



جامعة عبد المالك السعدي
تولهات | تونس | Université Abdelmalek Essaâdi



كلية العلوم والتكنولوجيا
+٥٢٣٦٠٤٤٨٨ | +٣٣٦٣٤٤٧٤١١ | EoTo
Faculté des Sciences et Techniques de Tanger

Université Abdelmalek Essaâdi
Faculté des Sciences et Techniques de Tanger
Département Génie Informatique

Decentralized Microservices App for Real Estate Rental

Based on Ethereum Blockchain

DeRent 5

Réalisé par :

El Gorrim Mohamed
Kchibal Ismail
Mohand Omar Moussa
Essalhi Salma
El Azzouzi Achraf

Encadré par :

M. Lotfi El Aachak

Cycle Ingénieur – Logiciels et Systèmes Intelligents
Année Universitaire 2025–2026

Table des matières

1	Introduction	2
1.1	Introduction Générale	2
1.2	Composition de l'Équipe	2
2	Architecture du Système	4
2.1	Conception Architecturale	4
2.1.1	Diagramme d'Architecture Globale	4
2.2	Modèle de Données (Data Modeling)	5
2.2.1	Diagramme de Classes (UML)	5
2.2.2	Analyse des Entités Clés	6
3	Microservices Backend	7
3.1	Choix Technologiques (Tech Stack)	7
3.2	Architecture Interne des Microservices	7
3.3	Détails des Microservices Clés	8
3.3.1	1. User Service (Gestion d'Identité)	8
3.3.2	2. Property Service (Gestion Immobilière)	8
3.3.3	3. Booking Service (Réservations)	8
3.3.4	4. Blockchain Service (Middleware Web3)	9
3.3.5	5. AI Service (Intelligence Artificielle)	9
3.4	Communication Asynchrone avec RabbitMQ	9
3.5	Sécurité Avancée et API Gateway	10
4	Développement Frontend	11
4.1	Interface Utilisateur (Screenshots)	11
4.1.1	Page d'Accueil (Landing Page)	11
4.1.2	Catalogue des Biens (Listings)	11
4.1.3	Détail d'une Propriété & Réservation	12
5	Couche Blockchain	13
5.1	Analyse du Smart Contract : BookingPaymentContract	13
5.1.1	Structures de Données (State Variables)	13
5.1.2	Logique Transactionnelle	13
5.2	Patterns de Sécurité Implémentés	14
5.3	Interaction Web3 avec MetaMask	14
5.3.1	Connexion du Wallet Ethereum	14
5.3.2	Confirmation du Paiement et Wallet Actif	15
5.3.3	Validation de la Transaction dans MetaMask	16

6 Intelligence Artificielle & Data Science	17
6.1 Modèle Phare : Prédiction Dynamique des Prix	17
6.1.1 Collecte de Données (Web Scraping)	17
6.1.2 Pipeline ETL & Feature Engineering	17
6.1.3 Modélisation et Optimisation (XGBoost)	18
6.2 Autres Modules Intelligents	18
6.2.1 Score de Risque Locataire (Tenant Risk Scoring)	18
6.2.2 Segmentation de Marché (Clustering)	18
6.2.3 Tendances du Marché (Market Trends)	18
6.3 Intégration dans l'Application (Screenshots)	18
6.3.1 Hôte : Suggestion de Prix Intelligente	19
6.3.2 Hôte : Analyse de Risque Locataire	19
6.3.3 Investisseur : Tableau de Bord du Marché	20
6.3.4 Locataire : Recommandations Personnalisées	20
7 DevOps & CI/CD	21
7.1 Orchestration Globale	21
7.2 Pipeline Standard (Backend Java)	22
8 Orchestration Kubernetes	25
8.1 Architecture du Cluster	25
8.2 Déploiement et Opérations	25
8.2.1 État du Cluster (Pods)	25
8.2.2 Exposition des Services (NodePort)	26
8.3 Observabilité et Monitoring	27
8.3.1 Prometheus (Métriques)	27
8.3.2 Grafana (Visualisation)	27
9 Infrastructure Cloud (AWS)	29
9.1 Architecture Modulaire Terraform	29
9.2 Schéma d'Architecture Cible	29
10 Conclusion	31
10.1 Réalisations Clés	31
10.2 Perspectives	31

Chapitre 1

Introduction

1.1 Introduction Générale

L'industrie immobilière traditionnelle est souvent entravée par des processus lents, des coûts d'intermédiation élevés et un manque de transparence. Ce projet, intitulé **Decentralized Real Estate Rental dApp**, propose une solution de rupture en tirant parti de la technologie **Blockchain** pour réinventer la location immobilière.

Il s'agit d'une plateforme **Peer-to-Peer (P2P)** qui permet aux propriétaires (*Hosts*) et aux locataires (*Tenants*) d'interagir directement via des **Smart Contracts** sur le réseau **Ethereum**. Cette approche "Trustless" garantit que les termes du contrat sont exécutés automatiquement, sécurisant ainsi les dépôts de garantie et les paiements sans nécessiter de tiers de confiance.

Le système est construit sur une architecture **Microservices** moderne, intégrant des technologies de pointe telles que **Spring Boot** pour le backend, **Next.js** pour le frontend, et une infrastructure **Cloud Native** orchestrée par **Kubernetes** sur **AWS**. De plus, l'intégration de modules d'**Artificial Intelligence** optimise l'expérience utilisateur grâce à la tarification dynamique et l'analyse de risque.

1.2 Composition de l'Équipe

La réussite de ce projet complexe repose sur une équipe multidisciplinaire de 5 ingénieurs, chacun expert dans son domaine technologique (*Backend, Frontend, Blockchain, DevOps / AI & Data, Cloud*).

Photo	Membre & Rôle	Responsabilités Clés
	El Azzouzi Achraf <i>Backend Engineer</i>	Core Development : Conception et implémentation des microservices avec Spring Boot 3 . API Design : Création d'APIs RESTful sécurisées et communication inter-services via RabbitMQ . Data Layer : Modélisation de la base de données PostgreSQL et intégration Hibernate.
	Essalhi Salma <i>Frontend Engineer</i>	UX/UI Design : Développement d'une interface utilisateur réactive avec Next.js et Tailwind CSS . Web3 Integration : Connexion des wallets (MetaMask) et interaction avec les smart contracts via Ethers.js . Mapping : Intégration de cartes interactives pour la localisation des biens.
	Mohand Omar Moussa <i>Blockchain Engineer</i>	Smart Contracts : Développement en Solidity des contrats de location et d'escrow. Testing & Security : Tests unitaires avec Hardhat et audit de sécurité avec Slither . Deployment : Gestion des migrations sur les réseaux de test (Testnets).
	El Gorrim Mohamed <i>DevOps / AI Engineer</i>	CI/CD : Automatisation des pipelines de déploiement avec Jenkins et Docker . Orchestration : Configuration et gestion des clusters Kubernetes (K8s) . AI Models : Développement de modèles de <i>Machine Learning</i> pour le <i>Tenant Risk Scoring</i> et le <i>Dynamic Pricing Suggestion</i> .
	Kchibal Ismail <i>Cloud Engineer</i>	IaC : Provisionnement de l'infrastructure AWS via Terraform . Networking : Configuration sécurisée du VPC, des subnets et des load balancers. Storage : Gestion du stockage S3 et des registres ECR .

Chapitre 2

Architecture du Système

2.1 Conception Architecturale

Pour répondre aux exigences de scalabilité, de résilience et de sécurité de la plate-forme, nous avons adopté une **Architecture Microservices** stricte. Contrairement à une approche monolithique, cette stratégie permet de découpler les fonctionnalités métier en services autonomes, chacun déployable et maintenable indépendamment.

2.1.1 Diagramme d'Architecture Globale

L'architecture repose sur un point d'entrée unique qui orchestre les échanges entre les clients (Web/Mobile) et les services backend.

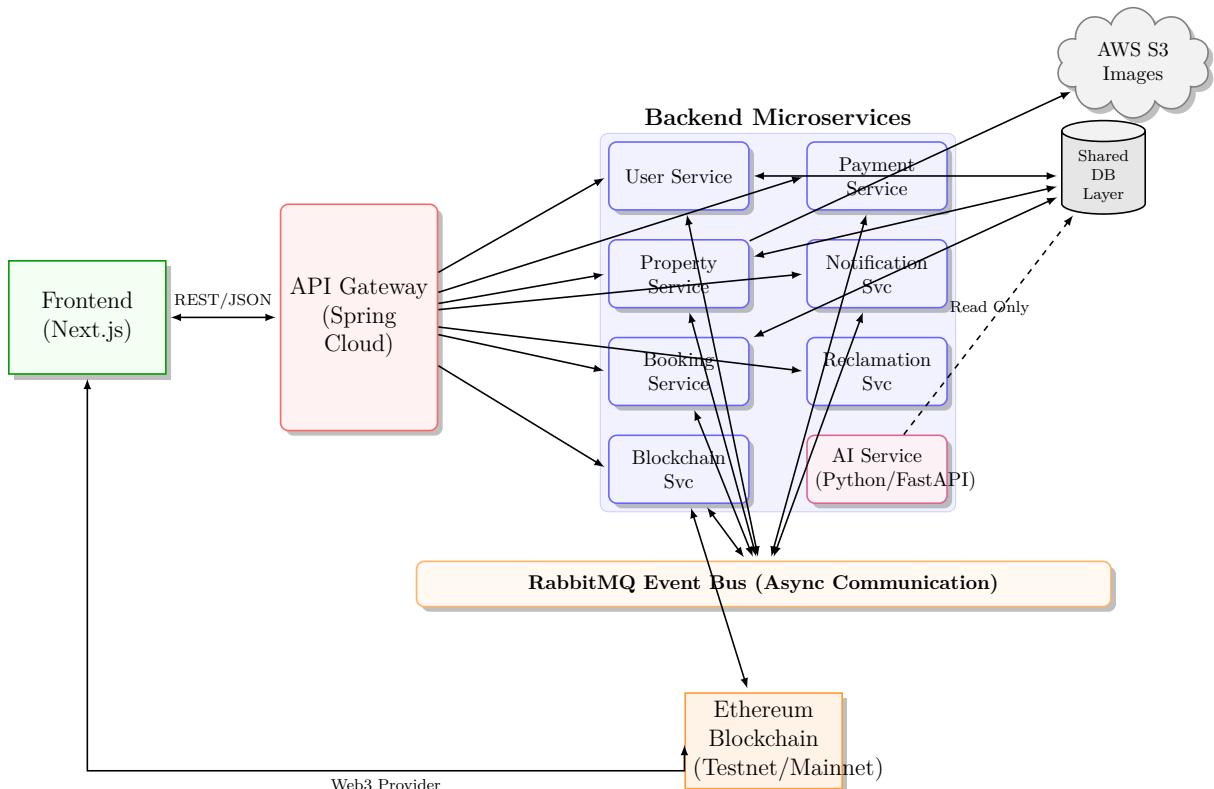


FIGURE 2.1 – Architecture Microservices Complète avec Flux de Communication

L'architecture se décompose en quatre couches principales :

1. **Client Layer (Frontend)** : Développée en Next.js, cette couche gère l'interface utilisateur. Elle est "stateless" et communique avec le backend exclusivement via des appels API REST sécurisés. Elle intègre également la librairie Ethers.js pour interagir directement avec la Blockchain Ethereum (Web3).
2. **Gateway Layer (API Gateway)** : Basée sur Spring Cloud Gateway, elle agit comme un "Reverse Proxy" intelligent.
 - **Routing Dynamique** : Redirection des requêtes (ex : `/api/v1/properties/ → property-service`).
 - **Sécurité Centralisée** : Validation des JWT avant de transmettre la requête.
 - **Cross-Cutting Concerns** : CORS, Rate Limiting et Logging centralisé.
3. **Business Layer (Microservices)** : Services isolés implémentant la logique métier. Ils communiquent de manière :
 - **Synchrone** : Appels HTTP via *OpenFeign* pour les opérations bloquantes (ex : vérifier l'existence d'un utilisateur lors d'une réservation).
 - **Asynchrone** : Échange d'événements via **RabbitMQ** pour les tâches de fond (ex : envoi de notification après paiement).
4. **Persistence Layer** : Approche Polyglotte pour le stockage de données :
 - **PostgreSQL** : Données relationnelles structurées (utilisateurs, réservations). Chaque microservice possède son propre schéma logique ("Database-per-Service").
 - **AWS S3** : Stockage d'objets (BLOB) pour les médias (photos des biens).
 - **Blockchain Ethereum** : "Ledger" immuable pour les contrats de location et l'historique des transactions.

2.2 Modèle de Données (Data Modeling)

Le schéma de base de données a été conçu pour garantir l'intégrité des transactions immobilières tout en restant flexible.

2.2.1 Diagramme de Classes (UML)

Le diagramme suivant illustre les relations entre les entités persistantes du système.

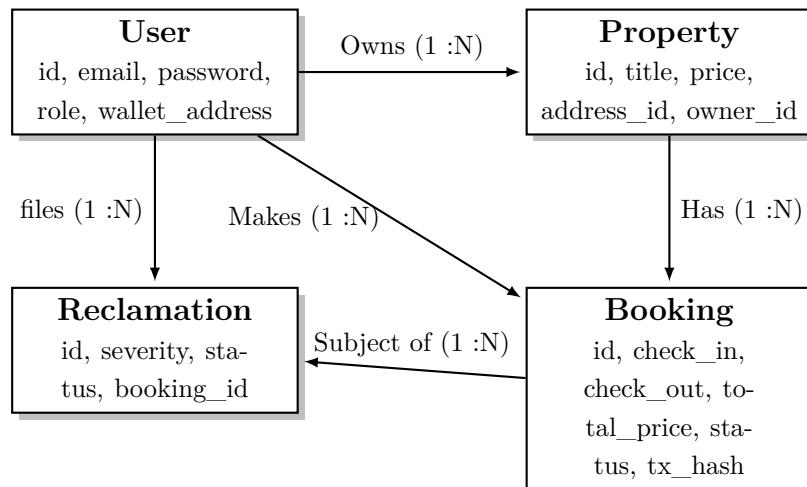


FIGURE 2.2 – Diagramme de Classes Simplifié (Entités Clés)

2.2.2 Analyse des Entités Clés

1. Gestion des Utilisateurs (`users`)

L'entité `User` est centrale. Elle utilise une stratégie d'héritage (ou de rôles) pour distinguer `Tenants` et `Hosts`.

- **Attributs** : `id`, `email`, `wallet_address` (lien Web3), `is_verified`.
- **Sécurité** : Les mots de passe sont hashés (BCrypt) et jamais stockés en clair.

2. Gestion Immobilière (`properties`)

Cette entité stocke les caractéristiques des biens.

- **Relation** : One-to-Many avec `property_images` (plusieurs photos par bien).
- **Localisation** : Liée à une table `addresses` contenant les coordonnées GPS (`latitude`, `longitude`) pour la carte interactive.
- **Disponibilité** : Relation avec `availabilities` pour gérer le calendrier.

3. Réservations et Transactions (`bookings`)

C'est l'entité pivot du système.

- **Cycle de Vie** : Géré par un champ `status` (enum : PENDING, ACCEPTED, PAID, COMPLETED, CANCELLED).
- **Blockchain Link** : Le champ `on_chain_tx_hash` stocke la référence du Smart Contract déployé pour cette réservation.
- **Contraintes** : Clés étrangères strictes vers `users` (locataire) et `properties`.

4. Gestion des Litiges (`reclamations`)

Permet de traiter les conflits post-séjour.

- **Preuves** : Relation One-to-Many avec `reclamation_attachments` pour stocker les photos/preuves.
- **Workflow** : Statuts OPEN, IN_REVIEW, RESOLVED.

Chapitre 3

Microservices Backend

Le backend constitue le noyau transactionnel de la plateforme *Derent*. Il a été conçu pour être modulaire, résilient et hautement scalable, en suivant strictement les principes de l'architecture Microservices et de la méthodologie *12-Factor App*.

Chaque service est une unité autonome, responsable d'un domaine métier spécifique (Bounded Context), possédant sa propre base de données pour garantir un couplage faible et une forte cohésion.

3.1 Choix Technologiques (Tech Stack)

Pour assurer robustesse et maintenabilité, nous avons sélectionné la stack technique suivante :

- **Langage & Framework** : **Java 17** (LTS) avec **Spring Boot 3.3**. Ce choix offre un excellent écosystème, une sécurité mature et une intégration native avec le Cloud.
- **Build Tool** : **Maven** pour la gestion des dépendances et du cycle de vie des projets (Multi-module).
- **Base de Données** : **PostgreSQL 15** pour la persistance relationnelle, avec **Hibernate/JPA** comme ORM.
- **Migration** : Les scripts SQL sont gérés par le mode `ddl-auto: update` en développement, mais prévus pour être gérés par **Liquibase** en production.
- **Communication** :
 - Synchrone : **Spring Cloud OpenFeign** pour les appels REST déclaratifs entre services.
 - Asynchrone : **RabbitMQ** pour l'échange de messages et le découplage événementiel.
- **Utilitaires** : **Lombok** (réduction du boilerplate), **MapStruct** (mapping DTO/Entity performant), **Spring Actuator** (monitoring).

3.2 Architecture Interne des Microservices

Chaque microservice suit une **Architecture en Couches** (inspirée de l'Architecture Hexagonale) pour séparer clairement les responsabilités :

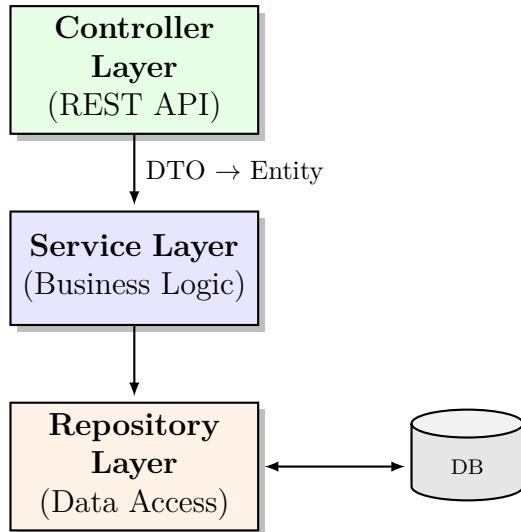


FIGURE 3.1 – Architecture en couches adoptée pour chaque Microservice

- **Controller** : Expose les endpoints REST, gère la validation des entrées ('@Valid') et renvoie les réponses HTTP standardisées ('ResponseEntity').
- **Service** : Contient la logique métier pure (ex : calculs, vérifications, appels externes). C'est la seule couche transactionnelle ('@Transactional').
- **Repository** : Interface étendant 'JpaRepository' pour l'accès aux données.
- **DTO (Data Transfer Object)** : Objets simples pour transférer les données entre le client et le serveur, évitant d'exposer directement les entités JPA.

3.3 Détails des Microservices Clés

3.3.1 1. User Service (Gestion d'Identité)

Ce service est le gardien des comptes utilisateurs.

- **Responsabilités** : Inscription, Authentification, Gestion de profil via **JWT**.
- **Logique Spécifique** : Il intègre une validation Web3 pour associer un portefeuille Ethereum ('wallet_address') à un compte classique. Cela permet une authentification hybride (Web2 + Web3).
- **Sécurité** : Les mots de passe sont hashés avec **BCrypt** avant stockage.

3.3.2 2. Property Service (Gestion Immobilière)

Le cœur du catalogue de la plateforme.

- **Fonctionnalités** : CRUD des propriétés, recherche par filtres (Ville, Prix, Date).
- **Stockage S3** : Lorsqu'un utilisateur upload une image, le service génère une URL présignée ou stocke le fichier directement sur **AWS S3** et sauvegarde l'URL en base.
- **Intégration** : Publie un événement 'property.created' écouté par le service IA pour l'analyse de prix.

3.3.3 3. Booking Service (Réservations)

L'orchestrateur des transactions. Il gère un cycle de vie complexe :

1. **Pending** : Création par le locataire. Vérification synchrone de la disponibilité (appel ‘PropertyService’).
2. **Accepted/Rejected** : Validation par l’hôte.
3. **Payment Pending** : Attente du paiement initial.
4. **Confirmed** : Paiement reçu + Smart Contract déployé (appel asynchrone ‘BlockchainService’).

3.3.4 4. Blockchain Service (Middleware Web3)

Ce service agit comme une passerelle sécurisée vers le réseau Ethereum.

- **Technologie** : Utilise la librairie **Web3j** pour interagir avec les noeuds Ethereum (via Infura/Alchemy ou Geth local).
- **Rôle** : Déploie les Smart Contracts de location (‘RentalAgreement’) et surveille les événements on-chain (‘DepositPaid’, ‘FundsReleased’).
- **Sécurité** : Il gère les clés privées du système (Server Wallet) stockées de manière sécurisée (Vault ou Variables d’Env chiffrées).

3.3.5 5. AI Service (Intelligence Artificielle)

Bien que développé en Python (FastAPI) pour l'accès aux librairies Data Science (Scikit-learn, Pandas), il est pleinement intégré à l'architecture.

- **Communication** : Il expose des API REST consommées par le Gateway.
- **Fonctions** :
 - *Price Prediction* : Suggère un prix à la création d'une annonce.
 - *Tenant Risk Score* : Analyse l'historique d'un locataire.

3.4 Communication Asynchrone avec RabbitMQ

Pour garantir la résilience, les processus non critiques sont découplés via un Bus d’Événements.

Cas d’usage : La Notification de Réservation

1. Le ‘Booking Service’ confirme une réservation.
2. Il publie un message sur l'échange ‘booking.exchange’ avec la routing key ‘booking.confirmed’.
3. Le ‘Notification Service’, abonné à la queue correspondante, consomme le message.
4. Il construit un email HTML et l'envoie via SMTP (Gmail/SendGrid), sans ralentir le processus de réservation initial.

Listing 3.1 – Exemple de Producteur RabbitMQ (Booking Service)

```

1 @Service
2 public class BookingEventProducer {
3     private final RabbitTemplate rabbitTemplate;
4
5     public void sendBookingConfirmedEvent(Booking booking) {
6         BookingEvent event = new BookingEvent(booking.getId(),
7             booking.getUserEmail(), "CONFIRMED");
8     }
9 }
```

```

7   rabbitTemplate.convertAndSend("booking.exchange", " 
8     booking.confirmed", event);
9   log.info("Event sent for booking:{}", booking.getId());
10 }

```

3.5 Sécurité Avancée et API Gateway

L'**API Gateway** (Spring Cloud Gateway) est le point d'entrée unique. Elle implémente une sécurité en profondeur :

- **Authentication Filter** : Intercepte chaque requête pour valider le Token JWT (signature et expiration) avant de la router.
- **CORS Configuration** : Restreint l'accès aux seules origines autorisées (Frontend Next.js).
- **Rate Limiting** : Protège les microservices contre les attaques DDoS en limitant le nombre de requêtes par IP.

Chapitre 4

Développement Frontend

L'interface utilisateur de *Derent* a été conçue pour offrir une expérience fluide, masquant la complexité de la Blockchain derrière une ergonomie moderne.

Conformément aux directives du projet, cette section présente les vues principales de l'application.

4.1 Interface Utilisateur (Screenshots)

4.1.1 Page d'Accueil (Landing Page)

Point d'entrée de l'application, invitant l'utilisateur à explorer les biens ou à connecter son Wallet.

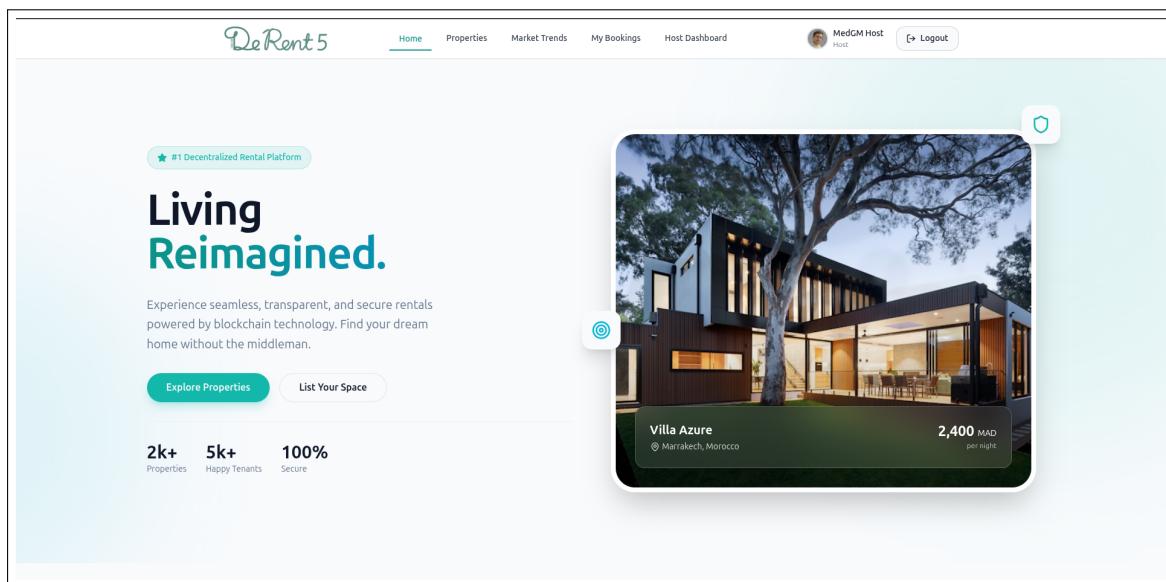


FIGURE 4.1 – Page d'Accueil de la plateforme

4.1.2 Catalogue des Biens (Listings)

Affichage des propriétés disponibles avec filtres et carte interactive.

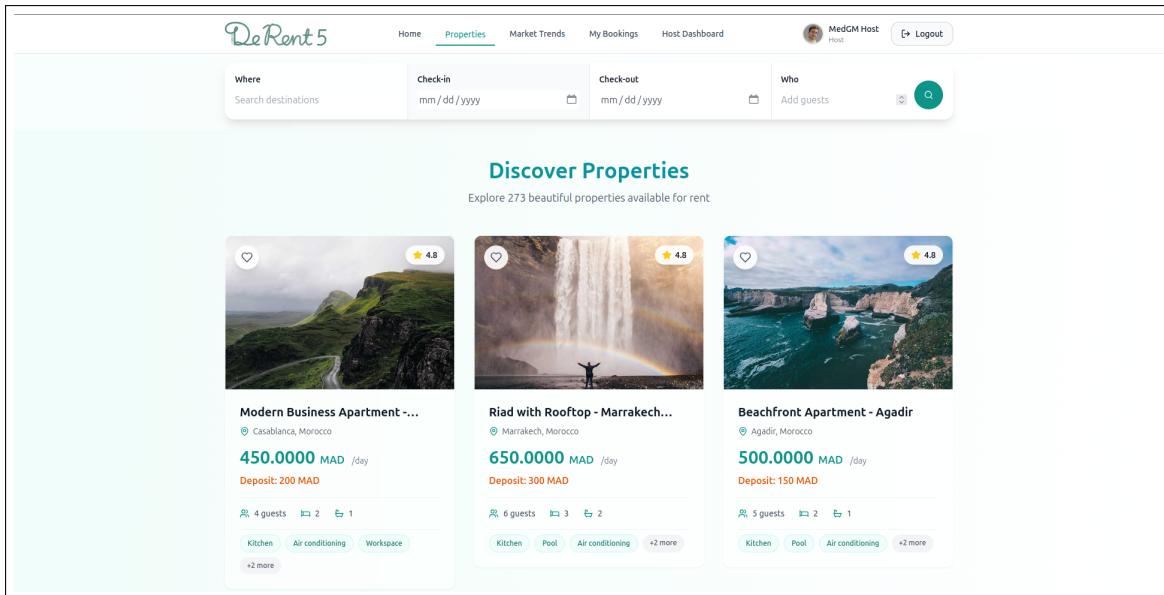


FIGURE 4.2 – Liste des propriétés avec recherche géolocalisée

4.1.3 Détail d'une Propriété & Réservation

Vue détaillée permettant au locataire de voir les photos, les équipements et d'initier une transaction.

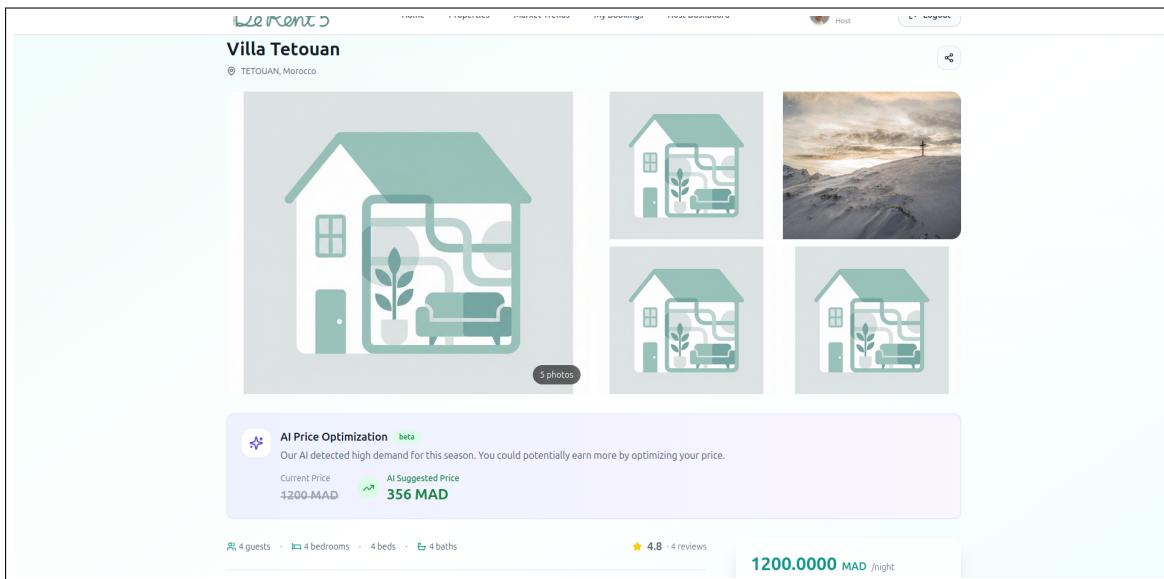


FIGURE 4.3 – Page de détail d'un bien avec bouton de réservation Web3

Chapitre 5

Couche Blockchain

La confiance et la sécurité des transactions financières sont assurées par un **Smart Contract** personnalisé déployé sur Ethereum. Nous avons analysé en détail le code source du contrat `BookingPaymentContract.sol` situé dans le service Blockchain.

5.1 Analyse du Smart Contract : BookingPayment-Contract

Ce contrat agit comme un tiers de confiance automatisé (Escrow). Il sécurise les fonds du locataire jusqu'à la fin de la location.

5.1.1 Structures de Données (State Variables)

Le contrat repose sur deux structures principales définissant l'état du système :

Listing 5.1 – Struct Booking (Solidity)

```
1 struct Booking {
2     address guest;          // Adresse ETH du locataire
3     address host;           // Adresse ETH de l'hôte
4     uint256 rentAmount;     // Montant du loyer (en Wei)
5     uint256 depositAmount;  // Caution (en Wei)
6     bool completed;         // Statut de la transaction
7     bool hasActiveReclamation; // Flag de litige
8     uint256 completedAt;    // Timestamp de fin
9 }
```

Le contrat maintient également des constantes critiques pour le modèle économique :

- `PLATFORM_FEE_PERCENT = 10` : La plateforme préleve 10% sur chaque transaction réussie.
- `PLATFORM_WALLET` : Adresse hardcodée recevant les commissions.

5.1.2 Logique Transactionnelle

1. Crédation et Paiement (`createBookingPayment`)

Cette fonction est `payable`. Elle bloque les Ethers envoyés par le locataire dans le contrat.

- **Validation** : Vérifie que `msg.value` correspond exactement à `rentAmount + depositAmount`.
- **Intégrité** : Empêche l'hôte d'être son propre locataire ('`host != tenant`').
- **Événement** : Émet `BookingPaymentCreated` pour notifier le backend off-chain.

2. Finalisation (`completeBooking`)

Appelée à la fin du séjour si aucun litige n'est en cours.

- **Distribution** :
 - Calcule la commission : $Fee = Rent \times 10\%$.
 - Transfère le Loyer Net ($Rent - Fee$) à l'Hébergeur.
 - Transfère la Commission à la Plateforme.
- La caution (`depositAmount`) reste virtuellement disponible pour être remboursée (géré par une logique de retrait séparée ou implicite selon le workflow de fin).

3. Gestion des Litiges (`processReclamationRefund`)

Fonction administrative ('`onlyAdmin`') permettant de résoudre les conflits. Elle offre une granularité fine :

- Peut rembourser le locataire ('`refundAmount`') ou l'hôte.
- Peut prélever une pénalité ('`penaltyAmount`') vers la plateforme.
- Met à jour l'état `hasActiveReclamation = false` pour débloquer le reste des fonds.

5.2 Patterns de Sécurité Implémentés

L'analyse du code révèle l'utilisation de bonnes pratiques de sécurité Solidity :

1. **Protection Anti-Reentrancy** : Utilisation d'un modificateur `nonReentrant` avec un mutex booléen (`locked`) pour empêcher les attaques récursives lors des transferts d'Ether ('`.callvalue : ...`').
2. **Pattern Checks-Effects-Interactions** : Le contrat met à jour l'état (ex : `booking.completed = true`) *avant* d'effectuer les transferts de fonds externes, minimisant les risques de manipulation.
3. **Contrôle d'Accès (RBAC)** : Le modificateur `onlyAdmin` restreint les fonctions sensibles (changement d'admin, résolution de litiges, retrait d'urgence) à l'adresse de déploiement.

5.3 Interaction Web3 avec MetaMask

L'intégration Web3 permet à l'utilisateur d'interagir directement avec le Smart Contract via son portefeuille Ethereum. La bibliothèque `Ethers.js` est utilisée côté frontend pour établir la communication entre l'application décentralisée (dApp) et MetaMask.

5.3.1 Connexion du Wallet Ethereum

La Figure 5.1 illustre l'étape initiale où l'utilisateur connecte son portefeuille MetaMask à l'application. Cette étape est indispensable pour récupérer l'adresse Ethereum de l'utilisateur et autoriser toute transaction on-chain.

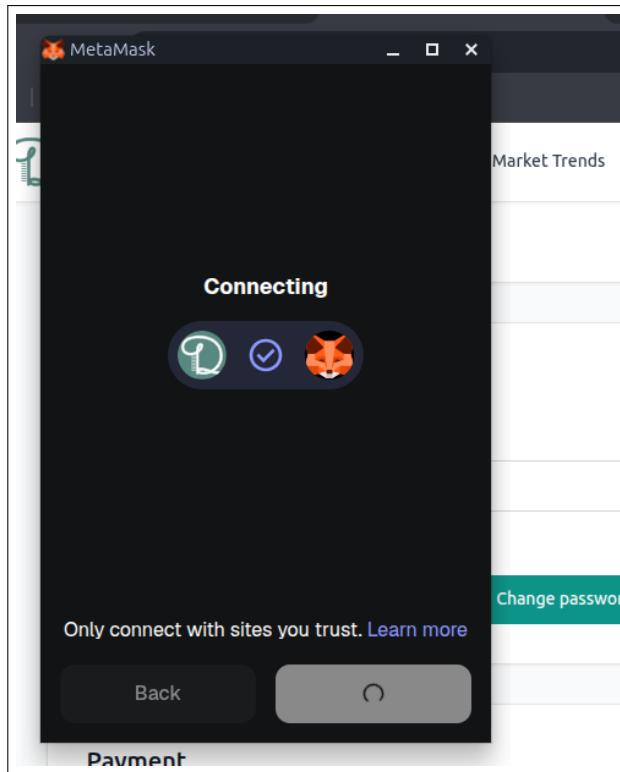


FIGURE 5.1 – Connexion de l’application au portefeuille MetaMask

5.3.2 Confirmation du Paiement et Wallet Actif

Après la sélection du bien et la validation de la réservation, l’utilisateur accède à la page de confirmation du paiement. La Figure 5.2 montre l’affichage du montant total à payer ainsi que l’adresse du wallet actuellement connecté via MetaMask, garantissant la transparence de l’opération.

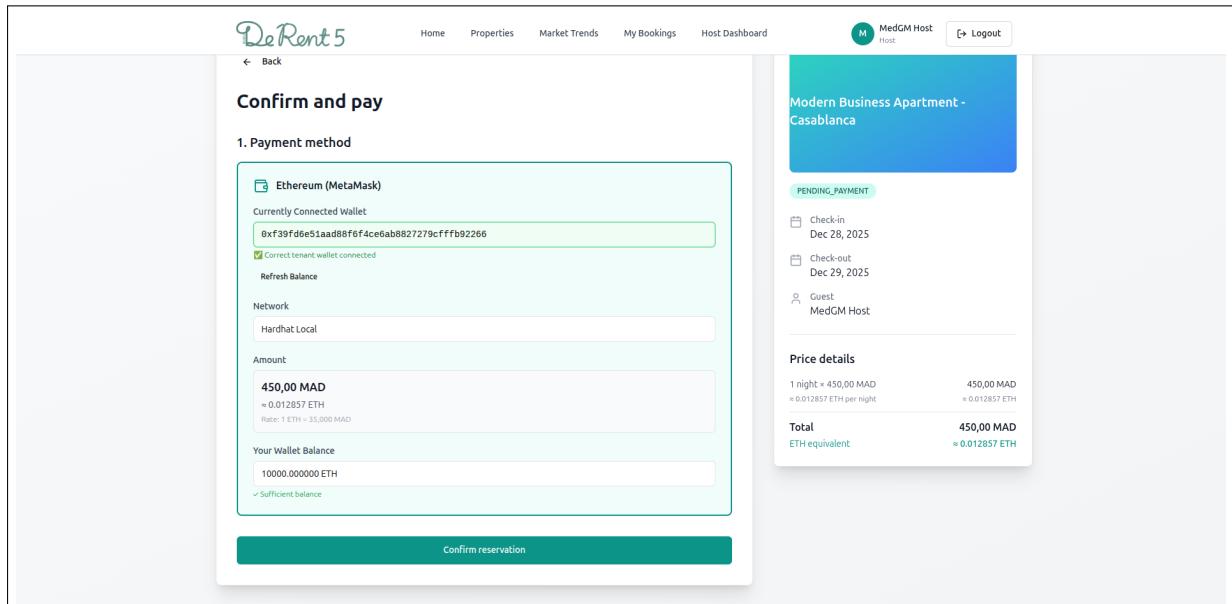


FIGURE 5.2 – Page de confirmation et paiement avec wallet MetaMask connecté

5.3.3 Validation de la Transaction dans MetaMask

Lors du paiement, MetaMask génère une demande de transaction Ethereum. Comme illustré dans la Figure 5.3, l'utilisateur doit confirmer le transfert d'Ether incluant le montant.

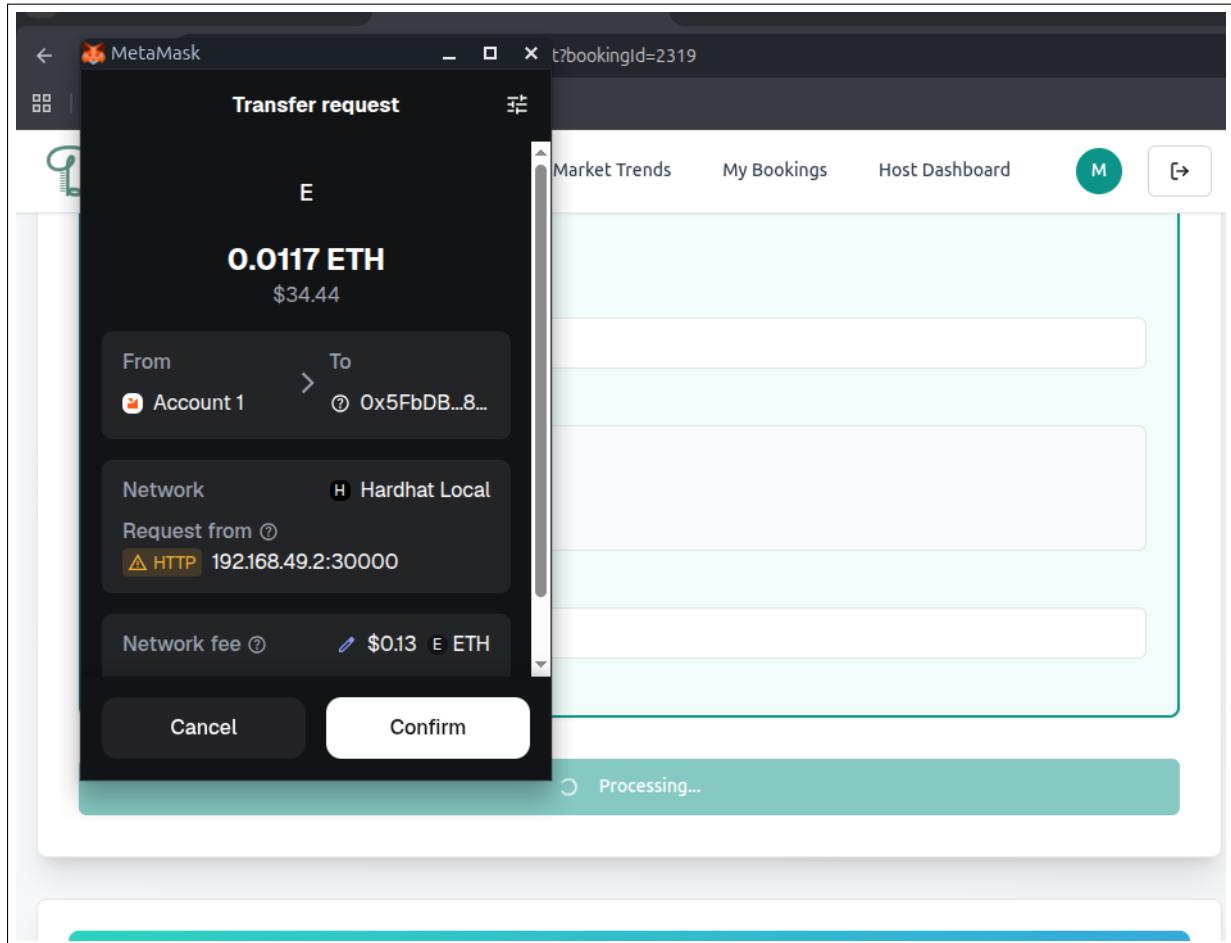


FIGURE 5.3 – Demande de transfert Ethereum dans MetaMask

Chapitre 6

Intelligence Artificielle & Data Science

Le cœur technologique de *Derent* réside dans son moteur d'intelligence artificielle, conçu pour apporter transparence et sécurité au marché locatif. Ce chapitre détaille le pipeline technique complet, de la collecte de données massive au déploiement de modèles de Machine Learning avancés.

Contrairement aux solutions standards, nous avons développé nos propres datasets et algorithmes "maison", validés par une méthodologie rigoureuse.

6.1 Modèle Phare : Prédiction Dynamique des Prix

Ce module ("Price Suggestion AI") assiste les hébergeurs en suggérant le prix optimal par nuit pour maximiser leurs revenus tout en restant compétitifs. C'est le résultat d'un projet de Data Science complet détaillé dans le rapport technique associé (*Morocco Airbnb Rental Price Prediction*).

6.1.1 Collecte de Données (Web Scraping)

Faute d'API publique, nous avons construit un pipeline d'extraction de données personnalisé :

- **Cible** : 13 villes majeures du Maroc (Casablanca, Marrakech, Tanger, Rabat, Agadir, etc.).
- **Volume** : Scraping de **65 988 annonces** uniques sur 4 saisons (Printemps, Été, Automne, Hiver 2025).
- **Outil** : Librairie Python `pyairbnb` pour l'extraction de 290 fichiers JSON bruts.

6.1.2 Pipeline ETL & Feature Engineering

Les données brutes ont subi un nettoyage intensif via le script `json_to_csv_pipeline.py` :

1. **Nettoyage** : Suppression des duplicitas et filtrage des outliers (prix > 10 000 MAD).
2. **Feature Engineering (44 variables)** :
 - **Géo-spatial** : Encodage des quartiers (`city_tier`) et coordonnées GPS.
 - **Temporel** : Création de flags comme `is_peak_season` ou `season_summer`.

- **Propriété** : Agrégation via `size_category` (Maison, Villa, Appartement) et détection du standing (`is_luxury`).

6.1.3 Modélisation et Optimisation (XGBoost)

Nous avons comparé plusieurs approches (Régression Linéaire, Random Forest) avant de sélectionner **XGBoost** pour ses performances supérieures.

Stratégie d'Optimisation :

- **Random Search** : Exploration large de 100 combinaisons d'hyperparamètres.
- **Grid Search** : Affinage précis sur les meilleurs paramètres identifiés.
- **Configuration Finale :**
 - `n_estimators=725, learning_rate=0.05` (pour la stabilité).
 - `max_depth=8` (capture des interactions complexes ville/saison).

Résultats : Le modèle final atteint une erreur moyenne absolue (**MAE**) de **48.55 MAD** (~4.5€) et explique plus de **91% de la variance** des prix ($R^2 = 0.9142$), surpassant largement le modèle Random Forest de base.

6.2 Autres Modules Intelligents

6.2.1 Score de Risque Locataire (Tenant Risk Scoring)

Ce modèle de classification binaire évalue la probabilité de défaut de paiement ou de litige.

- **Algorithm** : Random Forest Classifier (optimisé pour gérer le déséquilibre des classes via `class_weight='balanced'`).
- **Inputs** : Historique des transactions (échecs/succès), ancienneté du compte, réclamations passées.
- **Output** : Score de 0 à 100 affiché aux hôtes ("Confiance Élevée", "Risque Modéré").

6.2.2 Segmentation de Marché (Clustering)

Utilise l'algorithme **K-Means** non supervisé pour grouper les propriétés similaires. Cela alimente le moteur de recommandation ("Biens similaires"). L'algorithme maximise le **Silhouette Score** pour déterminer automatiquement le nombre optimal de segments de marché.

6.2.3 Tendances du Marché (Market Trends)

Un module d'analyse de séries temporelles (Régression Linéaire via `np.polyfit`) qui détecte les dynamiques de marché :

- Identification de la tendance (Hausse/Baisse).
- Détection de la saisonnalité et des pics de demande par ville.

6.3 Intégration dans l'Application (Screenshots)

L'IA n'est pas une "boîte noire" cachée, mais une fonctionnalité visible qui apporte de la valeur à chaque étape du parcours utilisateur.

6.3.1 Hôte : Suggestion de Prix Intelligente

Lors de la création d'une annonce, l'IA remplit automatiquement le prix suggéré en fonction des caractéristiques saisies.

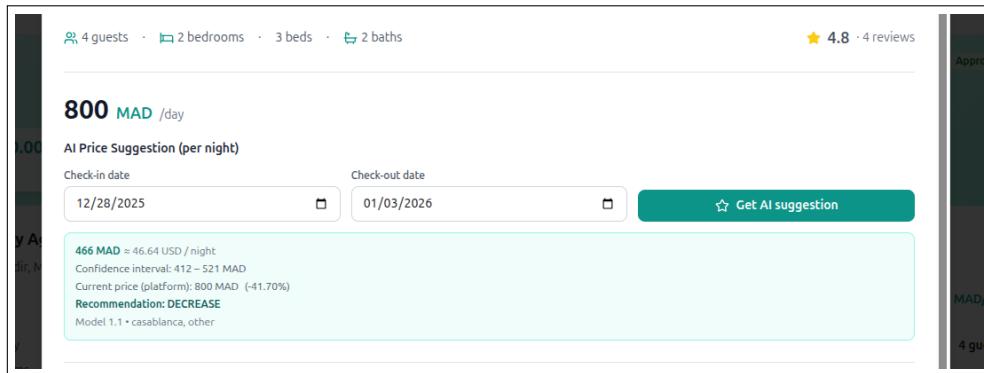


FIGURE 6.1 – Module de suggestion de prix : L'IA propose 466 MAD/nuit.

6.3.2 Hôte : Analyse de Risque Locataire

Avant d'accepter une réservation, l'hôte voit un score de confiance calculé par le modèle de risque.

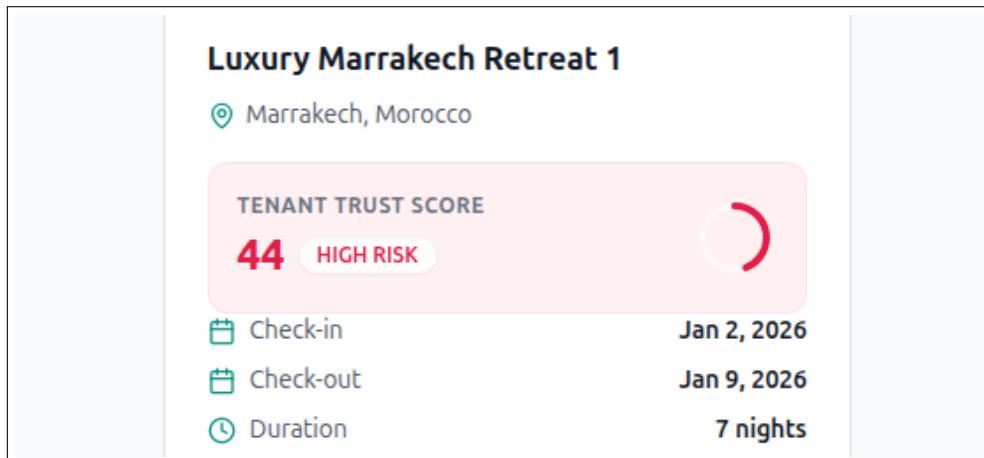


FIGURE 6.2 – Dashboard Hôte : Indicateur de fiabilité du locataire (Score Vert/Orange/-Rouge).

6.3.3 Investisseur : Tableau de Bord du Marché

Visualisation des tendances calculées par le module Market Trends.

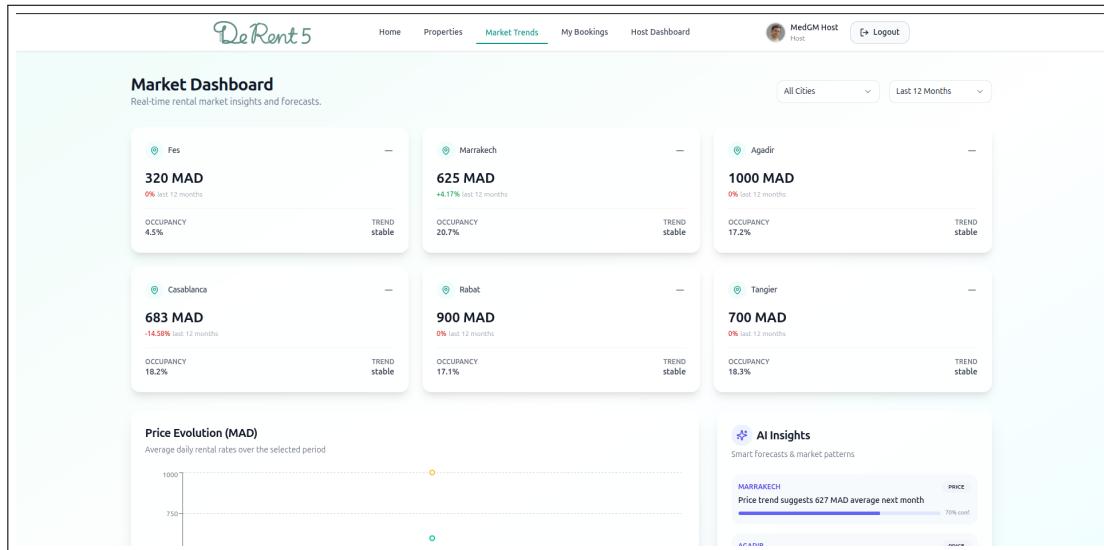


FIGURE 6.3 – Market Insights : Courbes de tendance des prix et taux d'occupation par ville.

6.3.4 Locataire : Recommandations Personnalisées

Le clustering permet de proposer des biens alternatifs pertinents.

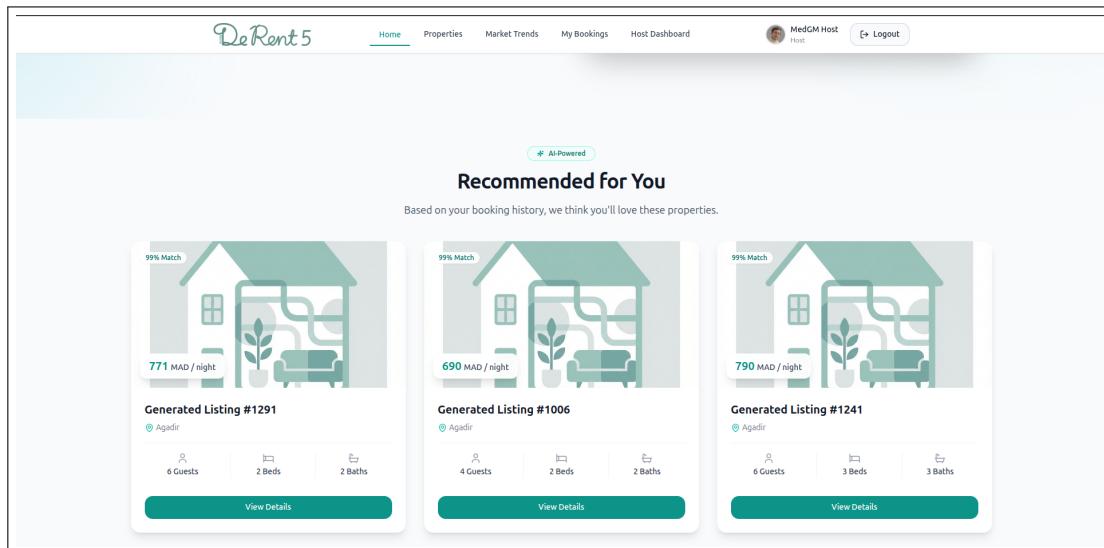


FIGURE 6.4 – Moteur de recommandation "Based on..." basé sur le clustering K-Means.

Chapitre 7

DevOps & CI/CD

L'industrialisation du développement est au cœur du projet *Derent*. Nous avons adopté une approche DevOps stricte pour garantir la qualité du code, la sécurité des livrables et la rapidité des déploiements.

L'ensemble de nos pipelines est défini "as code" via des **Jenkinsfiles**, stockés directement dans les dépôts Git de chaque microservice.

7.1 Orchestration Globale

Pour gérer la complexité de nos 10 microservices, nous avons mis en place un orchestrateur global nommé **Master of Pipeline** ('Jenkinsfile.orchestrator').

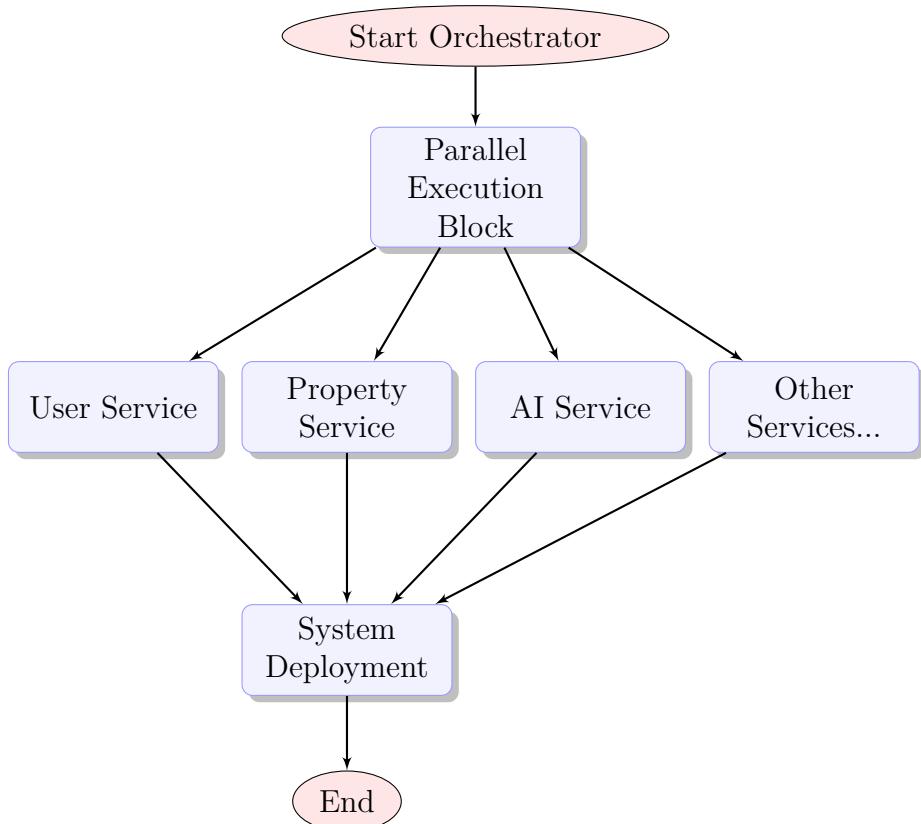


FIGURE 7.1 – Flux d'Orchestration Global : Exécution Parallèle

S	W	Name	Last Success	Last Failure	Last Duration
		Derent-Master-Release	2 hr 11 min #14	4 hr 5 min #11	23 min
		API-Gateway	2 hr 10 min #14	19 days #1	19 min
		blockchain-service	2 hr 10 min #15	N/A	4 min 11 sec
		booking-service	2 hr 10 min #13	N/A	20 min
		notification-service	2 hr 10 min #13	N/A	18 min
		payment-service	2 hr 10 min #13	2 days 22 hr #6	22 min
		property-service	2 hr 10 min #14	2 days 22 hr #8	18 min
		real-estate-frontend	2 hr 10 min #34	5 days 9 hr #19	17 min
		reclamation-service	2 hr 10 min #13	2 days 22 hr #7	22 min
		user-service	2 hr 10 min #61	2 mo 0 days #47	21 min
		morocco-pricing-api	36 min #58	56 min #56	12 min

FIGURE 7.2 – Vue d’ensemble Jenkins : Tous les pipelines (Backend, Frontend, IA)

- **Parallélisme** : Le pipeline lance la construction de tous les services (User, Booking, Property, Blockchain, AI, Frontend) en parallèle pour réduire le temps de feedback.
- **Indépendance** : L’échec d’un service non critique ne bloque pas nécessairement les autres, bien que le build global soit marqué en erreur.

7.2 Pipeline Standard (Backend Java)

Chaque microservice Spring Boot (ex : `user-service`) suit un cycle de vie standardisé en 7 étapes :

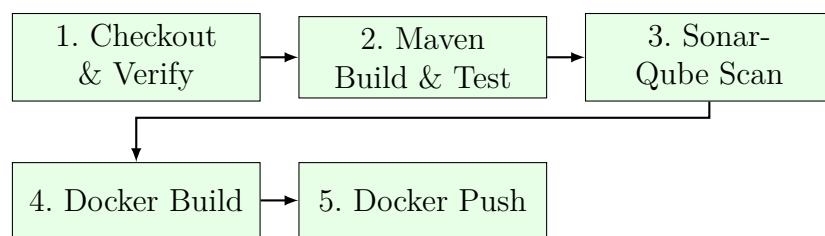


FIGURE 7.3 – Pipeline Standard pour Microservices Java

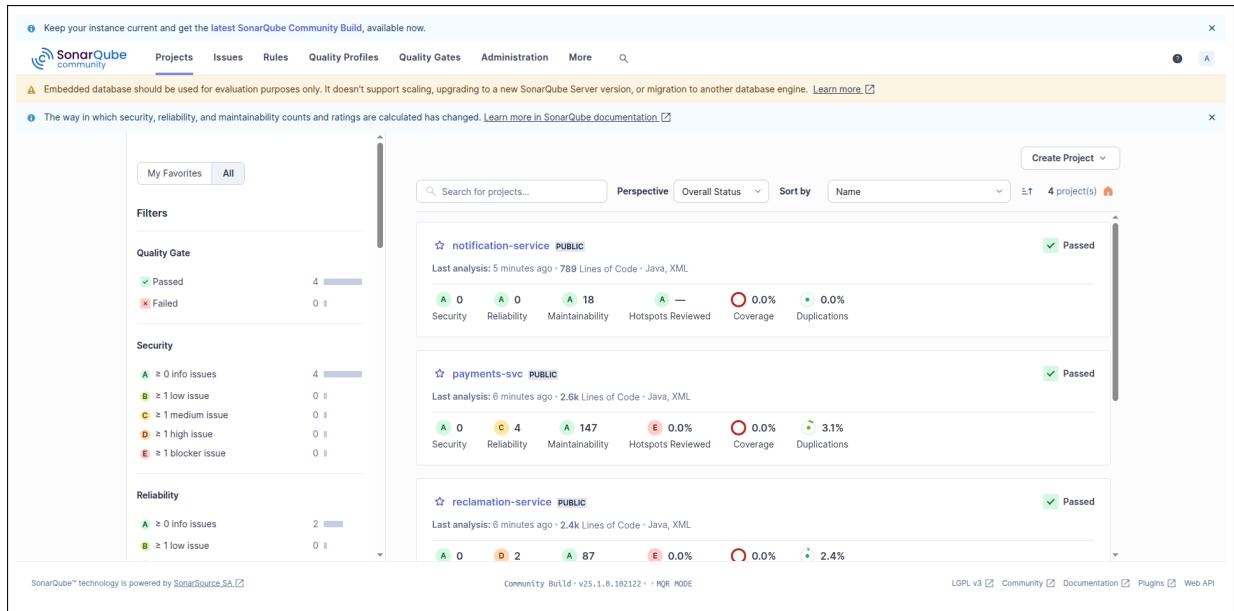


FIGURE 7.4 – SonarQube : Analyse de la qualité et couverture de code

- Checkout & Vérification** : Récupération du code et validation de la structure du projet (présence du `pom.xml`).
- Build & Test** : Compilation avec Maven et exécution des tests unitaires ('mvn clean package').
- Qualité de Code (SonarQube)** : Analyse statique pour détecter les "Code Smells" et bugs potentiels. Le pipeline s'arrête si le "Quality Gate" n'est pas respecté.
- Dockerisation via Multi-Stage Build :**
 - *Stage 1 (Builder)* : Image Maven pour compiler le projet.
 - *Stage 2 (Runtime)* : Image JRE légère ('eclipse-temurin :17-jre-alpine') pour l'exécution.
 - *Sécurité* : Création d'un utilisateur non-root ('appuser') pour l'exécution du conteneur.
- Push au Registre** : Envoi de l'image tagguée (hash du commit) vers Docker Hub et le registre local.

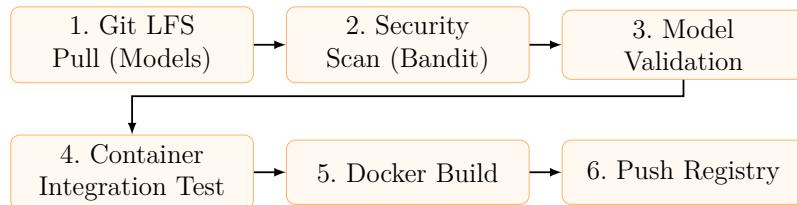


FIGURE 7.5 – Flux CI/CD Avancé pour l'IA

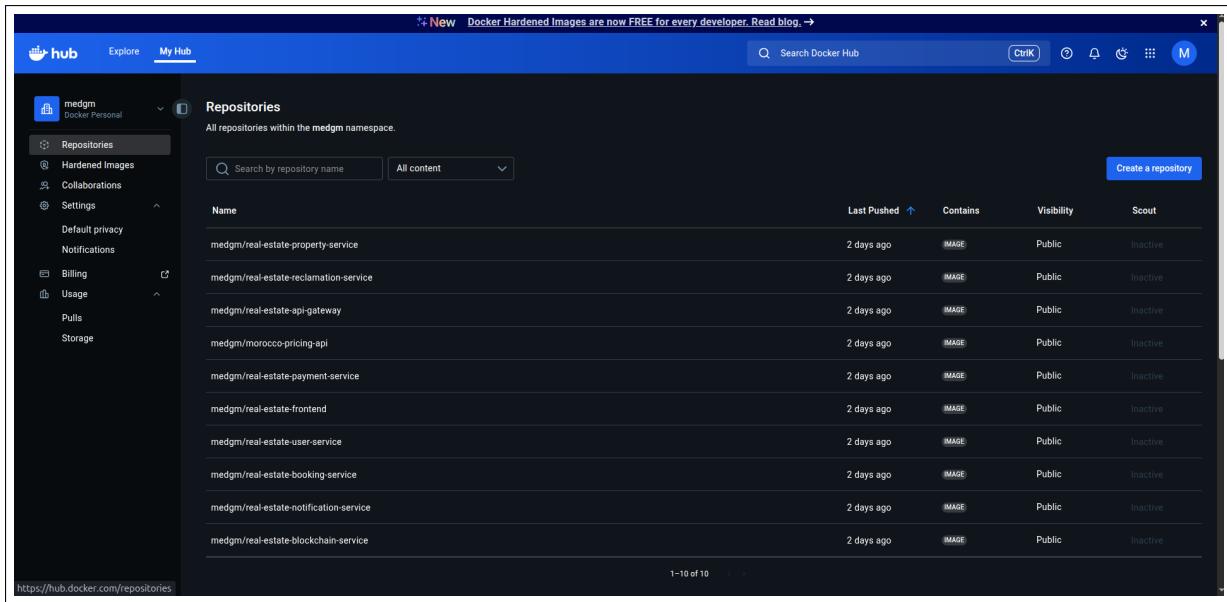


FIGURE 7.6 – Docker Hub : Repository public hébergeant les images multi-arch

- Gestion des Gros Fichiers (Git LFS)** : Le pipeline gère l'authentification Git LFS pour récupérer les modèles entraînés ('.pkl') qui dépassent 100 Mo.
- Scan de Sécurité (Bandit & Safety)** :
 - bandit** : Analyse le code Python pour détecter des failles de sécurité.
 - safety** : Vérifie les dépendances ('requirements.txt') contre une base de vulnérabilités connues (CVE).
- Validation du Modèle** : Un script Python charge le modèle '.pkl' pour vérifier son intégrité et s'assurer qu'il possède bien une méthode 'predict()' avant de construire l'image.
- Test du Conteneur** : Contrairement aux autres services, nous lançons le conteneur IA dans le pipeline pour tester ses endpoints ('/health', '/predict') avec de vraies requêtes HTTP avant la validation finale.

Chapitre 8

Orchestration Kubernetes

Pour gérer le déploiement de nos 10 microservices, nous utilisons **Kubernetes**, le standard de l'industrie pour l'orchestration de conteneurs. En phase de développement, nous utilisons **Minikube** pour simuler un cluster mono-nœud local.

8.1 Architecture du Cluster

L'ensemble des ressources est défini dans le dossier ‘Master/k8s‘, suivant une structure numérotée pour un déploiement ordonné :

- **00-04 (Base)** : ‘Namespace‘, ‘Secrets‘ (mots de passe chiffrés en base64), ‘Config Maps‘, et l’infrastructure de données (‘PostgreSQL‘, ‘RabbitMQ‘).
- **10-17 (Services Backend)** : Déploiements des microservices (User, Property, AI, etc.) exposés via des services **ClusterIP** (le trafic reste interne au cluster).
- **20-30 (Exposition)** : L’‘API Gateway‘ et le ‘Frontend‘ sont exposés via **Node-Port** pour être accessibles depuis l’hôte.
- **40+ (Monitoring)** : Stack d’observabilité complète avec Prometheus et Grafana.

8.2 Déploiement et Opérations

8.2.1 État du Cluster (Pods)

La commande ‘kubectl get pods‘ confirme que tous les composants sont en état ‘Running‘. On observe une séparation claire entre les services métiers, les bases de données et les outils de monitoring.

```
nedgm@medgm-HP-EliteBook-840-G7-Notebook-PC:~/vsc/Projet JEE$ kubectl get pods -n derent -w
NAME                               READY   STATUS        RESTARTS   AGE
api-gateway-68c99f7458-pjk54      0/1    ContainerCreating  0          78s
blockchain-service-6cb95b485b-24pz9 0/1    ContainerCreating  0          78s
booking-service-68b68b6c8b-j8p9j   0/1    ContainerCreating  0          79s
frontend-55c8859b74-q6zr9         0/1    ContainerCreating  0          78s
notification-service-65f5cf4f95-754pq 0/1    ContainerCreating  0          78s
payment-service-b577b9dcc-pnxmw   0/1    ContainerCreating  0          78s
postgres-0                         1/1    Running       0          79s
pricing-api-77d97d5cd6-5k4gd     0/1    ContainerCreating  0          78s
property-service-85cc5b48f6-4nxsn 0/1    ContainerCreating  0          79s
rabbitmq-86587c58d5-457zl        1/1    Running       0          79s
reclamation-service-779b57f455-khx2v 0/1    ContainerCreating  0          78s
user-service-655c99c66c-x8k56     1/1    Running       0          79s
property-service-85cc5b48f6-4nxsn 1/1    Running       0          93s
```

FIGURE 8.1 – Liste des Pods actifs dans le namespace 'derent-ns'

8.2.2 Exposition des Services (NodePort)

Minikube expose une IP locale ('192.168.49.2') permettant d'accéder aux services NodePort :

- **Frontend** : Port **30000** (Access via navigateur).
- **API Gateway** : Port **30090** (Point d'entrée pour les requêtes API).
- **Grafana** : Port **30092**.

```
nedgm@medgm-HP-EliteBook-840-G7-Notebook-PC:~/vsc/Projet JEE$ minikube service list
NAME           NAMESPACE   TARGET PORT   URL
kubernetes     default     No node port
api-gateway    derent      8090          http://192.168.49.2:30090
blockchain-service  derent      No node port
booking-service  derent      No node port
frontend        derent      3000          http://192.168.49.2:30000
notification-service  derent      No node port
payment-service  derent      No node port
postgres-service  derent      No node port
pricing-api      derent      8000          http://192.168.49.2:30002
property-service  derent      No node port
rabbitmq-service  derent      No node port
reclamation-service  derent      No node port
user-service      derent      No node port
kube-dns         kube-system No node port
```

FIGURE 8.2 – Liste des services exposés et URLs d'accès (Minikube Dashboard)

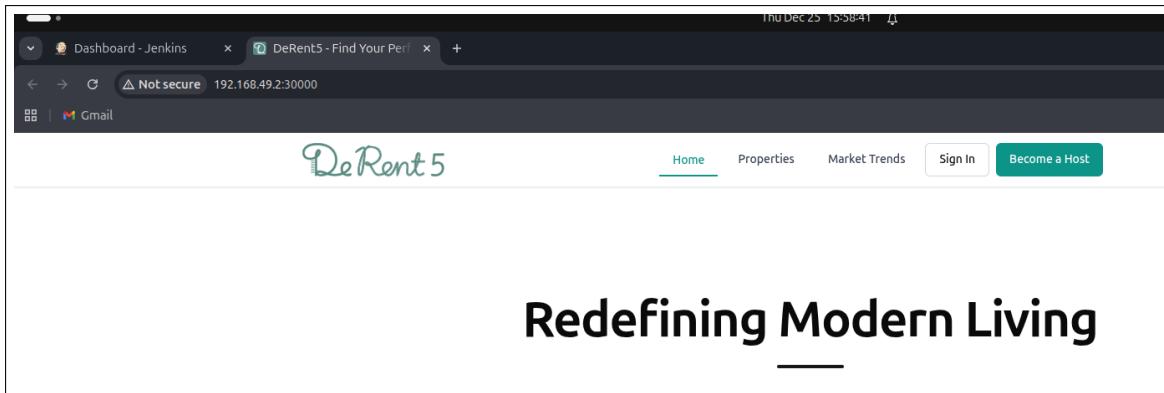


FIGURE 8.3 – Accès réussi au Frontend via le Port 30000 (Forwarding)

8.3 Observabilité et Monitoring

Un cluster distribué nécessite une visibilité accrue. Nous avons déployé une stack de monitoring :

8.3.1 Prometheus (Métriques)

Scrape les métriques techniques (CPU, RAM, JVM) exposées par les endpoints ‘/actuator/prometheus’ des services Spring Boot.

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://10.244.0.108:8081/actuator/prometheus	UP	application="property-service", instance="10.244.0.108:8081", job="spring-boot-actuator"	11.534s ago	4.716ms	
http://10.244.0.107:8082/actuator/prometheus	UP	application="user-service", instance="10.244.0.107:8082", job="spring-boot-actuator"	10.544s ago	4.473ms	
http://10.244.0.112:8091/actuator/prometheus	UP	application="recommendation-service", instance="10.244.0.112:8091", job="spring-boot-actuator"	4.593s ago	4.816ms	
http://10.244.0.109:8083/actuator/prometheus	UP	application="booking-service", instance="10.244.0.109:8083", job="spring-boot-actuator"	9.870s ago	12.693ms	
http://10.244.0.113:8090/actuator/prometheus	UP	application="api-gateway", instance="10.244.0.113:8090", job="spring-boot-actuator"	8.473s ago	8.978ms	
http://10.244.0.111:8084/actuator/prometheus	UP	application="authentication-service", instance="10.244.0.111:8084", job="spring-boot-actuator"	935.000ms ago	4.319ms	
http://10.244.0.110:8085/actuator/prometheus	UP	application="payment-service", instance="10.244.0.110:8085", job="spring-boot-actuator"	9.632s ago	5.901ms	

FIGURE 8.4 – Interface Prometheus : Requête sur l’état des cibles (Targets)

8.3.2 Grafana (Visualisation)

Tableaux de bord connectés à Prometheus pour visualiser la santé du cluster en temps réel.

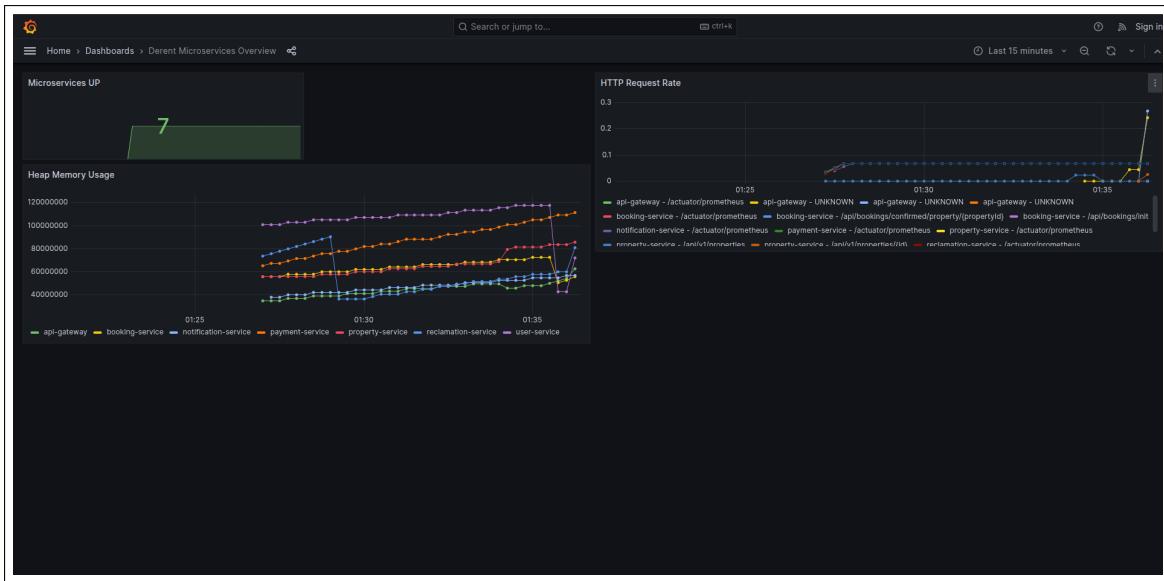


FIGURE 8.5 – Dashboard Grafana : Métriques système et applicatives

Chapitre 9

Infrastructure Cloud (AWS)

Pour le déploiement en production, nous visons le cloud **Amazon Web Services (AWS)** avec une approche *Infrastructure as Code* (IaC) utilisant **Terraform**. Cela garantit une infrastructure reproductible, versionnée et sécurisée.

9.1 Architecture Modulaire Terraform

L'infrastructure est définie dans le dossier ‘Master/terraform‘ et découpée en modules réutilisables pour chaque composant critique :

- **VPC (Network)** : Un réseau isolé (‘10.0.0.0/16‘) avec 3 sous-réseaux publics (pour les Load Balancers) et 3 sous-réseaux privés (pour les nœuds EKS et les bases de données) répartis sur plusieurs zones de disponibilité (AZ).
- **EKS (Compute)** : Cluster Kubernetes managé. Les nœuds de travail (Worker Nodes) sont placés dans les sous-réseaux privés pour la sécurité, et ne sont accessibles que via l’ALB (Application Load Balancer).
- **RDS (Database)** : Instance PostgreSQL managée, configurée avec des sauvegardes automatiques et le mode Multi-AZ pour garantir la haute disponibilité et la tolérance aux pannes.
- **ECR (Registry)** : Registres privés pour stocker de manière sécurisée les images Docker de nos microservices, générées par la pipeline CI/CD.
- **S3 (Storage)** : Stockage objet durable et scalable pour les assets statiques, notamment les images des propriétés et les justificatifs des réclamations.

9.2 Schéma d'Architecture Cible

L'architecture déployée respecte les bonnes pratiques AWS Well-Architected Framework :

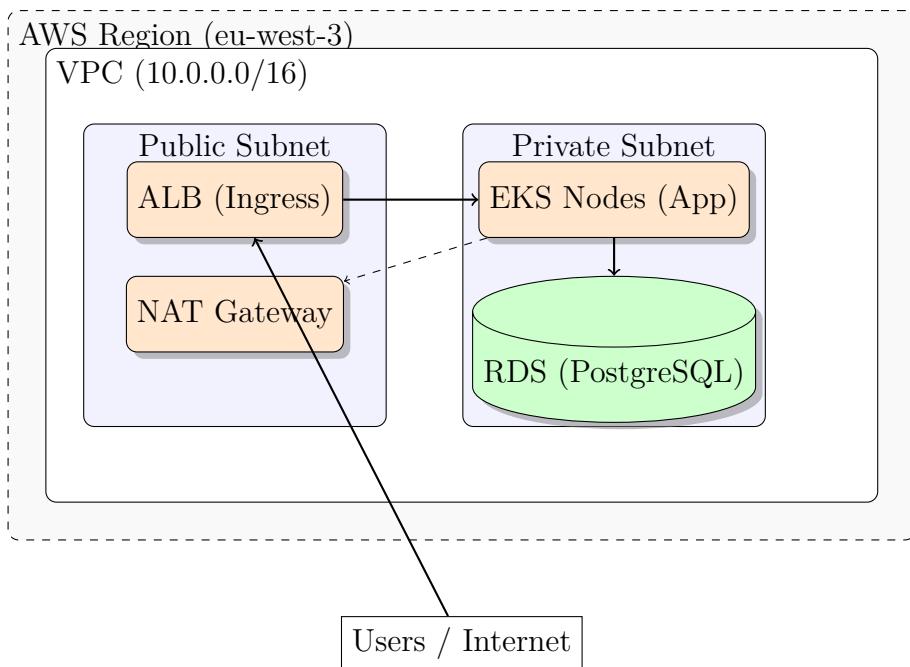


FIGURE 9.1 – Architecture Cloud AWS Cible (Production)

Chapitre 10

Conclusion

Ce projet a permis de réaliser une plateforme de location immobilière innovante, alliant la robustesse des microservices, la sécurité de la blockchain et l'intelligence de l'IA. L'implémentation actuelle est déployée en production sur AWS :

- Frontend Next.js diffusé via **CloudFront** en HTTPS, adossé à un service **EKS** exposé par un Load Balancer.
- API Gateway Spring Cloud sur **EKS**, servie derrière CloudFront (TLS de bout en bout) et connectée aux microservices internes (User, Booking, Property, Payment, Notification, Reclamation).
- **Blockchain Service** exposé en HTTPS via CloudFront pour MetaMask (RPC sécurisé), avec déploiements Hardhat.
- Stockage **S3** pour les médias (photos, pièces jointes) et registre **ECR** pour toutes les images Docker produites par la CI/CD Jenkins.
- Supervision Prometheus/Grafana et logs Kubernetes pour l'observabilité continue.

10.1 Réalisations Clés

- Architecture distribuée et résiliente sur **EKS**, découpée par microservices et sécurisée par le Gateway.
- Chaîne **CI/CD Jenkins** vers **ECR** et déploiements Kubernetes automatisés, pilotés par IaC (Terraform) pour l'infra AWS.
- Expérience utilisateur fluide (Next.js, Tailwind) avec intégration Web3 (MetaMask) et module IA de suggestion de prix.
- Exposition sécurisée via **CloudFront** (frontend, API, RPC) avec certificats TLS et CORS maîtrisé.

10.2 Perspectives

Pour l'avenir, nous envisageons :

- Passage au **Mainnet** ou à un **Layer 2** (ex. Polygon) pour réduire les coûts de gas.
- Enrichissement des modèles IA avec des données réelles et déploiement A/B pour affiner la précision.
- Ajout d'une messagerie temps réel (WebSocket) et d'alertes push pour renforcer l'engagement utilisateur.

- Renforcement de la résilience multi-région (réPLICATION S3, RDS cross-AZ) et mise en place d'un WAF devant CloudFront.