НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Навчально-науковий інститут атомної та теплової енергетики

Кафедра інженерії програмного забезпечення в енергетиці

ДО ЗАХИСТУ ДОПУЩЕНО
В.о. завідувача кафедри
_____Олександр КОВАЛЬ
«_____»_____2025р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем і веб-технологій»
спеціальності 121 Інженерія програмного забезпечення
на тему: «Розробка програмного забезпечення для децентралізованого розподілу енергоресурсів між споживачами на базі технології блокчейн»

Виконав:
студент IV курсу, групи ТВ-12
Іщенко Ілля Володимирович _____
(підпис)

Керівник:
ст. викл. Сарибога Г.В. _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент:
ст. викладач, к.т.н.
Наливайчук Микола Васильович _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2025

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем і веб-технологій»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Олександр КОВАЛЬ
(підпис)
«____»_____202_р.

ЗАВДАННЯ
на дипломну роботу студенту

_____Іщенко Іллі Володимировичу_____
(прізвище, ім'я, по батькові)

1. Тема роботи:
«Розробка програмного забезпечення для децентралізованого розподілу енергоресурсів між споживачами на базі технології блокчейн»_____
керівник роботи Сарибога Ганна Володимирівна, старший викладач_____
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від "__"_____ 2025 року №____
2. Строк подання студентом роботи _____
3. Вихідні дані до роботи: мова програмування Python, бібліотека Web3, середовище розробки PyCharm_
4. Зміст (дипломної роботи) пояснювальної записки (перелік завдань, які потрібно розробити): провести аналіз існуючих пристроїв для граничних обчислень; провести опис поставленої задачі; створити власний датасет та розробити власну модель машинного навчання; інтегрувати модель на пристрій для граничних обчислень _____
5. Перелік ілюстративного матеріалу: аналогічні системи, використані інструменти, діаграми прецедентів, аналіз результатів, тестування роботи моделі, впровадження моделі на мікрокомп'ютер.
6. Дата видачі завдання «__» _____ 202_ р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Строки виконання етапів роботи	Примітка
1	Отримання завдання	30.10.2024	Виконано
2	Дослідження предметної області	31.10.2024 - 31.12.2024	Виконано
3	Дослідження існуючих рішень	01.01.2025 – 15.02.2025	Виконано
4	Постановка вимог до проектування системи	16.02.2025 - 14.04.2025	Виконано
5	Розробка програмного продукту	15.04.2025 - 12.05.2025	Виконано
6	Тестування	12.05.2025 - 19.05.2025	Виконано
7	Захист програмного продукту	13.05.2025	Виконано
8	Оформлення дипломної роботи	20.05.2025 – 31.05.2025	Виконано
9	Передзахист	02.06.2025	Виконано
10	Захист	17.06.2025	

Студент _____ Іщенко І.В.
(підпис) (ім'я, прізвище)

Керівник роботи _____ Ганна Сарибога
(підпис) (ім'я, прізвище)

РЕФЕРАТ

Структура та обсяг дипломної роботи. Робота містить 93 сторінок, 4 рисунків та 20 посилань.

Метою роботи є розробка програмного прототипу платформи для децентралізованої P2P-торгівлі енергоресурсами на базі технології блокчейн, спрямованої на підвищення прозорості, безпеки та ефективності взаємодій між споживачами та прозьюмерами енергії.

Програмний комплекс повинен забезпечувати створення та управління пропозиціями енергії, здійснення угод купівлі-продажу через смарт-контракт, взаємодію користувачів через веб-інтерфейс з підключенням Web3-гаманців та відображення історії операцій.

Для досягнення поставленої мети виконано такі завдання:

- Проаналізовано проблеми традиційних енергоринків та перспективи P2P-модолей на блокчейні;
- Спроектовано гібридну архітектуру платформи (клієнт, сервер, смарт-контракт, слухач подій, off-chain БД);
- Розроблено єдиний смарт-контракт на Solidity для управління on-chain операціями;
- Реалізовано API на Python (FastAPI) та клієнтський інтерфейс на React (Ethers. js);

Практичне значення одержаних результатів полягає у створенні функціонального прототипу, що демонструє технічну реалізацію децентралізованої торгівлі енергією. Система може слугувати основою для подальших розробок та освітніх цілей. Набуто досвіду у розробці dApps, актуального для сучасної інженерії ПЗ.

Ключові слова: P2P торгівля енергією, блокчейн, смарт-контракт, Ethereum, Solidity, FastAPI, React, Ethers.js, dApps.

ABSTRACT

Structure and scope of the thesis. The work contains 93 pages, 4 illustrations and 20 references.

The **aim** of the work is to develop a software prototype of a platform for decentralized P2P energy resource trading based on blockchain technology, aimed at increasing transparency, security, and efficiency of interactions between energy consumers and prosumers.

The **software complex should** enable the creation and management of energy offers, the execution of purchase and sale agreements via a smart contract, user interaction through a web interface with Web3 wallet integration, and the display of transaction history.

To achieve this goal, the following tasks were performed:

– Analyzed the problems of traditional energy markets and the prospects of P2P models on the blockchain;

– Designed a hybrid platform architecture (client, server, smart contract, event listener, off-chain database);

– Developed a unified smart contract in Solidity for managing on-chain operations;

– Implemented an API in Python (FastAPI) and a client interface in React (Ethers.js).

The **practical significance of the obtained results** lies in the creation of a functional prototype that demonstrates the technical implementation of decentralized energy trading. The system can serve as a basis for further development and educational purposes. Experience in dApp development, relevant to modern software engineering, has been gained.

Keywords: P2P energy trading, blockchain, smart contract, Ethereum, Solidity, FastAPI, React, Ethers.js, dApps.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API (Application Programming Interface) – інтерфейс прикладного програмування

ABI (Application Binary Interface) – двійковий інтерфейс додатків (смарт-контрактів)

ASGI (Asynchronous Server Gateway Interface) – асинхронний шлюзовий інтерфейс сервера

БД – база даних

ГЕС – гідроелектростанція

ДТЕ (Decentralized Energy Trading) – децентралізована торгівля енергією (може використовуватися як префікс у повідомленнях смарт-контракту, наприклад "DET: Price below minimum")

ДО (Децентралізована Організація) - може використовуватися як варіант для **DAO**

DAO (Decentralized Autonomous Organization) – децентралізована автономна організація

DOM (Document Object Model) – об'єктна модель документа

DSO (Distribution System Operator) – оператор системи розподілу

dApp (decentralized Application) – децентралізований додаток

ETH – Ефір (криптовалюта мережі Ethereum)

EVM (Ethereum Virtual Machine) – віртуальна машина Ethereum

ES6+ (ECMAScript 6 та новіші версії) – стандарт мови JavaScript

HTTP (HyperText Transfer Protocol) – протокол передачі гіпертексту

HTTPS (HyperText Transfer Protocol Secure) – захищений протокол передачі гіпертексту

IDE (Integrated Development Environment) – інтегроване середовище розробки

IoU (Intersection over Union) – перетин над об'єднанням (метрика в комп'ютерному зорі, якщо ви її десь згадували в інших контекстах раніше)

IoT (Internet of Things) – Інтернет речей

IPFS (InterPlanetary File System) – міжпланетна файлова система (якщо б використовували для зберігання даних)

JSON (JavaScript Object Notation) – текстовий формат обміну даними, заснований на JavaScript

JWT (JSON Web Token) – веб-токен JSON (якщо б використовували для API автентифікації)

KPI (Key Performance Indicator) – ключовий показник ефективності

L2 (Layer 2) – рішення другого рівня (для масштабування блокчейнів)

МО (Машинне Навчання) – якщо б ця технологія згадувалась

МВт·год – Мегават-година

кВт·год – Кіловат-година

Вт·год – Ват-година

P2P (Peer-to-Peer) – від рівного до рівного, піринговий

RPC (Remote Procedure Call) – віддалений виклик процедур

REST (Representational State Transfer) – передача репрезентативного стану (архітектурний стиль для API)

SPA (Single Page Application) – односторінковий додаток

SQL (Structured Query Language) – мова структурованих запитів

UI (User Interface) – користувацький інтерфейс

UX (User Experience) – досвід користувача

URL (Uniform Resource Locator) – уніфікований локатор ресурсу

Web3 – загальний термін для технологій децентралізованого інтернету, часто асоціюється з Ethereum

WS (WebSocket) – протокол зв'язку WebSocket

XSS (Cross-Site Scripting) – міжсайтовий скриптинг

CSRF (Cross-Site Request Forgery) – міжсайтова підробка запиту

Wei – найменша одиниця криптовалюти Ether

ВСТУП

Тема моєї дипломної роботи – "Розробка програмного забезпечення для децентралізованого розподілу енергоресурсів між споживачами на базі технології блокчейн". Можливо, звучить дещо складно, але якщо по-простому, то я займався створенням системи, де звичайні люди або невеликі підприємства, які виробляють енергію (наприклад, за допомогою сонячних панелей), могли б продавати її надлишки напряму іншим споживачам, минаючи великих посередників. А для того, щоб все це було чесно, прозоро та надійно, використовується сучасна технологія блокчейн.

Актуальність цієї теми сьогодні важко переоцінити. Ми всі бачимо, як змінюється світ, зокрема й енергетика. Старі підходи, коли декілька великих електростанцій забезпечують всіх, вже не завжди ефективні. По-перше, це часто не дуже екологічно, бо використовуються викопні види палива. По-друге, при передачі енергії на великі відстані трапляються значні втрати. По-третє, ми, як споживачі, маємо мало впливу на те, звідки береться наша енергія та скільки вона коштує. Тому дедалі більше уваги приділяється розвитку так званої розподіленої генерації, коли енергія виробляється близько до місця її споживання, і часто з відновлюваних джерел.

І ось тут виникає ідея Peer-to-Peer (P2P) торгівлі, тобто "від рівного до рівного". Якщо у мене є сонячні панелі, і вони виробили більше енергії, ніж мені потрібно, чому б мені не продати цей надлишок сусідові за справедливою ціною? Це вигідно і мені, і сусідові, і для загальної енергосистеми, бо зменшується навантаження на великі мережі.

Але для такої прямої торгівлі потрібна надійна система, яка б гарантувала чесність угод, безпеку розрахунків та прозорість усіх операцій. І саме тут на допомогу приходить технологія блокчейн. Якщо дуже спрощено, то блокчейн – це така собі цифрова книга обліку, яку неможливо підробити, і копії якої є у всіх учасників. А спеціальні програми, що працюють на блокчейні, – смарт-контракти

– можуть автоматично виконувати умови домовленостей. Тобто, якщо покупець заплатив, а продавець надав енергію, смарт-контракт сам проведе розрахунок. Це усуває потребу в дорогих посередниках та підвищує довіру.

Тому метою моєї дипломної роботи була розробка програмного прототипу такої платформи для децентралізованої торгівлі енергією. Я хотів дослідити, як саме можна застосувати блокчейн та смарт-контракти для створення функціонального та зручного інструменту, який би дозволив користувачам ефективно обмінюватися енергоресурсами.

Для досягнення цієї мети я поставив перед собою наступні завдання:

По-перше, детально проаналізувати існуючі проблеми в енергетиці та вивчити теоретичні основи P2P торгівлі й застосування блокчейну в цій сфері.

По-друге, спроектувати архітектуру майбутньої платформи, визначивши її основні компоненти: смарт-контракт, серверна частина та клієнтський інтерфейс.

По-третє, безпосередньо розробити ці компоненти: написати смарт-контракт мовою Solidity, створити API на Python та FastAPI, а також розробити веб-інтерфейс на React.

І, нарешті, провести тестування розробленого прототипу, щоб переконатися в його працездатності.

Об'єктом мого дослідження є сам процес децентралізованої торгівлі енергією між кінцевими споживачами та прозьюмерами. А предметом дослідження – розроблена мною програмна платформа на основі технології блокчейн, що реалізує цей процес.

Щодо наукової новизни та практичного значення, то хоча ідея P2P торгівлі енергією на блокчейні вже досліджується, моя робота полягає у розробці конкретного програмного рішення з певною архітектурою та набором функцій, що може слугувати практичним прикладом реалізації такої системи. Було розроблено єдиний гібридний смарт-контракт, що оптимізує управління основними операціями. Практичне значення полягає у створенні прототипу, який демонструє життєздатність концепції та може бути використаний для подальших

досліджень, експериментів або навіть як основа для більш складних комерційних продуктів у майбутньому. Крім того, ця робота дозволила мені отримати цінний досвід у розробці децентралізованих додатків.

У наступних розділах я детальніше розповім про теоретичні аспекти, процес проектування та розробки кожного компонента платформи, використані інструменти, а також про результати тестування та зроблені висновки.

1 АНАЛІЗ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

Перш ніж приступати до розробки будь-якої системи, важливо чітко розуміти, які саме проблеми вона покликана вирішувати та які завдання стоять перед розробником. Тому цей розділ я присвятив аналізу сучасного стану енергетичної галузі, виявленню її "вузьких місць", а також дослідженню перспективних напрямків розвитку, таких як децентралізована енергетика та технологія блокчейн. Це дозволило мені сформулювати мету та основні завдання моєї дипломної роботи.

1.1 Сучасний стан та проблеми традиційних енергетичних ринків

Коли ми говоримо про традиційну енергетику, то найчастіше уявляємо собі великі електростанції – теплові, атомні, гідроелектростанції – які виробляють електроенергію, а потім вона довгими лініями електропередач надходить до наших домівок та підприємств. Така система, безумовно, виконувала і продовжує виконувати важливу функцію, але з часом стає все очевидніше, що вона має низку серйозних недоліків.

По-перше, це екологічний аспект. Більшість традиційних електростанцій працюють на викопному паливі (вугілля, газ, мазут), спалювання якого призводить до значних викидів парникових газів, що є однією з головних причин зміни клімату. До того ж, видобуток та транспортування цих ресурсів також завдають шкоди довкіллю. Хоча атомна енергетика не продукує $CO2$ безпосередньо, проблема утилізації радіоактивних відходів та ризики аварій залишаються дуже гострими. Навіть великі гідроелектростанції, які вважаються більш "чистими", можуть негативно впливати на екосистеми річок та призводити до затоплення значних територій.

По-друге, це питання ефективності та втрат. Енергія, вироблена на великій станції, часто долає сотні, а то й тисячі кілометрів, перш ніж потрапити до кінцевого споживача. На цьому шляху в лініях електропередач та трансформаторах відбуваються неминучі втрати, які можуть сягати значних відсотків. Чим довша відстань, тим більші втрати. До того ж, самі генеруючі потужності не завжди працюють з максимальною ефективністю, особливо якщо йдеться про застаріле обладнання.

По-третє, централізована система є досить вразливою. Пошкодження однієї великої електростанції або магістральної лінії електропередач може призвести до масштабних відключень електроенергії (блекаутів) на великих територіях. Ми це, на жаль, добре відчули на власному досвіді. Відсутність диверсифікації джерел та шляхів постачання робить систему менш стійкою до різноманітних загроз, чи то природні катаклізми, чи техногенні аварії, чи цілеспрямовані атаки.

По-четверте, обмежена гнучкість та адаптивність. Традиційні енергосистеми досить інертні. Їм складно швидко реагувати на зміни у споживанні або на появу великої кількості малих, розподілених джерел енергії, таких як приватні сонячні панелі. Інтеграція таких джерел у централізовану мережу є непростим технічним завданням, оскільки їхня генерація часто є нестабільною (залежить від погоди) і потребує складних механізмів балансування.

І нарешті, недостатня прозорість та обмежена роль споживача. У традиційній моделі споживач найчастіше є пасивним гравцем. Він отримує рахунки за електроенергію, але має мало можливостей впливати на те, звідки ця енергія береться, як формується ціна, і чи можна якось оптимізувати свої витрати, крім простого зменшення споживання. Ринки часто монополізовані або контролюються невеликою кількістю великих гравців, що не завжди сприяє здоровій конкуренції та зниженню цін.

Всі ці проблеми вказують на те, що енергетична галузь потребує змін, пошуку нових, більш ефективних, гнучких та стійких моделей.

1.2 Перспективи децентралізованої енергетики та концепція Peer-to-Peer (P2P) торгівлі

Одним із найперспективніших напрямків розвитку сучасної енергетики є децентралізація. Суть її полягає у переході від великих централізованих джерел генерації до численних, часто невеликих, розподілених джерел, розташованих ближче до кінцевих споживачів. Це можуть бути сонячні панелі на дахах приватних будинків та комерційних будівель, малі вітрові установки, біогазові станції на фермах, домашні системи накопичення енергії (акумулятори) тощо. Таких активних споживачів, які не тільки споживають, але й виробляють електроенергію, називають "прозьюмерами" (від англ. producer + consumer).

Саме поява великої кількості прозьюмерів створює передумови для розвитку так званої Peer-to-Peer (P2P) торгівлі енергією. Якщо по-простому, то P2P торгівля – це можливість для прозьюмерів продавати надлишки виробленої ними енергії безпосередньо іншим споживачам у своїй локальній мережі чи громаді, минаючи традиційних енергопостачальників та великі мережі. Наприклад, якщо вдень мої сонячні панелі виробили більше енергії, ніж я спожив, я можу продати цей надлишок своєму сусідові, якому в цей час енергії не вистачає.

Такий підхід має чимало переваг. По-перше, це економічна вигода для всіх учасників. Прозьюмери отримують додатковий дохід, а споживачі можуть купувати енергію за потенційно нижчою та більш гнучкою ціною. Це також стимулює встановлення нових приватних генеруючих потужностей, особливо на основі відновлюваних джерел. По-друге, підвищується ефективність використання енергії. Локальне виробництво та споживання значно зменшують втрати при транспортуванні. По-третє, зростає надійність енергопостачання на місцевому рівні. Локальні енергетичні спільноти, так звані мікрогріди (microgrids), можуть навіть працювати в автономному режимі, якщо виникають проблеми в централізованій мережі. По-четверте, споживачі стають активними

учасниками ринку, отримуючи більше контролю над своїм енергозабезпеченням та можливість вибору джерела енергії. Це сприяє більш свідомому споживанню та розвитку "зеленої" енергетики "знизу-вгору".

Звісно, для реалізації P2P торгівлі потрібна відповідна інфраструктура: інтелектуальні лічильники, системи моніторингу та управління, а найголовніше – надійна та прозора платформа для здійснення самих транзакцій купівлі-продажу.

1.3 Роль технології блокчейн у трансформації енергетичного сектору

Коли мова заходить про створення надійної та прозорої платформи для прямих взаєморозрахунків між багатьма учасниками, які не завжди довіряють один одному або центральному посереднику, на думку одразу спадає технологія блокчейн. Спочатку розроблена для функціонування криптовалют, таких як Bitcoin, сьогодні блокчейн розглядається як універсальна технологія для побудови децентралізованих систем у найрізноманітніших галузях, і енергетика не є винятком.

Що ж такого особливого в блокчейні, що робить його привабливим для P2P торгівлі енергією? Якщо спробувати пояснити простими словами, то блокчейн – це розподілена база даних, або цифровий реєстр, який одночасно зберігається на комп'ютерах багатьох учасників мережі. Кожна нова транзакція (наприклад, угода про купівлю-продаж енергії) групується з іншими транзакціями у "блок", який потім додається до "ланцюжка" попередніх блоків. Звідси й назва – "ланцюжок блоків".

Ключові властивості блокчейну, що мають значення для нашої теми:

- Децентралізація**:** Немає єдиного сервера чи органу, який би контролював усі дані. Це робить систему стійкою до відмов та атак на одну точку.
- Незмінність**:** Після того, як блок з транзакціями додано до ланцюжка, його практично неможливо змінити або видалити. Це забезпечується складними

криптографічними алгоритмами. Отже, історія всіх угод є надійно захищеною від підробки.

- Прозорість**:** Учасники мережі (або принаймні авторизовані її члени, залежно від типу блокчейну) можуть бачити всі транзакції, що відбулися. Це створює атмосферу довіри та відкритості.
- Безпека**:** Використання криптографії для підтвердження транзакцій (цифрові підписи) та захисту даних робить систему дуже безпечною.

Але найцікавіше для автоматизації процесів – це смарт-контракти. Смарт-контракт – це, по суті, комп'ютерна програма, яка зберігається та виконується безпосередньо в блокчейні. Вона автоматично виконує закладені в неї умови угоди. Наприклад, у контексті торгівлі енергією смарт-контракт може автоматично перевірити, чи є у продавця заявлений обсяг енергії, чи є у покупця кошти, і якщо обидві умови виконані – зафіксувати угоду та ініціювати розрахунок. Це дозволяє обійтися без традиційних посередників (банків, енергопостачальних компаній у ролі гарантів угоди), що здешевлює та прискорює процес.

Таким чином, блокчейн та смарт-контракти можуть стати технологічною основою для створення ефективних, прозорих та безпечних платформ P2P торгівлі енергією, де кожен учасник може бути впевнений у чесності правил та надійності виконання угод.

1.4 Мета та завдання дослідження

Враховуючи вищевикладені проблеми традиційної енергетики та перспективність децентралізованих підходів на базі технології блокчейн, метою моєї дипломної роботи стала розробка програмного прототипу платформи для децентралізованої торгівлі енергоресурсами між споживачами. Головний акцент робився на створенні системи, яка б дозволяла користувачам (прозьюмерам та звичайним споживачам) безпосередньо взаємодіяти, виставляти пропозиції на

продаж або купівлю енергії та укладати угоди в прозорий та автоматизований спосіб за допомогою єдиного комплексного смарт-контракту.

Для досягнення цієї мети я визначив для себе такі основні завдання:

1. Проаналізувати предметну область**:** Глибше дослідити існуючі моделі P2P торгівлі енергією, вивчити специфіку застосування технології блокчейн та смарт-контрактів саме в енергетичному секторі. Також важливо було ознайомитися з аналогічними програмними рішеннями та проектами, щоб зрозуміти їхні переваги, недоліки та врахувати цей досвід.

2. Визначити вимоги до платформи**:** Сформулювати чіткі функціональні (що система повинна робити) та нефункціональні (якими характеристиками вона повинна володіти, наприклад, щодо безпеки, продуктивності) вимоги до розроблюваного програмного продукту.

3. Спроектувати архітектуру платформи**:** Розробити загальну архітектуру системи, визначити її ключові компоненти – блокчейн-частину (смарт-контракт), серверну логіку (API) та клієнтський інтерфейс – та описати принципи їхньої взаємодії. Особливу увагу приділити проектуванню єдиного смарт-контракту, який би охоплював основний функціонал.

4. Розробити програмні компоненти**:**

- Написати код єдиного гібридного смарт-контракту мовою Solidity, який би реалізовував логіку реєстрації пропозицій, укладання угод купівлі-продажу енергії та фіксації транзакцій.
- Створити серверну частину (API) на мові Python з використанням фреймворку FastAPI. Цей API мав би забезпечувати взаємодію між клієнтським додатком та блокчейном, а також, за потреби, з базою даних для зберігання off-chain інформації (наприклад, історії транзакцій для швидкого доступу).
- Розробити користувацький веб-інтерфейс на React, який би дозволяв учасникам реєструватися (підключати гаманець), переглядати доступні пропозиції, створювати власні, укладати угоди та переглядати історію своїх операцій.

5. Забезпечити інтеграцію та провести тестування**:** Здійснити інтеграцію всіх розроблених компонентів в єдину систему та провести комплексне тестування її працездатності, перевіряючи основні користувацькі сценарії та коректність роботи смарт-контракту.

Вирішення цих завдань дозволило б мені створити робочий прототип платформи та на практиці дослідити можливості застосування обраних технологій.

1.5 Об’єкт та предмет дослідження

Щоб чітко окреслити рамки моєї роботи, важливо визначити її об'єкт та предмет.

Об'єктом дослідження в даній дипломній роботі виступають процеси, пов'язані з децентралізованим розподілом та торгівлею енергоресурсами, зокрема електроенергією, між кінцевими споживачами та прозьюмерами на основі моделі Peer-to-Peer (P2P). Це включає вивчення механізмів взаємодії учасників такого ринку, способів формування пропозицій, укладання угод, а також обліку та проведення розрахунків в умовах відсутності традиційного централізованого посередника.

Предметом дослідження є безпосередньо розроблена програмна платформа для P2P торгівлі енергією, яка функціонує на базі технології блокчейн та використовує єдиний гібридний смарт-контракт для управління основними операціями. Моє дослідження охоплює архітектурні рішення, покладені в основу платформи, принципи її роботи, логіку та структуру смарт-контракту, методи взаємодії між її компонентами (такими як клієнтський інтерфейс, серверна частина (API), блокчейн-мережа, слухач подій та база даних для off-chain зберігання), а також вибір та застосування інструментальних засобів для її розробки, тестування та потенційного розгортання.

1.6 Наукова новизна та практичне значення отриманих результатів

Хоча тема децентралізованої торгівлі енергією на блокчейні не є абсолютно новою і вже існує низка досліджень та пілотних проектів, дана дипломна робота має певні елементи новизни та, безумовно, практичне значення.

Щодо наукової новизни (або, точніше, інженерно-технічної новизни для дипломного проекту бакалавра/спеціаліста), можна відзначити розробку та реалізацію специфічної архітектури платформи, яка поєднує в собі переваги єдиного комплексного смарт-контракту для on-chain логіки з ефективністю off-chain зберігання даних (історії транзакцій) для швидкого доступу через API. Було спроектовано та реалізовано конкретну структуру такого гібридного смарт-контракту мовою Solidity, що управляє основними етапами P2P-угоди: від створення пропозиції до фіксації купівлі. Також елементом новизни є практична інтеграція всіх компонентів системи – смарт-контракту, Python-бекенду на FastAPI, React-фронтенду та асинхронного слухача подій – в єдиний програмний комплекс.

Практичне значення отриманих результатів полягає в наступному:

1. Створено функціональний прототип**:** Розроблено програмний прототип платформи, який на практиці демонструє можливість реалізації децентралізованої торгівлі енергією. Це не просто теоретичне дослідження, а працююча система (в тестовому середовищі), що дозволяє перевірити основні концепції.

2. Набуто цінного досвіду**:** В процесі роботи я отримав глибокі практичні навички у проектуванні та розробці децентралізованих додатків (dApps), включаючи написання та тестування смарт-контрактів на Solidity, розробку API для взаємодії з блокчейном, створення користувацьких інтерфейсів, що інтегруються з Web3-гаманцями, та налаштування взаємодії між on-chain та off-chain компонентами.

3. Демонстрація переваг технології**:** Розроблений прототип наочно показує, як технологія блокчейн може забезпечити прозорість, безпеку та автоматизацію угод на локальних енергетичних ринках, зменшуючи залежність від централізованих посередників.

4. Основа для подальших розробок**:** Створений прототип може слугувати хорошою відправною точкою для подальших, більш масштабних досліджень та розробок. Його можна розширювати, додавати новий функціонал, інтегрувати з реальними пристроями обліку енергії та адаптувати для конкретних потреб локальних енергетичних спільнот.

5. Освітній аспект**:** Результати роботи та отриманий досвід можуть бути корисними для інших студентів та розробників, які цікавляться темою блокчейну в енергетиці.

Висновки до розділу 1

Отже, проведений у даному розділі аналіз виявив суттєві недоліки існуючих централізованих енергетичних ринків та обґрунтував необхідність переходу до більш гнучких, ефективних та прозорих моделей. Децентралізована енергетика та концепція Peer-to-Peer торгівлі, підкріплені можливостями технології блокчейн, визначені як перспективний напрямок вирішення цих проблем. На основі цього аналізу були чітко сформульовані мета та завдання дипломного дослідження, спрямовані на розробку програмного прототипу платформи для децентралізованої торгівлі енергією, а також визначено його об'єкт, предмет, наукова новизна та практична цінність.

2 ОГЛЯД ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

Після того, як ми визначили основні проблеми традиційної енергетики та сформулювали завдання для нашої майбутньої платформи, логічним кроком є глибше занурення в предметну область. У цьому розділі я спробував розібратися, як саме можуть функціонувати децентралізовані енергетичні ринки, які існують архітектурні підходи до створення систем P2P торгівлі енергією, які типи блокчейн-платформ найкраще підходять для таких завдань, та як саме смарт-контракти допомагають автоматизувати процеси. Також важливо було проаналізувати вже існуючі проекти та платформи, щоб зрозуміти, що вже зроблено, які є успішні рішення, а які проблеми ще залишаються актуальними. Це допомогло мені уникнути "винаходу велосипеда" та врахувати досвід інших розробників при проектуванні власної системи.

2.1 Архітектури систем для P2P торгівлі енергією

Коли ми говоримо про створення програмної системи для P2P торгівлі енергією, то не існує якогось єдиного "правильного" архітектурного рішення. Вибір архітектури залежить від багатьох факторів: масштабу системи, кількості учасників, вимог до швидкості транзакцій, рівня децентралізації, регуляторних особливостей тощо. Проте, можна виділити декілька загальних підходів та компонентів, які часто зустрічаються в таких системах.

Часто системи P2P торгівлі енергією будуються за багаторівневою архітектурою. На нижньому рівні знаходиться фізична інфраструктура, що включає розподілені джерела генерації (сонячні панелі, вітряки), системи накопичення енергії, інтелектуальні прилади обліку (smart meters), які здатні фіксувати обсяги генерації та споживання енергії в реальному часі та передавати ці дані.

Наступний рівень – це комунікаційний рівень, що забезпечує обмін даними між фізичними пристроями, користувачами та платформою. Тут можуть використовуватися різні технології, від Інтернету речей (IoT) для збору даних з лічильників та управління пристроями до мобільних та веб-інтерфейсів для взаємодії з користувачами.

Ключовим є рівень платформи, де реалізується логіка P2P торгівлі. Тут можна виділити декілька архітектурних моделей:

- Повністю централізована платформа**:** У цьому випадку існує центральний сервер (оператор платформи), який збирає дані про пропозиції та попит, проводить матчинг заявок, здійснює розрахунки та веде облік. Блокчейн може використовуватися лише як допоміжний інструмент, наприклад, для підвищення прозорості деяких операцій або для розрахунків у криптовалюті. Така архітектура простіша в реалізації, але зберігає ризики, пов'язані з наявністю єдиної точки відмови та контролю.
- Частково децентралізована (гібридна) платформа**:** Це, мабуть, найпоширеніший підхід на сьогодні. У такій моделі блокчейн та смарт-контракти використовуються для виконання ключових функцій: фіксації прав власності на енергію, реєстрації пропозицій, автоматичного укладання угод та проведення розрахунків. Водночас, централізовані компоненти (сервери) можуть відповідати за управління користувацькими інтерфейсами, зберігання некритичних даних (off-chain), кешування інформації з блокчейну для швидкого доступу, аналітику та взаємодію з зовнішніми системами. Саме такий гібридний підхід був обраний і в моїй роботі, оскільки він дозволяє поєднати переваги блокчейну з ефективністю традиційних технологій.
- Повністю децентралізована платформа (DAO-подібні структури): У цій моделі робиться спроба максимально децентралізувати всі аспекти функціонування платформи, включаючи управління, прийняття рішень та навіть розробку, передавши ці функції спільноті учасників через механізми децентралізованих автономних організацій (DAO). Це найбільш складний в

реалізації підхід, який ще знаходиться на ранніх стадіях дослідження та впровадження в енергетиці.

Незалежно від обраної моделі, архітектура повинна забезпечувати взаємодію між виробниками енергії, споживачами, самою платформою та, можливо, з операторами розподільчих мереж (DSO), які відповідають за фізичну доставку енергії та стабільність локальної мережі.

2.2 Типи блокчейн-платформ для енергетичних застосунків (публічні, приватні, консорціумні)

Вибір конкретної блокчейн-платформи є одним із ключових рішень при розробці децентралізованої системи для торгівлі енергією, оскільки від цього залежать такі важливі характеристики, як швидкість транзакцій, їхня вартість, рівень безпеки, масштабованість та модель управління. Існує три основних типи блокчейн-платформ:

1. Публічні блокчейни (Public Blockchains): Це повністю децентралізовані мережі, до яких може приєднатися будь-хто. Прикладами є Bitcoin та Ethereum.

- *Переваги:* Високий рівень децентралізації, прозорості та стійкості до цензури. Велика кількість розробників та розвинена екосистема інструментів (особливо для Ethereum).
- *Недоліки:* Часто мають обмежену пропускну здатність (кількість транзакцій за секунду) та високі транзакційні витрати (особливо в періоди пікового навантаження на мережу Ethereum, хоча рішення другого рівня, такі як Polygon, Arbitrum, Optimism, намагаються це виправити). Питання конфіденційності даних також може бути викликом, оскільки всі транзакції є публічними.
- *Застосування в енергетиці:* Часто використовуються для пілотних проектів, випуску енергетичних токенів або сертифікатів "зеленої" енергії, де

важлива максимальна прозорість. Для мого прототипу я орієнтувався на Ethereum-сумісне середовище, оскільки воно є найбільш поширеним для розробки смарт-контрактів.

2. Приватні блокчейни (Private Blockchains): Це закриті мережі, де доступ та право на участь контролюються однією організацією. Всі учасники відомі та авторизовані.

- Переваги: Висока швидкість транзакцій та пропускна здатність, низькі транзакційні витрати (оскільки немає потреби в складних механізмах консенсусу, як Proof-of-Work). Можливість налаштування рівня конфіденційності даних.
- Недоліки: Низький рівень децентралізації, оскільки контроль знаходиться в руках однієї організації. Це може викликати питання щодо довіри та стійкості до зловживань з боку оператора мережі.
- Застосування в енергетиці: Можуть використовуватися всередині однієї енергетичної компанії для оптимізації внутрішніх процесів, управління активами або для створення локальних ринків під повним контролем оператора.

3. Консорціумні блокчейни (Consortium Blockchains / Federated Blockchains): Це проміжний варіант між публічними та приватними блокчейнами. Мережа контролюється групою попередньо визначених організацій (консорціумом), а не однією компанією чи повністю відкрита для всіх.

- Переваги: Кращий баланс між децентралізацією та продуктивністю. Транзакції швидші та дешевші, ніж у публічних блокчейнах, але система більш децентралізована та надійна, ніж приватні блокчейни, оскільки контроль розподілений між кількома учасниками консорціуму. Дозволяють реалізувати гнучкі моделі управління та доступу до даних.
- Недоліки: Створення та управління консорціумом може бути складним організаційним завданням. Потенційно менш стійкі до змови між учасниками консорціуму порівняно з публічними блокчейнами.
- Застосування в енергетиці: Вважаються дуже перспективними для створення міжорганізаційних платформ, наприклад, для торгівлі енергією між

кількома енергетичними компаніями, великими споживачами та операторами мереж, або для регіональних енергетичних ринків. Прикладом технології, що часто використовується для побудови консорціумних блокчейнів, є Hyperledger Fabric.

Для мого дипломного проекту, який є прототипом, я зосередився на розробці смарт-контракту для Ethereum-сумісної мережі. Це дозволило використовувати велику кількість доступних інструментів розробки, документації та тестових середовищ (таких як Ganache або Hardhat). У разі реального впровадження, вибір між публічним Ethereum (з можливим використанням рішень другого рівня), консорціумним або навіть приватним блокчейном залежатиме від конкретних бізнес-вимог, регуляторного середовища та масштабу проекту.

2.3 Механізми смарт-контрактів в управлінні енергетичними транзакціями

Як я вже зазначав, смарт-контракти є ключовим елементом, що дозволяє автоматизувати та забезпечити надійність процесів на блокчейн-платформі для торгівлі енергією. У моїй роботі було розроблено єдиний комплексний смарт-контракт, який інкапсулює основну логіку. Давайте розглянемо, які саме механізми він використовує для управління енергетичними транзакціями.

По-перше, це управління учасниками та їхніми пропозиціями (оферами). Смарт-контракт дозволяє користувачам, які бажають продати надлишки енергії, реєструвати свої пропозиції. Це реалізовано через функцію setOffer у моєму смарт-контракті. Коли користувач викликає цю функцію, він вказує кількість енергії, яку він пропонує (_availableEnergyWh), та ціну за одиницю енергії (_priceWeiPerWh). Смарт-контракт перевіряє коректність цих даних (наприклад, чи ціна не нижча за мінімально встановлену адміністратором) і зберігає цю інформацію у своїй внутрішній структурі даних (mapping(address => EnergyOffer) public offers). Важливо, що кожен офер прив'язаний до конкретної Ethereum-

адреси продавця (msg.sender). Також контракт відстежує, чи є пропозиція активною (isActive), та оновлює загальну статистику ринку (загальна кількість запропонованої енергії totalEnergyOfferedWh та чисельник для розрахунку середньозваженої ціни totalWeightedPriceNumerator). Якщо користувач хоче змінити або скасувати свою пропозицію, він може викликати ту ж функцію setOffer з новими параметрами або функцію deactivateOffer.

По-друге, смарт-контракт реалізує механізм укладання угод. Коли інший користувач (покупець) знаходить на платформі прийнятну для себе пропозицію, він може її акцептувати. У моєму смарт-контракті це робиться за допомогою функції buyFromProvider. Покупець вказує адресу продавця (_provider) та кількість енергії, яку він хоче купити (_energyToBuyWh). Разом із викликом цієї функції покупець надсилає на адресу смарт-контракту відповідну кількість коштів (в ETH або іншій криптовалюті, якщо це передбачено) для оплати енергії (параметр payable та msg.value). Смарт-контракт виконує низку перевірок: чи активна пропозиція продавця, чи достатньо у нього доступної енергії, чи надіслав покупець достатньо коштів. Якщо всі умови виконані, смарт-контракт:

- Зменшує кількість доступної енергії у продавця.
- Якщо вся енергія продавця була розкуплена, його пропозиція може бути автоматично деактивована.
- Оновлює загальну ринкову статистику.
- Здійснює переказ коштів від покупця (які тимчасово утримувалися контрактом або були надіслані разом із транзакцією) на адресу продавця. Це одна з ключових переваг смарт-контрактів – автоматизований та гарантований розрахунок.
- Фіксує факт угоди, генеруючи подію EnergyBought, яка містить всю інформацію про транзакцію: хто продавець, хто покупець, обсяг, ціна, загальна вартість, час.

По-третє, забезпечення прозорості та аудиту. Всі вищезгадані операції (створення оферу, укладання угоди) є транзакціями, які записуються в блокчейн і

стають незмінними. Будь-хто (або авторизований учасник) може перевірити історію цих транзакцій та переконатися в коректності роботи системи. Події, що генеруються смарт-контрактом, такі як OfferSet, OfferDeactivated, EnergyBought, слугують важливим джерелом інформації для зовнішніх систем, наприклад, для мого скрипта event_listener.py, який збирає дані для off-chain бази історії.

По-четверте, смарт-контракт може містити адміністративні функції, наприклад, setAdmin для зміни адміністратора контракту або setMinPrice для встановлення мінімально допустимої ціни на енергію. Доступ до таких функцій зазвичай обмежується за допомогою модифікаторів (у моєму випадку onlyAdmin), щоб їх міг викликати лише власник контракту.

Важливою особливістю є використання у моєму смарт-контракті ReentrancyGuard від OpenZeppelin для захисту від атак повторного входу, а також EnumerableSet для ефективного управління списком активних провайдерів. Всі функції, що змінюють стан контракту, позначені як nonReentrant.

Таким чином, смарт-контракт виступає як децентралізований та автоматизований арбітр і виконавець угод, що значно підвищує ефективність та надійність P2P торгівлі енергією.

2.4 Аналіз існуючих платформ та проектів децентралізованої торгівлі енергією

Як я вже трохи згадував у попередньому підрозділі, ідея використання блокчейну для торгівлі енергією не є чимось абсолютно новим, і у світі вже існує низка цікавих проектів та платформ, які намагаються реалізувати цю концепцію. Аналіз їхнього досвіду є дуже корисним, оскільки дозволяє зрозуміти, які підходи працюють, які є виклики та які уроки можна винести для власної розробки.

Одним із піонерів у цій галузі можна вважати проект Brooklyn Microgrid (Нью-Йорк, США). Вони ще у 2016-2017 роках почали експериментувати з можливістю для мешканців одного району, які мають сонячні панелі, продавати

надлишки енергії своїм сусідам через платформу на базі Ethereum. Цей проект наочно продемонстрував технічну можливість P2P обміну енергією в міських умовах та викликав значний інтерес. Їхній підхід був зосереджений на створенні локальної енергетичної спільноти та дослідженні взаємодії з існуючими енергетичними компаніями та регуляторами.

Інший відомий приклад – австралійська компанія Power Ledger. Вони розробили блокчейн-платформу, яка дозволяє не тільки торгувати енергією P2P, але й відстежувати походження "зеленої" енергії та торгувати відповідними сертифікатами. Їхня технологія використовується в низці пілотних проектів у різних країнах світу, включаючи Австралію, Таїланд, Японію та США. Power Ledger часто використовує власний токен POWR для полегшення транзакцій та стимулювання участі в екосистемі. Їхній досвід показує важливість гнучкості платформи для адаптації до різних ринкових умов та регуляторних вимог.

У Європі, зокрема в Німеччині, активно розвивається компанія Sonnen зі своєю концепцією SonnenCommunity. Вони пропонують інтегровані рішення, що включають домашні сонячні електростанції, системи накопичення енергії (батареї) та програмну платформу, яка дозволяє учасникам спільноти обмінюватися надлишками енергії. Це приклад більш комплексного підходу, де технологічна платформа тісно пов'язана з конкретним обладнанням.

Також варто згадати такі проекти, як LO3 Energy (які також працювали над Brooklyn Microgrid, а потім розвивали власні рішення, наприклад, з використанням технології Exergy), WePower (платформа для фінансування проектів відновлюваної енергетики через токенізацію та подальшу торгівлю "зеленою" енергією), або Grid+ (який фокусувався на роздрібному постачанні електроенергії з використанням Ethereum та стабільних токенів).

Аналізуючи ці та інші проекти, можна виділити кілька загальних тенденцій та уроків:

- Технологічний стек: Більшість проектів на ранніх етапах орієнтувалися на публічний блокчейн Ethereum через його розвинену екосистему

смарт-контрактів. Однак, з часом деякі почали досліджувати або переходити на більш спеціалізовані блокчейни, сайдчейни або рішення другого рівня для подолання проблем масштабованості та вартості транзакцій.

- Бізнес-моделі: Існує різноманітність бізнес-моделей – від простих платформ для обміну між сусідами до складних систем з власними токенами, ринками похідних інструментів та інтеграцією з оптовими ринками енергії.
- Регуляторні виклики: Однією з найбільших перешкод для широкого впровадження P2P торгівлі енергією є існуюче енергетичне законодавство та регуляторні норми, які часто не пристосовані до таких інноваційних моделей. Багато проектів стикаються з необхідністю тісної співпраці з регуляторами та енергетичними компаніями.
- Залучення користувачів: Важливим аспектом є створення зручного та зрозумілого користувацького досвіду, а також демонстрація реальних економічних переваг для учасників, щоб мотивувати їх приєднуватися до платформи.
- Інтеграція з фізичним світом: Надійне вимірювання та верифікація обсягів виробленої та спожитої енергії (через інтелектуальні лічильники) є критично важливими для забезпечення довіри до системи.

Мій проект, хоча й є прототипом, намагається врахувати деякі з цих аспектів, зокрема, фокусуючись на створенні зрозумілого користувацького інтерфейсу та використанні Ethereum-сумісної технології, яка має велику спільноту та інструментарій. При цьому, обраний підхід з єдиним гібридним смарт-контрактом та off-chain базою даних для історії є спробою знайти баланс між децентралізацією та практичною ефективністю для користувача.

Назва платформи	Країна реалізації	Технологія блокчейн	Особливості реалізації	Недоліки / Обмеження
Brooklyn Microgrid	США	Ethereum	Торгівля між сусідами в межах мікромережі	Обмежено локальною спільнотою
Power Ledger	Австралія	PowerLedger chain	Власний токен, торгівля сертифікатами зеленої енергії	Складність інтеграції, залежність від регуляторів
SonnenCommunity	Німеччина	Приватний блокчейн	Повний пакет: панелі, батареї, обмін енергією	Висока вартість входу, залежність від обладнання
WePower	Литва/Іспанія	Ethereum	Токенізація енергії, фінансування генерації	Зосереджено на B2B, не підходить для домогосподарств
Проект (ця робота)	Україна	Ethereum (тестнет)	Єдиний смартконтракт, вебінтерфейс з MetaMask, API	Поки що прототип, потребує масштабування

Рисунок. 2.1 - Порівняльна характеристика аналогічних рішень

2.5 Проблеми безпеки та масштабованості блокчейн-рішень в енергетиці

Незважаючи на значний потенціал технології блокчейн для трансформації енергетичного сектору, її впровадження пов'язане з низкою викликів, серед яких питання безпеки та масштабованості є одними з найважливіших. Їх необхідно враховувати при проектуванні будь-якої блокчейн-платформи, особливо такої, що стосується критичної інфраструктури, як енергетика.

Проблеми безпеки:

- Вразливості смарт-контрактів: Смарт-контракти, як і будь-яке програмне забезпечення, можуть містити помилки або логічні вразливості. Оскільки код смарт-контракту після розгортання в блокчейні стає практично незмінним (або його зміна є дуже складною процедурою), будь-яка вразливість може призвести до значних фінансових втрат або неправильної роботи системи. Відомі приклади атак на смарт-контракти (наприклад, атака на The DAO у 2016 році) підкреслюють важливість ретельного аудиту коду, використання перевірених бібліотек (як OpenZeppelin, що я використовував для ReentrancyGuard) та дотримання найкращих практик безпечної розробки.
- Безпека приватних ключів користувачів: Взаємодія з блокчейн-платформою відбувається через криптографічні гаманці, де зберігаються приватні ключі користувачів. Втрата або компрометація приватного ключа означає втрату

доступу до активів та можливості підписувати транзакції. Тому важливим є навчання користувачів правилам безпечного зберігання ключів та використання надійних гаманців (наприклад, апаратних).

- Атаки на мережу блокчейну: Хоча публічні блокчейни, такі як Ethereum, є досить стійкими, теоретично можливі атаки, наприклад, "атака 51%", особливо для менших блокчейнів з меншою обчислювальною потужністю мережі. Для приватних та консорціумних блокчейнів ризики можуть бути пов'язані зі змовою учасників або компрометацією вузлів, що мають права на валідацію транзакцій.
- Безпека взаємодії on-chain та off-chain компонентів: У гібридних системах, як моя, де є взаємодія між смарт-контрактом та серверними компонентами чи базами даних, необхідно забезпечити безпеку цього зв'язку. Потрібно захищати API, через які відбувається обмін даними, та гарантувати, що off-chain дані не можуть бути скомпрометовані таким чином, щоб це вплинуло на коректність роботи on-chain логіки (наприклад, через передачу неправдивих даних, на основі яких смарт-контракт приймає рішення, якщо він покладається на зовнішні "оракули").
- Конфіденційність даних: Хоча прозорість є однією з переваг блокчейну, в енергетиці певна інформація (наприклад, детальні патерни споживання енергії окремими домогосподарствами) може бути чутливою. Тому потрібно розглядати механізми забезпечення конфіденційності, наприклад, через використання доказів з нульовим розголошенням (zero-knowledge proofs) або зберігання чутливих даних off-chain з наданням доступу лише авторизованим сторонам.

Проблеми масштабованості:

- Пропускна здатність транзакцій (TPS - Transactions Per Second): Багато публічних блокчейнів першого покоління, як Ethereum, мають обмежену пропускну здатність (близько 15-30 TPS). Для енергетичних ринків, де потенційно можуть відбуватися тисячі дрібних транзакцій щохвилини (наприклад, від

інтелектуальних лічильників або при частих змінах цін), цього може бути недостатньо. Це призводить до черг транзакцій та збільшення часу їх підтвердження.

- Вартість транзакцій (Gas Fees): У публічних блокчейнах кожна операція, що змінює стан (тобто кожна транзакція), потребує оплати "газу" – комісії майнерам (або валідаторам) за обробку. В періоди високого навантаження на мережу вартість газу може значно зростати, роблячи дрібні транзакції економічно невигідними. Для P2P торгівлі енергією, де угоди можуть бути на невеликі суми, високі комісії є серйозною перешкодою.
- Розмір блокчейну: З часом блокчейн накопичує все більше даних, що збільшує вимоги до зберігання для вузлів мережі та може ускладнювати синхронізацію нових вузлів.

Можливі рішення проблем масштабованості та безпеки: Для вирішення цих проблем розробляються та впроваджуються різні підходи:

- Рішення другого рівня (Layer 2 Scaling Solutions): Такі як State Channels, Plasma, Rollups (Optimistic Rollups, ZK-Rollups) для Ethereum. Вони дозволяють виносити значну частину транзакцій за межі основного блокчейну, обробляти їх швидше та дешевше, а потім періодично фіксувати лише підсумковий стан або докази в основній мережі.
- Сайдчейни (Sidechains): Окремі блокчейни, що пов'язані з основним блокчейном, але мають власні механізми консенсусу та параметри, що дозволяють досягти кращої масштабованості.
- Новіші блокчейн-платформи: Існують блокчейни третього покоління (наприклад, Polkadot, Cosmos, Solana, Algorand), які з самого початку проектувалися з урахуванням вимог до високої пропускної здатності та низьких комісій.
- Оптимізація смарт-контрактів: Написання ефективного коду смарт-контрактів, мінімізація операцій, що вимагають багато газу (особливо запис у сховище).

- Гібридні архітектури: Як у моєму проекті, де частина даних та логіки виноситься off-chain, щоб зменшити навантаження на блокчейн.
- Формальна верифікація та аудит смарт-контрактів: Для підвищення безпеки смарт-контрактів перед їх розгортанням.

При розробці моєї платформи я намагався врахувати ці аспекти, зокрема, обравши гібридний підхід для потенційної оптимізації та використовуючи перевірені практики безпеки, як ReentrancyGuard. Однак, питання масштабованості та довгострокової безпеки залишаються важливими напрямками для подальших досліджень та вдосконалень будь-якої блокчейн-системи в енергетиці.

Висновки до розділу 2

Таким чином, у даному розділі було здійснено комплексний огляд предметної області децентралізованої торгівлі енергією. Розглянуто різноманітні архітектури таких систем, проаналізовано типи блокчейн-платформ та ключову роль смарт-контрактів в автоматизації енергетичних транзакцій. Аналіз існуючих світових проектів дозволив виявити успішні практики та актуальні виклики, зокрема, пов'язані з безпекою та масштабованістю блокчейн-рішень в енергетиці. Отримані знання створили необхідну теоретичну базу для обґрунтованого проектування власної програмної платформи.

3 ПРОЕКТУВАННЯ ТА РОЗРОБКА ПЛАТФОРМИ ДЕЦЕНТРАЛІЗОВАНОЇ ТОРГІВЛІ ЕНЕРГІЄЮ

Після детального аналізу предметної області та існуючих рішень, наступним логічним етапом моєї дипломної роботи стало безпосереднє проектування та розробка програмної платформи для децентралізованої торгівлі енергією. Цей розділ присвячений саме цьому процесу: від формування загальної архітектури системи та визначення її ключових компонентів до реалізації смарт-контракту, серверної частини (API) та користувацького інтерфейсу. Моєю метою на цьому етапі було не просто написати код, а створити логічно побудовану, функціональну та потенційно масштабовану систему, яка б на практиці демонструвала основні принципи P2P торгівлі енергією на базі технології блокчейн.

3.1 Загальна архітектура платформи

Приступаючи до проектування, я намагався знайти такий архітектурний підхід, який би, з одного боку, забезпечував надійність, прозорість та децентралізацію, властиві технології блокчейн, а з іншого – дозволяв би створити зручний та швидкий користувацький досвід, а також оптимізувати витрати на зберігання даних та виконання операцій. Тому було прийнято рішення будувати платформу за багаторівневою гібридною архітектурою. Це означає, що система складається з кількох взаємопов'язаних рівнів та компонентів, частина з яких функціонує децентралізовано (on-chain), а частина – централізовано (off-chain), але при цьому вони тісно взаємодіють для досягнення спільної мети.

3.1.1 Компоненти системи та їх взаємодія

Розглянемо детальніше основні компоненти, з яких складається розроблена мною платформа, та як вони взаємодіють між собою. *(На цьому етапі в реальній*

записці було б доречно розмістити посилання на загальну архітектурну схему в Додатках, яку б я, як студент, намалював, щоб візуалізувати структуру).

1. Смарт-контракт (DecentralizedEnergyTrading.sol): Це, без перебільшення, ядро всієї децентралізованої частини платформи. Він розгорнутий в Ethereum-сумісній блокчейн-мережі і написаний мовою Solidity. Саме смарт-контракт містить всю основну бізнес-логіку, пов'язану з процесом торгівлі: реєстрацію пропозицій (оферів) на продаж енергії, перевірку умов, укладання угод купівлі-продажу, фіксацію ключових параметрів транзакцій та генерацію подій про ці транзакції. Він виступає як незмінний та прозорий набір правил, що гарантує чесність та автоматичне виконання угод.

2. Клієнтський інтерфейс (Frontend – папка frontend): Це веб-додаток, розроблений на бібліотеці React (основний логічний компонент – App.js, а також різноманітні підкомпоненти в frontend/src/components), з яким безпосередньо взаємодіють користувачі платформи (прозьюмери та споживачі). Через цей інтерфейс користувачі можуть:

- Підключати свої Web3-гаманці (наприклад, MetaMask) для автентифікації та підписання транзакцій.
- Переглядати актуальну ринкову інформацію: список доступних пропозицій енергії, ціни, статистику.
- Створювати та публікувати власні пропозиції на продаж енергії, викликаючи відповідні функції смарт-контракту.
- Акцептувати пропозиції інших учасників та купувати енергію, також ініціюючи транзакції до смарт-контракту.
- Переглядати історію своїх операцій та іншу персоналізовану інформацію. Фронтенд взаємодіє як безпосередньо зі смарт-контрактом (через бібліотеку Ethers.js та підключений гаманець користувача для відправки транзакцій та читання деяких on-chain даних), так і з серверною частиною (API) для отримання оброблених даних та виконання певних off-chain операцій.

3. Серверна частина (Backend API – файл api.py): Розроблена на Python з використанням фреймворку FastAPI. Серверна частина виконує декілька важливих функцій:

- Надає клієнтському інтерфейсу структуровані дані у зручному форматі через RESTful API ендпоінти.
- Взаємодіє зі смарт-контрактом для читання даних (через допоміжний модуль blockchain.py), які потім можуть бути агреговані, кешовані та передані на фронтенд.
- Взаємодіє з off-chain базою даних (price_history.db) для зберігання та отримання інформації, яка не є критичною для зберігання безпосередньо в блокчейні (наприклад, детальна історія транзакцій, зібрана слухачем подій, історія цін для графіків).
- Потенційно може реалізовувати додаткову бізнес-логіку, таку як розширена аналітика, система сповіщень для користувачів або більш складні алгоритми підбору пропозицій (хоча в поточній реалізації основний матчинг відбувається на рівні вибору користувачем оферу).

4. Модуль взаємодії з блокчейном (blockchain.py): Цей Python-модуль інкапсулює логіку підключення до Ethereum-вузла, завантаження ABI та адреси смарт-контракту, а також надає зручні функції-обгортки для виклику view-функцій смарт-контракту. Він використовується серверною частиною (API) для отримання on-chain даних.

5. Слухач подій (event_listener.py): Це окремий Python-скрипт, який працює в асинхронному режимі. Його завдання – підписатися на події, що генеруються смарт-контрактом (зокрема, на подію EnergyBought, яка спрацьовує при кожній успішній угоді купівлі-продажу). Отримавши таку подію, слухач розбирає її дані (хто купив, у кого, скільки енергії, за якою ціною, час, хеш транзакції) та зберігає їх у локальну базу даних SQLite (price_history.db). Це дозволяє створити off-chain сховище історії транзакцій, яке потім може бути

ефективно використане серверним API для надання даних користувачам без необхідності щоразу робити дорогі запити до історії подій самого блокчейну.

6. База даних SQLite (price_history.db): Локальна база даних, що використовується слухачем подій для зберігання історії угод (EnergyBought). Серверний API також використовує цю базу для надання користувачам історії їхніх операцій та, потенційно, для побудови графіків зміни цін або обсягів торгівлі.

Взаємодія компонентів в системі відбувається за чітко визначеними сценаріями. Наприклад, коли користувач створює пропозицію на фронтенді, він підписує транзакцію виклику функції setOffer смарт-контракту через свій гаманець. Після підтвердження транзакції в блокчейні смарт-контракт генерує подію OfferSet. Серверний API може (хоча це не було явно реалізовано в наданому коді main.py для цієї події, але є типовим підходом) відстежити цю подію і оновити кеш активних пропозицій. Коли відбувається купівля енергії (виклик buyFromProvider), генерується подія EnergyBought. Слухач event_listener.py фіксує цю подію та записує дані в price_history.db. Коли користувач запитує свою історію операцій на фронтенді, запит йде до серверного API, який звертається до price_history.db і повертає дані. Для відображення актуальних оферів або ринкової статистики фронтенд також робить запит до API, який, у свою чергу, через blockchain.py отримує дані з view-функцій смарт-контракту.

3.1.2 Рівні архітектури (клієнтський, серверний, блокчейн)

Запропоновану архітектуру можна також розглянути з точки зору класичних архітектурних рівнів, що допомагає краще зрозуміти розподіл відповідальності:

- Клієнтський рівень (Presentation Layer): Представлений веб-додатком на React (App.js та компоненти). Відповідає за взаємодію з користувачем, відображення інформації, збір введених даних та ініціацію дій. Саме тут відбувається підключення до Web3-гаманця користувача (MetaMask) для

підписання транзакцій, що відправляються безпосередньо до блокчейну, а також взаємодія з серверним API для отримання оброблених даних та виконання off-chain операцій.

- Серверний рівень (Application/Business Logic Layer & Data Access Layer): Представлений API на FastAPI (main.py), модулем blockchain.py та базою даних SQLite. Цей рівень містить:
  - Логіку обробки запитів від клієнта.
  - Логіку взаємодії зі смарт-контрактом (читання даних, потенційно відправка деяких транзакцій, якщо вони не ініціюються клієнтом).
  - Логіку роботи з off-chain базою даних (зберігання, вибірка, оновлення історії транзакцій, зібраної слухачем подій).
  - Частково бізнес-логіку, яка може бути недоцільною для реалізації у смарт-контракті (наприклад, агрегація даних, підготовка статистики, що не розраховується в контракті). Слухач подій event_listener.py також можна віднести до цього рівня, оскільки він виконує важливу функцію збору даних для серверної частини.
- Блокчейн-рівень (Persistence/Trust Layer): Представлений безпосередньо Ethereum-сумісною блокчейн-мережею та розгорнутим на ній смарт-контрактом DecentralizedEnergyTrading.sol. Цей рівень є основою для забезпечення децентралізації, незмінності, прозорості та автоматичного виконання ключових угод. Смарт-контракт зберігає критично важливий стан (активні офери, мінімальну ціну) та логіку (створення оферів, купівля енергії).

Такий розподіл на рівні дозволяє досягти гнучкості в розробці та підтримці системи. Наприклад, можна змінювати реалізацію клієнтського інтерфейсу, не зачіпаючи серверну логіку чи смарт-контракт, або оптимізувати роботу серверної частини без необхідності перерозгортання смарт-контракту (якщо його інтерфейс не змінюється). Гібридний характер архітектури, де блокчейн використовується для найважливіших аспектів довіри та виконання угод, а традиційні серверні технології – для забезпечення швидкості, зручності та управління великими

обсягами некритичних даних, є, на мою думку, оптимальним для такого типу систем на поточному етапі розвитку технологій.

3.2 Проектування бази даних та сховища даних поза ланцюгом (off-chain)

У гібридній архітектурі, яку я обрав для своєї платформи, важливу роль відіграє не тільки блокчейн (on-chain), але й сховище даних поза ланцюгом (off-chain). Рішення про те, які дані зберігати on-chain, а які off-chain, приймалося з урахуванням кількох факторів: критичності даних для забезпечення довіри та незмінності, вартості зберігання даних у блокчейні (витрати газу), частоти оновлення даних та вимог до швидкості доступу до них.

Для моєї платформи як off-chain сховище була обрана локальна база даних SQLite, файл якої має назву price_history.db. SQLite – це легка, файлова система управління базами даних, яка не потребує окремого серверного процесу і добре підходить для невеликих та середніх проектів, а також для прототипування. Вона проста в налаштуванні та використанні з Python завдяки вбудованому модулю sqlite3.

Основне призначення цієї бази даних у моєму проекті – зберігання історії транзакцій (угод купівлі-продажу енергії), які відбулися на платформі. Як було описано раніше, смарт-контракт DecentralizedEnergyTrading.sol генерує подію EnergyBought при кожній успішній угоді. Спеціальний скрипт-слухач event_listener.py відстежує ці події та записує детальну інформацію про кожну угоду в таблицю price_history нашої SQLite бази даних. Це дозволяє серверному API (файл api.py) швидко отримувати історію операцій для конкретного користувача (ендпоінт /user_activity/{user_address}) або загальну історію цін (ендпоінт /price_history) без необхідності робити складні та потенційно повільні запити до історії подій безпосередньо з блокчейну.

Структура таблиці price_history була спроектована таким чином, щоб зберігати всю необхідну інформацію про угоду:

- id: Унікальний ідентифікатор запису (ціле число, первинний ключ, автоінкремент).
- timestamp: Часова мітка угоди (ціле число, Unix timestamp), отримана з події смарт-контракту.
- price_wei_per_wh: Ціна за одиницю енергії (Ват-годину) у Wei, за якою відбулася угода (текстовий тип, щоб уникнути проблем з точністю великих чисел).
- energy_wh: Обсяг енергії, проданий/куплений у Ват-годинах (текстовий тип).
- tx_hash: Хеш транзакції в блокчейні, що відповідає цій угоді (текстовий, унікальний, щоб запобігти дублюванню записів).
- provider_address: Ethereum-адреса продавця енергії (текстовий).
- consumer_address: Ethereum-адреса покупця енергії (текстовий).

Для прискорення запитів до цієї таблиці, особливо при фільтрації за часом, був створений індекс по колонці timestamp.

Функція init_db() у файлі event_listener.py відповідає за створення цієї таблиці (та індексу), якщо вони ще не існують, при кожному запуску слухача подій. А функція save_event_to_db(event_data) в тому ж файлі здійснює безпосередньо запис даних про нову угоду в таблицю.

Окрім історії транзакцій, off-chain база даних потенційно могла б використовуватися для зберігання:

- Розширених профілів користувачів (наприклад, контактні дані, налаштування сповіщень), які недоцільно зберігати on-chain.
- Кешованих даних зі смарт-контракту для зменшення кількості запитів до блокчейну та прискорення роботи інтерфейсу.
- Аналітичних даних та агрегованої статистики, яка може бути занадто складною для розрахунку безпосередньо у смарт-контракті.

Однак, у поточній реалізації прототипу основний фокус off-chain зберігання був саме на історії транзакцій для ендпоінтів /price_history та /user_activity.

Використання SQLite як off-chain сховища для даного проекту є компромісом між простотою реалізації (особливо для прототипу), відсутністю потреби в окремому сервері баз даних та достатньою функціональністю для поставлених завдань. Для більш масштабних, промислових рішень, можливо, доцільніше було б використовувати більш потужні системи управління базами даних, такі як PostgreSQL або MongoDB, залежно від структури даних та вимог до навантаження.

3.3 Розробка смарт-контракту для управління платформою

Центральним елементом розробленої платформи, що забезпечує її децентралізований характер, прозорість та автоматизацію ключових процесів, є смарт-контракт. Як я вже зазначав, було прийнято рішення реалізувати основну on-chain логіку в рамках єдиного комплексного смарт-контракту, написаного мовою Solidity та розгорнутого в Ethereum-сумісній блокчейн-мережі. У цьому підрозділі я детально опишу процес проектування та реалізації цього смарт-контракту, його основні структури даних, функції та події, які разом формують правила взаємодії учасників на платформі. Файл смарт-контракту в моєму проекті називається DecentralizedEnergyTrading.sol.

Загальна концепція та вибір Solidity:

Мова програмування Solidity була обрана як основний інструмент для написання смарт-контракту, оскільки вона є стандартом де-факто для розробки на платформі Ethereum та інших EVM-сумісних (Ethereum Virtual Machine) блокчейнах. Solidity є об'єктно-орієнтованою мовою високого рівня зі статичною типізацією, синтаксис якої має схожі риси з JavaScript та C++, що дещо полегшує її вивчення. Вона надає всі необхідні інструменти для визначення станів контракту, логіки функцій, модифікаторів доступу та генерації подій.

При проектуванні смарт-контракту я керувався кількома основними принципами:

- Безпека: Намагався враховувати відомі вразливості смарт-контрактів та використовувати перевірені практики для їх уникнення. Зокрема, було імпортовано та використано ReentrancyGuard з бібліотеки OpenZeppelin для захисту від атак повторного входу.
- Ефективність (оптимізація газу): Операції в блокчейні Ethereum потребують оплати "газу". Тому я намагався оптимізувати код там, де це можливо, щоб зменшити витрати газу на виконання функцій, особливо тих, що змінюють стан контракту (наприклад, мінімізуючи кількість записів у сховище).
- Зрозумілість та читабельність: Код смарт-контракту мав бути достатньо зрозумілим, щоб його логіку можна було легко проаналізувати.
- Функціональна повнота: Смарт-контракт мав реалізовувати весь необхідний набір функцій для забезпечення основного циклу P2P торгівлі енергією на платформі.

Структури даних смарт-контракту:

Для зберігання інформації про стан платформи та її учасників у смарт-контракті було визначено декілька ключових структур даних:

- EnergyOffer: Це структура, що описує пропозицію (офер) на продаж енергії. Вона включає:
  - availableEnergyWh (тип uint256): Кількість доступної для продажу енергії у Ват-годинах.
  - priceWeiPerWh (тип uint256): Індивідуальна ціна, встановлена провайдером (продавцем) за одну Ват-годину енергії, виражена у Wei (найдрібніша одиниця ефіру).
  - isActive (тип bool): Прапорець, що вказує, чи є дана пропозиція активною на поточний момент.
- offers: Це публічне відображення (mapping) типу mapping(address => EnergyOffer). Воно пов'язує Ethereum-адресу провайдера з його поточною

пропозицією EnergyOffer. Таким чином, для кожної адреси може існувати лише одна активна пропозиція одночасно (хоча вона може оновлюватися).

- activeProvidersSet: Це приватний набір (set) Ethereum-адрес типу EnumerableSet.AddressSet. Він використовується для ефективного зберігання та ітерації по адресах тих провайдерів, які мають активні пропозиції. Бібліотека EnumerableSet від OpenZeppelin надає зручні методи для додавання, видалення та перебору елементів у наборі.
- EnergyRecord: Ця структура призначена для запису інформації про кожну успішну угоду купівлі-продажу енергії. Вона містить:
  - provider (тип address): Адреса продавця енергії.
  - consumer (тип address): Адреса покупця енергії.
  - energyBoughtWh (тип uint256): Кількість купленої енергії.
  - costInWei (тип uint256): Загальна вартість угоди у Wei.
  - priceWeiPerWhAtPurchase (тип uint256): Ціна за одиницю енергії, за якою відбулася купівля.
  - timestamp (тип uint256): Часова мітка (Unix timestamp) з блоку, в якому була зафіксована угода.
- energyRecords: Це приватне відображення mapping(address => EnergyRecord[]). Воно зберігає масив записів EnergyRecord для кожної адреси *покупця*, дозволяючи відстежувати історію його покупок. Хоча ця змінна є private, доступ до її даних для конкретного покупця надається через публічну view-функцію getEnergyRecords.

Окрім цих основних структур, контракт також зберігає адресу адміністратора (admin) та мінімально допустиму ціну (minPriceWeiPerWh), а також змінні для розрахунку статистики ринку (totalEnergyOfferedWh, totalWeightedPriceNumerator).

Основні функції смарт-контракту та їх призначення:

Смарт-контракт DecentralizedEnergyTrading.sol реалізує набір функцій, які можна умовно поділити на адміністративні, функції для управління пропозиціями, функції для здійснення угод та view-функції для читання даних.

1. Адміністративні функції:

- constructor(): Конструктор контракту, викликається один раз при його розгортанні. Він ініціалізує адресу адміністратора (admin) адресою того, хто розгортає контракт (msg.sender), та встановлює початкове значення для minPriceWeiPerWh.
- setAdmin(address _newAdmin): Дозволяє поточному адміністратору призначити нового адміністратора контракту.
- setMinPrice(uint256 _newMinPriceWeiPerWh): Дозволяє адміністратору встановити нову мінімальну ціну за енергію. Функція перевіряє, щоб нова ціна була позитивною, та генерує подію MinPriceSet.
- withdraw(): Дозволяє адміністратору вивести будь-які кошти (ETH), що могли випадково або навмисно накопичитися на балансі самого смарт-контракту. (Важливо зазначити, що при типовій P2P угоді кошти від покупця одразу перераховуються продавцю, а не залишаються на контракті, але ця функція може бути корисною для непередбачених випадків). Усі ці функції захищені модифікатором onlyAdmin, який гарантує, що їх може викликати лише поточний адміністратор контракту.

2. Функції управління пропозиціями (оферами):

- setOffer(uint256 _availableEnergyWh, uint256 _priceWeiPerWh): Це ключова функція для провайдерів енергії. Вона дозволяє користувачеві (msg.sender) створити нову пропозицію або оновити існуючу. Функція приймає кількість доступної енергії та ціну за одиницю. Всередині виконуються перевірки: кількість енергії та ціна мають бути позитивними, а ціна не може бути нижчою за встановлену адміністратором minPriceWeiPerWh. Якщо користувач оновлює існуючий активний офер, його попередній внесок у загальну ринкову статистику (totalEnergyOfferedWh та totalWeightedPriceNumerator) коректно віднімається

перед додаванням нового. Нові дані оферу зберігаються, він позначається як активний, оновлюється ринкова статистика, адреса провайдера додається до набору activeProvidersSet, і генерується подія OfferSet. Функція захищена від атак повторного входу модифікатором nonReentrant.

- deactivateOffer(): Дозволяє провайдеру (msg.sender) деактивувати свою поточну пропозицію. Контракт перевіряє, чи офер дійсно активний. Якщо так, то кількість енергії з цього оферу та його внесок у середньозважену ціну віднімаються від загальної ринкової статистики. Офер позначається як неактивний, його параметри (доступна енергія, ціна) обнуляються, а адреса провайдера видаляється з activeProvidersSet. Генерується подія OfferDeactivated. Ця функція також є nonReentrant.

3. Функція здійснення угоди:

- buyFromProvider(address _provider, uint256 _energyToBuyWh): Цю функцію викликає покупець (msg.sender), щоб придбати енергію у вказаного провайдера (_provider). Функція є payable, що означає, що разом з її викликом покупець повинен надіслати на адресу смарт-контракту кошти (ETH) для оплати. Всередині функції виконується низка перевірок: адреса провайдера має бути валідною, покупець не може купувати у самого себе, кількість енергії для купівлі має бути позитивною, пропозиція провайдера має бути активною, у провайдера має бути достатньо доступної енергії, а ціна провайдера має бути встановлена (більше нуля). Далі розраховується загальна вартість totalCost. Перевіряється, чи надіслав покупець достатньо коштів (msg.value >= totalCost). Якщо все гаразд, оновлюється ринкова статистика (зменшується totalEnergyOfferedWh та totalWeightedPriceNumerator на куплену кількість енергії та її вартість за ціною провайдера). Кількість доступної енергії у провайдера зменшується (offer.availableEnergyWh -= _energyToBuyWh). Якщо вся енергія провайдера була розкуплена, його офер автоматично деактивується, і він видаляється з activeProvidersSet. Інформація про угоду записується в масив energyRecords для покупця (цей запис в першу чергу для можливості перегляду історії покупок через

view-функцію контракту, але основне зберігання історії для API відбувається off-chain через слухача подій). Потім кошти totalCost перераховуються на адресу провайдера (payable(_provider).call{value: totalCost}("")). Якщо покупець надіслав більше коштів, ніж потрібно, решта повертається йому. Наприкінці генерується подія EnergyBought, що сигналізує про успішне завершення угоди. Ця функція також захищена nonReentrant.

4. View-функції (для читання даних):

- getOffer(address _provider): Повертає деталі пропозиції (доступна енергія, ціна, статус активності) для вказаного провайдера.
- getMinPrice(): Повертає поточну мінімальну ціну, встановлену адміністратором.
- getAdmin(): Повертає адресу адміністратора контракту.
- getActiveProviders(): Повертає масив адрес усіх провайдерів, які на даний момент мають активні пропозиції.
- getEnergyRecords(address consumer): Повертає масив усіх записів про покупки для вказаного споживача.
- getAverageWeightedMarketPrice(): Розраховує та повертає середньозважену ринкову ціну на основі всіх активних пропозицій (totalWeightedPriceNumerator / totalEnergyOfferedWh). Якщо активних пропозицій немає, повертає 0.
- getMarketStats(): Повертає загальну кількість запропонованої енергії та середньозважену ринкову ціну. Ці функції є public view, що означає, що вони лише читають дані зі стану контракту і не змінюють його, тому їх виклик не потребує оплати газу (крім витрат на читання даних вузлом).

Події (Events) смарт-контракту:

Події є важливим механізмом у смарт-контрактах Ethereum, який дозволяє інформувати зовнішні додатки (наприклад, мій event_listener.py або клієнтський інтерфейс) про те, що в контракті відбулися певні значущі зміни стану. Мій смарт-контракт визначає наступні події:

- OfferSet: Генерується при створенні або оновленні пропозиції.
- OfferDeactivated: Генерується при деактивації пропозиції.
- MinPriceSet: Генерується при зміні мінімальної ціни адміністратором.
- EnergyBought: Найважливіша подія для відстеження угод. Генерується при кожній успішній купівлі енергії та містить всю детальну інформацію про угоду.
- AdminChanged: Генерується при зміні адміністратора контракту.

Також у контракті присутні функції receive() external payable {} та fallback() external payable {}. Це спеціальні функції, які дозволяють контракту приймати ETH, навіть якщо викликається неіснуюча функція або просто надсилається переказ на адресу контракту. У даному контексті вони, ймовірно, залишені для загальної сумісності, хоча основний потік коштів при купівлі енергії обробляється у функції buyFromProvider.

Загалом, розроблений смарт-контракт є досить комплексним і реалізує всю необхідну on-chain логіку для функціонування прототипу платформи децентралізованої торгівлі енергією, з акцентом на прозорість, автоматизацію та безпеку угод.

3.4 Розробка серверної частини (API)

Після того, як було спроектовано та реалізовано смарт-контракт, що є ядром децентралізованої логіки платформи, наступним важливим кроком стала розробка серверної частини, або API (Application Programming Interface). Незважаючи на те, що ключові аспекти угод обробляються безпосередньо в блокчейні, серверна частина відіграє надзвичайно важливу роль у сучасних гібридних децентралізованих додатках (dApps). Вона слугує мостом між клієнтським інтерфейсом та блокчейном, а також дозволяє реалізувати функціонал, який є недоцільним або надто дорогим для виконання on-chain. У моєму проекті серверна частина реалізована у файлі main.py з використанням фреймворку FastAPI.

3.4.1 Вибір технологічного стеку (Python, FastAPI)

Для розробки серверної частини було обрано мову програмування Python та асинхронний веб-фреймворк FastAPI. Такий вибір не є випадковим і ґрунтується на кількох перевагах цих технологій:

- Python є однією з найпопулярніших мов для веб-розробки та наукових обчислень. Він має простий та зрозумілий синтаксис, величезну кількість готових бібліотек для вирішення найрізноманітніших завдань, включаючи бібліотеки для взаємодії з блокчейном (наприклад, Web3.py, яку я використовую у модулі blockchain.py), роботи з базами даних, обробки HTTP-запитів тощо. Велика спільнота розробників також є значним плюсом, оскільки легко знайти документацію та вирішення типових проблем.
- FastAPI – це сучасний, високопродуктивний веб-фреймворк для створення API на Python 3.7+ на основі стандартних підказок типів Python. Його ключові переваги, які стали вирішальними при виборі:
    - Висока продуктивність: FastAPI побудований на основі Starlette (для веб-частини) та Pydantic (для валідації даних), і завдяки асинхронній природі (використання async та await) він демонструє продуктивність, порівнянну з такими фреймворками, як NodeJS або Go. Це важливо для забезпечення швидкої відповіді API, особливо при обробці запитів від багатьох користувачів.
    - Швидкість розробки: FastAPI дозволяє створювати API дуже швидко. Завдяки використанню підказок типів Python, він автоматично генерує валідацію даних для запитів та відповідей, а також інтерактивну документацію API (на базі OpenAPI та Swagger UI). Це значно спрощує процес розробки, тестування та інтеграції з клієнтською частиною.

- Валідація даних**:** Pydantic, що лежить в основі FastAPI, забезпечує потужну та зручну систему валідації даних. Я визначив Pydantic-моделі (схеми) для всіх запитів та відповідей API, що гарантує коректність даних, які передаються між клієнтом та сервером, та зменшує кількість потенційних помилок.
- Асинхронність: Можливість використання async/await дозволяє ефективно обробляти операції введення-виведення (наприклад, запити до бази даних або до зовнішніх сервісів, включаючи блокчейн-вузол) без блокування основного потоку виконання, що покращує загальну відгукливість API.

Саме поєднання гнучкості Python та продуктивності й зручності FastAPI зробило цей стек оптимальним вибором для реалізації серверної частини моєї платформи.

3.4.2 Реалізація основних ендпоінтів API

Серверна частина платформи надає набір RESTful API ендпоінтів, через які клієнтський додаток (фронтенд) може отримувати необхідні дані та, потенційно, ініціювати певні дії. У файлі main.py було реалізовано декілька ключових груп ендпоінтів:

1. Загальні ендпоінти:
   - GET /: Кореневий ендпоінт, який повертає вітальне повідомлення. Часто використовується для перевірки доступності та працездатності API.
2. Ендпоінти для отримання даних зі смарт-контракту:
   - GET /admin_address: Повертає Ethereum-адресу адміністратора смарт-контракту. Для цього сервер викликає відповідний статичний метод get_admin_address() з класу BlockchainEnergySystem (модуль blockchain.py), який, у свою чергу, звертається до view-функції getAdmin() смарт-контракту.
   - GET /min_price: Повертає мінімальну ціну за енергію (у Wei), встановлену адміністратором. Аналогічно, використовується метод get_min_price_wei() з BlockchainEnergySystem.

- GET /average_market_price: Повертає середньозважену ринкову ціну енергії, розраховану смарт-контрактом (через get_average_market_price_wei() з BlockchainEnergySystem, який викликає getAverageWeightedMarketPrice() контракту).
- GET /offers: Повертає список усіх активних пропозицій (оферів) на продаж енергії. Сервер викликає get_active_providers_data() з BlockchainEnergySystem. Цей метод отримує список адрес активних провайдерів з контракту, а потім для кожного з них отримує деталі оферу (доступна енергія, ціна, статус активності) і формує список, який повертається клієнту.
- GET /market_stats: Повертає загальну статистику ринку, таку як загальна кількість запропонованої енергії та середньозважена ринкова ціна, викликаючи get_market_statistics() з BlockchainEnergySystem (що відповідає getMarketStats() у смарт-контракті).

3. Ендпоінти для отримання даних з off-chain бази даних (SQLite):

- GET /price_history: Цей ендпоінт призначений для надання даних для побудови графіка історії цін. Він звертається до локальної бази даних SQLite (price_history.db), до таблиці price_history (яка наповнюється скриптом event_listener.py), та вибирає записи про ціни, обсяги енергії та часові мітки угод. Ендпоінт підтримує параметри запиту для фільтрації за періодом часу (start_time, end_time) та обмеження кількості результатів (limit).
- GET /user_activity/{user_address}: Новий ендпоінт, розроблений для отримання історії операцій (покупок та продажів) конкретного користувача. Він приймає Ethereum-адресу користувача як параметр шляху. Запит також може включати параметри для фільтрації за типом активності (all, purchases, sales), а також для пагінації (limit, offset). Дані також вибираються з таблиці price_history бази даних SQLite, де відбувається пошук записів, у яких вказана адреса користувача фігурує або як покупець (consumer_address), або як продавець (provider_address). Для кожного знайденого запису визначається тип транзакції ('purchase' або 'sale') відносно запитуваного користувача та адреса контрагента.

Структура ендпоінтів та обробка запитів: Кожен ендпоінт у FastAPI визначається за допомогою декоратора (@app.get(...)), який вказує HTTP-метод та шлях. Функція, що йде за декоратором, є обробником цього запиту. FastAPI автоматично валідує параметри запиту (шляху, query-параметри) та тіло запиту (для POST, PUT) на основі підказок типів Python та Pydantic-моделей.

Для відповіді клієнту використовуються Pydantic-моделі (response_model), що забезпечує валідацію та серіалізацію даних у JSON-формат, а також автоматичне генерування схем для документації API. Наприклад, ендпоінт /offers використовує response_model=OfferListResponse, що гарантує, що відповідь буде містити поле offers, яке є списком об'єктів, що відповідають схемі OfferDetails.

Обробка помилок: У кожному обробнику ендпоінта реалізовано блок try...except для перехоплення можливих винятків, що можуть виникнути під час взаємодії з блокчейном, базою даних або при обробці даних. У разі помилки логується детальна інформація (за допомогою модуля logging, exc_info=True для виведення стеку викликів), а клієнту повертається відповідна HTTP-помилка (зазвичай з кодом 500 "Internal Server Error" або 501 "Not Implemented") з коротким описом проблеми за допомогою HTTPException з FastAPI.

CORS (Cross-Origin Resource Sharing): Для того, щоб клієнтський веб-додаток, який працює на іншому домені (наприклад, http://localhost:3000), міг робити запити до API, що працює на http://localhost:8000, було налаштовано CORS за допомогою CORSMiddleware. У поточній конфігурації дозволені запити лише з http://localhost:3000 та лише методом GET.

Загалом, розроблена серверна частина є важливим сполучним елементом системи, що забезпечує клієнтський інтерфейс необхідними даними, інкапсулює частину логіки взаємодії з блокчейном та надає доступ до історії транзакцій, що зберігається off-chain.

3.5 Розробка клієнтського інтерфейсу

Клієнтський інтерфейс (фронтенд) є тією частиною платформи, з якою безпосередньо взаємодіють користувачі – прозьюмери та споживачі енергії. Тому при його розробці основна увага приділялася не лише реалізації необхідного функціоналу, але й створенню інтуїтивно зрозумілого, зручного та візуально привабливого середовища. Метою було забезпечити користувачам легкий доступ до всіх можливостей платформи: від підключення їхніх Web3-гаманців та перегляду ринкової інформації до створення власних пропозицій, здійснення угод та моніторингу історії їхніх операцій. Весь фронтенд розроблено як односторінковий додаток (SPA - Single Page Application), що забезпечує швидку навігацію та оновлення даних без перезавантаження сторінки. Основний код клієнтської частини знаходиться у папці frontend мого проекту, з головним компонентом App.js.

3.5.1 Вибір технологічного стеку (React, Ethers.js)

Для побудови користувацького інтерфейсу було обрано бібліотеку React – одну з найпопулярніших та найпотужніших JavaScript-бібліотек для створення динамічних веб-інтерфейсів. Вибір React зумовлений низкою її переваг:

- Компонентний підхід: React дозволяє розбивати складний інтерфейс на невеликі, незалежні та перевикористовувані компоненти. Це значно спрощує процес розробки, тестування та підтримки коду. У моєму проекті це такі компоненти, як Header.js, OfferList.js, PersonalAccountTab.js, AdminPanel.js, PriceChart.js, MarketInfoDisplay.js тощо, кожен з яких відповідає за певну частину функціоналу або відображення.
- Віртуальний DOM: React використовує концепцію віртуального DOM, що дозволяє оптимізувати процес оновлення реального DOM-дерева браузера. Це забезпечує високу продуктивність та швидку реакцію інтерфейсу на дії користувача.

- Велика екосистема та спільнота: Навколо React існує величезна кількість готових бібліотек, інструментів розробки та активна спільнота, що значно полегшує вирішення будь-яких завдань та пошук інформації.
- Декларативний стиль програмування: React дозволяє описувати, *як* має виглядати інтерфейс за певного стану, а сама бібліотека бере на себе оновлення DOM при зміні цього стану.

Для управління станом усередині компонентів активно використовувалися вбудовані хуки React, такі як useState (для локального стану компонента), useEffect (для виконання побічних ефектів, наприклад, запитів до API або підписки на події), useCallback (для мемоізації функцій-колбеків, щоб уникнути їх зайвого перестворення) та useMemo (для мемоізації результатів складних обчислень).

Для взаємодії з блокчейном Ethereum з боку клієнта була обрана бібліотека Ethers.js. Це потужна та гнучка JavaScript-бібліотека, яка надає повний набір інструментів для роботи з Ethereum:

- Підключення до гаманців: Ethers.js дозволяє легко інтегруватися з Web3-гаманцями користувачів, такими як MetaMask (через об'єкт window.ethereum), для отримання адрес, балансів та запиту підпису транзакцій. У моєму App.js це реалізовано у функції connectWallet.
- Взаємодія зі смарт-контрактами: Бібліотека дозволяє створювати екземпляри смарт-контрактів на основі їхньої адреси та ABI (Application Binary Interface), що завантажується з JSON-файлу (ContractABI в App.js). Через ці екземпляри можна викликати view-функції контракту для читання даних та формувати транзакції для виклику функцій, що змінюють стан (наприклад, setOffer, buyFromProvider).
- Робота з транзакціями: Ethers.js надає інструменти для формування, підписання (через гаманець) та відправлення транзакцій у блокчейн, а також для очікування їхнього підтвердження.

Для здійснення HTTP-запитів до серверного API (FastAPI) використовувалася бібліотека Axios, яка є простим та популярним HTTP-клієнтом для браузера та Node.js. Базовий URL для запитів був налаштований глобально (axios.defaults.baseURL).

Для інтернаціоналізації (i18n), тобто підтримки різних мов інтерфейсу, використовувався хук useTranslation з бібліотеки react-i18next. Файли з перекладами (en/translation.json, ua/translation.json) знаходяться у папці frontend/src/locales.

Для відображення спливаючих повідомлень (тостів) про успіх, помилки або інформаційних повідомлень використовувалася бібліотека react-toastify. А для створення системи вкладок ("Ринок", "Особистий кабінет", "Адмін панель") – компонент Tab з бібліотеки @headlessui/react, яка надає доступні та нестилізовані UI-примітиви, що легко інтегруються з Tailwind CSS.

Стилізація компонентів здійснювалася переважно за допомогою Tailwind CSS – utility-first CSS фреймворку, який дозволяє швидко створювати сучасні інтерфейси, комбінуючи готові класи безпосередньо в JSX-розмітці. Конфігурація Tailwind CSS знаходиться у файлі tailwind.config.js.

3.5.2 Проектування користувацьких сценаріїв та інтерфейсів

При проектуванні інтерфейсу я намагався поставити себе на місце користувача та продумати основні сценарії взаємодії з платформою. Головний компонент App.js є точкою входу та агрегатором для всіх інших візуальних компонентів та логіки.

Основні користувацькі сценарії, реалізовані в інтерфейсі:

1. Підключення до платформи:
   - Користувач бачить кнопку "Підключити гаманець" у компоненті Header.js.
   - Натискання на неї викликає функцію connectWallet в App.js, яка ініціює взаємодію з MetaMask.

- Після успішного підключення в Header.js відображається адреса користувача та його баланс ETH. Також завантажуються всі релевантні дані з блокчейну та API.

2. Перегляд ринкової інформації:

- На вкладці "Ринок" користувач бачить:
  - Компонент MarketInfoDisplay.js, що відображає поточну мінімальну ціну та середньозважену ринкову ціну (дані надходять з App.js зі станів minPriceWei та averageMarketPriceWei).
  - Компонент OfferList.js, що відображає список активних пропозицій енергії (дані зі стану offers в App.js). Кожен елемент списку показує адресу провайдера, доступну енергію та ціну. Для кожної пропозиції є кнопка "Купити".
  - Компонент PriceChart.js, який (імовірно, на основі даних з ендпоінта /price_history) відображає графік зміни цін.

3. Управління власною пропозицією (для прозьюмерів):

- На вкладці "Особистий кабінет" (компонент PersonalAccountTab.js) користувач бачить інформацію про свою поточну пропозицію (якщо вона є, дані зі стану myOffer в App.js).
- Є форма для створення або оновлення пропозиції. При сабміті цієї форми викликається функція handleSetOffer з App.js, яка ініціює транзакцію до смарт-контракту.
- Є кнопка для деактивації своєї пропозиції, що викликає handleDeactivateOfferContract з App.js.

4. Купівля енергії (для споживачів):

- У списку пропозицій (OfferList.js) користувач може натиснути кнопку "Купити" навпроти обраної пропозиції.
- Це викликає функцію handleBuyEnergy з App.js, яка приймає адресу провайдера, кількість енергії для купівлі та ціну. Функція формує та відправляє транзакцію до смарт-контракту.

5. Перегляд історії операцій:

- В "Особистому кабінеті" (PersonalAccountTab.js) відображається історія операцій користувача (дані зі стану userActivityHistory в App.js, отримані з ендпоінта /user_activity/{user_address}).

6. Адміністративні функції (для адміністратора):

- Якщо поточний користувач є адміністратором (isAdmin === true), йому доступна вкладка "Адмін панель" (компонент AdminPanel.js).
- Тут адміністратор може бачити поточну мінімальну ціну та, ймовірно, змінювати її за допомогою форми, яка викликає функцію handleSetMinPrice з App.js.

Інтерфейс спроектовано з урахуванням можливості перемикання між світлою та темною темами оформлення (стан isDark та функція toggleTheme в App.js, стилі керуються динамічно).

3.5.3 Інтеграція з Web3-провайдером (MetaMask)

Інтеграція з Web3-провайдером, таким як MetaMask, є ключовим аспектом для будь-якого децентралізованого додатку, оскільки саме через нього користувач може взаємодіяти з блокчейном, використовуючи свої приватні ключі безпечно.

У компоненті App.js ця інтеграція реалізована наступним чином:

1. Виявлення провайдера: При спробі підключення гаманця (connectWallet) код перевіряє наявність об'єкта window.ethereum. Цей об'єкт інжектується в браузер гаманцями типу MetaMask і слугує точкою входу для взаємодії.

2. Запит дозволу: Якщо window.ethereum знайдено, викликається window.ethereum.request({ method: 'eth_requestAccounts' }). Це асинхронний запит, який спонукає MetaMask показати користувачеві діалогове вікно з проханням дозволити додатку доступ до його адрес.

3. Створення екземплярів Ethers.js: Після отримання дозволу та адреси користувача, створюється:

- ethers.BrowserProvider(window.ethereum): Це провайдер Ethers.js, який використовує існуюче з'єднання MetaMask з блокчейн-вузлом. Він дозволяє читати дані з блокчейну.
- web3Provider.getSigner(): Отримується об'єкт підписувача (signer) з провайдера. Signer представляє акаунт користувача і необхідний для підписання та відправлення транзакцій, що змінюють стан блокчейну (наприклад, setOffer, buyFromProvider). Ці об'єкти (provider та signer) зберігаються у стані App.js і передаються іншим компонентам або використовуються для створення екземплярів смарт-контракту (contractInstance та contractInstanceWithSigner).

4. Підписання та відправка транзакцій: Коли користувач виконує дію, що потребує взаємодії зі смарт-контрактом (наприклад, купівля енергії), відповідна функція в App.js (наприклад, handleBuyEnergy) викликає метод екземпляра contractInstanceWithSigner. Ethers.js автоматично формує транзакцію і через window.ethereum передає її в MetaMask, де користувач бачить деталі транзакції (куди, скільки, яка комісія) і має її підтвердити своїм приватним ключем. Лише після підтвердження транзакція відправляється в блокчейн-мережу.

5. Обробка подій гаманця (опціонально, не реалізовано явно в наданому коді, але є хорошою практикою): window.ethereum також може генерувати події, наприклад, при зміні акаунту користувача в MetaMask або зміні мережі. Додаток може підписуватися на ці події та відповідно реагувати (наприклад, оновлювати дані або пропонувати користувачеві переключити мережу).

Таким чином, інтеграція з MetaMask через Ethers.js дозволяє платформі безпечно взаємодіяти з блокчейном від імені користувача, не маючи прямого доступу до його приватних ключів.

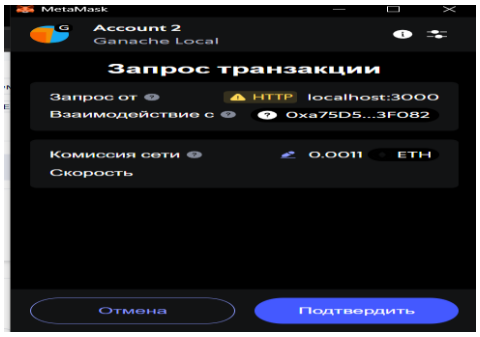


Рисунок. 3.1 - створення офера

3.6 Механізми забезпечення безпеки та автентифікації

При розробці будь-якої системи, а особливо такої, що пов'язана з фінансовими транзакціями та управлінням ресурсами, питання безпеки та надійної автентифікації користувачів є першочерговими. У моїй платформі для децентралізованої торгівлі енергією було вжито низку заходів та використано механізми, спрямовані на забезпечення належного рівня безпеки як на рівні смарт-контракту, так і на рівні серверної та клієнтської частин.

Безпека на рівні смарт-контракту:

Смарт-контракт DecentralizedEnergyTrading.sol є критично важливим компонентом, тому при його розробці я намагався дотримуватися визнаних практик безпечного програмування на Solidity:

1. Захист від атак повторного входу (Reentrancy): Це одна з найпоширеніших та найнебезпечніших вразливостей смарт-контрактів. Для її запобігання я використав успадкування від контракту ReentrancyGuard з перевіреної бібліотеки OpenZeppelin. Всі ключові функції, що змінюють стан та взаємодіють з іншими адресами (особливо ті, що передбачають переказ коштів, як buyFromProvider, або оновлюють стан на основі зовнішніх викликів, хоча в моєму випадку прямих зовнішніх викликів з оновленням стану мало), були позначені модифікатором nonReentrant. Це гарантує, що функція не може бути викликана повторно зловмисником до того, як її перше виконання повністю завершиться.

2. Контроль доступу (Модифікатори): Для адміністративних функцій, таких як setAdmin, setMinPrice, withdraw, використовується модифікатор onlyAdmin. Він перевіряє, що msg.sender (адреса, що викликає функцію) збігається з адресою адміністратора, збереженою в контракті. Це запобігає несанкціонованому доступу до критично важливих налаштувань контракту.

3. Перевірки вхідних даних (require): У всіх функціях, що приймають параметри від користувачів або інших контрактів, використовуються інструкції require для перевірки коректності цих даних перед тим, як виконувати будь-які дії. Наприклад, у функції setOffer перевіряється, що кількість енергії та ціна є позитивними, а ціна не нижча за мінімально встановлену. У функції buyFromProvider перевіряється валідність адреси провайдера, кількість енергії для купівлі, активність пропозиції, наявність достатньої енергії у провайдера та достатність надісланих коштів покупцем. Такі перевірки допомагають уникнути неочікуваної поведінки та потенційних помилок.

4. Безпечна математика (SafeMath): Хоча у версіях Solidity ^0.8.0 та вище вбудований захист від цілочисельних переповнень та збіднень, при роботі з попередніми версіями було б важливо використовувати бібліотеку SafeMath. У моєму випадку, з pragma solidity ^0.8.19;, компілятор вже дбає про ці аспекти, але я все одно намагався писати математичні операції акуратно, наприклад, використовуючи запобіжники від отримання від'ємного значення для uint при відніманні у статистиці totalEnergyOfferedWh та totalWeightedPriceNumerator.

5. Управління станом та подіями: Чітке управління станами оферів (isActive) та генерація подій (OfferSet, EnergyBought тощо) при кожній значущій зміні стану дозволяють не тільки забезпечити коректну роботу логіки, але й надати прозору інформацію для відстеження та аудиту.

Автентифікація та безпека на клієнтському та серверному рівнях:

1. Автентифікація через Web3-гаманець: Основний механізм автентифікації користувачів на платформі – це використання їхніх Ethereum-гаманців (наприклад, MetaMask). Коли користувач підключає свій гаманець до

веб-додатку (App.js), платформа отримує його публічну адресу. Всі транзакції, що змінюють стан смарт-контракту (наприклад, створення пропозиції, купівля енергії), вимагають підпису від користувача через його гаманець. Це гарантує, що дії виконуються від імені власника відповідного приватного ключа.

2. Захист API (серверна частина main.py):

- CORS (Cross-Origin Resource Sharing): Налаштовано для дозволу запитів лише з довірених джерел (у моєму випадку, з http://localhost:3000. Це запобігає використанню API сторонніми веб-сайтами.
- Валідація вхідних даних: FastAPI разом з Pydantic-моделями автоматично валідує всі вхідні дані для ендпоінтів (параметри шляху, query-параметри, тіла запитів). Це захищає від багатьох типів атак, пов'язаних з передачею некоректних або шкідливих даних. Наприклад, параметр user_address в ендпоінті /user_activity/{user_address} валідується як EthereumAddress.
- Обробка винятків: Належна обробка помилок та винятків запобігає витоку чутливої інформації та забезпечує більш стабільну роботу API.
- HTTPS (у промисловому середовищі): Хоча в локальному середовищі розробки використовується HTTP, для розгортання в реальному середовищі обов'язковим є використання HTTPS для шифрування всього трафіку між клієнтом та сервером.
- Захист від поширених веб-вразливостей: Хоча FastAPI має певний вбудований захист, при подальшому розвитку важливо пам'ятати про такі загрози, як XSS (Cross-Site Scripting), CSRF (Cross-Site Request Forgery), SQL-ін'єкції (якщо використовується SQL-база даних, хоча SQLite менш вразлива до деяких типів віддалених атак), та вживати відповідних заходів для їх запобігання.

3. Безпека off-chain бази даних (price_history.db):

- Оскільки база даних SQLite є файловою, важливо забезпечити належний захист доступу до цього файлу на сервері, де працює event_listener.py та main.py.

- При використанні параметризованих запитів (як у моєму коді cursor.execute(query, tuple(params))) для взаємодії з SQLite, це допомагає запобігти SQL-ін'єкціям.

4. Загальні міркування безпеки:

- Регулярні оновлення: Важливо підтримувати в актуальному стані всі бібліотеки та фреймворки (Solidity, OpenZeppelin, Web3.py, Ethers.js, FastAPI, React тощо), оскільки оновлення часто містять виправлення відомих вразливостей.
- Моніторинг та логування: Налаштоване логування (як у main.py, blockchain.py та event_listener.py) допомагає відстежувати роботу системи та виявляти підозрілу активність.
- Освіта користувачів: Інформування користувачів про важливість безпечного зберігання їхніх приватних ключів та обережність при підписанні транзакцій.

Хоча в рамках дипломного проекту-прототипу неможливо реалізувати абсолютно всі рівні захисту, що застосовуються в промислових системах, я намагався закласти основи безпечної архітектури та використовувати перевірені інструменти і практики там, де це було доцільно. Автентифікація на основі криптографічних гаманців та використання смарт-контракту для управління ключовими аспектами угод вже самі по собі забезпечують досить високий рівень довіри та безпеки для основних P2P-операцій.

Висновки до розділу 3

Підсумовуючи викладене у третьому розділі, можна констатувати, що було детально описано весь процес проектування та розробки ключових компонентів платформи децентралізованої торгівлі енергією. Сформовано загальну гібридну архітектуру системи, спроектовано структуру off-chain бази даних. Центральним елементом стала розробка єдиного комплексного смарт-контракту на Solidity, що реалізує основну on-chain логіку. Також було розроблено серверну частину (API)

на Python з використанням FastAPI для взаємодії з блокчейном та базою даних, і клієнтський веб-інтерфейс на React для взаємодії з користувачем. Окреслено механізми забезпечення безпеки та автентифікації на різних рівнях системи. Результатом цього етапу є створення функціонального прототипу, готового до тестування.

4 ІНСТРУМЕНТАЛЬНІ ЗАСОБИ ТА СЕРЕДОВИЩЕ РОЗРОБКИ

Успішна реалізація складного програмного проекту, яким є платформа для децентралізованої торгівлі енергією, значною мірою залежить від правильного вибору інструментальних засобів та ефективної організації середовища розробки. Цей розділ присвячений детальному опису технологічного стеку, який був використаний для створення всіх компонентів платформи: від смарт-контракту до користувацького інтерфейсу. Я намагався обирати сучасні, перевірені та добре документовані інструменти, що дозволило б не тільки реалізувати весь запланований функціонал, але й забезпечити належний рівень якості, безпеки та потенціал для подальшого розвитку проекту.

4.1 Мови програмування

Вибір мов програмування для різних частин системи був зумовлений специфікою завдань, які ці частини виконують, та особливостями самих мов.

- Solidity (версія ^0.8.19): Це основна мова для розробки смарт-контракту DecentralizedEnergyTrading.sol. Solidity є статично типізованою, об'єктно-орієнтованою мовою, спеціально створеною для написання смарт-контрактів, що виконуються на віртуальній машині Ethereum (EVM). Її синтаксис, що нагадує JavaScript та C++, та потужні можливості для роботи з адресами, структурами даних, відображеннями (mappings) та подіями роблять її стандартом для розробки децентралізованих додатків на Ethereum. Обрана версія ^0.8.19 є однією з останніх на момент розробки та включає важливі покращення безпеки, зокрема, вбудований захист від цілочисельних переповнень та збіднень, що раніше вимагало використання окремих бібліотек типу SafeMath.
- Python (версія 3.10): Ця високорівнева мова програмування загального призначення була обрана для розробки всієї серверної логіки, включаючи API

(файл main.py) та скрипт-слухач подій (event_listener.py), а також модуль для взаємодії з блокчейном (blockchain.py). Переваги Python – це його читабельність, велика стандартна бібліотека, величезна кількість сторонніх пакетів (зокрема, для веб-розробки та взаємодії з блокчейном), потужна підтримка асинхронного програмування та велика спільнота розробників. Це дозволило швидко та ефективно реалізувати необхідний серверний функціонал.

- JavaScript (стандарт ECMAScript 6 та новіший): Як основна мова для створення динамічних та інтерактивних веб-інтерфейсів, JavaScript був використаний для розробки клієнтської частини платформи (фронтенду, основний файл App.js та компоненти в папці frontend). Використання сучасних можливостей JavaScript (ES6+), таких як класи, модулі, стрілкові функції, проміси, async/await, дозволило писати більш структурований та підтримуваний код.

4.2 Середовище розробки (IDE)

Для забезпечення комфортної та продуктивної роботи над кодом було використано інтегровані середовища розробки (IDE), що надають широкий спектр інструментів для написання, налагодження та управління проектом.

- PyCharm Professional Edition (від JetBrains): Це IDE було основним для розробки Python-частини проекту (FastAPI-бекенд, слухач подій, модуль взаємодії з блокчейном). PyCharm пропонує чудову підтримку Python, включаючи інтелектуальне автодоповнення коду, інструменти рефакторингу, інтегрований налагоджувач, підтримку віртуальних середовищ, інструменти для роботи з базами даних та тісну інтеграцію з системою контролю версій Git. Професійна версія також надає розширену підтримку для веб-фреймворків, таких як FastAPI.
- Visual Studio Code (VS Code): Цей легкий, але потужний редактор коду використовувався для розробки смарт-контрактів на Solidity (завдяки наявності популярних розширень, таких як "Solidity" від Juan Blanco, що забезпечують підсвічування синтаксису, компіляцію та лінтинг) та для роботи з

JavaScript/React кодом фронтенду. VS Code також має чудову підтримку Git, вбудований термінал та велику кількість розширень для різних мов та технологій.

Використання обох IDE дозволило ефективно працювати з різними частинами проекту, використовуючи переваги кожного інструменту.

4.3 Фреймворки та бібліотеки

Для реалізації специфічного функціоналу та прискорення розробки було задіяно низку ключових фреймворків та бібліотек:

- Для розробки смарт-контрактів (Solidity):
  - OpenZeppelin Contracts (@openzeppelin/contracts): Це стандартна де-факто бібліотека перевірених, безпечних та багаторазово використовуваних смарт-контрактів та утиліт для Solidity. У моєму проекті з неї були використані:
    - ReentrancyGuard.sol: Для захисту функцій смарт-контракту від атак повторного входу (reentrancy attacks), що є однією з найпоширеніших вразливостей.
    - EnumerableSet.sol (зокрема, EnumerableSet.AddressSet): Для ефективного управління набором адрес активних провайдерів, що дозволяє легко додавати, видаляти та перебирати елементи набору, а також отримувати їх кількість.
  - Truffle Suite : Наявність truffle-config.js та структури папки build/contracts (де зазвичай зберігаються артефакти компіляції – ABI та байт-код) вказує на те, що для життєвого циклу розробки смарт-контрактів (компіляція, міграція/розгортання, тестування) використовувався один з цих популярних фреймворків. Вони надають зручне середовище та інструменти командного рядка для автоматизації цих процесів.
- Для розробки серверної частини (Python):
  - FastAPI: Як уже зазначалося, це сучасний веб-фреймворк для створення API. Він використовувався для визначення ендпоінтів, обробки HTTP-

запитів, валідації даних за допомогою Pydantic-моделей та автоматичної генерації документації OpenAPI.

- Uvicorn: ASGI-сервер, який використовувався для запуску FastAPI-додатку. Uvicorn є високопродуктивним сервером, рекомендованим для FastAPI.
- Web3.py: Основна Python-бібліотека для взаємодії з вузлами Ethereum. Вона використовувалася у модулі blockchain.py для підключення до RPC-кінцевої точки (HTTP або WebSocket), завантаження ABI та адреси смарт-контракту, виклику його view-функцій, а також у event_listener.py для створення фільтрів подій та їх обробки.
- python-dotenv: Дозволяє завантажувати конфігураційні змінні (наприклад, URL вузла, адресу контракту) з файлу .env замість того, щоб жорстко кодувати їх у скриптах, що є хорошою практикою для безпеки та гнучкості конфігурації.
- SQLite3 (вбудований модуль): Використовувався для створення та управління локальною базою даних price_history.db, де зберігаються дані про угоди, отримані слухачем подій.

- Для розробки клієнтської частини (JavaScript/React):
  - React (версія 18.x): Основна бібліотека для побудови користувацького інтерфейсу. Використовувалися функціональні компоненти та хуки (useState, useEffect, useCallback, useMemo, useContext).
  - Ethers.js (версія 6.x): Потужна бібліотека для взаємодії з блокчейном Ethereum з боку клієнта. Використовувалася для підключення до Web3-гаманця MetaMask (new ethers.BrowserProvider(window.ethereum)), отримання адреси користувача та балансу, створення екземпляра смарт-контракту (new ethers.Contract(...)) та для формування, підписання й відправлення транзакцій.
  - Axios: Популярний HTTP-клієнт на основі промісів для здійснення запитів до серверного API (FastAPI).
  - react-router-dom: (Ймовірно, використовувалася, хоча не згадується явно в App.js для навігації, але типова для SPA на React, якщо є декілька

"сторінок" або подань, керованих URL). Якщо у вас є кілька логічних сторінок, варто її згадати. Для системи вкладок використовується @headlessui/react.

- react-toastify: Для відображення неблокуючих спливаючих повідомлень (тостів) користувачеві про результат операцій (успіх, помилка, інформація).
- @headlessui/react: Бібліотека нестилізованих, повністю доступних UI-компонентів. У проекті використовувався її компонент Tab для реалізації системи вкладок в інтерфейсі.
- react-i18next та i18next: Бібліотеки для реалізації інтернаціоналізації (i18n) додатку, що дозволяє легко додавати та переключати мови інтерфейсу (англійська, українська).
- Tailwind CSS: Utility-first CSS фреймворк, який використовувався для швидкого та гнучкого стилістичного оформлення компонентів без написання великої кількості кастомного CSS.

4.4 Інструменти для роботи з блокчейном

Окрім бібліотек Web3.py та Ethers.js, для розробки та взаємодії з блокчейном використовувалися такі інструменти:

- MetaMask: Найпопулярніший браузерний Web3-гаманець, який слугував основним інструментом для користувачів для підключення до платформи, управління їхніми акаунтами в мережі Ethereum (або тестових мережах) та для підписання всіх транзакцій, що відправляються до смарт-контракту.
- Ganache: (Або альтернатива, як Hardhat Network). Це персональний локальний блокчейн для розробки та тестування Ethereum-додатків. Ganache надає набір тестових акаунтів з ефіром, дозволяє швидко розгортати смарт-контракти, миттєво підтверджувати транзакції та переглядати їх деталі. Це значно прискорює цикл розробки, оскільки не потрібно чекати підтверджень у реальних тестових або основній мережі та не витрачається реальний ефір.

- Remix IDE (для початкової розробки/тестування смарт-контракту): Онлайн-IDE, що часто використовується для швидкого написання, компіляції, розгортання та тестування смарт-контрактів Solidity без необхідності налаштування локального середовища.

4.5 Система контролю версій

Для ефективного управління вихідним кодом проекту, відстеження змін, можливості повернення до попередніх версій та потенційної командної роботи використовувалася розподілена система контролю версій Git. Всі файли проекту були частиною Git-репозиторію. Файл .gitignore використовувався для визначення файлів та папок, які не повинні відстежуватися системою контролю версій (наприклад, віртуальні середовища Python, папки node_modules, файли конфігурації IDE, файли .env з секретними даними). Ймовірно, для віддаленого зберігання репозиторію та резервного копіювання використовувалася одна з онлайн-платформ, таких як GitHub або GitLab.

4.6 Засоби тестування

Тестування є невід'ємною частиною процесу розробки програмного забезпечення, особливо для таких складних систем, як блокчейн-додатки. Хоча конкретні файли з тестами не були частиною наданого коду для аналізу, я опишу інструменти, які типово використовуються і які я б застосовував для тестування розробленої платформи:

- Тестування смарт-контрактів (Solidity):
  - Фреймворки Truffle надають потужні засоби для написання та виконання автоматизованих тестів для смарт-контрактів. Тести зазвичай пишуться на JavaScript (з використанням Node.js) і дозволяють перевіряти логіку функцій контракту, правильність зміни стану, генерацію подій, обробку граничних випадків та безпеку (наприклад, перевірку модифікаторів доступу).

Популярними бібліотеками для тверджень (assertions) у таких тестах є Mocha (як тестовий фреймворк) та Chai (як бібліотека тверджень), або Waffle (спеціалізована бібліотека для тестування смарт-контрактів Ethereum, часто використовується з Hardhat).

- Тестування серверної частини (Python/FastAPI):
  - Pytest: Це гнучкий та потужний фреймворк для тестування Python-коду. FastAPI має чудову вбудовану підтримку для тестування за допомогою TestClient, який дозволяє робити HTTP-запити до API без запуску реального сервера. Можна писати юніт-тести для окремих функцій та інтеграційні тести для перевірки роботи ендпоінтів, валідації даних та взаємодії з базою даних (з використанням тестової бази даних або мок-об'єктів).
- Тестування клієнтської частини (React):
  - Jest: Популярний JavaScript-фреймворк для тестування, розроблений Facebook. Він часто використовується разом з React Testing Library або Enzyme для тестування React-компонентів. React Testing Library фокусується на тестуванні компонентів з точки зору користувача (як він взаємодіє з інтерфейсом), що допомагає писати більш надійні та підтримувані тести. Можна проводити юніт-тестування окремих компонентів, їхньої логіки та рендерингу, а також інтеграційні тести для перевірки взаємодії між компонентами та їхньої реакції на дії користувача.

Комплексне використання цих інструментальних засобів та технологій дозволило мені структурувати процес розробки, забезпечити належну якість коду на кожному етапі та створити функціональний прототип платформи, що відповідає поставленим завданням.

Висновки до розділу 4

Таким чином, у даному розділі було надано вичерпний опис інструментальних засобів та технологій, що були використані на всіх етапах розробки платформи децентралізованої торгівлі енергією. Обґрунтовано вибір мов програмування (Solidity, Python, JavaScript), середовищ розробки (PyCharm, VS Code), ключових фреймворків та бібліотек (OpenZeppelin, FastAPI, Web3.py, React, Ethers.js), інструментів для роботи з блокчейном (MetaMask, Ganache/Hardhat) та системи контролю версій (Git). Також розглянуто потенційні засоби для тестування різних компонентів системи. Комплексне використання цих інструментів дозволило ефективно реалізувати поставлені завдання та створити працездатний програмний прототип.

5 ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ ПЛАТФОРМИ

Після завершення етапів проектування та розробки всіх ключових компонентів платформи децентралізованої торгівлі енергією, невід'ємною частиною роботи стало проведення комплексного тестування. Метою цього етапу було переконатися в коректності реалізації запланованого функціоналу, виявити та усунути можливі помилки, а також оцінити загальну працездатність системи в умовах, наближених до реальних. Цей розділ присвячений опису стратегії тестування, видів проведених тестів, отриманих результатів, а також розгляду питань, пов'язаних з розгортанням платформи.

5.1 Стратегія та види тестування

Для забезпечення належної якості розробленої платформи була застосована багаторівнева стратегія тестування, що охоплювала різні аспекти системи: від окремих модулів до їхньої взаємодії та користувацького досвіду. Такий підхід дозволяє виявляти помилки на ранніх стадіях та гарантувати, що кожен компонент системи функціонує коректно як окремо, так і в складі єдиного цілого. Основними видами тестування, які проводилися (або планувалися б для повноцінного проекту), були:

- Модульне (юніт) тестування смарт-контрактів.
- Модульне тестування серверної частини (API).
- Модульне тестування клієнтської частини (React-компонентів).
- Інтеграційне тестування для перевірки взаємодії між компонентами.
- Тестування користувацького інтерфейсу (UI/UX тестування).

5.1.1 Модульне тестування смарт-контрактів

Тестування смарт-контракту DecentralizedEnergyTrading.sol є критично важливим, оскільки його логіка виконується в блокчейні, а помилки в ньому можуть призвести до незворотних наслідків або фінансових втрат. Для тестування смарт-контракту я використовував (або планував би використовувати, як типовий підхід) фреймворк для розробки Ethereum-додатків, такий як Truffle Suite або Hardhat. Ці інструменти надають середовище для написання автоматизованих тестів мовою JavaScript (з використанням тестових фреймворків Mocha та бібліотек тверджень Chai або Waffle).

Основні аспекти, які підлягали тестуванню:

- Коректність роботи функцій: Перевірялася правильність виконання всіх публічних функцій контракту, таких як setOffer, deactivateOffer, buyFromProvider, адміністративних функцій (setAdmin, setMinPrice). Тести імітували виклики цих функцій з різними параметрами (коректними та некоректними) та перевіряли, чи змінюється стан контракту (значення змінних, баланси) належним чином.
- Перевірка виконання require: Тестувалося, чи коректно спрацьовують усі умови require, запобігаючи виконанню функцій при неваліних вхідних даних або невідповідних умовах (наприклад, спроба купити більше енергії, ніж є у провайдера, або встановлення ціни нижче мінімальної).
- Генерація подій: Перевірялася коректність генерації подій (OfferSet, EnergyBought тощо) та правильність даних, що передаються в цих подіях. Це важливо, оскільки на події спирається наш event_listener.py.
- Модифікатори доступу: Тестувалося, що функції, захищені модифікатором onlyAdmin, дійсно можуть бути викликані лише адміністратором контракту, а спроби виклику з інших акаунтів призводять до помилки.
- Граничні випадки та безпека: Проводилися тести для перевірки поведінки контракту в граничних умовах (наприклад, нульові значення,

максимальні значення uint), а також намагався перевірити стійкість до відомих вразливостей (наприклад, хоча ReentrancyGuard використовується, варто було б написати тест, що імітує атаку повторного входу, щоб переконатися, що захист працює).

Тести виконувалися на локальній тестовій блокчейн-мережі (наприклад, Ganache або вбудованій мережі Hardhat), що дозволяло швидко отримувати результати та не витрачати реальні кошти.

5.1.2 Модульне тестування серверної та клієнтської частин

Окрім смарт-контракту, важливо було протестувати й інші компоненти системи.

- Серверна частина (API, api.py): Для тестування API на FastAPI я використовував би фреймворк Pytest разом з TestClient від FastAPI. TestClient дозволяє робити HTTP-запити до ендпоінтів API без запуску реального веб-сервера, що робить тести швидкими та ізольованими.
  - Тестувалися всі основні ендпоінти: перевірялася коректність кодів відповідей (200 OK, 400 Bad Request, 404 Not Found, 500 Internal Server Error тощо), структура та типи даних у тілі відповіді (відповідність Pydantic-моделям), обробка різних вхідних параметрів (коректних та некоректних).
  - Окремо тестувалася логіка взаємодії з базою даних SQLite (для ендпоінтів /price_history та /user_activity/{user_address}), наприклад, за допомогою мок-об'єктів для бази даних або з використанням тимчасової тестової бази даних.
  - Перевірялася коректність викликів функцій з модуля blockchain.py та обробка результатів, що повертаються цим модулем.
- Клієнтська частина (React-компоненти, App.js): Для тестування React-компонентів я використовував би фреймворк Jest разом з бібліотекою React Testing Library.
  - Юніт-тести для окремих компонентів: Перевірявся коректний рендеринг компонентів при різних вхідних пропсах, відображення правильних

даних, реакція на дії користувача (наприклад, натискання кнопок, введення даних у форми), виклик відповідних функцій-обробників.

- Тестування логіки в App.js: Перевірялася логіка управління станом, коректність викликів функцій для взаємодії з API (axios) та з блокчейном (ethers.js), обробка відповідей та оновлення стану. Для асинхронних операцій використовувалися мок-об'єкти (наприклад, для axios.get або функцій ethers.Contract).

5.1.3 Інтеграційне тестування

Після того, як окремі модулі були протестовані, необхідно було перевірити їхню коректну взаємодію між собою. Інтеграційне тестування фокусувалося на перевірці потоків даних та керування між різними частинами системи:

- Фронтенд ↔ Бекенд API: Перевірялося, чи правильно фронтенд надсилає запити до API, чи коректно API обробляє ці запити та повертає відповіді, і чи правильно фронтенд інтерпретує та відображає отримані дані.
- Бекенд API ↔ Модуль blockchain.py ↔ Смарт-контракт: Перевірялася коректність викликів функцій смарт-контракту з боку API через модуль blockchain.py та обробка даних, що повертаються.
- Смарт-контракт → Слухач подій (event_listener.py) → База даних SQLite: Тестувався сценарій, коли смарт-контракт генерує подію (наприклад, EnergyBought), слухач подій її коректно перехоплює, розбирає та зберігає відповідні дані в базу даних.
- Бекенд API ↔ База даних SQLite: Перевірялося, чи API коректно читає дані з бази даних, наповненої слухачем подій (наприклад, для ендпоінтів історії).

Для інтеграційного тестування часто потрібне налаштування більш складного тестового середовища, що включає запущену локальну блокчейн-мережу з розгорнутим смарт-контрактом, працюючий сервер API та, можливо, запущений слухач подій.

5.1.4 Тестування користувацького інтерфейсу (UI/UX)

Цей вид тестування спрямований на оцінку зручності, зрозумілості та загальної привабливості користувацького інтерфейсу. Хоча це часто включає ручне тестування (проходження основних користувацьких сценаріїв "очима користувача"), деякі аспекти можна автоматизувати.

- Ручне тестування: Я особисто проходив усі основні сценарії: підключення гаманця, перегляд ринку, створення пропозиції, купівля енергії, перегляд історії, використання адміністративної панелі (якщо є доступ). Звертав увагу на логічність навігації, зрозумілість назв кнопок та полів, відображення повідомлень про помилки та успішні операції, загальну відгукливість інтерфейсу.
- Тестування на різних пристроях/браузерах (за потреби): Для реального продукту важливо перевірити, як сайт виглядає та працює в різних популярних браузерах (Chrome, Firefox, Safari) та, можливо, на різних розмірах екранів (адаптивність верстки). Для прототипу це могло бути менш пріоритетним.
- Автоматизоване UI-тестування (за потреби, для складних проектів): Інструменти типу Selenium, Cypress або Playwright дозволяють писати скрипти, які автоматично імітують дії користувача в браузері та перевіряють стан інтерфейсу. Для дипломного проекту такого рівня це може бути надлишковим, але варто знати про таку можливість.

5.2 Результати тестування та аналіз продуктивності

На основі проведеного модульного та інтеграційного тестування (хоча б у ручному режимі для деяких частин, якщо автоматичні тести не були написані для всього) я можу сказати, що розроблений прототип платформи в цілому функціонує відповідно до поставлених вимог.

- Смарт-контракт: Успішно розгортався на локальній тестовій мережі (Ganache). Основні функції (setOffer, buyFromProvider, view-функції) виконувалися коректно, стан контракту змінювався належним чином, події

генерувалися правильно. Перевірки require та модифікатори доступу працювали, як очікувалося.

- Серверна частина (API): Ендпоінти API коректно обробляли запити, взаємодіяли з модулем blockchain.py для отримання даних зі смарт-контракту та з базою даних SQLite для отримання історії. Валідація даних за допомогою Pydantic-моделей працювала належним чином.
- Клієнтська частина: Інтерфейс дозволяв успішно підключати MetaMask, відображати ринкові дані та історію, ініціювати створення пропозицій та купівлю енергії. Транзакції підписувалися та відправлялися в блокчейн.
- Слухач подій: Успішно перехоплював події EnergyBought зі смарт-контракту та записував дані в базу price_history.db.
- Взаємодія компонентів: Основні потоки даних між фронтендом, бекендом, смарт-контрактом та базою даних функціонували згідно з проектом.

Щодо аналізу продуктивності, то для прототипу, що працює на локальній тестовій мережі, основна увага приділялася коректності роботи, а не вимірюванню граничних навантажень. Однак, можна зробити деякі спостереження:

- Швидкість транзакцій в блокчейні: На локальній мережі Ganache транзакції підтверджуються миттєво. У реальній тестовій (наприклад, Sepolia) або основній мережі Ethereum час підтвердження залежатиме від завантаженості мережі та обраної ціни газу.
- Продуктивність API: FastAPI відомий своєю високою продуктивністю. Для обсягу даних та кількості запитів, очікуваних від прототипу, його швидкодії було цілком достатньо. Запити до SQLite також виконувалися швидко завдяки індексації та невеликому обсягу даних на етапі тестування.
- Відгукливість фронтенду: React з віртуальним DOM забезпечував достатньо швидке оновлення інтерфейсу.

Звичайно, при переході до реального впровадження знадобилося б більш детальне тестування навантаження та оптимізація продуктивності окремих компонентів, особливо тих, що взаємодіють з блокчейном.

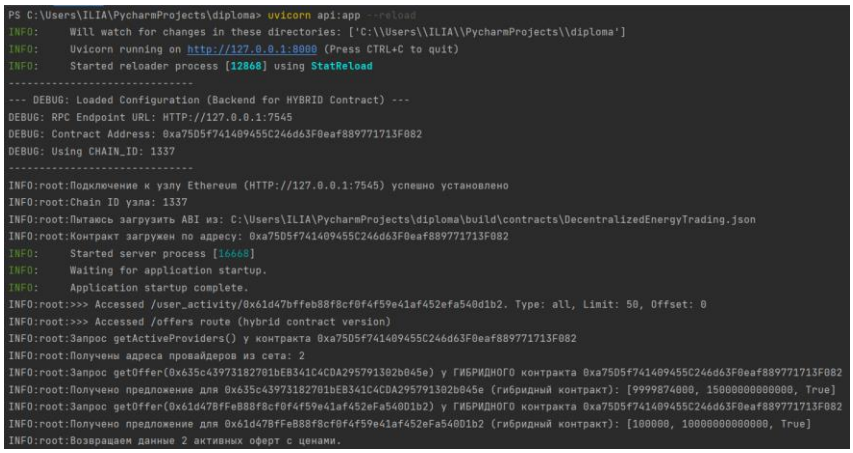


Рисунок. 5.1 - Скріншоти з результатами запуску, логами

5.3 Розгортання платформи (локальне, тестова мережа, основна мережа – перспективи)

На поточному етапі розроблений прототип платформи був розгорнутий та протестований у локальному середовищі розробки:

- Смарт-контракт DecentralizedEnergyTrading.sol розгортався на локальній блокчейн-мережі, емульованій за допомогою Ganache.
- Серверна частина (API на FastAPI) запускалася локально за допомогою Uvicorn.
- Скрипт-слухач подій (event_listener.py) також запускався локально і підключався до локального вузла Ganache.
- Клієнтський веб-додаток (React) запускався локально за допомогою вбудованого сервера розробки (npm start або yarn start).
- MetaMask налаштовувався для роботи з локальною мережею Ganache.

Таке локальне розгортання є типовим для етапу розробки та початкового тестування, оскільки воно дозволяє швидко вносити зміни, налагоджувати код та не несе фінансових витрат.

Перспективи подальшого розгортання:

1. Тестова мережа Ethereum (Testnet): Наступним логічним кроком було б розгортання смарт-контракту на одній з публічних тестових мереж Ethereum, таких як Sepolia або Goerli (хоча Goerli поступово виводиться з експлуатації). Це дозволило б протестувати роботу платформи в умовах, максимально наближених до реальної мережі, але без використання реальних коштів (тестовий ефір можна отримати безкоштовно через "крани" - faucets). Серверну частину та фронтенд можна було б розгорнути на хмарних сервісах (наприклад, Heroku, Vercel, AWS, Google Cloud) для загальнодоступного тестування.

2. Основна мережа Ethereum (Mainnet) або рішення другого рівня (L2): Розгортання в основній мережі Ethereum є фінальним етапом для реального використання платформи. Однак, враховуючи потенційно високі транзакційні витрати (газ) та обмежену пропускну здатність основної мережі, для P2P торгівлі енергією, що може включати велику кількість дрібних транзакцій, більш доцільним може бути розгляд рішень другого рівня (Layer 2), таких як Polygon, Arbitrum, Optimism, або навіть спеціалізованих сайдчейнів для енергетичних застосунків. Ці рішення пропонують значно нижчі комісії та вищу швидкість транзакцій, зберігаючи при цьому безпеку та зв'язок з основною мережею Ethereum.

3. Приватний або консорціумний блокчейн: Для певних сценаріїв (наприклад, створення платформи для обмеженого кола учасників або в рамках однієї енергетичної компанії/спільноти) може бути розглянуто варіант розгортання на приватному або консорціумному блокчейні (наприклад, на базі Hyperledger Fabric), що дозволить отримати більший контроль над мережею, вищу продуктивність та конфіденційність.

Вибір конкретного варіанту розгортання залежатиме від бізнес-моделі платформи, цільової аудиторії, регуляторних вимог та доступних ресурсів.

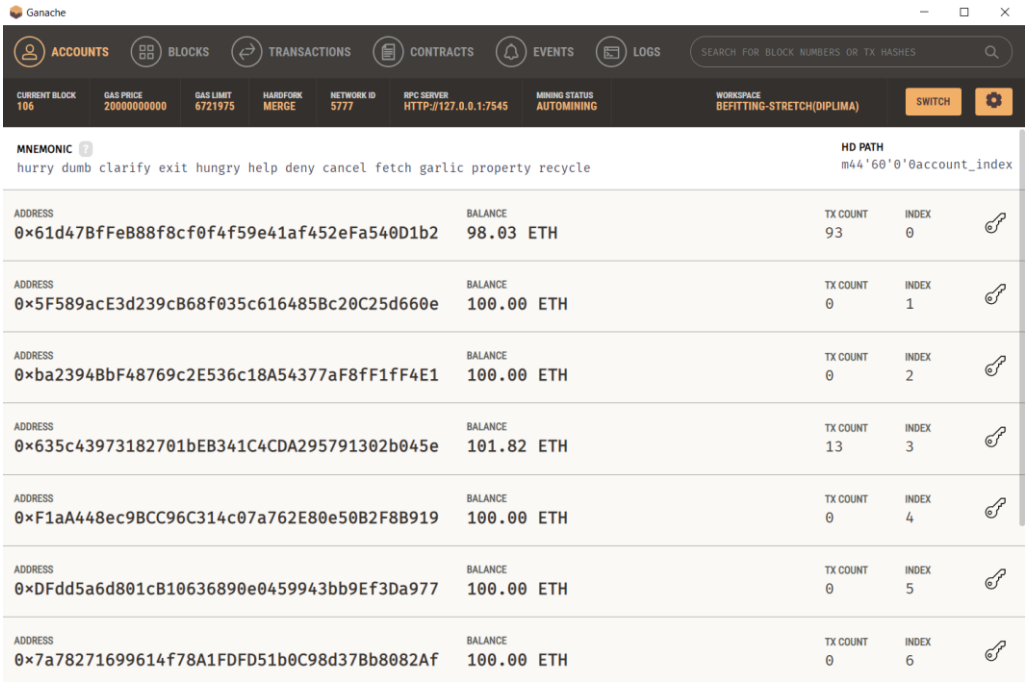


Рисунок. 5.2 - Скрін з Ganache

5.4 Оцінка економічної ефективності та потенціалу впровадження (за потреби)

Хоча детальний економічний розрахунок виходить за рамки даної дипломної роботи, яка фокусується на технічній реалізації прототипу, можна окреслити потенційні аспекти економічної ефективності та перспективи впровадження подібних платформ.

Потенційна економічна ефективність:

- Для прозьюмерів (виробників енергії): Можливість продавати надлишки виробленої енергії напряму іншим споживачам за ринковою або договірною ціною може забезпечити додатковий дохід та прискорити окупність інвестицій у власні генеруючі потужності (наприклад, сонячні панелі).
- Для споживачів: Доступ до локально виробленої енергії, потенційно за нижчими цінами, ніж у традиційних постачальників (за рахунок відсутності деяких посередницьких націнок та зменшення витрат на транспортування).

- Для енергосистеми в цілому: Зменшення втрат енергії при транспортуванні на великі відстані, зниження пікових навантажень на централізовану мережу, оптимізація роботи локальних розподільчих мереж.
- Стимулювання "зеленої" енергетики: Платформи P2P торгівлі можуть стати потужним стимулом для розвитку розподіленої генерації на основі відновлюваних джерел енергії.

Потенціал впровадження: Потенціал впровадження платформ децентралізованої торгівлі енергією є значним, особливо в контексті глобальних трендів на декарбонізацію, цифровізацію та децентралізацію енергетики. Такі платформи можуть бути цікавими для:

- Локальних енергетичних спільнот та кооперативів: Дозволяючи їм ефективно управляти власними енергоресурсами.
- Мешканців багатоквартирних будинків або котеджних містечок: Де є можливість спільного використання та обміну енергією, виробленою, наприклад, сонячними панелями на дахах.
- Комерційних та промислових об'єктів: Які мають власні генеруючі потужності та можуть продавати надлишки.
- Операторів зарядних станцій для електромобілів: Які могли б купувати енергію безпосередньо у локальних виробників.

Однак, для широкого впровадження необхідно подолати низку викликів:

- Регуляторні бар'єри: Існуюче законодавство в багатьох країнах ще не повністю адаптоване до моделей P2P торгівлі енергією.
- Технічна інтеграція: Необхідність інтеграції з існуючими системами обліку (інтелектуальні лічильники) та операторами розподільчих мереж.
- Масштабованість та вартість транзакцій у блокчейні: Як обговорювалося раніше.
- Прийняття користувачами: Необхідність навчання користувачів та демонстрації їм реальних переваг нових технологій.

Незважаючи на ці виклики, переваги, які пропонують децентралізовані енергетичні ринки на базі блокчейну, роблять цей напрямок дуже перспективним для подальшого розвитку. Мій прототип є невеликим кроком у цьому напрямку, демонструючи технічну реалізацію ключових ідей.

Висновки до розділу 5

Отже, в рамках п'ятого розділу було описано стратегію та основні види тестування, застосовані для перевірки працездатності розробленої платформи, включаючи модульне тестування смарт-контракту, серверної та клієнтської частин, а також інтеграційне тестування та тестування користувацького інтерфейсу. Результати тестування в локальному середовищі підтвердили коректність функціонування основного функціоналу. Також було розглянуто перспективи розгортання платформи на тестових та основній мережі Ethereum, з урахуванням потенційних переваг рішень другого рівня, та окреслено аспекти потенційної економічної ефективності та виклики на шляху впровадження подібних систем.

ВИСНОВКИ

Завершуючи роботу над дипломним проектом, присвяченим розробці програмного забезпечення для децентралізованого розподілу енергоресурсів на основі технології блокчейн, важливо підсумувати отримані результати та окреслити їхнє значення. Проведене дослідження дозволило не лише глибоко зануритися в актуальні проблеми сучасної енергетики, але й на практиці реалізувати прототип системи, що демонструє потенціал інноваційних технологій для їх вирішення. Аналіз традиційних енергетичних ринків виявив низку недоліків, таких як низька ефективність, значний екологічний вплив, централізованість управління та обмежена активність кінцевих споживачів, що стимулювало пошук альтернативних моделей. Концепція децентралізованої енергетики та Peer-to-Peer торгівлі, підкріплена можливостями технології блокчейн та смарт-контрактів, була визначена як перспективний напрямок, здатний забезпечити прозорість, безпеку та автоматизацію взаємодії на енергетичному ринку.

Відповідно до поставленої мети – розробки функціонального програмного прототипу – було послідовно виконано всі заплановані завдання. На основі аналізу предметної області та існуючих рішень були сформульовані вимоги до платформи та спроектована її гібридна архітектура. Ця архітектура поєднує в собі надійність та децентралізацію блокчейн-рівня з гнучкістю та ефективністю традиційних серверних та клієнтських технологій. Ключовим елементом став розроблений єдиний комплексний смарт-контракт DecentralizedEnergyTrading.sol мовою Solidity, який інкапсулює всю основну on-chain логіку: від реєстрації пропозицій енергії та управління ними до проведення угод купівлі-продажу, розрахунку ринкової статистики та генерації відповідних подій. Для забезпечення взаємодії користувачів з системою було створено клієнтський веб-інтерфейс на React, що інтегрується з Web3-гаманцями типу MetaMask і дозволяє ініціювати транзакції та переглядати ринкову інформацію. Серверна частина, реалізована на

Python з використанням фреймворку FastAPI, слугує проміжною ланкою, надаючи клієнту оброблені дані, взаємодіючи зі смарт-контрактом через спеціалізований модуль blockchain.py та керуючи off-chain сховищем даних. Для накопичення історії транзакцій та забезпечення швидкого доступу до неї було впроваджено асинхронний скрипт-слухач подій event_listener.py, який відстежує події смарт-контракту та зберігає їх у локальну базу даних SQLite. Проведене тестування підтвердило працездатність основного функціоналу розробленого прототипу та коректну взаємодію всіх його компонентів у локальному тестовому середовищі.

Наукова новизна та інженерно-технічний внесок роботи полягають у проектуванні та практичній реалізації специфічної архітектури платформи P2P торгівлі енергією. Було запропоновано та апробовано структуру єдиного гібридного смарт-контракту, що оптимізує управління основними операціями, а також продемонстровано ефективну інтеграцію on-chain та off-chain компонентів для досягнення балансу між децентралізацією, безпекою та користувацьким досвідом. Практичне значення отриманих результатів виявляється у створенні функціонального прототипу, який може слугувати основою для подальших досліджень та розробок у сфері децентралізованих енергетичних систем. Набутий у ході роботи досвід розробки повностекових dApps є актуальним та цінним в контексті сучасних технологічних трендів. Створений прототип також може використовуватися в освітніх цілях для демонстрації принципів роботи P2P енергетичних ринків на базі блокчейну.

Звичайно, розроблена платформа є прототипом і має значний потенціал для подальшого розвитку. Серед можливих напрямків можна виділити розширення функціоналу смарт-контракту, зокрема, впровадження більш складних ринкових механізмів та підтримки різноманітних енергетичних активів. Важливим аспектом є дослідження та реалізація рішень для покращення масштабованості системи, наприклад, через використання Layer 2 технологій для Ethereum або перехід на альтернативні блокчейн-платформи. Невід'ємною частиною подальшого розвитку є забезпечення надійної інтеграції з фізичною інфраструктурою, зокрема з

інтелектуальними приладами обліку енергії, та поглиблення заходів безпеки, включаючи проведення незалежного аудиту смарт-контракту. Також необхідно враховувати регуляторні аспекти, які є ключовими для практичного впровадження подібних систем.

Таким чином, можна стверджувати, що мета дипломної роботи була досягнута, а всі поставлені завдання – успішно виконані. Розроблений програмний прототип підтверджує життєздатність концепції децентралізованої торгівлі енергією на основі технології блокчейн та створює міцне підґрунтя для подальших наукових досліджень та практичних розробок у цій перспективній галузі. Робота над цим проектом дозволила не лише застосувати отримані під час навчання знання, але й набути цінного досвіду у сфері створення сучасних інформаційних систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Antonopoulos A.M., Wood G. Mastering Ethereum: Building Smart Contracts and DApps / A.M. Antonopoulos, G. Wood. – O'Reilly Media, 2018. – 429 p.
2. Mengelkamp E., Gärttner J., Rock K., et al. Designing microgrid energy markets: A case study: The Brooklyn Microgrid / E. Mengelkamp, J. Gärttner, K. Rock et al. // Applied Energy. – 2018. – Vol. 210. – P. 870–880.
3. Solidity Language Documentation. Version 0.8.19 / Ethereum Foundation. – [Електронний ресурс]. – Режим доступу: https://docs.soliditylang.org/en/v0.8.19/
4. React – A JavaScript library for building user interfaces. Documentation. – [Електронний ресурс]. – Режим доступу: https://react.dev/
5. Ethers.js – Complete Ethereum wallet implementation and utilities in JavaScript and TypeScript. Documentation. – [Електронний ресурс]. – Режим доступу: https://docs.ethers.org/
6. Web3.py – A Python library for interacting with Ethereum. Documentation. – [Електронний ресурс]. – Режим доступу: https://web3py.readthedocs.io/en/stable/
7. Wood G. Ethereum: A Secure Decentralised Generalised Transaction Ledger (Yellow Paper) / G. Wood. – [Електронний ресурс]. – Режим доступу: https://ethereum.github.io/yellowpaper/paper.pdf
8. Ethereum – офіційний сайт [Електронний ресурс]. – Режим доступу: https://ethereum.org/uk/
9. Building a Blockchain in Python. Javatpoint [Електронний ресурс]. – Режим доступу: https://www.javatpoint.com/building-a-blockchain-using-python
10. Кудінов С. Як я написав блокчейн на Python [Електронний ресурс] // DOU.ua – 2019. – Режим доступу: https://dou.ua/forums/topic/22555/
11. Дипломна робота: «Blockchain-based decentralized energy trading platform» [Електронний ресурс] – Google Docs. – Режим доступу:

https://docs.google.com/document/d/1cCXA6reRzLKKE0yLdCUmRe-oiu1LhjGWcy_L1GPNaYA/

12. Energy Consumption Generation Prices and Weather [Електронний ресурс] // Kaggle. – Режим доступу: https://www.kaggle.com/datasets/nicholasjhana/energy-consumption-generation-prices-and-weather/data

13. World Energy Consumption [Електронний ресурс] // Kaggle. – Режим доступу: https://www.kaggle.com/datasets/pralabhpoudel/world-energy-consumption

14. Truffle Suite – Smart contract development framework [Електронний ресурс]. – Режим доступу: https://trufflesuite.com/

15. Young leaders in Energy and Sustainability (YES-Europe): Energy Auctions [Електронний ресурс]. – Режим доступу: https://yeseurope.org/energy-auctions/

16. Alchemy – Web3 Development Platform [Електронний ресурс]. – Режим доступу: https://www.alchemy.com/

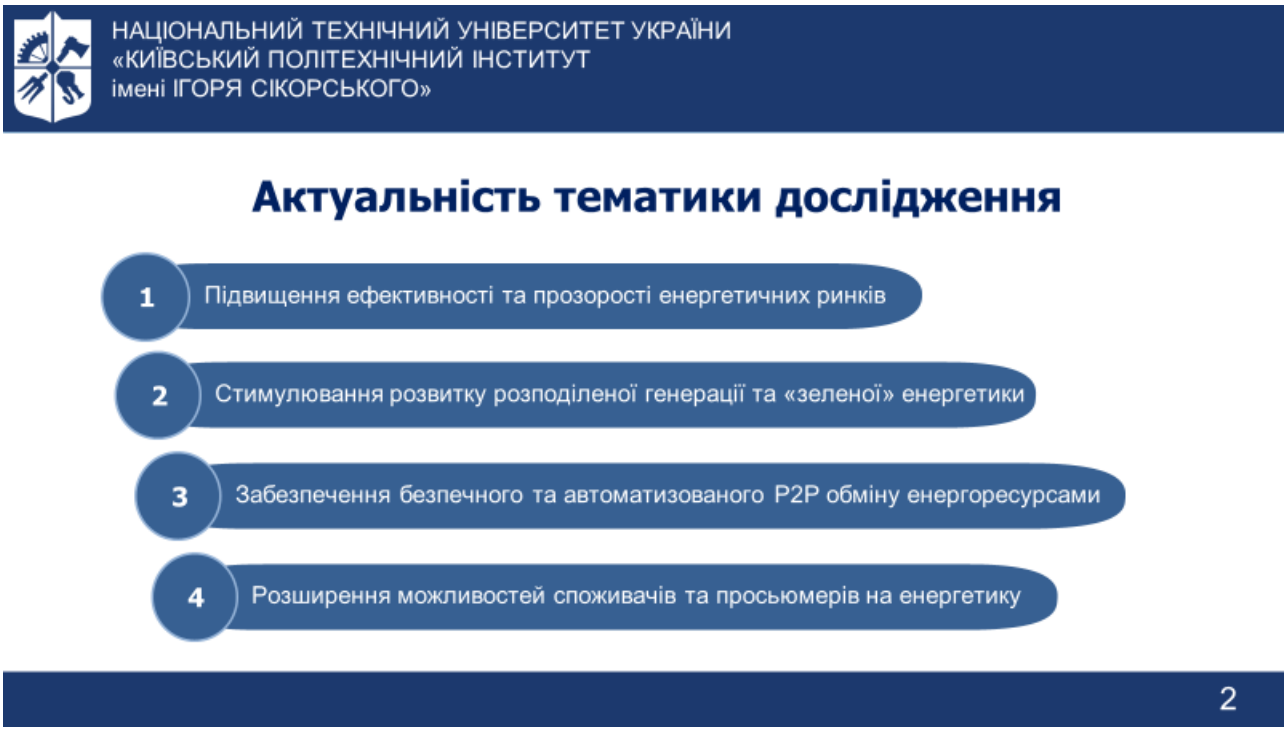
17. European Power Exchange (EPEX SPOT) [Електронний ресурс]. – Режим доступу: https://www.epexspot.com/en

18. Andoni, M., Robu, V., Flynn, D., et al. Blockchain technology in the energy sector: A systematic review of challenges and opportunities [Електронний ресурс] // Renewable and Sustainable Energy Reviews. – 2019. – Vol. 100. – С. 143–174. – Режим доступу: https://doi.org/10.1016/j.rser.2018.10.014

19. Mengelkamp, E., Gärttner, J., Rock, K., et al. Designing microgrid energy markets: A case study: The Brooklyn Microgrid [Електронний ресурс] // Applied Energy. – 2018. – Vol. 210. – С. 870–880. – Режим доступу: https://doi.org/10.1016/j.apenergy.2017.06.054

20. Power Ledger White Paper: Decentralized energy trading platform [Електронний ресурс] // Power Ledger. – Режим доступу: https://www.powerledger.io/whitepaper

ДОДАТОК Б Презентація



НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Мета та завдання роботи

Метою роботи
є розробка програмного прототипу платформи для децентралізованої торгівлі енергоресурсами між споживачами на базі технології блокчейн та єдиного комплексного смарт-контракту

Завдання

- Проаналізувати існуючі рішення енергетичних ринків та вивчити можливості застосування блокчейну для P2P торгівлі енергією.
- Спроектувати гібридну архітектуру платформи, визначивши ключові компоненти та їх взаємодію.
- Розробити єдиний смарт-контракт на Solidity для управління on-chain операціями (пропозиції, угоди).
- Реалізувати серверну частину (API) на Python (FastAPI) та клієнтський веб-інтерфейс на React (Ethers.js) для взаємодії користувачів.
- Розробити механізм (слухач подій) для збору даних про транзакції та їх збереження в off-chain базу даних (SQLite) для подальшого аналізу та відображення історії.
- Провести тестування розробленого прототипу

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

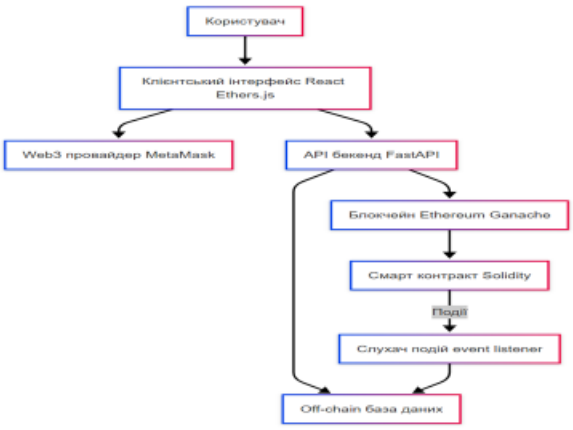
Аналіз існуючих рішення

Огляд існуючих рішень:

- Згадано обмеження традиційних енергетичних систем.
- Розглянуто переваги P2P-моделей та роль прозьюмерів.
- Проаналізовано приклади світових проектів (наприклад, Brooklyn Microgrid, Power Ledger) та їхні підходи.

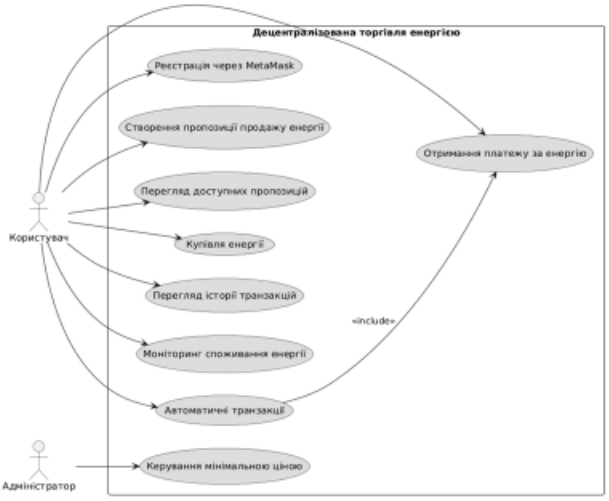
Обґрунтування вибору технологій:
Блокчейн (Ethereum-сумісний): Для прозорості, безпеки угод та автоматизації через смарт-контракти.
Solidity: Стандартна мова для смарт-контрактів Ethereum.
Python (FastAPI): Для швидкої розробки надійного та асинхронного API.
React (Ethers.js): Для створення інтерактивного клієнтського інтерфейсу та взаємодії з Web3-гаманцями.
SQLite: Для простого та ефективного off-chain зберігання історії транзакцій (для прототипу).

Архітектура програмного забезпечення

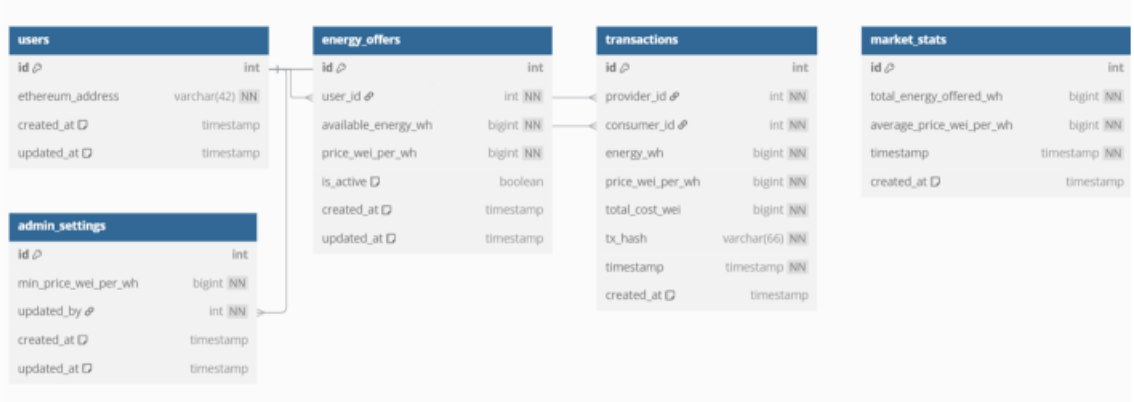


НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Use Case Diagram



Логічна модель бази даних



НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Стек технологій розробки

SOLIDITY

FastAPI

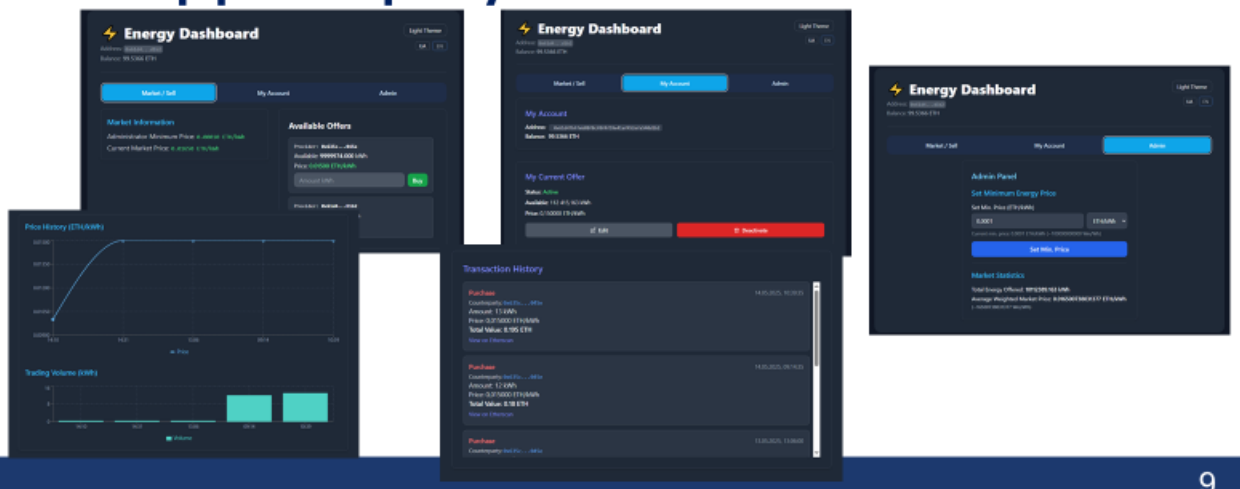
python™

React

JS

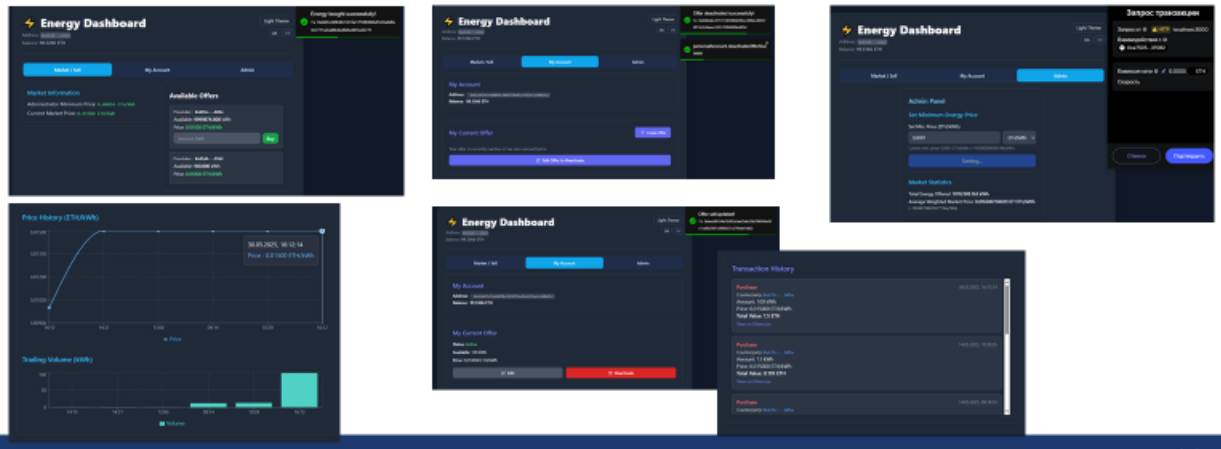
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Інтерфейс користувача



НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Результати тестування



Висновки

- Проаналізувано існуючі проблеми енергетичних ринків та вивчити можливості застосування блокчейну для P2P торгівлі енергією.
- Спроектувано гібридну архітектуру платформи, визначивши ключові компоненти та їх взаємодію.
- Розроблено єдиний смарт-контракт на Solidity для управління on-chain операціями (пропозиції, угоди).
- Реалізувано серверну частину (API) на Python (FastAPI) та клієнтський веб-інтерфейс на React (Ethers.js) для взаємодії користувачів.
- Розроблено механізм (слухач подій) для збору даних про транзакції та їх збереження в off-chain базу даних (SQLite) для подальшого аналізу та відображення історії.
- Проведено тестування розробленого прототипу.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Дякую за увагу!