Державний торговельно-економічний університет
Кафедра інженерії програмного забезпечення та кібербезпеки

ВИПУСКНА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«API ДЛЯ АВТОМАТИЗОВАНОЇ ГЕНЕРАЦІЇ ТА РОЗГОРТАННЯ ERC - 20 ТОКЕНІВ»

Студентки 2мз курсу, 2 групи, спеціальності 121 «Інженерія програмного забезпечення» освітньої програми «Інженерія програмного забезпечення» _____ підпис студента — Задорожної Аліни Віталіївни

Науковий керівник кандидат педагогічних наук, доцент кафедри інженерії програмного забезпечення та кібербезпеки _____ підпис керівника — Жирова Тетяна Олександрівна

Гарант освітньої програми кандидат педагогічних наук, доцент кафедри інженерії програмного забезпечення та кібербезпеки _____ підпис гаранта — Котенко Наталія Олексіївна

КИЇВ – 2023

Факультет інформаційних технологій
Кафедра інженерії програмного забезпечення та кібербезпеки
Освітній ступінь магістр
Освітня програма 121 «Інженерія програмного забезпечення»

Затверджую
Зав. кафедри інженерії програмного забезпечення та кібербезпеки
Криворучко О. В.
«13» грудня 2022 р.

Завдання
на випускну кваліфікаційну роботу студентові

Задорожній Аліні Віталіївні
(прізвище, ім'я, по батькові)

1. Тема випускної кваліфікаційної роботи «API для автоматизованої генерації та розгортання ERC - 20 токенів»
Затверджена наказом ректора від 9 грудня 2022 р. № 3339
2. Строк здачі студентом закінченої роботи 27 листопада 2023р.
3. Цільова установка та вихідні дані до роботи
Мета роботи розробка та аналіз API для автоматизованої генерації та розгортання ERC-20 токенів на блокчейні Ethereum.
Об'єкт дослідження процес створення та розгортання ERC-20 токенів на блокчейні Ethereum
Предмет дослідження розробка та аналіз функціональності API, яке дозволяє автоматизовано створювати та розгортати ERC-20 токени на блокчейні Ethereum.

4. Консультанти роботи із зазначенням розділів, які консультують:

Розділ	Консультант (прізвище, ініціали)	Підпис, дата	
		Завдання видав	Завдання прийняв

5. Зміст випускної кваліфікаційної роботи (перелік питань за кожним розділом)

ВСТУП

РОЗДІЛ 1 ДОКУМЕНТАЦІЯ ДЛЯ РОЗРОБКИ API ТА СТВОРЕННЯ СТРУКТУРИ ІНФОРМАЦІЙНИХ ПОТОКІВ

1.1. Документація API для автоматизованої генерації та розгортання ERC-20 токенів

1.2. Використання Web3.js та OpenZeppelin при розробці API для взаємодії з блокчейном Ethereum

1.3. REST архітектура

1.4. Swagger як інструмент документації якою забезпечується досліджуваний інформаційний процес

1.5. Структура інформаційних потоків API для автоматизованої генерації та розгортання ERC - 20 токенів

1.6. Висновки до розділу 1

РОЗДІЛ 2 ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ

2.1.Програмні можливості

2.2.Переваги та недоліки інтеграції API

2.3.Використання проксі

2.4. Побудова структури БД

2.5. Висновки до розділу 2

РОЗДІЛ 3 ЗАСОБИ РОЗРОБКИ

3.1. Середовище розробки

3.2. Інструменти для вирішення задачі

3.2.1 Docker

3.2.2 Postman

3.2.3 Adminer
3.2.4 MetaMask
3.3. Платформа Node.js
3.4. Мова програмування Solidity
3.5. Бібліотеки та фреймворки
3.7. Висновок до розділу 3
РОЗДІЛ 4 ОПИС РЕАЛІЗАЦІЇ API
4.1. Цільова аудиторія API
4.2.Опис системи
4.3. Висновки до розділу 4
ВИСНОВКИ ТА ПРОПОЗИЦІЇ
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ
ТЕХНІЧНЕ ЗАВДАННЯ
ТЕСТУВАННЯ ДОДАТКА
ДОДАТКИ

6. Календарний план виконання роботи

№ пор.	Назва етапів випускної кваліфікаційної роботи	Строк виконання етапів роботи	
		за планом	фактично
1	2	3	4
1.	*Вибір теми випускної кваліфікаційної роботи*	*07.11.2022*	*07.11.2022*
2.	*Розробка та затвердження завдання на роботу магістра (стац/заоч)*	*13.12.2022*	*13.12.2022*
3.	*Вступ та перелік літературних джерел*	*24.02.2023*	*24.02.2023*
4.	*Розробка технічного завдання*	*15.03.2023*	*15.03.2023*
5.	*Розділ 1. Документація для розробки api та створення структури інформаційних потоків*	*10.04.2023*	*10.04.2023*
6.	*Розділ 2. Опис предметної області*	*24.05.2023*	*24.05.2023*
7.	*Розділ 3. Засоби розробки*	*06.09.2023*	*06.09.2023*
8.	*Розділ 4. Опис реалізації API*	*06.09.2023*	*06.09.2023*
9.	*Розробка програми та методики тестування*	*18.10.2023*	*18.10.2023*
10.	*Написання наукової статті*	*17.05.2023*	*17.05.2023*
11.	*Висновки та пропозиції*	*01.11.2023*	*01.11.2023*
12.	*Здача випускної кваліфікаційної роботи на кафедру (перша перевірка)*	*06.11.2023*	*06.11.2023*
13.	*Підготовка автореферату та презентації доповіді*	*06.11.2023*	*06.11.2023*
14.	*Попередній захист випускної кваліфікаційної роботи*	*20.11.2023 – 24.11.2023*	*20.11.2023 – 24.11.2023*
15.	*Здача зброшурованої випускної кваліфікаційної роботи*	*27.11.2023*	*27.11.2023*
16.	*Зовнішнє рецензування випускної кваліфікаційної роботи*	*29.11.2023*	*29.11.2023*
17.	*Підготовка до публічного захисту випускної кваліфікаційної роботи*	*05.12.2023-06.12.2023*	*05.12.2023-06.12.2023*

7. Дата видачі завдання «13» грудня 2022 р.

8. Науковий керівник випускної кваліфікаційної роботи Жирова Т.О.
(прізвище, ініціали, підпис)

9. Гарант освітньої програми Котенко Н.О.
(прізвище, ініціали, підпис)

10. Завдання прийняв до виконання студент Задорожна А. В.
(прізвище, ініціали, підпис)

11. Відгук керівника випускного кваліфікаційного проєкту

Науковий керівник випускної кваліфікаційної роботи

_____

(підпис, дата)

Відмітка про попередній захист _____

(ПІБ, підпис, дата)

12. Висновок про випускну кваліфікаційну роботу

Випускна кваліфікаційна робота студента _____Задорожна А. В._____

(прізвище, ініціали)

може бути допущена до захисту екзаменаційній комісії.

Гарант освітньої програми _____Котенко Н.О._____

(прізвище, ініціали, підпис)

Завідувач кафедри _____Криворучко О. В._____

(підпис, прізвище, ініціали)

«_____»_____20 ___ р.

АНОТАЦІЯ

Мета роботи - розробка та аналіз API для автоматизованої генерації та розгортання ERC-20 токенів на блокчейні Ethereum.

Ця робота присвячена вивченню та аналізу можливостей використання API для автоматизованої генерації та розгортання токенів стандарту ERC-20 на платформі Ethereum. В роботі розглянуті основні аспекти стандарту ERC-20.

Досліджено переваги використання API для створення ERC-20 токенів та важливість гарантій безпеки.

Використання API для автоматизованої генерації та розгортання ERC-20 токенів є потужним інструментом для швидкого впровадження власних токенів у проєктах на базі Ethereum.

Обсяг дипломної роботи складає 43 сторінки, 14 рисунків, 18 використаних літературних джерел та 1 додатку.

ABSTRACT

The goal of the work is the development and analysis of an API for automated generation and deployment of ERC-20 tokens on the Ethereum blockchain.

This work is dedicated to the exploration and analysis of the possibilities of using an API for the automated generation and deployment of ERC-20 standard tokens on the Ethereum platform. The main aspects of the ERC-20 standard are covered in the work.

The advantages of using an API for creating ERC-20 tokens and the importance of security guarantees have been explored.

The use of an API for the automated generation and deployment of ERC-20 tokens is a powerful tool for quickly implementing custom tokens in projects based on Ethereum.

The thesis consists of 43 pages, 14 figures, 18 references, and 1 appendix.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ERC-20 – Ethereum Request for Comment 20

API – Application Programming Interface

REST – Representational State Transfer

HTTP – Hypertext Transfer Protocol

URI – Uniform Resource Identifier

JSON – JavaScript Object Notation

XML – eXtensible Markup Language

БД – база даних

ЗМІСТ

ВСТУП **5**

РОЗДІЛ 1 ДОКУМЕНТАЦІЯ ДЛЯ РОЗРОБКИ API ТА СТВОРЕННЯ СТРУКТУРИ ІНФОРМАЦІЙНИХ ПОТОКІВ **7**

1.1. Документація API для автоматизованої генерації та розгортання ERC-20 токенів 7

1.2. Використання Web3.js та OpenZeppelin при розробці API для взаємодії з блокчейном Ethereum 8

1.3. REST архітектура 9

1.4. Swagger як інструмент документації якою забезпечується досліджуваний інформаційний процес 10

1.5. Структура інформаційних потоків API для автоматизованої генерації та розгортання ERC - 20 токенів 13

1.6. Висновки до розділу 1 14

РОЗДІЛ 2 ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ **15**

2.1. Програмні можливості 15

2.2. Переваги та недоліки інтеграції API 17

2.3. Використання проксі 20

2.4. Побудова структури БД 21

2.5. Висновки до розділу 2 24

РОЗДІЛ 3 ЗАСОБИ РОЗРОБКИ **26**

3.1. Середовище розробки 26

3.2. Інструменти для вирішення задачі 27

3.2.1.Docker 27

3.2.2.Postman 28

3.2.3.Adminer 29

3.2.4.MetaMask 30

3.3. Платформа Node.js 31

3.4. Мова програмування Solidity 32

3.5. Бібліотеки та фреймворки 33

3.6. Висновок до розділу 3 33

РОЗДІЛ 4 ОПИС РЕАЛІЗАЦІЇ API **35**

4.1. Цільова аудиторія API 35

					ДТЕУ 121 02з-5.МР			
Зм.	*Аркуш*	*№ докум.*	*Підпис*	*Дата*				
Зав. каф.		Криворучко О.В.		01.11.23	*API для автоматизованої генерації та розгортання ERC - 20 токенів*	*Стадія*	*Аркуш*	*Аркушів*
Керівник		Жирова Т. О.		01.11.23		*Зміст*	*3*	*43*
Гарант		Котенко Н.О.		01.11.23		*Факультет інформаційних технологій 2мз курс, 2 група*		
Розробив		Задорожна .А В.		01.11.23	*Зміст*			

4.2. Опис системи 36
4.3. Висновок до розділу 4 38
ВИСНОВКИ ТА ПРОПОЗИЦІЇ 40
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 42
ТЕХНІЧНЕ ЗАВДАННЯ 44
ПРОГРАМА ТА МЕТОДИКА ТЕСТУВАННЯ 46
ДОДАТКИ 49

ВСТУП

ERC-20 є стандартом токенів на базі блокчейну Ethereum, що використовується для створення цифрових активів. Автоматизована генерація та розгортання ERC-20 токенів стає все більш популярною, оскільки вона надає зручний та швидкий спосіб створення власних цифрових активів з встановленими правилами торгівлі та управління.

Актуальність. У сучасному світі блокчейн - технології отримують все більше популярності, особливо в контексті фінансових операцій та децентралізованих фінансових інструментів. ERC-20 токени є одним з найпопулярнішим стандартів токенів на блокчейні Ethereum. Автоматизована генерація та розгортання таких токенів може спростити та прискорити процес створення власних криптовалют.

Метою дослідження є розробка та аналіз API для автоматизованої генерації та розгортання ERC-20 токенів на блокчейні Ethereum.

Об'*єктом дослідження* є процес створення та розгортання ERC-20 токенів на блокчейні Ethereum.

Предметом дослідження є розробка та аналіз функціональності API, яке дозволяє автоматизовано створювати та розгортати ERC-20 токени на блокчейні Ethereum.

Задачі дослідження:

1. Аналіз протоколів створення та розгортання ERC-20 токенів.
2. Розробка API для генерації параметрів токена та для розгортання токена на блокчейні Ethereum.
3. Тестування розробленого API на тестовій мережі Ethereum.

					ДТЕУ 121 02з-5.МР			
Зм.	*Аркуш*	*№ докум.*	*Підпис*	*Дата*				
Зав. каф.		Криворучко О.В.		24.02.23	*API для автоматизованої генерації та розгортання ERC - 20 токенів*	*Стадія*	*Аркуш*	*Аркушів*
Керівник		Жирова Т. О.		24.02.23		*В*	*5*	*43*
Гарант		Котенко Н.О.		24.02.23		*Факультет інформаційних технологій 2мз курс, 2 група*		
Розробив		Задорожна А. В.		24.02.23	*Вступ*			

У роботі розглядається розробка API для автоматизованої генерації та розгортання ERC-20 токенів. Детально розглянуто використання бібліотек Web3.js та OpenZeppelin для взаємодії з блокчейном Ethereum та створення смарт-контрактів ERC-20. Також розглянуто REST архітектуру та Swagger як інструмент документації, які допоможуть створити зручну та добре задокументовану API.

Опис предметної області API включатиме розгляд використання проксі, які гарантують безпеку та ефективність взаємодії з блокчейном. Також буде висвітлено побудову структури бази даних, яка дозволяє зберігати та керувати важливою інформацією про токени, контракти та взаємодію з користувачами.

У результаті розробка API для автоматизованої генерації та розгортання ERC-20 токенів буде корисною для розробників, які прагнуть створити свої власні токени або інтегрувати ERC-20 в свої додатки та платформи. Завдяки API, розробники матимуть зручний спосіб автоматизувати процес створення та розгортання токенів, що значно скоротить час і зусилля, необхідні для цих операцій.

Крім того, надання документації для API дозволить розробникам швидко ознайомитись з функціональністю та можливостями API. Використання REST архітектури та інструменту Swagger для документації дозволить створити зрозумілу інформаційну модель, яка допоможе користувачам ефективно використовувати API та взаємодіяти з ним без зайвих зусиль.

РОЗДІЛ 1
ДОКУМЕНТАЦІЯ ДЛЯ РОЗРОБКИ API ТА СТВОРЕННЯ СТРУКТУРИ ІНФОРМАЦІЙНИХ ПОТОКІВ

1.1. Документація API для автоматизованої генерації та розгортання ERC-20 токенів

Документація для розробки API та створення структури інформаційних потоків має на меті забезпечити розробникам та користувачам зрозумілу та повну інформацію щодо використання та функціональності API. Також допомагає забезпечити документований інформаційний процес.

Документація ERC-20 стандарту визначає набір правил та специфікацій для створення та роботи з токенами на блокчейні Ethereum. Цей стандарт став популярним в індустрії криптовалют і встановив основні норми, які повинні бути дотримані при розробці ERC-20 токенів [1].

Документація ERC-20 визначає такі ключові елементи:

- Реалізація стандартних методів, які передбачені ERC-20 стандартом. Це включає методи для отримання балансу токенів, переказу токенів, отримання загального постачання токенів тощо. Потрібно забезпечити роботу API згідно з вимогами ERC-20 стандарту.
- Документація ERC-20 визначає структури даних, які використовуються для зберігання інформації про баланси токенів та інші пов'язані дані.

- ERC-20 визначає певні події, які повинні бути сповіщені у разі здійснення певних дій, наприклад, переказу токенів. Це дозволяє моніторити події, пов'язані з токенами, та реагувати на них.
- Документація ERC-20 також встановлює ряд інших вимог, таких як обов'язковість реалізації методу "approve" для дозволу на переказ токенів від імені користувача та вимогу про наявність певних загальних змінних і функцій [2].

Розробка API для автоматизованої генерації та розгортання ERC-20 токенів повинна враховувати ці специфікації та реалізувати відповідні методи та структури даних.

1.2. Використання Web3.js та OpenZeppelin при розробці API для взаємодії з блокчейном Ethereum

Використання бібліотеки Web3.js та фреймворку OpenZeppelin є популярним підходом при розробці API для взаємодії з блокчейном Ethereum.

Web3.js бібліотека дозволяє взаємодіяти з Ethereum мережею за допомогою JavaScript. она надає інтерфейс для створення запитів до Ethereum мережі, взаємодії зі смарт-контрактами, надсилання транзакцій та отримання даних з блокчейну. Web3.js допомагає забезпечити взаємодію з ERC-20 токенами та їхніми смарт-контрактами [3].

OpenZeppelin - це популярна бібліотека для розробки безпечних контрактів на Ethereum. Вона містить готові реалізації контрактів ERC-20, які можна використовувати як основу для API. Використання цих інструментів спрощує розробку, покращує безпеку та прискорює процес розгортання додатків на блокчейні [4].

1.3. REST архітектура

Архітектура REST (Representational State Transfer) - це структурований підхід до розробки мережевих додатків, який базується на кількох ключових принципах. REST є архітектурним стилем, який використовується для створення веб-сервісів та API.

Основні принципи REST включають наступне:

- REST використовує HTTP методи, такі як GET, POST, PUT та DELETE, для взаємодії з ресурсами. Ці методи використовуються для створення, отримання, оновлення та видалення ресурсів, пов'язаних з ERC-20 токенами.
- Кожен ресурс в API має свій унікальний URI, за яким можна звертатися до нього. Наприклад, для отримання інформації про певний токен, URI може бути у форматі "/tokens/{id}", де {id} є ідентифікатором токену.
- REST використовує стандартні формати обміну даними, такі як JSON або XML, для представлення даних, що передаються через API.
- REST може використовувати механізми кешування, щоб зберегти ресурси та зменшити навантаження на сервер [5].

У API для автоматизованої генерації та розгортання ERC-20 токенів, можна використовувати ці принципи REST для визначення шляхів (endpoints) та взаємодії з ресурсами, такими як токени, користувачі, транзакції тощо [6].

REST є популярним стилем для створення веб-сервісів та API, оскільки він спрощує взаємодію між різними системами та дозволяє створювати гнучкі та масштабовані додатки. Використання HTTP - протоколу для комунікації дозволяє легко інтегрувати REST - сервіси у різні мови програмування та платформи.

1.4. Swagger як інструмент документації якою забезпечується досліджуваний інформаційний процес

Swagger є інструментом для створення та публікації документації API, який забезпечує досліджуваний інформаційний процес у сфері розробки програмного забезпечення та веб-сервісів. Він сприяє розумінню та використанню API шляхом створення документації з його використанням.

- Swagger дозволяє автоматично генерувати документацію API на основі самого коду. За допомогою анотацій та метаданих у коді, Swagger генерує описові та зрозумілі документи, що допомагають розробникам, тестувальникам та іншим зацікавленим сторонам зрозуміти, як працює API.
- Swagger надає візуальне подання специфікацій API за допомогою інтерактивних інструментів. Це включає відображення шляхів запитів, параметрів, відповідей та дозволяє випробовувати запити безпосередньо з інтерфейсу Swagger.
- Swagger допомагає стандартизувати спосіб документування API, що полегшує роботу з різними API. Він використовує специфікацію OpenAPI, яка забезпечує уніфікований формат для опису API.
- Swagger дозволяє легко знаходити та доступатися до документації API, що сприяє швидкому зрозумінню функцій та можливостей, які надає API.
- Swagger допомагає розробникам під час реалізації та тестування API, оскільки надає документацію, що дозволяє реалізовувати правильні запити та впевнитися, що API працює правильно.
- Swagger сприяє взаєморозумінню між різними командами, оскільки надає однозначний та доступний для всіх спосіб спілкування про функції та можливості API.

У результаті Swagger виступає як інструмент, що полегшує досліджуваний інформаційний процес у сфері розробки програмного

забезпечення, роблячи API більш зрозумілим, доступним та зручним для користувачів та розробників [7].

Під час розробки API для автоматизованої генерації та розгортання ERC - 20 токенів створюється функція swaggerGenerate (Рис.1.1. Функція swaggerGenerate). Ця функція буде автоматично заповнювати файл swagger.json - це і є документація по API.

```
function swaggerGenerate(app) {
  const config = new DocumentBuilder()
    .setTitle('Fantoken token builder')
    .setDescription(
      `In order to create a token, you must: \n
    1. If you don't have an account, you can create a new one - POST /accounts
    2. Create a token - POST /tokens
    3. If you don't have a network, you can create a new one - POST /networks
    4. Deploy a token - POST /accounts/{accountAlias}/tokens/{symbol}/deploy/{networkName}
    5. Deploy a proxy - POST /accounts/{accountAlias}/tokens/{symbol}/deploy/{networkName}/proxy`,
    )
    .setVersion('1.0')
    .build();

  const document = SwaggerModule.createDocument(app, config, {
    deepScanRoutes: true,
  });
  const outputPath = path.resolve(process.cwd(), 'swagger.json');

  writeFile(outputPath, JSON.stringify(document, null, 4), {
    encoding: 'utf8',
  });
}
```

Рис.1.1. Функція swaggerGenerate

Для того щоб побачити документацію потрібно використати онлайн інструмент Swagger Editor завантаживши туди файл swagger.json.

В результаті буде отримано таку документацію API:

- Інструкція по користуванню, яку також можна побачити в самій функції swaggerGenerate.
- Запити за замовчуванням, кожен з яких можна розгорнути та побачити метод використання (Рис.1.2. Запити за замовчуванням).
- Схеми, які також можна розгорнути і побачити всі змінні та їх типи даних, що міститься в API (Рис.1.3. Схеми даних).

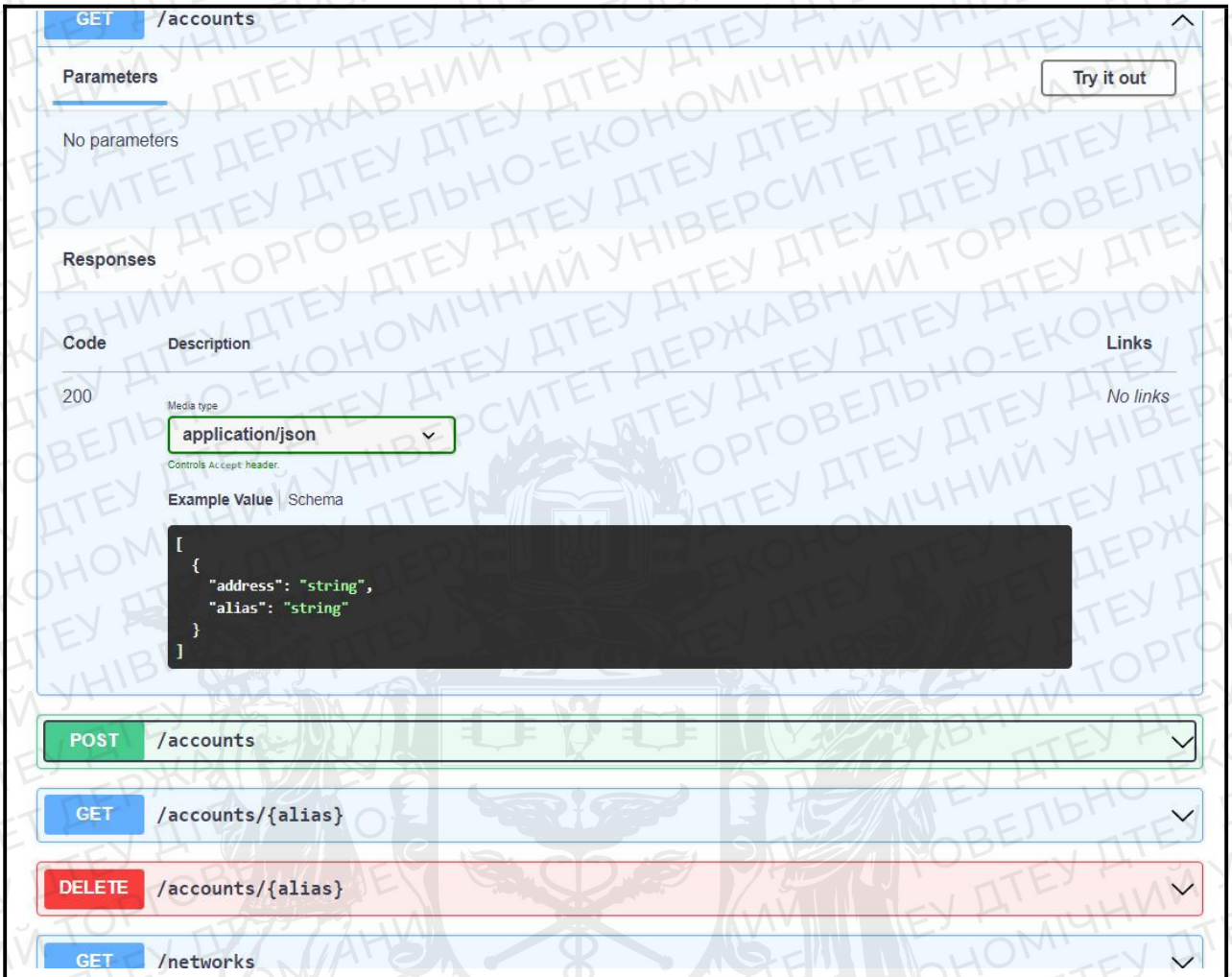

Рис.1.2. Запити за замовчуванням

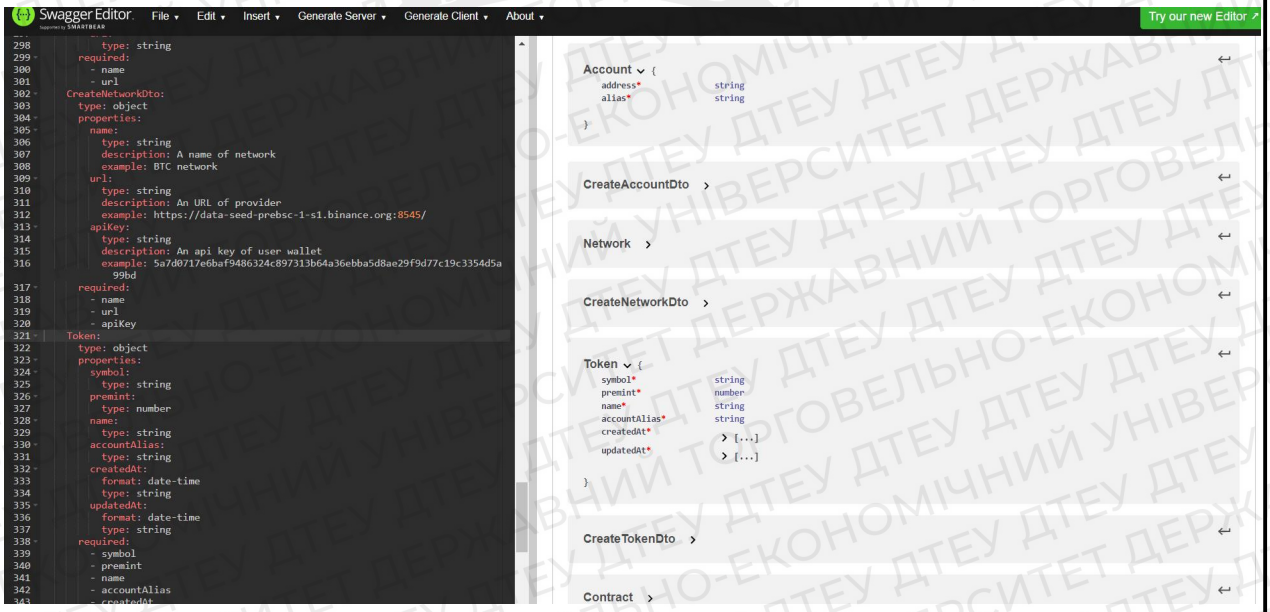

Рис.1.3. Схеми даних

1.5. Структура інформаційних потоків API для автоматизованої генерації та розгортання ERC - 20 токенів

Структура інформаційних потоків API для автоматизованої генерації та розгортання ERC - 20 токенів:

- API може мати ендпоінт для створення нового контракту ERC-20. Користувачі можуть передавати необхідні параметри, такі як назва токену, символ, загальна кількість токенів, десяткові знаки тощо. API повинен перевіряти ці параметри і використовувати Web3.js або OpenZeppelin для створення контракту ERC-20 на блокчейні Ethereum.
- API може мати ендпоінт для розгортання створеного контракту на блокчейні Ethereum. Цей етап включає в себе відправку транзакції з використанням підпису користувача та виконання контракту на блокчейні. API повинен використовувати Web3.js або аналогічні інструменти для взаємодії з Ethereum мережею та розгортання контракту.
- API повинен мати валідацію вхідних даних, щоб перевірити, чи вони відповідають очікуваним форматам та правилам. Наприклад, перед створенням контракту, API повинен перевірити, чи всі необхідні поля передані та чи мають вони правильні значення. Помилки валідації повинні бути повернуті відповідними статусами помилок і повідомленнями про помилки, щоб клієнтські програми могли адекватно реагувати.
- Необхідно розробити документацію API, яка пояснює доступні ендпоінти, параметри, типи даних, очікувані відповіді та приклади використання. Це допоможе розробникам інтегрувати ваше API та зрозуміти його функціонал.
- При розробці API для автоматизованої генерації та розгортання ERC-20 токенів необхідно враховувати безпекові практики. Це включає застосування аутентифікації, авторизації та шифрування для захисту конфіденційності та цілісності даних [8][9].

1.6. Висновки до розділу 1

Метою структури інформаційних потоків та документації API для автоматизованої генерації та розгортання ERC-20 токенів є забезпечення зручного та ефективного використання API, покращення комунікації між розробниками та користувачами, та гарантування безпеки та надійності взаємодії з блокчейном Ethereum.

РОЗДІЛ 2
ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ
2.1. Програмні можливості

API для автоматизованої генерації та розгортання ERC-20 токенів надає широкий набір програмних можливостей, які дозволяють розробникам створювати та управляти токенами на блокчейні Ethereum.

Програмних можливостей API:

- API надає можливість створювати нові ERC-20 токени. Розробники можуть визначити основні параметри токенів, такі як назва, символ, кількість десяткових знаків та загальний обсяг токенів.
- Розробники можуть використовувати API для зберігання та оновлення балансів користувачів ERC-20 токенів. Це дозволяє здійснювати операції з переказом токенів між різними гаманцями.
- API дозволяє створювати та підтверджувати транзакції ERC-20 токенів. Це включає в себе переказ, покупку, продаж та інші операції з токенами.
- API дозволяє автоматизовано створювати гаманці для користувачів, які будуть володіти ERC-20 токенами. Це спрощує процес реєстрації користувачів та надання їм доступу до токенів.
- Розробники можуть використовувати API для моніторингу та аналізу транзакцій ERC-20 токенів. Це допомагає відстежувати рух токенів та проводити аналіз активності користувачів.
- API може бути інтегровано з іншими фінансовими та блокчейн - сервісами, такими як обмінні біржі або платіжні системи. Це розширює

можливості використання ERC-20 токенів.

- API надає можливості для забезпечення безпеки транзакцій та доступу до токенів. Він може включати автентифікацію, авторизацію та шифрування даних.
- API надає документацію, що допомагає розробникам розуміти його можливості та використання. Також, підтримка API може надавати додаткові консультації та відповіді на запитання.

Ці програмні можливості дозволяють розробникам створювати та управляти ERC-20 токенами з використанням стандартизованого та документованого API, що полегшує розробку криптовалютних додатків та послуг.

Опис предметної області API:

- Методи для генерації нових ERC-20 токенів з заданими параметрами, такими як назва токену, символ, обсяг, десяткові знаки та інші властивості.
- Методи для керування токенами, а саме перевірки балансу токенів на рахунку або адресі. Можливість виконання переказів токенів між рахунками або адресами.
- API повинно відповідати вимогам стандарту ERC-20 для забезпечення сумісності з іншими додатками та сервісами, що працюють з ERC-20 токенами.Надання методів та функцій, які дозволяють інтегрувати ERC-20 токени в інші сервіси або використовувати їх у розумінні стандарту.
- Гарантування безпеки взаємодії з API шляхом використання шифрування та інших заходів безпеки. Механізми автентифікації, такі як використання токенів доступу, ключів або інших методів ідентифікації, для контролю доступу до API та його ресурсів.
- Надання детальної документації, яка описує функціональність, параметри та використання API для автоматизованої генерації та розгортання ERC-20 токенів.

API для автоматизованої генерації та розгортання ERC-20 токенів стане важливим інструментом для розробників, які бажають створити власні криптовалютні токени і впровадити їх в різноманітні додатки та сервіси. Предметна область API для ERC-20 токенів є ключовою для розробки криптовалютних додатків та послуг на блокчейн - платформі Ethereum. Розробники, користувачі та фінансові установи можуть використовувати це API для створення та управління ERC-20 токенами, що розширює можливості інвестицій та фінансових послуг на основі блокчейну.

Крім того предметна область API для автоматизованої генерації та розгортання ERC-20 токенів стосується роботи з криптовалютами. Ця область включає в себе різні аспекти, пов'язані з створенням, управлінням та використанням токенів на платформі Ethereum.

Криптовалюта це цифрові або криптографічні активи, які використовуються як засіб обміну та зберігання значення. Ethereum є однією з популярних блокчейн - платформ для роботи з криптовалютами.

2.2. Переваги та недоліки інтеграції API

Інтеграція API для автоматизованої генерації ERC-20 токенів надає кілька значущих переваг:

- Автоматизована генерація ERC-20 токенів через API дозволяє швидше та ефективніше створювати велику кількість токенів без необхідності в ручній роботі. Це полегшує і прискорює процес створення нових токенів.
- Використання API для автоматизації генерації токенів може допомогти уникнути людських помилок, які можуть виникнути під час ручного процесу створення та розгортання токенів.
- API надає можливість масштабування процесу генерації токенів, дозволяючи створювати та розгортати токени відповідно до зростаючих

потреб.

- За допомогою API можна легко налаштувати параметри нових токенів (назва, символ, обсяг тощо) через визначені запити та параметри.
- Інтегровані API можуть забезпечити вбудовані механізми безпеки та слідкування за транзакціями, зменшуючи ризики шахрайства або недостатньої відстежуваності.
- Використання API дозволяє встановлювати стандартизовані процедури та оптимізувати процес генерації ERC-20 токенів, забезпечуючи однаковий підхід до їх створення.
- Багато API постачаються з докладною документацією та демонстраційними прикладами, що полегшує їх інтеграцію та розуміння для розробників.

Незважаючи на переваги інтеграції API для автоматизованої генерації ERC-20 токенів, вона також деякі недоліки:

- Використання API може створити додаткові точки входу для потенційних атак, якщо не забезпечити належні заходи безпеки. Недостатня увага до захисту API може призвести до порушень безпеки та витоку конфіденційної інформації.
- Інтегровані системи стають залежними від надійності та доступності зовнішнього API. Якщо API не доступне або має перебої в роботі, це може призвести до призупинення функціонування генерації токенів.
- Деякі API можуть мати обмежені можливості або функціонал, що не відповідає конкретним потребам або вимогам вашого проекту. Це може обмежити гнучкість та можливості системи.
- Деякі API можуть бути платними або мати обмеження на кількість запитів, що можуть бути виконані безкоштовно. Це може призвести до витрат на використання платних планів або до обмеження можливостей через обмеження кількості запитів.

- Недостатня або невідповідна документація може ускладнити інтеграцію та розуміння API. Також, відсутність або недостатня підтримка з боку розробників може ускладнити вирішення можливих проблем або запитань.
- Будь-яке програмне забезпечення, включаючи API, піддається технічним помилкам та відмовам, які можуть призвести до недоліків у функціонуванні системи.

При інтеграції API для автоматизованої генерації ERC-20 токенів важливо бути уважним та враховувати ці переваги та недоліки для забезпечення безпечної та надійної роботи системи.

Щоб досягти успішного генерування та розгортання токенів на блокчейні Ethereum через API, важливо враховувати наступні аспекти:

- Вибрати API, яке має надійність, гарну документацію та підтримку.
- Важливо, щоб API надавало засоби автентифікації та шифрування для захисту конфіденційності та цілісності даних.
- Вибір API, яке дозволяє налаштовувати параметри токенів, такі як назва, символ, обсяг тощо, для відповідності вашим потребам.
- Забезпечити систему моніторингу та логування подій, щоб мати можливість відстежувати та аналізувати взаємодію з API.
- Обране API має відповідати стандартам ERC-20 та вимогам блокчейну Ethereum.
- Потрібно забезпечити належне резервне копіювання та можливість відновлення даних у випадку втрати інформації або інцидентів.

Використання API для генерації та розгортання токенів на блокчейні Ethereum може спростити процес, але вимагає уважності та дотримання вищезазначених принципів для досягнення успішного результату.

2.3. Використання проксі

Основна функція проксі-сервера - це перенаправлення запитів, які надходять від клієнтського додатку до вузлів блокчейну, обробка цих запитів і повернення відповідей назад до клієнта [10].

Проксі-сервер має декілька важливих функцій у контексті розгортання ERC-20 токенів:

- Проксі-сервер може виконувати функцію фільтрації та перевірки запитів, що надходять до блокчейну. Це дозволяє контролювати доступ до функціональності блокчейну та запобігати несанкціонованому виконанню операцій.
- Проксі-сервер може кешувати деякі дані або використовувати механізми кешування для прискорення відповідей на запити. Це необхідно, коли потрібно обробити велику кількість запитів на розгортання та взаємодію з багатьма ERC-20 токенами. Це має такі переваги, як покращена продуктивність та зменшення навантаження на сервер. Локальне кешування дозволяє клієнтам отримувати доступ до даних швидше, оскільки вони не повинні чекати на відповідь від сервера. Проксі може зменшити кількість запитів, які досягають сервера, що допомагає зменшити навантаження на сервер та поліпшити його продуктивність.
- Проксі може встановити обмеження на кількість запитів, які клієнт може зробити до сервера. Це допомагає запобігти перевантаженню сервера або зловживанню API.
- Проксі-сервер може виконувати функцію розподілення навантаження, розподіляючи запити між різними вузлами блокчейну. Це допомагає забезпечити більшу масштабованість та доступність системи при великому обсязі запитів.
- Проксі-сервер може виконувати роль посередника для створення резервних копій даних, що передаються через API. Це може бути корисно в

разі втрати даних або необхідності відновлення попереднього стану системи [11].

Загальною метою використання проксі в API для автоматизованої генерації та розгортання ERC-20 токенів є поліпшення безпеки, продуктивності і доступності API для користувачів. Цей посередник допомагає оптимізувати взаємодію між клієнтами та сервером, забезпечуючи більш ефективну та надійну роботу з ERC-20 токенами.

2.4. Побудова структури БД

Побудова структури бази даних для API для автоматизованої генерації та розгортання ERC-20 токенів грає важливу роль у забезпеченні надійності та продуктивності системи.

База даних API для автоматизованої генерації та розгортання ERC - 20 токенів включає такі елементи:

- Account;
- Tokens;
- Token network;
- Networks;
- Contracts (Рис.2.1 Структура БД).

Рис. 2.1. Структура БД

Один акаунт може мати багато токенів, а токен має лише одного власника.

Кожний токен прив’язаний до нетворка (мережі). Наприклад Ethereum, Binance Chain, і тд.

Токен може бути запущений одночасно на декількох нетворках. Відповідно один нетворк може мати багато токенів, так як багато різних токенів можуть бути запущені на одному нетворкі.

Токен має два контракти, один з яких є контрактом токена безпосередньо, а інший контракт проксі токена, якщо є проксі.

У одного токена є один контракт і один проксі контракт.

Проксі контракт створений для можливості зміни реального контракту. Проксі контракт замість реального контракту буде відповідати на запити до нього.

Так як в базі даних не можна зберігати паролі, а тільки хеші паролів, то PrivateKey має бути в байтах. Якщо до бази даних отримають доступ сторонні люди, то вони не зможуть побачити і скористатися паролями.

Таблиця 2.1

ACCOUNT

Ім’я	***Визначення***	***Опис***
Address	Адреса	Унікальний ідентифікатор облікового запису
Alias	Коротке ім’я	Зручне ім’я або позначення для облікового запису. Також буде використовуватися як логін
PrivateKey	Пароль	Пароль для доступу до обліковго запису

Таблиця 2.2

TOKENS

Ім'я	***Визначення***	***Опис***
Symbol	Символ токена	Унікальний символьний код, що ідентифікує токен. Наприклад BNB
Premint	Кількість токена після створення	Кількість токенів, які передбачено виділити перед розподілом
Name	Назва токена	
AccountAlias	Посилання на акаунт	Псевдонім або позначення облікового запису, що створив токен
CrearedAt	Дата створення запису	Дата та час створення токена
UpdatedAt	Дата оновлення запису	Остання дата та час оновлення інформації про токен
ContractId	Ідентифікатор контракту	
ProxyContractId	Ідентифікатор проксі-контракту	

Таблиця 2.3

TOKEN NETWORK

Ім'я	***Визначення***	***Опис***
TokenSymbol	Символ токена	Унікальний символьний код, що ідентифікує токен. Наприклад BNB
NetworksName	Назва мережі	Назва блокчейн мережі, в якій токен може бути розгорнутий

Таблиця 2.4

NETWORKS

Ім'я	***Визначення***	***Опис***
Name	Назва мережі	Унікальна назва блокчейн мережі, до якої API здійснює підключення
Url	URL API	Адреса (URL) API блокчейн мережі для взаємодії з нею
ApiKey	Ключ API	Ключ аутентифікації, який використовується для доступу до API

Таблиця 2.5

CONTRACTS

Ім'я	***Визначення***	***Опис***
Address	Адреса контракту	Унікальна адреса контракту ERC-20 токена, розгорнутого в блокчейні
Id	Ідентифікатор контракту	Унікальний ідентифікатор контракту ERC-20 токена в системі або базі даних
TransactionHash	Хеш транзакцій, в якій було створено контракт	Хеш транзакції, яка була використана для розгортання контракту в блокчейні
CreatedAt	Дата створення	Дата та час створення контракту

2.5. Висновки до розділу 2

Проксі-сервер є посередником між клієнтами та блокчейном Ethereum, і він виконує ряд важливих функцій. Використання проксі дозволяє

забезпечити ефективну та безпечну взаємодію з блокчейном, а також оптимізувати продуктивність API.

Інтеграція API для автоматизованої генерації ERC-20 токенів відображає значні переваги, але також супроводжується певними недоліками, які вимагають уважного врахування та керування.

Правильна структура БД відіграє важливу роль у забезпеченні ефективного зберігання та керування даними, пов'язаними з токенами, контрактами та взаємодією з користувачами.

РОЗДІЛ 3
ЗАСОБИ РОЗРОБКИ
3.1. Середовище розробки

Створення API для автоматизованої генерації та розгортання ERC-20 токенів вимагає використання різних засобів розробки для програмування, тестування та взаємодії з блокчейном.

Ключові засоби, які будуть використані:

- Мови програмування.
- Інтегровані середовища розробки (IDEs).
- Blockchain Development Framework.
- Бібліотеки для взаємодії з блокчейном.
- Тестові мережі та вузли Ethereum.
- Документація.

Visual Studio Code (VS Code) - це безкоштовний та потужний текстовий редактор, який став одним із найпопулярніших інструментів серед розробників. Він надає широкий набір функцій для комфортної та продуктивної розробки, включаючи створення смарт-контрактів та бекенду для API для автоматизованої генерації та розгортання ERC-20 токенів.

Основні можливості VS Code:

- VS Code підтримує ряд розширень для різних мов програмування. Для Solidity можна встановити розширення, яке надає синтаксичне підсвічування, автодоповнення та відладку для смарт-контрактів.
- VS Code має вбудований термінал, що дозволяє виконувати

команди безпосередньо у вікні редактора. Це особливо зручно для запуску тестів, взаємодії з блокчейном та інших команд.

- VS Code надає потужний інструмент для налагодження коду. Можна встановлювати точки зупину, крокувати по коду та аналізувати змінні під час налагодження смарт-контракту.
- У VS Code є багато розширень, які полегшують розробку. Для роботи з Ethereum можна встановити плагіни для роботи з мовою Solidity, блокчейном та інші корисні інструменти.
- VS Code надає автодоповнення для коду, що полегшує написання коду та уникнення помилок. Також можна швидко переглядати документацію для функцій та методів.

Все це робить Visual Studio Code зручним середовищем розробки для створення API для автоматизованої генерації та розгортання ERC-20 токенів [12].

3.2. Інструменти для вирішення задачі

3.2.1.Docker

Docker в API для автоматизованої генерації та розгортання ERC-20 токенів є важливим інструментом для забезпечення ефективності, надійності та зручності у процесі розробки, тестування та розгортання.

Docker - це платформа для контейнеризації додатків, яка дозволяє упаковувати програми та їхні залежності в стандартизовані контейнери. Це відіграє ключову роль в оптимізації розробних процесів.

Docker дозволяє розробникам інтегрувати різні компоненти системи, такі як смарт-контракти, бекенд API, бази даних та інші, в окремі контейнери. Це забезпечує відокремленість та ізоляцію компонентів, що допомагає уникнути конфліктів та забезпечити стабільність системи. Використання Docker також дозволяє стандартизувати середовище

розробки, тестування та розгортання, що спрощує спільну роботу в команді та гарантує однаковість у різних етапах розробки.

Один з ключових переваг Docker - це здатність легко переносити контейнери між різними середовищами. Це дозволяє розгортати створені контейнери на різних серверах без зайвих зусиль з підготовки середовища. Також Docker забезпечує більш ефективне використання ресурсів сервера та підтримує масштабованість, що важливо для забезпечення високої продуктивності [13].

Застосування Docker до розробки API для автоматизованої генерації та розгортання ERC-20 токенів сприяє поліпшенню процесів розробки, забезпеченню надійності та зручності розгортання, а також сприяє взаємодії з іншими інструментами, які використовуються у створенні і керуванні смарт-контрактами на блокчейні Ethereum (Рис.3.1 Використання Docker Desktop).

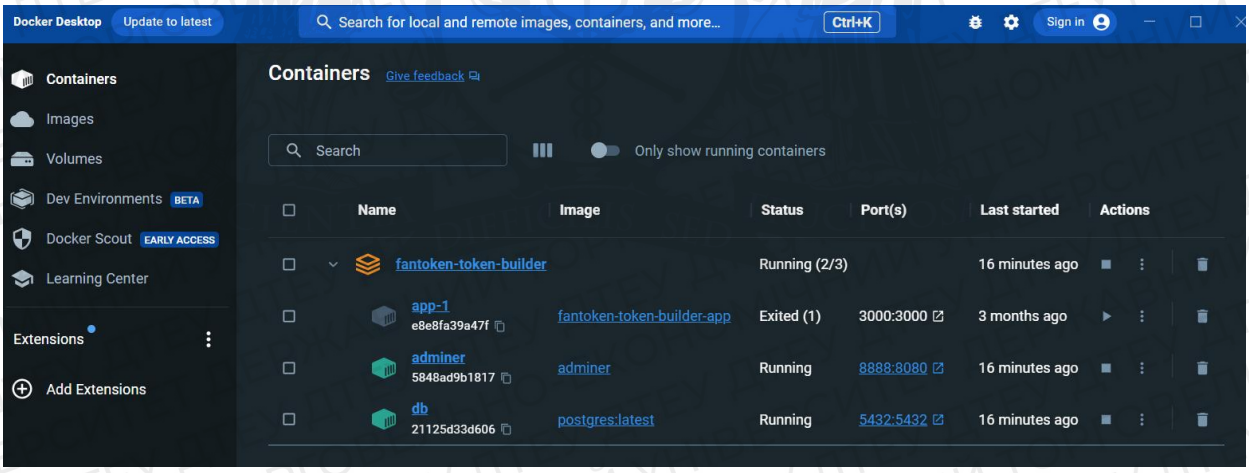

Рис. 3.1. Використання Docker Desktop

3.2.2.Postman

Postman це інструмент для тестування, документування та взаємодії з API, і його використання в проекті API для автоматизованої генерації та розгортання ERC-20 токенів може бути важливим.

Postman дозволяє створювати та відправляти HTTP-запити до API для

перевірки його роботи та взаємодії зі смарт-контрактами. Можна встановлювати різні типи запитів (GET, POST, PUT, DELETE) та надсилати їх разом із необхідними параметрами для тестування різних сценаріїв [14].

Postman з можливістю введення різних параметрів та даних дозволяє автоматизувати та спростити тестування різних функцій вашого API. Крім того, є можливість створити колекції запитів для різних сценаріїв взаємодії зі смарт - контрактами та зберігати їх для майбутнього використання та документування.

Загалом, використання Postman допомагає вдосконалити якість та надійність API для автоматизованої генерації та розгортання ERC-20 токенів шляхом забезпечення ефективного та зручного тестування та взаємодії з ним (Рис. 3.2. Використання Postman).

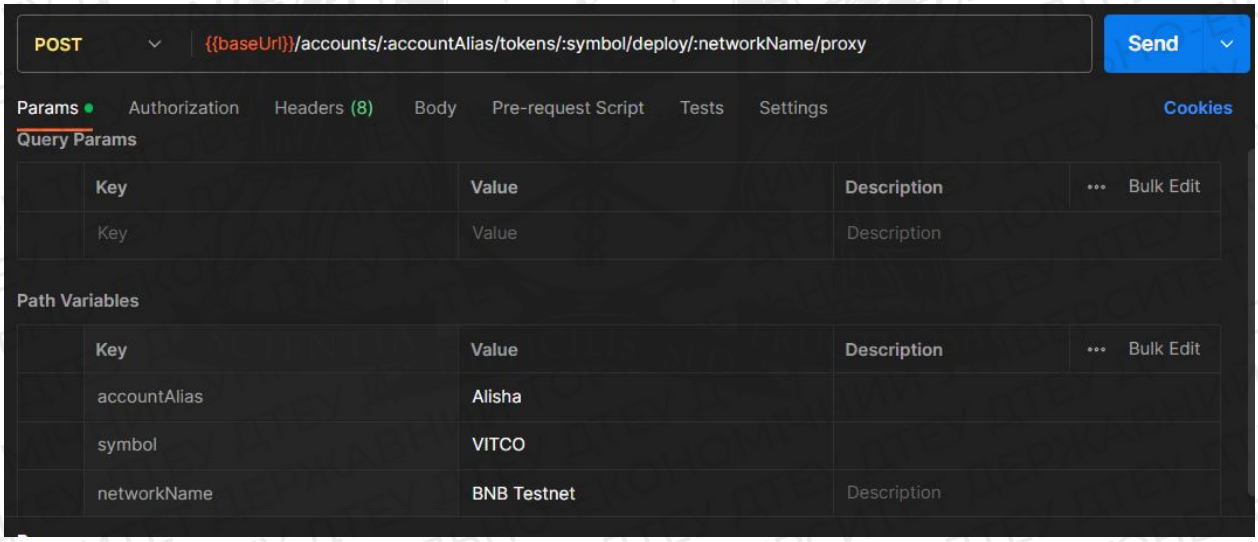

Рис. 3.2. Використання Postman

3.2.3.Adminer

Adminer це легкий інструмент для управління базами даних, який є корисним для проекту API для автоматизованої генерації та розгортання ERC-20 токенів.

Adminer надає інтерфейс для взаємодії з різними типами баз даних, включаючи SQL, що може бути важливим для збереження даних про токени

та їхні транзакції. Інструмент надає можливість виконувати SQL-запити, переглядати та редагувати дані в базі, створювати таблиці та здійснювати інші операції безпосередньо через веб-інтерфейс [15].

Для проєкту, який включає в себе розгортання смарт-контрактів та взаємодію з базою даних, Adminer є корисним інструментом для контролю та управління даними. Він дозволяє ефективно переглядати та адмініструвати дані, пов'язані з токенами ERC-20, що в свою чергу сприяє вдосконаленню якості та надійності системи (Рис. 3.3 Використання Adminer).

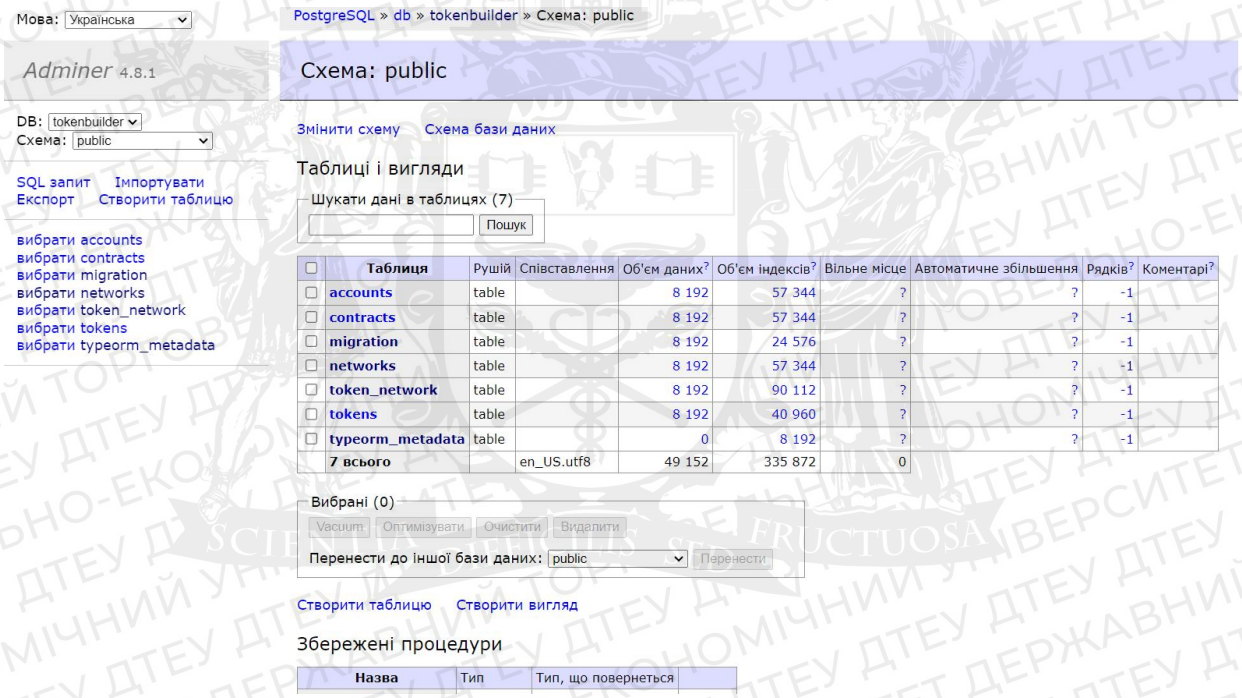

Рис. 3.3. Використання Adminer

3.2.4.MetaMask

Metamask це розширення для веб-браузерів, яке надає можливість взаємодіяти з блокчейном Ethereum та управляти Ethereum гаманцем безпосередньо з браузера. Для проекту API для автоматизованої генерації та розгортання ERC-20 токенів Metamask має важливу роль на рівні безпеки та зручності взаємодії зі створеними токенами.

Metamask дозволяє створювати, керувати та використовувати Ethereum гаманці прямо у браузері. Це особливо корисно для тестування та взаємодії зі смарт-контрактами на тестовій мережі, так як можна легко встановити та використовувати тестовий гаманець без необхідності використання реальних коштів [16].

Крім того, Metamask надає зручний спосіб підписувати транзакції, що є важливим для реалізації автоматизованої генерації та розгортання ERC-20 токенів. Можна використовувати Metamask для підпису транзакцій, які створені API під час розгортання нових токенів (Рис.3.3 Використання MetaMask).

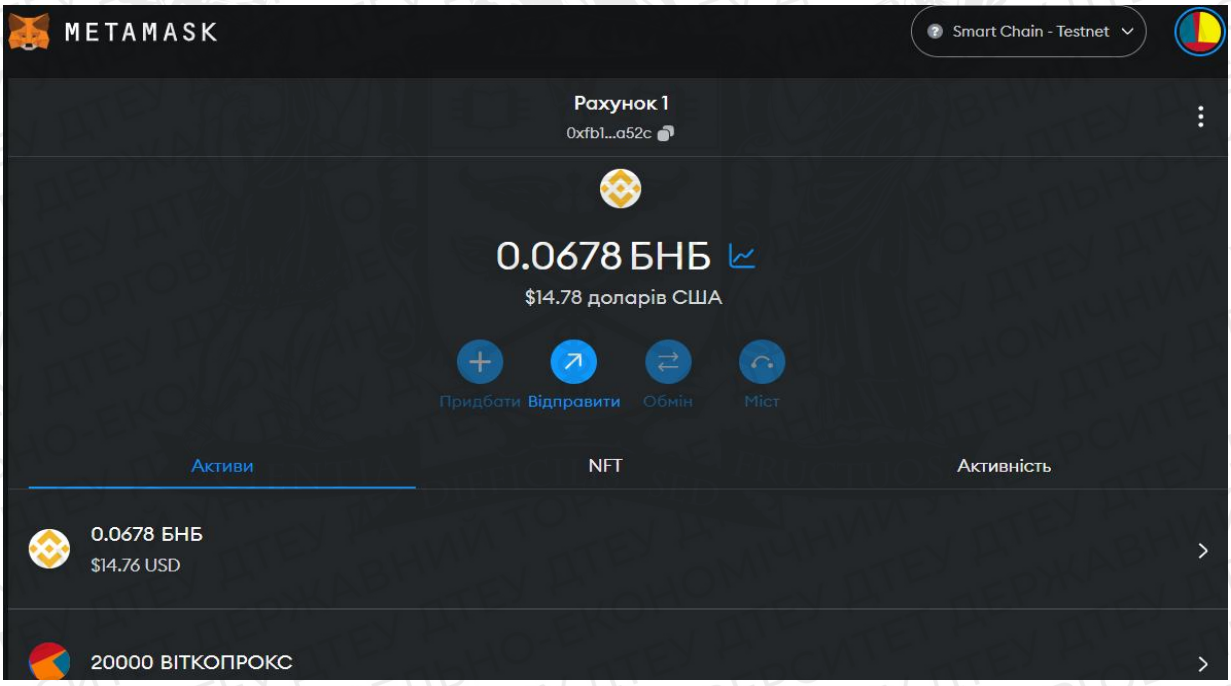

Рис. 3.4. Використання MetaMask

3.3. Платформа Node.js

Платформа Node.js дозволяє розробникам створювати швидкі та масштабовані додатки за допомогою JavaScript на серверному боці.

Однією з ключових переваг Node.js є його асинхронний підхід, що дозволяє ефективно обробляти багато одночасних запитів без блокування потоків. Це важливо для системи автоматизованої генерації та розгортання

токенів, оскільки можуть виникнути ситуації, коли необхідно одночасно обробляти багато запитів.

Node.js також має широкий вибір модулів та бібліотек, які полегшують розробку. Для взаємодії з Ethereum можна використовувати бібліотеку Web3.js, яка надає зручний інтерфейс для роботи з смарт-контрактами.

Крім того, Node.js має активну спільноту та багато ресурсів для вивчення, що робить його доступним та підтримує розробників на різних етапах проекту [17].

Загалом, Node.js допомагає розробникам ефективно реалізувати API для автоматизованої генерації та розгортання ERC-20 токенів, забезпечуючи високу продуктивність та зручність розробки.

3.4. Мова програмування Solidity

Мова програмування Solidity є ключовим інструментом при розробці API для автоматизованої генерації та розгортання ERC-20 токенів на платформі Ethereum. Вона спеціально розроблена для написання смарт-контрактів і володіє рядом властивостей, що роблять її ідеальним вибором для цього завдання [18].

Основні причини використання Solidity:

- Solidity дозволяє розробникам створювати смарт-контракти, які є основними компонентами токенів ERC-20. Вона надає необхідний функціонал для визначення поведінки та логіки токену, включаючи стандартні функції переказу, балансування та інші.
- Solidity використовується для реалізації стандарту ERC-20, який визначає стандартний інтерфейс для токенів на платформі Ethereum. Це дозволяє забезпечити сумісність та взаємодію з іншими токенами та додатками.

3.5. Бібліотеки та фреймворки

Для розробки API для автоматизованої генерації та розгортання ERC-20 токенів зручно використовувати бібліотеки Web3.js та Openzeppelin-contracts, що спрощують та прискорюють роботу з блокчейном Ethereum та смарт-контрактами.

Ці бібліотеки допомагають зосередитися на функціональності програми, спрощуючи складні аспекти взаємодії з Ethereum блокчейном та смарт-контрактами. Вони підтримують швидку та надійну розробку, що дозволить ефективно створити API для ERC-20 токенів

Також можна використати фреймворк NestJS, який спрощує та стандартизує процес розробки.

NestJS - це фреймворк для створення масштабованих та модульних веб-додатків на Node.js. NestJS надає вбудовану підтримку для створення API з використанням різних структур та архітектурних підходів.

3.6. Висновок до розділу 3

Інструменти для вирішення задачі, такі як Docker, дозволяють ізолювати, упаковувати та розгортати різні компоненти системи.

Додаткові інструменти, такі як Postman, допомагають тестувати та налагоджувати API, Adminer спрощує роботу з базою даних, а розширення MetaMask забезпечує взаємодію з Ethereum блокчейном. Платформа Node.js надає засіб для розробки серверної частини API з використанням JavaScript.

Мова програмування Solidity є необхідною для створення смарт-контрактів на блокчейні Ethereum.

Бібліотеки та фреймворки допомагають створити ефективний та функціональний код, що взаємодіє з Ethereum мережею та забезпечує структурованість розробки.

Загалом, правильний вибір та використання цих засобів сприяють розробці надійного, швидкого та функціонального API для автоматизованої генерації та розгортання ERC-20 токенів на блокчейні Ethereum.

РОЗДІЛ 4
ОПИС РЕАЛІЗАЦІЇ API

4.1. Цільова аудиторія API

Потенційними користувачами API для автоматизованої генерації та розгортання ERC-20 токенів можуть бути:

- Блокчейн-розробники, які працюють з блокчейном Ethereum та створюють власні токени на основі стандарту ERC-20, можуть використовувати це API для автоматизованої генерації токенів та їх розгортання на блокчейні.
- Компанії та стартапи, що планують випустити власні токени для реалізації програм лояльності або інвестицій, можуть використовувати це API для ефективної створення та управління своїми токенами.
- Фінансові установи, такі як банки, біржі або платіжні системи, можуть бачити користь у використанні цього API для забезпечення автоматизованого процесу емісії та управління токенами для своїх клієнтів.
- Децентралізовані додатки. Розробники децентралізованих додатків, які працюють на платформі Ethereum, можуть використовувати це API для легкого взаємодії з токенами та їх інтеграції у свої додатки.
- Інвестори та трейдери, що працюють з токенами на Ethereum, можуть бути зацікавлені в цьому API для зручного створення та управління своїми токенами.
- Екосистема блокчейн розробки. Інші розробники, інструменти, та сервіси у сфері блокчейн розробки можуть інтегрувати це API в свої продукти або використовувати його для тестування та експериментів.

- Фахівці з блокчейн технологій, які надають консультаційні послуги, можуть використовувати це API для допомоги своїм клієнтам у створенні та управлінні токенами.

4.2. Опис системи

API для автоматизованої генерації та розгортання ERC - 20 токенів має основний фасад створення токенів(Рис.4.1 Фасад логіки токенів(deployToken), Рис.4.2 Фасад логіки токенів(deployTokenProxy)).

```
export class TokenBuilderFacade {
  constructor(
    @Inject(TokenBuilderService)
    private readonly tokenBuilderService: TokenBuilderService,
    @Inject(forwardRef(() => TokensService))
    private readonly tokensService: TokensService,
    @Inject(NetworksService) private readonly networksService: NetworksService,
    @Inject(AccountsService) private readonly accountsService: AccountsService,
    @Inject(ContractsService)
    private readonly contractsService: ContractsService,
  ) {}
  public async deployToken(
    networkName: string,
    accountAlias: string,
    tokenSymbol: string,
  ) {
    const token = await this.tokensService.getAccountTokenBySymbol(
      accountAlias,
      tokenSymbol,
    );
    const alreadyDeployed = token.tokenNetworks.some(
      (tokenNetwork) => tokenNetwork.networkName === networkName,
    );
    if (alreadyDeployed) {
      throw new DeployError('Token already deployed');
    }
    const [network, account] = await Promise.all([
      this.networksService.getNetworkByName(networkName),
      this.accountsService.getAccountByAlias(accountAlias),
    ]);
    const tokenContract = await this.tokenBuilderService.deployToken(
      network,
      account,
    );
    return this.contractsService.createContract(tokenContract);
  }
```

Рис. 4.1. Фасад логіки токенів (deployToken)

```
  public async deployTokenProxy(
    networkName: string,
    accountAlias: string,
    tokenSymbol: string,
  ) {
    const [contracts, token, network, account] = await Promise.all([
      this.tokensService.getAccountTokenContracts(
        networkName,
        accountAlias,
        tokenSymbol,
      ),
      this.tokensService.getAccountTokenBySymbol(accountAlias, tokenSymbol),
      this.networksService.getNetworkByName(networkName),
      this.accountsService.getAccountByAlias(accountAlias),
    ]);
    if (contracts.proxyContract) {
      throw new DeployError('Proxy already deployed');
    }
    const proxyContract = await this.tokenBuilderService.deployProxyForToken(
      network,
      account,
      token,
      contracts.tokenContract,
    );
    return this.contractsService.createContract(proxyContract);
  }
}
```

Рис.4.2 Фасад логіки токенів (deployTokenProxy)

Систему можна розбити на декількох основних функціональних блоків:

1. Блок адміністрування системи:

Цей блок відповідає за адміністрування системи і дозволяє адміністраторам додавати та оновлювати мережі та акаунти в системі. Основні функції цього блоку включають можливість додавання, читання, оновлення та видалення (CRUD - Create, Read, Update, Delete) інформації про мережі та акаунти. Адміністратори можуть додавати нові мережі блокчейнів, вказувати їх параметри та додавати акаунти для керування токенами. Це забезпечує гнучкість і можливість розширення системи.

2. Бібліотека попередньо визначених смарт-контрактів:

Цей блок містить попередньо визначені смарт-контракти або шаблони, які можуть бути розгорнуті системою. Ці шаблони можуть містити заздалегідь визначену логіку та параметри, що дозволяє користувачам швидко розгортати стандартні токени ERC-20. Користувачі можуть вибирати із доступних шаблонів, налаштовувати їх параметри та розгортати смарт-контракти з вже визначеною функціональністю.

3. Інтеграція з блокчейн мережею Ethereum та розгортання контрактів

Цей функціонал забезпечує інтеграцію системи з блокчейн мережею Ethereum. Він надає можливість користувачам розгортати контракти на вибраній мережі. Користувачі можуть вибирати певну мережу Ethereum, вказувати параметри контракту та запускати процес розгортання. Це дозволяє легко та швидко створювати нові токени та інші смарт - контракти.

4. Блок для валідності операцій над токенами та бізнес логіки

Цей блок слідкує за виконанням бізнес критеріїв, необхідних для коректної роботи застосунку. Він перевіряє валідність операцій, що виконуються над токенами, такі як перекази, заморозки або інші маніпуляції. Також цей блок відповідає за виконання бізнес логіки, яка може бути вбудована в смарт - контракти. Наприклад, він може контролювати правила видачі токенів, обмеження на операції або реакції на певні події.

Ці основні функціональні блоки спільно працюють для створення, управління та взаємодії з ERC-20 токенами в автоматизований та ефективний спосіб.

4.3. Висновок до розділу 4

Детальний опис системи, пропозиції щодо використання та визначення цільової аудиторії становлять основу для успішного використання API у процесі генерації та розгортання токенів на блокчейні

Ethereum.

Цільова аудиторія API різноманітна і включає розробників, компанії та особи, які хочуть взаємодіяти з токенами на блокчейні Ethereum шляхом автоматизованої генерації та розгортання ERC-20 токенів.

Проведено детальний опис функціональності API, його можливостей для створення ERC-20 токенів. Висвітлено аспекти безпеки, моніторингу та управління процесами через API.

ВИСНОВКИ ТА ПРОПОЗИЦІЇ

У ході роботи над дипломним проектом було вивчено та описано процес розробки API для автоматизованої генерації та розгортання ERC-20 токенів на блокчейні Ethereum. Досліджена тема має велике практичне значення в контексті широкого застосування токенів ERC-20 у сучасних блокчейн-проектах, які вимагають ефективної та автоматизованої інфраструктури для створення, управління та розгортання токенів.

Розглянуто документацію для розробки API та створення структури інформаційних потоків. Встановлено, що використання Web3.js та OpenZeppelin при розробці API є ключовим для забезпечення взаємодії з блокчейном Ethereum. Вибір REST архітектури було обгрунтовано для підтримки стандартних підходів до взаємодії між компонентами. Важливість використання Swagger як інструменту для документації дозволила забезпечити якісну документацію та полегшити процес розробки та співпраці.

Досліджено різні засоби розробки, які використовуються для розробки, тестування та взаємодії з розроблюваним API. Описано Docker як інструмент для контейнеризації та розгортання, Postman для тестування API, Adminer для роботи з базою даних, MetaMask для взаємодії з Ethereum мережею. Також були вивчені платформа Node.js, мова програмування Solidity, бібліотеки та фреймворки, що надали комплексний набір інструментів для розробки.

Досліджена тема розробки API для автоматизованої генерації та розгортання ERC-20 токенів виявилася важливою та актуальною для

розвитку блокчейн - індустрії та децентралізованих додатків.Реалізація такого API дозволить розробникам ефективно взаємодіяти з блокчейном, створюючи, управляючи та розгортаючи свої ERC-20 токени з меншими зусиллями. Використання сучасних інструментів сприяє покращенню якості розробки та забезпечує стандартні підходи до взаємодії з блокчейном та базами даних.

У ході роботи було налагоджено і протестовано API, що продемонструвало правильність його функціонування. API виявилася ефективною і здатною вирішувати всі встановлені завдання.

Потенційними користувачами API є блокчейн - розробники, компанії та стартапи, фінансові установи, децентралізовані додатки, інвестори та трейдери, екосистема блокчейн розробки та блокчейн - консультанти.

В цілому, розробка API для автоматизованої генерації та розгортання ERC-20 токенів є важливим кроком у напрямку полегшення та прискорення процесу розробки блокчейн - додатків. Використання сучасних технологій та інструментів дозволяє забезпечити високу якість розробки, зручний інтерфейс для розробників та підвищену безпеку операцій. Інтеграція API для автоматизованої генерації ERC-20 токенів є перспективною технологією, але вимагає комплексного підходу до керування ризиками та постійного контролю за безпекою та доступністю для забезпечення успішного впровадження.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What Are ERC-20 Tokens on the Ethereum Network? \\ URL: https://www.investopedia.com/news/what-erc20-and-what-does-it-mean-ethereum/
2. ERC-20 token standard. \\ URL: https://ethereum.org/en/developers/docs/standards/tokens/erc-20/
3. Web3.js - Ethereum JavaScript API \\ URL: https://web3js.readthedocs.io/en/v1.10.0/
4. ERC - 20 \\ URL: https://docs.openzeppelin.com/contracts/2.x/api/token/erc20
5. REST API Tutorial \\ URL: https://restfulapi.net/
6. The REST Architecture \\ URL: https://www.baeldung.com/cs/rest-architecture
7. Swagger \\ URL: https://www.techtarget.com/searchapparchitecture/definition/Swagger
8. How to Create an ERC20 Token the Simple Way \\ URL: https://www.toptal.com/ethereum/create-erc20-token-tutorial
9. Exploring the Ultimate ERC20 Token API \\ URL: https://moralis.io/exploring-the-ultimate-erc20-token-api/
10. What Is an API Proxy? \\ URL: https://heynode.com/tutorial/what-api-proxy/
11. API Proxy vs API Gateway: What Are The Differences And Which Should You Use? \\ URL: https://www.moesif.com/blog/technical/api-gateways/API-Proxy-Vs-API-Gateway-What-Are-The-Differences-And-Which-Should-You-Use/
12. What is Visual Studio Code? \\ URL: https://www.educative.io/answers/what-is-visual-studio-code

13. Що таке Docker і навіщо він? \\ URL: https://qagroup.com.ua/publications/shcho-take-docker-i-navishcho-vin/
14. Postman - must have для QA \\ URL: https://qagroup.com.ua/publications/znajomtes-postman-must-have-dlia-qa/
15. Play databases with Adminer and Docker \\ URL: https://medium.com/@etiennerouzeaud/play-databases-with-adminer-and-docker-53dc7789f35f
16. What is MetaMask? How to Use the Top Ethereum Wallet \\ URL: https://decrypt.co/resources/metamask
17. Node.js \\ URL: https://www.tutorialsteacher.com/nodejs/what-is-nodejs
18. Introduction to Smart Contracts \\ URL: https://docs.soliditylang.org/en/v0.8.20/introduction-to-smart-contracts.html

ТЕХНІЧНЕ ЗАВДАННЯ

Назва проекту: Розробка API для автоматизованої генерації та розгортання ERC-20 токенів

Загальний опис:

Проект передбачає створення комплексної системи API, яка дозволить автоматизовано генерувати та розгортати токени стандарту ERC-20 на блокчейні Ethereum.

Функціональні вимоги:

1. Генерація токенів:

- Забезпечити можливість генерування ERC-20 токенів з встановленою кількістю та параметрами (назва, символ, десяткові знаки тощо).

2. Розгортання токенів:

- Дозволити автоматичне розгортання згенерованих токенів на блокчейні Ethereum.
- Забезпечити можливість вказати адресу власника токенів.
- Додати можливість вибору мережі для розгортання токену.

3. Документація та інтерфейс:

Надати зручний та інтуїтивно зрозумілий інтерфейс для розробників для взаємодії з API.

Технічні вимоги:

1. Мова програмування:

- Використання мови програмування з підтримкою блокчейн розробки, такої як JavaScript (Node.js).

					ДТЕУ 121 02з-5.МР		
Зм.	*Аркуш*	*№ докум.*	*Підпис*	*Дата*			
Зав. каф.		Криворучко О.В.		15.03.23	*API для автоматизованої генерації та розгортання ERC - 20 токенів*	*Стадія*	*Аркуш*	*Аркушів*
Керівник		Жирова Т. О.		15.03.23		*ТЗ*	*44*	*43*
Гарант		Котенко Н.О.		15.03.23		*Факультет інформаційних технологій 2мз курс, 2 група*		
Розробив		Задорожна А. В.		15.03.23	*Технічне завдання*			

2. Технології та бібліотеки:

- Використання Web3.js та OpenZeppelin для взаємодії з блокчейном та створення смарт-контрактів.
- Використання Docker для контейнеризації та розгортання системи.

3. Документація:

- Докладна документація API з використанням Swagger або аналогічного інструменту.

4. Безпека:

- Забезпечити шифрування конфіденційних даних криптоакаунта на випадок компроментації системної бази даних.

Вимоги прийняття системи:

- Система має бути протестованою.
- Система повинна бути зручною у використанні та зменшувати зусилля, необхідні для створення та управління токенами.

Очікуваний результат:

Створення повноцінного та функціонального API для автоматизованої генерації, розгортання та управління ERC-20 токенами на блокчейні Ethereum. Документація та інтерфейс повинні бути зрозумілими та зручними для розробників

ПРОГРАМА ТА МЕТОДИКА ТЕСТУВАННЯ

Задля економії ресурсів часу API протестовано мануальним метод.

Мануальне тестування API для автоматизованої генерації та розгортання ERC-20 токенів має деякі переваги, особливо в певних ситуаціях та етапах розробки.

На ранніх етапах розробки, коли система ще не стабільна та може зазнавати змін, мануальне тестування дозволяє швидко перевірити базовий функціонал та визначити основні проблеми.

Мануальне тестування може допомогти виявити неочікувані проблеми або аномалії в роботі системи, які можуть бути пропущені автоматизованими тестами.

Коли система зазнає частих змін або знаходиться в активній розробці, мануальне тестування дозволяє оперативно адаптувати тестові сценарії до нових умов.

Тестування допомагає впевнитися, що генеровані токени та виконані операції є унікальними та відповідають вимогам бізнесу.

Система взаємодіє з іншими засобами розробки, такими як Docker, Postman, Adminer. Проведене тестування взаємодії показало, що інтеграція працює без помилок та збоїв (Див. розділ 3, Рис.3.1, розділ 3 Рис.3.2 та розділ 3 Рис.3.3).

Процес тестування API для автоматизованої генерації та розгортання ERC-20 токенів з використанням MetaMask:

- Підключення MetaMask до тестової мережі;

- Створення нового гаманця на тестовій мережі Smart Chain - Testnet;
- Отримання тестових Ether на створений гаманець (Рис.1 Отримання тестових ефірів).

Це необхідно для виконання транзакцій в мережі.

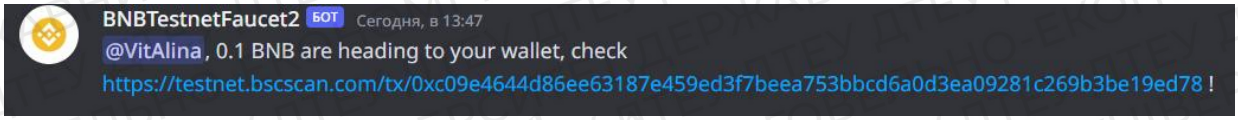

Рис. 1. Отримання тестових ефірів

- Виконання запитів до API для автоматизованої генерації та розгортання ERC-20 токенів. Це може включати створення нових токенів, налаштування параметрів та інші операції (Рис.3 Виконання запитів).

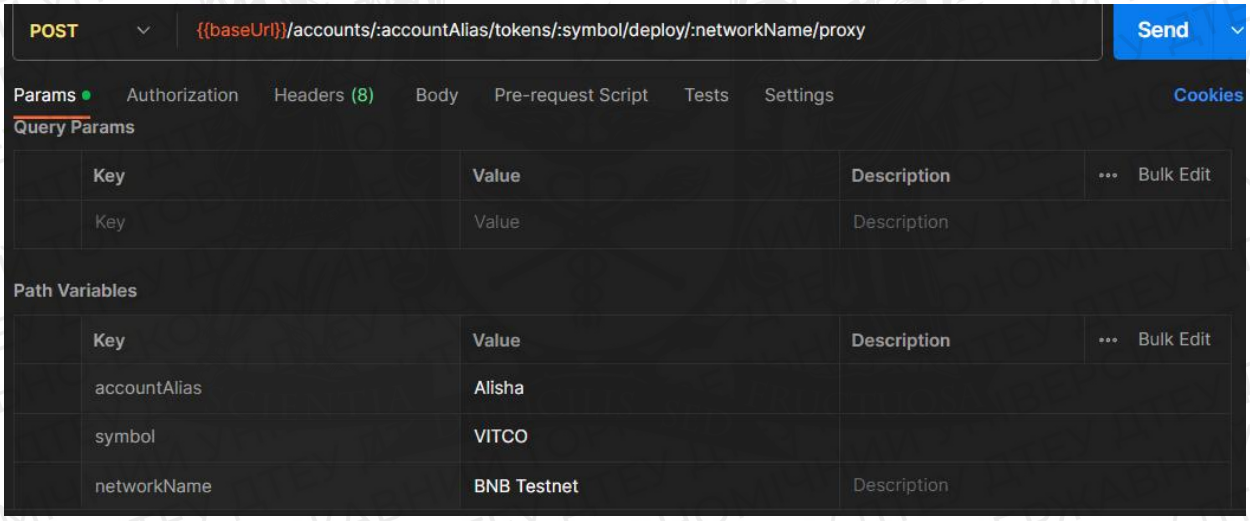

Рис. 2. Виконання запитів

- Генерація необхідних даних для підпису транзакцій, такі як адреса одержувача, кількість токенів тощо.
- Перевірка статусу транзакцій та результатів взаємодії з API(Рис.3 Результат транзакції, Рис.4 Результат взаємодії з API).

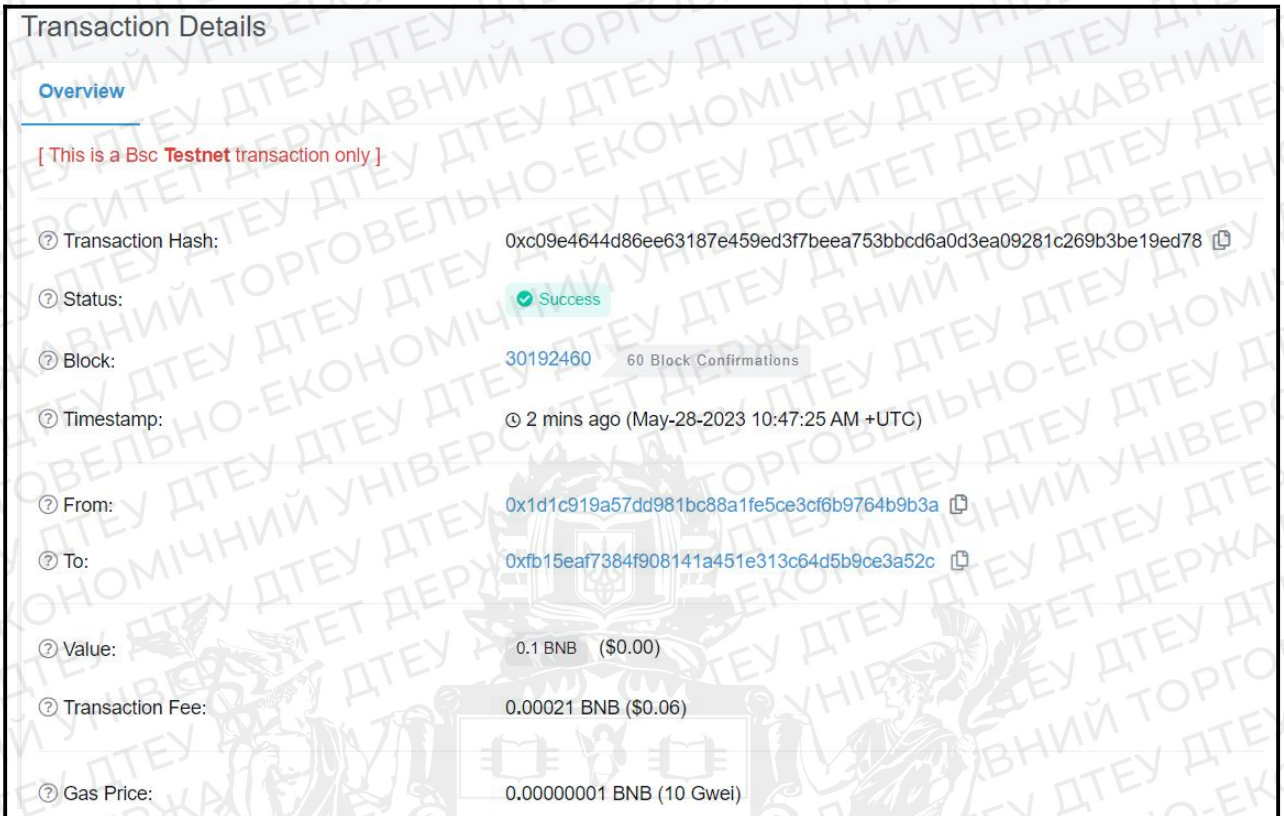

Рис. 3. Результат транзакції

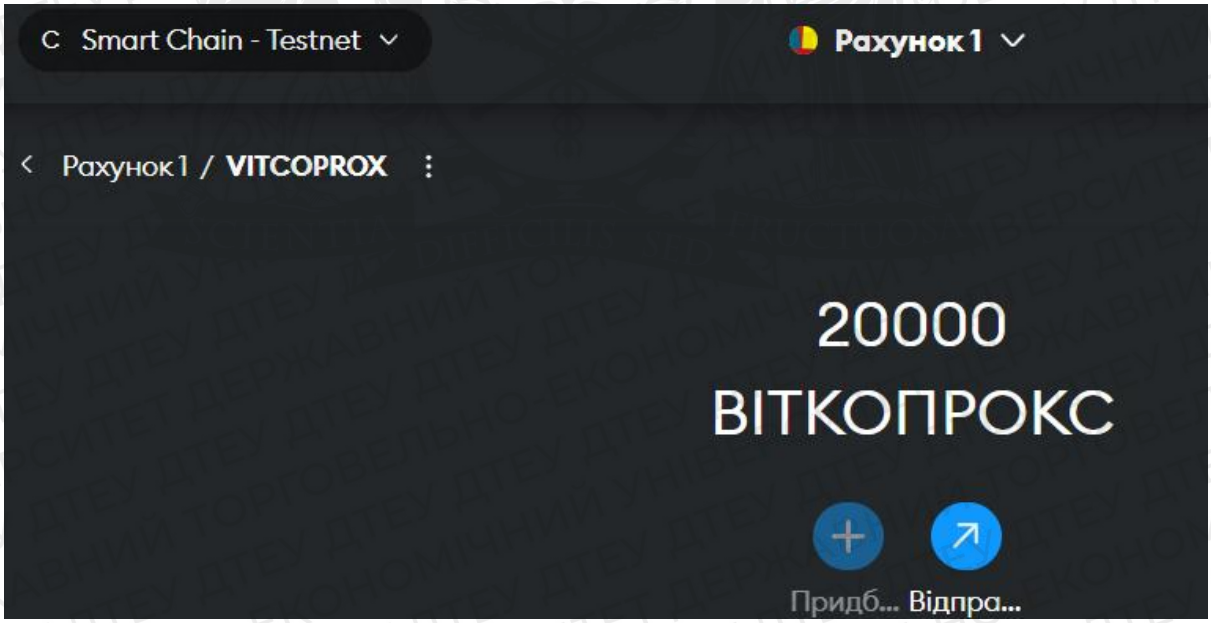

Рис. 4. Результат взаємодії з API

Під час тестування всі виявлені недоліки були усунені. Проведено тестування виправлень, яке підтвердило правильну роботу системи. Це свідчить про це, що API для автоматизованої генерації та розгортання ERC-20 токенів працює коректно і готовий до використання.

ДОДАТКИ

Додаток А

Лістинг програмного коду

Main.ts

```
import { ValidationPipe } from '@nestjs/common';
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';
import { HttpAdapterHost, NestFactory } from '@nestjs/core';
import { AllExceptionsFilter } from './all-exceptions.filter';
import { AppModule } from './app.module';
import * as path from 'path';
import { writeFile } from 'fs/promises';
import { configService } from 'src/config/config.service';

async function bootstrap() {
  const app = await NestFactory.create(AppModule, {
    logger: ['log', 'error', 'warn', 'debug', 'verbose'],
  });

  const httpAdapterHost = app.get(HttpAdapterHost);
  app.useGlobalFilters(new AllExceptionsFilter(httpAdapterHost));
  app.useGlobalPipes(new ValidationPipe({ transform: true, whitelist: true }));

  if (!configService.isProduction()) swaggerGenerate(app);

  const port = configService.getPort();
  await app.listen(port);
}

function swaggerGenerate(app) {
  const config = new DocumentBuilder()
    .setTitle('Fantoken token builder')
    .setDescription(
      `In order to create a token, you must: \n
    1. If you don't have an account, you can create a new one - POST /accounts
    2. Create a token - POST /tokens
    3. If you don't have a network, you can create a new one - POST /networks
    4. Deploy a token - POST /accounts/{accountAlias}/tokens/{symbol}/deploy/{networkName}
    5. Deploy a proxy - POST
/accounts/{accountAlias}/tokens/{symbol}/deploy/{networkName}/proxy`,
    )
    .setVersion('1.0')
    .build();

  const document = SwaggerModule.createDocument(app, config, {
    deepScanRoutes: true,
  });
  const outputPath = path.resolve(process.cwd(), 'swagger.json');
```

```
  writeFile(outputPath, JSON.stringify(document, null, 4), {
    encoding: 'utf8',
  });
}

bootstrap();
```

App.module.ts

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { AccountsModule } from './accounts/accounts.module';
import { TypeOrmModule } from '@nestjs/typeorm';
import { configService } from './config/config.service';
import { NetworksModule } from './networks/networks.module';
import { TokensModule } from './tokens/tokens.module';
import { ContractsModule } from './contracts/contracts.module';
import { APP_FILTER } from '@nestjs/core';
import { AllExceptionsFilter } from './all-exceptions.filter';

@Module({
  imports: [
    TypeOrmModule.forRoot(configService.getTypeOrmConfig()),
    AccountsModule,
    NetworksModule,
    TokensModule,
    ContractsModule,
  ],
  controllers: [AppController],
  providers: [
    AppService,
    {
      provide: APP_FILTER,
      useClass: AllExceptionsFilter,
    },
  ],
})
export class AppModule {}
```

Token.service.ts

```
import { forwardRef, Inject, Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { DeleteResult, Repository } from 'typeorm';
import { TokenBuilderFacade } from '../token-builder/token-builder.facade';
import { CreateTokenDto } from './dto/create-token.dto';
import { Token } from './entities/token.entity';
import { TokenNetwork } from './entities/tokenNetwork.entity';

@Injectable()
export class TokensService {
  constructor(
    @InjectRepository(Token)
```

```
    private readonly tokenRepository: Repository<Token>,
    @InjectRepository(TokenNetwork)
    private readonly tokenNetworkRepository: Repository<TokenNetwork>,
    @Inject(forwardRef(() => TokenBuilderFacade))
    private readonly tokenBuilderFacade: TokenBuilderFacade,
  ) {}

  public getTokensList() {
    return this.tokenRepository.find();
  }

  public getAccountTokens(accountAlias: string) {
    const account = { alias: accountAlias };
    return this.tokenRepository.find({ account });
  }

  public getAccountTokenBySymbol(accountAlias: string, symbol: string) {
    const account = { alias: accountAlias };
    return this.tokenRepository.findOneOrFail({
      where: { account, symbol },
      relations: [
        'tokenNetworks',
        'tokenNetworks.proxyContract',
        'tokenNetworks.tokenContract',
      ],
    });
  }

  public getTokenBySymbol(symbol: string) {
    return this.tokenRepository.findOneOrFail({ symbol });
  }

  public async createToken(tokenDTO: CreateTokenDto): Promise<Token> {
    const token = this.tokenRepository.create(tokenDTO);
    await this.tokenRepository.insert(token);
    return token;
  }

  public deleteTokenBySymbol(symbol: string): Promise<DeleteResult> {
    return this.tokenRepository.delete({ symbol });
  }

  public deleteAccountTokenBySymbol(accountAlias: string, symbol: string) {
    const account = { alias: accountAlias };
    return this.tokenRepository.delete({ account, symbol });
  }
  public getAccountTokenContracts(
    networkName: string,
    accountAlias: string,
    symbol: string,
  ) {
    return this.tokenNetworkRepository.findOneOrFail({
```

```
      where: {
        token: { accountAlias, symbol },
        networkName,
      },
      relations: ['proxyContract', 'tokenContract'],
    });
  }

  public async deployAccountToken(
    networkName: string,
    accountAlias: string,
    symbol: string,
  ) {
    const tokenContract = await this.tokenBuilderFacade.deployToken(
      networkName,
      accountAlias,
      symbol,
    );
    await this.tokenNetworkRepository.insert({
      tokenSymbol: symbol,
      networkName,
      tokenContractId: tokenContract.id,
    });
    return tokenContract;
  }

  public async deployAccountTokenProxy(
    networkName: string,
    accountAlias: string,
    symbol: string,
  ) {
    const proxyContract = await this.tokenBuilderFacade.deployTokenProxy(
      networkName,
      accountAlias,
      symbol,
    );
    await this.tokenNetworkRepository.update(
      {
        tokenSymbol: symbol,
        networkName,
      },
      {
        proxyContract,
      },
    );
    return proxyContract;
  }
}
```

Token.module.ts

```
import { forwardRef, Module } from '@nestjs/common';
import { TokensService } from './tokens.service';
```

```
import { TokensController } from './tokens.controller';
import { TypeOrmModule } from '@nestjs/typeorm';
import { Token } from './entities/token.entity';
import { TokenBuilderModule } from '../token-builder/token-builder.module';
import { TokenNetwork } from './entities/tokenNetwork.entity';

@Module({
  controllers: [TokensController],
  providers: [TokensService],
  imports: [
    TypeOrmModule.forFeature([Token, TokenNetwork]),
    forwardRef(() => TokenBuilderModule),
  ],
  exports: [TokensService],
})
export class TokensModule {}
```

Token.controller.ts

```
import {
  Controller,
  Get,
  Post,
  Body,
  Param,
  Delete,
  HttpException,
  HttpStatus,
} from '@nestjs/common';
import { TokensService } from './tokens.service';
import { CreateTokenDto } from './dto/create-token.dto';
import { ApiParam } from '@nestjs/swagger';
import { DeployError } from '../token-builder/token-builder.service';

@Controller('accounts/:accountAlias/tokens')
export class TokensController {
  constructor(private readonly tokenService: TokensService) {}

  @Get()
  public getAccountTokens(@Param('accountAlias') alias: string) {
    return this.tokenService.getAccountTokens(alias);
  }

  @Get(':symbol')
  public getAccountToken(
    @Param('accountAlias') alias: string,
    @Param('symbol') symbol: string,
  ) {
    return this.tokenService.getAccountTokenBySymbol(alias, symbol);
  }

  @ApiParam({
    name: 'accountAlias',
```

```
  type: String,
  description: 'An alias of user account',
  example: 'thebigboss',
 })
 @Post()
 public createToken(
  @Param('accountAlias') alias: string,
  @Body() createTokenDto: CreateTokenDto,
 ) {
  return this.tokenService.createToken({
   ...createTokenDto,
   accountAlias: alias,
  });
 }

 @Delete(':symbol')
 public async deleteToken(
  @Param('accountAlias') alias: string,
  @Param('symbol') symbol: string,
 ): Promise<void> {
  await this.tokenService.deleteAccountTokenBySymbol(alias, symbol);
 }

 @Post(':symbol/deploy/:networkName')
 public async deployToken(
  @Param('accountAlias') alias: string,
  @Param('symbol') symbol: string,
  @Param('networkName') networkName: string,
 ) {
  return this.tokenService
   .deployAccountToken(networkName, alias, symbol)
   .catch((err: Error) => {
    if (err instanceof DeployError) {
     throw new HttpException(err.message, HttpStatus.UNPROCESSABLE_ENTITY);
    }
    throw err;
   });
 }

 @Post(':symbol/deploy/:networkName/proxy')
 public async deployProxy(
  @Param('accountAlias') alias: string,
  @Param('symbol') symbol: string,
  @Param('networkName') networkName: string,
 ) {
  return this.tokenService
   .deployAccountTokenProxy(networkName, alias, symbol)
   .catch((err: Error) => {
    if (err instanceof DeployError) {
     throw new HttpException(err.message, HttpStatus.UNPROCESSABLE_ENTITY);
    }
    throw err;
```

```
    });
  }
}
```

Token-builder.service.ts

```
import { Injectable } from '@nestjs/common';
import * as fs from 'fs/promises';
import * as path from 'path';
import Web3 from 'web3';
import { TransactionReceipt } from 'web3-core';
import { AbiItem, toWei } from 'web3-utils';
import { NetworkDTO } from './dto/network.dto';
import { AccountDTO } from './dto/account.dto';
import { TokenDTO } from './dto/token.dto';
import { ContractDTO } from './dto/contract.dto';

const ARTIFACTS_PATH = path.join(__dirname, 'contracts', 'artifacts');

export class DeployError extends Error {
  constructor(message: string) {
    super(`Deploy Error: ${message}`);
  }
}

export enum ContractName {
  Proxy = 'ERC1967Proxy',
  Token = 'TokenTemplateERC20',
}

export interface ContractData {
  abi: AbiItem | AbiItem[];
  bytecode: string;
}

@Injectable()
export class TokenBuilderService {
  private readonly web3: Web3;
  constructor() {
    this.web3 = new Web3();
  }

  private async loadContract(
    contractName: ContractName,
  ): Promise<ContractData> {
    const artifact = await fs.readFile(
      path.join(ARTIFACTS_PATH, `${contractName}.json`),
      'utf8',
    );
    const contractFile = JSON.parse(artifact);
    return {
      abi: contractFile.abi,
      bytecode: contractFile.data.bytecode.object,
```

```
  };
}

private prepareContractToDeploy(
  contractData: ContractData,
  constructorArgs: unknown[],
): string {
  const contract = new this.web3.eth.Contract(contractData.abi);
  return contract
    .deploy({
      data: contractData.bytecode,
      arguments: constructorArgs,
    })
    .encodeABI();
}

private encodeContractFunctionCall(
  contractData: ContractData,
  functionName: string,
  args: unknown[],
) {
  const fn = [contractData.abi]
    .flat()
    .find(({ name }) => name === functionName);
  return this.web3.eth.abi.encodeFunctionCall(
    fn,
    args.map((el) => el.toString()),
  );
}

private async deploy(
  network: NetworkDTO,
  account: AccountDTO,
  encodedABI: string,
): Promise<TransactionReceipt> {
  const web3 = new Web3([network.url, network.apiKey].join('/'));
  const gas = await web3.eth.estimateGas({
    from: account.address,
    data: encodedABI,
  });
  const createTransaction = await web3.eth.accounts.signTransaction(
    {
      from: account.address,
      data: encodedABI,
      gas,
    },
    account.privateKey,
  );
  const receipt = await web3.eth.sendSignedTransaction(
    createTransaction.rawTransaction,
  );
  return receipt;
```

```
}

private async deployContract(
  network: NetworkDTO,
  account: AccountDTO,
  contractName: ContractName,
  constructorArgs: unknown[],
): Promise<ContractDTO> {
  const contractData = await this.loadContract(contractName);
  const deployContractABI = this.prepareContractToDeploy(
    contractData,
    constructorArgs,
  );
  return this.deploy(network, account, deployContractABI).then((receipt) => {
    if (!receipt.status) throw Error('Transaction rejected');
    return {
      address: receipt.contractAddress,
      transactionHash: receipt.transactionHash,
    };
  });
}

public async deployToken(
  network: NetworkDTO,
  account: AccountDTO,
): Promise<ContractDTO> {
  return this.deployContract(network, account, ContractName.Token, []);
}

public async deployProxyForToken(
  network: NetworkDTO,
  account: AccountDTO,
  token: TokenDTO,
  contract: ContractDTO,
): Promise<ContractDTO> {
  const { name, symbol, premint } = token;
  const { address = '' } = contract;
  if (!address)
    throw new DeployError(
      'Proxies cannot be deployed for an undeployed contract',
    );
  const tokenContract = await this.loadContract(ContractName.Token);
  const data = this.encodeContractFunctionCall(tokenContract, 'initialize', [
    name,
    symbol,
    toWei(premint.toString()),
  ]);
  return this.deployContract(network, account, ContractName.Proxy, [
    address,
    data,
  ]);
}
```

```
}
```

Token-builder.module.ts

```
import { forwardRef, Module } from '@nestjs/common';
import { AccountsModule } from '../accounts/accounts.module';
import { ContractsModule } from '../contracts/contracts.module';
import { NetworksModule } from '../networks/networks.module';
import { TokensModule } from '../tokens/tokens.module';
import { TokenBuilderFacade } from './token-builder.facade';
import { TokenBuilderService } from './token-builder.service';

@Module({
 providers: [TokenBuilderService, TokenBuilderFacade],
 imports: [
  ContractsModule,
  forwardRef(() => TokensModule),
  NetworksModule,
  AccountsModule,
 ],
 exports: [TokenBuilderFacade],
})
export class TokenBuilderModule {}
```

Token-builder.facade.ts

```
import { forwardRef, Inject, Injectable } from '@nestjs/common';
import { AccountsService } from '../accounts/accounts.service';
import { ContractsService } from '../contracts/contracts.service';
import { NetworksService } from '../networks/networks.service';
import { TokensService } from '../tokens/tokens.service';
import { DeployError, TokenBuilderService } from './token-builder.service';

@Injectable()
export class TokenBuilderFacade {
 constructor(
  @Inject(TokenBuilderService)
  private readonly tokenBuilderService: TokenBuilderService,
  @Inject(forwardRef(() => TokensService))
  private readonly tokensService: TokensService,
  @Inject(NetworksService) private readonly networksService: NetworksService,
  @Inject(AccountsService) private readonly accountsService: AccountsService,
  @Inject(ContractsService)
  private readonly contractsService: ContractsService,
 ) { }
 public async deployToken(
  networkName: string,
  accountAlias: string,
  tokenSymbol: string,
 ) {
  const token = await this.tokensService.getAccountTokenBySymbol(
   accountAlias,
   tokenSymbol,
  );
```

```
    const alreadyDeployed = token.tokenNetworks.some(
      (tokenNetwork) => tokenNetwork.networkName === networkName,
    );
    if (alreadyDeployed) {
      throw new DeployError('Token already deployed');
    }
    const [network, account] = await Promise.all([
      this.networksService.getNetworkByName(networkName),
      this.accountsService.getAccountByAlias(accountAlias),
    ]);
    const tokenContract = await this.tokenBuilderService.deployToken(
      network,
      account,
    );
    return this.contractsService.createContract(tokenContract);
  }
  public async deployTokenProxy(
    networkName: string,
    accountAlias: string,
    tokenSymbol: string,
  ) {
    const [contracts, token, network, account] = await Promise.all([
      this.tokensService.getAccountTokenContracts(
        networkName,
        accountAlias,
        tokenSymbol,
      ),
      this.tokensService.getAccountTokenBySymbol(accountAlias, tokenSymbol),
      this.networksService.getNetworkByName(networkName),
      this.accountsService.getAccountByAlias(accountAlias),
    ]);
    if (contracts.proxyContract) {
      throw new DeployError('Proxy already deployed');
    }
    const proxyContract = await this.tokenBuilderService.deployProxyForToken(
      network,
      account,
      token,
      contracts.tokenContract,
    );
    return this.contractsService.createContract(proxyContract);
  }
}
```