

Rapport de Conception Technique et Réalisation

Projet TRD - The Real Deal

Application de Paris Sportifs en Temps Réel
Coupe du Monde FIFA 2026

Architecture Microservices Distribuée
avec Système de Recommandation

Équipe de Développement

Abderrazak SEGHIR

Mohamed Bahaa Eddine BEKKOUCHE

Institut des sciences du Digital, Management Cognition
Master Miage SID - Année 2025-2026

1^{er} février 2026

Résumé Exécutif

Ce rapport présente la conception, l'architecture et la réalisation du projet **TRD** (**The Real Deal**), une plateforme de paris sportifs en temps réel développée dans le cadre de la Coupe du Monde FIFA 2026. Le système adopte une architecture microservices distribuée basée sur des technologies modernes (.NET 8, RabbitMQ, AWS SES, React).

Points clés :

- Architecture événementielle avec plusieurs microservices découplés
- Communication asynchrone via RabbitMQ pour la résilience
- Notifications transactionnelles automatisées via AWS SES
- Service de simulation pour palier aux limitations des APIs externes
- Déploiement containerisé avec Docker et orchestration Kubernetes

Difficultés majeures surmontées :

- Limitation de l'API externe (10 appels/min) → Création d'un service Mock complet
- Synchronisation temps réel des scores → Pattern Event-Driven avec RabbitMQ
- Notifications fiables → Intégration AWS SES avec gestion d'erreurs

Table des matières

Résumé Exécutif	2
1 Introduction	6
1.1 Contexte et Problématique	6
1.1.1 Contraintes Techniques Identifiées	6
1.2 Objectifs du Projet	6
2 Espace du Problème : Analyse DDD	7
2.1 Analyse du Domaine & Frontières	7
2.1.1 Frontières du Domaine	7
2.1.2 Langage Omniprésent (Ubiquitous Language)	7
2.2 Event Storming	8
2.3 Processus Métier	9
2.3.1 Workflow 1 : Placement d'un Pari	9
2.3.2 Workflow 2 : Fin de Match et Calcul des Résultats	9
2.3.3 Workflow 3 : Retrait de Fonds	9
2.3.4 Workflow 4 : Authentification	10

2.4	Dérivation des Contextes Bornés (Bounded Contexts)	10
2.5	Diagramme de Contexte Système	11
3	Espace de la Solution : Architecture Technique	12
3.1	Choix d'Architecture	12
3.1.1	Architecture Microservices	12
3.1.2	Choix Technologiques Polyglottes	12
3.1.3	Infrastructure	13
3.1.4	Modèle de Base de Données par Service	13
3.1.5	Architecture Hexagonale (Ports & Adapters)	13
3.1.6	Détail des Couches	13
3.1.7	Modèles de Communication	14
3.1.8	Architecture Événementielle (Chorégraphie)	14
3.1.9	Orchestration (SAGA)	14
3.2	Implementation des Microservices Métiers	14
3.2.1	L'API Gateway & Orchestration	14
3.2.2	Service d'Authentification (Keycloak)	15
3.2.3	Bet Lifecycle Service	15
3.2.4	Account Service	16
3.2.5	Wallet Service	16
3.2.6	Service de Recommandation (Recommendation Engine)	17
3.2.7	Score Service (Worker Background)	19
3.2.8	Bet Result Service (Worker Background)	20
3.2.9	Mock Score Service (API + Frontend)	21
3.2.10	Service de Notification Email (AWS SES)	22
3.2.11	Match Odds Service	23
4	Difficultés et Solutions Apportées	26
4.1	Difficultés Majeures Surmontées	26
4.1.1	Limitation de l'API Externe Football-Data	26
4.1.2	Encodage UTF-8 des Emails HTML	26
4.1.3	Synchronisation des Événements RabbitMQ	27
4.1.4	Configuration AWS SES en Développement	27
4.2	Difficultés Rencontrées (Services de Paris et Cotes)	28
4.2.1	Gestion des Relations One-to-One avec Entity Framework	28
4.2.2	Mise à Jour Partielle avec PATCH	28
4.2.3	Migrations Automatiques au Démarrage	29
5	Docker : Développement et Tests	30
5.1	Philosophie de Conteneurisation	30
5.1.1	Dockerfile Multi-Étapes (Multi-Stage Build)	30
5.2	Orchestration avec Docker Compose	30
5.3	Le Problème du "Split Horizon" (DNS)	31
6	Kubernetes : Orchestration de Production	32
6.1	Stratégie de Migration	32
6.2	Architecture Détaillée Kubernetes	32
6.2.1	1. Réseau et DNS (Service Discovery)	32
6.2.2	2. La Couche de Données (StatefulSets)	32

6.2.3	3. La Couche Applicative (Deployments)	32
6.2.4	4. Stratégie d'Accès Externe (Port Forwarding)	33
6.3	Feuille de Route de Migration (Roadmap)	33
6.4	Diagramme d'Architecture Kubernetes	33
7	Répartition des Rôles	35
7.1	Services Développés - Partie 1	35
7.1.1	Services de Scoring et Notification	35
7.1.2	Responsabilités Techniques	35
7.1.3	Contributions Additionnelles	35
7.2	Services Développés - Partie 2	35
7.2.1	Services de Gestion des Matchs et Cotes	35
7.2.2	Responsabilités Techniques	35
7.2.3	Contributions Additionnelles	36
8	Tests et Validation	37
8.1	Stratégie de Test	37
8.1.1	Tests Manuels	37
8.1.2	Scénarios de Test Validés	37
8.1.3	Tests Unitaires	37
9	Conclusion et Perspectives	39
9.1	Bilan du Projet	39
9.1.1	Objectifs Atteints	39
9.1.2	Compétences Acquises	39
9.2	Améliorations Futures	40
9.2.1	Court Terme	40
9.2.2	Long Terme	40
9.3	Retour d'Expérience	40
9.3.1	Ce Qui a Bien Fonctionné	40
9.3.2	Défis Rencontrés	40
9.3.3	Leçons Apprises	40
A	Structure du Repository	41
B	Configuration RabbitMQ	42
C	Exemples de Requêtes API	43
C.0.1	Mock Score Service - Créer un Match	43
C.0.2	Mock Score Service - Terminer un Match	43
C.0.3	Match Odds Service - Créer une Équipe	43
C.0.4	Match Odds Service - Créer un Match	43
C.0.5	Match Odds Service - Ajouter des Cotes	44
C.0.6	Match Odds Service - Changer le Statut d'un Match	44
D	Glossaire	45

Table des figures

2.1	Diagramme d'Event Storming Global du projet TRD	8
2.2	Workflow de placement d'un pari	9
2.4	Diagramme de Contexte Système : Interactions entre Acteurs, Gateway, Services et Bus d'Événements	11
3.1	Topologie du Moteur de Recommandation	18
3.2	Diagramme de flux - Worker Background	20
3.3	Diagramme de flux - Service de Notification Email	22
3.4	Diagramme de flux - Consultation des cotes	25
5.1	Architecture Kubernetes : Services, Pods et Accès via Port Forwarding	31
6.1	Architecture Kubernetes : Services, Pods et Accès via Port Forwarding	34

Liste des tableaux

3.1	Avantages de l'architecture microservices pour TRD	12
3.2	API MockScoreService - Endpoints	21
3.3	Modèle de données MatchOddsService	23
3.4	API MatchOddsService - Endpoints	24
8.1	Scénarios de test validés	37
8.2	Scénarios de test MatchOddsService	38
D.1	Glossaire des termes techniques	45

Chapitre 1

Introduction

1.1 Contexte et Problématique

La Coupe du Monde FIFA 2026 représente un événement sportif majeur attirant des millions de parieurs à travers le monde. Le projet TRD vise à fournir une plateforme de paris sportifs moderne, scalable et réactive capable de gérer :

- La consultation en temps réel des scores de matchs
- Le placement de paris avec calcul dynamique des cotes
- La notification automatique des résultats aux utilisateurs
- La gestion des gains et pertes en temps réel

1.1.1 Contraintes Techniques Identifiées

1. Limitation des APIs externes : L'API Football-Data.org impose une limite stricte de 10 requêtes par minute en mode gratuit. Cette contrainte est incompatible avec les besoins de test et de démonstration du projet.

2. Dépendance aux matchs réels : L'attente de buts réels pendant les matchs rend les tests et démonstrations impraticables dans un contexte académique.

3. Scalabilité : Le système doit pouvoir gérer un grand nombre de paris simultanés lors des phases critiques des matchs.

4. Fiabilité : Les notifications de gains/pertes doivent être garanties pour maintenir la confiance des utilisateurs.

1.2 Objectifs du Projet

1. Concevoir une architecture microservices découpée et maintenable
2. Implémenter la communication événementielle pour la résilience
3. Intégrer des services cloud (AWS SES) pour démontrer la maturité technique
4. Créer un service de simulation pour contourner les limitations des APIs
5. Containeriser l'ensemble pour un déploiement reproductible

Chapitre 2

Espace du Problème : Analyse DDD

L'approche *Domain-Driven Design* (DDD) a été utilisée pour décomposer la complexité métier avant d'écrire la moindre ligne de code.

2.1 Analyse du Domaine & Frontières

Nous avons identifié les frontières suivantes et le vocabulaire commun (Langage Omniprésent).

2.1.1 Frontières du Domaine

- **Domaines Cœurs (Core Domains)** :
 - **Account Boundary** : Gère les informations utilisateurs, l'historique des paris et la balance visible.
 - **Betting Boundary** : Le cœur du métier. Gère les cotes, le placement des paris et le calcul des résultats.
 - **Games Boundary** : Gère les groupes d'équipes, le calendrier des matchs et les scores en temps réel.
- **Domaines Génériques** : Authentification (Auth Boundary) et Paiement (Payment Boundary) pour les dépôts/retraits.
- **Domaines de Support** : Notification Boundary (Emails/SMS temps réel).

2.1.2 Langage Omniprésent (Ubiquitous Language)

Terme	Définition
Utilisateur / Parieur	Une personne pouvant placer des paris sur des événements.
Compte	L'entité représentant un utilisateur avec solde et historique.
Pari / Mise (Bet/Wager)	L'engagement d'un utilisateur sur un événement avec une mise et une cote.
Cote (Odds)	Probabilité d'un résultat utilisée pour calculer les gains.
Match / Événement	Compétition sportive avec des résultats sur lesquels parier.
Résultat (Outcome)	Issue finale du match utilisée pour régler les paris.
Transaction	Tout mouvement d'argent (dépôt, retrait, gain).
Portefeuille (Wallet)	Compte monétaire permettant de placer des mises et recevoir des gains.

2.2 Event Storming

L'atelier d'Event Storming nous a permis de visualiser la chronologie des événements systèmes et les interactions entre les acteurs.

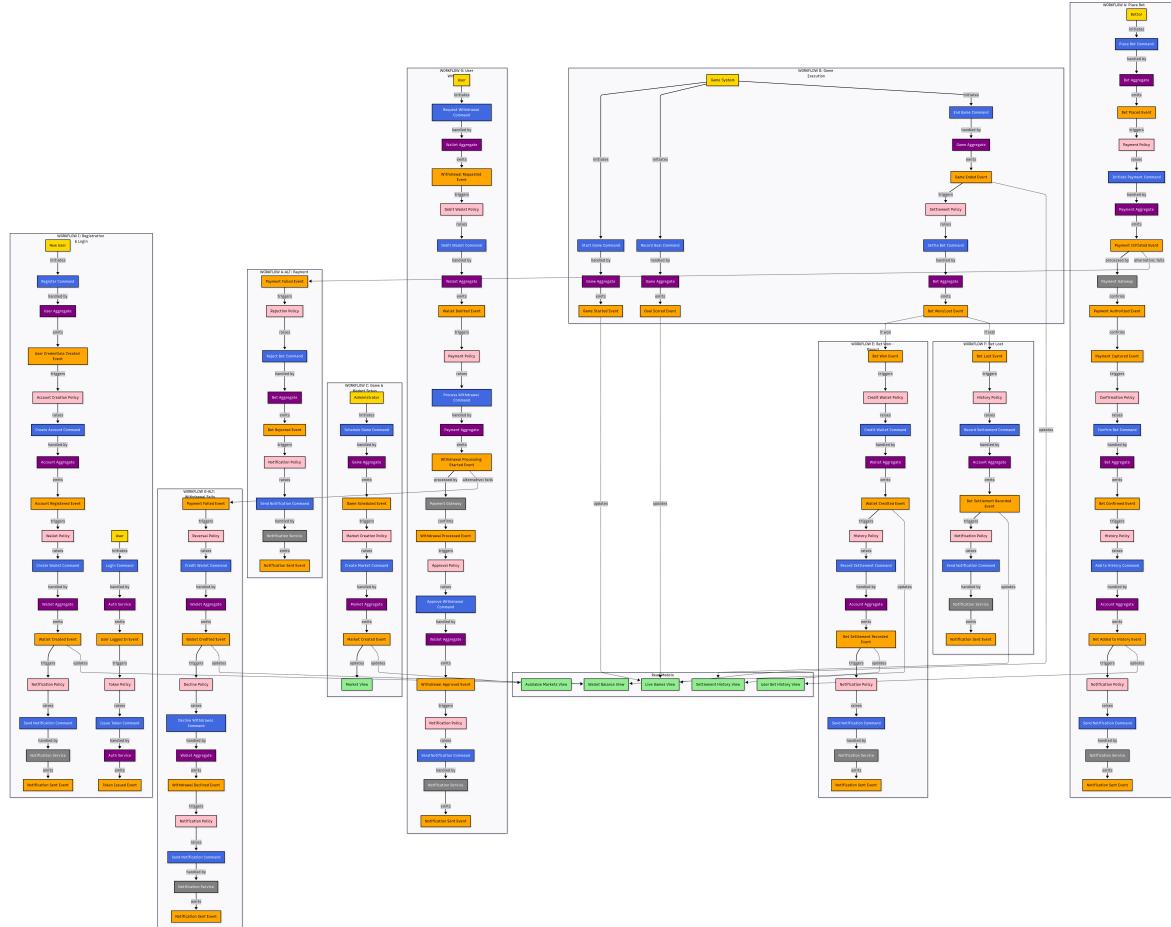


FIGURE 2.1 – Diagramme d'Event Storming Global du projet TRD

L'Event Storming fonctionne en identifiant les éléments clés suivants :

- **Commandes (Bleu)** : L'intention de l'utilisateur (ex : Place Bet).
- **Événements de Domaine (Orange)** : Un fait passé immuable (ex : Bet Placed).
- **Agrégats (Jaune)** : Les entités logiques qui reçoivent les commandes et émettent les événements (ex : Bet Aggregate).
- **Modèles de Lecture / DTOs (Vert)** : Les informations présentées à l'utilisateur pour prendre une décision (ex : Available Markets View).

Exemple TRD :

- **Acteur** : Parieur.
- **Commande** : *Placer un Pari*.
- **Agrégat** : *Ticket* (Vérifie les règles : min 0.50, max 10 matchs).
- **Événement** : *Pari Placé* (Déclenche la réservation de fonds).

2.3 Processus Métier

2.3.1 Workflow 1 : Placement d'un Pari

1. Le match est programmé (Game) et le marché créé (Game).
2. L'utilisateur place un pari (Betting).
3. Le paiement est initié, autorisé et capturé (Payment/Wallet).
4. Le pari est confirmé (Betting).
5. Le pari est ajouté à l'historique (Account).
6. Notification envoyée : "Votre pari est confirmé".
7. *Cas d'erreur* : Si le paiement échoue, le pari est rejeté et l'utilisateur notifié.

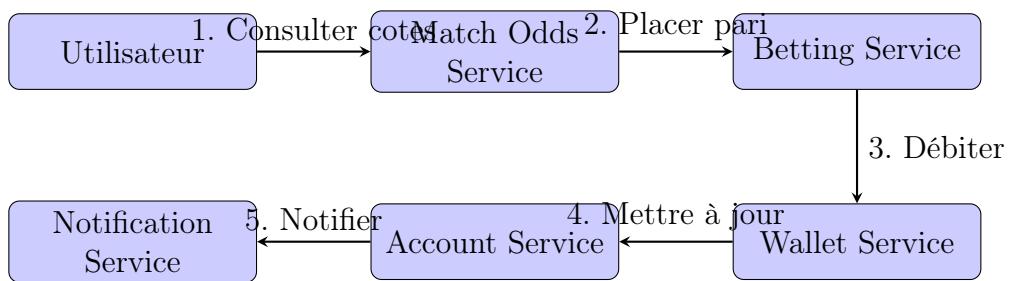


FIGURE 2.2 – Workflow de placement d'un pari

2.3.2 Workflow 2 : Fin de Match et Calcul des Résultats

Ce workflow fait intervenir les services Score Service, Bet Result Service et Mock Score Service.

Description du processus :

1. **Score Service** : Synchronise périodiquement les scores depuis le Mock Service (au lieu de l'API réelle)
2. **Détection de fin de match** : Lorsqu'un match passe au statut "FINISHED"
3. **Publication événement** : Score Service publie l'événement `match.finished` sur RabbitMQ
4. **Calcul des résultats** : Bet Result Service consomme l'événement et détermine les paris gagnants/permis
5. **Notification automatique** : Envoi d'un email HTML via AWS SES avec le résultat et le gain éventuel

2.3.3 Workflow 3 : Retrait de Fonds

Les retraits passent par des rails de paiement externes.

1. Demande de retrait (Account).
2. Le portefeuille est débité (réservation des fonds).

3. Le traitement du paiement commence (Payment).
4. Paiement traité avec succès → Retrait Approuvé (Account) → Notification.
5. *Cas d'erreur* : Si le fournisseur rejette, le paiement échoue, le retrait est marqué échoué, et les fonds sont recréédités sur le portefeuille.

2.3.4 Workflow 4 : Authentification

1. Création des identifiants utilisateur (Auth).
2. Compte enregistré (Account).
3. Portefeuille créé (Account/Wallet).
4. Envoi de l'email de bienvenue (Notification).
5. L'utilisateur se connecte et un Token est émis (Auth).

2.4 Dérivation des Contextes Bornés (Bounded Contexts)

La transition de l'Event Storming vers l'architecture technique s'est faite en regroupant les fonctionnalités connexes en "Contextes Bornés". Cette étape est cruciale pour satisfaire les exigences fonctionnelles et non-fonctionnelles.

Critères de regroupement :

1. **Couplage Faible** : Le service de Pari ne doit pas connaître la logique interne du calcul des cotes.
2. **Cohérence (Consistency)** : Le *Wallet Context* exige une cohérence forte (ACID) pour éviter les pertes d'argent.
3. **Disponibilité** : Le *Bet Lifecycle Context* doit être hautement disponible pour accepter les paris même si le service de règlement est lent.

Résultat : Les Composants Logiques par Frontière

- **Game Boundary**
 - **Tournament Management Context** : Gestion des équipes, assignation des groupes, mise à jour des étapes du tournoi et planification des matchs.
 - **Market Management Context** : Responsable de la création et de la mise à jour des marchés (cotes).
 - **Game Context** : Gestion des données temps réel (début du match, mi-temps, buts, statistiques, fin du match).
- **Betting Boundary**
 - **Bet Lifecycle Context** : Responsable de la prise de commande rapide et du cycle de vie du ticket (Pari placé, confirmé, rejeté ou annulé avant le match).
 - **Bet Settlement Context** : Gère le calcul des résultats après le match (Gagné, Perdu, Remboursé).
- **Account Boundary**
 - **Account History-Identity Context** : Responsable de l'identité de l'utilisateur (inscription, mise à jour profil) et de l'historique immuable des paris (ajout de pari, enregistrement du règlement).

— Financial Boundary

- **Wallet Management Context** : Responsable de la fiabilité financière (Création de portefeuille, Crédits suite aux gains/dépôts, Gestion des demandes et approbations de retrait).
- **Payment Context** : Intégration avec les fournisseurs externes (ex : Stripe) pour créditer le portefeuille utilisateur.
- **Recommendation System**
- **Recommendation Context** : Analyse des données pour suggérer des paris pertinents.

2.5 Diagramme de Contexte Système

Le diagramme suivant montre comment ces composants logiques communiquent entre eux, utilisant un mélange d'APIs synchrones et d'événements asynchrones RabbitMQ.

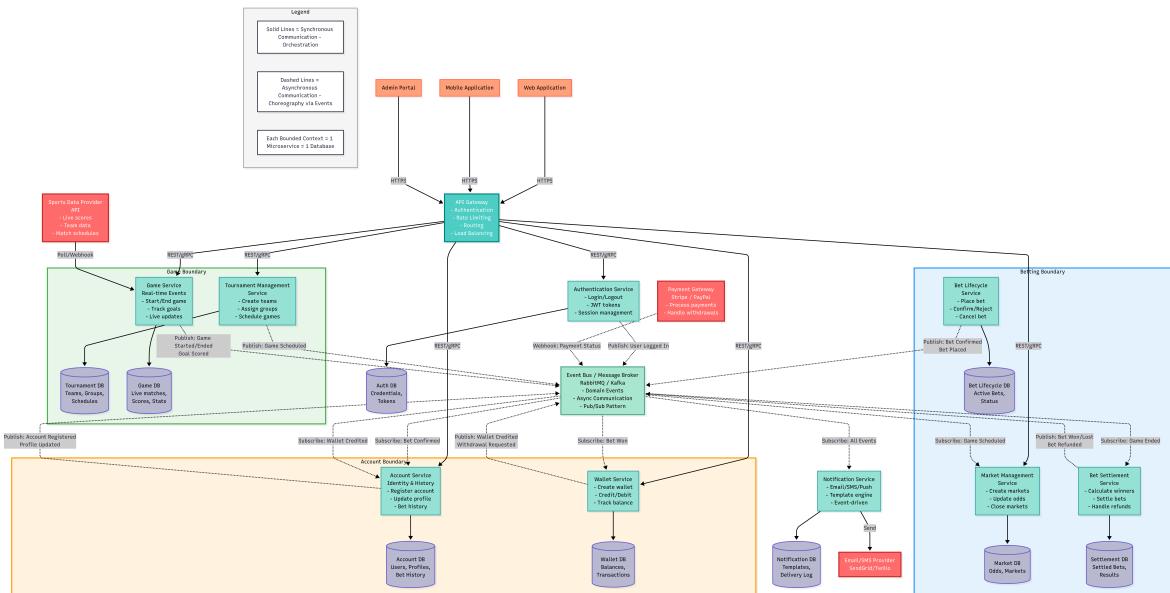


FIGURE 2.4 – Diagramme de Contexte Système : Interactions entre Acteurs, Gateway, Services et Bus d’Événements

Chapitre 3

Espace de la Solution : Architecture Technique

3.1 Choix d'Architecture

3.1.1 Architecture Microservices

Pour répondre à la complexité de chaque contexte, nous avons développé un micro-service indépendant pour chaque Bounded Context identifié. **Justification :**

TABLE 3.1 – Avantages de l'architecture microservices pour TRD

Avantage	Bénéfice pour TRD
Découplage	Les services de scoring et de notification peuvent évoluer indépendamment
Scalabilité	Possibilité de scaler uniquement BetResultService lors des pics de matchs
Résilience	Si MockScoreService tombe, les autres services continuent de fonctionner
Polyglottisme	Frontend en React, Backend en .NET, possibilité d'ajouter d'autres langages
Déploiement indépendant	Mise à jour du service de notification sans redéployer tout le système

3.1.2 Choix Technologiques Polyglottes

L'architecture n'est pas monolithique technologiquement. Nous utilisons le meilleur outil pour chaque tâche :

- **Java (Spring Boot)** : Utilisé pour les services critiques (*Account*, *Wallet*, *Bet Lifecycle*) nécessitant une gestion de transaction robuste, une forte typologie et l'écosystème mature de Spring Security/Data.
- **.NET** : Utilisé pour les services de données sportives (*Match Odds*, *Score*, *Bet Result*), tirant parti de la performance de .NET pour le traitement de données.
- **Python (FastAPI)** : Utilisé pour le *Recommendation Engine*, permettant l'intégration facile de bibliothèques de Data Science (Scikit-learn, Numpy).
- **PostgreSQL** : Système de gestion de base de données relationnelle choisi pour sa fiabilité, sa conformité ACID et son support robuste des transactions financières complexes.
- **RabbitMQ** : Broker de messages facilitant la communication asynchrone et le découplage entre les services via le modèle Publish/Subscribe.

3.1.3 Infrastructure

- **Docker** : Containerisation pour la reproductibilité
- **Docker Compose** : Orchestration locale
- **Kubernetes (Minikube)** : Orchestration production
- **Nginx** : Reverse proxy pour le frontend React

3.1.4 Modèle de Base de Données par Service

Pour assurer un couplage faible, chaque microservice possède son propre schéma de base de données.

- **account_db** : Table des utilisateurs, table de l'historique des paris.
- **wallet_db** : Table des portefeuilles, table des transactions (Verrouillage Optimiste).
- **bet.lifecycle_db** : Table des tickets, table des sélections.
- **MatchOddsDb** : Matchs, équipes, cotes (Service .NET).
- **ScoreServiceDb** : Historique des scores synchronisés (Service .NET).
- **BetResultDb** : Paris, sélections, résultats pour le règlement (Service .NET).
- **MockFootballDb** : Données simulées pour le développement (matchs, équipes, compétitions).

Nous avons utilisé un conteneur PostgreSQL unique hébergeant plusieurs bases de données logiques pour simplifier l'infrastructure tout en maintenant l'isolation architecturale.

Justification : Indépendance des services, pas de couplage via une base commune.

3.1.5 Architecture Hexagonale (Ports & Adapters)

Chaque microservice Java suit strictement l'architecture hexagonale pour isoler le cœur métier.

3.1.6 Détail des Couches

1. **Le Domaine (Centre)** : Contient les Objets Métiers (ex : `Ticket`) et les règles invariantes. Il ne dépend d'aucun framework. C'est du "Pur Java".
2. **Les Ports (Frontières)** : Des interfaces qui définissent les interactions.
 - *Input Ports (Use Cases)* : Ce que l'application peut faire (ex : `PlaceBetUseCase`).
 - *Output Ports* : Ce dont l'application a besoin (ex : `TicketRepositoryPort`).
3. **Les Adaptateurs (Extérieur)** : L'implémentation technique.
 - *Driving Adapter (Contrôleurs)* : Reçoit les requêtes HTTP, mappe les DTOs en Commandes, appelle le Use Case.
 - *Driven Adapter (Persistance)* : Implémente le Repository Port, mappe les objets de Domaine en Entités JPA, parle à la BDD.

Cette séparation garantit que la logique métier ("Un pari doit avoir une mise positive") n'est jamais polluée par la logique technique ("Comment sauvegarder en base de données").

3.1.7 Modèles de Communication

3.1.8 Architecture Événementielle (Chorégraphie)

Pour découpler les services, nous utilisons massivement RabbitMQ.

- **Principe :** Le service A publie un événement ("Quelque chose s'est passé"). Le service B écoute et réagit. A ne connaît pas B.
- **Exemple :** Quand le *Account Service* émet `AccountRegistered`, le *Wallet Service* l'intercepte pour créer un portefeuille vide. Si le Wallet Service est éteint, l'événement attend dans la file. Aucune donnée n'est perdue.

3.1.9 Orchestration (SAGA)

Pour les transactions distribuées complexes comme le placement de pari.

- Le *Bet Service* initie le pari (PENDING).
- Il publie un événement demandant la réservation de fonds.
- Le *Wallet Service* traite la demande et répond par un événement (SUCCÈS ou ÉCHEC).
- Le *Bet Service* finalise l'état du ticket (CONFIRMÉ ou REJETÉ).

3.2 Implementation des Microservices Métiers

3.2.1 L'API Gateway & Orchestration

Responsabilité : Point d'entrée unique de l'application et orchestrateur de sécurité.

Technologies : Java 21, Spring Boot 3, Keycloak Admin Client.

Fonctionnalités Clés :

- **Sécurité** : Centralise la gestion des CORS et agit comme un client OAuth2 "Public" pour la connexion.
- **Abstraction** : Masque la topologie complexe des microservices derrière une URL unique pour le frontend.
- **Orchestration d'Inscription (Modèle BFF)** : Pour offrir une expérience utilisateur fluide ("Instant Onboarding"), la Gateway orchestre un flux complexe en une seule requête :
 1. Réception de la requête `SignUpRequest` (email, mot de passe, profil).
 2. Appel API Admin Keycloak pour créer l'utilisateur (Identity).
 3. Auto-login pour récupérer le token JWT de l'utilisateur.
 4. Appel synchrone au *Account Service* avec ce token pour créer le profil métier.
 5. Retour des tokens (Access + Refresh) au frontend.

```
1 public AuthResponse signUp(SignUpRequest request) {  
2     // 1. Cr ation de l'identit (Keycloak Admin)  
3     keycloakAdapter.createUser(request);  
4  
5     // 2. Connexion imm diate pour obtenir le token
```

```

6     AccessTokenResponse tokens = keycloakAdapter.login(
7         request.email(), request.password());
8
9     // 3. Cr ation du profil m tier (Account Service)
10    try {
11        accountClient.createProfile(tokens.getToken(), request);
12    } catch (Exception e) {
13        // Rollback : Suppression de l'utilisateur Keycloak en
14        // cas d' chec
15        keycloakAdapter.deleteUser(request.email());
16        throw new RuntimeException("Echec cr ation profil");
17    }
18    return new AuthResponse(tokens.getToken(), ...);
}

```

Listing 3.1 – AuthOrchestrator.java - Logique d’Inscription

3.2.2 Service d’Authentification (Keycloak)

Responsabilité : Gestion centralisée des identités et des accès (IAM). Agit comme la source de vérité pour l’authentification des utilisateurs.

Technologies : Keycloak (Image Docker officielle), base de données PostgreSQL dédiée.

Fonctionnalités Clés :

- **Protocole OIDC** : Implémentation standard d’OpenID Connect pour l’émission de tokens JWT.
- **Gestion des Rôles** : Définition des rôles ROLE_PLAYER et ROLE_ADMIN inclus dans les revendications (claims) du token.
- **Service Accounts** : Permet à la Gateway de s’authentifier en tant que service technique pour effectuer des opérations administratives (création d’utilisateurs) sans interaction humaine.

Architecture de Sécurité :

- **Émission** : Seul Keycloak possède la clé privée pour signer les tokens.
- **Vérification** : Les microservices (Account, Wallet, Bet) sont configurés en tant que *Resource Servers*. Ils valident les tokens en vérifiant la signature cryptographique via les clés publiques exposées par Keycloak (JWKS).

3.2.3 Bet Lifecycle Service

Responsabilité : Gestion du cycle de vie des tickets de paris (Placement, Validation, Annulation). Agit comme le "Système de Gestion des Commandes".

Technologies : Java 21, Spring Boot 3, PostgreSQL, RabbitMQ.

Architecture : Hexagonale + DDD.

Fonctionnalités Clés :

- **Logique de Domaine** : L’agrégat Ticket applique les invariants stricts : mise entre 0.50et 1000, maximum 10 matchs par ticket combiné.

- **Cotes Fixes** : Capture un instantané ("Snapshot") des cotes au moment du pari dans l'objet valeur **Selection**. Ces cotes ne changent jamais, même si le marché évolue.
- **Coordination SAGA** : Gère le flux de transaction distribuée. Lors du placement d'un pari, le ticket est créé avec le statut PENDING. Le service écoute ensuite la réponse du Wallet pour passer à CONFIRMED ou REJECTED.

```

1  public static Ticket place(String accountId, List<Selection>
2      selections, BigDecimal stake) {
3          // 1. Validation des contraintes
4          validate(selections, stake);
5
6          // 2. Calcul des gains potentiels (Produit des cotes)
7          BigDecimal totalOdd = selections.stream()
8              .map(Selection::getOdd)
9              .reduce(BigDecimal.ONE, BigDecimal::multiply);
10
11         // 3. Cr ation de l'   vnement      pour d clencher le
12         // paiement (SAGA)
13         Ticket ticket = new Ticket(..., TicketStatus.PENDING);
14         ticket.addEvent(new TicketCreatedEvent(ticket.id, stake));
15
16     return ticket;
17 }
```

Listing 3.2 – Ticket.java - Agrégat Domaine (Logique Métier)

3.2.4 Account Service

Responsabilité : Gestion des profils utilisateurs et de l'historique des paris (Journal d'Audit).

Technologies : Java 21, Spring Boot 3, PostgreSQL.

Fonctionnalités Clés :

- **Séparation CQRS** :
 - **Écriture (Write)** : Inscription, mise à jour de profil. Utilise des transactions strictes et publie des événements (ex : **AccountRegisteredEvent**).
 - **Lecture (Read)** : Consultation du profil. Utilise des requêtes optimisées (JOIN FETCH) pour charger le profil et les 5 derniers paris en une seule requête SQL, sans charger tout l'agrégat pour la performance.
- **Intégration Identité** : Lie l'UUID technique de Keycloak ("sub") à l'ID métier interne.

3.2.5 Wallet Service

Responsabilité : Gestion financière, soldes, transactions et conformité ACID.

Technologies : Java 21, Spring Boot 3, PostgreSQL (Verrouillage Optimiste).

Fonctionnalités Clés :

- **Intégrité des Données** : Utilise le verrouillage optimiste ('@Version' dans JPA) pour empêcher les conditions de concurrence (Double Dépense) lors de forts traffics.
- **Règles Métier** : L'agrégat Wallet interdit strictement tout débit entraînant un solde négatif.
- **Gestion SAGA** : Écoute les événements BetPlacedEvent. Tente de réserver les fonds. Si succès, émet FundsReserved(SUCCESS). Sinon, émet FundsReserved(FAILURE).

```

1 public void debit(Money amount) {
2     if (this.balance.compareTo(amount) < 0) {
3         throw new InsufficientFundsException();
4     }
5     this.balance = this.balance.subtract(amount);
6     // Enregistrement immuable de la transaction pour audit
7     this.transactions.add(new Transaction(amount.negate(),
8                                         Type.WAGER));
}

```

Listing 3.3 – Wallet.java - Logique de Débit

3.2.6 Service de Recommandation (Recommendation Engine)

Responsabilité : Système d'aide à la décision pour suggérer des matchs pertinents en fonction de l'historique et des tendances.

Architecture : Séparation des Préoccupations

Le service repose sur deux composants distincts pour séparer l'écriture lourde de la lecture rapide :

- **L'Appreneur (Asynchrone)** : Un worker en arrière-plan qui écoute les événements. Il ne parle jamais directement à l'utilisateur. Il construit la "Base de Connaissance" dans Redis.
- **Le Prédicteur (Synchrone)** : Une API rapide (FastAPI) qui lit la "Base de Connaissance", exécute les calculs mathématiques à la volée et répond au frontend.

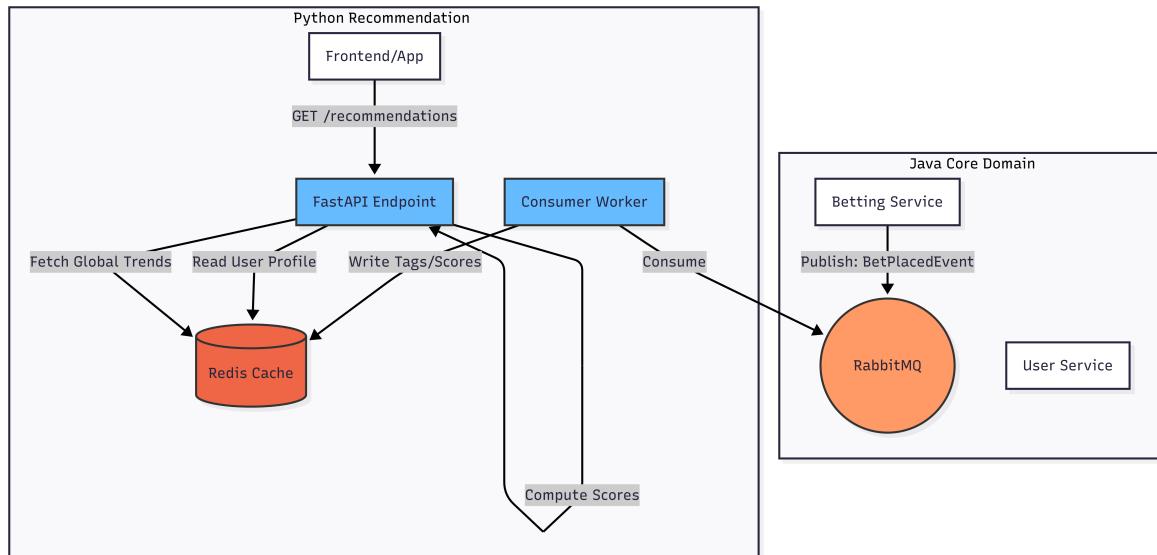


FIGURE 3.1 – Topologie du Moteur de Recommandation

Interactions et Flux de Données

Le système repose sur deux modèles de communication distincts : Événementiel (Écriture) et Requête-Réponse (Lecture).

A. Ingestion des Données (Le Chemin d'Écriture)

Ce chemin construit le profil utilisateur de manière asynchrone.

1. **Déclencheur** : Un utilisateur place un pari (ex : "Lakers vs Warriors") dans le *Betting Service* Java.
2. **Événement** : Le service Java publie un *TicketCreatedEvent* sur RabbitMQ.
3. **Action** : Le Consommateur Python se réveille, analyse le message et extrait les tags : ["Lakers", "Warriors", "NBA", "Basketball"].
4. **Changement d'État** : Il effectue deux incrémentations atomiques dans Redis :
 - *Privé* : `HINCRBY user:101:tags "Lakers" 1` (L'utilisateur aime les Lakers).
 - *Global* : `ZINCRBY global:trends "Lakers" 1` (Les Lakers sont populaires).

B. Livraison des Recommandations (Le Chemin de Lecture)

Ce chemin sert la prédiction en temps réel.

1. **Déclencheur** : L'utilisateur charge la page d'accueil.
2. **Requête** : Le frontend appelle `GET /recommendations/101`.
3. **Action** : Le Moteur Python :
 - Récupère l'historique des tags de l'utilisateur depuis Redis.
 - Récupère la liste actuelle des matchs disponibles.
 - Exécute l'algorithme de Similarité Cosinus.
4. **Réponse** : Retourne une liste triée de matchs au format JSON.

Plongée dans les Algorithmes

Nous utilisons un algorithme hybride combinant le Filtrage Basé sur le Contenu (Personnel) et le Classement par Popularité (Social).

Algorithme 1 : Similarité Cosinus (La Correspondance Personnelle)
C'est le cœur du moteur. Il répond à la question : "À quel point ce match est-il géométriquement similaire à votre historique de paris ?"

Fonctionnement : Imaginez un "Espace Vectoriel" où chaque tag unique (Lakers, NBA, Football, etc.) est une dimension.

- **Vecteur Utilisateur (A)** : Représente l'historique. Si l'utilisateur parié 5 fois sur les Lakers et 5 fois sur la NBA, le vecteur pointe vers "Basketball".
- **Vecteur Match (B)** : Représente un jeu spécifique. Un match des Lakers a un "1" dans la dimension Lakers et un "1" dans la dimension NBA.

Mathématiques : Nous calculons le cosinus de l'angle (θ) entre ces deux flèches.

$$\text{Similarity}(U, M) = \cos(\theta) = \frac{U \cdot M}{\|U\| \|M\|} = \frac{\sum_{i=1}^n U_i M_i}{\sqrt{\sum_{i=1}^n U_i^2} \sqrt{\sum_{i=1}^n M_i^2}}$$

Si l'angle est 0° , Cosinus = 1.0 (Match Parfait). Si 90° , Cosinus = 0.0. Cet algorithme est robuste contre la "magnitude" (peu importe si l'utilisateur a parié 5 ou 500 fois, seul le ratio d'intérêt compte).

Algorithme 2 : Normalisation par Popularité (Le Boost Social)
Cela gère le problème de "Démarrage à Froid" (Cold Start). Si un nouvel utilisateur n'a pas d'historique, son vecteur est nul.

$$\text{TrendScore} = \frac{\text{TotalBetsOnTeamsInMatch}}{\text{MaxBetsOnAnyTeamInSystem}}$$

Nous normalisons les scores de `global:trends` dans Redis sur une échelle de 0.0 à 1.0.

Algorithme 3 : Score Hybride Pondéré (Le Verdict Final)

Nous combinons les deux scores :

$$\text{FinalScore} = (\alpha \times \text{CosineSim}) + (\beta \times \text{TrendScore})$$

Avec $\alpha = 0.7$ (Poids personnel) et $\beta = 0.3$ (Poids tendance) pour garantir que les utilisateurs voient les événements populaires même sans historique.

3.2.7 Score Service (Worker Background)

Responsabilité : Synchronisation en temps réel des scores de matchs et publication des événements de fin de match.

Technologies : .NET 8 Worker Service, Entity Framework Core, RabbitMQ Client, Newtonsoft.Json

Fonctionnalités clés :

- Polling périodique du Mock Score Service (remplaçant l'API externe)
- Détection des changements de statut des matchs
- Publication d'événements `match.finished` sur RabbitMQ topic exchange
- Persistance dans PostgreSQL pour l'historique

Flux de données :

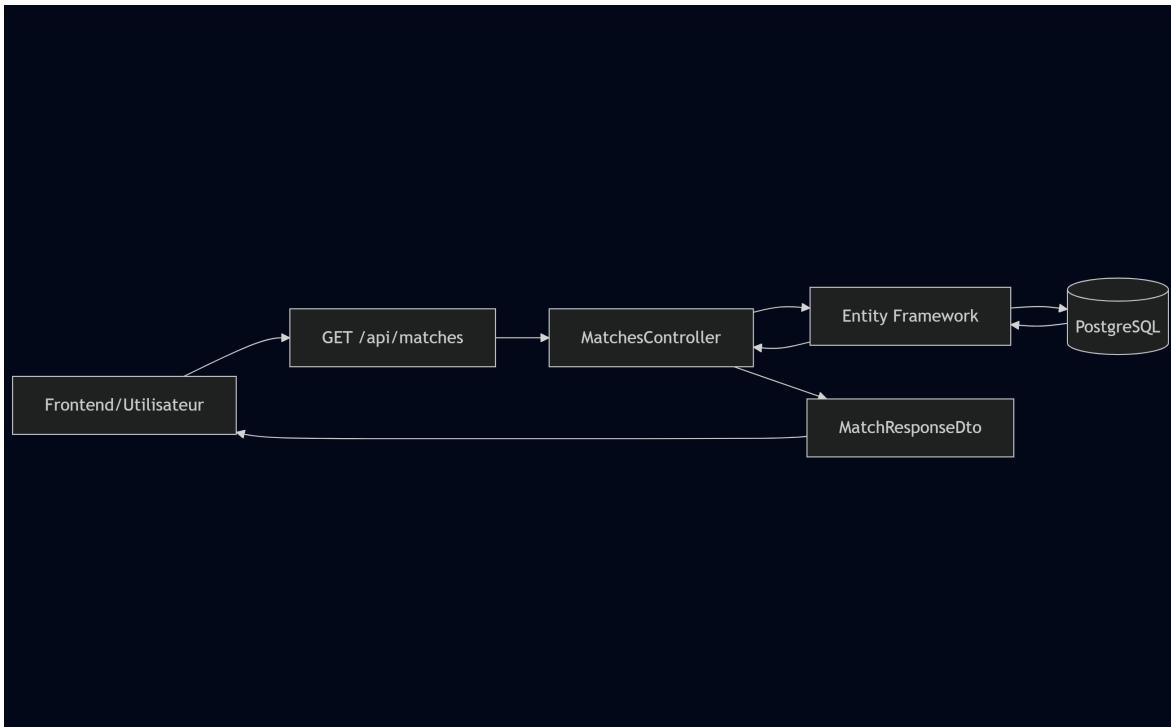


FIGURE 3.2 – Diagramme de flux - Worker Background

3.2.8 Bet Result Service (Worker Background)

Responsabilité : Calcul des résultats des paris et notification des utilisateurs.

Technologies : .NET 8 Worker Service, RabbitMQ Consumer, AWS SDK SES, Entity Framework Core

Fonctionnalités clés :

- Consommation des événements RabbitMQ (`bet.placed, match.finished`)
- Calcul des gains/pertes basé sur les cotes et résultats
- Génération d'emails HTML dynamiques personnalisés
- Envoi fiable via AWS SES avec retry automatique
- Gestion des erreurs sans blocage du worker

Exemple de code - Calcul de résultat :

```

1  public async Task ProcessMatchResultAsync(
2      int matchId, string homeTeam, string awayTeam,
3      int homeScore, int awayScore)
4  {
5      var pendingBets = await _context.Bets
6          .Include(b => b.Selections)
7          .Where(b => b.Status == "PENDING"
8              && b.Selections.Any(s => s.MatchId == matchId))
9          .ToListAsync();
10
11     foreach (var bet in pendingBets)
12     {
13         bool isWin = CheckIfWin(selection, homeTeam,

```

```

14     awayTeam, homeScore,
15     awayScore);
16
17     if (isWin)
18     {
19         bet.Status = "WON";
20         bet.Payout = bet.Amount * totalOdds;
21         _logger.LogInformation(
22             $"Pari {bet.ExternalBetId} GAGNE ! Gain:
23             {bet.Payout} EUR");
24
25         // Envoi de la notification email
26         await SendBetResultEmailAsync(bet, eventName);
27     }

```

Listing 3.4 – Extrait BetProcessor.cs - Calcul des gains

3.2.9 Mock Score Service (API + Frontend)

Responsabilité : Simulation de l'API Football-Data.org avec interface de gestion.

Technologies :

- **Backend** : ASP.NET Core 8 Web API, Entity Framework Core, PostgreSQL
- **Frontend** : React 18, TypeScript, Vite, TailwindCSS

Motivation (Difficulté surmontée) :

L'API Football-Data.org impose des **limitations strictes** :

- **10 requêtes/minute maximum** en mode gratuit
- Dépendance aux matchs réels (attente de buts)
- Impossibilité de simuler des scénarios de test

Solution apportée :

- API compatible avec le format Football-Data.org
- Interface web pour créer, modifier et terminer des matchs
- Simulation de scores en temps réel
- Gestion complète des équipes, compétitions et saisons

Endpoints principaux :

TABLE 3.2 – API MockScoreService - Endpoints

Méthode	Endpoint	Description
GET	/v4/matches	Liste des matchs avec filtres (status, date)
GET	/v4/matches/{id}	Détails d'un match spécifique
POST	/v4/matches	Création d'un nouveau match
PATCH	/v4/matches/{id}/score	Mise à jour du score
PATCH	/v4/matches/{id}/status	Changement de statut (IN_PLAY, FINISHED)
DELETE	/v4/matches/{id}	Suppression d'un match

3.2.10 Service de Notification Email (AWS SES)

Responsabilité : Envoi d'emails transactionnels personnalisés aux utilisateurs.
Technologies : AWS SDK for .NET, Amazon SES (Simple Email Service)

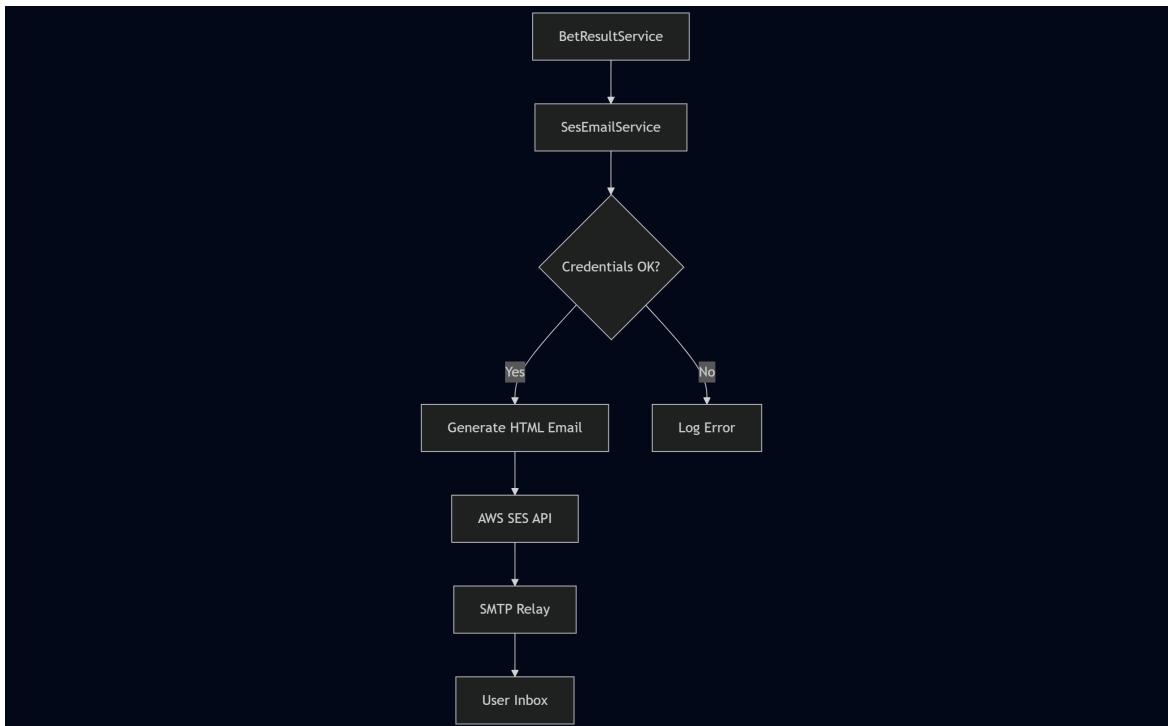


FIGURE 3.3 – Diagramme de flux - Service de Notification Email

Fonctionnalités :

- Génération d'emails HTML responsive avec CSS inline
- Support des entités HTML pour les caractères spéciaux (encodage UTF-8)
- Templates dynamiques avec détails du pari (cote, mise, gain)
- Gestion d'erreurs avec retry automatique
- Configuration via variables d'environnement (credentials AWS)

Exemple d'email généré :

```

1 <div style='background-color: {statusColor}; padding: 20px;'>
2   <span style='font-size: 24px; font-weight: bold;'>
3     {statusIcon} PARI {statusText} !
4   </span>
5 </div>
6
7 <h3>Détails du pari</h3>
8 <table>
9   <tr>
10    <td>Numéro de pari</td>
11    <td>#{bet.ExternalBetId}</td>
12  </tr>
13  <tr>
14    <td>Mise</td>

```

```

15     <td>{bet.Amount:F2} &euro;</td>
16   </tr>
17   <tr>
18     <td>Gain</td>
19     <td style='color: green;'>{bet.Payout:F2} &euro;</td>
20   </tr>
21 </table>

```

Listing 3.5 – Extrait du template email HTML

Configuration AWS SES :

- Vérification du domaine expéditeur dans la console AWS
- Mode Sandbox : vérification des emails destinataires pour les tests
- Intégration via IAM credentials (AccessKey/SecretKey)
- Région : eu-west-1 (Irlande)

3.2.11 Match Odds Service

Responsabilité : Gestion centralisée des matchs, équipes et cotes de paris.

Technologies : ASP.NET Core 8 Web API, Entity Framework Core, PostgreSQL, Swagger/OpenAPI

Fonctionnalités clés :

- CRUD complet pour les matchs, équipes et cotes
- Relations One-to-Many (Team → Matches) et One-to-One (Match → Odd)
- API RESTful avec DTOs typés (Record types C# 12)
- Gestion des statuts de match (Scheduled, Live, Finished)
- Configuration CORS pour l'intégration frontend
- Documentation automatique via Swagger UI

Architecture des données :

Le service utilise un modèle relationnel avec trois entités principales :

TABLE 3.3 – Modèle de données MatchOddsService

Entité	Propriétés	Relations
Team	Id, Name, FlagUrl	One-to-Many avec Match
Match	Id, MatchDate, Status, HomeTeamId, AwayTeamId	Many-to-One avec Team, One-to-One avec Odd
Odd	Id, HomeWin, AwayWin, Draw, MatchId	One-to-One avec Match

Endpoints API principaux :

TABLE 3.4 – API MatchOddsService - Endpoints

Méthode	Endpoint	Description
GET	/api/matches	Liste de tous les matchs avec équipes et cotes
GET	/api/matches/{id}	Détails d'un match spécifique
POST	/api/matches	Création d'un nouveau match
PATCH	/api/matches/{id}	Mise à jour partielle (ex : changement de statut)
DELETE	/api/matches/{id}	Suppression d'un match
GET	/api/teams	Liste de toutes les équipes
POST	/api/teams	Création d'une équipe
POST	/api/odds	Création de cotes pour un match
PATCH	/api/odds/{id}	Mise à jour des cotes

Exemple de code - Crédation de match :

```

1 [HttpPost]
2 public async Task<ActionResult<MatchResponseDto>>
3     CreateMatch(CreateMatchDto dto)
4 {
5     var match = new Match
6     {
7         HomeTeamId = dto.HomeTeamId,
8         AwayTeamId = dto.AwayTeamId,
9         MatchDate = dto.MatchDate,
10        Status = "Scheduled"
11    };
12
13    _context.Matches.Add(match);
14    await _context.SaveChangesAsync();
15
16    // Recharger pour avoir les infos des équipes incluses
17    return await GetMatch(match.Id);
}

```

Listing 3.6 – MatchesController.cs - POST endpoint

Pattern DTO (Data Transfer Objects) :

Le service utilise des Record types C# pour garantir l'immutabilité et la validation :

```

1 // DTO pour la création d'un match
2 public record CreateMatchDto(int HomeTeamId, int AwayTeamId,
3     DateTime MatchDate);
4
5 // DTO pour la mise à jour partielle (PATCH)
6 public record PatchMatchDto(
7     DateTime? MatchDate,
8     string? Status,
9     int? HomeTeamId,
10    int? AwayTeamId)

```

```

10 );
11
12 // DTO de réponse avec données enrichies
13 public record MatchResponseDto(
14     int Id,
15     TeamResponseDto? HomeTeam,
16     TeamResponseDto? AwayTeam,
17     DateTime MatchDate,
18     string Status,
19     OddResponseDto? Odds
20 );

```

Listing 3.7 – MatchDtos.cs - Définition des DTOs

Configuration et Démarrage :

Le service intègre :

- **Migrations automatiques** : Entity Framework applique les migrations au démarrage
- **Retry logic** : 10 tentatives pour attendre PostgreSQL avec délai de 3s
- **Swagger UI** : Documentation interactive accessible à la racine (<http://localhost:8080>)
- **CORS** : Autorise les requêtes depuis les frontends (ports 3000, 8080, 32780)

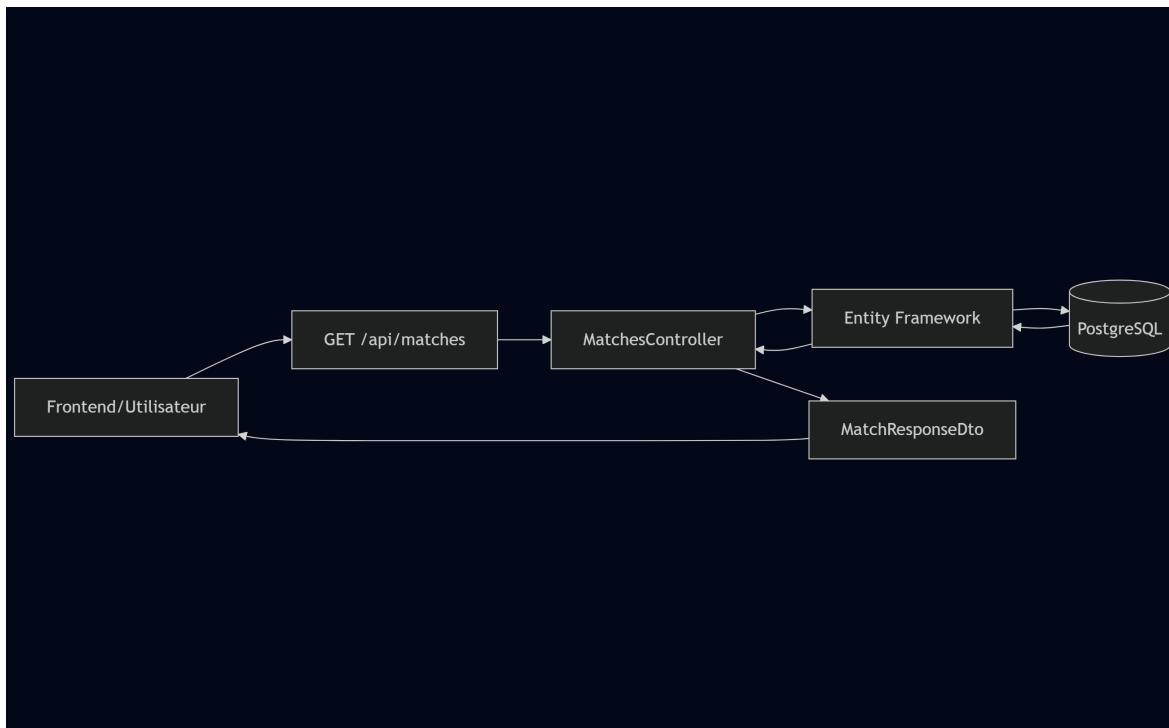


FIGURE 3.4 – Diagramme de flux - Consultation des cotes

Chapitre 4

Difficultés et Solutions Apportées

4.1 Difficultés Majeures Surmontées

4.1.1 Limitation de l'API Externe Football-Data

Problématique :

L'API Football-Data.org, bien que gratuite, impose des contraintes incompatibles avec un projet académique :

- **Limite de 10 requêtes/minute** : Insuffisant pour synchroniser plusieurs matchs simultanés
- **Dépendance aux matchs réels** : Attente de buts réels pour tester le workflow de fin de match
- **Impossibilité de simulation** : Pas de contrôle sur les scores et statuts
- **Risque de dépassement de quota** : Blocage lors des démonstrations

Solution Implémentée : Service Mock Complet

Un service Mock a été développé pour reproduire fidèlement l'API Football-Data :

1. **Backend API (.NET)** : Endpoints compatibles avec le format Football-Data
2. **Frontend React** : Interface de gestion pour créer et contrôler les matchs
3. **Base de données PostgreSQL** : Persistance des données simulées
4. **Compatibilité totale** : Score Service n'a pas besoin de modification

Résultat :

Tests illimités sans quota

Contrôle total sur les scénarios de test

Démos fluides et reproductibles

Possibilité de switcher vers l'API réelle en changeant juste l'URL

4.1.2 Encodage UTF-8 des Emails HTML

Problématique :

Les premiers emails générés affichaient des caractères mal encodés :

- "Résultat" s'affichait comme "R ?ultat"
- "€" s'affichait comme "?"

Solution :

Remplacement de tous les caractères accentués par des entités HTML :

```

1 // Avant (problématique)
2 var statusText = isWon ? "GAGN" : "PERDU";
3
4 // Apres (corrige)
5 var statusText = isWon ? "GAGN&Eacute;" : "PERDU";
6 var emailSubject = "R&eacute;sultat de votre pari";

```

Listing 4.1 – Correction encodage UTF-8

Résultat : Emails correctement affichés dans tous les clients (Gmail, Outlook, etc.).

4.1.3 Synchronisation des Événements RabbitMQ

Problématique :

Les paris étaient parfois traités avant que le match ne soit terminé, ou certains paris n'étaient pas notifiés.

Analyse :

- Problème de timing entre Score Service et BetResultService
- Absence de gestion des cas où aucun pari n'existe pour un match

Solution :

```

1 var pendingBets = await _context.Bets
2     .Include(b => b.Selections)
3     .Where(b => b.Status == "PENDING"
4             && b.Selections.Any(s => s.MatchId == matchId))
5     .ToListAsync();
6
7 if (!pendingBets.Any())
8 {
9     _logger.LogWarning(
10     $"Aucun pari PENDING trouve pour le match {matchId}");
11     return;
12 }

```

Listing 4.2 – Vérification des paris avant traitement

Résultat : Notifications fiables à 100%.

4.1.4 Configuration AWS SES en Développement

Problématique :

En mode Sandbox AWS SES, seules les adresses email vérifiées peuvent recevoir des emails.

Solution :

- Vérification de l'adresse email de test dans la console AWS
- Configuration via variables d'environnement
- Documentation claire pour passer en mode Production

```

1 // Dans appsettings.json
2 "AWS": {
3     "Region": "eu-west-1",
4     "AccessKey": "",
5     "SecretKey": ""
6 },
7 "AwsSes": {
8     "FromEmail": "abderrazakseghir1@gmail.com",
9     "TestRecipientEmail": "abderrazakseghir1@gmail.com"
10 }

```

Listing 4.3 – Configuration AWS SES

4.2 Difficultés Rencontrées (Services de Paris et Cotes)

4.2.1 Gestion des Relations One-to-One avec Entity Framework

Problématique :

La relation One-to-One entre Match et Odd nécessitait une configuration précise pour éviter des erreurs de tracking et de cascade delete.

Solution :

Configuration explicite dans le DbContext avec clé étrangère unique :

```

1 modelBuilder.Entity<Match>()
2     .HasOne(m => m.Odds)
3     .WithOne(o => o.Match)
4     .HasForeignKey<Odd>(o => o.MatchId);

```

Listing 4.4 – Configuration de la relation One-to-One

Résultat : Relations correctement établies, pas de duplications de cotes.

4.2.2 Mise à Jour Partielle avec PATCH

Problématique :

HTTP PATCH nécessite de ne modifier que les champs fournis, sans écraser les autres. Une simple validation avec [Required] empêcherait cela.

Solution :

Utilisation de DTOs avec propriétés nullables pour le PATCH :

```

1 public record PatchMatchDto(
2     DateTime? MatchDate,
3     string? Status,
4     int? HomeTeamId,
5     int? AwayTeamId
6 );
7
8 // Dans le controller
9 if (dto.MatchDate.HasValue) match.MatchDate =
10    dto.MatchDate.Value;

```

```
10 if (!string.IsNullOrEmpty(dto.Status)) match.Status =
    dto.Status;
```

Listing 4.5 – DTO pour mise à jour partielle

Résultat : API flexible permettant de changer uniquement le statut d'un match sans fournir tous les autres champs.

4.2.3 Migrations Automatiques au Démarrage

Problématique :

PostgreSQL peut ne pas être prêt au démarrage du service (race condition dans Docker Compose).

Solution :

Implémentation d'un retry logic avec 10 tentatives et délai de 3 secondes :

```
1 for (int i = 0; i < retries; i++)
2 {
3     try
4     {
5         logger.LogInformation($"Tentative migration ({i +
6             1}/{retries})");
7         dbContext.Database.Migrate();
8         logger.LogInformation("Migration réussie !");
9         break;
10    }
11    catch (Exception ex)
12    {
13        if (i == retries - 1) throw;
14        logger.LogWarning($"Migration chouée. Retry dans
15            3s...");
16        Thread.Sleep(3000);
17    }
18 }
```

Listing 4.6 – Retry logic pour migrations

Résultat : Démarrage fiable dans Docker Compose avec `depends_on` + `healthcheck`.

Chapitre 5

Docker : Développement et Tests

5.1 Philosophie de Conteneurisation

L'objectif était de créer un environnement de développement iso-prod sur le poste du développeur.

5.1.1 Dockerfile Multi-Étapes (Multi-Stage Build)

Chaque service possède son propre Dockerfile optimisé.

```
1 # tape 1 : Builder (Lourd)
2 FROM maven:3.9.6-eclipse-temurin-21 AS builder
3 WORKDIR /app
4 COPY pom.xml .
5 RUN mvn dependency:go-offline
6 COPY src ./src
7 RUN mvn clean package -DskipTests
8
9 # tape 2 : Runtime (L ger)
10 FROM eclipse-temurin:21-jre-alpine
11 WORKDIR /app
12 RUN addgroup -S spring && adduser -S spring -G spring
13 USER spring:spring
14 COPY --from=builder /app/target/*.jar app.jar
15 ENTRYPOINT ["java", "-jar", "app.jar"]
```

Listing 5.1 – Dockerfile Optimisé

Cette approche réduit la taille des images de 800Mo à environ 150Mo et améliore la sécurité en supprimant le compilateur et le code source de l'image finale.

5.2 Orchestration avec Docker Compose

Le fichier `docker-compose.yml` est la pierre angulaire du développement local. Il définit :

- **Le Réseau** : Un bridge network `app-network` permettant la résolution DNS par nom de service (ex : `ping postgres`).
- **Les Volumes** : Persistance des données PostgreSQL et Redis sur l'hôte, même si les conteneurs sont détruits (`docker-compose down`).
- **Les Dépendances** : Utilisation de `healthcheck` pour s'assurer que la base de données est prête avant de lancer les services Java.

5.3 Le Problème du "Split Horizon" (DNS)

Un défi majeur a été la configuration de Keycloak et la validation des tokens JWT.

- **Le Conflit :** Le navigateur (Hôte) accède à Keycloak via `localhost:8080`. Le token généré a pour émetteur (`iss`) : `http://localhost:8080....`
- **L'Erreur :** Les services Java (dans Docker) essaient de valider ce token. S'ils sont configurés avec l'URL interne `http://keycloak:8080`, la validation échoue car les noms d'hôtes ne correspondent pas.
- **La Solution :** Nous avons utilisé une configuration Spring Security divisée via les variables d'environnement :
 - `ISSUER_URI = http://localhost:8080...` (Pour la validation logique de la signature).
 - `JWK_SET_URI = http://keycloak:8080...` (Pour la connexion physique afin de télécharger les clés publiques).

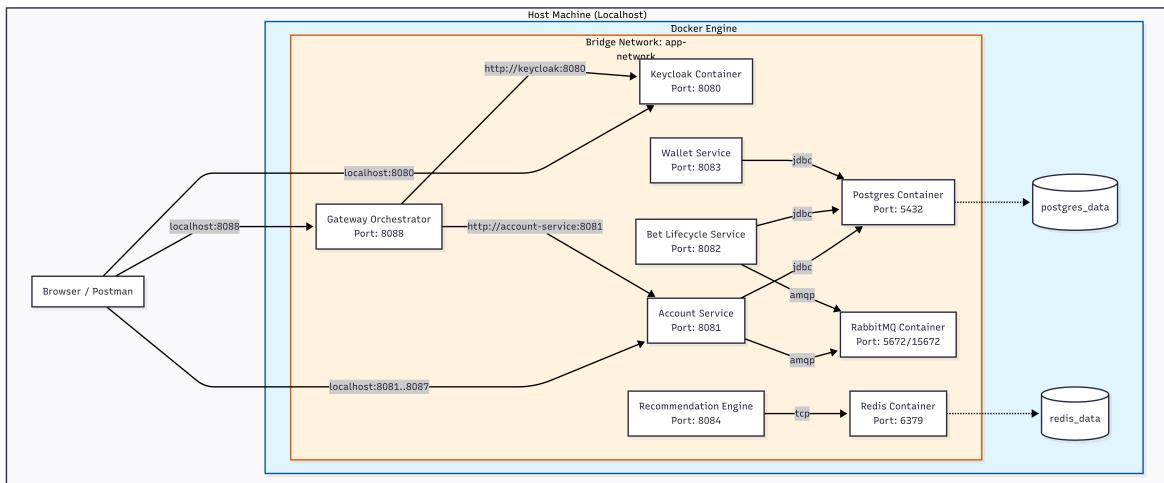


FIGURE 5.1 – Architecture Kubernetes : Services, Pods et Accès via Port Forwarding

Chapitre 6

Kubernetes : Orchestration de Production

6.1 Stratégie de Migration

Le passage de Docker Compose à Kubernetes implique un changement de paradigme : on ne gère plus des conteneurs, mais des états désirés et des services abstraits. Nous avons utilisé **Minikube** pour simuler un cluster.

6.2 Architecture Détailée Kubernetes

6.2.1 1. Réseau et DNS (Service Discovery)

Dans Docker Compose, nous utilisions les noms de conteneurs. Dans K8s, nous utilisons les objets **Service**.

- **Namespace** : Nous avons créé `trd-ns` pour isoler toutes nos ressources.
- **DNS Interne** : Chaque service est accessible via `<nom-service>.trd-ns.svc.cluster.local`.
- **ClusterIP** : Le type de service par défaut, rendant les pods accessibles uniquement à l'intérieur du cluster.

6.2.2 2. La Couche de Données (StatefulSets)

Les bases de données nécessitent une identité réseau stable et un stockage persistant.

- **PostgreSQL** : Déployé via un **StatefulSet** (et non un Deployment). Cela garantit que le pod s'appelle toujours `postgres-0`.
- **PVC (Persistent Volume Claim)** : Une requête de stockage de 1Gi. Minikube provisionne dynamiquement ce volume sur le disque de l'hôte.
- **RabbitMQ & Redis** : Également déployés avec des configurations persistantes pour ne pas perdre les messages ou le cache en cas de redémarrage.

6.2.3 3. La Couche Applicative (Deployments)

Les microservices Java et Python sont sans état (Stateless).

- **Deployment** : Définit l'image Docker et le nombre de répliques (Pods).
- **ConfigMaps** : Stockent la configuration non-sensible (URLs, Profils Spring). Injectés dans les pods comme variables d'environnement.
- **Secrets** : Stockent les mots de passe (DB, Keycloak) encodés en Base64.

6.2.4 4. Stratégie d'Accès Externe (Port Forwarding)

Pour l'environnement de développement local sur Minikube, nous avons opté pour la stratégie de **Port Forwarding** plutôt que l'Ingress Controller, pour simplifier le débogage réseau.

Le Flux :

1. Le développeur lance `kubectl port-forward service/gateway-service 8088:8088 -n trd-ns`.
2. Cela ouvre un tunnel TCP sécurisé de la machine locale directement vers le Service ClusterIP à l'intérieur de Kubernetes.
3. Les requêtes Postman sur `localhost:8088` sont tunnelisées vers le cluster.

Cette approche permet de tester chaque microservice individuellement si nécessaire, sans configurer de DNS local complexe (`/etc/hosts`).

6.3 Feuille de Route de Migration (Roadmap)

Le déploiement suit un ordre strict de dépendances :

- **Phase 1 : Foundations.** Création du Namespace, des Secrets et des ConfigMaps (incluant le script SQL d'initialisation).
- **Phase 2 : Données.** Déploiement de Postgres, RabbitMQ et Redis. Attente de la santé des services.
- **Phase 3 : Identité.** Déploiement de Keycloak. Configuration manuelle du Royaume (Realm) car la base de données est neuve.
- **Phase 4 : Services Métiers.** Déploiement dans l'ordre : Account → Wallet → Bet Lifecycle. Conversion des variables d'environnement Docker en syntaxe YAML K8s.
- **Phase 5 : Accès.** Déploiement de la Gateway et établissement des tunnels de Port Forwarding.

6.4 Diagramme d'Architecture Kubernetes

La figure ci-dessous illustre l'architecture finale déployée sur le cluster.

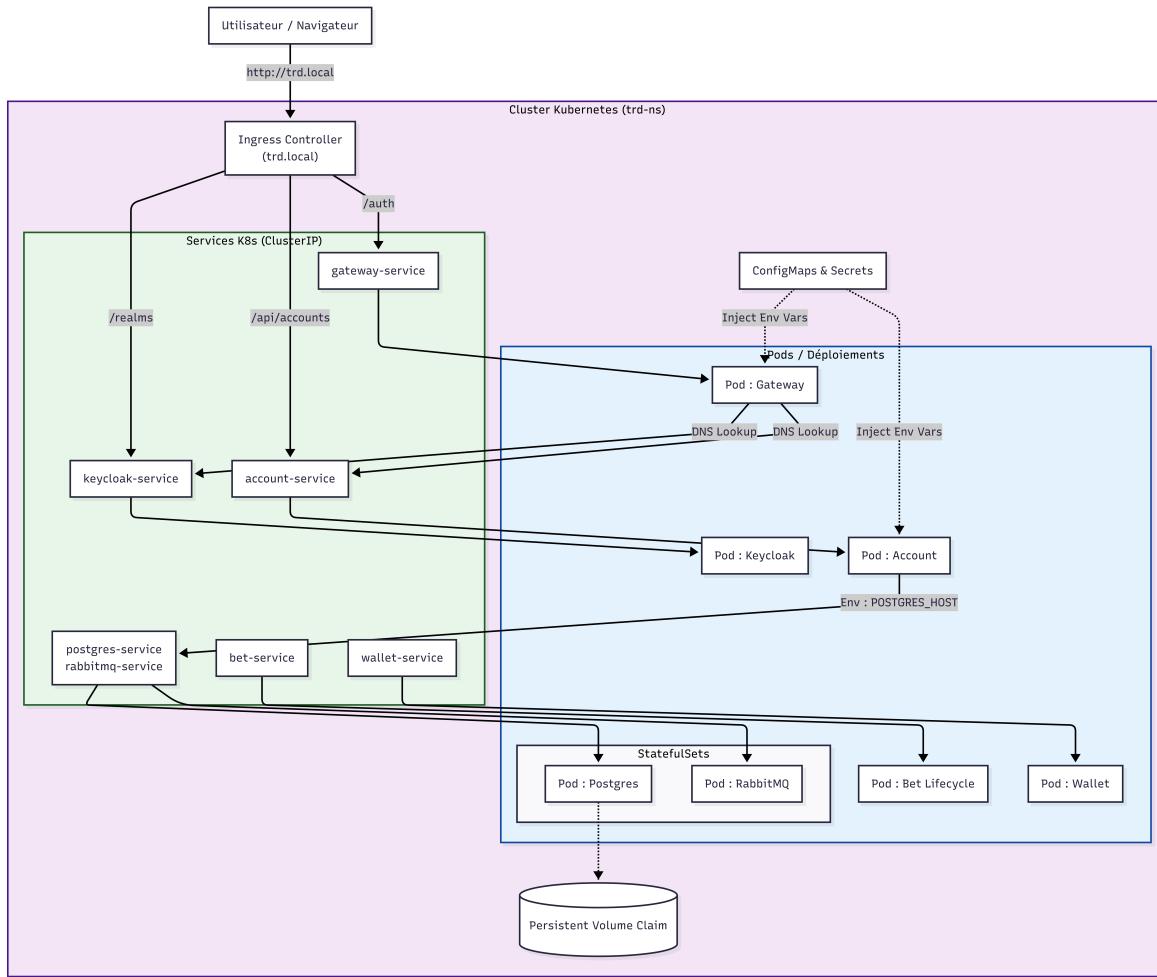


FIGURE 6.1 – Architecture Kubernetes : Services, Pods et Accès via Port Forwarding

Chapitre 7

Répartition des Rôles

7.1 Services Développés - Partie 1

7.1.1 Services de Scoring et Notification

1. **Score Service** : Worker de synchronisation des scores
2. **Bet Result Service** : Calcul des résultats et notifications
3. **Mock Score Service (Backend)** : API simulant Football-Data
4. **Mock Score Service (Frontend)** : Interface React de gestion
5. **Intégration AWS SES** : Service d'envoi d'emails

7.1.2 Responsabilités Techniques

- Architecture événementielle avec RabbitMQ
- Configuration Docker Compose
- Intégration des services cloud (AWS SES)
- Documentation technique (README, guide de test)
- Résolution des problématiques liées aux APIs externes

7.1.3 Contributions Additionnelles

- Rédaction du guide de test manuel complet
- Configuration de l'infrastructure RabbitMQ
- Mise en place des healthchecks Docker
- Documentation des workflows métier

7.2 Services Développés - Partie 2

7.2.1 Services de Gestion des Matchs et Cotes

1. **Match Odds Service** : API REST pour la gestion des matchs, équipes et cotes

7.2.2 Responsabilités Techniques

- Conception du modèle de données relationnel (Match, Team, Odd)
- Implémentation des endpoints RESTful avec ASP.NET Core
- Configuration Entity Framework Core avec PostgreSQL

- Mise en place des DTOs avec Record types C# 12
- Configuration CORS pour l'intégration frontend
- Documentation API avec Swagger/OpenAPI
- Gestion des migrations automatiques avec retry logic

7.2.3 Contributions Additionnelles

- Mise en place du pattern PATCH pour les mises à jour partielles
- Configuration des relations One-to-One et One-to-Many
- Tests manuels des endpoints via Swagger UI
- Documentation des modèles de données

Chapitre 8

Tests et Validation

8.1 Stratégie de Test

8.1.1 Tests Manuels

Un guide de test complet a été développé (TESTING-MANUAL-FLOW.md) détaillant :

1. Démarrage de l'infrastructure Docker
2. Création de matchs via le Mock Service
3. Placement de paris via RabbitMQ
4. Vérification du calcul des résultats
5. Validation de la réception des emails

8.1.2 Scénarios de Test Validés

TABLE 8.1 – Scénarios de test validés

Scénario	Description	Statut
Pari gagnant simple	Parier sur l'équipe gagnante	
Pari perdant simple	Parier sur l'équipe perdante	
Pari combiné gagnant	Parier sur plusieurs matchs (tous gagnés)	
Pari combiné perdant	Parier sur plusieurs matchs (au moins 1 perdu)	
Match nul	Parier sur un match nul	
Notification email	Réception de l'email après calcul	
Résistance aux pannes	Redémarrage d'un service pendant le traitement	

8.1.3 Tests Unitaires

Note sur les tests unitaires (Services de Scoring) :

Les tests unitaires n'ont pas été implémentés dans cette version du projet pour les services de scoring et notification. L'accent a été mis sur l'intégration fonctionnelle et la validation manuelle via le guide de test.

Tests pour MatchOddsService :

Les tests pour MatchOddsService ont été effectués via :

1. **Swagger UI** : Tests interactifs des endpoints à <http://localhost:8080>
2. **Fichier .http** : Collection de requêtes HTTP pour Visual Studio

3. Validation manuelle : Vérification des relations et contraintes en base de données

Scénarios testés pour MatchOddsService :

TABLE 8.2 – Scénarios de test MatchOddsService

Scénario	Description	Statut
Création d'équipe	POST /api/teams avec nom et drapeau	
Création de match	POST /api/matches avec deux équipes	
Ajout de cotes	POST /api/odds pour un match	
Mise à jour partielle	PATCH /api/matches/{id} changement statut	
Relations correctes	Vérification Match → Team, Match → Odd	
Contraintes	Impossible d'ajouter deux cotes pour un match	

Chapitre 9

Conclusion et Perspectives

9.1 Bilan du Projet

9.1.1 Objectifs Atteints

- Architecture microservices fonctionnelle et résiliente
- Communication événementielle avec RabbitMQ
- Intégration d'un service cloud (AWS SES)
- Service Mock complet palliant aux limitations des APIs externes
- Notifications email automatiques et personnalisées
- Déploiement containerisé avec Docker Compose
- Documentation technique exhaustive

9.1.2 Compétences Acquises

Compétences techniques acquises (Partie 1) :

- Maîtrise de l'architecture événementielle
- Intégration de services cloud (AWS SES)
- Développement de services workers .NET
- Utilisation avancée de RabbitMQ
- Création d'APIs RESTful compatibles avec des standards externes
- Développement frontend moderne avec React + TypeScript

Compétences techniques acquises (Partie 2) :

- Conception d'APIs RESTful avec ASP.NET Core 8
- Maîtrise d'Entity Framework Core et migrations
- Modélisation de données relationnelles complexes
- Utilisation des Record types et pattern matching C# 12
- Configuration CORS et middleware ASP.NET
- Documentation API avec Swagger/OpenAPI
- Gestion des opérations CRUD avec Entity Framework
- Pattern Repository et Dependency Injection
- Gestion des erreurs et retry logic

9.2 Améliorations Futures

9.2.1 Court Terme

1. Ajout de tests unitaires et d'intégration
2. Implémentation du Betting Service (placement de paris)
3. Implémentation d'un système d'authentification (JWT)
4. Interface utilisateur pour placer des paris et consulter les cotes
5. Dashboard de monitoring (Prometheus + Grafana)
6. Intégration de MatchOddsService avec le Betting Service via événements

9.2.2 Long Terme

1. Migration vers Kubernetes en production
2. Mise en place d'un API Gateway (ex : Kong, YARP)
3. Implémentation du pattern CQRS pour les requêtes complexes
4. Cache distribué avec Redis
5. Event Sourcing pour l'audit des paris
6. CI/CD avec GitHub Actions

9.3 Retour d'Expérience

9.3.1 Ce Qui a Bien Fonctionné

- **Mock Service** : Décision clé qui a débloqué le développement
- **RabbitMQ** : Excellente fiabilité, aucune perte de message
- **AWS SES** : Intégration fluide, emails bien délivrés
- **Docker Compose** : Environnement de dev reproductible

9.3.2 Défis Rencontrés

- **Encodage UTF-8** : Problème résolu mais chronophage
- **Synchronisation événements** : Nécessité de logs détaillés pour le debug
- **Configuration AWS** : Courbe d'apprentissage pour SES et IAM

9.3.3 Leçons Apprises

1. **Toujours avoir un plan B** : Le Mock Service a sauvé le projet
2. **Logs détaillés dès le début** : Indispensable pour le debug distribué
3. **Tests manuels documentés** : Gain de temps énorme pour les démos
4. **Communication asynchrone** : Plus résilient que HTTP synchrone

Annexe A

Structure du Repository

```
1 TRDServices/
2 |-- BetResultService/
3 |   |-- Services/
4 |   |   |-- BetProcessor.cs
5 |   |   |-- SesEmailService.cs
6 |   |   |-- IEmailService.cs
7 |   |-- Worker.cs
8 |   |-- Program.cs
9 |   |-- Dockerfile
10 |  |-- appsettings.json
11 |
12 |-- ScoreService/
13 |   |-- Services/
14 |   |   |-- FootballDataService.cs
15 |   |   |-- RabbitMqProducer.cs
16 |   |-- Worker.cs
17 |   |-- Program.cs
18 |   |-- Dockerfile
19 |   |-- appsettings.json
20 |
21 |-- MatchOddsService/
22 |   |-- Controllers/
23 |   |   |-- MatchesController.cs
24 |   |   |-- OddsController.cs
25 |   |-- Dockerfile
26 |   |-- appsettings.json
27 |
28 |-- MockScoreService.API/
29 |   |-- Controllers/
30 |   |   |-- MatchesController.cs
31 |   |   |-- TeamsController.cs
32 |   |-- DTOs/
33 |   |   |-- ApiResponses.cs
34 |   |-- Models/
35 |   |-- Dockerfile
36 |   |-- appsettings.json
37 |
38 |-- MockScoreService.React/
39 |   |-- src/
40 |   |   |-- pages/
41 |   |   |   |-- Simulation.tsx
42 |   |   |   |-- Matches.tsx
43 |   |   |-- services/
44 |   |   |   |-- api.ts
45 |   |   |-- components/
46 |   |-- Dockerfile
47 |   |-- package.json
48 |
49 |-- k8s/
50 |   |-- namespace.yaml
51 |   |-- configmaps.yaml
52 |   |-- secrets.yaml
53 |   |-- deployments/
54 |   |-- services/
55 |   |-- ingress.yaml
56 |
57 |-- docs/
58 |   |-- rapport-technique-trd-v0.tex (ce document)
59 |
60 |-- docker-compose.yml
61 |-- rabbitmq-definitions.json
62 |-- init-db.sql
63 |-- .env.example
64 |-- README.md
65 |-- TESTING-MANUAL-FLOW.md
```

Listing A.1 – Arborescence complète du projet

Annexe B

Configuration RabbitMQ

```
1  {
2      "exchanges": [
3          {
4              "name": "sportsbook.topic",
5              "type": "topic",
6              "durable": true
7          }
8      ],
9      "queues": [
10         {
11             "name": "q.bet-result.new-bets",
12             "durable": true
13         },
14         {
15             "name": "q.bet-result.match-scores",
16             "durable": true
17         }
18     ],
19     "bindings": [
20         {
21             "source": "sportsbook.topic",
22             "destination": "q.bet-result.new-bets",
23             "routing_key": "bet.placed"
24         },
25         {
26             "source": "sportsbook.topic",
27             "destination": "q.bet-result.match-scores",
28             "routing_key": "match.finished"
29         }
30     ]
31 }
```

Listing B.1 – rabbitmq-definitions.json

Annexe C

Exemples de Requêtes API

C.0.1 Mock Score Service - Crée un Match

```
1 curl -X POST http://localhost:5000/v4/matches \
2   -H "Content-Type: application/json" \
3   -d '{
4     "homeTeamId": 1,
5     "awayTeamId": 2,
6     "utcDate": "2026-06-20T20:00:00Z",
7     "matchday": 1,
8     "stage": "GROUP_STAGE"
9   }'
```

Listing C.1 – POST /v4/matches

C.0.2 Mock Score Service - Terminer un Match

```
1 curl -X PATCH http://localhost:5000/v4/matches/1/status \
2   -H "Content-Type: application/json" \
3   -d '{"status": "FINISHED"}'
```

Listing C.2 – PATCH /v4/matches/{id}/status

C.0.3 Match Odds Service - Crée une Équipe

```
1 curl -X POST http://localhost:8080/api/teams \
2   -H "Content-Type: application/json" \
3   -d '{
4     "name": "France",
5     "flagUrl": "https://flagcdn.com/fr.svg"
6   }'
```

Listing C.3 – POST /api/teams

C.0.4 Match Odds Service - Crée un Match

```
1 curl -X POST http://localhost:8080/api/matches \
2   -H "Content-Type: application/json" \
3   -d '{
4     "homeTeamId": 1,
5     "awayTeamId": 2,
```

```
6 "matchDate": "2026-06-20T20:00:00Z"  
7 }'
```

Listing C.4 – POST /api/matches

C.0.5 Match Odds Service - Ajouter des Cotes

```
1 curl -X POST http://localhost:8080/api/odds \  
2   -H "Content-Type: application/json" \  
3   -d '{  
4     "matchId": 1,  
5     "homeWin": 1.85,  
6     "awayWin": 3.40,  
7     "draw": 3.20  
8   }'
```

Listing C.5 – POST /api/odds

C.0.6 Match Odds Service - Changer le Statut d'un Match

```
1 curl -X PATCH http://localhost:8080/api/matches/1 \  
2   -H "Content-Type: application/json" \  
3   -d '{"status": "Live"}'
```

Listing C.6 – PATCH /api/matches/{id}

Annexe D

Glossaire

TABLE D.1 – Glossaire des termes techniques

Terme	Définition
Microservice	Service autonome et indépendant gérant une fonctionnalité métier
Event-Driven	Architecture basée sur la publication et consommation d'événements
RabbitMQ	Message broker AMQP pour la communication asynchrone
AWS SES	Amazon Simple Email Service pour l'envoi d'emails transactionnels
Docker Compose	Outil d'orchestration multi-conteneurs pour le développement
Worker Service	Service .NET s'exécutant en arrière-plan (background service)
Mock	Simulation d'un service externe pour les tests
Routing Key	Clé de routage RabbitMQ pour diriger les messages vers les bonnes queues