Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
«КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мультимедійних систем факультету інформатики

Електрона залікова книжка студента на основі технології Blockchain

Текстова частина до курсової роботи
за спеціальністю „Інженерія програмного забезпечення” 6.040302

Керівник курсової роботи
асистент
Кирило Гороховський

“____” _____ 2021 р.

Виконав студент ФІ-3
Поліщук Ю. О.

“____” _____ 2021 р.

Анотація

У даній роботі розглянути основні принципи роботи Blockchain і фреймворку Hyperledger Fabric та описаний процес розробки електронної залікової книжки студента на основі технології блокчейн.

Вступ

Важко назвати технологію, яка викликала більшу кількість суперечок в колах IT-спеціалістів, аніж блокчейн. Багато хто вважає його появу найреволюційнішою подією 21 століття. Величезна кількість стартапів була створена з метою втілення даної технології у наше повсякденне життя. Прихильники блокчейну стверджують, що він дозволить перенести звітність та фінансові ринки на якісно новий рівень.

Проте не менша кількість людей не згодна з даним твердженням і вважають євангелістів блокчейну шарлатанами. І дійсно не зважаючи на запевнення прихильників, досі важко знайти справжні застосування даної технології поза межами криптовалют.

Так що ж таке блокчейн, і в чому його особливості в порівнянні з традиційними методами збереження даних? У даній роботі я хочу розглянути основні особливості його роботи та створити власний проект на його основі. Даний проект полягає у створенні електронної залікової книжки студента. У теорії це унеможливить ймовірність фальсифікації оцінок та дозволить кожному упевнитись, що така оцінка дійсно існує.

Зміст

1. Основні поняття технології Blockchain

1.1 Розподілена книга обліку

Blockchain (англ. block – блок, chain – ланцюг) – це розподілена база даних, яка являє собою ланцюжок блоків з даними, де кожен блок залежить від попереднього. Даний ланцюг зберігає в собі всі дані які колись були у ньому записані. При оновленні певної інформації ми не перезаписуємо дані в блоці, а додаємо новий блок з актуальними даними.

У багато чому блокчейн нагадує цифрову книгу бухгалтерського обліку чи реєстр. Для прикладу давайте уявимо як виглядатиме реєстр оцінок для студента.

Рисунок 1.1.1

З рисунку 1.1 ми можемо відповісти одразу на декілька запитань: з яких предметів виставленні оцінки (ООП, Історія, JavaEE), в якому році студент вивчав історію (2020), скільки він отримав з ООП (92 бали). Як ми бачимо це дуже зручний і простий у роботі запис. Проте хто може засвідчити, що ці дані правдиві? Будь-хто може змінити інформацію і ми не помітимо різниці.

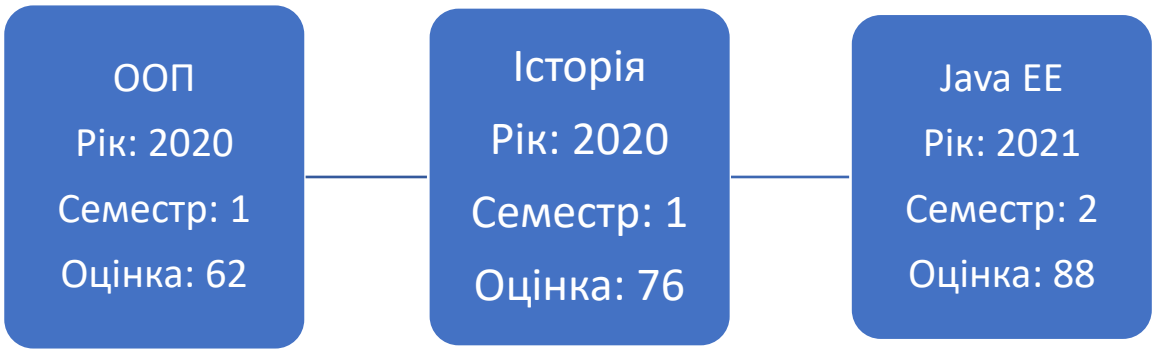

Рисунок 1.1.2

Для того щоб запобігти цьому в повсякденному житті, ми делегуємо одній особі чи групам осіб (у даному випадку державі) право на введення єдиної копії реєстру. При цьому суб'єкт, який відповідає за реєстр, повинен забезпечити належний рівень безпеки даного документу від сторонніх осіб. Усі записи зазвичай засвідчуються підписом, щоб знати хто відповідальний за дану зміну даних.

Дана система базується на нашій довірі обліковцю в тому, що він не зловживатиме своїми правами. Звісно він не зможе переписати весь реєстр, проте досить часто спливають історії про відібране майно за допомогою "необережних" нотаріусів[1].

Технологія blockchain обіцяє вирішити проблему довіри. Кожний має право тримати власну копію ланцюжка і при цьому дані у ньому будуть правдивими.

Надалі ми розглянемо головні принципи, які це забезпечують.

1.2 **Хешування**

Хешування є одним з найважливішим компонентів будь-якого блокчейну. Функція хешування приймає на вхід дані будь-якого розміру, а повертає рядок фіксованої довжини, який називається хешом. При цьому важливо зазначити що вона повертає завжди один і

той самий результат для одних і тих же даних. Хеші часто використовують для ідентифікації об’єктів.

Найпростіша хеш-функція – остача від ділення.

$$3\ mod\ 5 = 3$$

$$7\ mod\ 5 = 2$$

$$12\ mod\ 5 = 2$$

$$100\ mod\ 5 = 0$$

Як бачимо, вона приймає на вхід числа різного діапазону, а повертає число з фіксованого [0;5). Також ми бачимо що функція повертає теж саме значення у двох випадках – 7 і 12. Це називається колізією, і бажано аби вони виникали якомога рідше, тому хеш-функції зазвичай є складнішими ніж зазначена вище.

Також потрібно зазначити особливість, яка властива деяким функціям – розрахування хешу повинно бути простим, а відновлення інформації з нього дуже складним, або неможливим. Такі хеш-функції називаються криптографічними і саме їх ми використовуємо в блокчейні[2].

1.3 **Хеш-чейн**

Давайте змінимо наш реєстр, додавши до кожного запису хеш попереднього запису.

Рисунок 1.3.1

Перший блок не має посилання на попередника. Такий блок називають **генезисом**.

Дана структура називається **хеш-чейном**.

Тепер наші записи містять інформацію про стан в якому перебував ланцюжок при записі. Давайте змінимо інформацію у першому блоці.

Рисунок 1.3.2

Змінивши дані у блоці, ми змінили результат хеш-функції. Тепер хеш записаний у наступному блоці не співпадає зі справжнім хешом. Отже всі блоки розміщенні після першого відтепер недійсні.

Звісно, на даному етапі цю проблему досить легко вирішити. Досить просто створити нові блоки з перерахованими хешами.

Рисунок 1.3.3

Нам потрібно запобігти фальсифікації блоків.

1.4 **Цифровий підпис**

Шифрування з відкритим ключем дещо схоже на хешування. Ми створюємо пару ключів публічний та приватний. Ми зашифровуємо дані за допомогою функції шифрування та публічного ключа. Тепер, щоб доступитися до даних нам потрібно розшифрувати, але вже використовуючи приватний ключ. Таким чином хоча і зашифровані дані і публічний ключ можуть бути доступним усім, ніхто все одно не зможе прочитати дані.

Проте, насправді, це можна використати і в інший бік. Давайте зашифруємо хеш блоку нашим приватним ключем. Це буде нашим цифровим підписом. Відтепер щоб упевнитися що саме ми підписали цей блок досить захешувати блок самостійно, розшифрувати підпис за допомогою публічного ключа та порівняти результати. Якщо вони співпали, то блок дійсно підписаний нами.[2], [3]

Додамо до наших блоків цифровий підпис власника і публічний ключ наступного власника. Тепер поточний власник при створенні наступного блоку запише туди ключ наступного власника та підпише його своїм власним ключем.[2]

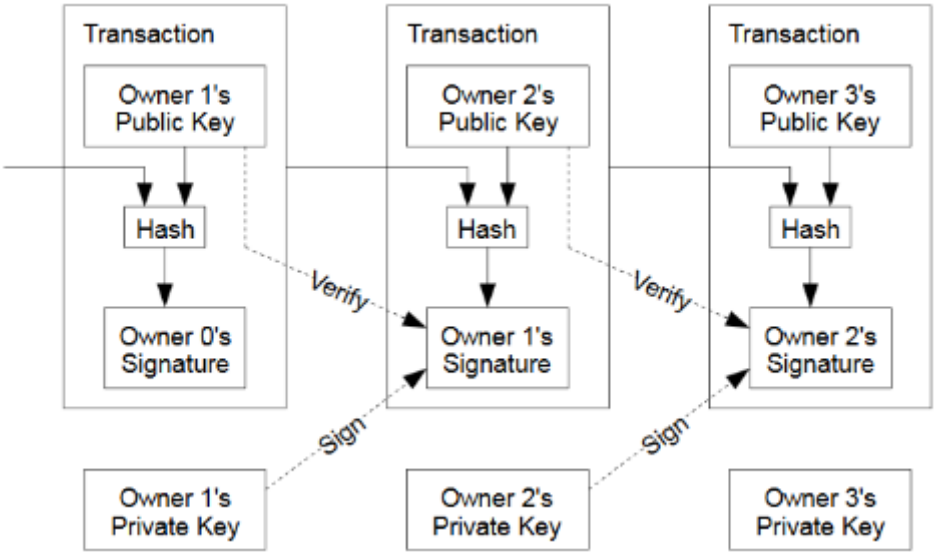

Рисунок 1.4.1 (Джерело: Satoshi Nakamoto)

1.5 **Блокчейн**

Уявимо, що існує джерело, яке постійно генерує нові блоки для хеш-чейну. Ми можемо покласти будь-яку інформацію в ці блоки, включаючи власний хеш-чейн. Фактично, ми отримаємо хеш-чейн у хеш-чейні. Саме цю конструкцію називають блокчейном[2]. Блок блокчейну складається з елементів, які називаються транзакціями.

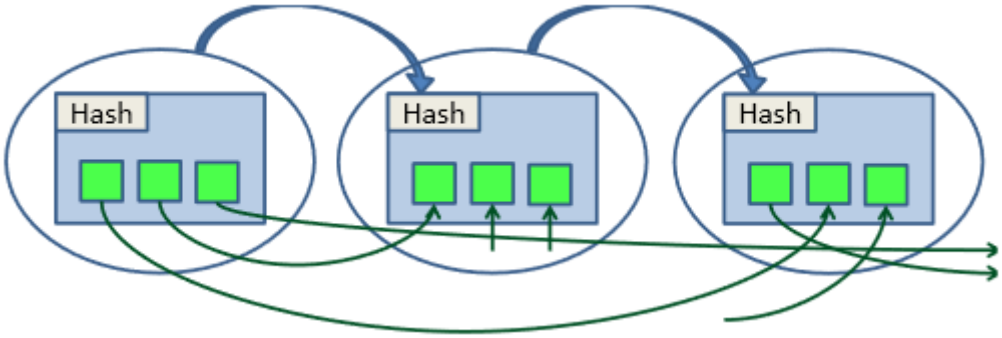

Рисунок 1.5.1 (Джерело: Oleg Mazonka)

Блокчейн мережа складається з комп'ютерів-учасників, які називаються **пірами**. Кожен пір зберігає власну копію блокчейну. Але при цьому усі копії підтримуються в єдиному стані за допомогою алгоритмів консенсусу.

Існує декілька різних підходів за якими можна організувати блокчейн. Розглянемо декілька з них.

2. Класифікація блокчейнів

2.1 За публічністю

Блокчейни можна поділити на два типи: публічні та приватні.

Публічний блокчейн дозволяє кожному доєднатися до нього та читати його дані.

Приватний чи консорціумний блокчейн належить певній організації чи консорціуму і не доступний для широкого загалу.

Зазвичай приватні блокчейни більш вразливі до фальсифікацій, але при цьому більш енергоефективні. Причини розглянемо нижче.

2.2 За алгоритмом консенсусу

Блокчейн – це децентралізована система. У ньому декілька нод можуть одночасно генерувати нові блоки до ланцюжка. Для того щоб запобігти виникненню хаосу, в системі діють правила за якими валідуються нові блоки які додаються до блокчейну. Існують три основні алгоритми консенсусу:

А) **Proof-of-Work (PoW)** – найпопулярніший алгоритм, використовується в Bitcoin. Кожна нода може створити новий блок, якщо вона обчислить математичну задачу. Складність задач налаштовується під кількість майнерів у мережі. Той, хто перший обчислить задачу, отримає винагороду. У біткоїні в середньому новий блок генерується раз в 10 хвилин. [4], [5]

Декілька нод можуть одночасно вирішити задачу. Тоді інші ноди будуть додавати блоки до однієї з нових нод і утвориться розгалуження. Щоб запобігти цьому, система обере ту гілку, яка має більше приєднаних блоків (можливі і інші алгоритми).[6]

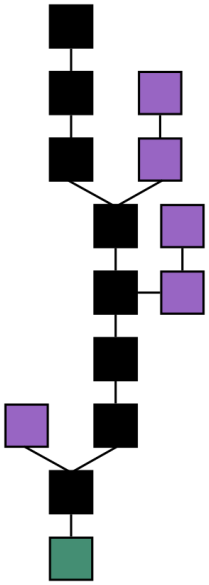

Рисунок 2.2.1 (Джерело: Theymos, Bitcoin Wiki)

Теоретично, якщо атакуючий контролюватиме 51% обчислюючих ресурсів, він зможе контролювати весь блокчейн. Саме тому надійність блокчейну напряму залежить від кількості підключених нод.

Проте основним мінусом даного алгоритму є його енергозатратність. Алгоритм спроектований марно витрачати ресурси майнерів, щоб сповільнити генерацію блоків. На сьогодні Bitcoin мережа витрачає більше електроенергії ніж уся Аргентина і практично наздогнала Україну.[7]

Б) **Proof-of-Stake (PoS)** – на відміну від PoW, тут ноди не розв'язують математичні задачі. Система обирає випадково ноду, яка буде генерувати наступний блок. При цьому чим більше валюти належить даній ноді, тим вищий у неї шанс на перемогу. Цей алгоритм набагато енергоефективніший аніж попередній. Але він також вразливий до атаки 51%. Якщо нода володіє 51% валюти вона контролюватиме блокчейном. [8]

В) **Proof-of-Autority (PoA)** – у багато чому схожа на PoS, але лише певні ноди-валідатори, які задані системою можуть генерувати нові блоки. Найефективніший алгоритм з точки зору енергоефективності та

швидкодії, але при цьому він є найбільш централізованим. Тому зазвичай використовується у непублічних блокчейнах.[9]

3. Розробка за допомогою Hyperledger Fabric

Пандемія Covid-19 наочно показала проблему використання паперових заліковок в 21 столітті. До того ж викладачі кажуть, що при виникненні конфліктних ситуацій, вони будуть орієнтуватися на запис у відомості, а не книжці. Це логічно, адже винахідливий студент може “поправити” свою оцінку в заліковій книжці.

Тому було б досить добре розробити електрону систему, яка б зберігала оцінки студентів і при цьому була б захищеною від втручання.

Як ми бачимо ці вимоги досить добре накладаються на можливості блокчейну і є хорошою можливістю протестувати його можливості на практиці.

Додаток повинен складатися з системи, яка зберігає оцінки та веб-застосунку, що вміє взаємодіяти з нею. Додаток має вміти авторизувати викладача чи студента, створювати нові предмети та додавати оцінки по предмету студентам.

Фреймворк (англ. **framework**) – інфраструктура програмних рішень, що полегшує розробку складних систем.[10]

У якості фреймворку для розробки блокчейну ми використаємо Hyperledger Fabric. Це проект з відкритим кодом, який належить Linux Foundation та розроблений за підтримки IBM. На сьогодні він є одним із найпопулярніших рішень у даній сфері. У 2019 році 30 з 50 стартапів зі списку “Forbes Blockchain 50” використовували саме його для розробки своїх систем.[11]

3.1 Чому приватний Blockchain

Hyperledger Fabric призначений для розробки приватних блокчейн мереж, у рамках однієї або декількох компаній. Це дає йому декілька переваг, які є привабливими для багатьох організацій.

По-перше, у приватних мережах організація контролює хто має доступ до блокчейну. Це означає, що приватні дані компанії, не будуть скопійованими по всім куточкам планети. І хоча ці дані зазвичай є зашифрованими, жодне шифрування не є стовідсотково надійним. Набагато краще зменшити ризик витоку даних завдяки зменшенню поверхні атаки.

По-друге, приватні мережі мають більший рівень довіри між співучасниками. Усі публічні блокчейн мережі побудовані на принципу відсутності довіри між учасниками. Саме тому в них використовуються складні та витратні алгоритми консенсусу. Проте така недовіра є надмірною у випадку приватної мережі. Учасники в ній не анонімні, оскільки ми контролюємо хто має доступ до неї. Тому погані гравці будуть досить легко виявлені, оскільки всі їх дії записуються. Також, оскільки ми контролюємо наш блокчейн, ми можемо налаштовувати обмеження для кожного учасника окремо.

Приватна мережа не має обов’язково бути розгорнута в середині лише однієї структури. Організації, що не довіряють одна одній, усе одно можуть створити спільну мережу. Для цього їм достатньо узгодити логіку блокчейну, права кожного учасника та правила, за якими керується дана мережа. Будь-які зміни до конфігурації потребуватимуть згоди інших учасників (згідно заданих раніше правил). Таким чином, ми одержуємо ігрове поле, в якому важко грати не за правилами.[12]

З попереднього пункту випливає, що немає потреби в складних алгоритмах консенсусу. Замість витратного **PoW**, можна використати

PoA, де нода-валідатор буде перевіряти вірність транзакцій згідно правил, що були встановлені в мережі.

Відсутність довіри в публічних мережах також накладає особливості в тому, як вони поширюють стан між нодами. В публічних мережах використовується модель **order-execute**. Згідно неї, транзакції спочату сортуються за допомогою консенсусу, а потім виконуються на кожній ноді, що присутня в мережі. Такий підхід називається активною реплікацією. Очевидно, що це надзвичайно марнотратно, адже кожен комп'ютер повторює одні й ті самі кроки, обраховуючи новий стан. Також це накладає деякі обмеження на транзакції, такі як те, що код має обов'язково детермінованим (фактично, це забороняє використання мов програмування загального призначення).[13]

На противагу публічним мережам, Hyperledger Fabric використовує модель **execute-order-validate**. У ній лише декілька нод виконавців виконують код. Кожна з них повертає свій результат (до і після). Далі він порівнюється між ними. Якщо результат певної кількості (задається при конфігурації) виконавців, транзакції сортуються і передаються іншим нодам. При цьому вони вже не виконують їх, а просто записують новий стан, після мало затратної валідації.[14] Окрім швидкодії, це дає перевагу в гнучкості (для кожного застосування може бути заданий свій рівень консенсусу) та дає можливість писати логіку блокчейну на традиційних мовах програмування.[13]

3.2 Основні принципи роботи Hyperledger Fabric

Реєстр Hyperledger Fabric складається з двох частин: стан світу та журнал транзакцій. **Стан світу (world state)** – це стан реєстру в даний момент часу. Він зберігається у сховищі ключ-значення. Журнал транзакцій зберігає історію всіх транзакцій, що призвели до поточного

стану світу. Це дозволяє за потреби відновити минулий стан, але при цьому не потрібно виконувати всі транзакції для отримання поточного стану.[12]

Логіка блокчейну пишеться за допомогою чейнкоду. **Чейнкод (chaincode) –** це застосунок, що має доступ до стану світу, і взаємодіє з ним отримуючи чи змінюючи інформацію у ньому. Він також має доступ до журналу транзакцій, але на практиці це рідко використовується. Hyperledger Fabric підтримує три мовні екосистеми: Go, Node.js та Java.[12]

Канали (channels) дозволяють забезпечити приватність. Мережа Fabric може містити велику кількість організацій. Якщо деякі з них хочуть забезпечити приватність свої контрактів вони можуть створити свій власний канал. Кожний канал має свій власний реєстр, чейнкод та правила за якими він оперує. До його даних мають доступ лише організації, що є його учасниками[12]. Більш того, кожна організація може створити власну приватну колекцію, де вона може зберігати дані приховані навіть від інших організацій-членів даного каналу[15]. Таким чином навіть у приватній мережі може бути ще один рівень приватності.

Існує декілька різних видів нод у Hyperledger Fabric.

Піри (peers) – це ноди, які зберігають реєстр та виконують чейнкод. Пір може належати багатьом каналам.

Сервіс впорядковування (ordering service) один на канал. Він отримує транзакції, впорядковує їх, збирає у блок та надсилає пірам на запис.

Клієнт – може створювати транзакції надсилаючи їх пірам.[12]

В Hyperledger Fabric всі учасники є ідентифікованими. Для ідентифікації використовуються сертифікати. Принцип дії публічних та приватних ключів, і як вони можуть використовуватись для

ідентифікації був описаний раніше. Фактично сертифікат – це файл, що містить ім'я власника та його публічний ключ. Сертифікат видається органом сертифікації (**Certificate Authority** або **CA**), який підписує його своїм цифровим підписом. Hyperledger Fabric має власний сервер CA, який можна використовувати для реєстрації користувачів. Проте він не є обов'язковим, адже Fabric використовує сертифікати у форматі X.509. Це означає, що компанія може використовувати інші рішення для автентифікації, як власний CA або OAuth сервіс спеціалізованих компаній.[3], [12]

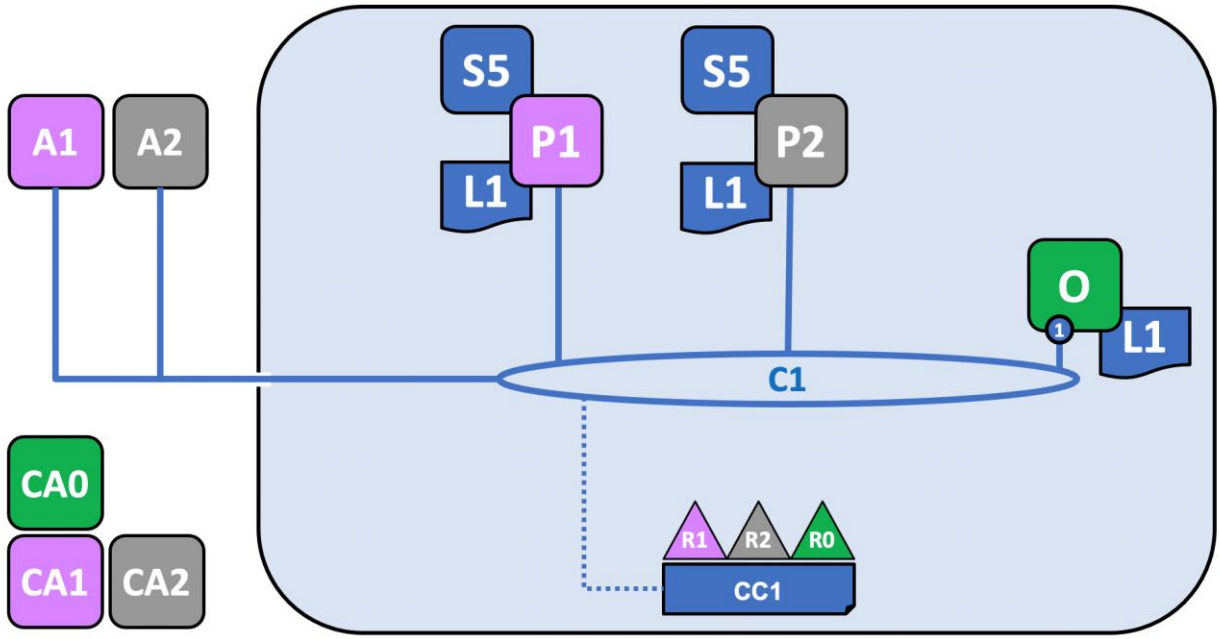

Рисунок 3.2.1 (Джерело: Документація Hyperledger Fabric)

На рисунку 3.1 зображено один з прикладів структури мережі Hyperledger Fabric. Три організації R0, R1, R2 домовилися створити спільний канал C1 з конфігурацією CC1. Кожна з них має власний орган сертифікації CA0, CA1 та CA2, які відповідають за ідентифікацію членів відповідних організацій.

Кожний канал повинен сервіс впорядкування. У даному випадку це O, що належить компанії R0.[16]

Організації R1 та R2 володіють парами P1 та P2 відповідно. Кожний з цих пірів та сервіс впорядкування мають власну копію реєстру L1. Також на обох

пірах встановлений чейнкод S5. Слід зазначити, що чейнкод не обов'язково повинен бути встановлений на кожному пірі. Кожний чейнкод містить інформацію про його політику схвалення (**endorsement policy**) – це правила, що зазначають, які піри мають виконати транзакцію для її схвалення.

Тепер застосунки A1 та A2 можуть викликати транзакції, що містяться в S5, звертаючись до пірів, що належать їх організаціям.[16]

Організації можуть бути учасниками декількох каналів одночасно.

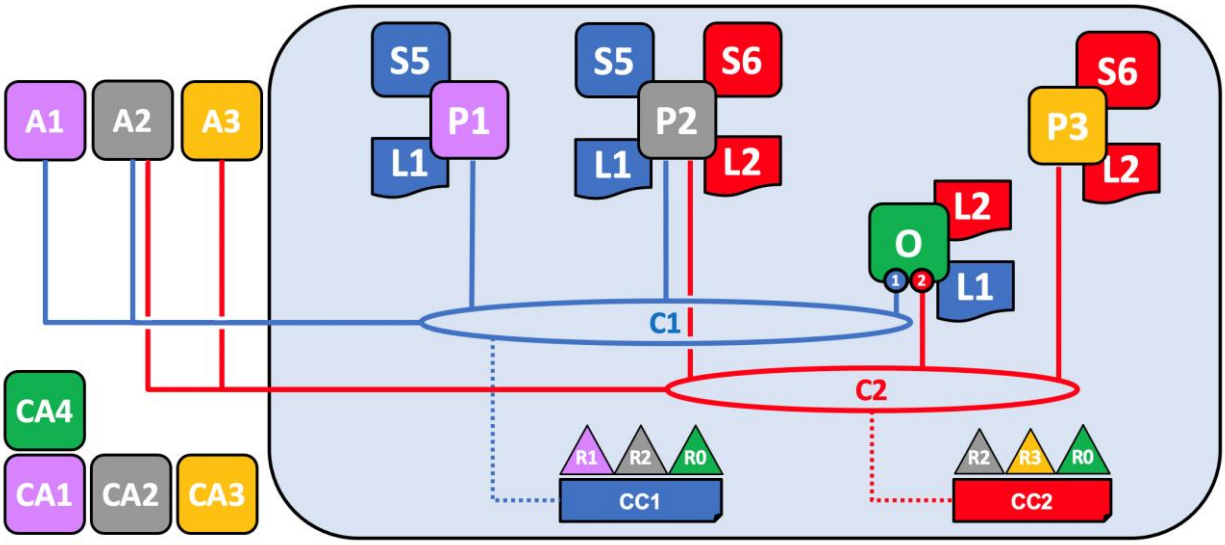

Рисунок 3.2.2 (Джерело: Документація Hyperledger Fabric)

На рисунку 3.2 до попередньої схеми був доданий новий канал C2. Його учасниками є організації R0, R2 та R3. Зверніть увагу, що пір P2 тепер належить обом каналам одночасно і містить другий реєстр L2 та чейнкод S6. Сервіс O тепер містить дві ноди, кожна з яких відповідає за окремий канал. Організація R1 не має жодного відношення до каналу C2 та навіть не знає про його існування.

4. Реалізація залікової книжки студента

Фронтенд (англ. **front-end**) – це частина застосунку, яка виконується на клієнтському апаратному забезпеченні. В основному відповідає за інтерфейс користувача.

Бекенд (англ. **back-end**) – це частина застосунку, яка виконується на сервері. В основному відповідає за виконання бізнес-логіки системи.

4.1 Структура проекту

Даний проект складається з трьох основних частин:

- Мережа Hyperledger Fabric, де зберігається стан залікових книжок та виконується вся бізнес-логіка нашої системи.
- Бекенд застосування, яке безпосередньо спілкується з блокчейном та надає REST API, які споживаються клієнтами.
- Фронтенд застосування, яке спілкується з бекендом, надає інтерфейс користувача та можливість переглянути оцінки, реєструвати користувачів, додавати предмети та оцінки.

У якості фронтенд-фреймворку був використаний React. На даному рівні технологія не має жодного значення. Єдина вимога – підтримка HTTP запитів.

Для написання чейнкоду було обрано мову Typescript. Вона надає баланс між зручністю написання коду та безпекою від неправильного використання типів. Hyperledger Fabric підтримує Typescript і всі бібліотеки в його SDK містять типи “з коробки”.

У якості бекенд-фреймворку ми повинні використати один з платформи, що має SDK для з’єднання з мережею Fabric. Це може бути Go, Node.js або Java. У даному випадку був Node.js, а саме Nest.js. Це фреймворк-надбудова над Express, який підтримує Typescript, Dependency Injection та модульну систему з чітким розділенням компонентів.

Також слід зазначити, ще один плюс використання Typescript - його можна використовувати на всіх рівнях нашої системи. Це спрощує підтримку системи та дозволяє використовувати ті самі моделі даних на клієнті та сервері.

Рисунок 4.1.1 - Порядок обробки запитів користувача

Для того щоб запустити блокчейн-мережу на одному комп'ютері нам потрібно використати віртуалізацію для емуляції роботи кожної з нод. На сьогоднішній день найзручнішим рішенням для цього є Docker. Він дозволяє швидко розгорнути мережу, споживає мінімальну кількість ресурсів та ізолює всі компоненти одне від одного. Саме тому він використовується в навчальних матеріалах Fabric і в даному проекті.

4.2 Структура мережі

Даний проект побудований за допомогою тестової мережі, що надається в навчальних матеріалах Hyperledger Fabric.

Усі файли з конфігурацією нашої мережі знаходяться в папці record_blockchain/network. Тут також знаходяться скрипти, надані командою Fabric, для швидкого розгортання мережі.

Для початку поглянемо в папку organizations. Тут знаходяться визначення організацій, які будуть наявні в даній мережі. Папка cryptogen містить конфігураційні файли для однойменної тестової утиліти, яка генерує сертифікати для організацій у випадку коли ми не розгортаємо органи сертифікації в мережі. Проте, оскільки ми хочемо мати можливість реєструвати нових користувачів, дана опція нам не підходить. У даному випадку Hyperledger Fabric надає власний CA, конфігурація якого знаходиться в папці fabric-ca.

Тут знаходяться три папки, по-одній для кожної організації в мережі. В даній мережі присутні три організації:

а). OrdererOrg – організація, що відповідає за сервіс впорядкування транзакцій.

б). Org1 – організація, яка представляє співробітників університету. Її члени матимуть змогу вносити зміни в чейнкод.

в). Org2 – організація, яка представляє студентів.

Конфігурація організацій залежить від проекту. Зазвичай в одній компанії використовується одна організація, а її підструктури конфігуруються за допомогою організаційних одиниць (**organizational units** або **OU**)[17]. Тут використано дві організації, щоб показати можливість їх взаємодії між собою.

Кожна з папок містить конфігураційний файл **fabric-ca-server-config.yaml**. Тут задається назва органу сертифікації, параметри шифрування та кореневого сертифікату, підрозділи організації тощо. Також тут задане визначення користувача-адміністратора організації.

```
registry:
  identities:
     - name: admin
       pass: adminpw
```

Адміністратор має право реєструвати нових користувачів. Оскільки кожна організація має свій власний CA, то і користувачі у них різні. Кожна з організацій має власного адміністратора, який може реєструвати нових користувачів лише в своїй організації.

Тепер повернемось до папки network та переглянемо файл **configtx/configtx.yaml**. Тут знаходиться конфігурація каналу.

Перший розділ цього файлу називається Organizations. Тут задаються параметри постачальника послуг членства (**Membership Service Provider** або **MSP**) для кожної з організацій в мережі. Під час запиту на виконання транзакції MSP перевіряє належність сертифікату, який був наданий користувачем, до організації. Також тут задаються політики доступу для членів організації. У Fabric CA є 4 види користувачів[17]:

а). **admin** – адміністратор

б). **peer** – пір

в). **orderer** – упорядковувач

г). **client** – клієнт

Вид користувача задається при його реєстрації.

При цьому в самому Hyperledger Fabric присутній, ще один розподіл – за їх правами[18]:

а). Readers – можуть читати

б). Writers – можуть писати

в). Admins – адміністратори

г). Endorsement – ті, хто виконують і перевіряють транзакції.

Для кожної з наведених вище політик потрібно вказати, які члени організацій належать їм.

```
Policies:
            Writers:
                Type: Signature
                Rule: "OR('Org1MSP.admin', 'Org1MSP.client')"
```

Параметр “Type: Signature”, означає, що в правилі “Role” задані ролі користувачів, які мають дану привілею[18]. Так, в прикладі вище, адміністратори і клієнти мають право на запис.

Проте нижче у файлі можна побачити інший вид правил – ImplicitMeta.

```
Endorsement:
            Type: ImplicitMeta
            Rule: "MAJORITY Endorsement"
```

Ці правила агрегують політики з усіх організацій в одну[18]. Так, в прикладі вище, для того щоб транзакція була схвалена, її мають схвалити більшість членів з політикою схвалювання.

В даному файлі записані політики за-замовчуванням для чейнкоду, каналу і сервісу впорядкування.

У папці docker знаходяться файли Docker Compose для побудови мережі за допомогою Docker. Docker Compose дозволяє задати які образи нам потрібні та їхню конфігурацію. Це дозволить Docker швидко побудувати кластер без нашого втручання.

Завантажимо двійкові файли з програмами, потрібними для побудови мережі, виконавши скрипт bootstrap.sh. (Для виконання потрібно середовище bash).

Спочатку дозволимо виконання скриптів в папці.

```
record_blockchain$ sudo chmod -R 775 network
```

Потім виконаємо скрипт.

```
record_blockchain/network$ ./bootstrap.sh
```

Тепер ми можемо викликати команду, яка розгорне мережу.

```
./network.sh up createChannel -ca -s couchdb
```

Скрипт network.sh дозволяє розгортати та видаляти мережу.

Він приймає декілька параметрів:

- createChannel – при розгортанні мережі одразу створювати канал
- -ca – розгорнути органи сертифікації Fabric CA
- -s couchdb – використовувати в якості сховища стану світу Apache CouchDB

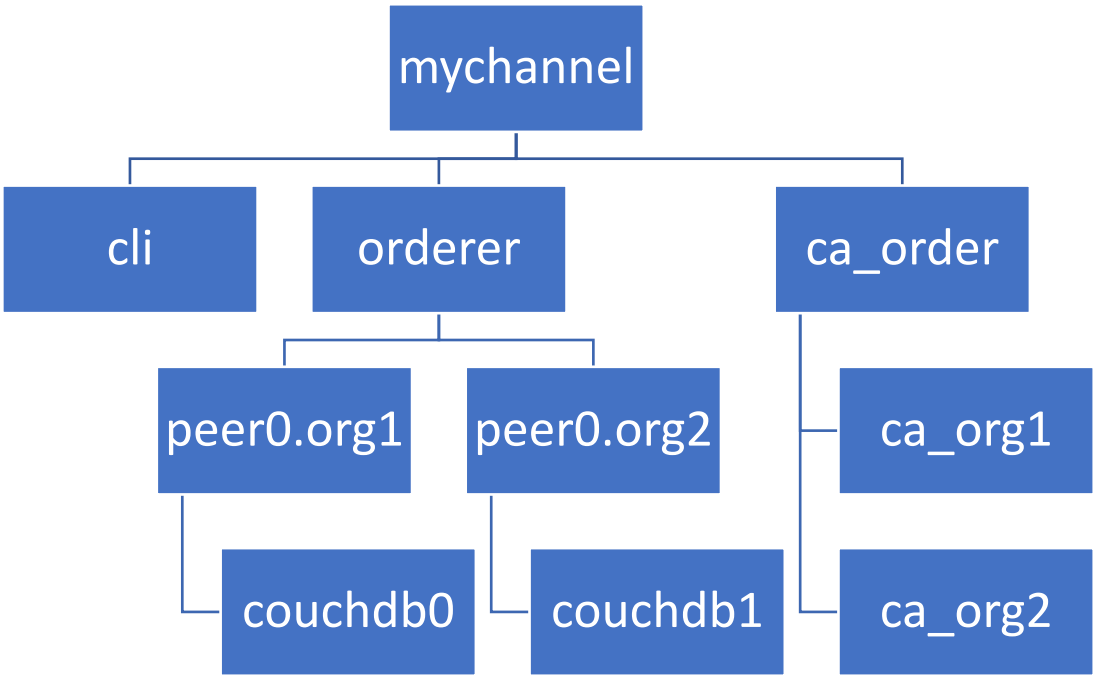

Рисунок 4.2.1 - Схема розгорнутої мережі

Для видалення мережі достатньо виконати команду

```
record_blockchain/network$ ./network.sh down
```

Тепер, маючи розгорнуту мережу ми можемо перейти до чейнкоду.

3.3 Чейнкод залікової книжки

Чейнкод залікової книжки знаходиться в папці record_chain. Це звичайний Node.js проект. Для роботи з чейнкодом Hyperledger Fabric нам потрібно встановити дві залежності з NPM.

```
"dependencies": {
    "fabric-contract-api": "^2.2.1",
    "fabric-shim": "^2.2.1"
```

```
  }
```

У першу чергу нам потрібно створити модель даних, які ми будемо зберігати в блокчейні. Поглянемо на recordBookState.ts (Додаток Б).

Для визначення моделі використовуються звичайні ES6 класи. Кожен клас анотується спеціальним декоратором Object, а поля – Property.

У даній моделі кожен студент визначається за його електронною адресою. Кожен студент зберігає інформацію про семестри в яких він отримав оцінки, а саме рік семестру та його порядковий номер (1, 2, 3 …). В кожному семестрі зберігається інформація про оцінки. Кожна оцінка зберігає інформацію про назву предмету, адресу викладача та сам бал.

Можна помітити, що дані про викладача та предмети можуть дублюватися між оцінками. Причиною цього є те, що стан світу в Hyperledger Fabric зберігається у вигляді JSON в сховищі ключ-значення, яке не є реляційним. Звичайною практикою при використанні NoSQL баз даних є зберігання усієї потрібної інформації, яка потрібна буде в запитах[19]. Звичайно, це може призвести до того, що при оновленні електронної пошти викладача, записи в книжці виявляться з застарілою інформацією. Проте, потрібно згадати, що наша система прийшла на заміну паперовим книжкам, де підтримка інформації в актуальному вигляді не можлива в апріорі. До того ж, записи в таких документах і не повинні змінюватись після їх створення.

Тепер ми можемо перейти до створення чейнкоду залікової книжки. Подивимось на файл **recordBookContract.ts** (Додаток В). Тут знаходиться вся логіка нашого блокчейну. Для визначення чейнкоду в Typescript потрібно створити клас, що унаслідується від класу Contract.

```
@Info({ title: "RecordBook", description: "Chaincode for student books" })
export class RecordBookContract extends Contract {
```

Тепер, давайте додамо транзакцію, що ініціалізує залікову книжку для заданого студента.

```
@Transaction()
  public async CreateBook(ctx: Context, email: string) {
    const book: RecordBook = {
      email,
      periods: [],
    };

    await ctx.stub.putState(
      RecordBookContract.getStudentKey(ctx, email),
      Buffer.from(JSON.stringify(book))
    );
  }
```

Метод-транзакція позначається декоратором Transaction і приймає першим параметром об’єкт типу Context. Наступні параметри – це аргументи, які повинен надати користувач при виклику транзакції. Параметр Context містить всі інформацію про контекст поточної транзакції, а саме інформацію про користувача, що її викликав, методи для взаємодії з реєстром та логером. Також розробник може додати свої поля до контексту, унаслідувавшись від класу Context.

Для збереження даних в стан світу, потрібно викликати метод putState. Він приймає два параметри: ключ типу string, та значення у форматі JSON. Як ми бачимо, процес створення та збереження об’єкту є напрочуд простим. Проте є і мінус – стан лише один і він не типізований. Це означає, що нам потрібно зберігати в одній колекції інформацію і про заліковки, і про предмети. До того ж не можна точно визначити, якого типу є об’єкт за ключем “3” та можливі конфлікти коли в сутностей різних типів співпадають ключі.

Задля вирішення даної проблеми можна використати композитні ключі. Композитний ключ складається з префіксу, що позначає тип сутності та власне значення ключа. Наприклад “recordBook~yurii.polishchuk@ukma.edu.ua” – ключ для залікової книжки

студента з електронною адресою “yurii.polishchuk@ukma.edu.ua”. Це усуває можливість конфліктів та дає інформацію про тип сутності за даним ключем. Hyperledger Fabric підтримує композитні ключі та надає низку методів для роботи з ними.

Проте композитні ключі все одно не захищають від збереження об’єктів невірного типу за ними. Задля зменшення ризику виникнення такої ситуації, я не рекомендую викликати putState напряму в транзакціях. Натомість краще визначити типізовану функцію-обгортку над викликом і використовувати її.

```
private static async SaveBook (ctx: Context, book: RecordBook) {
    await ctx.stub.putState(
      RecordBookContract.getStudentKey(ctx, book.email),
      Buffer.from(JSON.stringify(book))
    );
  }
```

Для того, щоб Fabric не вважав допоміжні методи за транзакції, потрібно або визначити їх як статичні члени класу, або додати символ ‘_’ перед назвою методу.

Для обробки виключних ситуацій використовується стандартні винятки (**exceptions**) JavaScript. Наприклад, поглянемо на транзакцію, що зчитує залікову книжку для заданого студента.

```
@Transaction(false)
  public async ReadBook(ctx: Context, email: string): Promise<RecordBook>
{
    const bookJSON = await ctx.stub.getState(
      RecordBookContract.getStudentKey(ctx, email)
    );
    if (!bookJSON || bookJSON.length === 0) {
      throw new Error(`The book of student ${email} does not exist`);
    }
    return JSON.parse(bookJSON.toString()) as RecordBook;
  }
```

Користувач може запитати залікову книжку студента, якого немає в базі. У такому випадку програма поверне помилку з відповідним повідомленням. Як

можна помітити, для зчитування об’єкту зі стану світу використовується метод getState, що приймає ключ, а повертає JSON-значення об’єкту.

У деяких ситуаціях транзакції не повинні бути доступні всім користувачам. Так, в нашій системі студенти не повинні мати право створювати предмети, додавати собі оцінки, тощо. Щоб досягти цього, ми можемо зчитати ідентичність користувача.

```
private static ensureUniversityStaff(ctx: Context) {
    if (ctx.clientIdentity.getMSPID() !== "Org1MSP")
      throw new Error("You do not have access to this function");
  }
```

У прикладі вище, ми зчитуємо інформацію про організацію, якій належить клієнт. Якщо вона не дорівнює “Org1MSP” (організація співробітників університету), то програма повертає відповідну помилку.

Hyperledger Fabric підтримує дві бази даних для збереження стану світу – LevelDB або Apache CouchDB. За замовчуванням використовується LevelDB, проте розробник може сам обрати CouchDB (що ми і зробили при розгортанні мережі). У такому разі ми отримаємо можливість шукати об’єкти за допомогою складніших запитів (Додаток Г).[20]

Задля пришвидшення таких запитів можна створювати індекси. Їх потрібно визначати в папці META-INF/statedb/couchdb/indexes за допомогою конфігураційного файлу в форматі JSON (Додаток Ґ).[20]

У полі “fields” потрібно вказати поля, за якими потрібно індексувати запити (ті поля, за якими ми фільтруємо дані). “ddoc” і “name” – це ідентифікатори індексу.

4.2.1 Встановлення чейнкоду до мережі

Після створення чейнкоду, ми можемо перейти до його встановлення до мережі. Для цього нам потрібно повернутись до папки network та викликати скрипт network.sh з новими параметрами.

network$ ./network.sh deployCC -ccn record_book -ccp ../record_chain/ -ccv 1 -ccl typescript

Команда deployCC встановлює чейнкод у мережу і приймає декілька параметрів:

а) -ccn – назва чейнкоду в мережі

б) -ccp – розташування проекту з чейнкодом

в) -ccv – версія чейнкоду

г) -ccl – мова, на якій написаний чейнкод (typescript, javascript, go, java)

Ця команда виконує декілька кроків, щоб встановити чейнкод до мережі.

1. Викликає “npm install” і “tsc”, щоб скомпілювати проект з чейнкодом.
2. Запаковує файл з проектом в архів за допомогою команди “peer”. Утиліта “peer” використовується для взаємодії з пірами і знаходиться в папці network/bin.
3. Встановлює чейнкод на кожному з пірів. Для цього потрібно використати згенеровані сертифікати адміністраторів для кожної з організацій. Вони були згенеровані при створенні мережі та
4. Схвалює чейнкод з кожної організації.
    a. Отримує ідентифікатор чейнкоду встановленого на піру
    b. Схвалює чейнкод за заданим ID
5. Одна з організацій підтверджує чейнкод в каналі [21]

Як ми бачимо, багато кроків повинні бути виконані повторно для кожної з організацій окремо. Це логічно, адже кожна з структур відповідає лише за свою частину і не має доступу до інших. Для того, щоб встановити чейнкод потрібно отримати згоду більшості членів каналу (згідно налаштувань за-замовчуванням). І, звісно, організація повинна схвалити чейнкод, якщо вона хоче використати його на своїх пірах.

Тепер, встановивши чейнкод до мережі, ми можемо приступити до розробки клієнтського застосунку.

4.3 Бекенд частина застосунку

Для зв’язку з нашим блокчейном, нам потрібно встановити дві залежності.

```
"dependencies": {
    "fabric-ca-client": "^2.2.5",
    "fabric-network": "^2.2.5",
  }
```

Для того, щоб викликати транзакцію, ми повинні авторизуватись. Як вже було зазначено раніше, Hyperledger Fabric використовує сертифікати X.509 для авторизації користувачів. Утім, спочатку потрібно їх отримати.

Процес реєстрації користувача в Fabric CA складається з двох кроків. Спочатку адміністратор організації робить запит на реєстрацію користувача вказуючи його логін та роль. У відповідь сервер надсилає йому “секретну” стрічку – випадковий хеш (адміністратор також може вказати “секрет” при реєстрації самостійно). Далі адміністратор або сам користувач надсилає другий запит до CA на зарахування (**enrollment**), вказуючи логін та “секретне” значення. І лише тоді сервер повертає сертифікат для користувача.[22]

Сертифікати користувачів зберігаються у спеціальному гаманці з бібліотеки Fabric. При надсиланні запиту до блокчейну, користувач вказує цей гаманець та який логін з нього використовувати. Існує декілька видів гаманців: для зберігання в оперативній пам’яті, на диску, тощо. Ми будемо використовувати файловий гаманець.

У авторизації з використанням сертифікатів є декілька мінусів, які проявляються в нашому проекті. По-перше, дуже складно підтримувати авторизовану сесію користувача, адже гаманець повинен знаходитися на сервері. По-друге, якщо користувач втратить файл з сертифікатом, то він не

зможе авторизуватись. Відновити сертифікат він зможе лише якщо пам’ятає секрет.

Тому ми використаємо інший підхід. Користувачі і надалі використовуватимуть логін та пароль. При реєстрації на сайті, сервер надсилатиме запит в Fabric CA на отримання сертифікату. Отримавши сертифікат, сервер кладе його в свій гаманець та зв’язує з логіном користувача. Надалі кожен раз, коли авторизований користувач надсилає запит, сервер надсилає запит до мережі Fabric, використовуючи сертифікат пов’язаний з логіном.

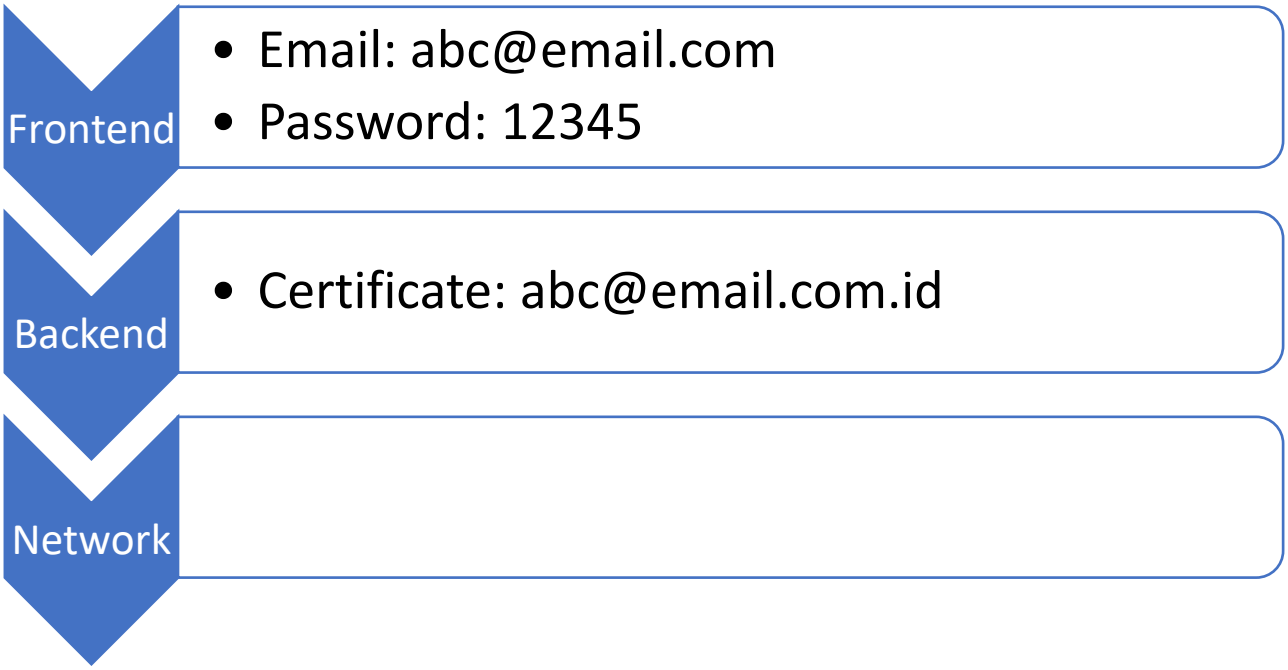

Рисунок 4.3.1 – Порядок ідентифікації користувача в мережі

Для авторизації фронтенд-застосування, я використовую JWT-токени. Через цей підхід до ідентифікації нам усе ж таки знадобиться “традиційніша” база даних. Для даного проекту я обрав PostgreSQL. Вона також доступна у вигляді контейнера Docker.

Для зв’язку з мережею Fabric, нам потрібно знати за якою адресою до неї звертатися. Якщо ми поглянемо в папку network/organizations при запущеній мережі, то побачимо там нові папки. Перейшовши в папку peerOrganizations/ org1.example.com, ми знайдемо файл connection-org1.json. Це файл з

налаштуваннями для з’єднання з нодами організації. Саме ці налаштування нам потрібні для з’єднання з мережею. Потрібно зауважити, що вміст файлів буде відрізнятися при кожному створенні мережі, оскільки вони містять сертифікати нод організації, які генеруються Fabric CA.

Бекенд налаштований брати конфігурацію для кожної організації з відповідних папок. За створення та збереження сертифікатів відповідає клас CAService (додаток Д). Можна помітити, що кожна організація має свій окремий гаманець, адже організації можуть мати користувачів з однаковими логінами.

Для виклику транзакції використовуються об’єкти класу Contract. За їх створення відповідає клас ContractService (додаток Е). При створенні необхідно вказати організацію, користувача, гаманець та конфігураційний файл мережі.

Тепер, ми можемо взаємодіяти з блокчейном. Для прикладу поглянемо на клас RecordBookService (додаток Є).

Метод AddMark додає оцінку за предмет студентові. Для того щоб викликати транзакцію використовується метод submitTransaction. Першим аргументом він приймає назву методу в чейнкоді, який ми викликаємо. Наступними йдуть аргументи у вигляді стрічок.

Таким же чином відбувається отримання даних. Для запитів на отримання даних можна також використати метод evaluateTransaction. На відміну від submitTransaction, evaluateTransaction не записує факт виклику транзакції в реєстр.

4.4 Фронтенд частина

Фронтенд частина застосунку написана на SPA-фреймворку React, і за реалізацією не відрізняється від інших додатків даного типу. У якості бібліотеки для управління станом використовується Recoil. Оскільки в нашій

системі є декілька ролей з різними можливостями, то деякі сторінки будуть недоступні для різних користувачів.

5. Огляд застосунку

При відкритті сторінки, користувач потрапляє на екран входу.

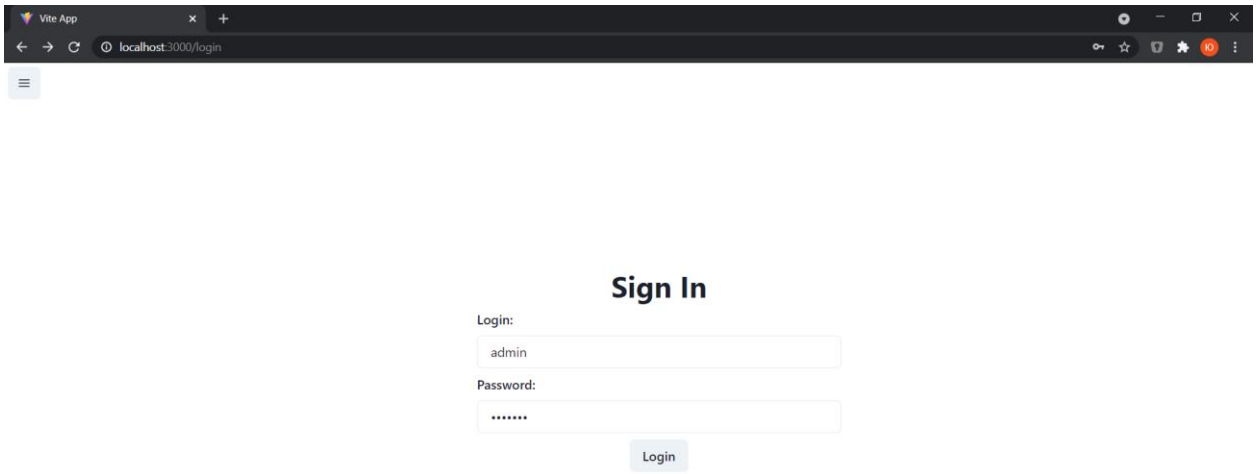

Рисунок 5.1

Введемо вхідні дані адміністратора (логін “admin” та пароль “adminpw”). Після натискання на кнопку Login, користувач перейде на основну сторінку для його ролі. Для студента це сторінка залікової книжки, викладача – додавання оцінок, адміністратора – реєстрації нових користувачів.

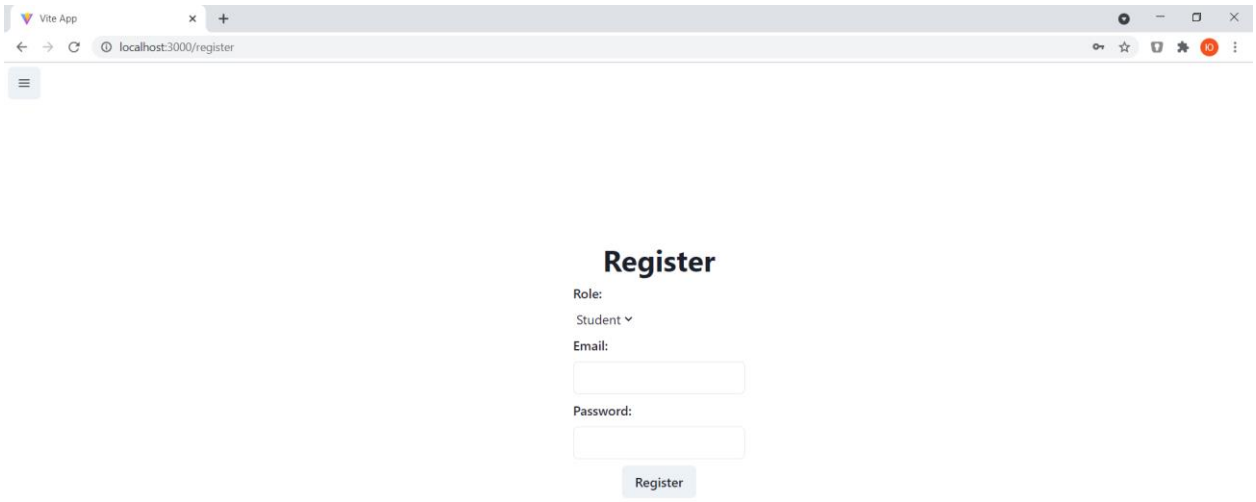

Рисунок 54.4.2

Натиснувши на кнопку в лівому верхньому куті, користувач може перемикатися між доступними йому сторінками.

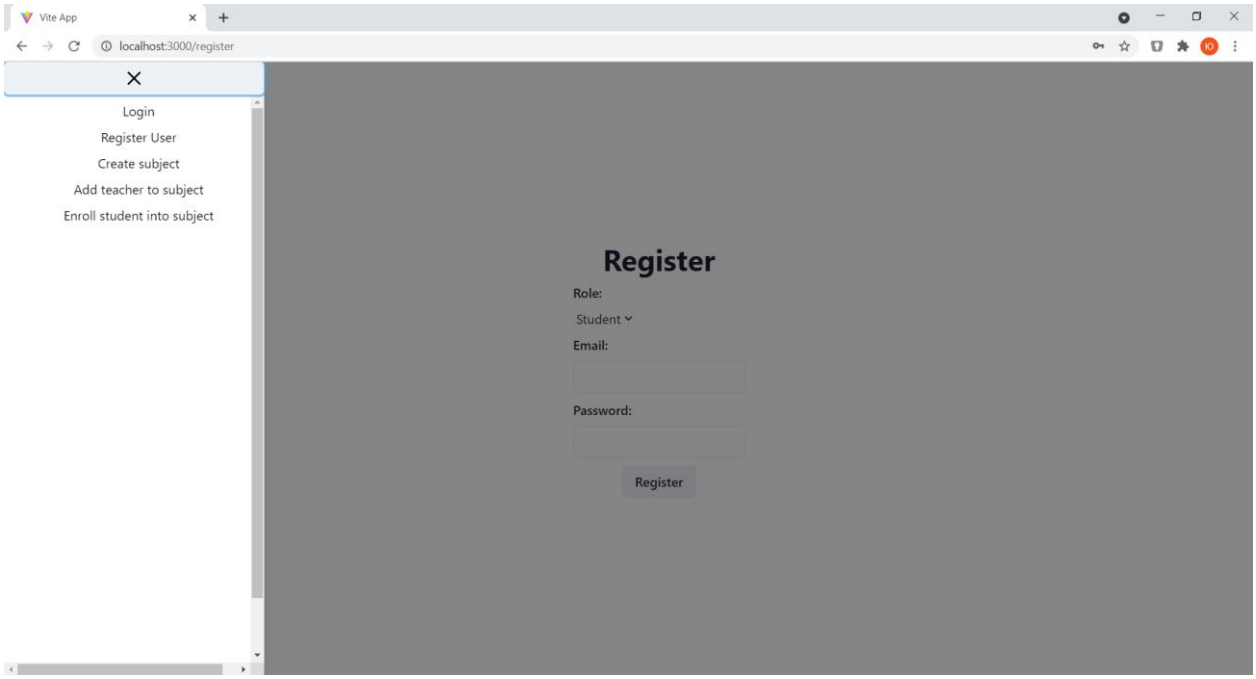

Рисунок 5.3

Зареєструємо одного викладача та студента.

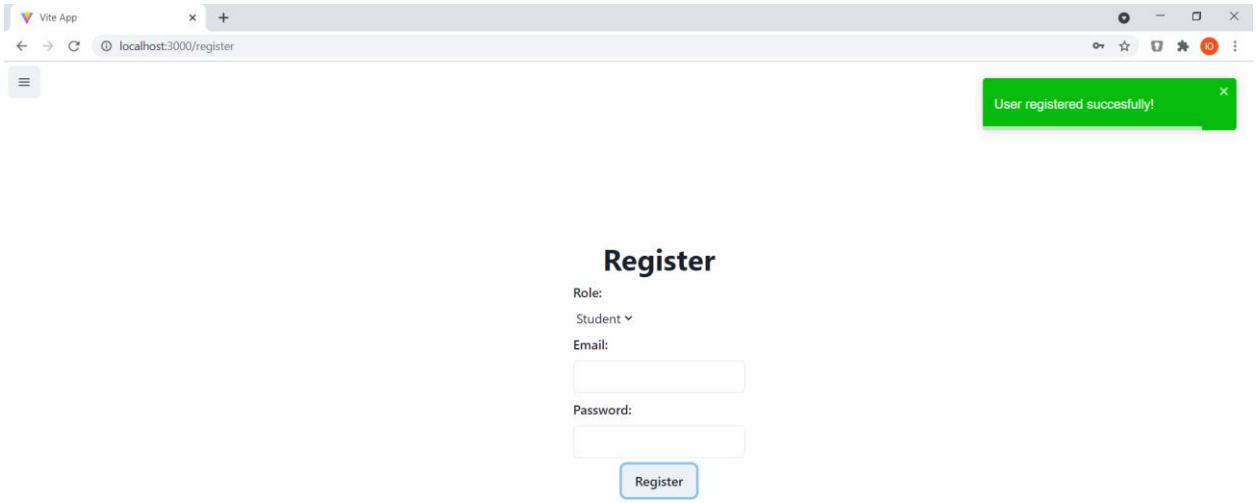

Рисунок 5.4

Перейдемо на сторінку створення предмету та додамо нові предмети.

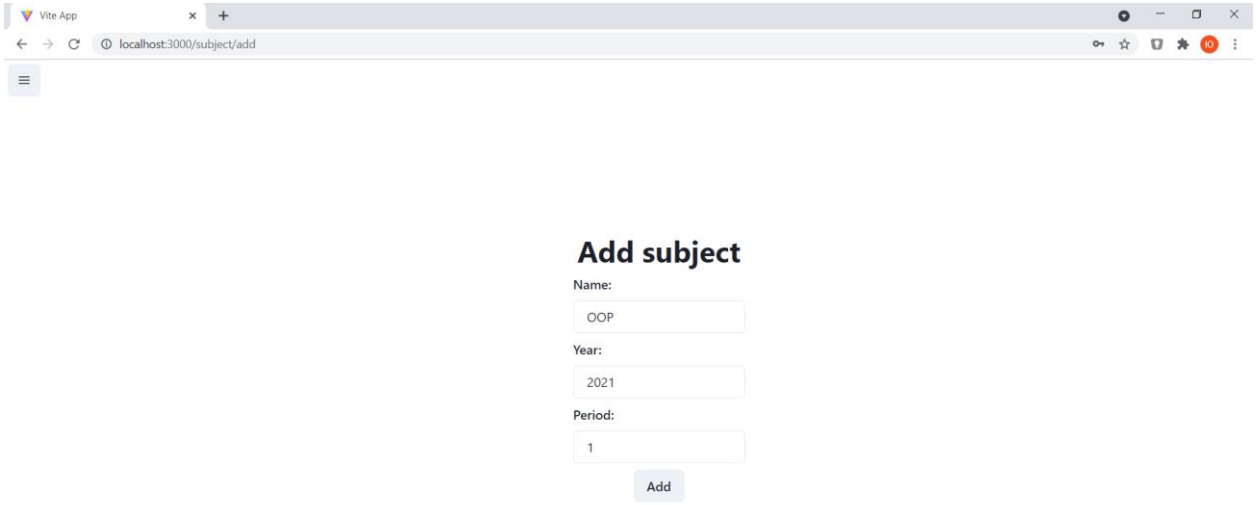

Рисунок 54.4.5

Перейдемо на сторінку запису викладача до предмету та додамо викладача.

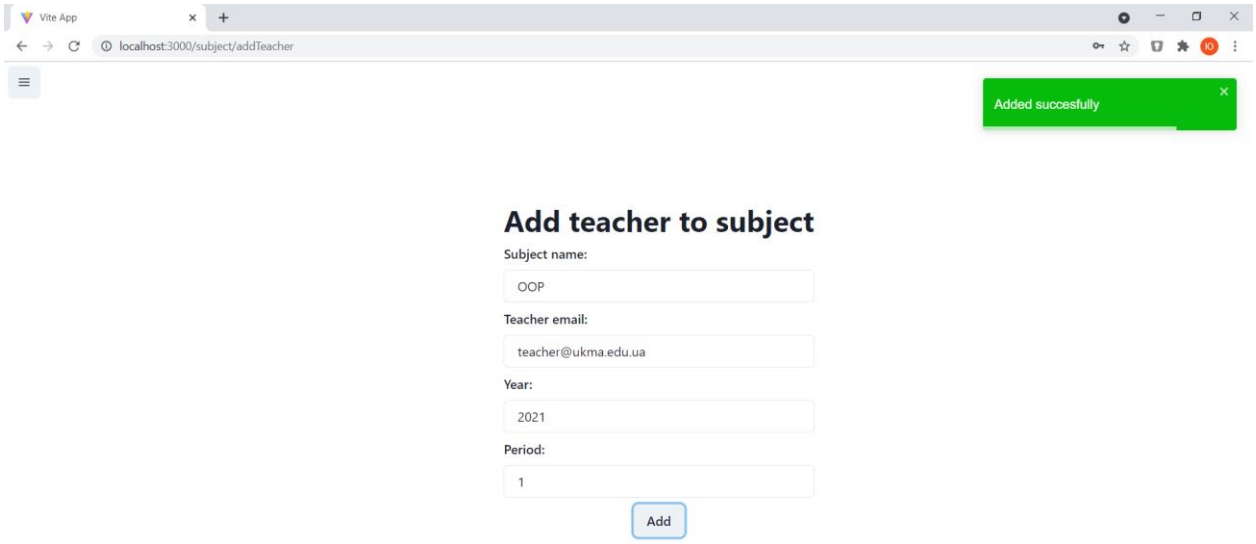

Рисунок 5.6

Також перейдемо на сторінку запису студента до предмету та додамо студента.

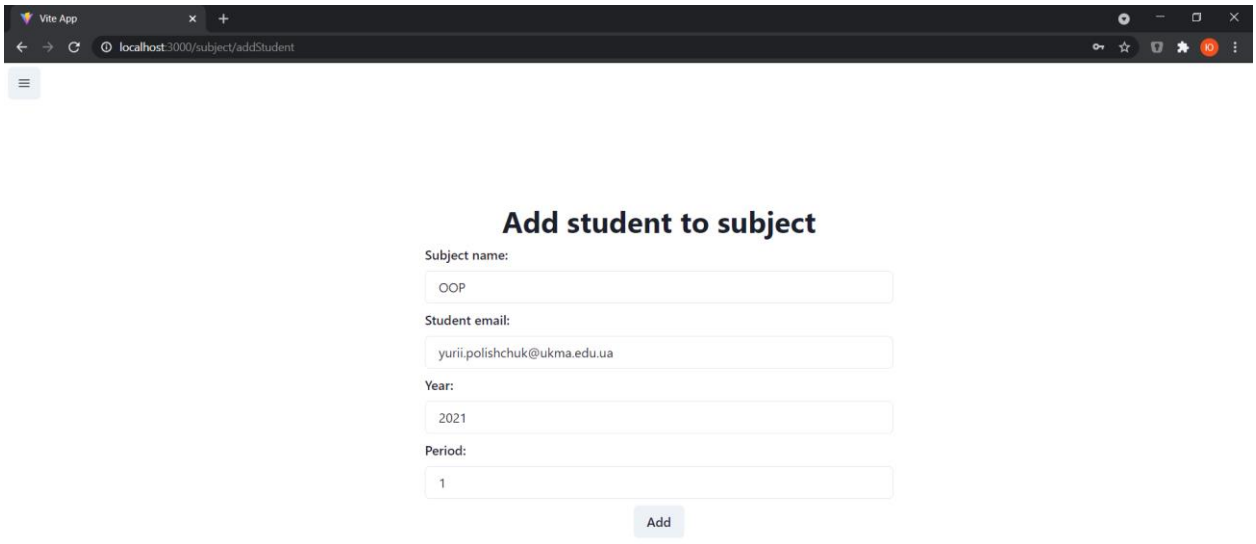

Рисунок 5.7

Тепер увійдемо в акаунт викладача та додамо оцінки студентові.

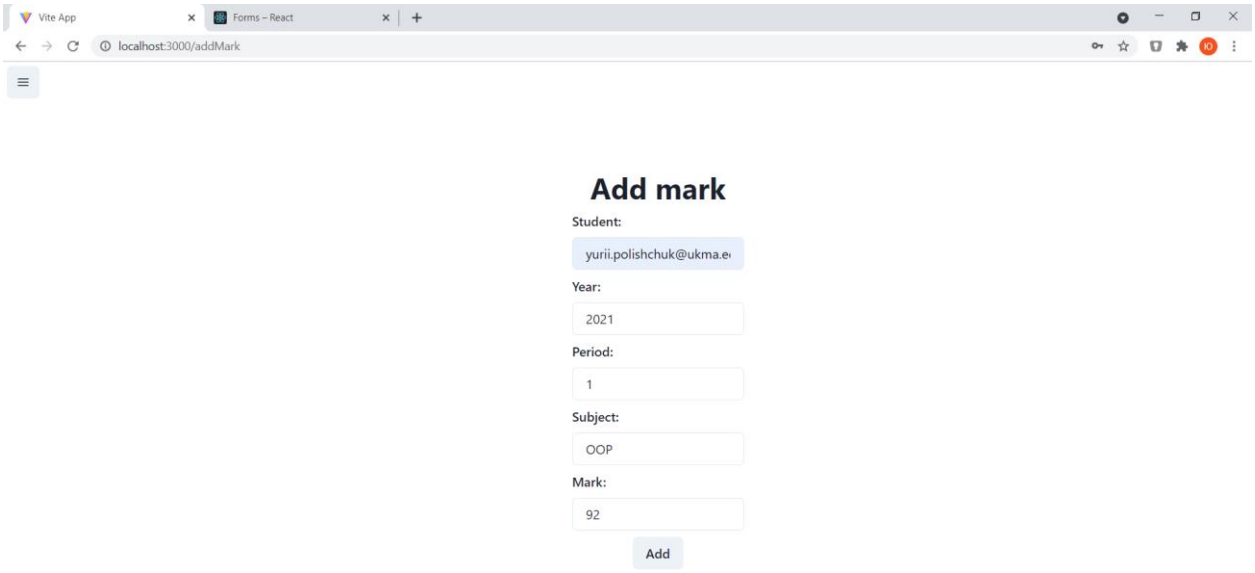

Рисунок 5.8

Тепер, увійшовши в свій акаунт, студент побачить свої оцінки.

Рисунок 5.9

Висновки

Під час виконання курсової роботи, була створена система електронної залікової книжки студента. Вона має розділення на ролі, дозволяє реєструвати нових користувачів, створювати предмети та додавати й переглядати оцінки по ним. При цьому вся інформація про предмети та оцінки зберігається в блокчейні.

Було розглянуто основні принципи роботи блокчейну та доведено, що його можна використовувати в справжніх проектах. Потрібно зауважити і деякі мінуси з досвіду розробки даної системи: дані зберігаються в нереляційному форматі (неможливо ефективно з’єднувати різні колекції) та швидкість запису даних помітно менша за звичайну базу даних.

На мою думку, використання блокчейну всередині однієї структури є малодоцільним. Найкраще він проявляє себе коли декілька організацій хочуть створити спільне сховище даних і жодна з них не хоче покладатися на іншу. Проте, навіть всередині однієї організації блокчейн може додати елемент прозорості в систему.

Тож, мета й завдання курсової роботи виконані.

Список літератури

[1] Я. Корнієнко, «Втратити квартиру і не знати про це. Як аферисти і суди позбавляють киян житла», 2019. [Online]. Доступний у: https://www.epravda.com.ua/publications/2019/08/22/650865/

[2] O. Mazonka, «What is Blockchain: a Gentle Introduction», 2016. [Online]. Доступний у: https://www.researchgate.net/publication/311572122_What_is_Blockchain_a_Gentle_Introduction

[3] «Hyperledger Fabric - Identity». [Online]. Доступний у: https://hyperledger-fabric.readthedocs.io/en/release-2.2/identity/identity.html

[4] B. Safak Zirhli, «Consensus Algorithms in Blockchain», 2020. [Online]. Доступний у: https://www.researchgate.net/publication/341626342_Consensus_Algorithms_in_Blockchain

[5] «Confirmation». [Online]. Доступний у: https://en.bitcoin.it/wiki/Confirmation

[6] «Consensus». [Online]. Доступний у: https://substrate.dev/docs/en/knowledgebase/advanced/consensus

[7] «Cambridge Bitcoin Electricity Consumption Index». https://cbeci.org/cbeci/comparisons

[8] J. Frankenfield, «Proof of Stake (PoS)». https://www.investopedia.com/terms/p/proof-stake-pos.asp

[9] «What is Proof of Authority?» https://www.coinhouse.com/coinhouse-academy/blockchain/what-is-proof-of-authority/

[10] «Wikipedia - Програмний каркас». [Online]. Доступний у: https://uk.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BD%D0%B8%D0%B9_%D0%BA%D0%B0%D1%80%D0%BA%D0%B0%D1%81

[11] «Forbes Blockchain 50». https://www.forbes.com/sites/michaeldelcastillo/2019/04/16/blockchain-50-billion-dollar-babies/?sh=5650fefe57cc

[12] «Hyperledger Fabric - Key Concepts». [Online]. Доступний у: https://hyperledger-fabric.readthedocs.io/en/release-2.2/blockchain.html

[13] J. Chiu, «The Design Philosophy of Hyperledger Fabric». https://medium.com/bsos-taiwan/the-design-philosophy-of-hyperledger-fabric-acb1db99be5e

[14] «The Ordering Service». [Online]. Доступний у: https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html

[15] «Hyperledger Fabric - Private data». [Online]. Доступний у: https://hyperledger-fabric.readthedocs.io/en/release-2.2/private-data/private-data.html

[16] «Hyperledger Fabric - Blockchain network». [Online]. Доступний у: https://hyperledger-fabric.readthedocs.io/en/release-2.2/network/network.html

[17] «Hyperledger Fabric - Membership Service Provider (MSP)». [Online]. Доступний у: https://hyperledger-fabric.readthedocs.io/en/release-2.2/membership/membership.html

[18] «Hyperledger Fabric - Policies». [Online]. Доступний у: https://hyperledger-fabric.readthedocs.io/en/release-2.2/policies/policies.html

[19] G. Arnicans і G. Karnitis, «Migration of Relational Database to Document-Oriented Database: Structure Denormalization and Data

Transformation», представлена на 2015 7th International Conference on Computational Intelligence, Communication Systems and Networks, 2015.

[20] «Hyperledger Fabric - Using CouchDB». [Online]. Доступний у: https://hyperledger-fabric.readthedocs.io/en/release-2.2/couchdb_tutorial.html

[21] «Hyperledger Fabric - Deploying a smart contract to a channel». [Online]. Доступний у: https://hyperledger-fabric.readthedocs.io/en/release-2.2/deploy_chaincode.html

[22] «Hyperledger Fabric - Registering and enrolling identities with a CA». [Online]. Доступний у: https://hyperledger-fabric-ca.readthedocs.io/en/latest/deployguide/use_CA.html

Додаток А

Файл configtx.yaml

Визначення політик для членів організації (на основі навчальних матеріалів Hyperledger Fabric)

```
Organizations:
        Name: Org1MSP

        # ID to load the MSP definition as
        ID: Org1MSP

        MSPDir: ../organizations/peerOrganizations/org1.example.com/msp

        # Policies defines the set of policies at this level of the config
tree
        Policies:
            Readers:
                Type: Signature
                Rule: "OR('Org1MSP.admin', 'Org1MSP.peer', 'Org1MSP.client')"
            Writers:
                Type: Signature
                Rule: "OR('Org1MSP.admin', 'Org1MSP.client')"
            Admins:
                Type: Signature
                Rule: "OR('Org1MSP.admin')"
            Endorsement:
                Type: Signature
                Rule: "OR('Org1MSP.peer')"
```

Визначення правил типу ImplicitMeta

```
Application: &ApplicationDefaults

    Organizations:

    # Policies defines the set of policies at this level of the config tree
    # For Application policies, their canonical path is
    #   /Channel/Application/<PolicyName>
    Policies:
        Readers:
            Type: ImplicitMeta
            Rule: "ANY Readers"
        Writers:
            Type: ImplicitMeta
            Rule: "ANY Writers"
        Admins:
            Type: ImplicitMeta
            Rule: "MAJORITY Admins"
        LifecycleEndorsement:
            Type: ImplicitMeta
            Rule: "MAJORITY Endorsement"
        Endorsement:
            Type: ImplicitMeta
            Rule: "MAJORITY Endorsement"

    Capabilities:
        <<: *ApplicationCapabilities
```

Додаток Б

Визначення моделі даних. Файл recordBookState.ts

```
import { Object, Property } from "fabric-contract-api";

@Object()
export class RecordBook {
  @Property()
  public email!: string;
  @Property()
  public periods!: Period[];
}

@Object()
export class Period {
    @Property()
    year!: number;
    @Property()
    num!: number;
    @Property()
    marks!: Mark[];
}

@Object()
export class Mark {
  @Property()
  public subject!: string;
  @Property()
  public teacher!: string;
  @Property()
  public mark!: number;
}
```

Додаток В

Визначення чейнкоду залікової книжки. Файл recordBookContract.ts (частково)

```
import {
  Context,
  Contract,
  Info,
  Transaction,
} from "fabric-contract-api";
import { Iterators } from "fabric-shim";
import { Query } from "./query";
import { RecordBook, Period } from "./recordBookState";
import { SubjectInfo } from "./subjectInfo";

@Info({ title: "RecordBook", description: "Smart contract for student book
s" })
export class RecordBookContract extends Contract {
  private static getStudentKey(ctx: Context, id: string): string {
    return ctx.stub.createCompositeKey("student", [id]);
  }

  private static getSubjectKey(ctx: Context, name: string, year: number, p
eriod: number): string {
    return ctx.stub.createCompositeKey("subject", [year.toString(), period
.toString(), name]);
  }

  @Transaction()
  public async CreateBook(ctx: Context, email: string) {
    const book: RecordBook = {
      email,
      periods: [],
    };

    await RecordBookContract.SaveBook(ctx, book);
  }


  @Transaction()
  public async AddSubject(ctx: Context, subject: string, year: number, per
iod: number) {
    const subjectInfo: SubjectInfo = {
      subject,
      year,
      period,
      students: [],
      teachers: []
    };
```

```
    await RecordBookContract.SaveSubject(ctx, subjectInfo);
  }

  @Transaction(false)
  public async ReadBook(ctx: Context, email: string): Promise<RecordBook>
{
    const bookJSON = await ctx.stub.getState(
      RecordBookContract.getStudentKey(ctx, email)
    );
    if (!bookJSON || bookJSON.length === 0) {
      throw new Error(`The book of student ${email} does not exist`);
    }
    return JSON.parse(bookJSON.toString()) as RecordBook;
  }

  @Transaction(false)
  public async ReadSubject(ctx: Context, subject: string, year: number, pe
riod: number): Promise<SubjectInfo> {
    const subjectJSON = await ctx.stub.getState(
      RecordBookContract.getSubjectKey(ctx, subject, year, period)
    );
    if (!subjectJSON || subjectJSON.length === 0) {
      throw new Error(`The subject ${subject} does not exist`);
    }
    return JSON.parse(subjectJSON.toString()) as SubjectInfo;
  }

  private static async SaveBook (ctx: Context, book: RecordBook) {
    await ctx.stub.putState(
      RecordBookContract.getStudentKey(ctx, book.email),
      Buffer.from(JSON.stringify(book))
    );
  }

  private static async SaveSubject (ctx: Context, subject: SubjectInfo) {
    await ctx.stub.putState(
      RecordBookContract.getSubjectKey(ctx, subject.subject, subject.year,
 subject.period),
      Buffer.from(JSON.stringify(subject))
    );
  }
  @Transaction()
  public async AddMark(
    ctx: Context,
    studentEmail: string,
    year: number,
    period: number,
    subject: string,
    mark: number,
    teacher: string
```

```
  ): Promise<void> {
    RecordBookContract.ensureUniversityStaff(ctx);
    await this._ensureTeacherStudentSubject(ctx, teacher, studentEmail, su
bject, year, period);

    let book = await this.ReadBook(ctx, studentEmail);

    let periodInfo = book.periods.find(
      (p) => p.year === year && p.num === period
    );
    if (!periodInfo) {
      periodInfo = { year, num: period, marks: [] };
      RecordBookContract.insertSorted(
        book.periods,
        periodInfo,
        RecordBookContract.periodComparator
      );
    }

    if (periodInfo.marks.some(m => m.subject === subject))
      throw new Error(`Student ${studentEmail} already has mark from subje
ct ${subject}.`)


    periodInfo.marks.push({ subject, teacher, mark });

    await RecordBookContract.SaveBook(ctx, book);
  }

private async _ensureTeacherStudentSubject(
    ctx: Context,
    teacher: string,
    student: string,
    subject: string,
    year: number,
    period: number
  ) {
    const subjectInfo = await this.ReadSubject(ctx, subject, year, period)
;
    if (!subjectInfo.teachers.some((s) => s === teacher))
      throw new Error(
        `Teacher ${teacher} isn't enrolled to the subject ${subject}.`
      );

    if (!subjectInfo.students.some((s) => s === student))
      throw new Error(
        `Student ${student} isn't enrolled to the subject ${subject}.`
      );
  }
```

```
private static ensureUniversityStaff(ctx: Context) {
    if (ctx.clientIdentity.getMSPID() !== "Org1MSP")
      throw new Error("You do not have access to this function");
  }
```

Додаток Г

Складний запит до бази даних CouchDB

```
const query: Query = {
      selector: {
        periods: {
          $elemMatch: {
            $and: [
              { year: year },
              { period: period },
              {
                marks: {
                  $elemMatch: {
                    subject: subject,
                  },
                },
              },
            ],
          },
        },
      },
      useIndex: "subjectStudentsDoc",
    };
    const response = await ctx.stub.getQueryResult(JSON.stringify(query));
```

Додаток Ґ

Визначення індексу CouchDB

```
{
    "index": {
      "fields": [
        "students"
      ]
    },
    "ddoc": "subjectStudentsDoc",
    "name": "subjectStudents",
    "type": "json"
  }
```

Додаток Д

Визначення класу CAService (Частково)

```
import { Injectable } from '@nestjs/common';
import { Wallet, Wallets } from 'fabric-network';
import organizations from 'config/organizations.json';
import adminCredentials from 'config/adminCredentials.json';
import { channels } from './channel-info';
import path from 'path';
import FabricCAServices from 'fabric-ca-client';

@Injectable()
export class CAService {
  public wallets: Record<string, Wallet> = {};

  constructor() {
    for (const [orgName, org] of Object.entries(organizations)) {
      Wallets.newFileSystemWallet(
        path.join(process.cwd(), 'identities', orgName),
      )
        .then((w) => (this.wallets[orgName] = w))
        .then(async (w) => {
          const client = this.buildCAClient(channels[orgName], org.caHostN
ame);
          await this.ensureAdmin(client, w, org.orgMspId);
        });
    }
  }

  async registerUser(login: string, organization: string, role: string) {
    const org = organizations[organization];
    const channel = channels[organization];

    const client = this.buildCAClient(channel, org.caHostName);

    await this.registerAndEnrollUser(
      client,
      this.wallets[organization],
      org.orgMspId,
      login,
      org.affiliation,
      role,
    );
  }

  buildCAClient(
    ccp: Record<string, any>,
    caHostName: string,
  ): FabricCAServices {
    const caInfo = ccp.certificateAuthorities[caHostName];
```

```
    return new FabricCAServices(
      caInfo.url,
      { trustedRoots: caInfo.tlsCACerts.pem, verify: false },
      caInfo.caName,
    );
  }
}
```

Додаток Е

Визначення класу ContractService

```
import { Injectable } from '@nestjs/common';
import { CAService } from 'src/ca/ca.service';
import { channels } from 'src/ca/channel-info'
import { Gateway } from 'fabric-network';



@Injectable()
export class ContractService {
  constructor(private caService: CAService) {}

  private getOptions(identity: string, organization: string) {
    return {
      identity,
      wallet: this.caService.wallets[organization],
      discovery: { enabled: true, asLocalhost: true },
    };
  }

  public async getContract(identity: string, organization: string) {
    const options = this.getOptions(identity, organization);
    const gateway = new Gateway();
    await gateway.connect(channels[organization], options);
    const network = await gateway.getNetwork('mychannel');
    const contract = network.getContract('record_book');
    return contract;
  }
}
```

Додаток Є

Визначення класу RecordbookService (частково).

```
@Injectable()
export class RecordbookService {

  constructor(private contracts: ContractService){}
  async addMark(
    identity: string,
    studentEmail: string,
    year: number,
    period: number,
    subject: string,
    mark: number,
  ) {
    const contract = await this.contracts.getContract(identity, 'org1');

    await contract.submitTransaction(
      'AddMark',
      studentEmail,
      year.toString(),
      period.toString(),
      subject,
      mark.toString(),
      identity,
    );
  }

  async getBook(email: string): Promise<RecordBook> {
    const contract = await this.contracts.getContract(email, 'org2');

    const recordResponse = await contract.evaluateTransaction(
      'ReadBook',
      email,
    );
    return JSON.parse(recordResponse.toString()) as RecordBook;
  }
}
```