КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет комп'ютерних наук та кібернетики

Кафедра математичної інформатики

Кваліфікаційна робота

на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки

на тему:

Побудова серверної частини та розгортання веб-застосунку для моніторингу пожертв волонтерам.

(Building server and deploying web application for real-time charity tracking for volunteers)

Виконав студент 4-го курсу
Цимбал Костянтин Олександрович

(підпис)

Науковий керівник:
Асистент, кандидат технічних наук
Федорус Олексій Мстиславович

(підпис)

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань

Студент

(підпис)

Київ – 2022

РЕФЕРАТ

Обсяг роботи 40 сторінок, 17 рисунків, 15 джерел посилання.

Назва роботи: побудова серверної частини веб-застосунку для публічного моніторингу некомерційних банківських рахунків зі зберіганням актуальної інформації в блокчейн.

Мета роботи: розробити надійну та підтримувану серверну частину додатку. Створити безпечну та зручну інфраструктуру у хмарі та розгорнути сервер. Додати інтеграції з блокчейном та графічною частиною застосунку.

План роботи: дослідити існуючі архітектури та найкращі практики створення серверної частини застосунку для інтеграції з блокчейном. Дослідити існуючі фреймворки. Знайти найкраще хмарне рішення. Реалізувати та розгорнути застосунок.

Результат роботи: робоча серверна частина застосунку, яку інтегровано з графічною частиною та з блокчейном. Застосунок розгорнуто на AWS та зроблено доступним у публічному інтернеті за посиланням.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

gRPC – Google Remote Procedure Call

API - Application Programming Interfac

AWS - Amazon Web Services

APM – Application Portfolio Management

ПЗ – Програмне забезпечення

Проект – ціль для збору коштів

ВСТУП

Оцінка сучасного стану об’єкта розробки.

Часто людям потрібно зібрати велику суму для тієї, або іншої цілі. Таких людей можна часто зустріти у житті: на вулиці, у метро, тощо. Часто, у них є якась табличка, або документи які підтверджують благородність мети, однак немає ні гарантій того, що ці гроші справді витраться з вказаною ціллю, ні того, що у людини справді є потреба. Також, не відомо, скільки грошей вже зібрано, а тому невідомо, скільки залишилось. Немає можливості відправити карткою, або крипто валютою.

З розвитком технологій, такі люди перекочували у інтернет. Я помічав відповідну рекламу перед переглядом відео на “ютубі” або у “Інстаграмі”. І, знову ж таки, ми бачимо схожі проблеми, що немає ні того, скільки коштів зібрано, ні гарантії того, що їх правильно витратять. Моментами, навіть небезпечно вводити данні про свою картку на невідомий сайт.

Це все призводить до того, що люди, які можуть надіслати гроші, мають багато вищевказаних сумнівів. Таким чином, чесні особи, яким справді необхідна така допомога, отримують менше коштів, або загалом не отримують їх, адже до них менше довіри.

Ці доводи стосуються фізичних осіб, проте не варто забувати і про юридичних осіб, до яких мало би бути більше довіри. Адже завжди можна перевірити на кого він зареєстрований, дізнатись баланс тощо. Але людині, яка хоче пожертвувати кілька сотень гривень, скоріше за все буде лінь це робити або вона просто не вміє.

Варто зазначити, що на цьому всьому можуть наживатись недобросовісні шахраї[1][2]. І це ще більш підриває довіру людей

Актуальність роботи та підстави для її виконання.

Сьогодні працюють багато волонтерів, які бажають допомагати нашій армії. Багато людей хочуть полегшити життя переселенців та інших людей, які потрапили у халепу. Через це, попит на збір коштів сильно виріс. Майже кожен другий збирав

кошти, або збирає їх зараз. Це все відбувається через історії у “Інстаграм”, або, у найкращому випадку створюється доволі примітивний сайт для збору з коротким описом та номером картки для пожертви.

Є й великі фонди, та можна напряму пожертвувати державі. Проте, у деяких випадках, хочеться точково допомогти волонтерам, які швидко все організують і доставлять. Тип паче, є якесь моральне задоволення, коли допомагаєш своїм знайомим.

Під час дослідження ринку, ми зрозуміли, що зараз немає структурованого застосунку, де волонтери могли би викладати свої потреби, а бажаючі могли би та фільтрувати їх за своїми правилами. А потім зручно надсилати гроші, та бути впевненими, що кошти точно дійдуть до виконавцю та будуть витрачені з благою ціллю.

Мета й завдання роботи. Метою кваліфікаційної роботи є розробка надійної та підтримуваної серверної частини додатку, яку буде зручно інтегрувати з графічною частиною та блокчейном у вигляді смарт контракту. Ще однією метою було розгортання усієї інфраструктури на AWS та забезпечення її надійності. Задля успішного виконання роботи, було зроблено:

- Дослідження існуючих архітектур серверної частини
- Дослідження найкращих практик розробки серверної частини(безпека та підримуванність)
- Дослідження існуючих фреймворків та вибір найкращого рішення
- Дослідження існуючих хмарних рішень для розгортання застосунку та вибір найкращого
- Реалізація застосунку
- Зберігання даних у БД
- Реалізація інтеграцій
- Розгортання на хмарі
- Тестування

Об’єкт, методи й засоби розробки.

Методологією розробки забезпечення було екстремальне програмування

Можливі сфери застосування.

Застосунок може бути використаним в усіх сферах, де необхідно збирати кошти. Це не обмежуються лише благодійністю. Можна використати як аналог kickstarter[3], але зі своїми гарантіями.

РОЗДІЛ 1. ВИБІР РІШЕННЯ ДЛЯ РОЗРОБКИ СЕРВЕРНОГО ДОДАТКУ

1.1 Вибір архітектури

Архітектура додатку описує поведінку програм(так звану бізнес логіку), взаємодії одного додатку з іншим та з кінцевими користувачами. Її фокус знаходиться на даних, які споживаються та створюються застосунками, а не на внутрішній структурі додатків(коді). У АРМ програми зіставляються з корисними для бізнесу функціями та процесами, для яких було створено ту чи іншу програму. Також враховують витрати, функціональну та технічну якість. Це необхідно для оцінки наданої користі.

Саму архітектуру визначають бізнес та функціональні вимоги. Передбачено взаємодію між базами даних, проміжними системами та пакетами програм з функціональної точки зору. Гарна архітектура допоможе ще на етапі розробки виявити прогалини у функціональності та проблеми інтеграції. Після цього буде можливість скласти плани міграції для відповідних систем. Такі плани не принесуть збитків, або збити буде мінімізовано.

Архітектура, у розрізі великою корпорації, повинна забезпечувати надійність, доступність та масштабування усіх додатків.

Архітектура додатків визначає, як вони будуть працювати разом. Не варто плутати її з архітектурою ПЗ, яка відповідає за технічні деталі будування окремої системи.

Архітектура ПЗ, у свою чергу, відноситься до фундаментальних структур програмної системи та дисципліни створення таких структур і систем. Кожна структура містить програмні елементи, зв'язки між ними, властивості як елементів, так і відносин.[4] Така архітектура є метафорою до архітектури будинку.

Архітектура ПЗ полягає в тому, щоб на початкових етапах розробки зробити фундамент програми, адже потім його буде дуже дорого змінювати. Вибір архітектури включає у себе структурні особливості системи та потреби бізнесу. Наприклад, автопілот для автомобіля Тесла повинен дуже швидко працювати, бути надійним та стійким до помилок. Для цього необхідно вибрати мову, яка дозволяє

робити швидкі обчислення у режимі реального часу. Для надійності та стійкості до помилок можна запустити кілька копій, та порівнювати результати у реальному часі.

Важливо не забувати і про документацію архітектури. Це полегшує спілкування, робить менш складним адаптацію нових співробітників та дозволяю перевикористовувати незалежні модулі.

1.1.1 Цибулева архітектура(Onion Architecture)

Більшість традиційних моделей архітектури проблеми тісного зв'язку та розділення проблем, які є фундаментальними. Джеффрі Палермо запропонував використовувати Onion Architecture, щоб забезпечити новий спосіб створення застосунків з точки зору надійності, підтримуванності та можливості тестування. Вона вирішує проблеми, з якими стикаються застосунку, у яких більше 3-х рівнів та пропонує відповіді на більш загальні питання. Рівні у цій архітектурі взаємодіють за допомогою абстракій(інтерфейсів)

Принципи

Цибулева архітектура базується на принципі інверсії контролю(inversion of control). Сама архітектура складається з кількох(3 та більше) концентричних шарів. Вона також обмежує взаємодію між шарами у напрямку до ядра, яке являє собою сутності. Вона відрізняється від класичних багаторівневих архітектур тим, що залежить не від рівня даних, а від моделей.

Рівні

Onion Architecture використовує концепцію шарів. Однак вони відрізняються від класичних 3-рівневих та n-рівневих архітектурних шарів. Повинні бути такі рівні:

Доменний рівень

У ядрі існує шар домену, який представляє самі дані. Суть у тому, щоб усі сутності були у цьому ядрі. Також туди можна додати інтерфейси та мапери. Не буде грубою помилкою, якщо винести це у окремий рівень. Ці об'єкти не повинні мати жодних

залежностей, окрім інших сутностей, або інтерфейсів.

Рівень репозиторіїв

Цей рівень відповідає за абстракції між сутностями у ядрі та бізнес-логікою застосунку. На цьому рівні можна додати, вказані вище перетворювачі та інтерфейси. Цей рівень відповідає за роботу з сутностями, проте потрібно мінімізувати специфічні деталі реалізації роботи з даними. Цей рівень все ще не містить логіки

Рівень сервісів

Цей рівень працює зі звичайними для користовуча, такими як: створення, видалення, редагування, тощо. Він же і відповідай за зв'язок з UI. Цей рівень містить бізнес логіку. Він залежить від абстрактних інтерфейсів репозиторія, а не від самих репозиторіїв.

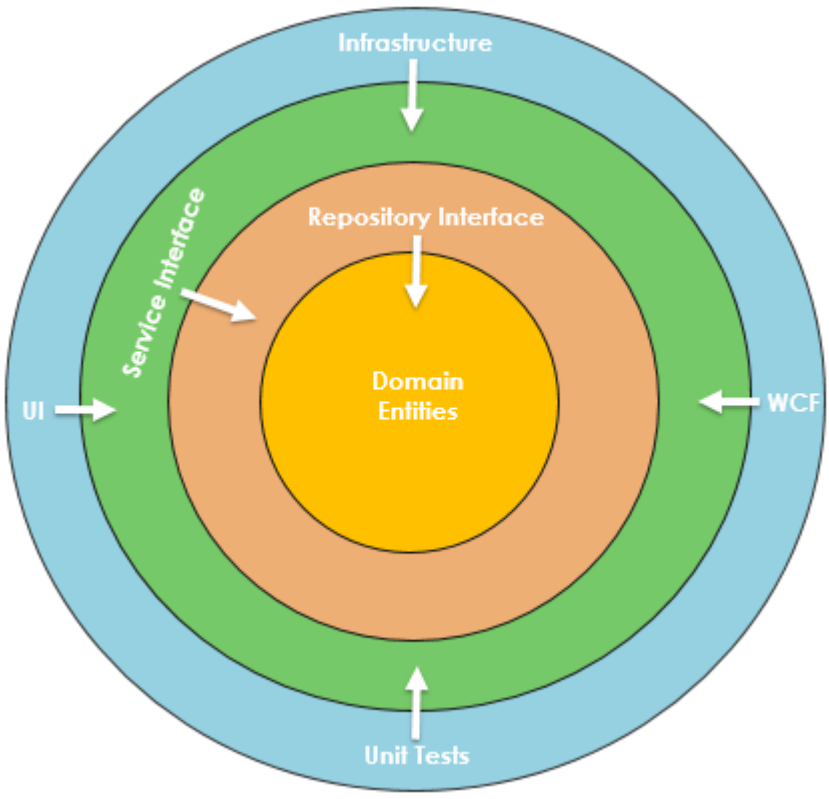


Рисунок 1.1.1.1 – Шари архітектури[5]

1.1.2 Гексагональна архітектура(Hexagonal architecture)

Була створена Алістером Кокберном у 2005 році, проте широкий інтерес та

застосування викликала лише у 2015 році. Вона була створена з такою метою: Дозволити програмі однаково управляти користувачами, програмами, автоматизованими тестовими або пакетними сценаріями, а також розробляти й тестувати ізольовано від пристроїв і баз даних, які можуть виконуватися[7]

Принципи

1. Відділення юзерів(User-side) від бізнес логіки(Business logic) та серверної частини(Server-side).

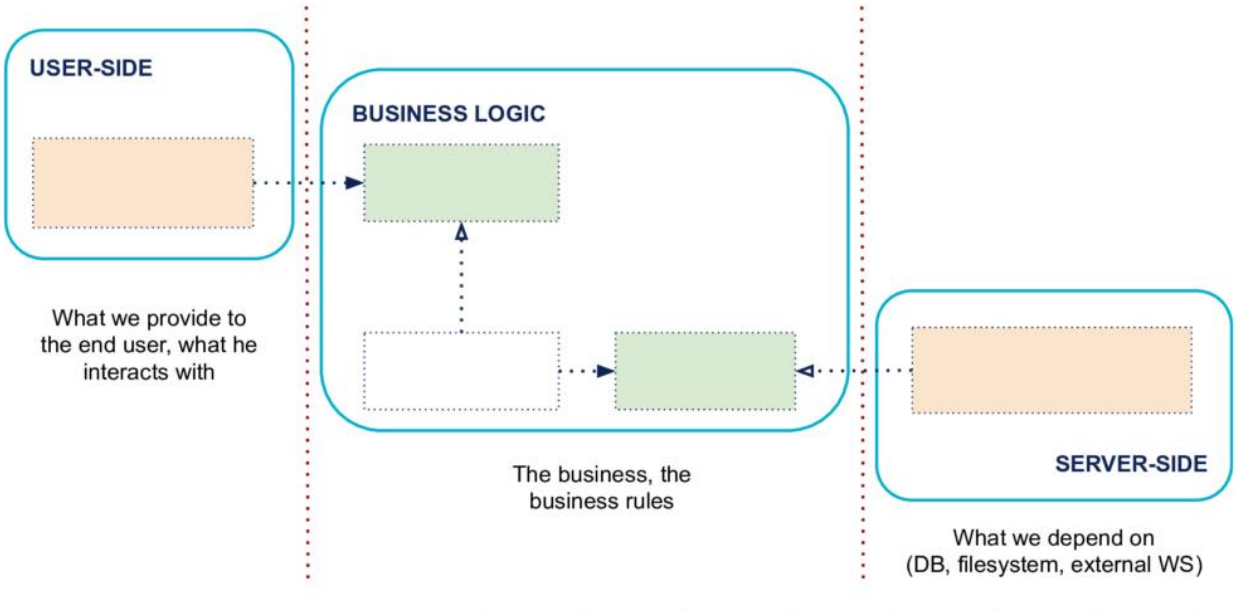


Рисунок 1.1.2.1 – Приклад розділення [6]

2. Залежності напрямлені в середину

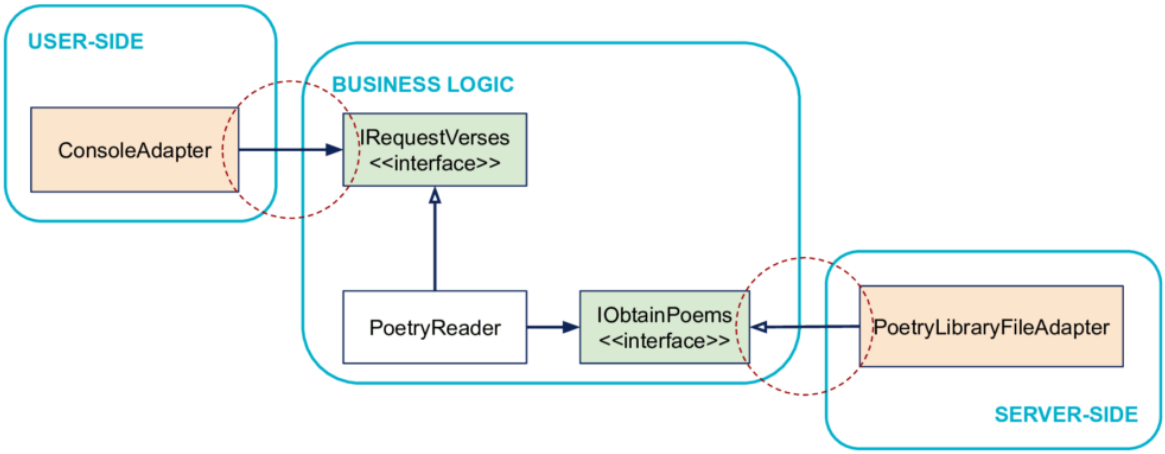


Рисунок 1.1.2.2 – Приклад напрямку залежностей [6]

3. Кордони ізольовано інтерфейсами

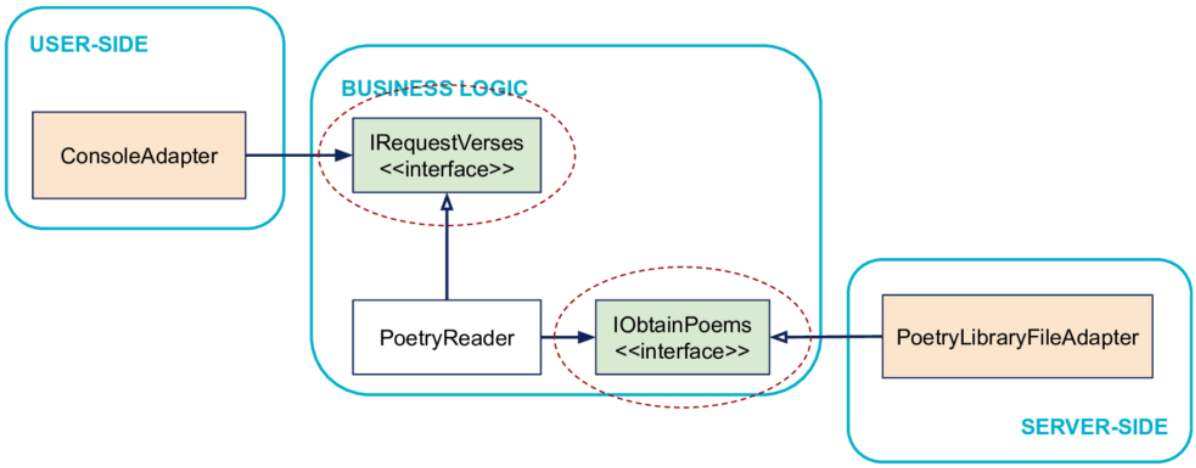


Рисунок 1.1.2.2 – Приклад ізоляції [6]

Рівні

1. Сутності(Entities) – це об’єкти домени, які не знають про те, як та де вони зберігаються.
2. Репозиторії (Repositories) – це інтерфейси для оновлення та створення сутностей. Вони зберігають методи для роботи з сутностями та повертають сутності або масив сутностей

3. Interactors(інтерактори) – іпмлементують бізнес логіку та перевірку.
4. Data Sources – адаптери для різних імплементацій роботи з даними. Це може бути як SQL база даних, так і якийсь примітивний CSV файл. На цьому рівні може міститись логіка для обробки даних(наприклад хешування)
5. Transport Layer – використовуються для того, щоб повідомити інтеракторам про необхідність виконання тієї, чи іншої бізнес логіки. Найбільш поширеним прикладом у мікросервісній архітектурі буде HTTP API Layer та набір контролерів для обробки запитів. Враховуючи те, що наша бізнес логіка містить у інтеракторах, ми не обмежені у використанні різних транспортних протоколів.

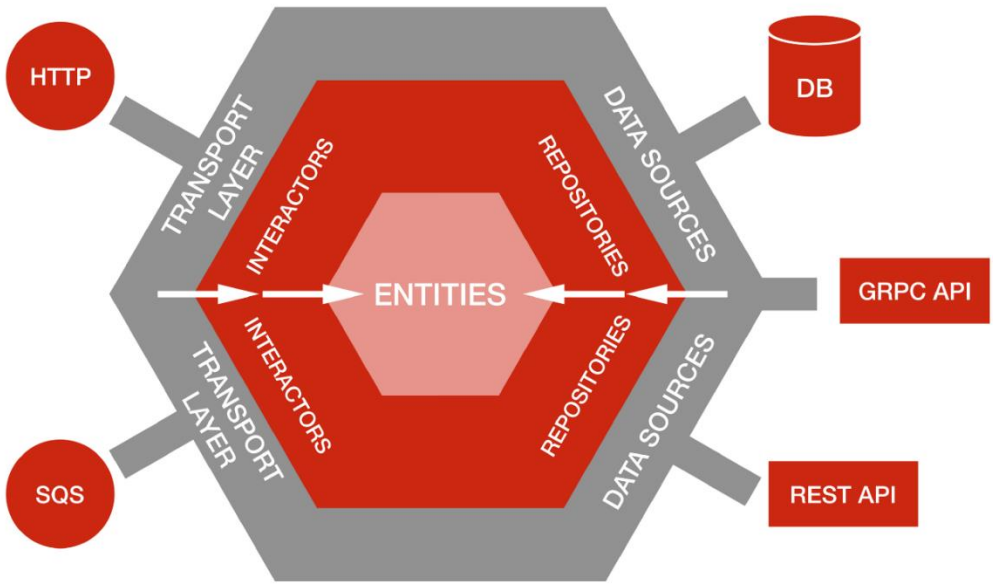


Рисунок 1.1.2.3 – Приклад комунікації між рівнями [8]

1.1.3 Висновок

Проаналізувавши обрані архітектури, я прийшов до висновку, що наші потреби з головою задовольняє цибулева архітектура, адже вона дасть необхідну надійність та структурованість коду, з однієї сторони, а з іншої сторони, її досить швидко можна імплементувати.

1.2 Вибір фреймворку

Я обрав Node.js для розробки серверної частини, адже вона швидка, зручна у розробці, має багато інтеграцій з блокчейном та надійна. Також вона має багато фреймворків для розробки сервера, 2 з яких ми розглянемо

1.2.1 Express

Філософія Express полягає в тому, що він надає невеликі та надійні інструменти для HTTP-серверів. Це робить його гарним рішенням для single-page додатків, звичайних веб-сайтів та навіть для мікросервісів.

Express не обмежує у використанні шаблонізаторів чи ORM. Він підтримує понад 14 шаблонів за допомогою Consolidate.js, тому дає змогу швидко створити свій ідеальний фреймворк.[9]

Із плюсів Express слід відмітити те, що на ньому дуже швидко розробляти, є багато інтеграцій, він мало важить, надійний та протестований роками. Фреймворк лежить у ядрі багатьох інших фреймворків.

Із мінусів: він погано товаришує з тайпскриптом, важко зробити гарну архітектуру. Завжди буде пекло з мідлварів(middleware hell)

1.2.2 Nest.js

В останні роки, завдяки Node.js, JavaScript став «lingua franca» інтернету як для front-end(веб сторінок), так і для серверних додатків. Це породило чудові проекти, такі як Vue, Angular та React, які підвищують продуктивність розробників і дозволяють створювати швидкі, перевірені та розширювані інтерфейси програми. Проте, незважаючи на те, що для Node (і серверного JavaScript) існує безліч чудових бібліотек, допоміжних засобів та інструментів, жодна з них ефективно не вирішує головну проблему — архітектури.

Nest надає готову архітектуру додатків, яка дозволяє розробникам і командам створювати добре перевірені, масштабовані, слабо пов’язані та легко підтримувані програми. Angular послужило головним джерелом для архітектури.[10]

Із плюсів: готова архітектура, яку просто продовжувати розвивати. Це швидкий фреймворк, адже під капотом все ще працює Express або Fastify. Багато готових рішень або інтеграцій, які добре задокументовані.

Із мінусів: це завеликий фреймворк для примітивного сервера. Також багато часу витрачається на налаштування з нуля.

1.2.3 Висновок

Nest.js пропонує готове рішення для цибулевої архітектури, на якій зупинився на попередньому етапі. Також він добре працює з тайпскриптом та інтерфейсами, які необхідні для чистої архітектури. Отже, я зупинився на ньому

РОЗДІЛ 2. РЕАЛІЗАЦІЯ ЗАСТОСУНКУ

2.1 Визначення

Опенсорсне рішення – рішення з відкритим програмним кодом, яке підтримується волонтерами та компаніями.

Кросс платформне рішення – рішення, яке не залежить від середовища запуску та має гарантії, що воно буде однаково працювати на найпопулярніших операційних системах.

Node.js – це опенсорсне, кросс платформне рішення для виконання Javascript поза браузером.

Javascript – мова програмування.

Тайпскрипт(Typescript) – це сильно типізована мова програмування, яка базується на Javascript.

Фреймворк – це абстракція, яка надає готове вирішення найпопулярніших базових проблем та може бути модифікована розробником.

Мапер(перетворювач) – функція, яка перетворює один об’єкт у інший.

Контролер – набір обробників для того чи іншого HTTP запиту.

Гейтвей – набір обробників, який допомагає передавати дані між різними мережами. У контексті цієї роботу, гейтвей повинен передавати дані по протоколу, відмінному від HTTP.

Cross-Origin Resource Sharing (CORS) – механізм, який дозволяє серверу вказувати будь-які джерела походження інформації(крім власного).

Вебхук – це варіант комунікації між різними серверами. Один сервер підписується на інший і після до першого сервера будуть приходити відповідні

Ендпоінт – веб адреса, за якою користувачі можуть отримати доступ до відповідного ресурсу

2.2 Архітектура

Застосунок створено на основі Node.js з використанням Тайпскрипта та фреймфорку Nest.js[10].
База даних: PostgreSQL[11]
Для роботи з базою даних використовується TypeORM

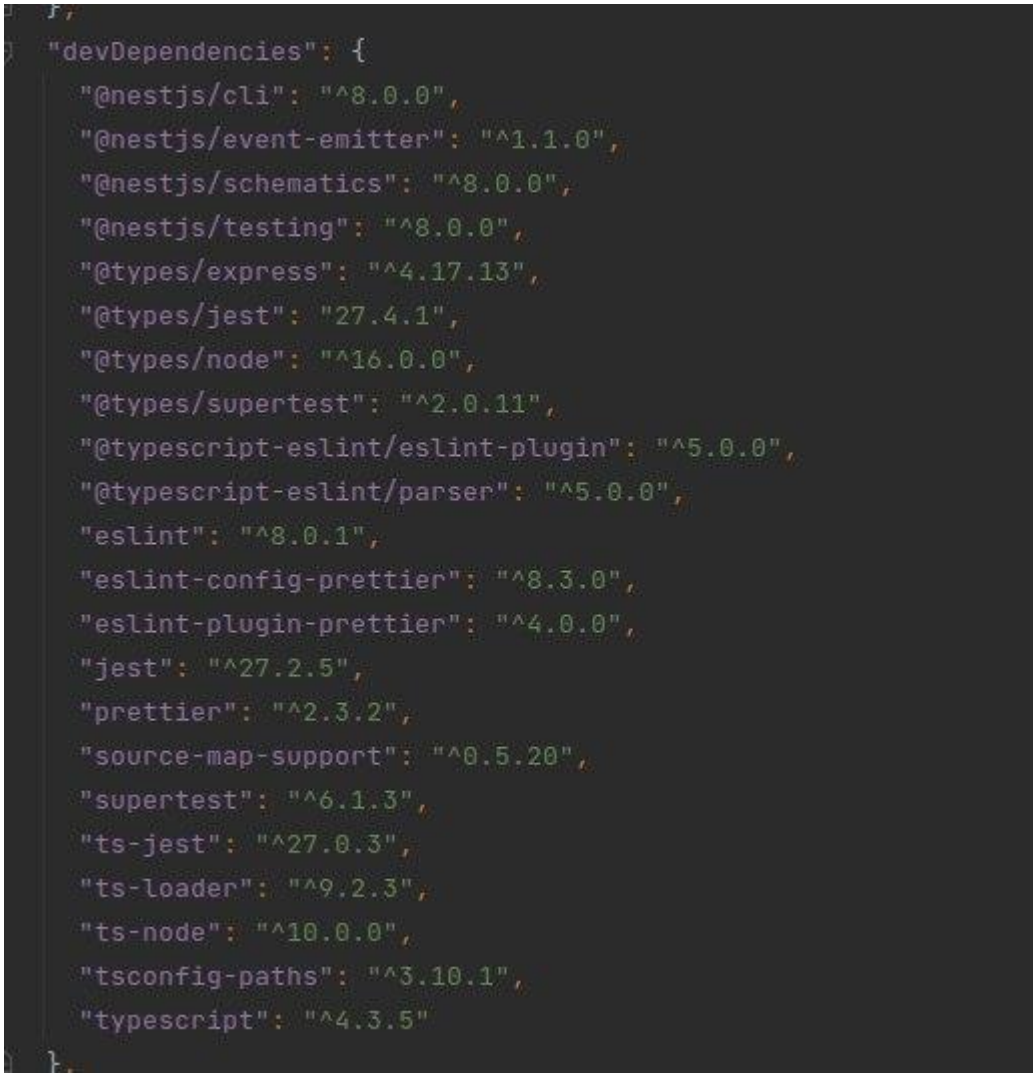


Рисунок 2.2.1 Залежності додатку

Проект має цибулеву архітектуру

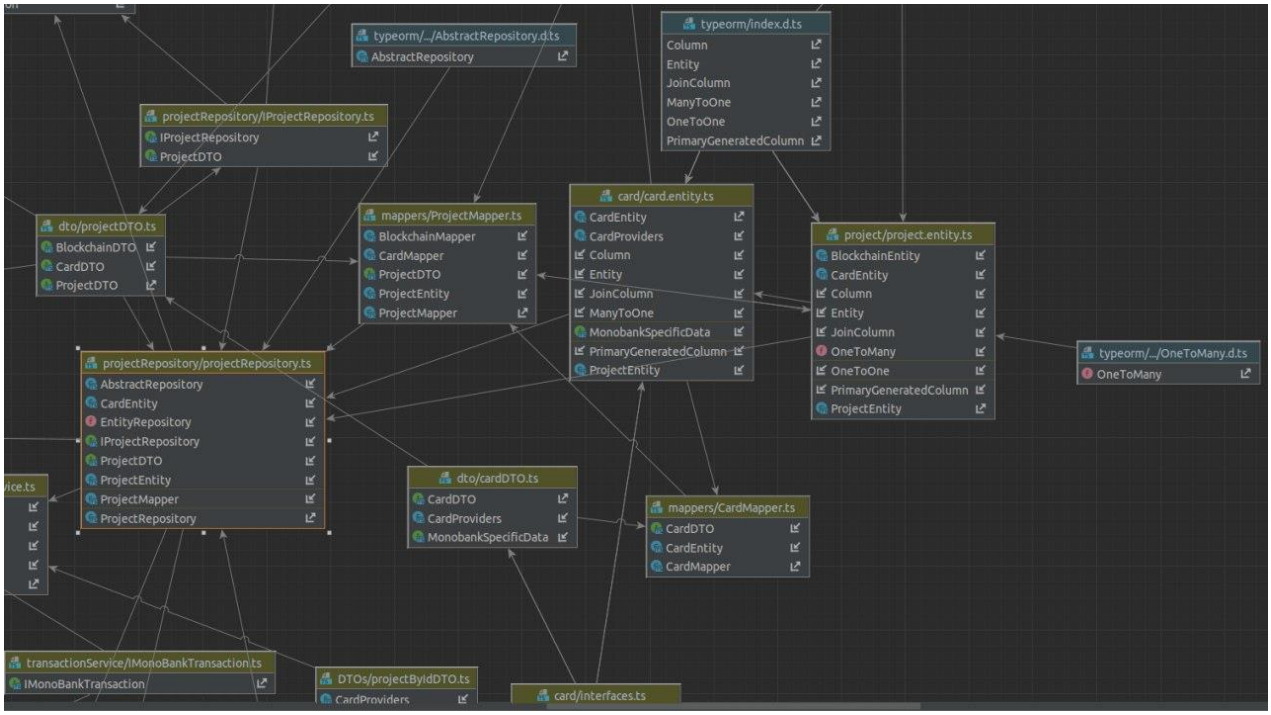


Рисунок 2.2.2 Напрямок залежностей у проекті

Під час розробки були використано такі дизайн патерни:

Singleton – для того, щоб не створювати окреме з’єднання до БД на кожний запит, використовуємо цей патерн та зберігаємо з’єднання у пам'яті процесу.

Dependency injection – зроблено для зручності підміни залежностей за необхідності

Factory method – для створення серверу

У програмі використовуються сокети для зв’язку з графічним інтерфейсом у режимі реального часу. Ця частина базується на подіях: тобто, ми відправляємо подію, який слухає наш додаток. Коли додаток отримує подію, він у свою чергу повідомляє графічному інтерфейсу про подію.

```
@OnEvent(Events.TRANSACTION_INITIATED)
public handleTransactionInitiatedEvent(
  payload: ITransactionInitiatedEvent,
): void {
  const clientData = Object.entries(this.clientProjectHashMap).find(
    ([_clientId :string , projectId :string ]) => projectId === payload.projectId,
  );

  if (clientData) {
    const [clientId] = clientData;

    this.server.to(clientId).emit(SocketEvents.TRANSACTION_INITIATED, args: {
      amount: payload.amount,
      id: payload.id,
    });
  }
}

@OnEvent(Events.TRANSACTION_FINALIZED)
public handleTransactionFinalizedEvent(
  payload: ITransactionFinalizedEvent,
): void {
  const clientData = Object.entries(this.clientProjectHashMap).find(
    ([_clientId :string , projectId :string ]) => projectId === payload.projectId,
  );

  if (clientData) {
    const [clientId] = clientData;

    this.server.to(clientId).emit(SocketEvents.TRANSACTION_FINALIZED, args: {
      id: payload.id,
      transactionSignature: payload.transactionSignature,
    });
  }
}
```

Рисунок 2.2.5 Показано підписку на події

```
async addMonobankTransaction({
  transaction,
  project,
}: {
  transaction: ITransaction;
  project: ProjectDTO;
}): Promise<void> {
  try {
    const transactionSignature =
      await blockchainService.sendTransactionToBlockchain(
        fromJsonToKeypair(project.blockchain.key),
        transaction.amount,
      );

    const eventData: ITransactionFinalizedEvent = {
      projectId: project.id,
      id: transaction.id,
      transactionSignature,
    };

    this.eventEmitter.emit(Events.TRANSACTION_FINALIZED, eventData);
  } catch (e) {
    console.error(e);
  }
}
```

Рисунок 2.2.5 Показано відправлення події

Таким чином модуль, який підписаний на події ніяк не пов'язаний з модулем, який їх надсилає, а тому кожен із цих модулів можна без проблем замінити.

Для роботи з сутностями використовуються мапери, які є репрезентаціями сутності у застосунку. Сутності є репрезентаціями реальних сутностей у базі даних та реальних відносин між сутностями.

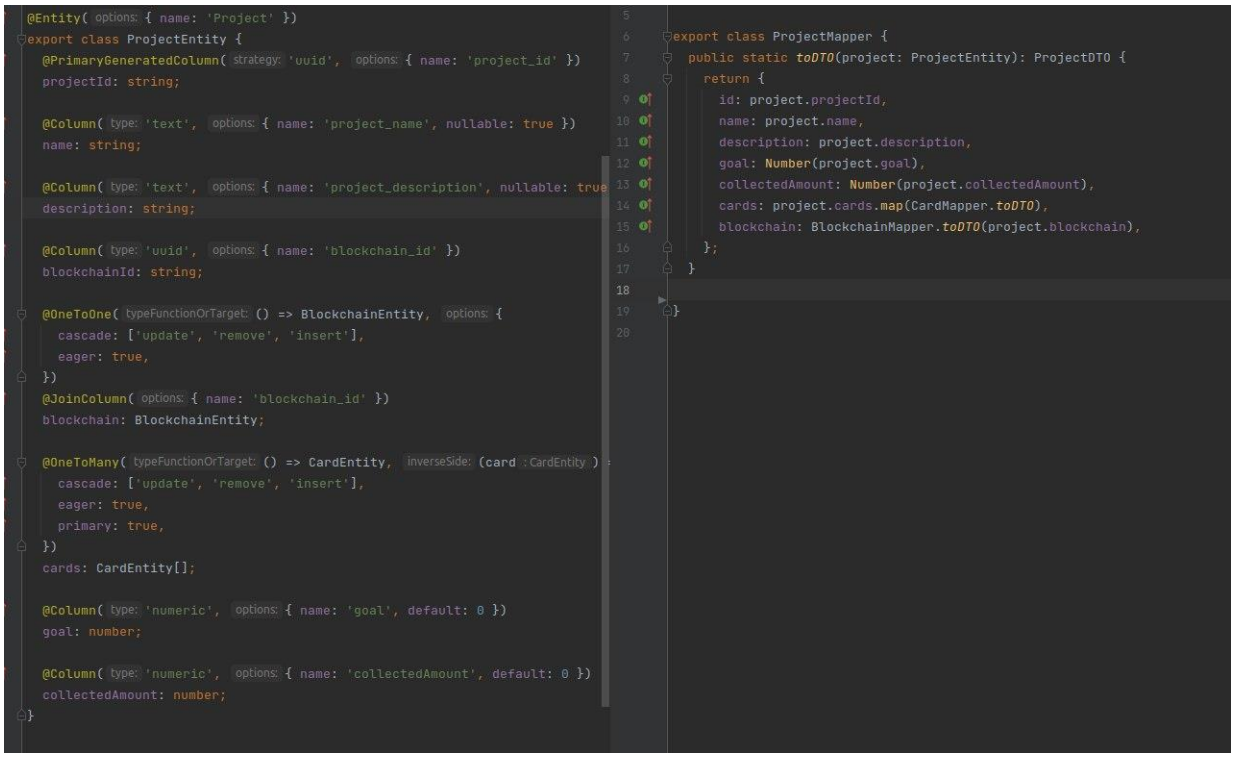


Рисунок 2.2.6 Приклад сутності та відповідного мапера

Слід зазначити, що застосунок також спілкується з блокчейном Solana по протоколу rpc та з Монобанком за допомогою вебхуків(webhook).

2.3 Реалізація

Проект складається з 2 контролерів, 1 гейтвея, 4 незалежних модулів, 3 сутностей та одного репозиторію, який поєднує у собі роботи з усіма сутностями. Основним файлом є main.ts. Він виглядає так:

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import * as morgan from 'morgan';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.enableCors();
  app.use(morgan('tiny'));
```

```
  await app.listen(3000);
}
bootstrap();
```

У цьому файлі ми описуємо основну логіку. Спочатку імпортуємо критичні залежності: AppModule, NestFactory та morgan.

Потім йде основна функція bootstrap, яка відповідає за запуск усього сервера. Першої стрічкою у функції ми створюємо наш застосунок.

Друга стрічка відповідає за те, щоб на сервері використовувався CORS. Це необхідно для спілкування з клієнтом, який розміщено на домені, відмінному від домена сервера.

Третьої стрічкою додаємо логування того, скільки часу було використано на обробку запита.

І п’ята стрічка відповідає за порт, на якому буде доступний сервер.

Програма запускається командою

npm run start

```
kostyatsymbal@kostyatsymbal-Swift-SF314-42:~/Develop/university/OpenCharityTracker/backend$ npm run start

> open-charity-tracker-backend@0.0.1 start /home/kostyatsymbal/Develop/university/OpenCharityTracker/backend
> nest start

[Nest] 30381  - 06/02/2022, 8:55:07 PM     LOG [NestFactory] Starting Nest application...
[Nest] 30381  - 06/02/2022, 8:55:07 PM     LOG [InstanceLoader] TypeOrmModule dependencies initialized +58ms
[Nest] 30381  - 06/02/2022, 8:55:07 PM     LOG [InstanceLoader] DiscoveryModule dependencies initialized +1ms
[Nest] 30381  - 06/02/2022, 8:55:07 PM     LOG [InstanceLoader] AppModule dependencies initialized +0ms
[Nest] 30381  - 06/02/2022, 8:55:07 PM     LOG [InstanceLoader] EventEmitterModule dependencies initialized +0ms
[Nest] 30381  - 06/02/2022, 8:55:08 PM     LOG [InstanceLoader] TypeOrmCoreModule dependencies initialized +524ms
[Nest] 30381  - 06/02/2022, 8:55:08 PM     LOG [InstanceLoader] TypeOrmModule dependencies initialized +1ms
[Nest] 30381  - 06/02/2022, 8:55:08 PM     LOG [InstanceLoader] ProjectModule dependencies initialized +1ms
[Nest] 30381  - 06/02/2022, 8:55:08 PM     LOG [InstanceLoader] WebhookModule dependencies initialized +1ms
[Nest] 30381  - 06/02/2022, 8:55:08 PM     LOG [RoutesResolver] AppController {/}: +15ms
[Nest] 30381  - 06/02/2022, 8:55:08 PM     LOG [RouterExplorer] Mapped {/create, GET} route +3ms
[Nest] 30381  - 06/02/2022, 8:55:08 PM     LOG [RoutesResolver] WebhookController {/webhook}: +1ms
[Nest] 30381  - 06/02/2022, 8:55:08 PM     LOG [RouterExplorer] Mapped {/webhook/monobank, POST} route +0ms
[Nest] 30381  - 06/02/2022, 8:55:08 PM     LOG [RoutesResolver] ProjectController {/project}: +1ms
[Nest] 30381  - 06/02/2022, 8:55:08 PM     LOG [RouterExplorer] Mapped {/project/:id, GET} route +0ms
[Nest] 30381  - 06/02/2022, 8:55:08 PM     LOG [NestApplication] Nest application successfully started +5ms
```

Рисунок 2.3.1 Приклад запуску сервера

Після виконання вказаною команди, бачимо логи від фреймворку Nest.js які вказають на те, що всі залежності успішно створено. Також є список контролерів, які потім зіставляють з відповідним шляхом у HTTP запиті. Остання стрічка каже, що наш додаток успішно запущено.

Розглянемо реалізацію контролера, який використовується для інтеграції з Монобанком:

Починається все з модуля, який описує залежності

```
@Module({
  imports: [TypeOrmModule.forFeature([ProjectRepository])],
  controllers: [WebhookController],
  providers: [
    WebhookService,
    {
      provide: Services.TRANSACTION,
      useClass: TransactionService,
    },(1)
  ],
})
```

Тут ми вказали усі залежності, і після цього можем переходити до реалізації самого контролера:

```
@Controller('webhook')
export class WebhookController {
  constructor(private readonly webhookService: WebhookService) {}

  @Post('monobank')
  @HttpCode(200)
  async processMonobankTransaction(
    @Body() transaction: IMonoBankTransaction,
```

```
  ): Promise<string> {
    await this.webhookService.processMonobankTransaction(transaction);
    return 'ok';
  }
}
```

Тут немає нічого особливого, просто вказали шлях для всього контролера (webhook), тип та шлях для окремого запиту (monobank). Тобто, щоб викликати функцію webhookService.processMonobankTransaction, нам необхідно зробити Post запит на {url}/webhook/monobank.

Цей ендпоінт відповідає за те, щоб після отримання донату, на сервер надійшла транзакція, а сервер у свою чергу повідомив графічній частині про цю транзакцію та записав відповідні дані у блокчейн.

Перейдемо до реалізації сервісу, який відповідає за обробку транзакції.

Сервіс містить лише 1 функцію, тому говорячи сервіс варто думати про функцію та її обгортку.

Цей сервіс варто розділити на 2 частини:

```
@Injectable()
export class WebhookService {
  constructor(
    @InjectRepository(ProjectRepository)
    private projectRepository: ProjectRepository,
    @Inject(Services.TRANSACTION)
    protected _transactionService: ITransactionService,
    private eventEmitter: EventEmitter2,
  ) {}
  async processMonobankTransaction(
    transaction: IMonoBankTransaction,
  ): Promise<void> {
    const isAccountTrackable = transaction.data.account === config.accountId;
    const isRefillTransaction = transaction.data.statementItem.amount > 0;
    if (isAccountTrackable && isRefillTransaction) {
      const relatedProject =
        await this.projectRepository.getProjectByMonobankAccountId(
          transaction.data.account,
        );
```

Рисунок 2.3.2 Показано першу частину реалізації сервісу для обробки вебхуків

Тут можна помітити явне використання шаблону Dependency Injection. Провайдер, який виділено (1) явно вказує що під токеном Services.TRANSACTION буде використовуватись TransactionService. Однак у реалізації Webhook сервісу(стрічки 16 та 17), ми опираємось на абстрактний інтерфейс ITransactionService та токен Services.TRANSACTION. Тобто цей сервіс ніяк не залежить від TransactionService.

Цей сервіс викликає метод у репозиторії:

```
public async getProjectByMonobankAccountId(
  accountId: string,
 ): Promise<ProjectDTO> {
  const project = await this.createQueryBuilder('project')
   .select(['project.project_id'])
   .innerJoin(
    CardEntity,
    'card',
    card.issuer_specific_data ->> 'accountId' = :accountId,(2)
    {
     accountId,
    },
   )
   .getRawOne();

  const projectEntity = await this.repository.findOne({
   projectId: project.project_id,
  });

  return ProjectMapper.toDTO(projectEntity);
 }
```

Цей метод трохи порушує нашу архітектуру, адже він залежить від сутності. Однак це не є грубим порушенням і є необхідним для роботи обраної ORM. Тут цікаво розглянути inner join. Ми робимо об’єднання за умовою (2). Я прийняв рішення зберігати специфічні для банку дані у форматі jsonb, адже формат такого об’єкту буде відрізнятись від банку до банку, а використовуватись він буде лише на рівні репозиторіїв, та ймовірніше всього, у одному запиті. Отже, це є безпечним. У самій умові ми дивимось на колонку issuer_specific_data у таблиці, яка відповідає сутності CardEntity. Обираємо проперті об’єкту issuer_specific_data під назвою accountId та порівнюємо з першим аргументом функції.

Після цього у сервісі відбуваються маніпуляції з даними. Завершується робота функції створенням івента та його розповсюдженням, а також додаванням транзакції у блокчейн:

```
this.eventEmitter.emit(
    Events.TRANSACTION_INITIATED,
    transactionInitiatedEvent,
  );

this._transactionService.addMonobankTransaction({
    transaction: transactionData,
    project: relatedProject,
  });
```

transactionService.addMonobankTransaction відповідає за виклик методу blockchainService.sendTransactionToBlockchain, очікування відповіді від нього, логування та за створення ще однієї події та поширення її на графічну частину через гейтвей.

Розглянемо сам метод у blockchainService:

```
async sendTransactionToBlockchain(account, amount: number): Promise<string> {
```

```
  const lastHash = await this.getLastHash(account);
  const hash = hashString(`${lastHash}${amount}`);
  return this.program.methods
   .addTransaction(${amount}, hash)
   .accounts({
    baseAccount: account.publicKey,
    user: this.provider.wallet.publicKey,
   })
   .rpc();
 }
```

Тут ми отримуємо відповідний хеш за акаунтом, формуємо хеш, який буде враховувати кількість грошей у транзакції та викликаємо метод addTransaction у смарт контракту. Для смарт контракту необхідно передати 2 аккаунти: аккаунт користувача та аккаунт програми. Завершуються ланцюжок викликів, викликом методу rpc(), який і формує сам запит у блокчейн. На цьому завершується робота розглядуваного ендпоінту.

Застосунок ще має 2 ендпоінти:
{url}/create – службовий ендпоінт для створення аккаунт у блокчейні
{url}/project/:id – для отримання даних про проект

2.4 БАЗА ДАНИХ

База даних використовується для зберігання даних про проекти, зберігання аккаунтів для блокчейна та зберігання карток з відповідними токенами.

2.4.1 Сутності

Програма явно працює з 3 сутностями:

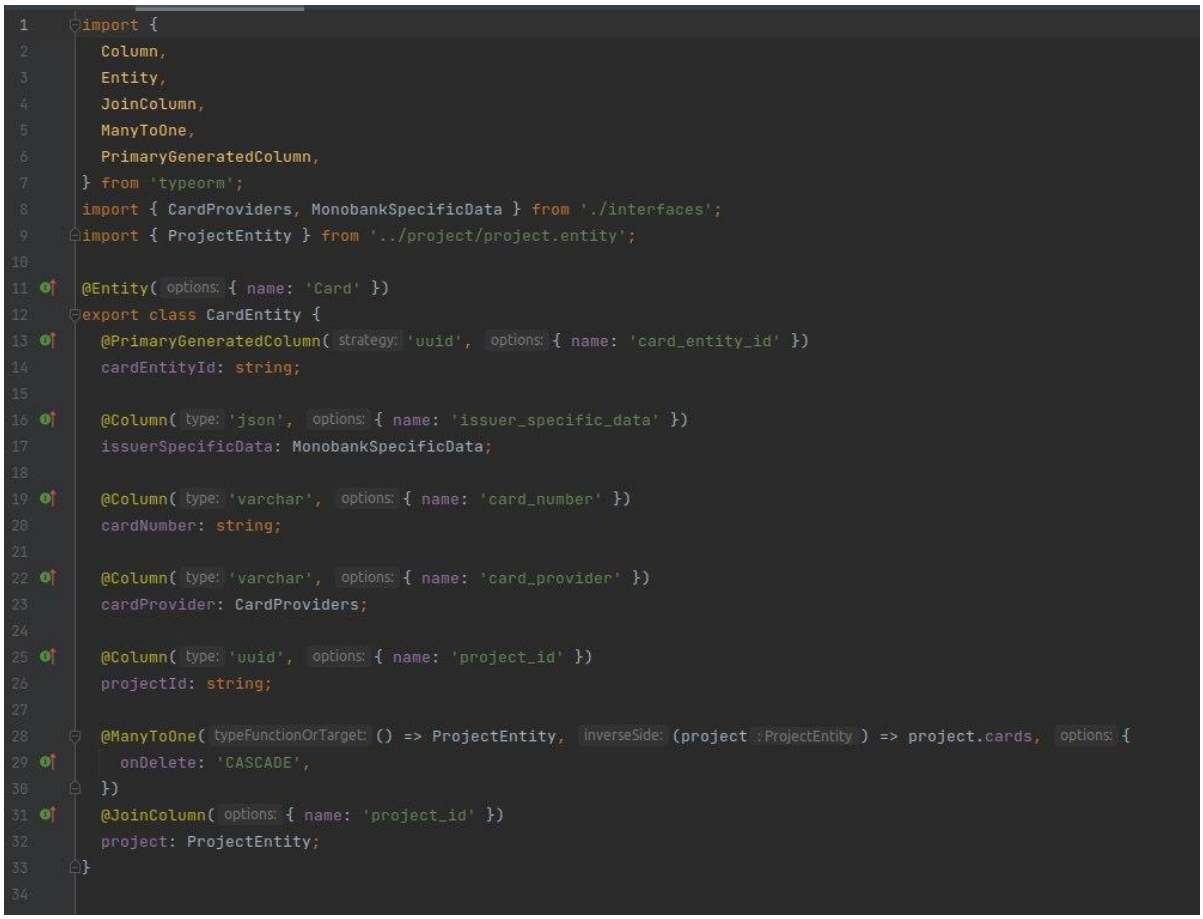
- CardEntity(відповідає таблиця Card)
- ProjectEntity(відповідає таблиця Project)
- BlockchainEntity(відповідає таблиця Blockchain)

CardEntity відповідає за зберігання картки будь-якого банку. Вона має 5 колонок:

- issuer_specific_data, в якій записано специфічну дані, які залежить від банка, з яким інтегруємось. Наприклад, для монобанка, тут буде записано об’єкт з властивістю accountId. Тому, тип цієї колонки – json
- card_number – varchar, для зберігання номеру картки
- card_provider – varchar. У коді використовується як enum. Вказує на назву банку, який випустив картку
- project_id – uuid, службова колонка для створення звязку з сутністю ProjectEntity
- card_entity_id – uuid, унікальний ключ стрічки

Код для створення таблиці виглядає так:

```
CREATE TABLE "Card"
    (
        "card_entity_id"      uuid            NOT NULL DEFAULT uuid_generate_v4(),
        "issuer_specific_data" json           NOT NULL,
        "card_number"         character varying NOT NULL,
        "card_provider"       character varying NOT NULL,
        "project_id"          uuid            NOT NULL,
        CONSTRAINT "PK_4ce471d53d8a7ffee9992ef4ddd" PRIMARY KEY ("card_entity_id")
    )
```



```
import {
  Column,
  Entity,
  JoinColumn,
  ManyToOne,
  PrimaryGeneratedColumn,
} from 'typeorm';
import { CardProviders, MonobankSpecificData } from './interfaces';
import { ProjectEntity } from '../project/project.entity';

@Entity( options: { name: 'Card' })
export class CardEntity {
  @PrimaryGeneratedColumn( strategy: 'uuid', options: { name: 'card_entity_id' })
  cardEntityId: string;

  @Column( type: 'json', options: { name: 'issuer_specific_data' })
  issuerSpecificData: MonobankSpecificData;

  @Column( type: 'varchar', options: { name: 'card_number' })
  cardNumber: string;

  @Column( type: 'varchar', options: { name: 'card_provider' })
  cardProvider: CardProviders;

  @Column( type: 'uuid', options: { name: 'project_id' })
  projectId: string;

  @ManyToOne( typeFunctionOrTarget: () => ProjectEntity, inverseSide: (project : ProjectEntity ) => project.cards, options: {
    onDelete: 'CASCADE',
  })
  @JoinColumn( options: { name: 'project_id' })
  project: ProjectEntity;
}
```

Рисунок 2.4.1.1 Вигляд сутності у коді

ProjectEntity відповідає за зберігання проекту. Вона має 6 колонок:

- project_name – text, назва проекту, для відображення у пошуку
- project_description – text, опистого, на що будуть витрачені кошти
- blockchain_id – uuid, службова колонка для зв’язку з сутністю BlockchainEntity
- goal – numeric, колонка, яка вказує на необхідну кількість коштів для виконання цілі
- collectedAmount – numeric, колонка, яка вказує на те, скільки коштів вже зібрано
- project_id – uuid, унікальний ключ стрічки

BlockchainEntity відповідає за зберігання даних про доступ до акаунту блокчейн
Має 3 колонки:

- blockchain_key – json, зберігає значення публічного ключа
- blockchain_description – text, відповідає за опис ключа
- blockhain_id – uuid, унікальний ключ стрічки

2.4.2 Схема

Проект відноситься до блокчейну, як один до одного.

Між проектом та карткою з’являється відношення один до багатьох, адже користувач може отримувати пожертви на багато карток.

Також, є 2 службових таблиці: одна для запису міграцій, які вже відбулись та інша метаданих ORM.

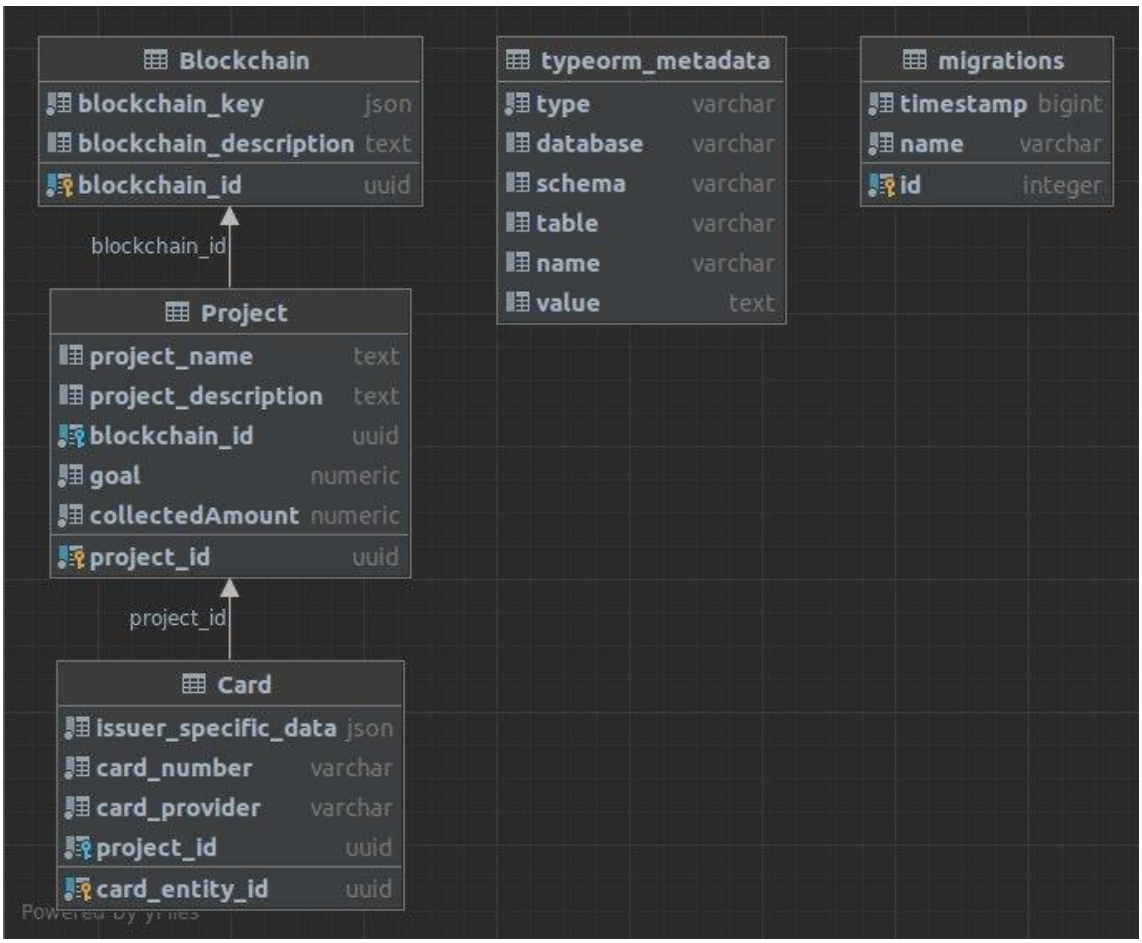


Рисунок 2.4.2.1 Графічне зображення схеми БД

2.4.3 Міграції

Міграція – це керування інкрементними, зворотніми змінами схеми реляційної бази даних. Міграція схеми виконується над базою даних коли

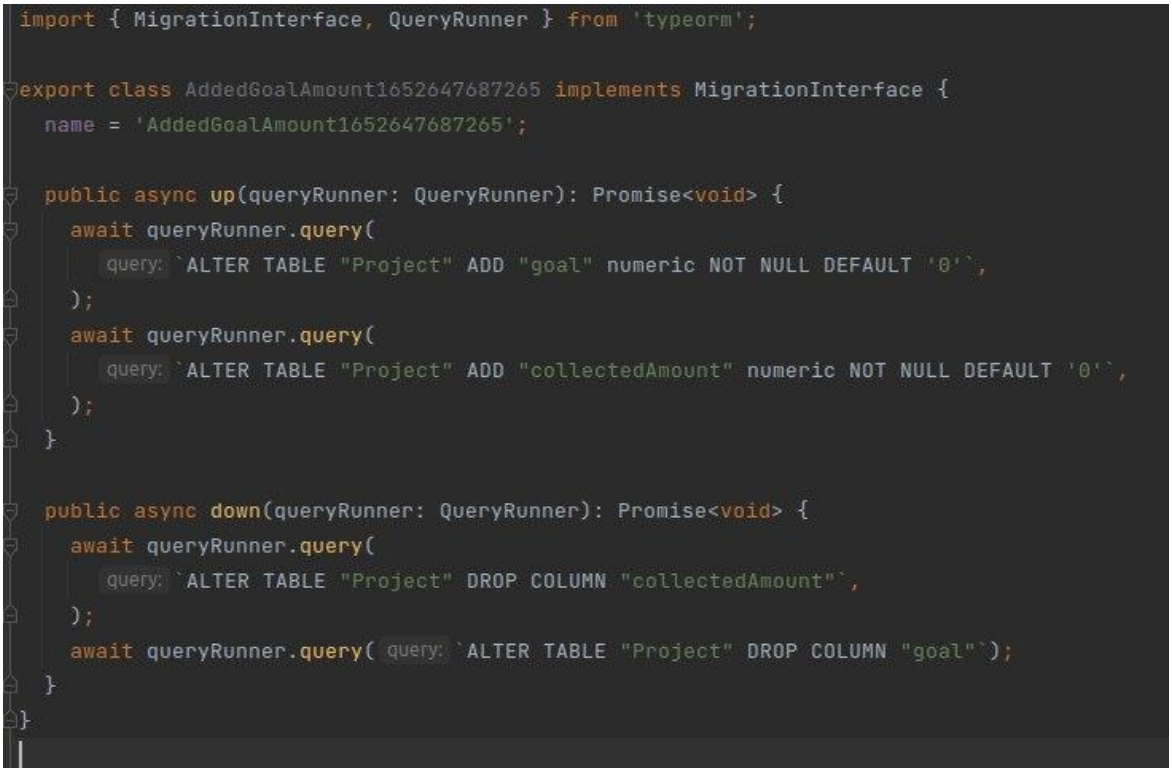
необхідно оновити, або повернути схему бази даних до якоїсь новішої чи старішої версії.[13]

Створення міграцій автоматизовано за допомогою фреймворку TypeORM та додано скрипти, які спрощують взаємодію.

Для створення міграції потрібно виконати один з 2 скриптів:

- *npm run migrate:create { FileName }* для створення пустого файлу
- *npm run migrate:generate{FileName}* для передзаповнення файлу змінами, які фреймворк зміг відслідкувати

Після виконання другої команди, отримуємо файл, який буде такий вигляд:



```
import { MigrationInterface, QueryRunner } from 'typeorm';

export class AddedGoalAmount1652647687265 implements MigrationInterface {
  name = 'AddedGoalAmount1652647687265';

  public async up(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.query(
      query: `ALTER TABLE "Project" ADD "goal" numeric NOT NULL DEFAULT '0'`,
    );
    await queryRunner.query(
      query: `ALTER TABLE "Project" ADD "collectedAmount" numeric NOT NULL DEFAULT '0'`,
    );
  }

  public async down(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.query(
      query: `ALTER TABLE "Project" DROP COLUMN "collectedAmount"`,
    );
    await queryRunner.query( query: `ALTER TABLE "Project" DROP COLUMN "goal"`);
  }
}
```

Рисунок 2.4.3.1 Приклад файлу з міграцією

Після правильної конфігурації, фреймворк сам зрозумів, що було додано 2 колонки: goal, collectedAmount до таблиці Project і створив базовий код.

Тут бачимо 2 функції up та down. Перша відповідає за додавання відповідних колонок та створення обмежень. Друга – за те, щоб після невдалої міграції можна було повернути систему у стан, до цієї міграції.

Для записку міграцій потрібно виконати команду:

npm run migrate

Вона сама відслідкує, які міграції було помічено як виконані та пропустить їх, а потім виконає усі інші. Цю команду можна налаштувати у системах CI/CD для автоматизації, а можна і просто виконати з локального комп’ютера, передавши необхідні параметри.

Для повернення системи у стан до міграції, я додав команду:

npm run migrate:undo

РОЗДІЛ 3. ВИБІР ХМАРНОГО РІШЕННЯ ТА РОЗГОРТАННЯ ЗАСТОСУНКУ

На ринку є 3 найпопулярніші хмарні рішення: AWS, Google Cloud та Microsoft Azure

3.1 Огляд хмарних рішень

3.1.1 Google Cloud

Google Cloud Platform(GCP) — це хмарне рішення, яке пропонує Google. GCP є частиною великої хмари від Google.

Перший публічний реліз відбувся у 2010 році. Зараз Google Cloud Platform пропонує понад 100 різних сервісів, що включають роботу з великими даними, мережею, важкими обчисленнями, тощо.

Найважливішими із цих сервісів є Google Workspace, корпоративний Android та Chrome OS.

Якщо порівняти його з AWS та Azure, то він буде найменшим постачальником. Тим не менш, він пропонує достатню кількість хмарних сервісів для розгортання та підтримки додатків будь-якої складності.

Він присутній у 27 регіонах, має 82 дата центри та 146 точок доступу.

Відомі клієнти, які користується послугами:

- Nintendo
- Spotify
- Twitter
- Paypal
- UPS

Плюси:

- Гарна інтеграція з іншими сервісами Google
- Глобальна мережа з оптоволокна

- Ідеальна підтримка контейнерізації

Мінуси:

- Обмежена підтримка для великих клієнтів, у порівнянні з іншими рішеннями
- Менше сервісів

3.1.2 Microsoft Azure

Microsoft Azure — другий по величині хмарний провайдер.

Дебютувала у 2010 році у публічному доступі. За 12 років, Azure змогла перетворитись на платформу, яка має понад 200 сервісів. Це найбільш швидкозростаюча платформа з оглянутих.

Так, як Azure – це продукт Microsoft, то він пропоную багато послуг, які здатні полегшити міграцію на хмарне, або гібридне середовище для компаній, орієнтованих на Microsoft. Наслідком цього є те, що 90% компаній з Fortune 500 використовують саме це рішення. Microsoft має перевірений досвід у сфері обслуговування корпоративних користувачів.

Варто відмітити, що Azure не обмежуються продуктами на базі Windows. Він має можливість працювати з іншими мовами, технологіями та платформами. Але часто необхідною умовою є відкритий код. Це все дає можливість створити та підтримувати будь-який додаток на базі цього рішення.

Має більше, ніж 60 географічних регіонів, з більше ніж трьома дата центрами у кожному. Має 116 точок присутності.

Відомі клієнти:

- Asos
- Starbucks
- Renault
- HP
- eBay

Плюси:

- У більшості випадків, дешевше за інших конкурентів
- Доступно багато сервісів, включаючи найкращі сервіси для штучного інтелекту, машинного навчання та аналітики
- Чудова підтримка гібридних стратегій
- Інтеграція з сервісами Microsoft

Мінуси:

- Менше сервісів, ніж у AWS
- Орієнтація на корпоративних клієнтів

3.1.3 AWS

Безумовним лідером ринку є Amazon Web Services, дочірня компанія Amazon.com, Inc. Публічний реліз відбувся у 2006 році, а отже це рішення є найстаршим. Клієнти AWS не обмежені великими підприємництвами та окремими розробниками. Серед клієнтів можна знайти навіть уряди.

Історія AWS почалась з сервісу для внутрішнього користування у компанії. У 2006 році відбувся реліз лише кількох сервісів(S3, EC2). Сьогодні у AWS можна знайти більше ніж 200 різних сервісів, які використовують мільйони користувачів.

Відомі клієнти:

- Ainbnb
- Netflix
- Twitch
- BBC
- Lyft

Плюси:

- Найбільша кількість доступних сервісів
- Найстаріше, а отже протестоване роками

- Більшість відомих вендорів програмного забезпечення роблять свої продукти доступними на AWS
- Признано золотим стандартом надійності та підтримуванності
- Має більше обчислювальних потужностей, ніж Azure або GCP

Мінуси:

- Потрібно платити за підтримку, навіть для аккаунта розробника
- Не так багато опцій для гібридних рішень
- Може злякати нових користувачів інтерфейсом та кількістю сервісів[12]

3.1.4 Висновок

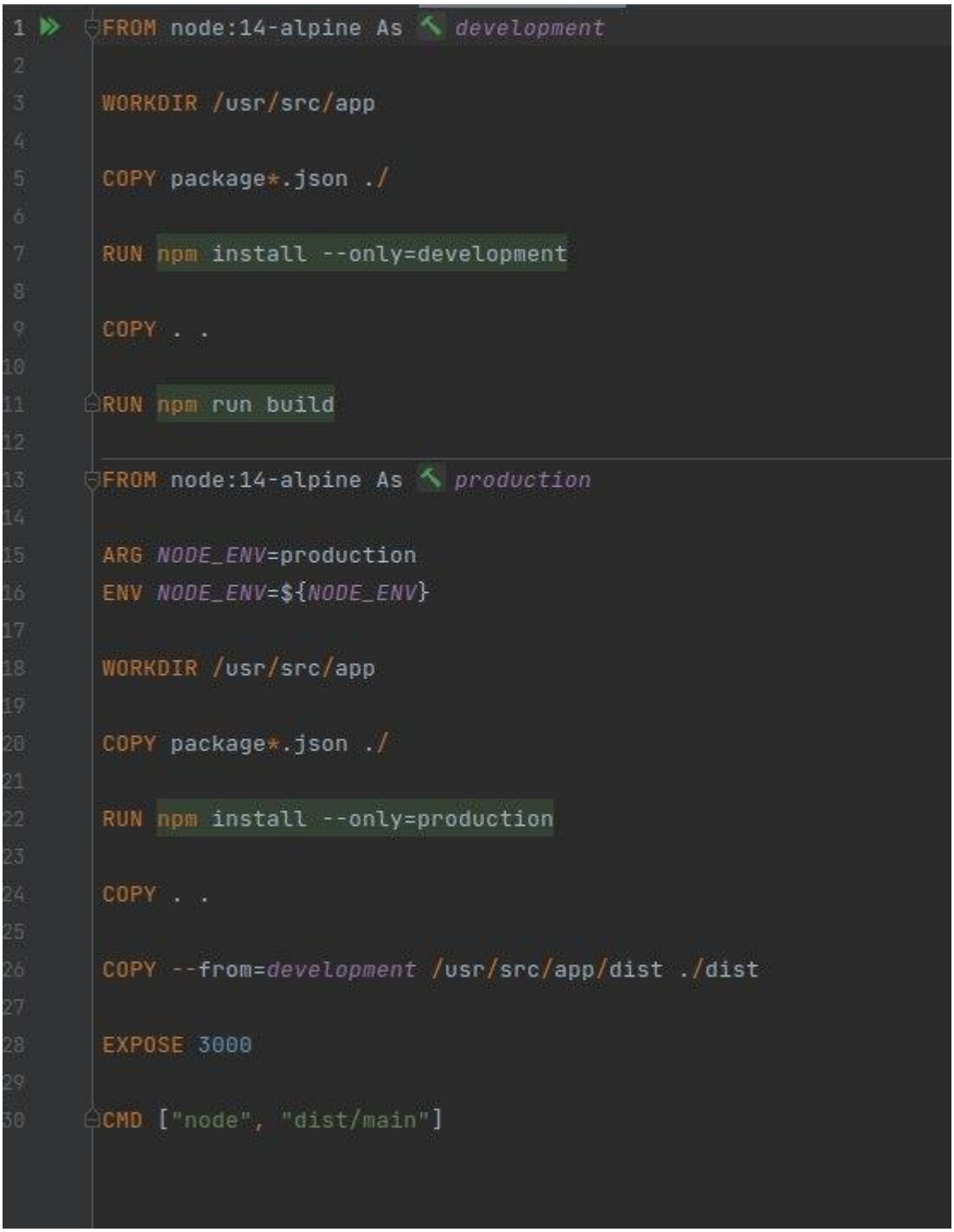
Проаналізувавши хмарні рішення, я вирішив використати AWS, тому що він пропонує своє рішення для блокчейну та має потужний безкоштовний період.

3.2 Розгортання додатку

Базу даних розгорнуто засобами Amazon RDS[15].

Сам додаток розгорнуто за допомогою сервісу Amazon Elastic Container Service. Для цього було створено нову сутність у Amazon EC2, на якій і буде запускатись контейнер.

Для контейнеризації я створив докер файл:



```
FROM node:14-alpine As development

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install --only=development

COPY . .

RUN npm run build

FROM node:14-alpine As production

ARG NODE_ENV=production
ENV NODE_ENV=${NODE_ENV}

WORKDIR /usr/src/app

COPY package*.json ./

RUN npm install --only=production

COPY . .

COPY --from=development /usr/src/app/dist ./dist

EXPOSE 3000

CMD ["node", "dist/main"]
```

Рисунок 3.2.1 Докер файл для розгортання у хмарі

При створенні файлу було використано найкращі практики для того, щоб мінімізувати розмір контейнеру.

Після розгортання контейнеру, було додано load balancer(ELB[14]) та відповідну групу(ASG), яка буде розширюватись горизонтально, відповідно до навантаження.

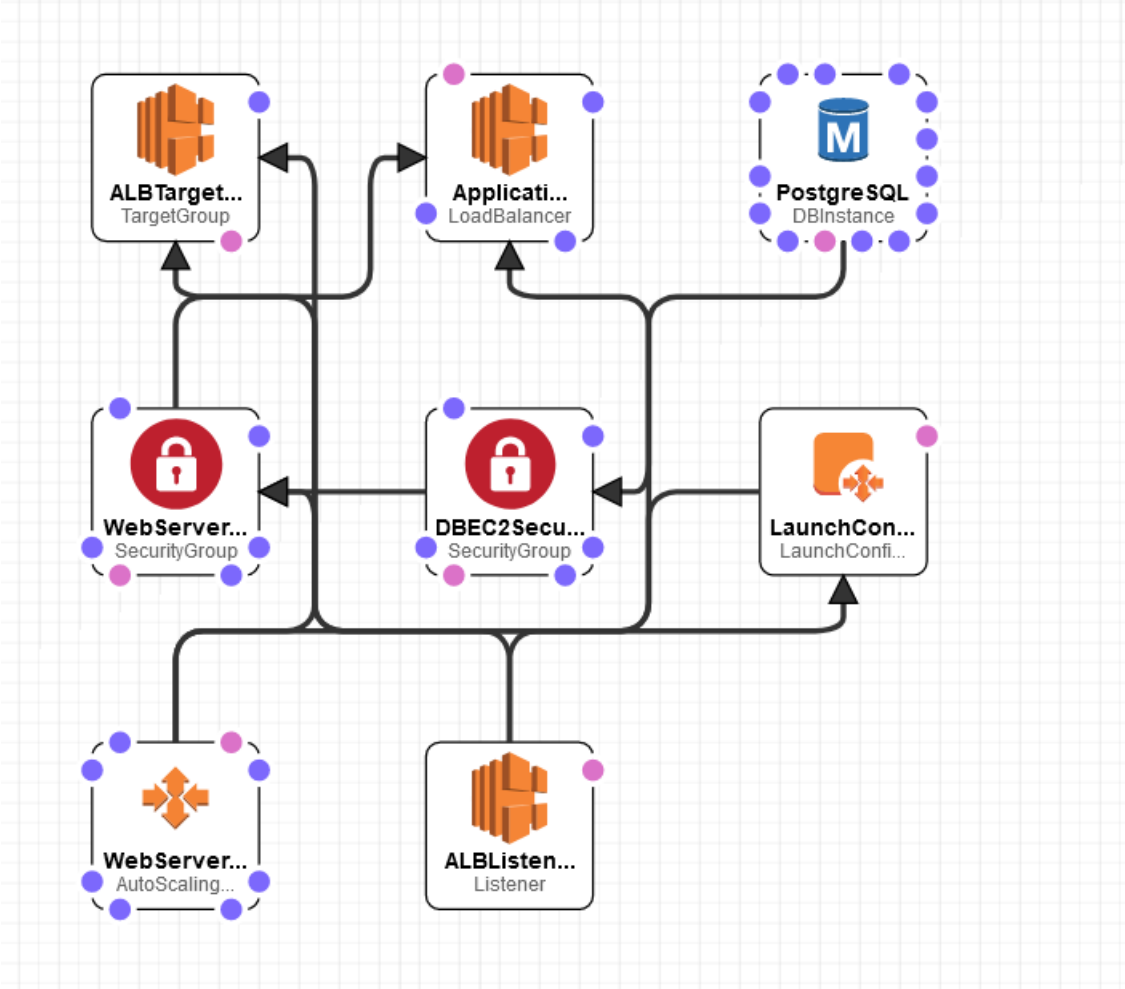


Рисунок 3.2.2 Архітектура у хмарі

ВИСНОВОК

Актуальність жертвувати на ті, чи інші потреби, завжди зростала. Можна подивитись на пафосні благодійні вечори, на яких люди жертвують баснословні суми. Слід звернути увагу на людей, які жертвують кілька десятків гривень на новий бронежилет. Усіх їх об’єднує те, що вони хочуть, щоб їх гроші використали з благою ціллю. Перші можуть бути впевненими, що їх не обікрадуть, адже вони можуть перевірити компанії, спиратись на репутацію збирачів, тощо. На жаль, є ймовірність, що других просто обдурять та заберуть пожертвувані гроші, або навіть більше.

Розглянемо, як можуть обдурити людей, які жертвують невеликі суми:

1. Вони не знають, скільки вже пожертвували інші гроші, а отже автор може зібрати більше та забрати різницю собі
2. Сервіс-посередник, який збирає гроші та вказує, скільки вже надіслано, може привласнити їх собі
3. Людина, яка збирає гроші, може зникнути

Наш додаток вирішує перших 2 проблеми, адже ми вказуємо, скільки грошей вже зібрано та зберігаємо це все у блокчейн. А отже, розробники не зможуть змінити значення того, що вже зібрано. Це виключає можливість змови. Ми не зберігаємо гроші на наших аккаунтах, а використовуємо безпечні інтеграції, а тому доступу до грошей в нас немає. Також ми не бачимо персональних даних про картку і не зможемо нічого вкрасти. Застосунок має відкритий вихідний код і кожен, хто хоче ним користуватись може подивитись його. Третю проблеми ми не можемо вирішити напряму, але після таких ситуацій, людина просто не зможе більше користуватись нашим застосунком.

Отже, наш застосунок робить пожертви доступнішими для широких верств населення та надає гарантії. Це збільшить кількість пожертв та зменшить кількість шахраїв.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Новини Західної України. Новина про шахраїв [Електронний ресурс] // URL:https://zahid.espreso.tv/shakhrai-prosyat-pozhertvi-cherez-storinki-kloni-vidomikh-u-lvovi-svyashchennosluzhiteliv
2. BBC News. Новина про фейкових волонтерів [Електронний ресурс] // URL: https://www.bbc.com/ukrainian/news-61337606
3. Kickstarter [Електронний ресурс] // URL: https://www.kickstarter.com/
4. Documenting Software Architectures: Views and Beyond, Second Edition. Boston: Addison-Wesley. Автори: Clements, Paul; Felix Bachmann; Len Bass; David Garlan; James Ivers; Reed Little; Paulo Merson; Robert Nord; Judith Stafford
5. Onion arcitecrure [Електронний ресурс] // URL: https://www.codeguru.com/csharp/understanding-onion-architecture/
6. Hexagonal architecture [Електронний ресурс] // URL: https://blog.octo.com/hexagonal-architecture-three-principles-and-an-implementation-example/
7. Hexagonal architecture [Електронний ресурс] // URL: https://alistair.cockburn.us/hexagonal-architecture/#:~:text=Intent,run%2Dtime%20devices%20and%20databases.
8. Hexagonal architecture [Електронний ресурс] // URL: https://netflixtechblog.com/ready-for-changes-with-hexagonal-architecture-b315ec967749
9. Express [Електронний ресурс] // URL: https://github.com/expressjs/express#philosophy
10. Nest.js [Електронний ресурс] // URL: https://docs.nestjs.com/#philosophy
11. PostgreSQL [Електронний ресурс] // URL: https://www.postgresql.org/

12. Comparing cloud providers [Електронний ресурс] //
URL: https://www.bmc.com/blogs/aws-vs-azure-vs-google-cloud-platforms/
13. Міграція бази даних [Електронний ресурс] //
URL:
https://uk.wikipedia.org/wiki/%D0%9C%D1%96%D0%B3%D1%80%D0%B0%D1%86%D1%96%D1%8F_%D0%B1%D0%B0%D0%B7%D0%B8_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85
14. Amazon ELB [Електронний ресурс] //
URL: https://aws.amazon.com/elasticloadbalancing/
15. Amazon RDS [Електронний ресурс] //
URL: https://aws.amazon.com/rds/?trk=2fad3d88-146d-4d95-858e-c860297a48cb&sc_channel=ps&sc_campaign=acquisition&sc_medium=ACQ-P|PS-GO|Brand|Desktop|SU|Database|Solution|EEM|EN|Text|EU&s_kwcid=AL!4422!3!548915587590!e!!g!!amazon%20relational%20database%20service&ef_id=EAIaIQobChMI3-Dpt_yQ-AIVDpftCh0giQNMEAAYASABEgIn6vD_BwE:G:s&s_kwcid=AL!4422!3!548915587590!e!!g!!amazon%20relational%20database%20service

ДОДАТКИ

https://github.com/kostia1359/OpenCharityTracker — посилання на репозиторій гіта.