КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

автоматизації і інформаційних технологій

(факультет)

інформаційних технологій

(кафедра)

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО АТЕСТАЦІЙНОЇ ВИПУСКНОЇ РОБОТИ
НА ЗДОБУТТЯ ОСВІТНЬОГО РІВНЯ «БАКАЛАВР»

на тему: «Бек-енд розробка автоматизованої системи оцінки психологічного стану»

СУСІДКО ВЛАДИСЛАВ ВІТАЛІЙОВИЧ

(прізвище, ім’я та по батькові студента повністю)

Київ 2024 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

автоматизації і інформаційних технологій

(факультет)

інформаційних технологій

(кафедра)

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТ
к.т.н., доцент Гончаренко Т. А.

„___” _____ 2024 року

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО АТЕСТАЦІЙНОЇ ВИПУСКНОЇ РОБОТИ
НА ЗДОБУТТЯ ОСВІТНЬОГО РІВНЯ «БАКАЛАВР»

на тему: «Бек-енд розробка автоматизованої системи оцінки психологічного стану»

Виконав: студент 4-го курсу, групи КНс-21

Спеціальності: 122 «Комп`ютерні науки»

Спеціалізація: «Інформаційні управляючі системи і технології»
(шифр і назва напряму підготовки, спеціальності)

Сусідко В. В.
(прізвище та ініціали)

Керівник к.т.н., доц. Горда О. В.
(прізвище та ініціали)

Рецензент к.т.н., доц. Шабала Є. Є.
(прізвище та ініціали)

Київ, 2024 р.

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
БУДІВНИЦТВА І АРХІТЕКТУРИ

Факультет: автоматизації і інформаційних технологій
Кафедра: інформаційних технологій
Освітній рівень: «бакалавр» за ОП
Спеціальність: 122 «Комп`ютерні науки»
Спеціалізація: Інформаційні управляючі системи і технології

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТ
к.т.н., доцент Гончаренко Т. А.

„ 12 ” лютого 2024 року

З А В Д А Н Н Я
ДО ВИКОНАННЯ АТЕСТАЦІЙНОЇ ВИПУСКНОЇ РОБОТИ
НА ЗДОБУТТЯ ОСВІТНЬОГО РІВНЯ «БАКАЛАВР»

Сусідко Владислав Віталійович
Тема роботи: Бек-енд розробка автоматизованої системи оцінки психологічного стану
затверджена наказом ректора КНУБА № 2650/2 від «12» лютого 2024 р.
2. Керівник роботи: Горда Олена Володимирівна, к.т.н, доцент кафедри інформаційних технологій проектування та прикладної математики
3. Строк подання студентом роботи до захисту:
4. Зміст пояснювальної записки за розділами:
Р.1. Аналіз предметної області та постановка задачі
Р.2. Проєктування програмного забезпечення
Р.3. Розробка програмного забезпечення
Р.4. Впровадження системи в експлуатацію
5. Інформаційні слайди:
С.1. Актуальність проблеми
С.2. Архітектура побудови серверного додатку
С.3. UML діаграма послідовності реєстрації користувача у системі
С.4. Електронний лист з підтвердженням реєстрації облікового запису
С.5. UML діаграма класів функціоналу перевірки тестування

С.6. UML діаграма класів сервісу зберігання та обробки файлів
С.7. UML діаграма класів сервісу відправки електронних листів
С.8. Огляд рішень для розгортання системи
С.9. Автоматизація розгортання інфраструктури
С.10. Граф залежностей компонентів інфраструктури системи
С.11. Налаштування CI/CD процесів
С.12. Тестовий приклад

6. Календарний план виконання атестаційної випускної роботи

Види робіт та їх зміст	Дата виконання
Р.1. Аналіз предметної області та постановка задачі	Лютий 2024 р.
Р.2. Проєктування програмного забезпечення	Березень 2024 р.
Р.3. Розробка програмного забезпечення	Квітень 2024 р.
Р.4. Впровадження системи в експлуатацію	Травень 2024 р.
Остаточне оформлення роботи	Травень 2024 р.
Направлення роботи на рецензування	Червень 2024 р.
Попередній захист роботи на кафедрі	Червень 2024 р.

7. Консультанти розділів атестаційної випускної роботи

Розділ	Прізвище, ініціали та посада консультанта, представника комісії	дата	підпис
Впровадження системи в експлуатацію	к.т.н., доц. Рябчун Ю.В.		
Прийом програмного продукту	к.т.н., доц. Шабала Є.Є.		

8. Дата видачі завдання: 12 лютого 2024 р.

Керівник _____ Горда О. В.
(підпис) (прізвище та ініціали)

Бакалавр _____ Сусідко В. В.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сусідко В. В. Бек-енд розробка автоматизованої системи оцінки психологічного стану

Атестаційна випускна робота бакалавра за спеціальністю: 122 «Комп’ютерні науки», спеціалізація: «Інформаційні управляючі системи і технології». – Київський національний університет будівництва та архітектури. – Київ, 2024.

Робота присвячена розробці серверної частини автоматизованої системи оцінки психологічного стану людей, які постраждали внаслідок бойових дій. Реалізація системи включає проєктування, розробку програмного забезпечення та впровадження системи в експлуатацію з використанням сучасних технологій.

Ключові слова: психологічний стан, діагностика, автоматизована система, бек-енд, .NET, GraphQL, Docker, Kubernetes, Terraform, Azure, CI/CD.

SUMMARY

Susidko V. V. Back-end development of the automated psychological assessment system.

Bachelor's thesis in the specialty: 122 "Computer Science", specialization: "Information managing systems and technologies". – Kyiv National University of Construction and Architecture. – Kyiv, 2024.

The thesis is dedicated to the development of the server part of the automated system for assessing the mental state of people affected by hostilities. The implementation of the system includes design, software development, and deployment of the system using modern technologies.

Keywords: mental state, diagnostics, automated system, back-end, .NET, GraphQL, Docker, Kubernetes, Terraform, Azure, CI/CD.

ЗМІСТ

Перелік умовних позначень

ООП – Об'єктно-орієнтоване програмування

AKS – Azure Kubernetes Service

API – Application Programming Interface

AWS – Amazon Web Services

BDI – Beck Depression Inventory

CD – Continuous Delivery / Continuous Deployment

CI – Continuous Integration

CI/CD – Continuous Integration/Continuous Delivery (або Continuous Deployment)

CLI – Command Line Interface

DevOps – Development and Operations

GCP – Google Cloud Platform

GPU – Graphics Processing Unit

HTML – Hypertext Markup Language

HTTP – Hypertext Transfer Protocol

I/O – Input/Output

IP – Internet Protocol

IaC – Infrastructure as Code

IoC – Inversion of Control

JWT – JSON Web Token

REST – Representational State Transfer

S3 – Simple Storage Service

SMTP – Simple Mail Transfer Protocol

SSL – Secure Sockets Layer

TLS – Transport Layer Security

UML – Unified Modeling Language

URL – Uniform Resource Locator

Вступ

Актуальність дослідження. В сучасному світі зростає поширеність психічних захворювань. Після повномасштабного вторгнення РФ в Україну чисельність людей, страждаючих від психічних хвороб, стрімко зросла через пережиті ними травматичні події. Запорукою вдалого лікування психічних захворювань є рання діагностика і виявлення цих самих захворювань. Щоб полегшити процес діагностики і зробити його більш доступним, доцільно розробити автоматизовану систему оцінки психічного стану людини. Дана система складається з наступних компонентів: клієнтська частина, серверна частина та база даних. Кожен з цих компонентів являється невід'ємною частиною інформаційної системи та не може бути спростований. Саме тому бек-енд розробка даної системи є актуальною темою дослідження, оскільки дана система дасть змогу полегшити процес діагностики і зробить його більш доступним, що призведе до вчасного виявлення проблеми з психічним станом людини і в результаті цього збільшить вірогідність одужання пацієнта.

Мета дослідження. Розробка бек-енд системи оцінки психологічного стану людей, які постраждали від бойових дій або від інших психологічно-травмуючих подій. Після тестування користувач побачить вірогідність різноманітних психічних порушень та рекомендації, які потрібно вжити для їх усунення. Система буде складатися з тесту, який буде включати питання про симптоми і пережиті події. На основі отриманих даних буде формуватися оцінка поточного стану особи та надаватись відповідні рекомендації. Якщо система виявить, що людина має високий ризик розвитку психічного захворювання, їй дадуть рекомендації подальшого обстеження у психіатра або психолога.

Об'єкт дослідження. Серверне програмне забезпечення автоматизованої системи оцінки психічного стану, яке буде відповідати за весь функціонал системи, включаючи реєстрацію, проходження тесту та надання відповідних результатів та рекомендацій користувачу.

Предмет дослідження. Методи, моделі та інструментальні засоби побудови архітектури, розробки та впровадження в експлуатацію серверного програмного забезпечення. Цей предмет також визначає дослідження взаємодії бек-енд системи з іншими системами проєкту: клієнтською частиною та базою даних.

Методи дослідження: методи системного аналізу, моделювання, об’єктно орієнтованого програмування, розробки та впровадження веб-систем.

Практична значимість. Результати дослідження можуть бути використані для розробки та реалізації ефективних систем оцінки психічного стану людини. Така система буде сприяти швидшому одужанню постраждалих людей від військових дій та інших психічних травм, та буде сприяти їхній подальшій адаптації у суспільстві.

Короткий зміст розділів:

У першому розділі частково описана загальна та теоретична інформація щодо відомостей, концепцій та проблематики розробки бек-енд частини автоматизованої системи оцінки психічного стану. Описується постановка та аналіз проблеми, дерево цілей, встановлюються вимоги та особливості проєктування системи, аналізуються існуючі розробки автоматизованих систем оцінки психічного стану, проводиться аналіз технологій розробки серверного програмного забезпечення, встановлюється задача дипломного проєкту.

Другий розділ присвячений проєктуванню програмного забезпечення. В ньому наведений перелік функціональних та не функціональних вимог до програмного забезпечення, розглянути сучасні архітектурні рішення побудови складних веб-систем, проєктування інфраструктури додатку та моделювання основних процесів системи.

Третій розділ описує реалізацію системи. Описується створення та налаштування проєкту, розробка основного функціоналу системи включаючи реєстрацію користувача, авторизацію та проходження загального тестування. Також описується розробка інфраструктури додатку, тобто сервіс зберігання та обробки файлів та сервіс відправки електронних листів. Заключною частиною

даного розділу є налаштування конфігурації проєкту та контейнеризація додатку з використанням платформи Docker.

У четвертому розділі розглянуто впровадження системи в експлуатацію, а саме описані ергономічні вимоги до проєкту, оглянуті рішення для розгортання системи і обґрунтований вибір використання послуг хмарного провайдера Azure. Окрім цього, описаний процес створення та налаштування інфраструктури на хмарному провайдері Azure, автоматизована інфраструктура з використанням Infrastructure as Code рішення Terraform та налаштовані CI/CD (Continuous Integration/Continuous Delivery (або Continuous Deployment)) процеси проєкту з використанням платформи GitHub Actions.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Постановка та аналіз проблеми

Для визначення напрямку даної роботи, необхідно спочатку визначити об’єкт дослідження і предмет дослідження, а також надати опис, за допомогою яких методів планується проводити дослідження і вирішення проблеми.

Отже, для дослідження і побудови імітаційної моделі підсистеми виділяємо наступні сутності:

- Об’єктом дослідження даної роботи є серверне програмне забезпечення автоматизованої системи оцінки психічного стану.
- Предмет дослідження роботи є побудова архітектури, розробка та впровадження в експлуатацію серверного програмного забезпечення. В основі програмного забезпечення лежить додаток, який відповідає за комунікацію з клієнтом та взаємодію з базою даних.
- Методи дослідження – на початку необхідно провести дослідження існуючих систем для оцінки психічного стану людини, таких як психологічні тестування; вивчити наукові статті та книги, що стосуються теорій психічного здоров'я, методів оцінки та інформаційних технологій в цій галузі; визначити параметри, які можуть бути використані для оцінки психічного стану; обрати стек технологій для розробки серверного програмного забезпечення та побудувати архітектуру всього додатку.

Розробка автоматизованої системи оцінки психічного стану людини є новою та перспективною сферою людської діяльності, оскільки має потенціал для суттєвого покращення охорони здоров’я людини. Автоматизовані системи оцінки психічного стану мають ряд переваг порівняно з традиційними методами. Вони можуть бути більш об’єктивними, точними та швидкими. Розробка інформаційного забезпечення автоматизованої системи оцінки психічного стану є важливим завданням, яке має потенціал для трансформації охорони здоров’я людини.

Поетапний процес розробки інформаційного забезпечення автоматизованої системи оцінки психічного стану можливо представити у вигляді життєвого циклу, серед яких виділяють наступні основні етапи:

1. Етап планування;
2. Етап збору вимог;
3. Етап проєктування;
4. Етап розробки програмного забезпечення;
5. Етап тестування програмного забезпечення;
6. Використання готового продукту на ринок;
7. Етап експлуатації та технічного обслуговування.

На першому етапі узгоджуються всі деталі майбутньої автоматизованої системи оцінки психічного стану людини, відповідаючи на важливі питання. Яка мета розробки? Які проблеми воно має вирішувати? Навіщо створюється система? Після відповіді на ці питання формується єдине бачення майбутньої системи. Що полегшує розробку та мінімізує ризики при створенні ПЗ.

Етап аналізу та збору вимог є одним із найважливіших етапів життєвого циклу розробки автоматизованої системи оцінки психічного стану людини. На цьому етапі збираються всі необхідні дані для успішної розробки та впровадження системи.

Після того, як стали зрозумілими цілі та завдання системи, а також технології, які будуть використовуватися для її розробки, можна переходити до проектування та дизайну. На цьому етапі проєктується майбутня архітектура проєкту у вибраній технології. Створюється адаптивний та зручний дизайн, продумується зв'язок фронт-енд частини програми з сервером, розробляються модулі та продумується система безпеки ресурсу.

Стадія розробки – це етап, на якому відбувається створення програмного продукту. На цьому етапі береться проєктна документація, прототипи, дизайн та архітектура, і на основі їх створюють код, який реалізує всі функціональні можливості системи. Завдання на цьому етапі розділені між членами команди відповідно до їхньої галузі спеціалізації. Розробники front-end частини

відповідають за створення інтерфейсу користувача, який буде відображатися на веб-браузері. Розробники back-end частини відповідають за створення серверної частини системи, яка обробляє дані та відповідає за функціональність системи. Адміністратори бази даних відповідають за створення та наповнення бази даних системи. Результатом етапу розробки є готовий програмний продукт, який відповідає вимогам, визначеним на етапі аналізу та збору вимог.

На етапі тестування команда перевіряє, чи відповідає розроблена система вимогам, визначеним на етапі аналізу та збору вимог. Тестування включає в себе такі основні завдання:

- Перевірка функціональності системи;
- Перевірка ефективності системи;
- Перевірка безпеки системи.

Після того, як система протестована і готова до використання, її запускають в експлуатацію. Система стає доступною для користувачів, які можуть її використовувати для оцінки свого психічного стану. Замовник збирає відгуки від користувачів, щоб зрозуміти, чи відповідає система їхнім потребам. Якщо в результаті збору відгуків виявляються помилки в системі, їх виправляють розробники.

Технічне обслуговування – це етап, на якому система підтримується в робочому стані і забезпечується її ефективне використання. На цьому етапі система постійно перевіряється на наявність помилок і проблем. Також на цьому етапі можуть вноситися зміни в систему для її поліпшення. Ці зміни можуть бути спрямовані на:

- усунення виявлених проблем;
- розширення функціональності системи;
- вдосконалення інтерфейсу користувача;
- підвищення продуктивності системи;
- забезпечення безпеки системи.

1.2 Дерево цілей

Основною метою роботи є отримання функціональної серверної частини автоматизованої системи оцінки психологічного стану. Дерево цілей опису даної мети наведено на рис. 1.1.

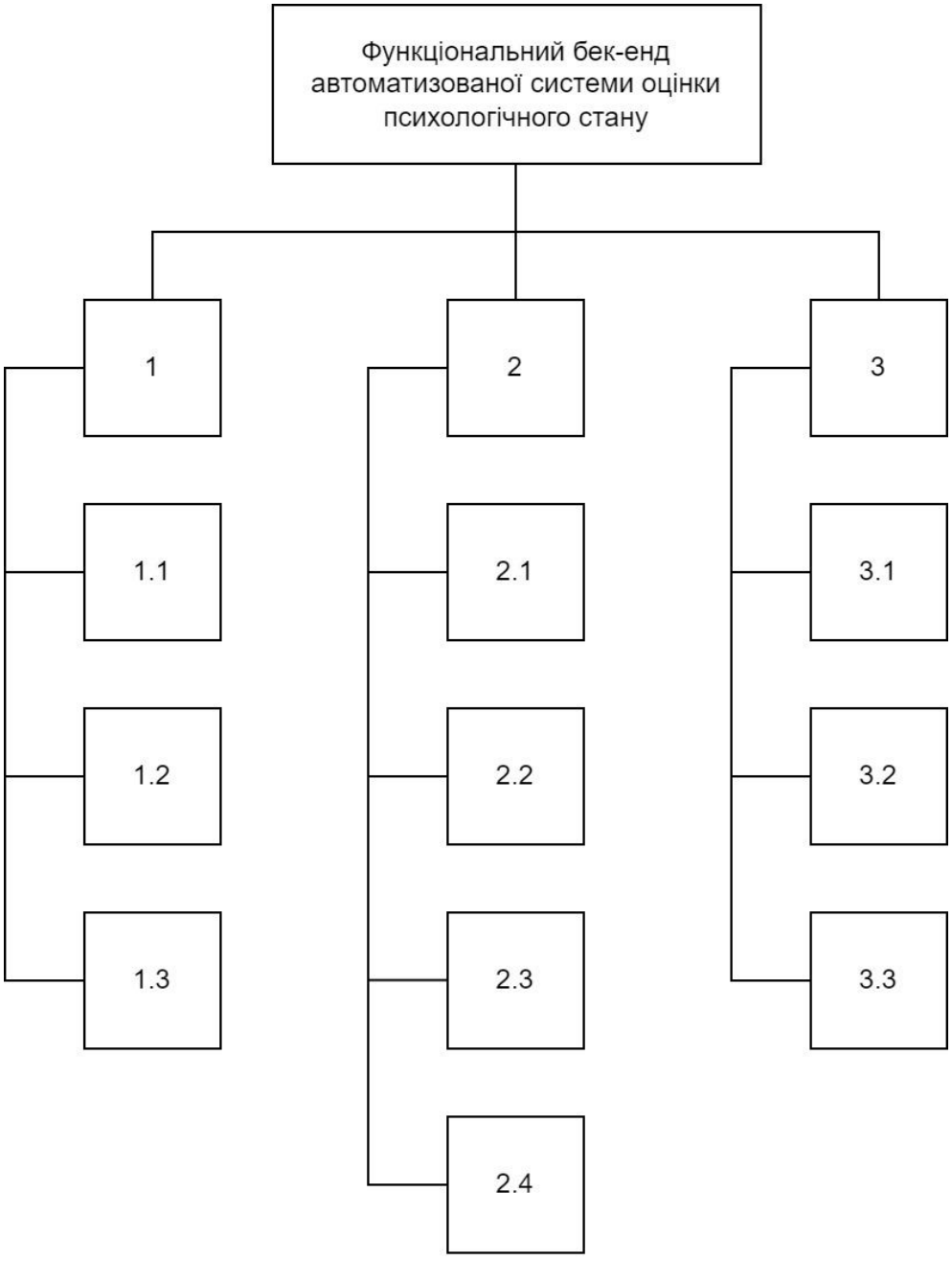

Рисунок 1.1. Дерево цілей

1. Налаштування циклу розробки програмного забезпечення.
    1.1. Налаштування IDE.
    1.2. Налаштування віддаленого репозиторію.
    1.3. Налаштування CI/CD процесів.
2. Розробка системи.
    2.1. Аналіз технічного завдання.
    2.2. Вибір стеку технологій.
    2.3. Побудова архітектури серверного додатку.
    2.4. Імплементація функціоналу.
3. Впровадження в експлуатацію.
    3.1. Контейнеризація додатку.
    3.2. Налаштування хмарної інфраструктури.
    3.3. Розгортання додатку на хмарній інфраструктурі.

1.3 Вимоги та особливості проєктування системи

Бек-енд частина системи автоматизованої оцінки психологічного стану людини є відповідальною за обробку даних, отриманих від користувача, і надання результатів оцінки. Вона повинна відповідати ряду вимог, які забезпечать її ефективність і надійність.

Однією з основних вимог є безпека. Дані, які надходять від користувача, можуть бути конфіденційними, тому вони повинні бути захищені від несанкціонованого доступу. Для цього необхідно використовувати сучасні методи шифрування та аутентифікації.

Іншою важливою вимогою є стабільність. Бек-енд система повинна працювати безперебійно, навіть при великому навантаженні. Для цього необхідно використовувати надійні сервери і мережеві технології.

Крім того, бек-енд система повинна бути ефективною. Вона повинна обробляти дані швидко і точно, щоб не затримувати час оцінки. Для цього

необхідно використовувати сучасні алгоритми обробки даних і ефективні бази даних.

Для забезпечення безпеки бек-енд системи необхідно використовувати такі заходи:

- Шифрування даних. Всі дані, які надходять від користувача, повинні бути зашифровані перед їх передачею на сервер. Для цього можна використовувати такі алгоритми шифрування, як AES, RSA або ECC.
- Аутентифікація користувачів. Кожен користувач повинен бути ідентифікований перед тим, як він зможе отримати доступ до системи. Для цього можна використовувати такі методи аутентифікації, як паролі, біометрія або двофакторна аутентифікація.
- Контроль доступу. Доступ до різних ресурсів системи повинен бути обмежений відповідно до ролі користувача. Для цього можна використовувати такі механізми контролю доступу, як ролі, дозволи та аудит.

Для забезпечення стабільності бек-енд системи необхідно використовувати такі заходи:

- Надійні сервери. Бек-енд система повинна працювати на надійних серверах, які мають достатньо ресурсів для обробки даних.
- Мережеві технології. Бек-енд система повинна використовувати надійні мережеві технології, які забезпечують безперебійну передачу даних.
- Автоматичне відновлення. Система повинна мати механізм автоматичного відновлення в разі непередбачених збоїв.

Для забезпечення ефективності бек-енд системи необхідно використовувати наступні заходи:

- Сучасні алгоритми обробки даних. Бек-енд система повинна використовувати сучасні алгоритми обробки даних, які дозволяють швидко і точно обробляти великі обсяги даних.

- Ефективні бази даних. Бек-енд система повинна використовувати ефективні бази даних, які дозволяють швидко шукати та отримувати дані.

Крім того, бек-енд система повинна бути розроблена з урахуванням вимог користувачів. Вона повинна бути легкою у використанні і надавати користувачам доступ до всіх необхідних функцій.

1.4 Аналіз існуючих розробок систем оцінки психологічного стану

У сучасному світі автоматизовані системи оцінки психічного стану людини через тестування стають все більш актуальними та важливими для забезпечення ментального здоров'я населення. Такі системи дозволяють здійснювати швидку та ефективну оцінку психічного стану, забезпечуючи можливість раннього виявлення та лікування різних психічних розладів. У даному аналізі розглянуто найбільш відомі системи автоматизованої оцінки психічного стану, які вже існують на ринку.

Система «Я-ПСИХОЛОГ» – інформаційна платформа, що дозволяє проводити психологічні тестування та діагностику в режимі онлайн. Основна перевага системи полягає не лише у визначенні психологічного стану, але й у створенні індивідуальних психологічних характеристик для кожного користувача. Платформа використовує діагностичні методики та психологічні тести, які отримали рекомендації від Міністерства освіти і науки, молоді та спорту, Міністерства охорони здоров'я, Міністерства оборони України та інших важливих установ для використання в Україні.

MARTA – проєкт, анонсований міністерством ще у 2021 році під час зустрічі з представниками компанії Apple, представляє інноваційну систему для психологічної підтримки та оцінки психічного здоров’я учасників бойових дій. Проект був презентований Apple у формі демонстраційної версії, яка мала дозволити бійцям отримувати психологічну підтримку та оцінювати своє психічне здоров'я через смартфон або комп'ютер. Згодом MARTA стала

доступною широкому загалу користувачів. Система включає онлайн-тести, призначені для широкого кола користувачів, та допомагає визначити попередню характеристику психічного стану.

Beck Depression Inventory (BDI) Online – онлайн-версія відомого тесту Аарона Бека для визначення рівня депресії. Тест включає в себе питання, спрямовані на виявлення різних аспектів депресивних симптомів. Його перевагами є науково обґрунтована методика та широке застосування в клінічній практиці. Однак необхідно враховувати, що він вимагає від користувача високого рівня самосвідомості.

Anima – онлайн-платформа, яка дозволяє оцінити психологічне здоров'я за допомогою веб-камери. Весь процес передбачає проведення трьоххвилинного тесту, під час якого показуються різноманітні зображення, такі як їжа, пейзажі, портрети тощо. Програма аналізує рухи очей та їх реакцію на зображення, роблячи конкретні висновки. Наприклад, вона допомагає визначити рівень тривожності чи депресії у людини. Аналіз здійснюється за допомогою технології відстеження очей, використовуючи вбудовані математичні моделі для фіксації рухів очей.

Самопоміч – сайт, розроблений Національним інститутом психічного здоров’я Чеської Республіки у співпраці з іншими національними та міжнародними організаціями, які беруть участь у співпраці. Вміст сайту ґрунтується на наукових даних, що означає, що представлена на веб-сайті інформація є достовірною та науково перевіреною. На сайті є набір корисних тестів, які широко використовуються для виявлення проблем зі психічним здоров'ям. Присутній базовий тест, який визначає стан поточного психічного стану. Інші тести зосереджені на конкретних аспектах психічного здоров'я, таких як депресія та тривога. Друга частина містить корисний огляд симптомів, які можуть супроводжувати різні проблеми з психічним здоров'ям.

Порівняння існуючих розробок систем оцінки психічного стану наведено у табл. 1.1.

Таблиця 1.1. Опис існуючих розробок систем оцінки психічного стану

Назва	Метод оцінювання	Країна	Аудиторія	Особливості
«Я-ПСИХОЛОГ»	Тести з питаннями	Україна	Цивільні	Створення індивідуальних психологічних характеристик для кожного користувача
MARTA	Тести з питаннями	Україна	Військові	Надання психологічної підтримки та оцінки психічного здоров’я учасників бойових дій
Beck Depression Inventory (BDI) Online	Тести з питаннями	Чехія	Цивільні	Науково обґрунтована методика та широке застосування в клінічній практиці
Anima	Тест за допомогою веб-камери та тести з питаннями	Україна	Цивільні	Програма аналізує рухи очей та їх реакцію на зображення, роблячи конкретні висновки

Продовження таблиці 1.1. Опис існуючих розробок систем оцінки психічного стану

Назва	Метод оцінювання	Країна	Аудиторія	Особливості
Самопоміч	Тести з питаннями	Чехія	Цивільні	Представлена на веб-сайті інформація є достовірною та науково перевіреною. Містить корисний огляд симптомів, які можуть супроводжувати різні проблеми з психічним здоров'ям

Порівнюючи різноманітні системи тестування психологічного стану, можна визначити декілька ключових аспектів, що варто враховувати при розробці власної автоматизованої системи з оцінки психічного стану. Усі розглянуті системи використовують тести з питаннями як основний метод оцінювання. Проте Anima вирізняється використанням веб-камери для аналізу рухів очей, що може надати додаткову інформацію про реакцію користувача. MARTA спеціалізується на військових, що робить її специфічною для цього сегменту користувачів. У той час як "Я-ПСИХОЛОГ", Beck Depression Inventory, Anima та Самопоміч орієнтовані на цивільну аудиторію. Розробка систем велась в різних країнах (Україна, Чехія), що може вплинути на особливості їх функціоналу та адаптацію до конкретного культурного та соціального середовища. Beck Depression Inventory та Самопоміч вирізняються науковою

обґрунтованістю своїх методик, що підсилює достовірність інформації, яку вони надають. Усі ці фактори свідчать про те, що вибір можливостей та особливостей для системи тестування психологічного стану повинен бути здійснений з урахуванням конкретних потреб користувачів, їх контексту та специфіки ситуації, для якої вони шукають рішення.

1.5 Постановка задачі.

Метою дипломного проєкту є розробка серверної частини інформаційної системи для автоматизованої оцінки психічного стану користувача, яка дозволить на основі отриманих даних з тестування формувати оцінку поточного стану особи та надавати їй відповідні рекомендації. Реалізація проєкту передбачає вибір стеку технологій для розробки, побудову архітектури бек-енд частини інформаційної системи, її розробку та розгортання для подальшого використання. Система повинна бути надійною та захищеною, витримувати навантаження та добре масштабуватись при необхідності. Очікуваним результатом є система, яка відмінно працює, відповідає всім вимогам та готова для подальшої експлуатації.

Основні завдання дипломного проекту:

1. Аналіз вимог користувачів та фахівців у галузі психічного здоров'я для визначення потреб і функціональності системи.
2. Вибір стеку технології, вивчення особливостей та можливостей обраного стеку технологій.
3. Проєктування архітектури серверної частини системи, організація структури проєкту.
4. Налаштування циклу розробки програмного забезпечення.
5. Розробка бек-енд функціоналу системи.
6. Тестування системи.
7. Впровадження системи в експлуатацію.

1.6 Аналіз та вибір технології для реалізації проєкту

Для реалізації серверної частини проєкту автоматизованої системи оцінки психологічного стану людини необхідно вибрати технологію, яка відповідатиме таким вимогам:

- Можливість масштабування. Система повинна бути здатна обробляти великі обсяги даних та забезпечувати високу пропускну здатність.
- Безпека. Система повинна бути захищена від несанкціонованого доступу та злому.
- Швидкість та продуктивність. Система повинна забезпечувати швидку обробку даних та відповідати вимогам користувачів.
- Легкість у використанні. Система повинна бути легкою у використанні для розробників та користувачів.

Для бек-енд розробки існує багато різних технологій, і вибір залежить від конкретних потреб проекту, особистих вподобань команди розробників, а також від екосистеми та підтримки, яку пропонує мова. Серед найпопулярніших мов програмування, які можна використовувати у бек-енд розробці, належать: Java, C#, PHP, Node.js та Python. Для вибору конкретної мови програмування необхідно зробити аналіз кожної з них з урахуванням вищезазначених вимог.

Java – це об'єктно-орієнтована мова програмування, розроблена компанією Sun Microsystems у 1995 році. Вона є однією з найпопулярніших мов програмування у світі, і широко використовується для розробки веб-додатків, мобільних додатків, а також серверних додатків.

Сильні сторони:

- Переносимість: Java код може виконуватися на різних платформах без переписування.
- Об'єктно-орієнтований підхід: Вбудована підтримка ООП (Об’єктно орієнтоване програмування) сприяє розробці складних систем.
- Велика спільнота і екосистема: Широка база користувачів і багатий вибір бібліотек та фреймворків.

Слабкі сторони:

- Зайва кількість коду: Деякі розробники вважають, що Java вимагає більше коду для досягнення того ж функціоналу порівняно з іншими мовами.
- Повільний запуск: Додатки на Java можуть запускатися повільніше порівняно з іншими мовами.

C# – це об'єктно-орієнтована мова програмування, розроблена компанією Microsoft у 2000 році. Вона є однією з найпопулярніших мов програмування для розробки веб-додатків та серверних додатків у середовищі .NET.

Сильні сторони:

- Інтеграція з Microsoft технологіями: C# добре інтегрується з продуктами Microsoft, що робить його ідеальним для розробки на платформах Windows.
- Ефективність та швидкодія: Код на C# може працювати ефективно та швидко завдяки оптимізації виконання.

Слабкі сторони:

- Портативність: C#-код може виконуватися лише на платформах, які підтримують середовище .NET, на даний момент це операційні системи Windows, Linux та MacOS.

PHP – це скриптова мова програмування, розроблена Rasmus Lerdorf у 1994 році. Вона широко використовується для розробки веб-додатків, а також для розробки мобільних додатків та серверних додатків.

Сильні сторони:

- Легкість вивчення: PHP є досить простою мовою для вивчення, що робить її популярною серед новачків.
- Велика спільнота: Велика кількість розробників і велика кількість готових рішень.

Слабкі сторони:

- Не найкраща підтримка ООП: Історично PHP не була повністю об'єктно-орієнтованою, хоча останні версії мають покращену підтримку ООП.
- Проблеми з безпекою: Недоліки в безпеці були проблемою в минулому, хоча останні версії PHP покращили цю ситуацію.

Node.js – це платформа з відкритим кодом, заснована на JavaScript, яка використовується для розробки веб-додатків, а також для розробки мобільних додатків та серверних додатків.

Сильні сторони:

- Асинхронність: Інтерфейс на основі подій та асинхронна модель дозволяють побудову високопродуктивних застосунків.
- JavaScript на обох сторонах: Якщо ви вже використовуєте JavaScript на стороні клієнта, Node.js дозволяє вам використовувати його і на сервері.
- Простота використання: Node.js є відносно простою платформою, що її робить хорошим вибором для початківців.
- Розширюваність: Node.js має велику бібліотеку модулів, які можна використовувати для додавання додаткових функцій до платформи.

Слабкі сторони:

- Не найкраща обробка великих обчислень: Node.js краще підходить для I/O-орієнтованих (Input / Output) застосунків, і великі обчислення можуть впливати на продуктивність.
- Брак вбудованої підтримки для багатозадачності: Високонавантажені застосунки можуть вимагати використання додаткових бібліотек для реалізації багатозадачності.

Python – це мова програмування загального призначення, розроблена Guido van Rossum у 1991 році. Python широко використовується для розробки веб-додатків, використовуючи фреймворки, такі як Django або Flask.

Сильні сторони:

- Простота читання і вивчення: Python вважається однією з найбільш читабельних мов, що полегшує розробку та підтримку коду.
- Широкий вибір бібліотек: Python має велику кількість бібліотек для різних завдань, що полегшує розробку.

Слабкі сторони:

- Швидкодія: У порівнянні з деякими іншими мовами, такими як C# або Java, Python може бути менш продуктивним з точки зору швидкодії.

З урахуванням цих вимог, для реалізації серверної частини проєкту автоматизованої системи оцінки психологічного стану людини найбільш доцільно використовувати мову програмування C# разом з платформою .NET. Дана мова відповідає всім вимогам проєкту, є популярною мовою з великою спільнотою розробників, а також забезпечує хорошу підтримку від Microsoft.

Переваги використання C# .NET для реалізації серверної частини проєкту:

- Безпека. C# .NET є безпечною мовою програмування, яка підтримує такі можливості, як типізація, управління пам'яттю та захист від переповнення буферів.
- Масштабованість. C# .NET є масштабованою мовою, яка може використовуватися для розробки великих і складних застосунків.
- Продуктивність. C# .NET є продуктивною мовою, яка забезпечує високу швидкість виконання коду.
- Легкість у використанні. C# .NET є відносно простою мовою для освоєння та використання.

2 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Опис функціональних та нефункціональних вимог програмного забезпечення

Функціональні та нефункціональні вимоги до програмного забезпечення є ключовим етапом у процесі розробки будь-якої інформаційної системи. Функціональні вимоги визначають, які функції має виконувати програмне забезпечення, які операції воно повинно забезпечувати та які результати мають бути отримані після виконання цих операцій. Нефункціональні вимоги визначають якість програмного забезпечення, такі як продуктивність, надійність, безпека, сумісність та інші аспекти, які не стосуються конкретних функцій програми.

Система повинна реалізовувати наступні функціональні вимоги:

- Реєстрація користувачів. Система повинна мати можливість реєстрації нових користувачів з подальшою можливістю збереження результатів тестування
- Авторизація користувачів. Доступ до системи повинен бути обмежений та контрольований за допомогою авторизації. Кожен користувач повинен мати можливість увійти до свого облікового запису.
- Проведення тестів. Система повинна мати можливість проведення тестів для оцінки психологічного стану користувача.
- Аналіз результатів. Система автоматично оброблятиме результати тестування та надаватиме список виявлених потенціальних психологічних проблем користувачу.
- Надання рекомендацій. Користувач повинен мати можливість не лише дізнатись свої потенційні розлади, але й отримати повний список рекомендацій, завдяки яким можна покращити свій психологічний стан.

- Особистий профіль користувача. Користувач повинен мати можливість переглядати та редагувати особисту інформацію та за бажанням переглядати результати своїх попередніх тестувань.

До нефункціональних вимог належать наступні вимоги:

- Ефективність. Система повинна забезпечувати швидку та точну оцінку психологічного стану людини без зайвого затримання. Це означає, що час, необхідний для проведення тестування та отримання результатів, повинен бути мінімальним, щоб користувачі могли швидко отримати інформацію про свій психологічний стан.
- Надійність. Система повинна працювати безперебійно та надійно, забезпечуючи коректну інтерпретацію результатів тестування. Надійність системи важлива для того, щоб користувачі могли довіряти отриманим результатам та використовувати їх для подальшого аналізу свого психологічного стану.
- Безпека. Система повинна гарантувати конфіденційність даних користувачів, забезпечуючи захист від несанкціонованого доступу. Це означає, що всі особисті дані користувачів повинні бути зашифровані та зберігатися в безпечному середовищі, щоб уникнути витоку конфіденційної інформації.
- Масштабованість. Система повинна бути готовою до масштабування, здатною обробляти великий потік даних та користувачів без втрати продуктивності. Це дозволить системі ефективно працювати навіть при збільшенні обсягу користувачів та даних.
- Сумісність. Система повинна бути сумісною з різними операційними системами та пристроями, що дозволить користувачам отримати доступ до неї з будь-якого пристрою. Це забезпечить доступність системи для широкого кола користувачів та зручність її використання.
- Документація. Наявність докладної та зрозумілої документації з описом функціоналу системи, процесу взаємодії та вимог до

користувачів. Це допоможе користувачам швидко ознайомитися з системою, її можливостями та правилами використання, що сприятиме успішній інтеграції системи у їхню роботу.

2.2 Побудова архітектури системи

2.2.1 Опис клієнт-серверної архітектури

Всі сучасні веб додатки розробляються на основі клієнт-серверної архітектури (рис. 2.1), де клієнтом може виступати веб-браузер, який взаємодіє з веб-сервером для отримання даних для відображення їх користувачу. Дана архітектура є однією з основних архітектурних моделей в розробці програмного забезпечення. Також ця архітектура застосовується у багатьох інших областях, де потрібна розподілена обробка даних та взаємодія між різними компонентами системи [1].

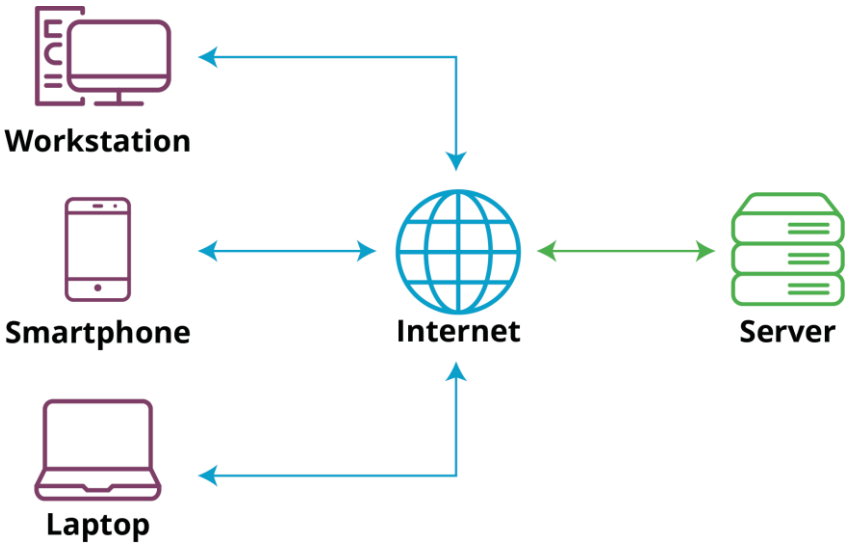

Рисунок 2.1. Клієнт-серверна архітектура

У клієнт-серверній архітектурі існують два основних типи компонентів:

- Клієнт – це програмний модуль або пристрій, який відправляє запити на сервер для отримання даних або виконання певних операцій. Клієнт може бути реалізований у вигляді десктопної програми, веб-браузера, мобільного додатку тощо.

- Сервер – це програма або пристрій, який приймає запити від клієнтів, обробляє їх і надсилає відповідь. Сервер зазвичай має доступ до ресурсів, баз даних або інших системних компонентів, які використовуються для виконання запитів.

Основні принципи клієнт-серверної архітектури включають:

- Розділення функціональності між клієнтом і сервером, що дозволяє покращити масштабованість та ефективність системи.
- Клієнт та сервер можуть взаємодіяти через мережу, що дозволяє розподіленій роботі системи.
- Система може бути розширена додаванням нових клієнтів або серверів без необхідності змінювати всю архітектуру.

Використання клієнт-серверної архітектури має безліч переваг, які роблять її дуже доцільною для різноманітних типів програмних систем. Ось деякі з головних переваг цієї архітектури:

- Розділення обов'язків: Клієнт та сервер виконують різні функції, що дозволяє розділити завдання між ними. Це сприяє покращенню швидкодії, масштабованості та обслуговування системи.
- Масштабованість: Клієнт-серверна архітектура дозволяє легко масштабувати систему шляхом додавання нових клієнтів або серверів. Це дозволяє підтримувати високу продуктивність та ефективність системи навіть при збільшенні обсягів даних чи користувачів.
- Централізоване управління: Сервер може забезпечувати централізоване управління даними, логікою додатку та безпекою. Це спрощує розгортання, моніторинг та адміністрування системи.
- Безпека: Клієнт-серверна архітектура дозволяє контролювати доступ до ресурсів та даних через сервер, що підвищує рівень безпеки системи.
- Можливість розширення та модифікації: Ця архітектура дозволяє легко розширювати та модифікувати систему, додавати нові функції та компоненти без значних змін у загальній архітектурі.

- Робота в реальному часі: Клієнт-серверна архітектура дозволяє взаємодіяти з даними та виконувати операції в реальному часі, що дуже важливо для багатьох додатків.

2.2.2 Вибір архітектурного стилю проєктування API системи

API (Application Programming Interface) – це набір правил та протоколів, які визначають, як різні компоненти програмного забезпечення можуть взаємодіяти один з одним. API дозволяє різним програмам та сервісам обмінюватися даними та функціональністю без необхідності розкриття джерела цих даних або функціональності. У веб-додатках API використовується для забезпечення комунікації між клієнтською та серверною частинами програми для отримання або надсилання даних клієнтом на сервер для обробки та їх відображення. Одним із архітектурних стилів, який описує ці правила взаємодії, є REST (Representational State Transfer).

REST – це архітектурний стиль, який використовується для створення веб-додатків. REST базується на принципах взаємодії між клієнтом і сервером через передачу та обмін станом ресурсів за допомогою стандартних HTTP (Hypertext Transfer Protocol) методів [2].

Основні принципи REST включають в себе:

- Клієнт-серверна архітектура: розділення між клієнтом і сервером, що дозволяє їм розвиватися незалежно один від одного.
- Відсутність збереження стану: кожен запит від клієнта містить всю необхідну інформацію для сервера для обробки запиту, без необхідності зберігання стану на сервері.
- Кешування: можливість кешування відповідей сервера для покращення продуктивності та зниження навантаження на сервер.
- Єдина точка доступу: кожен ресурс має унікальний URL (Uniform Resource Locator), що дозволяє однозначно ідентифікувати його.
- Шарування: можливість розділити логіку додатку на різні шари, що спрощує розробку та підтримку.

REST архітектура реалізована на основі протоколу HTTP. Даний протокол використовується для взаємодії між веб-серверами та клієнтськими програмами, такими як веб-браузери. HTTP використовує модель клієнт-сервер, де клієнтська програма ініціює запит до веб-сервера, який обробляє цей запит і повертає відповідь.

Протокол HTTP базується на текстових повідомленнях, які складаються з трьох основних елементів: запит, заголовки та тіло повідомлення. Запит включає метод (наведені у табл. 2.1), URL ресурсу та версію протоколу. Заголовки містять додаткову інформацію про запит або відповідь, таку як тип контенту, куки, довжина повідомлення тощо. Тіло повідомлення містить дані, які передаються між клієнтом та сервером [3].

Таблиця 2.1. Опис HTTP методів

HTTP метод	Опис
GET	Використовується для отримання ресурсів з сервера
POST	Використовується для надсилання даних на сервер для обробки
PUT	Використовується для оновлення або створення ресурсу на сервері
DELETE	Використовується для видалення ресурсу на сервері
PATCH	Використовується для часткового оновлення ресурсу на сервері
HEAD	Аналогічний до GET, але повертає тільки заголовки без тіла відповіді
OPTIONS	Використовується для запиту можливих методів комунікації з сервером

Продовження таблиці 2.1. Опис HTTP методів

HTTP метод	Опис
TRACE	Використовується для діагностики мережевих проблем та відстеження шляху запиту
CONNECT	Використовується для встановлення тунелю до сервера за допомогою проксі

HTTP використовується для передачі різних типів даних, включаючи HTML (Hypertext Markup Language) сторінки, зображення, відео, аудіо та інші ресурси. Протокол також підтримує безпеку за допомогою шифрування SSL (Secure Sockets Layer) / TLS (Transport Layer Security), що дозволяє забезпечити конфіденційність та цілісність даних під час їх передачі.

Одним з недоліків REST архітектури є відсутність механізму для часткового повернення даних. При запиті певного ресурсу веб-сервер повертає весь ресурс цілком, в не залежності від того, чи потрібен нам весь ресурс чи лише певна його частина. Також REST не передбачає механізму для об’єднання запитів на отримання або створення декількох ресурсів одночасно. Для повернення декількох ресурсів необхідно виконати окремий запит на кожен ресурс відповідно. Для вирішення даних проблем, була створена мова запитів GraphQL.

GraphQL – це мова запитів та новий підхід до розробки API, який дозволяє розробникам ефективно керувати даними, які вони отримують від сервера. В якості протоколу для комунікації GraphQL використовує HTTP. Однією з ключових особливостей даної технології є можливість клієнтів запитувати лише ті дані, які їм потрібні для конкретного запиту. Наприклад, замість того, щоб отримувати всі дані про користувача, клієнт може запитати лише ім'я та електронну адресу. Крім того, GraphQL підтримує вкладеність запитів, що дозволяє отримувати пов'язані дані в одному запиті, що робить його більш

потужним та зручним у використанні. GraphQL підтримує 3 типи операцій, які описані в табл. 2.2.

Таблиця 2.2. Операції над даними в GraphQL

Операція	Опис
Мутація	Використовується для зміни даних на сервері. Наприклад, додавання нового користувача, оновлення даних або видалення об'єктів. Мутації виконуються за допомогою ключового слова "mutation" і можуть приймати аргументи. Після виконання мутації сервер повертає дані, що вибрані клієнтом.
Запит	Слугує для отримання даних з сервера. Клієнт може вказати, які саме дані він хоче отримати, і у якому форматі. Запити виконуються за допомогою ключового слова "query". Клієнт може вкладати запити один в одного для отримання пов'язаних даних.
Підписка	Дозволяє клієнтам підписуватися на події на сервері і отримувати оновлення в реальному часі. Наприклад, клієнт може підписатися на оновлення нових повідомлень у чаті. Підписки виконуються за допомогою ключового слова "subscription" і можуть використовувати WebSocket технологію для передачі даних.

Основні переваги GraphQL включають:

- Гнучкість: клієнти можуть запитувати тільки необхідні дані, що дозволяє ефективно використовувати ресурси сервера.
- Однорідність: всі дані повертаються у вигляді JSON об'єктів, що спрощує їх обробку.

- Автоматична документація: GraphQL має вбудований механізм для створення документації API, що полегшує розробку та розуміння API.

GraphQL використовується в багатьох веб-додатках та мобільних додатках для отримання та надсилання даних між клієнтом та сервером. Він дозволяє розробникам створювати потужні та ефективні API, що відповідають потребам сучасних додатків. Саме тому буде доцільним використати дану технологію для побудови API частини системи.

2.2.3 Вибір архітектури побудови серверного додатку

Вибір архітектури відіграє дуже важливу роль у подальшій розробці програмного забезпечення. Вони дозволяють розділити функціональність програми на логічні компоненти, що спрощує розробку, тестування та підтримку коду. Монолітна та мікросервісна архітектура – це два основні підходи до побудови програмних додатків [4].

Мікросервісна архітектура – це підхід до розробки програмного забезпечення, в якому додаток розбивається на невеликі автономні компоненти, які називаються мікросервісами. Кожен мікросервіс відповідає за виконання певної функціональності і може бути розгорнутий, масштабований і оновлений незалежно від інших мікросервісів.

Основні принципи мікросервісної архітектури включають:

- Розбиття на дрібні компоненти: додаток розбивається на невеликі мікросервіси, кожен з яких відповідає за конкретну функціональність.
- Незалежність: кожен мікросервіс може бути розгорнутий, масштабований і оновлений незалежно від інших.
- Комунікація через API: мікросервіси спілкуються між собою через API, що дозволяє їм працювати разом.
- Гнучкість та масштабованість: мікросервіси можуть бути масштабовані незалежно один від одного, що дозволяє підтримувати високу доступність системи.

- Легкість у впровадженні нового функціоналу: завдяки незалежності мікросервісів, нові функції можуть бути швидко впроваджені без зміни всього додатку.

Мікросервісна архітектура дозволяє розробникам швидше впроваджувати нові функції, підтримувати високу доступність та масштабованість системи, а також полегшує роботу над великими проектами завдяки розбиттю на дрібні компоненти. Однак вона також вимагає додаткових зусиль та навичок у керуванні мікросервісами та у забезпеченні їхньої взаємодії. Побудова додатку на основі даної архітектури займає велику кількість часу та ресурсів, і на початковому етапі розробки проєкту даний підхід частіше за все не виправдовує затрачених на нього ресурсів [5].

Протилежною архітектурою до мікросервісної є монолітна архітектура, яка являє собою підхід до розробки програмного забезпечення, де весь додаток будується як єдиний, нероздільний блок. У монолітній архітектурі весь код, логіка та функціонал програми знаходяться в одному монолітному застосунку. Це означає, що всі компоненти додатка, такі як бізнес-логіка, база даних та інші, розглядаються як частина одного цілого.

Основні характеристики монолітної архітектури включають в себе простоту розгортання та відсутність складності в управлінні додатку, а також легкий процес налагодження коду. Однак, з часом, коли додаток росте та розвивається, монолітна архітектура може почати проявляти свої недоліки, такі як складність у підтримці та розвитку, велика залежність компонентів один від одного, а також обмежені можливості масштабування.

Підсумовуючи вище зазначені плюси та мінуси кожного з підходів можна зробити висновок, що для системи оцінки психологічного стану доцільно використовувати саме монолітну архітектуру, принаймні на початковому етапі розробки системи.

Але постає питання, яким чином будувати монолітну систему? Для рішення цієї задачі існує інший тип архітектур, який пропонує підходи до організації коду в межах програмного додатку. Самими поширеними

архітектурами даного типу є Трирівнева, Onion, Clean та Vertical Slice архітектури.

Трирівнева архітектура (рис. 2.2) – це концепція проєктування програмного забезпечення, яка передбачає розділення програми на три основні рівні або шари [6]. Нижче наведений опис кожного з шарів:

1. Інтерфейс користувача (presentation layer) – це рівень, який відповідає за повернення інформації клієнту. Цей рівень також відповідає за обробку введених даних користувачем і передачу їх на логічний рівень.
2. Логічний рівень (business logic layer) – це рівень, де знаходиться логіка програми. Тут відбувається обробка даних, виконання операцій та прийняття рішень на основі цих даних. Цей рівень відповідає за виконання бізнес-логіки програми, тобто за те, що програма робить з даними.
3. Рівень доступу до даних (data access layer) – це рівень, який відповідає за зберігання та доступ до даних. Тут відбувається взаємодія з базою даних або іншими джерелами даних. Цей рівень відповідає за отримання, зберігання та оновлення даних, необхідних для роботи програми.

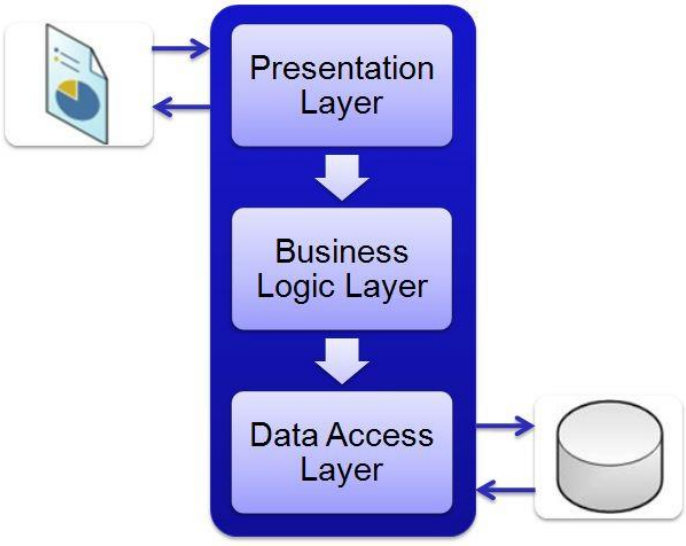

Рисунок 2.2. Трирівнева архітектура

Трирівнева архітектура дозволяє розділити програму на логічні частини, що спрощує розробку, тестування та підтримку програмного забезпечення.

Кожен рівень виконує свою функцію і може бути змінений або модифікований незалежно від інших рівнів, що полегшує роботу розробників. Але, в порівнянні з іншими архітектурами, вона має проблему з масштабованістю.

Onion архітектура (рис. 2.3) є дещо схожою на трирівневу, за виключенням організації шарів, яких може бути значно більше за 3. Шари розташовуються один над одним. Кожен шар може взаємодіяти з будь-яким іншим шаром, але має чітку ієрархію. Дана архітектура сприяє більшій гнучкості та легкості змін у системі [7].

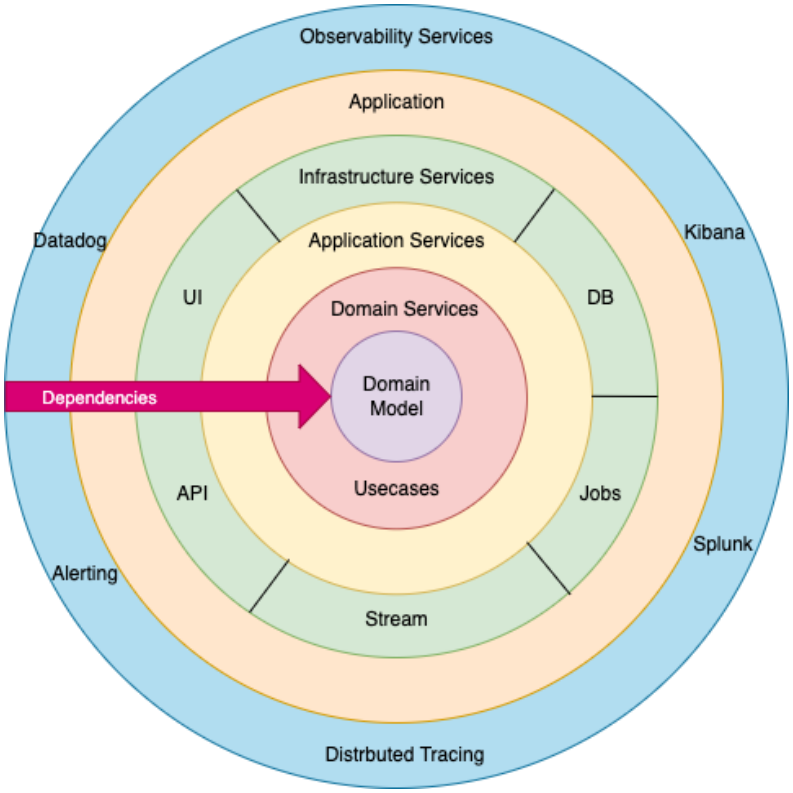

Рисунок 2.3. Onion архітектура

Clean архітектура (рис. 2.4) – це інший підхід до дизайну програмного забезпечення, який ставить основний акцент на розділення функціональності програми на чітко визначені шари та забезпечення високої рівня модульності і розширюваності системи. Цей підхід дозволяє створити систему, яка буде легко змінюватися, тестуватися та підтримуватися з мінімальними зусиллями. Дана архітектура базується на принципах SOLID, які допомагають створити гнучку та масштабовану архітектуру [8]. Основні компоненти Clean архітектури включають в себе:

- Entities: Це об'єкти, що представляють основні сутності вашої системи. Вони повинні бути незалежними від будь-яких інших шарів програми.
- Use cases: Це класи, які містять бізнес-логіку вашої програми. Вони виконують конкретні дії з використанням сутностей.
- Interface adapters: Цей шар містить всі необхідні адаптери для взаємодії між зовнішніми інтерфейсами (наприклад, UI, база даних) та внутрішніми компонентами програми.
- Frameworks and drivers: Це зовнішні компоненти, які взаємодіють з вашою програмою (наприклад, фреймворки, бібліотеки).

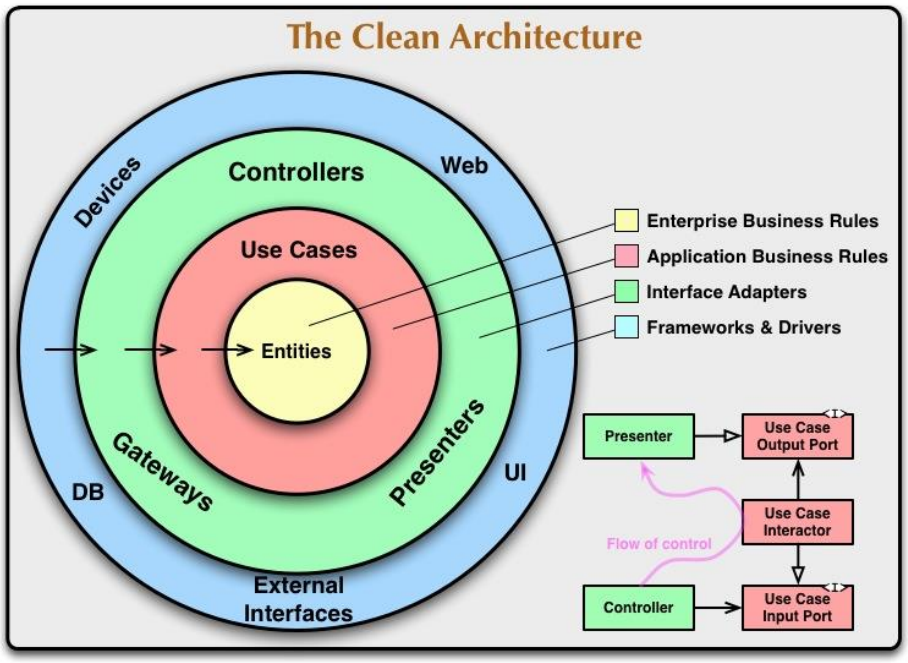

Рисунок 2.4. Clean архітектура

Clean архітектура дозволяє зберігати бізнес-логіку окремо від деталей реалізації, що робить систему більш модульною, стабільною та легкою для розширення. Крім того, він сприяє тестуванню коду, оскільки окремі компоненти можуть бути протестовані незалежно один від одного.

Vertical Slice (рис. 2.5) архітектура може розглядатись як альтернатива до Clean архітектури. Даний підхід до розробки програмного забезпечення полягає в тому, щоб розділити функціонал програми на вертикальні сегменти, які представляють повний стек функціоналу. Кожен вертикальний сегмент або

"вертикальний зріз" охоплює всі шари програми, включаючи інтерфейс взаємодії, бізнес-логіку та доступ до даних [9].

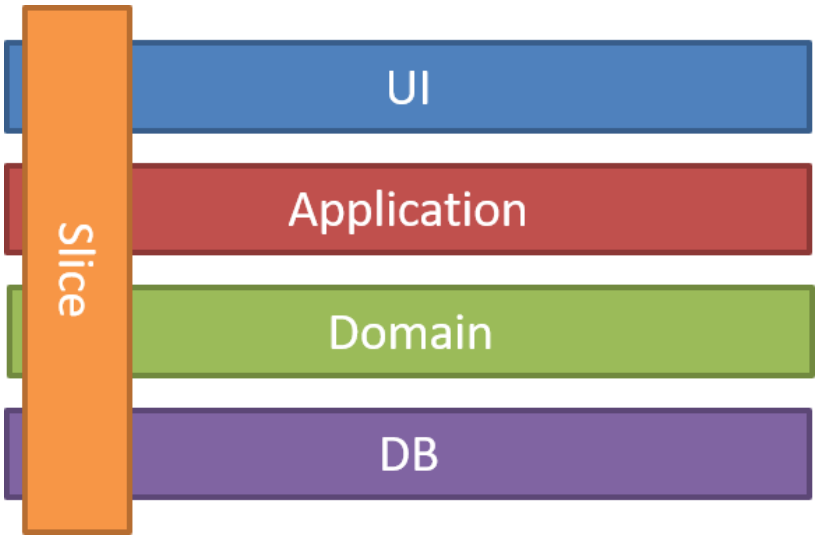

Рисунок 2.5. Vertical Slice архітектура

Основні принципи Vertical Slice архітектури включають:

- Розділення на вертикальні сегменти. Функціонал розбивається на невеликі вертикальні сегменти, які представляють повний стек функціоналу. Кожен вертикальний зріз відповідає за певну функцію або можливість програми.
- Самостійність. Кожен вертикальний зріз може працювати незалежно від інших, що сприяє зменшенню залежностей між компонентами програми.
- Інкапсуляція. Кожен вертикальний зріз має свою внутрішню структуру, яка приховує деталі реалізації від інших вертикальних зрізів.
- Швидка розробка нового функціоналу. Дана архітектура сприяє швидкому впровадженню нового функціоналу, оскільки розвивати і тестувати можна кожен вертикальний зріз окремо.
- Легкість розширення та змін. Завдяки розділенню на вертикальні сегменти, додавання нового функціоналу або зміни існуючого стає більш простим та менш ризикованим.

Vertical Slice архітектура дозволяє командам розробників працювати над різними вертикальними зрізами паралельно, що сприяє швидкій розробці та

поставці програмного забезпечення. Також цей підхід допомагає зменшити залежності між компонентами та полегшує тестування та супровід програми.

Отже, проаналізувавши всі вищезазначені архітектури, можна зробити висновок, що найбільш доцільно використовувати Clean або Vertical Slice архітектуру, так як вони більше підходять під проекти середнього та великого масштабів. Я вирішив зупинитися на Vertical Slice, так як з цією архітектурою швидше можна розробляти функціонал, а також з нею простіше працювати та масштабувати проєкт в перспективі.

2.3 Проєктування інфраструктури додатку

2.3.1 Огляд рішень для зберігання та обробки файлів

В будь якій системі є робота з файлами. Система оцінки психологічного стану – не виключення. Користувачі матимуть можливість налаштовувати власний профіль, в тому числі завантажувати аватар. Ці зображення необхідно десь зберігати, і при необхідності відображати у клієнтській частині додатку. Для рішення цієї задачі потрібно обрати підхід для збереження та обробки файлів.

Одним із самих примітивних підходів для зберігання файлів є локальне зберігання файлів, де запущений додаток. Цей підхід може бути зручним для деяких випадків, але має свої мінуси.

Одним з головних мінусів локального зберігання файлів є обмеженість простору на сервері. Якщо додаток обробляє велику кількість файлів або файли мають великий обсяг, це може призвести до переповнення диску і виникнення проблем з продуктивністю. Крім того, локальне зберігання файлів може бути менш безпечним, оскільки файли можуть бути доступні на сервері, що може призвести до можливості несанкціонованого доступу до них. Це особливо важливо в разі зберігання конфіденційної інформації. Також важливо враховувати, що локальне зберігання файлів може ускладнити масштабування додатку, оскільки розподілений доступ до файлів може бути складним у випадку розширення серверного парку.

Зберігання файлів у базі даних є ще альтернативним підходом, який може бути використаний у веб додатках. У цьому випадку, сам файл зберігається у вигляді бінарних даних у базі даних, а не на файловій системі сервера. Цей підхід також має свої переваги та недоліки.

Однією з переваг зберігання файлів у базі даних є простота управління даними. Файли зберігаються разом з іншими даними у базі даних, що дозволяє здійснювати операції з ними так само, як і з будь-якими іншими даними. Це також спрощує резервне копіювання та відновлення даних.

Однак, зберігання файлів у базі даних може призвести до збільшення розміру самої бази даних, що може вплинути на продуктивність та швидкість доступу до даних. Також, це може ускладнити масштабування системи, оскільки база даних може потребувати додаткових ресурсів для зберігання великих обсягів файлів. Крім того, зберігання файлів у базі даних може бути менш ефективним з точки зору оптимізації для великих файлів, оскільки це може призвести до збільшення обсягу даних, які потрібно передавати між базою даних та додатком.

Самим сучасним підходом до вирішення цієї задачі є використання хмарних сервісів. Самими популярними сервісами, які дозволяють зберігати та оброблювати файли, є S3 (Simple Storage Service) Bucket від хмарного провайдера AWS (Amazon Web Services) та Azure Blob Storage від хмарного провайдера Azure.

Однією з переваг використання даних сервісів для зберігання файлів є масштабованість та гнучкість. За допомогою них можна зберігати великі обсяги файлів у хмарному середовищі та масштабувати ресурси згідно з потребами додатку. Окрім того, сервіси хмарного зберігання можуть мати вбудовані механізми забезпечення безпеки, які допомагають захистити дані від несанкціонованого доступу та втрати. Також, вони можуть надавати різноманітні інструменти для резервного копіювання та відновлення даних.

Опираючись на огляд рішень для зберігання та обробки файлів, я вирішив делегувати дану функцію сторонньому сервісу, а саме обрав Azure Blob Storage.

Даний сервіс є дуже зручним у використанні та відповідає всім стандартам по захисту інформації.

2.3.2 Огляд рішень для відправки електронних листів

В наш час майже будь який додаток має можливість надсилання електронних листів користувачам. Це може бути як підтвердження електронної адреси при реєстрації облікового запису, відновлення паролю або інформування користувача про важливі події. Кожен додаток, який працює з електронною поштою, потребує ефективного механізму для відправлення листів.

Одним з можливих рішень для відправлення електронних листів є налаштування власного SMTP (Simple Mail Transfer Protocol) серверу. Цей підхід дозволяє повністю контролювати процес відправлення листів і забезпечує високий рівень безпеки. Однак, налаштування та підтримка власного SMTP серверу можуть бути складними завданнями для більшості користувачів. Вимагається належна експертиза з мережевої безпеки та адміністрування серверів.

Одним з недоліків використання власного SMTP серверу є його обмежена масштабованість. У випадку великого обсягу листів або потреби в надійному відправленні спам-фільтрам може бути складно обробити листи, відправлені з власного серверу. Крім того, існує ризик потрапляння в чорний список серверів через недостатню репутацію IP-адреси (Internet Protocol), що може призвести до тимчасового або постійного блокування відправлення листів.

Альтернативним популярним рішенням для відправлення електронних листів є використання сторонніх сервісів, таких як SendGrid. Ці сервіси надають інфраструктуру та технічну підтримку для відправлення листів, забезпечуючи високу швидкість доставки, надійність та безпеку. Крім того, вони забезпечують моніторинг відправлень, аналітику та можливість налаштування різних параметрів відправлення.

Одним з головних переваг використання сторонніх сервісів є їх масштабованість. Вони можуть легко впоратися з великим обсягом листів і забезпечити їх вчасну доставку. Крім того, вони допомагають уникнути проблем з чорними списками та підтримують високу репутацію IP-адрес. У той же час, використання сторонніх сервісів може вимагати додаткових витрат, особливо у випадку великого обсягу відправлень.

У підсумку, у якості підходу для відправки електронних листів я вирішив обрати хмарну платформу SendGrid, яка дозволяє розробникам і компаніям надсилати масові розсилки листів електронної пошти з високим рівнем доставки та безпеки. SendGrid надає API та інструменти для автоматизації процесу надсилання поштових повідомлень, включаючи можливість персоналізації контенту, відстеження метрик доставки та аналізу результатів кампаній. Ця платформа широко використовується бізнесами для маркетингових цілей, сповіщень про події, підтвердження реєстрації та інших цілей, пов'язаних з електронною поштою.

2.4 Моделювання

2.4.1 Моделювання процесу реєстрації користувача

Процес реєстрації користувача краще всього можна зобразити за допомогою UML (Unified Modeling Language) діаграми послідовності (рис. 2.6). На ній можна побачити користувача системи, клієнтський додаток, серверний додаток та сервіс для відправки електронних листів. За допомогою даної діаграми можна побачити взаємодію компонентів системи під час реєстрації користувача, так само як і взаємодію користувача з системою.

Реєстрація починається з введення користувачем необхідної для реєстрації інформації. Після цього, клієнтський додаток відправляє запит на реєстрацію користувача разом з уведеними даними. Серверний додаток надсилає запит на відправлення електронного листа з підтвердженням реєстрації сервісу для відправки електронних листів. Після того, як користувач отримав повідомлення

– він переходить за посиланням у листі для підтвердження реєстрації. Клієнтський додаток робить запит на підтвердження реєстрації разом з тимчасовим токеном, який міститься у посиланні в електронному листі. Сервер перевіряє токен та підтверджує реєстрацію, після чого клієнтський додаток перенаправляє користувача на сторінку з входом в особистий кабінет.

Рисунок 2.6. UML діаграма послідовності реєстрації користувача у системі

2.4.2 Побудова діаграми класів функціоналу перевірки тестування

При проєктуванні даного функціоналу я вирішив використати шаблон проєктування "Стратегія", який відноситься до поведінкових шаблонів і дозволяє визначити сімейство алгоритмів, розкласти їх у відповідні класи і зробити їх взаємозамінними. Цей шаблон дозволяє об'єкту вибирати один з алгоритмів (або декілька з них одразу) на льоту, залежно від ситуації. Даний шаблон визначає

інтерфейс або абстрактний клас, який представляє собою алгоритм. Потім створюються конкретні класи, які реалізують цей інтерфейс і включають в себе конкретні реалізації алгоритмів. Об'єкт контексту містить посилання на об'єкт стратегії і може викликати методи цього об'єкта для виконання певного алгоритму.

На діаграмі класів функціоналу перевірки тестування (рис. 2.7) можна побачити інтерфейс IPotentialProblemsDetection, який узагальнює стратегії по перевірці тестування. Кожен клас, який реалізує даний інтерфейс – являє собою стратегію перевірки тестування. Кожна стратегія має свій алгоритм перевірки на виявлення відповідного їй психологічного розладу. У ролі контексту виступає клас GeneralTestAnswersProcessor, який приймає список стратегій при створенні у якості параметру конструктора. Даний клас відповідає за перевірку загального тестування, використовуючи надані йому стратегії. Даний клас, в свою чергу, реалізує інтерфейс IGeneralTestAnswersProcessor.

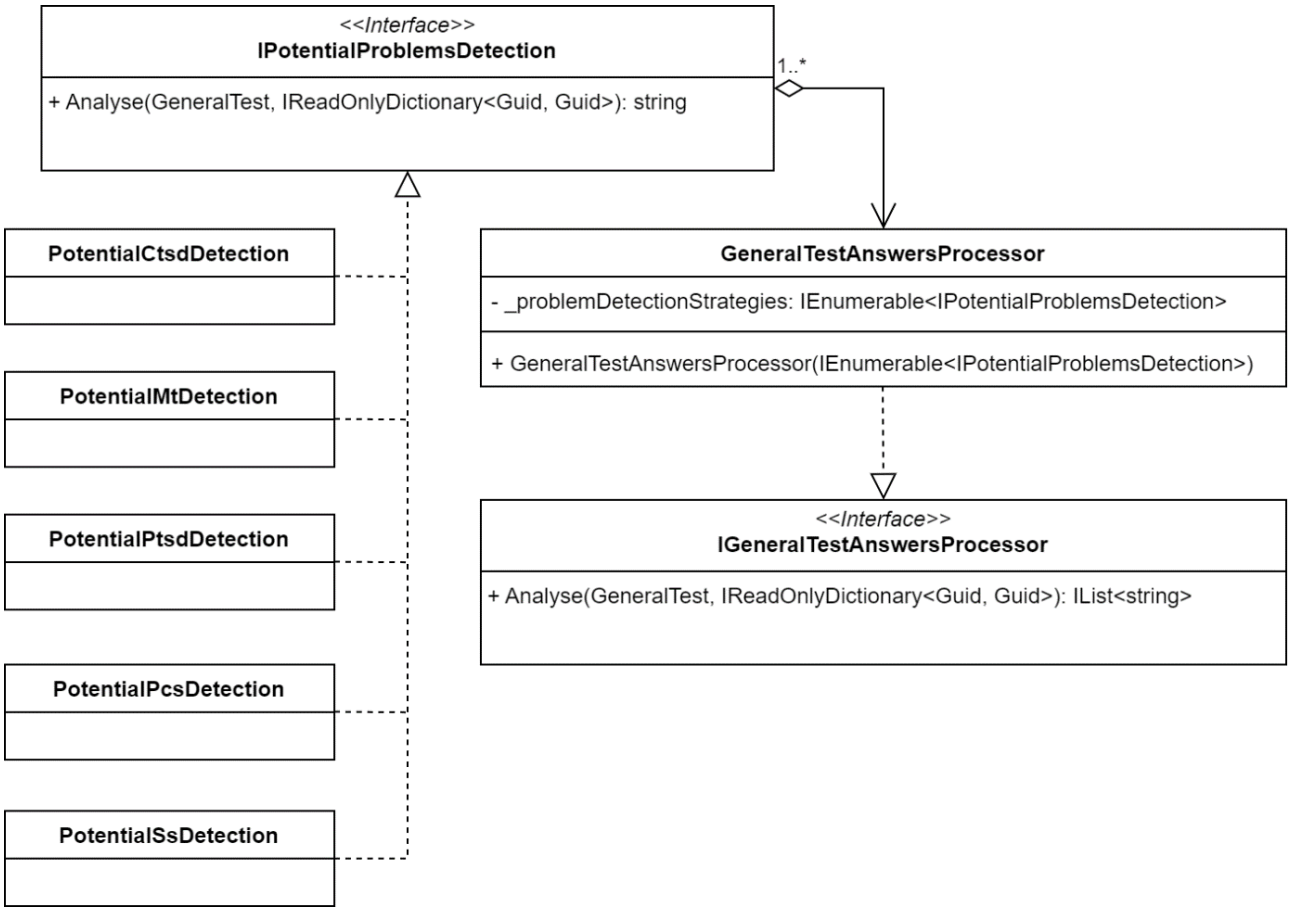

Рисунок 2.7. UML діаграма класів функціоналу перевірки тестування

3 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Створення та налаштування проєкту

3.1.1 Створення проєкту

Перш ніж почати розробку системи, необхідно створити проєкт. Є декілька варіантів створення проєкту: через консоль використовуючи CLI (Command Line Interface) або використовуючи графічний інтерфейс середовища розробки. Для пришвидшення процесу можна скористатись графічним інтерфейсом. Першим чином необхідно створити рішення проєкту (рис. 3.1), після чого необхідно створити 2 проєкти: WebAPI (рис. 3.2) та Application.

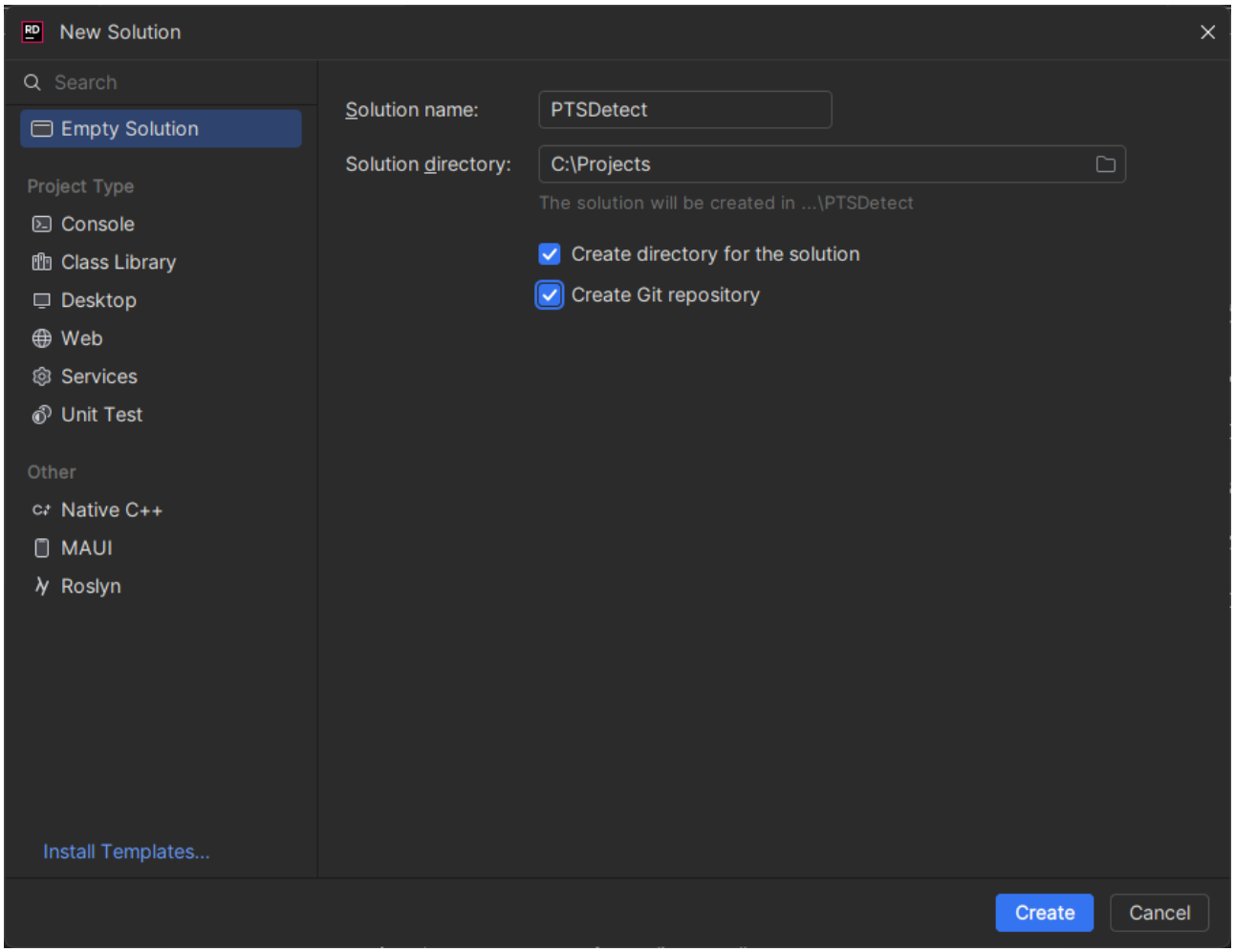

Рисунок 3.1. Створення пустого рішення проєкту

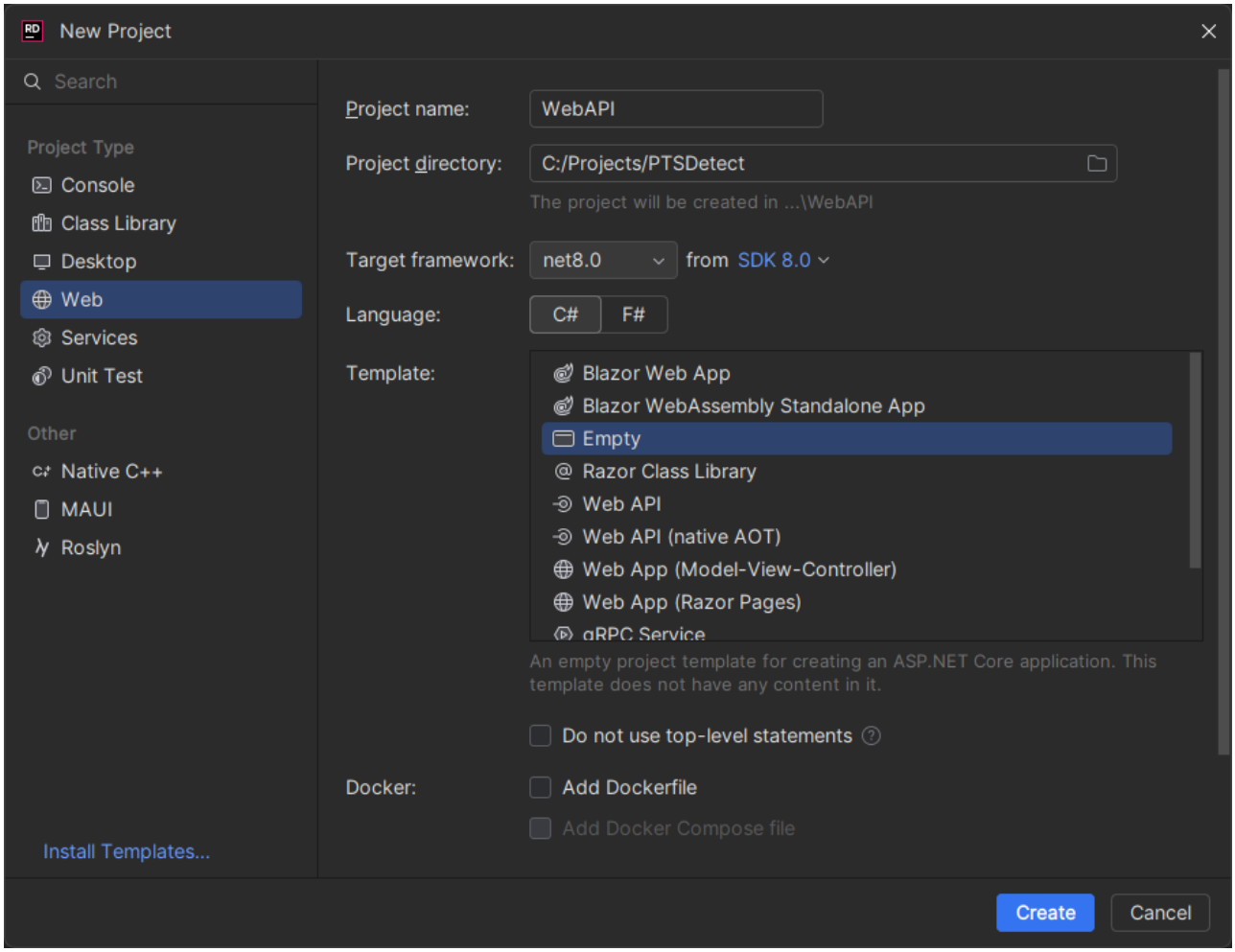

Рисунок 3.2. Створення пустого проєкту WebAPI

3.1.2 Встановлення залежностей

Наступним етапом необхідно встановити всі залежності проєкту використовуючи NuGet – менеджер пакетів для платформи розробки .NET, який дозволяє легко встановлювати, оновлювати та керувати залежностями різних бібліотек та компонентів в проектах .NET. NuGet дозволяє розробникам швидко і зручно використовувати сторонні бібліотеки та інші ресурси, що сприяє покращенню продуктивності та якості розробки програмного забезпечення на платформі .NET. Встановленні залежності проєкту зображені на рис. 3.3.

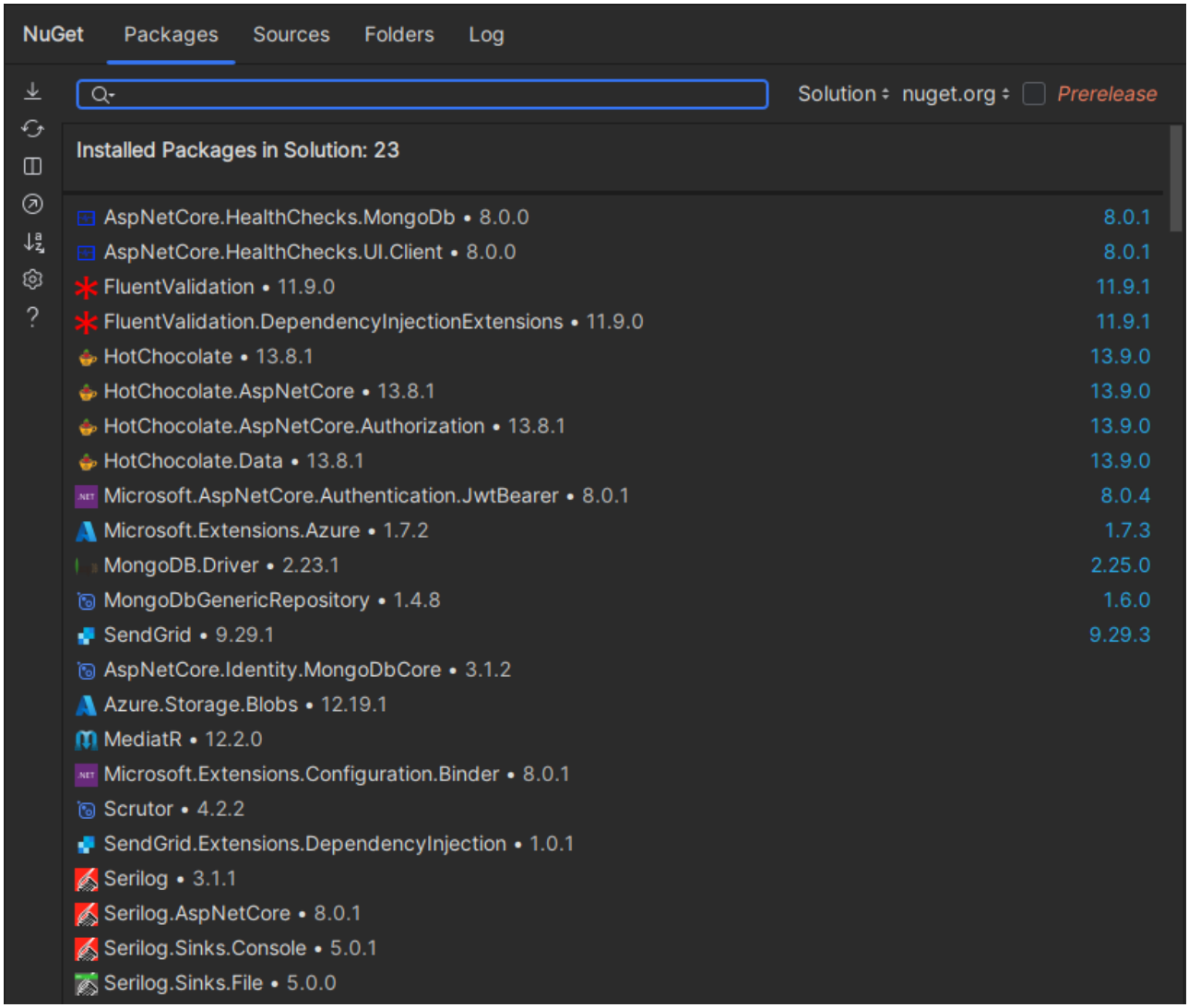

Рисунок 3.3. Встановленні залежності проєкту

3.1.3 Підключення та налаштування залежностей

Після встановлення залежностей їх необхідно правильно підключити та налаштувати згідно потреб системи. За це відповідає статичний клас DependencyInjection, який додає залежності до IoC (Inversion of Control) контейнеру. Завдяки цьому можна використовувати ті чи інші сервіси у проєкті використовуючи Dependency Injection – підхід до керування залежностями в програмуванні, який дозволяє вводити залежності в об'єкт зовні, замість того, щоб об'єкт створював їх самостійно. Це дозволяє полегшити тестування, збільшити перевикористання коду та зменшити зв'язаність між компонентами програми. В результаті, програма стає більш гнучкою та легше змінюється.

Нижче наведені основні методи класу DependencyInjection:

- AddAuthentication (рис. 3.4) – налаштовує аутентифікацію з використанням JWT (JSON Web Token) токена
- AddGeneralTestProcessor (рис. 3.5) – додає класи відповідальні за перевірку загального тестування
- AddSendGrid (рис. 3.6) – налаштовує сервіс відправки електронних листів SendGrid
- AddAzureServices (рис. 3.7) – налаштовує сервіси Azure, зокрема сервіс для зберігання файлів.

```
1 usage  Vlad Susidko
private static IServiceCollection AddAuthentication(this IServiceCollection services, JwtOptions authJwtOptions)
{
    services // IServiceCollection
        .AddAuthentication()
        .AddJwtBearer(x :JwtBearerOptions =>
        {
            x.TokenValidationParameters =
                new TokenValidationParameters
                {
                    ValidateIssuer = true,
                    ValidIssuer = authJwtOptions.Issuer,
                    ValidateAudience = true,
                    ValidAudience = authJwtOptions.Audience,
                    ValidateIssuerSigningKey = true,
                    IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(authJwtOptions.Secret)),
                    ValidateLifetime = true,
                    ClockSkew = authJwtOptions.ClockSkew
                };
        });

    return services;
}
```

Рисунок 3.4. Підключення та налаштування аутентифікації з використанням JWT токена

```
1 usage  Vlad Susidko
private static IServiceCollection AddGeneralTestProcessor(this IServiceCollection services)
{
    services.AddTransient<IPotentialProblemsDetection, PotentialCtsdDetection>();
    services.AddTransient<IPotentialProblemsDetection, PotentialMtDetection>();
    services.AddTransient<IPotentialProblemsDetection, PotentialPcsDetection>();
    services.AddTransient<IPotentialProblemsDetection, PotentialPtsdDetection>();
    services.AddTransient<IPotentialProblemsDetection, PotentialSsDetection>();

    services.AddTransient<IGeneralTestAnswersProcessor, GeneralTestAnswersProcessor>();

    return services;
}
```

Рисунок 3.5. Додання класів відповідальних за перевірку загального тестування

```
1 usage  Vlad Susidko
private static IServiceCollection AddSendGrid(this IServiceCollection services, IConfiguration configuration)
{
    var sendGridOptions = configuration // IConfiguration
        .GetRequiredSection(key: SendGridOptions.SectionName) // IConfigurationSection
        .Get<SendGridOptions>();

    if (sendGridOptions is null)
    {
        throw new InvalidOperationException(message: "Failed to obtain SendGrid options from configuration.");
    }

    services.AddSendGrid(configureOptions: x :SendGridClientOptions => { x.ApiKey = sendGridOptions.ApiKey; });

    return services;
}
```

Рисунок 3.6. Налаштування сервісу відправки електронних листів SendGrid

```
1 usage  Vlad Susidko
private static IServiceCollection AddAzureServices(this IServiceCollection services, IConfiguration configuration)
{
    var azureBlobStorageOptions = configuration // IConfiguration
        .GetRequiredSection(AzureStorageOptions.SectionName) // IConfigurationSection
        .Get<AzureStorageOptions>();

    if (azureBlobStorageOptions is null)
    {
        throw new InvalidOperationException(message: "Failed to obtain AzureBlobStorageOptions options from configuration.");
    }

    services.AddAzureClients(x :AzureClientFactoryBuilder => { x.AddBlobServiceClient(azureBlobStorageOptions.ConnectionString); });

    return services;
}
```

Рисунок 3.7. Налаштування сервісів Azure

3.1.4 Опис файлової структури проєкту

Згідно обраної архітектури Vertical Slice, проєкт складається з двох під-проєктів: WebAPI та Application.

WebAPI проєкт відповідає за реалізацію API шару системи. Він надає зовнішній доступ до функціоналу вашого додатку через HTTP протокол. У цьому проєкті реалізовані ендпоінти, які обробляють HTTP запити, точка входу у програму, статичні ресурси та конфігурація всього проєкту. Структура проєкту зображена на рис. 3.8 та описана в табл. 3.1.

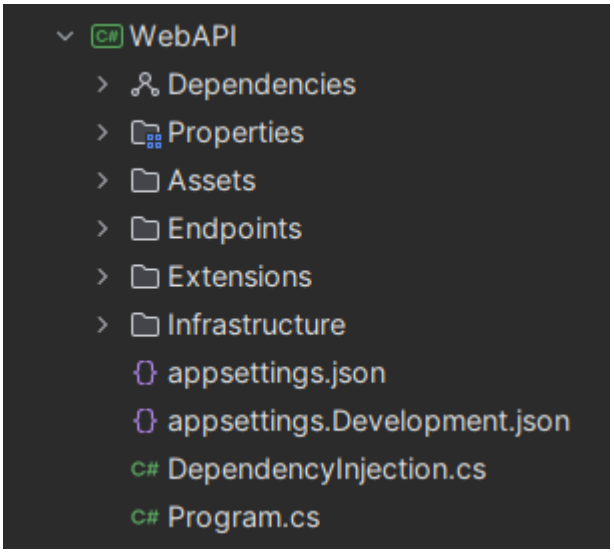

Рисунок 3.8. Файлова структура проєкту WebAPI

Таблиця 3.1. Опис файлової структури проєкту WebAPI

Структурна одиниця	Опис
Properties	Директорія, яка містить в собі параметри запуску проєкту.
Assets	Директорія зі статичними файлами.
Endpoints	Директорія, яка реалізує HTTP ендпоінт для GraphQL та ендпоінт перевірки здоров’я системи.

Продовження таблиці 3.1. Опис файлової структури проєкту WebAPI

Extensions	Директорія з реалізацією методів розширення для спрощення розробки.
Infrastructure	Директорія з абстрактними класами інфраструктури.
appsettings.json	JSON файл конфігурації проєкту.
appsettings.Development.json	JSON файл конфігурації проєкту який застосовується лише при запуску проєкту в режимі Development.
DependencyInjection.json	Код підключення та налаштування залежностей проєкту.
Program.cs	Код точки входу у програму.

Application проєкт містить бізнес-логіку додатку. Тут реалізовані основний функціонал системи, інфраструктура, обробники GraphQL запитів, валідатори, репозиторії, та інші компоненти, які відповідають за обробку даних та виконання бізнес-логіки. Структура проєкту зображена на рис. 3.9 та описана в табл. 3.2.

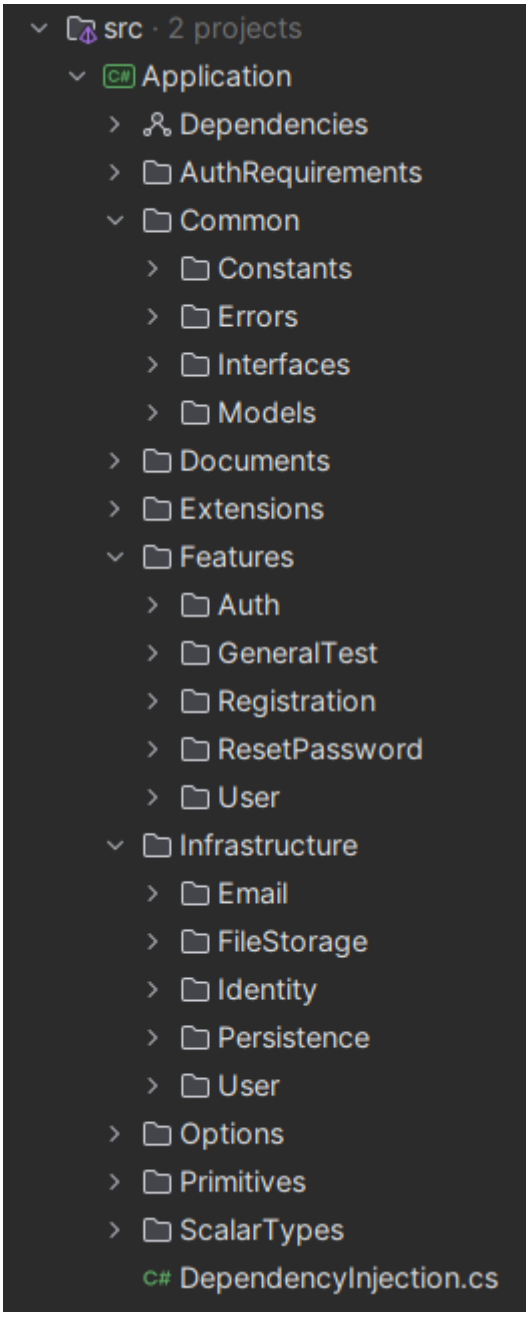

Рисунок 3.9. Файлова структура проєкту Application

Таблиця 3.2. Опис файлової структури проєкту Application

Структурна одиниця	Опис
AuthRequirements	Директорія з вимогами до авторизації, які перевіряються при кожній авторизації користувача.

Продовження таблиці 3.2. Опис файлової структури проєкту Application

Структурна одиниця	Опис
Common	Директорія з об’єктами, які використовуються по всьому проєкту. До них належать константи, помилки, інтерфейси та моделі.
Documents	Директорія, яка містить документи, які зберігаються в нереляційній базі даних, наприклад загальний тест або списки порад до психологічних розладів.
Extensions	Директорія з реалізацією методів розширення для спрощення розробки.
Features	Директорія, в якій знаходиться основний функціонал додатку, тобто аутентифікація, проходження загального тестування, реєстрація, відновлення втраченого паролю та функціонал кабінету користувача.
Infrastructure	Директорія, в якій знаходиться функціонал взаємодії з інфраструктурою, який використовується в основному функціоналі системи. До інфраструктури належить робота з сервісом відправки електронних листів, функціонал збереження та обробки файлів та взаємодія з базою даних.
Options	Директорія з класами, які використовуються для отримання конфігурації з appsettings.json файлів.

Продовження таблиці 3.2. Опис файлової структури проєкту Application

Структурна одиниця	Опис
Primitives	Директорія з примітивними об’єктами, які використовуються в основному функціоналі системи.
ScalarTypes	Директорія, в якій знаходяться додаткові типи даних GraphQL.
DependencyInjection	Код підключення та налаштування залежностей проєкту.

3.2 Розробка основного функціоналу серверної частини додатку

3.2.1 Реєстрація нового користувача

При реєстрації користувач має надати свою адресу електронної пошти та пароль. На стороні бекенду необхідно прийняти ці дані та перевірити їх на коректність.

Для реєстрації відповідного користувача необхідно створити мутацію – GraphQL операція, яка використовується для зміни даних на сервері. GraphQL мутації використовуються для виконання операцій запису, таких як створення, оновлення або видалення даних. Вони дозволяють клієнтському додатку взаємодіяти з сервером та змінювати дані в базі даних. Клієнт може викликати мутацію, передаючи необхідні параметри, і сервер відповідає результатом операції.

Мутація RegisterUser (додаток А) інкапсулює в собі логіку реєстрації нового користувача. Спочатку відбувається валідація даних, введених користувачем з клієнтської частини. Якщо дані введені у коректному форматі – відбувається реєстрація користувача за допомогою виклику метода RegisterUserAsync сервісу IdentityService. Якщо реєстрація успішна –

викликається метод генерації токену для підтвердження реєстрації GenerateEmailVerificationTokenAsync, генерується URL адреса яка містить в собі даний токен, та відправляється в електронному листі користувачу (рис. 3.10) у вигляді кнопки Finish registration. Для завершення реєстрації користувачу необхідно перейти за посиланням натиснувши на кнопку. Варто зазначити, що підтвердження реєстрації є необхідним етапом реєстрації щоб впевнитись, що користувач увів свою електронну адресу коректно.

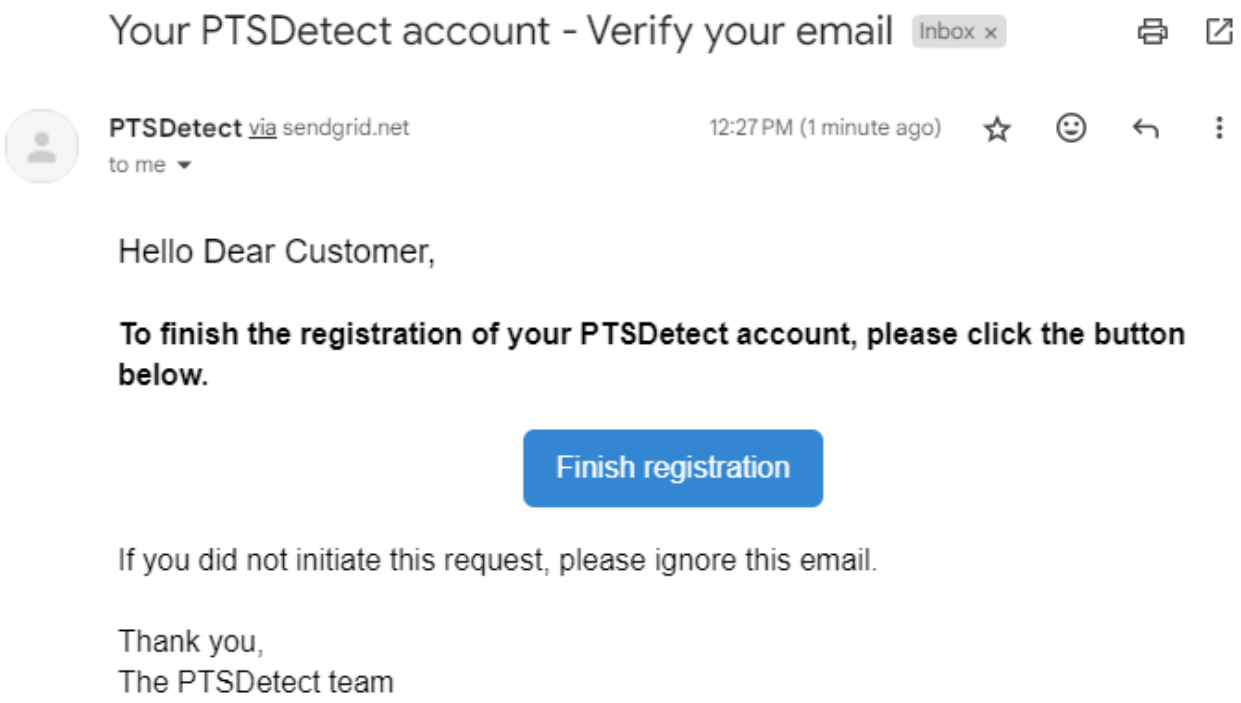

Рисунок 3.10. Електронний лист з підтвердженням реєстрації облікового запису

3.2.2 Аутентифікація користувача

Для того, щоб користувач мав можливість увійти у систему під своїм обліковим записом необхідно розробити аутентифікацію – процес перевірки і підтвердження ідентичності користувача, який намагається отримати доступ до певного ресурсу або сервісу. Цей процес включає в себе представлення вірних ідентифікаційних даних, таких як електронна адреса і пароль, для підтвердження права доступу до системи. Аутентифікація є важливим етапом в забезпеченні безпеки даних і захисту від несанкціонованого доступу.

За обробку логіки авторизації відповідає мутація Login (додаток Б). Спочатку виконується валідація введених даних користувачем, після чого викликається метод LoginByPasswordAsync для перевірки паролю користувача, і якщо пароль вірний – генерується пара токенів за допомогою метода GenerateTokenPairAsync: токен доступу (access token) та токен для оновлення токену доступу (refresh token). Наступним етапом токен для оновлення додається до кукі за допомогою метода розширення AddRefreshToken, а токен доступу повертається користувачу у якості відповіді на запит.

3.2.3 Проходження загального тестування

Для початку тестування клієнтська частина повинна отримати перелік запитань з можливими відповідями, щоб відобразити їх користувачу. За це відповідає запит GeneralTestQuestions (додаток В). Даний запит приймає від користувача код мови, і повертає список запитань у відповідній мові. Наразі підтримується 3 мови: українська, англійська та російська.

Після того, як користувач відповів на всі запитання, клієнт відправляє їх на сервер для перевірки та збереження результату викликаючи мутацію SubmitGeneralTestAnswers (додаток Г). На даному етапі виконується перевірка тестування за допомогою класу GeneralTestAnswersProcessor, який в свою чергу паралельно викликає метод Analyse у кожної стратегії для виявлення психологічних розладів. Паралельність необхідна для пришвидшення обробки результатів тестування.

Однією із стратегій є стратегія по виявленню пост-травматичного стресового розладу. Логіка роботи даної стратегії полягає в ітерації через групи запитань, які відносяться до пост-травматичного розладу, та перевірки запитань кожної з груп наступним чином: якщо хоча б одна відповідь на запитання має мітку пост-травматичного стресового розладу – група дала позитивний результат. Якщо кожна група дала позитивний результат – тоді користувач, скоріше за все, має даний розлад.

Після перевірки всіх можливих розладів – результат зберігається і клієнтській частині повертається ідентифікатор збереженого результату. Для відображення результату користувачу, клієнт повинен зробити запит на отримання результату тестування передавши наданий йому ідентифікатор результату.

3.3 Розробка інфраструктурної частини

3.3.1 Сервіс для відправки електронних листів

Під час проєктування інфраструктурної частини було прийнято рішення використовувати сервіс SendGrid для відправки електронних листів. Використання даного сервісу напряму в бізнес-логіці серверної частини додатку не є ефективним рішенням, оскільки бізнес-логіка почне залежати від стороннього сервісу, який складно буде замінити за потреби в майбутньому, також це ускладнить юніт-тестування системи.

Для вирішення даної проблеми доречно розробити абстракцію навколо сервісу для відправки електронних листів. Для початку треба виділити методи, які необхідні нашій системі для відправки листів, а саме:

- SendVerificationEmailAsync – метод відправки електронного листа для верифікації електронної адреси користувача. Використовується при реєстрації користувача у системі.
- SendResetPasswordEmailAsync – метод відправки електронного листа для відновлення користувачем втраченого паролю.

У ролі абстракції виступатиме інтерфейс IMailService, від якого буде залежати бізнес-логіка системи. Даний інтерфейс складається з вище описаних методів, які необхідні для функціонування системи. У якості реалізації інтерфейсу виступатиме клас SendGridMailService, який інкапсулює в собі взаємодію з сервісом SendGrid. Таким чином функціонал взаємодії з стороннім сервісом повністю відокремлений від основної чистини системи, і обмежується лише інтерфейсом IMailService. У майбутньому при потребі ми можемо замінити

сервіс SendGrid на будь який інший без зміни основної логіки системи. Для цього потрібно буде лише створити новий клас, який реалізує інтерфейс IMailService. UML діаграма класів сервісу відправки електронних листів зображена на рис. 3.11.

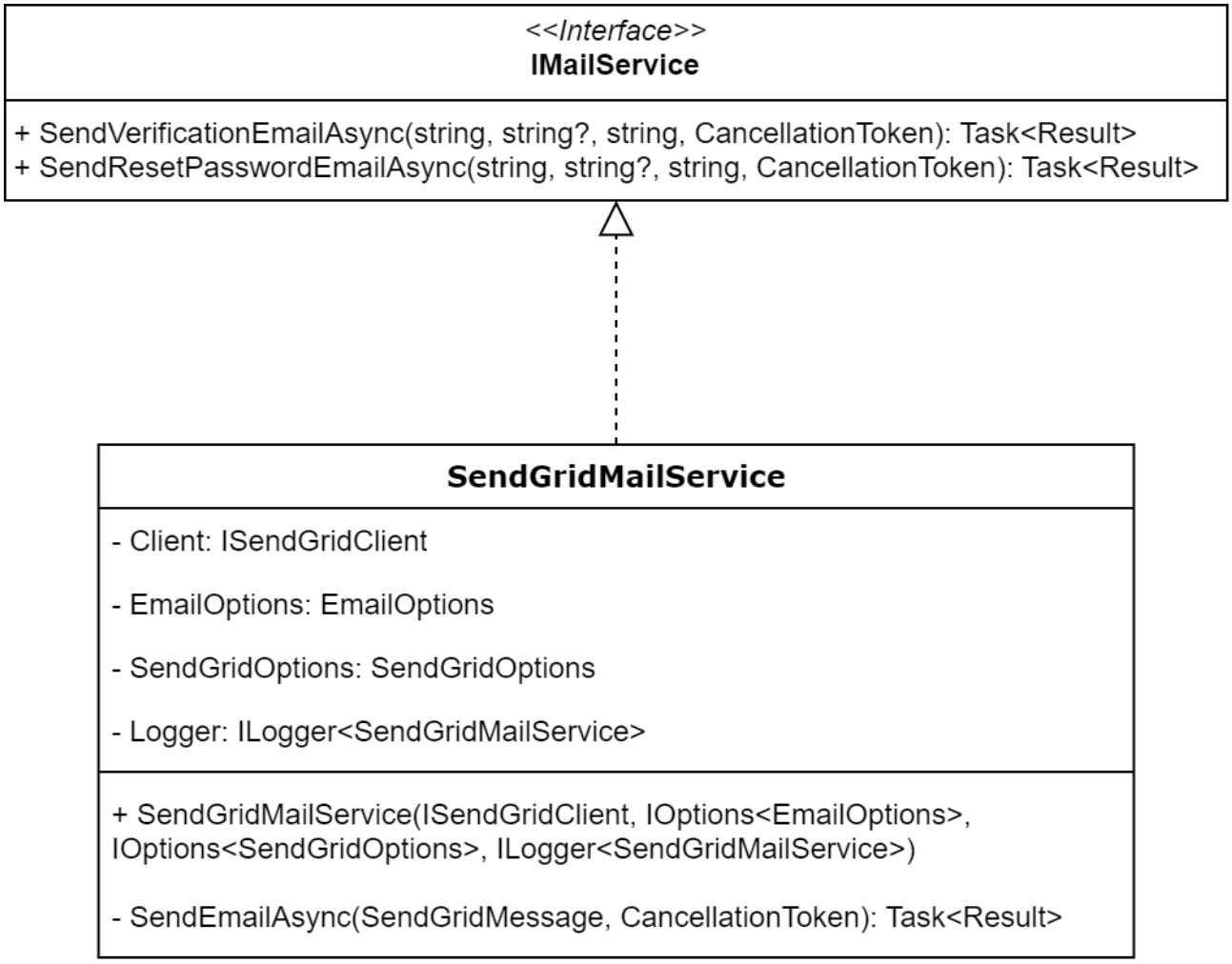

Рисунок 3.11. UML діаграма класів сервісу відправки електронних листів

3.3.2 Сервіс для зберігання та обробки файлів

Як і з сервісом відправки електронних листів, функціонал для зберігання та обробки файлів також має бути абстрагований від бізнес-логіки додатку з тих самих причин. Методи роботи з файлами, які необхідні системі:

- GeneratePreviewUrl – метод генерації підписаного посилання для перегляду файлу.

- GenerateUploadUrl – метод для генерації підписаного посилання для прямого завантаження файлу на сервіс.
- DeleteAsync – метод для видалення файлу.
- ExistsAsync – метод для перевірки існування файлу.

Дані методи доречно включити до інтерфейсу IFileService, який буде виступати у ролі абстракції функціоналу для взаємодії з файлами. У якості реалізації даного інтерфейсу виступає клас AzureBlobStorageService, який інкапсулює в собі взаємодію з хмарним сервісом Azure Blob Storage. Завдяки даній абстракції, бізнес-логіка системи не прив’язана до конкретного сервісу а залежить виключно від інтерфейсу IFileService, реалізацію якого можна змінити за потреби без необхідності вносити зміни до основного функціоналу системи. UML діаграма класів сервісу зберігання та обробки файлів зображена на рис. 3.12.

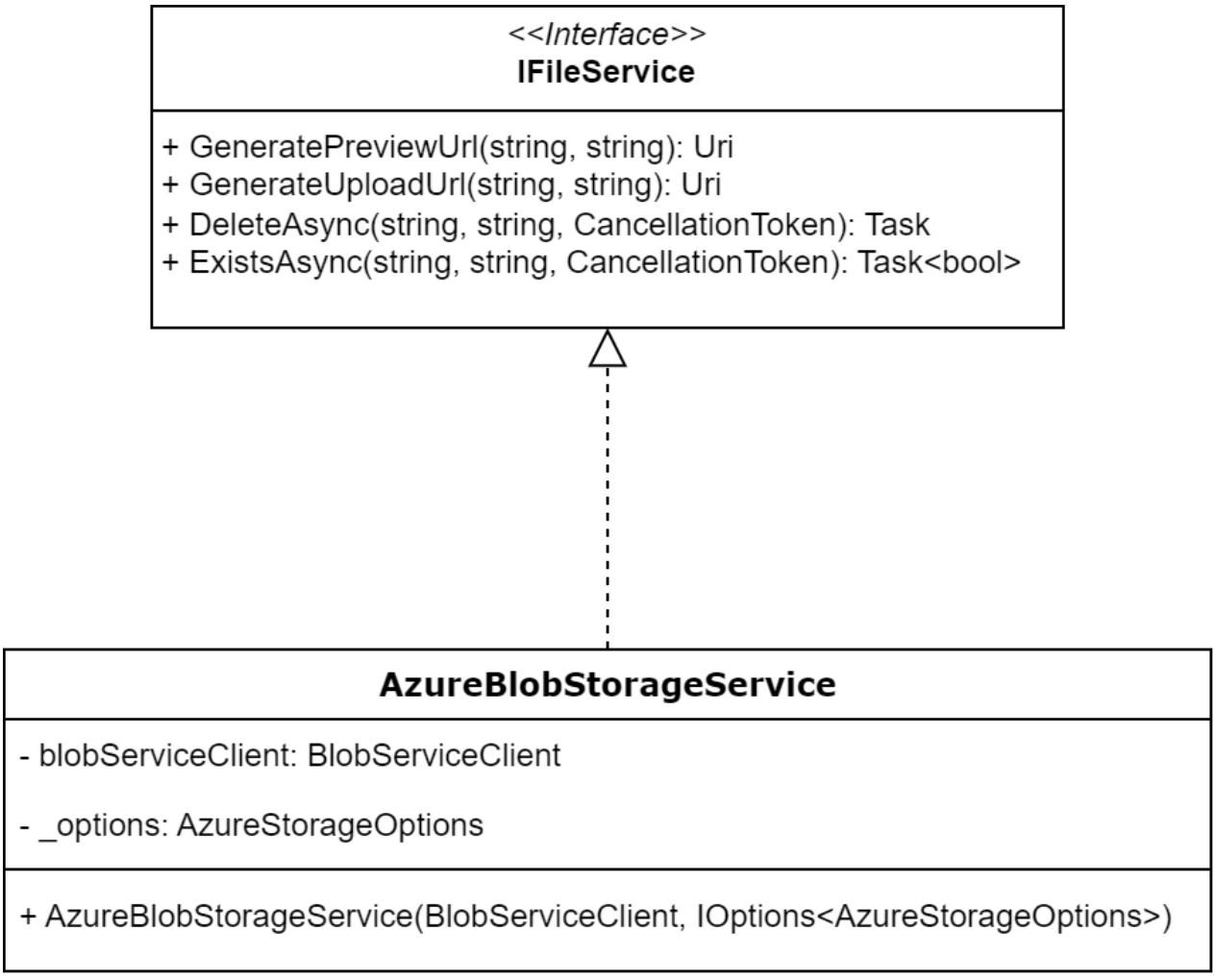

Рисунок 3.12. UML діаграма класів сервісу зберігання та обробки файлів

3.4 Налаштування конфігурації проєкту

Конфігурація проекту – це набір параметрів і налаштувань, які визначають поведінку програмного проекту. Ці параметри можуть включати в себе різноманітні дані, такі як рядки підключення до бази даних, URL-адреси зовнішніх сервісів, налаштування безпеки та інші дані.

У .NET проектах, таких як ASP.NET Core, для збереження конфігурації часто використовується файл appsettings.json. Цей файл містить пари ключ-значення, які визначають різні параметри проєкту. Наприклад, у файлі appsettings.json можуть бути вказані налаштування підключення до бази даних, налаштування кешування, налаштування логування тощо [10].

Використання appsettings.json дозволяє розділити конфігурацію від коду програми, що робить проект більш гнучким і легким у підтримці. Також цей файл може бути легко змінений без необхідності перекомпіляції програми, що дозволяє швидко змінювати налаштування в залежності від потреб проекту або середовища, в якому він виконується. Конфігурація проєкту за допомогою файлу appsettings.json наведена у додатку Д.

3.5 Контейнеризація додатку

Контейнеризація – це технологія, яка дозволяє упаковувати програмне забезпечення та всі його залежності в ізольовані контейнери, які можна запускати на будь-якій підтримуваній платформі без необхідності конфігурації середовища вручну. Контейнери дозволяють запускати програми у стандартизованому середовищі, що гарантує їх правильну роботу незалежно від операційної системи або обладнання, на якому вони виконуються. Крім того, контейнери є легкими та швидкими у використанні, оскільки вони використовують спільні ресурси з операційною системою хоста.

Основні переваги контейнеризації включають забезпечення ізоляції програм, спрощення розгортання та масштабування програм, підвищення

ефективності роботи розробників та адміністраторів, а також забезпечення безпеки та стабільності програмного забезпечення.

У якості інструментарію для контейнеризації додатку я обрав Docker – відкрите програмне забезпечення, яке дозволяє упаковувати, розгортати та управляти програмними додатками в контейнерах. Docker став популярним інструментом у сфері розробки програмного забезпечення, оскільки він спрощує процес розгортання та управління додатками, дозволяючи розробникам швидко та ефективно працювати з великими проектами.

Для контейнеризації додатку необхідно створити Dockerfile – текстовий файл, який містить інструкції для побудови Docker-контейнера. У Dockerfile вказуються кроки для створення середовища виконання програми, встановлення необхідних залежностей, копіювання файлів та налаштування контейнера. Після написання Dockerfile його можна використовувати для автоматизованої збірки Docker-імеджу, який можна запустити як контейнер. Dockerfile дозволяє створювати повністю відтворювані середовища для розгортання програм у контейнерах. Вихідний код даного файлу наведений у додатку Е.

Для додатків які складаються з декількох або більше контейнерів доречно використовувати Docker Compose – інструмент для оркестрації та управління багатьма контейнерами Docker одночасно. Він дозволяє описати конфігурацію вашого додатку у файлі YAML, включаючи інформацію про контейнери, мережі, образи, змінні середовища та інші параметри. Після цього з’являється можливість запуску всіх контейнерів за допомогою однієї команди, що значно спрощує розгортання та управління складними додатками, які складаються з декількох сервісів.

Docker Compose дозволяє визначити залежності між сервісами, налаштувати мережі для спілкування між контейнерами, визначити змінні середовища для кожного сервісу та багато іншого. Він робить розгортання і управління контейнерами більш простим та ефективним, дозволяючи швидко створювати та запускати складні додатки в ізольованому середовищі.

4 ВПРОВАДЖЕННЯ СИСТЕМИ В ЕКСПЛУАТАЦІЮ

4.1 Ергономічні вимоги до проєкту

Ергономіка – це наука, що вивчає вплив факторів робочого середовища на людину з метою оптимізації її працездатності, безпеки і комфорту. Впровадження ергономічних принципів у розробку програмного забезпечення сприяє створенню зручних, безпечних і ефективних систем для користувачів. Для автоматизованої системи оцінки психологічного стану людини ергономічні вимоги відіграють важливу роль, оскільки система повинна бути не лише функціональною, але й зручною для користування як для кінцевих користувачів, так і для адміністраторів та технічного персоналу [11].

Ергономіка, або ергономічні вимоги, стосується науки про оптимізацію взаємодії людини з системами, машинами та середовищем для підвищення ефективності, безпеки та комфорту. У контексті бек-енд розробки автоматизованої системи оцінки психологічного стану людини, ергономічні вимоги включають низку аспектів, які забезпечують зручність використання, надійність і продуктивність системи для кінцевих користувачів та адміністраторів.

Перш за все, важливо забезпечити високу продуктивність і надійність серверної частини системи. Це означає, що сервери повинні бути здатні обробляти великий обсяг запитів без затримок, що критично важливо для систем, які оцінюють психологічний стан користувачів у режимі реального часу. Використання сучасних технологій, таких як контейнеризація та мікросервіси, дозволяє розподілити навантаження між різними компонентами системи, що підвищує її стійкість і масштабованість.

Ще одним важливим аспектом є безпека даних. Оскільки система обробляє чутливу інформацію про психологічний стан користувачів, необхідно забезпечити високий рівень захисту даних. Це включає шифрування даних як у стані спокою, так і під час передачі, а також використання сучасних методів

аутентифікації та авторизації. Важливо також забезпечити регулярне оновлення системи безпеки для захисту від нових загроз.

Ергономічні вимоги також включають забезпечення зручності адміністрування системи. Інтерфейси адміністрування повинні бути інтуїтивно зрозумілими і дозволяти швидко виконувати необхідні операції, такі як моніторинг стану системи, управління користувачами і налаштування параметрів системи. Важливо, щоб адміністратори мали доступ до докладної документації, яка пояснює всі аспекти роботи з системою.

Крім того, система повинна бути легко інтегрованою з іншими системами і сервісами. Це передбачає використання стандартних протоколів і форматів даних, таких як REST API або GraphQL, що дозволяє легко підключати нові модулі або інтегрувати систему з існуючими рішеннями. Успішна інтеграція з іншими системами може значно підвищити ефективність роботи і забезпечити більш повну картину психологічного стану користувачів.

Важливо також врахувати аспекти масштабованості системи. Система повинна бути здатна обробляти зростаючий обсяг даних і запитів без зниження продуктивності. Це може бути досягнуто за рахунок використання горизонтального масштабування, коли нові сервери додаються до існуючої інфраструктури для розподілу навантаження. Використання хмарних сервісів також може допомогти забезпечити необхідну масштабованість і гнучкість системи.

Нарешті, важливо забезпечити зручність використання системи для кінцевих користувачів. Це включає забезпечення швидкого і точного аналізу даних, а також надання результатів у зручному і зрозумілому форматі. Використання алгоритмів машинного навчання і штучного інтелекту може допомогти забезпечити високу точність оцінки психологічного стану користувачів і надати більш персоналізовані рекомендації.

Таким чином, ергономічні вимоги до бек-енд розробки автоматизованої системи оцінки психологічного стану людини включають забезпечення високої продуктивності і надійності системи, захисту даних, зручності адміністрування,

інтеграції з іншими системами, масштабованості і зручності використання для кінцевих користувачів. Виконання цих вимог дозволить створити ефективну і зручну систему, яка буде відповідати потребам користувачів і забезпечить високу якість оцінки психологічного стану.

4.2 Огляд рішень для розгортання системи

Розгортання системи для оцінки психологічного стану потребує ретельного вибору інфраструктури, яка забезпечить її ефективне та безперебійне функціонування. Існує два основні підходи до розгортання таких систем: використання фізичних серверів та хмарних провайдерів. У цьому розділі розглянуті обидва підходи, їхні переваги та недоліки, що допоможе визначитися з оптимальним рішенням.

Фізичні сервери – це власні або орендовані сервери, які встановлюються у дата-центрі або власному офісі. Вони забезпечують повний контроль над обладнанням та програмним забезпеченням, що встановлюється на них. Розгортання програмного забезпечення на фізичних серверах має наступні переваги:

- Контроль та безпека. Використання фізичних серверів дозволяє повністю контролювати всі аспекти безпеки, включаючи фізичний доступ до серверів, мережеві налаштування та налаштування програмного забезпечення. Це особливо важливо для систем, які обробляють конфіденційну інформацію.
- Продуктивність. Фізичні сервери можуть забезпечити високу продуктивність завдяки можливості налаштування апаратного забезпечення під специфічні вимоги системи. Відсутність віртуалізації також може знизити затримки та підвищити ефективність використання ресурсів.
- Налаштовуваність. Можливість повного налаштування апаратного забезпечення та програмного забезпечення відповідно до потреб

системи. Це може включати використання спеціалізованих компонентів, таких як графічні процесори GPU (Graphics Processing Unit) для обробки великих обсягів даних.

Поміж тим, з серверами пов’язані й недоліки, а саме:

- Високі витрати. Вартість придбання, встановлення та обслуговування фізичних серверів може бути значною. Це включає витрати на апаратне забезпечення, електроенергію, охолодження, а також заробітну плату технічного персоналу.
- Масштабованість. Масштабування фізичних серверів може бути складним і дорогим процесом. Додавання нових серверів потребує часу на їх закупівлю, доставку, встановлення та налаштування.
- Обслуговування та підтримка. Підтримка фізичних серверів вимагає високого рівня технічних знань та постійної уваги. Це включає оновлення програмного забезпечення, управління безпекою та вирішення апаратних проблем.

Альтернативою фізичним серверам є використання послуг хмарних провайдерів, які пропонують послуги оренди віртуальних серверів з можливістю швидкого їх розгортання та налаштовування відповідно до потреб користувача. Вони забезпечують гнучкість та високу доступність завдяки своїй інфраструктурі, розташованій у різних дата-центрах по всьому світу.

Переваги хмарних провайдерів:

- Гнучкість та масштабованість. Хмарні провайдери дозволяють швидко масштабувати ресурси в залежності від потреб системи. Це може включати збільшення обчислювальної потужності, додавання нових серверів або використання додаткових сервісів, таких як бази даних або системи зберігання.
- Зниження початкових витрат: Оренда віртуальних серверів дозволяє уникнути значних початкових витрат на придбання обладнання.

Користувачі сплачують лише за використані ресурси, що може бути економічно вигідним, особливо для стартапів та малих підприємств.

- Доступність та надійність: Хмарні провайдери забезпечують високу доступність своїх сервісів завдяки розподіленій інфраструктурі та резервуванню даних. Це знижує ризики простоїв та втрати даних.
- Підтримка та обслуговування: Хмарні провайдери беруть на себе обслуговування інфраструктури, включаючи оновлення програмного забезпечення, управління безпекою та вирішення технічних проблем. Це дозволяє користувачам зосередитися на розробці та вдосконаленні своєї системи.

Недоліки хмарних провайдерів

- Контроль та безпека. Використання хмарних провайдерів може обмежувати контроль над безпекою та конфіденційністю даних. Хоча більшість провайдерів пропонують високий рівень безпеки, все ж таки існує ризик витоку даних або несанкціонованого доступу.
- Залежність від інтернет-з'єднання. Для доступу до хмарних сервісів необхідне стабільне та швидке інтернет-з'єднання. У випадку проблем з інтернетом, доступ до системи може бути ускладненим або неможливим.
- Вартість при довготривалому використанні. Хоча початкові витрати на використання хмарних сервісів можуть бути низькими, при довготривалому використанні вартість може зрости. Це особливо актуально для систем з високими вимогами до обчислювальних ресурсів.
- Обмежена налаштовуваність. Хоча хмарні провайдери пропонують широкий спектр налаштувань, можливості повної кастомізації апаратного забезпечення можуть бути обмеженими в порівнянні з фізичними серверами.

Обидва підходи до розгортання системи – використання фізичних серверів та хмарних провайдерів – мають свої переваги та недоліки. З огляду на систему оцінки психологічного стану стає зрозумілим, що доцільним є використання обладнання та послуг хмарних провайдерів, оскільки це дуже економить ресурси, а саме кошти та час на початковому етапі розвитку проєкту. Варто зазначити, що вибір хмарного провайдеру також відіграє важливу роль в провадженні системи в експлуатацію. На ринку існує багато провайдерів різного розміру та з різними послугами. Самими популярними провайдерами можна виділити AWS, Microsoft Azure та Google Cloud Platform (GCP), кожен з яких пропонує широкий спектр послуг для підтримки різноманітних потреб бізнесу та розробників.

AWS є лідером на ринку хмарних послуг. Відомий своїми масштабованими і надійними інфраструктурними рішеннями, AWS пропонує понад 200 послуг, включаючи обчислення, зберігання, бази даних, машинне навчання, аналітику та багато інших. Він відзначається своєю гнучкістю, багатством функцій і потужною мережею центрів обробки даних по всьому світу.

Microsoft Azure є другим за популярністю хмарним провайдером, що забезпечує широкий спектр послуг для підприємств усіх розмірів. Azure інтегрується з багатьма іншими продуктами Microsoft, такими як Office 365, Dynamics 365 та Windows Server, що робить його привабливим вибором для організацій, які вже використовують Microsoft продукти. Azure також пропонує рішення для обчислення, зберігання, баз даних, штучного інтелекту, машинного навчання і DevOps (Development and Operations).

GCP є третім великим провайдером на ринку хмарних послуг, зосереджуючись на високопродуктивних обчисленнях, великих даних і машинному навчанні. GCP використовує ту ж інфраструктуру, що і Google, забезпечуючи швидку і надійну продуктивність. GCP відзначається своїми інноваційними рішеннями в сфері аналізу даних та штучного інтелекту, а також пропонує повний набір хмарних послуг, включаючи обчислення, зберігання, бази даних і мережеві рішення.

З огляду системи оцінки психологічного стану кожен з цих провайдерів чудово впорається з розгортанням та хостингом системи. Але найбільш доцільним буде використати провайдер Microsoft Azure, так як дана платформа найкраще інтегрується з .NET платформою, яка використовується в серверній розробці системи.

4.3 Впровадження інструменту для оркестрування контейнерів

Сучасний підхід до розробки програмного забезпечення включає в себе контейнеризацію додатків для подальшого їх розгортання у будь-якому середовищі, яке підтримує дану технологію. Система оцінки психологічного стану не є виключенням – усі її додатки також контейнеризовані. Але постає питання – яким чином розгортати контейнери, керувати масштабуванням, перезапускати контейнери які зупинили своє виконання через помилку? Для вирішення даних проблем існують інструменти для оркестрування контейнерів

Інструменти для оркестрування контейнерами використовуються для автоматизації розгортання, керування, масштабування і роботи контейнеризованих застосунків. Вони дозволяють легко керувати великою кількістю контейнерів, розподілених по різних середовищах, і забезпечують необхідні функції для безперебійної роботи.

Основні функції оркестрації контейнерів:

- Автоматизація розгортання та оновлення: оркестратори автоматично розгортають та оновлюють контейнери відповідно до визначених політик.
- Управління конфігурацією: вони дозволяють зберігати конфігурацію застосунків окремо від коду і легко змінювати її.
- Масштабування: оркестратори дозволяють автоматично масштабувати кількість працюючих контейнерів в залежності від навантаження.

- Моніторинг та ведення логів: вони забезпечують засоби для моніторингу стану контейнерів та ведення логів для відстеження їхньої роботи.
- Балансування навантаження та високодоступність: оркестратори розподіляють навантаження між контейнерами та забезпечують високодоступність застосунків.
- Резервне копіювання та відновлення: вони дозволяють автоматично створювати резервні копії та відновлювати контейнери.

Популярні інструменти для оркестрації контейнерів:

- Kubernetes – найпопулярніший оркестратор, розроблений Google. Підтримує автоматичне масштабування, самовідновлення контейнерів, управління конфігураціями, та багато інших функцій. До переваг можна віднести велику спільноту, широку підтримку та активний розвиток, але даний оркестратор є відносно складним в налаштуванні та управлінні [12].
- Docker Swarm – інструмент оркестраї, який вбудований в Docker. Відносно простий у налаштуванні, використанні та інтеграції з Docker, але має менше функцій порівняно з Kubernetes.
- Apache Mesos – платформа для розподіленого управління ресурсами, яка підтримує оркестрацію контейнерів через Marathon або інші фреймворки. До переваг можна віднести підтримку різних видів навантажень та високу масштабованість. Водночас дана платформа є досить складною у налаштуванні.
- OpenShift – платформа на основі Kubernetes, яка надає додаткові інструменти для розробників і адміністраторів. Підтримує CI/CD, інтеграцію з корпоративними системами, але також досить складна для налаштування.

Для системи оцінки психологічного стану доцільно буде використати Kubernetes, оскільки це найбільш популярний та потужний оркестратор, який

дозволяє розробникам і адміністраторам керувати контейнерами на кластерах серверів, забезпечуючи гнучкість, масштабованість і високу доступність застосунків. Він має наступні переваги:

- Автоматизація: автоматизує більшість операцій з контейнерами, таких як розгортання, масштабування, відновлення та управління конфігураціями.
- Масштабованість: підтримує горизонтальне та вертикальне масштабування.
- Портативність: підтримка різних середовищ, включаючи локальні кластери та хмарні платформи.
- Стійкість до збоїв: автоматичне відновлення та реплікація забезпечують високу доступність.
- Ізоляція: простори імен та інші механізми забезпечують ізоляцію ресурсів.

Основні концепції Kubernetes:

1. Кластер – складається з одного або декількох вузлів (nodes). Є два типи вузлів: Master Node та Worker Nodes. Master Node контролює кластер, керує розгортаннями, масштабуванням та іншими операціями, в той час як Worker Nodes виконують робочі навантаження, тобто запускають контейнери.
2. Под (Pod) – найменша та найпростіша одиниця у Kubernetes. Він може містити один або декілька контейнерів, які мають спільну мережу та зберігання. Поди створюються, масштабуються та видаляються в міру необхідності.
3. Реплікаційні контролери та ReplicaSets. Реплікаційний контролер (ReplicationController) забезпечує бажану кількість реплік одного й того ж поду в кластері. ReplicaSet є покращеною версією реплікаційного контролера і використовується для підтримки заданої кількості подів.

4. Деплойменти (Deployments) – керують створенням та оновленням подів і ReplicaSets. Вони дозволяють визначати бажаний стан застосунку, автоматично змінювати кількість реплік та оновлювати поди без зупинки сервісу.
5. Сервіси (Services) – забезпечують постійний доступ до подів. Вони визначають логічний набір подів та політику доступу до них. Сервіси дозволяють іншим подам у кластері або зовнішнім користувачам звертатися до подів за стабільними DNS-іменами або IP-адресами.
6. Конфігурації та секрети (ConfigMaps and Secrets). ConfigMaps дозволяють зберігати конфігураційну інформацію, яка не є секретною, у вигляді ключ-значення. Secrets використовуються для зберігання конфіденційної інформації, такої як паролі, ключі доступу та сертифікати.
7. Простори імен (Namespaces). Простори імен дозволяють розділяти кластер на віртуальні кластери, що дозволяє ізолювати ресурси між різними командами або проєктами.

Kubernetes кластер можна розгортати як на власній інфраструктурі, так і на хмарній, але розгортання кластеру з нуля є досить складним завданням, та потребує відповідної кваліфікації. Оскільки було прийнято рішення використати хмарний провайдер Azure для розгортання системи, доцільним буде використати сервіс AKS (Azure Kubernetes Service), який спрощує розгортання, управління та масштабування контейнеризованих додатків за допомогою Kubernetes. AKS автоматизує багато аспектів управління кластером, зменшуючи навантаження на адміністратора та дозволяючи зосередитися на розробці та розгортанні додатків.

AKS інтегрується з іншими сервісами Azure, такими як Azure Active Directory для аутентифікації та Azure Monitor для моніторингу та логування. Це дозволяє створити цілісну та ефективну екосистему для розробки та експлуатації додатків.

Однією з ключових переваг AKS є автоматичне масштабування. Це включає горизонтальне масштабування подів у кластері та масштабування самих

вузлів кластера відповідно до навантаження. Крім того, AKS забезпечує автоматичне оновлення Kubernetes, що дозволяє зберігати кластер в актуальному стані без значних зусиль з боку адміністратора.

Безпека також є важливим аспектом AKS. Він підтримує мережеві політики, шифрування даних у спокої та під час передачі, а також інші засоби захисту для забезпечення безпечного функціонування контейнеризованих додатків.

Для керування ресурсами у Kubernetes кластері використовуються YAML-файли, які описують бажаний стан ресурсів. Команди kubectl дозволяють застосовувати ці конфігурації до кластера, моніторити та управляти станом ресурсів.

Конфігурація системи оцінки психологічного стану складається з двох YAML файлів:

- ptsdetect-api.yaml – містить налаштування сервісу з типом LoadBalancer та деплойменту, який підтримує 1 репліку ptsdetect-api поду. Вихідний код даного файлу наведений у додатку Є.
- ptsdetect-api-config.yaml – містить ConfigMap конфігурацію, яка застосовується у якості змінних оточення ptsdetect-api контейнера.

Для застосування конфігурації достатньо авторизуватися у кластері та виконати команду *kubectl apply -f k8s*, де k8s – директорія, в якій знаходяться вище зазначені файли конфігурації. Після цього, Kubernetes завантажить ptsdetect-api контейнер та створить сервіс та под з відповідним контейнером та конфігурацією.

4.4 Створення та налаштування Azure ресурсів

4.4.1 Створення групи ресурсів

Група ресурсів в Azure – це логічний контейнер, який використовується для управління ресурсами, такими як віртуальні машини, бази даних та мережеві інтерфейси, що працюють разом у рамках одного проекту. Вона дозволяє

організовувати та керувати ресурсами, спрощуючи їх розгортання, моніторинг і підтримку. Завдяки групам ресурсів можна застосовувати загальні політики, відстежувати витрати та забезпечувати безпеку для всіх ресурсів в межах одного контейнера.

Для створення групи ресурсів необхідно перейти в розділ «Resource groups», натиснути кнопку «Create» і заповнити 3 поля (рис. 4.1), після чого натисну кнопку «Review + create» і після перевірки ще раз «Create». Нижче наведений опис полів, які обов'язкові для заповнення:

- Subscription – підписка, в якій буде створена група.
- Resource group – назва групи ресурсів.
- Region – регіон, у якому будуть зберігатись метадані групи ресурсів.

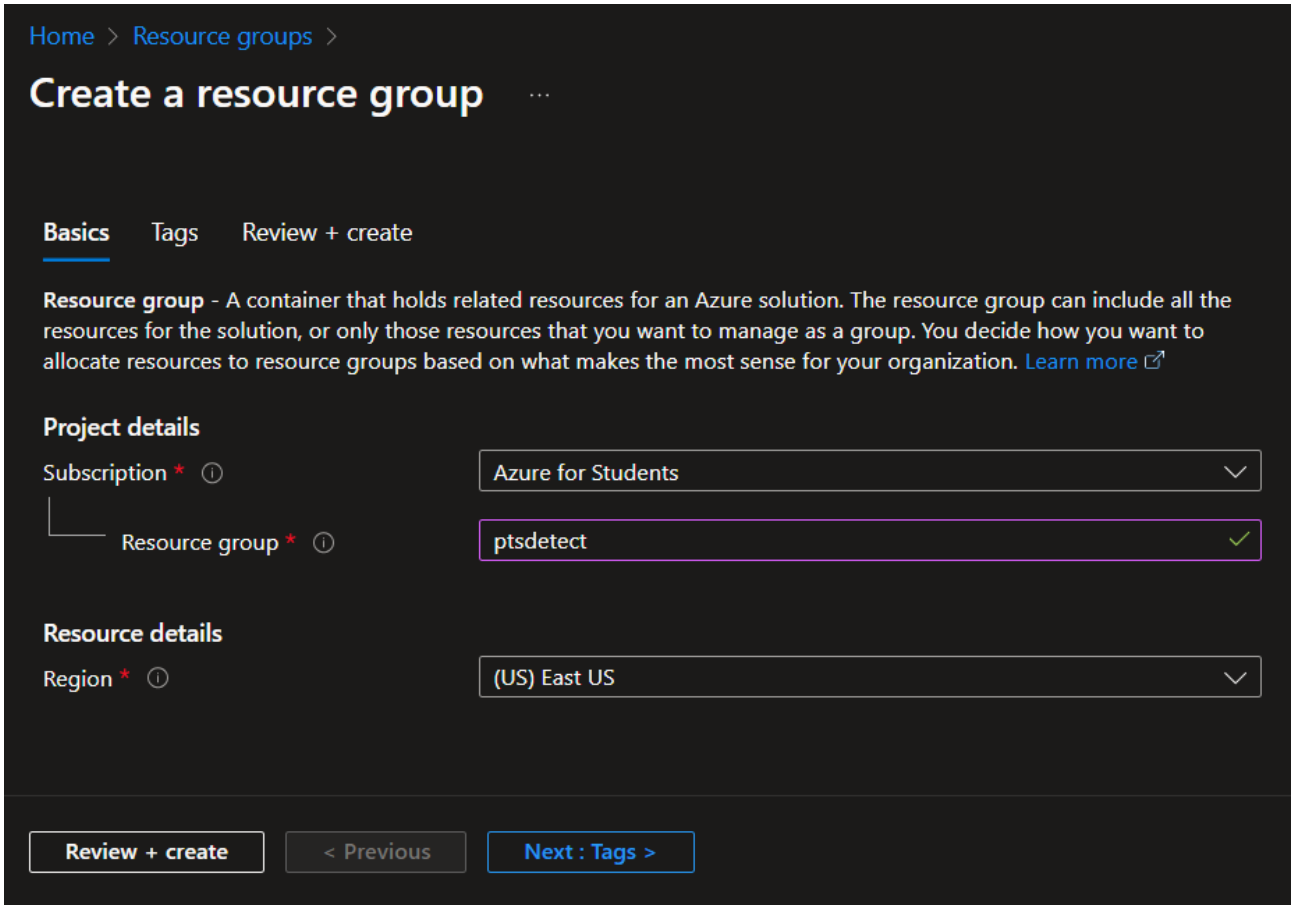

Рисунок 4.1. Створення групи ресурсів

4.4.2 Створення та налаштування облікового запису зберігання

Обліковий запис зберігання в Azure – це обліковий запис, який надає уніфікований доступ до послуг зберігання даних в Azure, таких як Blob Storage, File Storage, Queue Storage і Table Storage. Він служить контейнером для даних, дозволяючи зберігати та управляти великими обсягами структурованих і неструктурованих даних. Обліковий запис зберігання забезпечує масштабованість, високу доступність, безпеку даних та гнучкість у виборі рівня продуктивності та вартості.

Форма створення облікового запису зберігання (рис. 4.2) має значно більше налаштувань, порівняно з групою ресурсів. Нижче наведений опис основних полів:

- Subscription – підписка, в якій буде створений аккаунт.
- Resource group – група ресурсів, в якій буде створений обліковий запис.
- Storage account name – унікальна назва облікового запису зберігання.
- Region – регіон, у якому буде зберігатись інформація.
- Performance – рівень продуктивності зберігання. Оскільки система не потрибує низької затримки, доцільно вибрати рівень Standard.
- Redundancy – надлишковість інформації. Оскільки для функціонування системи доступність даних не є критичним параметром, доцільно вибрати локально-надлишкове зберігання, яке захищає інформацію від її втрати у разі виходу із ладу обладнання.

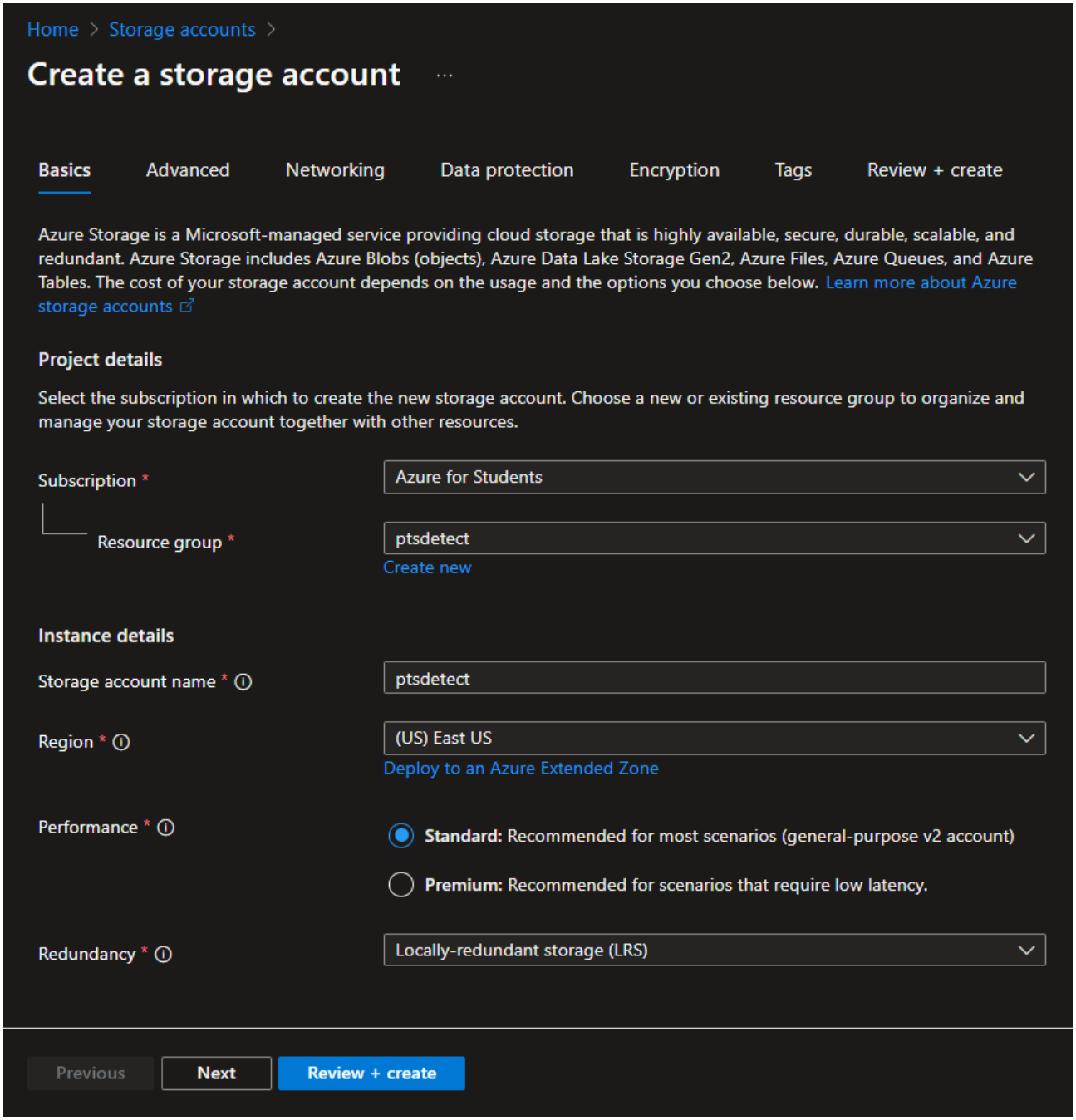

Рисунок 4.2. Створення облікового запису зберігання

4.4.3 Створення контейнеру для зберігання аватарів користувачів

Контейнер в обліковому записі зберігання Azure – це логічна одиниця, яка слугує для організації даних в хмарному сховищі. Він подібний до папки в файловій системі і використовується для зберігання набору об'єктів даних або blob-об'єктів. Кожен контейнер може містити неосяжну кількість цих об'єктів, що дозволяє ефективно організувати та керувати великими обсягами даних.

Контейнери створюються всередині облікового запису зберігання Azure і надають структуру для зберігання даних, дозволяючи розподіляти дані по логічним сегментам для полегшення доступу та керування. Кожен контейнер має унікальну назву в межах облікового запису і може бути налаштований з різними рівнями доступу, що дозволяє контролювати, хто і як може взаємодіяти з вмістом контейнера. Наприклад, можна налаштувати доступ для публічного читання або зробити дані доступними лише для певних користувачів або додатків.

Контейнери також забезпечують ізоляцію даних між різними програмами або користувачами, що є важливим для безпеки та організації інформації. Вони є основною складовою структури облікового запису зберігання і забезпечують гнучкість та масштабованість при роботі з великими обсягами даних у хмарі Azure.

Для коректного функціонування системи оцінки психологічного стану необхідно створити контейнер з назвою user-avatars. В даному контейнері будуть зберігатись аватари користувачів, які вони завантажують в налаштуваннях особистого профілю. Для створення контейнеру необхідно перейти до облікового запису зберігання, відкрити вкладку Containers та натиснути кнопку Create, у результаті чого відкриється діалогове вікно (рис. 4.3). Опис основних полів, які необхідно заповнити при створенні контейнеру:

- Name – ім’я контейнеру, в даному випадку user-avatars.
- Anonymous access level – рівень доступу до файлів у контейнері. В даному випадку згідно налаштувань облікового запису зберігання рівень автоматично виставлений в Private – тобто за замовчуванням файли не доступні в мережі інтернет. Для доступу до аватарів користувачів система генерує підписане посилання, яке дійсне протягом обмеженого періоду часу.

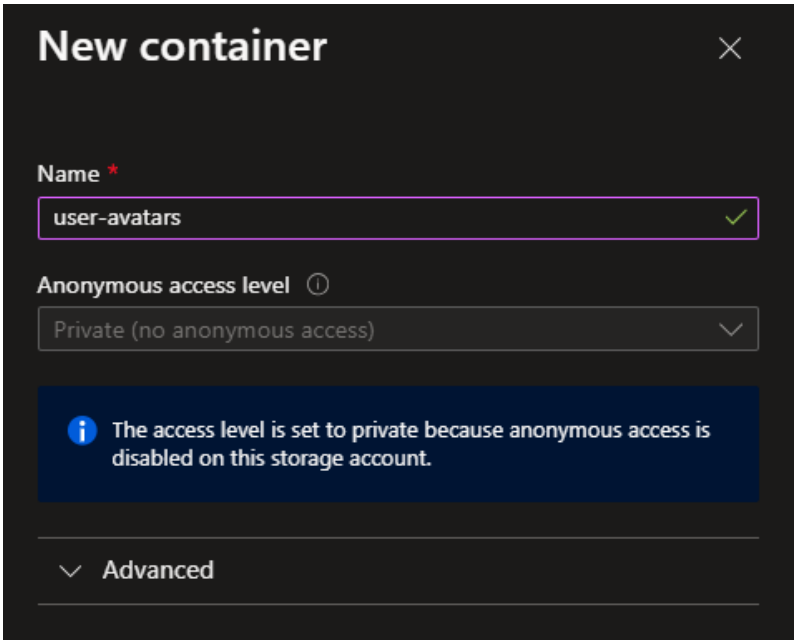

Рисунок 4.3. Створення контейнеру для зберігання аватарів користувачів

4.4.4 Створення та налаштування Kubernetes кластеру

Azure Kubernetes Service – це керований сервіс від компанії Microsoft для оркестрації контейнерів на платформі Azure. Він дозволяє розгортати, керувати та масштабувати контейнеризовані додатки за допомогою Kubernetes. AKS автоматизує багато аспектів керування Kubernetes-кластером, таких як оновлення, моніторинг і масштабування, що спрощує адміністрування та зменшує витрати на обслуговування.

Для створення Kubernetes кластеру необхідно відкрити розділ Kubernetes Services, натиску кнопку Create та вибрати Create a Kubernetes cluster, після чого відкриється відповідна сторінка з формою для створення кластеру (рис. 4.4). Нижче наведений опис основних полів форми:

- Subscription – підписка, в якій буде створений кластер.
- Resource group – група ресурсів, до якої буде доданий кластер.
- Cluster preset configuration – конфігурація кластера. Для потреб системи достатньо буде вибрати Dev/Test, оскільки дана конфігурація призначена для розробки та тестування і є безкоштовною.
- Kubernetes cluster name – назва кластеру.

- Region – регіон, у якому буде знаходитись кластер.
- Availability zones – фізично ізольовані локації в межах одного регіону Azure, між якими будуть розподілені вузли кластеру.
- AKS pricing tier – визначає рівень функціональності та ресурси, доступні кластеру, впливаючи на його вартість. Для системи достатнім є рівень Free.
- Kubernetes version – версія Kubernetes, яка буде використана для розгортання кластера.

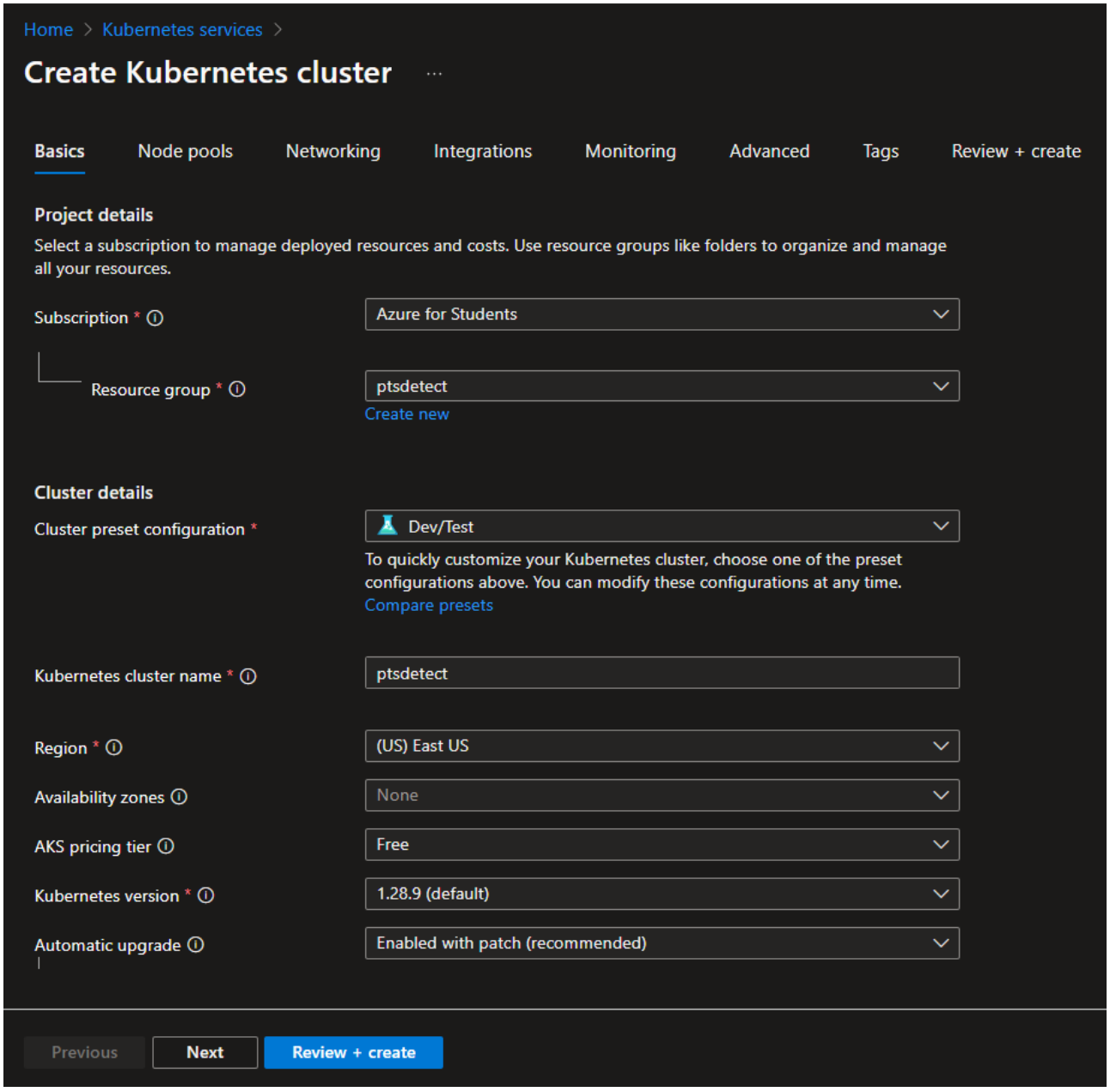

Рисунок 4.4. Створення Kubernetes кластеру

4.5 Автоматизація розгортання інфраструктури

Створення та налаштування хмарної інфраструктури вручну може здаватися простим рішенням для невеликих проектів або одноразових завдань, проте цей підхід має численні недоліки, особливо у довгостроковій перспективі. Використання Infrastructure as Code (IaC) рішень, таких як Terraform або Pulumi, має численні переваги, які роблять цей підхід більш ефективним та надійним [13].

Недоліки ручного налаштування хмарної інфраструктури:

1. Людські помилки: ручне налаштування завжди піддається ризику людських помилок. Неправильно введені параметри, пропущені кроки або некоректні налаштування можуть призвести до проблем, які важко виявити та виправити.
2. Непослідовність: ручний процес налаштування важко стандартизувати. Відсутність консистенції може призвести до того, що різні середовища (розробка, тестування, продакшн) будуть відрізнятися один від одного, що ускладнює відлагодження та підтримку.
3. Трудомісткість: ручне налаштування та управління інфраструктурою займає багато часу і зусиль. Це особливо помітно у великих проектах або у випадках частих змін.
4. Складність у масштабуванні: зі зростанням проекту і необхідністю додавати нові ресурси, ручне управління стає дедалі складнішим і менш ефективним.

Переваги Infrastructure as a Code:

1. Автоматизація: IaC дозволяє автоматизувати процес створення та налаштування інфраструктури, зменшуючи ризик людських помилок. Скрипти та конфігураційні файли автоматично налаштовують ресурси, забезпечуючи високу точність і повторюваність.
2. Відстежуваність і контроль версій: за допомогою систем контролю версій (наприклад, Git) можна відстежувати всі зміни в інфраструктурі,

повертатися до попередніх версій, аналізувати історію змін і контролювати процес оновлень.

3. Консистентність середовищ: IaC забезпечує однаковість налаштувань у різних середовищах (розробка, тестування, продакшн). Це зменшує кількість помилок, пов’язаних з відмінностями у конфігураціях.
4. Масштабованість: IaC дозволяє легко масштабувати інфраструктуру, додаючи нові ресурси або змінюючи існуючі конфігурації за допомогою скриптів. Це значно спрощує процес управління великими та складними системами.
5. Швидкість розгортання: автоматизоване налаштування дозволяє значно швидше розгортати нові середовища або вносити зміни в існуючі. Це підвищує ефективність розробки і скорочує час виходу нових функцій на ринок.
6. Прозорість і співпраця: IaC сприяє кращій співпраці між командами, оскільки всі конфігурації зберігаються у вигляді коду. Це забезпечує прозорість процесів та спрощує спільну роботу над проектом.
7. Інтеграція з CI/CD: IaC інтегрується з інструментами безперервної інтеграції та доставки (CI/CD), що дозволяє автоматизувати весь процес розгортання та тестування.

Найбільш популярними IaC інструментами є Terraform, Ansible та Pulumi, але вони мають різні підходи та вирішують різні задачі.

Terraform робить фокусом на декларативний підхід. Він дозволяє визначати інфраструктуру в конфігураційних файлах, які описують бажаний кінцевий стан ресурсів. Terraform має широку спільноту користувачів і великий набір провайдерів, що дозволяє працювати з різними хмарними сервісами та іншими інфраструктурними компонентами [14].

Ansible, з іншого боку, орієнтований на управління конфігураціями і автоматизацію розгортання. Він використовує YAML синтаксис для опису завдань і ролей, які визначають, як налаштовувати систему. Ansible також є

агентом, що означає, що він не потребує встановлення агентів на серверах, з якими працює, що робить його простішим у використанні в деяких сценаріях.

Pulumi використовує підхід інфраструктури як коду з можливістю програмувати інфраструктуру за допомогою загальноприйнятих мов програмування, таких як TypeScript, Python, Go або C#. Це дозволяє розробникам використовувати звичні інструменти та методи для роботи з інфраструктурою, що може спростити інтеграцію з існуючими процесами розробки.

Отже, найбільш доцільним є використання Terraform, оскільки це найбільш популярне рішення, має велику спільноту розробників, підтримує багато хмарних платформ та інших сервісів. Конфігурація складається з декількох файлів:

- versions.tf – файл з конфігурацією версії Terraform, провайдера Azure та налаштування Terraform Cloud.
- variables.tf – файл з описом змінних, значення яких має бути встановлено під час створення інфраструктури.
- main.tf – файл з основною конфігурацією інфраструктури (додаток Ж).
- output.tf – файл з оголошенням змінних, які будуть містити в собі інформацію про інфраструктуру після її створення.

Під час створення інфраструктури Terraform створює граф залежностей її компонентів. Після цього відбувається створення компонентів, починаючи з вершини (або вершин) даного графа, які не мають залежностей. Після цього відбувається процес створення залежних компонентів. Граф залежностей компонентів інфраструктури системи наведений на рис. 4.5.

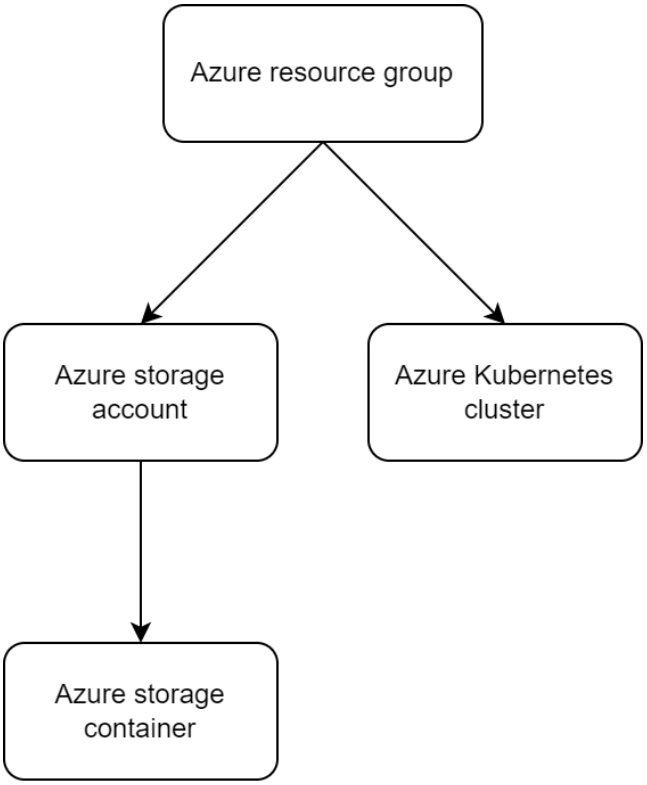

Рисунок 4.5. Граф залежностей компонентів інфраструктури системи

4.6 Налаштування CI/CD процесів

CI/CD, що розшифровується як Continuous Integration/Continuous Delivery (або Continuous Deployment), є набором практик, що застосовуються в програмній розробці для автоматизації та покращення процесів інтеграції та доставки програмного забезпечення [15].

Continuous Integration (CI) – це практика регулярного інтегрування змін у вихідний код до головної гілки репозиторію. Цей процес зазвичай автоматизується за допомогою інструментів, які збирають та тестують код кожного разу, коли в нього вносяться зміни. Мета CI полягає в тому, щоб виявляти та виправляти помилки на ранніх стадіях розробки, що допомагає уникнути великих проблем на етапі фінального тестування. Щоразу, коли розробник вносить зміни в код, вони інтегруються в основну гілку, і CI-система запускає серію автоматизованих тестів для перевірки того, чи нові зміни не призвели до регресії або нових помилок. Це дозволяє підтримувати високий рівень якості коду протягом всього процесу розробки.

Continuous Delivery (CD) – це практика, яка йде на крок далі від Continuous Integration. Вона передбачає автоматизацію процесу доставки програмного забезпечення до середовищ, таких як тестові, передпродуктивні та продуктивні системи. Мета CD полягає в тому, щоб забезпечити можливість випускати зміни у програмному забезпеченні швидко, надійно та з мінімальними зусиллями. Це означає, що після проходження всіх тестів та перевірок, зміни в коді можуть бути автоматично готові для розгортання у виробниче середовище. Хоча сама доставка може бути автоматизована, рішення про фактичне розгортання у виробниче середовище часто залишають за людиною для додаткового контролю.

Continuous Deployment – це ще один крок від Continuous Delivery, де кожна зміна, що проходить автоматизовані тести, автоматично розгортається у виробничому середовищі без втручання людини. Це забезпечує максимальну швидкість доставки змін до кінцевих користувачів, але вимагає високого рівня автоматизації та надійності тестів.

Інструменти CI/CD допомагають у реалізації цих практик, забезпечуючи автоматизацію збору, тестування та розгортання програмного забезпечення. Прикладами таких інструментів є Jenkins, GitHub Actions, GitLab CI, CircleCI, Travis CI та інші. Завдяки CI/CD процес розробки стає більш передбачуваним, швидким та менш ризикованим, що сприяє більш частим та надійним релізам програмного забезпечення.

У даному проєкті доцільно використати GitHub Actions – сервіс для автоматизації робочих процесів, інтегрований безпосередньо в платформу GitHub. Він дозволяє створювати та налаштовувати робочі процеси для автоматизації різних завдань, включаючи CI/CD, без необхідності використання зовнішніх інструментів.

Однією з головних переваг GitHub Actions є його глибока інтеграція з екосистемою GitHub, що дає можливість налаштувати автоматичне виконання дій у відповідь на події в репозиторії, такі як коміти, створення pull requests, відкриття issues та інше. Інтеграція з GitHub надає можливість використовувати

інформацію про репозиторій, гілки, коміти та інші метадані безпосередньо у робочих процесах [16].

GitHub Actions використовує YAML-файли для визначення робочих процесів, що робить їх зрозумілими та легкими для редагування. Даний сервіс надає можливість створювати складні багатоступеневі процеси з умовами, матрицями збірок та різними середовищами виконання. Однією із значних переваг GitHub Actions є можливість використання готових дій (actions) з GitHub Marketplace. Там можна знайти багато готових до використання дій, що значно спрощує налаштування типових завдань, таких як запуск тестів, розгортання на різні платформи, перевірка коду за допомогою статичного аналізу та багато іншого.

Continuous Integration даного проєкту (рис. 4.6) налаштований на запуск при створенні пул реквесту у гілку main та при кожному новому коміті, який потрапляє в даний пул реквест. CI складається з двох задач, які виконуються паралельно: побудови проєкту та валідації Terraform конфігурації. Перша необхідна для виявлення помилок у коді системи: якщо побудова проєкту завершилась з помилкою – це означає що код містить помилки, які необхідно виправити перед зливанням пул реквесту. Друга задача необхідна для перевірки синтаксису Terraform конфігурації та її валідації на коректність.

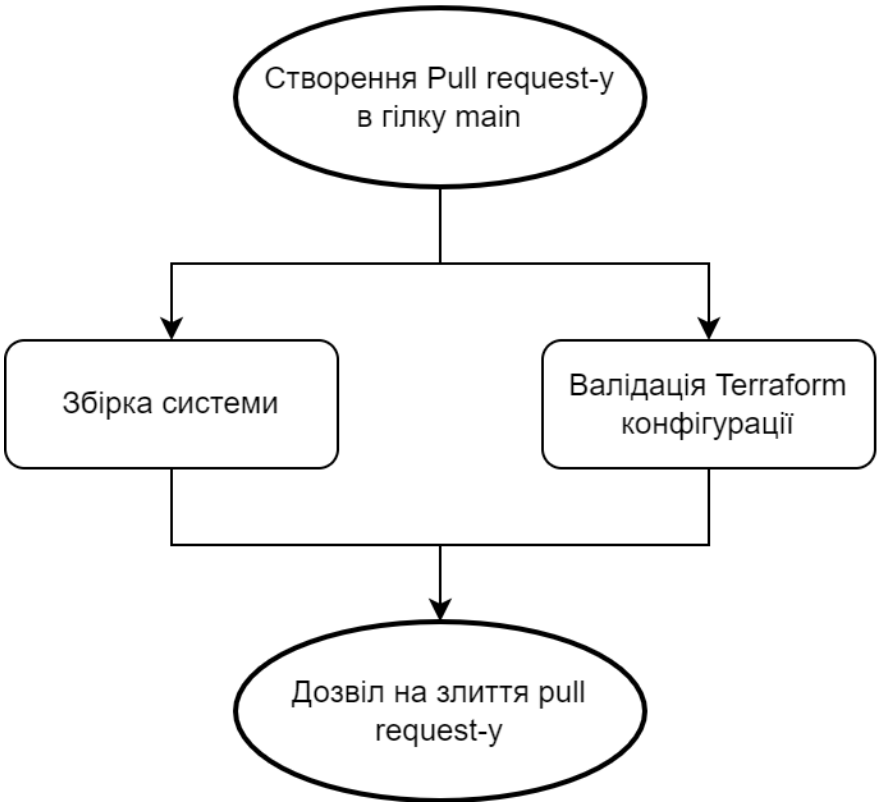

Рисунок 4.6. Блок схема Continuous Integration процесу

Налаштування репозиторію проєкту мають захист гілки main як головної гілки проєкту. Розробник не має можливості напряму пушити коміти до цієї гілки. Він може лише створити пул реквест. Але і на пул реквест також є обмеження – його можна злити в основну гілку лише при наступних двох умовах: всі зауваження від інших розробників повинні бути вирішеними, і CI процеси повинні відпрацювати без помилок. Якщо дані критерії виконані – розробник має можливість злити пул реквест в основну гілку main.

Для запуску Continuous Deployment процесу (рис. 4.7) необхідно створити тег (або реліз в GitHub) на гілці main. Сам процес складається з наступних етапів:

1. Збірка Docker образу та завантаження його у Docker Hub репозиторій (додаток З).
2. Розгортання інфраструктури за допомогою Terraform.
3. Розгортання системи на Kubernetes кластері.

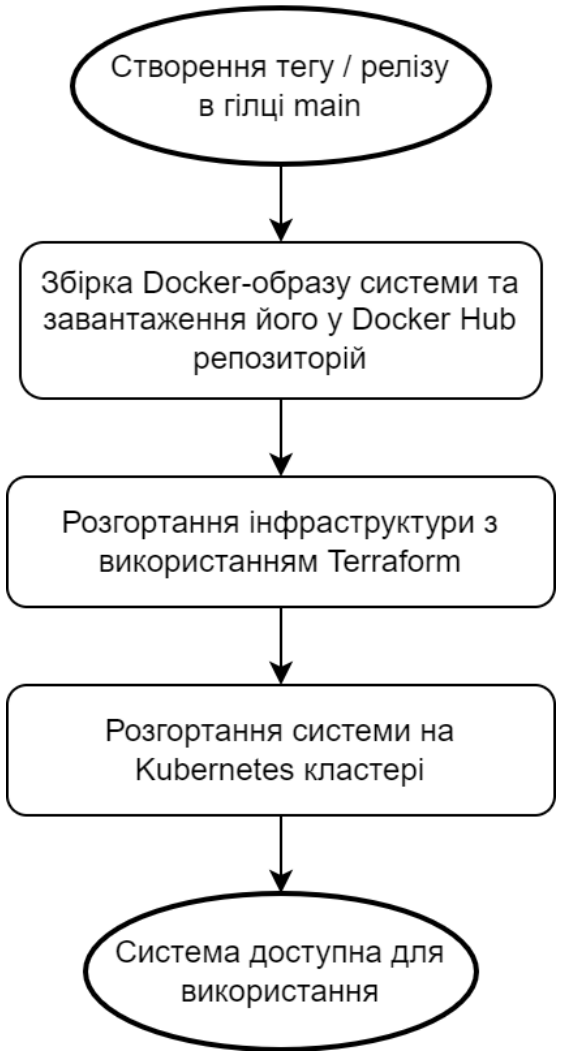

Рисунок 4.7. Блок схема Continuous Deployment процесу

Після завершення Continuous Deployment процесу API системи доступний за адресою створеного ptsdetect-api-service сервісу в Kubernetes кластері.

ВИСНОВОК

У дипломній роботі на тему "Бек-енд розробка автоматизованої системи оцінки психологічного стану" було реалізовано комплексний підхід до створення інноваційного програмного забезпечення, яке надає можливість своєчасно діагностувати та оцінювати психологічний стан користувачів. Даний висновок підсумовує основні результати дослідження та розробки, а також перспективи подальшого вдосконалення системи.

Проведений аналіз предметної області та постановка задачі дозволили чітко окреслити основні вимоги до розробки системи, а також визначити її структуру та функціональні можливості. Важливим аспектом роботи було дослідження існуючих рішень у сфері автоматизованої оцінки психічного стану, що дало змогу визначити переваги та недоліки сучасних технологій, а також обрати оптимальний стек технологій для реалізації проєкту.

Проєктування програмного забезпечення включало визначення функціональних та нефункціональних вимог, а також вибір архітектурного стилю, що забезпечує високу масштабованість та надійність системи. Особливу увагу було приділено проектуванню інфраструктури додатку та моделюванню основних процесів системи, що включає реєстрацію користувача, авторизацію, проходження тестування та надання результатів.

Реалізація системи була проведена з використанням сучасних технологій, що забезпечують високу продуктивність та надійність роботи серверної частини. Особливу увагу було приділено розробці основного функціоналу, що включає сервіс для зберігання та обробки файлів, а також сервіс відправки електронних листів. Налаштування конфігурації проекту та контейнеризація додатку з використанням платформи Docker дозволили створити гнучку та портативну систему, яка може бути легко розгорнута в різних середовищах, включаючи локальні та хмарні платформи.

Впровадження системи в експлуатацію включало детальний опис ергономічних вимог до проєкту, огляд рішень для розгортання системи та

обґрунтування вибору хмарного провайдера Microsoft Azure. Також було реалізовано автоматизовану інфраструктуру з використанням Infrastructure as Code рішення Terraform та налаштовані CI/CD процеси проєкту з використанням платформи GitHub Actions.

У ході розробки було досягнуто основної мети дослідження – створення бек-енд системи оцінки психологічного стану людей, яка дозволяє своєчасно виявляти психічні порушення та надавати рекомендації щодо їх усунення. Дана система значно полегшує процес діагностики, робить його більш доступним та збільшує вірогідність одужання пацієнтів.

Подальший розвиток проекту може включати розширення функціональності системи, інтеграцію з іншими медичними сервісами та розробку додаткових модулів для детальнішого аналізу психічного стану. Також важливим аспектом є проведення додаткових досліджень з метою вдосконалення алгоритмів оцінки та рекомендацій.

Таким чином, результати даної дипломної роботи мають значний практичний внесок у розвиток систем автоматизованої оцінки психічного стану та можуть бути використані як основа для подальших досліджень та розробок у цій галузі.

Список використаних джерел

1. What is Client-Server Architecture? Everything You Should Know, John T., 2023 [Електронний ресурс] // Режим доступу: https://www.simplilearn.com/what-is-client-server-architecture-article
2. What is a RESTful API? A Detailed Look, Alexander O., 2023 [Електронний ресурс] // Режим доступу: https://medium.com/@AlexanderObregon/what-is-a-restful-api-a-detailed-look-2b7b182e1def
3. An overview of HTTP [Електронний ресурс] // Режим доступу: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview
4. Len B., Paul C., Rick K. Software Architecture in Practice (SEI Series in Software Engineering), 3rd Edition, 2012, pp. 34-66.
5. Martin K. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems, 1st Edition, 2017, pp. 3-103.
6. What is three-tier architecture? [Електронний ресурс] // Режим доступу: https://www.ibm.com/topics/three-tier-architecture
7. Onion Architecture. Let’s slice it like a Pro, Ritesh K., 2022 [Електронний ресурс] // Режим доступу: https://medium.com/expedia-group-tech/onion-architecture-deed8a554423
8. Robert M. Clean Architecture: A Craftsman's Guide to Software Structure and Design, 1st Edition, 2017, pp. 203-227
9. Vertical Slice Architecture, Jimmy B., 2018 [Електронний ресурс] // Режим доступу: https://www.jimmybogard.com/vertical-slice-architecture/
10. Configuration in ASP.NET Core [Електронний ресурс] // Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/configuration/?view=aspnetcore-8.0
11. Горда О. В. Ергономіка інформаційних технологій, 2020, с. 6-10.
12. Kubernetes overview [Електронний ресурс] // Режим доступу: https://kubernetes.io/docs/concepts/overview/

13. What is Infrastructure as Code (IaC)? [Електронний ресурс] // Режим доступу: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac
14. What is Terraform? [Електронний ресурс] // Режим доступу: https://developer.hashicorp.com/terraform/intro
15. What is CI/CD? [Електронний ресурс] // Режим доступу: https://www.redhat.com/en/topics/devops/what-is-ci-cd
16. Understanding GitHub Actions [Електронний ресурс] // Режим доступу: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions

Додатки

Додаток А

Вихідний код мутації RegisterUser

```
Vlad Susidko
public async Task<MutationResult<Void>> RegisterUser(
    [Service] IIdentityService identityService,
    [Service] IValidator<RegisterUserInput> inputValidator,
    [Service] IOptions<FrontendOptions> frontendOptions,
    [Service] IMailService mailService,
    [Service] ILogger<RegisterUserMutation> logger,
    RegisterUserInput input,
    CancellationToken cancellationToken = default)
{
    var validationResult = inputValidator.ValidateToResult(input);

    if (validationResult.IsFailure){...}

    var registrationResult = await identityService // IIdentityService
        .RegisterUserAsync(input.Email, input.Password, cancellationToken); // Task<Result<...>>

    if (registrationResult.IsFailure){...}

    var emailVerificationTokenResult = await identityService // IIdentityService
        .GenerateEmailVerificationTokenAsync(input.Email, cancellationToken); // Task<Result<...>>

    if (emailVerificationTokenResult.IsFailure){...}

    var urlEncodedToken :string = HttpUtility.UrlEncode(emailVerificationTokenResult.Value);
    var emailVerificationLink :string = $"{frontendOptions.Value.VerifyEmailUrl}" +
                                $"?userId={registrationResult.Value.UserId}" +
                                $"&token={urlEncodedToken}";

    var sendMailResult = await mailService.SendVerificationEmailAsync(
        registrationResult.Value.UserEmail,
        userName: null,
        emailVerificationLink,
        cancellationToken); // Task<Result>

    return sendMailResult.ToMutationResult();
}
```

Додаток Б

Вихідний код мутації Login

```
Vlad Susidko
public async Task<MutationResult<Token>> Login(
    [Service] IIdentityService identityService,
    [Service] ITokenService tokenService,
    [Service] IHttpContextAccessor httpContextAccessor,
    [Service] IValidator<LoginInput> inputValidator,
    LoginInput input,
    CancellationToken cancellationToken = default)
{
    var httpContext = httpContextAccessor.HttpContext!;

    var validationResult = inputValidator.ValidateToResult(input);
    if (validationResult.IsFailure){...}

    var loginResult = await identityService.LoginByPasswordAsync(login: input.Email, input.Password, cancellationToken);
    if (loginResult.IsFailure){...}

    var userInfo = loginResult.Value;
    var tokenPairResult = await tokenService.GenerateTokenPairAsync(userInfo.Id, userInfo.Roles, cancellationToken);

    if (tokenPairResult.IsFailure){...}

    var (accessToken, refreshToken) = tokenPairResult.Value;
    httpContext.Response.AddRefreshToken(refreshToken);

    return accessToken;
}
```

Додаток В

Вихідний код запиту для отримання питань загального тестування GeneralTestQuestions

```
[Authorize]
Vlad Susidko
public async Task<GeneralTestQuestionsPayload> GeneralTestQuestions(
    [Service] IValidator<GeneralTestQuestionsInput> inputValidator,
    [Service] ITestRepository testRepository,
    GeneralTestQuestionsInput input,
    CancellationToken cancellationToken = default)
{
    var validationResult = inputValidator.ValidateToResult(input);

    if (validationResult.IsFailure)
    {
        return new GeneralTestQuestionsPayload(null,
            validationResult.UnionErrors<IGeneralTestQuestionsErrorUnion>());
    }

    var questions:IQueryable<Question> = await testRepository// ITestRepository
        .GetGeneralTestQuestions(input.LanguageCode, cancellationToken);// Task<IQueryable<...>>

    return new GeneralTestQuestionsPayload(questions, Errors:null);
}
```

Додаток Г

Вихідний код мутації перевірки та збереження результатів загального тестування SubmitGeneralTestAnswers

```
Vlad Susidko
public async Task<MutationResult<SubmitGeneralTestAnswersPayload>> SubmitGeneralTestAnswers(
    [Service] ICurrentUser currentUser,
    [Service] ITestRepository testRepository,
    [Service] IUserRepository userRepository,
    [Service] IGeneralTestAnswersProcessor iGeneralTestAnswersProcessor,
    [Service] IValidator<SubmitGeneralTestAnswersInput> inputValidator,
    SubmitGeneralTestAnswersInput input,
    CancellationToken cancellationToken = default)
{
    var validationResult = inputValidator.ValidateToResult(input);
    if (validationResult.IsFailure){...}

    var test = await testRepository.GetGeneralTest(cancellationToken);
    var answers :Dictionary<Guid,Guid> = input.Answers // IList<AnsweredQuestion>
        .ToDictionary(x :AnsweredQuestion => x.QuestionId, x :AnsweredQuestion => x.AnswerId);

    var answersValidationResult = ValidateAnswers(test, answers);
    if (answersValidationResult.IsFailure){...}

    var potentialProblems :IList<string> = iGeneralTestAnswersProcessor.Analyse(test, answers);

    var testResult = new GeneralTestResult
    {
        CompletionDate = DateTimeOffset.Now,
        PotentialProblems = potentialProblems,
        Answers = input.Answers
    };

    await userRepository.SaveGeneralTestResult(currentUser.Id, testResult, cancellationToken);

    return new SubmitGeneralTestAnswersPayload(testResult.Id);
}
```

Додаток Д

Конфігурація проєкту з використанням appsettings.json файлу

```
{
  "AllowedHosts": "*",
  "IdentityOptions": {
    "RequireUniqueEmail": true,
    "RequiredPasswordLength": 8
  },
  "JwtOptions": {
    "ClockSkew": "00:00:10",
    "Auth": {
      "Lifetime": "00:30:00"
    },
    "Refresh": {
      "Lifetime": "30.00:00"
    }
  },
  "AppDbCollectionNames": {
    "Users": "applicationUsers",
    "Tests": "tests",
    "AdviceLists": "advice-lists"
  },
  "AssetOptions": {
    "GeneralTestFilePath": "Assets/general-test.json",
    "AdviceListsFilePath": "Assets/advice-lists.json"
  },
  "EmailOptions": {
    "ServiceName": "PTSDetect"
  },
  "FrontendOptions": {
    "VerifyEmailPath": "/verify-email",
    "ResetPasswordPath": "/reset-password"
  },
  "AzureStorageOptions": {
    "PreviewUrlExpirationTime": "02:00:00",
    "UploadUrlExpirationTime": "00:02:00"
  }
}
```

Додаток Е

Вихідний код Dockerfile файлу

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
RUN apt-get update && apt-get install -y curl
USER $APP_UID
WORKDIR /app
EXPOSE 5000
EXPOSE 5001

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
ARG BUILD_CONFIGURATION=Release
WORKDIR /src
COPY ["src/WebAPI/WebAPI.csproj", "WebAPI/"]
COPY ["src/Application/Application.csproj", "Application/"]
RUN dotnet restore "WebAPI/WebAPI.csproj"
WORKDIR /
COPY . .
WORKDIR /src/WebAPI
RUN dotnet build "WebAPI.csproj" -c $BUILD_CONFIGURATION -o /app/build

FROM build AS publish
ARG BUILD_CONFIGURATION=Release
RUN dotnet publish "WebAPI.csproj" -c $BUILD_CONFIGURATION -o /app/publish /p:UseAppHost=false

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "WebAPI.dll"]
```

Додаток Є

Вихідний код ptsdetect-api.yaml файлу конфігурації Kubernetes

```yaml
apiVersion: v1
kind: Service
metadata:
  name: ptsdetect-api-service
spec:
  type: LoadBalancer
  selector:
    app: ptsdetect-api
  ports:
    - port: 5000
      targetPort: 5000
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ptsdetect-api-deployment
  labels:
    app: ptsdetect-api
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ptsdetect-api
  template:
    metadata:
      labels:
        app: ptsdetect-api
    spec:
      containers:
        - name: ptsdetect-api
          image: exemory/ptsdetect-api:latest
          ports:
            - containerPort: 5000
          envFrom:
            - configMapRef:
                name: ptsdetect-api-config
            - secretRef:
                name: ptsdetect-api-secrets
          livenessProbe:
            httpGet:
              path: /healthz
              port: 5000
          readinessProbe:
            httpGet:
              path: /healthz
              port: 5000
          startupProbe:
            httpGet:
              path: /healthz
              port: 5000
            failureThreshold: 30
            periodSeconds: 1
            timeoutSeconds: 3
```

Додаток Ж

Вихідний код main.tf файлу Terraform конфігурації

```
You, last month | 1 author (You)
# Configure the Azure provider
You, last month | 1 author (You)
provider "azurerm" {
  features {}
}

You, last month | 1 author (You)
resource "azurerm_resource_group" "default" {
  name     = var.resource_group_name
  location = var.resource_location
}

You, last month | 1 author (You)
resource "azurerm_storage_account" "default" {
  name                     = var.storage_account_name
  resource_group_name      = azurerm_resource_group.default.name
  location                 = azurerm_resource_group.default.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}

You, last month | 1 author (You)
resource "azurerm_storage_container" "user_avatars" {
  name                  = "user-avatars"
  storage_account_name  = azurerm_storage_account.default.name
  container_access_type = "private"
}

You, last month | 1 author (You)
resource "azurerm_kubernetes_cluster" "default" {
  name                = "ptsdetect-aks"
  location            = azurerm_resource_group.default.location
  resource_group_name = azurerm_resource_group.default.name
  dns_prefix          = "ptsdetect-k8s"

  You, last month | 1 author (You)
  default_node_pool {
    name                = "default"
    node_count          = 1
    vm_size             = "Standard_B2s"
    enable_auto_scaling = true
    min_count           = 1
    max_count           = 1
  }

  You, last month | 1 author (You)
  identity {
    type = "SystemAssigned"
  }
}
```

Додаток З

Вихідний код задачі побудови Docker образу системи та завантаження його у Docker Hub репозиторій

```yaml
build-and-publish:
  name: Build and publish PTSDetect API image
  runs-on: ubuntu-latest
  environment: production
  steps:
    - name: Checkout repository
      uses: actions/checkout@v4

    - name: Build Docker image
      working-directory: backend
      run: docker build -t ${{vars.PTSDETECT_API_IMAGE}}:latest -t ${{vars.PTSDETECT_API_IMAGE}}:${{env.IMAGE_TAG}} .

    - name: Login to Docker Hub
      run: docker login -u ${{vars.DOCKER_HUB_USERNAME}} -p ${{secrets.DOCKER_HUB_TOKEN}}

    - name: Push Docker image to Docker Hub
      run: docker push -a ${{vars.PTSDETECT_API_IMAGE}}

    - name: Logout from Docker Hub
      run: docker logout
```