

CAHIER DES CHARGES - INTÉGRATION BACKEND/FRONTEND

Projet d'Adaptation des Interfaces Restaurant

Version : 3.0

Date : 18 janvier 2026

Auteur : Équipe Technique

Contexte : Adaptation IHM à l'environnement

Parler des champs du back non utilisés pour le moment (billing dans les tableOrders par ex)

CAHIER DES CHARGES - INTÉGRATION BACKEND/FRONTEND.....	1
Résumé Exécutif.....	3
I. Contexte et objectifs.....	4
II. État des Lieux Technique.....	5
II.1 API du Backend fourni.....	5
II.2 Besoins Frontend généraux et par adaptation.....	8
II.3 Synthèse de l'écart entre les deux systèmes.....	9
3. Analyse des Trois Solutions Architecturales.....	11
3.1 Solution 1 : Frontend Manages the Fork (Adaptation côté Frontend).....	11
Description de la solution.....	11
Gestion des adaptations fonctionnelles.....	12
Avantages.....	13
Inconvénients.....	13
Cas d'usage recommandés.....	14
3.2 Solution 2 : Backend For Frontend (BFF).....	14
Description de la solution.....	14
Gestion des adaptations fonctionnelles.....	16
Avantages.....	18
Inconvénients.....	18
Cas d'usage recommandés.....	19
3.3 Solution 3 : Modification du Backend (Micro-Frontend Approach).....	19
Description de la solution.....	19
Gestion des adaptations fonctionnelles.....	22
Avantages.....	23
Inconvénients.....	23
Cas d'usage recommandés.....	24
4. Comparaison Multi-Critères.....	24
4.1 Tableau comparatif synthétique.....	24
4.2 Analyse détaillée par critère.....	25
5. Recommandation et Justification.....	25
5.1 Classement des solutions.....	26
5.2 Décision finale.....	27
6. Plan d'Implémentation de la Solution Retenue.....	27
Phase 1 : Enrichissement Backend (Semaine 1).....	27
Phase 2 : Migration Données (Semaine 1-2).....	27
Phase 3 : Nouveaux Endpoints (Semaine 2).....	28
Phase 4 : Service API Frontend (Semaine 3).....	28
Phase 5 : Adaptation Composants React (Semaine 3-4).....	28
Phase 6 : Tests et Validation (Semaine 4-5).....	29
Phase 7 : Déploiement (Semaine 5).....	29
Estimation globale.....	29
7. Conclusion.....	30
7.1 Synthèse.....	30
7.2 Décision.....	30

7.3 Bénéfices attendus.....	31
7.4 Risques et mitigation.....	31
7.5 Perspective.....	31

Résumé Exécutif

Ce rapport analyse trois approches architecturales distinctes pour l'intégration d'un frontend React existant avec un backend NestJS nouvellement fourni. Le frontend, développé initialement avec des données simulées localement, doit être connecté à une API réelle sans perte de fonctionnalités. Les trois solutions étudiées sont : (1) l'adaptation côté frontend (Frontend Manages the Fork), (2) l'utilisation d'une couche intermédiaire BFF (Backend For Frontend), et (3) la modification du backend existant. Chaque approche est évaluée selon des critères de complexité, maintenabilité, performance, coût et évolutivité. Une recommandation finale est formulée sur la base de cette analyse comparative.

I. Contexte et objectifs

Le projet consiste à intégrer deux systèmes développés indépendamment :

Le frontend existant :

Une application web React/TypeScript fonctionnelle, développée pour un système de commande en restaurant sur mobile ou tablette par les clients ou un serveur.

Elle propose deux modes d'interaction distincts :

- Le mode Standard, qui comporte des filtres, des suggestions contextuelles de repas et la possibilité de demander à manger rapidement en cas d'affluence (Mode Rush).
- Le mode Enfant, qui consiste en une interface ludique avec un système de récompenses.

Actuellement, cette application fonctionne en totale autonomie avec des données simulées localement dans le code source du frontend.

Le backend fourni :

Une API REST développée avec NestJS et PostgreSQL, exposant des endpoints pour la gestion des plats et des commandes. Ce backend n'a pas été développé spécifiquement pour notre application et présente donc un décalage entre son modèle de données et celui attendu par l'interface (le backend idéal présenté à mi-projet), particulièrement en ce qui concerne les adaptations de l'interface étant donné qu'elles ont été choisies différemment par chaque groupe d'étudiants.

Dans l'objectif d'intégrer le backend fourni au frontend actuel, nous étudierons dans ce rapport trois méthodes d'intégration possibles :

- Une adaptation côté Frontend
- L'utilisation d'un service intermédiaire BFF (Backend For Frontend)
- Une adaptation des microservices

Dans chaque cas, nous décrirons le workflow et les diagrammes de séquence à implémenter pour chacune des adaptations choisies pour notre application à l'étape 1, à savoir :

- Mode Rush
- Mode Enfant
- Suggestions de plat en cas de réflexion prolongée
- Mode Tablette / Smartphone

II. État des Lieux Technique

II.1 API du Backend fourni

D'après la documentation swagger déployée localement, le backend NestJS est une architecture microservices avec 3 services distincts :

Menu - Gestion des plats au menu (port 3001, gateway /menus)

Dining - Gestion des Tables et Commandes (port 3002, gateway /dining)

Kitchen - Gestion des Préparations (port 3003, gateway /kitchen)

Les endpoints concernant la partie ciblée par notre application sont les suivantes :

Menu Service - MenuItem :

GET /menus

Description: Récupère tous les plats du menu

Response: MenuItem[]

GET /menus/{menuItemid}

Description: Récupère un plat par son ID

Response: MenuItem

POST /menus

Description: Crée un nouveau plat

Body: CreateMenuItemDto

Response: MenuItem

Schema MongoDB MenuItem

```
{
  _id: string,
  fullName: string,    // Ex: "Steak avec frites maison"
  shortName: string,  // Ex: "Steak Frites"
  price: number,      // Ex: 18.50
  category: string,    // Enum : ["STARTER" , "MAIN" , "DESSERT" , "BEVERAGE"]
  image: string        // URL de l'image
}
```

Dining Service - Table :

GET /tables

Description: Liste toutes les tables du restaurant

Response: Table[]

POST /tables

Description: Ajouter une table au restaurant

Response: Table, ou erreur si table existante

GET /tables/{tableNumber}

Description: Renvoie une table du restaurant

Response: Table

Dining Service - TableOrder :

GET /tableOrders

Description: Récupère la liste des commandes

Response: TableOrder[]

POST /tableOrders

Description: Crée une commande pour une table

Body: { tableNumber: number, customersCount: number }

Response: TableOrder

GET /tableOrders/{tableOrderId}

Description: Récupère une commande

Response: TableOrder

POST /tableOrders/{tableOrderId}

Description: Ajoute des plats à une commande

Body: { "menuItem": string, "menuItemShortName": string, "howMany": 1 }

Response: TableOrder

POST /tableOrders/{tableOrderId}/bill

Description: Confirme la facturation de la commande spécifiée

Response: TableOrder

Schema MongoDB Table :

```
{
  _id: string,
  number: number,
  taken: boolean,
  tableOrderId: string,
}
```

Schema MongoDB TableOrder :

```
{
  _id: string,
  tableNumber: number,
  customersCount: number,
  opened: string (date),
  lines: OrderingLine[],
  preparations: PreparationDto[],
  bill : null || string (date)
}
```

Dining Service - PreparationDto :

POST /tableOrders/{tableOrderId}/prepare
Description: Ajoute des plats à une commande
Body: { "menuItem" : string,
"menuItemShortName" : string,
"howMany" : 1 }
Response: PreparationDto[]

Schema MongoDB PreparationDto :

```
{  
    _id: string,  
    shouldBeReadyAt: string (date),  
    preparedItems: OrderingItem[]  
}
```

Nous ne détaillons pas l'API de KitchenService qui concerne plutôt une autre partie du logiciel que notre frontend de commande mais on notera que la durée des préparations est par exemple codée dans cette partie du backend actuellement, on utilisera cette information indirectement via shouldBeReadyAt.

II.2 Besoins Frontend généraux et par adaptation

Adaptation système : Rush Hour Mode

- **Besoin** : Filtrer les plats par temps de préparation
- **Données manquantes** : isQuick, prepTime (temps de préparation en minutes, absence compensée par shouldBeReadyAt des PreparationDto et le temps de préparation des recettes dans la kitchen API), rushStatus (mode rush activé / désactivé)
- **Logique** : Afficher uniquement les plats avec prepTime <= 30min ou prepTime <= 15min

Adaptation à l'âge : Child Mode

- **Besoin** : Afficher uniquement les plats adaptés aux enfants
- **Données manquantes** :
 - kidFriendly (boolean)
 - kidFriendlyDescription?
 - ChildModeConfig (messages de chef Léo, d'encouragements)
 - Liste des récompenses
- **Logique** : Filtrer kidFriendly === true, interface simplifiée avec dialogues, afficher les récompenses que l'enfant peut choisir.

Adaptation cognitive : Panel Suggestions

- **Besoins** :
 - Plat du jour : isSpecialOfDay (boolean)
 - Plats populaires : popularity (note 1-5)
 - Trending : Statistiques de commandes
- **Données manquantes** : isSpecialOfDay, popularity, orderCount, lastOrdered
- **Logique** : Lorsque l'utilisateur reste plus de 7 secondes sur une carte de plat, une suggestion pertinente en fonction des données citées est proposée par le système

Adaptation au dispositif : Mode Tablette / Smartphone (+ Table POC) :

Géré côté frontend, inclusion du backend insignifiante

Autres :

Advanced Filters

- **Besoins** : Filtres diététiques et recherche par ingrédients
- **Données manquantes** :
 - description: string
 - subcategory: string
 - ingredients: string[]
 - isVegetarian: boolean
 - isVegan: boolean
 - isGlutenFree: boolean
 - spicyLevel: number
 - isLight : boolean
 - isLocal: boolean
 - allergens: string[]
 - cuisine: string

Général

- **Besoins** : Configurer l'application et récupérer la charte graphique
- **Données manquantes** : RestaurantConfig : Configuration générale (nom, logo, message d'accueil, configuration Rush, features)

II.3 Synthèse de l'écart entre les deux systèmes

- Absence de 14+ champs du frontEnd dans MenuItem : kidFriendly, prepTime, popularity, isQuick, ingredients[], isVegetarian, isVegan, isGlutenFree, spicyLevel, isLight, isLocal, hasVegetables, isSpecialOfDay, cuisine
- Catégories incompatibles : Backend utilise STARTER/MAIN/DESSERT/BEVERAGE vs Frontend attend "entrée"/"plat"/"dessert" + subcategory
- Absence de l'entité ChildReward : Aucune table pour les récompenses du mode enfant
- Absence d'endpoint `/api/rush-status` : Pas de détection dynamique du nombre de commandes
- Absence d'endpoint `/api/restaurant-config` : Configuration non accessible via API
- Absence d'endpoint `/api/recommendations` : Système de suggestions non implémenté

Fonctionnalité	Frontend (attendu)	Backend (fourni)	Écart
Endpoint plats	GET /api/dishes	GET /menus	Nom différent
Champs Menuitem	20+ champs (kidFriendly, prepTime, etc.)	6 champs (fullName, shortName, price, category, image, _id)	14+ champs manquants
Catégories	"entrée"/"plat"/"dessert" + subcategory	"STARTER"/"MAIN"/"DESSERT"/"BEVERAGE"	Format incompatible
Récompenses enfant	GET /api/child-rewards	Inexistant	Entité + endpoint manquants
Configuration	GET /api/restaurant-config	Inexistant	Endpoint manquant
Statut Rush	GET /api/rush-status (polling 10s)	Inexistant	Endpoint manquant
Recommandations	POST /api/recommendations	Inexistant	Endpoint manquant
Type de BDD	Format frontend flexible	MongoDB avec _id	Adaptation nécessaire

Cet écart important nécessite une stratégie d'adaptation robuste pour réconcilier les deux systèmes.

3. Analyse des Trois Solutions Architecturales

3.1 Solution 1 : Frontend Manages the Fork (Adaptation côté Frontend)

Description de la solution

Cette approche consiste à “gérer l'adaptation des données entièrement côté frontend”. Le frontend interroge le backend existant sans modification, puis enrichit, transforme et complète les données reçues avant de les utiliser dans les composants React.

Architecture :

Backend NestJS (inchangé) → Service d'Adaptation Frontend → Composants React

Principe de fonctionnement :

1. Le frontend appelle l'API backend existante (`GET /dishes`)
2. Une couche d'adaptation côté frontend transforme les données :
 - Ajoute les champs manquants avec des valeurs par défaut
 - Enrichit certains champs à partir de données locales complémentaires
 - Calcule des valeurs via des heuristiques (ex: détection kidFriendly par mots-clés)
3. Les données transformées sont passées aux composants React

Exemple d'implémentation :

```
``typescript
class BackendAdapter {
  async getDishes(): Promise<Dish[]> {
    // Appel du VRAI endpoint du backend fourni
    const response = await fetch('http://localhost:3000/menus');
    const menuItems = await response.json(); // MenuItem[]
    return menuItems.map(item => this.enrichMenuItem(item));
  }

  private enrichMenuItem(menuItem: MenuItem): Dish {
    return {
      id: menuItem._id,
      name: menuItem.fullName,
      description: menuItem.shortName, // Pas de vraie description
      category: this.convertCategory(menuItem.category),
      subcategory: menuItem.category,
      price: menuItem.price,
      imageUrl: menuItem.image,
      // Champs estimés avec heuristiques
      prepTime: this.estimatePrepTime(menuItem.fullName),
      popularity: 3, // Valeur par défaut
    };
  }
}
```

```

    isQuick: false, // Impossible à déterminer
    kidFriendly: this.detectKidFriendly(menuItem.fullName),
    ingredients: this.extractIngredients(menuItem.shortName),
    // ... 8 autres champs avec valeurs par défaut
  };
}

private convertCategory(backendCat: string): string {
  // Conversion STARTER/MAIN/DESSERT → entrée/plat/dessert
  const map = { STARTER: 'entrée', MAIN: 'plat', DESSERT: 'dessert' };
  return map[backendCat] || 'plat';
}

private detectKidFriendly(name: string): boolean {
  const kidWords = ['nuggets', 'frites', 'pizza', 'pâtes', 'glace'];
  return kidWords.some(word => name.toLowerCase().includes(word));
}
}
...

```

Pour les données totalement absentes du backend (récompenses enfant, configuration), elles restent stockées localement.

Gestion des adaptations fonctionnelles

Mode Enfant (Chef Léo)

Problématique : Nécessite l'identification des plats adaptés aux enfants (kidFriendly), la gestion des récompenses (étoiles, cadeaux), et les messages encourageants de Chef Léo.

Implémentation :

- Détection kidFriendly : Heuristique par mots-clés dans le nom du plat ("nuggets", "frites", "pizza", "pâtes", "glace"). Limitation majeure : Imprécis, ne détecte pas tous les plats adaptés, risque de faux positifs.
- Récompenses : Stockées dans un fichier local `/data/restaurant-data.ts` (5-6 récompenses hardcodées). Non synchronisées avec le backend.
- Messages Chef Léo : Configuration locale dans le même fichier. Modifiable uniquement en redéployant le frontend.
- Calcul des étoiles : Logique côté frontend (entrée = +2, plat = +4, dessert = +2).

Qualité : Dégradée - Le mode enfant fonctionne mais avec des données peu fiables.

Mode Rush

Problématique : Détection dynamique de l'affluence pour afficher une interface simplifiée avec plats rapides.

Implémentation :

- Détection Rush : Simulation locale dans `/data/rushService.ts` retournant un nombre fixe (ex: 15 commandes). Pas de connexion au backend réel.
- Identification plats rapides : Champ `isQuick` défini à `false` par défaut car impossible à estimer. Le mode Rush affichera tous les plats sans distinction.
- Temps de préparation : Estimé par heuristique simple (salades = 10min, pizzas = 20min, défaut = 15min). Imprécis.

Qualité : Dégradée - Le mode Rush s'active mais n'affiche pas les bons plats rapides.

Suggestions Intelligentes

Problématique : Recommandations basées sur l'hésitation de l'utilisateur et l'historique des commandes.

Implémentation :

- Détection d'hésitation : Fonctionnelle côté frontend (timer 3-4 secondes).
- Génération recommandations : Impossible avec cette solution. Aucune donnée d'association de plats, aucun historique. Les suggestions seront soit désactivées, soit basées sur des règles simplistes (popularité par défaut, même catégorie).
- Personnalisation : Aucune, car pas d'accès aux données de commandes passées.

Qualité : Inexistante ou très basique - Fonctionnalité compromise.

Avantages

- Indépendance vis-à-vis du backend : Aucune modification du backend n'est nécessaire, aucune coordination avec l'équipe backend, déploiement frontend indépendant
- Rapidité de mise en œuvre : Développement uniquement côté frontend, pas de migration de base de données, temps estimé : 1-2 semaines
- Flexibilité : Contrôle total sur la logique de transformation, possibilité d'ajuster rapidement les règles d'enrichissement
- Coût nul : Pas d'infrastructure additionnelle à déployer ou maintenir

Inconvénients

- Limitations fonctionnelles majeures :
 - Mode enfant dégradé : Le champ `kidFriendly` sera calculé de manière heuristique (mots-clés), imprécis et sujet à erreurs
 - Temps de préparation estimés : Impossible de calculer précisément `prepTime` sans information du backend, impact sur le mode Rush
 - Filtres incomplets : Les filtres avancés (végétarien, sans gluten, cuisine locale) seront peu fiables car basés sur des estimations

- Pas de données historiques : Impossible de calculer la vraie popularité des plats ou les associations pour les recommandations
- Complexité croissante côté frontend : Le code frontend devient responsable de la logique métier (détection kidFriendly, estimation des temps), violation du principe de séparation des responsabilités
- Maintenabilité faible : Duplication de données entre frontend et backend (configuration, récompenses), risque d'incohérence, dette technique qui s'accumule
- Performance : Traitement côté client des transformations (consommation batterie sur mobile), pas de cache centralisé

Cas d'usage recommandés

Cette solution est appropriée pour :

- Projet à court terme ou prototype nécessitant une mise en œuvre rapide
- Backend non modifiable (contrainte externe, API tierce)
- Équipe backend indisponible ou non collaborative

Pour ce projet : Cette solution est acceptable mais non optimale car elle compromet la qualité du mode enfant et du mode Rush, fonctionnalités centrales de l'application.

3.2 Solution 2 : Backend For Frontend (BFF)

Description de la solution

Le pattern BFF (Backend For Frontend) consiste à créer une couche intermédiaire dédiée entre le frontend et le backend existant. Ce BFF :

- Expose une API spécifiquement conçue pour les besoins du frontend
- Interroge le backend existant pour les données de base
- Enrichit, transforme et agrège les données
- Possède sa propre base de données pour stocker les données complémentaires

Architecture :

Frontend React → BFF (NestJS) → Backend NestJS existant
 ↓
 Base de données BFF

Principe de fonctionnement :

1. Le frontend interroge le BFF via des endpoints dédiés (`GET /api/dishes`, `GET /api/child-rewards`, etc.)

2. Le BFF récupère les données du backend existant (`GET /dishes`)
3. Le BFF enrichit ces données avec :
 - Informations stockées dans sa propre base de données (kidFriendly, prepTime, etc.)
 - Données calculées en temps réel (popularité, recommandations)
 - Configuration et récompenses stockées localement
4. Le BFF retourne au frontend exactement le format attendu

Base de données BFF (PostgreSQL ou MongoDB) :

```
```sql
```

```
-- Tables propres au BFF pour enrichir les données de /menus
```

```
CREATE TABLE dish_metadata (
 menu_item_id VARCHAR(24) PRIMARY KEY, -- Correspond au _id MongoDB du backend
 kid_friendly BOOLEAN,
 prep_time INTEGER,
 is_quick BOOLEAN,
 ingredients TEXT[],
 popularity INTEGER,
 subcategory VARCHAR(50),
 is_vegetarian BOOLEAN,
 is_vegan BOOLEAN,
 is_gluten_free BOOLEAN,
 spicy_level INTEGER,
 is_light BOOLEAN,
 is_local BOOLEAN,
 has_vegetables BOOLEAN,
 is_special_of_day BOOLEAN,
 cuisine VARCHAR(50)
);
```

```
CREATE TABLE child_rewards (
 id VARCHAR(50) PRIMARY KEY,
 name VARCHAR(100),
 emoji VARCHAR(10),
 stars INTEGER CHECK (stars IN (3, 6)),
 description TEXT
);
```

```
CREATE TABLE restaurant_config (
 config_key VARCHAR(100) PRIMARY KEY,
 config_value JSONB
);
```

```
CREATE TABLE dish_associations (
 dish_id VARCHAR(10),
 associated_dish_id VARCHAR(10),
 frequency INTEGER,
 score FLOAT,
 PRIMARY KEY (dish_id, associated_dish_id)
```

```
);
...
```

Implémentation du BFF :

```
``typescript
@Injectable()
export class DishesService {
 constructor(
 private httpService: HttpService,
 private bffDatabase: DatabaseService
) {}

 async getEnrichedDishes(): Promise<Dish[]> {
 // 1. Récupérer les MenuItems du backend réel (port 3000)
 const response = await this.httpService.get('http://localhost:3000/menus');
 const menuItems: MenuItem[] = response.data;

 // 2. Récupérer les métadonnées depuis la BDD BFF
 const metadata = await this.bffDatabase.query(
 'SELECT * FROM dish_metadata'
);

 // 3. Fusionner MenuItem + metadata → Dish complet
 return menuItems.map(item => {
 const meta = metadata.find(m => m.menu_item_id === item._id);
 return {
 id: item._id,
 name: item.fullName,
 description: item.shortName,
 category: this.convertCategory(item.category),
 price: item.price,
 imageUrl: item.image,
 ...meta // Ajoute les 14+ champs enrichis
 };
 });
 }
}
```

## Gestion des adaptations fonctionnelles

### Mode Enfant (Chef Léo)

Problématique : Identifier les plats adaptés aux enfants, gérer les récompenses et les messages de Chef Léo.

Implémentation :

- Champ kidFriendly : Stocké de manière fiable dans la table `dish\_metadata` du BFF. Renseigné manuellement ou via import des données initiales. Précision maximale.
- Récompenses : Table dédiée `child\_rewards` dans la BDD BFF avec toutes les récompenses (nom, emoji, nombre d'étoiles, description). Endpoint `GET /api/child-rewards`.
- Messages Chef Léo : Stockés dans `restaurant\_config` avec la clé `child\_mode\_config`. Modifiables via l'API sans redéploiement. Endpoint `GET /api/child-mode-config`.
- Calcul des étoiles : Logique côté frontend (identique) mais les données kidFriendly sont fiables.

Qualité : Excellente - Fonctionnalité complète et fiable.

## Mode Rush

Problématique : Détection dynamique de l'affluence et identification des plats rapides.

Implémentation :

- Détection Rush : Le BFF expose `GET /api/rush-status` qui interroge le Dining Service (`GET http://localhost:3001/tableOrders`), compte les commandes ouvertes (`opened: true`), et retourne le nombre. Détection en temps réel basée sur la charge réelle.
- Identification plats rapides : Champ `isQuick` dans `dish\_metadata`, renseigné avec les vrais temps de préparation. Filtrage précis des plats < 15 minutes.
- Temps de préparation : Champ `prepTime` fiable stocké en base. Affiché correctement dans l'interface.
- Polling : Le frontend interroge l'endpoint toutes les 10 secondes. Le BFF peut mettre en cache la réponse 5-10 secondes pour optimiser.

Qualité : Excellente - Mode Rush pleinement fonctionnel avec données réelles.

## Suggestions Intelligentes

Problématique : Recommandations personnalisées basées sur l'hésitation et l'historique.

Implémentation :

- Endpoint : `POST /api/recommendations` exposé par le BFF.
- Table d'associations : `dish\_associations` dans la BDD BFF contenant les paires de plats fréquemment commandés ensemble (dishId, associatedDishId, frequency, score). Données calculées depuis l'historique des commandes du Dining Service.
- Algorithme : Le BFF analyse la table, filtre selon le contexte (userMode, isRushMode, currentCart), priorise par score et popularité, retourne top 3-5 plats.
- Personnalisation : Filtrage kidFriendly en mode enfant, filtrage isQuick en mode Rush, exclusion du panier actuel.
- Détection hésitation : Côté frontend (timer 3-4s), envoie requête au BFF.

Qualité : Excellente - Recommandations intelligentes et contextuelles.

## Avantages

- Séparation claire des responsabilités : Le frontend reste concentré sur l'UI, le BFF gère la logique métier de transformation, le backend existant continue de fonctionner normalement
- API optimisée pour le frontend : Endpoints sur mesure correspondant exactement aux besoins de l'interface, réduction du nombre d'appels réseau, format de réponse exactement conforme
- Enrichissement avec base de données dédiée : Stockage des métadonnées manquantes (kidFriendly, prepTime, ingredients), gestion des données spécifiques au mode enfant, possibilité de créer des tables d'association (recommandations)
- Évolutivité : Ajout facile de nouvelles fonctionnalités (cache, analytics), possibilité d'interroger plusieurs backends différents à l'avenir, versionning de l'API frontend sans impacter le backend
- Performance : Cache centralisé au niveau du BFF (Redis, mémoire), réduction de la charge sur le backend existant, agrégation côté serveur

## Inconvénients

- Complexité d'infrastructure : Nécessite un nouveau service à déployer (serveur Node.js/NestJS), nécessite une nouvelle base de données (PostgreSQL, MySQL), gestion de deux connexions réseau (frontend ↔ BFF, BFF ↔ backend)
- Coûts opérationnels : Serveur supplémentaire à héberger (~20-50€/mois), base de données supplémentaire (~15-30€/mois), monitoring et logs, estimation : +30-40% de coûts d'infrastructure
- Temps de développement : Développement complet d'un nouveau service (contrôleurs, services, modèles), création et migration de la base de données BFF, tests unitaires et d'intégration, temps estimé : 3-4 semaines
- Dépendance du BFF : Si le BFF est en panne, le frontend ne fonctionne plus, point de défaillance unique si non redondé
- Synchronisation des données : Les métadonnées dans la BDD BFF doivent être synchronisées avec les plats du backend, risque d'incohérence
- Overhead réseau : Deux appels réseau au lieu d'un (frontend → BFF → backend), latence légèrement augmentée (+20-50ms)

## Cas d'usage recommandés

Le BFF est recommandé dans les cas suivants :

- Écart important entre le modèle backend et les besoins frontend (notre cas : 14+ champs manquants)
- Fonctionnalités complexes nécessitant de l'agrégation (recommandations, analytics)
- Backend non modifiable ou partagé par plusieurs applications
- Plusieurs frontends à supporter (web, mobile native, desktop)

Pour ce projet : Cette solution est très adaptée car l'écart entre les modèles est significatif et les fonctionnalités spéciales (mode enfant, rush, recommandations) nécessitent des données enrichies.

### 3.3 Solution 3 : Modification du Backend (Micro-Frontend Approach)

#### Description de la solution

Cette approche consiste à modifier directement le backend NestJS fourni pour qu'il expose nativement toutes les données et endpoints requis par le frontend. Le backend devient la source unique de vérité, contenant l'intégralité du modèle de données enrichi.

Architecture :

...

Frontend React (appels directs) → Backend NestJS modifié (PostgreSQL enrichie)

...

Principe de fonctionnement :

1. Le modèle de données backend est enrichi avec tous les champs nécessaires
2. De nouveaux endpoints sont créés pour les fonctionnalités manquantes
3. Le frontend interroge directement le backend sans couche intermédiaire
4. Toutes les données (plats, récompenses, configuration) sont stockées en base de données PostgreSQL

Modifications du backend :

Enrichissement de l'entité MenuItem existante :

```
``typescript
// Backend réel actuel (MongoDB)
interface MenuItem {
 _id: string;
 fullName: string;
 shortName: string;
 price: number;
 category: 'STARTER' | 'MAIN' | 'DESSERT' | 'BEVERAGE';
 image: string;
}
```

```

// À TRANSFORMER EN (PostgreSQL avec TypeORM)
@Entity('menu_items')
export class MenuItem {
 @PrimaryColumn()
 _id: string; // Garder compatibilité MongoDB

 @Column()
 fullName: string;

 @Column()
 shortName: string;

 @Column('decimal')
 price: number;

 @Column()
 category: string;

 @Column()
 image: string;

 // ===== 14+ NOUVEAUX CHAMPS AJOUTÉS =====
 @Column({ nullable: true })
 subcategory: string;

 @Column({ nullable: true })
 prepTime: number;

 @Column({ default: 3 })
 popularity: number;

 @Column({ default: false })
 kidFriendly: boolean;

 @Column('simple-array', { nullable: true })
 ingredients: string[];

 @Column({ default: false })
 isVegetarian: boolean;

 @Column({ default: false })
 isVegan: boolean;

 @Column({ default: false })
 isGlutenFree: boolean;

 @Column({ default: 0 })

```

```

 spicyLevel: number;

 @Column({ default: false })
 isLight: boolean;

 @Column({ default: false })
 isLocal: boolean;

 @Column({ default: false })
 hasVegetables: boolean;

 @Column({ default: false })
 isSpecialOfDay: boolean;

 @Column({ default: false })
 isQuick: boolean;

 @Column({ nullable: true })
 cuisine: string;
}
...

```

Création de nouvelles entités :

```
``typescript
```

```

@Entity('child_rewards')
export class ChildReward {
 @PrimaryColumn()
 id: string;

```

```

 @Column()
 name: string;

```

```

 @Column()
 stars: number;
}

```

```

@Entity('restaurant_config')
export class RestaurantConfig {
 @PrimaryColumn()
 configKey: string;

```

```

 @Column('jsonb')
 configValue: any;
}
...

```

Nouveaux endpoints : `GET /child-rewards`, `GET /restaurant-config`, `GET /rush-status`,  
`POST /recommendations`

## Gestion des adaptations fonctionnelles

### Mode Enfant (Chef Léo)

Problématique : Identifier les plats adaptés aux enfants, gérer les récompenses et les messages de Chef Léo.

Implémentation :

- Champ kidFriendly : Ajouté directement dans l'entité `MenuItem` du Menu Service avec validation. Renseigné dans la base MongoDB lors de la création/modification des plats. Précision maximale, source unique de vérité.
- Récompenses : Nouvelle collection MongoDB `child\_rewards` dans le Menu Service. Nouveau module `ChildRewardsModule` avec endpoints `GET /child-rewards`, `POST /child-rewards` (admin). Gestion complète des récompenses (CRUD).
- Messages Chef Léo : Collection `restaurant\_config` avec document `child\_mode` . Nouveau module `RestaurantConfigModule` avec endpoints `GET /restaurant-config`, `GET /child-mode-config` . Modifiables via API sans redéploiement.
- Calcul des étoiles : Logique côté frontend, données kidFriendly fiables du backend.

Qualité : Excellente - Fonctionnalité native du backend, entièrement intégrée.

### Mode Rush

Problématique : Détection dynamique de l'affluence et identification des plats rapides.

Implémentation :

- Détection Rush : Nouveau module `RushStatusModule` dans le Menu Service exposant `GET /rush-status` . Le service interroge le Dining Service (`GET http://localhost:3001/tableOrders`), compte les commandes ouvertes, retourne le nombre. Communication inter-services native.
- Identification plats rapides : Champ `isQuick` ajouté à l'entité `MenuItem`, calculé automatiquement à partir de `prepTime < 15` . Filtrage natif via query param `GET /menus?isQuick=true` .
- Temps de préparation : Champ `prepTime` dans `MenuItem`, renseigné en base avec les vraies valeurs. Affiché directement dans les réponses.
- Polling : Le frontend interroge `/rush-status` toutes les 10 secondes. Le backend peut cacher la réponse.

Qualité : Excellente - Mode Rush natif avec données en temps réel.

### Suggestions Intelligentes

Problématique : Recommandations personnalisées basées sur l'hésitation et l'historique.

Implémentation :

- Endpoint : `POST /recommendations` dans un nouveau module `RecommendationsModule` du Menu Service.
- Collection d'associations : `dish\_associations` dans MongoDB avec paires de plats (dishId, associatedDishId, frequency, score). Calculées depuis l'historique du Dining Service via un job batch périodique ou événements.
- Algorithme : Service NestJS dédié qui analyse les associations, interroge la collection `menu\_items` pour filtrer (kidFriendly, isQuick), applique les règles métier, retourne top 3-5 plats.
- Personnalisation : Requête enrichie avec contexte (userMode, isRushMode, currentCart), filtrage natif MongoDB avec agrégation pipeline pour performances.
- Évolutivité : Le système peut évoluer vers du machine learning (calcul de score dynamique basé sur historique complet).

Qualité : Excellente - Recommandations natives, évolutives, performantes.

## Avantages

- Architecture simplifiée : Un seul backend à maintenir et déployer, pas de couche intermédiaire, moins de points de défaillance
- Source unique de vérité (Single Source of Truth) : Toutes les données métier sont dans la base de données PostgreSQL, pas de duplication, cohérence garantie
- Performance optimale : Un seul appel réseau entre frontend et backend, latence minimale (pas d'overhead), requêtes SQL optimisées possibles
- Données fiables et complètes : Tous les champs sont renseignés correctement en base de données (pas d'estimation), validation des données côté backend
- Coûts réduits : Pas de serveur BFF supplémentaire, une seule base de données, coûts identiques au backend initial
- Évolutivité cohérente : Toutes les évolutions futures se font au même endroit, réutilisable pour d'autres clients

## Inconvénients

- Modification du backend existant : Nécessite l'accès et la permission de modifier le backend fourni, risque de casser l'existant si le backend est utilisé par d'autres applications
- Temps de développement significatif : Migrations de base de données complexes (14+ nouveaux champs), création de nouveaux modules, mise à jour de la documentation API, temps estimé : 4-5 semaines

- Couplage fort frontend-backend : Le backend est spécifiquement adapté aux besoins de ce frontend, risque de créer un backend monolithique
- Migration des données : Il faut peupler les nouveaux champs pour tous les plats existants, écriture de scripts de migration
- Tests impactés : Tous les tests backend existants doivent être mis à jour, risque de régressions

### Cas d'usage recommandés

Cette solution est recommandée dans les cas suivants :

- Backend entièrement contrôlé par l'équipe (pas de contrainte externe)
- Pas d'autres clients dépendant du backend existant
- Projet à long terme nécessitant une architecture unifiée
- Équipe backend disponible et collaborative

Pour ce projet : Cette solution est optimale si le backend peut être modifié sans contrainte. Elle offre la meilleure architecture à long terme avec une maintenance simplifiée et des performances maximales.

## 4. Comparaison Multi-Critères

### 4.1 Tableau comparatif synthétique

Critère	Frontend Manages Fork	BFF	Backend Modifié
<b>Complexité implémentation</b>	2/5 Faible	4/5 Élevée	5/5 Très élevée
<b>Temps développement</b>	1-2 semaines	3-4 semaines	4-5 semaines
<b>Coût infrastructure</b>	0€	+45-90€/mois (service supplémentaire)	0€
<b>Performance</b>	3/5 Moyenne	4/5 Bonne	5/5 Excellente
<b>Fiabilité données</b>	2/5 Faible	5/5 Excellente	5/5 Excellente
<b>Maintenabilité</b>	2/5 Faible	4/5 Bonne	5/5 Excellente

<b>Évolutivité</b>	2/5 Limitée	5/5 Excellente	4/5 Bonne
<b>Qualité mode enfant</b>	2/5 Dégradée	5/5 Complète	5/5 Complète
<b>Qualité mode rush</b>	2/5 Dégradée	5/5 Complète	5/5 Complète
<b>Indépendance backend</b>	5/5 Totale	4/5 Élevée	1/5 Aucune
<b>Risque projet</b>	Faible	Moyen	Élevé

## 4.2 Analyse détaillée par critère

Temps et complexité :

- Frontend Fork : Rapide (1-2 sem) mais complexité technique frontend
- BFF : Moyen (3-4 sem), complexité infrastructure
- Backend Modifié : Long (4-5 sem), complexité migration

Coûts :

- Frontend Fork et Backend Modifié : 0€ additionnel
- BFF : +45-90€/mois (serveur + BDD)

Performance :

- Frontend Fork : Calculs client, pas de cache centralisé
- BFF : Bon avec cache, légère latence (+20-50ms)
- Backend Modifié : Optimal, communication directe

Fiabilité :

- Frontend Fork : Critique - Mode enfant compromis (kidFriendly imprécis), mode Rush dégradé (prepTime estimé)
- BFF et Backend Modifié : Données fiables en BDD, fonctionnalités complètes

Maintenabilité :

- Frontend Fork : Faible (dette technique, logique métier dans UI)
- BFF : Bonne (séparation claire)
- Backend Modifié : Excellente (source unique, évolutions centralisées)

## 5. Recommandation et Justification

### 5.1 Classement des solutions

### 1) Solution recommandée : Modification du Backend (Solution 3)

Justification principale :

1. Contexte académique favorable : Backend modifiable sans contraintes externes, pas d'autres clients dépendants
2. Qualité fonctionnelle prioritaire : Mode enfant avec Chef Léo est une fonctionnalité différenciante nécessitant données fiables (kidFriendly ne peut être estimé)
3. Architecture pérenne : Source unique de vérité, maintenabilité optimale long terme
4. Performance maximale : Communication directe, latence minimale
5. Coûts nuls : Budget académique limité, évite coûts infra additionnels
6. Apprentissage complet : Pratique développement full-stack (migrations BDD, NestJS, intégration)

Conditions de validité :

- Backend NestJS modifiable (confirmation obtenue)
- Pas d'autres clients dépendants actuellement
- Projet long terme (plusieurs années)
- Équipe a accès au code backend et peut effectuer les migrations

### 2) Solution alternative : BFF (Solution 2)

Justification : Excellente séparation des responsabilités, fiabilité complète des fonctionnalités, évolutivité maximale

Conditions : Backend non modifiable (contrainte externe), budget suffisant (+45-90€/mois), équipe capable de développer un service backend additionnel

Si backend non modifiable : Cette solution est la meilleure alternative.

### 3) Solution à éviter : Frontend Manages Fork (Solution 1)

Raisons : Compromet gravement mode enfant et Rush (fonctionnalités clés), dette technique importante, maintenabilité faible

Uniquement pour : Prototype rapide (< 6 mois), POC, aucune possibilité de modifier backend ni de créer BFF

Cette solution n'est pas recommandée pour ce projet.

## 5.2 Décision finale

Solution retenue : Modification du Backend (Solution 3)

Cette décision s'appuie sur :

- Le contexte académique permettant modification backend sans contraintes
- La nécessité absolue de données fiables pour le mode enfant (fonctionnalité différenciante)
- L'architecture la plus pérenne et maintenable long terme
- L'absence de coûts d'infrastructure additionnels
- L'opportunité d'apprentissage complet du développement full-stack

## 6. Plan d'Implémentation de la Solution Retenue

### Phase 1 : Enrichissement Backend (Semaine 1)

Objectif : Ajouter champs manquants en BDD et mettre à jour entités NestJS

Tâches :

- Créer migrations SQL (14+ champs dishes, tables child\_rewards, restaurant\_config, dish\_associations)
- Mettre à jour entité Dish avec tous nouveaux champs
- Mettre à jour DTOs avec validations
- Créer nouvelles entités (ChildReward, RestaurantConfig)

Livrables : Migrations SQL testées, entités NestJS mises à jour

### Phase 2 : Migration Données (Semaine 1-2)

Objectif : Peupler PostgreSQL avec données actuelles de `restaurant-data.ts`

Tâches :

- Script migration plats (parser restaurant-data.ts → SQL INSERT)
- Script migration récompenses enfant
- Script migration configuration restaurant
- Vérification intégrité (nombre enregistrements, valeurs)

Livrables : BDD PostgreSQL peuplée, rapport validation migration

### Phase 3 : Nouveaux Endpoints (Semaine 2)

Objectif : Créer endpoints manquants requis par frontend

Tâches :

- Module ChildRewards : contrôleur, service, endpoint `GET /api/child-rewards`
- Module RestaurantConfig : endpoints `GET /api/restaurant-config`, `GET /api/child-mode-config`
- Module RushStatus : endpoint `GET /api/rush-status` (compte commandes en cours)
- Module Recommendations : endpoint `POST /api/recommendations` (logique recommandation)
- Enrichir `GET /api/dishes` avec tous les champs et query params filtrage
- Enrichir `POST /api/orders` (champs deviceType, userMode, childModeData)

Livrables : Tous endpoints créés et testés (Postman/Thunder Client)

## Phase 4 : Service API Frontend (Semaine 3)

Objectif : Créer service centralisé frontend pour gérer appels API

Tâches :

- Créer `ApiService.ts` (encapsulation fetch avec retry automatique)
- Configurer React Query (cache, stale-while-revalidate)
- Créer hooks personnalisés (useDishes, useRestaurantConfig, useChildRewards)
- Créer hook `useRushStatus` avec polling 10s
- Hook `useSubmitOrder` avec useMutation

Livrables : ApiService complet, hooks React Query configurés

---

## Phase 5 : Adaptation Composants React (Semaine 3-4)

Objectif : Remplacer imports locaux par appels API

Tâches :

- Remplacer imports dans composants (App.tsx, MenuView.tsx, ChildMode.tsx, RushHourMode.tsx)
- Ajouter états chargement (skeletons, spinners)
- Ajouter gestion erreurs (messages utilisateur, bouton "Réessayer")
- Supprimer fichiers locaux (restaurant-data.ts, rushService.ts)

Livrables : Composants adaptés, états chargement/erreurs, fichiers locaux supprimés

---

## Phase 6 : Tests et Validation (Semaine 4-5)

Objectif : Tester intégration complète, valider tous scénarios

Tests backend : Unitaires (services), Intégration (endpoints avec BDD test)

Tests frontend : Hooks React Query, Composants, End-to-end (Cypress/Playwright)

Scénarios validation :

- Mode Normal : Chargement plats, filtres, recherche, ajout panier, validation commande
- Mode Enfant : Plats kidFriendly, messages Chef Léo, étoiles, récompenses, childModeData envoyé
- Mode Rush : Activation auto (>10 commandes), bannière, plats rapides, désactivation (<10)
- Recommandations : Hésitation 4s → requête API → panneau suggestions

Tests performance : Temps chargement < 2s, latence API, cache fonctionnel

Livrables : Suite tests automatisés, rapport validation, rapport performance

---

## Phase 7 : Déploiement (Semaine 5)

Objectif : Déployer en production avec monitoring

Tâches :

- Configurer variables environnement (DATABASE\_URL, REACT\_APP\_API\_URL)
- Déployer backend (Heroku/Render/Railway) et exécuter migrations production
- Déployer frontend (Vercel/Netlify)
- Configurer CORS (autoriser domaine frontend)
- Monitoring : Sentry (erreurs frontend), logs backend (Winston)
- Tests production (tous scénarios)

Livrables : Application déployée, monitoring configuré, documentation déploiement

## Estimation globale

Durée totale : 4-5 semaines

Phase	Durée	Effort
Enrichissement backend	3-4 jours	Moyen

Migration données	2-3 jours	Faible
Nouveaux endpoints	4-5 jours	Élevé
Service API frontend	3-4 jours	Moyen
Adaptation composants	5-6 jours	Élevé
Tests validation	4-5 jours	Élevé
Déploiement	2-3 jours	Faible

Ressources : 1 développeur full-stack, accès PostgreSQL, services déploiement (Vercel + Heroku/Render)

## 7. Conclusion

### 7.1 Synthèse

Ce rapport a analysé trois approches pour intégrer le frontend React avec le backend NestJS :

1. Frontend Manages Fork : Rapide (1-2 sem) mais fonctionnalités dégradées (mode enfant/rush imprécis), dette technique
2. BFF : Excellent compromis si backend non modifiable, fiabilité complète, coûts additionnels (+45-90€/mois), 3-4 semaines
3. Modification Backend : Architecture optimale long terme, performance maximale, fiabilité complète, coûts nuls, 4-5 semaines

### 7.2 Décision

Solution retenue : Modification du Backend (Solution 3)

Justification :

- Contexte académique favorable (backend modifiable)
- Qualité complète mode enfant (fonctionnalité différenciante)
- Architecture pérenne (source unique vérité)
- Performance maximale (communication directe)
- Coûts nuls (budget limité)
- Apprentissage complet développement full-stack

## 7.3 Bénéfices attendus

- Expérience utilisateur complète (tous modes fonctionnels)
- Données fiables (PostgreSQL)
- Performance optimale
- Architecture maintenable et évolutive
- Base solide pour extensions futures (app mobile, back-office)

## 7.4 Risques et mitigation

Risques :

- Régression backend → Mitigation : tests exhaustifs
- Temps développement → Mitigation : planning détaillé, jalons clairs
- Complexité migration → Mitigation : scripts testés en dev d'abord

## 7.5 Perspective

Cette analyse illustre les défis de l'architecture logicielle moderne : équilibrer rapidité, qualité, coûts et maintenabilité. La méthodologie comparative présentée est réutilisable pour projets d'intégration similaires.

L'implémentation permettra de livrer une application restaurant complète, performante et évolutive, avec une expérience d'apprentissage enrichissante en développement full-stack.