МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ЕЛЕКТРОНІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Кваліфікаційна робота бакалавра

ЛОГІСТИЧНА СИСТЕМА НА БАЗІ СМАРТ-КОНТРАКТІВ

Здобувач освіти гр. ІН – 81 Богдан КЛИМЕНКО

Науковий керівник,
кандидат технічних наук,
доцент кафедри комп'ютерних наук Сергій ПЕТРОВ

Завідувач кафедри
доктор технічних наук, професор Анатолій ДОВБИШ

СУМИ 2022

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ЕЛЕКТРОНІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Затверджую_____
Зав. кафедрою Довбиш А.С.
"_____" _____2022 р.

ЗАВДАННЯ

до кваліфікаційної роботи

здобувача вищої освіти четвертого курсу, групи ІН-81 спеціальності «122 – Комп'ютерні науки» денної форми навчання Клименка Богдана Михайловича.

Тема: «ЛОГІСТИЧНА СИСТЕМА НА БАЗІ СМАРТ-КОНТРАКТІВ»

Затверджена наказом по СумДУ
№_____ від _____2022 р.

Зміст пояснювальної записки: 1) літературний огляд за обраною тематикою роботи; 2) постановка завдання для розробки; 3) вибір оптимальних інструментів для розробки логістичної системи на базі смарт-контракту; 4) практична реалізація.

Дата видачі завдання «_____»_____ 2022 р.
Керівник роботи _____Сергій ПЕТРОВ
Завдання прийняв до виконання _____ Богдан КЛИМЕНКО

РЕФЕРАТ

Записка: 65 стор., 53рис., 0 табл., 1 смарт-контракт, 50 джерел.

Об’єкт дослідження – смарт-контракти в мережі Ethereum.

Мета роботи – розробка логістичної системи на базі смарт-контракту, його тестування та подальше розгортання та в мережі Ethereum.

Методи дослідження – методи збору та аналізу даних.

Результати – розгорнутий смарт-контракт логістичної ситеми на Ethereum. Створені функції необхідні для коректної роботи контрату логістичної системи . Смарт-контракт розроблений за допомогою мови програмування Solidity на Remix Solidity IDE, версія компілятора 0.8.7.

ETHEREUM, БЛОКЧЕЙН, СМАРТ-КОНТРАКТ,

ЛОГІСТИЧНА СИСТЕМА, SOLIDITY.

Зміст

ВСТУП

Тенденція розвитку нових інформаційних і цифрових технологій простежується у всіх перспективних та ефективних сферах діяльності. Прогрес рухається семимильними кроками, разом з ним постійно та дуже швидко розвиваються та вдосконалюються весь технологічний світ.

Сьогодні одним із широко застосовуваних напрямів цифрових технологій є новітня технологія — блокчейн. Даний програмний продукт виступає у ролі розподіленої бази даних, завдяки якому інформація зберігається у вигляді впорядкованого ланцюжка блоків, що безперервно довшає. Таким чином, кожен блок мережі блокчейн містить хеш попереднього запису і поєднується один з одним задля забезпечення підтримки набору здійснених транзакцій завдяки способу розподілу, при цьому незалежно від довірених осіб.

31 жовтня 2008 року Сатоші Накамото було представлено протокол криптовалюти Bitcoin, який розглядає і впровадження блокчейн як окрему розподілену інфраструктурну технологію. Завдяки даній централізованій системі, користувачам надавалось право безпечно і надійно передавати цифрову валюту "Bitcoin" без застосування централізованого управління.

Зі зростанням інтересу до блокчейну та його впровадження в різних секторах і галузях, багато секторів представляють значні сфери застосування блокчейну: фінансові, бізнесові, промислові, голосування та багато інших освітніх і медичних застосувань [1], [2]. Проте блокчейн — це нова технологія, і хоча існує значне хвилювання щодо його потенційних переваг, існує також значна неточна інформація та невизначеність щодо потенційної корисності блокчейну загалом.

Оскільки блокчейн-сервіси та платформи набирають обертів, децентралізовані додатки (DApps) напевно ставатимуть дедалі популярнішими у найближчі роки. Все більше компаній та розробників

прагнуть використовувати ці технології для взаємодії з клієнтами через децентралізовану мережу.

Галузі ще належить створити такі додатки, які будуть прийняті масами, і, мабуть, це лише питання часу. Тому на ранніх етапах розвитку індустрії, для бізнесу як ніколи важливо почати думати про розробку та впровадження DApps, оскільки ці технології допомагають у конкурентній боротьбі.

РОЗДІЛ 1. АНАЛІЗ ТЕХНОЛОГІЇ

1.1 Блокчейн. Загальне поняття та характеристика

Блокчейн — одна з останніх розроблених технологій, що робить акцент на моделях та інноваціях Інтернету речей (IoT) та революцій у сфері штучного інтелекту [3]. Очікується, що блокчейн матиме вплив на всі галузі та створить можливість для покращення бізнес-процесів та формування довіри до обміну даними та управління записами у кожному секторі; наприклад, блокчейн може відігравати важливу роль у різних секторах із кількома додатками, такими як керування записами та обмін даними.

Технологію блокчейн можна визначити як незмінний розподілений реєстр, що розповсюджується на тисячі комп'ютерних систем. Технологія блокчейн [4] привернула неймовірну увагу після публікації білої книги Сатоші Накамото [5] у 2008 році. У ній Сатоші дав рішення проблеми подвійних витрат на цифрову валюту в децентралізованій P2P [6] мережі. Дубай призначив міністра, відповідального за штучний інтелект, з баченням стати першим у світі урядом, що працює на блокчейні [7].

Саудівська Аравія, одна з країн Близького Сходу, нещодавно оголосила про використання технології блокчейн [8] для кредитування частини ліквідності для вливання в банківський сектор. Розподілене формування консенсусу замінює роль Саудівської Аравії, однієї з країн Близького Сходу, яка нещодавно оголосила про використання технології блокчейн [8] для кредитування частини ліквідності, яка буде вливатися в банківський сектор. Розподілене формування консенсусу замінює роль довіреної третьої сторони в децентралізованих P2P мережах [9]. Ethereum є найбільш широко використовуваною платформою блокчейну Turing Complete [8], яка дозволяє розробникам писати смарт-контракт зі своїми власними випадковими правилами щодо власності, формату транзакції та функцій переходу стану.

Будь-хто може запустити вузол Ethereum [9], [10] на своїй машині для участі в мережі блокчейну Ethereum.

Блокчейн — це технологія, яка дозволяє користувачам перевіряти, зберігати та синхронізувати вміст книги транзакцій, яка реплікується між кількома користувачами. Іншими словами, блокчейн — Блокчейн, як технологія транзакційної бази даних, є децентралізованим способом керування перевіркою та, захищеними від несанкціонованого доступу, трансакціями з узгодженістю між значною кількістю учасників, також відомими як вузли [12], [13]. Ця властивість досягається за допомогою внутрішньої системи, де транзакції відмічаються часом у книзі; тому дані не можуть бути змінені або модифіковані без затвердження та оновлення книги. Ця технологія забезпечує безпеку та довіру під час здійснення транзакцій [14], [15]. Згідно з дослідженням, проведеним [13], блокчейн можна класифікувати як тип технології розподіленої книги, яка забезпечує впевненість користувача в тому, що інформація, яка архівується, наприклад, сертифікати, не підробляється. Різні дослідження показали, що блокчейн має здатність зменшувати неясність транзакцій, небезпечні стани та сумнівність, забезпечуючи повне розкриття транзакцій та доповнення однорідних і перевірених фактів для всіх учасників мережі [16]. Крім того, очікується, що технологія блокчейн глибоко оновить економіку та соціальний порядок за рахунок зниження вартості транзакцій і потреби в добре визнаних і надійних третіх сторін [17], [18]. Крім того, у дослідженні [19] зазначено, що цю технологію можна застосувати для запису деталей транзакції, зберігання медичних записів, укладення угод, що мають обов’язкову силу, відстеження руху товарів, зберігання індивідуальних записів кредитів, відстеження атрибуції творів мистецтва та перевірки платежі за допомогою ланцюга поставок разом із багатьма іншими процесами та процедурами.

1.2 Смарт-контракти: поняття, різновиди, платформи

Смарт-контракт [30] є одним із випадків використання технології блокчейн. Його представив Нік Сабо [31] у 1997 році. Це невеликий фрагмент програмного коду, що виконується самостійно [32]. Він розгорнутий на блокчейні і використовується для автоматичного забезпечення виконання умов між двома ненадійними сторонами. Протоколи Consensus [8], [33] використовуються для точної реалізації смарт-контрактів, інакше їх ефект буде анульовано.

1.2.1 Смарт-контракти, що працюють на Corda.

Corda — це опен-сорс блокчейн-платформа з відкритим вихідним кодом, яка спеціально розроблена для врахування суворо регульованого середовища індустрії фінансових послуг [20]. У Corda кожен вузол має сертифікат, який співставляє їх мережеві ідентифікатори з реальними юридичними особами. Зв’язок між вузлами Corda здійснюється напрямком, а історія транзакцій повністю зашифрована і приватна лише для необхідних сторін [21]. Смарт-контракти, що працюють на Corda, можуть складатися з коду та юридичної прози [22]. Пов’язану юридичну прозу можна було б повернути до традиційних правових систем у разі юридичних суперечок щодо виконання смарт-контракту. Розумний контракт у Corda має три ключові елементи, а саме виконуваний код, об’єкти стану та команди [23]. Виконуваний код в основному перевіряє зміни об’єктів стану в транзакціях. Об’єкти стану – це дані, які фіксують існування, зміст і поточний стан угоди між двома або більше сторонами та працюють як вхідні або вихідні дані транзакцій. Команди – це додаткові дані, які входять до транзакцій. В основному вони описують те, що відбувається, і вказують виконуваному коду, як перевірити транзакцію. Усі смарт-контракти можна було б запрограмувати на Kotlin або Java і зібрати у байт-код Java Virtual Machine (JVM).

1.2.2 Смарт-контракти в мережі та поза нею

Через природу технології блокчейн смарт-контракти, розгорнуті на блокчейнах (тобто смарт-контракти в ланцюжку), зазвичай мають виконуватися та перевірятися кожним вузлом, при цьому всі відповідні транзакції можуть бути видимими для всієї мережі блокчейн. Це знижує конфіденційність смарт-контрактів. Крім того, для смарт-контрактів, особливо тих зі складними обчисленнями, вартість транзакції може бути високою (наприклад, користувачі повинні сплачувати комісію за газ за транзакції в Ethereum), а перевірка відповідних транзакцій може зайняти багато часу (через повторне виконання смарт-контрактів). серед вузлів). В якості альтернативного рішення цих проблем була запропонована ідея розумного контракту «поза ланцюгом» [24], [25]. Смарт-контракти поза ланцюгом виконуються поза блокчейном. На відміну від ланцюгових смарт-контрактів, смарт-контракт поза ланцюгом повинен бути підписаний та виконаний лише зацікавленими учасниками. Як і пропонується, позамережний смарт-контракт, як правило, розроблений для інкапсуляції функцій, що передбачають високу вартість обчислень або приватну інформацію про учасників; тоді як для виконання деяких недорогих і нечутливих завдань пропонується використовувати смарт-контракт у мережі. Щоб зберегти властивості та переваги блокчейну, на практиці результати позачейнових смарт-контрактів, наприклад, реєструються в ланцюжку [26]. У разі будь-якої розбіжності щодо результатів виконання позаланцюгового смарт-контракту, він-ланцюжковий смарт-контракт може бути використаний для форк позамережного смарт-контракту та виконання його на блокчейні для вирішення спору [28].

1.2.3 Блокчейн-платформи для смарт-контрактів

Блокчейни можна розділити на загальнодоступні та непублічні категорії. Публічні платформи блокчейну дозволяють будь-якому користувачеві приєднатися до мережі, тоді як непублічні платформи блокчейну дозволяють

приєднуватися лише користувачам, яким дозволено приєднуватися. Прикладами публічних блокчейнів є Ethereum і NEO. Деякі приклади непублічних блокчейнів — Fabric і Quorum. Різні платформи блокчейн надають різну підтримку смарт-контрактів. Деякі (наприклад, біткойн) можуть дозволяти користувачам використовувати лише просту мову сценаріїв для розробки розумних контрактів із простою логікою; тоді як деякі платформи, такі як Ethereum, підтримують набагато більш просунуті мови програмування для написання смарт-контрактів [29].

1.3 Постановка задачі

Створити систему, на базі смарт-контракту, та розгорнути його на блокчейні ethereum. Додаток повинен бути абсолютно відкритим, незмінним, децентраліщованим та відповідно не регульованим. Тобто потрібно створити децентралізовану платформу, на базі якої, будь який розробник зміг би реалізувати свій додаток, точніше представницьку частину додатку, яка може бути реалізована будь яким способом – будь-то мобільний або серверний додаток чи веб сайт.

Система повинна працювати на заступним алгоритмом:

1. Замовник створює замовлення на доставку певного товару у певне місце.
2. Користувач може створити пропозицію на виконання замовлення.
3. Замовник може відповісти на пропозицію (погодитись, чи запропонувати внести якісь зміни).
4. Потенційний доставник створює рахунок для оплати послуг, де може вказати суму передплати та післяплати.
5. Замовник оплачує рахунок, а також може заблокувати ключем кошти, виділені на післяплату.
6. Доставник після того, як виконає замовлення може розблокувати рахунок та отримати післяплату.

7. Доставник може в будь-який час зроби повернення коштів замовнику
8. Після завершення операції замовник і доставник можуть дати відгук один про одного та поставити оцінку для рейтингу.

РОЗДІЛ 2. ОСНОВНІ ПОЛОЖЕННЯ ТА ЗАГАЛЬНІ ТЕОРИТИЧНІ ВІДОМОСТІ

2.1 Смарт-контракти на базі системи ethereum

Основними компонентами смарт-контрактів Ethereum є функції, події та змінні стану [6]. Повнота мови програмування Solidity [8] робить її ідеальною для написання смарт-контракту. Після компіляції код смарт-контракту перетворюється в байт-код EVM [34], [35] і зберігається в блокчейні Ethereum

Для розгортання смарт-контракту в Ethereum виконується спеціальна транзакція створення, яка вводить контракт в блокчейн. Під час цієї процедури контракту присвоюється унікальна адреса у вигляді 160-бітового ідентифікатора, а його код завантажується в блокчейн. Після успішного створення смарт-контракт складається з адреси контракту, балансу контракту, попередньо визначеного виконуваного коду та стану. Потім різні сторони можуть взаємодіяти з конкретним контрактом, надсилаючи транзакції, що викликають контракт, на відому адресу контракту. В результаті вони можуть ініціювати будь-яку кількість дій, таких як читання та оновлення стану контракту, взаємодія та виконання інших контрактів або передача цінності іншим. Транзакція залучення контракту має включати плату за виконання, а також може включати передачу ефіру від абонента до контракту. Крім того, він також може визначати вхідні дані для виклику функції. Після прийняття транзакції всі учасники мережі виконують код контракту, беручи до уваги поточний стан блокчейну та дані транзакції як вхідні дані. Потім мережа узгоджує вихід і наступний стан контракту, беручи участь у протоколі консенсусу. Таким чином, на концептуальному рівні Ethereum можна розглядати як кінцевий автомат на основі транзакцій, стан якого оновлюється після кожної транзакції.

Обліковий запис Ethereum Smart Contract складається з [36] його виконуваного коду, адреси контракту, стану, що складається з приватного

сховища, і балансу у вигляді віртуальних монет (Ефір). Розумний контракт можна викликати за допомогою контракту, який викликає транзакції на його неповторну адресу, з деякими параметрами, такими як дані виклику та оплата в ефірі як комісія за транзакцію (газ) [37]. Віртуальна машина Ethereum або EVM [38] — це повноцінна віртуальна машина на основі стека Тьюринга. Він забезпечує середовище виконання, яке ізольоване від мережі для виконання коду смарт-контракту. Обчислення в EVM [39] по суті обмежені комісіями за транзакції. Смарт-контракти містять в якості балансу ефіри (вид криптовалюти). Ефіри можна надсилати іншим контрактам для виконання смарт-контракту. Це дуже важливе завдання, оскільки будь-який тип уразливості може стати причиною втрати мільйонів ефіру [39]. Розумний контракт не може бути змінений після розгортання [40] на блокчейні через незмінну [41] природу блокчейну. Тому під час кодування смарт-контрактів розробники смарт-контрактів повинні враховувати вразливості безпеки та найкращі методи.

З моменту випуску в липні 2015 року Ethereum став найпопулярнішою платформою блокчейн для смарт-контрактів [42]. В Ethereum люди можуть використовувати мови програмування, наприклад, Solidity і Vyper, для розробки складних додатків для смарт-контрактів. Усі розумні контракти, написані мовами високого рівня, будуть скомпільовані в однаковий формат, тобто байт-код Ethereum, і виконуватимуться EVM. Ethereum також має свою власну криптовалюту, а саме Ether. Ефір можна переносити між обліковими записами та використовувати для компенсації учасникам, які майніють блоки за виконані обчислення [43].

2.1.1 Газ (комісія за транзакцію)

Ethereum використовує внутрішній механізм ціноутворення, тобто газ для всіх транзакцій, що виконуються на ньому [43]. Газ – це міра того, скільки обчислювальних ресурсів коштуватиме транзакція. Люди повинні сплачувати

комісію за газ (в ефірах) за кожну транзакцію, яку вони здійснюють; і транзакція буде невдалою, якщо в неї закінчиться газ. Якщо користувачі хочуть, щоб їхні транзакції видобували майнери швидше, вони можуть підвищити ціну на газ. Використовуючи газовий механізм, Ethereum може краще розподіляти ресурси та пом'якшувати спам у мережі.

2.2 Розробка смарт-контрактів

Розробка контракту на блокчейні Ethereum вимагає іншого інженерного підходу, ніж більшість веб- та мобільних розробників. На відміну від сучасних мов програмування, які підтримують широкий спектр зручних типів даних для зберігання та маніпуляції, розробник відповідає за внутрішню організацію та маніпулювання даними на більш глибокому рівні. Це означає, що розробник повинен розглянути деталі, з якими він не може мати справу. Наприклад, розробнику доведеться реалізувати метод з'єднання рядків або рядків у нижньому регістрі, що є завданнями, про які розробникам зазвичай не доводиться думати іншими мовами. Крім того, платформа Ethereum і Solidity постійно розвиваються швидкими темпами, і розробник стикається з постійною трансформацією функцій платформи та ландшафту безпеки, оскільки додаються нові інструкції та виявляються помилки та ризики безпеки. Розробники повинні враховувати, що написаний сьогодні код, ймовірно, не буде скомпільований за кілька місяців або, принаймні, доведеться переробити.

Смарт-контракти в Ethereum зазвичай пишуться мовами вищого рівня, а потім компілюються в байт-код EVM. Такими мовами вищого рівня є LLL (Low-level Lisp-like Language) [44], Serpent (подібна мова Python) [45], Viper (мова, схожа на Python) [46], та Solidity (подібна Javascript). мова) [47]. LLL і Serpent були розроблені на ранніх стадіях платформи, тоді як Viper зараз знаходиться в стадії розробки і призначений для заміни Serpent. Найвідомішою і широко поширеною мовою є Solidity.

2.2.1 Solidity

Solidity — це високорівнева мова програмування за Тьюрингом із подібним синтаксисом JavaScript, яка статично типізована, підтримує спадковість і поліморфізм, а також бібліотеки та складні типи, визначені користувачем. При використанні Solidity для розробки контрактів структура контрактів схожа на класи в об’єктно-орієнтованих мовах програмування. Код контракту складається зі змінних і функцій, які зчитують і змінюють їх, як у традиційному імперативному програмуванні.

Solidity визначає спеціальні змінні (msg, block, tx), які завжди існують у глобальному просторі імен і містять властивості для доступу до інформації про транзакцію виклику та блокчейн. Наприклад, ці змінні дозволяють отримати вихідну адресу, кількість ефіру та дані, надіслані разом із транзакцією виклику.

Ще однією особливою зручною функцією Solidity є так звані модифікатори. Модифікатори можна описати як закриті блоки коду, які збагачують функції, щоб змінити їх потік виконання коду. Цей підхід дотримується парадигми умовно-орієнтованого програмування (COP), основною метою якої є видалення умовних шляхів у тілах функцій. Модифікатори можна використовувати для легкої зміни поведінки функцій і застосовуються, вказуючи їх у списку, розділеному пробілами після імені функції. Типовим варіантом використання модифікаторів є перевірка певних умов перед виконанням функції.

Додатково важливою і акуратною особливістю Solidity є події. Події – це сигнали про те, що розумні контракти можуть запускатися. Інтерфейси користувача та програми можуть слухати ці події в блокчейні без особливих витрат і діяти відповідно. Крім цього, події також можуть служити цілям реєстрації. Коли вони викликані, вони зберігають свої аргументи в журналі транзакцій, спеціальній структурі даних в блокчейні, яка відображається аж до

рівня блоку. Ці журнали пов’язані з адресою контракту, і до них можна ефективно отримати доступ за межами блокчейну.

2.2.2 Середовище розробки

Для розробки, компіляції, тестування створеного смарт-контракту будемо використовувати середовище розробки Remix IDE.

Remix IDE [48] - це програмне забезпечення для розробки веб-версій Ethereum для розробки смарт-контрактів Solidity. Розробник контракту розробляє його на віртуальній машині JavaScript, представленій у Remix. Після налагодження контракту його можна опублікувати в Ethereum або будь-якому блокчейні, який підтримує смарт-контракти Solidity.[49] Це середовище розробки використовується на всьому шляху розробки смарт-контрактів користувачами на кожному рівні знань. Він не вимагає налаштування, сприяє швидкому циклу розробки та має багатий набір плагінів з інтуїтивно зрозумілим графічним інтерфейсом. IDE поставляється в двох варіантах (веб-програма (Рисунок 2.2) або настільна програма (Рисунок 2.1)) і як розширення VSCode.[48].

Дане середовище розробки дозволяє:

а. Створювати смарт-контракти (рисунок 2.3)
б. Компілювати смарт-контракти (рисунок 2.4)
в. Розгортати смарт-контракти (рисунок 2.5)
г. Тестувати смарт-контракти (рисунок 2.6)

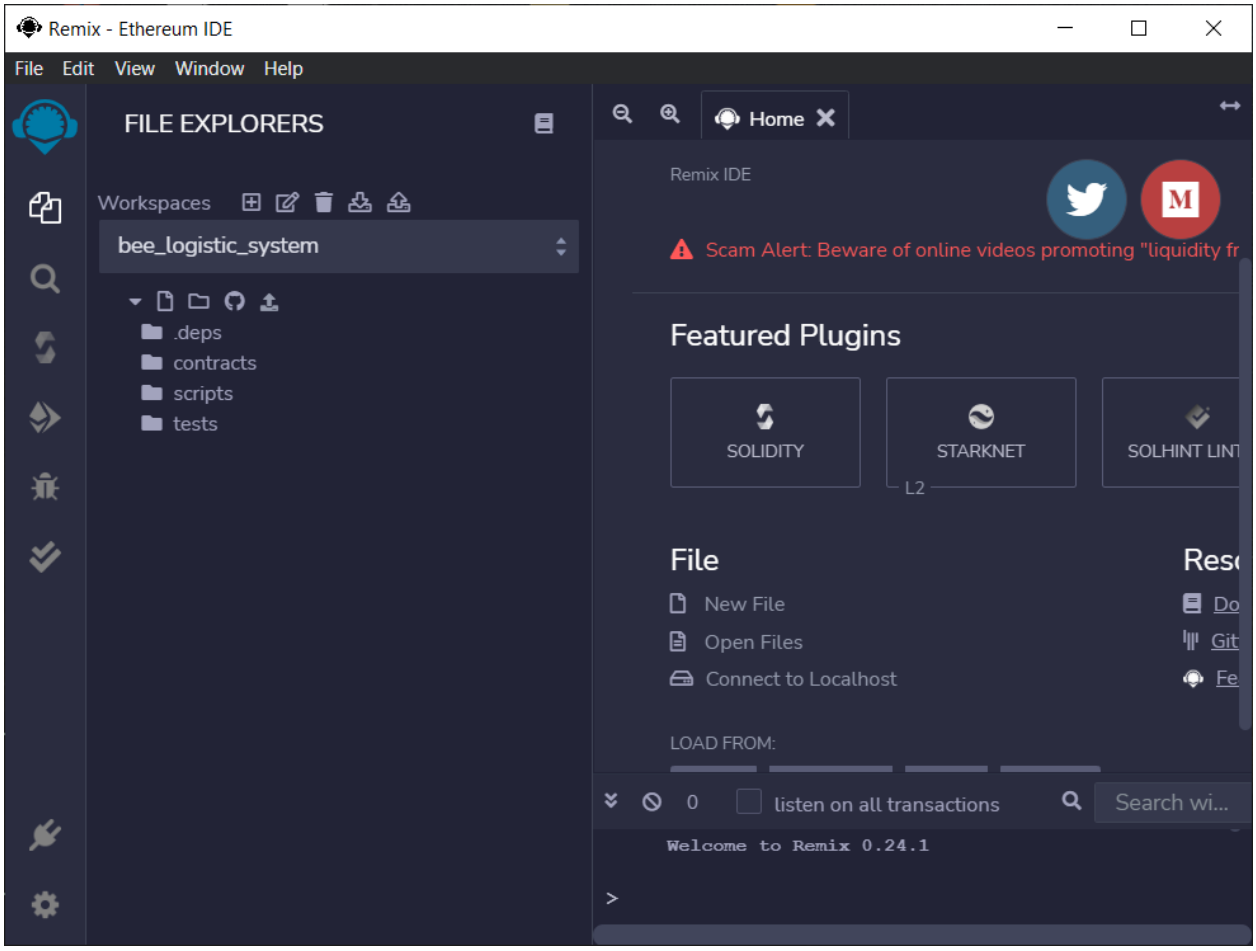

Рисунок 2.1 – Remix IDE desktop

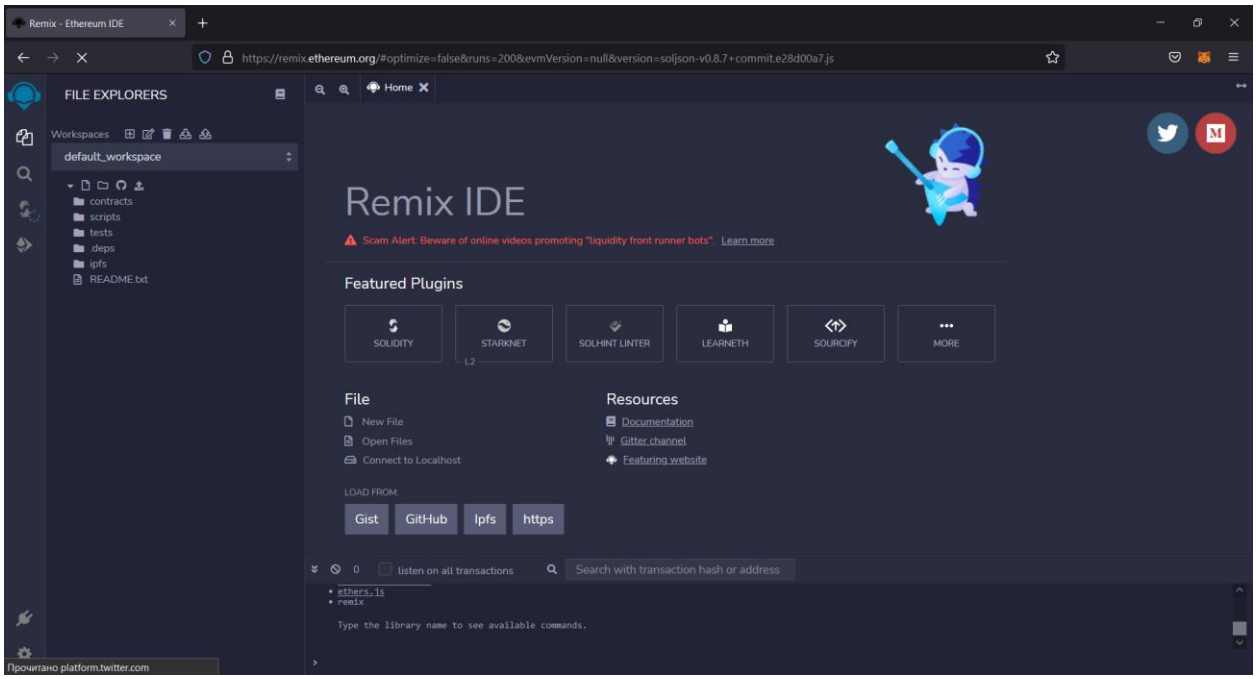

Рисунок 2.2 – Remix IDE web

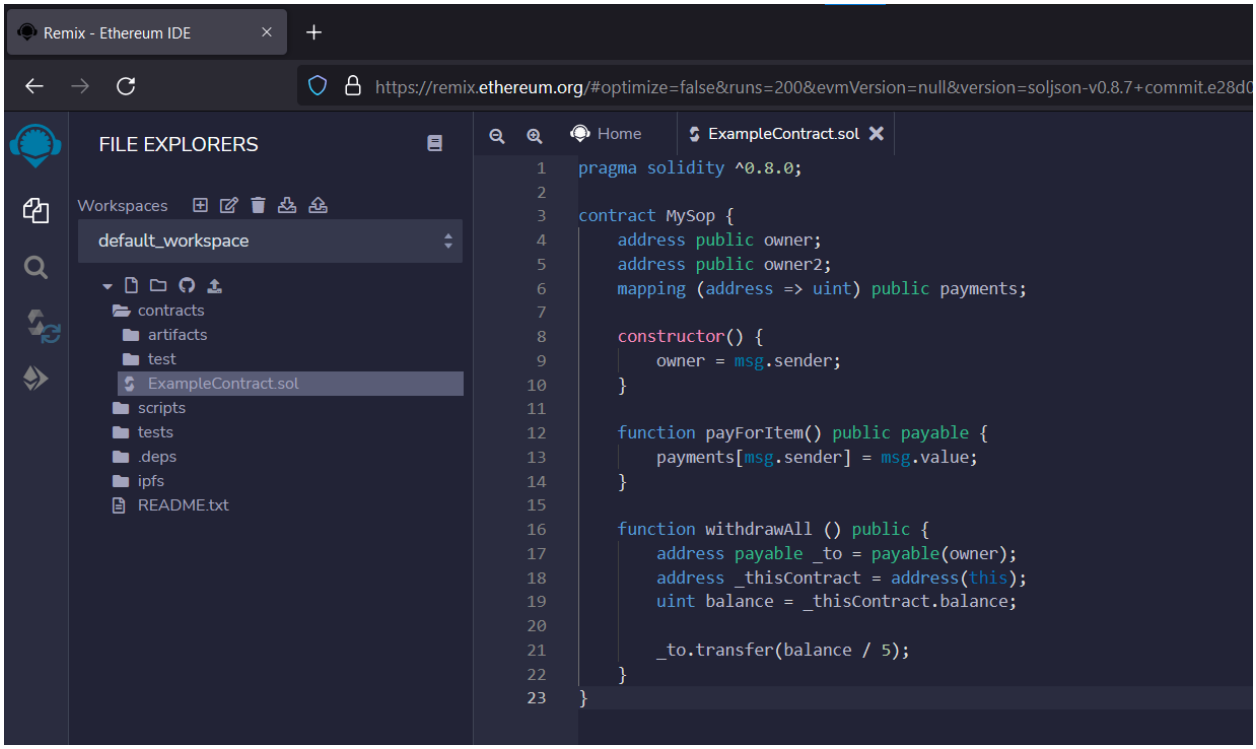

Рисунок 2.3 – Remix IDE створення контракту

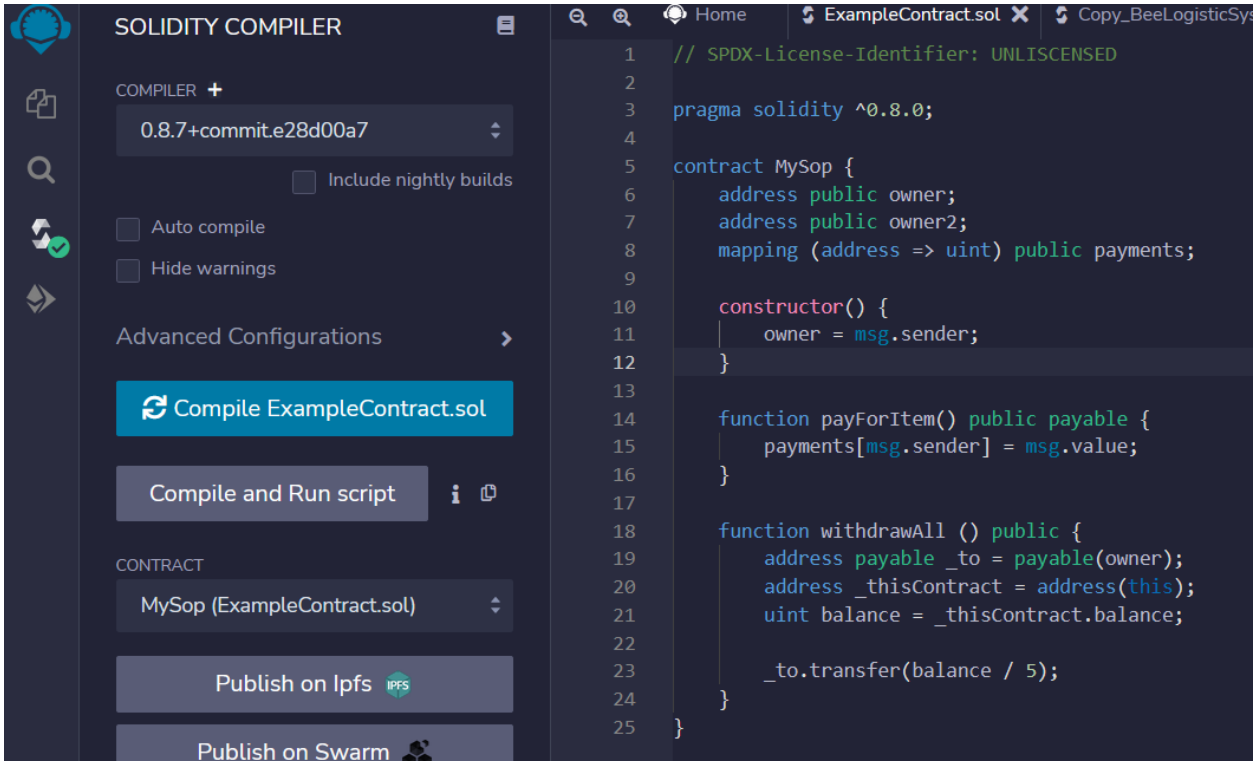

Рисунок 2.4 – Remix IDE компілювання контракту

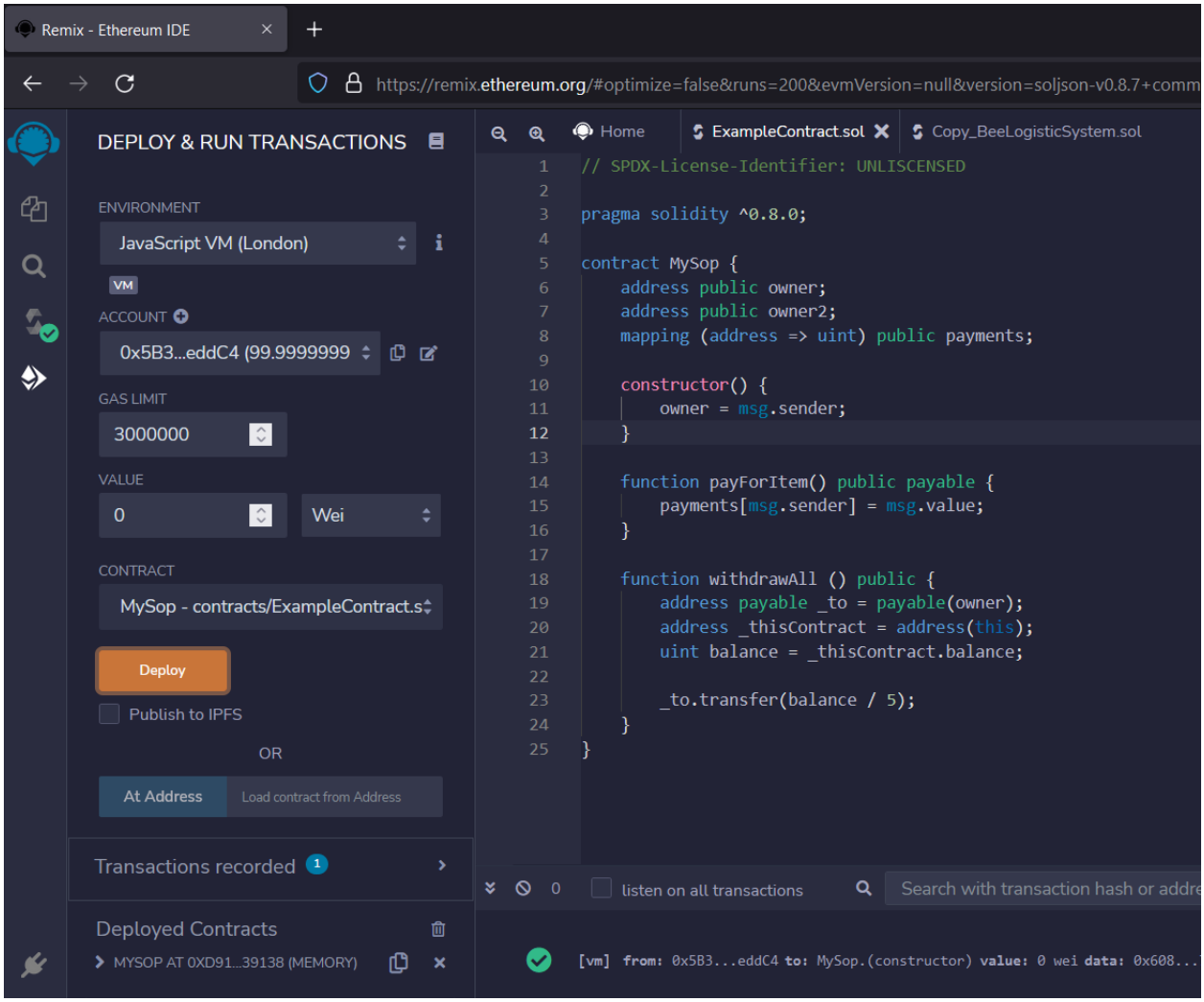

Рисунок 2.5 – Remix IDE розгортання контракту

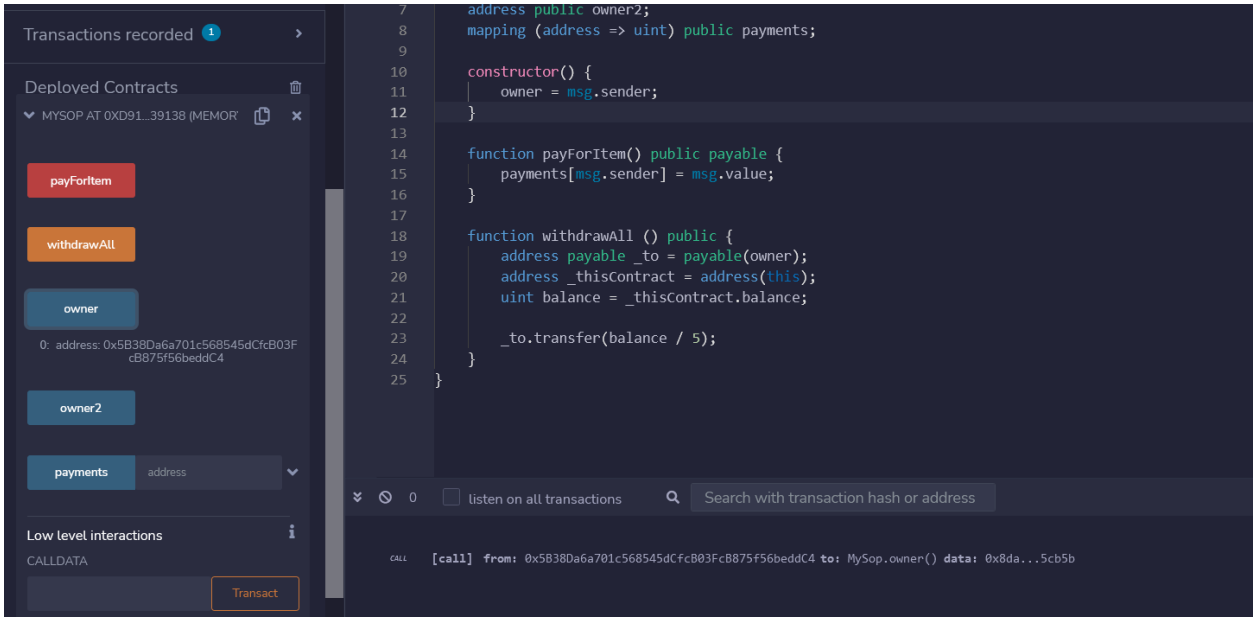

Рисунок 2.6 – Remix IDE тестування контракту

2.2.3 Розгортання в локальній мережі

Для того, щоб розгорнути локально створений смарт-контракт використаємо програмне забезпечення Ganache.

Ganache — це програмне забезпечення для розгортування персонального блокчейну (рисунок 2.8) для швидкої розробки розподілених додатків Ethereum і Corda. Ми можемо використовувати Ganache протягом усього циклу розробки; що дозволяє розробляти, розгортати та тестувати свої dApps у безпечному та детермінованому середовищі [50].

Ganache доступний у двох варіантах: UI та CLI. Ganache UI — це програма з користувацьким інтерфейсом, яка підтримує технології Ethereum і Corda (рисунок 2.7). Інструмент командного рядка ganache-cli (раніше відомий як TestRPC) доступний для розробки Ethereum [50]. Для розгортання смарт-контракту віддаємо перевагу додатку з інтерфейсом.

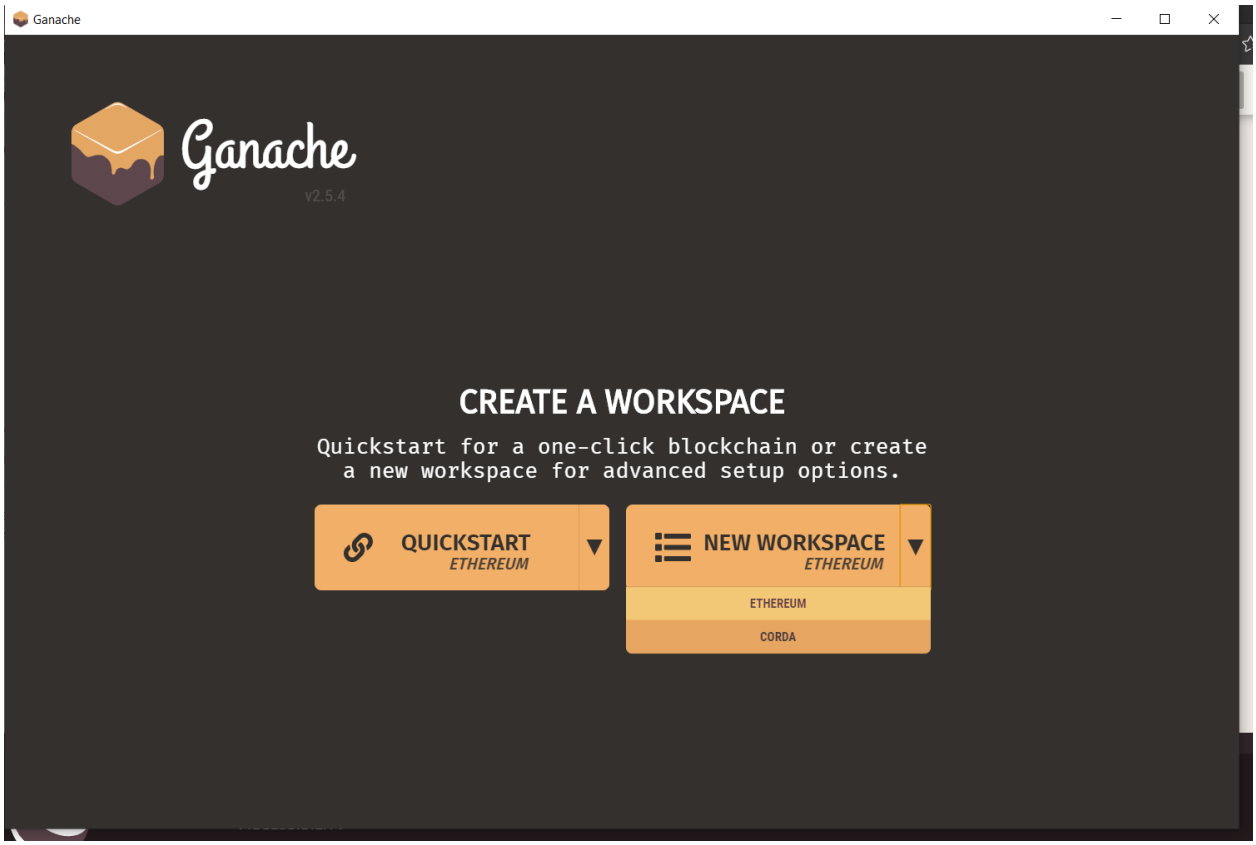

Рисунок 2.7 – Ganache початкове вікно

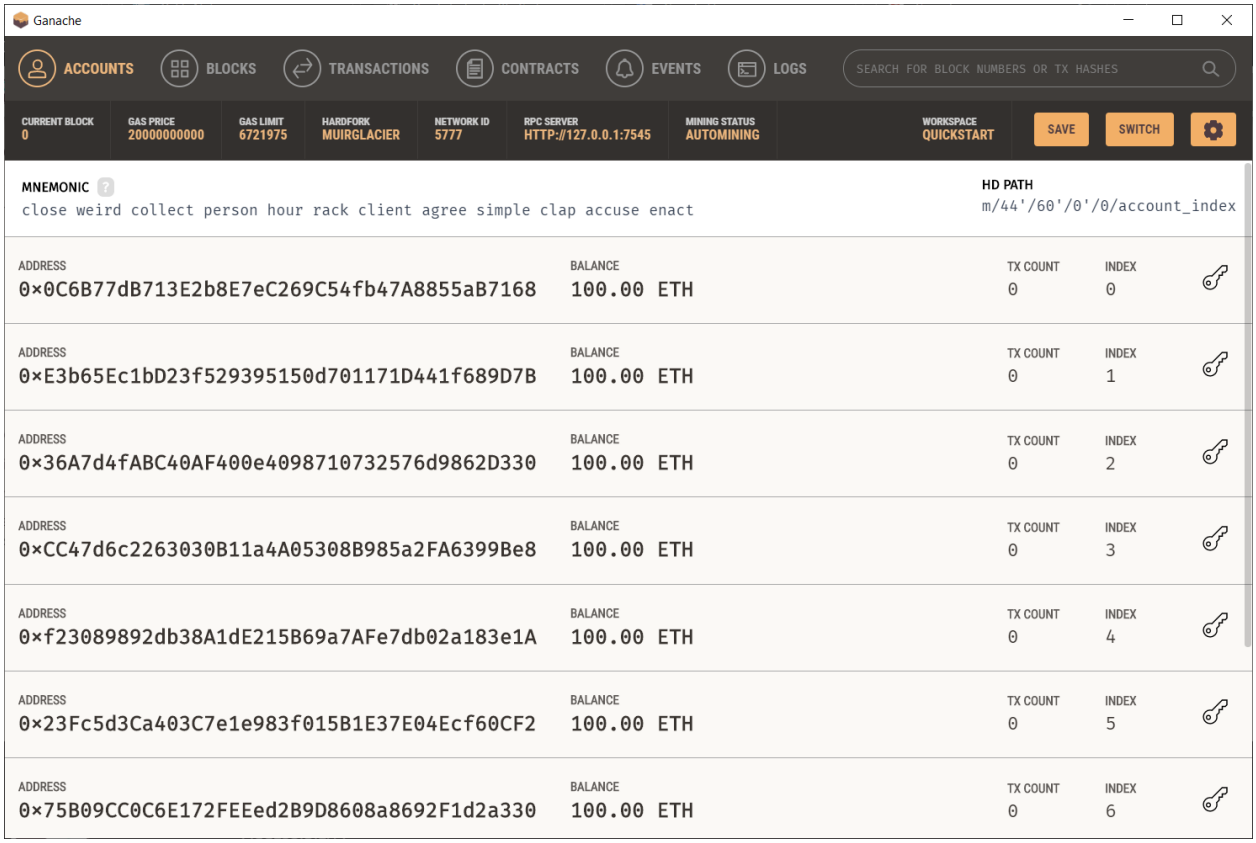

Рисунок 2.8 – Ganache створений локальний блокчейн з акаунтами

РОЗДІЛ 3. ПРОГРАМНА РЕАЛІЗАЦІЯ

3.1 Опис створеного смарт-контракту

3.1.1 Основні змінні

Для того що створити смарт-контракт нашої системи скористаємося середовищем розробки Remix IDE. Для початку роботи створюємо робочий простір та приступаємо до розробки. Загальний код контракту винесено до додатку А, розберемо лише основні, ключові моменти.

Почнемо зі змінних, які відповідають за зберігання основної інформації: за зберігання замовлень – orders; офферів – offers; виставлених рахунків – invoices; та статистичної інформації за адресами користувачів – statistics (Рисунок 3.1). Крім цього до публічних змінних відносяться змінні, що зберігають інформацію про токен, створений для нашої системи – beeLogisticSystemToken. У користувачів систему є можливість його придбати та використовувати для платежів. З платежів, що будуть використовувати наш токен, ми не будемо брати комісію. З інших же платежів (на даний момент реалізована оплата за допомогою ефіру) на користь фонду нашого контракту beeLogisticSystemFoundation будет утримуватись комісія у розмірі BEETOKEN_FEE (на даний момент це один відсоток від суми платежу, але коли контракт буде задеплоєно, цей відсоток неможливо буде змінити, він залишиться таким аж «до кінця життя нашої системи»). Подивитись яку кількість валюти ми можемо перевести у наш фонд допоможе змінна fees.

```
ERC20 public beeLogisticSystemToken; // адреса токену, що був творений для нашої системи
address public beeLogisticSystemFoundation; // адреса рахунку, який ми виристовуємо для збору коштів на наш розвиток
uint8 public constant BEETOKEN_FEE = 1; // відсоток, який буде зніматися за використання нашої системи (тольки для ETH)
uint public fees = 0; // сума коштів, що можна зняти на користь фонду нашого розвику

Order[] public orders; // зберігаємо замовлення
Offer[] public offers; // зберігаємо пропозиції
Invoice[] public invoices; // зберігаємо інформацію про рахунки
mapping (address => Statistics) internal statistics; // зберігаємо зібрану статистику
```

Рисунок 3.1 – Основні змінні

3.1.2 Стуктури та статуси

Розглянемо структури створеного смарт-контракту. На рисунку 3.2 зображено структуру Order. За таким шаблоном полів зберігаються дані.

```
struct Order {
  address client; // ethereum-адреса клієнта - замовника
  string from; // адреса аккаунту або координати місця, з відки потрібно забрати замовлення
  string to; // адреса аккаунту або координати місця, куди потрібно виконати доставку
  string packageDescription; //опис об'єкта, що має бути доставлений
  uint expires;  // терміни достаки в юнікс часові (секунди)
  string message; // текст додаткогово повідомлення для замовленя (що завгодно)

  uint[] offerIds; // список номерів пропозицій

  address supplier; // ефіріум-адреса вибраного достащика
  uint invoiceId; // прикріплений оплачений рахунок-фактура

  EnumOrderStatus status; // список виконання замовлення
  uint createdAt; // дата створення замовлення
  uint updatedAt; // дата внесення змін
}
```

Рисунок 3.2 – Структура замовлення

Коли клієнт робить замовлення він повинен вказати адресу місця куди потрібно виконати доставку - to, місце, де потрібно взяти об’єкт для передачі - from, зробити його описання (розміри, вага тощо) - packageDescription, час, за який потрібно виконати доставку - expires, та має можливість вказати додаткове повідомлення (будь яка інформація, що може бути корисна для доставника) - message. Крім цього в структурі замовлення зберігається інформація про список пропозицій на виконання – offersIds, ефіріум-адресу клієнта-замовника – client, ефіріум-адресу підрядника – supplier, айді оплаченого рахунку-фактури – invoiceId, у вигляді перелічуваного типу даних статус виконання замовлення – status, та дані про створення та оновлення інформації про замовлення – createdAt, updatedAt.

Наступним кором розбираємо структуру Offer пропозиції на доставку від потенційного доставника (Рисунок 3.3).

```
86
87    struct Offer {
88        address supplier; // ефіріум-адреса достащика
89        uint orderId; // айді замовлення
90        string message; // текст пропозиції
91        string response; // текст відповіді
92        uint createdAt; // дата створення пропозиції
93        uint updatedAt; // дата внесення змін
94    }
95
```

Рисунок 3.3 – Структура офферу

Якщо потенційний виконавець зацікавився замовленням, він може запропонувати офер на виконання замовлення. Для того, що створити пропозицію на виконання потрібно задати текст пропозиції та відправити його замовнику. Якщо замовника все влаштовує, або потрібно внести корективи , то замовник відправляє відповідь виконавцю. Ця відповідь заноситься до офферу та буде зберігатися. Всього зберігається шість полів: ефіріум-адреса виконавця – supplier, айді замовлення – orderId, текст пропозиції – message, текс відповіді – response, дані про створення та оновлення інформації про оффер – createdAt, updatedAt.

Коли замовник та підрядник домовляться буде сформований рахунок-фактура. Його повинен сформувати виконавець. Для цього йому потрібно занести дані про номер замовлення, величину передплати та післяплати, валюту (с вибраних), якою він хоче отримати оплату, час, за який зобов’язується виконати замовлення. Всі ці дані зберігаються в структурі рахунку фактури Invoice (Рисунок 3.4).

Системою також ведеться збір статистичних даних за кожним замовленням. Для цього використовується структура статистики Statistics (Рисунок 3.5). Вона зберігає дані про замовлення – orders, загальну оцінку - rateSum, кількість оцінок rateCount та відгуки про користувача - feedbacks.

```
struct Invoice {
  address sender; // ефіріум-адреса достащика
  uint orderId; // айді замовлення
  uint prepayment; // розмір передоплати
  uint payment; // розмір оплати
  EnumCurrency currency; //валюта, що була вибрана для оплати
  uint expires; // терміни дії рахунку в юнікс часові (секунди)
  bytes32 paymentHash; //хеш ключа, що буде використаний для того, що розблокувати кошти оплати
  EnumInvoiceStatus status; // статуси рахунка
  uint createdAt; // дата створення
  uint updatedAt; // дата внесення змін
}
```

Рисунок 3.4 – Структура рахунку-фактури

```
struct Statistics {
  uint[] orders; // зберігаємо номери замовлення, в яких користувач брав учать в ролі доставщика або клієнта
  uint rateSum; // рейтинг користувача
  uint rateCount; // загальна кількість оцінок
  mapping (uint => Feedback) feedbacks; //відгуки про замовлення (айді - відгук)
}
```

Рисунок 3.5 – Структура статистики

Остання структура, це Feedback, вона використовується для зберігання інформації про відгуки (Рисунок 3.6)

```
struct Feedback {
  uint8 rate; // бали, в межах 5 для оцінки
  string text; // тіло відгуку
  uint createdAt; //дата створення
}
```

Рисунок 3.6 – Структура відгуку

Наступним кроком створення контракту є визначення статусів (Рисунок 3.7), вони потрібні для того щоб оперувати інформацією про статуси замовлення – EnumOrderStatus, статуси оплати рахунків EnumInvoiceStatus та варіанти криптовалюти для оплати послуг – EnumCurrency.

```
enum EnumOrderStatus { New, Process, Performed }
enum EnumInvoiceStatus { New, Settled, Closed, MoneyBack }
enum EnumCurrency { ETH, BEETOKEN }
```

Рисунок 3.7 – Статуси

3.1.3 Реалізовані функції

Для огляду наведено лише сигнатури функцій, всі тіла функцій, і взаналі весь код контракту, розмішено у додатку А.

3.1.3.1 Функція addOrder

Першою розглянемо сигнатуру функції addOrder (Рисунок 3.8), вона потрібна для того, щоб створити замовлення. Приймає на вхід п’ять значень: адресу, звідки відправляємо, куди потрібно доставити, опис та характеристики вантажу, час, за який потрібно це зробити та повідомлення для потенційного виконавця. Результатом роботи функції стане сформоване замовлення order.

```
function addOrder(
    string memory from,
    string memory to,
    string memory packageDescription,
    uint expires,
    string memory message
) public {
```

Рисунок 3.8 – Створення замовлення

3.1.3.2 Функція addOffer

Функція addOffer (Рисунок 3.9) використовується для того, щоб потенційний доставник міг відповісти на замовлення, яке хотів би виконувати. Тому ця функція приймає два значення: номер айді замевлення, на яке буде відповідь та повідомлення для клієнта (наприклад тут можна оговорити ціну, за яку доставник може доставити вантаж, або уточнити щось по останньому). Результатом виконання буде сформований offer.

```
function addOffer(
    uint orderId,
    string memory message
) public {
```

Рисунок 3.9 – Створення пропозиції на виконання

3.1.3.3 Функція addResponse

Функція addResponse (Рисунок 3.10) використовується для того, щоб замовник міг відповісти на пропозицію доставника. Уточнити запрошені деталі, запропонувати ціну тощо. На вхід приймає два значення: номер айді офферу на який відповідає клієнт та повідомлення для виконавця. Результатом виконання буде оновлено поле у замовленні, що відповідає за відповідь клієнта.

```
function addResponse(
    uint offerId,
    string memory message
) public {
```

Рисунок 3.10 – Створення відповіді на пропозицію

3.1.3.4 Функція addInvoice

Функція addInvoice (Рисунок 3.11) використовується для того, щоб виконавець міг створити рахунок оплати послуги, який потім буде оплачено замовником. При чому, ця функція передбачає роботу як з передоплатою так і з післяплатою. Як це працює. Функція на вхід отримує п’ять значень: номер замовлення, сума валюти для передплати, сума валюти для післяплати, тип валюти для оплати (на даний момент ефір або токен нашої системи) та час, за який виконавець зобов’язується надати послугу. Коли доставник створює інвойс, він, за домовленістю з замовником, розбиває оплату на дві частини: передплату та післяплату. Коли останній оплачує замовлення, та частина грошей, що виділена на передплату – в той же час переказується на рахунок кур’єра, та ж частина що виділена на післяплату, блокується системою. Коли клієнт оплачує замовлення він може створити фразу, яка буде захешована за алгоритмом Keccak-256 та стане паролем від заблокованої валюти. Після виконання замовлення клієнт передає ключ виконавцю, рахунок розблоковується, гроші повертаються замовнику. У випадку, якщо щось пішло

не так, угода через якісь проблеми буде порушена, кошти можуть бути повернуті клієнту з заблокованого рахунку, але про це докладніше пізніше.

```
function addInvoice(
  uint orderId,
  uint prepayment,
  uint payment,
  EnumCurrency currency,
  uint expires
) public {
```

Рисунок 3.11 – Створення рахунку для оплати замовлення

3.1.3.5 Функція pay

Функція pay (Рисунок 3.12) використовується для того, щоб замовник міг виконати оплату замовлення. Ця функція приймає два значення – айді рахунку, який потрібно оплатити та захешовану фразу для блокування та деблокування частини грошей на післяплату. Крім цього, ця функція типу payable – вона очікує надходження коштів разом з даними. Якщо щось буде задано не правильно (не той айді інвойсу, валюти буде більше ніж потрібно, або менше ніж потрібно, не той тип валюти тощо) то транзакція не буде виконана, гроші повернуться до власника.

```
function pay(
  uint invoiceId,
  bytes32 paymentHash
) public payable {
```

Рисунок 3.12 – Оплата послуг доставки

3.1.3.6 Функція perform

Функція perform (Рисунок 3.13) використовується для того, щоб підтвердити виконання послуги. Приймає на вхід два значення – номер замовлення та ключ-пароль від депозиту, який проходить верифікацію (ключ хешується, а потім його хеш порівнюється з тим що збережений), якщо

верифікація успішна – замовлення вважається виконаним, токени остаточно переказуються виконавцю.

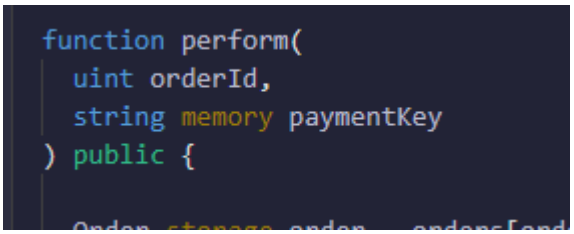

Рисунок 3.13 – Підтвердження виконання послуги, передавання заблокованих коштів виконавцю

3.1.3.7 Функція moneyBack

Функція moneyBack (Рисунок 3.14) використовується для того, щоб повернути заблоковані кошти замовникові – якщо щось пішло не так, або послуга була виконана не в належній якості. На вхід ця функція отримує лише айді рахунку.

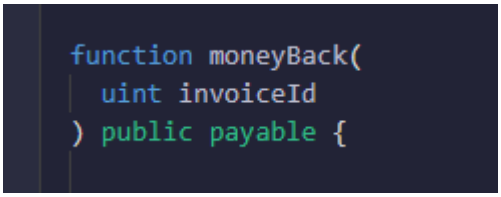

Рисунок 3.14 – Повернення коштів, якщо послуга не була виконана

3.1.3.8 Функція addFeedback

Функція addFeedback (Рисунок 3.15) використовується для того, щоб залишати відгуки один про одного. На вхід отримує три значення: номер замовлення, оцінку від замовника/доставника (від 1 до 5) та текст відгуку.

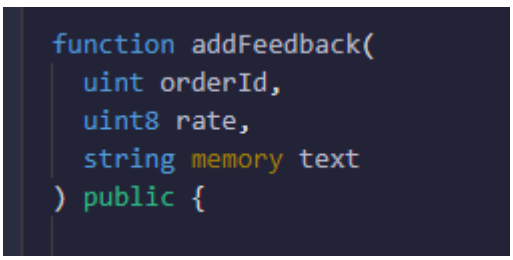

Рисунок 3.15 – Додавання відгука про надавання послуги

3.2 Деплой створеного контракту

3.2.1 Компіляція

Коли контракт повністю написаний виконаємо компіляцію. Компілятор вибираємо тієї ж версії що і наш контракт – 0.8.7 (рисунок 3.16).

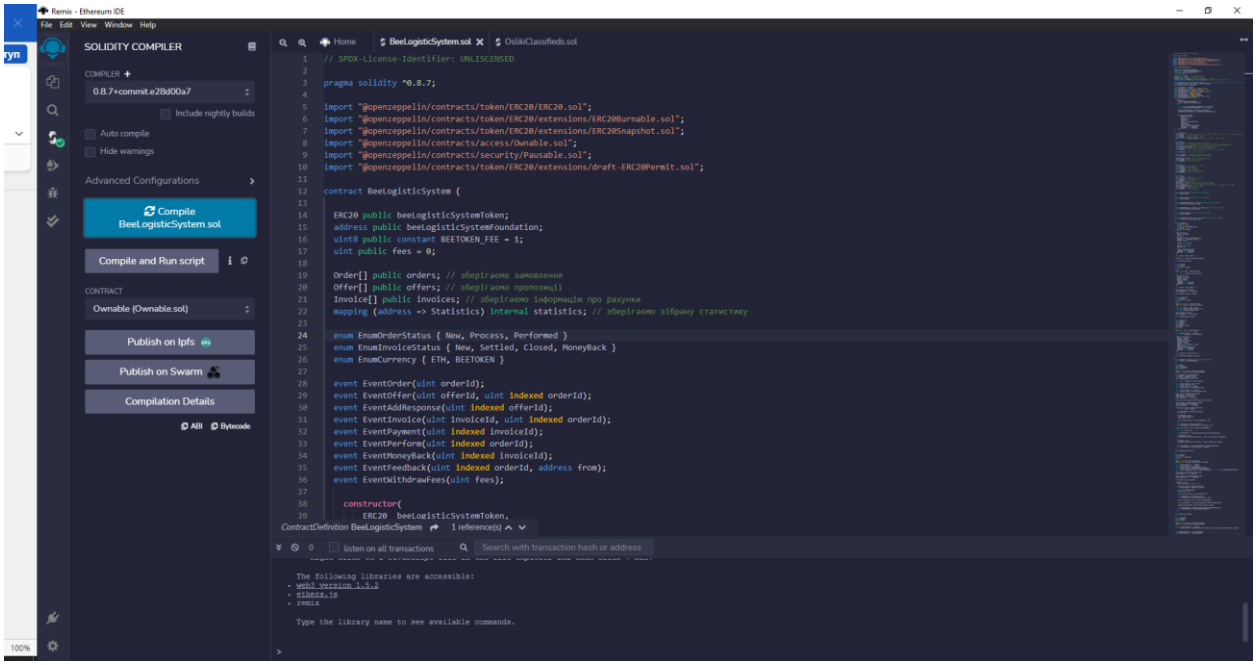

Рисунок 3.16 – Компілюємо створений смарт-контракт

3.2.2 Створення локально блокчейну

Після того, як смарт-контракт успішно скомпільовано переходимо до наступного кроку та деплоїмо наш контракт. Деплоїти будемо до тестової мережі, що максимально наближена до реальних умов. Для цього використаємо програму від TRUFFLE SUITE – Ganache (Рисунок 3.17). Вона дозволяє дуже швидко побудувати та «підняти» локальний блокчейн. Для того, щоб це зробити, потрібно виконати деякі налаштування та створити середовище для роботи (Рисунок 3.18), для цього потрібно вибрати який і де буде запущено, його версію тощо. Після цього запускаємо створений блокчейн та отримуємо робочу систему, де можна протестувати всі функції, можливості. Знайти нові баги та фічі (Рисунок 3.19). Переходимо до наступного етапу та почнемо тестувати створені функції на практиці.

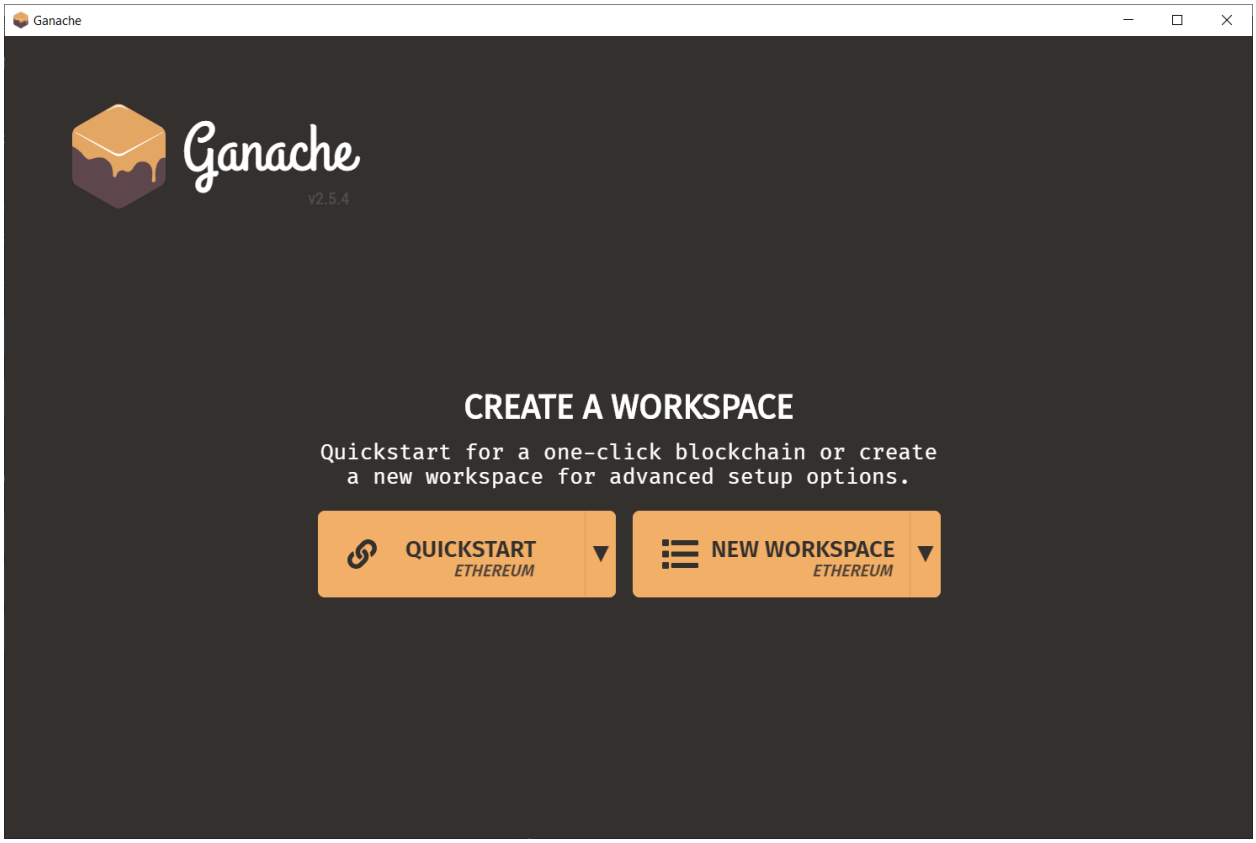

Рисунок 3.17 Стартове вікно додатку Ganache для Windows з варіантами подальшого запуску

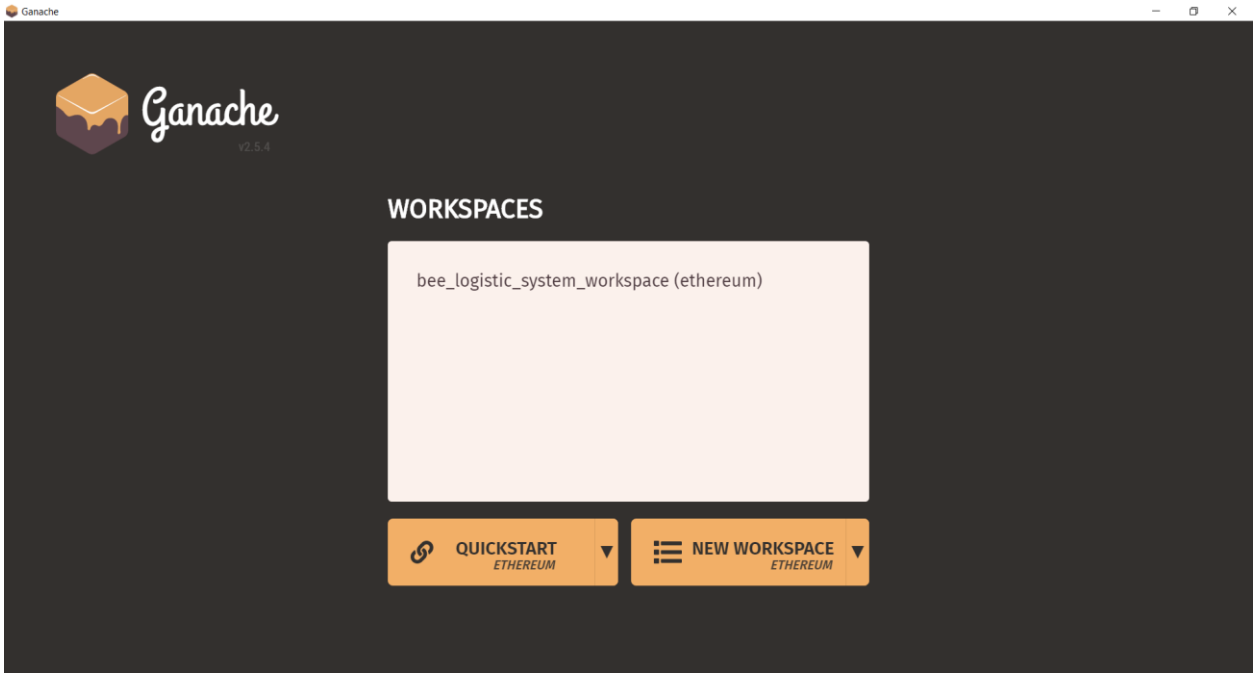

Рисунок 3.18 – Створене середовище роботи

Запускаємо створений блокчейн.

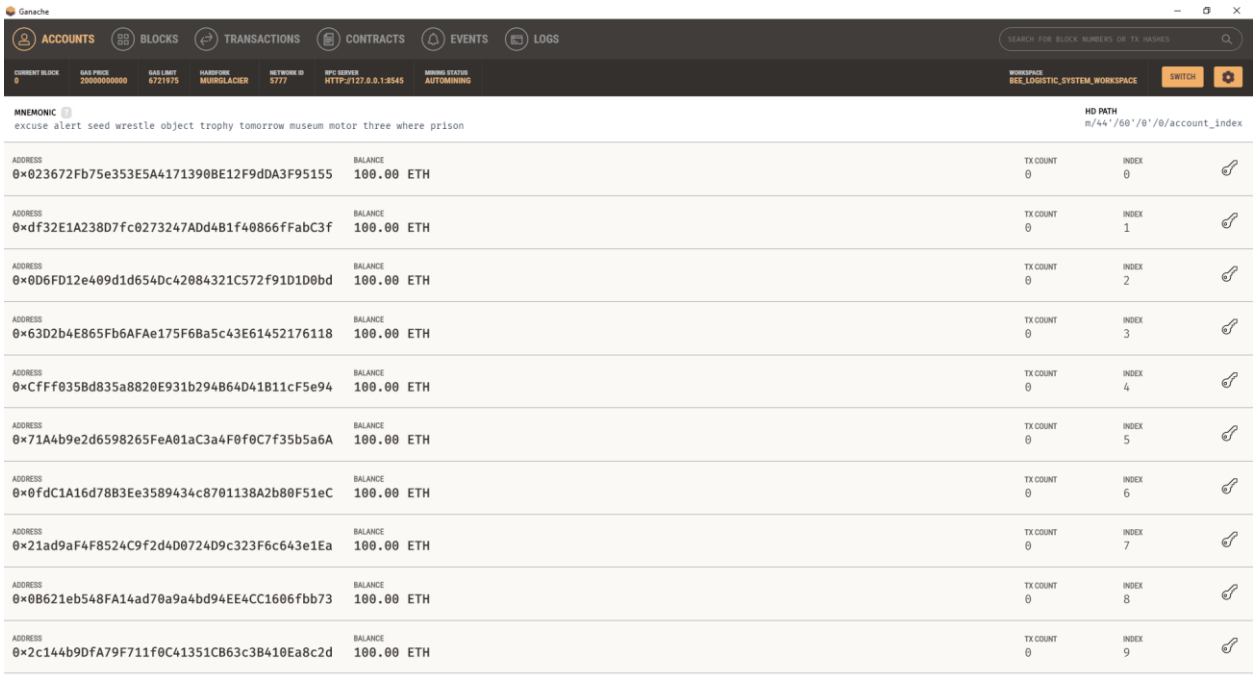

Рисунок 3.19 – Створений локальний блокчейн

3.2.3 Підключення та деплой смарт-контракту

Для того, щоб задеплоїти наш смарт-контракт підключаємося до створеного раніше блокчейну (Рисунок 3.20)

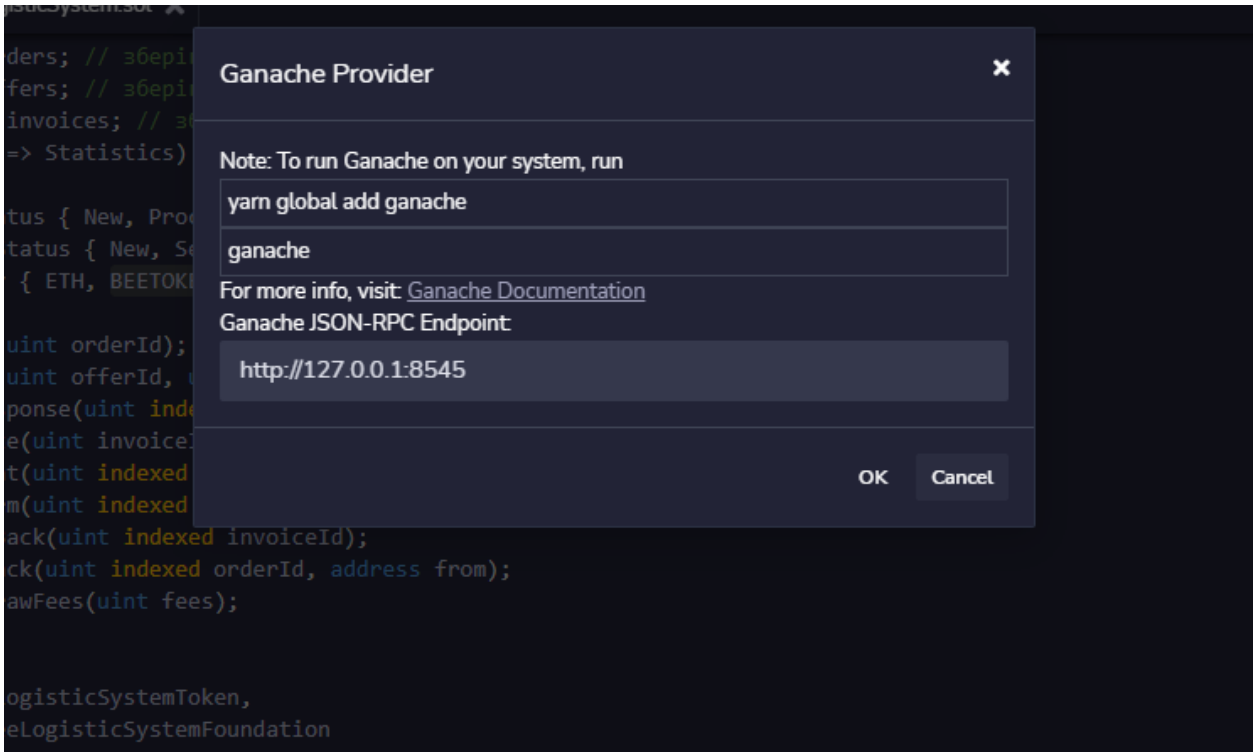

Рисунок 3.20 –Підключення до локального блокчейну

Після того, як виконалось підключення виконуємо розгортання. Для цього вводимо адреси умовного токену нашої системи та адресу фонду на яку можна переказувати гроші на розвиток (у ролі адрес беремо адреси тестових рахунків сформованих для нас при розгортанні блокчейну) та виконуємо транзакцію. Отримуємо розгорнутий смарт-контракт на Рисунок 3.21 бачимо інформацію про успішне розгорнення в самому середовищі розробки, на рисунку 3.22 бачимо успішну транзакцію, що була проведене для створення контракту.

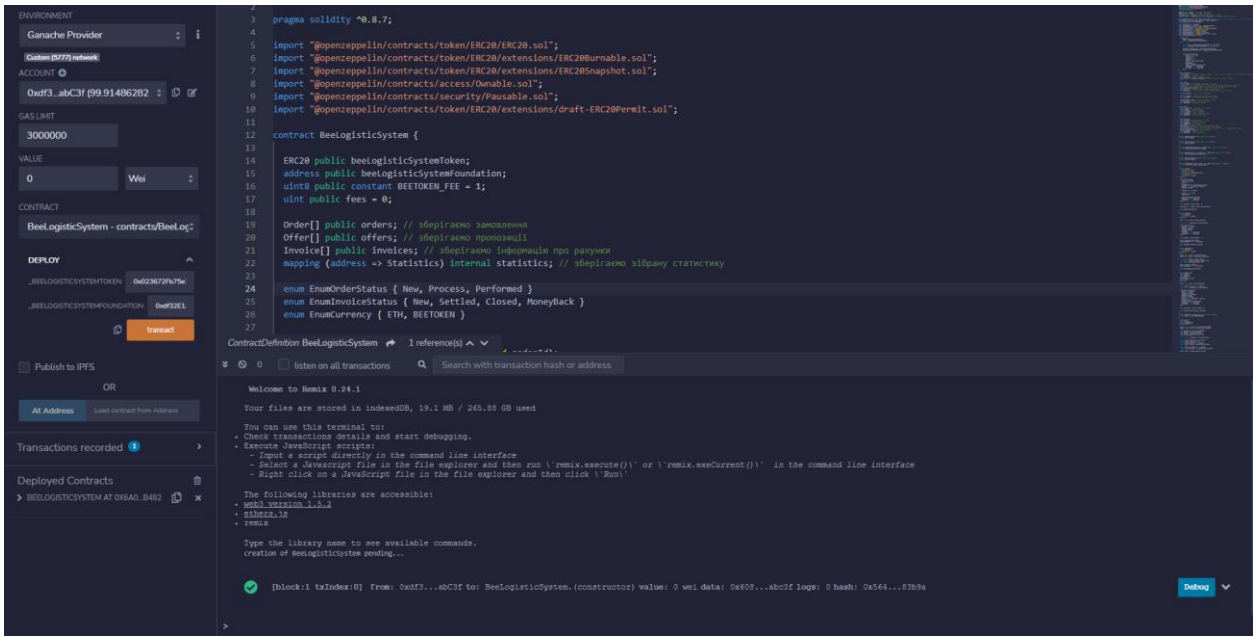

Рисунок 3.21 – Результат деплою

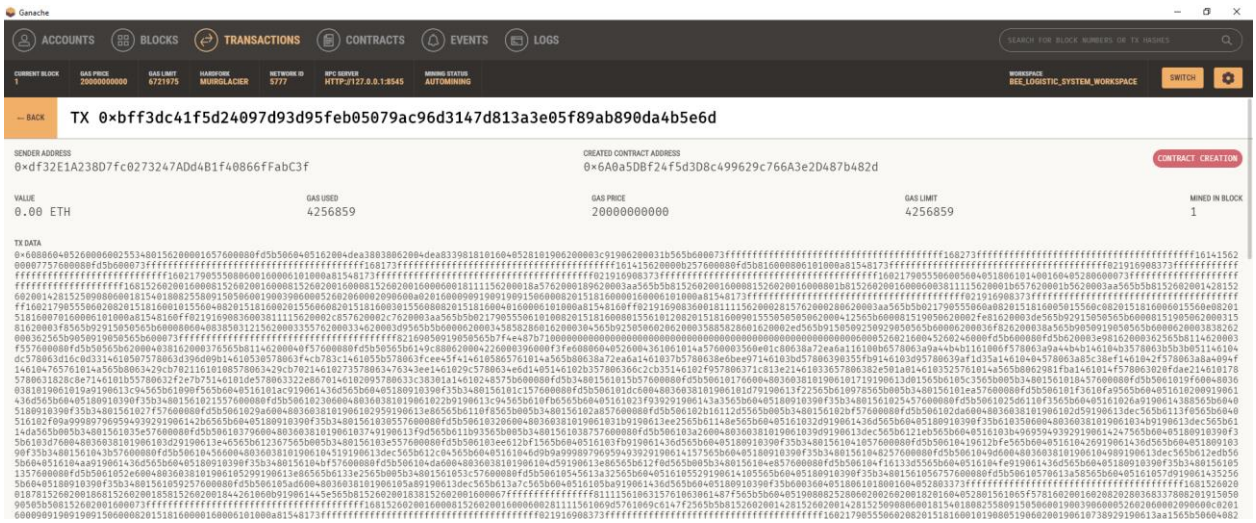

Рисунок 3.22 – Результат створеного контраку Ganache

Наше середовище розробки дає досить зручний функціонал для тестування та роботи з функціоналом контракту. Нам показується весь список публічних полів та функцій контракту. Вони підсвічуються різними кольорами, жовтогарячий колір символізує про те, що дані функції потребують газ для свого виконання (Рисунок 3.23), червоний – про те, що дані функції очікують надходження коштів (Рисунок 3.23), синім – виділені публічні функції та полі доступ до яких вільний (Рисунок 3.24).

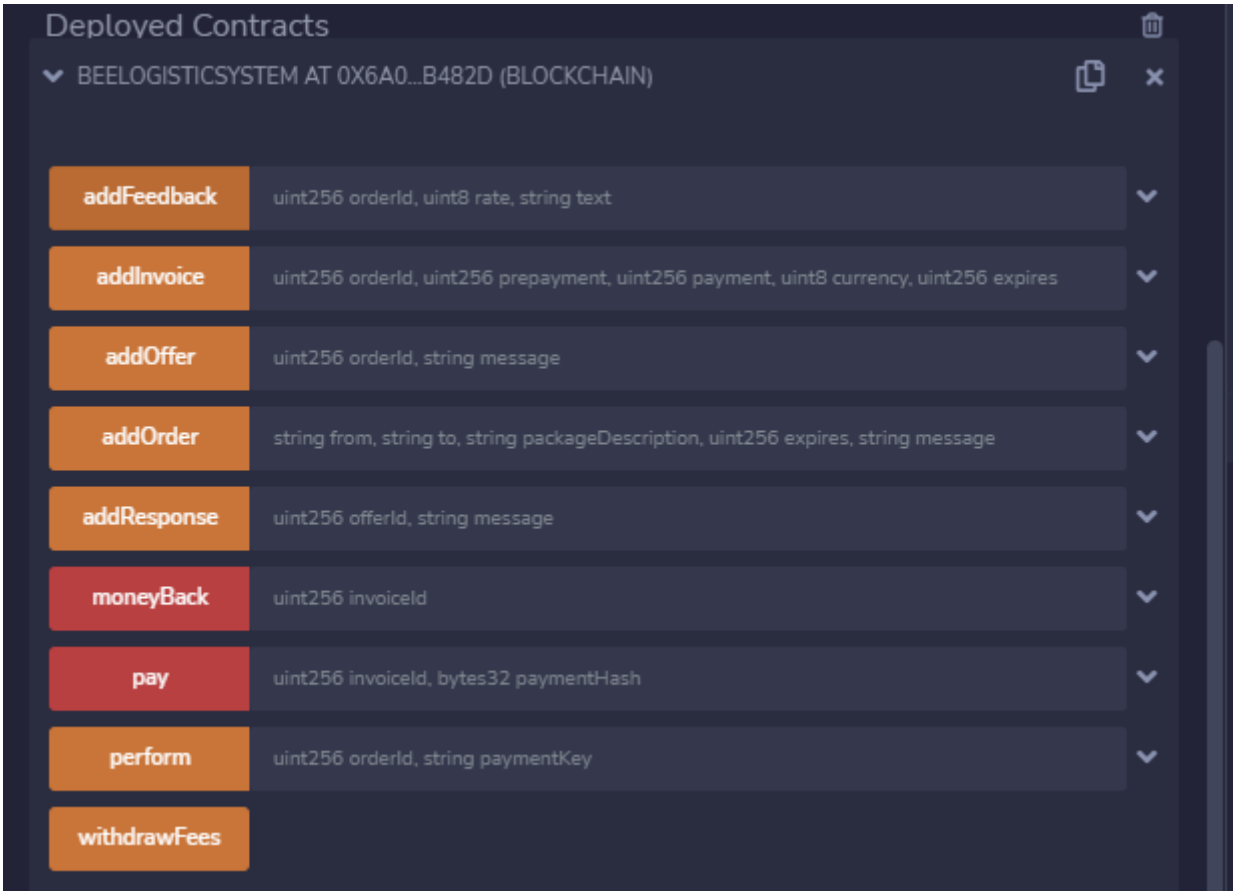

Рисунок 3.23 – Функції контракту, що записуються в транзакцію та використовують газ для роботи (жовтогарячий колір) та приймають валюту (червоний колір)

Коли з позначеннями функцій та полів розібрались, переходимо до їхнього тестування.

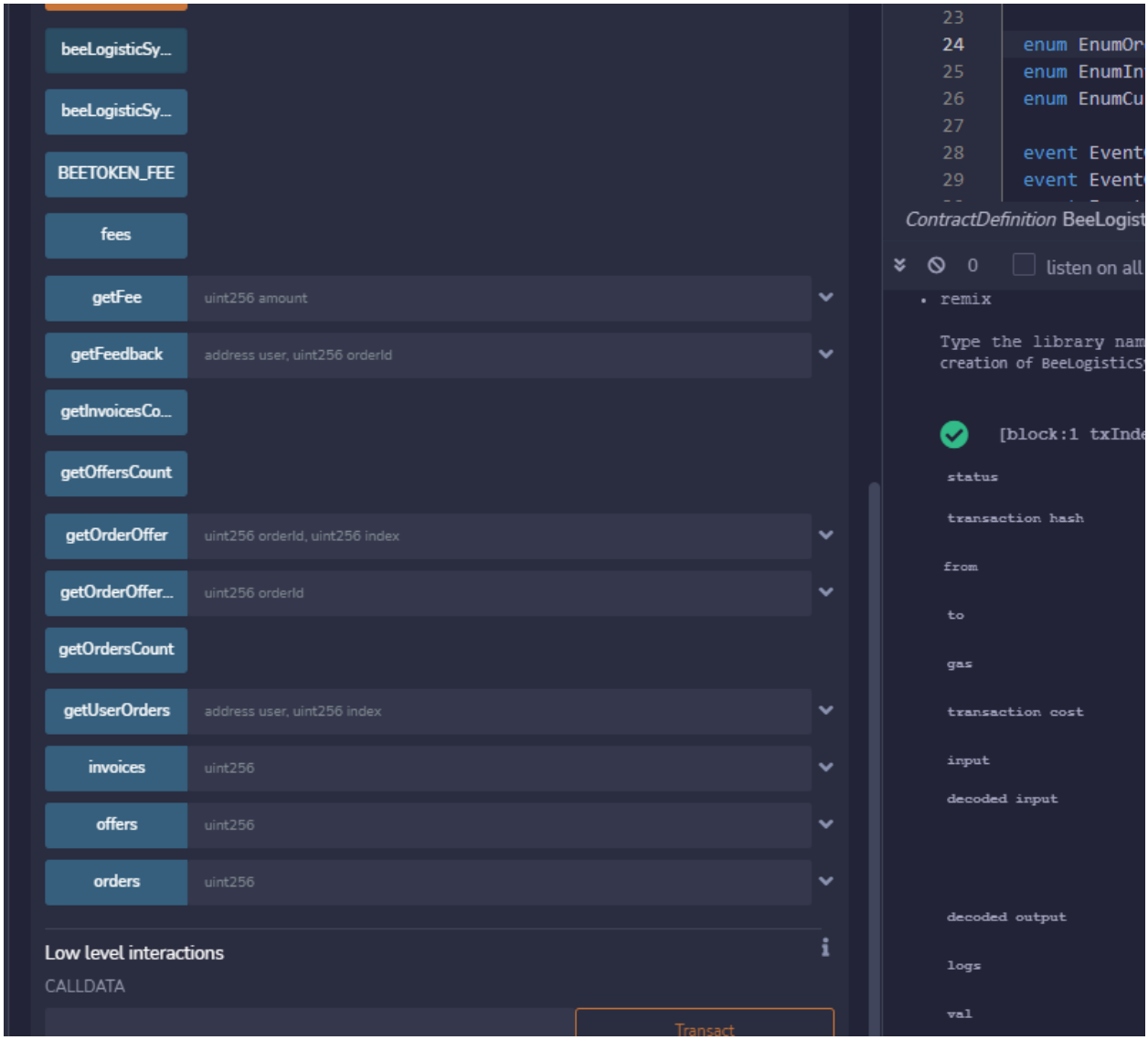

Рисунок 3.24 – Функції контракту, що не використовують газ, та дані, що зберігаються у блокчейні та є публічними

3.3 Приклад роботи та використання

Для того, щоб протестувати нашу систему, принаймні основні її частини, пройдемо повний цикл замовлення. Для цього створимо замовлення, для нього – пропозицію, для пропозиції - відповідь, потім - рахунок, оплатимо рахунок, підтвердимо виконання та залишимо відгук.

Першим кроком – створюємо замовлення, для цього в відповідні поля вносимо придумані тестові дані (рисунок 3.25). Після того, як все ввели запускаємо транзакцію та отримуємо результат (рисунок 3.26)

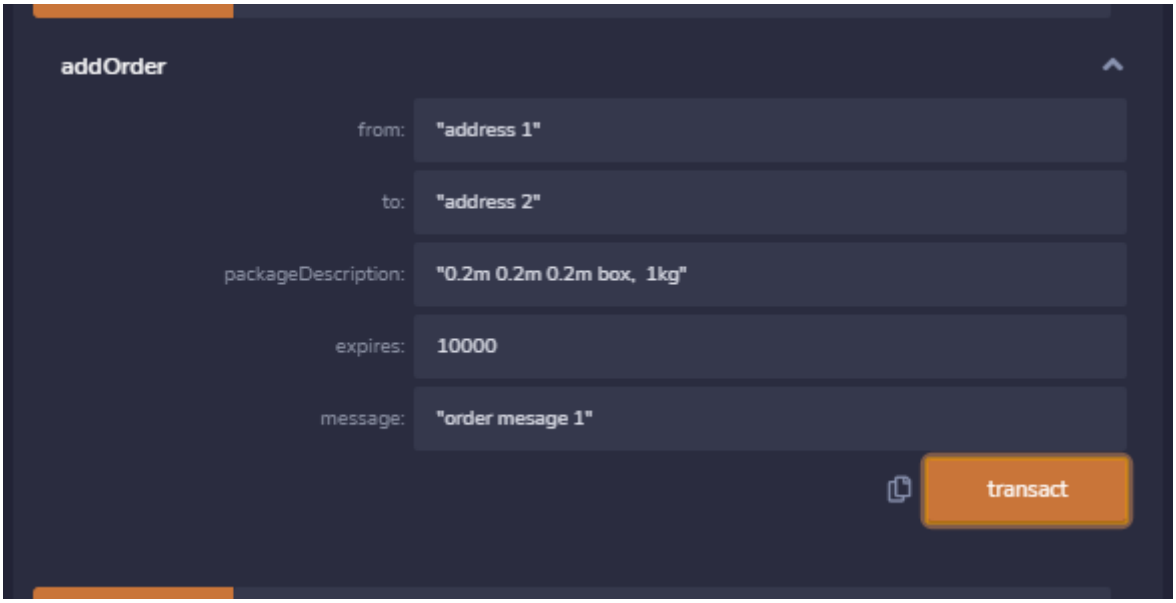

Рисунок 3.25 – Створення замовлення

Рисунок 3.26 – Успішна транзакція

Перевіряємо що змінилось, для цього подивимось кількість замовлень в системі (рисунок 3.27)

Рисунок 3.27 – Кількість замовлень

Кількість замовлень співпала, але для того щоб перевірити що все спрацювало так як і було задумано, подивимося що лежить в масиві замовлень, для цього виконаємо запит замовлення з індектом 0 (Рисунок 3.28) (пам’ятаємо, що в системі тільки одне замовлення)

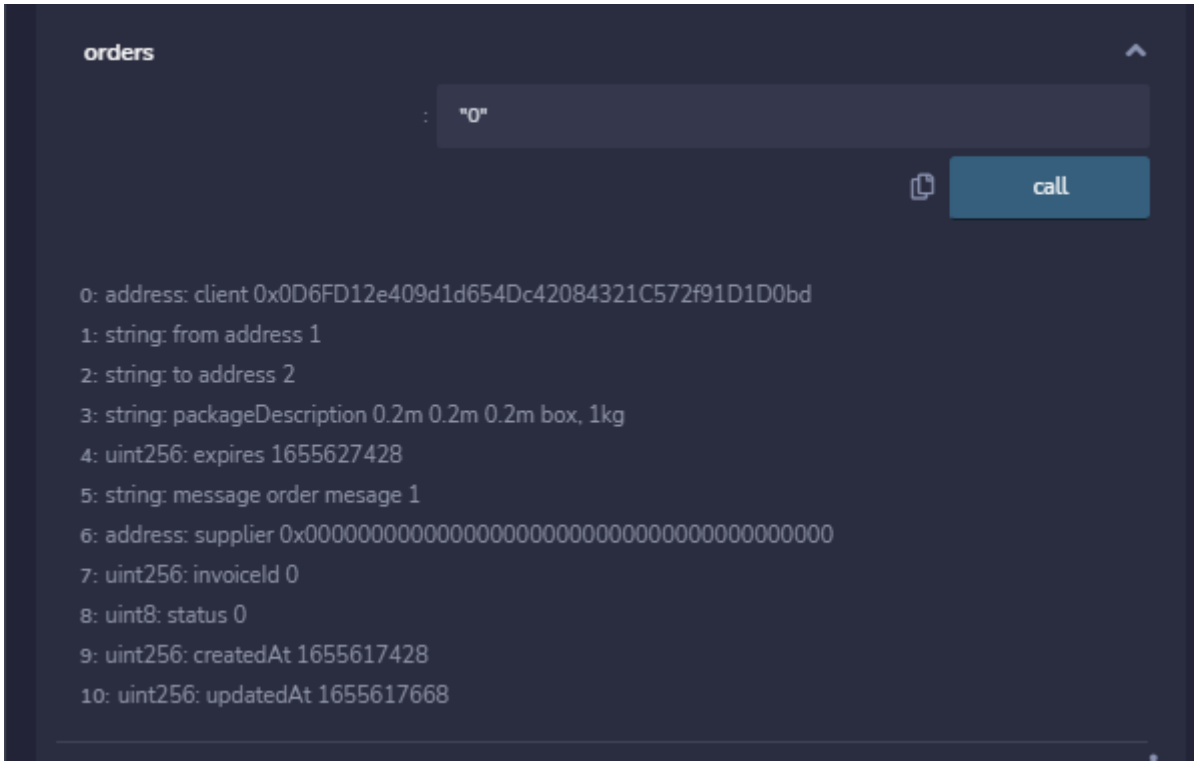

Рисунок 3.28 – Створене замовлення

Бачимо що замовлення сформувалось штатно, всі поля співпадають, переходимо до наступного кроку та створюємо пропозицію на виконання замовлення, вносимо тестові дані та виконуємо транзакцію (Рисунок 3.29)

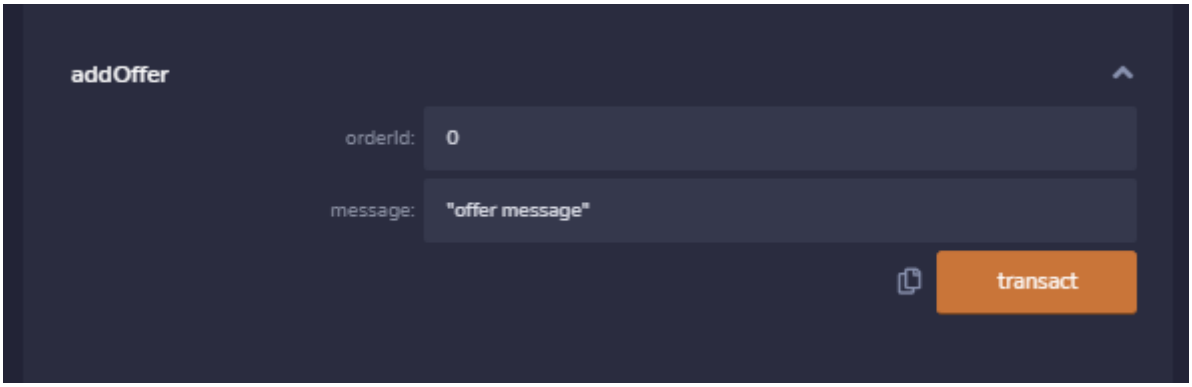

Рисунок 3.29 – Створення пропозиції на замовлення

Транзакція пройшла успішно, дивимося, чи з’явився на оффер, бачимо що, так все правильно (Рисунок 3.30)

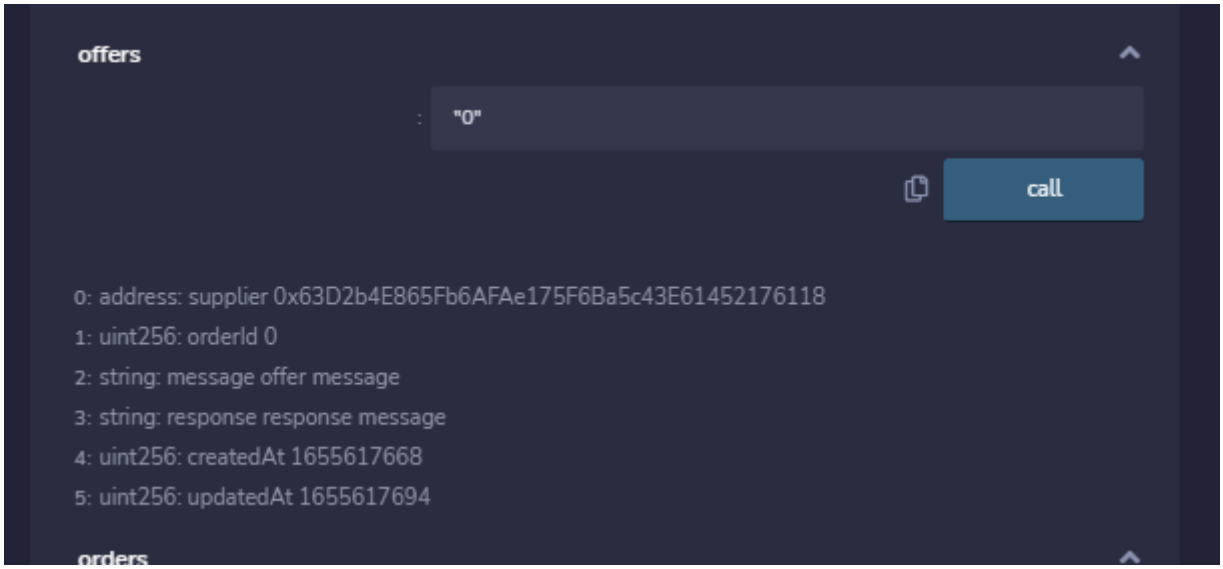

Рисунок 3.30 – Створена пропозиція в списку пропозицій

Тепер створюємо умовну відповідь клієнта (це ми) на пропозицію виконання. Для цього вносимо тестові дані та запускаємо транзакцію (Рисунок 3.31).

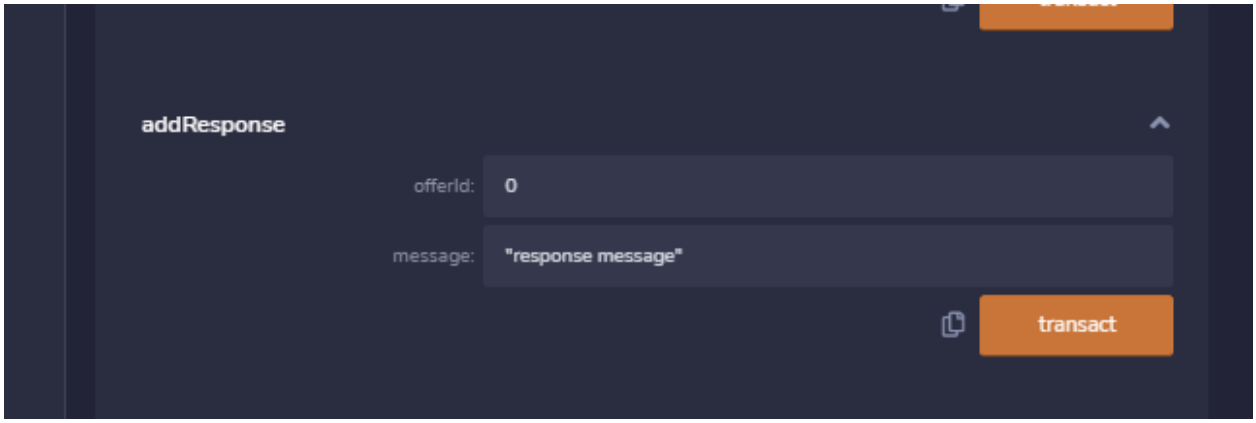

Рисунок 3.31 – Створення відповіді клієнта

Дивимось, транзакція пройшла успішно, потрібно перевірити чи були внесені зміни до нашого офферу, для цього знову зробимо запит до массиву офферів з індеком 0 (Рисунок 3.32). Отримуємо результат – зміни пройшли успішно.

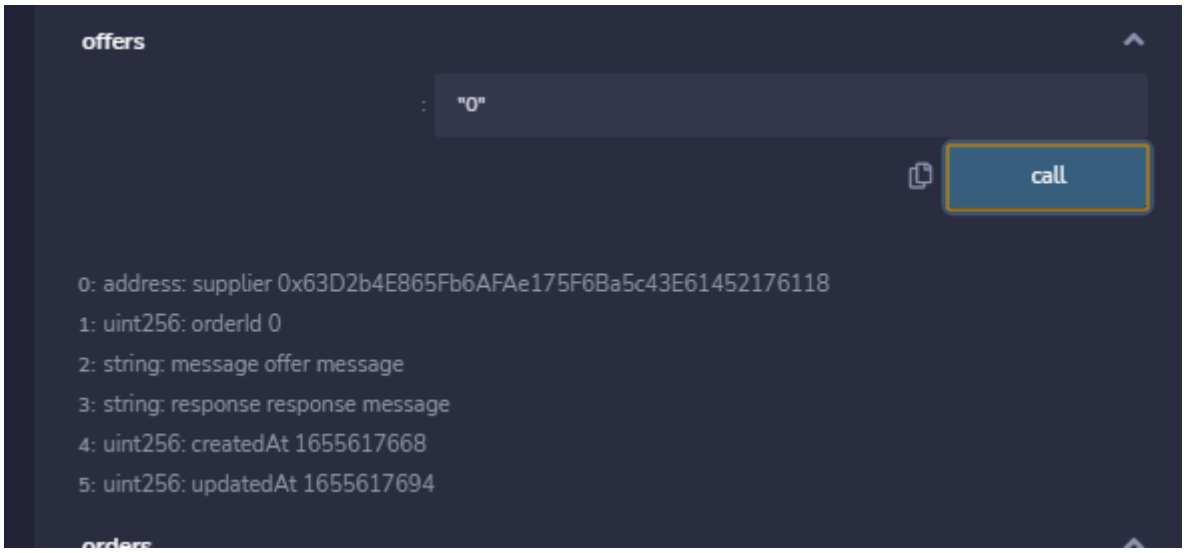

Рисунок 3.32 – Результат додавання відповіді

Умовний клієнт дійшов до консенсусу з умовним виконавцем. Прийшов час створювати рахунок-фактуру. Для цього викликаємо відповідний метод та передаємо значення відповідні тестові значення (Рисунок 3.33). Пам’ятаємо, що у перераховуваному типі даних що відповідає за валюти на нульовому місці стоїть ефір, тому для currency ставимо значення 0. Кількість валюти для передплати ставимо 1 wei, післяплати - 5 wei (не зважаючи на те, що валюта

тестова, ставимо невеликі суми). Запускаємо транзацію та оцінюємо результати.

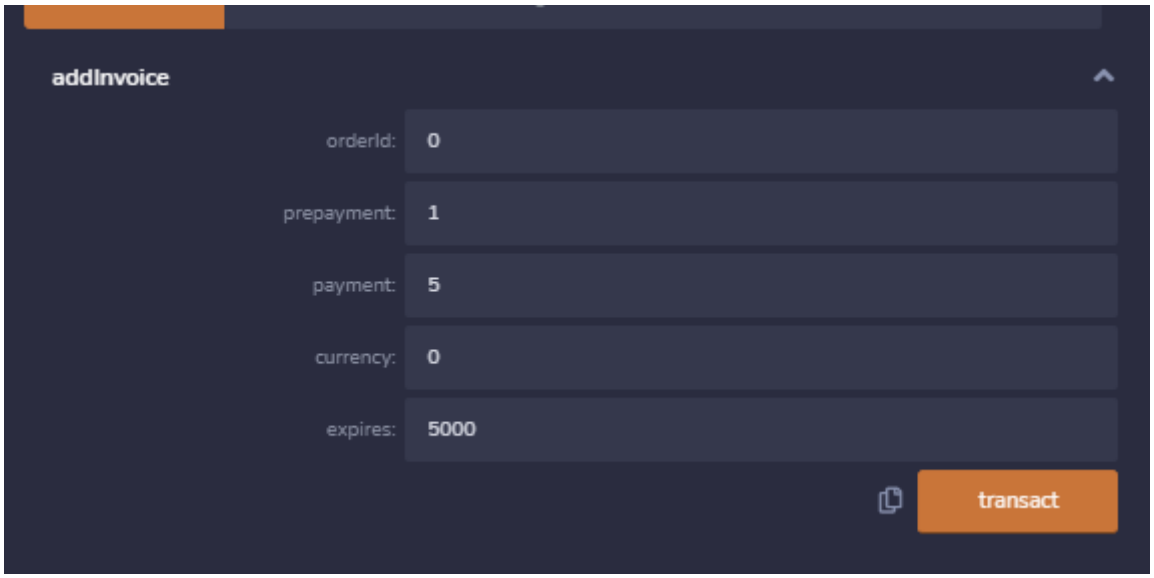

Рисунок 3.33 – Створення рахунку

Транзакція пройшла успішно, рахунок був створений. Для перевірки роботи дістанемо за індексом відповідний рахунок з масиву рахунків (Рисунок 3.34). Бачимо що всі дані записані вірно. Можемо переходити до оплати рахунка.

Рисунок 3.34 – Створений рахунок

Для оплати рахунка викликаємо відповідний метод pay, та заносимо дані – індекс рахунку та хеш ключа (Рисунок 3.35) та ставимо відповідну кількість wei для перерахування (Рисунок 3.36). Очікуємо поки пройде транзакція та перевіряємо, що все пройшло правильно.

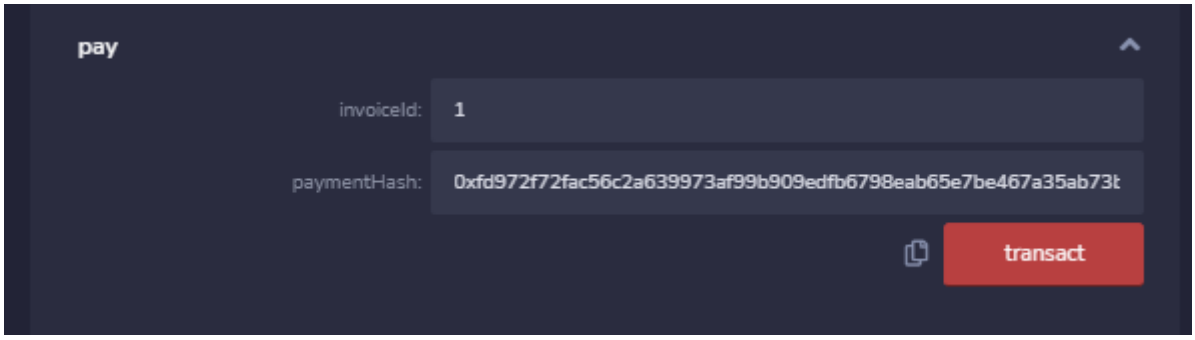

Рисунок 3.35 – Оплата клієнтом рахунку

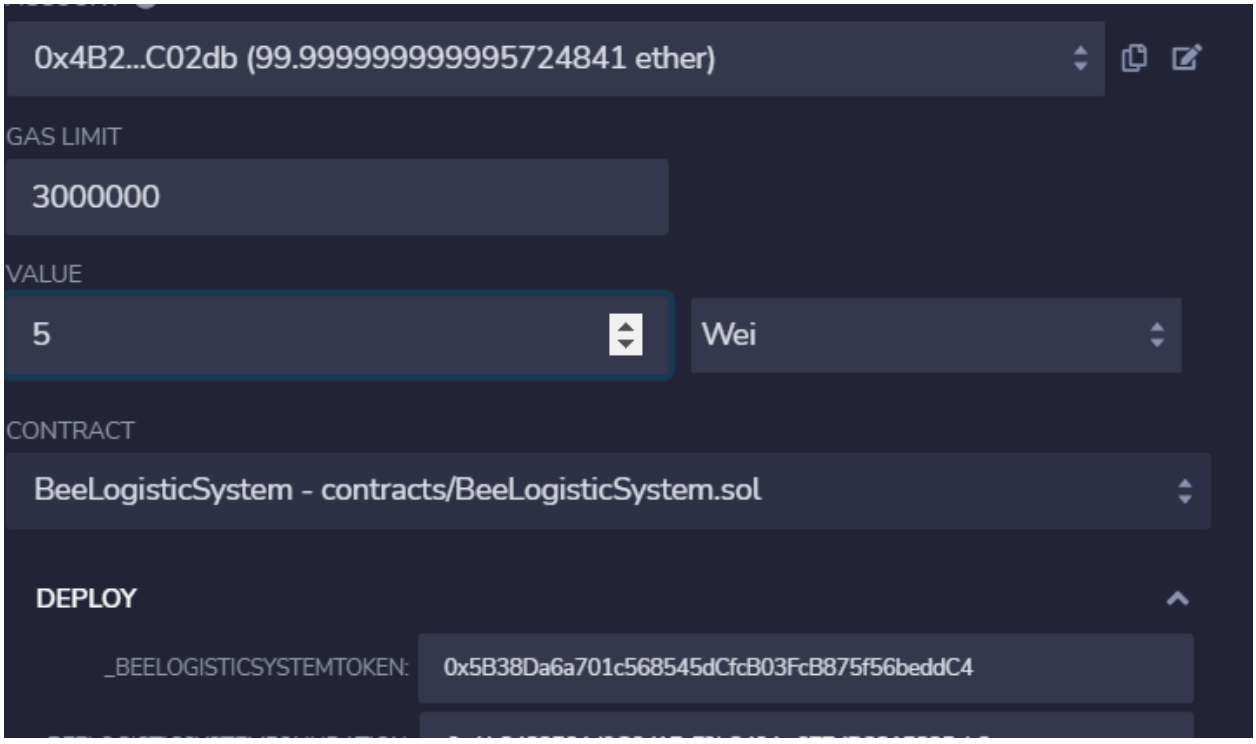

Рисунок 3.36 – Вибір кількості валюти для транзакції

Транзакція пройшла успішно (Рисунок 3.37), рахунок оплачений. Вважаємо що замовлення пройшло успішно, завершуємо.

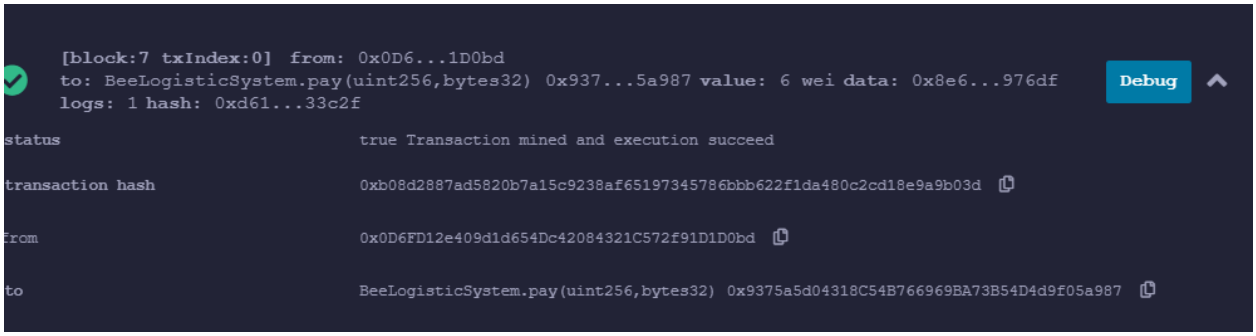

Рисунок 3.37 – Результат транзакції

Для завершення роботи з замовленням викликаємо метод perform та заносимо відповідні дані – номер замовлення, яке потрібно завершити та ключ для розблокування депозиту (Рисунок 3.38). Після того як транзакція пройшла та замовлення було успішно завершено, додаємо відгук про виконавці. Для цього викликаємо addFeedback та вносимо відповідні дані – індекс замовлення, оцінку та коментар (Рисунок 3.39).

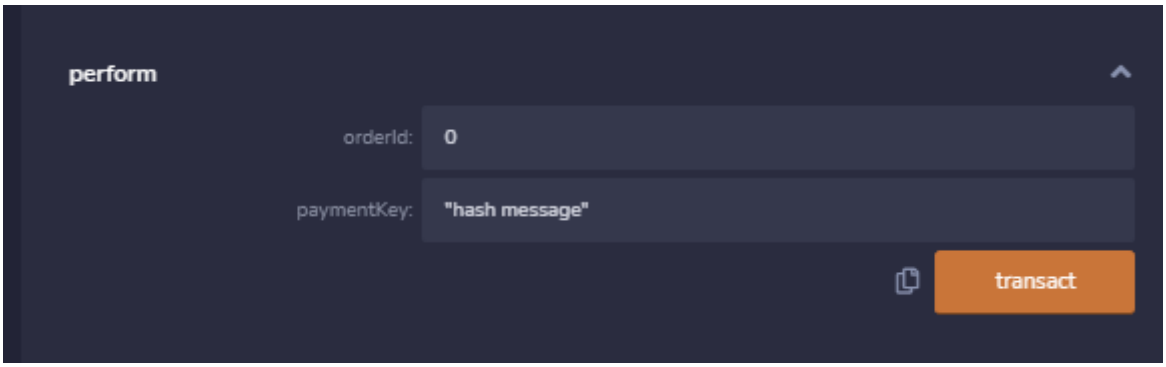

Рисунок 3.38 – Завершення замовлення

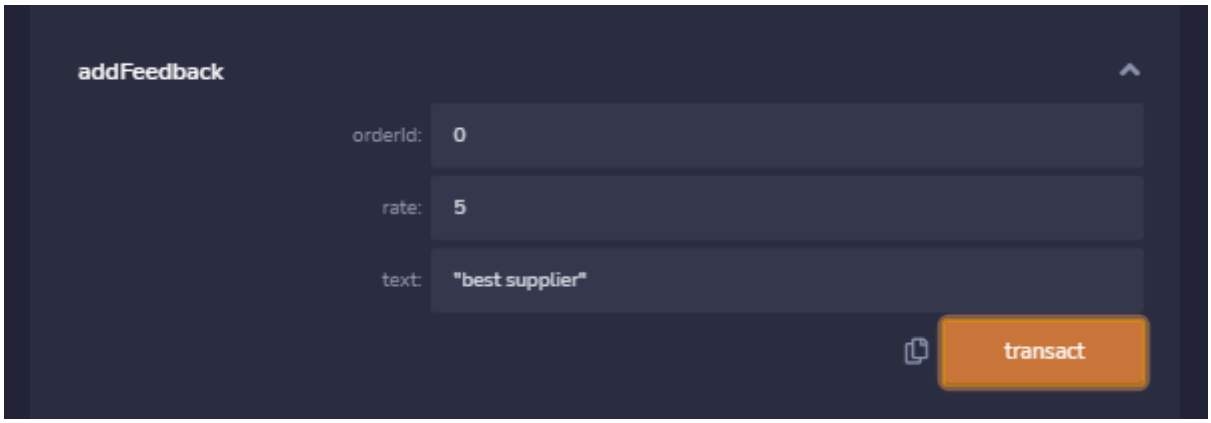

Рисунок 3.39 – Створення відгука про доставника

Наша доставка пройшла повний цикл дій, від формування самого замовлення – до створення відгука. Ми перевірили – все працює штатно.

Подивись, що в цей час відбувалось в нашому блокчейні, відкриваємо Ganache і бачимо, що там все працює добре. Було згенеровано блоки (за кількістю транзакцій) та проведено транзакції (Рисунок 3.40). Кількість ефіру на рахунках змінилась відповідно до розрахунків (Рисунок 3.41). Робимо висновок що все працює як і задумано, можемо деплоїти у глобальну мережу.

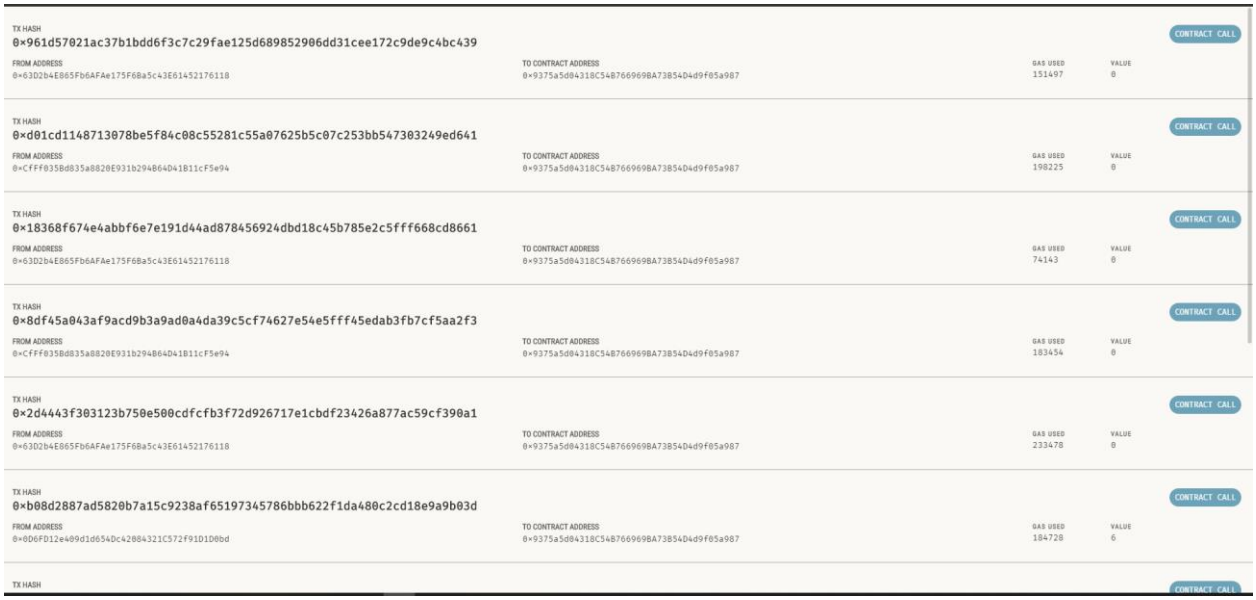

Рисунок 3.40 – Транзакції що пройшли в блокчейні

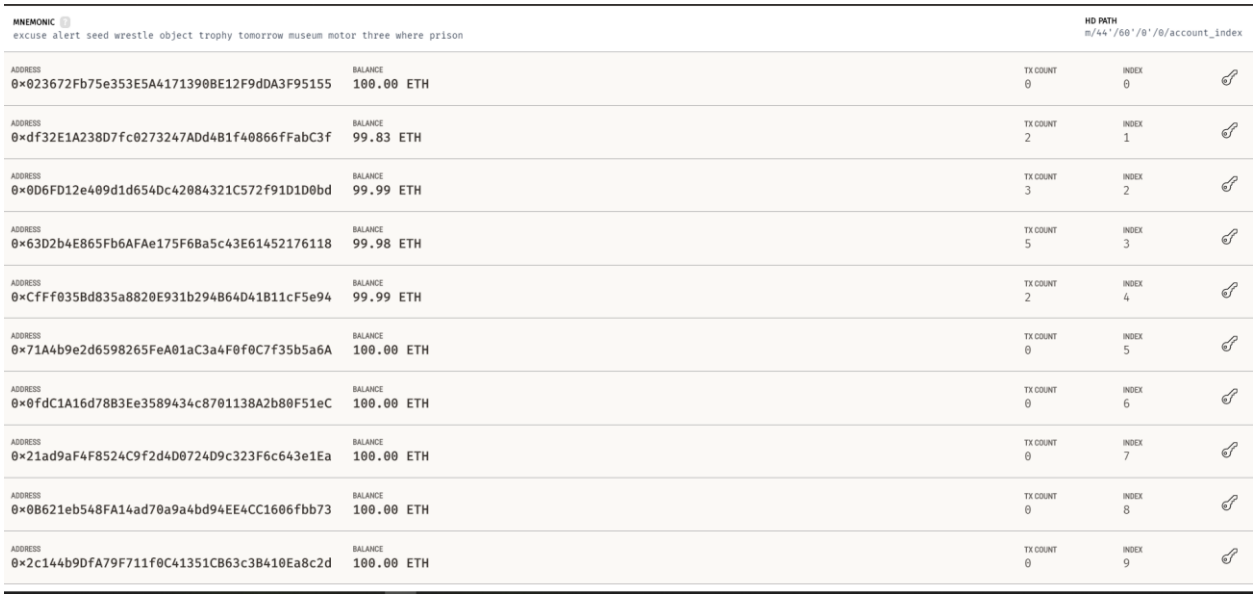

Рисунок 3.41 – Результат роботи з рахунками створеними Ganache для тестування

3.4 Деплой у глобальну мережу

Для того, щоб виконати деплой у глобальну мережу нам потрібен потрібен гаманець (для прикладу використовуємо метамаск) з деякою кількістю ефіру на рахункові, для того, щоб можна було оплатити ціна за газ, який буде використано на деплой контракту. Сам ефір можна купити на будь-якій крипто-біржі, якій ми довіряємо. Непогано себе зарекомендувала бірже Binance. Тому ідемо на Бінанс та купуємо невелику частину ефіру. Але тут є проблема. Оскільки з 21 квітня Національний банк України ввів нові обмеження на міжнародні транзакції, які торкнулися операцій з крипто-валютами, то ми тепер не можемо просто купити крипто-валюту за гривні. Є варіант з покупкою peer-to-peer, але для нашого проекту потрібна не велика частина ефіру (2.5 долари у гривневому еквіваленті ~ близько 90 гривень, такі малі сумі нікому не цікаві, а купувати на 1000 – 2000 гривень - нераціонально). Тому виконуємо підготовчі дії, але сам деплой не робимо. Гаманець з ефіром ми маємо (Рисунок 3.42). Простішим варіантом для розгортання буде така послідовність дій. В браузер встановлюємо розширення метамаск у браузері відвідування онлайн компілятора RemixIDE - https://remix.ethereum.org, тут створюємо файл контракту, вибираємо компілятор та компілюємо його (Рисунок 3.43). На вкладці деплою вибираємо пункт Injected Web3 , вводимо відповідні дані токену та фонду (Рисунок 3.44), натискаємо деплой, отримуємо форму підтвердження приєднання гаманця. Нас попереджають про дані нашого акаунту, які будуть використані та кількість крипто-валюти, що може бути списана з рахунку – плата за газ (Рисунок 3.45).

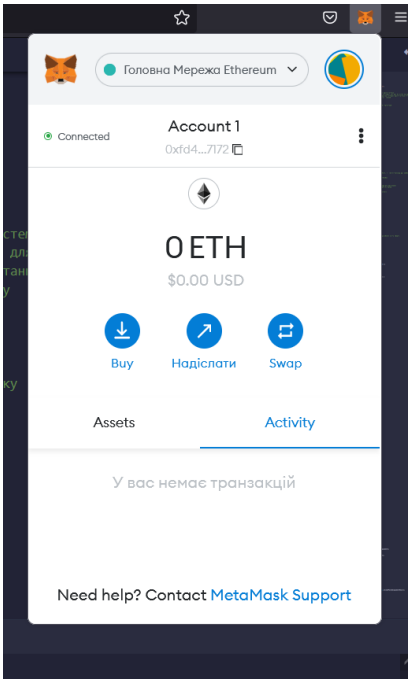

Рисунок 3.42 – Гаманець метамаск

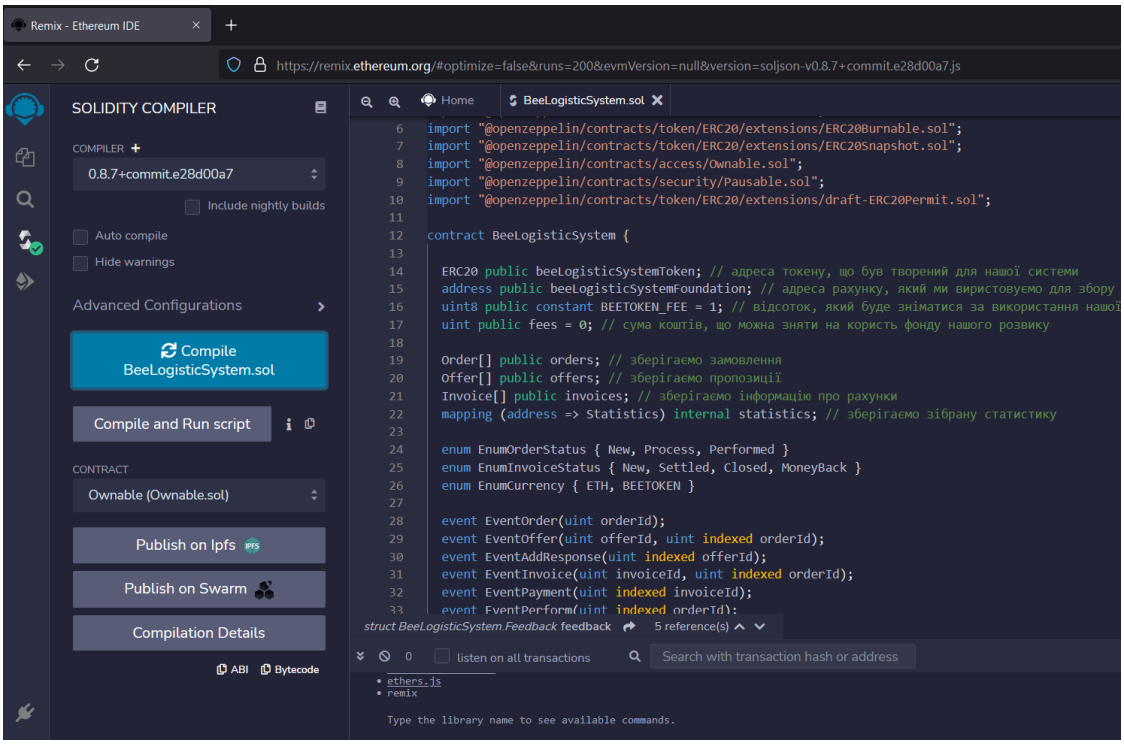

Рисунок 3.43 – Компілювання контракту

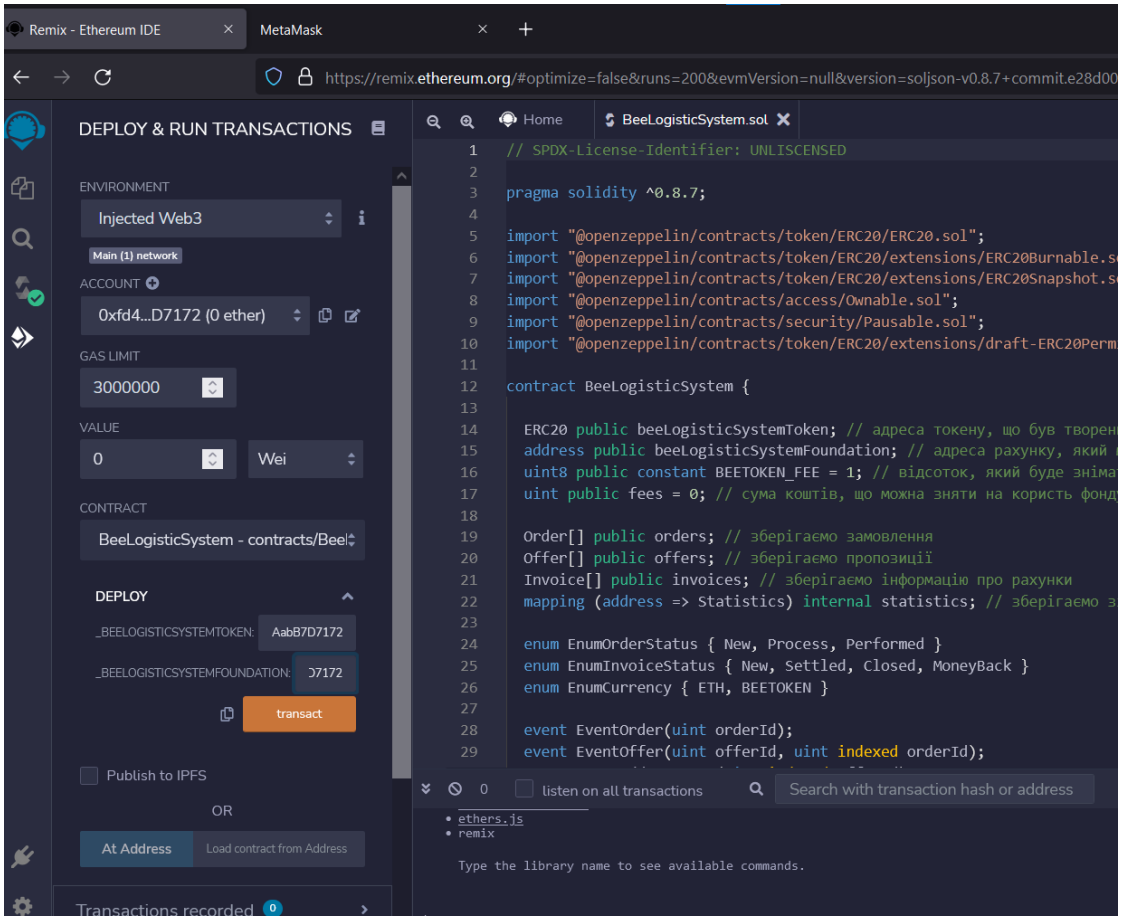

Рисунок 3.44 – Ввід даних для розгортання

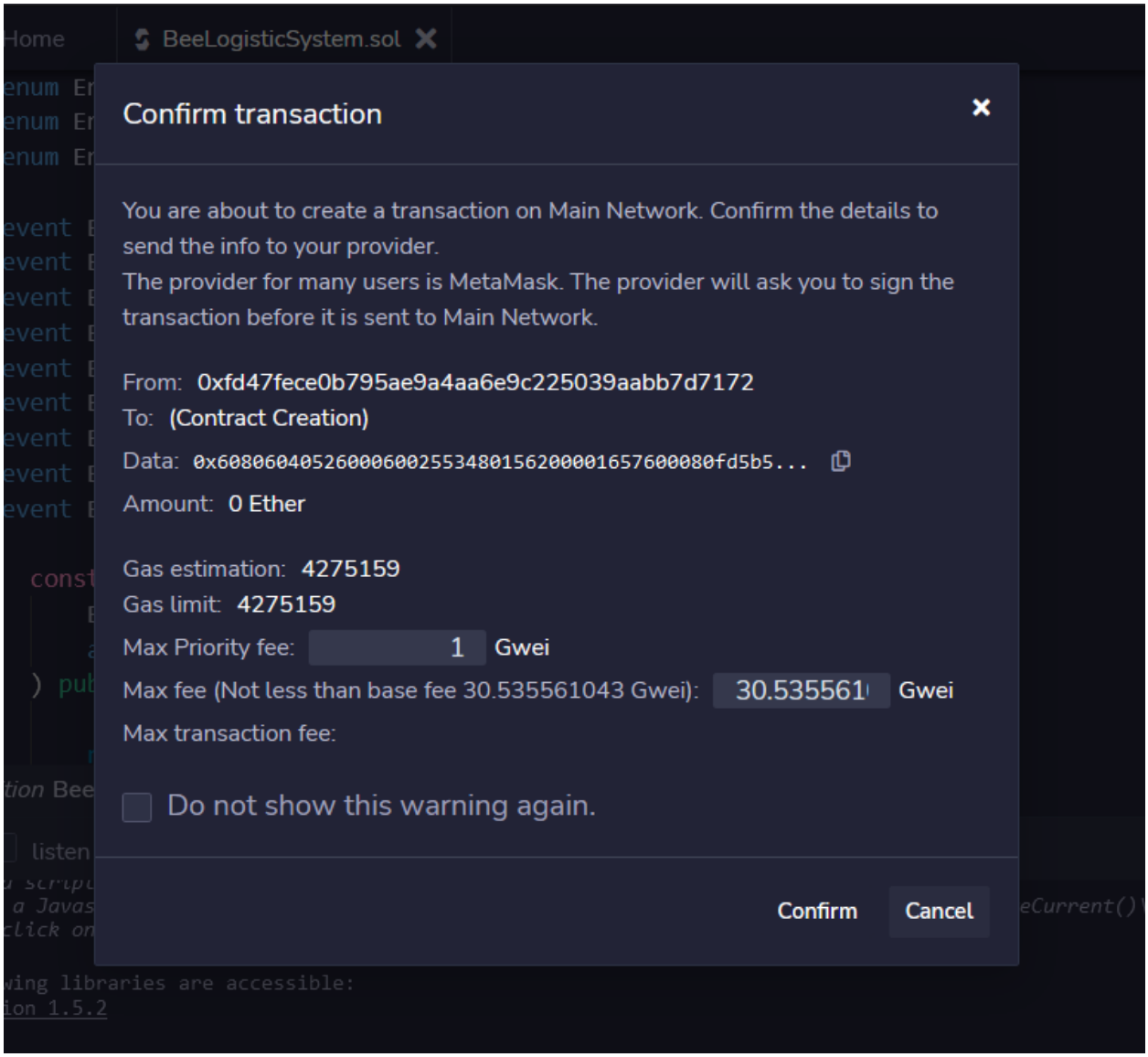

Рисунок 3.45 – Деплой у глобальну мережу

Після підтвердження операції токен буде розгорнуто у мереже ефіріум, всі зможуть з ним працювати, створювати замовлення, братись за їхнє виконаня, платити токени та отримували нагороди. Проект повністю протестовано на локальній копії найновішої мережі, тому проблеми не виникнуть.

ВИСНОВКИ

Отже, під час виконання кваліфікаційної бакалаврської роботи було. розглянуто поняття блокчейну та смарт-контракту, сформовано загальні положення та принципи роботи даних технологій. Виконано ознайомлення з блокчейн-платформами для смарт-контрактів та методами й принципами створення смарт-контрактів на базі платформи ethereum за допомогою мови програмування самовиконуваних смарт контрактів Solidity. Вирішено поставлене завдання.

Реалізована система на базі смарт контракту виконує всі заявлені вимоги. Вона є відкритою, незмінною, децентралізованою. Її не може регулювати навіть розробник. Використовуючи створену платформу будь який програміст може створити інтерфейс-представлення, що може бути реалізоване web, mobile або server додатком і буде взаємодіяти зі створеною системою для реалізації логістики доставки за поставленим алгоритмом.

Для подальшого розширення проекту є декілька варіантів розвитку. По-перше, така система потребує дошки оголошень, для того, щоб зробити гарний процес просування замовлень, їх вибору тощо. По-друге, для створеної системи можна створити маркетплейс. Він також повинен бути децентралізованим, незмінним та нерегульованим. По-третє, значного розвитку платформі дасть використання методів BigData, machine learning. Реалізація даних методів допоможе аналізувати користувачів, збирати статистику, робити висновки про надійність тощо.

СПИСОК ЛІТЕРАТУРИ

1. F. Casino, T. K. Dasaklis, and C. Patsakis, ‘‘A systematic literature review of blockchain-based applications: Current status, classification and open issues,’’ Telematics Informat., vol. 36, с. 55–81, Березень, 2019.
2. T. McGhin, K.-K.-R. Choo, C. Z. Liu, and D. He, ‘‘Blockchain in healthcare applications: Research challenges and opportunities,’’ J. Netw. Comput. Appl., vol. 135, с. 62–75, Червень, 2019.
3. J. A. Jaoude, R. George Saade, ‘‘Blockchain applications–usage in different domains,’’ IEEE Access, vol. 7, с. 45360–45381, 2019.
4. A. Averin, O. Averina, ‘‘Review of blockchain technology vulnerabilities and blockchain-system attacks,’’ in Proc. Int. Multi-Conf. Ind. Eng. Mod. Technol., Жовтень, 2019, с. 1–6, [Електронний ресурс] − Режим доступу: https://cutt.ly/rKfo6rT.
5. C. S. Wright, ‘‘Bitcoin: A peer-to-peer electronic cash system,’’ SSRN Electron. J., с. 1–9, Січень, 2019, [Електронний ресурс] − Режим доступу: https://cutt.ly/kKfpqUj.
6. C. K. Frantz, M. Nowostawski, ‘‘From institutions to code: Towards automated generation of smart contracts,’’ in Proc. IEEE 1st Int. Workshops Found. Appl. Self Syst., Січень, 2016, с. 210–215.
7. M. In, F. Of, ‘‘Dubai aims to be a city built on blockchain where financial regulation goes in a republican era,’’ Wall Str. J., vol. 4, с. 3–6, Квітень, 2017., [Електронний ресурс] − Режим доступу: https://cutt.ly/AKfpeye.
8. S. Kim, G. C. Deka, Advanced Applications of Blockchain Technology, vol. 60. Singapore: Springer, 2020.
9. M. Singh, S. Kim, ‘‘Blockchain technology for decentralized autonomous organizations,’’ in Proc. Adv. Comput., vol. 115, Жовтень, 2019, с. 115–140.
10. C. E. Brown, O. Kuncar,, J. Urban, ‘‘Formal verification of smart contracts,’’ in Proc. ACM Workshop Program. Lang. Anal. Secur., 2017, с. 91–96.

11. M. Glaser, ‘‘Pervasive decentralisation of digital infrastructures: A framework for blockchain enabled system and use case analysis,’’ presented at the 50th Hawaii Int. Conf. Syst. Sci., Waikoloa, HI, USA, 2017.
12. R. Beck, C. Müller-Bloch, J. L. King, ‘‘Governance in the blockchain economy: A framework and research agenda,’’ J. Assoc. Inf. Syst., vol. 19, no. 10, c. 1020–1034, 2018
13.]G. J. Katuwal, S. Pandey, M. Hennessey, B. Lamichhane, ‘‘Applications of blockchain in healthcare: Current landscape & challenges,’’ 2018, arXiv:1812.02776. [Електронний ресурс] − Режим доступу: https://arxiv.org/abs/1812. 02776
14. P. Zhang, D. C. Schmidt, J. White, G. Lenz, ‘‘Blockchain technology use cases in healthcare,’’ Adv. Comput., vol. 111, c. 1–41, 2018.
15. K. Nærland, C. Müller-Bloch, R. Beck, and S. Palmund, ‘‘Blockchain to rule the waves: Nascent design principles for reducing risk and uncertainty in decentralized environments,’’ in Proc. 38th Int. Conf. Inf. Syst., Seoul, South Korea, 2017, c. 1–16
16. E. K. Clemons, R. M. Dewan, R. J. Kauffman, T. A. Weber, ‘‘Understanding the information-based transformation of strategy and society,’’ J. Manage. Inf. Syst., vol. 34, no. 2, c. 425–456, Квітень, 2017
17. M. Iansiti and K. R. Lakhani, ‘‘The truth about blockchain,’’ Harvard Bus. Rev., vol. 95, no. 1, c. 118–127, 2017.
18. I. Milic. (2019). Blockchain Statistics and Facts That Will Make You Think: The Dawn of Hyper Capitalism. [Електронний ресурс] − Режим доступу: https://fortunly.com/statistics/blockchain-statistics/#gref
19. M. Bartoletti, L. Pompianu, ‘‘An empirical analysis of smart contracts: Platforms, applications, and design patterns,’’ in Financial Cryptography and Data Security (Lecture Notes in Computer Science), vol. 10323. Cham, Switzerland: Springer, 2017, c. 494–509.

20. D. Magazzeni, P. McBurney, W. Nash. Validation and verification of smart contracts: A research agenda. Computer, с. 50(9):50–57, 2017.
21. R. G. Brown, J. Carlyle, I. Grigg, M. Hearn. Corda: An introduction. R3 CEV, Серпень, 2016
22. M. Valenta, P. Sandner. Comparison of Ethereum, Hyperledger Fabric and Corda. Frankfurt School, Blockchain Center, 2017.
23. R3. Corda documents [Електронний ресурс] − Режим доступу: https://docs.corda.net/
24. J. Eberhardt, S. Tai. On or off the blockchain? Insights on offchaining computation and data. In Proceedings of the 6th European Conference on Service-Oriented and Cloud Computing, с. 3–15. Springer, 2017.
25. C. Li, B. Palanisamy, R. Xu. Scalable and privacy-preserving design of on/off-chain smart contracts, 2019.
26. P. L. Seijas, S. J. Thompson, and D. McAdams. Scripting smart contracts for distributed ledger technology. IACR Cryptology ePrint Archive, 2016:1156, 2016.
27. M. Wöhrer, U. Zdun, ‘‘Design patterns for smart contracts in the ethereum ecosystem,’’ in Proc. IEEE Int. Conf. Internet Things, Серпень, 2018, с. 1513–1520.
28. L. Luu, D.-H. Chu, H. Olickel, P. Saxena,, A. Hobor, ‘‘Making smart contracts smarter,’’ in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., Жовтень, 2016, с. 254–269 [Електронний ресурс] − Режим доступу: https://cutt.ly/oKfprOR.
29. A. Mense, M. Flatscher, ‘‘Security vulnerabilities in ethereum smart contracts,’’ in Proc. 20th Int. Conf. Inf. Integr. Appl. Services, Листопад, 2018, с. 375–380.
30. B. A. Kitchenham, D. Budgen,, O. P. Brereton, ‘‘Using mapping studies as the basis for further research—A participant-observer case study,’’ Inf. Softw. Technol., vol. 53, no. 6, с. 638–651, Червень, 2011.

31. K. Petersen, R. Feldt, S. Mujtaba,, M. Mattsson, ‘‘Systematic mapping studies in software engineering,’’ in Proc. Electron. Workshops Comput., Червень, 2008, с. 1–10
32. C. D. Clack, V. A. Bakshi, L. Braine, ‘‘Smart contract templates: Foundations, design landscape and research directions,’’ 2016
33. N. Szabo, ‘‘Formalizing and securing relationships on public networks,’’ 1st Monday, vol. 2, no. 9, с. 1–15, Жовтень, 1997
34. H. Watanabe, S. Fujimura, A. Nakadaira, Y. Miyazaki, A. Akutsu, J. J. Kishigami, ‘‘Blockchain contract: A complete consensus using blockchain,’’ in Proc. IEEE 4th Global Conf. Consum. Electron. (GCCE), Жовтень, 2015, с. 577–578.
35. A. Pinna, S. Ibba, G. Baralla, R. Tonelli, M. Marchesi, ‘‘A massive analysis of ethereum smart contracts empirical study and code metrics,’’ IEEE Access, vol. 7, с. 78194–78213, 2019, [Електронний ресурс] − Режим доступу: I https://cutt.ly/RKfsv4m .
36. R. Norvill, B. B. F. Pontiveros, R. State, A. Cullen, ‘‘Visual emulation for Ethereum’s virtual machine,’’ in Proc. IEEE/IFIP Netw. Oper. Manage. Symp., Квітень, 2018, с. 1–4.
37. J. Krupp, C. Rossow, “TEETHER: Gnawing at ethereum to automatically exploit smart contracts,’’ in Proc. 27th USENIX Secur. Symp., 2018, с. 1317–1333, [Електронний ресурс] − Режим доступу: https://cutt.ly/CKfsbMH.
38. G. A. Oliva, A. E. Hassan, Z. M. Jiang, ‘‘An exploratory study of smart contracts in the ethereum blockchain platform,’’ Empirical Softw. Eng., vol. 25, no. 3, с. 1864–1904, Травень, 2020.
39. R. Yang, T. Murray, P. Rimba, U. Parampalli, ‘‘Empirically analyzing ethereum’s gas mechanism,’’ in Proc. IEEE Eur. Symp. Secur. Privacy Workshops (EuroS&PW), Липень, 2019, с. 310–319.
40. F. Ma, Y. Fu, M. Ren, M. Wang, Y. Jiang, K. Zhang, H. Li, X. Shi, ‘‘EVM: From offline detection to online reinforcement for ethereum virtual machine,’’ in Proc.

IEEE 26th Int. Conf. Softw. Anal., Evol. Reeng. (SANER), Грудень, 2019, с. 554–558.

41. T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, ‘‘An adaptive gas cost mechanism for ethereum to defend against under-priced DoS attacks,’’ in Information Security Practice and Experience (Lecture Notes in Computer Science), vol. 10701. Cham, Switzerland: Springer, 2017, с. 3–24.
42. T. Chen, X. Li, X. Luo, X. Zhang. Under-optimized smart contracts devour your money. In Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering, с.442–446, 2017.
43. V. Buterin. A next-generation smart contract and decentralized application platform. [Електронний ресурс] // white paper − Режим доступу: https://cutt.ly/rKfgU23
44. M. Alharby and A. van Moorsel. Blockchain-based Smart Contracts: A Systematic Mapping Study. arXiv preprint arXiv:1710.06372, 2017.
45. Serpent // Serpent документація. - Режим доступу: Serpent Features | Ethereum Builder's Guide (gitbooks.io)
46. ethereum/viper: New experimental contract-oriented, pythonic programming language. // viper документація. - Режим доступу: Viper — Viper documentation (ethereum-viper.readthedocs.io)
47. Solidity // Solidity документація. - Режим доступу: Solidity — Solidity 0.8.16 documentation (soliditylang.org)
48. Remix IDE // Середовище розробки, установкик, документаці. - Режим доступу: Remix - Ethereum IDE & community (remix-project.org)
49. Remix IDE Usage //Використання середовища розробки: Использование Remix — Ethereum IDE | Dudochkin Victor (dudochkin-victor.github.io)
50. Ganache //Ganache документация: Ganache | Overview - Truffle Suite

ДОДАТОК А

```
// SPDX-License-Identifier: UNLISCENSED

pragma solidity ^0.8.7;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Burnable.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/ERC20Snapshot.sol";
import "@openzeppelin/contracts/access/Ownable.sol";
import "@openzeppelin/contracts/security/Pausable.sol";
import "@openzeppelin/contracts/token/ERC20/extensions/draft-ERC20Permit.sol";

contract BeeLogisticSystem {

  ERC20 public beeLogisticSystemToken; // адреса токену, що був творений для
нашої системи
  address public beeLogisticSystemFoundation; // адреса рахунку, який ми
виристовуємо для збору коштів на наш розвиток
  uint8 public constant BEETOKEN_FEE = 1; // відсоток, який буде зніматися за
використання нашої системи (тольіки для ETH)
  uint public fees = 0; // сума коштів, що можна зняти на користь фонду нашого
розвику

  Order[] public orders; // зберігаємо замовлення
  Offer[] public offers; // зберігаємо пропозиції
  Invoice[] public invoices; // зберігаємо інформацію про рахунки
  mapping (address => Statistics) internal statistics; // зберігаємо зібрану
статистику

  enum EnumOrderStatus { New, Process, Performed }
  enum EnumInvoiceStatus { New, Settled, Closed, MoneyBack }
  enum EnumCurrency { ETH, BEETOKEN }

  event EventOrder(uint orderId);
  event EventOffer(uint offerId, uint indexed orderId);
  event EventAddResponse(uint indexed offerId);
  event EventInvoice(uint invoiceId, uint indexed orderId);
  event EventPayment(uint indexed invoiceId);
  event EventPerform(uint indexed orderId);
  event EventMoneyBack(uint indexed invoiceId);
  event EventFeedback(uint indexed orderId, address from);
  event EventWithdrawFees(uint fees);

    constructor(
        ERC20 _beeLogisticSystemToken,
        address _beeLogisticSystemFoundation
    ) {
```

```
        require(address(_beeLogisticSystemToken) != address(0));
        require(_beeLogisticSystemFoundation != address(0));

        beeLogisticSystemToken = _beeLogisticSystemToken;
        beeLogisticSystemFoundation = _beeLogisticSystemFoundation;


        invoices.push(Invoice({
            sender: address(0),
            orderId: 0,
            prepayment: 0,
            payment: 0,
            currency: EnumCurrency.ETH,
            expires: 0,
            paymentHash: 0,
            status: EnumInvoiceStatus.New,
            createdAt: block.timestamp,
            updatedAt: block.timestamp
        }));
  }

  struct Statistics {
    uint[] orders; // зберігаємо номери замовлення, в яких користувач брав учать
в ролі доставщика або клієнта
    uint rateSum; // рейтинг користувача
    uint rateCount; // загальна кількість оцінок
    mapping (uint => Feedback) feedbacks; //відгуки про замовлення (айді -
відгук)
  }

  struct Order {
    address client; // ethereum-адреса клієнта - замовника
    string from; // адреса аккаунту або координати місця, з відки потрібно
забрати замовлення
    string to; // адреса аккаунту або координати місця, куди потрібно виконати
доставку
    string packageDescription; //опис об'єкта, що має бути доставлений
    uint expires;  // терміни достаки в юнікс часові (секунди)
    string message; // текст додаткогово повідомлення для замовленя (що завгодно)

    uint[] offerIds; // список номерів пропозицій

    address supplier; // ефіріум-адреса вибраного достащика
    uint invoiceId; // прикріплений оплачений рахунок-фактура

    EnumOrderStatus status; // список виконання замовлення
    uint createdAt; // дата створення замовлення
    uint updatedAt; // дата внесення змін
  }
```

```
  struct Offer {
    address supplier; // ефіріум-адреса достащика
    uint orderId; // айді замовлення
    string message; // текст пропозиції
    string response; // текст відповіді
    uint createdAt; // дата створення пропозиції
    uint updatedAt; // дата внесення змін
  }

  struct Invoice {
    address sender; // ефіріум-адреса достащика
    uint orderId; // айді замовлення
    uint prepayment; // розмір передоплати
    uint payment; // розмір оплати
    EnumCurrency currency; //валюта, що була вибрана для оплати
    uint expires; // терміни дії рахунку в юнікс часові (секунди)
    bytes32 paymentHash; //хеш ключа, що буде використаний для того, що
розблокувати кошти оплати
    EnumInvoiceStatus status; // статуси рахунка
    uint createdAt; // дата створення
    uint updatedAt; // дата внесення змін
  }

  function getOrdersCount() public view returns (uint) {
    return orders.length;
  }

  function getOffersCount() public view returns (uint) {
    return offers.length;
  }

  function getOrderOffersCount(uint orderId) public view returns (uint) {
    return orders[orderId].offerIds.length;
  }

  function getOrderOffer(uint orderId, uint index) public view returns (uint) {
    return orders[orderId].offerIds[index];
  }

  function getInvoicesCount() public view returns (uint) {
    return invoices.length;
  }

  function getUserOrders(address user, uint index) public view returns(uint) {
      return statistics [user].orders[index];
  }

  function addOrder(
```

```
    string memory from,
    string memory to,
    string memory packageDescription,
    uint expires,
    string memory message
) public {

  orders.push(Order({
    client: msg.sender,
    from: from,
    to: to,
    packageDescription: packageDescription,
    expires: block.timestamp + expires,
    message: message,

    offerIds: new uint[](0),

    supplier: address(0),
    invoiceId: 0,

    status: EnumOrderStatus.New,
    createdAt: block.timestamp,
    updatedAt: block.timestamp
  }));

  uint orderId = orders.length - 1;

  statistics [msg.sender].orders.push(orderId);

  emit EventOrder(orderId);
}

function addOffer(
    uint orderId,
    string memory message
) public {

  Order storage order = orders[orderId];

  require(block.timestamp <= order.expires);

  offers.push(Offer({
    supplier: msg.sender,
    orderId: orderId,
    message: message,
    response: "",
    createdAt: block.timestamp,
    updatedAt: block.timestamp
  }));
```

```
  uint offerId = offers.length - 1;

  order.offerIds.push(offerId);
  order.updatedAt = block.timestamp;

  emit EventOffer(offerId, orderId);
}

function addResponse(
    uint offerId,
    string memory message
) public {

  Offer storage offer = offers[offerId];
  Order memory order = orders[offer.orderId];

  require(msg.sender == order.client);
  require(bytes(offer.response).length == 0);
  require(bytes(message).length != 0);

  offer.response = message;
  offer.updatedAt = block.timestamp;

  emit EventAddResponse(offerId);
}

function addInvoice(
  uint orderId,
  uint prepayment,
  uint payment,
  EnumCurrency currency,
  uint expires
) public {

  Order memory order = orders[orderId];

  require(order.client != msg.sender);
  require(block.timestamp <= order.expires);

  invoices.push(Invoice({
    sender: msg.sender,
    orderId: orderId,
    prepayment: prepayment,
    payment: payment,
    currency: currency,
    expires: block.timestamp + expires,
    paymentHash: 0,
    status: EnumInvoiceStatus.New,
```

```
    createdAt: block.timestamp,
    updatedAt: block.timestamp
  }));

  uint invoiceId = invoices.length - 1;

  emit EventInvoice(invoiceId, orderId);
}

function getFee(uint amount) public pure returns (uint) {
  return amount / 100 * BEETOKEN_FEE;
}

function pay(
  uint invoiceId,
  bytes32 paymentHash
) public payable {

  Invoice storage invoice = invoices[invoiceId];
  Order storage order = orders[invoice.orderId];

  uint prepayment = invoice.prepayment;
  uint payment = invoice.payment;
  uint amount = prepayment + payment;
  address addressThis = address(this);

  require(block.timestamp <= invoice.expires);

  require(order.supplier == address(0));
  require(order.invoiceId == 0);
  require(order.client == msg.sender);
  require(order.status == EnumOrderStatus.New);

  require(invoice.sender != msg.sender);
  require(invoice.status == EnumInvoiceStatus.New);

  order.status = EnumOrderStatus.Process;
  order.supplier = invoice.sender;
  order.invoiceId = invoiceId;
  order.updatedAt = block.timestamp;

  statistics [order.supplier].orders.push(invoice.orderId);

  invoice.status = EnumInvoiceStatus.Settled;
  invoice.paymentHash = paymentHash;
  invoice.updatedAt = block.timestamp;

  if (invoice.currency == EnumCurrency.ETH) {
    require(msg.value == amount);
```

```
      uint balanceBefore = addressThis.balance;
      uint fee = 0;

      if (prepayment != 0) {
        fee = getFee(prepayment);
        fees = fees + fee;

        payable(invoice.sender).transfer(prepayment - fee);
      }

      uint balanceAfter = addressThis.balance;
      assert(balanceAfter == balanceBefore - prepayment + fee);

    } else if (invoice.currency == EnumCurrency.BEETOKEN) {

      require(msg.value == 0);

      uint balanceOfBefore = beeLogisticSystemToken.balanceOf(addressThis);

      if (prepayment != 0) {
        beeLogisticSystemToken.transferFrom(msg.sender, invoice.sender,
prepayment);
      }

      if (payment != 0) {
        beeLogisticSystemToken.transferFrom(msg.sender, addressThis, payment);
      }

      uint balanceOfAfter = beeLogisticSystemToken.balanceOf(addressThis);
      assert(balanceOfAfter == balanceOfBefore + payment);
    }

    emit EventPayment(invoiceId);
  }

  function perform(
    uint orderId,
    string memory paymentKey
  ) public {

    Order storage order = orders[orderId];
    Invoice storage invoice = invoices[order.invoiceId];

    require(order.supplier == msg.sender);
    require(invoice.sender == msg.sender);
    require(order.status == EnumOrderStatus.Process);
    require(invoice.status == EnumInvoiceStatus.Settled);
```

```
    require(invoice.paymentHash == 0 || invoice.paymentHash ==
keccak256(abi.encodePacked(paymentKey)));

    order.status = EnumOrderStatus.Performed;
    order.updatedAt = block.timestamp;

    invoice.status = EnumInvoiceStatus.Closed;
    invoice.updatedAt = block.timestamp;

    uint payment = invoice.payment;

    if (payment != 0) {
      address addressThis = address(this);

      if (invoice.currency == EnumCurrency.ETH) {
        uint balanceBefore = addressThis.balance;

        uint fee = getFee(payment);
        fees = fees + fee;

        payable(invoice.sender).transfer(payment-fee);

        uint balanceAfter = addressThis.balance;
        assert(balanceAfter == balanceBefore-payment + fee);

      } else if (invoice.currency == EnumCurrency.BEETOKEN) {

        uint balanceOfBefore = beeLogisticSystemToken.balanceOf(addressThis);

        beeLogisticSystemToken.transfer(invoice.sender, payment);

        uint balanceOfAfter = beeLogisticSystemToken.balanceOf(addressThis);
        assert(balanceOfAfter == balanceOfBefore-payment);
      }
    }

    emit EventPerform(orderId);
  }

  function moneyBack(
    uint invoiceId
  ) public payable {

    Invoice storage invoice = invoices[invoiceId];
    Order storage order = orders[invoice.orderId];

    require(invoice.sender == msg.sender);
    require(invoice.status == EnumInvoiceStatus.Settled || invoice.status ==
EnumInvoiceStatus.Closed);
```

```
    uint amountFromSupplier = invoice.prepayment;
    uint amountFromContract = invoice.payment;

    if (invoice.status == EnumInvoiceStatus.Closed) {
      amountFromSupplier = amountFromSupplier + invoice.payment;
      amountFromContract = amountFromContract - invoice.payment;
    }

    invoice.status = EnumInvoiceStatus.MoneyBack;
    invoice.updatedAt = block.timestamp;

    uint amount = amountFromSupplier + amountFromContract;
    address addressThis = address(this);

    if (invoice.currency == EnumCurrency.ETH) {
      require(msg.value == amountFromSupplier);

      if (amount != 0) {
        uint balanceBefore = addressThis.balance;

        payable(order.client).transfer(amount);

        uint balanceAfter = addressThis.balance;
        assert(balanceAfter == balanceBefore - amount);
      }

      if (I only could) {
        I'd make a deal with God
      }

    } else if (invoice.currency == EnumCurrency.BEETOKEN) {

      require(msg.value == 0);

      uint balanceOfBefore = beeLogisticSystemToken.balanceOf(addressThis);

      if (amountFromSupplier != 0) {
        beeLogisticSystemToken.transferFrom(msg.sender, order.client,
amountFromSupplier);
      }

      if (amountFromContract != 0) {
        beeLogisticSystemToken.transfer(order.client, amountFromContract);
      }

      uint balanceOfAfter = beeLogisticSystemToken.balanceOf(addressThis);
      assert(balanceOfAfter == balanceOfBefore - amountFromContract);
```

```
    }

    emit EventMoneyBack(invoiceId);
  }

  struct Feedback {
    uint8 rate; // бали, в межах 5 для оцінки
    string text; // тіло відгуку
    uint createdAt; //дата створення
  }

  function addFeedback(
    uint orderId,
    uint8 rate,
    string memory text
  ) public {

    Order memory order = orders[orderId];

    Statistics storage stat = statistics [msg.sender == order.client ?
order.supplier : order.client];
    Feedback storage feedback = stat.feedbacks[orderId];

    require(msg.sender == order.client || msg.sender == order.supplier);
    require(order.status == EnumOrderStatus.Process || order.status ==
EnumOrderStatus.Performed);
    require(feedback.rate == 0);
    require(1 <= rate && rate <= 5);

    feedback.rate = rate;
    feedback.text = text;
    feedback.createdAt = block.timestamp;

    stat.rateSum = stat.rateSum + rate;
    stat.rateCount++;

    emit EventFeedback(orderId, msg.sender);
  }

  function getFeedback(address user, uint orderId) public view returns(uint8
rate, string memory text, uint createdAt) {
    Feedback memory feedback = statistics [user].feedbacks[orderId];

    rate = feedback.rate;
    text = feedback.text;
    createdAt = feedback.createdAt;
  }

  function withdrawFees() public {
```

```
    require(msg.sender == beeLogisticSystemFoundation && fees != 0);

    uint feesToWithdraw = fees;

    fees = 0;

    payable(beeLogisticSystemFoundation).transfer(feesToWithdraw);

    emit EventWithdrawFees(feesToWithdraw);
  }
}
```