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

Побудова багаторівневого веб-застосування з високою доступністю на платформі Google Cloud Platform (GCP)

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

Керівник курсової роботи
Черкасов Д.І.
_____
(підпис)
“____” _____ 2022 р.

Виконав студент
Возбранний Р.С.
“____” _____ 2022 р.

Київ 2022

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

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

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу
студенту Возбранному Роману
факультету інформатики 1 курсу магістерської програми

ТЕМА: Побудова багаторівневого веб-застосування з високою доступністю на платформі Google Cloud Platform (GCP)

Вихідні дані:

Інструкції та готові рішення для проблем розробки, розгортання та підтримки застосунків.

Зміст ТЧ до курсової роботи:

Вступ

Анотація

1. Огляд бібліотек та технологій для побудови застосунку

2. Структурна розробка

3. Автоматизоване розгортання, CI/CD інтеграція

Висновки

Список використаної літератури та електронних ресурсів

Додатки

Дата видачі “_____” _____201_ р.

Керівник _____ Завдання отримано _____

Тема: Побудова багаторівневого веб-застосування з високою доступністю на платформі Google Cloud Platform (GCP)

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 6

Вступ 7

Розділ 1. Огляд бібліотек та технологій для побудови застосунку 8

1.1 Інтерфейс користувача 8

1.1.1 Бібліотека React.js 8

1.1.2 Бібліотека React Query 8

1.1.3 React Router 9

1.1.4 Axios 10

1.2 Бізнес логіка 10

1.2.1 Node.js 10

1.2.2 Express 11

1.2.3 Docker 12

1.3 Робота з базою даних 13

1.3.1 PostgreSQL 13

1.3.2 Mikro ORM 13

1.4 Висновки до розділу 1 15

Розділ 2. Структурна розробка 16

2.1 Frontend 16

2.1.1 Бібліотека Axios 16

2.1.2 Бібліотека React Query 17

2.1.3 Бібліотека React Router 20

2.1.4 Обробка query параметрів клієнстького застосунку (useQueryParams) 20

2.2 Backend 20

2.2.1 Структура Express REST API 21
2.2.2 Mikro orm 22
2.2.3 Аутентифікація та авторизація 24
2.3 Висновки до розділу 2 26
Розділ 3. Автоматизоване розгортання, CI/CD інтеграція 27
3.1 Підготовка GitHub workspace 27
3.2 Build застосунку 28
3.3 Github Actions 28
3.3.1 Dependabot 28
3.4 CD – розгортання в середовищі 29
3.5 CI/CD Google Cloud Platform 30
3.5.1 Аутентифікація 30
3.5.2 Deployment 30
3.6 Backend CI / CD 32
3.6.1 Sentry monitoring 32
3.6.2 Cloud SQL and Postgres dumps 36
3.7 Frontend CI / CD 37
3.7.1 Sentry monitoring 37
3.7.2 Cypress 38
3.3 Висновки до розділу 3 39
Висновки 40
Список використаної літератури та електронних ресурсів 41

Анотація

В роботі проаналізовано процес розробки, розгортання та підтримки багаторівневих застосунків. Також, описуються особливості налаштування проектів для розгортання в Google Cloud Platform. В результаті, розроблено інструкції та проаналізовано готові рішення для розробки, розгортання та підтримки застосунків.

Ключові слова:

Розробка, система, розгортання, Google Cloud Platform, CI, CD.

Вступ

З розвитком технологій, хмарні обчислювальні сервіси стали ефективним інструментом, який дозволяє розробникам не турбуватися про налаштування інфраструктури та обслуговування, а зосередитися на найголовнішому – розробці. Оплата в міру використання, автоматичне масштабування та надання за запитом, – значно полегшують процес розробки застосунків.

В роботі проводиться аналіз та опис основних аспектів розробки, розгортання, підтримки та засобів моніторингу багаторівневого веб-застосування на хмарній платформі.

Об’єкт дослідження: розгортання багаторівневого веб-застосування на хмарній платформі Google Cloud Platform.

Мета дослідження: визначити та описати особливості розробки, налаштування проекту для подальшого автоматизованого розгортання застосунку на платформі Google Cloud Platform.

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 автоматизоване розгортання застосунку та CI/CD інтеграція.

Розділ 1. Огляд бібліотек та технологій для побудови застосунку

1.1 Інтерфейс користувача

1.1.1 Бібліотека React.js

ReactJS - це бібліотека JavaScript, що використовується для створення інтерфейсів користувача та компонентів. Бібліотека дозволяє динамічно відображати дані без перезавантаження сторінки в великих односторінкових застосунках [1].

Особливості бібліотеки:

- JSX - це розширення синтаксису JavaScript, яке рекомендується використовувати при розробці React застосунків;
- Віртуальний DOM - забезпечити високу швидкість роботи застосунку, оскільки бібліотека самостійно вирішує які компоненти сторінки треба оновити.

1.1.2 Бібліотека React Query

React.js - це потужна JavaScript бібліотека для створення клієнтських застосунків. Як і будь-яка інша бібліотека JavaScript, React.js забезпечує безперебійну роботу при створенні реактивних і декларативних інтерфейсів користувача. Однак у React є кілька негативних аспектів, одним з яких є керування станом і отримання даних. React Query - це попередньо налаштована бібліотека, яка спрямована на вирішення цих складнощів [7].

React Query дозволяє отримувати, кешувати й оновлювати дані в програмах на базі React у простий і декларативний спосіб, не змінюючи глобальний стан.

Переваги використання React-Query в React.js застосунках:

- Використання window focus pre-fetching механізму для завантаження та оновлення даних зважаючи на активність вкладки застосунку.
- Можливість встановити кількість повторних спроб запиту для будь-якого запиту у разі випадкових помилок.
- React-Query на задньому плані виконує автоматичне оновлення застарілих даних.
- Обробка кешування складних програм, для оптимізації операцій запиту даних.

1.1.3 React Router

Для веб-програми маршрутизація - це механізм відтворення всієї або часткової сторінки на основі наданої URL-адреси, параметрів або взаємодії користувача з інтерактивними елементами сторінки.

Бібліотека React не включає маршрутизацію. React Router сумісний з останніми версіями React і є найпопулярнішим інструментом маршрутизації для React застосунків. React Router забезпечує навігацію між представленнями різних компонентів. Він зчитує і керує URL-адресою браузера, а також підтримує синхронізацію інтерфейсу користувача з URL-адресою.

Основні елементи навігації [6]:

- URL (Uniform Resource Locator) - це адреса даного унікального ресурсу в Інтернеті. У браузері є адресний рядок, у який користувачі можуть вводити певну URL-адресу. Також URL може бути контрольований програмою.
- Location - об’єкт, який відповідає за представлення URL location, в основі якого лежить об’єкт [removed] браузера.
- History - відстежує навігацію по кожній вкладці.

1.1.4 Axios

Виконання HTTP-запитів для отримання та збереження даних є звичайним завданням для будь-якого клієнтського JavaScript застосунку. Axios - це JavaScript бібліотека, яка використовується для виконання HTTP-запитів, яка працює як у браузері, так і на платформах Node.js.

Основні особливості Axios:

- XMLHttpRequests запити з браузера;
- http запити з node.js;
- підтримка Promise API;
- перехоплення запиту та відповіді;
- трансформація даних запиту та відповіді;
- скасування запитів;
- автоматичне перетворення для JSON даних;
- підтримка на стороні клієнта запобігання XSRF.

1.2 Бізнес логіка

1.2.1 Node.js

Node.js - це кросплатформне середовище виконання JavaScript з відкритим вихідним кодом. Це популярний інструмент практично для будь-яких проектів. Node.js запускає V8 JavaScript engine, ядро Google Chrome, поза браузером. Це дозволяє Node.js бути дуже продуктивним [2].

Коли Node.js виконує операцію введення-виведення, наприклад, читання з мережі, доступ до бази даних або файлової системи, замість того, щоб блокувати потік і витрачати цикли ЦП на очікування, Node.js відновлює операції після отримання відповіді. Це дозволяє Node.js обробляти тисячі одночасних підключень з одним сервером, без необхідності використання паралельних потоків, що може бути значним джерелом помилок.

З точки зору розробки веб-сервера, Node має ряд переваг [2]:

- Висока продуктивність. Node був розроблений для оптимізації пропускної здатності та масштабованості у веб-додатках і є хорошим рішенням для багатьох поширених проблем веб-розробки (наприклад, real-time web applications).
- Код написаний на "звичайному старому JavaScript", що означає, що менше часу витрачається на "зміщення контексту" між мовами, при написанні клієнтського і серверного коду.
- Менеджер пакетів (NPM) надає доступ до сотень тисяч повторно використовуваних пакетів. Він також має найкращу в своєму класі dependency resolution, а також може використовуватися для автоматизації більшості інструментів для створення.
- Node.js є портативним. Він доступний в Microsoft Windows, macOS, Linux, Solaris, FreeBSD, OpenBSD, WebOS і NonStop OS. Крім того, він добре підтримується багатьма провайдерами веб-хостингу, які часто надають специфічну інфраструктуру та документацію для розміщення сайтів Node.
- Активна спільнота розробників.

1.2.2 Express

Express - найпопулярніший веб-фреймворк Node і є базовою бібліотекою для ряду інших популярних веб-фреймворків Node. Він забезпечує механізми для [3]:

- Написання обробників запитів з різними на різних URL-шляхах.
- Інтеграції з механізмами візуалізації "view", щоб генерувати відповіді, вставляючи дані в шаблони.

- Встановлення загальних параметрів веб-програми, таких як порт для підключення і розташування шаблонів, які використовуються для рендеру відповіді.
- Використання додаткових "middleware" для обробки запитів у будь-якій точці процесу обробки.

1.2.3 Docker

Docker - це відкрита платформа для розробки, доставляння та запуску програм. Docker дозволяє відокремити програму від інфраструктури. Контейнери дозволяють відокремлювати програми від інфраструктури. Це дозволяє значно пришвидшити процес розробки.

Docker надає можливість пакувати та запускати програму в слабо ізольованому середовищі, яке називається контейнером. Ізоляція та безпека дозволяють запускати багато контейнерів одночасно на даному хості. Контейнери легкі та містять все необхідне для запуску програми, тому розробнику не потрібно хвилюватися про те, що наразі встановлено на хості. Це дозволяє ділитися контейнерами та бути впевненим що контейнер буде працювати так само на інших машинах.

Docker надає інструменти та платформу для керування життєвим циклом контейнерів:

- Розробка застосунків та допоміжних компонентів за допомогою контейнерів.
- Контейнер - блок для розповсюдження та тестування застосунку.

Docker спрощує життєвий цикл розробки, дозволяючи розробникам працювати в стандартизованих середовищах, використовуючи локальні контейнери. Контейнери чудово підходять для безперервної інтеграції та CI/CD процесів.

1.3 Робота з базою даних

Під час розробки програми першим кроком після створення дизайну MVP є створення моделі даних, яка пізніше буде перекладена у базу даних.

1.3.1 PostgreSQL

PostgreSQL - це гнучка реляційна база даних з відкритим вихідним кодом, здатна впоратися з різними випадками використання, від окремих машин і сховищ даних до веб-сервісів з багатьма одночасними користувачами. PostgreSQL використовує та розширює SQL.

Головні переваги Postgres:

- чудова інтеграція з ORM;
- багато типів даних;
- хороша підтримка JSON;
- хороша продуктивність;
- легко масштабується.

Нижче наведено недоліки та обмеження PostgreSQL:

- Зміни, внесені для підвищення швидкості, вимагають більше роботи, ніж при використанні MySQL, оскільки PostgreSQL зосереджується на сумісності;
- Багато програм з відкритим кодом підтримують MySQL, але можуть не підтримувати PostgreSQL;
- За показниками продуктивності він повільніше, ніж MySQL.

1.3.2 Mikro ORM

Не зважаючи на те, що важливо розуміти внутрішню роботу сервера баз даних і механізмів запитів, розробник рідко стикається з ними. Замість цього

програми створюються за допомогою ORM, рівня абстракції поверх інтерфейсу бази даних.

ORM - це об’єктно-реляційний проєкція, який працює як середовище виконання запитів і створення об’єктів між клієнтським кодом і реляційною базою даних. ORM надають безліч потужних функцій, таких як кешування об’єктів і запитів, контроль паралельності, об’єктно-орієнтовані мови запитів і багато іншого.

MikroORM - чудовий ORM для Node.js. Він має надійне ядро, але відносно новий, тому не всі функції є надійними.

Головні переваги Mikro Orm:

- Має простий спосіб визначення моделі на основі декораторів (рисунок 1.3.2.1);
- Інтуїтивно зрозумілий спосіб написання запитів;
- Висока продуктивність завдяки розумній поведінці поділу запитів;
- Зручна генерація міграцій.

./entities/Book.ts

```
@Entity()
export class Book extends CustomBaseEntity {

  @Property()
  title!: string;

  @ManyToOne(() => Author)
  author!: Author;

  @ManyToOne(() => Publisher, { wrappedReference: true, nullable: true })
  publisher?: IdentifiedReference<Publisher>;

  @ManyToMany({ entity: 'BookTag', fixedOrder: true })
  tags = new Collection<BookTag>(this);

}
```

Рис. 1.3.2.1 Визначення моделі в Mikro Orm

1.4 Висновки до розділу 1

Для аналізу розробки серверної частини (Rest API), було обрано платформу Node.js та фреймворк Express. Роботу з базою PostgresSQL забезпечує Mikro ORM, яка легко інтегрується в TypeScript проекти. В основі клієнтського застосунку лежить бібліотека React. За управління запитами та кешування даних відповідає бібліотека React Query.

Розглянуті технології та інструменти дозволяють створювати надійні, легкі в підтримці та легко масштабовані веб застосунки.

Розділ 2. Структурна розробка

2.1 Frontend

2.1.1 Бібліотека Axios

Для підключення бібліотеки Axios до проекту необхідно виконати команду “yarn add axios”. Створення Axios instance дозволяє виконати метод “axios.create(config)” (рисунок 2.1.1.1), який приймає об’єкт з налаштуваннями. Одним з основних налаштувань є baseURL для всіх запитів.

```
import axios from 'axios';
import setupAxiosInterceptorsTo from './setupAxiosInterceptorsTo';

const api = setupAxiosInterceptorsTo(
  axios.create({
    baseURL: process.env.API_BASE_URL,
  }),
);
export default api;
```

Рис. 2.1.1.1 Створення Axios instance

Проте, головним в налаштуванні роботи Axios в React застосунку є визначення інтерцепторів. Для цього була використана власна функція “setupAxiosInterceptorsTo” (рисунок 2.1.1.2), яка застосовує інтерцептори onRequest, onResponse та onResponseError до інстансу axios:

- onRequest - відповідає за додавання JWT токену до хедерів запиту, а також здійснює перевірку його валідності. У разі закінчення строку дії токена, відбувається автоматичне оновлення за допомогою refresh токена.
- onResponse - при отриманні відповіді на запит про отримання токенів, виконує їх зберігання в cookies або localStorage.

- onResponseError - перевіряє статус помилки, та видаляє збережені токени у разі їх не валідності.

```
function setupAxiosInterceptorsTo(axiosInstance: AxiosInstance): AxiosInstance {
  axiosInstance.interceptors.request.use(onRequest);
  axiosInstance.interceptors.response.use(onResponse, onResponseError);
  return axiosInstance;
}
```

Рис. 2.1.1.2 Функція «setupAxiosInterceptorsTo»

2.1.2 Бібліотека React Query

Для забезпечення роботи бібліотеки React Query в React застосунку, необхідно встановити саму бібліотеку react-query командою “yarn add react-query”. React Query сумісна з React v16.8+ і працює з ReactDOM. Щоб запустити React-Query, використовується конфігурацію у файлі кореневого каталогу index.js.

QueryClient приймає параметри, для налаштування повторного надсилання запитів, та способів обробки помилок. В прикладі демонтрується опція використання ErrorBoundary тільки при визначених помилках в queries хуках (рисунок 2.1.2.1). Весь застосунок необхідно обгорнути в QueryClientProvider, в який передається створений раніше queryClient.

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

Рис. 2.1.2.1 Створення QueryClient instance

Для організації файлів для роботи з React-Query використовується дві директорії: api та hooks. Директорія api містить axios запити (рисунок 2.1.2.2), які пізніше можуть використовуватися як в React-Query hooks, так і в інших частинах застосунку. Часто вони також використовуються для отримання даних для селектів з можливістю пошуку [8].

```
import api from 'data/axiosApi';
import { User, EditUserValues } from 'data/utils/types';

export const getUser = (id: string): Promise<User> => api.get(`/users/${id}`);

export const editUser = (id: string, values: EditUserValues): Promise<unknown> =>
  api.patch(`/users/${id}`, values);

export const deleteUser = (id: string): Promise<unknown> => api.delete(`/users/${id}`);
```

Рис. 2.1.2.2 Axios запити

Ключі запиту є важливою концепцією в React Query. Вони необхідні для того, щоб бібліотека могла внутрішньо кешувати дані, правильно та автоматично витягувати їх, коли залежність від запиту змінюється. Також, це дозволяє взаємодіяти з кешем запитів вручну, коли це необхідно, наприклад, під час оновлення даних після мутації або коли потрібно вручну скасувати деякі запити.

Для зручної організації ключів рекомендується створювати Query Key фабрику для кожної функції. Фабрика Query Key (рисунок 2.1.2.3) це простий об’єкт ключами та функціями, які генерують query ключі [8].

```ts
export const usersKeys = {
  all: () => ['user'] as const,
  lists: () => [...usersKeys.all(), 'list'] as const,
  current: () => [...usersKeys.all(), 'current'] as const,
  list: (filters: string) => [...usersKeys.lists(), { filters }] as const,
  details: () => [...usersKeys.all(), 'detail'] as const,
  detail: (id: string) => [...usersKeys.details(), id] as const,
};
```

Рис. 2.1.2.3 Фабрика Query Key

При написанні хуків React Query, у файл імпортуються попередньо створені ключі, та api запити. React Query надає хук useQuery для отримання та контролю стану даних. На відміну від хука useQuery, мутації (рисунок 2.1.2.4) використовуються для операцій CREATE, UPDATE або DELETE як операцій на стороні сервера.

```ts
export const useDepartments = (teamId = '', searchParams: CommonSearchParams) => {
  const queryString = stringify(searchParams);
  return useQuery<Departments, Error>(departmentsKeys.list(teamId, queryString), () =>
    api.fetchDepartments(teamId, queryString),
  );
};

export const useCreateDepartment = (): UseMutationResult<
  DepartmentEntity,
  IError,
  { teamId: string; values: EditDepartment },
  unknown
> => {
  const queryClient = useQueryClient();
  return useMutation(({ teamId, values }) => api.createDepartment(teamId, values), {
    onSuccess: () => {
      queryClient.invalidateQueries(departmentsKeys.all());
    },
  });
};
```

Рис. 2.1.2.4 Хуки useQuery та useMutation

2.1.3 Бібліотека React Router

Для встановлення бібліотеки React Router, необхідно виконати команду "yarn add react-router-dom@6”. Після створення проекту і встановлення React Router як залежності, у файлі src/index.js потрібно імпортувати BrowserRouter з react-router-dom та обгорнути застосунок в <BrowserRouter>. Після цього React Router стане доступним у всьому застосунку.

2.1.4 Обробка query параметрів клієнстького застосунку (useQueryParams)

Створюючи додаток з URL-адресами, якими можна легко поділитися, часто потрібно закодувати стан як параметри запиту, але всі параметри запиту мають бути закодовані як рядки. useQueryParams дозволяє легко кодувати та декодувати дані будь-якого типу як параметри запиту за допомогою розумного запам'ятовування, щоб запобігти створенню непотрібних повторюваних об'єктів. Він використовує serialize-query-params. UseQueryParams Працює з React Router і Reach Router і підтримує TypeScript.

2.2 Backend

Вимоги до архітектури:

- В основі серверної частини лежить REST архітектура.
- Для управління даними використовується 4 базові функції CRUD (create, read, update, delete) «створення, зчитування, зміна і видалення».
- Для авторизації використовується JWT (JSON Web Token).

2.2.1 Структура Express REST API

Багаторівнева архітектура - шаблон, який використовується в розробці програмного забезпечення, де ролі та обов'язки в програмі (додатку) розділені на рівні.

Для організації структури Express REST API застосунку було розглянуто розподіл на три рівні (рисунок 2.2.1.1): API, Service та Integration. Головна ідея цього підходу полягає у винесенні бізнес-логіки з API раутів [4][5].

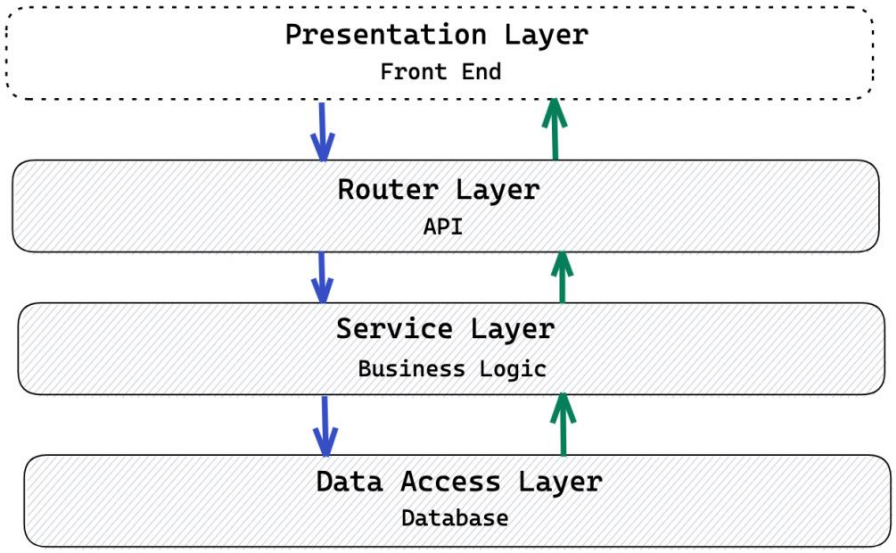

Рис. 2.2.1.1 Трирівнева архітектура

Опис рівнів:

- API layer - містить API раути програми. Цей рівень відповідає за отримання HTTP запиту та аналіз його даних. Після аналізу даних, вони передаються на Service layer. Перевірка структури отриманих даних відбувається на цьому кроці. Також, на рівень API відповідає за трансформацію результат виконання запиту з Service layer в HTTPстатус код та JSON відповідь. Express.js використовується тільки на цьому рівні.

- Service Layer - обробляє бізнес-логіку програми. Він не залежить від будь-яких конструкцій, специфічних для HTTP, і виклик може бути здійснений з будь-якої частини програми. Вхідними є звичайні об’єкти JavaScript. Сервіси виконують бізнес-логіку. Вони перевіряють введені дані на відповідність бізнес-правилам і викликають інші служби на рівні сервісу або звертаються до Integration layer.
- Integration layer - код на рівні інтеграції відповідає за виконання вводу-виводу поза межами процесу. Він спілкується з базами даних і робить HTTP-запити до сторонніх веб-API.

Така архітектура легко переноситься в структуру проекту. Для кожного рівня створюється окремий каталог (рисунок 2.2.1.2) [5].

Рис. 2.2.1.2 Структура проекту серверної частини

2.2.2 Mikro orm

Перед початком налаштування необхідно встановити mikro-orm в проект виконавши команду “yarn add @mikro-orm/core @mikro-orm/postgresql”. Після цього для правильної роботи в typescript проекті,

потрібно увімкнути підтримку декораторів та esModuleInterop в файлі tsconfig.json (рисунок 2.2.2.1).

```
"experimentalDecorators": true,
"emitDecoratorMetadata": true,
"esModuleInterop": true
```

Рис. 2.2.2.1 Налаштування mikro-orm

Для доступу до специфічних методів драйвера postgresql, потрібно вказати тип драйвера при виклику функції “MikroORM.init()” (рисунок 2.2.2.2).

```
import type { PostgreSqlDriver } from '@mikro-orm/postgresql'; // or any other driver package

const orm = await MikroORM.init<PostgreSqlDriver>({
  entities: ['./dist/entities'], // path to your JS entities (dist), relative to `baseDir`
  dbName: 'my-db-name',
  type: 'postgresql',
});
console.log(orm.em); // access EntityManager via `em` property
```

Рис. 2.2.2.2 Виклик функції MikroORM.init()

Також, важливим кроком є використання entity manager для кожного запиту окремо. Для цього використовується express middleware. Це необхідно для запобігання колізії identity maps (рисунок 2.2.2.3).

```
const app = express();

app.use((req, res, next) => {
  RequestContext.create(orm.em, next);
});
```

Рис. 2.2.2.3 Запобігання колізії identity maps

Після виконання всіх попередніх кроків, у відповідних каталогах потрібно створити файли з визначенням сутностей (рисунок 2.2.2.4).

./entities/Item.ts

```
@Entity()
export class Item {
  @PrimaryKey()
  id: number;

  @Property()
  title: string;

}
```

Рис. 2.2.2.4 Приклад визначення сутностей

2.2.3 Аутентифікація та авторизація

Для забезпечення безпеки застосунку використовуються токени доступу. Детальніше розглянуті саме JWT (JSON Web Token). JWT - це спосіб шифрування даних json в токен за допомогою секрету або пари ключів. Ці дані будуть використані для аутентифікації та авторизації запиту.

Спочатку відбувається перевірка та декодування JWT з секретним підписом. Якщо це не вдається, токен вважається недійсним і запит відхиляється. Якщо JWT перевірено та успішно декодовано, відбувається перевірка отриманих з токена даних:

- хто є користувачем;
- яка роль користувача та яку інформацію він запитує;
- в якому контексті відбувається запит.

Після отримання відповідей на ці питання, починається процес авторизації, який перевіряє, чи має даний користувач до запитуваних ресурсів.

Додатково до токенів доступу створюються refresh токени, які зберігаються в базі даних. Термін дії JWT токенів зазвичай закінчується досить швидко, залежно від налаштувань.

Щоб не змушувати користувача входити в систему щоразу, коли закінчується термін дії токена, клієнтська частина застосунку може використати refresh токен, щоб отримати новий токен доступу. Термін дії refresh токенів зазвичай набагато довший і його також можна скасувати, видаливши з бази даних. Refresh токен також можна використовувати для отримання інших типів токенів доступу, який дозволить користувачеві виконувати різні дії в API.

2.2.3.1 Role-based system

У базі даних поруч із таблицею User зберігається відношення oneToMany до таблиці Role. Ці ролі визначають доступ до певних ресурсів. Найпростішою роллю може бути щось загальне, наприклад адміністратор або користувач. Сутність Role також може бути розширена додатковими даними. Наприклад, до ролі можна приєднати команду, щоб визначити роль користувача в цій команді.

2.2.3.2 Перевірка вимог доступу до роута

Одним з рішень для реалізації аутентифікації в express застосунках є бібліотека “routing-controllers”. Для визначення обмежень безпеки для раута використовується декоратор “@Authorized” (рисунок 2.2.3.2.1). В нього також можна передати додаткові вимоги безпеки. Щоб авторизований декоратор зрозумів, як обробляти ці вимоги, потрібно налаштувати “authorizationChecker” у параметрах контролерів маршрутизації “useExpressServer”.

```
@Authorized(aRequirement)
@Authorized([aRequirement, aSecondRequirement])
@Authorized([[aRequirement, aSecondRequirement]])
@Authorized([[aRequirement, aSecondRequirement], aThirdRequirement])
@Authorized([[aRequirement, aSecondRequirement], [aThirdRequirement, aFourthRequirement]])
```

Рис. 2.2.3.2.1 Приклад визначення сутностей

2.3 Висновки до розділу 2

В процесі виконання роботи було розглянуто процес створення серверного та клієнтського застосунків. Описано важливі процеси інтеграції бібліотек та інструментів інфраструктури бібліотеки React. Визначено структуру серверної частини застосунку, процес підключення Mikro ORM та описано важливі етапи розробки системи аутентифікації і авторизації.

Розділ 3. Автоматизоване розгортання, CI/CD інтеграція

Усі процеси мають бути автоматизовані та налаштовані за допомогою дій Github. Щоб розгорнути програму, потрібно виконати кілька кроків:

- підготовка GitHub workspace;
- build застосунку;
- deploy застосунку.

3.1 Підготовка GitHub workspace

Кожен workflow створює робочу область, в якій будуть виконуватися всі дії. Щоб отримати доступ до свого сховища, потрібно змінити робочий каталог за замовчуванням. Для цього потрібно використати GitHub checkout action (рисунок 3.1.1).

```
- name: Checkout
  uses: actions/checkout@v3
```

Рис. 3.1.1 Використання GitHub checkout action

Крім того, щоб використовувати команди проекту, потрібно інсталювати Node у цю робочу область (setup-node action) (рисунок 3.1.2).

```
- name: Use Node.js v16
  uses: actions/setup-node@v3
  with:
    node-version: 16.x
```

Рис. 3.1.2 Інсталяція Node

3.2 Build застосунку

Першим кроком білда застосунку є встановлення залежностей. bahmutov/npm-instal - GitHub Action для встановлення залежностей npm з кешуванням без будь-якої конфігурації (рисунок 3.2.1).

```
- name: Install dependencies
  uses: bahmutov/npm-install@v1
```

Рис. 3.2.1 Встановлення залежностей

Також можливе використання власних, попередньо написаних кроків, не використовуючи заздалегідь готові actions (рисунок 3.2.2).

```
- name: Build app
  run: yarn run build:production //write needed script here
```

Рис. 3.2.2 Використання власних кроків

3.3 Github Actions

Файли CI/CD Github action workflow повинні розміщуватись в каталозі .github/workflows. На початку файлу визначаються тригери, де зазвичай вказуються гілки, на яких робочий процес запускатиметься автоматично. Також слід включити тригер «workflow_dispatch», щоб вручну запустити робочий процес з інтерфейсу github.

3.3.1 Dependabot

Dependabot допомагає підтримувати залежності в актуальному стані. Щодня він перевіряє файли залежностей проекту на наявність застарілих і відкриває окремі PR для будь-яких знайдених.

Події, ініційовані Dependabot, не матимуть доступу до secrets. Це може призвести до невдачі PR від залежного бота. У налаштуваннях репозиторію передбачено розділ для налаштування секретів для dependabot (рисунок 3.3.1.1).

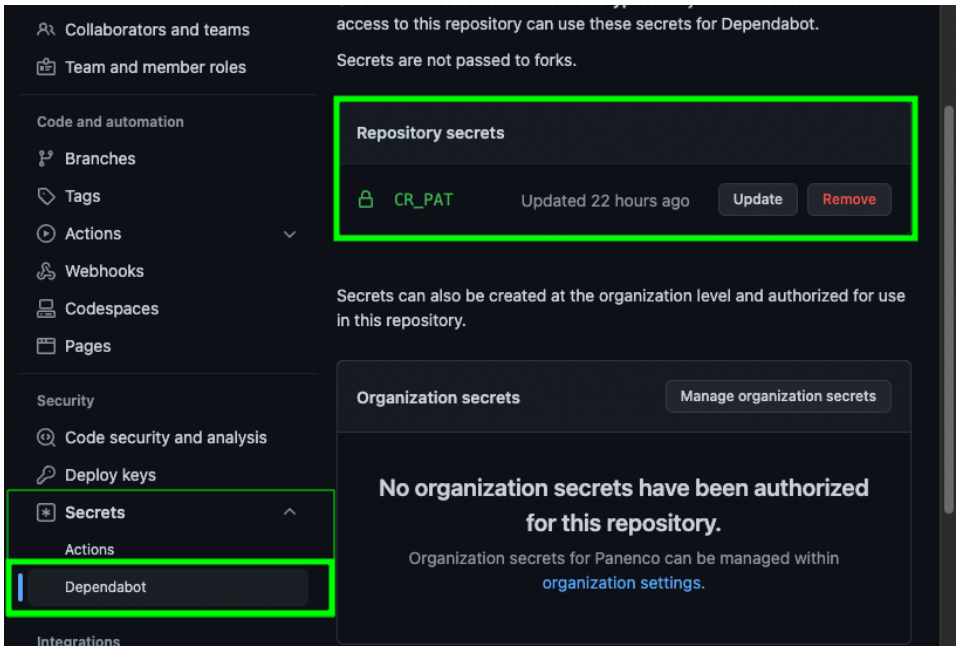

Рис. 3.3.1.1 Налаштування секретів для dependabot

3.4 CD – розгортання в середовищі

Після автоматизації білду та тестування в CI, continuous delivery автоматизує релізи валідованого коду. Розгортання в середовищі завжди має бути автоматизованим процесом. Кроки для розгортання:

- Merge to staging для деплою staging.
- Merge to main для деплою production.
- Workflow запускається автоматично.
- Build застосунку.
- Деплой до хмарного провайдера.
- Success/Error сповіщення.

3.5 CI/CD Google Cloud Platform

Google Cloud використовується для розміщення програм через служби App Engine. App Engine - це керована, безсерверна платформа для розробки та розміщення веб-додатків. App Engine самостійно керує наданням серверів і масштабуванням застосунків відповідно до необхідності.

Стандарте середовище Google App Engine має такі особливості:

- Постійне сховище із запитами, сортуванням і транзакціями.
- Автоматичне масштабування та балансування навантаження.
- Асинхронні черги завдань для виконання роботи за межами запиту.
- Заплановані завдання для ініціювання подій через регулярні або визначені часові проміжки.
- Інтеграція з іншими хмарними сервісами та API Google.

3.5.1 Аутентифікація

Actions отримують доступ до проекту GCP через service account. Service account credentials повинні зберігатися в Github secrets.

3.5.2 Deployment

Google надає попередньо створений github action, який дозволяє легко розгортати застосунок на їхніх серверах: google-github-actions/setup-gcloud. Ця утиліта очікує два параметри: service_account_email та service_account_key. Також потрібно надати deployment файл з деталями сервісу (рисунок 3.5.2.1) [9][10].

Розгортання відбуваються до визначеного проекту в gcp. Для цього необхідно налаштувати ідентифікатор проекту в Github secrets.

Ідентифікатор проекту можна знайти в консолі gcp: gcp console > select project > dashboard > Project info > Project ID.

```yaml
- name: Google cloud auth
  uses: google-github-actions/setup-gcloud@v0
  with:
    service_account_email: ${{ secrets.GCP_SA_EMAIL }}
    service_account_key: ${{ secrets.GCP_SA_KEY }}
- name: Deploy
  run: gcloud --project ${{ secrets.PROJECT_ID }} app deploy staging.yml
```

Рис. 3.5.2.1 Використання google-github-actions/setup-gcloud

Щоб уникнути перевищення квоти, версії потрібно регулярно видаляти. Коли квота перевищена, розгортання починає давати збій. Цей action слід інтегрувати, щоб спростити й уніфікувати спосіб розгортання.

Для роботи action необхідні наступні дані:

- файл credentials_json;
- service_id - обов’язковий ідентифікатор app engine service;
- project_id - обов’язковий ідентифікатор app engine service;
- app_yaml_path - обов’язковий шлях до файлу GCP app.yaml;
- extra_deploy_args - Додаткові аргументи для використання під час розгортання програми.

Action виконує два кроки (рисунок 3.5.2.2):

- Видалення усіх app engine service versions, крім останніх 6;
- Розгорнути нову версію.

```yaml
on: [push]
jobs:
  deploy_job:
    runs-on: ubuntu-latest
    name: deploy
    steps:
      - name: Deploy to app engine
        uses: Panenco/gcp-deploy-action@v1
        with:
          credentials_json: "{{ secrets.GCP_CREDENTIALS_JSON }}"
          service_id: "{{ secrets.GCP_SERVICE_ID }}"
          project_id: "{{ secrets.GCP_PROJECT_ID }}"
          app_yaml_path: "staging.yml"
```

Рис. 3.5.2.2 Action для розгортання застосунку

3.6 Backend CI / CD

3.6.1 Sentry monitoring

3.6.1.1 Загальне відстеження помилок

Мінімум Sentry - це відстеження помилок. Для цього необхідно виконати наступні кроки:

1. Створення проекту в Sentry;
2. Налаштування sentry DSN в config.json або .env;
3. Ініціалізація в app.ts (рисунок 3.6.1.1.1).

Для typescript проектів також потрібні source maps, щоб легко ідентифікувати помилковий код. Для налаштування потрібно:

- Явно встановити tslib у проект;
- Запустити застосунок з підтримкою source map в hosted environment (ex. in gcp: add -r source-map-support/register в команді start в package.json).

```
// Initialize as first item in the constructor
private initSentry() {
    if (!['development', 'test'].includes(process.env.NODE_ENV)) return;
    Sentry.init({ dsn: config.sentryDns, environment: process.env.NODE_ENV });
}

// Initialize as last item in the constructor (end of the startup)
private initializeErrorHandling() {
    this.app.use(
    Sentry.Handlers.errorHandler({
        shouldHandleError(error: HttpError) {
        return !error.httpCode || error.httpCode === StatusCode.serverError;
        },
    }) as express.ErrorRequestHandler,
    );
    this.app.use(errorMiddleware);
}

// Initialize as first middleware
private initializeSentryMiddleware() {
    if (!['development', 'test'].includes(process.env.NODE_ENV)) return;
    this.app.use(Sentry.Handlers.requestHandler() as express.RequestHandler);
}
```

Рис. 3.6.1.1.1 Налаштування sentry

3.6.1.2 Відстеження продуктивності

Відстеження продуктивності дуже корисно для виявлення повільних ендпоінтів і для розуміння того, як часто вони використовуються. Sentry має чудову підтримку транзакцій, де надається можливість відстежувати всі кроки ендпоінтів і навіть глибоко зануритися в продуктивність запиту. Для цього необхідно ввімкнути кілька інтеграцій:

- http tracing;
- express;
- postgres.

Кроки для повного налаштування відстеження продуктивності:

1. Встановлення sentry tracing (yarn add @sentry/tracing).
2. Додавання інтеграцій та частоти дискретизації до ініціалізації Sentry (рисунок 3.6.1.2.1). Частота дискретизації – це число від

0 до 1. Вона вказується в config.json або .env та використовується для запобігання великих витрат на проект.

```
Sentry.init({
  dsn: config.sentryDsn,
  environment: process.env.NODE_ENV,
  integrations: [
    new Sentry.Integrations.Http({ tracing: true }),
    new Integrations.Express({ app: this.host, router: this.host._router }),
    new Integrations.Postgres({ usePgNative: false }),
  ],
  tracesSampleRate: config.sentrySampleRate,
});
```

Рис. 3.6.1.2.1 Додавання інтеграцій та частоти дискретизації до ініціалізації Sentry

3. Налаштування Sentry middleware (рисунок 3.6.1.2.2), щоб увімкнути відстеження користувачів та запитів:
    a. В обробнику запитів налаштовуємо використання ip, userId та даних запиту;
    b. В обробнику трасування - відстеження всіх запитів;
    c. У процесорі подій - встановлення userId та ігнорування JWT.

```
private initializeSentryMiddleware() {
    if (['development', 'test'].includes(process.env.NODE_ENV)) return;

    this.host.use(
        Sentry.Handlers.requestHandler({
            ip: true,
            user: ['id'],
            request: true,
        })
    );
    this.host.use(Sentry.Handlers.tracingHandler());

    this.host.use((req, _, next) => {
        Sentry.configureScope((scope) => {
            scope.addEventProcessor((e) => {
            e?.request?.headers && delete e?.request?.headers['x-auth'];
            e.user && (e.user.id = (req as any)?.token?.userId);
            return e;
            });
        });
        next();
    });
}
```

Рис. 3.6.1.2.2 Налаштування Sentry middleware

3.6.1.3 Відстеження релізів

Sentry може автоматично ідентифікувати реліз на основі хешу коміту, коли доступний контекст git. У GCP це не так, тому потрібно встановити це вручну.

Для налаштування відстеження релізу потрібно зробити дії:

1. Ідентифікація різних версій програми (рисунок 3.6.1.3.1):
    a. Додати COMMIT_HASH=unknown_version до .env або config.json;
    b. Встановити COMMIT_HASH як ${{github.sha}} в deployment workflows які будуть включені як env змінні in GCP.

```
release: process.env.COMMIT_HASH,
// OR
release: config.commitHash,
```

Рис. 3.6.1.3.1 Налаштування ідентифікації різних версій програми

2. Контекст Git і відстеження змін:
    a. Додавання кроку в deployment workflows (рисунок 3.6.1.3.2).

```
- name: Create Sentry release
  uses: getsentry/action-release@v1
  env:
    SENTRY_AUTH_TOKEN: ${{secrets.SENTRY_AUTH_TOKEN}}
    SENTRY_ORG: panenco
    SENTRY_PROJECT: <product>-api
  with:
    environment: <environment>
```

Рис. 3.6.1.3.2 Налаштування контексту Git і відстеження змін

3.6.2 Cloud SQL and Postgres dumps

Часом, під час розробки та підтримки застосунку виникає проблема в production або staging, яку не вдається відтворити локально. Завантаження dump з Google Cloud та імпорт його локально може допомогти знайти проблему. Це дозволить запустити специфічні тести локально.

Для генерації dump файлу, потрібно перейти до платформи Google Cloud та відкрити сторінку sql. В правому верхньому куті потрібно натиснути на кнопку експорту, обрати тип “sql” та storage bucket, в який буде відбуватися експорт. Після успішної генерації, файл потрібно завантажити.

Для імпорту даних в локальну базу локального докер контейнера, потрібно виконати команду:

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

3.7 Frontend CI / CD

3.7.1 Sentry monitoring

Щоб полегшити процес обробки помилок, можна завантажити код до Sentry (create Sentry release). Це дозволяє отримати чітке уявлення про помилки у програмі та показує де вони сталися. Для цього використовується action getsentry/action-release (рисунок 3.7.1.1).

```yaml
name: Sentry release

on:
  push:
    branches:
      - master

jobs:
  sentry-release:
    # The type of runner that the job will run on
    runs-on: ubuntu-latest
    env:
      # Secrets are set in a repo settings
      SENTRY_AUTH_TOKEN: ${{ secrets.SENTRY_AUTH_TOKEN }} # auth token
      SENTRY_ORG: ${{ secrets.SENTRY_ORG }} # panenco
      SENTRY_PROJECT: ${{ secrets.SENTRY_PROJECT }} # your app
    steps:
      - name: Checkout
        uses: actions/checkout@v2
      - name: Create Sentry release
        uses: getsentry/action-release@v1.0.0
        with:
          environment: production
          sourcemaps: './dist' // folder with builded sourcemaps
```

Рис. 3.7.1.1 Action getsentry/action-release

3.7.2 Cypress

3.7.2.1 Test setup with API Container

Для роботи з тестами End-To-End необхідно встановити nvm, docker та yarn. Для тестів використовується окремий API контейнер. Щоб аутентифікуватися в реєстрі контейнерів у GitHub Actions workflow, потрібно використовувати GITHUB_TOKEN. Для налаштування контейнера локально, потрібно згенерувати персональний токен доступу в налаштуваннях github.

При генерації ключа обов'язково потрібно обрати наступні пункти (рисунок 3.7.2.1.1):

- write:packages - для стягування та завантаження container images та доступу до читання метаданих;
- read:packages - для завантаження container images та доступу до читання метаданих;
- delete:packages - для видалення container images.

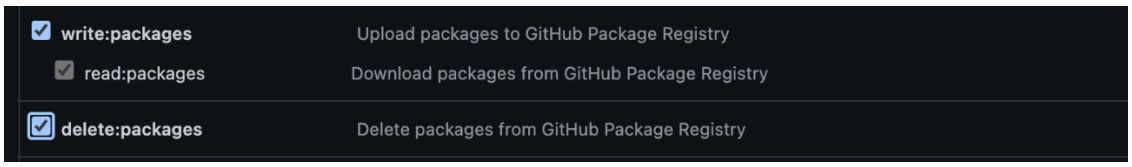

Рис. 3.7.2.1.1 Обов'язкові пункти при генерації ключа

Використовуючи CLI для обраного типу контейнера, потрібно виконати вхід у сервіс реєстру контейнерів на ghcr.io (рисунок 3.7.2.1.2).

```
echo <YOUR_PERSONAL_ACCESS_TOKEN> | docker login ghcr.io -u <YOUR_GITHUB_USERNAME> --password-stdin
```

Рис. 3.7.2.1.2 Команда для входу до реєстру контейнерів на ghcr.io

Для завантаження останньої версії API контейнера, потрібно виконати команду “docker pull” (рисунок 3.7.2.1.3).

```
docker pull ghcr.io/<API_CONTAINER_NAME>:latest
```

Рис. 3.7.2.1.3 Команда для завантаження останньої версії API контейнера

Для запуску docker контейнера потрібно виконати команду: “docker-compose up -d”. Для перегляду запущених контейнерів та щоб визначити назву контейнера API, потрібно виконати команду: “docker ps”.

Також, для написання тестів, локальну базу даних потрібно наповнити базовими тестовими даними. Для цього потрібно виконати відповідний API fixtures скрипт (рисунок 3.7.2.1.4).

```
docker exec <API_CONTAINER_NAME> yarn load-fixtures # Or the appropriate API fixtures script
```

Рис. 3.7.2.1.4 API fixtures скрипт

3.3 Висновки до розділу 3

Поетапно розглянуто процес підготовки клієнтської та серверної частини застосунку до розгортання на платформі Google Cloud. Налаштування Github репозиторію та написання Github actions для автоматичної перевірки та розгортання нових версій застосунку. Описано особливості підключення інструментів автоматизованого тестування та моніторингу та їх інтеграція в процес автоматичного розгортання.

Висновки

Під час виконання курсової роботи було досліджено бібліотеки та інструменти для розробки клієнтської та серверної частини веб застосунку. Розглянуто процес створення структури, процес підключення та налаштування важливих компонентів клієнтського застосунку. Описано багаторівневу архітектуру Express REST API, підключення та налаштування Mikro ORM в TypeScript проекті та детально визначено процес аутентифікації та авторизації на стороні сервера.

Важливою частиною дослідження є процес автоматизованого розгортання та CI/CD інтеграції. В роботі розглядається процес підготовки репозиторію, конструювання застосунку, створення інструкцій для автоматизованого розгортання, моніторингу та тестування клієнтської і серверної частини застосунку.

В результаті виконання роботи було визначено та описано особливості розробки, налаштування моніторингу, тестування, підготовки та налаштування проекту для подальшого автоматизованого застосунку на платформі Google Cloud Platform.

Список використаної літератури та електронних ресурсів

1. Документація бібліотеки React.js. [Електронний ресурс]. – Режим доступу: https://uk.reactjs.org/
2. Документація бібліотеки Node.js. [Електронний ресурс]. – Режим доступу: https://nodejs.org/en/docs/
3. Документація бібліотеки Express. [Електронний ресурс]. – Режим доступу: http://expressjs.com/en/api.html
4. Bulletproof node.js project architecture. [Електронний ресурс]. – Режим доступу: https://dev.to/santypk4/bulletproof-node-js-project-architecture-4epf
5. Route-Controller-Service structure for ExpressJS. [Електронний ресурс]. – Режим доступу: https://sodocumentation.net/node-js/topic/10785/route-controller-service-structure-for-expressjs
6. Документація бібліотеки React Router. [Електронний ресурс]. – Режим доступу: https://reactrouter.com/web/guides/quick-start
7. Документація бібліотеки React Query. [Електронний ресурс]. – Режим доступу: https://react-query.tanstack.com/
8. Practical React Query. [Електронний ресурс]. – Режим доступу: https://tkdodo.eu/blog/practical-react-query
9. Google Kubernetes Engine documentation. [Електронний ресурс]. – Режим доступу: https://cloud.google.com/kubernetes-engine/docs
10. Google App Engine documentation. [Електронний ресурс]. – Режим доступу: https://cloud.google.com/appengine/docs