МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

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

В. А. Яланецький

ОСНОВИ ТЕХНОЛОГІЇ БЛОКЧЕЙН

Комп'ютерний практикум

для студентів спеціальностей

126 «Інформаційні системи та технології»

та 121 «Інженерія програмного забезпечення»

всіх форм навчання

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за всіма освітніми програмами спеціальності 126
«Інформаційні системи та технології»

Київ
КПІ ім. Ігоря Сікорського
2022

Відповідальний редактор: Петро КРАВЕЦЬ, к.т.н., доц. каф. ІСТ

Рецензент: Артем ВОЛОКИТА, к.т.н., доц. каф. ОТ

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 3 від 27.01.2022 р.)
за поданням Вченої ради факультету Інформатики та обчислювальної техніки
(протокол № 4 від 28.12.2021 р.)

Електронне мережне навчальне видання

Валерій ЯЛАНЕЦЬКИЙ, ст. викладач

ОСНОВИ ТЕХНОЛОГІЇ БЛОКЧЕЙН
Комп'ютерний практикум

Основи технології блокчейн: комп'ютерний практикум [Електронний ресурс]: навч. посіб. для студентів спеціальностей 126 «Інформаційні системи та технології» та 121 «Інженерія програмного забезпечення» / КПІ ім. Ігоря Сікорського; уклад.: В.А. Яланецький. – Електронні текстові дані (1 файл: 2.1 Мбайт). – Київ: КПІ ім. Ігоря Сікорського, 2022. – 89 с.

Навчальний посібник призначений для студентів освітньої програми «Інтегровані інформаційні системи» спеціальності 126 «Інформаційні системи та технології» кафедри інформаційних систем та технологій всіх форм навчання. Тематика посібника відповідає силабусу дисципліни «Технологія блокчейн», яка є вибірковою дисципліною у навчальному плані підготовки бакалаврів зі спеціальності 126 «Інформаційні системи та технології». В навчальному посібнику викладені теоретичні відомості та комп'ютерний практикум до дисципліни «Технологія блокчейн». Комп'ютерний практикум є можливість виконувати як у фізичних лабораторіях кафедри так і дистанційно на власних робочих станціях. Комп'ютерний практикум виконуються мовою Python. В посібнику наводяться рекомендації до вирішення задач моделювання та програмування базового функціоналу прототипу блокчейн. Посібник може бути корисний студентам відповідних спеціальностей при вивченні дисциплін, пов’язаних із технологіями блокчейн та проектуванням розподілених децентралізованих систем.

ЗМІСТ

СПИСОК СКОРОЧЕНЬ ТА ТЕРМІНІВ

Скорочення:

ЛР – лабораторна робота (комп'ютерного практикуму)

ПК – персональний комп'ютер

ПЗ – програмне забезпечення (програмні засоби)

ЛКМ, ПКМ – ліва та права кнопки маніпулятора «Миша»

ООП – об'єктно-орієнтоване програмування

PoW – доказ виконаної роботи

PoS – доказ володіння долею

JSON – текстовий формат, контейнер даних

SHA-256 – криптографічний алгоритм шифрування

IDE – середовище програмування

Терміни:

деплой – завантаження та запуск на виконання контракту

міграція – перенесення контракту в робоче середовище

ВСТУП

В навчальному посібнику викладені цілі, завдання, базові теоретичні відомості, приклади виконання лабораторних робіт (ЛР) з вибіркової дисципліни «Технологія блокчейн» (ТБ), а також порядок їх захисту на кафедрі «Інформаційних систем та технологій» факультету «Інформатики та обчислювальної техніки» Національного технічного університету України «КПІ імені Ігоря Сікорського» для студентів спеціальності 126 «Інформаційні системи та технології» та 121 «Інженерія програмного забезпечення».

Навчальний посібник призначений орієнтований на односеместровий курс дисципліни для всіх форм навчання. Посібник містить 6 лабораторних робіт. Для денної форми навчання обов'язковим є виконання всіх лабораторних робіт. Для заочної форми – перших двох лабораторних робіт.

Комплекс ЛР з дисципліни ТБ є початковим етапом вивчення та практичного засвоєння сучасних інформаційних технологій, засобів та програмних інструментів щодо моделювання та програмування прототипу блокчейну.

Мета лабораторних робіт:

- вивчення середовища програмування PyCharm;
- ознайомлення із базовими принципами програмування блокчейну на прикладі публічного блокчейну Біткойн;
- закріплення теоретичних навичок з консесусу доказу виконаної роботи;
- оволодіння базовою грамотністю створення та корисування криптогаманцем на базі тестової мережі Біткойну;
- оволодіння прикладом створення та деплою смарт-контрактів.

Завдання лабораторних робіт:

- засвоїти методику розробки прототипу блокчейну;
- вивчити принцип складання прототипу блокчейну;

- практична реалізація контурів керування в програмі для ПЛК.

Протягом виконання кожної ЛР студент самостійно повинен:

- розробити програмне рішення;
- скомпілювати та завантажити розроблену програму до цільової ВП (ВК або ПЛК) й виконати її запуск;
- перевірити правильність роботи програми на ВП;
- оформити звіт та захистити ЛР.

Кожна ЛР виконується і оформлюється відповідно до завдання на ЛР, варіант якої можна знайти в таблиці або ж завдання видається викладачем студентові особисто в електронному (друкованому) вигляді.

Комплекс ЛР ТБ складається із наступних робіт:

1) Середовище розробки та скелет прототипу блокчейну;

2) Транзакції та майнінг блоків в прототипі блокчейну;

3) Взаємодія з прототипом блокчейну засобами Postman;

4) Організація консенсусу в прототипі блокчейну;

5) Транзакції в блокчейні Bitcoin;

6) Смарт-контракти на блокчейні Ethereum.

Навчальний посібний є базовим начальним документом для подальшого вивченні інформаційних технологій в сфері децентралізованих систем, розподілених та інтелектуальних інформаційних систем, блокчейн рішень, децентралізованих фінансів.

1 ЛАБОРАТОРНА РОБОТА № 1

Тема: Середовище розробки та скелет прототипу блокчейну

Мета: Ознайомитися з середовищем PyCharm або Eclipse та базовими поняттями блокчейну.

Завдання: Встановити PyCharm (або IDE для Java), створити проєкт, додати необхідні фреймворки, додати методи створення нових блоків, реалізувати алгоритм підтвердження роботи PoW.

1.1 Короткі теоретичні про необхідні фреймворки

Мова програмування **Java** через її **ООП** спрямованість дозволяє зручно та просто виконувати проєкти, що потребують наявність об'єктів. Даний проєкт не є виключенням, тому використання мови Java є виправданим.

Розробка на цій мові може вестися як за допомогою блокнота та консолі, так і з використанням сучасних інтегрованих середовищ розробки, серед яких найбільш популярними є **Eclipse**, **IntelliJ IDEA** та **NetBeans**.

Будь-яка програма на мові Java базується на класах – файлах певної структури, які являють собою шаблон, по якому далі створюються об'єкти – примірники класів з певним станом. «Точкою входу» або точкою початку виконання програми є метод **public static void main(String[] args)** у будь-якому класі. Будь-який код, що написано у цьому класі буде виконано.

Фреймворк **Spark**, який теж буде використано у даному курсі, дозволяє швидко та просто будувати так звані **REST** сервіси – програми, що отримують, обробляють та відповідають на **HTTP** запити клієнта.

Бібліотека **Gson** від **Google** буде використана для парсингу вхідних даних запиту в об'єкти та навпаки, для конвертування об'єктів у вихідні дані, у так званому форматі **JSON**.

Бібліотека **Guava** буде потрібна для зручного і швидкого хешування

даних, так як у своєму функціоналі має реалізацію для метода **SHA-256**.

Для швидкого і простого використання вищевказаних бібліотек використаємо інструмент для автоматизованого збору проєктів **Maven**. Для підключення бібліотек потрібно буде лише вказати їх у конфігурації інструменту **Maven**.

1.2 Практичний приклад розгортання середовища розробки №1

Розглянемо розробку блокчейну за допомогою мови **Python**, використовуючи **IDE PyCharm**.

PyCharm — інтегроване середовище розробки для мови програмування Python. Надає засоби для аналізу коду, графічний зневаджувач, інструмент для запуску юніт-тестів і підтримує веб-розробку на Django. PyCharm розроблена російською компанією JetBrains[1] на основі IntelliJ IDEA.

Завантажити PyCharm можна з офіційного сайту (https://www.jetbrains.com/pycharm/download/), цілком піддійте безкоштовна **Community** версія.

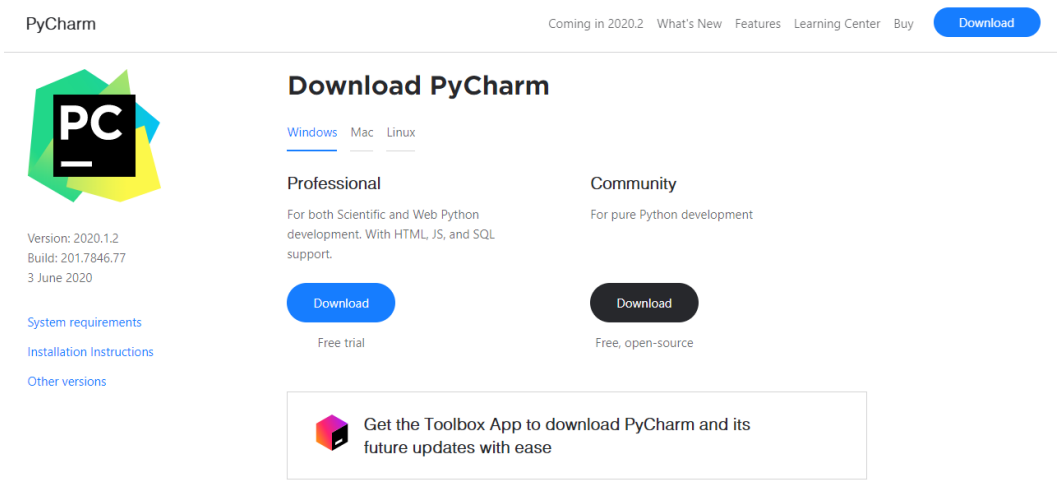


Рис. 1. Сторінка завантаження PyCharm

Після встановлення IDE потрібно створити новий проєкт. Для цього необхідно вибрати «**File → New Project**».

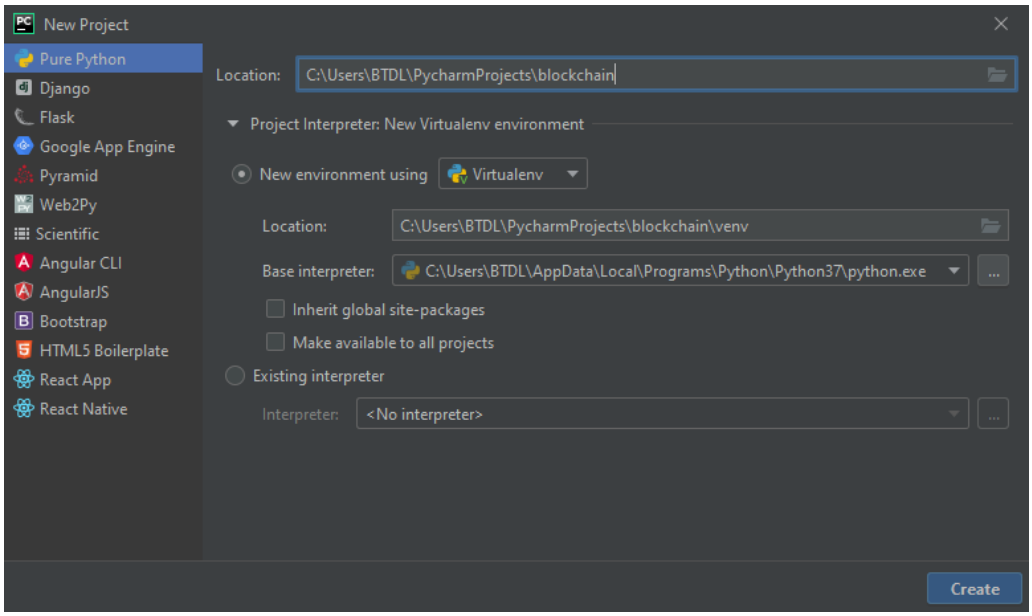


Рис. 2. Зовнішній вигляд вікна створення нового проєкту

Далі додаємо до проєкту новий файл «**File → New… → Python File**». У полі «**Name**» вводимо назву проєкту, наприклад, «**blockchain**».

Також для даного проєкту знадобляться фреймворки **Flask** та **Requests**. Щоб додати їх, необхідно зайти в налаштування проєкту« **File → Settings → Project → Project Interpreter** → “+” ». У рядку пошуку введіть **Flask**, потім натисніть **Install Package**. Аналогічно повторюємо для фреймворку **Requests**.

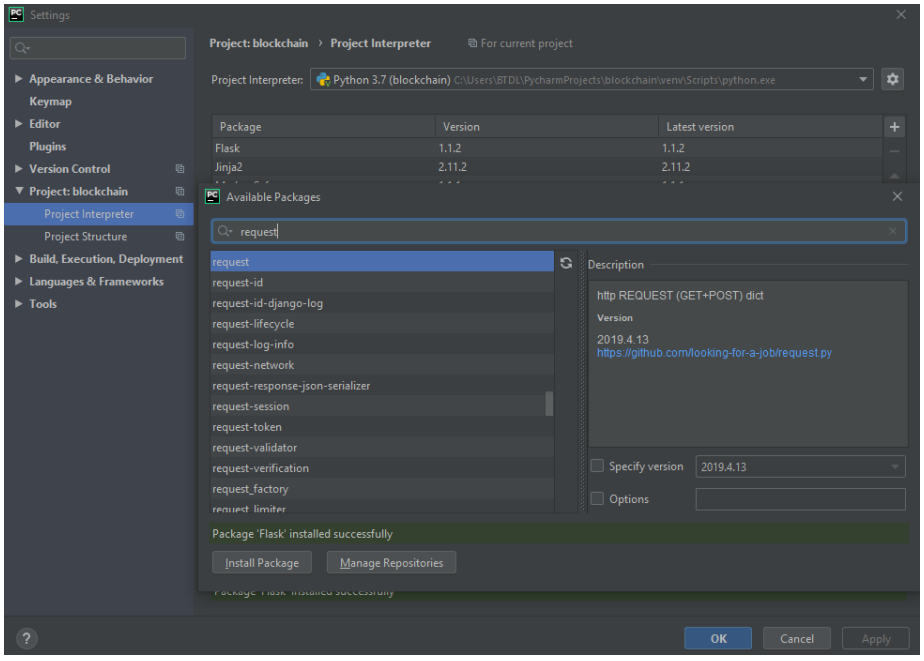


Рис. 3. Додавання фреймворку до проєкту

Тепер проєкт готовий до роботи.

1.3 Практичний приклад розгортання середовища розробки №2

Для цього прикладу будемо використовувати мову програмування Java, фреймворк Spark та бібліотеки Gson, Guava (для подальшої роботи на комп'ютері повинна бути встановлена Java 8 або новіше).

На вибір студента існує велика кількість середовищ розробки, у цьому прикладі буде описане використання Eclipse IDE.

Для завантаження потрібно перейти по посиланню та завантажити файл-встановлювач для Вашої операційної системи:

https://www.eclipse.org/downloads/download.php?file=/oomph/epp/2020-06/R/eclipse-inst-win64.exe

Після завантаження потрібно запустити файл та у вікні, що з'явиться, обрати перший пункт:

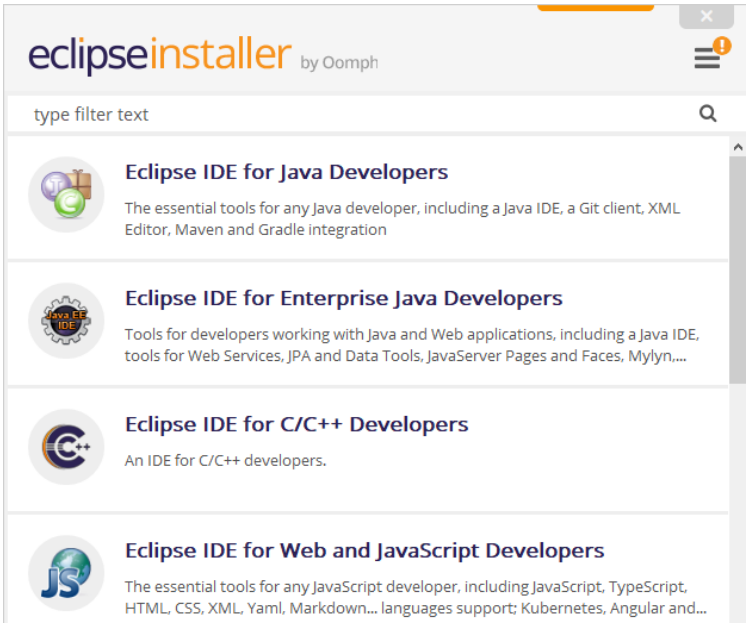


Рис.1 Вибір інсталяції Eclipse

Після встановлення IDE потрібно створити новий Maven проєкт. Для цього потрібно вибрати «**File → New → Other…**».

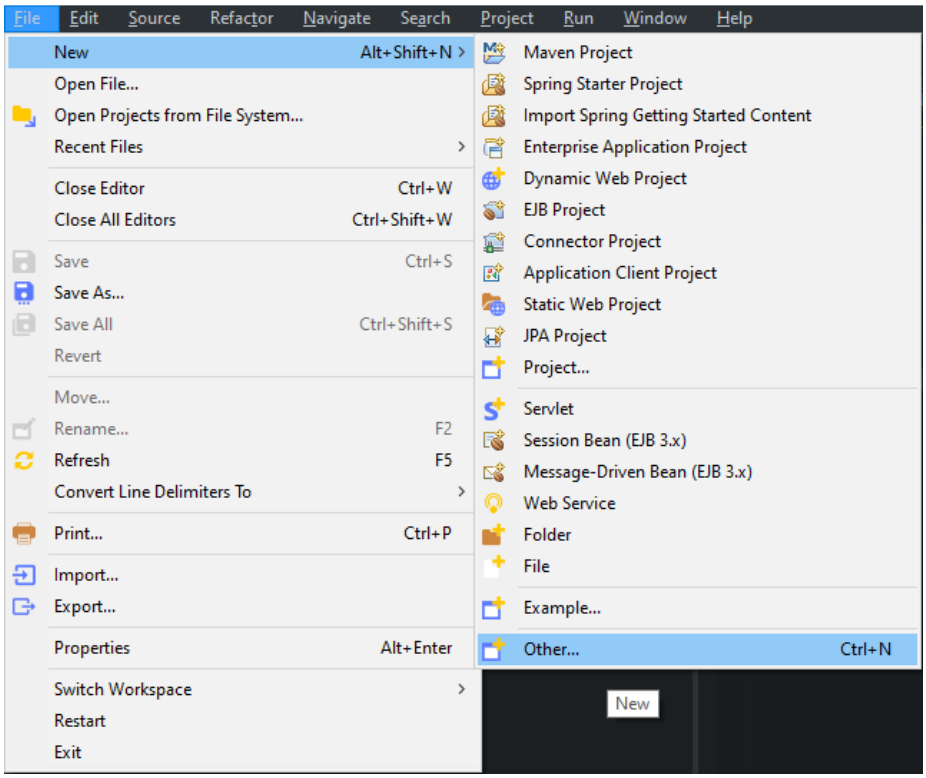


Рис.2 Вибір типу проєкта

Далі, знаходимо розділ **Maven** та обираємо пункт **Maven Project**

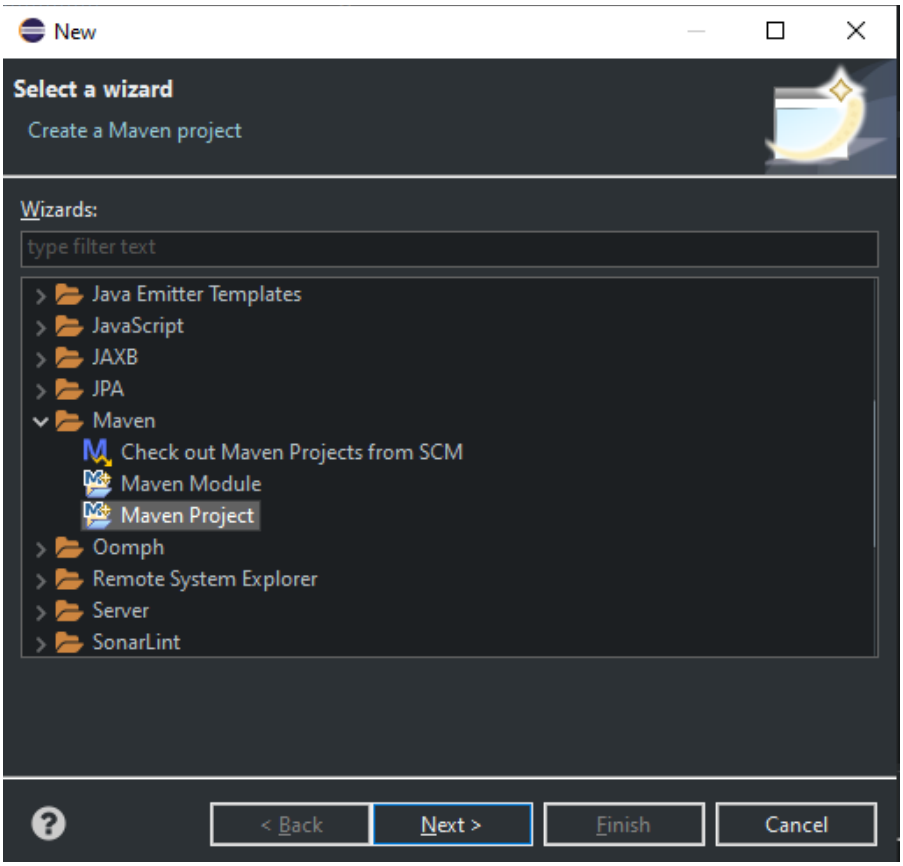


Рис.3 Продовження вибору типу проєкта

Далі, продовжуємо конфігурацію проєкта як на скріншотах

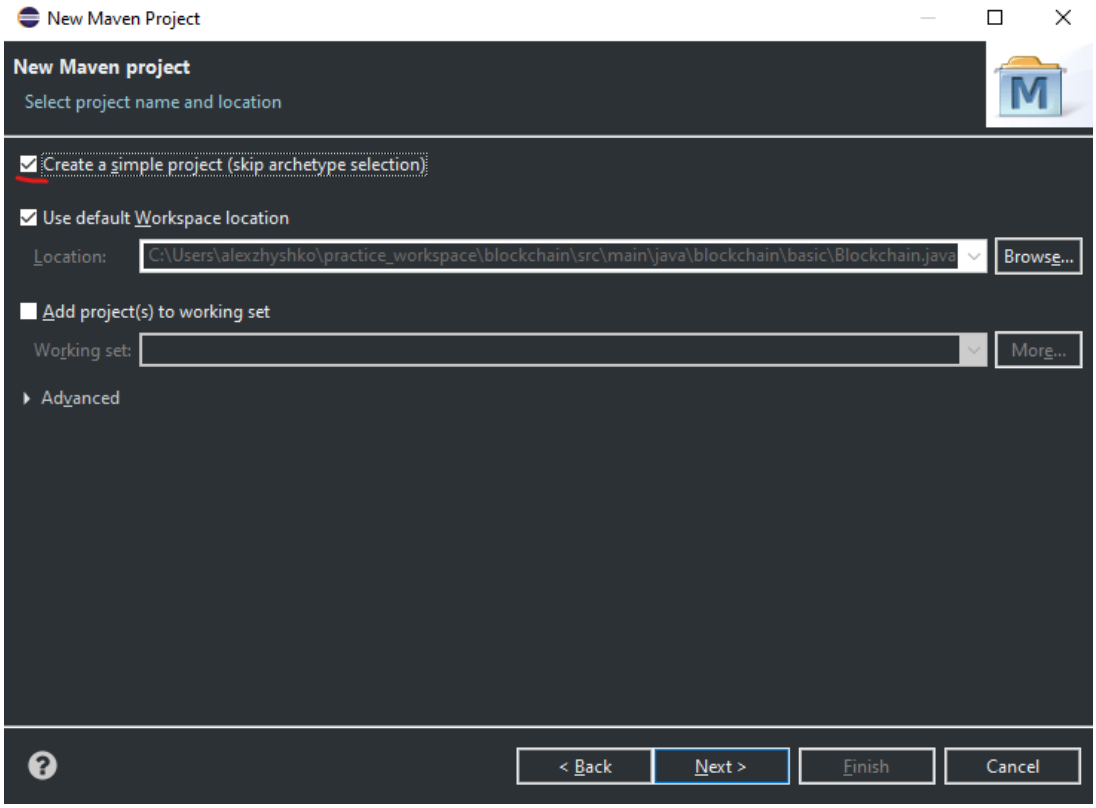


Рис.4 Створення проєкту

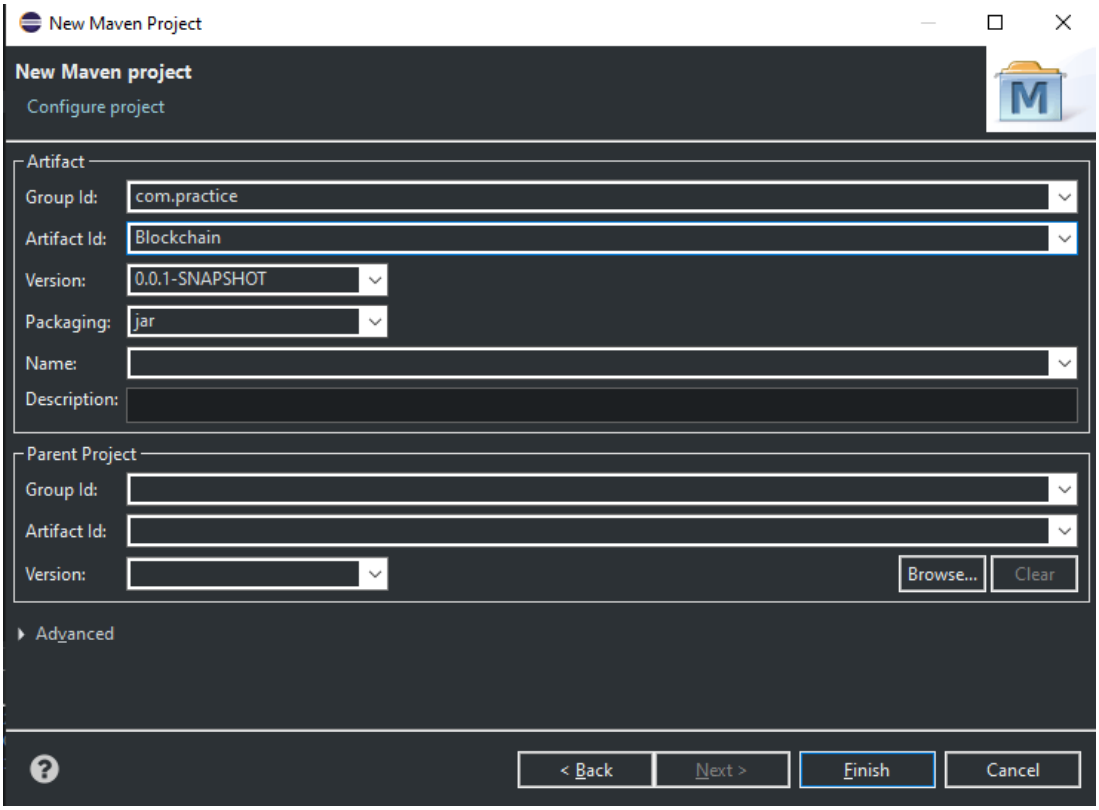


Рис.5 Продовження створення проєкту

Натискаємо Finish. Ствоення проєкту завершено.

Далі додаємо до проєкту новий файл. Правою кнопкою миші натискаємо по проєкту та обираємо пункт Class

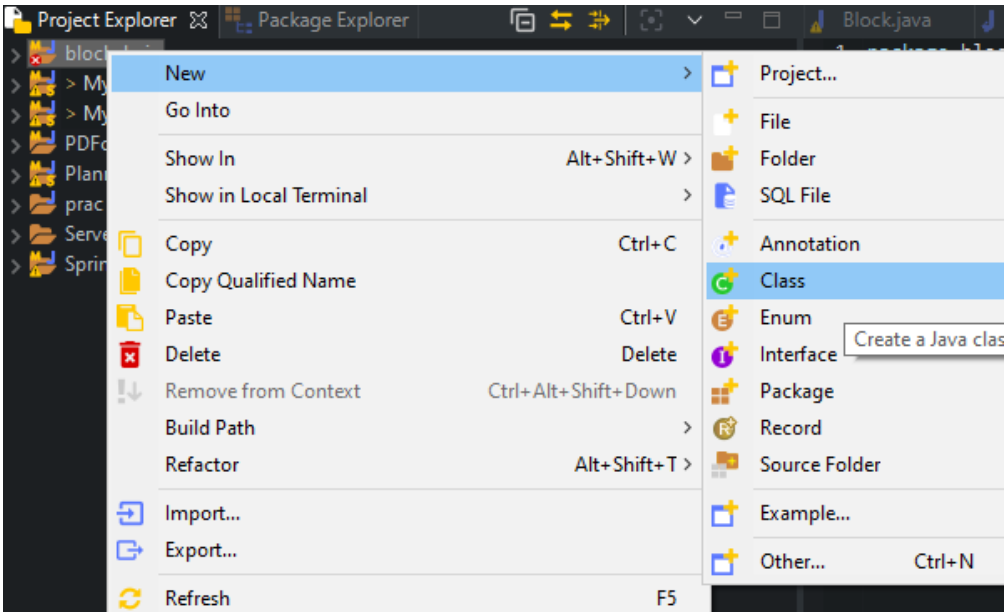


Рис.6 Додавання нового класу

У полі **«Name»** ввести назву файлу, наприклад, **«Blockchain».**

З прикладу №1 пригадуємо, що знадобиться фреймворк **Spark** та дві бібліотеки: **Gson** та **Guava.** Щоб додати його, необхідно скопіювати даний текст у файл pom.xml, що знаходиться у кореневій папці проєкту.

```
<dependencies>
    <dependency>
         <groupId>com.sparkjava </groupId>
         <artifactId>spark-core</artifactId>
         <version>2.6.0</version>
    </dependency>
    <dependency>
         <groupId>com.google.guava</groupId>
         <artifactId>guava</artifactId>
         <version>20.0</version>
    </dependency>
    <dependency>
         <groupId>com.google.code.gson</groupId>
         <artifactId>gson</artifactId>
         <version>2.8.6</version>
    </dependency>
</dependencies>
```

Файл матиме приблизно такий вигляд:

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.practice</groupId>
    <artifactId>Blockchain</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>com.sparkjava</groupId>
            <artifactId>spark-core</artifactId>
            <version>2.6.0</version>
        </dependency>
        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>20.0</version>
        </dependency>
        <dependency>
            <groupId>com.google.code.gson</groupId>
            <artifactId>gson</artifactId>
            <version>2.8.6</version>
        </dependency>
    </dependencies>
</project>
```

Рис.7 Кінцевий вигляд файлу pom.xml

Тепер проєкт готовий до роботи.

1.4 Практичний приклад створення нових блоків та PoW №1

У файлі **blockchain.py** створимо клас блокчейну, чий конструктор створює початковий порожній лист (для зберігання прототипу блокчейну), і ще один — для зберігання транзакцій. Ось макет класу:

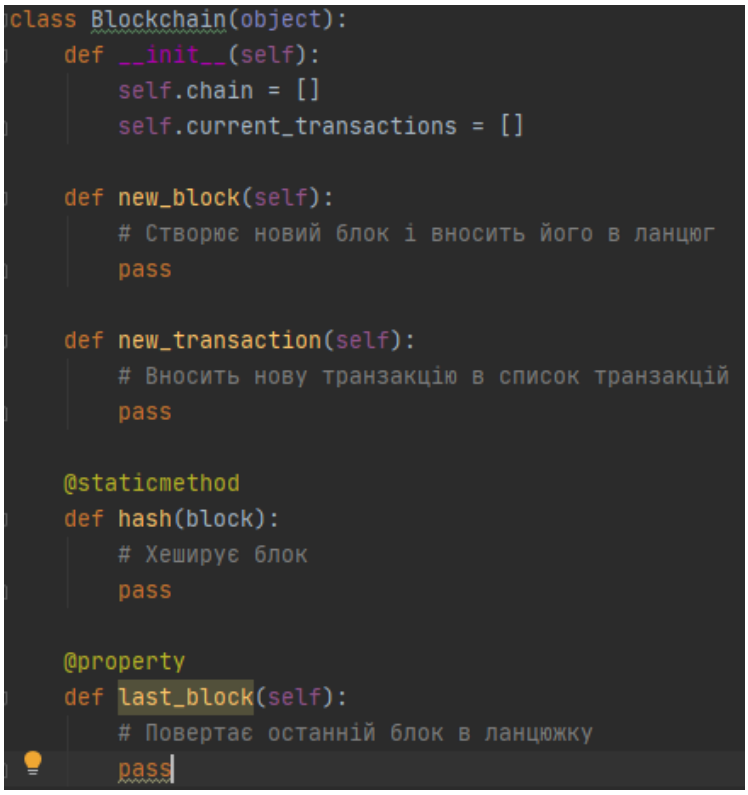


Рис. 1. Макет класу Blockchain

Клас Blockchain відповідає за управління ланцюгом. Він буде зберігати транзакції та матиме кілька допоміжних методів додавання нових блоків до ланцюга. Почнемо розробку деяких методів.

1.5 Структура блоку в блокчейні

Кожен блок містить *індекс*, *часову позначку* (час unix), *список транзакцій*, *доказ* і **хеш** попереднього блоку.

Ось приклад того, як виглядає один блок:

```
1 block = {
2     'index': 1,
3     'timestamp': 1506057125.900785,
4     'transactions': [
5         {
6             'sender': "8527147fe1f5426f9dd545de4b27ee00",
7             'recipient': "a77f5cdfa2934df3954a5c7c7da5df1f",
8             'amount': 5,
9         }],
10     'proof': 324984774000,
11     'previous_hash':
"2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824"
12 }
```

У цей момент ідея *ланцюга* повинна бути очевидною — кожен новий блок містить у собі, хеш попереднього блоку. **Це має вирішальне значення, оскільки саме це дає незмінність блокчейну**: якщо зловмисник зіпсував попередній блок у ланцюжку, то **всі** наступні блоки містять неправильні хеші.

1.6 Додавання транзакцій в блок

За **додавання транзакцій до блоку** буде відповідати метод **new_transaction()**:

```
def new_transaction(self, sender, recipient, amount):
    """
    Направляє нову транзакцію в наступний блок

    :param sender: <str> Адреса відправника
    :param recipient: <str> Адреса отримувача
    :param amount: <int> Сума
    :return: <int> Індекс блока, який буде зберігати цю транзакцію
    """

    self.current_transactions.append({
        'sender': sender,
        'recipient': recipient,
        'amount': amount,
    })

    return self.last_block['index'] + 1
```

Рис. 2. Метод new_transaction()

Після того, як **new_transaction()** додає транзакцію до списку, він повертає індекс блоку, до якого буде додана транзакція — а саме наступного, який буде замайнений (створений). У майбутньому, це буде корисно для користувача, який відправляє транзакцію.

1.7 Створення нових блоків

Після того, як екземпляр блокчейну буде створено, потрібно внести в нього **блок генезису** — перший блок без попередників. Також потрібно буде додати *"пруф"* в цей блок генезису, який являє собою результат *майнінгу* (або підтвердженням проведеної роботи). Майнінг розглядатимемо пізніше.

В доповнення до створення блоку генезису в конструкторі, також напишемо методи new_block(), new_transaction() і hash():

```
1 import hashlib
2 import json
3 from time import time
4
5 class Blockchain(object):
6     def __init__(self):
7         self.current_transactions = []
8         self.chain = []
9
10        # Створення блоку генезису
11        self.new_block(previous_hash=1, proof=100)
12
13    def new_block(self, proof, previous_hash=None):
14        """
15        Створення нового блока у блокчейні
16
17        :param proof: <int> Докази проведеної роботи
18        :param previous_hash: Хеш попереднього блока
19        :return: <dict> Новий блок
20        """
21
22        block = {
23            'index': len(self.chain) + 1,
24            'timestamp': time(),
25            'transactions': self.current_transactions,
26            'proof': proof,
27            'previous_hash': previous_hash or self.hash(self.chain[-1]),
28        }
29
30        # Перезавантаження поточного списку транзакцій
31        self.current_transactions = []
32
33        self.chain.append(block)
34        return block
```

```
35
36    def new_transaction(self, sender, recipient, amount):
37        """
38        Направляє нову транзакцію в наступний блок
39
40        :param sender: <str> Адреса відправника
41        :param recipient: <str> Адреса отримувача
42        :param amount: <int> Сума
43        :return: <int> Індекс блоку, який буде зберігати цю транзакцію
44        """
45        self.current_transactions.append({
46            'sender': sender,
47            'recipient': recipient,
48            'amount': amount,
49        })
50
51        return self.last_block['index'] + 1
52
53    @property
54    def last_block(self):
55        return self.chain[-1]
56
57    @staticmethod
58    def hash(block):
59
60        """
61        Створює хеш SHA-256 блока
62
63        :param block: <dict> Блок
64        :return: <str>
65        """
66
67        # Потрібно переконатися в тому, що словник впорядкований, інакше будуть непо-
слідовні хеші
68        block_string = json.dumps(block, sort_keys=True).encode()
69        return hashlib.sha256(block_string).hexdigest()
```

Майже закінчено зі скелетом прототипу блокчейну. Тепер дізнаємось, як створюються нові блоки.

1.8 Алгоритм підтвердження виконаної роботи

Алгоритм підтвердження роботи (Proof of Work, PoW) — це те, як нові блоки створюються або «майняться» в блокчейні. Мета PoW — це знайти число, яке вирішує проблему. Число повинне бути таким, щоб його важко було знайти, але легко підтвердити (кажучи про обчислення) ким

завгодно в мережі. Це головне завдання алгоритму.

Розглянемо простий приклад, щоб отримати краще уявлення.

Нехай хеш деякого числа *x*, помноженого на інше число повинен закінчуватися нулем. Таким чином, hash(*x* * *y*) = ac23dc...0. Для спрощеного прикладу, уявімо що *x* = 5. Як це працює в Python:

```
from hashlib import sha256

x = 5
y = 0  # Ще не знаємо, чому дорівнює y
while sha256(f'{x*y}'.encode()).hexdigest()[-1] != "0":
    y += 1

print(f'The solution is y = {y}')
```

Рішення тут наступне: y = 21, Оскільки отриманий хеш закінчується нулем:

```
hash(5 * 21) = 1253e9373e...5e3600155e860
```

1.9 Реалізація базового PoW

Давайте реалізуємо аналогічний алгоритм для прототипу блокчейну. Правило (задачу) приймемо аналогічним зазначеному раніше:

Знайдіть таке число *p*, що при хешуванні з рішенням попереднього блоку створює хеш з чотирма заголовними нулями.

```
import hashlib
import json

from time import time
from uuid import uuid4


class Blockchain(object):
    ...

    def proof_of_work(self, last_proof):
        """
        Проста перевірка алгоритму:
         - Пошук числа p`, так як hash(pp`) містить 4 заголовних нуля, де p — попередній
```

```
15        - p є попереднім доказом, а p` — новим
16
17        :param last_proof: <int>
18        :return: <int>
19        """
20
21        proof = 0
22        while self.valid_proof(last_proof, proof) is False:
23            proof += 1
24
25        return proof
26
27    @staticmethod
28    def valid_proof(last_proof, proof):
29        """
30        Підтвердження доказу: Чи містить hash(last_proof, proof) 4 заголовних нуля?
31
32        :param last_proof: <int> Попередній доказ
33        :param proof: <int> Поточний доказ
34        :return: <bool> True, якщо правильно, False, якщо ні.
35        """
36
37        guess = f'{last_proof}{proof}'.encode()
38        guess_hash = hashlib.sha256(guess).hexdigest()
39        return guess_hash[:4] == "0000"
```

Щоб скорегувати складність алгоритму, можемо змінити кількість заголовних нулів. У нашому випадку, 4 — досить. Внесення одного додаткового нуля створює колосальну різницю в часі, необхідного для пошуку рішення (**майнінгу**).

1.10 Практичний приклад створення нових блоків та PoW №2

У класі **Blockchain** створимо клас блокчейну із двома списками, один для зберігання блоків, ще один — для зберігання транзакцій. Ось макет такого класу:

```
1 package blockchain.basic;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class Blockchain {
7
8     private List<Block> chain = new ArrayList<>();
10    private List<Transaction> currentTransactions = new Ar-
      rayList<>();
11
```

```
12
13    public void newBlock() {
14
15    }
16
17    public void newTransaction() {
18
19    }
20
21    public static void hash(Block block) {
22
23    }
24
25    public Block lastBlock() {
26
27    return this.chain.size()>0?this.chain.get(this.chain.size()-
1):null;
28  }
29
30 }
```

Внесення транзакцій в блок

Попередньо потрібно створити новий клас з назвою Transaction та додати до нього декілька полів, геттери для них та конструктор:

```
1 package blockchain.basic;
2
3 public class Transaction {
4
5     private String sender;
6     private String recipient;
7     private int amount;
8
9
10    public Transaction(String sender, String recipient, int amount)
11{
12          this.sender = sender;
13          this.recipient = recipient;
14          this.amount = amount;
15    }
16
17    public String getSender() {
18          return sender;
19    }
20
21    public String getRecipient() {
22          return recipient;
23    }
24
25    public int getAmount() {
26          return amount;
27    }
28
29}
```

Далі потрібен спосіб **додавання транзакцій до блоку**. За це буде відповідати метод **newTransaction()**:

```
1     /**
2      *
3      * Направляє нову транзакцію в наступний блок
4      *
5      *
6      * @param sender    Адреса відправника
7      * @param recipient Адреса отримувача
8      * @param amount    Сума
9      * @return Індекса блока, що буде зберігати цю транзакцію
10     */
11
12    public int newTransaction(String sender, String recipient, int
      amount) {
13    this.currentTransactions.add(new Transaction(sender, recipient,
      amount));
14          return this.chain.size();
15    }
```

Після того, як **newTransaction()** додає транзакцію до списку, він повертає індекс блоку, до якого буде додана транзакція — а саме наступного, який буде замайнений. У майбутньому це буде корисно для користувача, який відправляє транзакцію.

Створення нових блоків

Створюємо новий клас Block та додаємо в нього потрібні поля, гетери для них та конструктор:

```
1 package blockchain.basic;
2
3 Import java.util.List;
4
5 public class Block {
6
7     private int index;
8     private long timestamp;
9     private List<Transaction> transactions;
10    private int proof;
11    private String previousHash;
12
13    public Block(int index, int proof, String previousHash,
      List<Transaction> transactions) {
14          this.index = index;
15          this.proof = proof;
16          this.previousHash = previousHash;
17          this.transactions = transactions;
18          this.timestamp = System.currentTimeMillis();
19    }
```

```
20
21    public int getIndex() {
22          return index;
23    }
24
25    public long getTimestamp() {
26          return timestamp;
27    }
28
29    public List<Transaction> getTransactions() {
30          return transactions;
31    }

32    public int getProof() {
33          return proof;
34    }
35
36    public String getPreviousHash() {
37          return previousHash;
38    }
39 }
```

Напишемо методи newBlock() і hash():

```
1 /**
2 *
3 * @param proof         Докази проведенної роботи
4 * @param previousHash Хеш попереднього блока
5 * @return Новий блок
6 */
7 public Block newBlock(int proof, String previousHash) {
8
9     // створюмо копію списка
10    List<Transaction> transactions = this.currentTransac-
      tions.stream().collect(Collectors.toList());
11
12    // створюємо новий об'єкт блока
13    Block newBlock = new Block(this.chain.size(), proof, previ-
      ousHash, transactions);
14
15    // очищаємо список транзакцій
16    this.currentTransactions.clear();
17
18    // додаємо новий блок у цепочку
19    this.chain.add(newBlock);
20
21    // повертаємо новий блок
22    return newBlock;
23 }
24
25 /**
26 *
27 * @param block Блок
28 * @return Хеш блока
29 */
30 public static String hash(Block block) {
31    StringBuilder hashingInputBuilder = new StringBuilder();
```

```
32
33    // додаємо параметри блока у змінну в певному незмінному по-
      рядку
34 hashingInputBuilder.append(block.getIndex())
      .append(block.getTimestamp()).append(block.getProof())
      .append(block.getPreviousHash());
36
37    String hashingInput = hashingInputBuilder.toString();

38    // генеруємо хеш блока на основі її полів за допомогою змінної
39    return Hashing.sha256().hashString(hashingInput, Standard-
      Charsets.UTF_8).toString();
40 }
```

Тепер необхідно створити конструктор, у якому буде додано блок генезису:

```
1 public Blockchain() {
2     newBlock(100, "0");
3 }
```

Реалізація базового PoW

Правило буде аналогічним зазначеному раніше:

Знайдіть число *p*, що при хешуванні з рішенням попереднього блоку створює хеш з чотирма заголовними нулями.

```
1 /**
2 * Проста перевірка алгоритму: Пошук числа p`, так як hash(pp`)
3 * містить 4
4 * заголовних нуля, де p — попередній p є попереднім доказом, а p`
— 5новим
6 *
7 * @param lastProofOfWork
8 * @return int
9 */
10 public int proofOfWork(int lastProofOfWork) {
11    int proof = 0;
12    while (!isProofValid(lastProofOfWork, proof)) {
13         proof++;
14    }
15    return proof;
16}
17
18 /**
19 * Підтвердження доказу: Чи містить hash(lastProof, proof)
      4 заголовних нуля
20 *
21 * @param lastProof
22 * @param proof
23 * @return
24 */
25 private boolean isProofValid(int lastProof, int proof) {
```

```
26
27    String guessString = Integer.toString(lastProof) + Inte-
      ger.toString(proof);
28    String guessHash = Hashing.sha256().hashString(guessString,
      StandardCharsets.UTF_8).toString();
29    return guessHash.endsWith("0000");
30 }
```

Щоб скорегувати складність алгоритму, можемо змінити кількість заголовних нулів.

1.11 Контрольні запитання

1) Для чого призначене середовище PyCharm?
2) Що потрібно виконати, щоб створити новий проєкт в PyCharm?
3) Як додавати сторонні фреймворки до проєкту?
4) Що таке блок генезису?
5) Що таке алгоритм підтвердження роботи?

1.12 Контрольне завдання

1) Засобами PyCharm (або інше) створити проєкт із назвою свого **Прізвища** (латиницею). Створити скелет прототипу блокчейну у складі: клас Блокчейн, методи Обрахунку хеш блоку та Додавання блоку в чейн. В назвах методів та змінних використовувати ідентифікатор **PIP** (латиницею перші літери Прізвища, Ім’я по-Батькові).
2) В генезис-блок передати попередній хеш зі своїм прізвищем (латиницею), а у якості **nonce** – [**деньмісяцьрік** народження].
3) У якості підтвердження доказу - наявність в кінці хешу [**місяць** народження].

Результати ЛР подати у вигляді серії скріншотів кодування та результатів роботи програми: виклик методів і доведення роботи протоколу виконаної роботи згідно цільового критерію знаходження хешу.

Компоненти програм починаються ідентифікатором **pip_**.

2 ЛАБОРАТОРНА РОБОТА № 2

Тема: Транзакції та майнінг блоків в прототипі блокчейну

Мета: Ознайомитися з фреймворком Flask, реалізувати функціонал взаємодії з прототипом блокчейну.

Завдання: Написати методи створення нової транзакції, майнінгу нового блоку та повернення ланцюга в прототипі блокчейну.

2.1 Теоретичні відомості

У цій роботі налаштуємо та побудуємо REST сервіс, що може отримувати, обробляти та відповідати на різноманітні HTTP запити клієнта. У цьому знадобиться уже згаданий вище мікрофреймворк під назвою Spark.

Цей фреймворк дозволяє розробнику у максимально короткі терміни створити власний REST сервіс. Для його запуску потрібно вказати точки входу певної структури у **main** методі класу. Після запуску сервер запускається та є доступним за замовчуванням на порті 4567 (можна змінити у конфігурації проєкта).

Навчальний приклад №1

Тепер задіємо фреймворк під назвою Flask. Це макро-фреймворк, який помітно спрощує зіставлення кінцевих точок з функціями Python. Це дозволяє взаємодіяти з прототипом блокчейну в мережі за допомогою HTTP-запитів.

Створимо три методи:

- /transactions/new для створення нової транзакції в блоці;
- /mine, щоб вказати серверу, що потрібно майнити новий блок;
- /chain для повернення всього блокчейну

2.2 Налаштування Flask

Сервер сформує єдиний вузол в мережі блокчейну. Давайте створимо шаблонний код:

```
1 import hashlib
2 import json
3 from textwrap import dedent
4 from time import time
5 from uuid import uuid4
6
7 from flask import Flask
8
9
10 class Blockchain(object):
11     ...
12
13
14 # Створюємо екземпляр вузла
15 app = Flask(__name__)
16
17 # Генеруємо унікальну на глобальному рівні адресу для цього вузла
18 node_identifier = str(uuid4()).replace('-', '')
19
20 # Створюємо екземпляр блокчейну
21 blockchain = Blockchain()
22
23  #Створення кінцевої точки /mine, яка є GET-запитом
24 @app.route('/mine', methods=['GET'])
25 def mine():
26     return "We'll mine a new Block"
27   # Створення кінцевої точки /transactions/new, яка є POST-запитом, так як будемо відп-
равляти туди дані;
28 @app.route('/transactions/new', methods=['POST'])
29 def new_transaction():
30     return "We'll add a new transaction"
31  # Створення кінцевої точки /chain, яка повертає весь блокчейн;
32 @app.route('/chain', methods=['GET'])
33 def full_chain():
34     response = {
35         'chain': blockchain.chain,
36         'length': len(blockchain.chain),
37     }
38     return jsonify(response), 200
39  # Запускає сервер на порт: 5000
40 if __name__ == '__main__':
41     app.run(host='0.0.0.0', port=5000)
```

2.3 Кінцева точка транзакцій

Ось так повинен буде виглядати запит транзакції. Це те, що користувач відправляє на сервер:

```
{
  "sender": "my address",
  "recipient": "someone else's address",
  "amount": 5
}
```

Так як вже є метод класу для додавання транзакцій в блок, справа залишилася за малим. Потрібно написати функцію для внесення транзакцій:

```
1 import hashlib
2 import json
3 from textwrap import dedent
4 from time import time
5 from uuid import uuid4
6
7 from flask import Flask, jsonify, request
8
9 ...
10
11 @app.route('/transactions/new', methods=['POST'])
12 def new_transaction():
13     values = request.get_json()
14
15     # Перевірка того, що необхідні поля знаходяться серед POST-даних
16     required = ['sender', 'recipient', 'amount']
17     if not all(k in values for k in required):
18         return 'Missing values', 400
19
20     # Створення нової транзакції
21     index = blockchain.new_transaction(values['sender'], values['recipient'], values['amount'])
22
23     response = {'message': f'Transaction will be added to Block {index}'}
24     return jsonify(response), 201
```

2.4 Кінцева точка майнінгу

Кінцева точка майнінгу — це частина, у якій:

- Рахується PoW;

- Нагороджується майнер, додавши транзакцію, і отримує 1 «коін»;
- Генерується наступний блок, додаванням його у ланцюг.

```
1 import hashlib
2 import json
3
4 from time import time
5 from uuid import uuid4
6
7 from flask import Flask, jsonify, request
8
9...
10
11 @app.route('/mine', methods=['GET'])
12 def mine():
13     # Запускаємо алгоритм підтвердження роботи, щоб отримати наступний пруф
14     last_block = blockchain.last_block
15     last_proof = last_block['proof']
16     proof = blockchain.proof_of_work(last_proof)
17
18     # Повинні отримати винагороду за знайдене підтвердження
19     # Відправник "0" означає, що вузол заробив коін
20     blockchain.new_transaction(
21         sender="0",
22         recipient=node_identifier,
23         amount=1,
24     )
26     # Створюємо новий блок, шляхом внесення його в ланцюг
27     previous_hash = blockchain.hash(last_block)
28     block = blockchain.new_block(proof, previous_hash)
29     response = {
30         'message': "New Block Forged",
31         'index': block['index'],
32         'transactions': block['transactions'],
33         'proof': block['proof'],
34         'previous_hash': block['previous_hash'],
35     }
36     return jsonify(response), 200
```

Зауважте, що отримувачем замайненого блоку є адреса активного вузла. І більшість того, що тут зроблено, — це лише взаємодія з методами Blockchain-класу. На даний момент базовий функціонал закінчено і можемо розпочати взаємодію з блокчейном.

Навчальний приклад №2

Використаємо фреймворк під назвою Spark. Це мікрофреймворк, що дозволяє зручно та швидко розробляти REST сервіси.

2.5 Налаштування Spark

Повторюємо, що сервер сформує єдиний вузол в мережі блокчейну. За замовчуванням, Spark працює на порті 4567. Давайте створимо шаблонний код:

```
1 public class Controller {
2
3    public static void main(String[] args) {
4
5          Spark.port(4568);
6          Blockchain blockchain = new Blockchain();
7          Gson gson = new Gson();
8          get("/mine", (req, res) -> {
9
10         });
11
12         post("/transactions/new", (req, res) -> {
13
14         });
15
16         get("/chain", (req, res) -> {
17
18         });
19
20   }
21
22 }
```

Кінцева точка транзакцій

Ось так повинен виглядати запит транзакції. Це те, що користувач відправляє на сервер:

```
{
  "sender": "my address",
  "recipient": "someone else's address",
  "amount": 5
}
```

Давайте напишемо функцію для внесення транзакцій:

```
1 post("/transactions/new", (req, res) -> {
2    try {
```

```
3     // створюємо об'єкт транзакції за отриманими даними з запиту
4     Transaction transaction = gson.fromJson(req.body(), Transac-
      tion.class);
5
6           // додаємо нову транзакцію
7     int index =blockchain.newTransaction(transaction.getSender(),
      transaction.getRecipient(),.getAmount());
8
9           // якщо помилок не відбулося, то відправляємо відповідь
10          res.status(201);
11          return "Transaction will be added to Block " + index;
12          } catch (JsonSyntaxException e) {
13
14    // якщо запит має неправильну структуру - повертаємо помилку
15          res.status(400);
16          return "Invalid JSON";
17    }
18 });
```

Рис. 2 Обробка транзакцій

Кінцева точка майнінгу

```
1 get("/mine", (req, res) -> {
2
3     // отримуємо останній блок у блокчейні
4     Block lastBlock = blockchain.lastBlock();
5
6     // отримуємо останнє підтвердження роботи
7     int lastProof = lastBlock.getProof();
8
9     // отримуємо нове підтвердження роботи
10    int proofOfWork = blockchain.proofOfWork(lastProof);
11
12    // створюємо нову транзакцію
13    blockchain.newTransaction("0",
      UUID.randomUUID().toString().replace("-", ""), 1);
14
15    // отримуємо останній хеш у блокчейні
16    String lastHash = Blockchain.hash(lastBlock);
17
18    // створюємо новий блок
19    Block newBlock = blockchain.newBlock(proofOfWork, lastHash);
20
21    // створюємо рядок, що є відображенням нового блока
22    String json = gson.toJson(newBlock);
23
24    // ставимо статус відповіді 200
25    res.status(200);
26
27    // відправляємо відомості про новий блок
28    return json;

29 });
```

2.6 Контрольні запитання

1) Для чого призначений фреймворк Flask або Spark?
2) Що таке кінцева точка майнінгу?

2.7 Контрольне завдання

1) Реалізувати функціонал додавання транзакцій (в мемпул).
2) Захешувати транзакції в меркель-дерево, виконати майнінг блоку із транзакціями.

Результати ЛР подати у вигляді скріншотів кодування та результатів роботи програми: виклик методів і доведення отримання винагороди в результаті майнінгу.

Компоненти програм починаються ідентифікатором **pip_**.

3 ЛАБОРАТОРНА РОБОТА № 3

Тема: Взаємодія з прототипом блокчейну засобами Postman

Мета: Ознайомитися з додатком Postman, навчитися взаємодіяти з прототипом блокчейну за допомогою HTTP-запитів.

Завдання: за допомогою додатка Postman створити GET-та POST-запити до прототипу блокчейну.

3.1 Теоретичні відомості

Щоб взаємодіяти з API прототипу блокчейну, потрібно встановити додаток Postman (https://www.postman.com/downloads/).

Postman — зручний HTTP-клієнт для тестування веб-сайтів та API. За допомогою додатку можна складати і редагувати прості або складні HTTP-запити. Складні запити автоматично зберігаються на майбутнє для повторного застосування. Відповіді від сервера можна зберігати як файли на жорсткому диску. Таким чином, Postman економить купу часу веб-розробникам.

3.2 Запуск сервісу Postman

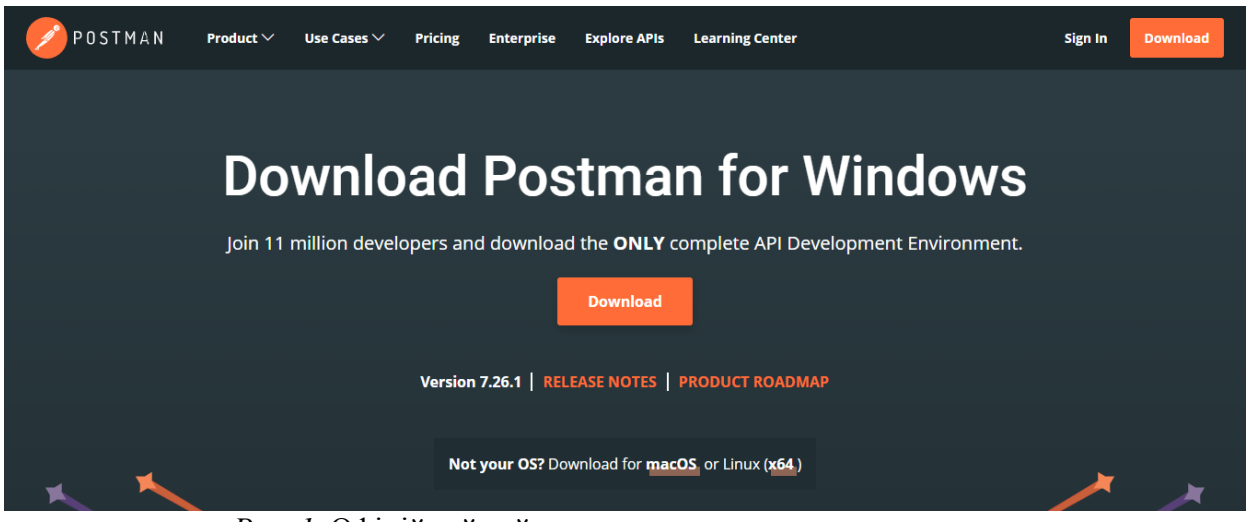


Рис. 1. Офіційний сайт, де можна завантажити додаток

Запускаємо сервер:

Python:

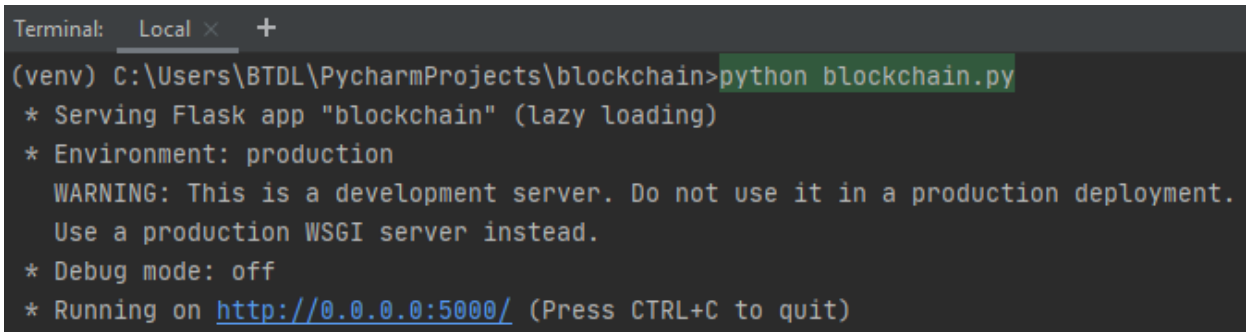


Рис. 2. Вікно терміналу, з завантаженим файлом blockchein.py

Java:

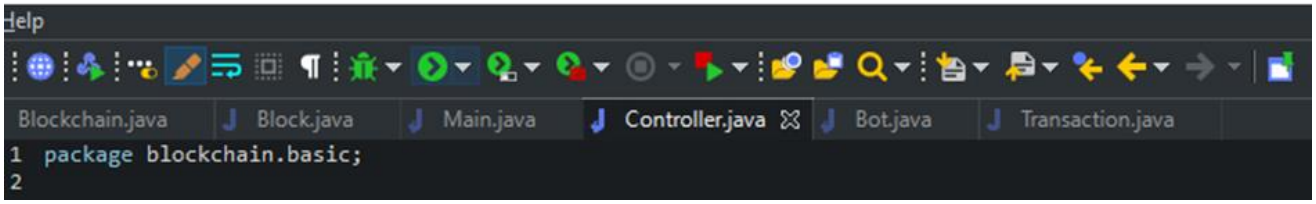


Рис.3 Запуск класу Controller.java

У випадку використання Java та Spark, далі порт 5000 слід замінити на 4567, оскільки за замовчуванням Spark використовує саме порт 4567.

Відповідь Flask та Spark може дещо відрізнятися.

3.3 Майнінг блоку в Postman

Тепер спробуємо майнити блок, створивши GET-запит до вузла http://localhost:5000/:

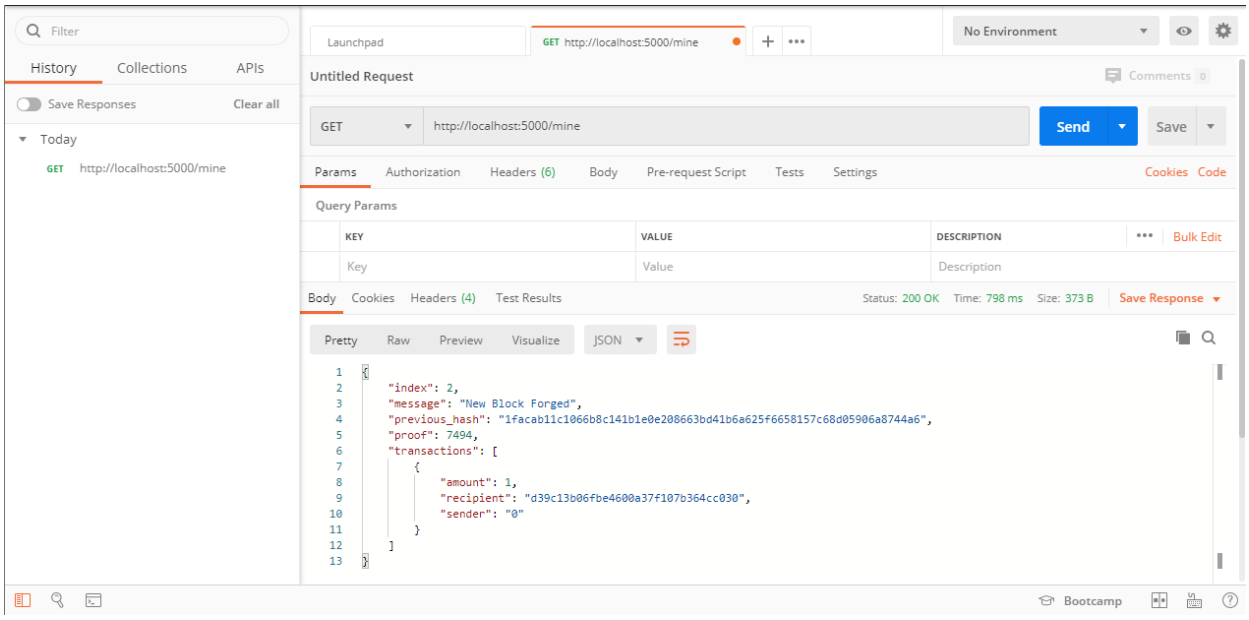


Рис. 3. GET-запит http://localhost:5000/mine у Postman

3.4 Додавання транзакцій в Postman

Тепер, давайте створимо нову транзакцію, відправивши POST-запит до вузла http://localhost:5000/transactions/new з тілом, що містить структуру потрібної транзакції:

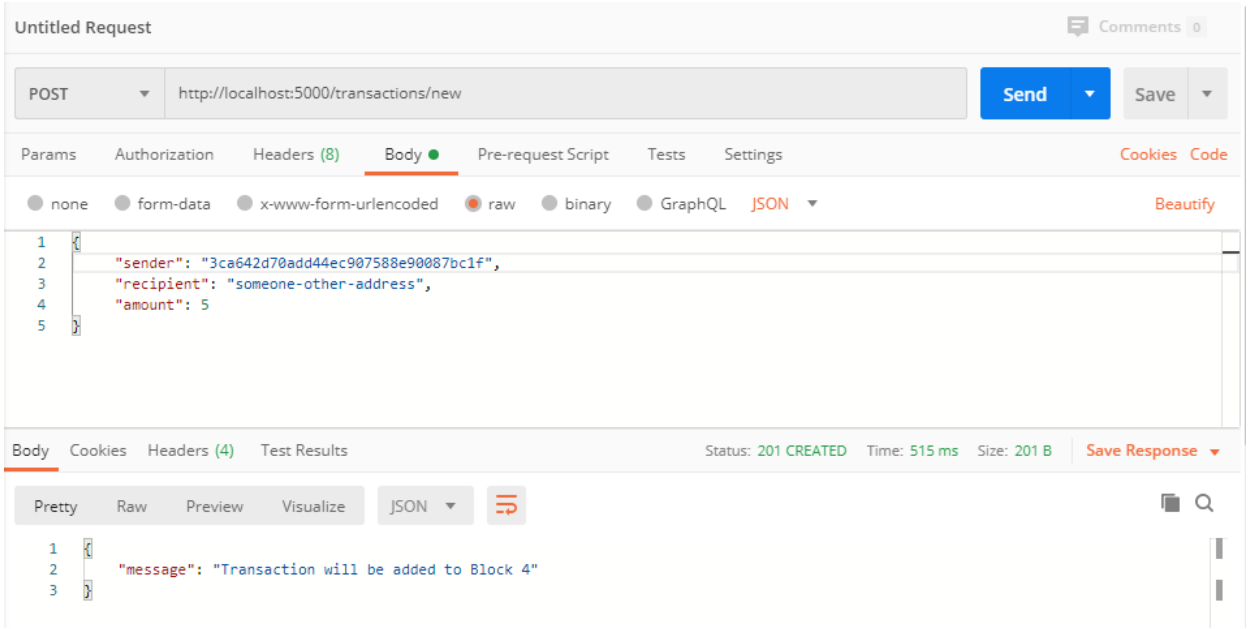


Рис. 4. POST-запит http://localhost:5000/transactions/new у Postman

3.5 Виведення всього ланцюга блокчейну

Тепер перевіримо весь ланцюжок, виконавши запит до вузла http://localhost:5000/chain. Отримаєм наступне:


```
{
    "chain": [
        {
            "index": 1,
            "previous_hash": "1",
            "proof": 100,
            "timestamp": 1592745911.531421,
            "transactions": []
        },
        {
            "index": 2,



            "previous_hash": "de0fed066542832d2c2c8ae17f4d45768ecc8345d13288935d114
4b3f37e0c6f",
            "proof": 62705,
            "timestamp": 1592745916.5031183,
            "transactions": [
                {
                    "amount": 1,
                    "recipient": "3ca642d70add44ec907588e90087bc1f",
                    "sender": "0"
                }
            ]
        },
        {
            "index": 3,
            "previous_hash": "58c1de6411473a27d556e8ad87d69e38f830f82a707772afef763
dcabfb4f74f",
            "proof": 51364,
            "timestamp": 1592745920.225162,
            "transactions": [
                {
                    "amount": 1,
                    "recipient": "3ca642d70add44ec907588e90087bc1f",
                    "sender": "0"
                }
            ]
        }
    ],
    "length": 3
}
```

3.6 Контрольні запитання

1) Що таке Postman? Навіщо він потрібен?
2) Що таке GET-запит?
3) Що таке POST-запит?

3.7 Контрольне завдання

1) Намайнити один блок та отримати винагороду у розмірі [**день** народження] монет.
2) Додати транзакцію передачі [**день** народження] монет відносно наявної каси.

Компоненти програм починаються ідентифікатором **pip_**.

4 ЛАБОРАТОРНА РОБОТА № 4

Тема: Організація консенсусу в прототипі блокчейну

Мета: Ознайомитися з алгоритмом Консенсусу.

Завдання: Реалізувати базовий алгоритм консенсусу, створити метод для реєстрації децентралізованих вузлів у мережі.

4.1 Теоретичні відомості

Отже в минулих роботах розроблено базовий блокчейн, який приймає транзакції і дає можливість майнити нові блоки. Але вся суть блокчейну в тому, що вони повинні бути децентралізованими. А якщо вони децентралізовані, тоді яким чином гарантувати, що всі вони відображають один і той самій ланцюг?

Це називається **проблемою Консенсусу**, і тепер доведеться реалізувати алгоритм Консенсусу, якщо буде потрібно більше одного вузла в робочій мережі блокчейну.

Практичний приклад №1

Реєстрація нових вузлів

Аби реалізувати **алгоритм Консенсусу**, потрібно знайти спосіб дати певному вузлу знати про існування сусідніх вузлів в мережі. Кожен вузол в робочому ланцюзі повинен містити регістр інших вузлів в ланцюзі. Отже, знадобитися ще кілька кінцевих точок:

- /nodes/register для прийняття списку нових вузлів в формі URL-ів;
- /nodes/resolve для реалізації алгоритму Консенсусу, який вирішує будь-які конфлікти, пов'язані з підтвердженням того, що вузол знаходиться у своєму ланцюзі.

Також потрібно буде змінити конструктор класу Blockchain і додати метод для реєстрації вузлів:

```
1 ...
2 from urllib.parse import urlparse
3 ...
4
5
6 class Blockchain(object):
7     def __init__(self):
8         ...
9         self.nodes = set()
10         ...
11
12     def register_node(self, address):
13         """
14         Вносимо новий вузол у список вузлів
15
16         :param address: <str> адреса вузла, іншими словами: 'http://192.168.0.5:5000'
17         :return: None
18         """
19
20         parsed_url = urlparse(address)
21         self.nodes.add(parsed_url.netloc)
```

Зверніть увагу на те, що використовується метод set() для зберігання списку вузлів. Це легкий спосіб переконатися в тому, що внесення нових вузлів є ідемпотентним — це означає, що незалежно від того, скільки разів вноситься певний вузол у список, він виникне лише один раз.

4.2 Реалізація алгоритму Консенсусу

Як відомо, конфлікт полягає в тому, що один вузол має ланцюг, пов'язаний з іншим вузлом. Щоб вирішити це, введемо правило, де найдовший і валідний ланцюг є авторитетним. Іншими словами, найдовший ланцюг мережі де-факто є єдиним. Використовуючи цей алгоритм, досягнемо Консенсусу серед вузлів робочої мережі.

```
1 ...
2 import requests
3
4
5 class Blockchain(object)
6     ...
7
```

```
8   def valid_chain(self, chain):
9       """
10      Перевіряємо, чи є внесений в блок хеш коректним
11
12      :param chain: <list> blockchain
13      :return: <bool>
14      """
15
16      last_block = chain[0]
17      current_index = 1
18
19      while current_index < len(chain):
20          block = chain[current_index]
21          print(f'{last_block}')
22          print(f'{block}')
23          print("\n-----------\n")
24          # Перевірка правильності хеша блока
25          if block['previous_hash'] != self.hash(last_block):
26              return False
27
28          # Перевіряємо, чи є підтвердження роботи коректним
29          if not self.valid_proof(last_block['proof'], block['proof']):
30              return False
31
32          last_block = block
33          current_index += 1
34
35      return True
36
37    def resolve_conflicts(self):
38      """
39      Це алгоритм Консенсусу, він вирішує конфлікти,
40      Змінюючи ланцюг на найдовший в мережі
41
42      :return: <bool> True, якщо ланцюг був змінений, False, якщо ні.
43      """
44
45      neighbours = self.nodes
46      new_chain = None
47
48      # Шукаємо тільки ланцюги, довші за наші
49      max_length = len(self.chain)
50
51      # Захоплюємо і перевіряємо всі ланцюги з усіх вузлів мережі
52      for node in neighbours:
53          response = requests.get(f'http://{node}/chain')
54
55          if response.status_code == 200:
56              length = response.json()['length']
57              chain = response.json()['chain']
58
59              # Перевіряємо, чи є довжина найдовшою, а ланцюг — валідним
```

```
60              if length > max_length and self.valid_chain(chain):
61                  max_length = length
62                  new_chain = chain
63
64          # Замінюємо ланцюг, якщо знайдемо інший валідний і довший
65          if new_chain:
66              self.chain = new_chain
67              return True
68
69          return False
```

Перший метод **valid_chain()** відповідає за перевірку того, чи є ланцюг валідним, запустивши цикл через кожен блок і проводячи верифікацію як хешу, так і пруфу.

Метод **resolve_conflicts()**, який запускає цикл через всі сусідні вузли, завантажує їх ланцюги і проводить перевірку, як і в попередньому методі. Якщо валідний ланцюг, довжина якого більше, ніж перевірочний, ми його замінюємо на перевірочний.

Зареєструємо дві кінцеві точки робочого API, одну для внесення сусідніх вузлів, а другу — для вирішення конфліктів:

```
1 @app.route('/nodes/register', methods=['POST'])
2 def register_nodes():
3     values = request.get_json()
4
5     nodes = values.get('nodes')
6     if nodes is None:
7         return "Error: Please supply a valid list of nodes", 400
8
9     for node in nodes:
10        blockchain.register_node(node)
11
12    response = {
13        'message': 'New nodes have been added',
14        'total_nodes': list(blockchain.nodes),
15    }
16    return jsonify(response), 201
17
18
19
20 @app.route('/nodes/resolve', methods=['GET'])
21 def consensus():
22    replaced = blockchain.resolve_conflicts()
```

```
23
24    if replaced:
25        response = {
26            'message': 'Our chain was replaced',
27            'new_chain': blockchain.chain
28        }
29    else:
30        response = {
31            'message': 'Our chain is authoritative',
32            'chain': blockchain.chain
33        }
34
35    return jsonify(response), 200
```

4.3 Реєстрація вузлів в локальній мережі

Тепер можна сісти за інший комп'ютер, і розгорнути різні вузли в локальній мережі. Також можна розгорнути процеси за допомогою різних портів на одному і тому ж комп'ютері. Спробуємо розгорнути ще один вузол на власному комп'ютері, але на іншому порті і зареєструвати його за допомогою поточного вузла. Таким чином, отримаємо два вузли: http://localhost:5000 і http://localhost:5001.

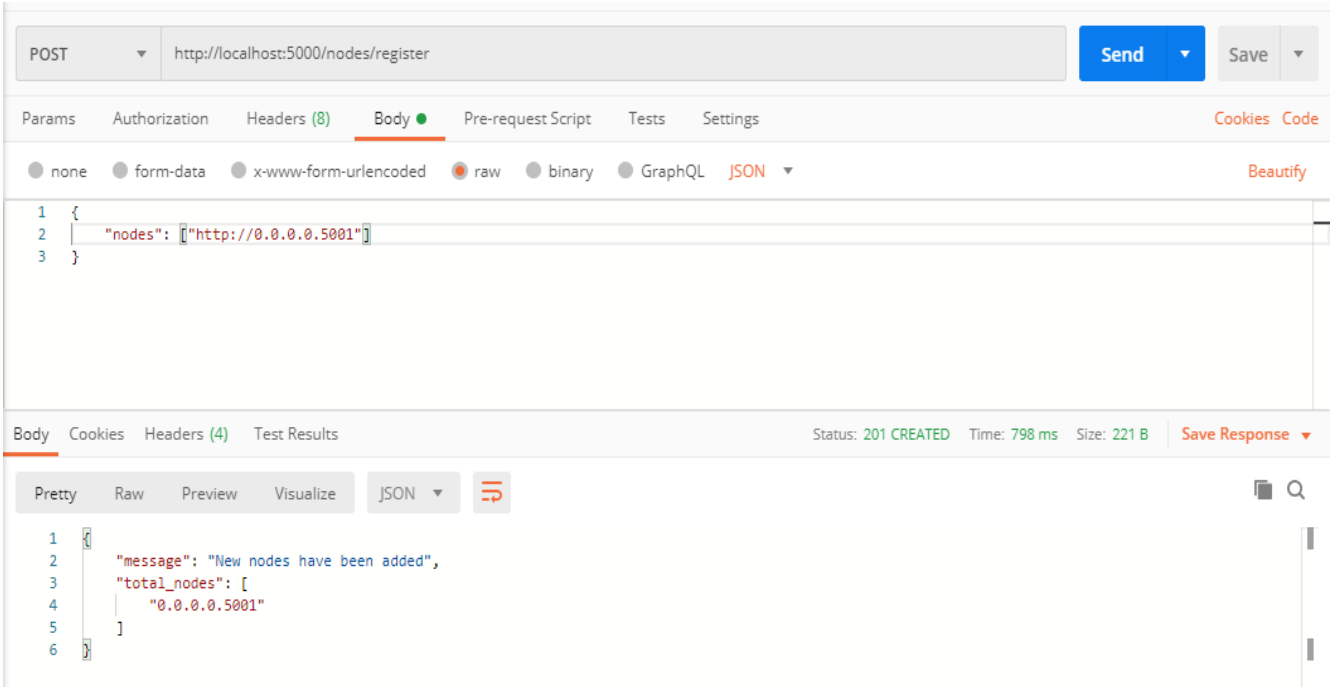


Рис. 1. POST-запит http://localhost:5000/node/register у Postman

Після цього отримаємо два нові блоки в вузлі 2, щоб переконатися в

тому, що ланцюг буде довшим. Після цього, викликаємо GET /nodes/resolve в вузлі 1, де ланцюг був замінений алгоритмом Консенсусу:

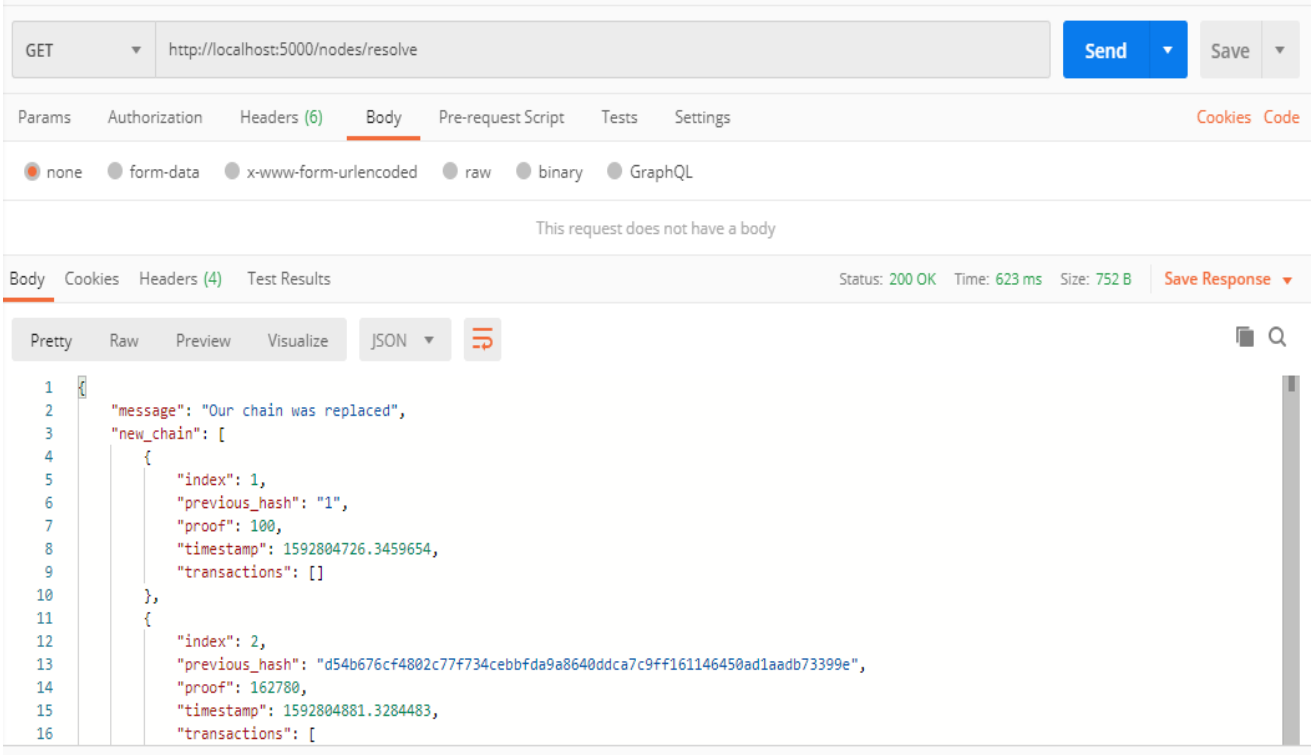


Рис. 2. GET-запит http://localhost:5000/node/resolve у Postman

Практичний приклад №2

Перед початком роботи необхідно створити два таких класи для перетворення тіла запитів у зручний вигляд:

```
1 package blockchain.basic;
2
3 import java.util.ArrayList;
4 import java.util.List;
5
6 public class ChainResponse {
7
8    List<Block> chain = new ArrayList<>();
9    int length;
10
11   public ChainResponse(List<Block> chain, int length) {
12         this.chain = chain;
13         this.length = length;
14   }
15 }
```

```
1 package blockchain.basic;
2
3 import java.util.List;
4
5 public class NodesResponse {
6
7    List<String> nodes;
8
9 }
```

Реєстрація нових вузлів

Додаємо методи для реєстрації вузлів

```
1 private Set<String> nodes = new HashSet<>();
2
3
4 public void registerNode(String netloc) {
5     nodes.add(netloc);
6 }
```

Реалізація алгоритму консенсусу

```
1 public boolean validChain(List<Block> chain) {
2     for (int i = 1; i < chain.size(); i++) {
3          Block lastBlock = chain.get(i - 1);
4          Block currentBlock = chain.get(i);
```

```
5          if (!currentBlock.getPrevi-
ousHash().equals(hash(lastBlock))) {
6                System.out.println("Hash don't match");
7                return false;
8          }
9          if (!isProofValid(lastBlock.getProof(), cur-
rentBlock.getProof())) {
10               System.out.println("Proof is not valid");
11               return false;
12         }
13   }
14   return true;
15 }
16 public boolean resolveConflicts() {
17   Gson gson = new Gson();
18   int maxLen = this.chain.size();
19   List<Block> newChain = this.chain;
20   try {
21         for (String host : this.nodes) {
22               URL url;
23               url = new URL(host + "/chain");
24               HttpURLConnection con = (HttpURLConnection)
url.openConnection();
25               con.setRequestMethod("GET");
26               int status = con.getResponseCode();
27               if (status == 200) {
28                     BufferedReader in = new Buff-
eredReader(new InputStreamReader(con.getInputStream()));
29                     String inputLine;
30                     StringBuffer content = new String-
Buffer();
31                     while ((inputLine = in.readLine()) !=
null) {
32                           content.append(inputLine);
33                     }
34                     in.close();
35                     con.disconnect();
36                     ChainResponse response =
gson.fromJson(content.toString(), ChainResponse.class);
37               if (response.length > maxLen && validChain(re-
     sponse.chain)) {
38                           maxLen = response.length;
39                           newChain = response.chain;
40                     }
41               }
42         }
43   } catch (IOException e) {
44         e.printStackTrace();
45   }
46   if(newChain != this.chain) {
47         this.chain = newChain;
48   }
49   return newChain == this.chain;
50 }
```

4.4 Створення кінцевих точок API

```
1 post("/nodes/register", (req, res) -> {
2     try {
3     List<String> nodes = gson.fromJson(req.body(), NodesRe-
      sponse.class).nodes;
4                 for(String node : nodes) {
5                       blockchain.registerNode(node);
6                 }
7                 return gson.toJson(blockchain.getNodes());
8     } catch (Exception e) {
9           res.status(400);
10          return "Incorrect host address";
11    }
12 });
13
14 get("/nodes/resolve", (req, res) -> {
15    if(blockchain.resolveConflicts()) {
16          return gson.toJson(new ChainResponse(block-
chain.getChain(), blockchain.getChain().size()));
17    }else {
18          return gson.toJson(new ChainResponse(block-
chain.getChain(), blockchain.getChain().size()));
19    }
20
21 });
```

Тепер запускаємо два сервери на різних портах робочого комп'ютера та реєструємо обидва у кожному з серверів:

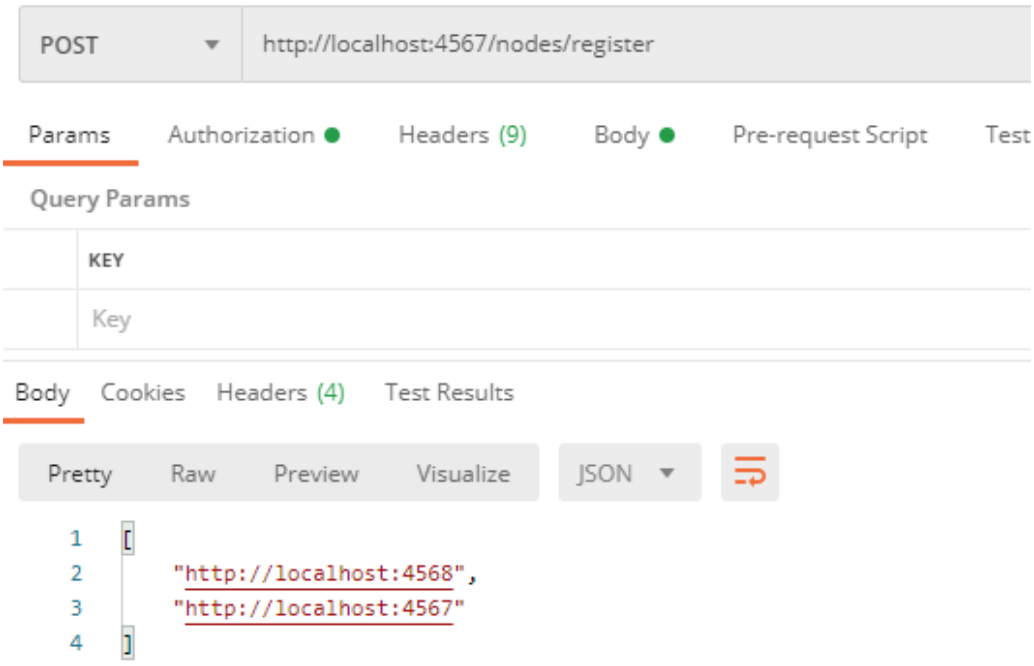


Рис.4 Запит про реєстрацію вузлів на перший сервер

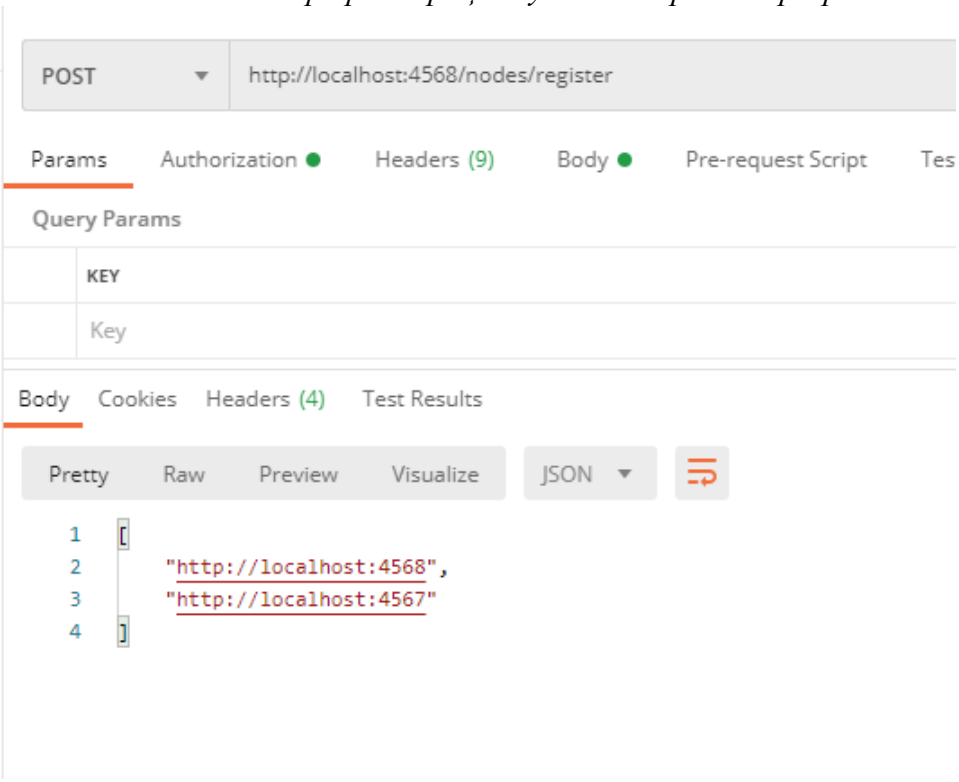


Рис.5 Запит про реєстрацію вузлів на другий сервер

Далі, створюємо ще один блок на першому вузлі:

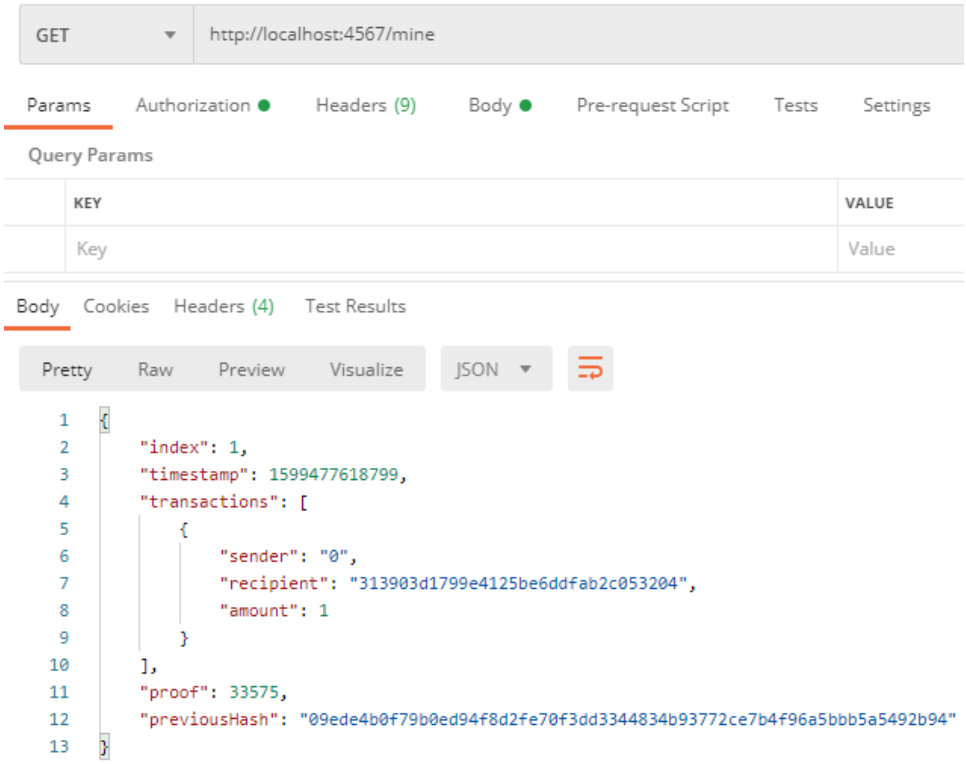


Рис.6 Запит на додавання блока першому вузлу

Далі, запускаємо процес Консенсусу на вузлі 2.

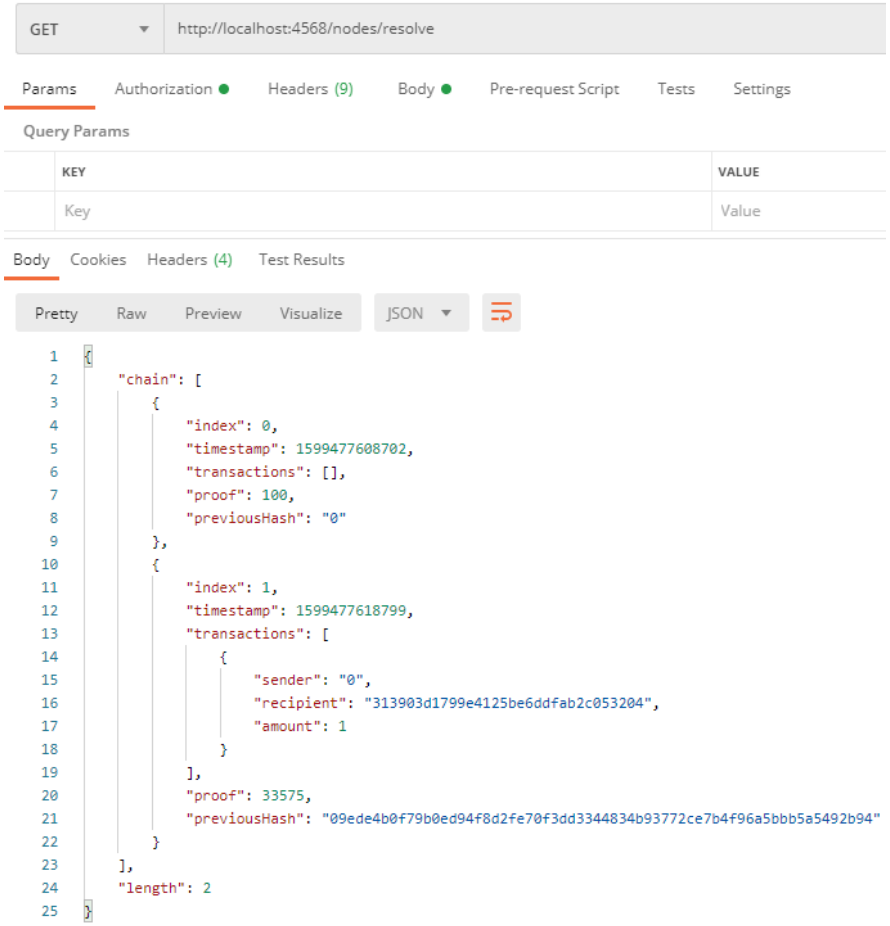


Рис.7 Запит для запуску Консенсусу на другому вузлі

Бачимо, що результат запиту є списком блоків першого вузла. Отже, можна зробити висновок, що вузол 2 валідував послідовність вузла 1 та записав її собі, як правильну. Механізм працює правильно.

4.5 Контрольні запитання

1) Що таке проблема Консенсусу?
2) Яким чином можна переконатися в тому, що ланцюг є валідним?

4.6 Контрольне завдання

1) Забезпечити роботу заданої кількості вузлів в мережі блокчейн – ціле ([**місяць** народження]/3), але не менше 2х та не більше 4х.
2) Протестувати алгоритм консенсусу.

Компоненти програм починаються ідентифікатором **pip_.**

5 ЛАБОРАТОРНА РОБОТА № 5

Тема: Транзакції в блокчейні Bitcoin

Мета: Ознайомитися із тестовою мережею Bitcoin.

Завдання: Створити онлайн-гаманець в тестовій мережі блокчейну Bitcoin. Отримати тестові монети. Передати монети до іншого гаманця.

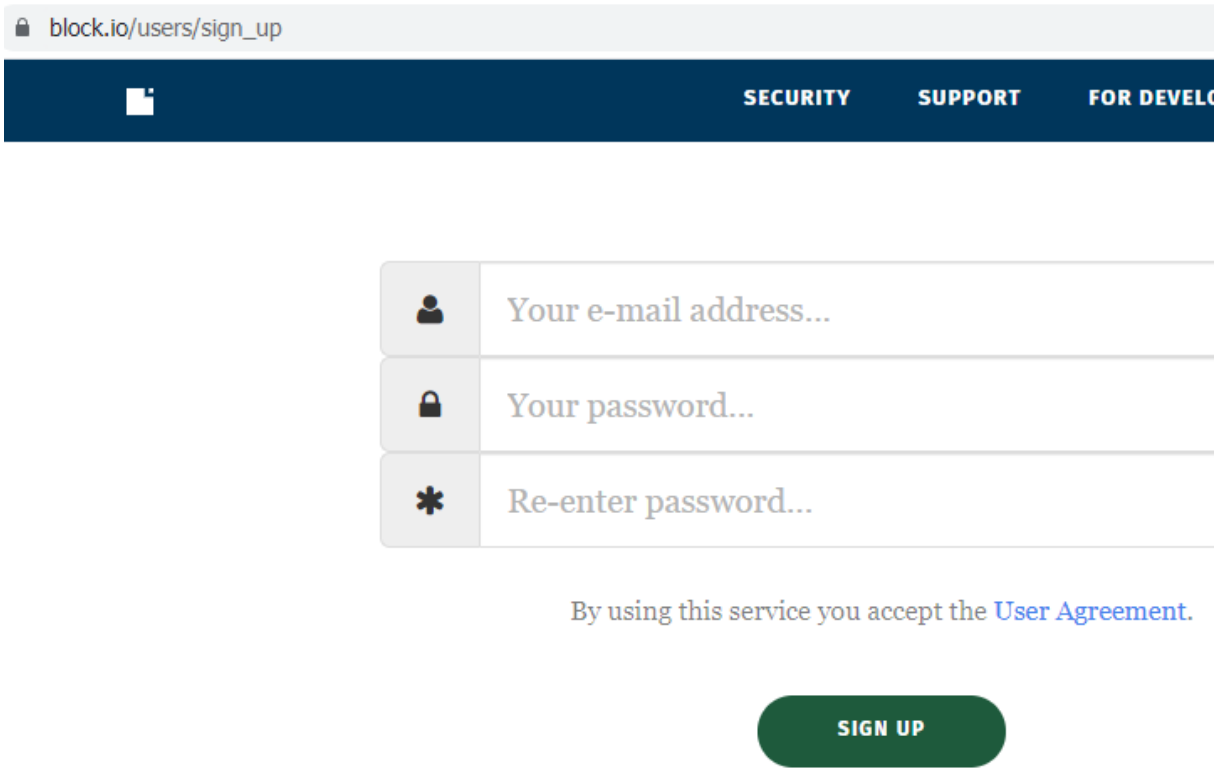
5.1 Теоретичні відомості

Testnet - це альтернативний блокчейн біткойнов, який розробники використовують для тестування. Монети тестової мережі Testnet **не мають ніякої цінності**. Розробники використовують тестову мережу, щоб експериментувати з блокчейном, не використовуючи справжні біткойни і не турбуючись про розрив (форк) основного ланцюжка.

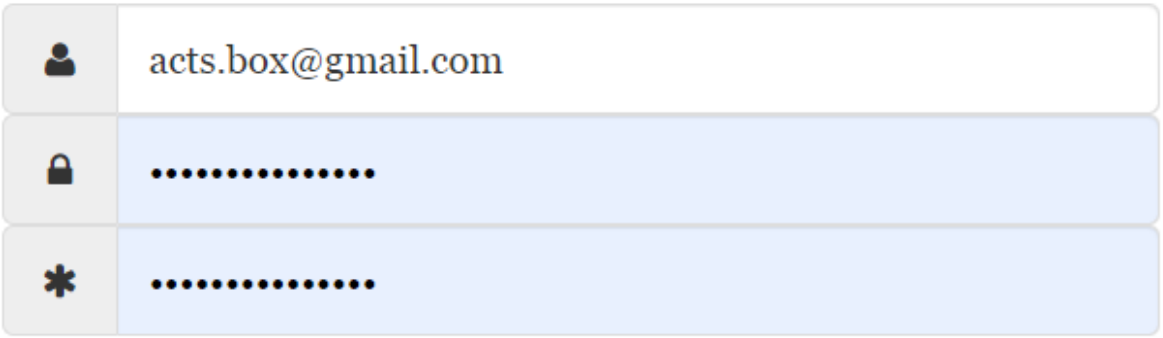
5.2 Приклад виконання

Найпростіший спосіб отримати гаманець для тестових біткойнів в мережі testnet - скористатися онлайн-гаманцем. Це те що треба для токенів testnet.

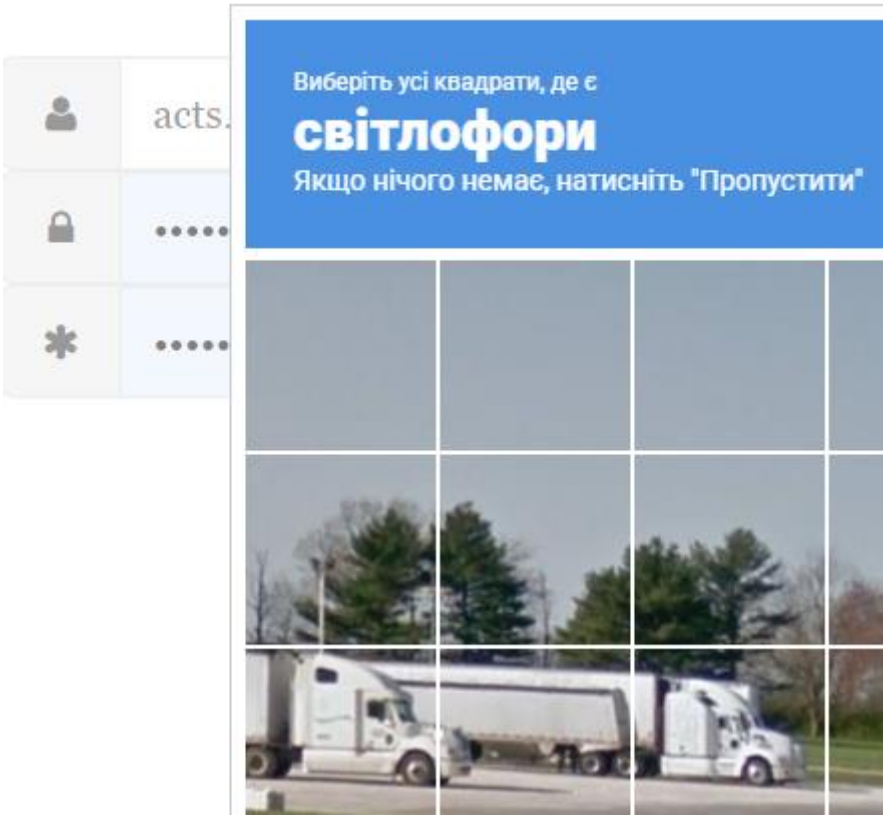
Для цього потрібно перейти на сторінку сайту https://block.io/users/sign_up і зареєструватись для створення облікового запису.



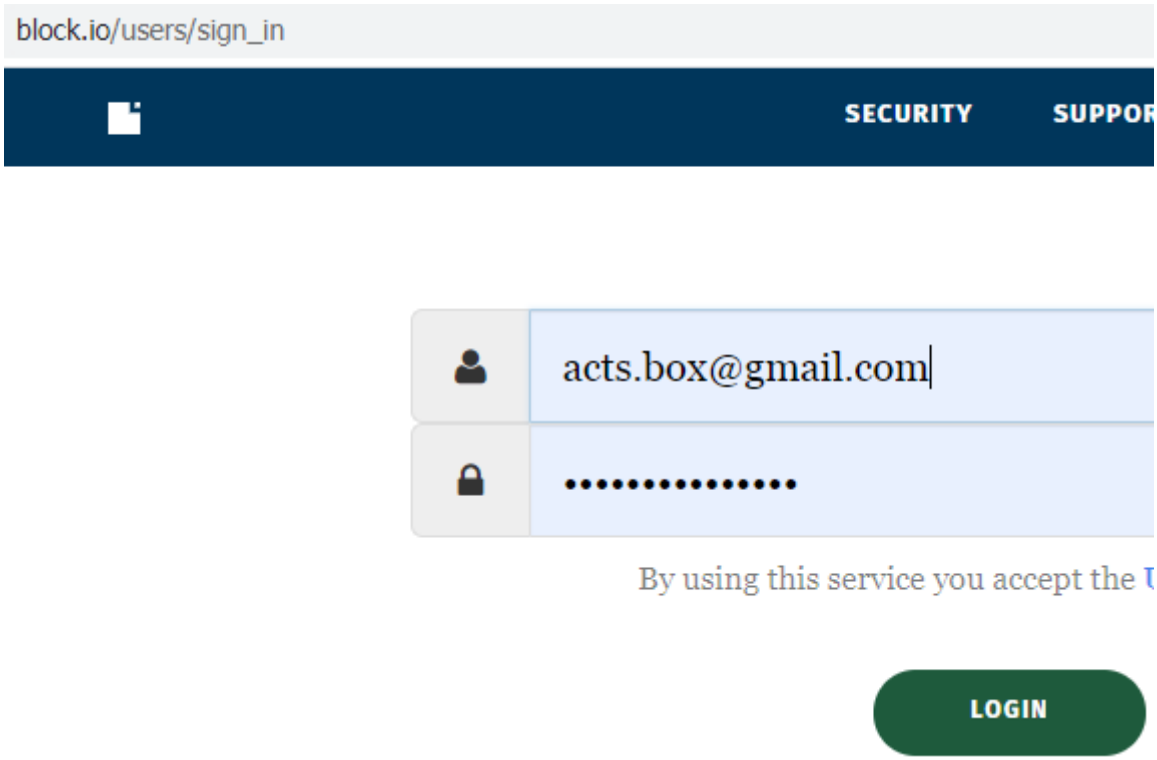
Приклад заповнення полів реєстрації:



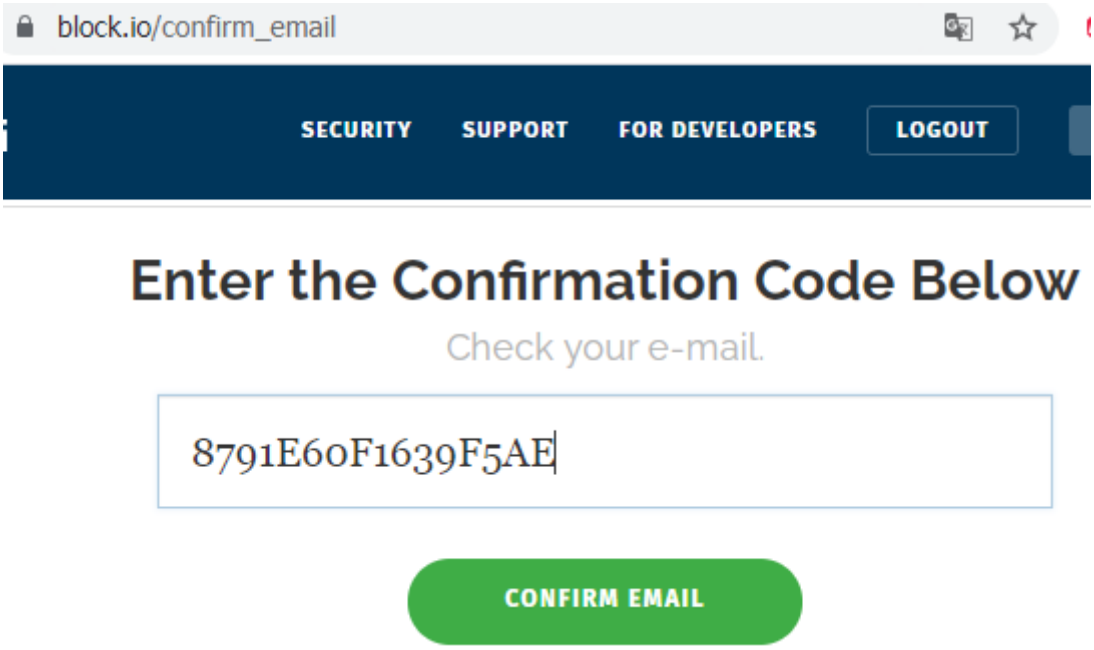
Далі ймовірно доведеться ввести капчу.



Далі логінимся:



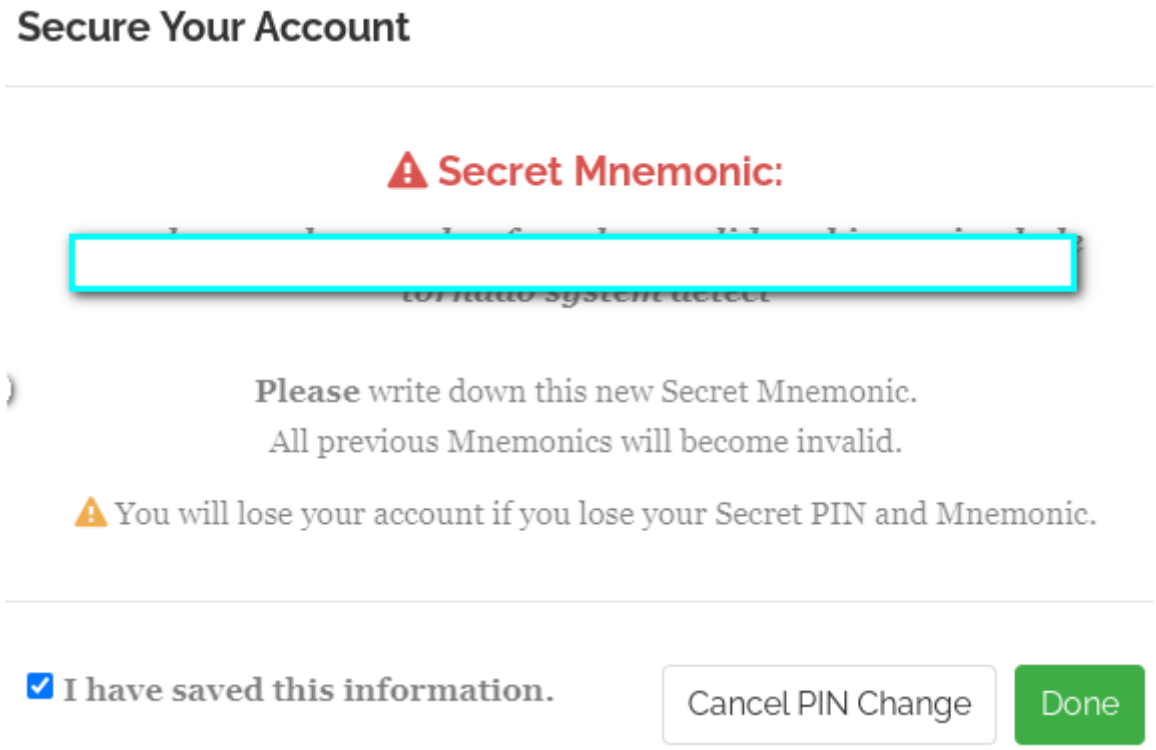
Вводимо код, що надійшов на імейл, потім ще один код підтвердження імейлу:



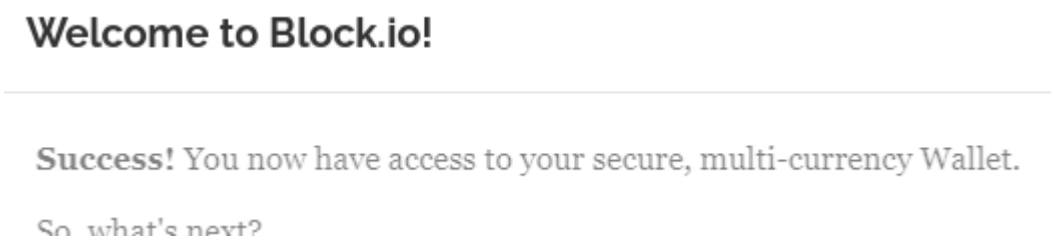
Пропускаємо вибір плану, натиснувши Skip:

Створюємо та запам’ятовуємо секретний ПІН-код:

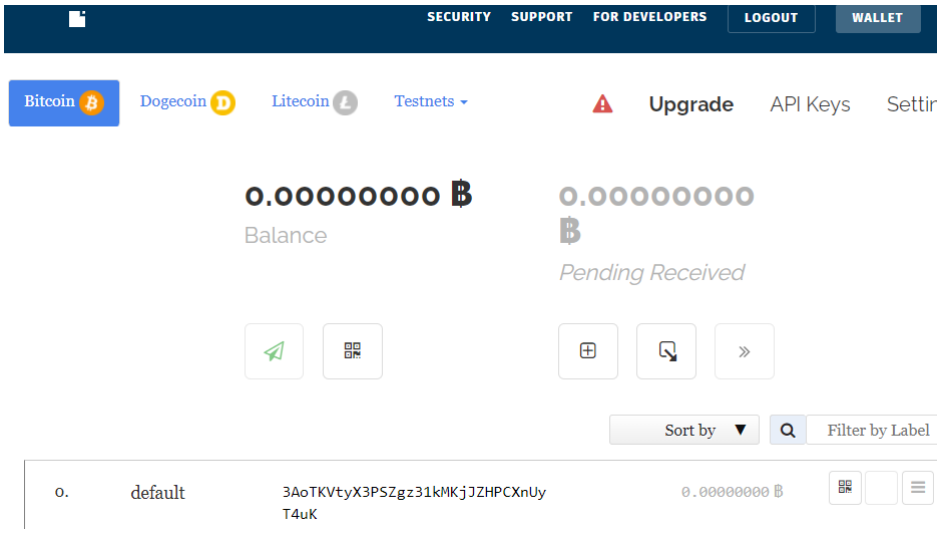
Далі, записуємо унікальну **мнемонічну фразу** – це пароль доступу до гаманця із тестовими криптомонетами:



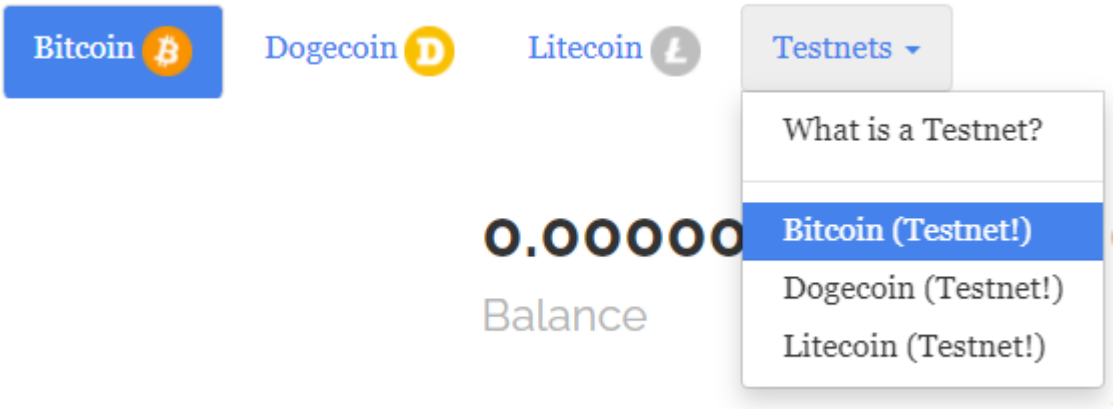
Якщо все гаразд, гаманець буде створено:



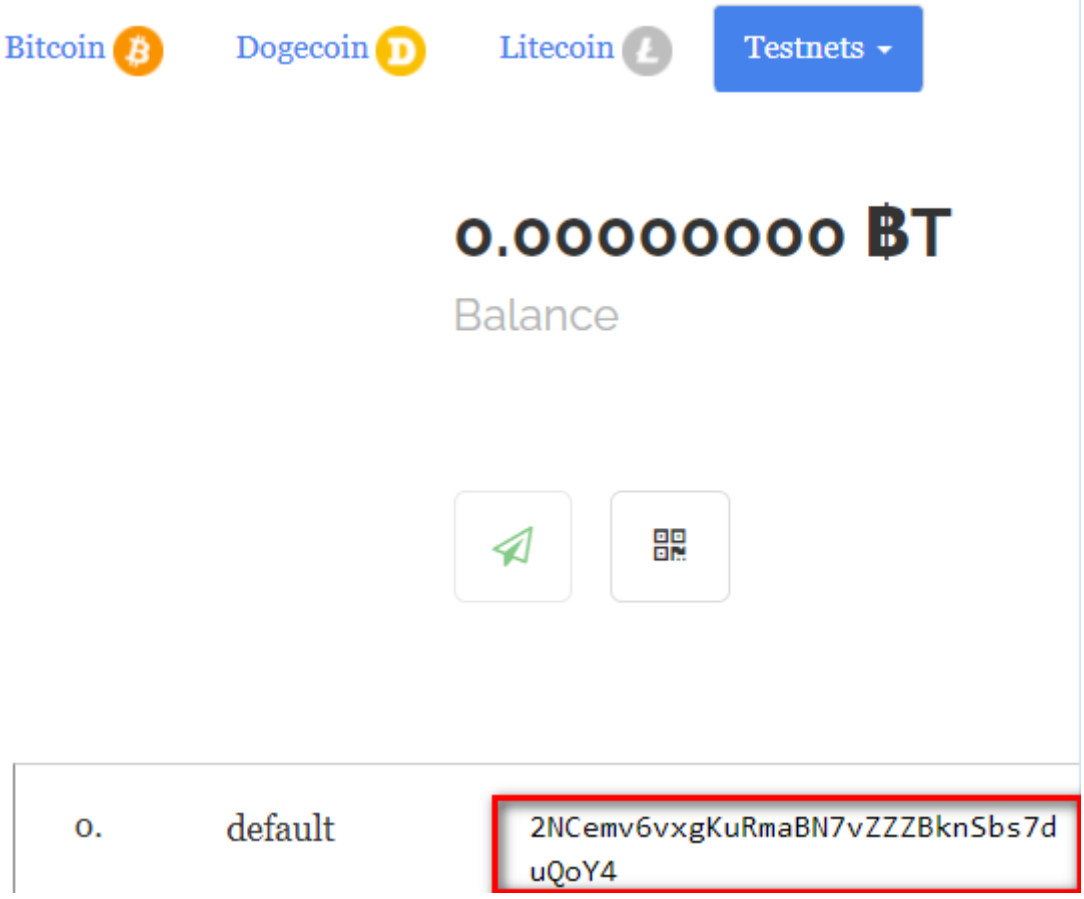
Далі потрапляємо до гаманця із трьома криптомонетами (за замовчування обрано Bitcoin та адресу bitcoin-гаманця). На балансі порожньо:



Обираємо тестову мережу Біткойн:



Згенерується нова адреса для гаманця в тестовій мережі:



Тепер відкриває один з наведених кранів (сайти, що роздають безкоштовно криптомонети):

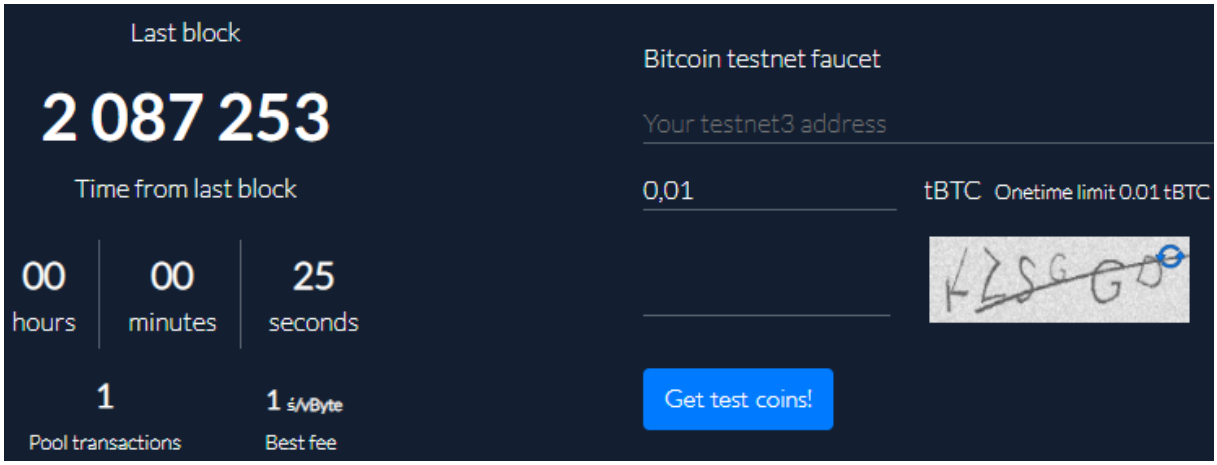
https://tbtc.bitaps.com/ (recommended)

https://coinfaucet.eu/en/btc-testnet/

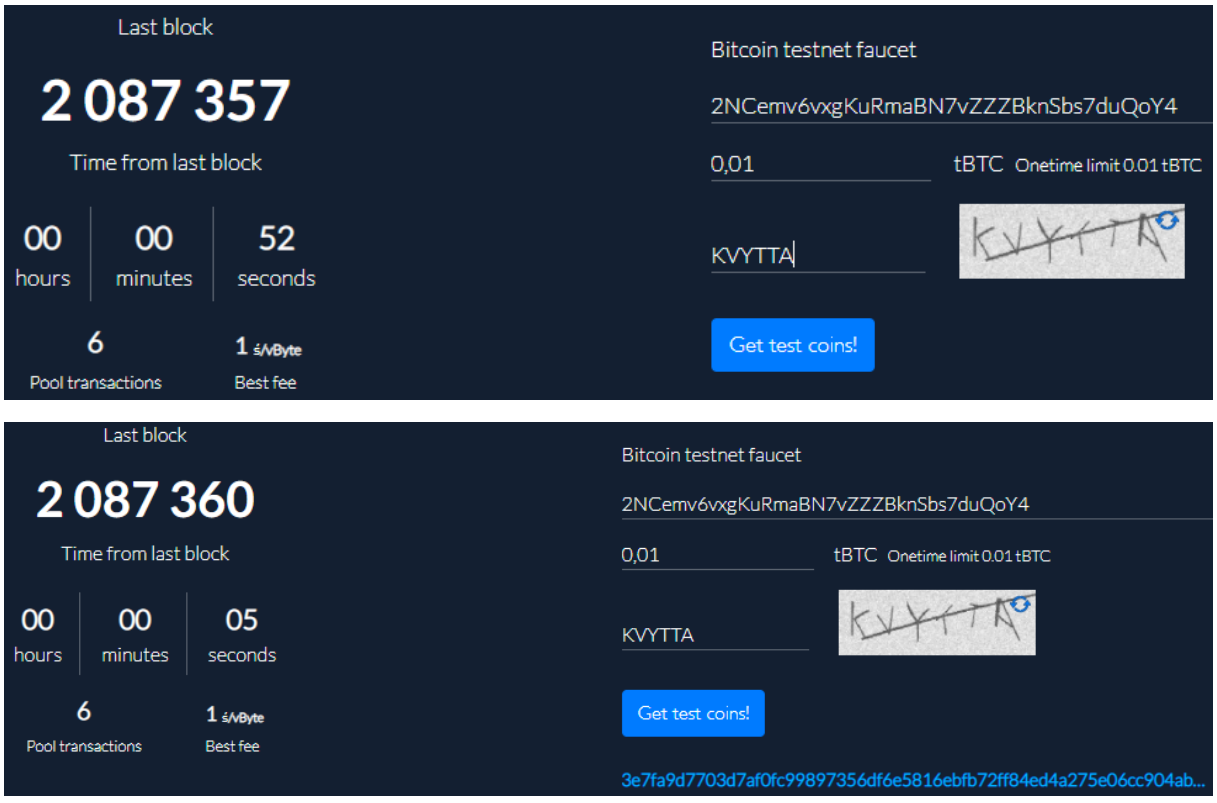
https://testnet-faucet.mempool.co/

https://bitcoinfaucet.uo1.net/

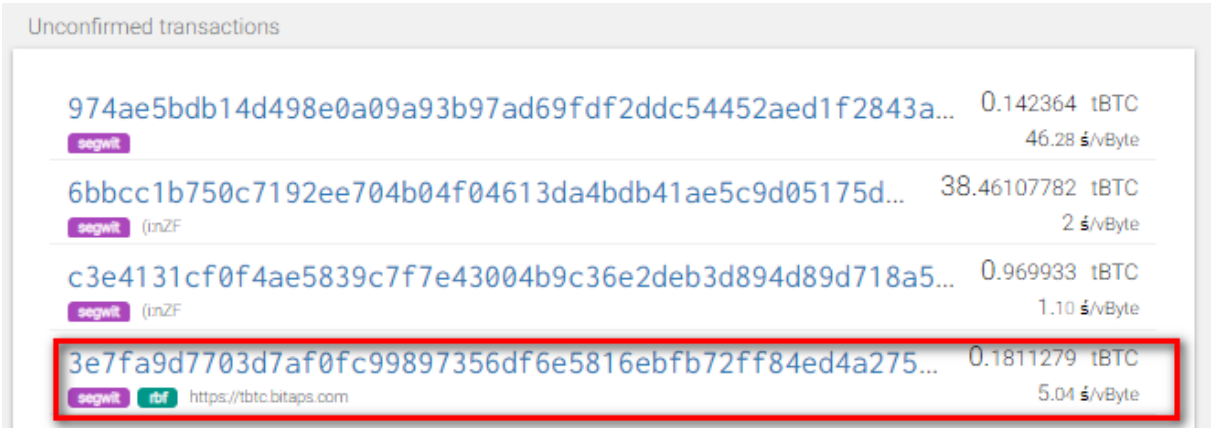
Виберемо перший сайт:



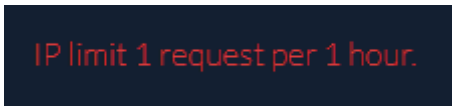
Далі в полі testnet3 address вводимо: адресу біткойн-гаманця, бажану кількість монет та капчу:



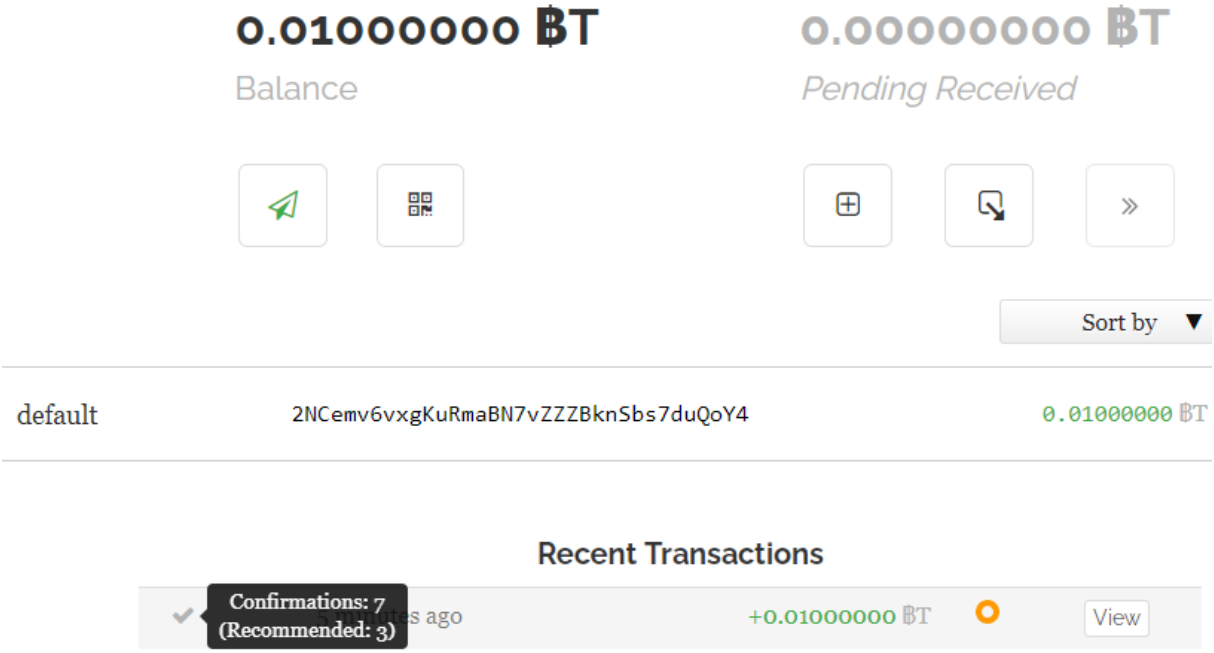
Можем спостерігати появу транзакції в черзі на підтвердження блоками:



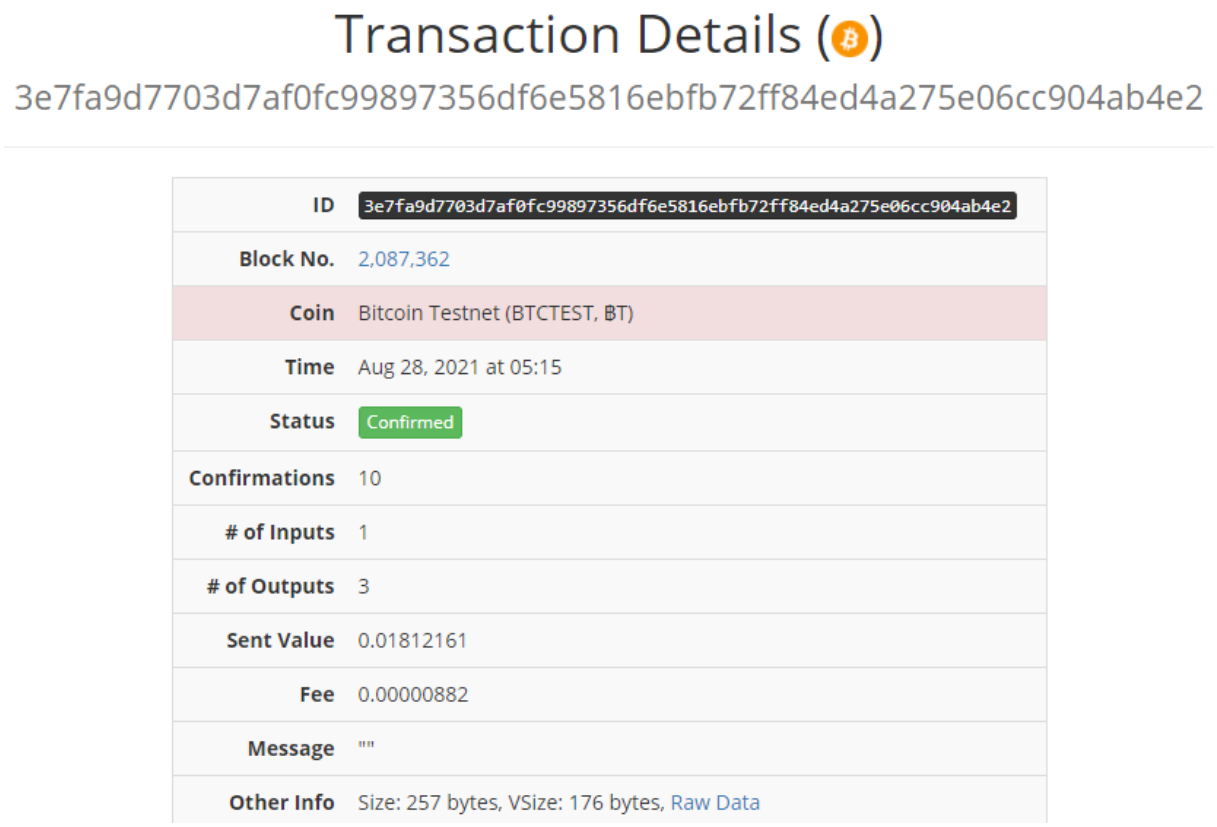
ВАЖЛИВО. Кран видає щогодини тестові монети.



Далі повертаємось до гаманця (оновлюємо сторінку) і очікуємо надходження тестових монет, бачимо що транзакція переводу спочатку не має підтверджень, але швидко (мережа зовсім не навантажена) підтверджується:



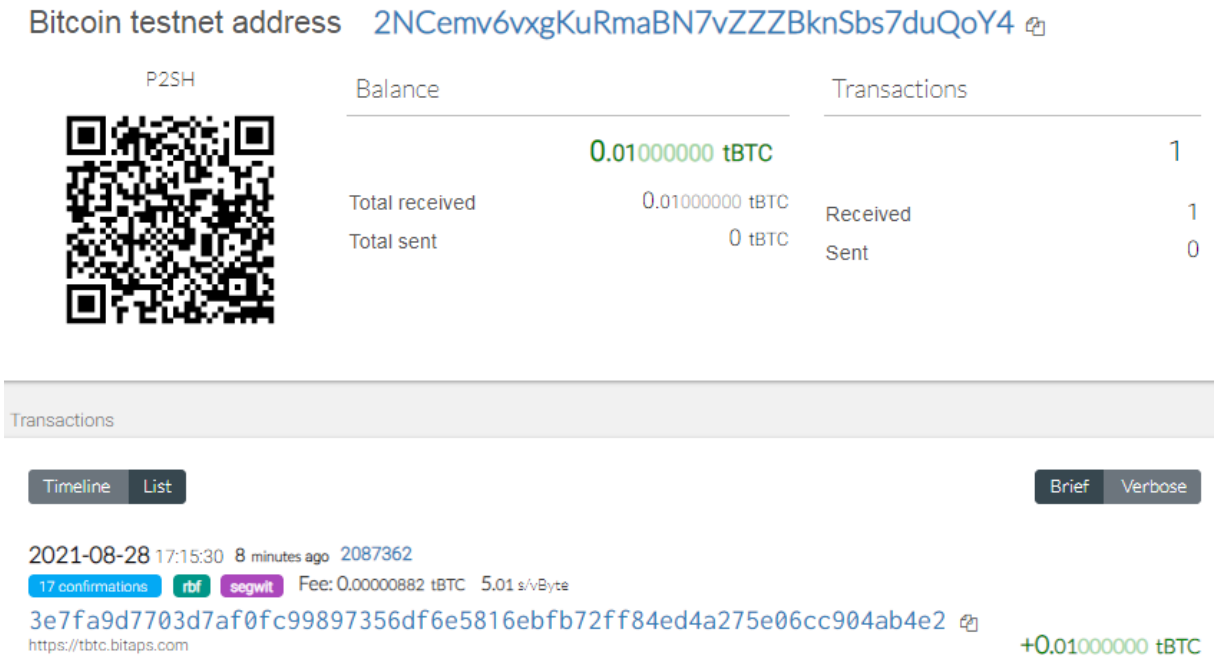
Натискаємо **View** та переходимо до експлореру sochain.com:



5.3 Перегляд транзакцій в експлорері блокчейну

Можна переглянути транзакцію у самому крані tbtc.bitaps.com:

І стан «балансу» гаманця в експлорері крана tbtc.bitaps.com:



5.4 Контрольні запитання

1) Що таке кран?
2) Що таке експлорер блокчейну?
3) Чим відрізняється тестнет від мейннет блокчейну.
4) Що таке підтвердження транзакцій.
5) Чому у гаманця немає балансу, але відображене число.

5.5 Контрольне завдання

1) Створити тестовий онлайн-гаманець Bitcoin. Отримати з крану необхідні кошти на гаманець.
2) Передати на тестову адресу гаманця кошти, в еквіваленті **0.[деньмісяцьрік]** тестових біткойнів (tBTC)**.**

Наприклад, якщо рік народження 28.08.2021, значить передаємо 0.00280821 tBTC**.**

Результати ЛР подати у вигляді скріншотів транзакцій в блокчейні.

6 ЛАБОРАТОРНА РОБОТА № 6

Тема: Смарт-контракти на блокчейні Ethereum

Мета: Опанувати теоретичну частину щодо смарт-контрактів та розгорнути приклад типового смарт-контракту.

Завдання: Запустити тестову локальну мережу Ethereum. Створити аккаунт у тестовій мережі. Створити контракт у тестовій мережі. Задеплоїти смарт контракт до мережі.

6.1 Теоретичні відомості

Ethereum - криптовалюта і платформа для створення децентралізованих онлайн-сервісів на базі блокчейна (децентралізованих додатків), що працюють на базі розумних контрактів. Реалізована як єдина децентралізована віртуальна машина. Концепт був запропонований засновником журналу Bitcoin Magazine Віталіком Бутеріним в кінці 2013 року, мережа була запущена 30 липня 2015 року.

В контексті ефіріума блокчейн діє як публічний реєстр, в якому фіксується все що відбувається в мережі в режимі реального часу. Деякі з основ технології блокчейн які дозволяють використовувати біткоіни забезпечують і роботу ефіріума. Структура блокчейна ефіріума схожа на те що ми бачимо у біткоіні. Так, копія кожної транзакції поширюється по всій мережі і кожен вузол (комп'ютер з програмним забезпеченням ефіріума) мережі зберігає копію цієї історії.

Цей розподілений цифровий реєстр може бути легко синхронізований через масштабну децентралізовану мережу. Це те що робить ефіріум доступним для будь-якої людини яка має доступ до інтернету незалежно від свого місця розташування.

У мережі ефіріума є блоки даних що складаються з транзакцій і смарт-контрактів. Ці блоки пов'язані один з одним і являють собою повний запис історії ефіріума починаючи з першого блоку. Блоки створюються деякими

користувачами і поширюються серед інших користувачів які перевіряють їх і підтверджують що вони «правильні».

Обмінні одиниці Ethereum називаються ефіром (англ. Ether). Для позначення використовується скорочення ETH і символ у вигляді грецької букви Кси Ξ. Дробові частини мають свої назви: 1/1000 - finney, 1/106 - szabo, 1/1018 - wei.

Ethereum - це криптовалюта, але її «розумні контракти» можуть використовуватися в різних фінансових областях. Про свій інтерес до платформі заявили різні організації, включаючи Microsoft, IBM, JPMorgan Chase. Bloomberg Businessweek стверджує, що розподілене програмне забезпечення Ethereum може бути використано усіма, кому потрібен захист від несанкціонованого втручання. «Ви можете спокійно робити бізнес з кимось, кого ви не знаєте, тому що умови прописані в розумному контракті, вбудованому в блокчейн».

6.2 Встановлення необхідних програмних засобів

У даному прикладі використовувався менеджер залежностей Chocolatey (доступний для Windows систем). Встановити потрібні залежності можливо і іншими шляхами. Щоб завантажити Chocolatey слідуйте інструкціям за посиланням [1].

Для встановлення потрібних залежностей скористаємось командою “ choco install <ім’я залежності>”. У нашому випадку потрібно встановити залежності node та ethereum.

- ➔ choco install nodejs.install –y
- ➔ choco install VisualStudioCode –y
- ➔ choco install ethereum
- ➔ choco install truffle

Перед тим як встановити ethereum потрібно створити папку для встановлення. Цього можна досягти командою “choco tap <шлях створення>”.

➔ choco tap ethereum/ethereum

Програмне забезпечення Ganache можна завантажити за посиланням [2], фреймворк Truffle [3].

Необхідне програмне забезпечення

1. **Chocolatey version 0.10.15**

Chocolatey створений для того, щоб зробити управління програмами більш зручним і більш схожим на те, як це відбувається в Linux. В рамках цього проєкту створено репозиторій програм і спеціальний клієнт для Windows, який здатний завантажувати, встановлювати та оновлювати практично в автоматичному режимі програми з цього каталогу. Клієнт Chocolatey вдає із себе утиліту, що працює в командному рядку.

Він не є обов’язковим у даній роботі, ви можете встановити усі додатки вручну з офіційних сайтів, але Chocolatey робить це швидше та без зайвих рухів.

2. **Ethereum (Geth) version 1.0.3-stable**

Ethereum - децентралізована платформа, на якій працюють смарт-контракти: додатки, виконувані строго запрограмованим чином, без можливості даунтайм, цензури, фрода або втручання третіх осіб. Як раз для створення приватного блокчейну Ethereum і потрібен Geth.

Інструкція включає в себе наступне:

Створення приватного блокчейна Ethereum за допомогою geth.

Створення гаманця MetaMask для роботи з приватним блокчейном.

Переказ коштів між кількома акаунтами.

Створення, розгортання і виклик смарт-контракту в приватному блокчейне за допомогою remix.

Створення оглядача блоків Ethereum поверх приватного блокчейна.

3. **Node 16.2.0**

У даній роботі Node.js потрібен для написання скриптів для блокчейну.

4. **Truffle v5.3.9**

«Truffle» це середовище розробки світового класу, що надає платформу тестування і портфель активів для «Blockchain» використовуючи «Máquina Virtual Ethereum (Ethereum Virtual Machine - EVM)», щоб полегшити життя розробникам. Серед його найбільш видатних особливостей:

- Вбудована компіляція смарт-контрактів, прив'язка, розгортання і двоичное управління.
- Автоматизоване тестування контрактів для швидкої розробки.
- Розширювана і програмована середу розгортання і міграції.
- Управління мережею для розгортання в будь-якій кількості публічних і приватних мереж.
- Управління пакетами з допомогою EthPM і NPM, використовуючи стандарт ERC190.
- Інтерактивна консоль для прямого спілкування з замовником.
- Конфігурується конструкція труби з підтримкою тісної інтеграції.
- Зовнішній виконавець сценаріїв, який запускає сценарії в середовищі Truffle.

5. Visual Studio Code

Visual Studio Code - це редактор вихідного коду, Розроблений Microsoft для Windows, Linux і macOS. Позиціонується як «легкий» редактор коду для кроссплатформенной розробки веб і хмарних додатків. Включає в себе відладчик, інструменти для роботи з Git, підсвічування синтаксису, IntelliSense і засоби для рефакторинга. Має широкі можливості для кастомізації: призначені для користувача теми, поєднання клавіш і файли конфігурації.

Ви можете обрати будь-який текстовий редактор, не обов'язково Visual Studio Code.

6. Ganache

«Ganache» є «Blockchain» особистий з «Ethereum» для створення і тестування розробки, наприклад «Dapps» і «Smart Contracts». Він доступний як настільний додаток, так і інструмент командного рядка (раніше відомий як TestRPC). Ganache доступний для Windows, Mac і Linux.

6.3 Запуск приватного блокчейну

Запуск приватного блокчейну виповнюється у терміналі наступною командою

➔ geth --dev --ipcpath ~/Library/Ethereum/geth.ipc

Для комфортної роботи geth надає наступні параметри при виконанні команд:

--vmdebug	(Virtual Machine debug output)
--maxpeers <n>	де n - це число можливих підключень, у разі встановлення 0, підключення неможливе
--gasprice <n>	де n - це вартість контракту

--port 0	підключення до певного порту, при значенні 0 відбувається підключення до будь якого порту
--shh	включення протоколу Whisper
--datadir	створює тимчасову папку на жорсткому диску
--ipcpath	встановлення стандартного шляху IPC, що спрощує роботу з іншими командами geth шляхом зменшення передаваємих параметрів

Після вводу даної команди у вікно терміналу буде виведена інформація стосовно блокчейну (рис. 9)

- максимальна кількість підключень (peers)
- інформація про протоколи
- інформація про створені / знищені елементи блокчейну
- розмір використаної пам’яті

Підключення до мережі виконується командою (рис. 10)

➔ geth --dev attach

Створення нового аккаунту у мережі виконується командою (рис. 11)

➔ personal.newAccount('<password>')

Поповнення аккаунту

➔ miner.start()

Завершення поповнення

➔ miner.stop()

Останні три команди (створення, поповнення, завершення поповнення) виконуються у JavaScript консолі, яка буде відкрита автоматично після успішного підключення до блокчейну (рис. 13).

6.4 Фреймворки Ganache та Truffle

Truffle Suite - це популярний фреймворк для розробки децентралізованих фулстек застосунків для мережі Ethereum. Перед початком використання потрібно впевнитись, що даний фреймворк встановлено глобально. Зручніше всього встановлювати його за допомогою менеджеру npm. Виконується це командою у терміналі:

➔ npm install -g truffle

Ganache - працює як локальний блокчейн для деплою та тестування функціоналу контрактів перед тим як задеплоїти їх у публічну мережу. Для встановлення Ganache достатньо слідувати інструкціям з офіційної сторінки.

Перед початком роботи з Truffle зручніше всього створити папку, у якій у подальшому буде створюватись проєкт.

Ініціалізація проєкту за допомогою Truffle (рис. 13) відбувається командою

➔ truffle init

Після успішного виконання команди у вікно терміналу буде виведена інформація щодо шляху у який було скопійовано файли проєкту. Проєкт містить папки:

- migrations
- contracts
- test

Та файл конфігурації truffle-config.js (рис. 14)

Підключення до Ganache. Підключення відбувається за допомогою функції Quickstart у вікні програми, що запустить локальний блокчейн (рис. 15). Перші 10 адресів є публічними, які Ganache створив для взаємодії з блокчейном та автоматично поповнив дані аккаунти на 100 ETH. На темній частині у верхній частині вікна можна побачити інформацію про блокчейн: Current Block, Gas Price, Gas Limit, та інше. Для подальшої роботи знадобиться Network ID та RPC Server. Network ID вводиться у файл конфігурації truffle-config.js (рис. 16). У даному файлі вказується де саме truffle повинен шукати локальний блокчейн. Усередині network вказується ім'я мережі, її host, port та networkID.

Для перевірки можливості підключення truffle до блокчейну використовується команда у терміналі:

➔ truffle test

При успішному виконанні вивід буде схожий як на рис. 17. Після цього truffle готовий до взаємодії з локальним блокчейн.

6.5 Файли смарт-контрактів

Смарт-контракти створюються та зберігаються у папці проєкту truffle contracts/. Приклад вказаний на рис. 18.

pragma solidity визначає версію Solidity, що використовується для компіляції контракту. Solidity — статично типізована JavaScript-подібна мова програмування, створена для розробки розумних контрактів, які працюють на віртуальній машині Ethereum (EVM). Програми на мові Solidity транслюються в байткод EVM. Solidity дозволяє розробникам створювати самодостатні програми, що містять бізнес-логіку, результуючу в транзакційні записи блокчейну. У прикладі використовується версія 0.5.0, потрібно перевіряти яка саме версія використовується для коректного компіліювання контракту. Для перевірки у терміналі вводиться команда:

➔ truffle version

Після ключового слова contact вводиться ім’я контракту. У контракту можуть бути змінні, функції та конструктори.

Для деплою контракту у блокчейн він повинен бути перетворений у машинний код. Для цього у терміналі вводиться команда:

➔ truffle compile

При успішному виконанні даної команди вивід буде схожий на приклад на рис. 15. та у дереві проєкту з’явиться папка build/.

Міграції - це процес деплою скомпільованих смарт контрактів до блокчейну. Файл міграції приведено на рис. 20. У даному файлі зберігаються інструкції щодо того як Truffle повинен задеплоїти смарт контракт. Міграція виконується наступною командою у терміналі:

➔ truffle migrate

У разі успішного виконання вивід буде схожий на рис. 21. У виводі нижче після імені кожного контракту буде вказана наступна інформація:

- hash транзакції
- блоки
- адреса контракту
- аккаунт
- баланс
- використаний gas
- ціна gas
- відправлена валюта
- повна вартість

У графі summary буде вказано число задеплоєних контрактів та повна вартість.

Після міграцій у вікні Ganache можна буде побачити зміни у аккаунтах. Там буде надана інформація стосовно суми використаної валюти та кількості транзакцій. Баланс ETH зменшився, бо був використаний gas для деплою у блокчейн.

6.6 Тестування смарт-контрактів

Для того аби переконатися, що смарт контракти працюють вірно можуть бути написані тести. У прикладі вказані Solidity тести. Тести створюються та зберігаються у папці tests.

Для тестування вводиться команда у терміналі:

➔ truffle test

У разі успішного виконання у вікно терміналу буде виведена інформація про проведені тест кейси та час їх виконання.

6.7 Деплой смарт-контрактів

Для деплою смарт контрактів до мережі, потрібно мати Ether даної мережі. Скористаємось тестовими монетами.

Для цього можна використовувати Kovan testnet [4]. У вікні Ganache потрібно скопіювати публічний ключ першого аккаунту та відправити у чат Kovan. Незабаром з’явиться інформація про відправку Ether на ваш адрес.

Надалі потрібно аби Truffle мав змогу підключитися до мережі Kovan. Сервіс Infura [5] спрощує дану роботу. На сервісі потрібно зареєструватися та створити новий проєкт. У секції Keys ввести ключі, та у випадаючому списку ‘ENDPOINT’ потрібно обрати Kovan. У проєкті потрібно зробити файл package.json та у тілі dependencies цього файлу вказати наступне:

```
"dependencies": {
    "truffle-hdwallet-provider": "1.0.4",
    "truffle-hdwallet-provider-privkey": "1.0.3",
    "web3": "1.0.0-beta.46"
  }
```

У терміналі потрібно ввести команду:

```
➔ npm install
```

Після завершення встановлення залежностей у файлі конфігурацій нижче списку мереж потрібно вказати наступне:

```
kovan: {
    provider: function() {
      return new HDWalletProvider(
        [privateKey],
        //url to ethereum node
        endpointUrl
      )
    },
    gas: 5000000,
    gasPrice: 25000000000,
    network_id: 42
  }
```

privateKey	масив ключів. Взяти у Ganache
endpointURL	посилання до ноду Ethereum. Ця інформація береться з сервісу Infura (ENDPOINT).

Після цього у терміналі потрібно виконати команду:

➔ truffle migrate --network kovan

При успішному виконанні даної команди буде вивід як на рис. 20. У вікні терміналу можна побачити наступну інформацію:

- hash транзакції
- кількість блоків
- адрес контракту
- баланс
- використаний gas
- ціна gas
- відправлена кількість валюти
- повна вартість

Використовуючи адрес контракту можна перейти за посиланням [6] та знайти даний контракт.

ДАЛІ створимо розподілений додаток (DAPP) на платформі Ethereum. Зокрема, створення контрактів, а також тестування їх на локальному блокчейні.

6.8 Приклад виконання смарт-контрактів

Перш ніж починати виконувати роботу, необхідно інсталювати потрібні нам програми, а саме:

- Node Js;
- Visual Studio Code (або інший текстовий редактор).

Звернемо увагу на менеджер управління пакетами Windows Chocolatey, він значно пришвидшує встановлення потрібних вам програм.

Встановити chocolately можна за інструкцією з офіційного сайту [1] в cmd у режимі адміністратора, або PowerShell у режимі адміністратора (всі

наступні дії виконуються у цьому режимі), розглянемо обидва методи його встановлення:

1. CMD

Для того щоб встановити chocolatey на вашу машину, введіть у командний рядок наступну команду:

```
@powershell -NoProfile -ExecutionPolicy Bypass -Command
"[System.Net.WebRequest]::DefaultWebProxy.Credentials =
[System.Net.CredentialCache]::DefaultCredentials; iex ((New-Object
System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))
" && SET PATH="%PATH%;%ALLUSERSPROFILE%\chocolatey\bin"
```

Важливо! При копіюванні будь-якого коду з файлів форматом .docx чи .pdf можлива деструктуризація скрипту, котрий вставлятиметься у cmd та PowerShell, шляхом додавання зайвих відступів через міжрядковий інтервал. Будьте, будь-ласка, уважні.

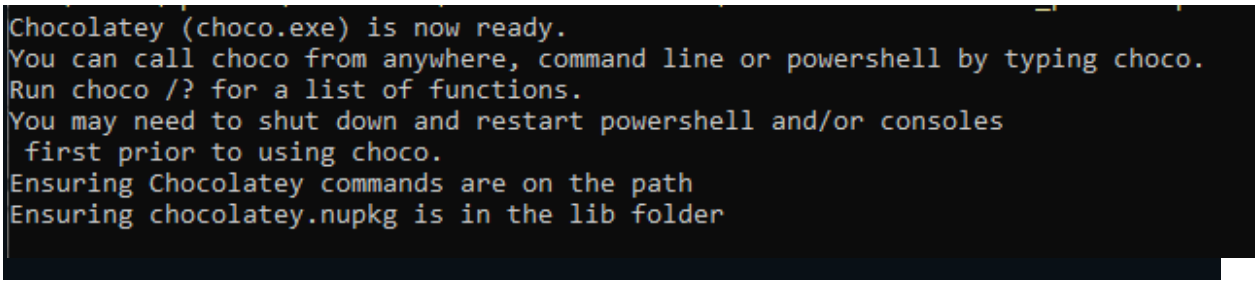


Рис. 1. Результат встановлення chocolately

2. PowerShell

Встановлення chocolatey в PowerShell відбувається так само як і в першому варіанті, проте команда відрізняється:

```
[System.Net.WebRequest]::DefaultWebProxy.Credentials =
[System.Net.CredentialCache]::DefaultCredentials; iex ((New-Object
System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))
```

Після встановлення потрібно закрити PowerShell та відкрити його знов для виконання наступних кроків.

ВАЖЛИВО! Якщо при виконанні цих кроків у вас виникли проблеми з встановленням, ви можете без проблем інсталювати усі потрібні програми з офіційних сайтів за посиланнями [2], [3], [7], [8], [9] та продовжити виконання роботи з 14 сторінки. Вони встановлюються наступним чином:

1. Visual Studio Code

   Перейдемо за посиланням (https://code.visualstudio.com/download) та оберемо вказаний на рис. 2 варіант встановлення програми [7].

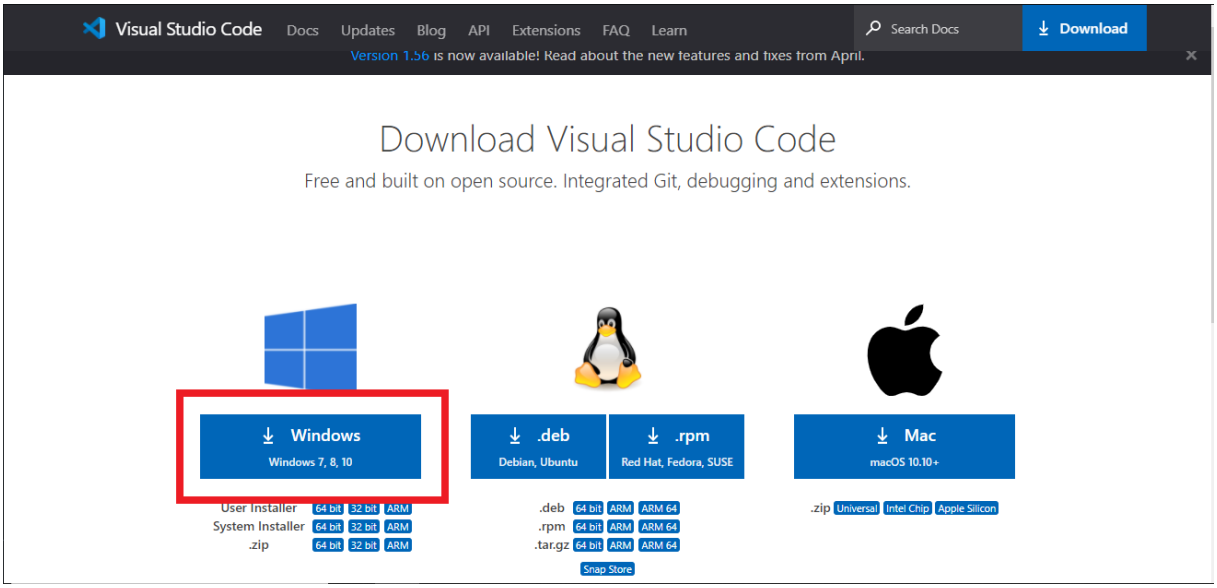


Рис. 2. Встановлення Visual Studio Code

Після завантаження файлу відкрийте його, та натискайте кнопку “Next” до її встановлення.

2. Geth

Для встановлення Geth перейдемо за посиланням (https://geth.ethereum.org/downloads/) та оберемо вказаний на рис. 3 варіант встановлення програми [8].

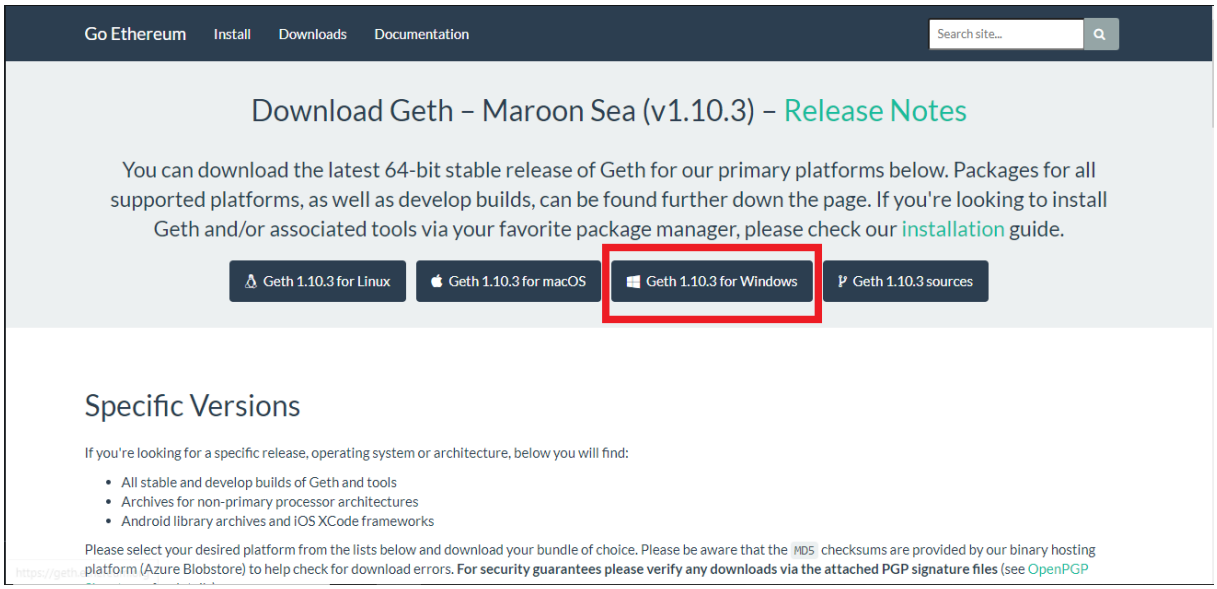


Рис. 3. Встановлення Geth

Після завантаження файлу відкрийте його, та натискайте кнопку “Next” до її встановлення.

3. Ganache

Для встановлення Ganache скористаємось посиланням (https://www.trufflesuite.com/ganache) та оберемо вказаний на рис. 4 варіант встановлення програми [2].

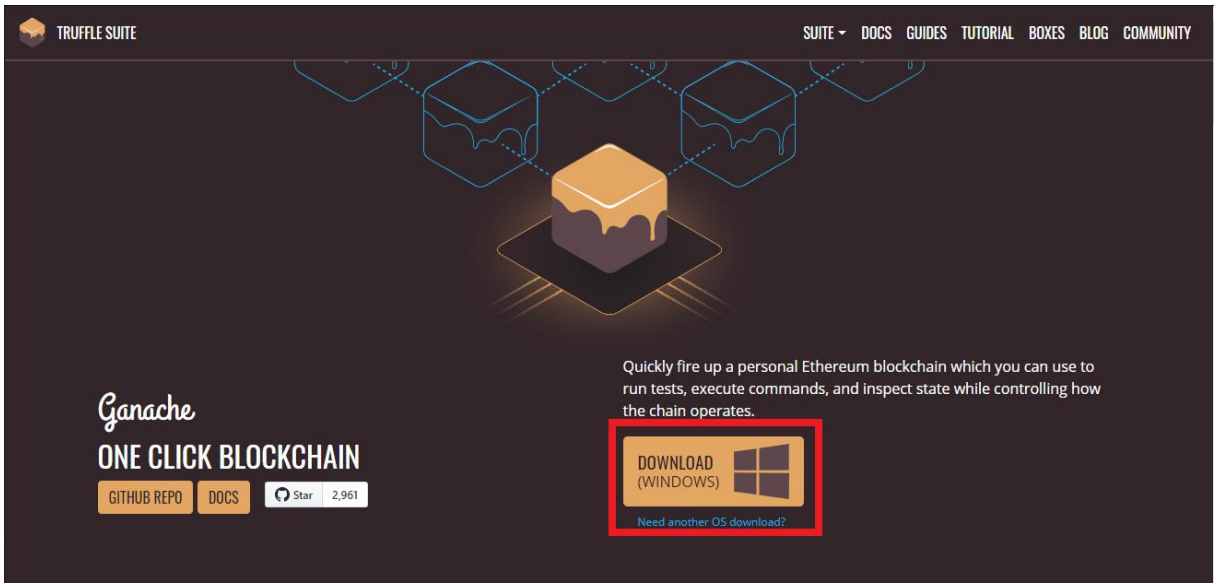


Рис. 4. Встановлення Ganache

Після завантаження файлу відкрийте його, та натискайте кнопку “Next” до її встановлення.

За допомогою chocolately продовжуємо встановлення потрібних програм. Власне, ввівши у консоль наступну команду, встановимо Node.js:

choco install nodejs.install –y

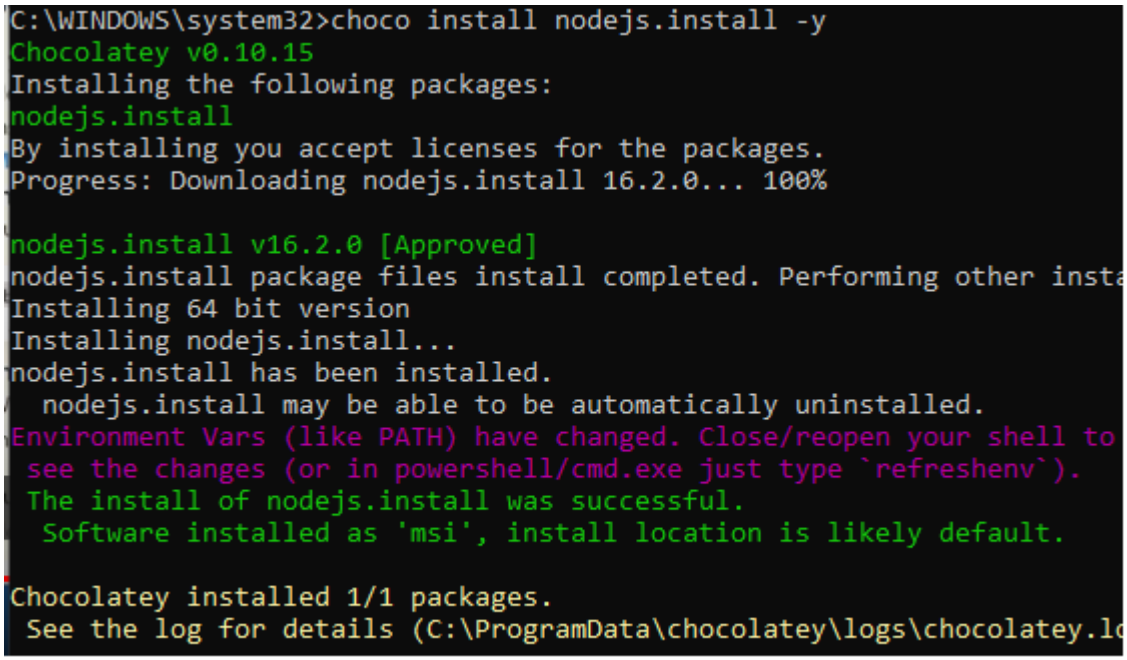


Рис. 5. Встановлення Nodejs

Таким самим чином встановимо і наступні програмні засоби.

CMD: Choco install VisualStudioCode –y

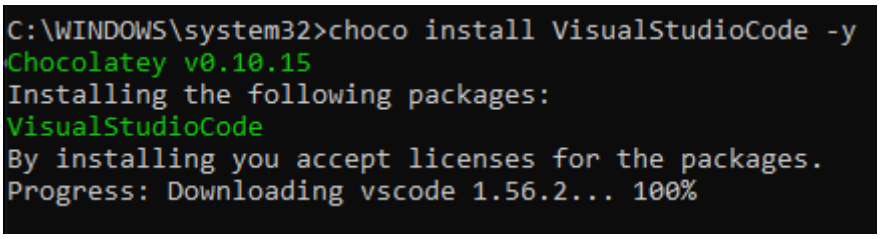


Рис. 6. Встановлення VisualStudioCode

Ganache та Geth встановлюємо з офіційних сайтів [2], [8].

Підключення до блокчейну

Встановлення Ethereum наступними командами в командному рядку.

CMD: geth account new --datadir %APPDATA%\Ethereum

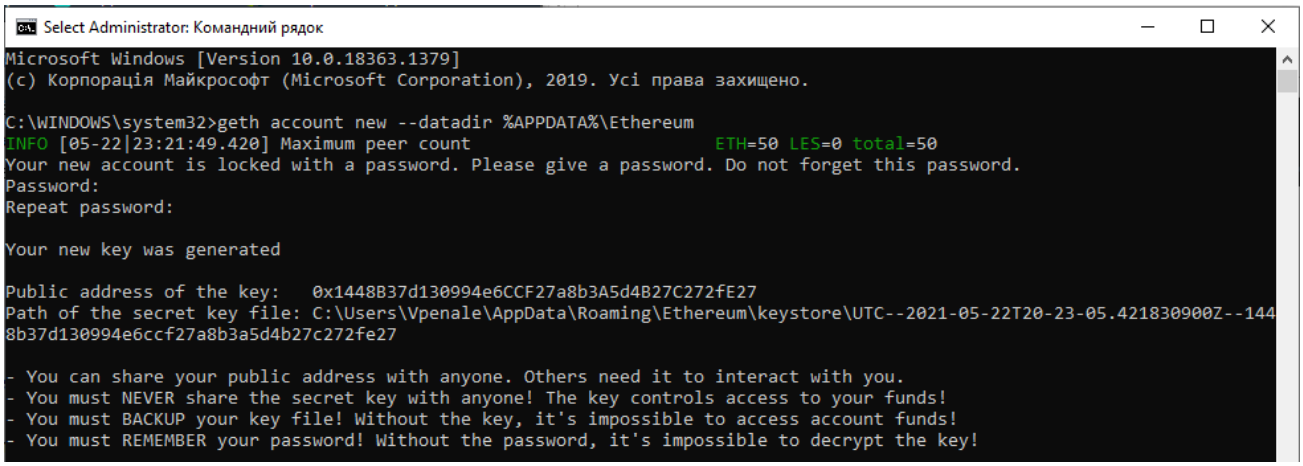


Рис. 7. Створення папки Ethereum

Перевірка версії geth виповнюється наступною командою: geth version

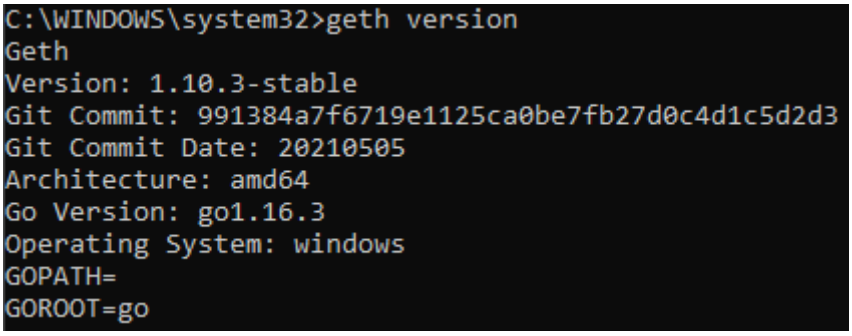


Рис. 8. Перевірка версії geth

Запуск приватного блокчейну виконується наступною командою у CMD:

geth --dev --ipcpath %APPDATA%\Ethereum/geth.ipc

Результат виконання цієї команди зображено на Рис. 9.

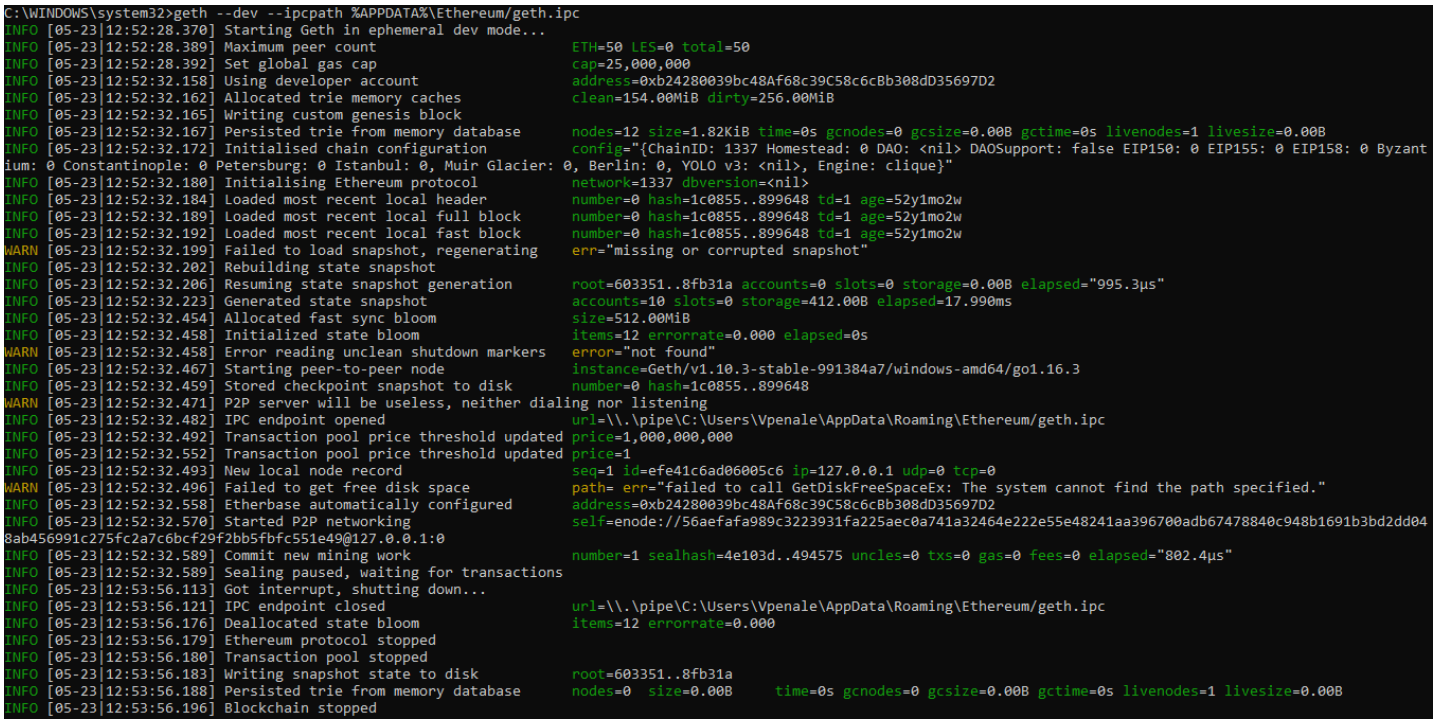


Рис. 9. Вивід команди geth ‘--dev --ipcpath <шлях>’

Задля створення акаунту у тестовій мережі потрібно відкрити нове вікно терміналу та ввести команду “geth --datadir datadir2 --port 30304 --nodiscover --ipcpath geth-data2.ipc --networkid 1234 console”.

Результатом цієї команди є відкриття JavaScript консолі. Тепер можна у новому командному рядку підключитись до створеного екземпляру наступною командою:

geth attach ipc:\\.\pipe\geth-data2.ipc

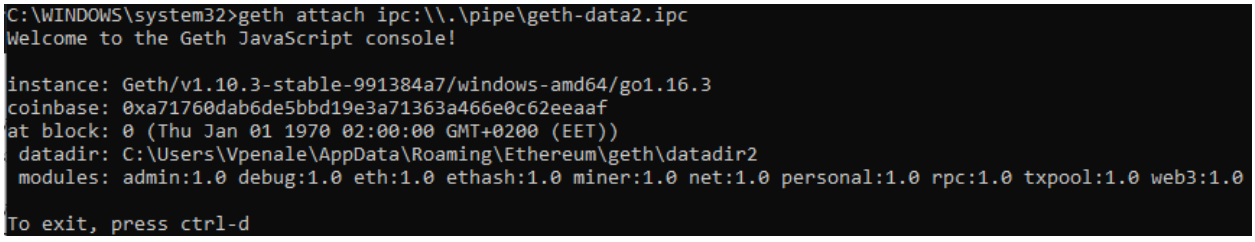


Рис. 10. Підключення до блокчейну

Створення акаунту відбувається наступним чином:

```
> personal.newAccount('qwerty')
"0x62352f8ae3c1598982f994fc8fe25310e812ffe5"
```

Рис. 11. Створення аккаунту через cmd

У дужках вказується пароль нового аккаунту. Поповнення аккаунту виконується наступною командою “miner.start()”. Після виконання цих дій вікна cmd можна закрити та продовжити виконання наступних кроків.

Створення смарт-контракту та тестування блокчейну

У процесі створення контракту буде використовуватись фреймворк Truffle. Для роботи з Ethereum цього разу буде використовуватись Ganache (https://www.trufflesuite.com/ganache).

Для подальшого розгортання контракту вам необхідно встановити на персональну машину **truffle** та **node.js** з декількома пакетами, які встановлюються в командному рядку наступними командами: “**npm install -g truffle**” та “**npm install -g @truffle/hdwallet-provider**”.

Truffle Suite - популярний фреймворк для розробки повнофункціональних децентралізованих додатків в мережі Ethereum.

Після цього необхідно створити папку в файловій системі у якій буде зберігатися ваш проєкт:

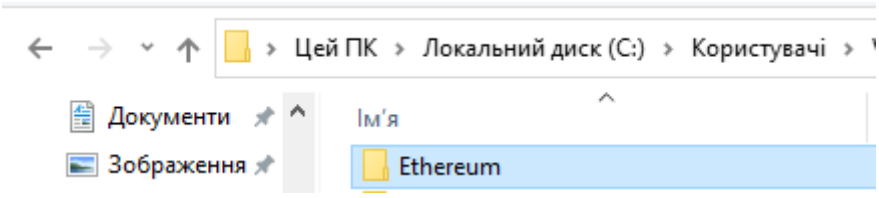


Рис. 12. Створена папка

Та ініціалізувати Truffle в ній командою “**truffle init**”.

Для цього перейдемо до цієї папки командою cd <скопійована адреса до створеної папки>.

Введемо вказану команду для ініціалізації truffle у створеній папці

```
C:\Windows\System32>cd C:\Users\Vpenale\AppData\Roaming\Ethereum\truffle-smart-contract

C:\Users\Vpenale\AppData\Roaming\Ethereum\truffle-smart-contract>truffle init

Starting init...
================

> Copying project files to C:\Users\Vpenale\AppData\Roaming\Ethereum\truffle-smart-contract

Init successful, sweet!
```

Рис. 13. Ініціалізація truffle проєкту

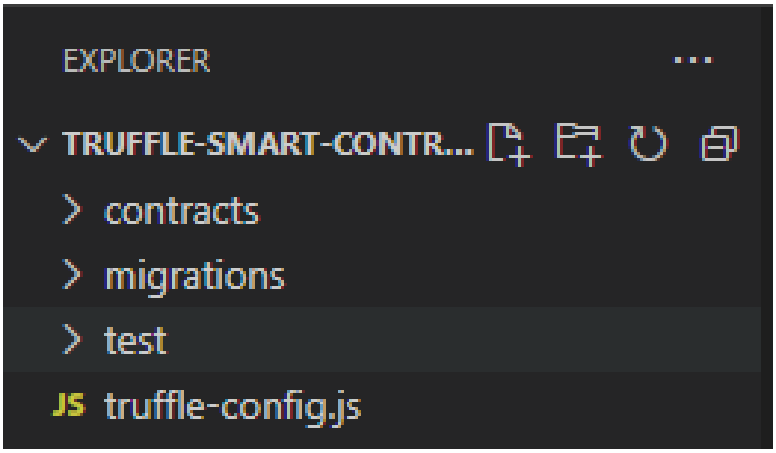


Рис.14. Створений проєкт truffle

У результаті маємо проєкт.

contract/ місце де зберігається код Solidity Smart Contract с. Truffle знає, що тут потрібно шукати .sol файл аби скомпілювати та задеплоїти до блокчейну.

migrations/ місце зберігання логіки міграцій

test/ тести для Smart Contracts

truffle-config.js файл зберігає інформацію про наші мережі, локації файлів та інші налаштування для Truffle

Запустмо Ganache **у режимі адміністратора** та запустимо його за допомогою Quickstart, що у результаті відкриє наступне вікно

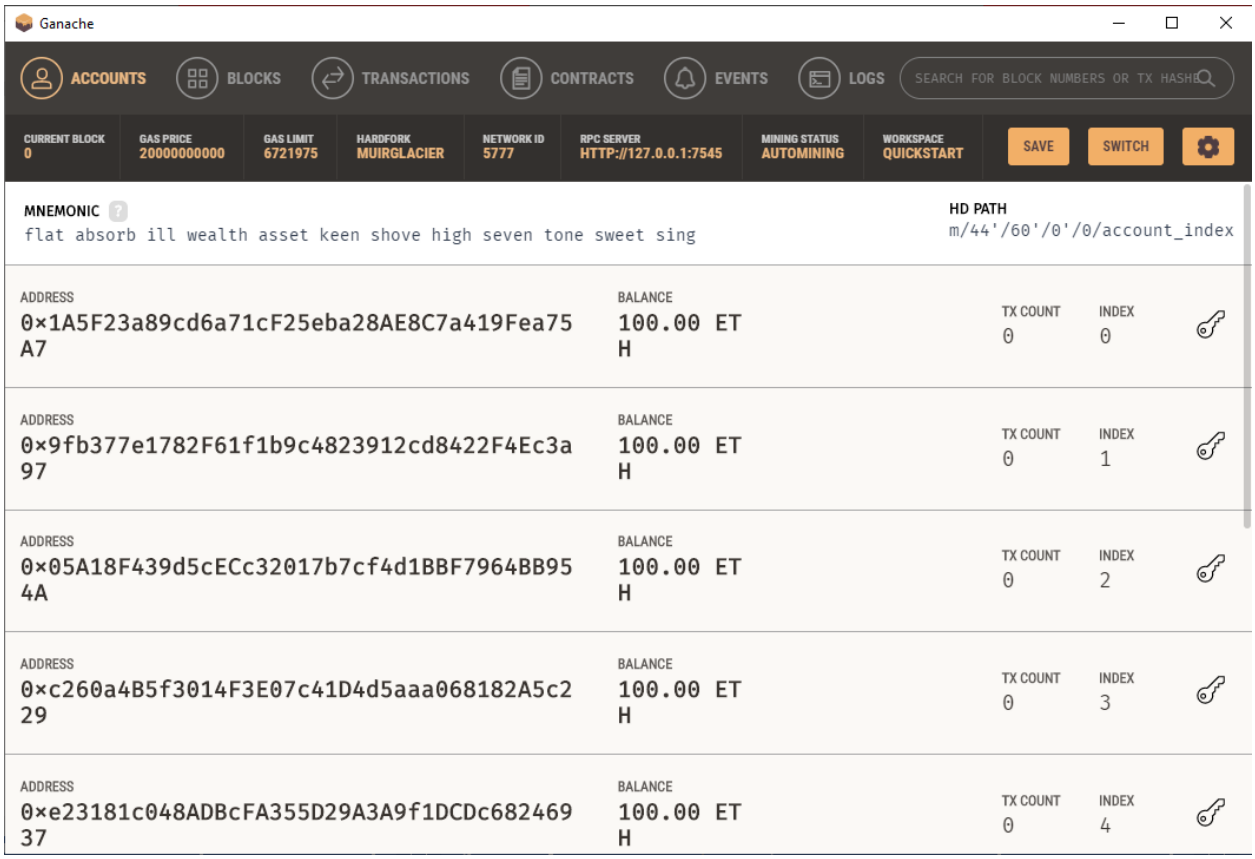


Рис. 15. Акаунти створенні за допомогою Ganache

Ganache діє як наш локальний блокчейн, тому ми можемо розгорнути і протестувати його функціональність локально, перш ніж розгортати його в загальнодоступній тестової мережі.

Перші 10 адресів є публічними, будуть використовуватись для взаємодії з блокчейном, і на них Ganache поклав 100 ETH.

У файл truffle-config.js додаємо наступний код

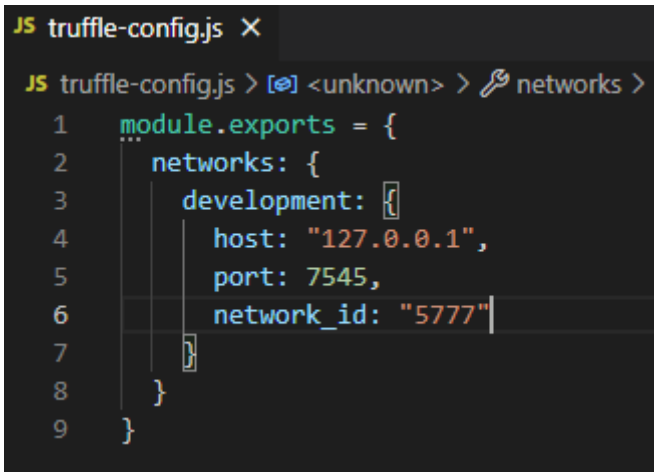


Рис. 16. Config файл truffle

У даному скрипті host, port та network_id вказуються такі ж як і на головному вікні Ganache у режимі Quickstart, що можна побачити з рисунку 15. І, як можна зрозуміти, цей скрипт виконує підключення блокчейну до Ganache.

Таким чином Tuffle буде знати як знайти локальний блокчейн, у даному файлі ми визначили мережу development. Для тесту введемо у командному рядку наступну команду:

truffle test

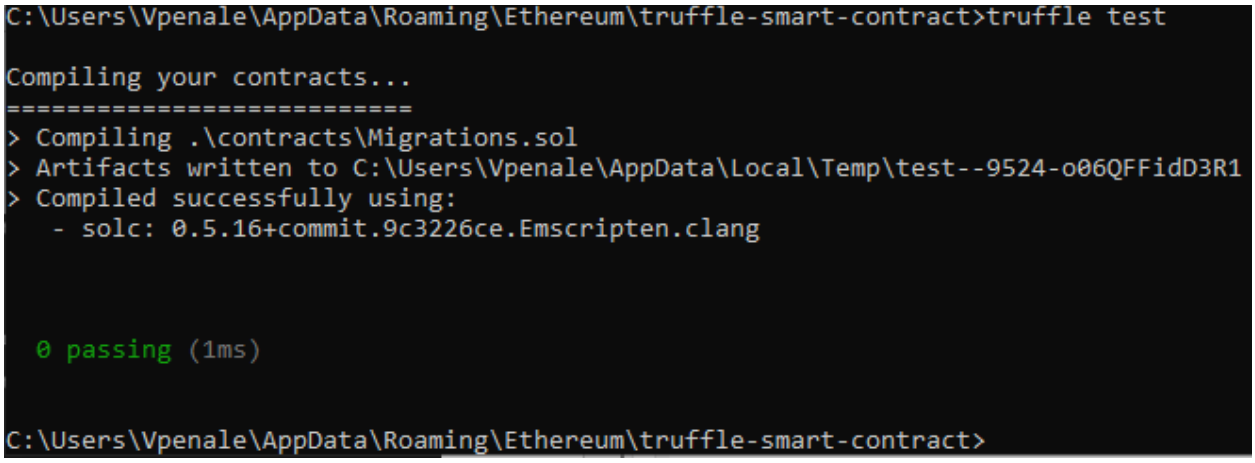


Рис. 17. Тестування проєкту Truffle

Додамо тестовий Smart Contract у папку /contracts

contracts > HelloWorld.sol

```
1   pragma solidity >=0.5.0;
2
3   contract HelloWorld {
4       string private greeting;
5
6       constructor() public {
7           greeting = "Hello World";
8       }
9
10      function getGreeting() public view returns(string memory){
11          return greeting;
12      }
13  }
```

Рис. 18. Контракт

Щоб скомпілювати даний код у машинний та задеплоїти до блокчейну введемо у cmd наступну команду:

truffle compile

```
C:\Users\Vpenale\AppData\Roaming\Ethereum\truffle-smart-contract>truffle compile

Compiling your contracts...
===========================
> Compiling .\contracts\HelloWorld.sol
> Compiling .\contracts\Migrations.sol
> Artifacts written to C:\Users\Vpenale\AppData\Roaming\Ethereum\truffle-smart-contract\build\contracts
> Compiled successfully using:
   - solc: 0.5.16+commit.9c3226ce.Emscripten.clang


C:\Users\Vpenale\AppData\Roaming\Ethereum\truffle-smart-contract>
```

Рис. 19. Компілювання проєкту

У цьому розділі ми скомпілювали наш новий смарт-контракт в машинний код і розгорнули в нашом локальному ланцюжку блоків.

Щоб задеплоїти створений контракт до блокчейну використовуємо міграції, для цього створимо наступний файл міграції.

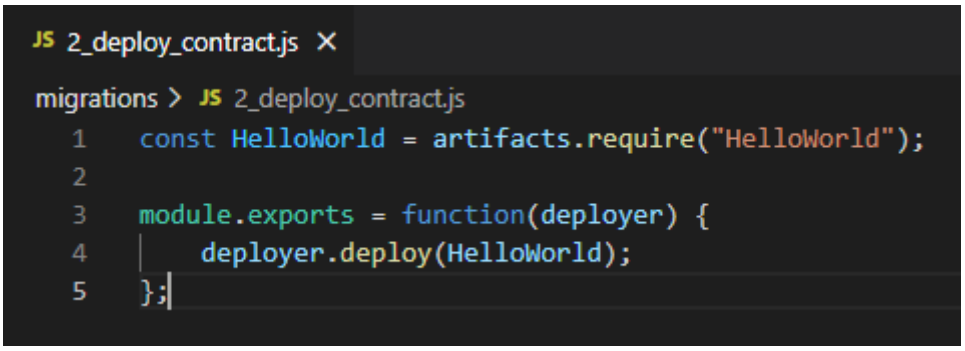


Рис. 20. Файл міграції для деплою контрактів

Міграція - це процес розгортання скомпільованих смарт-контрактів в блокчейні.

Для її виконання вводимо наступну команду:

truffle migrate

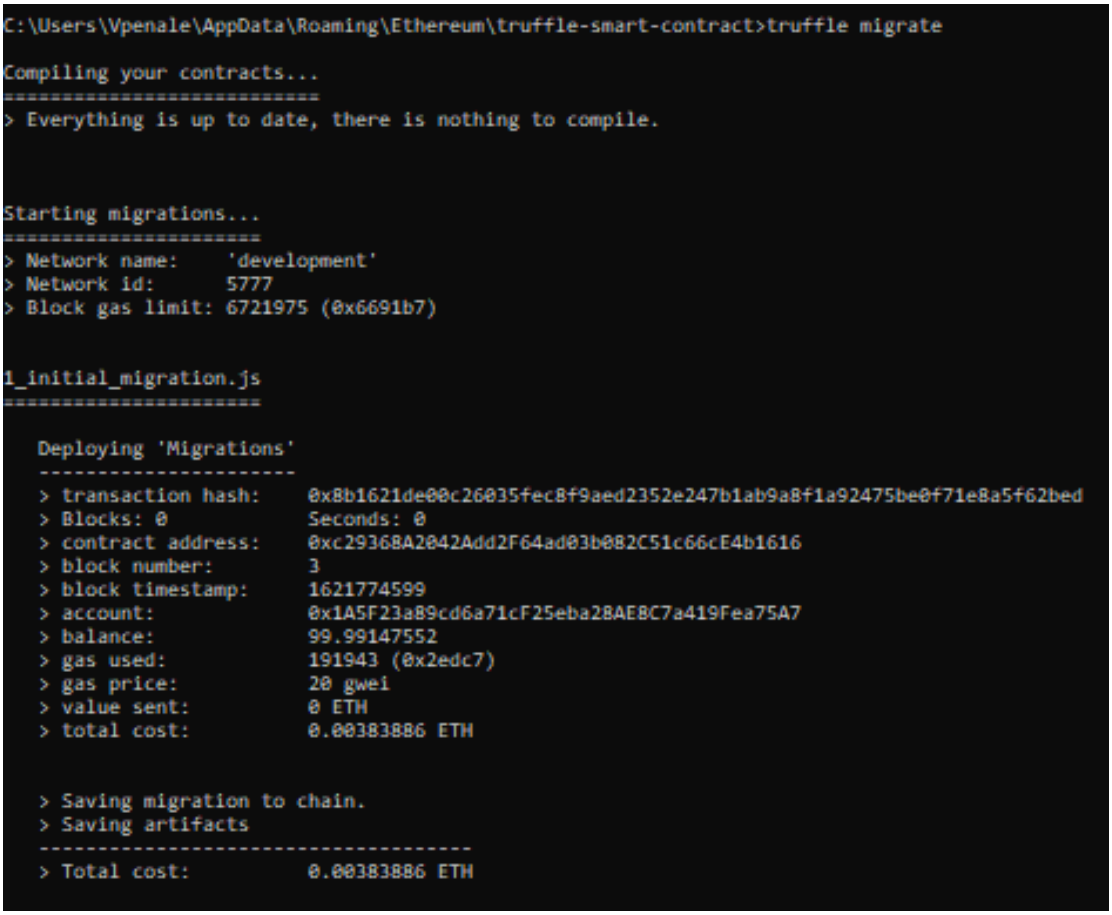


Рис. 21. Вивід команди ‘truffle migrate’

```
2_deploy_contract.js
====================

   Deploying 'HelloWorld'
   ----------------------
   > transaction hash:    0x0de912917b5ab553b1fb8b2627b6bd7373834ce80e617ff0d40f5480c6c88b27
   > Blocks: 0            Seconds: 0
   > contract address:    0x39aC184f63D7Ea0D1b40be288CdD7E6D2D439919
   > block number:        5
   > block timestamp:     1621774602
   > account:             0x1A5F23a89cd6a71cF25eba28AE8C7a419Fea75A7
   > balance:             99.98736614
   > gas used:            163131 (0x27d3b)
   > gas price:           20 gwei
   > value sent:          0 ETH
   > total cost:          0.00326262 ETH


   > Saving migration to chain.
   > Saving artifacts
   -------------------------------------
   > Total cost:          0.00326262 ETH


Summary
=======
> Total deployments:   2
> Final cost:          0.00710148 ETH


C:\Users\Vpenale\AppData\Roaming\Ethereum\truffle-smart-contract>
```

Рис. 22. Деплой контракту ‘HelloWorld’

Контракт задеплоївся, у вікні Ganache можемо бачити зняття 0.00326262 ETH з першого аккаунту.

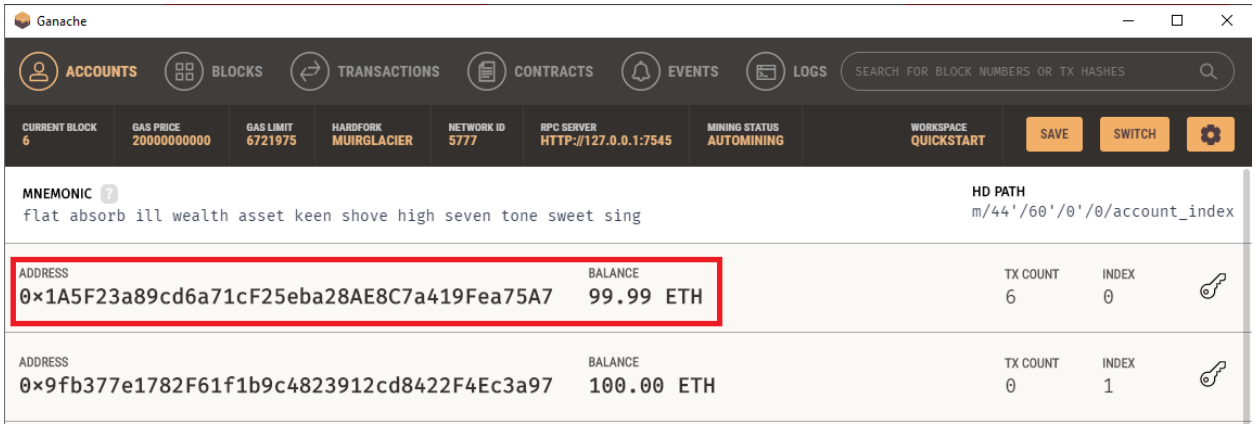


Рис. 23. Зміна балансу після деплою

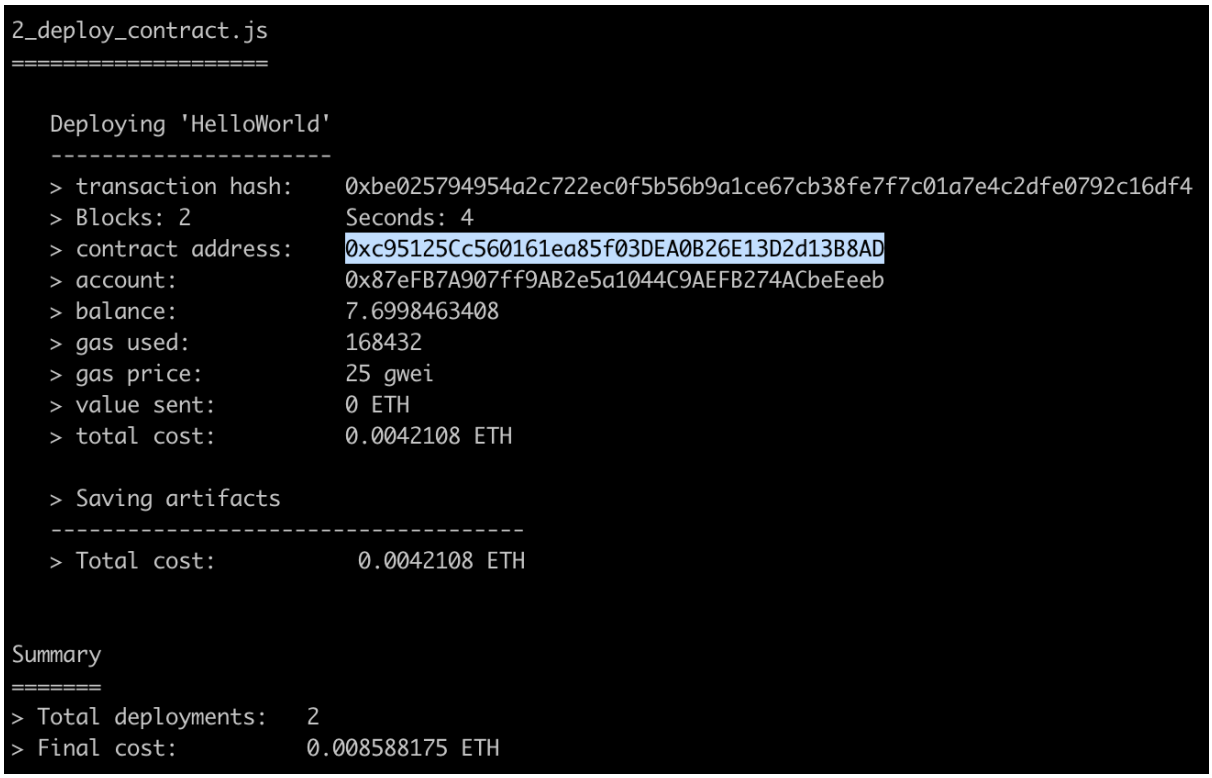


Рис. 24. Міграція контракту за допомогою kovan та сервісу Infura.

Наш контракт був розгорнутий локальному блокчейні. Перейшовши до Ganache, та подивившись на перший рядок в таблиці рахунків, можна побачити, що баланс ETH трохи знизився, і що кількість транзакцій зросла вище 0, як показано на Рисунку 23.

Баланс ETH падає, тому що GAS вимагає розгортання в блокчейні, а збільшена кількість транзакцій представляє собою транзакції, які використовувалися при розгортанні контракту.

За умови, якщо вартості вашого смарт-контракту не вистачило щоб внести зміни до кількості ефірума в Ganache, по аналогії створіть ще кілька контрактів та запустіть команду «truffle migrate --reset», щоб загальна їх вартість становила більше ніж 0.005 ефіріума і внесла відповідні зміни у Ganache.

6.9 Контрольні запитання

1) Що таке смарт-контракт?
2) Призначення Ganache.
3) Різниця між тестовою та головною мережею.

6.10 Контрольне завдання

Розгорнути середовище для створення смарт-контрактів та задеплоїти простий приклад смарт-контракту у блокчейн. **Повторити з методичних вказівок**.

Результатами роботи та виконання ЛР є скріншоти з IDE та OS.

СПИСОК ІНФОРМАЦІЙНИХ ДЖЕРЕЛ

1. Chocolatey:https://docs.chocolatey.org/en-us/choco/setup#non-administrative-install
2. Ganache: https://www.trufflesuite.com/ganache
3. Фреймворк Truffle: https://www.trufflesuite.com/truffle
4. Kovan testnet: https://gitter.im/kovan-testnet/faucet
5. Сервіс Infura: https://infura.io/
6. Сервіс Kovan: https://kovan.etherscan.io/
7. VisualStudioCode: https://code.visualstudio.com/download
8. Geth: https://geth.ethereum.org/downloads/
9. Node.js: https://nodejs.org/uk/
10. https://github.com/dvf/blockchain
11. Learn Blockchains by Building One https://medium.com/@vanflymen/learn-blockchains-by-building-one-117428612f46
12. https://medium.com/tixlorg/guide-get-a-btc-testnet-wallet-with-tokens-and-transfer-them-to-the-tixl-testnet-807f1a6ef8b2