ГРАЛЬНА КОМОРА (NE_PROGRAMISTY)

Сучасний E-commerce вебзастосунок для продажу настільних ігор

Команда: Ne_Programisty

Дата здачі: 16.12.2025

СКЛАД КОМАНДИ

🚀 Команда «Ne_Programisty»

- 🐻 **Лукащук Данило** — *Team Lead / Fullstack*
  Архітектура проєкту, Docker & Nginx, Адмін-панель.
- 🦊 **Чорноус Сергій** — *Frontend Dev*
  Дизайн інтерфейсу, Кошик, Логіка замовлень.
- 🐻 **Базюк Максим** — *Backend Dev*
  Система відгуків (Feedback), Новини, анімації.
- 🐼 **Кондратюк Дмитро** — *QA / UI/UX Designer*
  Тестування (Bug hunting), Адаптивність.

ОГЛЯД ПРОБЛЕМИ

Проблема: Ринок настільних ігор зростає, але користувачам часто не вистачає зручних, швидких та сучасних платформ для їх придбання. Багато існуючих сайтів мають застарілий дизайн, складну навігацію або відсутність мобільної адаптації.

Цільова аудиторія:

- Фанати настільних ігор (гіки).
- Компанії друзів, що шукають розваги.
- Батьки, що шукають розвиваючі ігри для дітей.

Актуальність: Створення "лампового", але технологічного простору для ком'юніті з можливістю швидкої покупки, читання новин та обміну відгуками.

РІШЕННЯ

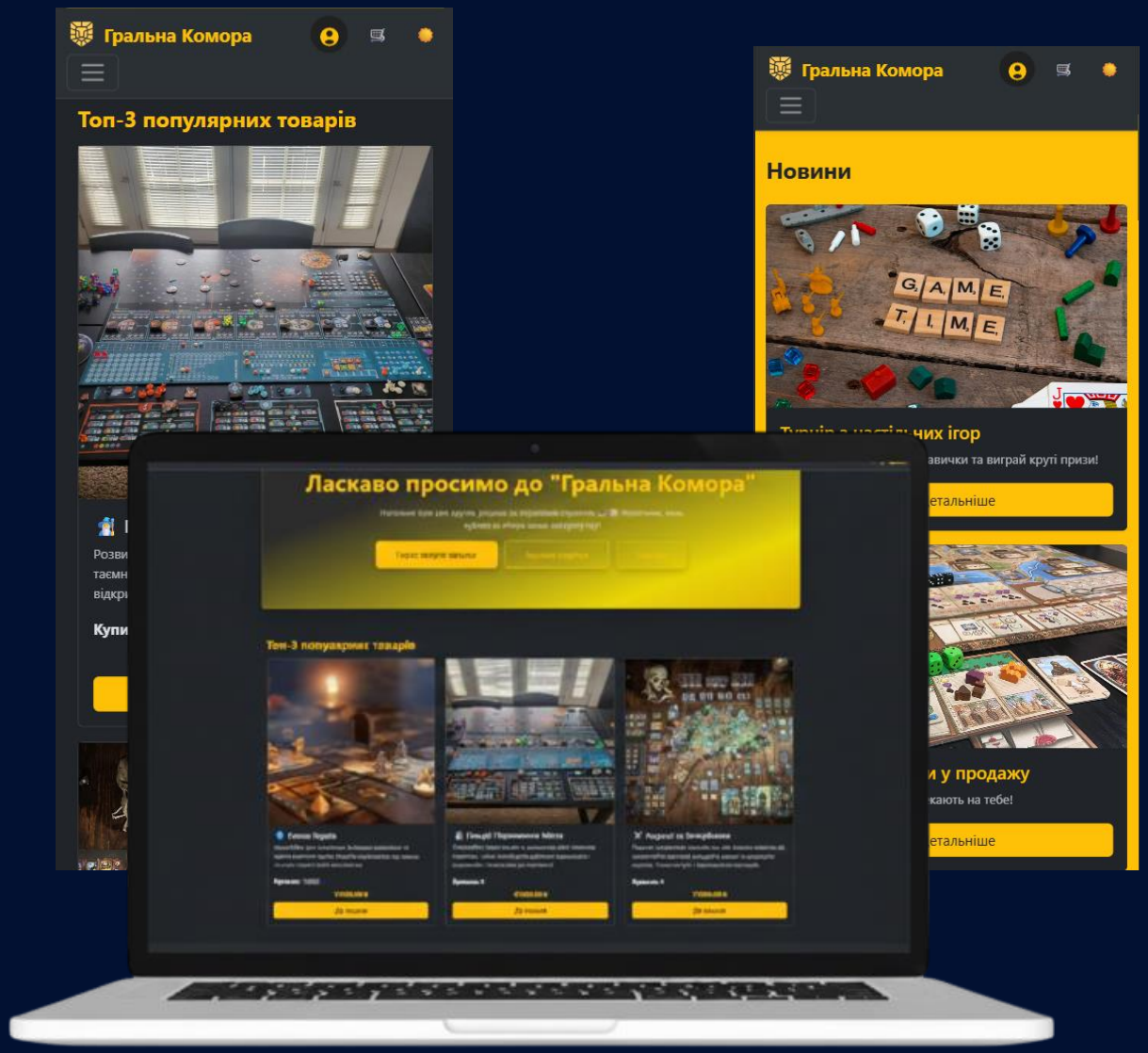
Наше рішення: **"Гральна Комора"** — це повнофункціональний вебсервіс, що поєднує в собі каталог товарів, CMS (систему управління контентом) та RESTful API.

Ключова цінність:

- Швидкість: Оптимізований бекенд та легкий фронтенд.
- Зручність: Адаптивний дизайн (Mobile-First) та інтуїтивний UX.
- Надійність: Безпечна авторизація та стабільна робота завдяки контейнеризації.

Унікальні особливості:

- Власні CSS-анімації та лоадер.
- Темна/Світла тема.
- Гнучка система ролей (User, Moderator, Admin).



ТЕХНОЛОГІЧНИЙ СТЕК

Ми обрали сучасний стек для балансу між швидкістю розробки та продуктивністю.

Frontend:

- **HTML5 / CSS3** (Custom Animations)
- **Bootstrap 5** (Адаптивність)
- **JavaScript (ES6+)** (Fetch API, динаміка)

Backend:

- **Python 3.13**
- **Flask 3.0** (Web Framework)
- **SQLAlchemy + Alembic** (ORM & Migrations)
- **Flask-JWT-Extended** (Security)

Інфраструктура:

- **Docker & Docker Compose** (Containerization)
- **Nginx** (Reverse Proxy)
- **Git & GitHub** (Version Control)

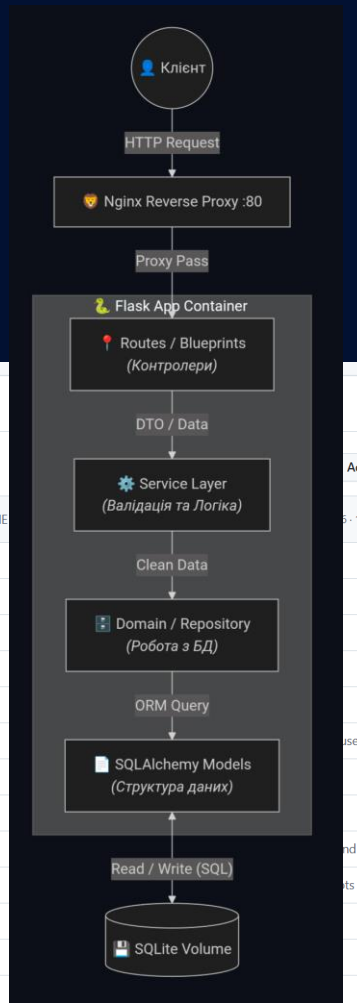
Архітектура системи

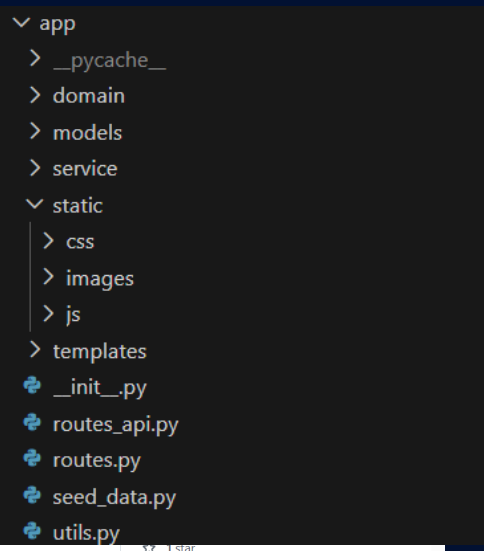
Ми реалізували **Layered Architecture** (Багатошарову архітектуру) для чистоти коду.

Внутрішня структура Flask:

1. **Routes:** Приймають запити (Controllers).
2. **Service Layer:** Бізнес-логіка та валідація даних.
3. **Domain/Repository:** Пряма робота з БД.
4. **Models:** Опис таблиць SQLAlchemy.
5. **Database:** SQLite Volume.

Взаємодія: Nginx приймає запити, передає їх у Flask, де вони проходять через шари валідації та бізнес-логіки перед записом у БД.

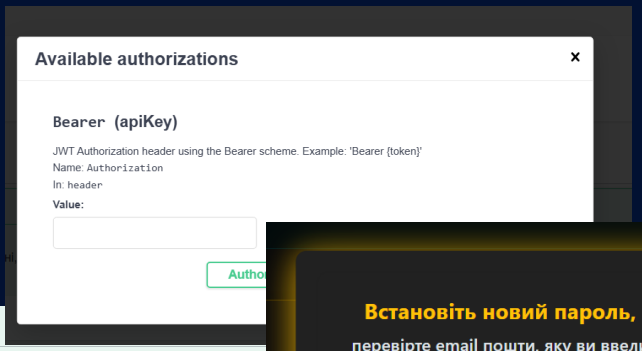


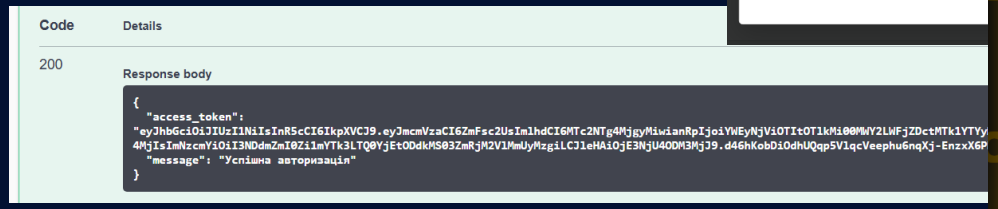


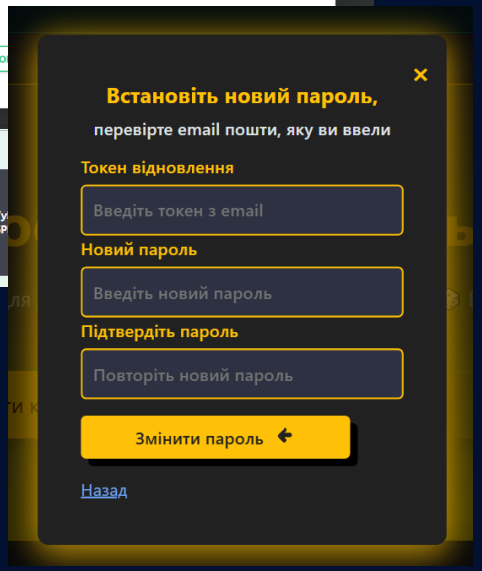
ОСНОВНИЙ ФУНКЦІОНАЛ

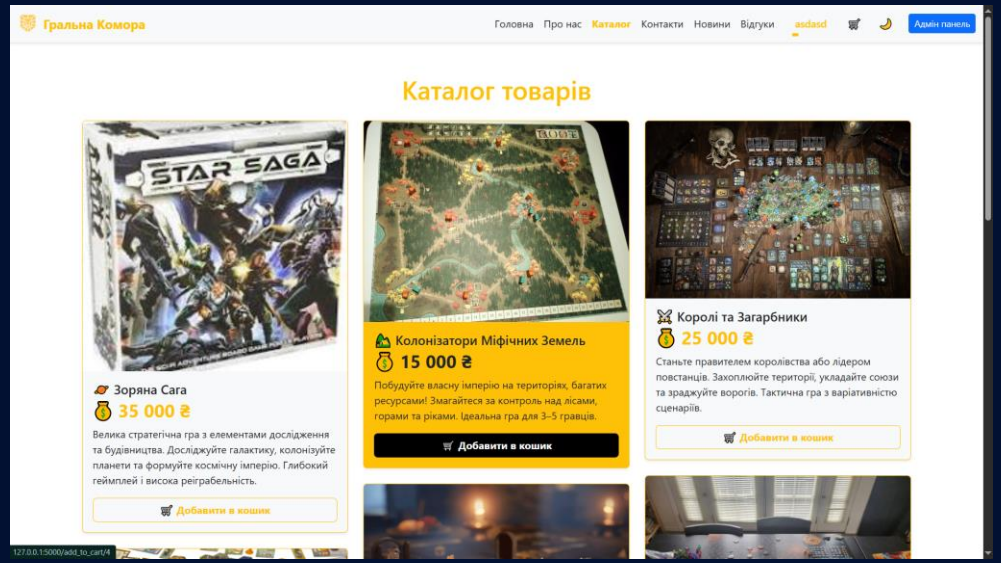
Аутентифікація:

- Реєстрація та вхід (JWT Access/Refresh токени).
- Відновлення пароля через email.









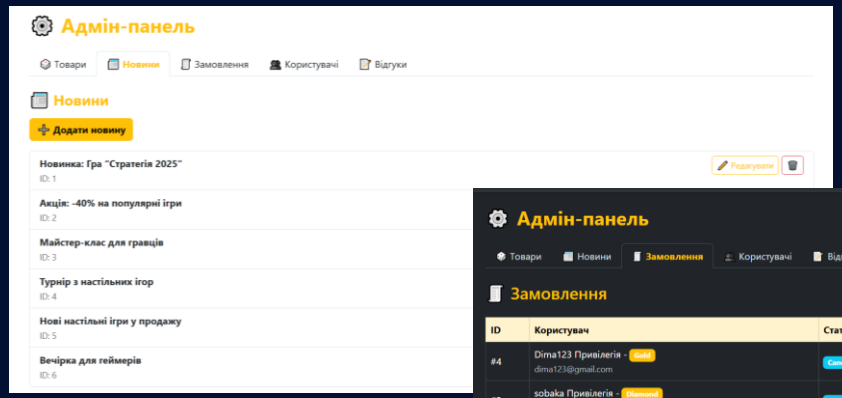
E-commerce:

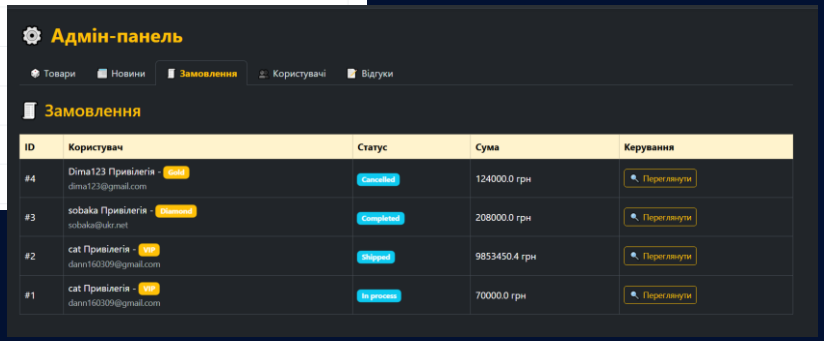
- Каталог товарів з фото та описом.
- Кошик (додавання, зміна кількості, видалення).
- Оформлення замовлення (Checkout).

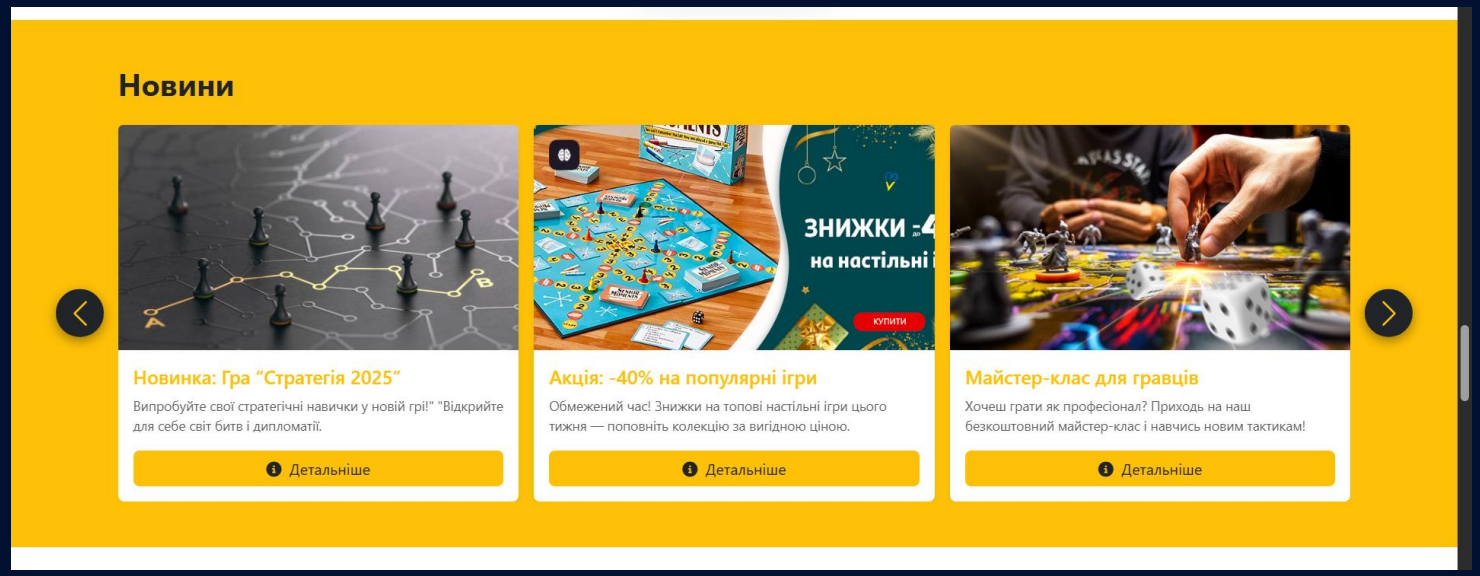
ОСНОВНИЙ ФУНКЦІОНАЛ

Адмін-панель:

- CRUD для товарів, новин, користувачів.
- Модерація відгуків.
- Керування статусами замовлень.







Контент:

- Стрічка новин.
- Система відгуків користувачів.

НОВІ ФУНКЦІЇ (LAB 8)

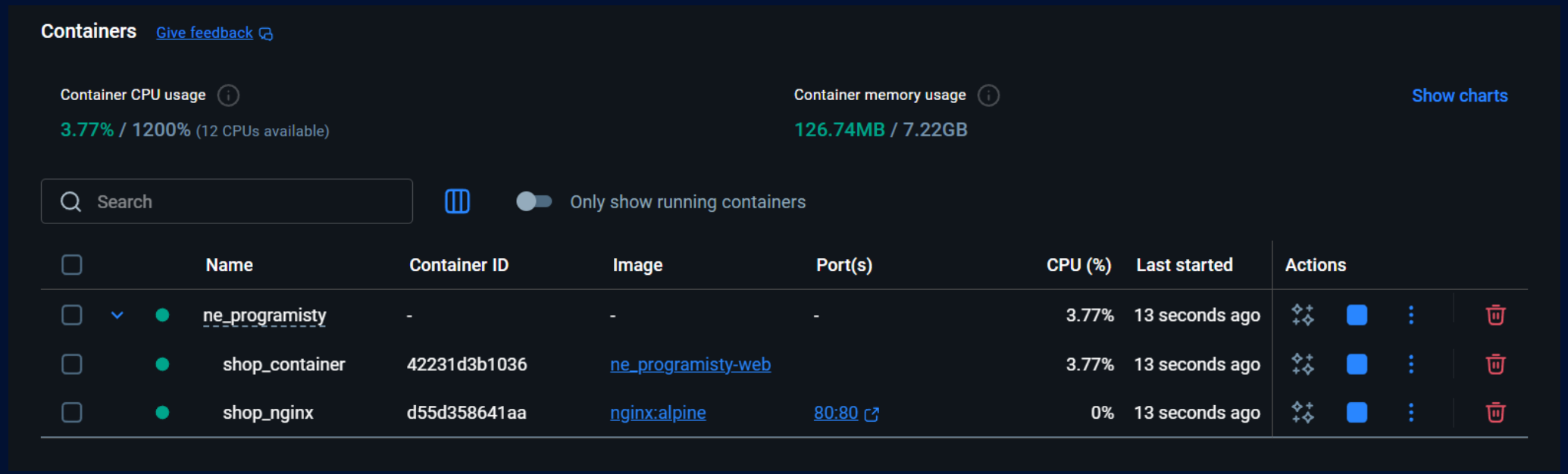
В рамках останнього етапу ми впровадили **Контейнеризацію.**

Що додано:

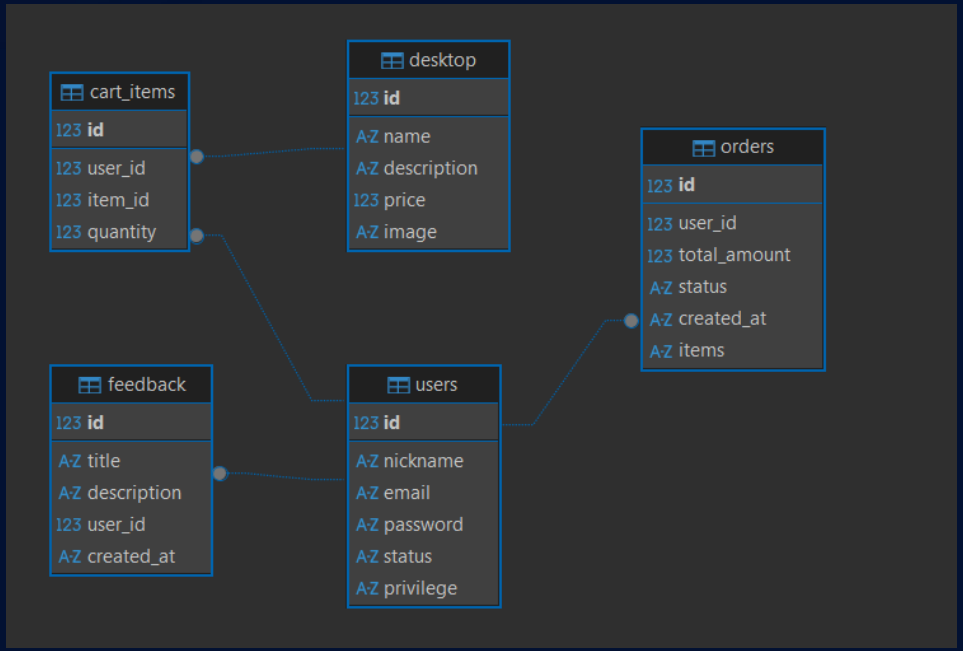
- **Docker Support:** Створено оптимізований **Dockerfile** на базі **python:3.11-alpine.**
- **Orchestration:** Налаштовано **docker-compose** для зв'язку сервісів **web** та **nginx.**
- **Nginx Configuration:** Налаштовано проксі-сервер для роздачі статики та захисту застосунку.

Як це покращує застосунок:

- **Ізоляція:** Жодних конфліктів залежностей ("It works on my machine" — у минулому).
- **Легкий деплой:** Розгортання однією командою **docker-compose up.**



СТРУКТУРА БАЗИ ДАНИХ



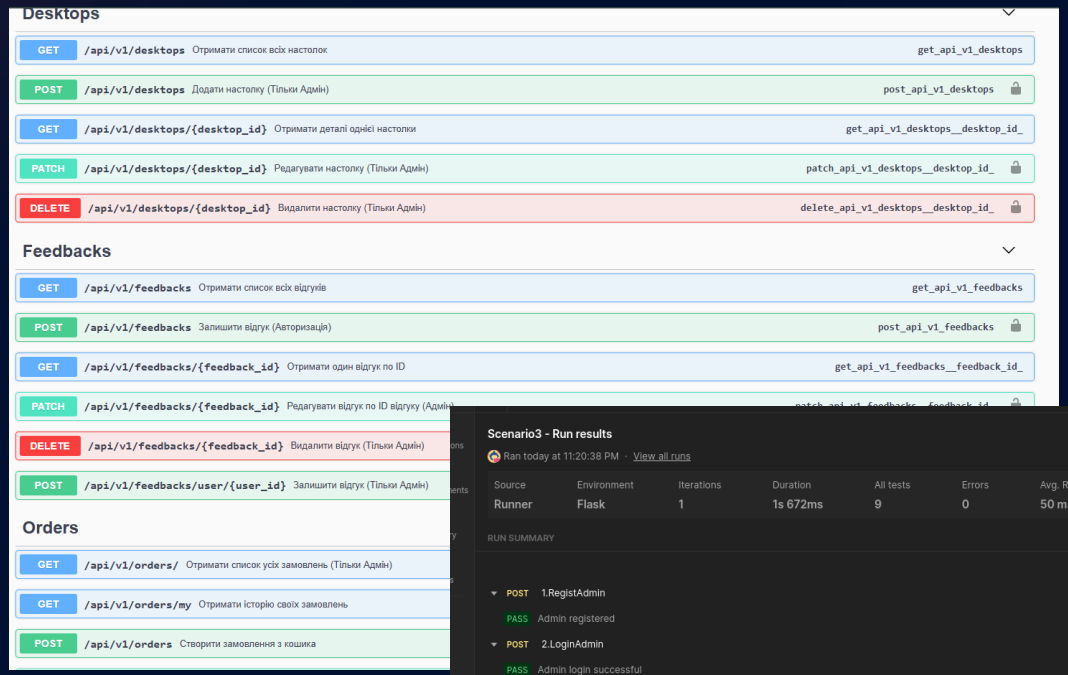
Використовується реляційна модель даних (SQLite).

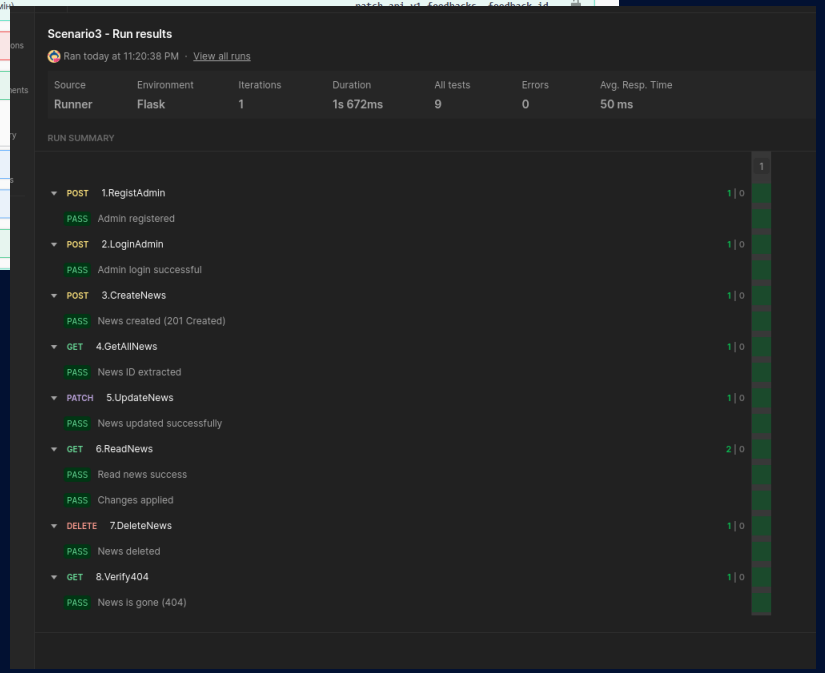
Основні сутності:

- **Users:** id, nickname, email, password_hash, role.
- **Desktop (Products):** id, name, price, image.
- **CartItems:** Зв'язок User <-> Desktop (Many-to-One).
- **Orders:** Історія покупок, статус, сума.
- **Feedback:** Відгуки користувачів.
- **News:** Новини магазину.

Зв'язки: Користувачі мають кошики та замовлення; Відгуки прив'язані до користувачів.

API ENDPOINTS





Ми розробили RESTful API, документоване через **Swagger UI**.

Приклади:

- POST /api/v1/auth/login — Отримання токена.
- GET /api/v1/desktops — Отримати каталог товарів.
- POST /api/v1/carts — Додати товар у кошик.
- GET /api/v1/orders/my — Перегляд історії замовлень.
- DELETE /api/v1/users/{id} — Видалення користувача (Admin only).

ТЕХНІЧНІ ВИКЛИКИ

Проблеми та рішення:

1. 🔘 **Infinite Reload Loop:** Фронтенд зациклювався при невалідному токені.
   ✅ *Рішення:* Виправлено ендпоінт (/carts замість /cart) та додано Circuit Breaker в JS (sessionStorage flag).

2. 🔘 **Permissions у Docker:** Помилка запису в БД при локальному запуску скриптом app.py.
   ✅ *Рішення:* Реалізовано "розумний" конфіг, який визначає середовище і підставляє абсолютний шлях до БД.

3. 🔘 **Nginx Static Files:** Проблема з відображенням картинок через проксі.
   ✅ *Рішення:* Правильне налаштування Volumes та директиви proxy_pass у nginx.conf.

ЦІКАВІ ТЕХНІЧНІ РІШЕННЯ

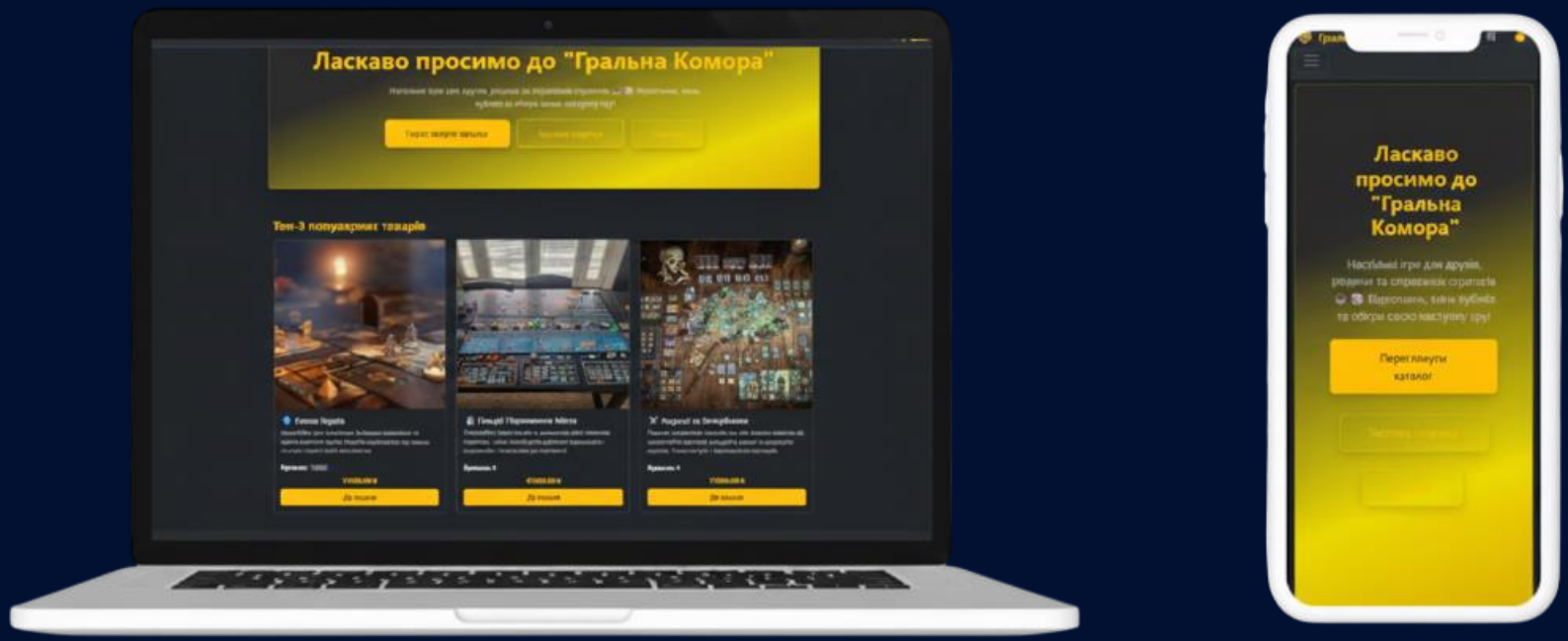
1. Service Layer Pattern:
Винесення бізнес-логіки з роутів у сервіси. Це робить код чистим і легким для тестування.

2. Docker Multi-stage Build:
Використання python:3.11-alpine та поетапної збірки дозволило зменшити розмір образу до ~120 MB, що пришвидшує деплой.

```python
@api.route("/carts/clear", methods=["DELETE"])
@jwt_required()
def clear_cart_endpoint():
    # 1. Отримання ID користувача з токена
    current_user_id = get_jwt_identity()

    # 2. Виклик сервісної функції
    # Припускаємо, що CartService - це клас, який містить clear_cart
    try:
        CartService.clear_cart(user_id=current_user_id)

        # 3. Успішна відповідь
        return jsonify({
            "message": "Ваш кошик успішно очищено.",
            "user_id": current_user_id
        }), 200

    except Exception as e:
        # Обробка можливих помилок бази даних або сервісу
        print(f"Помилка при очищенні кошика користувача {current_user_id}: {e}")
        return jsonify({"error": "Не вдалося очистити кошик через внутрішню помилку сервера."}), 500
```

```dockerfile
# === ЕТАП 1: Будівельник (Builder) ===
FROM python:3.11-alpine AS builder

# Робоча папка
WORKDIR /app

# Встановлюємо компілятори, потрібні для
# Це потрібно, щоб pip install не впав з
RUN apk add --no-cache gcc musl-dev libf

# Копіюємо список бібліотек
COPY requirements.txt .

# Встановлюємо залежності в user site-pa
RUN pip install --no-cache-dir --user -r

# === ЕТАП 2: Фінальний (Final) ===
FROM python:3.11-alpine

# Щоб Python показував логи відразу
ENV PYTHONUNBUFFERED=1

# Робоча папка
WORKDIR /app

# Копіюємо встановлені бібліотеки з етап
COPY --from=builder /root/.local /root/.
ENV PATH=/root/.local/bin:$PATH

# Встановлюємо wget для healthcheck
RUN apk add --no-cache wget

# Копіюємо ВЕСЬ код проекту в контейнер
COPY . .

# Підвантажуємо змінні середовища з .env
RUN pip install --no-cache-dir python-dc
```

ДЕМОНСТРАЦІЯ



МОЖЛИВОСТІ РОЗВИТКУ

Roadmap:

1. **REST-Api:** Перехід всього сайту на АПІ-запити
2. **PostgreSQL:** Міграція з SQLite для кращої продуктивності під навантаженням.
3. **Redis:** Кешування запитів API та сесій.
4. **Платіжна система:** Інтеграція LiqPay або Stripe.
5. **HTTPS:** Налаштування Certbot (Let's Encrypt) у Nginx-контейнері.
6. **Mobile App:** Розробка повноцінногшо мобільного застосунку, використовуючи наш готовий API.

ВИСНОВКИ

Результати:

Ми пройшли повний шлях SDLC (Software Development Life Cycle): від ідеї та макетів до контейнеризованого продукту, готового до деплою в хмару (Render.com).

Здобуті навички:

- Робота з **Git Flow** та вирішення конфліктів.
- Проєктування **REST API** та баз даних.
- Написання **Unit & Integration тестів**.
- **DevOps:** Docker, Nginx, CI/CD основи.

Проєкт "Гральна Комора" готовий до масштабування!

Дякуємо за увагу!