Міністерство освіти і науки України

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти перший (бакалаврський)

Застосунок для проведення онлайн-аукціону із використанням технології блокчейн

(тема)

Виконав:
здобувач 4 року навчання,
групи КІУКІ-21-4
Дмитро СИДОРОВ
(власне ім’я, прізвище)

Спеціальність
123 «Комп’ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма
Комп’ютерна інженерія
(повна назва освітньої програми)

Керівник: ст. викл. Олександр ФОМІЧОВ
(посада, власне ім’я, прізвище)

Допускається до захисту

Завідувач кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (власне ім’я, прізвище)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти перший (бакалаврський)

Спеціальність 123 «Комп’ютерна інженерія»
(код і повна назва)

Тип програми Освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Комп’ютерна інженерія
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
“____” _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Сидорову Дмитру Володимировичу
(прізвище, ім’я, по батькові)

1. Тема роботи Застосунок для проведення онлайн-аукціону із використанням технології блокчейн

затверджена наказом по університету від “26” травня 2025 р. № 424 Ст

2. Термін подання здобувачем роботи до екзаменаційної комісії 17 червня 2025

3. Вхідні дані до роботи

1) документація мови програмування JavaScript

2) інтегроване середовище Visual Studio Code

4. Перелік питань, що потрібно опрацювати у роботі

1) аналіз предметної області

2) принципи ооп

3) програмна реалізація застосунку онлайн-аукціону

4) інструкція користувача

5) висновки

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій

Слайд-презентація – 11 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Аналіз проблеми та огляд існуючих рішень	27.05.25-28.05.25	
2	Формування переліку вимог до програми	29.05.25-30.05.25	
3	Вибір технології розробки та	31.05.25-02.06.25	
	інструментальних засобів		
4	Розробка та тестування програми	03.06.25-09.06.25	
5	Оформлення матеріалів кваліфікаційної роботи	10.06.25-13.06.25	
6	Подання кваліфікаційної роботи керівнікові та	14.06.25-15.06.25	
	її попередній захист		
7	Подання кваліфікаційної роботи на	16.06.25-17.06.25	
	рецензування		

Дата видачі завдання " 26 " травня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ ст. викл. Олександр ФОМІЧОВ
(підпис) (посада, власне ім'я, прізвище)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 78 с., 11 рис., 2 дод., 20 джерел.

БЛОКЧЕЙН, СМАРТ-КОНТРАКТ, JAVASCRIPT, REACT, АУКЦІОН, ФРОНТЕНД, БЕКЕНД, METAMASK, SOLIDITY.

Метою кваліфікаційної роботи є створення децентралізованої платформи для проведення онлайн-аукціонів. Можливість авторизуватися через криптогаманець, створювати лоти, переглядати їх, приймати участь у торгах.

У ході виконання кваліфікаційної роботи було розроблено децентралізовану веб-платформу для проведення онлайн аукціонів з використанням технології блокчейн та смарт-контрактів Ethereum. Реалізоване рішення поєднує в собі фроненд на базі React, бекенд частину на Node.js та смарт-контракт розроблений на Solidity. Фронтенд частина для взаємодії з користувачем, бекенд для авторизаціїі, зберігання даних та взаємодії зі смарт-контрактом та смарт-контракт, який відповідає за логіку проведення торгів та транзакції. Авторизація відбувається через підпис повідомлення за допомоги криптографічного гаманця MetaMask. Усі транзакції фіксуються на блокчейні, що гарантує прозорість процесу торгів, незмінність даних та безпеку взаємодії між користувачами. Розроблена система була протестована та за результатами є повністю працюючою.

ABSTRACT

Bachelor’s thesis 78 pages, 11figures, 2 appendices, 20 sources.

BLOCKCHAIN, SMART CONTRACT, JAVASCRIPT, REACT, AUCTION, FRONTEND, BACKEND, METAMASK, SOLIDITY.

The major goal of this thesis is to create a decentralized platform for conducting online auctions. The ability to log in via a crypto wallet, create lots, view them, and participate in bidding.

In order to qualification work a decentralized web platform for conducting online auctions using blockchain technology and Ethereum smart contracts was developed. The implemented solution combines a frontend based on React, a backend part on Node.js and a smart contract developed on Solidity. The frontend part for interaction with the user, the backend for authorization, data storage and interaction with the smart contract and the smart contract, which is responsible for the logic of conducting trades and transactions. Authorization occurs through the signature of the message using the MetaMask cryptographic wallet. All transactions are recorded on the blockchain, which guarantees the transparency of the trading process, data immutability and security of interaction between users. The developed system has been tested and, according to the results, is fully operational.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

dApp – децентралізований додаток (англ. decentralized application)

DeFi – децентралізовані фінанси (англ. Decentralized Finance)

EVM – віртуальна машина, яка виконує смарт-контракти у блокчейн мережі Ethereum (англ. Ethereum Virtual Machine)

eWASM – нова віртуальна машина Ethereum (англ. Ethereum WebAssembly)

ВСТУП

У сучасному світі цифрова трансформація охопила майже всі сфери людського життя, від фінансів та логістики до охорони здоров’я. Однією з самих перспективних та революційних технологій став блокчейн. На його основі була створена нова технологія смарт-контракти, здатні автоматизувати виконання згод між сторонами без необходності посерднеків. Блокчейн та смарт-контракти можуть змінити принципи взаємодії у найрізноманітніших сферах, автоматизація бізнес-процесів та покращення безпеки транзакцій до створення нових моделей довіри у цифровій економіці. Їх використання відкриває можливості більш прозорим, надійним та ефективним системам керування, де людський фактор мінімальний, а довіра заснована не на репутації, а на алгорітмах.

У зв’язку зі стріким розвитком технологій Web3, децентралізованих фінансів, токенізаціїї активів та цифрової ідентичності, актуальність блокчейн – систем та смарт-контрактів зростає з кожним роком. Потенціал цих технологій полягає не тільки в технічних інноваціях, а і в можливості переосмислення організаційних, правових та економічних моделях.

На фоні усього вище сказаного цікавим є використання блокчейну та смарт-контрактів у сфері аукціонної торговлі. Блокчейн забезпечить прозорість та незмінність даних, а смарт-контракти дозволять автоматизувати виконання умов торгів без посередників. Це робить можливим створення децентралізованих аукціонних платформ де дотримання усіх правил буде забезпечуватися кодом, а не людьми. Використання смарт-контрактів у онлайн аукціонах дозволяє висключити можливість втручання третіх осіб, збільшити довіру учасників, а також забезпечити автоматичну фіксацію ставок та чесне визначення переможця. Більш того такі системи мають потенціал для масштабування та можуть використовуватися у різноманітних галузях.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Поняття блокчейну та смарт контракти

Блокчейн – це побудований за певними правилами безперервний ланцюжок блоків, зберігаючий деяку інформацію. Зв’язок між блоками забезпечується тим, що окрім нумірації кожен блок містить свою хеш-суму та хеш-суму попереднього блоку і тоді зміна інформації у блоці змінить його хеш-суму [1]. Щоб виконалися правила побудови ланцюжка зміни хеш-суми потрібно записати до наступного блоку, що вже змінить його хеш-суму, при цьому попередні блоки будуть незмінні. Якщо після блоку який потрібно змінити вже був сформованний наступний блок, то зміни майже неможливі. Уся проблема в тому, що зазвичай копії ланцюжків зберігаються на великій кількості інших комп’ютерів, що незалежні один від одного. Через це технологія блокчейну дуже стійка до втручання третіх осіб.

Блокчейн працює за принципом розподіленого консенсусу, що значить дані у ланцюжці підтверджуються ні одним центральним сервером, а багатою кількістю вузлів (комп’ютерів) по всьому світу. Кожен вузол зберігає копію усього ланцюжка та приймає участь у перевірці нових даних. До того як нова інформації потрапить до блокчейну, вона повинна бути підтверджена за певним принципом [2]. Такі принципи і є механізми консенсусу (рисунок 1.1).

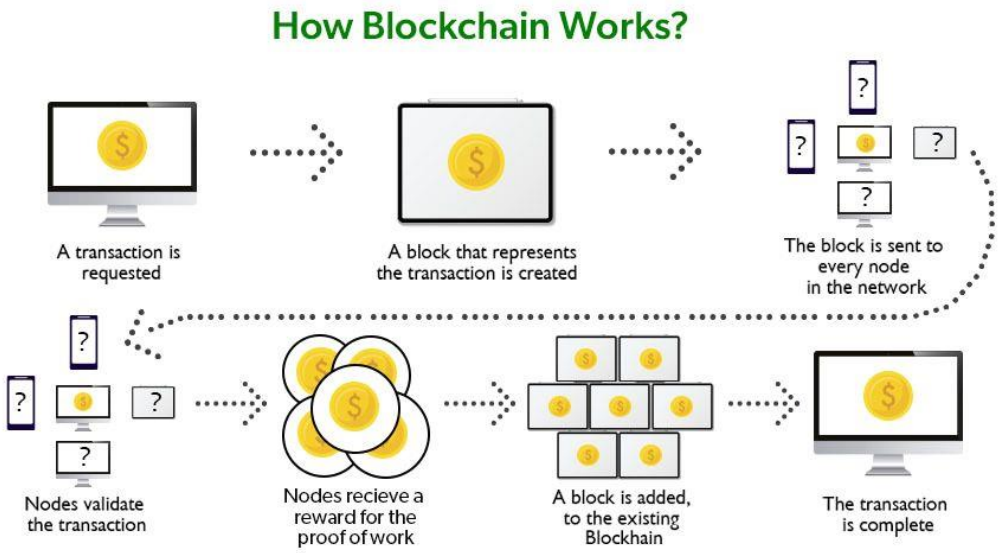

Рисунок 1.1 – Приклад роботи блокчейну

Існують різноманітні механізми консенсусу. Найвідоміші це Proof-of-Work та Proof-of-Stake . У Proof-of-Work учасники мережі вирішують складні математичні задачі, щоб мати право додати новий блок. У Proof-of-Stake нові блоки створюються валідаторами, які вибираються на основі належним ним токенам у мережі [3]. Окрім криптовалют блокчейн використовується у різних сферах: у логістиці, щоб відслідковувати ланцюжки поставок, у медицині для зберігання записів, в юрисприденції для автоматизації контрактів або навіть у політиці для проведення електронних виборів.

Смарт-контракт контракт який виконує сам себе а його умови напряму записані у коді. Цей код розгортається у блокчейні де надійно і незмінно записуються усі транзакції. Після того як контракт був розгорнут він автоматично забезпечує та виконує усі умови без необхідності третіх осіб, наприклад банку або правової системи, щоб прослідкувати за виконанням усіх поставлених умов [4].

Смарт-контракти це рішення, які працюють відповідно правилам які були прописані у коді, частіше такі правила називають “логікою” контракта.

Контракти відстежують блокчейн на виконання вказаних умов відомих як трігери і коли ці трігери виконуються контракт автоматично виконує відповідні дії. Наприклад якщо є контракт де вказано що до встановленої дати буде отримана деяка кількість криптовалюти і тоді кошти будуть переведені тому хто створив проєкт. Якщо цілі контракту не була досягнута то кошти повернуться до вкладчиків. Весь цей процес автоматизований завдяки смарт-контракту, тому сторонього втручання не потрібно та виконання усіх умов згоди грантується [5]. Смарт-контракти усуває потреби довіри між сторонами договіру оскільки усе прописано у самому незмінному коді. В особливості це корисно тоді, коли сторони не знають один одного або звичні механізми правового забезпечення багато коштують і непрактичні.

1.2 Мережі блокчейну

Ethereum децентралізована блокчейн платформа, призначена для запуску смарт-контрактів та децентралізованих додатків. На відміну від біткойна, який орієнтований тільки на передачу цифрової валюти, Ethereum створенний як більш гнучка система, яка дозволяє програмувати умови передачі цієї валюти. Технічно платформа це глобальний децентралізований комп’ютер, який складається з тисяч вузлів кожен з яких зберігає та підтримує одну й туж саму копію блокчейна [6]. В основі полягає концепція смарт-контрактів, які автоматично виконуються при заданих умовах. Ці контракти зберігаються у блокчейні та працюють без можливості втручання (рисунок 1.2).

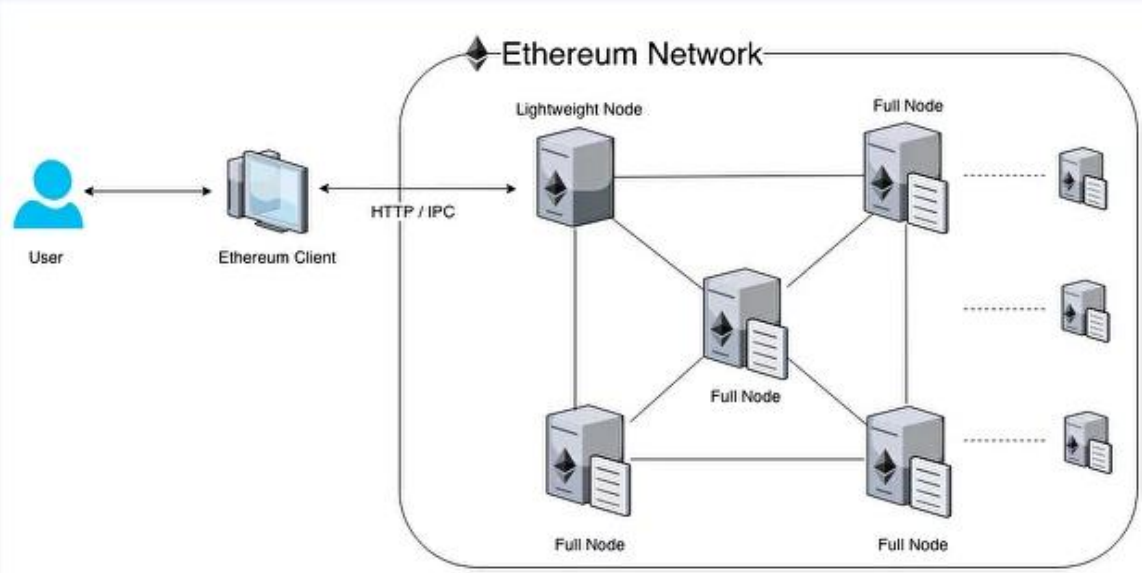

Рисунок 1.2 – Приклад мережі Ethereum

Кожна дія у мережі Ethereum потребує розрахунків, яка сплачується у одиницях яка називається «газ». Газ сплачується криптовалютою Ether, таким чином кожне виконання контракту, кожна транзакція і навіть просте зберігання даних у блокчейні потребує деякої плати. Сам Ethereum працює на віртулаьній машині EVM. Це спеціальний простір виконання в якому виконуються всі смарт-контракти. EVM інтерпретує байт-код, який скомпілювали мови як Solidity або Vyper, та виконує його однаково на всіх вузлах мережі, забезпечуючи однаковий результат незалежно від місця виконання [7].

Ethereum як і Bitcoin використовував механізм Proof-of-Work поки у вересні 2022 року не перейшов на механізм Proof-of-Stake. За новим принципом блоки виробляються валідаторами, які вибираються на основі того хто скільки поставив ETH на стейкінг. Це дозволило значно зменшити енергоспоживання та підвищити безпеку і стійкість мережі. Остання важлива особливість те що Ethereum відкрита платформа. Будь хто може створити свій децентралізований додаток, токен, смарт-контракт чи навіть власний ша взаємодії з користувачами та запустити його без згоди з боку централізованної організації.

Solana блокчейн-мережа, яку створили задля швидкості та доступності. Цілю мережі є стимулювання створення смарт-контрактів і dApps що і робить її корисною для DeFi, NFT та інших застосунків. Solana написана на мові програмування Rust, що відомий своєю продуктивністю та безпекою пам’яті. Також підтримуєтються мови C/C++ що дозволяє розробникам використовувати функціонал цих мов для конкретних задач в системі Solana.

На відміну від традиційних блокчейнів, які зазвичай використовують механізм Proof-of-Work, Solana використовую інший принцип який об’єднує два механізми [8].

Proof-of-Stake який дозволяє валідаторам забезпечувати безпеку мережі. Ці валідатори відповідають за додавання нових блоків до блокчейну та отримують нагороду за свій внесок.

Proof-of-History іноваційний підхід до часової відмітки транзакцій. PoH ставить часові відмітки усіх подій у мережі, формуючи перевіряючий запис їх порядку. Це усуває необхідність у комунікації між валідатором та збільшує швидкість оброки транзакцій.

Поєднання цих механізмів є ключовим фактором відомості Solana за дуже гарну швидкість.

Binannce Smart Chain високопродуктивна блокчейн мережа, яку створила криптовалютна біржа Binance. На відміну від Ethereum має більш швидку обробку транзакцій та менші комісії. Мережа функціонує як незалежна блокчейн-мережа, яка працює паралельно з BNB Beacon Chain. Вона створена задля забезпечення високої продуктивності при цьому зберігаючи сумісність з Ethereum за допомоги EVM (Ethereum Virtual Machine) [9]. Це дозволяє розробникам легко переносити dApps, використовувати інструменти розробника такі як MetaMask, Truffle та Remix, а також писати смарт контракти на Solidity.

У Binance Smart Chain використовується гібридний алгорітм Proof-of-Staked-Authority, що поєднує в собі принципи Delegated-Proof-of-Stake та Proof-of-Authority. У системі діють 21 валідатор, які вибирають зі списку

кандидатів на основі стейкінгу токенів BNB. Блоки створюються кожні 3 секунди, що і забезпечує високу пропускну здібність та швидке закриття транзакцій.

Мережа підтримує усі стандартні інструменти Ethereum-розробки, а також має свій власний блокчейн-експлорер BscScan, що є аналогом EtherScan. BSC активно розвивається у мережі працює десятки DeFi-протоколів, наприклад PancakeSwap, Venus, BakerySwap та ін. Також наявна інтеграція з кросчейновими мостами, що дозволяють переміщувати активи між різними мережами.

Polkadot блокчейн-протокол, котрий об’єднує велику кількість спеціалізованих блокчейнів у єдину універсальну мережу. Вона дозволяє подолати обмеження вже існуючих блокчейнів та передати владу від інтернет-організацій до користувачів, зберігаючи при цьому основні характеристики попередніх блокчейн-технологій та впроваджуючи нові важливі покращення. Переваги Polkadot у зрівнянні з іншими традиційними блокчейнами є те, що вона має шардингову мультичейн-структуру. Завдяки цій особливості вона може опрацьовувати багато транзакцій одночасно на декількох блокчейнах [10]. Через це Polkadot більш масштабована та універсальна, а також має великий потенціал для подальшого розвитку. В архітектурі мережі передбачено те що різним блокчейнам потрібні різні набори параметрів. Адаптивність надає користувачам можливість налаштовувати кожен блокчейн у відповідності до цілей для яких він призначений. Завдяки своїй універсальності мережа працює більш ефективно, безпечно та якісно, оскільки блокчейн не перевонюється надлишком кодом і завжди виконує свої функції.

Polkadot забезпечує злажену взаємодію між мережами та застосунками, подібно тому як це відбувається у мобільному телефоні. У цьому і є одна з ключових відменностей від більш старих ізольованих мереж. Це відкриває можливості для розробки просунутих сервісів і передачі інформації між блокчейнами. Також Polkadot дозволяє спільнотам адмініструвати власні

мережі, опираючись на принципи прозорості та демократичності. Користувачі можуть за необхідності варіативно змінювати та покращувати свої структури управління.

Avalanche – це блокчейн-платформа основна ціль якої представити високо масштабоване, безпечне та сумісне рішення для децентралізованих застосунків і користувацьких блокчейн-мереж. Платформа вирішує ці проблеми за допомоги алгорітму Proof-of-Stake, який називають Snowman Consensus Protocol, який забезпечує високу швидкість транзакцій, малу затримку та масштабованість. Avalanche має складну основу, яка і дозволяє досягти таких параметрів [11].

Платформа складається з трьох блокчейнів, тобто використовуються три окремі блокчейни X-Chain, C-Chain та P-Chain кожний з них виконує власну роль. Exchange Chain (X-Chain) відповідає за створення та передачу цифрових активів. Contract Chain (C-Chain) використовується для виконання смарт-контрактів та розгортання децентралізованних додатків на базі Ethereum. Platform Chain (P-Chain) керує валідаторами мережі, відслідковує активні субсети та координує загальний консенсус мережі. Протокол консенсуса Snowman використовує механізм, при якому вузли мережі спілкуються з декількома випадковими вузлами одночасно, а не зі всією мережею. Це робить процес швидше та більш масштабованим і дозволяє мережі досягти консенсусу з мінімальною комунікацією.

Також платформа підтримує субсети які можна налаштовувати та незалежні блокчейни, які можуть мати власні правила, моделі керування та токени. Ці субсети дозволяють різним децентралізованим додаткам та мережам працювати паралельно. Саме це і допомагає Avalanche масштабуватися підтримуючи різні варіанти використання та снижуючи перенавантаження на основну мережу. Також платформа сумісна з EVM через що її часто зв’язують з Ethereum і хоч обидві мережі достатнбо популярні для створення децентралізованих додатків підходи дуже відрізняються.

1.3 Мови програмування для Ethereum

Solidity високорівневий об’єктно-орієнтована мова програмування для написання смарт-контрактів на блокчейні. Вона найбільш відома завдяки зв’язку з домом DeFi, Ethereum, система була створена розробниками Ethereum для створення смарт-контрактів, які працювали б з віртуальною машиною Ethereum у мережі. У просторі web3-розробників Solidity є найбільш використовуємою мовою програмування, хоч у ньому і присутні елемнти деяких інших мов таких як JavaScript, C++ та Python [12]. Solidity статитично типізована мова, що підтримує наслідування, складні користувацькі типи та бібліотеки що необхідні для функцінування смарт-контрактів.

В основному мова використовується для написання смарт-контрактів, що є основою мережі Ethereum та відносяться до програм, які виконуються на EVM. Вони виконують багато функцій, в основному пов’язанних з автоматизацією транзакцій у мережі. EVM виступає як регістратор “глобального ком’ютеру” Ethereum, що значить він підтримує глобальний стан мережі, також адреси, залишки на рахунках і володіння токенами.

Контракти Solidity компілюються у байт код, який потім зчитує EVM, тому теорітично можна писати смарт-контракти безпосередньо на байт-коді. Але написання коду на байт-коді майже не використовується оскільки це є технічно більш складно і велика вірогіднітсь помилок. Виходячи з цього використання мов віщого рівняння як Solidity дозволяє розробникам зусередити увагу на логіці та функціональності коду замість того, щоб розбиратися зі складностями написання на байт-коді.

Vyper мова програмування створена на базі Python, високорівнева мова програмування створена спецально з урахуванням потрібностей у безпеці, простоті та читабельності коду смарт-контрактів, які розгортаються у EVM-сумісних блокчейнах. Окрім питань читабельності, зручності кодингу та пов’язаних з цим особливостей Vyper, окремої уваги заслуговує робота мови

з пам’яттю. На відміну від Python, у Vyper відсутня динамічна типізація. Тобто тип кожної змінної повинен бути визначен одразу. Така особливість пов’язана з тим що у Vyper відсутні динамічні структури данних. Тому програмісту що пише код потрібно точно вказувати наскільки великим буде об’єкт, який він записати до пам’яті.

При компіляції Vyper заздалегіть розуміє яку кількість пам’яті він повинен виділити для кожного об’єкту. Це дозволяє не використовувати вказівники вільної пам’яті, які є у Solidity. Цей поінтер завжди вказує на вільну комірку пам’яті, наступну за останьою зайнятою. Щоразу коли до пам’яті щось записується, вказівник зміщується до наступної вільнох комірки, тим самим вказуючи куди будуть записані данні далі [13]. З одного боку це дозволяє Solidity мати динамічні структури даних, а з іншого боку це збільшує витрати газу. Тому відсутність таких поінтерів у Vyper дозволяє йому біль ефективно використовувати газ.

Але всеж є і недоліки, Vyper наразі меньш популярний ніж Solidity, що значить спільнота розробників менше. Це може привести до складнощів пошуку інформації та ресурсів для проєктів на Vyper. Швидкість оновлень мови набагато нижча ніж у Solidity, тому нові функції з’являються рідше, а також через вбудовані обмеження деякі нові функції просто можуть не з’являтися у Vyper на відміну від мов конкурентів. Також Vyper гірше адаптований до інфрастуктури Web3 тому багото платформ підтримують Solidity.

Huff низькорівнева мова програмування для створення смарт-контрактів у мережі Ethereum. Вона орієнтована на максимальну ефективність використання газу та надає повний контроль над компіляцією контракту у байт-код EVM . Huff подібен до асемблеру для архітектури EVM та його пряме призначення створення дуже оптимізованих контрактів, коли ефективність використання ресурсів дуже критична.

Головною особливістю мови є те, що вона дозволяє писати інструкції в такому вигляді як вони будуть виконуватися у EVM. На відміну від

високорівневих мов як Solidity або Vyper, Huff не сховує деталі виконання, а потребує від розробника розуміння стекової архітектури EVM, що робить мову доволі складну у вивчені [14]. Програим написані на ній це ланцюжки інструкцій, які керуються стеком, де операції задаються явно власноруч. Таким чином, є можливість наприклад мінімалізувати кількість операцій, уникати зайвих переходів та використовувати точні інструкції для виклику та керування пам’яттю. Завдяки цьому смарт-контракти на Huff потребують набагато менше газу, чим такі самі контракти на Solidity.

Мова є популярною у розробників DeFi додатків, де важлива кожна одиниця газу. Наприклад деякі компоненти проєкту Uniswap були написані на Huff для збільшення ефективності. Але незважаючи на те що це доволі потужна середа розробки мова не користуються загальною популярністю у розробників звичайних додатків, оскільки потрібно багато часу на розробку та відладку. Мова ніяк не призначена для нових розробників та використовується зазвичай для створення бібліотек, ядер протоколів та інших компонентів.

Yul проміжна низькорівнева мова, яку розробили для EVM та eWASM. Вона була створена як уніфікована асемблоподібна мова, яка підходила би до декількох віртуальних машин, що робить її корисною для майбутньої модульності Ethereum. Мова була спроектована розробниками Solidity, як проміжний етап для більш високорівневих мов.

Призначення Yul надати розробнику контроль над генерацією байт-коду при цьому зберігаючи більш читаємий та структурований синтаксис, ніж у традиційних EVM-асемблерів. Вона більш проста у використані ніж Huff, а також надає доступ до більшості можливостей EVM. В Yul можна напряму звертатися до команд віртуальної машини, керувати стеком, пам’яттю та сховищем контракту. Спершу мова проектувалась, як низкорівнева ціль для компіляторів мов Solidity та Vyper. Тобто наприклад Solidity спочатку переводиться у Yul а потім вже в байт-код EVM [15]. Це робить Yul зручним не тільки для ручного програмування, а також для

оптимізації контрактів на стадії компіляції. Розробник може додавати фрагменти коду на Yul до коду Solidity використовуючи inline-assembly. Такий підхід дозволяє комбінувати високорівневу логіку та низькорівневу оптимізацію в одному контракті.

Yul+ розширення для Yul, яке використовується зазвичай у компіляторі Cairo. Воно додає деякі зручності наприклад локальні змінні, функції, базові керуючі функції та більш гнучку структуру кода. Такі нові можливості дозволяють легше писати складну логіку контракту у порівнянні з базовим Yul. Мови Yul та Yul+ підходять для оптимізації критичних ділянок смарт-контрактів, де важлива економія газу або потрібна особлива логіка, яку важко реалізувати на Solidity. Але через свою низькорівневу природу, робота з Yul вимагає хорошого розуміння EVM, її структури даних та обмежень.

2 ПРИНЦИПИ ООП

2.1 SOLID

У розробці програмного забезпечення велике значення має якість коду, наскільки він зрозумілий, легко змінний та надійний. Особливо це важливо при створенні масштабних проєктів, де над кодом працює декілька людей і зміни відбуваються постійно. Щоб спростити проектування таких проєктів були сформульовані принципи, які допомогають структурувати код правильно с самого початку. Одним з найвідоміших принципів та признаним суспільністю є принцип SOLID. SOLID абривіатура котра поєднює п’ять ключових принципів об’єктно-орієнтованого проектування, які допомагають писати гнучкий, стійкий та легко читаємий код (рисунок 2.1).

Рисунок 2.1 – Принцип SOLID

Перший принцип Single Responsibility Principle, про те, що у кожного класа повинна бути тільки одна причина для зміни. Іншими словами кожен клас повинен вирішувати тільки одну конкретну задачу. Якщо клас виконує декілька ролей, то у майбутньому зміна одніє з ролей може вплинути на

іншу, що робить проектування складнішим. Наприклад якщо клас User відповідає одночасно за зберігання інформації о користувачі, а також за виведення звітів та зберігання до бази даних то є порушення принципу. Краще розділити функціонал між трьома різними класами наприклад UserData, UserReport, UserStorage. Це зробить код більш зрозумілим та внесення змін до одного класу не змінить інші.

Другий принцип Open/Closed Principle, про програмні компоненти (класи, функції, модулі) вони повинні бути відкриті для розширення, але закриті до змін. Тобто коли потрібно додати новий функціонал не потрібно модифікувати існуючий код, а потрібно його розширити. Наприклад використання інтерфейсів та спадкування. Припустимо є інтерфейс Payment, а також його реалізації CreditCardPayment, PayPalPayment. Тоді щоб додати новий клас CryptoPayment, ми не чіпаємо старі класи, а просто створюємо новий, який буде реалізовувати той самий інтерфейс.

Третій принцип Liskov Substitution Principle, говорить, що об’єкти підкласу повинні без проблем замінювати об’єкти батьківського класу при цьому не порушуючи логіку програми. Якщо виникають проблеми, то скоріш за все порушена ієрархія спадкування. Наприклад є класи Bird та Penguin. Якщо базовий клас Bird має метод fly(), а Penguin як підклас не може літати, виникає логічна помилка. В такому випадку краще передивитися архітектуру та змінити базові класи, щоб не виникало помилок у підкласах.

Четвертий принцип Interface Segregation Principle, про те, що інтерфейси не повинні змушувати клієнтів реалізовувати методи, які їм не потрібні. Замість одного великого інтерфейсу краще створити декілька менших, які спеціалізуються на своїх задачах. Наприклад замість інтерфейсу Work, який мав би методи work(), eat(), sleep(), краще зробити три откремі інтерфейси. Це зробить інтерфейси більш гнучкими та збільшить зручність розробки.

Останній метод Depency Inversion Principle про те, що модулі верхнього рівня не повинні залежати від модулів нижчого рівня. Усі модули повинні

залежати від абстракцій, а вони не повинні залежати від деталей, навпаки деталі залежать від абстракцій. Практично це значить, що замість створення залежності від конкретного класу створюється залежнітсь від інтерфейсу. Такий підхід дозволяє легше замінняти реалізацію, робити модулі більш незалежними та простіше тестувати окремі компоненти.

2.2 GRASP

GRASP абрівіатура від General Responsibility Assigment Software Patterns він не такий відомий як SOLID, але є не менш важливим для архітектури програмного забезпечення. GRASP складається з дев’яти принципів, кожен з яких описує як і чому потрібно призначити відповідальність тому чи іншому класу або об’єкту (рисунок 2.2).

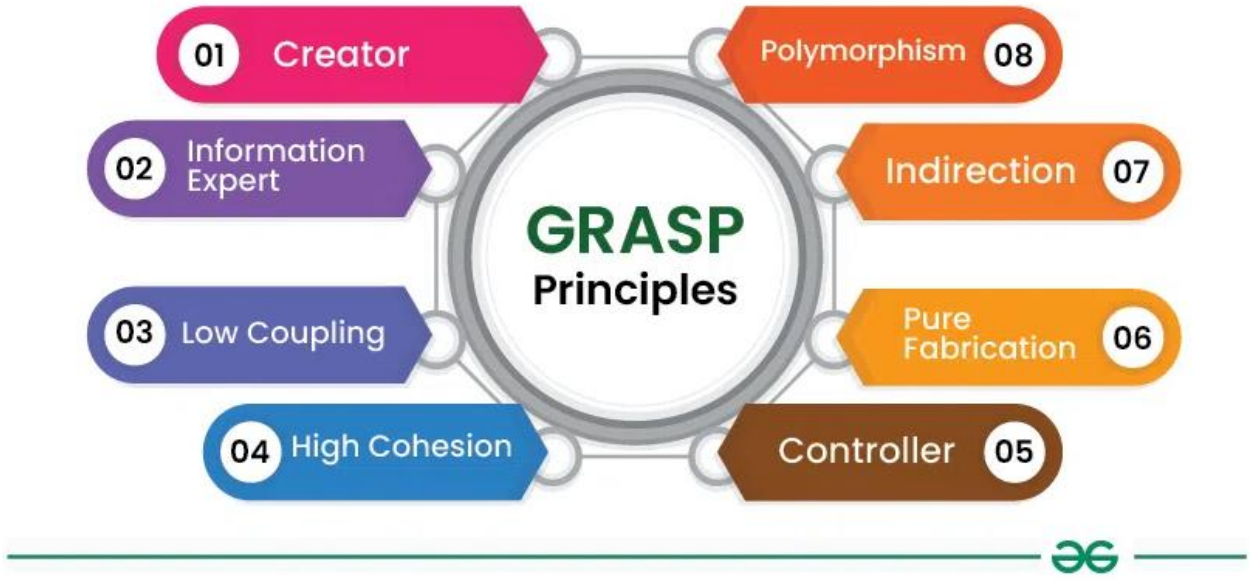

Рисунок 2.2 – Принцип GRASP

Information Expert це принцип про те, що відпоідальність потрібно призначати тому об’єкту який володіє достатньою інформацією для її виконання. Наприклад якщо потрібно порахувати суму замовлення то логічним буде, використати об’єкт Order, який знає список товарів та їх

кількість. Це дозволяє тримати логіку поряд з даними, роблячи код логічніше.

Creator принцип який вказує на те, який об’єкт повинен створювати інший. Замисел у тому що наприклад об’єкт А повинен створювати об’єкт В якщо: А містить В, А використовує В, А агрегує В, А ініціалізує В. Це допомогає дотримуватися зв'язності та робить код передбачуваним.

Controller це об'єкт, який отримує зовнішній запит (наприклад, від користувача чи іншого шару системи) та спрямовує їх у відповідні частини бізнес-логіки. Зазвичай контролер не містить логіки сам по собі, а лише координує роботу інших об'єктів. Він служить перехідною ланкою між інтерфейсом користувача і бізнес-логікою. Наприклад у веб-застосунку це може бути клас OrderController, який отримує запит на оформлення замовлення і передає його в сервіси.

Low Coupling принцип слабкої зв’язності значить, що всі класи та об’єкти повинні як можна менше залежати один від одного, чим слабкіша зв’язанність, тим легше змінювати та повторно використовувати компоненти. Це досягається наприклад через використання інтерфейсів та інверсій залежностей.

High Coupling навпроти сильна внутрішня пов'язаність. Об'єкт з високою зв'язністю відповідає за логічно пов'язані завдання і не розповзається в інші області. Такі об'єкти простіше зрозуміти, налагодити та повторно використовувати. Наприклад клас, який відповідає лише за надсилання листів, а не за генерацію звітів або обробку помилок, є прикладом високої зв'язності.

Polymorphism – цей принцип полягає у тому, що загальні поведінки різних об'єктів мають бути реалізовані через поліморфізм. Тобто замість того, щоб писати if-розгалуження за типами, краще використовувати загальний інтерфейс або абстрактний клас. Наприклад є інтерфейс PaymentMethod, а реалізація CreditCardPayment та PayPalPayment. Замість того щоб перевіряти тип платежу і викликати вручну потрібний метод,

просто викликається метод pay() – і потрібна реалізація буде обрана автоматично.

Pure Fabrication про те, що іноді для дотримання інших принципів, таких як слабка зв'язаність або висока зв'язність, корисно створити "штучний" клас, який сам не відображає реальну сутність з предметної області. Наприклад клас Logger або EmailService – це штучні абстракції, створені виділення технічної логіки в окремі компоненти.

Indirection принцип непрямості говорить, що іноді корисно запровадити проміжний шар чи об'єкт, щоб знизити зв'язність між іншими елементами системи. Наприклад інтерфейси та посередники (Mediator, Facade, Adapter) це реалізації непрямості. Завдяки цьому один модуль може бути замінений або протестований окремо від інших.

Protected Variations сенс цього принципу – ізолювати частини системи від змін інших частин, які можуть вплинути на них. Це досягається через стабільні інтерфейси та абстракції. Наприклад: якщо ви маєте різні способи зберігання даних (база даних, файли, хмара), варто створити абстракцію Storage, від якої залежатиме бізнес-логіка. Тоді при зміні способу зберігання не доведеться змінювати всю іншу систему.

2.3 Порівняння SOLID та GRASP

Шаблони GRASP більше фокусуються на початковій стадії проектування, коли система додатку тільки формується. Вони дозволяють визначити які об’єкти потрібні у системі, за що кожен з них буде відповідати та як уникнути зайвої зв’язності та дублювання коду. Принципи GRASP особливо корисні при створенні моделі предметної області, і часто застосовуються разом з UML-діаграмами та шаблонами проектування. Їхнє завдання – зробити систему зрозумілою та логічно збудованою ще до початку активної розробки.

З іншого боку, принципи SOLID спрямовані на побудову якісної,

гнучкої архітектури модулів та класів. Вони допомагають структурувати код так, щоб він легко розширювався, був стійкий до змін та максимально незалежний від деталей реалізації. На відміну від GRASP, SOLID застосовується найчастіше вже на етапі реалізації або при рефакторингу. Принципи особливо корисні серед, де проєкт активно розвивається, часто змінюється, а команда складається з кількох розробників.

Обидва підходи перетинаються загалом – підвищити якість архітектури, зробити код логічним та модульним. Проте різницю між ними очевидні. GRASP в першу чергу спрямований на те, щоб правильно розкласти обов'язки по об'єктах, а SOLID – на те, щоб забезпечити стійкість та гнучкість системи у процесі її розвитку. Іншими словами, GRASP більше про «що» і «кому доручити», а SOLID про «як саме це реалізувати» на рівні коду та структури.

Незважаючи на те, що обидва підходи є корисними та актуальними, у сучасній практиці найчастіше орієнтуються на SOLID. Це пояснюється тим, що принципи SOLID простіше інтегруються в сучасні фреймворки та архітектурні підходи, такі як мікросервісна архітектура, шарувата архітектура та DDD (Domain-Driven Design). Крім того, вони тісно пов'язані з такими практиками, як автоматичне тестування, модульність, використання залежностей і шаблони проектування. Завдяки цьому SOLID вважається більш універсальним та практично застосовним набором принципів, особливо в умовах комерційної розробки.

3 ПРОГРАМНА РЕАЛІЗАЦІЯ ЗАСТОСУНКУ ОНЛАЙН-АУКЦІОНУ

3.1 Загальна структура проєкта

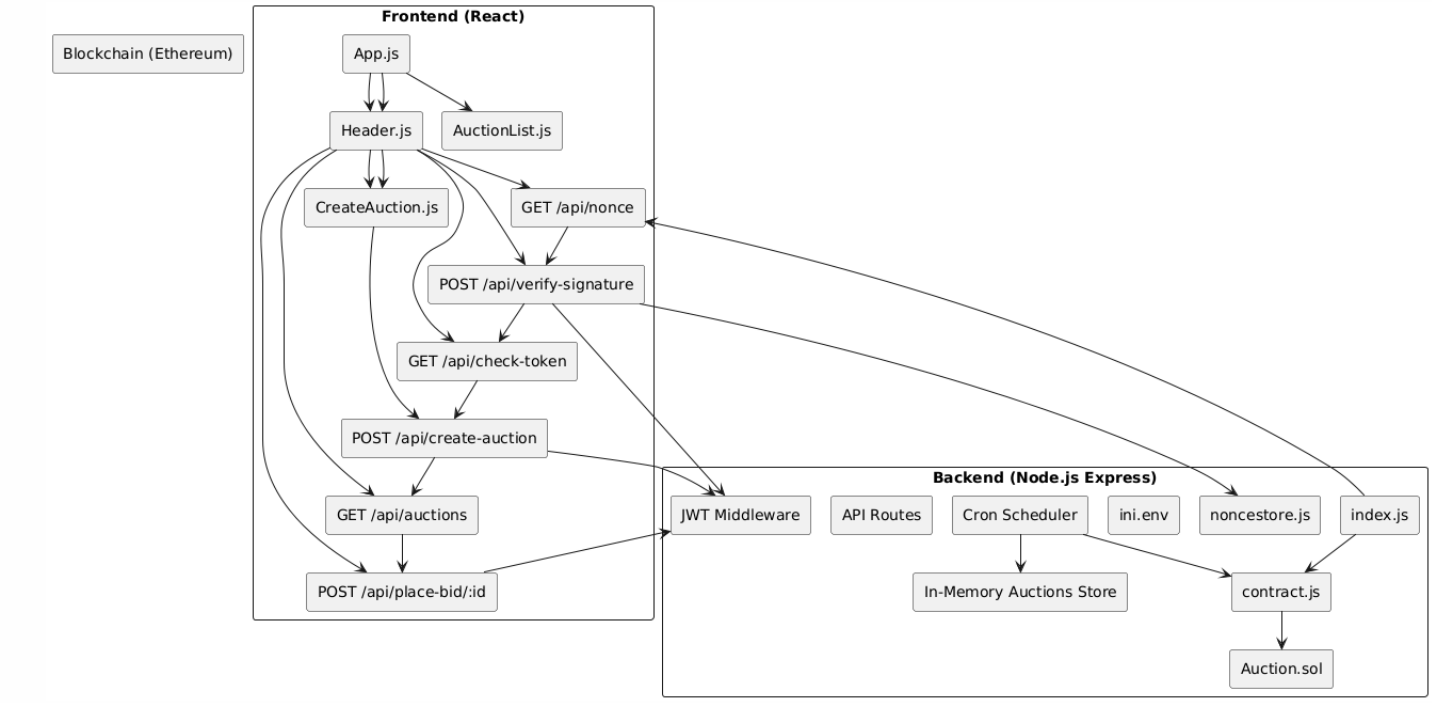

Рисунок 3.1 – Діаграма проєкту аукціона

Із діаграми проєкту бачимо (рисунок 3.1), що у розробленому вебзастосунку клієнтська частина, є основним засобом взаємодії користувача з системою аукціонів. Ця частина проєкта реалізована зі використанням сучасної бібліотеки для WEB розробки React [16]. Це дозволило побудувати застосунок зрозумілим до використання, з гнучкою структурою та високим рівнем інтерактивності. Початковую точку входу у застосунок є компонент App.js, що відповідає за загальне компонування інтерфейсу, зв’язок та маршутизацію між файлами застосунку. Перший компонент застонку з яким взаємодіє користувач це Header.js. Він виконує одну з важливих ролей у клієнтскій частині додатку. Зовні це звичайний на перший погляд заголовок

сайту, але саме у ньому реалізована логіка авторизації користувача.

У заголовку є кнопка входу, яка активує процес авторизації користувача через WEB3-гаманець, наприклад через MetaMask у нашому випадку. Далі відбувається ланцюжок, що охоплює як фронтєнд частину так і бекєнд. Спочатку надсилається запит на Get /api/nonce після чого сервер повертає унікальну згенеровану фразу, яку потім користувач підписує у своєму гаманці. Цей підпис підтвердить, що саме цей користувач володіє акаунтом у мережі Etherium з якого відбувається спроба входу. Після підписання nonce користувач відсилає його назад до сервера через запит Post /api/verify-signature. Якщо все пройшло успішно на бекєнді створюється JWT, який потім зберігається у браузері й буде використанний для подальших захищених запитів.

Також компонент завжди перевіряє стан аутентифікації завдяки внутрішньому менеджменту стану через локальні сховища. Для цього використовується постійна перевірка токену, чи залишився він активним при перезавантаженні сторінки чи ні перевіряючи його актуальність через запит GET /api/check-token. Якщо термін дії токену вийшов або він недійсний, користувача автоматично виведе із системи. Такий підхід на практиці дуже зручний для користувача, бо не потрібні зайві логіни та паролі, ніяких підтверджень пошти або номерів телефонів. Усе що потрбіно лише мати гаманець та погодитись на електронний підпис, який не має жодного доступу до коштів. У заголовку є навігація до створення аукціонів CreateAuction.js що визиває вікно створення аукціонів. Це все реалізовано через механізми внутрішньої маршутизації бібліотеки React [17]. Це дозволяє отримати зручну, плавну та швидку взаємодію з інтерфейсом додатку.

CreateAuction один з головних функіональних блоків клієтської частини застосунку, бо саме за його допомоги користувач має змогу створювати аукціони. Це не лише відображення форми для заповнення, він представляє логічний та зручний візуальний інтерфейс, що керує створення нових лотів, взаємодіє з бекєндом. Також блок перевіряє та валідує введені

дані, має логіку захищенного створеного лотів завдяки попередній аутинтифікації та має невелику взаємодію зі смарт-контрактом.

Перед надсиланням форми усі дані проходять попередню перевірку на клієнтській частині. Наприклад перевіряється, щоб поле з назвою не було порожнє, стартова ціна була більшою за нуль а дата закінчення аукціону у майбутньому. Коли форма пройшла усі перевірки створюється лот та ініціюється запит на бекєнд за маршрутом Post /api/create-auction. Разом зі змістом форми до запиту додається JWT-токен користувача, який забезпечує захист даних, бо тільки авторизовані користувачі можуть створювати аукціони. Якщо же токен за якимись обставинами не валідний, з’явиться повідомлення про необхідність авторизуватись знову.

Іншим важливим блоком клієнтської частини є AuctionList.js. Він відповідає за виведення списку усіх доступних аукціонів, фактично це головна вітрина додатку. Саме завдяки цій частині користувач може переглядати активні аукціони їх поточні ставки, дати завершення або власника. При завантаженні сторінки одразу надсилається запит до серверу за маршрутом GET /api/auctions спеціальний API-ендпоінт, що повертає список актуальних аукціонів у форматі JSON. Надалі дані вже зберігаються у внутришньому стані React-компоненту, після чого автоматично перерендерюються і відображаються у зручному вигляді картки [18].

Блок AuctionList підключається безпосередньо до App, тому при завантажені сторінки одразу відображується список лотів. Це зручно бо користувачеві не потрбіно робити зайвих дій для того щоб перейти до перегляду лотів. До того ж, якщо не виникає нійяких помилок на сервері то список завантажується дуже швидко і також швидко оновлюється. Також у компоненті передбачена обробка помилок, якщо сервер не відповідає або виникла якись помилка даних користувач побачить що аукціони завантажуються.

Бекєнд частина розробленого застосунку є невід’ємною частиною загальної архітектури системи. Саме вона відповідає за обробку запитів

клієнта, верифікацію каористувачів, взаємодію з блокчейном, обробку ставок та створення нових лотів. Сервірна логіка реалізована за допомоги Node.js середовища виконання JavaScript, що дозволяє ефективно реалізовувати асинхрону обробку подій, масштабування та швидкий запуск серверів. Основою для маршутизації та побудови HTTP API виступає Express.js потужний фреймворк для створення веб-серверів.

Основним файлом проєкту є index.js, який ініціалізує сервер, підключає всі маршрути API та конфігураційні параметри, а також запускає HTTP-з'єднання за вказаним портом. У файлі також відбувається інтеграція з middleware-компонентами, парсерами тіла запитів, обробником помилок та модулями аутинтифікації на базі JWT [19]. Цей блок центральний хаб, через який відбувається виклик усіх основних логічних модулів.

Один з базових блоків система аутинтифікації, що реалізована через комбінацію криптографічної перевірки користувача та JWT-токенів. При першому звернені користувача до сервера через маршрут GET /api/nonce бекенд генерує випадковий рядок nonce, який тимчасово зберігається в блоці noncestore.js. Цей nonce надсилається користувачу для підпису за допомоги криптогаманця MetaMask. Підписаний nonce надсилається назад за маршрутом Post /api/verify-signature, де сервер перевіряє дійсність підпису, розшифровуючи його та звіряючи з оригінальним значенням nonce. Якщо перевірка пройшла успішно, користувач отримує JWT-токен, який використовується в подальших запитах.

JWT-токен створюється на основі секретного ключа, що зберігається у конфігураційному файлі ini.env. Він містить основні параметри середовища із секретним ключом, портом сервера, адресою смарт-контракту і приватним ключем. Для захисту маршрутів серверної частини використовується проміжний маршрут JWT Middleware. Він обробляє всі вхідні запити до захищених маршрутів, перевіряючи наявність і дійсність токена. Якщо токен недійсний, неправильно підписаний або взагалі відсутній, запит не буде допущений до виконання, а користувач отримає повідомлення про відмову у

доступі. Це дозволяє чітко відокремити логіку авторизації та захистити критичні дії, як створення аукціонів або подання ставок.

Окремої уваги заслуговує блок контрактної взаємодії contract.js, у ньому реалізован код що відповідає за виклик методів смарт-контракту розгорнутий у мережі Ethereum. Для взаємодіїї із контрактом у цьому блоці використовується бібліотека Ethers.js [20]. Контракт ініціалізується за допомоги ABI-файлу та адреси розгортання, після чого сервер може викликати його методи, передаючи необхідні параметри.

При створенні нового аукціону сервер приймає запит від користувача, розпаковує отрримані параметри та далі звертається до блоку contracts, щоб викликати відповідну функцію створення аукціону в смарт-контракті. Важливим моментом також є, що перед викликом система перевіряє наявність газу на акаунті, перевіряючи баланс користувача та забезпечує локальну перевірку валідності параметрів, до того як передасть дані до блокчейну.

Наступний важливий маршрут POST /api/place-bid/:id реалізує функціонал ставок на поточні аукціони. Сервер перевіряє, чи активний аукціон, не перевищена ставка, а потім надсилає запит до смарт-контракту з вказанням суми ставки та адресою користувача, що зробив ставку. Такі транзакції потребують деякої невеликої комісії за виклик смарт-контракту, на даному єтапі спачуємою користувачем. Усі операції записуються до блокчейну через що неможливо провести фальсифікацію ставок та забезпечуючи прозорість процесу.

Для перегляду поточних аукціонів на бекєнді також створений маршрут GET /api/auctions. На разі працює за підтримки тимчасового локального сховища у пам'яті проєкту, де зберігаються необхідний набір даних. Наразі таке рішення дозволяє уникнути затримок пов'язаних із взаємодією з блокчейном та дозволяє отримувати список аукціонів майже моментово. Також реалізований фоновий елемент Cron Scheduler задача якого, полягає у переодичному оновлені інформації по стану аукціонів. Якщо

аукціон завершився викликається метод зі смарт-контракту який перераховує кошти до власника лоту та маркує завершені аукціони як архівовані.

Ключова складова застосуку для проведення аукціонів це смарт-контракт. Він визначає ділову логіку проведення аукціонів, перевіряє ставки користувачів, реєструє переможців та забезпечує цілісність та прозорість проведення торгів. Саме смарт-контракт пов'язаний з блокчейн системою Ethereum, що гарантує незмінність умов та їх децентралізоване виконання.

Контракт написаний на мові Solidity основній мові для розробки смарт-контрактів у мережі Ethereum. Структура файлу передбачає важливі моменти структур даних, функцій для створення та участі в аукціонах, а також автоматичні механізми завершення торгів. На відміну від традиційних централізованих систем, де вся логіка зберігається на сервері, у цьому проєкті саме смарт-контракт контролює увесь процес.

Створення нового аукціону відбувається викликом публічної функції createAuction, що приймає параметри: назва товару, стартова ціна, тривалість аукціону. Функція записує дані до блокчейну, ініціалізує новий запис до мапінгу та формує подію, яка повідомляє зовнішній системі про появу нового лоту. Кожен новий лот прив’язаний до адреси його творця. Подання ставки реалізовано через функцію placeBid, що дозволяє будь-якому користувачу, який має кошти на своєму гаманці, підвищити поточну ставку. Для цього з гаманця знімається сума відповідна його ставці, надалі ці кошти зберігаються у смарт-контракті.

Контракт автоматично перевіряє стан аукціону, чи він не завершився, чи більша нова ставка за поточну та чи не є користувач вже поточним лідером ставок. Якщо усі перевірки пройшли успішно нова сума записується як поточна ставка, а кошти попередньої ставки повертаються до попереднього лідера. Такий підхід захищає кошти користувачів та забезпечує добросовесність торгів.

3.2 Фронтенд частина проєкту

3.2.1 Файли App.js та Header.js

Файл App.js центральний елемент клієнтської архітектури поєднує в собі декілька інших функціональних компонентів, такі як заголовок та список аукціонів. У файлі не реалізовано складної логіки, але його структра впливає на впорядкування інших компонентів інтерфейсу (лістинг 3.1).

Лістинг 3.1 – Структура файлу App.js

```
import React from "react";
import Header from "./Header";
import AuctionList from "./AuctionList";
import "./App.css";

function App() {
  return (
    <div>
      <Header />
      <main className="main-content">
        <h1 className="main-title">Активні аукціони</h1>
        <AuctionList />
      </main>
    </div>
  );
}

export default App;
```

Спочатку імпортуються два компоненти Header що містить логіку авторизації та інтерфейсні елементи навігації, та AuctionList, який відповідає за відображення поточних аукціонів, які були отримані з бекєнду. Компонент App збирає все у спільну структуру, що потім відображається на сторінці. Такий підхід дозволяє розбити функціональність на різні модулі, зі своєю сферою діяльності, а потом зібрати все у гарно структурований вид. Це спрощує розробку, тестування і масштабування інтерфейсу.

Все починається з компоненту Header.js один з головних елементів застосунку. У ньому реалізована логіка авторизація користувача через

роширення MetaMask, а також компоненти, що залежать від стану авторизації. Після завантаження компонента відбувається перевірка активної сесії користувача, чи є дійсний JWT-токен у локальному сховищі. Ця перевірка відбувається через React-хук (лістинг 3.2).

Лістинг 3.2 – Перевірка дійсності токена користувача (Header.js)

```
useEffect(() => {
    const checkToken = async () => {
      const token = localStorage.getItem("token");
      const address = localStorage.getItem("userAddress");

      if (token && address) {
        try {
          const res = await
fetch("http://localhost:5000/api/check-token", {
            headers: { token },
          });

          if (!res.ok) throw new Error("Токен недійсний");
          const data = await res.json();

          if (data.success) {
            setUserAddress(address);
          } else {
            logout();
          }
        } catch (err) {
          console.warn("Помилка при перевірці токена:", err);
          logout();
        }
      }
    };
```

Фрагмент представляє ключовий етап у збереженні безперервної сесії користувача. Після першого рендеру застосунок перевіряє, чи збережений токен справді активний. Якщо токен не дійсний дані очищуються, та користувач вважається не авторізованим.

Реалізація авторизації через MetaMask заслуговує окремої уваги. На відміну від звичної авторизації через логін та пароль, у цьому варіанті використовується криптографічна перевірка права власності на гаманець Ethereum (лістинг 3.3).

Лістинг 3.3 – Функція авторизації користувача (Header.js)

```
const loginWithMetaMask = async () => {
    if (userAddress) return;
    if (!window.ethereum) {
      alert("Будь ласка, встановіть MetaMask!");
      return;
    }
    try {
      const provider = new
ethers.BrowserProvider(window.ethereum);
      await provider.send("eth_requestAccounts", []);
      const signer = await provider.getSigner();
      const address = await signer.getAddress();
      setUserAddress(address);
      localStorage.setItem("userAddress", address);

      const nonceRes = await
fetch(`http://localhost:5000/api/nonce?address=${address}`);
      if (!nonceRes.ok) throw new Error("Помилка отримання
nonce");
      const { nonce } = await nonceRes.json();
      const message = `Підпиши цей текст для входу: ${nonce}`;
      const signature = await signer.signMessage(message);

      const verifyRes = await
fetch("http://localhost:5000/api/verify-signature", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ address, signature, nonce }),
      });

      if (!verifyRes.ok) throw new Error("Помилка перевірки
підпису");

      const data = await verifyRes.json();

      if (data.success) {
        localStorage.setItem("token", data.token);
        alert("Успішно увійшли!");
      } else {
        alert("Помилка входу!");
        logout();
      }
    } catch (error) {
      console.error(error);
      alert("Помилка авторизації");
      logout();
    }
  };
```

Функція запитує доступ до гаманця користувача, отримує nonce, який

користувач підписує приватним ключем. Підпис далі надсилається до бекєнду, де верифікується за допомоги публічного ключа і якщо все пройшло успішно створюється JWT-токен.

Також реалізована і функція виходу, який видаляє токен та адресу користувача повертаючи його до неавторізованого стану (лістинг 3.4).

Лістинг 3.4 – Функція для виходу з акаунту (Header.js)

```
const logout = () => {
    localStorage.removeItem("token");
    localStorage.removeItem("userAddress");
    setUserAddress("");
    alert("Ви вийшли з акаунту.");
  };
```

3.2.2 Файл AuctionList.js

Одна з головних функціональних можливостей у застосунку є перегляд поточних аукціонів та можливість взаємодії з ними. Саме у файлі AuctionList реалізована логіка даного функціоналу, отримання даних з бекєнду та взаємодія зі смарт-контрактом. Функції реалізовані з використанням хука useState для збереження станів та useEffect для ініціалізації запитів до API. Поведінку компонента визначають основні стани auctions, loading, selectedAuction, які дозволяють реалізувати адаптивне оновлення UI без повного перезавантаження сторінки.

Після ініціалізації компонента листу аукціонів викликається функція fetchAuctions (лістинг 3.5) що надсилає Get запит до бекєнду. Функція гарантує, що аукціони будуть завантажені лише один раз при оновлені сторінки. У разі виникнення помилок користувач побачить відповідне повідомлення.

Лістинг 3.5 – Функція fecthAuctions (AuctionList.js)

```
const fetchAuctions = async () => {
  try {
    const res = await
```

```
fetch("http://localhost:5000/api/auctions");
    const data = await res.json();
    setAuctions(data.auctions);
  } catch (error) {
    console.error("Помилка при завантаженні аукціонів:", error);
  } finally {
    setLoading(false);
  }
};
```

Можливість робити ставки безпесередньо через блокчейн, ще одна функція реалізована у компоненті. Функція реалізована у handleMakeBid, яка визивається кнопкою зробити ставку (лістинг 3.6).

Лістинг 3.6 – Функція handleMakeBid щоб робити ставки (AuctionList.js)

```
const handleMakeBid = async (auctionId, currentBid) => {
  const token = localStorage.getItem("token");
  if (!token) {
    alert("Необхідно увійти для ставки.");
    return;
  }

  const amount = prompt("Введіть вашу ставку в ETH:");
  ...
};
```

Наступна функція перевіряє наявність JWT-токена користувача та валідність суми ставки. При успішному проходжені усіх перевірок підключається до провайдера MetaMask через бібліотеку ethers, створюється signer та відправляється транзакція виклику методу bid смарт-контракта (лістинг 3.7).

Лістинг 3.7 – Перевірки токена та валідності ставки (AuctionList.js)

```
const contract = new ethers.Contract(CONTRACT_ADDRESS,
auctionABI, signer);
const parsedAmount = ethers.parseEther(amount);
const tx = await contract.bid(auctionId, { value: parsedAmount
});
await tx.wait();
```

Після підтвердження транзакції через блокчейн дані синхронізуються зі

серверною частиною за допоги Post-запиту. Це дозволяє зберігати актуальні дані про ставки та реалізується двостороння синхронізація між блокчейном та бекєндом (лістинг 3.8).

Лістинг 3.8 – Синхронізація ставок з бекєндом (AuctionList.js)

```
await fetch(`http://localhost:5000/api/place-bid/${auctionId}`,
{
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    token,
  },
  body: JSON.stringify({ bidAmount: parseFloat(amount) }),
});
```

Також реалізовано невеличкий дизайн та конструкція створених лотів. Кожен аукціон представлений у вигляді картки у якій представлена інформація про назву, опис лоту, поточну ставку, час завершення та зображення товару (лістинг 3.9).

Лістинг 3.9 – Клас картки лотів аукціону (AuctionList.js)

```
 className={`auction-card ${auction.ended ? "ended" : ""}`}
            >
              {auction.imageUrl && (
                <img
                  src={auction.imageUrl}
                  alt={auction.title}
                  style={{
                    width: "100%",
                    borderRadius: "8px",
                    marginTop: "10px",
                  }}
                />
              )}
              <h3>{auction.title}</h3>
              <p
className="description">{auction.description}</p>
              <p>Початкова ціна: {auction.startPrice} ETH</p>
              <p>Поточна ставка: {auction.currentBid ||
auction.startPrice} ETH</p>
              <p>Завершення: {new
Date(auction.endTime).toLocaleString()}</p>
```

3.2.3 Файл CreateAuction.js

Останній функціональний файл CreateAuction реалізує повний цикл створення лотів користувачем. Задача компонента у зборі вхідних даних, взаємодії хі смарт-контрактом, а також синхронізація з бекєндом, де зберігається інформація не зв’язана з блокчейном. У цій частині використовується система станів на основі хуків React usestate, це дозволяє керувати вмістом форми та взємодією із зовнішніми сервісами. Також передбачена первинна валідація введених даних, перевірку на підключення MetaMask та правильну обробку часу завершення аукціону.

Коли віладація токена авторизації пройшла успішно та у користувача наявний гаманець підключений до MetaMask викликається блокчейн-взаємодія через наступну частину (лістинг 3.10).

Лістинг 3.10 – Ініціалізація блокчен-взаємодії (CreateAuction.js)

```
const provider = new ethers.BrowserProvider(window.ethereum);
      const accounts = await window.ethereum.request({ method:
"eth_requestAccounts" });
      const signer = await provider.getSigner(accounts[0]);

      const contract = new ethers.Contract(CONTRACT_ADDRESS,
auctionABI, signer);
```

Тривалість аукціону розраховується у секудах, через те що смарт-контракт оперує часом у цій одиниці виміру, у файлі весь час розраховується на основі введеної дати заверешння. Після розрахунку часу викликається метод createAuction зі смарт-контракту. Також важливо, що початкова ціна конвертується з ETH у Wei через parseEther, це гаратнтує точність переданої суми (лістинг 3.11).

Лістинг 3.11 – Компненти розрахунку часу та виклику методу створення аукціону

```
const currentTime = Math.floor(Date.now() / 1000);
      const endTimestamp = Math.floor(new
```

```
Date(endTime).getTime() / 1000);
      const biddingTime = endTimestamp - currentTime;

      if (biddingTime <= 0) {
        setStatus("Дата завершення має бути у майбутньому.");
        return;
      }

      const tx = await contract.createAuction(
        title,
        ethers.parseEther(startPrice),
        biddingTime
      );

      await tx.wait();
```

Після підтвердження транзації з блокчену дані дублюються до бекєнду через ендпоінт. Це дозволяє відстежувати та зберігати інформацію, яка не зберігається у блокчейні (лістинг 3.12).

Лістинг 3.12 – Дублювання інформації до бекєнду (CreateAuction.js)

```
const res = await fetch("http://localhost:5000/api/create-
auction", {
        method: "POST",
        headers: {
          "Content-Type": "application/json",
          token,
        },
        body: JSON.stringify({
          title,
          description,
          startPrice,
          imageUrl,
          endTime,
        }),
      });
```

3.3 Бекєнд частина проєкту

Бекєнд частина проєкту реалізована з використанням фреймворку Express.js. У цій частині реалізовано забезпечення безпечної авторизації користувачів, взаємодія з локальним сховищем лотів, підтримка авторизації за допомоги криптографічних підписів та періодичний контроль завершення

аукціонів з проведенням транзакцій на блокчейн.

На початку файлу здійснюється підключення усіх необхідних модулів. Бібліотеки path, express, cors, body-parser, jsonwebtoken для генерації токенів доступу, ethers для криптографічної перевірки підписів, та node-cron для реалізації періодичних фонових задач (лістинг 3.13).

Лістинг 3.13 – Підключання усіх необхідних залежностей

```
const path = require("path");
require("dotenv").config({ path:
path.resolve("E:/VSCodeProjects/Auction/backend", "ini.env") });
const express = require("express");
const cors = require("cors");
const bodyParser = require("body-parser");
const jwt = require("jsonwebtoken");
const { verifyMessage } = require("ethers");
const cron = require("node-cron");
const auctionContract = require("./contract");
const { getNonce, setNonce, generateNonce } =
require("./noncestore");
```

Система авторизації не потребує логінів та паролів, а реалізована через перевірку володіння користувача приватним ключем Ethereum-гаманця. Реалізація цього процеса представлена через nonce, одноразовий токен який повинен підписати власник гаманця (лістинг 3.14).

Лістинг 3.14 – Запит на отримання nonce (index.js)

```
app.get("/api/nonce", (req, res) => {
  const { address } = req.query;
  if (!address) return res.status(400).json({ error: "Address is
required" });
  const nonce = getNonce(address);
  res.json({ nonce });
});
```

Після відповіді від nonce користувач підписує його через MetaMask і потім на основі створеного токену сервер перевіряє валідність користувача. Наступний фрагмент файлу демонструє повний цикл перевірки валідності підпису, оновлення nonce та генерацію JWT-токену для подальшого захисту

API (лістинг 3.15).

Лістинг 3.15 – Фрагмент перевірки користувача (index.js)

```
app.post("/api/verify-signature", async (req, res) => {
  const { address, signature, nonce } = req.body;
  if (!address || !signature || !nonce)
    return res.status(400).json({ error: "Missing data" });

  try {
    const message = `Підпиши цей текст для входу: ${nonce}`;
    const signerAddress = await verifyMessage(message,
signature);

    if (signerAddress.toLowerCase() === address.toLowerCase()) {
      setNonce(address, generateNonce());
      const token = jwt.sign({ address },
process.env.JWT_SECRET, { expiresIn: "1h" });
      res.json({ success: true, token });
    } else {
      res.status(401).json({ error: "Invalid signature" });
    }
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: "Server error" });
  }
});
```

Також створена функція для перевірки актуальності токену. Вона використовується для всіх запитів наприклад створення аукціонів або робленя ставок. Завдяки цій перевірці коричтувач не може взаємодіяти з API без надання актуального токену, що гарантує, що всі дії будуть робитись лише від імені руального власника гаманця (лістинг 3.16).

Лістинг 3.16 – Первірка актуальності токену (index.js)

```
app.get("/api/check-token", (req, res) => {
  const { token } = req.headers;
  if (!token) return res.status(401).json({ success: false,
error: "No token provided" });

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    res.json({ success: true, address: decoded.address });
  } catch (err) {
    console.error(err);
    res.status(401).json({ success: false, error: "Invalid
```

```
token" });
  }
});
```

Далі реалізована логіка створення аукціонів із серверної частини, сюди відправляються дані про створення лоту з фронтенду, також перевіряється дійсність токену авторизації користувача. Також у створеному тимчасовому масиві auctions зберігаються лоти створені користувачами сайту. Це тимчасова заміна бази даних, яку у подальшому можна замінити на реальну (лістинг 3.17).

Лістинг 3.17 – Серверна частина створення аукціонів та їх зберігання (index.js)

```
app.post("/api/create-auction", (req, res) => {
  const { token } = req.headers;
  const { title, description, startPrice, imageUrl, endTime } =
req.body;

  if (!token) return res.status(401).json({ error: "No token
provided" });
  if (!endTime) return res.status(400).json({ error: "End time
is required" });

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    const newAuction = {
      id: auctions.length + 1,
      title,
      description,
      startPrice: parseFloat(startPrice),
      imageUrl,
      endTime: new Date(endTime),
      owner: decoded.address,
      createdAt: new Date(),
      ended: false,
    };
    auctions.push(newAuction);
    res.json({ success: true, auction: newAuction });
  } catch (err) {
    console.error(err);
    res.status(401).json({ error: "Invalid token" });
  }
});

app.get("/api/auctions", (req, res) => {
```

```
  res.json({ auctions });
});
```

Також на бекєнді реалізована логіка розміщення нових ставок. Сервер перевіряє поточний стан лоту та авторизацію користувача. Потім ідентифікує відповідний лот та оновлює поточну ставку (лістинг 3.18).

Лістинг 3.18 – Розміщення нових ставок та оновлення потоної ставки (index.js)

```
app.post("/api/place-bid/:id", (req, res) => {
  const { id } = req.params;
  const { token } = req.headers;
  const { bidAmount } = req.body;

  if (!token) return res.status(401).json({ success: false,
error: "No token provided" });

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    const auction = auctions.find((a) => a.id === parseInt(id));

    if (!auction) return res.status(404).json({ success: false,
error: "Аукціон не знайдено" });
    if (auction.ended) return res.status(400).json({ success:
false, error: "Аукціон вже завершено" });

    auction.currentBid = parseFloat(bidAmount);
    auction.highestBidder = decoded.address;
    auction.bids = auction.bids || [];
    auction.bids.push({
      amount: auction.currentBid,
      bidder: auction.highestBidder,
      time: new Date(),
    });

    res.json({
      success: true,
      message: "Ставка збережена в локальному сховищі",
      auction,
    });
  } catch (err) {
    console.error(err);
    res.status(401).json({ success: false, error: "Invalid
token" });
  }
});
```

Останній реалізований функціонал на серверній частині проєкту це автоматичне завершення аукціонів, яка реалізована через бібліотеку node-cron. Кожні 30 секунд функція перевіряє всі активні аукціони час який сплив, та викликає модуль endAuction із смарт-контракту, після чого вони завершуються на блокчейні та відбуваються усі необхідні транзакції (лістинг 3.19).

Лістинг 3.19 – Автоматична перевірка та завершення аукціонів (index.js)

```
cron.schedule("*/30 * * * * *", async () => {
  const now = new Date();

  for (let auction of auctions) {
    if (!auction.ended && new Date(auction.endTime) <= now) {
      try {
        const tx = await auctionContract.endAuction(auction.id);
        await tx.wait();
        auction.ended = true;
        console.log(`✅ Аукціон ${auction.id} завершено на
блокчейні`);
      } catch (err) {
        console.error(`❌ Помилка завершення аукціону
${auction.id}:`, err.message);
      }
    }
  }
});
```

3.4 Реалізація смарт-контракту

Смарт-контракт серце проєкту саме він забезпечує децентралізованість торгів, їх прозорість, чесну та безпечну взаємодію між учасниками аукціону. Смарт-контракт виступає незалежним посередником торгів, що гарантує виконання усіх правил, які неможливо змінити. Напочатку створення контракту важливо задати правила які стосуються ліцензії. Це дозволить використовувати контракт у відкритому середовищі без юридичних обмежень. Для цього визначається SPDX-License_Identifier: MIT, що значить код публікується під відкритою ліцензією MIT, яка дозволяє будь-кому копіювати та використовувати код.

Створена структура лота містить усі важливі поля для проведення та контролю процесу торгів, seller адреса користувача, який виставив лот на торги, itemName текстовий опис товару, startPrice початкова ціна торгів, яка задається при створені лоту, highestBid адреса учасника з найбільшою ставкою, endTime мітка часу до якого аукціон існує, ended прапорець, який показує чи завершен аукціон (лістинг 3.20).

Лістинг 3.20 – Структура лота у смарт-контракті (Auction.sol)

```
contract Auction {
    struct AuctionLot {
        address payable seller;
        string itemName;
        uint startPrice;
        uint highestBid;
        address payable highestBidder;
        uint endTime;
        bool ended;
    }
```

Так як смарт-контракт не має прямого доступу до інтерфейса користувача, то потрбіно створювати події які потім можна зчитувати через ethers. Подіїї створені для важливих моментів процесу аукціону. Подіїї фіксують створення нового аукціону, появу більшої ставки, завершення аукціону, успішне виведення коштів. (лістинг 3.21).

Лістинг 3.21 – Основні події (Auction.sol)

```
event AuctionCreated(uint auctionId, string itemName, uint
endTime);
event NewHighestBid(uint auctionId, address bidder, uint
amount);
event AuctionEnded(uint auctionId, address winner, uint amount);
event Withdrawal(uint auctionId, address seller, uint amount);
```

Модіфікатори у мові Solidity на якій написан смарт-контракт дозволяють вставляти стандартні перевірки у декілька функцій. У цьому файлі створені модіфікатори роблять перевірку завершення аукціону. OnlyBeforEnd дозволяє виконати функцію лише до завершення аукціону, а

onlyAfterEnd навпаки дозволяє виконання тільки після завершення (лістинг 3.22).

Лістинг 3.22 – Створені модіфікатори (Auction.sol)

```
modifier onlyBeforeEnd(uint _auctionId) {
        require(block.timestamp < auctions[_auctionId].endTime,
"Auction already ended");
        _;
    }

    modifier onlyAfterEnd(uint _auctionId) {
        require(block.timestamp >= auctions[_auctionId].endTime,
"Auction not yet ended");
        _;
    }
```

Наступна функція для створення нових лотів, після її виклику перед тим як створити аукціон вона перевіряє чи ціна та тривалість торгів більша за 0, ініціалізується структура AuctionLot та викликається подія AuctionCreated. Усі важливі параметри задаються вручну (лістинг 3.23).

Лістинг 3.23 – Функція створення нового аукціону (Auction.sol)

```
function createAuction(
        string memory _itemName,
        uint _startingPrice,
        uint _biddingTimeInSeconds
    ) external {
        require(_startingPrice > 0, "Starting price must be
positive");
        require(_biddingTimeInSeconds > 0, "Bidding time must be
positive");

        auctionCounter++;
        auctions[auctionCounter] = AuctionLot({
            seller: payable(msg.sender),
            itemName: _itemName,
            startPrice: _startingPrice,
            highestBid: _startingPrice,
            highestBidder: payable(address(0)),
            endTime: block.timestamp + _biddingTimeInSeconds,
            ended: false,
            withdrawn: false
        });

        emit AuctionCreated(auctionCounter, _itemName,
```

```
block.timestamp + _biddingTimeInSeconds);
    }
```

Далі іде функція розміщення ставки, яка дозволяє користувачу підвищити поточну ставку. У цієї функції також є декалька перевірок, неможливість зробити ставку на свій лот, попередній учасник отримає свою ставку назад, якщо перша ставка то просто становлюється як вища (лістинг 3.24).

Лістинг 3.24 – Функція для розміщення ставок (Auction.sol)

```
function bid(uint _auctionId) external payable
onlyBeforeEnd(_auctionId) {
        AuctionLot storage auction = auctions[_auctionId];
        require(msg.sender != auction.seller, "Seller cannot
bid");
        require(msg.value > auction.highestBid, "Bid must be
higher than current highest bid");

        // Повернення попередньої ставки, якщо є попередній
учасник
        if (auction.highestBidder != address(0)) {
            (bool refundSuccess, ) =
auction.highestBidder.call{value: auction.highestBid}("");
            require(refundSuccess, "Failed to refund previous
highest bidder");
        }

        auction.highestBidder = payable(msg.sender);
        auction.highestBid = msg.value;

        emit NewHighestBid(_auctionId, msg.sender, msg.value);
    }
```

Завершальна функція у смарт-контракті, для завершення аукціонів. Коли термін аукціону закінчується, якщо були ставки кошти надходять до продавця, та викликається подія завершення. Викликається тільки коли endTime минув (лістинг 3.25).

Літсинг 3.25 – Функція завершеня аукціону (Auction.sol)

```
function endAuction(uint _auctionId) external
onlyAfterEnd(_auctionId) {
```

```
        AuctionLot storage auction = auctions[_auctionId];
        require(!auction.ended, "Auction already ended");

        auction.ended = true;

        if (auction.highestBidder != address(0)) {
            (bool success, ) = auction.seller.call{value:
auction.highestBid}("");
            require(success, "Failed to send funds to seller");

            auction.withdrawn = true;
            emit Withdrawal(_auctionId, auction.seller,
auction.highestBid);
        }

        emit AuctionEnded(_auctionId, auction.highestBidder,
auction.highestBid);
    }
```

Окрема функція створена для зчитування інформації про будь-який конкретний лот. Він не змінює стан контракту і не потребує комісії газу, тому його зручно використовувати для інтеграції з бекендом та фронтендом (лістинг 3.26).

Лістинг 3.26 – Функція для зчитування інформації про лоти (Auction.sol)

```
function getAuction(uint _auctionId) external view returns (
        address seller,
        string memory itemName,
        uint startPrice,
        uint highestBid,
        address highestBidder,
        uint endTime,
        bool ended,
        bool withdrawn
    ) {
        AuctionLot memory auction = auctions[_auctionId];
        return (
            auction.seller,
            auction.itemName,
            auction.startPrice,
            auction.highestBid,
            auction.highestBidder,
            auction.endTime,
            auction.ended,
            auction.withdrawn
        );
    }
}
```

4 ІНСТРУКЦІЯ КОРИСТУВАЧА

При завантаженні сторінки аукціону користувач одразу побачить головну сторінку (рисунок 4.1) де будуть відображені активні аукціони та заголовок з функціональними кнопками.

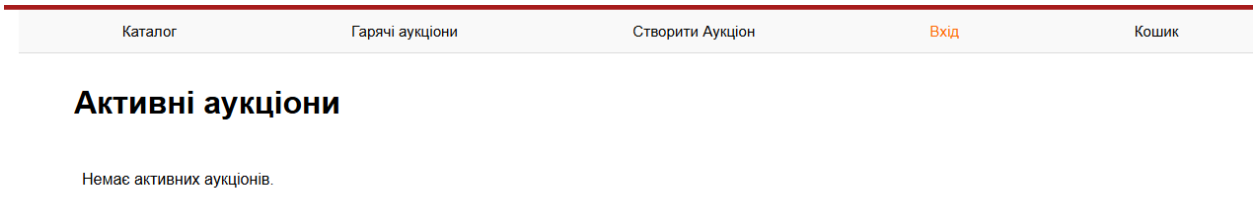

Рисунок 4.1 – Головна сторінка застосунку

Щоб взаємодіяти з аукціонами користувачу спочатку потрібно увійти до сайту. Це відбувається за допомоги розширення MetaMask, тож спочатку потрбіно його встановити до свого браузеру. Наприклад у браузері Chrome потрібно перейти до вкладки розширеня, управління розширенями та знайти розширення MetaMask (рисунок 4.2).

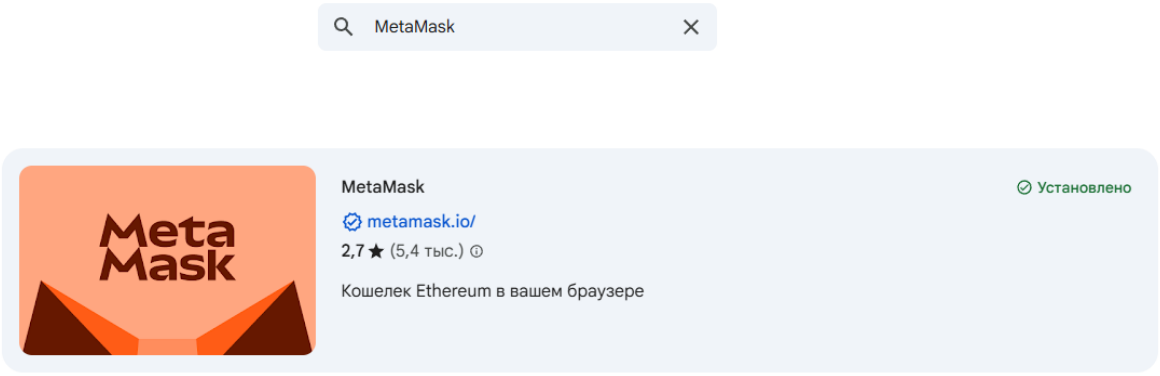

Рисунок 4.2 – Потрібне рощирення у магазині розширень Chrome

Після встановлення розширення вмикаємо його у браузері, переходимо до основної сторінки MetaMask та реєструємось. Після реєстрації додаємо свій гаманець Ethereum. Усе готово, тепер повертаємось до сайту натискаємо кнопку вхід та якщо усе було зроблено правильно відкриється вікно з підтвердженням підпису (рисунок 4.3), підписуємо його і тепер ми успішно авторизовані на сайті.

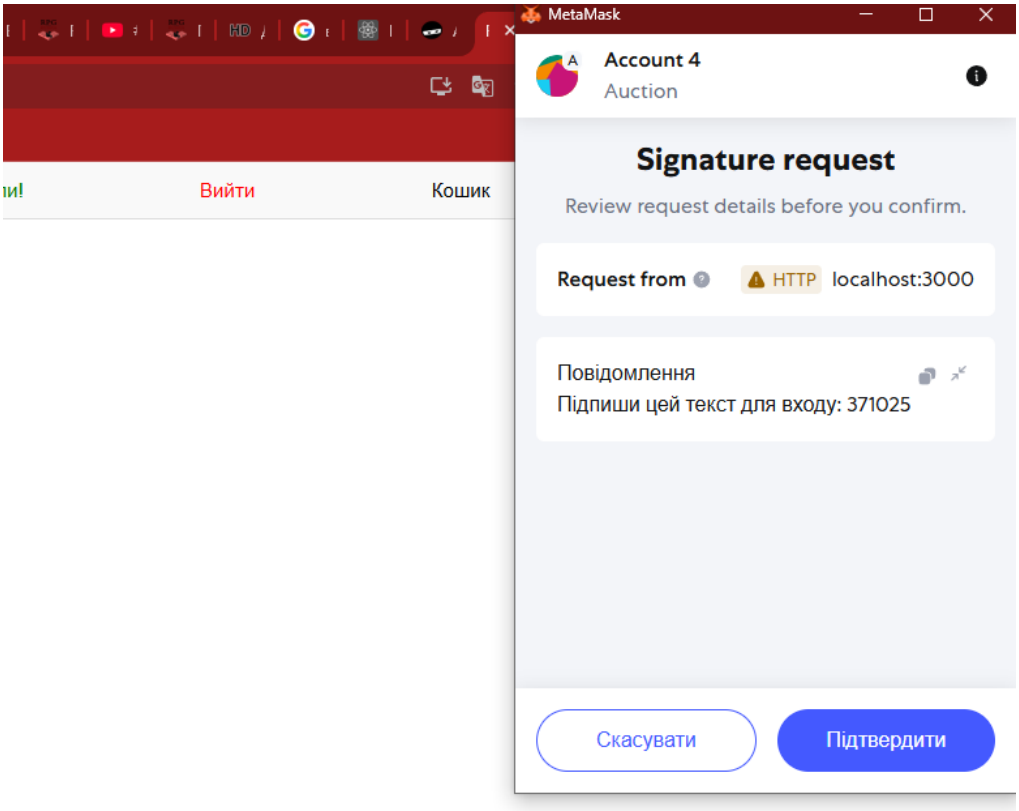

Рисунок 4.3 – Вікно підтвердження підпису

Після того як авторизувались до сайту відкривається можливість створювати аукціони, при натискані відповідної кнопки у заголовку. Відкриється вікно створення аукціонів (рисунок 4.4) де потрібно ввести необхідну інформацію про лот. Особливо важливо ввести коректні дані ставки та часу завершення аукціона, бо якщо вони будуть менше за 0 то висвітиться помилка та аукціон не буде створено. Також обов'язково потрбіно заповнити поля назви та опису лоту, фото не обов'язково прикріпляти, але якщо прикріпити це значно поліпшить візуальне сприйняття вашого лоту.

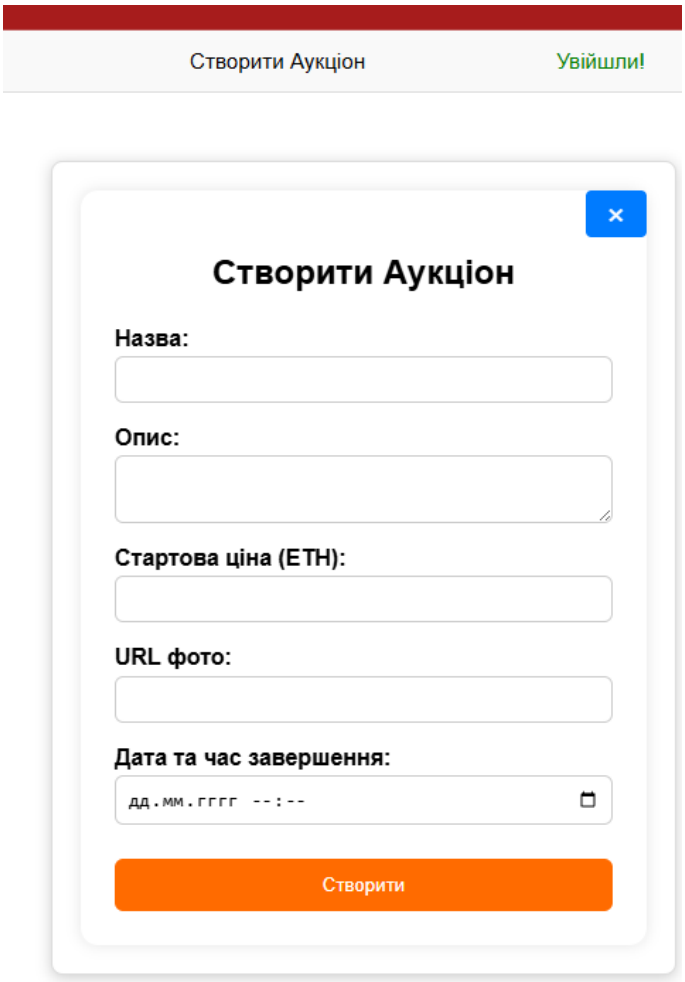

Рисунок 4.4 – Вікно створення аукціону

Після заповнення усіх полів, натискаємо створити, і тут розширення MetaMask знову попросить підпис, цей підпис для підтвердження комісії яку

приймає смарт-контракт за свій виклик, а також це слугує для повторної перевірки користувача. Підписуємо згоду на комісію транзакція проходить та лот створюється. Тепер після оновлення сторінки можна побачити тільки що створений лот (рисунок 4.5).

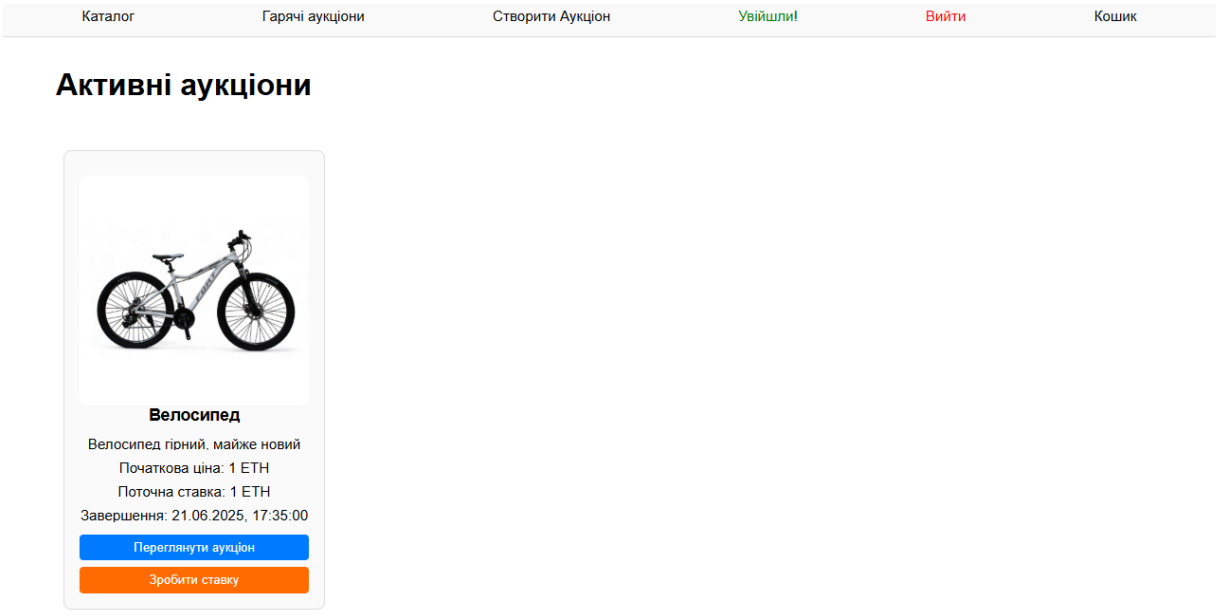

Рисунок 4.5 – Поява нового лоту на сторонці застосунку

Для тесту зайдемо на інший акаунт щоб тепер перевірити роботу розміщення нових ставок та перегляду аукціону. Тепер можна переглянути аукціон або зробити ставку. При перегляді лоту відкривається вікно де розписані усі відомості про лот, його фото та власник цього аукціону. При натисканні зробити ставку відкриється поле куди можна ввести свою ставку (рисунок 4.6). Важливо що неможливо ввести ставку меншу за 0 або меншу за поточну ставку.

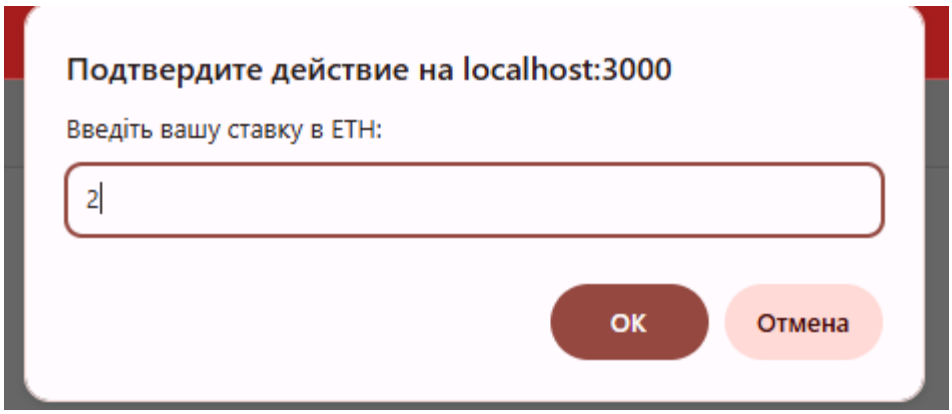

Рисунок 4.6 – Поле для введеня ставки

Якщо все було введено правильно, відкриється вікно яке попросить поставити підпис під транзакцією та кошти які були поставлені знімуться з рахунку та тимчасово перейдуть у смарт-контракт. Якщо вашу ставку переб’ють не хвилюйтесь усі кошти які ви поставили повернуться до рахунку. Якщо ж вашу ставку не переб’ють то кошти надійдуть продавцю після чого, буде змога зв’язатись з ним та домовитись про доставку товару.

ВИСНОВКИ

У результаті виконання кваліфікаційоної роботи був реалізований децентралізований застосунок для проведення онлайн аукціонів на базі технології блокчейн. Основною метою розробки було створити прозору, надійну та автоматизовану систему для проведення торгів, якак не потребує зайвих посередників та буде функціонувати за довіри публічному смарт-контракту.

Спершу було проведено аналіз предметної області, що дозволило обрати необхідні інструменти та методи для створення застосунку. Також визначити ключові вимоги до функціонування системи. В основі реалізації був обран смарт-контракт, який забезпечує повний цикл торгів, від створення лоту до повного завершення аукціону, з подальшим визначенням переможця та автоматичним переказом коштів до продавця. Кожен лот містить усю необхідну інформацію про товар,а також продавця та час завершення торгів. Система торгів базується на прицині стандартного англійского аукціону найвищої ставки, але з невиликою зміною, що при ставці гроші знімаються з гаманця, пока ставка не буде перебита. У випадку якщо ставку було перебито, кошти повертаються до гаманця попереднього учасника.

Особливу увагу також було приділено безпеці. Смарт-контракт містить різні перевірки, які не допускають до торгів, створення аукціонів та вивід коштів стороннім адресам. Також зроблена обробка різноматніх помилок які можуть виникнути у користувача при роботі із застосунком, кожна з яких супроводжується відповідним повідомленням.

У рамках роботи було реалізовано та протестовано усі аспекти застосунку. Робота серверної частини яка слугує ланцюжком що поєднує клієнтську частину та блокчейн. Завдяки добре структорованій логіці та підібраних методів, система працює синхронно з блокчейн-мережою, та гарантує, що увесь процес торгів буде проведнний від початку до кінця.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Drescher, D. Blockchain Basics: A Non-Technical Introduction in 25 Steps. – Apress, 2017. – 255 p.
2. Bashir, I. Mastering Blockchain: Unlocking the Power of Cryptocurrencies, Smart Contracts, and Decentralized Applications. – Packt Publishing, 2020. – 786 p.
3. Tapscott, D., Tapscott, A. Blockchain Revolution: How the Technology Behind Bitcoin and Other Cryptocurrencies is Changing the World. – Portfolio, 2016. – 384 p.
4. Lewis, A. The Basics of Bitcoins and Blockchains: An Introduction to Cryptocurrencies and the Technology that Powers Them. – Mango, 2020. – 408 p.
5. Arslanian, H., Fischer, F. Ethereum for Beginners: A Step-by-Step Guide to Learning Ethereum and the Blockchain Technology. – Independently Published, 2021. – 154 p.
6. Narayanan, A., Bonneau, J., Felten, E., Miller, A., Goldfeder, S. Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction. – Princeton University Press, 2016. – 336 p.
7. Laurence, T. Blockchain for Dummies. – Wiley, 2019. – 384 p.
8. Antonopoulos, A. M., Wood, G. Mastering Ethereum: Building Smart Contracts and DApps. – O'Reilly Media, 2018. – 384 p.
9. Modi, R. Solidity Programming Essentials: A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain. – Packt Publishing, 2018. – 254 p.
10. Infante, R. Building Ethereum DApps: Decentralized Applications on the Ethereum Blockchain. – Apress, 2019. – 352 p.
11. Dannen, C. Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners. – Apress,

2017. – 185 p.

12. Solorio, K., Kanna, R., Hoover, D. H. Hands-On Smart Contract Development with Solidity and Ethereum: From Fundamentals to Deployment. – O'Reilly Media, 2019. – 250 p.
13. Voshmgir, S. Token Economy: How the Web3 Reinvents the Internet. – Token Kitchen, 2020. – 350 p.
14. De Filippi, P., Wright, A. Blockchain and the Law: The Rule of Code. – Harvard University Press, 2018. – 256 p.
15. Tapscott, A. Smart Contracts: The Essential Guide to Using Blockchain Smart Contracts for Cryptocurrency Exchange, ICOs and Decentralized Finance. – Independently Published, 2020. – 112 p.
16. Banks, A., Porcello, E. Learning React: Modern Patterns for Developing React Apps. – O'Reilly Media, 2020. – 350 p.
17. Wieruch, R. The Road to React: Your Journey to Master Plain Yet Pragmatic React.js. – Independently Published, 2021. – 270 p.
18. Chevalier, A. Fullstack React: The Complete Guide to ReactJS and Friends. – Fullstack.io, 2017. – 836 p.
19. Flanagan, D. JavaScript: The Definitive Guide. Master the World's Most-Used Programming Language. – O'Reilly Media, 2020. – 706 p.
20. Freeman, E., Robson, E. Head First JavaScript Programming: A Brain-Friendly Guide. – O'Reilly Media, 2014. – 704 p.