# **Технічне завдання (ТЗ) на розробку платформи з картками, аукціонами та блокчейном**

## **1. Загальний опис проєкту**

### **1.1. Суть гри**

### Гравці можуть зареєструватися на платформі, після чого їм доступні наступні механіки:

### Лотерея – раз на певний період гравець може отримати випадкову картку.

### Колекціонування – зібрані картки зберігаються у профілі гравця.

### Торгівля – можливість продавати/купувати картки на аукціоні.

### Ігрова валюта – гравці отримують внутрішні гроші за активність (вхід у гру, запрошення друзів, продаж карток).

### Блокчейн – всі картки є NFT та закріплюються за власником у блокчейні.

### **1.2. Основні користувачі**

### Гравець – реєструється, отримує картки, бере участь у торгах.

### Адміністратор – модерування аукціонів, управління базою карток.

### **1.3. Основні фічі**

### ✅ Генерація карток з різними рідкісностями ✅ Аукціони та торги між гравцями ✅ Блокчейн для збереження власності карток ✅ Внутрішня валюта та механізм отримання бонусів ✅ Гнучка система API для масштабування

### 

## **2. Архітектура проєкту**

### **2.1. Загальна схема**

### Проєкт розробляється у мікросервісній архітектурі.

### Основні мікросервіси:

### service-user – реєстрація, авторизація, профілі

### service-card – управління картками

### service-lottery – механіка отримання карток

### service-trade – продаж та аукціони

### service-wallet – управління ігровою валютою

### service-blockchain – взаємодія з блокчейном

### service-gateway – єдина точка входу (API Gateway)

### **2.2. Взаємодія сервісів**

### Взаємодія між сервісами реалізована через:

### gRPC – для швидкої синхронної комунікації

### RabbitMQ – для обміну подіями між сервісами

### REST API – для зовнішнього клієнтського інтерфейсу

## **3. Технологічний стек**

### **3.1. Backend**

### ✅ Node.js + NestJS – для розробки мікросервісів ✅ TypeScript – для типізації ✅ PostgreSQL / MongoDB – зберігання користувачів, карток, історії ✅ Redis – кешування ✅ RabbitMQ – брокер повідомлень для обміну між сервісами ✅ gRPC / REST API – для взаємодії між сервісами

### **3.2. Blockchain**

### ✅ Ethereum / Polygon – для збереження NFT-карток ✅ Smart Contracts (Solidity) – смарт-контракти для торгівлі

### **3.3. Frontend**

### ✅ React + Next.js – для веб-інтерфейсу ✅ TailwindCSS – для стилізації

### **3.4. DevOps**

### ✅ Docker + Kubernetes – деплой всіх мікросервісів ✅ CI/CD (GitHub Actions / GitLab CI) – автоматизація деплою

## **4. База даних та основні сутності**

### **4.1. Основні таблиці та їх атрибути**

## **1. Таблиця Users (Користувачі)**

### Призначення: Зберігання даних про користувачів.

### Поля:

### user\_id (PK, INT, AUTO\_INCREMENT) – унікальний ідентифікатор користувача.

### email (VARCHAR) – електронна пошта (унікальне значення).

### password\_hash (VARCHAR) – хеш пароля.

### wallet\_address (VARCHAR) – адреса гаманця в блокчейні.

### balance (DECIMAL) – баланс внутрішньої валюти.

### created\_at (DATETIME) – час створення запису.

### updated\_at (DATETIME) – час останньої зміни запису.

### 

## **2. Таблиця Cards (Картки)**

### Призначення: Зберігання інформації про картки, їх характеристик та поточного стану.

### Поля:

### card\_id (PK, INT, AUTO\_INCREMENT) – унікальний ідентифікатор картки.

### name (VARCHAR) – назва картки.

### rarity (ENUM або посилання на таблицю Rarities) – рідкість (наприклад: 'common', 'rare', 'epic', 'legendary').

### image\_url (VARCHAR) – посилання на зображення картки.

### owner\_id (INT, FK → Users.user\_id) – поточний власник картки.

### status (ENUM: 'available', 'in\_auction', 'sold') – статус картки.

### created\_at (DATETIME)

### updated\_at (DATETIME)

### *Примітка:* Якщо можливе розширення набору значень для рідкості, можна створити окрему таблицю Rarities.

### 

## **3. Таблиця Auctions (Аукціони)**

### Призначення: Організація аукціонів для продажу карток.

### Поля:

### auction\_id (PK, INT, AUTO\_INCREMENT) – унікальний ідентифікатор аукціону.

### card\_id (INT, FK → Cards.card\_id, UNIQUE) – картка, що виставляється на аукціон. *Зауваження: у картки, що перебувають на аукціоні, статус оновлюється до in\_auction.*

### seller\_id (INT, FK → Users.user\_id) – продавець (власник картки під час виставлення).

### start\_price (DECIMAL) – стартова ціна.

### current\_price (DECIMAL) – поточна найвища ставка (може оновлюватися автоматично).

### start\_time (DATETIME) – час початку аукціону.

### end\_time (DATETIME) – час завершення аукціону.

### status (ENUM: 'active', 'closed', 'cancelled') – статус аукціону.

### winner\_id (INT, FK → Users.user\_id, NULLABLE) – переможець аукціону (визначається після завершення).

### created\_at (DATETIME)

### updated\_at (DATETIME)

### *Примітка:* Зберігання поточного стану аукціону дозволяє швидко отримувати інформацію про активні торги.

## **4. Таблиця Bids (Ставки)**

### Призначення: Збереження історії ставок для кожного аукціону.

### Поля:

### bid\_id (PK, INT, AUTO\_INCREMENT) – унікальний ідентифікатор ставки.

### auction\_id (INT, FK → Auctions.auction\_id) – посилання на аукціон.

### bidder\_id (INT, FK → Users.user\_id) – користувач, який зробив ставку.

### bid\_amount (DECIMAL) – сума ставки.

### bid\_time (DATETIME) – час розміщення ставки.

### *Примітка:* Такий підхід дозволяє зберігати повну історію торгів і забезпечує прозорість процесу.

## **5. Таблиця Transactions (Транзакції)**

### Призначення: Фіксація фінансових операцій, пов’язаних із системою (покупки, продажі, лотереї, внутрішні перекази тощо).

### Поля:

### transaction\_id (PK, INT, AUTO\_INCREMENT) – унікальний ідентифікатор транзакції.

### sender\_id (INT, FK → Users.user\_id, NULLABLE) – відправник коштів (може бути NULL для надходжень, наприклад, від системи чи лотереї).

### receiver\_id (INT, FK → Users.user\_id, NULLABLE) – отримувач коштів.

### amount (DECIMAL) – сума транзакції.

### type (ENUM: 'purchase', 'sale', 'lottery', 'transfer') – тип транзакції.

### related\_auction\_id (INT, FK → Auctions.auction\_id, NULLABLE) – посилання на аукціон (якщо транзакція пов’язана з торгами).

### related\_card\_id (INT, FK → Cards.card\_id, NULLABLE) – посилання на картку (за необхідності).

### created\_at (DATETIME)

### *Примітка:* Поля sender\_id та receiver\_id дозволяють гнучко обробляти різні типи операцій.

### 

## Додаткові зауваження та логіка взаємодії

### Зв’язки та цілісність даних:

### Для всіх зовнішніх ключів слід задати відповідні обмеження (foreign key constraints), що допомагає підтримувати цілісність даних.

### У випадку, коли картка виставляється на аукціон, необхідно оновити її статус (наприклад, змінити з available на in\_auction), а після завершення аукціону – відповідно, змінити власника та статус на sold або повернути статус у випадку скасування.

### Історія ставок:

### Таблиця Bids дозволяє зберігати повну історію торгів, що є важливим для аудиту та визначення переможця аукціону.

### current\_price в таблиці Auctions може оновлюватися при кожній новій ставці.

### Транзакції:

### Всі фінансові операції (наприклад, зарахування коштів від продажу, списання при покупці, участь у лотереї) фіксуються у таблиці Transactions.

### Поля related\_auction\_id та related\_card\_id допомагають встановити зв’язок між транзакцією і конкретним аукціоном чи карткою, що може бути корисно для розширеної аналітики.

### Модульність:

### При необхідності можна додати додаткові таблиці (наприклад, Rarities для гнучкого управління категоріями рідкості або таблицю Wallet\_Transactions для детального обліку операцій із блокчейном).

## **5. Бізнес-логіка мікросервісів**

### **5.1. Алгоритм отримання картки**

### Гравець запускає лотерею в service-lottery

### Система визначає випадкову рідкість картки

### Картка створюється у service-card

### Власник записується у blockchain через service-blockchain

### Картка з’являється у профілі гравця

### **5.2. Алгоритм продажу картки**

### Гравець додає картку на аукціон у service-trade

### Інші користувачі можуть робити ставки

### Коли аукціон завершується:

### Гроші переказуються через service-wallet

### Власність змінюється у blockchain

### Картка переходить до нового власника

### 

## **6. Безпека**

### Використання JWT для аутентифікації

### Шифрування паролів bcrypt

### Захист API за допомогою rate limiting

### Захист фінансових операцій через 2FA

### 

## **7. Масштабування**

### **🔹** Горизонтальне масштабування мікросервісів через Kubernetes 🔹 Використання Redis для кешування запитів 🔹 CDN (Cloudflare / AWS CloudFront) для оптимізації завантаження карток

### 

## **8. Очікувані результати**

### ✅ Запуск MVP-версії гри з базовими механіками ✅ Інтеграція блокчейну для управління картками ✅ Запуск торгового майданчика для карток

## **9. Моніторинг та логування**

### **9.1. Моніторинг мікросервісів**

### Щоб відстежувати стан сервісів і швидко реагувати на проблеми, використовуємо:

### Prometheus – збір метрик (нагрузка CPU, RAM, час відповіді сервісів)

### Grafana – візуалізація метрик та алерти

### Kubernetes Health Checks – перевірка живучості сервісів (livenessProbe, readinessProbe)

#### Основні метрики:

### ✅ Запити до API (кількість, успішні/помилки) ✅ Час відповіді сервісів (response time) ✅ Кількість активних користувачів ✅ Використання ресурсів (CPU, RAM, дисковий простір) ✅ Кількість подій у RabbitMQ (чи не зависла черга) ✅ Статистика транзакцій та аукціонів

### **9.2. Логування**

### Для логування подій та помилок використовуємо Winston + Elasticsearch:

### Winston – централізоване логування у всіх мікросервісах

### Elasticsearch + Kibana – збір логів та їх перегляд у реальному часі

#### Типи логів:

### 🟢 Інформаційні (INFO) – запити користувачів, зміни у профілях 🟠 Попередження (WARNING) – затримки в обробці запитів, нестача ресурсів 🔴 Помилки (ERROR) – збої сервісів, невдалі транзакції, критичні баги

#### Приклад формату логів (json):

### **{**

### **"timestamp": "2024-02-01T12:34:56Z",**

### **"level": "error",**

### **"service": "service-trade",**

### **"message": "Аукціон не завершився через помилку бази даних",**

### **"context": {**

### **"auctionId": "12345",**

### **"sellerId": "67890"**

### **}**

### **}**

### 

### **9.3. Сповіщення про критичні помилки**

### 📌 Slack / Telegram боти – надсилання повідомлень у канал розробників 📌 Email-сповіщення – якщо сервіс падає більше ніж на 5 хвилин 📌 Webhook-інтеграції – підключення до PagerDuty для швидкого реагування

### **9.4. Автоматичне відновлення сервісів**

### ⚙ Kubernetes Auto Healing – якщо сервіс впав, Kubernetes його перезапустить ⚙ Rate Limiting + Circuit Breaker – захист від перевантаження та DDoS

### **Пропонована мікросервісна структура:**

📂 game-platform/ (Головний репозиторій)  
├── 📂 service-user/ – Сервіс управління користувачами (реєстрація, аутентифікація)  
├── 📂 service-card/ – Сервіс карток (створення, отримання, перегляд)  
├── 📂 service-lottery/ – Сервіс лотереї (генерація карток, розіграш)  
├── 📂 service-trade/ – Сервіс торгівлі (аукціони, продаж, обмін карток)  
├── 📂 service-wallet/ – Сервіс внутрішньої валюти (перекази, баланс, платежі)  
├── 📂 service-blockchain/ – Інтеграція з блокчейном (збереження власності карток)  
├── 📂 service-gateway/ – API Gateway (єдина точка входу для всіх клієнтів)  
├── 📂 libs/ – Спільні бібліотеки (DTO, обмін повідомленнями)  
├── 📂 docker/ – Конфігурації для деплою  
└── 📄 docker-compose.yml – Запуск всіх сервісів

### 

### **Коротко про мікросервіси:**

1️⃣ **service-user**

* Реєстрація, вхід, управління профілем
* Використовує **JWT** для аутентифікації
* Зберігає користувачів у базі (MongoDB/PostgreSQL)

2️⃣ **service-card**

* Генерація карток
* Отримання списку карток користувача
* Взаємодія з **service-blockchain** для власності

3️⃣ **service-lottery**

* Механізм отримання карток через лотерею
* Обчислення рідкісності картки
* Підключення до **service-card**

4️⃣ **service-trade**

* Додавання картки на аукціон
* Купівля/продаж карток між гравцями
* Використовує **service-wallet** для платежів

5️⃣ **service-wallet**

* Управління внутрішньою валютою
* Баланс користувача
* Платежі за картки, бонуси за вхід

6️⃣ **service-blockchain**

* Збереження власності карток на блокчейні
* Перевірка транзакцій

7️⃣ **service-gateway**

* **API Gateway** для всіх запитів
* Глобальна аутентифікація

### **Взаємодія мікросервісів:**

✅ Використовуємо **RabbitMQ** для асинхронного обміну подіями  
✅ Всі сервіси працюють через **gRPC/HTTP**✅ Деплой в **Docker + Kubernetes**