Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Факультет інформатики
Кафедра мережевих технологій

# Магістерська робота

Освітній ступінь: магістр

На тему: «Побудова мережевого застосування з високою доступністю на хмарній платформі»

Текстова частина до дипломної роботи за спеціальністю «Інженерія програмного забезпечення» 121

Виконав: студент 2 року навчання

Спеціальності

121 Інженерія програмного забезпечення

Возбранний Роман Сергійович

Керівник: Черкасов Д.І.

Рецензент

Магістерська робота захищена

з оцінкою_____

Секретар ЕК С.А. Мелещенко

«___» _____2023

Київ 2023

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережевих технологій

ЗАТВЕРДЖУЮ
Зав. кафедри мультимедійних систем,
доктор техн. наук, декан ФІ
_____…
(підпис)
“____”_____ 202_ р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на магістерську роботу
студенту Возбранному Роману
факультету інформатики 2 курсу магістерської програми

ТЕМА: Побудова мережевого застосування з високою доступністю на хмарній платформі.

Вихідні дані:

Зміст ТЧ до курсової роботи:

Вступ

Анотація

1. Огляд технологій та сервісів для побудови застосунку.

2. Програмна реалізація

3. Автоматизоване розгортання, CI/CD інтеграція

Висновки

Список використаної літератури та електронних ресурсів

Додатки

Дата видачі “_____” _____202_ р.

Керівник _____ Завдання отримано ___ _____

**Тема:** Побудова мережевого застосування з високою доступністю на хмарній платформі.

**Календарний план виконання роботи:**

| № | Назва етапу | Термін виконання | Примітка |
|---|---|---|---|
| 1. | Отримання теми курсової роботи | | |
| 2. | Вивчення предметної області | | |
| 3. | Огляд засобів реалізації | | |
| 4. | Вивчення технологій | | |
| 5. | Написання першої частини курсової роботи | | |
| 6. | Написання другої частини курсової роботи | | |
| 7. | Написання третьої частини курсової роботи | | |
| 8. | Написання висновків курсової роботи | | |
| 9. | Перегляд змісту роботи з керівником | | |
| 10. | Внесення змін до курсової роботи відповідно до зауважень наукового керівника | | |
| 11. | Створення презентації | | |
| 12. | Захист роботи | | |

Студент Возбранний Р.С.
Керівник Черкасов Д.І.

“____” _____

# Зміст

# Анотація

Робота зосереджена на процесі розробки та розгортання багаторівневого веб-застосунку з високою доступністю на хмарній платформі. Центральна ідея полягає в тому, що хмарні обчислювальні сервіси, надаючи автоматичне масштабування та засоби моніторингу, полегшують процес розробки. Застосунок, який було створено в ході дослідження, має на меті автоматизувати процеси моніторингу, збирання, структурування та валідації даних з відкритих джерел.

В роботі описано використання технологій Node.js, Nest.js, React, PostgreSQL та TypeORM для розробки застосунку, що забезпечує високий рівень стабільності, безпеки та продуктивності. Визначено ключові етапи розробки серверної та клієнтської частини застосунку з акцентом на правильне використання інструментів та бібліотек для автоматизованого збирання даних, автентифікації та управління базами даних.

Також було проведено автоматизацію процесів розгортання та інтеграції CI/CD за допомогою Github, Github Actions, Docker, Google Cloud Run, Vercel та Sentry. В результаті, було розроблено надійний, високопродуктивний та легко масштабований веб-застосунок.

# Вступ

За останні роки хмарні обчислювальні сервіси стали популярним інструментом для розробки та розгортання веб-застосунків. Ці сервіси дозволяють розробникам не турбуватися про налаштування інфраструктури та обслуговування, що забезпечує більшу продуктивність та ефективність розробки застосунків з високою доступністю.

Висока доступність означає, що застосунок завжди доступний для користувачів в будь-який час та з будь-якого місця, незалежно від кількості запитів та навантаження на сервер. Надійна та стабільна роботу застосунку є критичним для бізнесу, оскільки може призвести до втрати прибутку та клієнтів.

Хмарні платформи надають можливість автоматичного масштабування. Крім того, хмарні платформи також надають різноманітні засоби моніторингу та аналізу продуктивності роботи компонентів, що дозволяє розробникам відстежувати та вирішувати проблеми в реальному часі. У підсумку, використання хмарних платформ може значно полегшити процес розробки.

Для реалізації було обрано створення веб-застосунку, який складатиметься з серверної та клієнтської частини, а також мікросервісів. Функціональним призначенням застосунку є автоматизація моніторингу, збирання, структурування та забезпечення можливості ручної валідації та корекції даних з відкритих джерел шляхом аналізу статей та структурування інформації про людей, компанії та зв’язків між ними.

Об’єкт дослідження: процес розробки та розгортання багаторівневого застосування з високою доступністю на хмарній платформі.

Мета дослідження: визначити та описати особливості розробки та налаштування застосування з високою доступністю для подальшого розгортання на хмарній платформі.

Робота має три розділи:

1) В першому розділі проводиться огляд та опис технологій, які використовуються для розробки застосунку.
2) В другому розділі розглядаються важливі моменти створення застосунку.
3) В третьому розділі розглядаєься CI/CD та огляд інструментів моніторингу і аналізу продуктивності.

# Розділ 1. Огляд технологій та сервісів для побудови застосунку

## 1.1 Технології та сервіси, які використовуються для розробки як клієнтської так і серверної частини

### 1.1.1 Пакетний менеджер

Node.js поставляється з npm (Node Package Manager) за замовчуванням, який є основним інструментом для керування бібліотеками та пакетами в екосистемі Node.js. npm надає доступ до великої кількості модулів та бібліотек, що створені спільнотою, і має простий інтерфейс командного рядка. Проте, існують і інші пакетні менеджери, такі як Yarn та pnpm, які пропонують деякі переваги порівняно з npm.

Yarn, який був розроблений Facebook, пропонує більшу швидкість та надійність порівняно з npm. Він використовує кешування та паралельне завантаження для підвищення швидкості інсталяції пакетів. Yarn також має систему блокування, яка гарантує, що встановлення пакетів є однаковими на всіх системах [2].

pnpm пропонує унікальну властивість зберігання пакетів. Він використовує "жорсткі посилання" для зберігання одного екземпляра пакета незалежно від того, скільки проектів використовують його. Це може суттєво зменшити використання дискового простору та прискорити процес встановлення пакетів. Окрім цього, pnpm має строгу ізоляцію між пакетами, що зменшує ймовірність появи проблем, пов'язаних з випадковими доступами - коли пакет має доступ до залежностей, які не були явно вказані в package.json [1].

Ключові особливості та переваги pnpm:

- pnpm забезпечує спільне використання пакетів між різними проектами, що допомагає зекономити місце на диску та пришвидшити встановлення залежностей.

- Завдяки ефективному кешуванню та оптимізації, швидкість встановлення пакетів використовуючи pnpm вища за такі популярні менеджери пакетів, як npm та Yarn.
- pnpm повністю сумісний зі стандартними файлами package.json та npm-скриптами, що дозволяє легко переключитися між менеджерами пакетів.
- pnpm використовує жорстку систему залежностей для забезпечення безпечності проекту, зменшуючи ризик непередбачених проблем, пов'язаних з пакетами.

Зважаючи на описані особливості пакетних менеджерів Node.js, для програмної реалізації було вирішено використовувати pnpm як на клієнтській так і на серверній частині.

### 1.1.2 TypeScript

TypeScript є сучасною надбудовою над JavaScript, яка додає статичну типізацію до мови програмування та сприяє розробці масштабованих програмних продуктів. Розроблений Microsoft, TypeScript використовується як інструмент для поліпшення якості коду, спрощення роботи в команді та покращення продуктивності розробки.

Характеристики та переваги TypeScript [3]:

- Дозволяє визначати типи даних для змінних, аргументів функцій та об'єктів, що забезпечує кращий контроль над кодом та попереджує багато помилок на етапі компіляції.
- Повністю сумісний з JavaScript, що означає, що розробник може використовувати будь-який JavaScript код у TypeScript проектах та навпаки. Це дозволяє легко інтегрувати TypeScript у вже існуючі проекти.

- Підтримка останніх версії ECMAScript, дозволяючи розробникам користуватися найновішими можливостями JavaScript у проектах.
- Широка підтримка різних інструментів розробки, середовищ програмування (IDE) та плагінів, що сприяє підвищенню продуктивності розробки та якості коду.

### 1.1.3 Технологія контейнеризації Docker

Docker є відомою та ефективною технологією контейнеризації, яка допомагає розробникам створювати, розгортати та керувати програмами в ізольованих контейнерах. Він полегшує процес розробки, тестування та розгортання програмних продуктів, забезпечуючи надійність та гнучкість в роботі.

Характеристики та переваги Docker [4]:

- Дозволяє розробникам створювати контейнери, які ізолюють програми та їх залежності від операційної системи хоста. Це забезпечує стабільність та однорідність середовища виконання на різних платформах.
- Завдяки контейнерам Docker, програми можуть працювати на різних платформах та операційних системах без змін у коді або додаткових налаштувань.
- Допомагає забезпечити відтворюваність програми на всіх стадіях розробки, тестування та розгортання, спрощуючи процеси розробки та співпраці.
- Дозволяє керувати ресурсами хоста та контейнера, що сприяє ефективному використанню системних ресурсів та зниженню витрат.

- Має активну спільноту розробників та широку екосистему інструментів, сервісів та ресурсів, які полегшують розробку, розгортання та керування контейнерами.

## 1.2 Інтерфейс користувача

### 1.2.1 Інструменти для збірки проекту

Збірка веб-додатків, особливо клієнтської частини, є важливим кроком у процесі розробки та розгортання з кількох причин:

- Ефективність завантаження: Великі веб-додатки складаються з десятків або навіть сотень різних файлів JavaScript, CSS та інших. Замість того, щоб відправляти кожен файл окремо, що може бути повільним та неефективним, збирачі проектів можуть згрупувати всі ці файли в один або декілька бандлів для більш ефективного завантаження.
- Мініфікація та оптимізація: Збірка проекту також дозволяє виконувати різні оптимізації, такі як мініфікація (видалення непотрібних символів), видалення "мертвого" коду (коду, який ніколи не використовується), та "tree shaking" (видалення невикористаних експортів), що можуть зменшити розмір збірки та покращити швидкість завантаження.
- Сумісність браузерів: Не всі браузери підтримують всі функції JavaScript та CSS. Бандлери можуть використовувати транспіляцію (перетворення коду з однієї версії мови на іншу) та поліфіли (код, який надає сучасні функції в старих браузерах), щоб забезпечити сумісність з широким спектром браузерів.
- Модульність: Бандлінг допомагає розробникам створювати більш структурований і модульний код, згруповуючи пов'язані частини

коду разом. Це може покращити якість коду, зробити його більш зрозумілим та легшим для підтримки.

Webpack та Vite - це два інструменти для збирання модулів JavaScript, які використовуються в сучасних веб-проектах. Хоча обидва інструменти виконують подібні завдання, їх підходи та можливості відрізняються.

Webpack - це надзвичайно потужний та популярний інструмент для бандлінга (групування) JavaScript коду та інших активів, таких як CSS і зображення (рисунок 1.2.1.1). Він має велику кількість налаштувань, плагінів та загрузчиків, які можуть бути налаштовані для виконання широкого спектру завдань. Однак, ця гнучкість може призвести до складності конфігурації та повільної збірки, особливо для великих проектів [6].

Vite є інноваційним інструментом збірки та розробки, який базується на нативних ECMAScript модулях (ESM) та призначений для роботи з сучасними JavaScript-фреймворками. Цей інструмент відрізняється високою швидкістю та легкістю використання, а також підтримкою гарячого оновлення модулів (HMR) без перезавантаження сторінки [7].

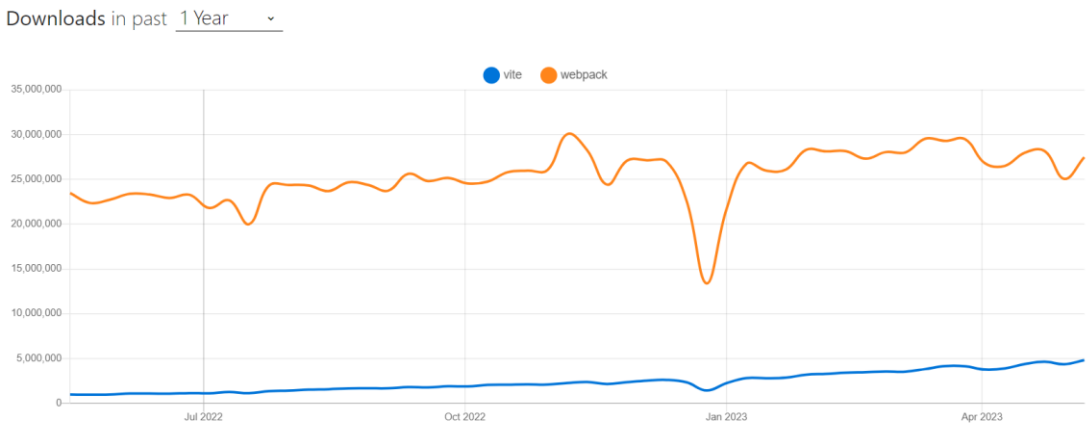


*Рис. 1.2.1.1 Порівняння кількості завантажень Vite та Webpack [5]*

Основні характеристики та переваги Vite:

- Завдяки оптимізованому процесу збірки, Vite пропонує значне прискорення старту застосунку для розробки та перебудови модулів (рисунок 1.2.1.2).
- Включає вбудовану підтримку HMR, яка дозволяє оновлювати компоненти в реальному часі без перезавантаження сторінки, що забезпечує зручність розробки.
- Забезпечує сумісність з плагінами Rollup, що дає можливість розширити функціональність та адаптувати інструмент для специфічних вимог проекту.
- Оптимізація коду за допомогою Tree Shaking та інших технік, які допомагають зменшити розмір файлів та покращити продуктивність.
- Автоматична обробка файлів CSS, SCSS та інших, що спрощує процес інтеграції стилів у проект.

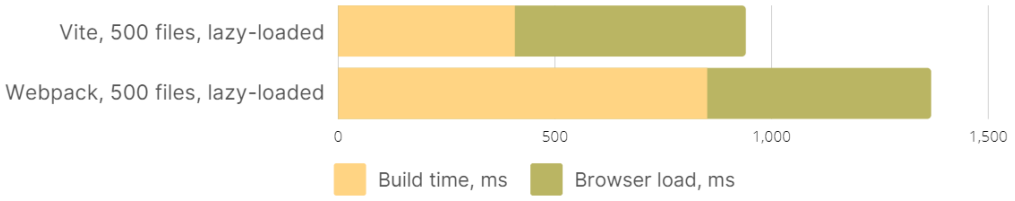


*Рис. 1.2.1.2 Порівняння швидкості роботи Webpack та Vite*

Отже, зважаючи на особливості, швидкість розробки та простоту конфігурації, Vite є кращим вибором для програмної реалізації застосунку порівняно з Webpack.

### 1.2.2 Інструменти для розробки інтерактивних веб-додатків

Найпопулярнішими інструментами для створення інтерактивних веб-додатків є React.js, Vue.js та Angular (рисунок 1.2.2.1).

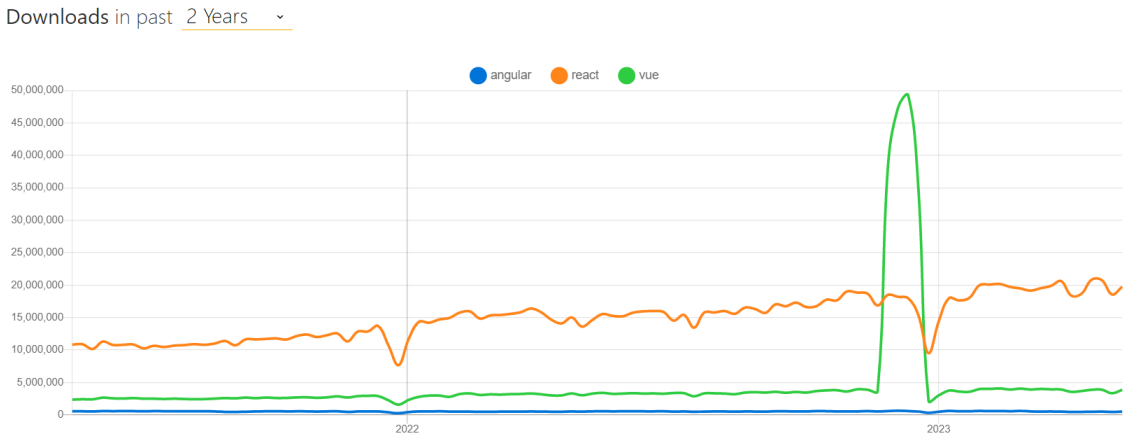


*Рис. 1.2.2.1 Порівняння кількості завантажень Angular, React та Vue [5]*

Angular від Google - це повноцінний фреймворк, який включає в себе все, що потрібно для створення складних веб-додатків. Він має багато вбудованих інструментів та функцій, включаючи систему для керування станом, формами, валідацією, маршрутизацією, інтернаціоналізацією та іншими. Але він має високий поріг входу і може бути складним для навчання.

Vue.js є легшим і більш гнучким фреймворком. Vue.js є гнучким та модульним, з легкою інтеграцією з іншими бібліотеками та існуючими проектами. Він також має дуже низький поріг входу та легкий у вивченні.

React.js - це бібліотека, яка зосереджена на створенні інтерфейсів користувача через компоненти. React є гнучким і потужним, дозволяючи створювати високопродуктивні додатки. Він має велику спільноту, багато ресурсів для навчання та широке використання.

Описані бібліотеки мають свої відмінності та особливості, але є ряд причин, чому React може вважатися кращим варіантом для практичної реалізації проекту:

- Гнучкість: React.js є більш гнучким порівняно з Vue та Angular, оскільки він не накладає строгих вимог на структуру проекту чи спосіб розробки. React пропонує лише базову бібліотеку для створення користувацького інтерфейсу, а решта функціональності може бути розширена за допомогою сторонніх бібліотек, що дозволяє вибирати найкращі інструменти для відповідних потреб.
- Продуктивність: React використовує віртуальний DOM для оптимізації рендерингу, що забезпечує високу продуктивність, навіть у великих додатках. Це також дозволяє легко створювати динамічні й інтерактивні інтерфейси з меншим впливом на продуктивність.
- Спільнота та екосистема: React має велику та активну спільноту розробників. Це означає, що існує велика кількість документації, наявних плагінів та компонентів, які можуть використовуватися для прискорення розробки та розв'язання проблем.

Отже, основою для програмної реалізації клієнтської частини застосунку було обрано бібліотеку React.js.

### 1.2.3 Управління станом в React застосунку

React.js є високопродуктивною JavaScript бібліотекою, призначеною для створення клієнтських додатків. Як і інші JavaScript бібліотеки, React.js гарантує надійність при розробці реактивних та декларативних користувацьких інтерфейсів. Однак у React є певні недоліки, одним з яких є керування станом та отриманням даних. Для вирішення цієї проблеми найпопулярнішою бібліотекою є Redux.

Redux - це бібліотека управління станом, яка використовує патерн Flux. Він надає єдине джерело правди для стану застосунку, дозволяючи контролювати стан за допомогою actions та reducers. Redux дуже потужний,

але це може призвести до великої кількості бойлерплейт коду, також потрібно вручну контролювати та управляти асинхронними даними.

TanStack Query – нова бібліотека із деякими важливими перевагами, яка полегшує отримання, кешування та оновлення даних у React застосунках у простий і декларативний спосіб без зміни глобального стану (рисунок 1.2.3.1) [8].

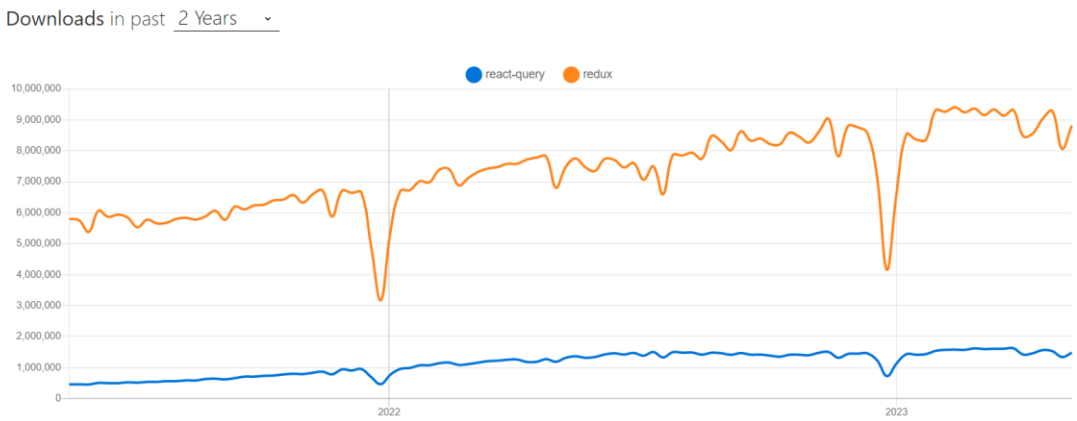


*Рис. 1.2.3.1 Порівняння кількості завантажень React-Query та Redux [5]*

Переваги TanStack Query над Redux [9]:

- Автоматичний кеш та недійсність: TanStack Query автоматично кешує запити і інвалідує їх, коли відбуваються відповідні мутації. Це означає, що не потрібно вручну контролювати кешування або інвалідацію даних, як це відбувається в Redux.
- Менше бойлерплейт коду: У TanStack Query є менше бойлерплейт коду порівняно з Redux. Основною ідеєю є використання хуків для виконання запитів і мутацій, замість створення окремих actions та reducers для кожної зміни стану.
- DevTools: TanStack Query має вбудовані DevTools, які надають корисний інтерфейс для перегляду стану кешу, активних запитів і мутацій.

- Підтримка фонових запитів: TanStack Query може обробляти фонові запити, що дозволяє додатку завантажувати дані на задньому плані, коли вони не потрібні для поточного відображення.

Для розробки сучасних React додатків, TanStack Query є більш ефективним і гнучким інструментом, особливо при роботі з асинхронними даними, отже для програмної реалізації застосунку було обрано саме цю бібліотеку.

### 1.2.4 Бібліотека компонентів

Material-UI (MUI), Chakra UI та Ant Design - це три найпопулярніші бібліотеки компонентів для React, кожна з яких має свої сильні сторони. Вони всі містять велику кількість готових до використання компонентів, що допомагають будувати інтерфейси швидко і ефективно.

MUI - бібліотека компонентів для React, яка базується на принципах Material Design від Google. Цей інструмент дозволяє швидко створювати інтуїтивно зрозумілі та естетично привабливі інтерфейси користувача для веб-додатків та мобільних застосунків. Також, MUI X пропонує набір розширених компонентів інтерфейсу користувача таких як Date and Time pickers та DataGrid

Ant Design включає великий набір компонентів та додаткові функції, такі як інтеграція з іконками та вбудованими формами. Бібліотека пропонує дуже великий вибір компонентів, проте деякі з них часто викликають питання у UX дизайнерів, що призводить до втрати часу на перевизначення структури та стилів компонентів.

Chakra UI має сильний акцент на доступність та простоту використання. Він містить гнучку систему стилів, що використовує Style Props для прямого застосування стилів до компонентів. Однак бібліотека

пропонує значно менший набір готових компонентів порівнюючи з бібліотеками описаними раніше.

Для програмної реалізації важлива швидкість розробки та можливість використовувати готові компоненти з мінімальною кастомізацією. Зважаючи на це MUI є найкращим вибором завдяки його гнучкості, обширному набору компонентів та стабільності.

## 1.3 Серверна частина

### 1.3.1 Node.js

Node.js представляє собою кросплатформне середовище виконання JavaScript з відкритим кодом, яке є популярним інструментом для широкого спектра проектів. Він працює на основі V8 JavaScript engine, яке є ядром Google Chrome, дозволяючи Node.js досягати високої продуктивності.

Під час виконання операцій введення-виведення, таких як доступ до мережі, баз даних або файлових систем, Node.js не блокує потік і не витрачає процесорний час на очікування, а продовжує обробку інших операцій після отримання відповіді. Це дозволяє Node.js ефективно обробляти велику кількість одночасних підключень до сервера без використання паралельних потоків.

Характеристики та переваги Node.js:

- Використання однієї мови програмування для клієнтської та серверної частин дозволяє розробникам зосередитися на своїй роботі без необхідності переключатися між мовами.
- Менеджер пакетів NPM надає доступ до сотень тисяч перевикористовуваних пакетів та може використовуватися для автоматизації різноманітних інструментів збірки.

- Node.js доступний на різних платформах, включаючи Windows, macOS, Linux, Solaris, FreeBSD, OpenBSD, WebOS та NonStop OS.

### 1.3.2 Фреймворк

Найпопулярнішим фреймворком для розробки серверних Node.js застосунків є Express. Це мінімалістичний і гнучкий веб-фреймворк, який надає набір різноманітних функцій для розробки веб-додатків. Він дозволяє швидко і легко налаштувати роутинг, обробку запитів та відповідей, мідлвари і т.д. Однак, це також означає, при розробці доведеться вирішувати багато речей самостійно або шукати додаткові бібліотеки.

NestJS, з іншого боку, - це прогресивний фреймворк для побудови ефективних, масштабованих серверних додатків на Node.js. Він використовує сучасний JavaScript або TypeScript і об'єднує елементи об'єктно-орієнтованого програмування, функціонального програмування і функціонально-реактивного програмування.

Характеристики та переваги Nest.js [10]:

- Nest.js пропонує модульну архітектуру, яка сприяє легкому та ефективному управлінню кодом. Це дозволяє розробникам легко організовувати свої додатки, розбиваючи їх на окремі компоненти, що спілкуються між собою.
- Використовує систему ін'єкції залежностей для спрощення та покращення коду, дозволяючи розробникам легко забезпечувати та змінювати залежності між компонентами.
- Вбудована підтримка TypeScript, яка дозволяє користуватися статичною типізацією для кращого контролю над кодом та забезпечення більш безпечної роботи додатків.

- Надає адаптери для різних платформ, таких як Express або Fastify, що дозволяє розробнику вибрати найбільш підходящу для проекту платформу.

У підсумку, для забезпечення зрозумілої структури і масштабованості серверного застосунку, NestJS є кращим вибором.

### 1.3.3 Бібліотеки class-transformer та class-validator

Бібліотеки class-transformer та class-validator використовуються для роботи з класами і об'єктами в TypeScript та JavaScript. Вони дозволяють легко перетворювати та валідувати об'єкти на основі декларативних правил.

Характеристики та переваги class-transformer та class-validator:

- class-transformer дозволяє перетворювати об'єкти між класами та різними форматами даних, такими як JSON, використовуючи декоратори та спеціальні методи. class-validator допомагає валідувати об'єкти, використовуючи гнучкі правила.
- Бібліотеки використовують декоратори для визначення правил перетворення та валідації, що дозволяє зручно структурувати код.
- Обидві бібліотеки розроблені з урахуванням TypeScript, що дозволяє користуватися їхніми перевагами, такими як статична типізація та інші сучасні можливості мови програмування.
- class-transformer та class-validator можуть бути використані разом або окремо, залежно від потреб розробки. Це дозволяє створювати модульні та гнучкі рішення.

### 1.3.4 Бібліотека для збирання даних з веб-сторінок

Найпопулярнішими JavaScript бібліотеками для тестування та скрепінгу даних з веб сторінок є Puppeteer і Cypress (рисунок 1.3.4.1).

Puppeteer - це інструмент від Google, який в основному був розроблений для автоматизації браузера Chrome, хоча тепер він також підтримує Firefox. Puppeteer може бути використаний для тестування, генерації знімків екрана, автоматичного створення PDF та веб-скрапінгу. Cypress - це ще один інструмент для автоматизації веб-браузера, який фокусується на підтримці енд-ту-енд тестування. Його основна перевага - швидкість та надійність, але він має обмеження, що стосуються підтримки різних браузерів.

Playwright, в свою чергу, нова, сучасна і високоефективна бібліотека для автоматизації тестування та веб-скрепінгу. Хоча Playwright має не таку поширеність та популярність як описані раніше бібліотеки, основною перевагою є підтримка асинхронних операцій та паралельне виконання завдань, що дозволяє оптимізувати роботу з веб-сторінками та забезпечити ефективне використання ресурсів [11].

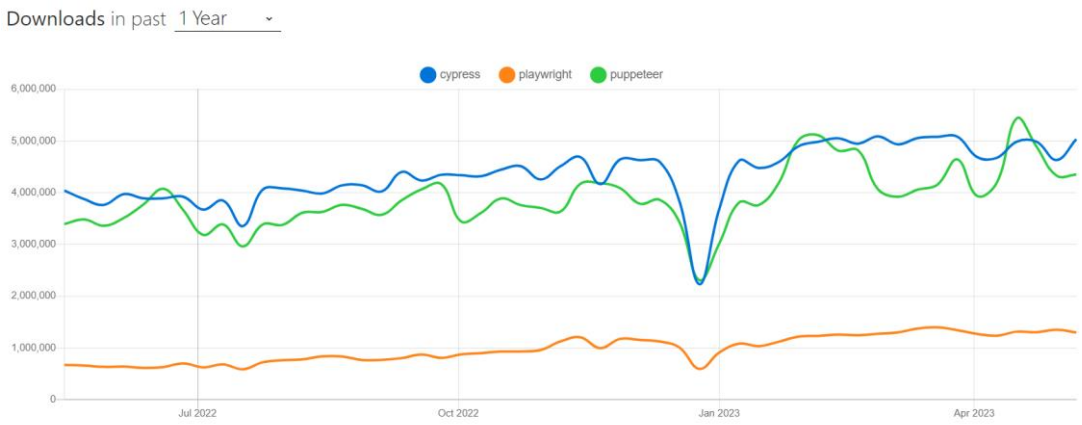


*Рис. 1.3.4.1 Порівняння кількості завантажень Cypress, Playwright and Puppeteer [5]*

## 1.4 Робота з базою даних

### 1.4.1 База даних

Бази даних – це організовані колекції даних, які зберігаються і доступні для обробки в комп'ютерній системі. Вони використовуються для зберігання великих обсягів інформації.

SQL (Structured Query Language) та NoSQL (Not only SQL) - це два основні типи систем баз даних. SQL-бази даних, такі як MySQL, Oracle та PostgreSQL, використовують реляційну модель бази даних, яка зберігає дані в структурованих таблицях. SQL використовується для написання запитів для створення, обробки та видалення даних. Реляційні бази даних добре підходять для обробки складних транзакцій та забезпечують високий рівень надійності і безпеки.

NoSQL-бази даних, такі як MongoDB, Cassandra та Redis, відрізняються від SQL тим, що вони можуть зберігати дані в нереляційних структурах, таких як ключ-значення, колоночні, документні та графові бази даних. NoSQL добре підходить для великих обсягів структурованих і неструктурованих даних і може легко масштабуватися.

Враховуючи потреби поточного проекту, PostgreSQL є гарним вибором вибором, оскілки це гнучка реляційна база даних з відкритим вихідним кодом, яка ідеально підходить для різних сценаріїв використання - від індивідуальних серверів до веб-служб з великою кількістю користувачів. Ця база даних підтримує та розширює можливості SQL.

Основні переваги PostgreSQL:

- відмінна інтеграція з ORM;
- широкий вибір типів даних;
- гарна підтримка JSON;
- висока продуктивність;
- легкість масштабування.

### 1.4.2 Object-Relational Mapping

ORM (Object-Relational Mapping) - це техніка, що дозволяє взаємодіяти з базою даних так, ніби це об'єктно-орієнтована база даних. Це спрощує розробку програмного забезпечення, дозволяючи працювати з базами даних, використовуючи об'єктно-орієнтовані парадигми. В JavaScript існує декілька популярних ORM, зокрема Sequelize, MikroORM та TypeORM.

Sequelize підтримує транзакції, зв’язки, завантаження та інше. Він підтримує Node.js і може працювати з різними SQL-діалектами, такими як PostgreSQL, MySQL, MariaDB, SQLite та Microsoft SQL Server.

MikroORM - це TypeScript ORM для Node.js, який зосереджений на перевагах статичної типізації і використовує декоратори для визначення схем. Він підтримує MongoDB, MySQL, MariaDB, PostgreSQL та SQLite.

TypeORM - це дуже гнучкий ORM, який було розроблено спеціально для TypeScript, але також може працювати з JavaScript. Він підтримує активні записи та об'єкти з даними, різні типи баз даних та має велику кількість функцій.

Для програмної реалізації застосунку було обрано TypeORM. Він має велику гнучкість, підтримує об'єктно-орієнтовані та функціональні підходи, має гарну підтримку TypeScript та багато іншого. Це робить його ідеальним для розробки складних застосунків.

Основні переваги TypeORM:

- Підтримка активних записів та репозиторіїв (Repository Pattern) для роботи з сутностями бази даних.
- Відмінна інтеграція з TypeScript і JavaScript, що полегшує інтеграцію та підтримку.
- Забезпечує універсальність, оскільки підтримує багато різних баз даних.

- Механізм міграції бази даних, що дозволяє легко та безпечно змінювати та оновлювати структуру бази даних.

## 1.5 Платформи для розгортання застосунку

### 1.5.1 Google cloud platform

GCP App Engine - це хмарна платформа як сервіс (PaaS), яка автоматично управляє масштабуванням до обслуговуванням. Це допомагає зосередитися на розробці, а не на управлінні інфраструктурою.

Cloud Run від Google - це хмарний продукт, який дозволяє розгортати та масштабувати контейнери без необхідності управляти інфраструктурою. Cloud Run використовує модель виконання без сервера, що забезпечує автоматичне масштабування.

App Engine і Cloud Run забезпечують автоматичне масштабування, але Cloud Run пропонує більшу гнучкість. Надає змогу використовувати будь-яку мову програмування, бібліотеку або фреймворк, які працюють у Docker-контейнері. Тим часом, App Engine має обмеження щодо мов програмування, технологій та бібліотек [12].

Cloud Run також масштабується автоматично відповідно до вхідного трафіку, і має інтегроване балансування навантаження. Він може використовувати екземпляри додатка, які розташовані в різних регіонах, що забезпечує високий рівень доступності.

### 1.5.2 Vercel

Vercel - це хмарна платформа, спеціально розроблена для розгортання фронтенд-додатків. Вона є ідеальним рішенням для статичних сайтів та JavaScript-додатків.

Vercel був розроблений для масштабування без будь-якого додаткового втручання. Він автоматично масштабується, щоб впоратися з будь-яким навантаженням, і забезпечує високий рівень продуктивності. Він також надає автоматичне балансування навантаження, що розподіляє трафік між серверами, що забезпечує стабільну роботу додатків.

Vercel підтримує розгортання з GitHub, GitLab та Bitbucket і автоматично створює прев'ю кожного пул-реквеста, що дозволяє легко тестувати зміни перед їх розгортанням. Крім того, Vercel пропонує вбудовану підтримку серверних функцій, що робить його ідеальним для розгортання серверних додатків на JavaScript [13].

## 1.6 Висновки до розділу 1

При аналізі технологій для розробки серверної частини (Rest API) для сучасних веб-застосунків, було обрано платформу Node.js та фреймворк Nest.js, який пропонує стабільність, гнучкість та безпеку. Для роботи з базою даних був проведений аналіз та опис  роботи з PostgreSQL, яка відбувається використовуючи TypeORM, що легко інтегрується в TypeScript проекти і спрощує роботу з реляційними базами даних.

Для огляду технологій для розробки клієнтського застосунку було обрано популярну та надійну бібліотека React, яка дозволяє створювати інтерактивні користувацькі інтерфейси та забезпечує відмінну продуктивність. Для оптимального керування запитами та кешування даних було обрано бібліотеку TanStack Query, яка пропонує декларативний підхід до отримання, кешування та оновлення даних.

Для розгортання застосунку було обрано Cloud Run через його гнучкість та підтримку Docker. Він дозволяє розгортати контейнери з будь-якими додатками, не маючи обмежень на мови програмування або бібліотеки. Крім того, Cloud Run підтримує розгортання в

мультирегіональному середовищі, що може покращити доступність та продуктивність додатка.

Для розгортання фронтенд застосунку було обрано Vercel, з огляду на його спеціалізований дизайн та високу продуктивність. Сервіс забезпечує автоматичне масштабується та балансування навантаження. Завдяки цьому, Vercel ідеально підходить для JavaScript-додатків.

Ці технології, інструменти та сервіси, дозволяють створювати надійні, легкі в підтримці та легко масштабовані веб-застосунки. Вони сприяють швидкій розробці, пропонують високу продуктивність, а також допомагають уникнути поширених проблем, що можуть виникати при розробці клієнтської та серверної частин.

# Розділ 2. Програмна реалізація

## 2.1 Технічне завдання

### 2.1.1 Функціональне призначення

Функціональним приначенням застосунку є автоматизація моніторингу, збирання, структурування та забезпечення можливості ручної валідації та корекції даних з відкритих джерел. Для реалізації було вирішено аналізувати статті та структурувати інформацію про люлей, компанії та зв’язки між ними.

### 2.1.2 Основні вимоги

Застосунок повинен складатися з чотирьох частин: база даних, Rest API, scheduled function і клієнтська частина (рисунок 2.1.2.1).

База даних зберігає всі дані, що використовуються застосунком. Вона містить інформацію про користувачів, їх дії, налаштування та багато іншого. Використовуючи SQL застосунок може здійснювати операції читання, запису, оновлення та видалення (CRUD) в базі даних.

Rest API - це спосіб, яким застосунок взаємодіє з базою даних. API служить мостом між клієнтською частиною та базою даних, приймаючи запити від клієнтської частини, обробляючи їх та передаючи їх базі даних. Коли база даних повертає відповідь, API передає цю відповідь назад до клієнтської частини.

Регулярно запланована функція - це скрипт, який запускаються автоматично за певним графіком. Вона відповідає за збір та оновлення даних. Функція взаємодіє з базою даних напряму.

Фронт-енд - це та частина веб-застосунку, яку бачить користувач. Він взаємодіє з Rest API для відправки запитів до бази даних та отримання відповідей.

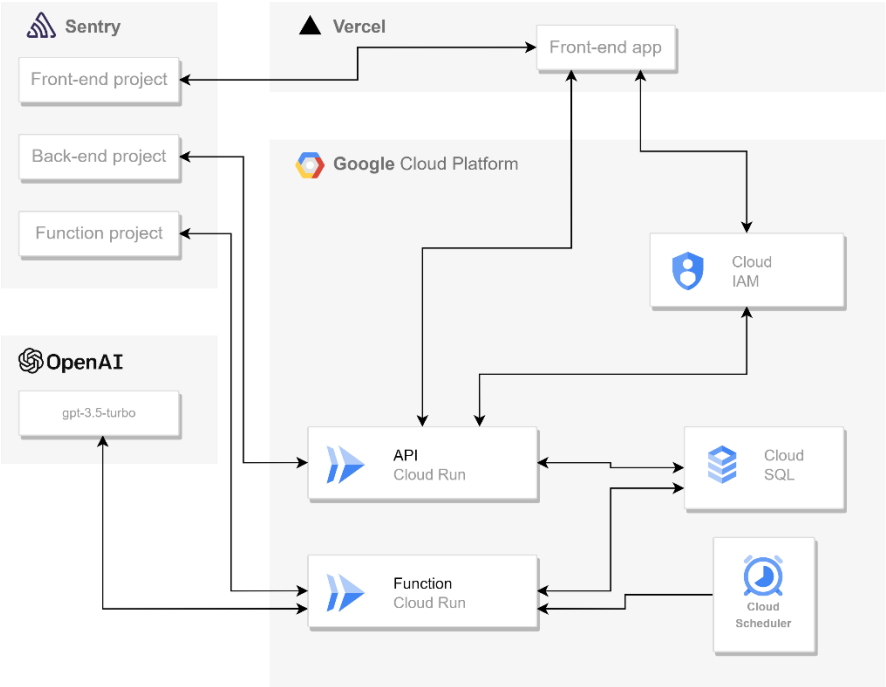


*Рис. 2.1.2.1 Взаємодія між компонентами застосунку*

### 2.1.2.1 Вимоги до серверної частини застосунку

Для ефективної структуризації даних необхідно розробити чіткі моделі даних, що відображають взаємозв'язки між різними сутностями бізнес-процесу. Кожна модель даних має бути чітко описана з урахуванням її полів, типів та обмежень.

Ручна валідація даних - це важлива частина функціоналу, яка повинна дозволяти користувачам перевіряти дані перед тим, як вони будуть збережені в базі даних. Це може бути реалізовано за допомогою сторінок для перегляду

та редагування, які дозволяють користувачам переглядати вхідні дані, редагувати їх та видаляти помилкові записи перед підтвердженням.

Також для роботи застосунку необхідна аутентифікація, яка дозволяє перевірити ідентичність користувача, який намагається отримати доступ до системи.

REST (Representational State Transfer) API - це програмний інтерфейс, який використовує HTTP запити для створення, читання, оновлення та видалення даних. REST API повинен бути гнучким та сумісним для інтеграції з іншими системами. Гнучкість REST API дає можливість майбутньої масштабованості та розширення функціоналу.

Функція для веб-скрапінгу має бути надійною та стабільною. Вона має коректно обробляти помилки та відмови, наприклад, якщо веб-сайт тимчасово недоступний або якщо структура веб-сайту змінилась. Після отримання даних потрібно їх обробити та структурувати перед збереженням у базі даних.

Також, потрібно визначити частоту запуску функції для веб-скрапінгу. Запуск може відбуватися кожні кілька хвилин, годин, днів або тижнів в залежності від вимог до свіжості даних.

Потрібно оптимізувати заплановані функції, щоб вони не створювали надмірного навантаження на цільовий веб-сайт. Занадто часті запити можуть призвести до блокування IP-адреси.

Архітектурні вимоги до серверної частини застосунку:

- Масштабованість
- Висока доступність
- Безпека
- Модульність
- Тестованість
- Документація API
- Логування і моніторинг

#### 2.1.2.2 Вимоги до клієнтської частини застосунку

Вимоги до клієнтської частини застосунку включають наступні ключові аспекти:

- Інтуїтивно зрозумілий інтерфейс: Інтерфейс застосунку має бути простим і зрозумілим для користувачів з різним досвідом використання подібних систем.
- Функціональність: Клієнтська частина повинна надавати всі необхідні можливості для виконання задач, передбачених бізнес-логікою застосунку.
- Швидкість і відповідь: Застосунок повинен швидко реагувати на дії користувача і надавати чітку зворотню відповідь.
- Безпека: Клієнтська частина повинна використовувати безпечні методи обробки даних і забезпечувати конфіденційність персональних даних користувача.
- Доступність: Застосунок має бути доступним для різних груп користувачів, включаючи людей з обмеженими можливостями.

## 2.2 Структура бази даних

Після аналізу вимог, було розроблену структуру бази даних з урахуванням вимог до масштабованості, продуктивності, та безпеки даних. Крім того, вони повинні бути нормалізовані для уникнення повторення даних та забезпечення цілісності даних (рисунок 2.2.1):

- user - зберігає інформацію про користувачів, інформацію для входу (наприклад, ім'я користувача, захищений хеш-код пароля), та інші персональні дані.
- raw_entity – зберігає дані, які необхідно валідувати. Це може включати сирий контент, отриманий від веб-скрапінгу.
- company – сутність, яка відповідає за поєднання всієї інформації про компанію.
- company_history – відповідає за зберігання історичних даних про компанії. Може включати зміни у структурі компанії, попередні адреси, зміни у керівництві та інші значні події.
- company_relation_type – визначає типи відносин між компаніями, які використовуються в таблиці company_relation.
- person – сутність, яка відповідає за поєднання всієї інформації про особу.
- person_company_relation – описує відношення між особами та компаніями, наприклад, чи є даний працівник CEO, менеджером, акціонером тощо.
- person_company_relation_type – визначає типи відносин між особами та компаніями, які можуть бути використані в таблиці person_company_relation.
- person_history – таблиця для зберігання історичних даних про осіб. Може включати зміни в позиціях, попередні компанії, значні досягнення, тощо.

- person_relation – описує взаємозв'язки між різними особами, наприклад, родинні зв'язки, професійні зв'язки тощо.
- person_relation_type – визначає типи відносин між особами, які використовуються в таблиці person_relation.
- scrape_keyword – містить ключові слова, які використовуються для веб-скрапінгу. Ключові слова можуть бути пов'язані з назвами компаній, іменами осіб, подіями, характеристиками та іншими даними, які потрібно зібрати.

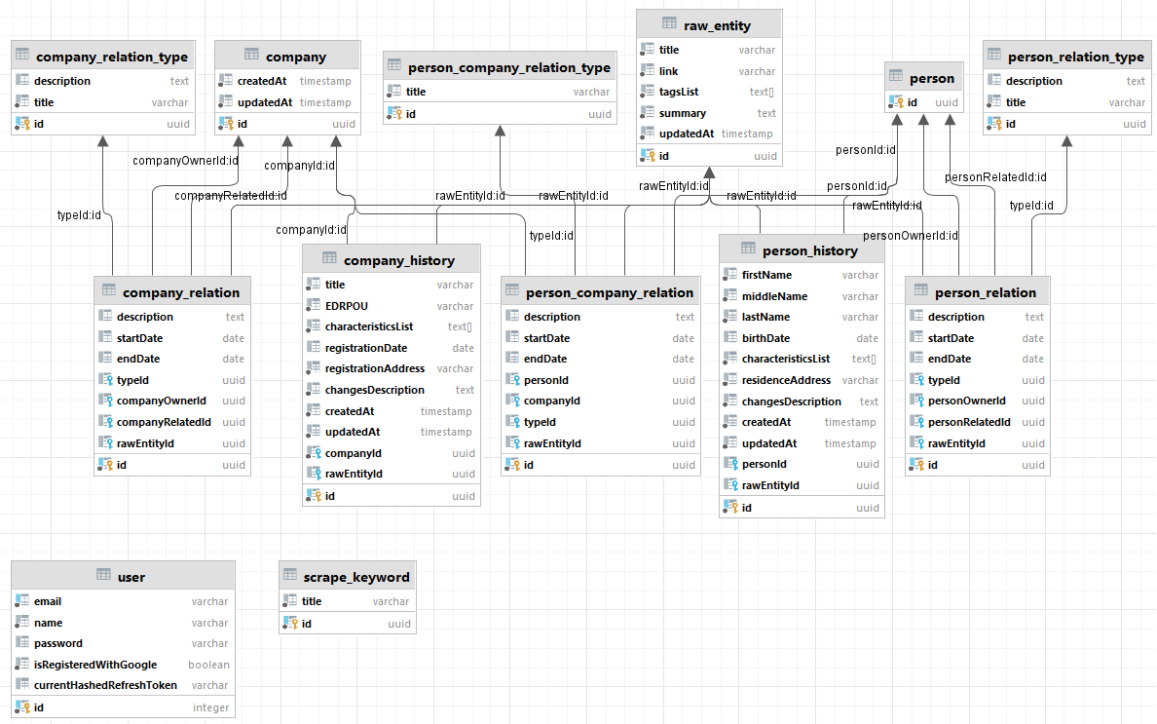


*Рис. 2.2.1 Структура бази даних*

## 2.3 Налаштування середовища розробки

### 2.3.1 Керування версіями Node.js

Node Version Manager (nvm) - це інструмент, який дозволяє керувати декількома версіями Node.js на комп'ютері. Для встановлення nvm на Linux або Mac потрібно виконати наступні кроки:

1. Виконати в терміналі одну з наступних команд. Можна використати curl або bash. Ці команди клонують репозиторій nvm до каталогу ~/.nvm.
   a. `curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.1/install.sh | bash`
   b. `wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.39.1/install.sh | bash`
2. Після встановлення nvm, потрібно відкрити файл конфігурації (наприклад, ~/.bashrc, ~/.zshrc або ~/.profile) використовуючи текстовий редактор та додати наступний рядок коду в кінці файлу:

```
export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh"
```

Коли nvm встановлено та налаштовано, його можна використовувати для керування версіями Node.js. Для розробки буде використовуватися версія 18.16.0:

- Перевірити доступні версії Node.js: `nvm ls-remote`
- Встановити певну версію Node.js: `nvm install <версія>`
- Переключитися між встановленими версіями Node.js: `nvm use <версія>`
- Видалити певну версію Node.js: `nvm uninstall <версія>`

### 2.3.2 Встановлення pnpm

Для роботи pnpm потрібно встановити Node.js принаймні 16.14 версії. Ця вимога задовольняється, оскільки на попередньому кроці було втсановлено версію 18.16.0. Для середовища виконання JavaScript Node.js менеджером пакетів за замовчуванням є npm. Отже, його можна використати для завантаження pnpm: `npm install -g pnpm`.

Список команд еквівалентів npm для початку роботи:

- `npm install: pnpm install`
- `npm i <pkg>: pnpm add <pkg>`
- `npm run <cmd>: pnpm <cmd>`

За допомогою pnpm залежність зберігатиметься в сховищі з адресою вмісту, тому якщо ви залежите від різних версій залежності, до сховища додаються лише ті файли, які відрізняються. Наприклад, якщо він містить 100 файлів, а нова версія містить зміни лише в одному з цих файлів, оновлення pnpm додасть лише 1 новий файл до сховища замість того, щоб клонувати всю залежність лише для єдиної зміни.

Усі файли зберігаються в одному місці на диску. Коли пакети інстальовано, їхні файли жорстко пов’язуються з цього єдиного місця, не займаючи додаткового місця на диску.

### 2.3.3 Встановлення та запуск Docker Desktop

Для того щоб встановити та запустити Docker Desktop на Mac виконати наступні кроки:

1. Завантажити файл Docker.dmg з офіційного сайту docker.com.
2. Виконати наступні команди в терміналі, щоб встановити Docker Desktop:

   ```
   sudo hdiutil attach Docker.dmg
   ```

```
sudo
/Volumes/Docker/Docker.app/Contents/MacOS/install
sudo hdiutil detach /Volumes/Docker
```

Після завершення встановлення, потрібно знайти та запустити Docker Desktop у папці "Програми". Він почне працювати у фоновому режимі та дозволить використовувати Docker для створення та управління контейнерами.

## 2.4 Front-end

### 2.4.1 Створення та налаштування проекту Vite

Vite вимагає Node.js версії 14.18+, 16+. Для створення нового проекту Vite з React та TypeScript, в терміналі потрібно виконати наступну команду: `pnpm create vite react-ts-app --template react-ts`. Після успішної виконання команди, в обраній директорії буде створено проект з Vite.

Для запуску проекту потрібно встановити залежності виконавши команду `pnpm install.` Після успішного встановлення залежностей проект можна запустити виконавши команду `pnpm dev`. За замовчуванням проет буде запущено локально на 5173 порті (рисунок 2.4.1.2).

```
VITE v4.3.5  ready in 2424 ms

→ Local:   http://localhost:5173/
→ Network: use --host to expose
→ press h to show help
```

*Рис. 2.4.1.2 Результат виконання команди pnpm dev*

Для тестування production збірки проету Vite пропонує наступні команди, які визначені у файлі package.json:

- `"build": "tsc && vite build"`
- `"preview": "vite preview"`

Команда build запускає збірку проекту в результаті якої буде створено папку dist в корені проекту (рисунок 2.4.1.3). Щоб запустити зібраний production build потрібно виконати команду `pnpm preview.`

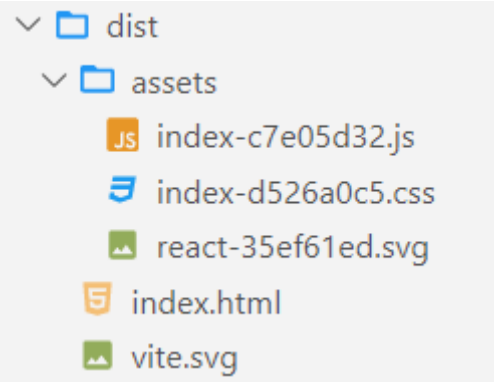


*Рис. 2.4.1.3 Вміст папки dist*

### 2.4.2 Налаштування TanStack Query

TanStack Query сумісний з React v16.8+ і працює з ReactDOM і React Native. Для завантаження необхідно виконати команду `pnpm add @tanstack/react-query` в директорії проекту.

Для виявлення помилок та невідповідностей рекомендується використовувати ESLint Plugin Query. Плагін - це окремий пакет, який потрібно встановити: `pnpm add -D @tanstack/eslint-plugin-query`. Після цього, додати `"@tanstack/query"` до розділу плагінів файлу конфігурації .eslintrc: `{ "plugins": ["@tanstack/query"] }`. Підключення рекомендованих правил для плагіна: `{ "extends": ["plugin:@tanstack/eslint-plugin-query/recommended"] }`.

Налаштування TanStack Query здійснюється у файлі src/main.tsx. QueryClient приймає різні параметри, що дозволяють налаштувати повторні запити та обробку помилок (рисунок 2.4.2.1). За замовчуванням, без використання дженериків, помилка буде визначена як `unknown`. Це може здатися помилкою, адже зазвичай помилка має тип Error. Але насправді, це зроблено навмисно, оскільки у JavaScript можна викидувати будь-що - не обов'язково об'єкт типу Error.

Оскільки TanStack Query не контролює функцію, яка повертає Promise, він також не може знати, які типи помилок вона може генерувати. Тому `unknown` тип є правильним. Отже, потрібно вручну звузити тип, виконавши перевірку за допомогою оператора `instanceof`. Після створення QueryClient instance, весь додаток слід обгорнути у QueryClientProvider, до якого передається раніше створений queryClient.

```
const queryClient :QueryClient = new QueryClient( config: {
  defaultOptions: {
    queries: {
      retry: (failureCount: number, error) :boolean => {
        if (error instanceof AxiosError) {
          return (
            (!error?.response?.status || error?.response?.status >= 500) &&
            failureCount <= 3
          );
        }
        return failureCount <= 3;
      },
      useErrorBoundary: (error) :boolean => {
        if (error instanceof AxiosError) {
          return (
            !error?.response?.status ||
            error?.response?.status === 404 ||
            error?.response?.status >= 500
          );
        }
        return true;
      },
    },
  },
});
```

*Рис. 2.4.2.1 Створення QueryClient instance*

Ключі запитів відіграють важливу роль у концепції TanStack Query. Вони потрібні для забезпечення кешування даних та автоматичного їх оновлення у разі зміни залежностей запиту. Більше того, це дозволяє користувачам вручну взаємодіяти з кешем запитів, коли це потрібно, наприклад, при оновленні даних після мутації, реалізації функції optimistic update, prefetching чи при необхідності ручного скасування запитів.

Всередині кешу запитів знаходиться простий об'єкт JavaScript, в якому ключі представлені у вигляді серіалізованих ключів запитів, а значення складаються з даних запиту та метаданих. Ключі формуються упорядкованим способом, тому можна також використовувати об'єкти (однак на найвищому рівні ключі мають бути рядками або масивами). Головне, що ключі мають бути унікальними для кожного запиту. Якщо TanStack Query виявляє запис для ключа в кеші, він використовується.

Зручним способом організації ключів є фабрика. Це простий об'єкт з записами та функціями, які генерують ключі запитів, які можуть використовуватися в хуках (рисунок 2.4.2.2).

```
export const companiesKeys : {all: readonly ["companies"], detail: (id: string) => readonly ["com...,
  all: ['companies'] as const,
  lists: () => [...companiesKeys.all, 'list'] as const,
  list: (filters: string) => [...companiesKeys.lists(), { filters }] as const,
  details: () => [...companiesKeys.all, 'detail'] as const,
  detail: (id: string) => [...companiesKeys.details(), id] as const,
};
```

*Рис. 2.4.2.2 Приклад фабрики ключів*

Для забезпечення можливості перевикористання та уникнення дублювання коду, було вирішено розділити TanStack Query хуки та axios запити та створити для них окремі директорії (рисунок 2.4.2.3). Це дозволтить використовувати axios запити не тільки разом з хуками, а і у селектах з можливістю пошуку.

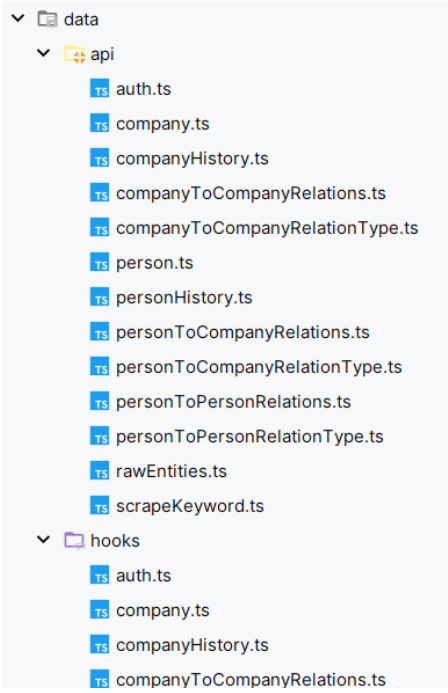


*Рис. 2.4.2.3 Файлова структура для обробки запитів*

Файли з axios запитами містять функції, які відносяться до відповідної тематики (рисунок 2.4.2.4). Кожна функція повертає Promise з певним типом, який автоматично буде використаний в TanStack Query хуках.

```
import axiosApi from '../axiosApi';
import { Company } from '../types';

5+ usages  Roman Vozbrannyi
export const fetchCompaniesList = async (search: string): Promise<Company[]> => {
  return await axiosApi
    .get( url: `/company?search=${search}`)
    .then((response :AxiosResponse<any, any> ) => response.data);
};

2 usages  Roman Vozbrannyi
export const fetchCompany = async (id: string): Promise<Company> => {
  return await axiosApi.get( url: `/company/${id}`).then((response :AxiosResponse<any, any> ) => response.data);
};

2 usages  Roman Vozbrannyi
export const deleteCompany = async (id: string): Promise<void> => {
  await axiosApi.delete( url: `/company/${id}`).then((response :AxiosResponse<any, any> ) => response.data);
};
```

*Рис. 2.4.2.4 Вміст файлу api/company.ts*

TanStack Query надає два основних хуки: useQuery та useMutation. Оскільки, типи вже були визначені у функціях запитів, TanStack Query також буде розуміти типи даних та дозволить використовувати їх в подальшому використанні створених хуків. useQuery та useMutation спрощують роботу з асинхронними запитами у React-додатках (рисунок 2.4.2.5):

- useQuery – це хук, який використовується для отримання даних із сервера. Він автоматично керує кешуванням та оновленням даних. Також, хук дозволяє легко відслідковувати стан завантаження, помилки та успіху запиту. useQuery приймає ключ запиту, функцію, яка повертає проміс (зазвичай запит до API), та додаткові параметри конфігурації.
- useMutation – це хук, призначений для зміни даних на сервері, таких як створення, оновлення чи видалення. Він допомагає управляти станом мутації, включаючи помилки та успішні

виконання. Цей хук приймає функцію, яка повертає проміс, та додаткові параметри конфігурації. Після успішного виконання мутації, useMutation може автоматично оновлювати кеш даних та надає можливість інвалідувати кешовані запити вручну.

```
import { companiesKeys, rawEntitiesKeys } from '../queryKeys';
import { useMutation, useQuery, useQueryClient } from '@tanstack/react-query';
import { deleteCompany, fetchCompaniesList, fetchCompany } from '../api/company';

4 usages  Roman Vozbrannyi
export const useCompaniesList = (search: string) =>
  useQuery(companiesKeys.list(search),  queryFn: () => fetchCompaniesList(search));

no usages  Roman Vozbrannyi
export const useCompany = (id: string) =>
  useQuery(companiesKeys.detail(id),  queryFn: () => fetchCompany(id));

2 usages  Roman Vozbrannyi
export const useDeleteCompany = () => {
  const queryClient :QueryClient  = useQueryClient();
  return useMutation( mutationFn: ({ id }: { id: string }) => deleteCompany(id),  options: {
    onSuccess: () :void  => {
      queryClient.invalidateQueries(companiesKeys.all);
      queryClient.invalidateQueries(rawEntitiesKeys.all);
    },
  });
};
```

*Рис. 2.4.2.5 Вміст файлу hooks/company.ts*

## 2.5 Серверна частина

### 2.5.1 Визначення структури та створення Nest.js REST API

Структура REST API-додатка на основі Nest.js має декілька ключових елементів, які допомагають організувати код та забезпечити чистоту архітектури.

Modules – це основні блоки додатка, які забезпечують групування пов'язаних функцій. Модулі можуть включати контролери, провайдери, сервіси та інші модулі, і вони допомагають створювати структуру додатка за принципом поділу на домени чи функціональність.

Controllers відповідають за обробку вхідних HTTP-запитів та відправку відповідей. Вони взаємодіють з сервісами, щоб отримати дані, і генерують відповіді на запити клієнтів (рисунок 2.5.1.2).

```
import { Body, Controller, Delete, Param, Post } from '@nestjs/common';
import { CompanyHistoryService } from './company-history.service';
import { CompanyHistory } from './company-history.entity';
import { CreateCompanyHistoryDto } from './dto/create-company-history.dto';
import { ApiTags } from '@nestjs/swagger';

2 usages  Roman Vozbrannyi
@ApiTags( tags: 'Company history')
@Controller( prefix: 'company-history')
export class CompanyHistoryController {
  no usages  Roman Vozbrannyi
  constructor(private readonly companyHistoryService: CompanyHistoryService) {}

  5+ usages  Roman Vozbrannyi
  @Post()
  create(
    @Body() createCompanyHistoryDto: CreateCompanyHistoryDto,
  ): Promise<CompanyHistory> {
    return this.companyHistoryService.create(createCompanyHistoryDto);
  }

  6+ usages  Roman Vozbrannyi
  @Delete( path: ':id')
  async delete(@Param( property: 'id') id: string): Promise<void> {
    return this.companyHistoryService.delete(id);
  }
}
```

*Рис. 2.5.1.2 Вміст файлу company-history.controller.ts*

Services містять бізнес-логіку та відповідають за обробку даних, отриманих від контролерів. Вони можуть використовувати репозиторії та інші засоби для отримання та збереження даних, а також обробляти помилки та валідацію (рисунок 2.5.1.3).

```
@Injectable()
export class CompanyHistoryService {
  no usages  Roman Vozbrannyi
  constructor(
    @InjectRepository(CompanyHistory)
    private readonly companyHistoryRepository: Repository<CompanyHistory>,

    @InjectRepository(Company)
    private readonly companyRepository: Repository<Company>,

    @InjectRepository(RawEntity)
    private readonly rawEntityRepository: Repository<RawEntity>,
  ) {}

  5+ usages  Roman Vozbrannyi
  async create(
    companyHistory: CreateCompanyHistoryDto,
  ): Promise<CompanyHistory> {
    let company: Company;

    if (companyHistory.companyId) {
      company = await this.companyRepository.findOneBy( where: {
        id: companyHistory.companyId,
      });
```

*Рис. 2.5.1.3 Вміст файлу company-history.service.ts*

Repositories - це класи, які відповідають за взаємодію з базою даних. Вони забезпечують абстракцію від конкретної бази даних та допомагають створювати тести без залежності від бази даних.

DTO (Data Transfer Objects) - це об'єкти, які використовуються для передачі даних між різними шарами додатка. Вони дозволяють валідувати дані та гарантувати безпеку даних, які відправляються та отримуються (рисунок 2.5.1.4).

```
import {
  IsArray,
  IsDateString,
  IsOptional,
  IsString,
  IsUUID,
} from 'class-validator';

4 usages   Roman Vozbrannyi
export class CreateCompanyHistoryDto {
  @IsOptional()
  @IsUUID( version: 4)
  companyId: string;

  @IsOptional()
  @IsUUID( version: 4)
  rawEntityId: string;

  @IsOptional()
  @IsString()
  title: string;

  @IsOptional()
```

*Рис. 2.5.1.4 Вміст файлу create-company-history.dto.ts*

Validators - це класи, які допомагають перевіряти вхідні дані та гарантувати, що вони відповідають встановленим вимогам.

Middlewares - це функції, які виконуються перед обробкою запиту контролером. Вони можуть використовуватися для перевірки автентифікації, валідації, логування та інших завдань, які потрібно виконати перед тим, як запит буде оброблений.

Interceptors використовуються для перехоплення та зміни відповідей або помилок перед тим, як вони будуть надіслані клієнту. Це може включати перетворення даних, додавання спеціальних заголовків та інше.

Exceptions - це класи, які можуть бути використані для обробки помилок у додатку. Вони допомагають забезпечити коректну обробку помилок та відправку коректних HTTP-кодів відповідей.

Guards дозволяють контролювати доступ до різних частин додатка на основі даних про автентифікацію користувача або інших умов. Вони забезпечують безпеку додатка шляхом встановлення правил доступу до ресурсів.

```
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

3 usages  Roman Vozbrannyi
@Injectable()
Roman Vozbrannyi
export default class JwtAuthenticationGuard extends AuthGuard( type: 'jwt') {}
```

*Рис. 2.5.1.5 Вміст файлу jwt-authentication.guard.ts*

Загалом, структура додатка Nest.js REST API допомагає створювати чисту, добре організовану архітектуру, яка спрощує розробку та підтримку коду. Використання таких елементів, як модулі, контролери, сервіси та інші, дозволяє розбити додаток на окремі частини, які легко підтримувати та тестувати (рисунок 2.5.1.6).

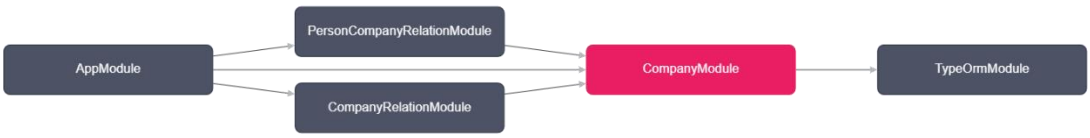


*Рис. 2.5.1.6 Структура модулів, що приймають участь у обробці запитів пов’язаних з компаніями*

На основі визначеної архітектури, було створено модулі, контролери та сервіси для кожної сутності додатка. Модулі групують пов'язані функції, контролери обробляють вхідні HTTP-запити та відправляють відповіді, а сервіси містять бізнес-логіку та відповідають за обробку даних. Вони

взаємодіють з репозиторіями для отримання та збереження даних, а також обробляють помилки та валідацію. Для керування відповідями або помилками використовуються перехоплювачі (interceptors), які перетворюють дані, додають спеціальні заголовки або виконують інші дії перед відправкою відповіді клієнту (рисунок 2.3.1.7).


*Рис. 2.5.1.7 Процес обробки запиту*

### 2.5.2 Аутентифікація за допомогою Google

Для аутентифікації використовуючи Google в Nest.js REST API було використано @nestjs/jwt, googleapis та JwtStrategy. По-перше, додаток було зареєстровано в Google Cloud Console, щоб отримати необхідні ключі та ідентифікатори клієнта. Ці дані будуть використані для налаштування модулю Google OAuth. Другим кроком є інтеграція бібліотеки googleapis. Ця бібліотека дозволяє використовувати Google APIs для аутентифікації користувачів. Вона буде використовуватися для перевірки токенів отриманих від Google (рисунок 2.5.2.1).

```
async authenticate(token) :Promise<...>  {
    const tokenInfo :TokenInfo  = await this.oauthClient.getTokenInfo(token);
    const email :string  = tokenInfo.email;
    try {
        const user = await this.usersService.getByEmail(email);
        return this.handleRegisteredUser(user);
    }
    catch (error) {
        if (error.status !== 404) {
            throw new error();
        }
        return this.registerUser(token, email);
    }
}
```

*Рис. 2.5.2.1 Використання googleapis oauthClient*

Після цього, для генерації та перевірки JWT (JSON Web Tokens) було використано модуль @nestjs/jwt. Ці токени потрібні для ідентифікації користувачів після того, як вони успішно автентифікувалися через Google. Наступним кроком є створення JwtStrategy. Ця стратегія використовується Passport.js для перевірки JWT та визначає, як токени повинні бути перевірені та як витягнути дані користувача з цих токенів (рисунок 2.5.2.2).

```
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  no usages   Roman Vozbrannyi
  constructor(
    private readonly configService: ConfigService,
    private readonly userService: UsersService,
  ) {
    super({
      jwtFromRequest: ExtractJwt.fromExtractors( extractors: [
        (request: Request) => {
          return request?.headers?.['access-token'] as string;
        },
      ]),
      secretOrKey: configService.get('JWT_ACCESS_TOKEN_SECRET'),
    });
  }

  no usages   Roman Vozbrannyi
  async validate(payload: TokenPayload) :Promise<User>  {
    return this.userService.getById(payload.userId);
  }
}
```

*Рис. 2.5.2.2 JwtStrategy*

Завершальним кроком є налаштування маршрутів та контролерів для обробки процесу аутентифікації. Потрібно створити маршрут для обробки відповіді від Google після аутентифікації (рисунок 2.5.2.3).

```
@ApiTags( tags: 'Google authentication')
@Controller( prefix: 'google-authentication')
@UseInterceptors(ClassSerializerInterceptor)
export class GoogleAuthenticationController {
  no usages   Roman Vozbrannyi
  constructor(
    private readonly googleAuthenticationService: GoogleAuthenticationService,
  ) {}

  1 usage   Roman Vozbrannyi
  @Post()
  async authenticate(
    @Body() tokenData: TokenVerificationDto,
    @Req() request: Request,
  ) :Promise<User>  {
    const { accessToken :string , refreshToken :string , user :User  } =
      await this.googleAuthenticationService.authenticate(tokenData.token);

    request.res.setHeader('access-token', accessToken);
    request.res.setHeader('refresh-token', refreshToken);

    return user;
  }
}
```

*Рис. 2.5.2.3 Процес обробки запиту*

### 2.5.3 Визначення структури та створення функції для збирання даних

Архітектура функції скрапера Playwright в Node.js та кроки її реалізації включають декілька ключових елементів. Спочатку, потрібно ініціалізувати Playwright. Другим кроком є створення нового екземпляра браузера нової сторінку в браузері. На третьому кроці відбувається перехід на потрібний URL. Після цього, Playwright може виконувати різні дії на сторінці, такі як кліки по елементам, введення тексту в поля та інше.

Playwright дозволяє вибирати елементи на сторінці за допомогою CSS-селекторів або XPath, і витягувати інформацію з них. Це може бути текст

елемента, атрибути, такі як href або src, та інше. Після того, як дані були зібрані, їх можна обробити, зберегти або відправити далі. Наприклад, дані можуть бути збережені в базі даних або відправлені через API. Останнім кроком є закриття браузера, щоб звільнити ресурси.

Для створення функції, спочатку було створено окремий файл . В цьому файлі було імпортовано бібліотеку Playwright та ініціалізовано її. Також було створено асинхронну функцію main(), яка призначена для виконання всієї роботи по збиранню даних. У цій функції створено новий екземпляр браузера та нову сторінку. Потім сторінка переходила на заданий URL, використовуючи метод page.goto().

Далі було реалізовано збирання даних. Для цього було створено декілька допоміжних функцій, які використовувались для вибору елементів на сторінці та витягування інформації з них. Використовувались CSS-селектори, а також методи Playwright для витягування тексту та атрибутів з цих елементів.

Спочатку функція отримує список посилань на статті зі списку на сторінці. Після цього, відбувається перехід на кожну, та аналіз тексту статті.

Аналіз статті включає:

- Визначення чи підходить стаття по ключовим словам, які мають бути визначені користувачем у користувацькому інтерфейсі клієнтської частини застосунку та збережені до бази даних (рисунок 2.5.3.1).

```
if (
  scrapeKeywordsList.some(
    (keyword : ObjectLiteral ) =>
      content.includes(keyword.title) || title.includes(keyword.title),
  )
) {
```

*Рис. 2.5.3.1 Перевірка ключових слів*

- Обробка тексту для статті для зменшення кількості слів.

- Використання OpenAI ChatGPT api для обробки тексту статті та структурування інформації (рисунок 2.5.3.2).

```
const response :AxiosResponse<CreateChatComple...  = await openai.createChatCompletion(
  model: 'gpt-3.5-turbo',
  messages: [
    {
      role: 'user',
      content: prompt,
    },
  ],
});
```

*Рис. 2.5.3.2 Використання OpenAI ChatGPT api*

- Перевірка структури оброблених даних та виконання повторних запитів до ChatGPT у разі помилки (рисунок 2.5.3.3). Перевірка здійснюється за допомогою бібліотеки Zod. Це об'єктно-орієнтована бібліотека, яка дозволяє створювати схеми, що відображають структуру даних. Робота Zod полягає в перевірці вхідних даних на відповідність певній схемі.

```
export const validateExtractedData = (
  data: unknown,
): z.infer<typeof ExtractionResult> | null => {
  const validationResult :SafeParseSuccess<{persons?: Zo... | SafeParseError<{persons?: ZodA...  = ExtractionResult.safeParse(data);

  if (validationResult.success === false) {
    return null;
  }
  return validationResult.data;
};
```

*Рис. 2.5.3.3 Перевірка структури оброблених даних*

Після успішного аналізу, назва статті, посилання та структурована інформація зберігається у базі даних, створюючи таким чином централізований репозиторій даних, що буде використаний для подальшої обробки та верифікації даних користувачем (рисунок 2.5.3.4).

```
await AppDataSource.getRepository( target: 'RawEntity').save( entity: {
  title,
  link: article,
  tagsList: scrapeKeywordsList
    .filter(
      (keyword : ObjectLiteral ) =>
        content.includes(keyword.title) ||
        title.includes(keyword.title),
    )
    .map((keyword : ObjectLiteral ) => keyword.title),
  summary: jsonString || '',
});
```

*Рис. 2.5.3.4 Зберігання обробленої інформації у базі даних*

## 2.4 Висновки до розділу 2

У даному розділі було визначено технічне завдання та детально розглянуто ключові етапи розробки серверної та клієнтської частини застосунку. Починаючи з визначення структури та створення Nest.js REST API, описано основні компоненти архітектури Nest.js, які включають модулі, контролери, сервіси, DTO та інші. Також, показано як розробляти ці компоненти таким чином, щоб забезпечити високу якість коду, зручність тестування та масштабованість додатку.

Також, було описано використання бібліотеки @nestjs/jwt та googleapis для створення безпечної системи автентифікації. Ця система надає засоби для верифікації токенів, що отримані від Google, і генерує власні JWT токени для подальшого використання у додатку.

Що стосується визначення структури та створення функції для збирання даних, було описано використання Playwright для автоматизованого

збирання даних з веб-сторінок, аналіз тексту за допомогою ChatGPT api, валідація за допомогою бібліотеки Zod та збереження структурованої інформації до бази даних.

# Розділ 3. Автоматизоване розгортання, CI/CD інтеграція

Автоматизація процесів CI/CD (Continuous Integration/Continuous Deployment) значно спрощує розробку, дозволяючи фокусуватися на коді, замість мануального оновлення серверів або тривалого тестування.

Для створення гнучкого та ефективого CI/CD пайплайну з використанням Github, Github Actions, Docker, Google Cloud Run та Vercel потрібно виконанти наступні кроки:

- Створення репозиторію на Github.
- Налаштування Github Actions.
- Ствоерння файлів Dockerfile, що дозволить пакувати ваш застосунок та його залежності в контейнер, що забезпечує єдність середовища розробки при розробці, тестуванні та продакшні.
- Налаштування Google Cloud Run, що дозволяє запускати контейнери в хмарі.
- Розгортання фронтенду на Vercel.

Загальна схема CI/CD передбачує, що розробник спочатку завантажує код свого проекту на Github, використовуючи систему контролю версій Git. Після того, як код завантажено, Github Actions, інструмент для автоматизації робочого процесу, починає свою роботу. Він налаштований для автоматичного виконання ряду завдань, таких як збірка, тестування та розгортання коду.

Github Actions використовує Docker для створення образів контейнерів з кодом проекту. Після створення Docker image може бути завантажено до Google Cloud Run. Vercel, в свою чергу надає Github app, який автоматизує збірку та розгортання коду клієнтської частини застосунку (рисунок 3.1).

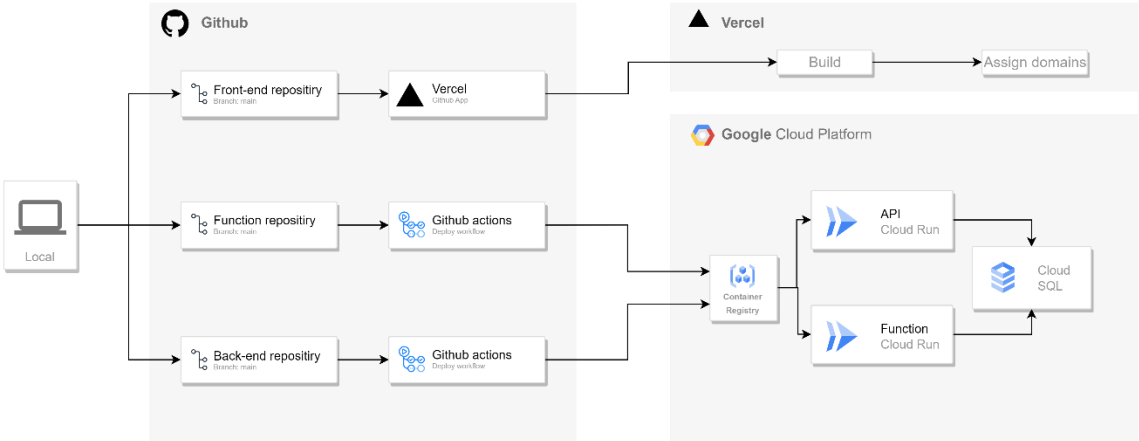


*Рис. 3.1 Схема розгортання застосунку*

## 3.1 CI/CD інтеграція Nest.js REST API та Scheduled function

### 3.1.1 Використання Dockerfile

Для запуску локально та розгортання застосунку в ізольованому середовищі було створено Dockerfile, який створює Docker-образ для Node.js-застосунку, використовуючи multistage процес збірки. Multistage збірка є ефективним способом мінімізації розміру образу та забезпечення, що в контейнері є лише те, що дійсно потрібно для запуску застосунку.

Опис кроків збірки, визначених у Dockerfile:

- Base stage: використовується офіційний образ Node.js (версія 18) на базі Alpine Linux, що є легким дистрибутивом Linux. Після цього відбувається втановлення pnpm.
- Dependencies stage: виконується створення робочої директорії /app і копіювання package.json та pnpm-lock.yaml у цю

директорію. Після цього – встановлення залежностей за допомогою pnpm install.

- Build stage: виконується копіювання всього коду застосунку в робочу директорію, а потім копіювання node_modules з dependencies stage. Після цього відбувається збірка застосуноку за допомогою pnpm build і видалення непотрібних залежностей, залишаючи тільки залежності для production.
- Deploy stage: це кінцева стадія, в якій створюється образ для розгортання. Відбувається копіювання зібраних файлів та node_modules з build stage в робочу директорію. Після цього відбувається встановлення команди, яка запускатиме застосунок під час запуску контейнера: node dist/main.js.

Для локальної розкробки та розгортання потрібно встановити Docker локально. Його можна завантажити з офіційного сайту Docker вибравши версію для відповідної операційної системи.

Після створення Dockerfile, яка була описана раніше, для тестування правильності збірки, можна виконати build Docker-образу за допомогою команди docker build. Наприклад, docker build -t my-app:latest. збудує образ використовуючи Dockerfile з поточної директорії. Після цього потрібно запустити контейнер за допомогою команди docker run -p 8000:8000 my-app:latest.

### 3.1.2 Використання Docker Compose

Для забезпечення зручності локальної розробки було вирішено використовувати  Docker і для запуску бази даних. Docker Compose - це інструмент, що дозволяє визначати та керувати багатоконтейнерними Docker-застосунками. Він дозволяє створювати і запускати всі служби

застосунку за допомогою однієї команди. Запустити визначені інтрукції можна за допомогою команди docker-compose up.

Створений файл docker-compose.yml визначає два сервіси: postgres та backend, що використовують спільну мережу data-shape-network (рисунок 3.1.2.1). Сервіс postgres використовує останній образ Docker для PostgreSQL. Він прослуховує порт 5432. Змінні середовища беруться з файлу docker.env. Сервіс backend збирає Dockerfile з поточної директорії і прослуховує порт 3000. Змінні середовища беруться з файлу .env.dev.

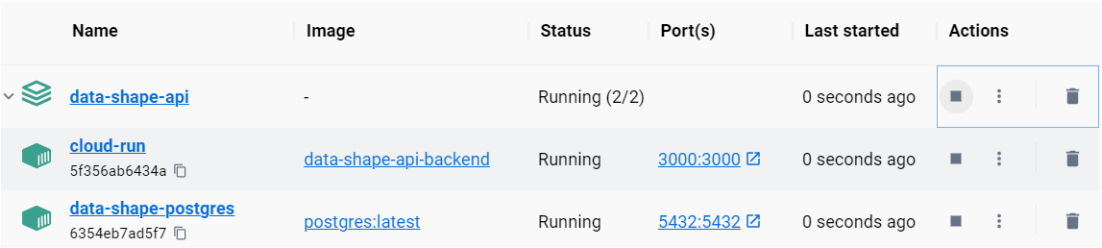


*Рис. 3.1.2.1 Сервіс postgres та backend запущені за допомогою* docker-compose.yml

### 3.1.3 Використання Github Actions

Для автоматизації розгортання було використано Github Actions. Ця система автоматизації вбудована безпосередньо в Github, що дозволяє створювати та запускати робочі процеси відповідно до різних подій у репозиторії (рисунок 3.1.3.1).

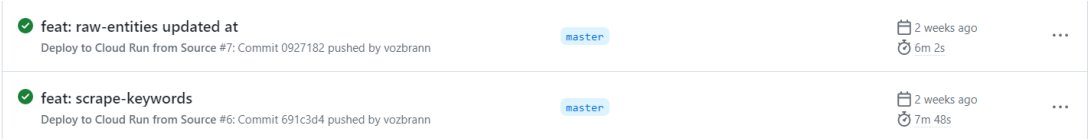


*Рис. 3.1.3.1 Приклад запуску робочих процесів*

В директорії репозиторію Github було створено файл google-cloudrun-source.yml у каталозі .github/workflows/. У файли визначено GitHub Actions

робочий процес, який автоматично розгортає код на Google Cloud Run, при виконанні змін у гілці "master".

Кроки робочого процесу:

- Checkout: отримання вихідного коду репозиторію.
- Google Auth: Використовує GitHub Actions google-github-actions/auth для аутентифікації в Google Cloud з допомогою workload identity federation. Він використовує секретні значення WIF_PROVIDER та WIF_SERVICE_ACCOUNT, які були визначені в налаштуваннях репозиторію GitHub.
- Deploy to Cloud Run: Використовує GitHub Actions google-github-actions/deploy-cloudrun для розгортання застосунку на Google Cloud Run (рисунок 3.1.3.2). Ця дія використовує змінні середовища.

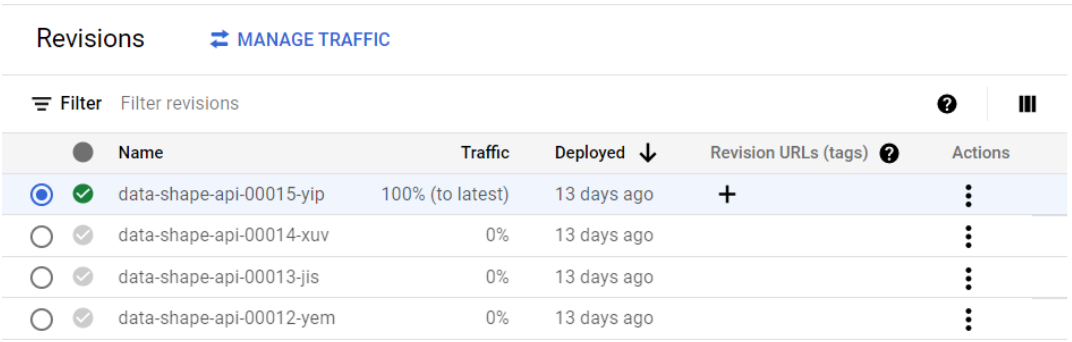


*Рис. 3.1.3.2 Розгортання API на Cloud Run*

### 3.1.4 Розгортання Scheduled function та використання GCP scheduler

Сервіси Cloud Run добре підходять для контейнерів, які працюють безстроково, прослуховуючи HTTP-запити, тоді як завдання Cloud Run краще підходять для контейнерів, які виконуються до кінця і не обслуговують запити (рисунок 3.1.4.1).

Filter Filter history

| | Execution ID | Creation time | Tasks | End time |
|---|---|---|---|---|
| ✓ | data-shape-function-vqgtk | May 21, 2023, 4:32:00 PM | 1/1 completed | May 21, 2023, 4:34:31 PM |
| ✓ | data-shape-function-5gfk4 | May 20, 2023, 4:32:01 PM | 1/1 completed | May 20, 2023, 4:34:32 PM |
| ✓ | data-shape-function-j6kfs | May 19, 2023, 4:32:00 PM | 1/1 completed | May 19, 2023, 4:37:03 PM |
| ✓ | data-shape-function-744lw | May 18, 2023, 4:32:01 PM | 1/1 completed | May 18, 2023, 4:37:26 PM |

*Рис. 3.1.4.1 Розгортання функції на Cloud Run*

Nest.js API та функція для скрепінгу розгортаються на Google Cloud Run. Docker файли та workflows не мають суттєвих відмінностей окрім того, що для розгортання функції використовується команда gcloud run jobs update. Після цього, після успішного розгортання, в інтерфейсі Google Cloud можна визначити тригери, які будуть відповідати за виконання функції за графіком (рисунок 3.1.4.2).

data-shape-function-scheduler-trigger

Schedule
32 14 * * *

Timezone
Etc/UTC

Region
europe-central2

VIEW DETAILS

*Рис. 3.1.4.2 Приклад тригеру, що запускає виконання функції*

### 3.1.5 Cloud SQL and Postgres dumps

В процесі розробки та обслуговування додатку можуть виникнути проблеми у продакшні чи стейджингу, які складно відтворити на локальній машині. Отримання дампу з Google Cloud та його імпортування на локальну машину може сприяти виявленню таких проблем. Це дозволить виконати конкретні тести безпосередньо на локальному рівні.

Для створення дамп-файлу необхідно зайти на платформу Google Cloud і перейти до розділу SQL. В правому верхньому куті слід натиснути кнопку експорту, вибрати тип "sql" та storage bucket для збереження експортованих даних. Після того, як файл буде успішно створено, його необхідно завантажити.

Щоб імпортувати дані в базу даних локального Docker контейнера, слід виконати наступну команду:

"docker exec -i DOCKER_CONTAINER_NAME psql postgresql://USERNAME:PASSWORD@HOST/DBNAME< ./dump.sql".

### 3.1.6 Nest.js Sentry integration

Основна функція Sentry - це відслідковування та ідентифікація помилок. Щоб використовувати цей інструмент, вам потрібно виконати декілька кроків:

- Створення проекту через веб-інтерфейс Sentry, де потрібно налаштувати параметри проекту у відповідності до технологій.
- Налаштування Sentry DSN у файлі .env. Sentry DSN - це унікальний ідентифікатор, який використовується для з’єднання додатку з проектом у Sentry.
- Ініціалізація Sentry, що дозволяє застосунку відправляти відомості про помилки в Sentry.

Для проектів, що написані на Typescript, також важливо використовувати source maps. Вони допомагають легко ідентифікувати код, що став причиною помилки. Для включення source maps вам потрібно:

- Встановити бібліотеку tslib у проект. Це можна зробити за допомогою команди "pnpm install tslib".
- Під час розгортання застосунку в Google Cloud Run, потрібно включити source maps в код застосунку. Це можна зробити за допомогою встановлення пакету source-map-support у проекті.
- В секцію "start" файлу package.json потрібно додати наступний рядок: "-r source-map-support/register". Це дозволить Node.js використовувати source maps під час виконання коду застосунку.

### 3.1.7 Node.js function Sentry integration

Для інтеграції Sentry з TypeScript у Node.js, спочатку потрібно встановити Sentry SDK для Node.js через pnpm. Після встановлення SDK його слід ініціалізувати, вказавши DSN (Data Source Name), який можна отримати після створення проекту на платформі Sentry. DSN вказує куди надсилати повідомлення про помилки.

Після цього, якщо у функції відбудеться необроблене виключення або невідомий відхилений проміс, Sentry автоматично відправить повідомлення про помилку на сервер. Також, Sentry можна використовувати для відслідковування транзакцій, що дозволяє отримувати більше деталей.

## 3.2 CI/CD інтеграція клієнтської частини

### 3.2.1 Розгортання на Vercel

Для автоматизації розгортання клієнтської частини застосунку, було використано Vercel, який використовує Github для доступу до коду і

автоматичного розгортання оновлень. Після реєстрації на сайті Vercel, потрібно підключити репозиторій. Під час цього процесу, Vercel попросить надати йому доступ до репозиторію, щоб він міг отримувати код для розгортання.

Також під час підключення репозиторію, Vercel пропонує налаштувати параметри розгортання. Це може включати вибір гілки для розгортання, вибір команди для збирання проекту та введення змінних середовища. Коли налаштування розгортання завершене, Vercel автоматично розгортатиме застосунок при кожній зміні у вибраній гілці. Також, запустити розгортання можна вручну через інтерфейс Vercel. Після успішного розгортання, Vercel надає URL, за яким можна переглянути розгорнутий додаток (рисунок 3.2.1.1).

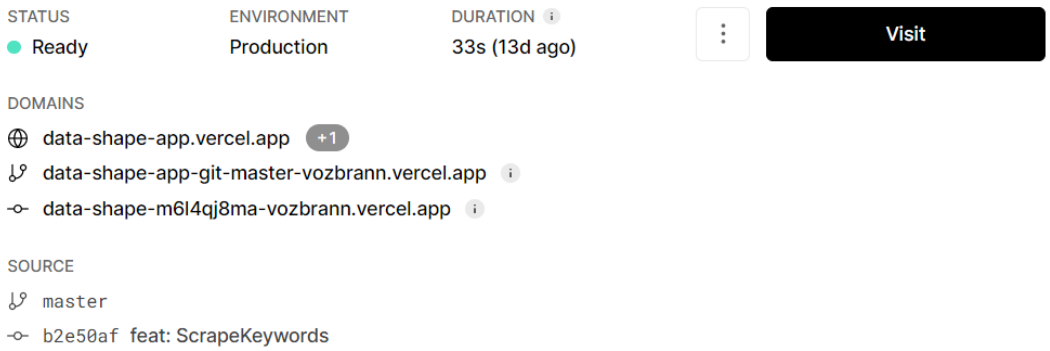


*Рис. 3.2.1.1 Перегляд деплою в інтерфейсі Vercel*

### 3.2.2 Sentry integration

Для інтеграції Sentry з React на Vite і TypeScript необхідно виконати наступні кроки:

1. Встановлення бібліотеки Sentry: pnpm add @sentry/react
2. Конфігурація Sentry. Конфігурація повинна відбуватися якомога раніше в життєвому циклі застосунку. У основному файлі з React потрібно імпортувати бібліотеку та виконати налаштування.

Після цього всі некеровані помилки автоматично будуть фіксуватися Sentry.

3. Налаштування Error Boundary. Компонент Error Boundary також може використовуватися для автоматичної відправки помилок до Sentry.
4. Налаштування React Router. При роботі з react-router-dom, потрібно обгорнути компонент Routes за допомогою Sentry.withSentryReactRouterV6Routing. Це створить компонент вищого порядку, який надасть Sentry доступ до контексту роутера.
5. Налаштування Sentry Vite Plugin. Це плагін Vite, який надає підтримку карти вихідного коду та керування релізами для Sentry.

### 3.2.3 Інтеграція end-to-end тестування

Для налаштування CI в мережі GitHub було обрано бібліотеку Playwright. Інтеграція включає в себе різноманітні кроки для автоматичного запуску при кожному надсиланні нових змін до коду або при створенні запиту на злиття до основних робочих гілок, визначених як "main" або "master" (рисунок 3.2.3.1).


*Рис. 3.2.3.1 Автоматичний запуск тестів*

На початковій стадії, система автоматично здійснює читання коду з репозиторію за допомогою акції actions/checkout@v2. Це дозволяє системі отримати найновішу версію коду для подальшого процесу.

Наступним етапом є отримання ідентифікаційного токену GitHub, що реалізується за допомогою actions/github-script@v5. Це критичний крок, який дозволяє наступним етапам мати доступ до потрібних даних.

Після цього, система встановлює Node.js за допомогою actions/setup-node@v2, що є основою для всієї подальшої роботи. Встановлюється пакетний менеджер pnpm, який використовується для встановлення залежностей проекту. Після цього відбувається встановлення браузерів Playwright.

Google Cloud SDK налаштовується використовуючи google-github-actions/setup-gcloud@v1, що дозволяє системі взаємодіяти з хмарною платформою Google. Вхід до Docker реалізується через docker/login-action@v1, що відкриває можливість використання Docker Buildx через docker/setup-buildx-action@v1.

Використовуючи Docker Compose, виконується завантаження та запуск Docker image серверної частини та бази даних, кроки для запуску яких визначені в docker-compose.yml. Потім, система робить збірку застосунку Vite використовуючи pnpm build та запускається з використанням команди pnpm preview. Після того, як всі попередні етапи виконані, тести Playwright запускаються за допомогою pnpm exec playwright test.

Нарешті, результати тестування завантажуються як артефакт, який зберігається протягом 30 днів, що дає можливість аналізувати результати тестування після завершення процесу CI (рисунок 3.2.3.2).

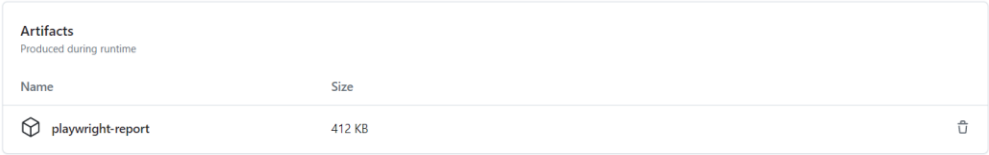


*Рис. 3.2.3.2 Зберігання результатів тестування*

## 3.3 Інтеграція Renovate

Renovate — це інструмент автоматичного оновлення залежностей для проектів програмного забезпечення. Він стежить за файлами залежностей (наприклад, package.json), автоматично виявляє, коли випускається нова версія пакета, який використовується в проекті, та створює запит на злиття або відгалуження з оновленням до цієї нової версії (рисунок 3.3.1).

Renovate Github app можна безкоштовно встановити як для публічних, так і для приватних сховищ. Послуга надається безкоштовно від Mend (раніше відома як WhiteSource). Після підключення, Renovate автоматично відриє запит злиття в всіма необхідними налаштуваннями та можливістю визначення графіку перевірок.

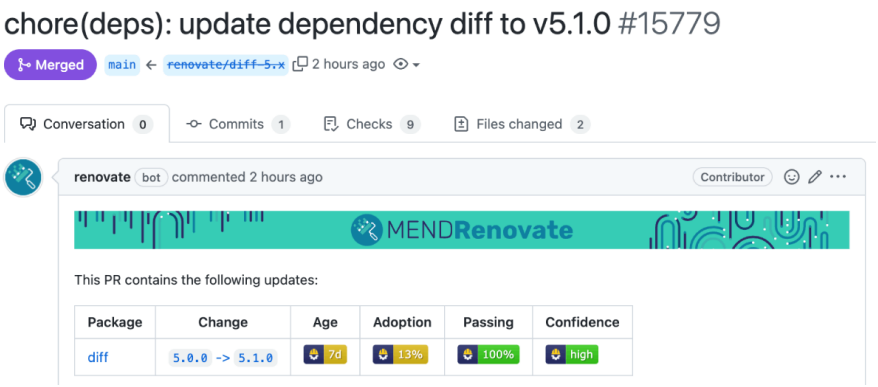


*Рис. 3.3.1 Приклад запиту на злиття від Renovate*

## 3.4 Висновки до розділу 3

У третьому розділі було розглянуто процеси автоматизованого розгортання та інтеграцію CI/CD, які сприяють значному спрощенню розробки. Наголос було зроблено на використанні інструментів, таких як Github, Github Actions, Docker, Google Cloud Run, Vercel, та Sentry для створення гнучкого та ефективного CI/CD пайплайну. В результаті було описано процес налаштування цих інструментів для оптимізації розробки, тестування і розгортання застосунків.

Крім того, було описано використання Playwright для автоматизованого end-to-end тестування. Цей інструмент, який інтегровано в процес CI/CD, дозволяє розробникам перевіряти функціональність всіх частин застосунку разом, що значно підвищує якість кінцевого продукту. Використання системи Docker дозволяє створювати умови, що максимально наближені до реального середовища користувача, що знову ж таки допомагає підвищити якість тестування та надійність застосунку. Використання Google Cloud SDK та Docker в GitHub Actions дає можливість автоматизувати розгортання та виконання тестів, що забезпечує більш швидку і ефективну роботу пайплайну. Загалом, налаштований CI/CD пайплайн з включеним автоматизованим тестуванням є важливим кроком до побудови ефективної системи розробки.

# Висновки

Використовуючи технології Node.js, Nest.js, React, PostgreSQL та TypeORM, було розроблено веб-застосунок з високим рівнем стабільності, безпеки та продуктивності. Дані технології сприяють швидкій розробці, гнучкості, легкій масштабованості та уникненню поширених проблем при розробці серверної та клієнтської частин.

В ході розробки було вивчено ключові етапи розробки серверної та клієнтської частини застосунку, зосереджуючись на глибокому розумінні архітектури застосунку та правильному використанні інструментів та бібліотек для автоматизованого збирання даних, автентифікації та управління базами даних.

Було проведено автоматизацію процесів розгортання та інтеграції CI/CD, використовуючи Github, Github Actions, Docker, Google Cloud Run, Vercel та Sentry. Ці інструменти допомогли оптимізувати процеси розробки, тестування та розгортання застосунку, підвищуючи продуктивність, швидкість розробки та надійність кінцевого продукту.

Впровадження системи автоматизованого тестування Playwright в CI/CD процес дозволило спростити розробку, перевірку та розгортання застосунку, а застосування технології Docker для створення середовища забезпечує стабільність та надійність.

Таким чином, на основі використаних технологій та глибокого технічного розуміння процесу розробки, було розроблено надійний, високопродуктивний та легко масштабований веб-застосунок.

## Список використаної літератури та електронних ресурсів


1. Pnpm documentation. [Online]. Available: https://pnpm.io/motivation
2. Yarn documentation. [Online]. Available: https://classic.yarnpkg.com/en/
3. TypeScript documentation. [Online]. Available: https://www.typescriptlang.org/docs/
4. Docker documentation. [Online]. Available: https://docs.docker.com/
5. Npm trends. Compare package download counts over time. [Online]. Available: https://npmtrends.com/
6. Webpack documentation. [Online]. Available: https://webpack.js.org/concepts/
7. Vite documentation. [Online]. Available: https://vitejs.dev/guide/
8. Tanstack query React documentation. [Online]. Available: https://tanstack.com/query/latest/docs/react/overview
9. Tkdodo blog. Practical React Query. [Online]. Available: https://tkdodo.eu/blog/practical-react-query
10. Nestjs documentation. [Online]. Available: https://docs.nestjs.com/
11. Playwright documentation. [Online]. Available: https://playwright.dev/docs
12. Google cloud documentation. Cloud Run for App Engine customers. [Online]. Available: https://cloud.google.com/appengine/docs/standard/cloud-run-for-gae-customers
13. Vercel documentation. [Online]. Available: https://vercel.com/docs