ChatGPT

Дорожня карта переходу до архітектури IUI v4 з єдиним Core API

1. Нова архітектура: єдине ядро, DRY та SRP-принципи

Мета: Відділити бізнес-логіку від клієнтів і зібрати її в одному центральному сервісі (Core API), який обслуговує всі платформи – Telegram, Discord, Web, Mobile. Це забезпечить дотримання принципів **DRY** (Don't Repeat Yourself) та **SRP** (Single Responsibility Principle). Згідно з принципом DRY, кожен шматок логіки має бути реалізований в одному місці – в нашому випадку в ядрі API [1]. Це підвищує підтримуваність: зміни в логіці вносяться один раз і автоматично застосовуються для всіх клієнтів, що зменшує ризик розбіжностей та помилок [2]. Одночасно SRP гарантує, що кожен компонент системи має одну чітку зону відповідальності – ядро займається **бізнес-функціями**, тоді як клієнтські додатки займаються **презентацією та взаємодією з користувачем**.

Концепція: У новій архітектурі **IUI v4** всі клієнти звертаються до централізованого **IUI Core API**. Клієнти (боти Telegram, Discord, веб-інтерфейс, мобільний додаток) більше не містять дубльованої логіки – вони лише відправляють запити до Core API і відображають результати. **Бізнес-правила, обробка даних, робота з БД та сторонніми сервісами виконуються єдиним ядром.** Такий підхід відповідає практиці, коли **REST API виступає "view" (представленням) бізнес-логіки, подібно до UI** [3]. Ідеально, щоб бізнес-логіка не залежала від того, хто її викликає – Telegram-бот чи веб-клієнт – вона працює однаково для всіх. **Клієнти стають "тонкими оболонками"**, що відповідають лише за прийом введення користувача та вивід відповіді, без дублювання внутрішньої логіки.

Приклад архітектури: Єдине ядро (сервер) обслуговує декілька різних клієнтів. Усі клієнти – чи то десктоп, мобільний додаток або бот – роблять однакові запити до центрального API [4].

Переваги підходу:
- *Єдиний "авторитетний" шар логіки:* будь-яка бізнес-функція реалізована один раз (відповідно до принципу "single authoritative representation" [1]), тому відсутній дублюючий код у різних клієнтах.
- *Простіше розширення на нові платформи:* новий клієнт можна підключити, реалізувавши тільки інтерфейс для користувача, без переписування логіки.
- *Легше тестування та підтримка:* можна окремо тестувати Core API (усю бізнес-логіку), а клієнти тестувати в основному на правильність інтеграції та відображення. Зміни в бізнес-правилах не потребують редагування коду кожного клієнта – достатньо змінити ядро.
- *Чіткий поділ обов'язків (Separation of Concerns):* ядро відповідає за опрацювання команд, зберігання даних, взаємодію з AI/базами даних, а клієнти – за специфіку платформи (наприклад, форматування повідомлень Telegram, Discord-слеш-команди, веб-UI тощо). Такий поділ відповідає рекомендаціям, де контролер/сервер не прив'язаний до конкретного інтерфейсу [3] – UI можна змінювати або додавати без впливу на бізнес-логіку.

2. Проєктування ядра IUI Core API

Технологічний стек ядра: Ми створюємо окремий асинхронний веб-сервіс на базі **FastAPI** (Python 3.11) – високопродуктивного фреймворка для REST API. Основні компоненти ядра:

- **FastAPI (async)** – основа веб-сервісу. Дозволяє визначати REST-ендпоїнти, використовувати асинхронні обробники для високої продуктивності та підтримує автоматичну документацію (OpenAPI/Swagger). Асинхронна архітектура гарантує, що ядро може обробляти багато запитів одночасно, не блокуючи події (наприклад, очікування відповіді від БД чи зовнішнього API не "заморожує" інші запити) [5]. Це особливо важливо, якщо клієнти активно звертаються до ядра або якщо інтегруються довгі операції (AI-запити тощо).

- **Pydantic v2** – використовується для визначення схем даних (моделей) та валідації вхідних/вихідних даних. Новітня версія Pydantic дає високу швидкість та зручний синтаксис. Ми створимо Pydantic-моделі для всіх запитів і відповідей API, що дозволить автоматично перевіряти структуру і типи даних. Це забезпечить консистентність між клієнтами: всі вони отримують дані в одному форматі, визначеному схемами. Також Pydantic допоможе валідувати конфігурації, .env змінні тощо.

- **PostgreSQL + SQLAlchemy 2.0** – основна база даних для зберігання постійних даних (профілі користувачів, їх контент, дані NFT/аукціонів, історія взаємодій тощо). Оберемо PostgreSQL за надійність і функціональність. ORM **SQLAlchemy 2.0** використовується для зручної роботи з БД – ця версія повністю підтримує асинхронний двигун та новий синтаксис ORM. Будемо використовувати **async engine** та **async session** SQLAlchemy 2.0, щоб всі операції з БД виконувалися без блокування основного потоку подій. Необхідно спроєктувати моделі БД (таблиці) під основні сутності: *User* (користувач), *Session/Token* (для OAuth2), *Guide/Content* (гайди), *NFT/Auction* (цифрові активи), *Interaction/Message* (лог дій чи історія AI-діалогів) тощо. Введемо міграції (наприклад, через Alembic) для контрольованого оновлення схем БД.

- **Redis** – швидке сервіси кешування та зберігання нетривалих даних. Redis можна використати для:

- *Кешування результатів*: наприклад, часто запитувані дані (останній результат AI-аналізу, статус аукціону) можна тимчасово зберігати, щоб прискорити відповіді.

- *Сесії та стани користувача*: якщо потрібен механізм збереження проміжного стану (наприклад, кроку “AI-флоу” або тимчасових OTP/verification кодів), Redis підійде завдяки швидкодії і підтримці TTL (час життя ключа).

- *Черги завдань/pub-sub*: для реалізації відкладених задач або повідомлень (наприклад, відправити сповіщення на всі клієнти) можна застосувати Redis Pub/Sub чи Stream, якщо виникне потреба.

- **Інтеграції зі сторонніми сервісами:** ядро виконуватиме всі звернення до зовнішніх API та сервісів, аби клієнти були максимально незалежними. Планується підтримка:

- **OpenAI API** – для реалізації AI-функціоналу (генерація тексту, відповіді на питання, AI-діалоги тощо). Наприклад, якщо бот має функцію “AI-чат” або аналіз контенту, Core API матиме модуль інтеграції, що відправляє запит до OpenAI (GPT-4/3.5 чи інші моделі) і отримує результат. Цей модуль буде асинхронним (через `httpx` або вбудований AsyncIO HTTP-клієнт) та може підтримувати потокові відповіді (stream) для довгих AI-відповідей.
- **Vision API / комп’ютерний зір** – для роботи із зображеннями. Якщо бот обробляє картинки (наприклад, розпізнає текст, визначає об’єкти або генерує зображення), ядро матиме відповідний підмодуль. Можливі рішення: використати сторонні API (наприклад, Google Vision, OpenCV на сервері, чи власну модель). В ядрі передбачимо сервіс `VisionService`, який отримує зображення (наприклад, у вигляді URL або переданого файлу), виконує потрібний аналіз/обробку і повертає результат клієнту (текст, посилання на оброблене зображення тощо).
- **Cloudinary** – хмарне сховище медіа. Для зберігання зображень, відео та інших файлів, що завантажують користувачі, інтегруємо Cloudinary (або аналог). Ядро приймає файл від клієнта (через upload-ендпоїнт), завантажує на Cloudinary і зберігає у БД лише посилання на файл та необхідні метадані. Це зніме навантаження з сервера та спростить доставку медіаконтенту клієнтам (Cloudinary може віддавати оптимізовані зображення по URL).

- **E-mail/Notification сервіси (за потреби)** – якщо потрібна email-верифікація чи інші нотифікації поза межами чат-платформ, ядро може містити інтеграцію з SMTP або стороннім сервісом розсилки для відправки листів користувачам (наприклад, підтвердження пошти при реєстрації на веб).

- **OAuth2 авторизація, JWT та refresh tokens** – ядро буде відповідати за автентифікацію користувачів для тих клієнтів, де це потрібно (веб, мобільний). План дій:

- Реалізувати в Core API механізм OAuth2 (наприклад, `OAuth2PasswordBearer` у FastAPI) для класичного логіну по username/email та паролю. При успішному логіні видавати **JWT access token** і **refresh token** (зберігати refresh-токени, наприклад, в БД або Redis).
- Передбачити ендпоїнти реєстрації нового користувача (для веб/мобільних клієнтів) та отримання нового токену по refresh-токену. Всі захищені ендпоїнти Core API будуть вимагати пред’явлення дійсного JWT (в заголовку Authorization).

- **Telegram/Discord:** ці платформи не мають традиційного логіну в нашій системі, там користувач вже “автентифікований” через свій акаунт на платформі. Ми реалізуємо для них **сервісну авторизацію**: бот (Telegram/Discord клієнт) буде звертатися до ядра з **сервісним токеном** або іншим механізмом довіри (наприклад, в заголовку запиту міститиме попередньо узгоджений API-ключ), а також передаватиме у запиті ідентифікатор користувача з тієї платформи. Ядро перевіряє ключ, ідентифікує користувача за platform ID (в БД користувач матиме прив’язку, див. пункт 6) і дозволяє доступ. Таким чином, Telegram/

Discord-клієнти не потребують ручного логіну користувачем, але безпечний доступ до ядра гарантовано.

- **Webhook'и та зворотні виклики** – ядро повинно вміти як приймати, так і ініціювати вебхуки:

- *Вхідні webhooks:* Наприклад, інтеграція з платежами або сторонніми сервісами. Якщо користувач здійснив оплату на сторонньому сервісі, той може викликати наш Core API (спеціальний URL) з повідомленням про успішну транзакцію. Ми спроєктуємо ендпоїнти для отримання таких webhook-викликів і обробки подій (оновлення статусу замовлення, надсилання сповіщень тощо).
- *Вихідні webhooks/callbacks:* Якщо ядро має самостійно повідомити зовнішню систему або навіть клієнтський додаток про подію, можна реалізувати механізм webhook-сповіщень. Приклад: у системі аукціону після завершення торгів ядро може викликати callback-URL, наданий фронтендом, для оновлення в реальному часі. Для чат-ботів це не дуже актуально (вони отримують повідомлення через свої API), але для веб/мобільного можна використовувати WebSocket або Server-Sent Events. В рамках Core API варто закласти можливість розширення реального часу (через WebSocket) для вебклієнта, якщо буде потрібно (FastAPI/Starlette це підтримують).

Архітектура ядра: Структуруємо код за шарами, дотримуючись SRP:
- **Models (Pydantic & SQLAlchemy)**: Опис моделей домену. SQLAlchemy-моделі для БД (таблиці: User, Item, NFT, Bid, Guide і т.п.), Pydantic-схеми для запитів/відповідей (CreateItemRequest, UserResponse тощо). Це забезпечить єдину схему даних для всіх клієнтів.
- **DAO/Repositories:** Класи або функції, які інкапсулюють логіку доступу до БД (через ORM). Вони виконують CRUD-операції, транзакції, але **не містять бізнес-логіки** – тільки зберігання/читання.
- **Services/Business Logic:** Основний шар бізнес-логіки. Тут реалізуються всі правила: що відбувається при тій чи іншій дії. Сервіси оперують моделями, звертаються до DAO, викликають зовнішні API (через інтеграційні модулі). **Важливо:** сервісні функції не знають нічого про Telegram або веб – вони приймають звичайні параметри (наприклад, `user_id`, `item_id`) і повертають результат (об'єкти або дані). Таким чином, цей шар повністю незалежний від способу виклику.
- **API (FastAPI routers):** Тонкий шар, що прив'язує URL-запити до викликів сервісних функцій. Кожен ендпоїнт:
- Отримує HTTP-запит, перевіряє/десеріалізує дані через Pydantic.
- Викликає відповідну функцію сервісного шару, передаючи їй необхідні параметри.
- Отримує від сервісу результат (наприклад, об'єкт або повідомлення) і віддає його клієнту у вигляді HTTP-відповіді (автоматично серіалізується за допомогою Pydantic).
- Реалізує тільки ту логіку, що стосується **транспорту** (HTTP): наприклад, витягання токену користувача із заголовку, встановлення кукі, обробка статус-кодів. Вся бізнес-логіка (перевірка прав, обчислення, формування текстів тощо) вже виконана в сервісах.

- **Конфігурація:** Використовуємо Pydantic для налаштувань (об'єкт Settings), щоб зчитувати конфіг (рядки підключення до БД, ключі API зовнішніх сервісів, секрет JWT і т.д.) з оточення (.env) безпечним способом.

- **Безпека:** Впроваджуємо OAuth2 для веб/моб, як зазначалось, з використанням FastAPI-засобів. Для Telegram/Discord – фільтрація за IP або секретним ключем для сервісних запитів. Всі важливі ендпоїнти перевіряють права користувача (наприклад, чи є користувач власником ресурсу, чи має адмін-права для певних дій).

- **Документація API:** Завдяки FastAPI автоматично отримаємо OpenAPI-документацію. Необхідно описати моделі й ендпоїнти (через docstring/опис параметрів), щоб згенерований Swagger був зрозумілим. Це спростить розробку клієнтів – вони можуть дивитися /docs і бачити, які є методи, які дані очікуються та повертаються.

Реалізація базових можливостей в Core API:
Переконаємося, що Core API охоплює всі функції, які раніше реалізовувалися на стороні Telegram-бота. Для кожної команди чи сценарію, наявного в боті, треба спроєктувати відповідний REST-метод. Наприклад:
- Команда `/start` (або запуск бота новим користувачем) – тепер це виклик, який реєструє користувача в системі. Ендпоїнт `POST /users/start` може створювати запис користувача (якщо ще нема) та повертати базову інформацію (вітальне повідомлення, налаштування).
- Отримання профілю користувача – ендпоїнт `GET /users/{id}` повертає дані профілю (статистика, баланс, тощо) у форматі, готовому для відображення.
- Оновлення профілю – `PUT /users/{id}` дозволяє змінювати налаштування користувача (в тому числі, можливо, прив'язувати соц.акаунти).
- Сценарії "гайдів" – можливо, ендпоїнти на кшталт `GET /guides` (список гайдів), `GET /guides/{id}` (деталі гайду), `POST /guides` (створити новий гайд адміністратором).
- NFT/аукціони – серія ендпоїнтів: `GET /auctions` (список активних аукціонів/NFT), `POST /auctions` (створити аукціон), `POST /auctions/{id}/bid` (зробити ставку), тощо. Логіка визначення переможця, часу завершення – все в ядрі.
- AI-флоу – можливо, ендпоїнт `POST /ai/chat` куди надсилається запит від користувача (його повідомлення), а ядро через OpenAI повертає відповідь. Або `POST /ai/analyze-image` – надсилання зображення для аналізу Vision-модулем.
- Вебхуки – наприклад, `POST /webhook/telegram` (якщо раптом вирішимо приймати оновлення від Telegram напряму), або `POST /webhook/payment` для отримання підтвердження платежу.

Продумуємо **статуси відповіді та помилки**: Core API завжди має повертати клієнтам зрозумілі коди (200 OK, 201 Created, 400 Bad Request з поясненням, 401 Unauthorized, 500 Internal Server Error для неочікуваних проблем і т.д.). Клієнти на основі цього зможуть правильно реагувати (наприклад, якщо 401 – запросити логін, якщо 400 – показати повідомлення про неправильний ввід).

3. Рефакторинг Telegram-бота: винесення логіки в сервіс

Поточний Telegram-бот (Python 3.11, aiogram 3.19) містить усю бізнес-логіку: обробники команд здійснюють усі перевірки, звернення до БД, формування відповідей. Наша задача – **максимально "знежирити" код бота**, перетворивши його з моноліту логіки на тонкий клієнт, що викликає Core API. Ось план рефакторингу крок за кроком:

3.1. Виокремлення модулів логіки:
Проаналізуйте код бота і виділіть основні функціональні блоки: авторизація/реєстрація, управління профілем, обробка конкретних команд (наприклад, команди для отримання гайдів, початку аукціону, виклику AI). Для кожного блоку визначте бізнес-логіку, яку він виконує. *Наприклад:* обробник команди `/profile` наразі сам дістає дані профілю з БД і формує повідомлення. Ця логіка повинна бути перенесена в Core API: створюємо ендпоїнт (наприклад, `GET /users/{telegram_id}`), який при виклику повертає потрібні дані (ім'я, рейтинг, тощо), а форматування повідомлення залишимо клієнту.

3.2. Реалізація логіки в Core API:
Для кожного виділеного модуля бізнес-логіки реалізуємо відповідні функції та ендпоїнти в новому сервісі:
- Переносимо код роботи з базою, перевірки умов, виклики сторонніх сервісів – у функції всередині Core API. Наприклад, усе, що Telegram-бот раніше робив при надходженні команди `/auction_start` (створення запису аукціону, перевірка, чи користувач має право, відповіді з підтвердженням) – тепер опиниться в ендпоїнті `POST /auctions` на боці ядра. Бот замість здійснювати це сам, викличе цей ендпоїнт і отримає результат.
- При перенесенні коду враховуємо асинхронність: у aiogram всі хендлери – async-функції. Виклики до БД (через SQLAlchemy) чи API (OpenAI) в ядрі – також `async/await`. Таким чином, як бот, так і ядро працюватимуть асинхронно і ефективно.

3.3. "Тонкі" Telegram-хендлери:
Після того, як логіка реалізована на стороні Core API, переписуємо обробники aiogram:
- Кожен обробник тепер лише отримує **вхідні дані** від Telegram (текст команди, параметри, ID користувача, вкладені файли) і робить **HTTP-запит** до відповідного ендпоїнту Core API. Для цього в Telegram-боті можна використовувати бібліотеку `aiohttp` або `httpx` для виконання запитів всередині async-хендлерів.
- Формуємо запит: визначаємо URL ендпоїнту, метод (GET/POST/PUT...), передаємо необхідні параметри. Наприклад, для команди `/profile`, хендлер може виконати:

```
response = await httpx.get(f"{CORE_API_URL}/users/{telegram_id}")
data = response.json()
```

(плюс обробка помилок, якщо статус не 200).
- **Отримання відповіді:** на виході Core API поверне структуровані дані (JSON). Бот їх приймає і на основі них формує повідомлення для Telegram. Тут і залишаємо специфічну для Telegram презентацію: наприклад, якщо ядро повернуло `{ "username": "Ivan", "rating": 100 }`, бот може сформувати строку `"📊 *Ваш рейтинг:* 100\n *Ім'я:* Ivan"`. Форматування (емодзі, Markdown розмітка) – це зона відповідальності Telegram-клієнта. Ядро таких деталей не знає, воно могло повернути просто текстові поля або навіть шаблоновані повідомлення.
- **Відправка повідомлення користувачу:** далі бот використовує `message.answer()` або інші методи aiogram, щоб відправити кінцеву відповідь. У випадку команди, що запускає якусь дію (наприклад, генерувати звіт), Core API може повертати повідомлення про успіх, яке бот просто передасть користувачу.

Такий підхід (розділення на "отримати дані" і "відобразити дані") відповідає рекомендаціям щодо розділення логіки: *"основна логіка відокремлена від Telegram-обробників"* [6]. У результаті код обробників стане набагато простішим і меншим: кожен – це по суті виклик до API і обробка відповіді.

3.4. Відв'язування обробників від бізнес-функцій:
Щоб впевнитись, що обробники Telegram не містять бізнес-логіки, слід дотримуватися правила: *"нехай обробник не вирішує* що *робити – він лише вирішує* куди *передати запит"*. Тобто:
- Якщо раніше в хендлері були умовні конструкції, розгалуження бізнес-правил – їх треба перенести в код ядра. У хендлері можна залишити лише просту перевірку, чи є необхідні аргументи, інакше – повідомити про неправильний ввід. Хоча і такі перевірки часто можна перенести на бік сервера, щоб логіка перевірки була єдина.
- **Валідація вводу:** Частину її (наприклад, чи число в діапазоні) можна доручити ядру (через

Pydantic-схеми з валідаторами). Але базові речі – на кшталт "користувач не надав обов'язковий параметр після команди" – можна і на клієнті перевірити, щоб одразу попросити повторити. Це не дублювання бізнес-логіки, а скоріше поліпшення UX конкретної платформи. В ідеалі ж, API повинне відповідати помилкою 400, а бот згенерує з тієї помилки повідомлення.
- **FSM та стан**: Якщо бот використовував **Finite State Machine** (Aiogram FSMContext) для поетапних сценаріїв (наприклад, кілька питань до користувача), треба вирішити, де тримати цей стан. Є два варіанти:
1. *Залишити FSM на боці Telegram:* Тоді бот продовжує керувати діалогом, але на кожному кроці викликає Core API для перевірки/збереження відповідей. Це простіше впровадити, але частково логіка послідовності залишається в коді бота.
2. *Перенести логіку послідовності на Core:* Бот кожен крок просто надсилає введення користувача на сервер, а сервер у відповідь повідомляє, що далі (напр., "ок, чекаю наступне питання" або "дані збережено, дякую"). Це вимагатиме на боці ядра реалізувати механізм сесій діалогу (зберігати в Redis чи БД стадію, на якій знаходиться користувач). Цей підхід усуне будь-яку FSM з клієнта – станами керуватиме сервер.
Обравши підхід, слід рефакторити існуючі сценарії відповідно. Наприклад, якщо був сценарій "створення нового гайду з кількох кроків", можна реалізувати в Core API ендпоїнт, який приймає поточну відповідь і віддає, яке наступне питання ставити. Telegram-бот просто слідує інструкціям.
- **Відокремлення презентаційного шару:** Частини коду, які відповідають чисто за інтерфейс Telegram, можна залишити. Це:
- Формування *Inline Keyboard* або *Reply Keyboard* – ядро може передати список опцій, але саме створення об'єктів `InlineKeyboardMarkup` з кнопками – робить бот. Логіка, які кнопки показати, теж може визначатися ядром (наприклад, повернути тип меню), але побудова – клієнт.
- Особливості форматування: Telegram дозволяє Markdown/HTML, Discord – свої форматування (Markdown, ембед), веб – взагалі HTML. Тому ядро може надсилати або **нейтральні дані** (наприклад, чистий текст без розмітки, а клієнт додає потрібні символи для жирності/курсиву), або універсальні позначки (наприклад, використовувати MarkdownV2 в текстах ядра, що Telegram сприйме, а інші просто відобразять як є – це неідеально). Краще, якщо **ядро надсилає структуровану відповідь**, а кожен клієнт сам вирішує, як її красиво відобразити. Наприклад, ядро повертає JSON: `{ "text": "Ваш рейтинг: 100", "buttons": ["Update", "Main menu"] }`, Telegram-бот перетворює це на повідомлення з кнопками, Discord-бот – на вбудоване повідомлення, веб – на HTML-картку.
- Обробка специфічних подій платформи: наприклад, Telegram надсилає окремо повідомлення про редагування повідомлення, або callback_data від кнопок. Ці події спочатку обробляє бот (отримує update). Потім він може або безпосередньо реагувати (наприклад, закрити клавіатуру), або дерегувати на сервер (наприклад, при натисканні кнопки "Refresh" бот звернеться до `/some-data` і потім оновить повідомлення).

3.5. Тестування після рефакторингу:
Після кожного кроку рефакторингу переконайтеся, що поведінка бота не змінилася з погляду користувача. Підніміть локально Core API і переналаштуйте бота на нього. Протестуйте основні команди: старт, отримання довідки, AI-запити, тощо. Якщо все зроблено правильно, користувач помітить хіба що швидкодію (можливо, незначна затримка через мережевий виклик). У разі проблем – налагоджуйте: дивіться логи ядра (чи доходить запит, чи коректно відпрацьовує), та логи бота (чи правильно він форматує запит і обробляє відповіді).

Переваги рефакторингу: Після завершення, **TelegramBot стане "тонким" клієнтом**, а вся бізнес-логіка буде сконцентрована і легко змінювана в ядрі. Це означає, що додавання, скажімо, нової перевірки чи функції більше не потребуватиме редагування *кожного* клієнта – достатньо внести зміни в Core API. Такий підхід відповідає принципу, що *REST API – це просто ще один шар представлення бізнес-логіки* [3]: якщо завтра замість Telegram з'явиться новий інтерфейс

(наприклад, Slack-бот), ми реалізуємо тільки клієнтський код для Slack, тоді як бізнес-логіка буде повторно використана на 100%. Більше того, ізольована логіка легше тестується і розвивається незалежно від деталей Telegram API.

4. План розробки клієнтів на базі єдиного Core API

Коли ядро створено і Telegram-бот переписано для роботи з ним, наступний крок – **побудувати інші клієнти (Discord, Web, Mobile)**, які аналогічно використовують Core API. Ключова ідея – **усі клієнти викликають одні й ті ж ендпоїнти Core API**, забезпечуючи єдину поведінку. Розробку клієнтів можна вести паралельно (різні команди) або послідовно, реюзуючи напрацювання. Нижче – дорожня карта для кожного з клієнтів:

4.1. Telegram Bot (на aiogram) – оновлена версія

Поточний бот вже частково перероблено на кроці 3. Завдання на цьому етапі: довести клієнт Telegram до чистого presentation-layer та впевнитись, що він повністю використовує Core API.
- **Перевести бота на webhook (опціонально):** Розгляньте можливість замість long polling використовувати вебхуки Telegram. У такому разі бот буде запущено як веб-сервер (можна навіть інтегрувати в FastAPI або окремо через Aiogram) і Telegram надсилатиме апдейти на URL (наприклад, `https://api.example.com/tg/webhook`). Це усуне зайву довготривалу петлю і полегшить масштабування (можна запустити кілька екземплярів бота за балансувальником). Однак webhook потребує HTTPS і доступного ззовні сервера – якщо це складно, залишаємо long polling, але тоді процес бота треба постійно тримати запущеним.
- **Налаштування URL ядра:** Зробити зручним конфігурований URL/host Core API для бота (через .env). У dev-режимі це може бути `http://localhost:8000`, в продакшені – `https://api.myservice.com`.
- **Використання спільних моделей:** Щоб мінімізувати помилки, можна згенерувати або написати спрощені клієнтські моделі відповідей на Python (або просто десеріалізувати JSON у словник). Наприклад, використати OpenAPI-специфікацію ядра, щоб згенерувати Python-клієнт (існують генератори клієнтів по OpenAPI). Це дасть Telegram-боту готові методи для звернення до API. Але це опціонально – можна писати запити вручну.
- **Обробка помилок API:** Додати загальний перехоплювач: якщо Core API відповів не 2xx статусом, бот має коректно це обробити. Наприклад, при 401 – повідомити користувача, що сесія не дійсна (хоча для Telegram це навряд станеться, бо сервісний доступ), при 400 – вивести текст помилки (ядро може повертати `{"detail": "Error message"}`), при 500 – вибачитись і спробувати пізніше. Таким чином UX буде дружнім.

4.2. Discord Bot (на discord.py)

Discord-бот розробляється аналогічно Telegram-боту, але з урахуванням специфіки Discord:
- **Створення Discord-бота:** Реєструємо бота в Discord Developer Portal, отримуємо токен. Налаштовуємо базове з'єднання через бібліотеку `discord.py`. Бот працюватиме як окремий процес, підключений до Discord gateway.
- **Обробка команд і подій:** У Discord є поняття slash-commands (новий інтерфейс команд) та текстових команд. Краще використовувати slash-commands для сучасності. Визначаємо набір команд, аналогічний до Telegram. Наприклад: `/start`, `/profile`, `/guide`, `/auction` і т.д., з відповідними опціями.
- **Виклики Core API:** При виклику команди в Discord, бот так само робить HTTP-запит до Core API. Наприклад, користувач вводить команду `/profile` в Discord – у коді бота пишемо обробник для цієї команди, який викликає `GET /users/{discord_id}` на ядрі. Discord надає нам `user.id`

(числовий ID користувача Discord) – використовуємо його для ідентифікації або (краще) зв'язуємо із записом користувача в нашій БД (див. секцію 6 про ідентифікацію). Можливо, доведеться спочатку зіставити discord_id з нашим user_id (наприклад, зберігати при першій взаємодії).
- **Формування відповіді:** Discord-бот отримує JSON з ядра і перетворює його на повідомлення у чаті. Discord підтримує **Embed**-повідомлення з гарним оформленням: можна використати це для кращої подачі (наприклад, профіль оформити у вигляді embed-картки з полями). Також підтримує і просто текст/Markdown. Необхідно перевести дані у відповідний формат. Знову ж, вся логіка – на боці ядра, бот тільки відображає.
- **Особливості Discord:**
- **Асинхронність:** `discord.py` теж асинхронна, тож можна без проблем використовувати `await httpx.get(...)` всередині обробників.
- **Rate limits:** Discord має обмеження на кількість повідомлень. Якщо ядро може віддати багато контенту, можливо варто розбивати повідомлення або використовувати пагінацію. Це можна врахувати: або ядро одразу підтримує параметри пагінації (`?page=2`), або бот сам розбиває масив даних на кілька повідомлень.
- **Командний інтерфейс:** Після налаштування slash-команд Discord, опис (description) і параметри команд можна зберігати статично або отримувати з ядра (наприклад, ядро може мати ендпоїнт, що описує доступні дії). Але на практиці, slash-команди реєструються через API Discord окремо. Це не дублювання бізнес-логіки, бо опис команди – це скоріше документація для користувача.

- **Тестування Discord-бота:** Перевіряємо основні сценарії так само, як у Telegram. Звертаємо увагу на різницю у контексті: в Discord один бот може бути на кількох серверах, треба врахувати, що `user_id` може повторюватися на різних серверах? (Ні, глобальний Discord ID унікальний). Але можливо, варто врахувати сервер (guild) контекст, якщо, приміром, функціонал має залежність від сервера. Це вже специфіка бізнес-логіки, яка має бути у ядрі (наприклад, аукціон на сервері). В ядро можна передавати і `guild_id` теж, якщо потрібно.

4.3. Веб-клієнт (Next.js або Vue)

Веб-інтерфейс надасть більш багатий досвід користувача для IUI. Обираємо фреймворк (або Next.js+React, або Vue.js – залежно від експертизи команди). План розробки:
- **Визначення стеку:** Наприклад, **Next.js** (React) – для SSR і SPA функціоналу, або **Vue 3** (можливо з Nuxt.js). Обидва підходи дозволяють будувати односторінковий додаток, що спілкується з API.
- **Аутентифікація в веб:** На відміну від ботів, веб-користувач заходить через браузер. Використовуємо OAuth2/JWT з ядра: тобто зробимо сторінки **логін/реєстрація**. Користувач вводить email/пароль (або інший механізм), фронтенд відправляє ці дані на ендпоїнт Core API (`/auth/login`). У разі успіху отримує JWT і refresh-токен. Зберігаємо JWT у **HttpOnly cookie** (безпечніше) або в пам'яті/LocalStorage (спрощено, але менш безпечно). Всі наступні запити з браузера надсилатимуть JWT (як Bearer заголовок або cookie автоматично). Реалізуємо механізм **refresh**: коли токен протухає, автоматично звертаємося за новим (`/auth/refresh`).
- **UI компоненти:** Розробляємо інтерфейс для основних розділів:
- *Профіль користувача:* сторінка з інформацією профілю, налаштуваннями. Запит до `GET /users/me` (використовуючи JWT для аутентифікації) підтягує дані, відображаємо їх. Можливість редагувати -> форма, яка на submit робить `PUT /users/me`.
- *Головна панель / Гайди:* якщо передбачені навчальні матеріали чи новини – отримуємо їх з ядра (`GET /guides`), відображаємо списком.
- *NFT/Аукціони:* список аукціонів (`GET /auctions`), детальна сторінка аукціону (`GET /auctions/{id}`) з можливістю зробити ставку (`POST /auctions/{id}/bid`). Ці виклики йдуть напряму до Core API. Реалізуємо оновлення даних на фронті після дій (ставку додано – оновити інформацію, або використати WebSocket для отримання реального часу).
- *AI-функції:* наприклад, окрема сторінка "AI Assistant", де користувач вводить запит. По submit

робимо запит до `/ai/chat` і відображаємо відповідь моделі. Можна в режимі чату: тоді зберігаємо контекст на фронті або отримуємо ідентифікатор сесії від ядра (краще, якщо ядро веде історію). Також, якщо є комп'ютерний зір – сторінка для завантаження зображення, яке відправляється на `/ai/vision` і виводиться результат (наприклад, текст з картинки).
- **Комунікація з API:** Використовуємо стандартні засоби: `fetch` API, бібліотека **Axios** або інша (в Next.js може підійти `getServerSideProps` для SSR, але в основному тут SPA calls). Формуємо запити згідно документації OpenAPI (URL, метод, тіло, заголовки). Оскільки API уніфіковане, фронтенд-розробникам буде зрозуміло з документації, що викликати.
- **Обробка відповіді:** На фронтенді обов'язково обробляємо помилки (HTTP статуси). Якщо приходить 401 – перекидаємо на логін (або виконуємо refresh token, якщо є). Якщо 403 – показуємо "Недостатньо прав". Якщо 400 – відображаємо повідомлення з detail, яке надішле сервер (можна красиво оформити).
- **Інтерфейс та UX:** Створюємо приємний UI, враховуючи i18n (див. пункт 6 – можливо, різні мови). Важливо зробити навігацію по основним функціям, щоби користувачі веб мали доступ до всіх тих же можливостей, що й користувачі ботів, і навіть більше (наприклад, перегляд списків, графічні елементи).
- **Integration з Telegram/Discord:** Можна додати опцію "зв'язати Telegram-акаунт" у веб-профілі: видавати код для верифікації, який користувач може надіслати боту, щоб лінкувати (про це нижче). Це більше про ідентифікацію між клієнтами.
- **Розгортання веб-клієнта:** Це окремий проект (Next/Vue). Можливо, він буде розгорнутий на Vercel/Netlify чи в тому ж сервері. Якщо SSR, він може виконуватися на Node.js сервері. Але архітектурно його тримаємо окремо від Core API (щоб був незалежним клієнтом).

4.4. Мобільний додаток (React Native або Flutter)

Мобільний клієнт забезпечить присутність IUI на телефонах користувачів з нативним досвідом. Вибираємо технологію: **React Native** (JavaScript, крос-платформова, близька до веб) або **Flutter** (Dart, потужний UI toolkit). План:
- **Створення проекту:** Якщо React Native – ініціалізуємо з `npx react-native init`, якщо Flutter – `flutter create`.
- **Аутентифікація:** Так само, як на веб, користувачі мобільного будуть логінитися. Використовуємо той самий Core API OAuth2. Реалізуємо екран входу, що відправляє запит до `/auth/login` і зберігає токен. На мобільному токен можна зберігати у захищеному сховищі (Keychain для iOS, Keystore для Android) або в SecureStorage (щоб не був видимий, на відміну від AsyncStorage). Refresh-токен так само зберігаємо і оновлюємо по необхідності.
- **UI та навігація:** Розробляємо екрани, аналогічні веб-функціям: Профіль, Список гайдів, Аукціони, AI-чат. Якщо RN, можна використати бібліотеки типу React Navigation для переходів між екранами; для Flutter – Navigator.
- **Виклики API:** У RN можна використовувати `fetch` або бібліотеку Axios. У Flutter – пакет `http` або більш просунутий `dio`. Усі виклики йдуть на ті самі ендпоїнти. Формати JSON такі ж, тому якщо API добре спроєктоване, мобільна розробка зводиться до відмалювання даних.
- **Відображення даних:** Формуємо нативні компоненти: списки (ListView/FlatList в RN, ListView.builder у Flutter), форми, кнопки – і наповнюємо даними з API. Наприклад, екран "Аукціони" робить GET /auctions, отримує JSON масив лотів, відображає їх у списку; при натисканні на лот – переходить на екран деталей, де викликає /auctions/{id} і показує інформацію, дозволяє зробити ставку (пост-запит).
- **Особливості мобільного UX:**
- Мобільний додаток може надсилати **push-нотифікації**. Наприклад, якщо на аукціоні перебили ставку користувача, добре б надіслати пуш. Для цього можна інтегрувати Firebase Cloud Messaging чи аналог. Ядро при певних подіях може через Firebase API відправляти push конкретному токену девайса. Налаштуємо, якщо планується (це не обов'язково на перший етап).

- Офлайновий режим: базово, додаток покладається на онлайн API, але можна кешувати деякі запити локально (SQLite/AsyncStorage) для швидкості та офлайн-доступу до статичної інформації (гайдів, можливо).
- Обновлення контенту в реальному часі: або через пуші, або через періодичні запити. Наприклад, в AI-чаті можна реалізувати web-socket до сервера (якщо зробимо), або опитування раз в n секунд, але краще викликати і чекати відповіді (що і так буде, якщо API повертає одразу).
- **Тестування на пристроях:** Перевірити на Android/iOS емуляторах (або реальних) всі функції. Звернути увагу на авторизацію (чи зберігається токен, чи оновлюється), на роботу з камерою/галереєю якщо потрібен аплоад зображень (наприклад, для Vision функціоналу – RN/Flutter мають окремі плагіни для вибору файлу і відправки multipart на `/ai/vision`).
- **Платформо-специфічні нюанси:** Якщо React Native – код в основному спільний для iOS/Android, але треба налаштувати App Transport Security (для iOS, якщо API не HTTPS) або права. Якщо Flutter – ще легше, один код. Релізні збірки через Xcode/Android Studio.

4.5. Уніфікація досвіду: Всі клієнти повинні надавати користувачу порівняльний набір можливостей. Хоч інтерфейси різні, важливо, щоб, наприклад, в Telegram та веб були доступні одні й ті ж дані (можливо, у текстовому vs графічному вигляді). Завдяки центральному API, додавання нової функції (скажімо, новий тип AI-аналізу) означає: реалізувати ендпоїнт в ядрі, а потім просто викликати його з кожного клієнта, де цей функціонал потрібен. Це синхронізує розвиток платформи. Як було зазначено: *"всі ці клієнти роблять одне й те саме – спілкуються з вашим API"* [4].

5. Стратегія CI/CD, тестування та документації

Для успішного запуску і підтримки нової архітектури потрібно налаштувати **безперервну інтеграцію і деплоймент (CI/CD)**, впровадити різні рівні тестування, а також підготувати вичерпну документацію для розробників і користувачів. Кроки:

5.1. CI (Continuous Integration):
- **Репозиторій коду:** Рекомендується зберігати код Core API та клієнтів у окремих репозиторіях (для чіткого поділу) або в монорепозиторії з відповідними папками (спрощує синхронізацію версій). Наприклад, один репо для `core-api`, інший для `telegram-bot`, `discord-bot`, `web-client`, `mobile-app`.
- **Налаштування автоматичних збірок:** Використовуємо інструменти CI, такі як GitHub Actions, GitLab CI, Jenkins чи інші, щоб при кожному пуші/мержі коду виконувалися скрипти:
- Запуск юніт-тестів (про них нижче).
- Перевірка якості коду: лінтери (`flake8`, `black` для Python; ESLint/Prettier для JS; `dart analyze` для Flutter).
- Збірка артефактів: для ядра можна збирати Docker-образ, для фронтенда – виконати build (наприклад, Next.js build генерує production bundle), для мобільного – CI може збирати .apk/.ipa (опціонально, можна вручну для релізу).
- **Контейнеризація:** Створюємо Dockerfile для Core API (на базі python:3.11-slim, встановлюємо залежності, запускаємо uvicorn сервер). Те ж саме для ботів (можна їх теж контейнеризувати). Це дозволить легко розгортати на сервері або в Kubernetes.
- **Аналіз коду і безпеки:** Інтегрувати в CI інструменти типу SonarQube або Snyk для статичного аналізу і перевірки відомих вразливостей залежностей. На кожному пуші корисно бачити, чи не додалося критичних уразливостей.

5.2. CD (Continuous Deployment/Delivery):
- **Середовища:** Налаштувати щонайменше два середовища – *Staging* (тестове) і *Production*. Staging

– це копія продакшен оточення, де можна перевірити нові версії без впливу на реальних користувачів.
- **Автоматичний деплой:** Наприклад, при мержі в гілку `main` (чи `release`) CI/CD пайплайн збирає Docker-образи і деплоїть їх:
- Якщо використовується Kubernetes – робить kubectl apply нових образів (можна з Rolling Update).
- Якщо простіше – на виділеному сервері з Docker Compose оновлює контейнер до нової версії.
- Важливо налаштувати безперервність: нова версія ядра сумісна зі старими клієнтами (тому краще, щоб усі клієнти оновлювалися разом із API, або API лишався backward-compatible).
- **Версіонування API:** Для безпеки, можна ввести версію API (v1, v2) у шляхах ендпоїнтів або в хедерах. Це на випадок майбутніх змін, щоб старі клієнти могли ще деякий час працювати. На етапі v4 можна позначити всі URL як `/api/v4/...`.
- **Розгортання клієнтів:**
- Telegram/Discord ботів – також деплоїмо, можливо, на той самий сервер або окремі. Важливо, щоб боти були перезапущені з новим кодом після оновлення ядра (особливо якщо змінилось щось у форматах).
- Веб-клієнт – деплой можна автоматизувати на Vercel/Netlify чи власний сервер nginx. Якщо Next.js SSR – запускаємо сервер через PM2 або Docker.
- Мобільний – це окрема історія: деплой на користувацькі пристрої йде через App Store / Google Play (а це вже не CD в класичному розумінні, а вручну випуск релізів). Можна, однак, автоматизувати збірку і розміщення в TestFlight / Google Play Internal Testing для QA.

5.3. Тестування (різнорівневе):
Розробляємо **стратегію тестування**, яка охоплює всі ключові частини системи:
- **Юніт-тести (Unit tests):** Пишемо тести для дрібних модулів, особливо бізнес-логіки ядра:
- Тестуємо окремо функції сервісного шару: наприклад, функція розрахунку результатів аукціону, функція валідації введених даних, функція взаємодії з OpenAI (можна замокати зовнішній API).
- Для ботів: можна протестувати утилітарні функції (якщо такі є) або прості перетворення. Основна логіка ботів тепер дуже проста, тож тут тестів може бути мінімум (можливо, тести на формування повідомлень: передаємо штучний JSON – перевіряємо, що бот формує потрібний текст).
- Використовуємо Pytest для Python-коду, jest/mocha для JS при потребі, etc.
- **Інтеграційні тести:** Перевіряємо взаємодію компонентів разом:
- **API + БД:** Піднімаємо тестовий інстанс PostgreSQL (можна через Docker в CI або використати SQLite в пам'яті, якщо підтримує) та Redis. Запускаємо тестовий сервер FastAPI (можливо з допомогою `TestClient` FastAPI) і проганяємо сценарії: наприклад, **тест реєстрації та логіну** (POST /auth/register, потім /auth/login – очікуємо токен), **тест створення та отримання ресурсу** (POST /auctions, GET /auctions – перевіряємо дані). Важливо перевірити, що бізнес-правила виконуються (не можна зробити ставку більшу за баланс – API повинно повернути 400). Ці тести дають впевненість, що ядро працює як задумано.
- **API + сторонні сервіси:** Можна написати тести, які мокають виклики до OpenAI чи Cloudinary. Або, у разі дозволу мережі, викликати справжні API (але це не завжди добре для CI через ключі). Ліпше замокати (наприклад, за допомогою бібліотеки responses або httpx.MockTransport) відповіді від OpenAI, щоб перевірити, що наш код правильно обробляє їх.
- **Боти + API (інтеграція):** Написати невеликі скрипти, що імітують поведінку ботів проти тестового API. Або підняти локально core-api, запустити бота з змінною середовища, що він вказує на локальний API, і через pytest відправити йому "штучне" повідомлення (aiogram дозволяє диспатчити Update об'єкти вручну). Перевірити, чи він звернувся до API (можливо, замоніторити лог чи використати мок). Це складніше, але можна реалізувати.
- **Web + API:** За допомогою фреймворків типу Cypress або Playwright можна написати e2e-тести для веб: наприклад, запустити локально (або в CI з xvfb) нашу веб-програму, автоматично увійти (використовуючи тестового користувача на тестовому API) і пройти сценарії – перевірити, що відображаються дані з API. Cypress може перехоплювати HTTP-запити і перевіряти їх, або просто

працювати як користувач і перевіряти текст на сторінці.
- **E2E (End-to-End) тести:** Це повні сценарії від імені користувача, які проходять через всі шари. Деякі з них можна автоматизувати, деякі доведеться проводити вручну:
- **Manual тестування ботів:** Мати тестового користувача в Telegram і Discord, запускати бота на staging-середовищі Core API. Проходити всі команди, перевіряти відповіді. Переконатися, що нові функції (NFT, аукціони, AI) працюють в чаті не гірше, ніж у веб.
- **Manual тестування веб та мобільного:** Протестувати, що весь основний шлях працює: реєстрація -> логін -> перегляд/створення контенту -> логіка з AI -> логіка з NFT -> вихід.
- **Автоматизовані E2E:** Крім Cypress для веб, можна розглянути **Appium** для мобільного (автоматизація UI на емуляторі). Але це досить ресурсно затратно, можна реалізувати базові тести (наприклад, запуск додатку, логін, перевірка екрану профілю).

5.4. Документація:
Документування проекту має декілька аспектів:
- **Автодокументація API (OpenAPI):** Як згадано, FastAPI генерує OpenAPI schema. Необхідно її підтримувати – додавати описи моделей, параметрів, приклади відповідей. На основі цієї схеми можна згенерувати документацію для зовнішнього використання або навіть клієнтські SDK. Власне, розглянемо генерацію API-клієнтів: існують інструменти, що беруть OpenAPI і генерують клієнтський код для різних мов [7] [8]. Можна згенерувати, наприклад, TypeScript-клієнт для веб (хоча Axios використати теж ок), або Python-клієнт для ботів. Це **не обов'язково**, але корисно.
- **README та репозиторна документація:** У головному репозиторії(Core API) створюємо детальний README.md, де описана архітектура, як запустити проект локально (інструкція з встановлення залежностей, запуск БД/Redis, запуск сервера). Також описуємо основні компоненти. Для клієнтських репозиторіїв – свої README з інструкціями запуску (наприклад, як запустити бота локально, які змінні потрібні).
- **Dev Quickstart:** Створимо документ "Developer Quickstart" – по суті, покрокова інструкція розгортання всього середовища:
1. Клонувати репозиторій(ї).
2. Встановити Docker та docker-compose (або python+node, залежно від підходу).
3. Запустити команду `docker-compose up` (ми можемо підготувати docker-compose.yaml, що піднімає postgres, redis, core-api, і, можливо, локально telegram-bot з webhook-режимом для тесту).
4. Вказати, як протестувати: наприклад, "відкрийте браузер на http://localhost:8000/docs щоб побачити API docs" або "надішліть команду /start у Telegram боту @YourTestBot".
Цей гайд допоможе новому розробнику чи контриб'ютору швидко підняти систему і побачити її в дії.
- **Документація для користувачів/адмінів:** Окремо, за потреби, створюємо **користувацькі гіди** – як користуватися ботом, вебом, з картинками і поясненнями. Це не про розробку, але якщо проект публічний, корисно мати.
- **Коментарі і стиль коду:** Дотримуємося чистого коду, додаємо коментарі там, де логіка неочевидна. Використовуємо Google Docstring або Sphinx-стиль, щоб при генерації документації (наприклад, через Sphinx) було зрозуміло, що робить той чи інший модуль.

Після завершення, нова команда розробників повинна мати все необхідне: репозиторії з інструкціями, автотести що перевіряють регресії, та живу документацію API для інтеграції.

6. Масштабування, розширення функціоналу та інші перспективи

При проєктуванні IUI v4 необхідно врахувати майбутнє зростання проекту: як в плані кількості користувачів, так і появи нових функцій. Нижче описано стратегії для забезпечення

масштабованості, **легкого додавання фіч**, **незалежності клієнтів**, а також підтримки **інтернаціоналізації (i18n)** та **єдиної системи ідентифікації користувачів** між платформами.

6.1. Горизонтальне масштабування та продуктивність:

- **Статус ядра:** Завдяки асинхронності та відмові від збереження стану в пам'яті, Core API буде практично **stateless** (не враховуючи кеш/БД). Це дозволяє легко масштабувати його горизонтально – запустити декілька екземплярів за балансувальником (наприклад, Kubernetes deployment з кількома репліками). Всі вони будуть обробляти запити, використовуючи спільну БД та Redis. Якщо навантаження виросте (більше користувачів, більше запитів), достатньо підняти більше контейнерів ядра.
- **Масштабування бази даних:** PostgreSQL можна масштабувати через реплікацію (читачі-репліки для розподілу read-навантаження) або вертикально (потужніший інстанс). На рівні коду важливо оптимізувати запити, мати необхідні індекси. Використовувати профайли запитів, щоб уникнути повільних операцій.
- **Кешування:** Вже згаданий Redis кеш дасть приріст при масштабуванні – наприклад, закешувати відповіді на популярні запити (топ-рейтинги, список останніх гайдів) з невеликим TTL, щоб при пікових навантаженнях не завантажувати БД.
- **Балансування навантаження:** Якщо клієнти (особливо веб) розподілені глобально, розглянути CDN для статики вебсайту, гео-рознесені сервера для API. Telegram та Discord трафік – глобальний, але наш сервер можна тримати в одному регіоні, це не критично.
- **Моніторинг і логування:** Для підтримки продуктивності впровадити логування (через структуровані логи або сервіси типу Sentry для винятків, Prometheus + Grafana для метрик). Моніторити кількість запитів, час відповіді, навантаження на CPU/RAM – щоб вчасно додати ресурси.
- **Оптимізація AI-викликів:** Якщо функціонал AI (OpenAI API) буде інтенсивно використовуватись, врахуйте їх ліміти і час: можливо, додати **чергу для AI-завдань**. Наприклад, якщо багато користувачів генерують зображення або текст, ставити задачі в чергу (RabbitMQ/Redis) і обробляти робітниками послідовно, щоб не перевищувати rate limit і не тримати HTTP-з'єднання відкритим. У такому разі можна організувати нотифікацію клієнта про готовність (в веб – через WebSocket/push, в ботах – бот просто надішле відповідь як тільки отримає результат через окремий потік). Але якщо навантаження помірне, можна і прямо чекати відповіді від OpenAI в запиті.

6.2. Розширення функціоналу (гайди, NFT, аукціони, AI-флоу):

Архітектура повинна бути **модульною**, щоби нові можливості додавалися без "ламаючих" змін.

- **"Гайди" (довідники/статті):** Ця функція може бути реалізована як окремий модуль ядра. Наприклад, модуль **GuideModule** з власними моделями (Guide, GuideCategory), своїми ендпоїнтами. При розробці модулю, дотримуємося тих самих принципів – логіка тільки на боці сервера. Клієнти просто відображають контент гайду. Якщо потрібно редагування/створення (для адміністраторів) – робимо через ті ж API. Цей модуль не залежить від інших частин, крім як загальної системи автентифікації (тобто можна легко змінювати чи видаляти його).
- **NFT/Аукціон:** Враховуючи специфіку, це, по суті, цілий піддомейн. Але спроєктуємо його в рамках моноліту ядра так, щоб був готовий до винесення. Наприклад, пакет `app/auction` зі своїми схемами, сервісами і роутерами. Якщо навантаження чи складність виростуть, цей код можна виділити в мікросервіс (окремий сервіс AuctionService, з власною БД, а Core API виступатиме gateway). Але на початку простіше тримати разом. Головне – ізолювати логіку.
- **Інтеграція з блокчейном:** Якщо NFT передбачає взаємодію з Ethereum/Solana тощо, потрібно інтегрувати відповідні SDK. Це зовнішня залежність, яка може бути ресурсомісткою. Можливо, її варто виконувати окремо (окрема черга транзакцій). Але знову ж, за модульності це не завадить іншим частинам.
- **Аукціон реального часу:** Питання: чи потрібні оновлення в реальному часі про перебиття

ставки? Якщо так, для веб варто мати WebSocket канал. Можна реалізувати WS-сервер у Core API (FastAPI дозволяє) або окремий microservice. Telegram/Discord не підтримують persistent connection від нас, але можна емулювати: бот може отримувати оновлення через webhook від ядра. Наприклад, ядро при новій ставці може дернути бот API метод відправки повідомлення всім, хто торгується. Це можливо, але тоді ядро має знати про Telegram (що ми хотіли уникати). Краще – клієнти самі періодично опитують API або, якщо користувач активно торгується – натискає "оновити" кнопку.

- **AI-флоу:** Можливі майбутні ускладнення AI-сценаріїв (наприклад, ланцюжки запитів, рекомендації). Тримаємо AI-логіку теж окремо. Зараз це просто функція в Core, але якщо захочемо додати, наприклад, свою ML-модель, можна підняти окремий сервіс (наприклад, модель на TensorFlow Serving) і звертатися з Core API до нього. Важливо, що API ядра для клієнтів залишиться тим же (`/ai/...`), просто під капотом поміняється реалізація. Така **замінність реалізації** – частина принципу Open/Closed: зовнішній контракт стабільний, реалізація розширюється.

6.3. Незалежність оболонок (клієнтів):

Хоча всі клієнти використовують спільний API, вони повинні бути максимально ізольовані один від одного на випадок збоїв чи змін:

- **Відмовостійкість клієнтів:** Якщо, скажімо, Discord-бот зіткнувся з помилкою і впав, це не повинно вплинути на роботу Telegram-бота чи веб – і навпаки. Завдяки тому, що вони окремі процеси, це і так виконується. Треба лише слідкувати, щоб один клієнт не перевантажив ядро. Якщо, наприклад, баг у Telegram-боті спричиняє шторм запитів на API, може постраждати сервіс для всіх. Щоб цього уникнути, можна впровадити **rate limiting** на рівні ядра (наприклад, не більше X запитів від одного IP/токена в секунду) або окремо для кожного бота.
- **Сумісність з новими платформами:** Якщо завтра вирішимо додати нового клієнта (Slack, Viber, CLI-інтерфейс), ми повинні лише написати новий фронт, використовуючи існуючі ендпоїнти. Переконуємося, що API має все необхідне і не залежить від специфіки старих клієнтів. Наприклад, не варто в назвах чи структурах відповідей використовувати терміни, специфічні для Telegram. Все має бути уніфіковано (лаконічні назви полів, нейтральні формати).
- **Версії клієнтів:** У мобільному додатку, користувачі можуть не оновитись одразу. Тому, якщо ми змінюємо API, старі версії апки можуть звертатися до нього. Треба або підтримувати сумісність, або мати версіонування API. На веб це менш проблема (користувач завжди отримує останню версію при оновленні сторінки), а от старі версії Telegram/Discord ботів можна примусово оновити з сервера devops-методами. Все ж, варто документувати зміни API чітко і, якщо можливо, не ламати його без необхідності.

6.4. Інтернаціоналізація (i18n):

Проект IUI має аудиторію, що, ймовірно, розмовляє різними мовами (наприклад, українська, англійська). Потрібно забезпечити багатомовність відповідей. План впровадження i18n:

- **Локалізація контенту ядра:** Весь текст, який генерує Core API (помилки, повідомлення, відповіді AI/гайди якщо перекладати) має підтримувати різні мови. FastAPI не має вбудованої i18n, але ми можемо використати стандартні підходи Python: бібліотеку **Babel** або **gettext** для перекладу рядків. Практика така: позначаємо рядки через функцію `_('text')` і маємо .po файли перекладів. Або зберігаємо переклади в JSON і робимо свій простий механізм.
- **Визначення мови користувача:**
- Для веб/моб – користувач може вибрати мову в налаштуваннях або автоматично по браузеру. Цю інформацію включаємо в запити (наприклад, заголовок `Accept-Language` або параметр ?lang=ua).
- Для Telegram – бот API дає поле `from.language_code` (яке містить мову інтерфейсу користувача Telegram). Можна використовувати його як початкову підказку. Дискорд – мовних налаштувань користувача API не дає, тож можемо за замовчуванням англійську, або мати команду /language.

- Зробимо в нашій БД поле `user.preferred_language`, яке оновлюється або вручну (якщо користувач вибрав), або ставиться при першій взаємодії з Telegram (якщо `language_code` є).
- **Middleware для мови:** На стороні FastAPI зробимо middleware, що визначає мову для кожного запиту і зберігає її в `request.state` або робить `gettext.translation(...).install()` перед виконанням ендпоїнту [9]. Джерела: заголовок, параметр або профіль користувача (якщо є аутентифікація, можемо взяти з БД). Наприклад, якщо автентифікований користувач – беремо його `preferred_language`; якщо ні – дивимось заголовок `Accept-Language`.
- **Переклад відповідей:** При формуванні повідомлень ядро вибирає відповідну локаль. Якщо використали gettext, то всі фрази вже перекладуться автоматично за вибраною локаллю. Якщо ж JSON словники, то треба вручну підставляти. Наприклад, при формуванні помилки:

```
if user_not_found:
    return {"detail": translations[lang]["USER_NOT_FOUND_ERROR"]}
```

Це трохи verbosive. Можливо, комбінація: статичні повідомлення через gettext, а динамічні дані підставляти форматуванням.
- **Локалізація клієнтів:** Самі клієнти теж потребують перекладу, але це окремо в UI (не стосується ядра). В веб – використати i18n фреймворк (react-i18next або Vue i18n). В моб – аналогічно (для RN – i18next, для Flutter – Flutter Intl). Боти: де вони ще формують текст (в ідеалі, мало де), можуть також звертатися до ядра за локалізованими повідомленнями. Наприклад, якщо клавіатура з кнопками – текст кнопок теж краще локалізувати. Можна це зробити на стороні сервера: сервер може передати текст кнопки вже в потрібній мові. Або зробити шаблони в боті для декількох мов, але тоді десь потрібно знати мову користувача на боці бота (можна зберігати). Правильніше – бот запитує сервер з вказанням мови, сервер дає потрібні тексти.
- **Додавання нових мов:** Організувати процес, щоби легко додати переклад. Наприклад, всі повідомлення зібрані в один файл, який віддаємо перекладачу. По отриманні перекладу, додаємо нову локаль, і все – сервер почне її віддавати при `Accept-Language`.

6.5. Єдина система сесій та ідентифікації користувачів між клієнтами:
Щоб користувач міг безперешкодно перемикатися між платформами (наприклад, почав діалог з ботом у Telegram, а продовжив на веб) і щоб система розуміла, що це той самий користувач, потрібно впровадити механізм зв'язування облікових записів та спільної ідентифікації:
- **Уніфікація користувача:** Зробимо в базі таблицю Users з унікальним внутрішнім `user_id`. Додамо колонки для кожної платформи: `telegram_id`, `discord_id`, можливо `email` (для веб), та ін. Коли користувач вперше взаємодіє через Telegram, ми створюємо нового User з заповненим telegram_id. Через Discord – аналогічно, окремий запис з discord_id. Через веб – реєстрація створює запис (email + пароль, telegram_id/discord_id порожні).
- **Лінкування акаунтів:** Забезпечуємо можливість об'єднати записи. Наприклад, користувач почав з Telegram (є запис з telegram_id=12345). Потім він заходить на веб і хоче мати там же дані. Він реєструється на веб (новий запис email=foo@bar.com). Тепер у нас дві сутності для однієї людини. Потрібно їх змерджити або прив'язати. Як варіант:
- **Через код підтвердження:** На веб у профілі показуємо "Прив'язати Telegram". При натисканні генеруємо одноразовий код (зберігаємо у Redis, прив'язаний до web-акаунта) і даємо інструкцію "відправте цей код нашому Telegram-боту". Користувач в Telegram вводить команду /link ABC123. Бот надсилає цей код на Core API (з інформацією, що від цього telegram_id прийшло). Ядро знаходить сесію з таким кодом, бере відповідний user_id веб і виконує об'єднання: дописує в цей запис `telegram_id=12345` (тобто прив'язує телеграм до веб-акаунта). Можливо, видаляє старий окремий запис, або помічає як дублікат. Аналогічно можна і Discord прив'язати (код або URL).
- **Через OAuth Telegram:** Telegram пропонує **Telegram Login Widget** для веб, що дозволяє зайти на сайт через свій Telegram (https://core.telegram.org/widgets/login). Це теж варіант: користувач на

веб натискає “Login with Telegram”, відбувається верифікація через Telegram API, і веб отримує його telegram_id та username. Тут можна одразу знайти користувача з таким telegram_id і прив’язати/логінити. Але це працює якщо користувач хоче зв’язати, а якщо він вже зареєстрований, то цей процес треба акуратно обробити (щоб дубліката не створити).
- **Через зовнішні OAuth:** Можна дозволити входити через Google/Facebook, але це вже інша площина – не напряму про Telegram/Discord, хоча теж розширює ідентифікацію.

- **Спільні сесії:** Коли акаунти прив’язані, цікаво реалізувати безшовний досвід: наприклад, після того як користувач прив’язав Telegram, він міг би отримувати повідомлення на Telegram про важливі події з веб (але це треба надсилати від бота). Також, чи можна продовжити розмову AI між платформами? Теоретично так: якщо зберігати історію в БД з прив’язкою до user_id, то користувач, зайшовши на веб, може побачити історію питань, які він задавав боту. Це додатковий плюс єдиного акаунта. Тому радимо зберігати історію AI-чатів/запитів із полем user_id – тоді інтерфейси можуть відображати це.
- **Single Sign-On між платформами:** Якщо у майбутньому буде власний OAuth-сервер, можна реалізувати SSO: наприклад, користувач авторизувавшись на веб, автоматично має токен, який дійсний і для інших фронтів. Але в випадку ботів – вони не вміють приймати наш JWT. Тому SSO в класичному сенсі тільки між веб і, скажімо, окремими веб-додатками. Наразі достатньо просто *об’єднання акаунтів*, без спільної сесії.
- **Зберігання сесій (JWT) між клієнтами:** JWT-токен, виданий веб-користувачу, не має сенсу використовувати у боті, бо бот не питає користувача токен. Навпаки, бот діє під довіреним сервісним обліком. Однак, якщо, наприклад, з веб користувач вмикає налаштування “надсилати мені сповіщення в Telegram”, то при настанні події ядро може ініціювати повідомлення через Telegram-бота цьому користувачу (знаючи telegram_id). Це можна зробити, і це не вимагає авторизації, бо ми самі викликаємо Telegram API. Але потрібно зберегти токен бота і chat_id. Оскільки у нас вже є прив’язка telegram_id до user_id, цього досить (токен бота глобальний, а telegram_id відомий).

6.6. Довгострокова підтримка та гнучкість:
- **Логування та аудит:** З часом, для розширення, можна додати аудит-лог – записувати ключові дії користувачів (хто що зробив, коли) в окрему таблицю. Це допоможе відслідковувати шахрайство (особливо якщо фінанси в аукціонах).
- **Безпека:** Переглядати безпекові аспекти: захист від SQL-ін’єкцій (ORM це дає), від XSS/CSRF на веб (використати HttpOnly cookies і CSRF tokens при необхідності), rate limiting, капча на реєстрації, тощо. В контексті Telegram/Discord, можливо, антиспам механізми (не давати одному юзеру спамити запитами).
- **Мікросервіси при необхідності:** Закладена модульність дозволяє у майбутньому відділити деякі частини. Наприклад, якщо AI-функціонал стане дуже важким, можна винести його в окремий сервіс (наприклад, на іншому сервері з GPU). Core API тоді буде викликати його по внутрішньому API. Для клієнтів це прозоро. Те ж стосується NFT: окремий сервіс міг би опрацьовувати взаємодію з блокчейном. Наша архітектура монолітна, але правильно розбита на “контексти”, тож вона **Ready for Microservices**, коли постане така вимога.
- **Розгортання нових клієнтів:** Нові платформи (скажімо, голосовий асистент, або AR/VR клієнт) легко підключити – достатньо, щоб вони використовували HTTP API. Документація OpenAPI і, можливо, згенеровані SDK допоможуть їм швидко почати.
- **Відгуки та аналітика:** Можна вбудувати збір подій (event tracking) в ядро: кожен ендпоїнт логуватиме анонімізовану інформацію (який функцією користуються найбільше, середній час відповіді). Ці дані допоможуть фокусувати розвиток (наприклад, якщо AI-флоу дуже популярний – приділити йому більше уваги).

Насамкінець, запропонована архітектура IUI v4 з єдиним Core API орієнтована на **чистий код, повторне використання, масштабованість і зручність розвитку**. Дотримання принципів DRY/SRP гарантує, що кожна частина системи відповідає за своє: **ядро** – "мозок" проекту, **клієнти** – "обличчя" для користувача. Це суттєво спростить підтримку проекту і відкриє можливості для розвитку нових напрямків (NFT, AI, гейміфікація тощо) без хаотичного нагромадження коду. З таким підходом IUI зможе еволюціонувати й масштабуватися, зберігаючи цілісність та стабільність на кожному етапі. [6] [3]

---

[1] [2] Striking the Balance with the DRY Principle in Software Development | by Muhammad Ur Rehman | Medium
https://medium.com/@muhammad-ur-rehman/striking-the-balance-with-the-dry-principle-in-software-development-8dc9fa0cfea2

[3] architecture - Considerations for decoupling and refactoring business logic to a REST API - Software Engineering Stack Exchange
https://softwareengineering.stackexchange.com/questions/319420/considerations-for-decoupling-and-refactoring-business-logic-to-a-rest-api

[4] [7] [8] Everything you need to know about OpenAPI and API client generation · Sander ten Brinke
https://stenbrinke.nl/blog/openapi-api-client-generation/

[5] Setting up a FastAPI App with Async SQLALchemy 2.0 & Pydantic V2 | by Thomas Aitken | Medium
https://medium.com/@tclaitken/setting-up-a-fastapi-app-with-async-sqlalchemy-2-0-pydantic-v2-e6c540be4308

[6] How I saved Business logic from Telegram integration - DEV Community
https://dev.to/egorblagov/how-i-saved-business-logic-from-telegram-integration-5al1

[9] Simple way to make i18n support with FastAPI - DEV Community
https://dev.to/whchi/simple-way-to-make-i18n-support-in-fastapi-27kd