Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій

Побудова багаторівневого веб-застосування з високою доступністю на хмарній платформі

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

Керівник дипломної роботи

кандидат технічних наук, старший викладач

Черкасов Д.І.

_____ (Підпис)

“___” _____ 2025 року

Виконав студент

ІПЗ-4 Іщенко Т. О.

“___” _____ 2025 року

Київ 2025

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій

ЗАТВЕРДЖУЮ

зав. кафедри мережних технологій, д.ф-м.н.

_____ Малашок Г.І. „_____” _____ 2025 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на дипломну роботу

студенту Іщенку Тіхону Олексійовичу факультету інформатики 4 курсу бакалаврської програми

ТЕМА: Побудова багаторівневого веб-застосування з високою доступністю на хмарній платформі.

Зміст ТЧ до дипломної роботи:

ЗМІСТ

АНОТАЦІЯ

ВСТУП

Розділ 1. Практики високої доступності та відмовостійкості у веб-застосунках

Розділ 2. Проектування та реалізація веб-застосунку

Розділ 3. Автоматизоване розгортання, CI/CD та продуктивність

Висновки

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

Дата видачі „___” _____ 2025 р.

Керівник_____

(підпис) Завдання отримав_____ (підпис)

Календарний план виконання роботи

№	Назва етапу дипломної роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу	23.10.2024	
2.	Огляд літератури за темою роботи	23.01.2025	
3.	Проведення дослідження	04.02.2025	
4.	Написання програмного застосунку	12.03.2025	
5.	Написання текстової частини	01.04.2025	
6.	Захист дипломної роботи	02.04.2025	

Студент _____

Керівник _____ “_____”_____ 2025

ЗМІСТ

АНОТАЦІЯ

Робота зосереджена на процесі дослідження, аналізу та впровадження архітектурних і технологічних рішень для розробки багаторівневого веб-застосунку з високою доступністю на хмарній платформі. Центральною ідеєю є застосування можливостей хмарних обчислювальних сервісів, які завдяки автоматичному масштабуванню, балансуванню навантаження та моніторингу суттєво полегшують побудову надійних веб-систем. Як приклад реалізації створено застосунок для допомоги безпритульним тваринам, що дозволяє користувачам розміщувати та переглядати оголошення.

У ході роботи було проведено огляд і аналіз сучасних технологій розробки веб-застосунків, таких як ASP.NET Core для серверної частини, React для клієнтської частини, SQL Server та Entity Framework для роботи з базами даних. Окрему увагу приділено вивченню підходів до забезпечення високої доступності, відмовостійкості, масштабованості та безпеки застосунків. Детально описано процес побудови багаторівневої архітектури, організації взаємодії між компонентами системи та впровадження механізмів управління доступом.

Значна увага приділена дослідженню та реалізації процесів автоматизованого розгортання і оновлення за допомогою інструментів CI/CD, зокрема GitHub Actions, та налаштуванню

хмарної інфраструктури на базі Microsoft Azure. В рамках роботи проаналізовано різні стратегії розгортання і обрано найбільш ефективні підходи для підвищення стабільності та надійності системи.
Результатом стала розробка масштабованого, стабільного та готового до реального використання веб-застосунку, що відповідає сучасним вимогам до хмарних архітектур.

ВСТУП

У сучасних умовах стрімкого розвитку цифрових технологій розробка веб-застосунків з високою доступністю є одним із основних викликів для інженерів програмного забезпечення. Збільшення навантаження на онлайн-сервіси, очікування безперебійного доступу та вимоги до стійкості систем вимагають використання хмарних технологій, що забезпечують автоматичне масштабування, моніторинг та відмовостійкість.
Особливу актуальність має побудова багаторівневих архітектур, що дозволяють розподіляти функціональні компоненти системи, підвищувати її надійність, гнучкість і стабільність.
Дана робота присвячена розробці багаторівневого веб-застосунку з високою доступністю на хмарній платформі. Як приклад практичного впровадження обрано соціально значущий проєкт - створення веб-сайту для допомоги безпритульним тваринам. Сервіс дозволяє користувачам - як приватним особам, так і представникам притулків - розміщувати оголошення про тварин, які потребують нового дому, переглядати існуючі публікації та знаходити потенційних власників. Таким чином, застосунок поєднує важливу соціальну функцію із технічними вимогами до високої доступності, масштабованості та стабільної роботи.

У ході дослідження було проведено аналіз сучасних архітектурних підходів, вивчено технології для побудови багаторівневих систем, а також оцінено інструменти автоматизації процесів розгортання. Для реалізації серверної частини обрано платформу ASP.NET Core, для клієнтської частини - бібліотеку React, для зберігання даних - SQL Server у поєднанні з ORM-технологією Entity Framework. Хмарне розгортання здійснюється на базі Microsoft Azure із впровадженням автоматизованих процесів CI/CD через GitHub Actions.

Об’єкт дослідження: процес створення багаторівневого веб-застосунку з високою доступністю на хмарній платформі.

Мета дослідження: вивчення та впровадження оптимальних архітектурних, інфраструктурних і технологічних рішень для забезпечення стабільної роботи багаторівневого веб-застосунку в умовах змінного навантаження.

Завдання роботи:

1. Провести аналіз сучасних технологій побудови високодоступних веб-застосунків.
2. Розробити архітектуру застосунку із забезпеченням відмовостійкості, масштабованості та безпеки.
3. Реалізувати застосунок для допомоги безпритульним тваринам на базі обраних технологій.
4. Налаштувати автоматизовані процеси CI/CD для спрощення розгортання та обслуговування системи.

Таким чином, робота спрямована на комплексне вирішення завдань побудови, розгортання та підтримки високодоступних багаторівневих веб-застосунків у хмарному середовищі.

Розділ 1. Практики високої доступності та відмовостійкості у веб-застосунках

1.1. Концепція високої доступності та відмовостійкості

Висока доступність та відмовостійкість є критично важливими характеристиками для хмарних веб-застосунків, оскільки вони забезпечують безперервність роботи та надійність сервісу для користувачів.

Висока доступність означає здатність системи залишатися працездатною та доступною для користувачів протягом максимально можливого часу. Це досягається шляхом мінімізації часу простою та швидкого відновлення після збоїв. Зазвичай, високою доступністю вважається показник безперебійної роботи на рівні 99% і вище. У контексті хмарних веб-застосунків це означає, що застосунок розгорнутий та налаштований таким чином, щоб забезпечити безперервну роботу та здатність відповідати на запити користувачів навіть у випадку відмови окремих компонентів системи.

Відмовостійкість стосується здатності системи продовжувати функціонувати належним чином у разі відмови одного або декількох її компонентів. У хмарних веб-застосунках відмовостійкість забезпечується шляхом розподілу навантаження між декількома серверами, використанням кластерів баз даних та інших методів, що дозволяють системі залишатися функціональною навіть при виникненні проблем з окремими її частинами.

Важливість високої доступності та відмовостійкості полягає в тому, що вони безпосередньо впливають на досвід користувачів та репутацію сервісу. Недоступність або збої у роботі веб-застосунку можуть призвести до втрати користувачів, зниження довіри та фінансових втрат, особливо якщо сервіс пов'язаний з бізнесом або наданням критично важливих послуг.

1.2. Архітектурні підходи для забезпечення високої доступності

1.2.1. Мікросервісна архітектура

Мікросервісна архітектура передбачає розподіл застосунку на невеликі незалежні сервіси, кожен з яких відповідає за конкретну бізнес-функцію. Такий підхід забезпечує ізоляцію компонентів, що дозволяє окремим сервісам працювати автономно, не впливаючи на загальну працездатність системи у випадку відмови одного з них. Крім того, мікросервіси можуть бути розгорнуті та масштабовані незалежно, що підвищує гнучкість та адаптивність системи до змінних навантажень. Основними елементами мікросервісної архітектури є:

- API-шлюзи (наприклад, Azure API Management, AWS API Gateway) для управління запитами між сервісами.
- Розподілене зберігання даних, де кожен сервіс може мати свою окрему базу даних (наприклад, SQL Server, NoSQL MongoDB).
- Сервіси балансування навантаження (наприклад, Azure Load Balancer, AWS ELB) для рівномірного розподілу трафіку.
- Система контейнеризації (наприклад, Docker, Azure Container Apps) для спрощення розгортання.

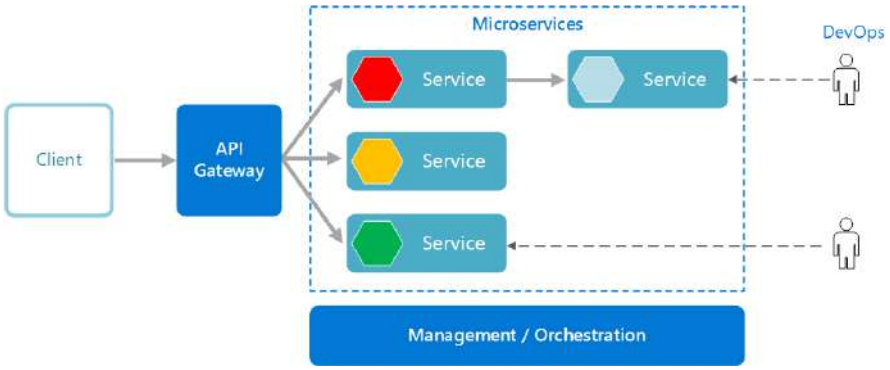


Рис.1 Діаграма мікросервісної архітектури

Мікросервісна архітектура сприяє високій доступності системи завдяки ізоляції сервісів. У разі відмови одного сервісу інші продовжують функціонувати незалежно, що мінімізує вплив на загальну працездатність системи. Додатково, використання контейнеризації та оркестрації

дозволяє автоматично перезапускати збоїлі сервіси та масштабувати їх відповідно до навантаження. Інструменти моніторингу та логування забезпечують своєчасне виявлення та реагування на проблеми, що підвищує загальну надійність системи.

1.2.2. Монолітні системи

Монолітна архітектура передбачає створення єдиного, цілісного застосунку, у якому всі функціональні компоненти інтегровані в один процес. Такий підхід дозволяє спрощувати розробку та тестування на початкових етапах, але може створювати виклики при масштабуванні та впровадженні змін.

Монолітні застосунки зручні для розробки оскільки не потребують окремого рогортання для окремих частин і інтеграції між ними, а також усі компоненти розгортаються та оновлюються разом, що спрощує управління версіями, та **:** всі модулі працюють з одними даними, що забезпечує узгодженість.
Це також створює і проблеми для відмовостійкості та високої доступності, оскільки компоненти дуже сильно залежать один від одного і кожен компонент є єдиною точкою відмови. Тобто при виникненні проблеми в одному модулі може бути порушено функціонування всього застосунку. Також є проблемою обмежена можливість масштабування та складність внесення змін без впливу на весь застосунок через монолітну архітектуру.

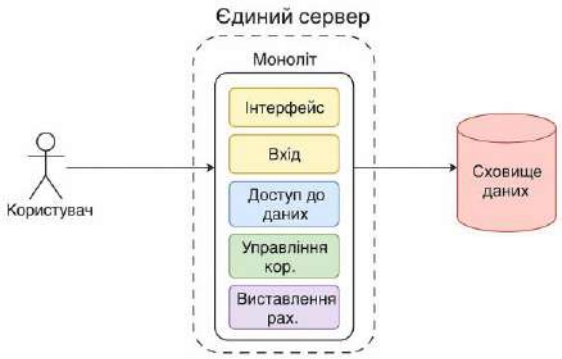


Рис.2 Діаграма монолітної архітектури

Монолітна архітектура має обмежену здатність до забезпечення високої доступності. Оскільки всі компоненти інтегровані в один процес, відмова одного модуля може призвести до збоїв у всій системі. Масштабування такої архітектури часто вимагає розгортання

додаткових копій всього застосунку, що не є ефективним. Хоча монолітні системи можуть бути простішими в розробці та розгортанні на початкових етапах, їхня стійкість до відмов знижується зі зростанням складності та обсягу функціоналу.

1.2.3. Модульні архітектури

Модульна архітектура (або модульний моноліт) передбачає розбиття великої кодової бази на автономні, логічно ізольовані модулі, які розгортаються як єдине виконуване середовище з чітко визначеними інтерфейсами взаємодії між собою. Кожен модуль реалізує окрему бізнес-область, зберігає власну внутрішню структуру даних та може бути розроблений і протестований незалежно від інших. Така структура поєднує простоту розгортання класичного моноліту з перевагами низької зв’язності й високої зв’язності компонентів, характерними для мікросервісів.

Застосування модульного підходу підвищує масштабованість системи, оскільки нові модулі можна додавати без зміни існуючої архітектури, а окремі області бізнес-логіки можуть оброблятися незалежно. Крім того, модулі полегшують підтримку та розвиток коду: команда може фокусуватися лише на межах одного модуля, не поглиблюючись у внутрішню реалізацію інших. Це сприяє швидшій ітеративній розробці та суттєво знижує ризик поширення дефектів по всій системі.

При цьому модульні системи використовують принципи предметно-обмежених контекстів (DDD), де для кожного модуля визначаються чіткі контракти та залежності, що обмежує зв’язність та сприяє підтриманню узгодженості даних.

Серед ключових викликів модульних архітектур - управління залежностями між модулями, підтримка зворотної сумісності інтерфейсів, а також організація автоматичних тестів для перевірки інтеграції на рівні всієї системи. Для уникнення «згортання» модулів необхідно впроваджувати механізми суворого розмежування відповідальності та використовувати інструменти аналізу статичної залежності.

В архітектурних студіях та промислових кейсах (наприклад, Shopify, Appsmith) модульні моноліти довели ефективність як проміжного етапу міграції від класичного моноліту до мікросервісів, поєднуючи простоту одноразового розгортання з можливістю подальшої

еволюції системи. Такий підхід забезпечує баланс між продуктивністю, відмовостійкістю та гнучкістю подальшого розвитку архітектури.

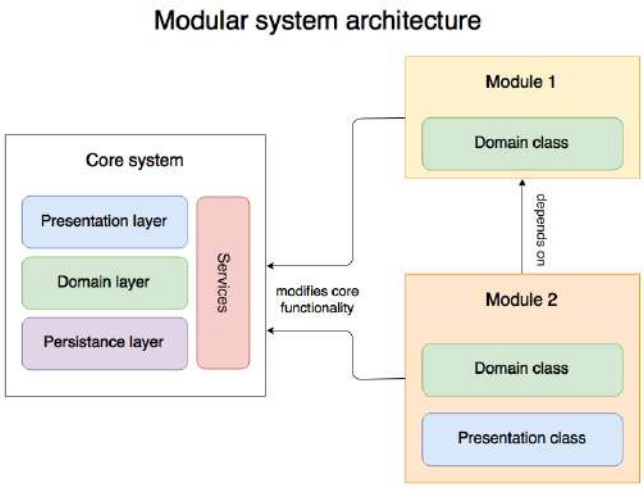


Рис.3 Діаграма модульної архітектури

Модульна архітектура поєднує переваги моноліту та мікросервісів, забезпечуючи кращу відмовостійкість порівняно з традиційним монолітом. Хоча всі модулі розгортаються як єдине ціле, чітке розмежування відповідальності та ізоляція модулів дозволяють локалізувати проблеми та зменшити їх вплив на всю систему. Використання принципів предметно-орієнтованого проектування та суворе дотримання контрактів між модулями сприяє підтриманню узгодженості та стабільності системи, що дає кращу відмовостійкість ніж монолітний підхід.

1.3. Архітектурні моделі розгортання

1.3.1. Зосереджені (централізовані) архітектури

Зосереджена або централізована архітектура передбачає розміщення всіх компонентів інформаційної системи на одному фізичному або логічно єдиному сервері чи групі серверів у межах одного датацентру або локації. Такий підхід традиційно використовувався у класичних клієнт-серверних системах та є поширеним у невеликих або локальних проєктах, де немає високих вимог до масштабування або географічного покриття. У централізованій архітектурі вся бізнес-логіка, обробка даних та зберігання інформації реалізується на єдиній інфраструктурі, що значно спрощує розгортання, адміністрування, моніторинг і захист системи.

Основною перевагою зосередженого підходу є простота реалізації, прозора логіка взаємодії компонентів та зниження складності при налагодженні та супроводі. Централізована архітектура знижує ризики, пов’язані з проблемами реплікації, консистентності даних або відмов мережевих з’єднань між окремими вузлами. Крім того, централізація дозволяє більш точно контролювати ресурси, забезпечує більш передбачувану продуктивність і може бути економічно ефективнішою в невеликих або контрольованих середовищах.

Водночас централізовані архітектури мають суттєві обмеження в контексті високої доступності та масштабованості. Відмова центрального вузла може призвести до повної зупинки системи, а збільшення навантаження - до обмежень у горизонтальному масштабуванні. Зважаючи на це, централізовані системи менш ефективні в умовах великої кількості одночасних користувачів або при необхідності обслуговування користувачів з різних регіонів.

Таким чином, зосереджена архітектура є доцільною у випадках, коли пріоритетом є простота, локальне розгортання та мінімальні вимоги до розподіленості. У масштабніших проєктах її часто комбінують з іншими архітектурними підходами або трансформують у гібридні рішення з частковою розподіленістю для досягнення більшої відмовостійкості.

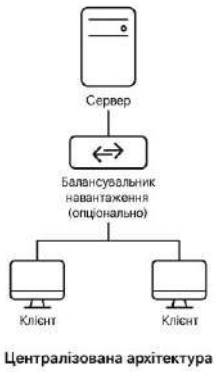


Рис. 4 Діаграма централізованої архітектури

1.3.2 Розподілені системи

Розподілена архітектура передбачає розміщення функціональних компонентів застосунку на кількох серверах або вузлах, які можуть бути як географічно рознесеними, так і розташованими в межах одного дата-центру. Такий підхід дозволяє досягти високої

доступності, масштабованості та відмовостійкості, оскільки відмова окремого вузла не призводить до зупинки всієї системи.

У порівнянні з централізованою архітектурою, де всі компоненти зосереджені на одному сервері або в одному дата-центрі, розподілені системи забезпечують кращу стійкість до відмов та гнучкість у масштабуванні. Наприклад, у централізованій системі відмова центрального вузла може призвести до повної зупинки сервісу, тоді як у розподіленій архітектурі навантаження може бути перерозподілено на інші вузли .

Розподілені системи можуть бути класифіковані за їх географічним розташуванням:

- Локально розподілені системи: Вузли розташовані в межах одного дата-центру або фізично близько один до одного. Така конфігурація дозволяє ефективно балансувати навантаження та забезпечувати відмовостійкість на рівні окремих серверів.
- Географічно розподілені системи: Вузли розміщені в різних географічних локаціях або дата-центрах. Це дозволяє забезпечити відмовостійкість на рівні дата-центрів, зменшити затримки для користувачів з різних регіонів та підвищити загальну доступність системи.

Переваги розподілених систем включають:

- Масштабованість: Можливість додавання нових вузлів для обробки зростаючого навантаження без значних змін у системі.
- Відмовостійкість: Відмова окремого вузла не призводить до зупинки всієї системи, що забезпечує безперервність сервісу.
- Гнучкість: Можливість розміщення вузлів у різних локаціях для оптимізації продуктивності та доступності.

Однак розподілені системи також мають виклики:

- Складність управління: Необхідність синхронізації даних між вузлами та забезпечення їх узгодженості.
- Залежність від мережі: Продуктивність системи може залежати від стабільності мережевих з'єднань між вузлами.

- Безпека: Розподіленість системи може ускладнювати забезпечення безпеки даних та доступу.

Таким чином, вибір між централізованою та розподіленою архітектурою залежить від конкретних вимог до системи, таких як масштабованість, доступність, відмовостійкість та географічне покриття.

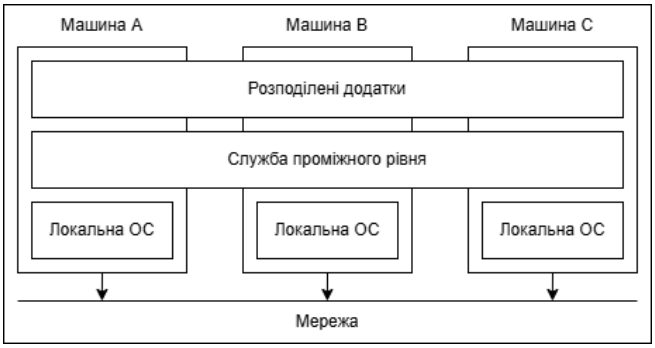


Рис. 5 Діаграма централізованої архітектури

1.4. Масштабування застосунків

Масштабування серверної частини веб-застосунків є ключовим аспектом забезпечення їх продуктивності та стійкості до навантажень. Цей процес включає застосування вертикального та горизонтального масштабування, а також балансування навантаження для оптимального розподілу ресурсів.

Вертикальне масштабування досягається за рахунок підвищення потужності окремого обчислювального пристрою. Після припинення дії закону Мура вертикальне масштабування зводиться до збільшення кількості обчислювальних ядер у комп’ютері. Такий підхід має свої обмеження, адже кількість процесорів не може зростати необмежено хоча б в силу обмеженості фізичних розмірів комп’ютерів. Крім того вартість багатопроцесорних комп’ютерів непропорційно зростає з кількістю обчислювальних ядер, що може підтримувати система. Крім того існують і принципові обмеження для розпаралелювання обчислень, що пов’язані з послідовною структурою традиційних методів програмування.[3]

Горизонтальне масштабування полягає у використанні великої кількості достатньо простих комп’ютерів.[3]

Ідеальним при масштабуванні є випадок, коли продуктивність обчислювальної системи лінійно залежить від кількості обчислювальних вузлів у системі. Така ситуація називається лінійною масштабованістю, та є практично недосяжною. Закон Амдала описує приріст

швидкодії системи залежно від кількості наявних обчислювальних ядер. Приріст швидкодії S(n) обчислювальної системи за рахунок використання n обчислювальних ядер визначається відношенням:

$$S(n) = \frac{T_s + T_p}{T_s + \frac{T_p}{n}},$$

Рис 6. Формула приросту швидкодії

де Ts – час обробки послідовної складової обчислення одним обчислювачем, Тр – час обробки паралельної складової обчислення одним обчислювачем.[3]

Для забезпечення ефективного балансування навантаження в мережі необхідно дотримуватися таких вимог:

1. Гарантована обробка запитів: Кожен вхідний запит до системи повинен бути належним чином оброблений, забезпечуючи безперебійну роботу сервісу.
2. Ефективність розподілу навантаження: Рівномірний розподіл трафіку між серверами дозволяє уникнути перевантаження окремих вузлів, забезпечуючи оптимальне використання ресурсів та стабільну продуктивність системи.
3. Зменшення часу відповіді: Рівномірний розподіл навантаження сприяє швидшій обробці запитів, що, в свою чергу, зменшує затримки та покращує користувацький досвід.
4. Передбачуваність алгоритмів балансування: Чітке розуміння та документування використовуваних алгоритмів розподілу навантаження дозволяє ефективно керувати системою та прогнозувати її поведінку під різними навантаженнями.
5. Масштабованість системи: Можливість адаптації до змін у навантаженні, зокрема при його різкому збільшенні, забезпечує стабільну роботу сервісу без втрати якості обслуговування.

1.5. Хмарна інфраструктура та балансування навантаження

Побудова хмарної інфраструктури є ключовим аспектом забезпечення високої доступності веб-застосунків. Використання хмарних сервісів дозволяє швидко адаптуватися до змінних навантажень, забезпечуючи безперебійний доступ до ресурсів. Балансування навантаження відіграє важливу роль у розподілі трафіку між серверами та регіонами, запобігаючи перевантаженню окремих компонентів системи.

Також важливим є постійний моніторинг стану системи для своєчасного виявлення та усунення несправностей, що забезпечує стабільну роботу хмарної інфраструктури.

1.5.1. Використання Microsoft Azure для побудови високодоступних систем

Платформа Microsoft Azure пропонує широкий набір рішень для побудови хмарних застосунків, які забезпечують високу доступність та відмовостійкість. Це включає розподіл ресурсів між дата-центрами, балансування навантаження, автоматичне масштабування та моніторинг.

Ключові принципи архітектури високої доступності:

- Регіональна надмірність
- Резервування критичних компонентів
- Автоматичне масштабування
- Моніторинг і відновлення

Azure дозволяє розгортати віртуальні машини з різними операційними системами та конфігураціями, що забезпечує гнучкість у виборі середовища для застосунків. Використання груп доступності та зон доступності гарантує відмовостійкість і високу доступність сервісів. Ці механізми розподіляють віртуальні машини між різними фізичними серверами та дата-центрами, мінімізуючи ризик одночасного виходу з ладу кількох ресурсів. Крім того, Azure підтримує автоматичне масштабування віртуальних машин, що дозволяє динамічно реагувати на змінні навантаження. Це забезпечує оптимальне використання ресурсів та економію витрат.

Служби додатків дозволяють швидко створювати, розгортати та масштабувати веб-додатки та API. Вбудовані можливості автоматичного масштабування та балансування навантаження забезпечують стабільну роботу додатків під час пікових навантажень. Служба додатків підтримує різні мови програмування, такі як .NET, Java, Node.js та Python, що робить його універсальним рішенням для розробників. Крім того, інтеграція з іншими сервісами Azure, такими як бази даних та системи моніторингу, спрощує управління та забезпечує високу доступність додатків. Це дозволяє зосередитися на розробці функціональності, не турбуючись про інфраструктуру.

Azure Kubernetes Service (AKS) спрощує розгортання, управління та масштабування контейнеризованих додатків, забезпечуючи автоматичне відновлення та балансування

навантаження для контейнерів. AKS інтегрується з іншими сервісами Azure, такими як моніторинг та безпека, що забезпечує комплексний підхід до управління додатками. Використання контейнерів дозволяє ізолювати середовище виконання додатків, що підвищує їхню стабільність та безпеку. Крім того, AKS підтримує автоматичне масштабування контейнерів залежно від навантаження, що забезпечує ефективне використання ресурсів. Це робить AKS потужним інструментом для розробки та експлуатації сучасних хмарних додатків.

Azure пропонує різні рішення для балансування навантаження, такі як Azure Load Balancer для внутрішнього та зовнішнього трафіку, а також Azure Application Gateway для управління HTTP/HTTPS-трафіком з можливістю веб-захисту. Ці сервіси забезпечують рівномірний розподіл трафіку між ресурсами, підвищуючи продуктивність та доступність додатків. Azure Load Balancer працює на мережевому рівні, забезпечуючи низьку затримку та високу пропускну здатність. Azure Application Gateway, у свою чергу, надає можливості веб-аплікаційного брандмауера (WAF), захищаючи додатки від загроз на рівні застосунків. Це дозволяє забезпечити безпеку та стабільність роботи веб-додатків.

Azure Traffic Manager дозволяє розподіляти користувацькі запити між різними регіонами, забезпечуючи низьку затримку та високу доступність додатків на глобальному рівні. Він підтримує різні методи маршрутизації, такі як пріоритетна, географічна та на основі продуктивності, що дозволяє налаштувати оптимальний розподіл трафіку. Traffic Manager інтегрується з іншими сервісами Azure, такими як Load Balancer та Application Gateway, забезпечуючи комплексне управління трафіком. Це дозволяє швидко реагувати на зміни в навантаженні та забезпечувати безперебійну роботу додатків. Крім того, використання Traffic Manager сприяє підвищенню задоволеності користувачів завдяки швидкому доступу до сервісів.

Azure SQL Database пропонує вбудовані механізми реплікації та аварійного відновлення, що забезпечує безперервність доступу до даних навіть у разі збоїв. Azure SQL Database підтримує активну гео-реплікацію, дозволяючи створювати копії бази даних у різних регіонах для забезпечення відмовостійкості. Крім того, цей сервіс підтримує автоматичне масштабування, що забезпечує стабільну продуктивність при змінних навантаженнях.

Azure Backup та Azure Site Recovery є ключовими компонентами стратегії забезпечення високої доступності та відмовостійкості в інфраструктурі Microsoft Azure. Ці сервіси забезпечують захист даних і безперервність бізнес-процесів у разі збоїв або аварійних ситуацій. Azure Backup це хмарне рішення для резервного копіювання, яке забезпечує захист даних як для локальних, так і для хмарних ресурсів. Воно підтримує резервне копіювання файлів, папок, додатків і віртуальних машин, забезпечуючи довгострокове збереження даних з можливістю гнучкого налаштування політик резервного копіювання. Azure Backup гарантує безпеку даних за допомогою шифрування під час передачі та зберігання, а також забезпечує захист від випадкового видалення або атак типу ransomware. Azure Site Recovery (ASR) це рішення для відновлення після аварій, яке забезпечує безперервність бізнесу шляхом реплікації робочих навантажень на фізичних або віртуальних машинах до альтернативного місця розташування. У разі збою основного сайту ASR дозволяє виконати автоматичне переключення на резервний сайт з мінімальними втратами даних і часу простою. Після відновлення основного сайту можливе повернення до початкового стану без порушення роботи системи. Використання Azure Backup та Azure Site Recovery разом забезпечує комплексний підхід до захисту даних та відновлення після аварій, що є критично важливим для підтримки високої доступності та відмовостійкості систем у хмарному середовищі.

1.5.2. Альтернативні платформи (AWS, Google Cloud) для відмовостійких рішень

Окрім Microsoft Azure, провідними платформами для побудови таких систем є Amazon Web Services (AWS) та Google Cloud Platform (GCP). Вони пропонують широкий спектр сервісів для автоматичного масштабування, балансування навантаження, резервного копіювання та відновлення після аварій.

Для наочного порівняння розглянемо таблицю основних функцій та можливостей, які забезпечують відмовостійкість:

	AWS	Google Cloud	Azure
Автоматичне масштабування	Amazon EC2 Auto Scaling	Compute Engine Managed Instance Groups	Virtual Machine Scale Sets (VMSS)
Балансування	Elastic Load Balancer	Google Cloud Load	Azure Load Balancer,

	AWS	Google Cloud	Azure
навантаження	(ELB)	Balancing	Azure Traffic Manager
Географічна реплікація БД	Amazon RDS Multi-AZ Deployments	Cloud SQL High Availability	Azure SQL Database with Geo-Replication
Резервне копіювання	AWS Backup	Google Cloud Backup and DR	Azure Backup
Відновлення після аварій	AWS Disaster Recovery	Google Cloud Disaster Recovery	Azure Site Recovery
Гібридна хмара	AWS Outposts	Google Cloud Anthos	Azure Arc

Таблиця 1. Порівняння хмарних платформ

Тобто всі ці сервіси мають схожий функціонал і приблизно однакові ціни для зберігання конкуренції.

1.5.3. Обгрунтування вибору Azure

Але Microsoft Azure є найбільш оптимізованою хмарною платформою для розгортання .NET-застосунків завдяки глибокій інтеграції з технологіями Microsoft. Використання Azure App Service дозволяє легко розгортати ASP.NET Core-додатки без необхідності налаштовувати серверну інфраструктуру. Azure Virtual Machines підтримують Windows Server та SQL Server із вбудованими ліцензіями, що зменшує витрати для додатків, які вже використовують екосистему Microsoft. Інструменти Azure DevOps та GitHub Actions забезпечують автоматизоване тестування та CI/CD-процеси для .NET-додатків, що пришвидшує їх оновлення та підтримку.

Azure також має потужні можливості для роботи з базами даних, такі як Azure SQL Database, яка підтримує Entity Framework Core та надає автоматичне резервне копіювання та гео-реплікацію. Використання Azure Functions дозволяє запускати .NET-код у serverless-середовищі, зменшуючи витрати на інфраструктуру. Крім того, Azure пропонує вбудовані засоби моніторингу та безпеки, такі як Azure Application Insights та Azure Active Directory, що полегшує контроль доступу та аналіз продуктивності додатків. Завдяки цьому Azure є найкращим вибором для додатків, що використовують .NET.

1.6. Вибір технологій

Вибір технологій є фундаментальним етапом розробки будь-якого багаторівневого веб-застосунку, оскільки він безпосередньо впливає на продуктивність, масштабованість та безпеку. Технологічний стек визначає, які інструменти й підходи будуть застосовані на кожному рівні архітектури, від сервера до клієнта, і від цього залежить, наскільки ефективно можна реалізувати бізнес-вимоги та підтримувати систему в подальшому. Некоректний вибір може призвести до значних затримок у розробці, складнощів із масштабуванням та підвищенню експлуатаційних витрат, тому вже на стадії планування слід врахувати критерії сумісності, рівня підтримки та довгострокової життєздатності обраних технологій.

1.6.1. Мови програмування та середовища виконання

Визначення мови програмування та середовища виконання на сервері формує основу для реалізації бізнес-логіки та є ключем до забезпечення продуктивності обчислень і обробки запитів. Вибір між такими підходами, як компільовані рішення (.NET Core), інтерпретовані середовища (Node.js) чи віртуальні машини (Java), впливає на затримки I/O-операцій, можливості паралелізму та готовність команди до підтримки коду. Сучасні платформи на кшталт .NET Core та Node.js надають крос-платформеність і мікросервісну архітектуру, тоді як Java зі Spring Boot або Python із Django залишаються репрезентативними для корпоративного сегмента з багаторічними практиками безперервного розгортання й моніторингу.

1.6.1.1. .NET

Платформа .NET Core з фреймворком ASP .NET Core, реалізована на мові C#, є сучасним рішенням для розробки високопродуктивних веб-застосунків із крос-платформеною підтримкою та модульною архітектурою. Завдяки відкритому вихідному коду під ліцензією MIT і поширенню функціональності через окремі NuGet-пакети, ASP .NET Core дозволяє мінімізувати розмір розгортання, включаючи лише ті компоненти, що необхідні для конкретного проєкту.

Перевагами .NET можна назвати:

- ASP .NET Core демонструє продуктивність на рівні або вищу за найшвидші альтернативи (наприклад, майже в 5 разів швидше Node.js) завдяки оптимізованому HTTP-серверу Kestrel і асинхронній моделі обробки запитів.
- Глибока інтеграція з Microsoft Azure спрощує налаштування автоматичного масштабування, моніторингу (Application Insights) та безперервного розгортання у хмарному середовищі.

- Вбудований контейнер залежностей, підтримка модульного тестування та якісні шаблони MVC/Web API сприяють чистій архітектурі й високому покриттю юніт-тестами.

Але також є і недоліки:

- Спільнота .NET Core менш численна, ніж у Node.js або Java Spring, що може уповільнити пошук рішень нетривіальних проблем.
- Інтеграція з іншими мовами не дуже зручна, хоч C# й TypeScript вбудовані добре, інтеграція з Python, Ruby чи Go не така гладка, як у Node.js-екосистемі.
- Для клієнтського коду без використання Blazor доводиться застосовувати зовнішні JavaScript-фреймворки.

  Таким чином, ASP .NET Core надає потужний інструментарій для створення високопродуктивних і масштабованих веб-систем із сучасними практиками розробки. Водночас необхідність використовувати зовнішні JavaScript-фреймворки та менша за розмірами спільнота може створити труднощі в розробці.

1.6.1.2. Node.js

Node.js вирізняється подієво-орієнтованою, неблокуючою моделлю вводу-виводу, що дозволяє ефективно обслуговувати велику кількість одночасних підключень із мінімальним споживанням ресурсів. Платформа базується на рушії V8 від Google та бібліотеці libuv, що забезпечує високу продуктивність при обробці I/O-інтенсивних операцій. Використання єдиної мови програмування - JavaScript/TypeScript - на клієнті й сервері сприяє скороченню кривої навчання та повторному використанню бізнес-логіки між шарами застосунку. Окрім того, екосистема npm налічує понад мільйон пакетів, що значно пришвидшує розробку й прототипування. Водночас однопотокова модель Node.js є слабкою ланкою при виконанні CPU-інтенсивних задач, а callback-орієнтована природа коду може призводити до глибокої вкладеності та ускладнювати читання й супровід.

Перевагами Node.js можна вважати:

- Неблокуючу архітектуру I/O, яка дозволяє обслуговувати тисячі одночасних запитів без створення додаткових потоків, що знижує накладні витрати та збільшує пропускну здатність сервера.
- Використання однієї мови програмування на фронтенді та бекенді сприяє уніфікації моделей даних і логіки, прискорює комунікацію між командами та полегшує навчання нових розробників.

- Реєстр npm містить понад мільйон модулів для різноманітних завдань - від ORM та реального часу до мікросервісної оркестрації - що скорочує час розробки готових рішень і дозволяє фокусуватися на бізнес-законах.

Недоліками Node.js можна назвати, те що:

- Важкі обчислювальні задачі (CPU-bound) блокують event loop, що призводить до зростання затримок і зниження пропускної здатності при виконанні складних обчислень.
- Глибока вкладеність callback-функцій ускладнює читання та відлагодження коду, створюючи ризик багів і труднощів у підтримці, хоча сучасні async/await значно полегшили цю проблему.

Порівняно з .NET Core, який у TechEmpower Benchmarks показує близько 76 % результатів для швидкості запитів і тісно інтегрується з Azure для автоматичного масштабування й моніторингу, Node.js забезпечує швидше розгортання та гнучкий JavaScript-стек, але поступається ASP .NET Core у плані обчислювальної потужності для складних транзакцій.

1.6.1.3. Інші

У класичній сфері розробки корпоративних веб-застосунків Java-фреймворк Spring Boot та Python-фреймворк Django займають провідні позиції завдяки своїй перевіреній стабільності та багатому набору вбудованих інструментів. Spring Boot надає «стартер»-модулі та автоматичну конфігурацію, що значно спрощує розгортання мікросервісів у великих проєктах. Django пропонуючи ORM, систему аутентифікації, механізми захисту від атак та адміністративний інтерфейс із коробки, пришвидшує створення прототипів і скорочує час виходу MVP на ринок.

Перевагами Spring Boot є висока інтеграція з Java-екосистемою, масштабованість та оптимізація для ресурсомістких завдань у корпоративному сегменті, проте його артефакти часто мають збільшений розмір і потребують більше пам’яті, а адаптивна автоматична конфігурація іноді створює надлишок невикористаних залежностей.

Django забезпечує велику швидкість розробки завдяки єдиному стеку Python, інтуїтивному шаблону MVT і вбудованим засобам безпеки, але його синхронна модель обробки запитів

може стати вузьким горлом при високому рівні паралелізму, і для складних бізнес-логік доводиться застосовувати зовнішні рішення для масштабування.

Порівняно з ASP .NET Core і Node.js, Spring Boot та Django пропонують більш зрілі, усталені механізми розгортання та потужні засоби інфраструктурної інтеграції (наприклад, Actuator у Spring Boot і система міграцій у Django), але поступаються в «холодному старті» та масштабуванні за допомогою контейнерів, де .NET Core у бенчмарках демонструє вищу швидкість обробки REST-запитів, а Node.js - ефективну роботу з великими обсягами I/O-запитів завдяки неблокуючій архітектурі. Водночас вибір між цими технологіями слід базувати на характері проєкту: для CPU-bound навантажень перевагу матиме .NET Core, для I/O-орієнтованих сервісів - Node.js, а для швидкого створення MVP із багатим функціоналом «з коробки» - Spring Boot або Django.

1.6.1.4. Обґрунтування вибору .NET

Після аналізу різних платформ для розробки серверної частини веб-застосунку - включно з Java/Spring Boot, Python/Django та Node.js - платформа .NET Core (C# / ASP .NET Core) вирізняється як оптимальне рішення для даного проєкту. По-перше, .NET Core пропонує уніфікований стек» із чітко визначеними API, потужною підтримкою dependency injection та зрілими патернами обробки запитів, що значно спрощує побудову і підтримку багаторівневої архітектури. По-друге, висока продуктивність сервера Kestrel у поєднанні з асинхронною моделлю обробки запитів забезпечує надзвичайно низькі затримки й високу пропускну здатність навіть під піковими навантаженнями, що критично для систем, що повинні гарантовано обслуговувати великі потоки користувацьких звернень. Крім того, тісна інтеграція з екосистемою Microsoft Azure дозволяє майже «з коробки» налаштувати горизонтальне та вертикальне масштабування, моніторинг за допомогою Application Insights і автоматичне аварійне відновлення, що значно знижує операційні витрати на підтримку інфраструктури.

Критерій	.NET	Node.js	Spring Boot (Java)	Django (Python)
Продуктивність	Надзвичайно висока для REST-запитів, низькі затримки завдяки Kestrel і асинхронній моделі	Висока для I/O-інтенсивних операцій, але CPU-bound задачі блокують Event Loop	Висока стабільність у JVM, але більші затримки на «холодний старт» та вище споживання пам’яті	Середня продуктивність, синхронна модель потребує WSGI-серверів для паралелізму
Модель паралелізму	Асинхронна + багатопоточна через Thread Pool	Однопоточна, неблокуючий I/O	Багатопоточна JVM з оптимізацією через GC	Синхронна; паралельні запити обробляються через кілька процесів/воркерів
Масштабованість	Вбудоване авто-масштабування в Azure, легка контейнеризація	Легка контейнеризація та автоскейлінг у Kubernetes	Мікросервісна архітектура, Docker/Kubernetes	Швидке вертикальне масштабування через Docker/Kubernetes
Екосистема	NuGet-пакети, тісна інтеграція з екосистемою Microsoft	npm, широка спільнота	Maven/Gradle, модулі Spring Ecosystem	PyPI, вбудовані ORM, адмін-інтерфейс, засоби аутентифікації
Хмарна підтримка	Azure	AWS/GCP/Azure, serverless	AWS/GCP/Azure, Spring Cloud	AWS/GCP/Azure, Django-Channels, Heroku-ready

Таблиця 2. Порівняння мов програмування та середовищ виконання

Останнім чинником, що схилив вибір на користь .NET Core, стала глибока інтеграція з корпоративними СУБД, зокрема SQL Server. Використання Entity Framework Core забезпечує зручну розробку та оптимізовану роботу з реляційними даними, а вбудовані засоби безпеки забезпечують високу стійкість до атак. У результаті поєднання продуктивності, корпоративної підтримки й багатої екосистеми робить .NET Core найкращим вибором для реалізації високодоступного, масштабованого і безпечного багаторівневого веб-застосунку.

1.6.2. Фреймворки для клієнтської частини

Фреймворк клієнтської частини визначає модель рендерингу, маршрутизацію та механізми управління станом, що безпосередньо впливає на швидкість завантаження та взаємодію

користувача із застосунком. Серед провідних рішень - React, Angular та Vue - кожен підхід має свої особливості: React пропонує віртуальний DOM і компонентний підхід для високої продуктивності, Angular - повноцінний «все в одному» фреймворк з потужною CLI та інжекцією залежностей, а Vue - просту інтеграцію й м’яку криву навчання. Вибір клієнтського фреймворка обумовлюється рівнем інтерактивності інтерфейсу та наявністю готових рішень для серверного рендерингу й статичної генерації сторінок.

1.6.2.1. React

React - це декларативна бібліотека для побудови користувацьких інтерфейсів, розроблена компанією Facebook, яка застосовує компонентний підхід для організації UI та використовує концепцію віртуального DOM для оптимізації оновлень сторінки. Кожен компонент у React виголошує свій інтерфейс у вигляді функції або класу, що приймає вхідні властивості та стан, а потім повертає опис UI у вигляді елементів JavaScript-об’єктів (JSX), що підлягають процесу реконсиляції з реальним DOM. Віртуальний DOM дозволяє React мінімізувати прямі маніпуляції з браузерним DOM, зменшуючи кількість дорогих операцій рендерингу та забезпечуючи високий рівень продуктивності навіть у складних односторінкових застосунках.

Серед головних переваг React слід відзначити швидку реалізацію оновлень завдяки алгоритму «diffing», що виконує порівняння попереднього та нового опису UI й вносить лише мінімальні зміни в реальний DOM. Компонентний підхід сприяє локалізації логіки та повторному використанню фрагментів інтерфейсу, що полегшує підтримку та тестування коду. Однак механізм віртуального DOM може іноді створювати додаткові затримки через підрахунок змін (особливо при гучних оновленнях стану), а синтаксис JSX і підхід до управління станом потребують додаткового навчання та суворого дотримання патернів оптимізації.

React популярним вибором для створення високодинамічних та інтерактивних веб-інтерфейсів, оскільки його віртуальний DOM і компонентна архітектура забезпечують надзвичайно ефективне оновлення UI без зайвих операцій рендерингу. Водночас розробникам необхідно ретельно продумувати стратегії управління станом і оптимізації рендерингу, щоб уникнути надмірних diff-операцій і забезпечити передбачувану продуктивність застосунку.

1.6.2.2. Vue.js

Vue.js - це декларативний фреймворк для побудови інтерфейсів користувача, який поєднує компонентний підхід із гнучкою реактивною системою. Vue рендерить зміни через віртуальний DOM, мінімізуючи операції з браузерним DOM та забезпечуючи високу продуктивність оновлень інтерфейсу. Компоненти у Vue ізольовані за допомогою Options API або Composition API, що дає змогу чітко структурувати код і повторно використовувати логіку в різних частинах застосунку.

Серед ключових переваг Vue.js варто відзначити його низький поріг входження та зрозумілу синтаксисну обгортку над HTML, CSS та JavaScript, що істотно спрощує швидке прототипування навіть для розробників із мінімальним досвідом у фреймворках. Гнучка реактивна система з автоматичним відстеженням залежностей і оновленням компонентів забезпечує плавний досвід користувачів без зайвих рендерів. Також Vue пропонує потужні інструменти для оптимізації продуктивності, зокрема lazy-loading компонентів, оптимізацію списків із ключами та можливість ручного контролю над реактивністю. Водночас Vue.js може стикатися з проблемами в масштабованості дуже великих застосунків через потенційне накопичення надмірних реакцій та необхідність тонкого налаштування watch-методів. Перехід від Options API до Composition API вводить круту криву навчання, а надмірне застосування composables без суворої структури може призводити до хаотичної організації коду. Крім того, хоча екосистема Vue постійно зростає, вона досі менша за React, що іноді ускладнює пошук готових рішень або плагінів для специфічних завдань.

Vue.js поєднує простоту й потужність реактивного програмування з високою продуктивністю віртуального DOM, що робить його привабливим вибором для середніх і великих SPA-проєктів із помірною складністю бізнес-логіки. Водночас для застосунків із надзвичайно великими інтерфейсними деревами та жорсткими вимогами до затримок варто передбачити експерименти з інструментами профілювання та, можливо, підібрати інші підходи до управління станом і компонування, враховуючи висновки масштабних досліджень продуктивності фронтенд-фреймворків.

1.6.2.3. Angular

Angular - це повноцінний фронтенд-фреймворк від Google, який реалізує підхід «все в одному» для побудови великих та складних односторінкових застосунків (SPA). Він базується на архітектурі компонентів із суворою типізацією через TypeScript і застосовує двосторонній зв’язок даних, що дозволяє автоматично синхронізувати модель і представлення без додаткових зусиль зі сторони розробника. Angular надає набір вбудованих інструментів - від інтегрованої CLI-утиліти для генерації коду до суттєвих модулів для роботи з формами, маршрутизацією та HTTP-запитами, забезпечуючи єдину екосистему для всіх аспектів клієнтського програмування.

Серед основних переваг Angular слід відзначити наявність строгої структури проєкту й стандартизованих патернів, що підтримують однаковий стиль написання коду та полегшують масштабування великих командних проєктів. Вбудована система двостороннього зв’язку скорочує обсяг шаблонного коду для синхронізації UI та стану, а TypeScript дає змогу виявляти помилки на етапі компіляції й забезпечує надійну рефакторингову підтримку. Водночас діагностика змін у великих DOM-деревах може відбуватися із затримками, а порівняно велика вага стартового бандла негативно впливає на час першого завантаження, що потребує впровадження додаткових технік оптимізації («lazy loading», «AOT-компіляція»).

У сукупності, Angular є доцільним вибором для проєктів, які потребують суворої організації коду, широких функціональних можливостей «із коробки» та корпоративного рівня підтримки. Однак для невеликих або дуже динамічних застосунків, де критично важливий час завантаження та мінімальний обсяг клієнтського коду, може бути доцільнішим використання легших бібліотек на зразок React або Vue.js.

1.6.2.4. Обґрунтування вибору React

Проведений аналіз основних фронтенд-фреймворків показав, що React вирізняється завдяки своїй декларативній компонентній архітектурі та оптимізованому механізму оновлення віртуального DOM. Це дозволяє зосередитися на описі кінцевого стану інтерфейсу без детального контролю за низькорівневими маніпуляціями з реальним DOM, що суттєво зменшує число помилок і підвищує продуктивність при частих змінах даних.

Вибір React обґрунтований також його широкою підтримкою спільноти та індустрії: велика кількість пакетів і плагінів спрощує реалізацію складних UI-рішень, а офіційні інструменти («Create React App», «React DevTools») суттєво пришвидшують розробку та налагодження. Компонентний підхід у поєднанні з «TypeScript» або «PropTypes» забезпечує строгість типізації й раннє виявлення помилок, що особливо цінно в проєктах зі складною бізнес-логікою та високими вимогами до надійності. Таким чином, React поєднує високу продуктивність, гнучкість інтеграції та стійкість архітектурних рішень, що робить його оптимальним вибором для розробки масштабованого клієнтського інтерфейсу в межах даного дипломного проєкту.

Критерій	**React**	**Vue.js**	**Angular**
Модель рендерингу	Віртуальний DOM із реконсиляцією для мінімальних змін у реальному DOM (React)	Віртуальний DOM плюс реактивні проксії для автоматичного відстеження змін стану (vuejs.org)	Пряма маніпуляція DOM через власний механізм виявлення змін без віртуального DOM (Angular)
Управління станом	Хуки (useState, useReducer) або зовнішні бібліотеки (Redux, MobX)	Вбудована реактивність (ref, reactive) та Vuex/Pinia для складних випадків	Сервіси + RxJS; двостороння прив’язка через NgModel
Крива навчання	Середня: необхідно освоїти JSX і патерни композиції компонентів	Низька: зрозумілий синтаксис шаблонів та Options API; Composition API додає глибину	Висока: TypeScript, декоратори, сувора архітектура «все в одному»
Розмір бандлу	Мінімальний за базу, але залежить від додаткових бібліотек	Компактний; можна налаштувати вручну через збірщик	Досить великий стартовий бандл, вимагає AOT, lazy loading

Прив'язка даних	Односторонння (дані→відображення); для двосторонньої — додаткові бібліотеки	Одно- і двосторонння через v-bind / v-model	Двосторонння через ngModel; повний контроль потоку даних
Екосистема та інструменти	Create React App, Next.js, велика кількість пакетів npm	Vue CLI, Vite, Nuxt.js	Angular CLI, Angular Universal, RxJS
Оптимізація продуктивності	Профайлер, shouldComponentUpdate /PureComponent, memo	Lazy-loading компонентів, оптимізація списків із ключами	AOT-компіляція, екстракція шаблонів, кешування

Таблиця 3. Порівняння фреймворків

1.7. Системи управління базами даних для високої доступності

Управління базами даних для забезпечення відмовостійкості є критичною складовою архітектури будь-якої сучасної системи, оскільки саме від надійності зберігання й обробки даних залежить доступність сервісу та мінімізація бізнес-ризиків у разі апаратних чи програмних збоїв. Комплекс заходів-від синхронної чи асинхронної реплікації до автоматичного failover і використання хмарних SLA-дозволяє підтримувати час відновлення (RTO) та точку відновлення (RPO) на прийнятному рівні, забезпечуючи безперервність роботи й збереження цілісності інформації.

Вибір СУБД визначає те, як саме дані зберігатимуться, запитуватимуться й масштабуватимуться. Реляційні бази даних, такі як SQL Server або PostgreSQL, забезпечують строгі гарантії ACID-транзакцій та стандартну мову SQL, що полегшує виконання складних запитів і нормалізацію даних. Натомість NoSQL-системи (MongoDB, Cassandra) пропонують горизонтальне масштабування та гнучкі схеми, придатні для великих обсягів неструктурованих даних і сценаріїв із високою частотою записів. Правильний вибір СУБД враховує обсяг і тип даних, очікуваний рівень узгодженості та характеристики навантажень, щоб забезпечити оптимальне співвідношення продуктивності, доступності та складності адміністрування.

1.7.1. Реляційні СУБД (SQL)

Реляційні системи управління базами даних (RDBMS) є фундаментальним рішенням для зберігання та маніпуляції структурованими даними з гарантіями цілісності й узгодженості завдяки реалізації ACID-транзакцій. У RDBMS інформація організовується у вигляді таблиць із чіткими схемами, що дозволяє виконувати складні запити мовою SQL та забезпечує оптимізовану роботу з відносинами між різними наборами даних.

Найпоширеніші комерційні та відкриті рішення включають Microsoft SQL Server, PostgreSQL, MySQL і Oracle Database. Microsoft SQL Server відзначається глибокою інтеграцією з екосистемою Windows і Azure, використовуючи Transact-SQL для розширених можливостей запитів. PostgreSQL вирізняється дотриманням стандартів SQL, розширюваністю та підтримкою складних типів даних. MySQL та Oracle пропонують різні комерційні моделі: MySQL відрізняється простотою розгортання в open-source середовищі, тоді як Oracle забезпечує високий рівень корпоративної підтримки та масштабованість для дуже великих навантажень.

До переваг реляційних СУБД належать сувора схема даних, що гарантує узгодженість і цілісність, а також потужний мову SQL для реалізації складних бізнес-запитів та аналітики. Обмеженнями є відносна складність горизонтального масштабування (шардинг) та фіксована структура схеми, яка може бути недостатньо гнучкою для динамічно змінних або неструктурованих даних. Крім того, комерційні рішення, такі як SQL Server та Oracle, можуть передбачати значні витрати на ліцензування та експертну підтримку.

Реляційні СУБД залишаються оптимальним вибором для застосунків із чітко структурованими та взаємопов’язаними даними, що потребують сильних гарантій транзакційності та можливості складної звітності.

1.7.2. Нереляційні СУБД (NoSQL)

Нереляційні системи управління базами даних (NoSQL) були розроблені для роботи з великими обсягами даних, що не вписуються в звичну табличну структуру реляційних баз, та забезпечення горизонтального масштабування «в один клік». На відміну від традиційних SQL-СУБД, NoSQL-рішення відмовляються від жорстко фіксованої схеми й пропонують гнучкі моделі даних, що робить їх привабливими для застосунків із високою частотою

записів, неструктурованими об’єктами або вимогою обробки великих потоків користувацької активності.

Серед основних категорій NoSQL-СУБД виділяють:

1. Документоорієнтовані (MongoDB, Couchbase), які зберігають дані у форматах JSON або BSON і дозволяють гнучко змінювати структуру записів без складних міграцій схем.
2. Ключ-значення (Redis, DynamoDB), які оптимізовані під дуже швидкий доступ до даних за простими ключами та використовуються як кеш або «сховище сесій».
3. Колончаті (Apache Cassandra, HBase), що ефективно обробляють великі обсяги записів, розподілених по вузлах, за рахунок горизонтального шардингу та колоночної організації даних.
4. Графові (Neo4j, Amazon Neptune), призначені для моделювання та запитів складних зв’язків між сутностями, наприклад, у соціальних мережах або рекомендательных системах.

До ключових переваг NoSQL-підходу належать можливість гнучкого розгортання масштабованих кластерів із автоматичним розподілом даних, підтримка неструктурованих форматів без необхідності змінювати схему та висока швидкість записів. Проте нереляційні СУБД втрачають сильні сторони реляційних моделей, такі як повноцінні ACID-транзакції та складні JOIN-операції, через що реалізація транзакційного узгодження стає складнішою і часто вимагає зовнішніх механізмів або компромісів у консистентності. Додатково, різноманітність моделей даних і конфігурацій може ускладнювати ознайомлення розробників та адміністраторів з кожним окремим рішенням і підвищувати загальну складність операційного супроводу.

Висновуючи, NoSQL-СУБД слід обирати для сценаріїв із динамічною або неструктурованою інформацією, необхідністю горизонтального масштабування за обсягом записів та швидкістю реакції на операції вставки чи читання. При цьому для забезпечення необхідного рівня узгодженості та надійності даних варто звернути увагу на конкретну модель NoSQL-СУБД і можливості її конфігурації, оскільки компроміси в сфері транзакційної цілісності можуть вплинути на архітектурні рішення в цілому.

1.7.3. Основні механізми відмовостійкості

У сучасних розподілених системах управління даними забезпечення відмовостійкості ґрунтується на поєднанні синхронної та асинхронної реплікації з автоматичним перемиканням, що дозволяє підтримувати високу доступність сервісу навіть за умов виходу з ладу окремих вузлів. Синхронна реплікація гарантує, що кожна транзакція фіксується одночасно на головному і резервному серверах, забезпечуючи жорстку консистентність, однак вона збільшує час відповіді через очікування підтверджень. Асинхронна реплікація зменшує затримки запису, копіюючи зміни із затримкою й допускаючи тимчасові розбіжності між вузлами, що оптимально для систем із високими вимогами до продуктивності. Вибір топології Active–Passive, де один вузол обробляє запити, а інший чекає готовим до ролі первинного, забезпечує простоту реалізації й відмовостійкості, тоді як Active–Active дозволяє одночасне обслуговування запитів усіма вузлами та підвищує масштабованість, але вимагає розв’язання конфліктів при одночасних записах. Кластеризація у «shared-nothing» архітектурі, яка розподіляє дані між незалежними вузлами без спільного доступу до пам’яті чи сховища, усуває єдину точку відмови та забезпечує ефективне горизонтальне масштабування через шардинг даних.

Автоматичне перемикання (failover) реалізується через безперервний моніторинг стану вузлів і негайне перенаправлення клієнтського трафіку на резервну репліку без участі оператора, що суттєво скорочує час простою. У AWS Multi-AZ розгортання SLA 99,99 % гарантує, що в разі збою первинного інстансу дані автоматично дублюються в іншій зоні доступності і сервіс перемикається на стендбай-репліку. Amazon Aurora Global Database розширює цю модель географічно, підтримуючи RPO менш ніж секунду між регіонами та автоматичний глобальний failover. Microsoft Azure пропонує аналогічні функції в Azure SQL Database та Managed Instance через Auto-Failover групи, які зберігають незмінні кінцеві точки підключення під час геоперемикань та забезпечують SLA до 99,995 % за допомогою зональної та регіональної надмірності. Такий підхід формує комплексний автоматизований механізм, що гарантує безперервний доступ до даних і відповідність суворим бізнес-вимогам до часу відновлення та точки відновлення.

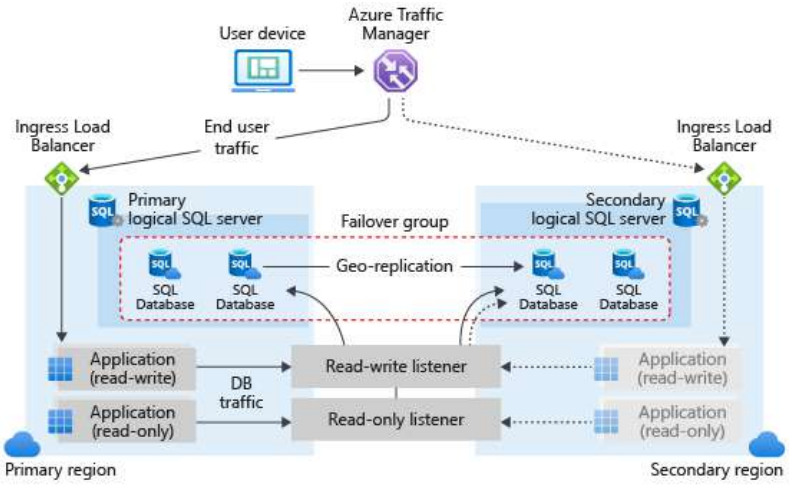


Рис. 6 Діаграма архітектури для максимальної відмовостійкості

1.7.4. Резервне копіювання та відновлення

Управління резервним копіюванням і відновленням даних є невід'ємною складовою забезпечення відмовостійкості, адже саме ці процеси дозволяють відновити працюючий стан системи після критичних збоїв чи втрати інформації. Підхід, побудований за принципом «3-2-1» - три копії даних, два різні носії та щонайменше одна off-site копія - забезпечує необхідний рівень застереження від одночасного виходу з ладу кількох середовищ зберігання. Використання знімкових (snapshot) бекапів дає змогу швидко фіксувати стан бази в конкретний момент часу, а операційні журнали транзакцій (transaction log backups) гарантують відновлення до заданої точки, що дозволяє мінімізувати втрати даних і скоротити RPO (Recovery Point Objective).

Автоматизація перевірки коректності резервних копій і регулярне виконання тестових процедур відновлення критично важливі для підтвердження працездатності бекапів та відповідності бізнес-вимогам щодо часу відновлення (RTO). Інтеграція з DevOps-пайплайнами та постановка alert-систем для повідомлень про невдалі операції резервного копіювання дозволяють своєчасно виявляти проблеми й усувати їх ще до настання аварійних ситуацій. Сучасні хмарні платформи, такі як Azure Backup, AWS Backup або Google Cloud Backup, надають вбудовані механізми знімків та журналювання, а також можливості політик «long-term retention» і «immutable backups», що підвищує рівень безпеки даних та спрощує виконання регуляторних вимог. Завдяки комплексному поєднанню цих технологій організація здатна забезпечити надійне зберігання, швидке відновлення та відповідність найвищим стандартам бізнес-безпеки.

1.7.5. Обґрунтування вибору SQL Server

У проєкті Microsoft SQL Server було обрано як основну реляційну СУБД через його тісну інтеграцію з екосистемою .NET Core і Azure, що спрощує налаштування безперервного розгортання, моніторингу та аварійного відновлення. Завдяки глибокій інтеграції з Visual Studio та Entity Framework Core, SQL Server забезпечує зручні інструменти для швидкого створення моделей даних зі схеми бази даних і навпаки. Крім того, платформа підтримує вбудовані механізми високої доступності, резервного копіювання «на гарячій» базі та розширені можливості безпеки – шифрування даних «at rest» і «in transit», а також контроль на рівні рядка. У порівнянні з відкритими рішеннями, такими як PostgreSQL або MySQL, SQL Server пропонує найвищу продуктивність у сценаріях із великими транзакційними навантаженнями завдяки автоматичному паралелізму запитів та оптимізованим алгоритмам індексації.

Критерій	SQL Server	PostgreSQL	MySQL	Oracle DB
Продуктивність	Автопаралелізм, висока TPS	Оптимальний для складних запитів	Швидкий CRUD	Висока для великих БД
Відмовостійкість	Always On AG з автоматичним failover	Синхронна/асинхронна реплікація	Master–slave, Galera-Cluster	Data Guard, RAC
Масштабованість	Шардінг	Шардінг	Реплікація та кластери	RAC, Sharding
Вартість	Дорога ліцензія	Безкоштовний	Безкоштовний	Дорога ліцензія
Інструменти	SSMS, Azure Data Studio, BI	pgAdmin, psql, розширення	Workbench, Shell	OEM, SQL Developer
Хмарна інтеграція	Azure SQL MI, Geo-реплікація	RDS/Aurora, Cloud SQL	RDS/Aurora, Cloud SQL	Autonomous DB, OCI DB Systems

Таблиця 4. Порівняння СУБД

1.8. Практики моніторингу та усунення несправностей у хмарному середовищі

У сучасних хмарних середовищах забезпечення стабільної роботи веб-застосунків вимагає впровадження ефективних практик моніторингу та усунення несправностей. Це включає систематичне спостереження за станом системи, виявлення аномалій, своєчасне реагування на інциденти та оптимізацію продуктивності.

Логування є фундаментальним компонентом моніторингу, що передбачає запис подій, помилок та інших важливих дій системи. Ці журнали дозволяють аналізувати поведінку застосунку, виявляти причини збоїв та відстежувати безпекові інциденти. Сучасні інструменти, такі як Amazon CloudWatch та Azure Monitor, забезпечують централізоване збирання, зберігання та візуалізацію логів, що спрощує процес аналізу та прийняття рішень.

Для забезпечення оперативного реагування на критичні події використовуються автоматичні сповіщення. Вони налаштовуються на основі визначених порогових значень або специфічних умов, таких як перевищення допустимого навантаження на процесор чи невдала спроба автентифікації. При спрацьовуванні таких умов системи надсилають сповіщення відповідальним особам через електронну пошту, SMS або інші канали зв'язку, що дозволяє швидко вжити необхідних заходів для усунення проблеми.

Системи моніторингу продуктивності застосунків (APM) надають глибоке розуміння роботи веб-застосунків, відстежуючи такі показники, як час відгуку, використання ресурсів та частоту помилок. Інструменти APM, такі як New Relic, AppDynamics та Dynatrace, дозволяють виявляти вузькі місця, аналізувати транзакції та забезпечувати оптимальний користувацький досвід. Використання APM сприяє проактивному підходу до управління продуктивністю, дозволяючи передбачати потенційні проблеми та запобігати їх виникненню.

Процес моніторингу та реагування на інциденти у хмарному середовищі можна представити у вигляді циклу, що складається з кількох етапів:

1. Виявлення інциденту: Цей етап передбачає безперервний моніторинг хмарної інфраструктури для виявлення аномалій або підозрілих дій. Використання систем управління інформацією та подіями безпеки (SIEM) дозволяє централізовано збирати та аналізувати журнали подій, що сприяє швидкому виявленню потенційних загроз.

2. Оцінка та класифікація: Після виявлення інциденту необхідно оцінити його вплив на систему та класифікувати за рівнем критичності. Це допомагає визначити пріоритетність реагування та залучити відповідні ресурси для вирішення проблеми.

3. Сповіщення та ескалація: На цьому етапі відповідальні особи інформуються про інцидент через автоматизовані сповіщення, такі як електронна пошта або SMS. У разі серйозних інцидентів може бути необхідна ескалація до вищого рівня управління або залучення додаткових фахівців.

4. Реагування та усунення: Команда реагування на інциденти аналізує причини виникнення проблеми та вживає заходів для її усунення. Це може включати ізоляцію уражених компонентів, застосування патчів або відновлення системи з резервних копій.

5. Відновлення та відстеження: Після усунення інциденту проводиться відновлення нормальної роботи системи та моніторинг для запобігання повторенню подібних ситуацій. Аналіз інциденту дозволяє виявити слабкі місця та вдосконалити заходи безпеки.

Використання сучасних інструментів моніторингу та реагування на інциденти, таких як рішення для виявлення та реагування на хмарні загрози (CDR), забезпечує комплексне управління інцидентами від їх виявлення до повного усунення. Це сприяє швидкій нейтралізації загроз з високою точністю та ефективністю.

Впровадження таких практик є критично важливим для підтримки високої доступності та безпеки хмарних веб-застосунків, забезпечуючи надійність та довіру користувачів до сервісів.

1.9. Висновки до розділу 1

У цьому розділі було розглянуто основні підходи до забезпечення високої доступності та відмовостійкості веб-застосунків у хмарному середовищі. Описано ключові архітектурні рішення, зокрема мікросервісну архітектуру, реплікацію баз даних та балансування навантаження, що сприяють стійкості системи до збоїв. Також розглянуто методи масштабування серверних ресурсів, включаючи горизонтальне та вертикальне масштабування, що дозволяють динамічно адаптувати ресурси до поточного навантаження. Використання автоматизованих механізмів резервного копіювання, як-от інкрементальне та диференційне копіювання, забезпечує безпеку даних та можливість швидкого відновлення у разі аварійних ситуацій. Таким чином, впровадження цих підходів дає змогу створювати надійні та продуктивні системи, здатні працювати безперебійно у змінних умовах експлуатації.

Розділ 2. Проектування та реалізація веб-застосунку

2.1 Визначення вимог до системи

Перед безпосередньою розробкою платформи для пошуку та усиновлення бездомних тварин проводиться детальне формулювання вимог, яке лягає в основу архітектурного й функціонального дизайну. Чітко визначені функціональні та нефункціональні вимоги забезпечують цілісність рішення, мінімізують ризики та гарантують відповідність кінцевого продукту очікуванням користувачів і бізнес-цілям.

2.1.1 Функціональні вимоги

Для коректного визначення масштабу та можливостей платформи необхідно сформулювати набір функціональних вимог, які безпосередньо відображають бізнес-логіку майданчика для пошуку та усиновлення бездомних тварин. Нижче наведено перелік ключових операцій, які повинні бути реалізовані у вигляді окремих функцій системи.

- Реєстрація та автентифікація користувачів із розподілом на ролі з використанням Cookie.
- Створення, редагування, видалення та перегляд оголошень про тварин з атрибутами: фото, вік, стан здоров'я, контактна інформація, теги.
- Пошук та фільтрація оголошень за породою, віком та іншими тегами.
- Обмін запитами на усиновлення, включно з історією повідомлень між користувачами.
- Можливість переглядати активності пов'язані з тваринами.
- Можливість переглядати актуальні благодійні збори коштів.
- Можливість спілкуватися з іншими користувачами за допомогою внутрішнього чату.

Реалізація цих функцій забезпечить основний користувацький сценарій роботи з майданчиком, що дозволить зручно публікувати інформацію про тварин, швидко знаходити відповідні оголошення та безпечно здійснювати процес усиновлення.

2.1.2 Нефункціональні вимоги

Нефункціональні вимоги визначають якість обслуговування та технічні характеристики системи, які не залежать від конкретних бізнес-процесів, але критично впливають на досвід користувачів і надійність платформи в цілому.

- Забезпечення Uptime ≥ 99,9 % через багаторегіональні репліки та автоматичне масштабування компонентів.
- Середній час відповіді API ≤ 200 мс, піковий час обробки запиту ≤ 2 с для 95 % запитів.
- Дотримання рекомендацій OWASP Top 10, шифрування даних, захист від SQL-ін'єкцій і CSRF.

- Автоматичне failover для сервісів і баз даних з RPO ≤ 5 хв та RTO ≤ 2 хв.
- Наявність модульних та інтеграційних тестів з охопленням ≥ 80 % критичних функцій.
- CI/CD-конвеєр: безперервна інтеграція й розгортання через GitHub Actions.

Забезпечення вищевказаних нефункціональних характеристик є основою для побудови стійкого, безпечного і зручного у використанні веб-застосунку, здатного витримувати великі навантаження і швидко відновлюватися після можливих збоїв.

2.2 Архітектурний дизайн застосунку

Застосунок спроєктовано з використанням багаторівневої архітектури, що сприяє чіткому розділенню відповідальностей, полегшує підтримку та масштабування системи. Архітектура складається з трьох основних рівнів: представлення, бізнес-логіки та доступу до даних.

Рівень представлення об’єднує дві частини – фронтенд і бекенд. Фронтенд реалізовано на React: він відповідає за взаємодію з користувачем, відображення інтерфейсу та відправку запитів до бекенду, обробляючи введення та дані, отримані з API. Бекенд виконано на ASP .NET Core у вигляді REST-контролерів, доступних за адресами api/[controller]/[action]; контролери приймають HTTP-запити, виконують базову валідацію і передають дані на рівень бізнес-логіки.

Рівень бізнес-логіки побудовано навколо сервісів, які реалізують усі ключові правила роботи платформи: вони перетворюють DTO з контролерів у доменні об’єкти, застосовують бізнес-правила та взаємодіють із шаром доступу до даних для зчитування чи збереження інформації. Сервіси також можуть звертатися до зовнішніх API та черг повідомлень для асинхронних процесів.

Рівень доступу до даних представлено репозиторіями, що абстрагують усю низькорівневу роботу з базою: вони надають методи CRUD, а всередині використовують Entity Framework Core у моделі Code-First із налаштованими міграціями. Контекст EF Core містить визначення моделей і їхнє мапінгування на таблиці SQL Server. Окрім того, репозиторії інтегруються з Azure Table Storage для логування й Azure Blob Storage для зберігання файлів.

Такий чіткий розподіл на рівні гарантує, що кожен шар можна розвивати, тестувати й масштабувати незалежно, зберігаючи при цьому цілісність бізнес-логіки та високу надійність системи в цілому.

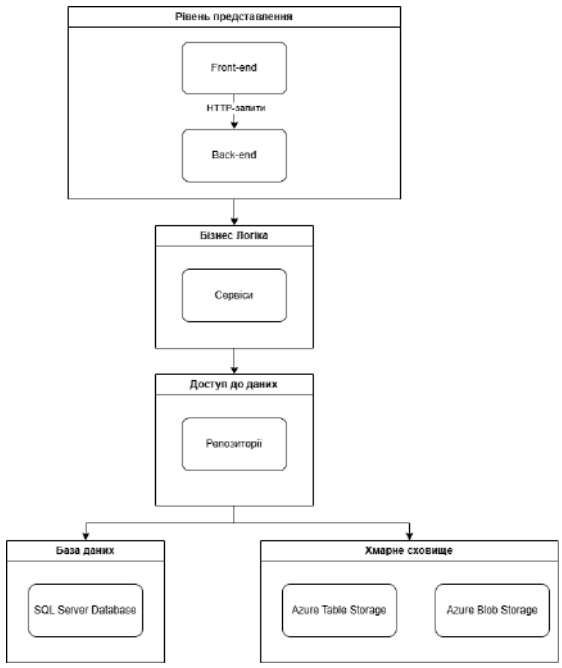


Рис. 7 Діаграма архітектури API застосунку .NET

2.3 Проєктування моделі даних

Модель даних платформи PetFinder створена за підходом Code-First у Entity Framework Core з використанням SQL Server як основної реляційної СУБД. В основі стоїть набір сутностей, що відображають ключові доменні об’єкти: користувачів (AspNetUsers + UserProfile), тварин (Animals), тегів (Tags), проміжної таблиці зв’язків AnimalTag, посилань на зображення (AnimalPictureLink), чатів (Chats) та повідомлень (Messages), а також додаткових сутностей Activity і Donation. Кожна сутність має свій первинний ключ PK_Id (Id типу uniqueidentifier) і, за потреби, зовнішні ключі FK_IdentityId, FK_UserProfileId, FK_AnimalId тощо. Відношення «один-до-багатьох» і «багато-до-багатьох» реалізовані через відповідні навігаційні властивості та проміжні колекції, що ілюструє наведена ER-діаграма (Мал.).

Для оптимізації продуктивності запитів до SQL Server застосовано кластеризовані індекси на первинних ключах кожної таблиці та некластеризовані індекси на стовпцях зовнішніх ключів. Додаткові некластеризовані індекси рекомендовано на полях, що використовуються у фільтрах (Animals.Type, AnimalTag.TagId, UserProfile.Email), для зменшення витрат на JOIN-операції та прискорення пошуку. Щодо організації високої доступності, обрано Always On Availability Groups із конфігурацією мультизональних реплік: первинна репліка виконує запис, а вторинні – читають дані у режимі Read-Only, що дозволяє відокремити пікові запити на читання й забезпечити автоматичний failover за неполадок із RTO ≤ 2 хв. Такий підхід

поєднує гнучкість горизонтального масштабування, низькі затримки читання та гарантовану стійкість до відмов.

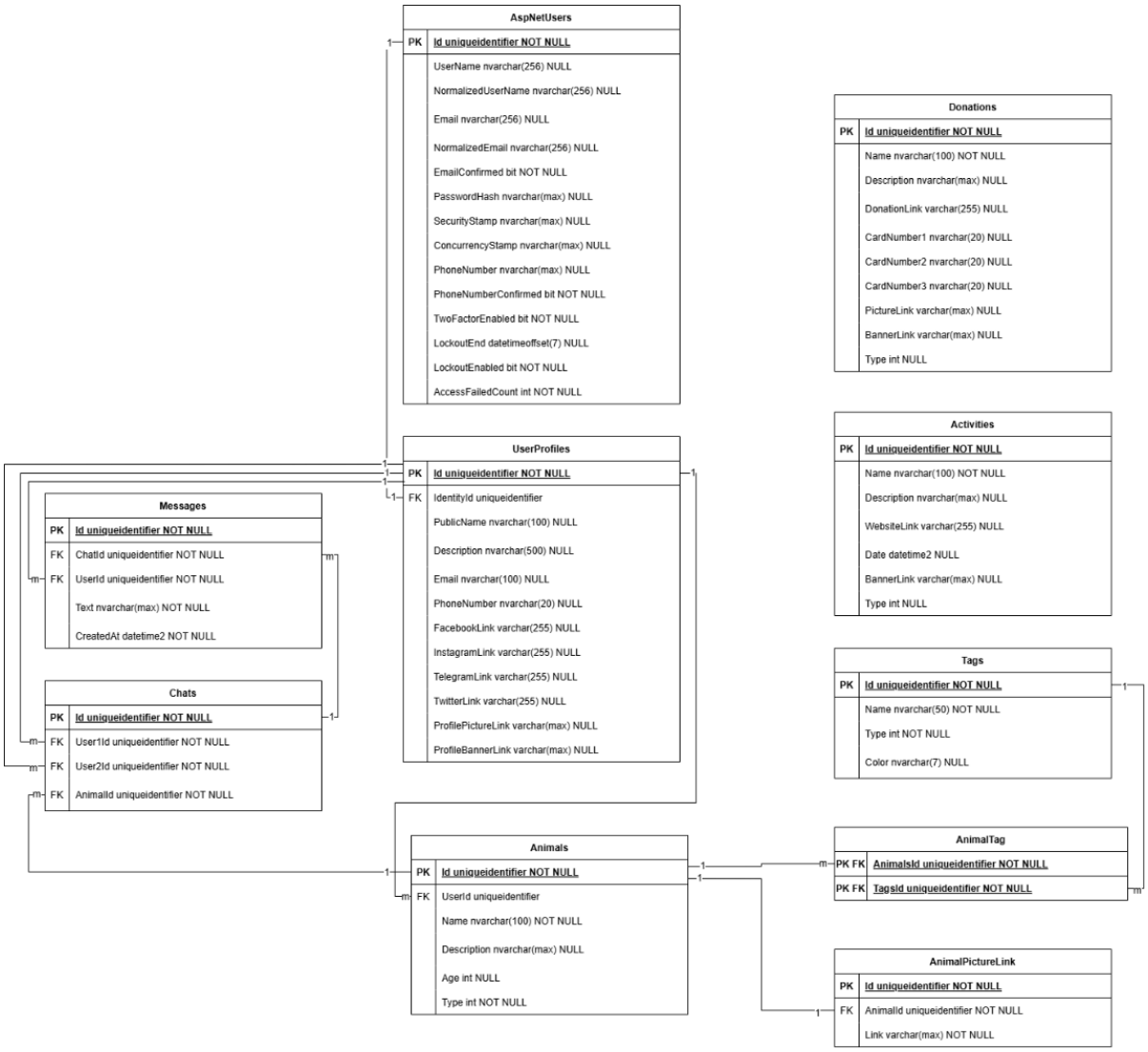


Рис. 8 Діаграма архітектури бази даних

2.4 Реалізація серверної частини на ASP.NET Core

2.4.1 Структура проєкту

Серверна частина застосунку "PetFinder" розроблена на платформі ASP.NET Core 8 і має чітко визначену структуру для підтримки принципів розділення відповідальностей та полегшення подальшої розробки та тестування.

Структура проєкту .NET:

- Директорія Controllers: Містить класи контролерів API (AnimalController.cs, UserProfileController.cs, AuthorizationController.cs, ChatController.cs тощо). Контролери відповідають за обробку HTTP-запитів, валідацію вхідних даних (DTO), виклик відповідних сервісів бізнес-логіки та формування HTTP-відповідей.
- Директорія Services: Містить інтерфейси та реалізації сервісів, що інкапсулюють бізнес-логіку застосунку. Кожен сервіс відповідає за певну функціональну область (наприклад, AnimalService, UserProfileService, ChatService, BlobStorageService, TableService). Вони взаємодіють з рівнем доступу до даних (репозиторіями) та іншими сервісами.
- Директорія Services/AnimalService, Services/UserProfileService: директорії для організації конкретних сервісів бізнес-логіки, кожен із яких інкапсулює операції з відповідної предметної області та звертається до репозиторіїв або інших сервісів.
- Директорія Services/AzureBlobService, Services/AzureTableService: містить сервіси для взаємодії з Azure Storage - AzureBlobService для роботи з Blob Storage та AzureTableService для запису й читання логів в Table Storage.
- Директорія DataLayer: містить інтерфейси та реалізації репозиторіїв, що абстрагують доступ до джерел даних. Кожен репозиторій використовує DbContext для операцій над базою даних.
- Директорія Data/Models: класи сутностей EF Core, які описують структуру таблиць у SQL Server.
- Директорія Data/Mapping: класи конфігурації що задають деталі мапінгу між C#-класами та таблицями бази даних.
- Директорія Migrations: файли міграцій EF Core, які відображають історію змін у моделі даних.

- Директорія Models/DTO: об’єкти передачі даних між шарами та клієнтом, що відділяють внутрішню структуру сутностей від зовнішнього API.
- Директорія Models/Hubs: класи SignalR, які відповідають за real-time комунікацію між клієнтами та сервером.
- Директорія Models/TableModels: моделі для роботи з Azure Table Storage, що визначають схему рядків у таблицях логів.

- Файл MappingProfiles.cs: конфігурація AutoMapper, яка автоматизує перетворення між DTO та сутностями бази даних.
- Директорія Constants: класи з константами, які централізовано зберігають налаштування та шаблони.
- Директорія Helpers: допоміжні утиліти та розширення, що використовуються по всьому проєкту.
- Файл Program.cs: точка входу в застосунок, де конфігуруються сервіси, middleware, CORS та маршрутизація.
- Файл appsettings.json: конфігурація застосунку.
- Файл PetFinder.csproj: файл проєкту, який містить залежності NuGet та налаштування збірки.

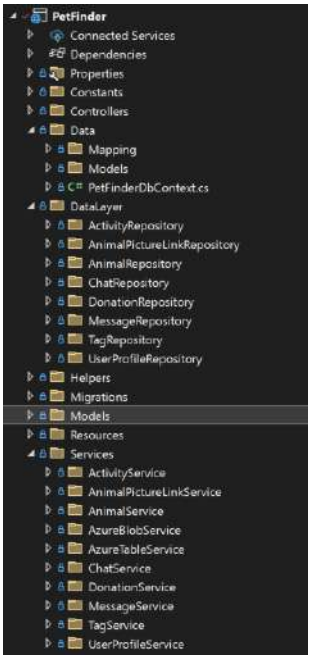


Рис. 9 Структура проєкту .NET

Основна конфігурація знаходиться у файлі Program.cs, який використовує мінімальний підхід до конфігурації, запроваджений в .NET 6+.

В ньому ми виконуємо такі налаштування:

- Налаштовується політика CORS для дозволу запитів з певних джерел.
- Реєструється PetFinderDbContext з використанням SQL Server та рядка підключення з конфігурації.
- Реєструються всі репозиторії та сервіси з життєвим циклом Scoped, що дозволяє інжектувати їх у контролери та інші сервіси.

- Реєструються клієнти для Azure Blob Storage та Table Storage.
- Додаються сервіси для підтримки контролерів API.
- Додаються сервіси для генерації документації API коли ми запускаємо проєкт локально.
- Налаштовується ASP.NET Core Identity для автентифікації та авторизації, використовуючи PetFinderDbContext для зберігання даних користувачів та ролей.
- Додаються сервіси SignalR та інтеграція з Azure SignalR Service.
- Конфігуруються параметри cookie для автентифікації, включаючи шляхи для логіну/логуауту, час життя cookie та налаштування безпеки.
- Реєструється AutoMapper для сканування збірки та знаходження профілів мапінгу.
- Налаштовується інтеграція з Azure Application Insights для моніторингу та телеметрії.
  Також ми налаштовуємо middleware:
- Налаштовується обробка винятків: сторінка для розробників у середовищі розробки та глобальний обробник для «production»-версії, який повертає JSON з повідомленням про помилку .
- Включається Swagger UI у середовищі розробки для тестування та документації API.
- Включається перенаправлення HTTP-запитів на HTTPS.
- Включається система маршрутизації ASP.NET Core.
- Застосовується налаштована раніше політика CORS.
- Включаються middleware для автентифікації та авторизації, які працюють на основі налаштувань Identity.
- Створюються кінцеві точки API контролерів та SignalR хаб на відповідні маршрути .
  Така структура та конфігурація забезпечують модульність, тестовність та масштабованість серверної частини застосунку, інтегруючи необхідні сервіси для роботи з даними, автентифікацією, логуванням та взаємодією в реальному часі.

2.4.2 REST-API контролери

Контролери в ASP.NET Core виступають як точки входу для HTTP-запитів від клієнта. Вони відповідають за:

- Маршрутизацію: Визначення URL-адрес, «ендпоінтів», за якими доступні певні дії.
- Прив'язку моделі: Автоматичне перетворення даних із запиту в об'єкти C#.

- Валідацію: Перевірка коректності вхідних даних за допомогою атрибутів валідації в DTO та перевірки ModelState.IsValid.
- Авторизацію: Перевірка прав доступу користувача до певних ресурсів або дій за допомогою атрибута [Authorize] та додаткових перевірок всередині методів.
- Виклик сервісного шару: Делегування виконання бізнес-логіки відповідним сервісам.
- Формування відповіді: Повернення HTTP-статусів (200 OK, 201 Created, 204 No Content, 400 Bad Request, 401 Unauthorized, 404 Not Found, 500 Internal Server Error) та даних (DTO) клієнту.

Тепер давайте розглянемо один з методів контролера який відповідає за видалення об’яви про тварину:

```
// DELETE: api/Animals/{id}
[Authorize] // 1. Авторизація: Доступ тільки для автентифікованих користувачів
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteAnimal(Guid id) // Приймає ID з URL
{
    // 2. Логування запиту: Фіксація спроби видалення
    await _tableService.RequestLogs(logEntity: new Models.TableModels.GenericLogEntity
    {
        Message = $"DeleteAnimal request for id: {id}",
        User = User.Identity?.Name ?? "Anonymous",
        Parameter = $"id={id}"
    });
    try // 3. Обробка винятків: Захист від непередбачених помилок
    {
        // 4. Отримання ID користувача: Визначення поточного користувача, що робить запит
        Guid userId = await _userProfileService.GetUserProfileIdByAspNetUserIdAsync(id: Guid.Parse(input: User.FindFirst(type: ClaimTypes.NameIdentifier)?.Value));
        // 5. Отримання ролей користувача
        List<Claim> roleClaims = HttpContext.User.FindAll(type: ClaimTypes.Role).ToList();
        // 6. Отримання даних для перевірки
        var animalToDelete = await _animalService.GetAnimalByIdAsync(id);
        // 6.1 Перевірка існування
        if (animalToDelete == null)
        {
            // Логування спроби видалення неіснуючого об'єкта
            await _tableService.LogError(logEntity: new Models.TableModels.GenericLogEntity { Message = $"Attempt to delete non-existent animal with id {id} by user
              {User.Identity?.Name}" });
            return NotFound(value: $"Animal with id {id} not found."); // Повернення 404 Not Found
        }

        // 7. Перевірка прав доступу
        if (animalToDelete.UserId != userId && !roleClaims.Select(selector: x => x.Value).Contains(value: "Admin"))
        {
            // 8. Логування несанкціонованої спроби
            await _tableService.LogError(logEntity: new Models.TableModels.GenericLogEntity
            {
                Message = $"Unauthorized attempt to delete animal with id {id}",
                User = User.Identity?.Name ?? "Anonymous",
                Parameter = $"id={id}"
            });

            return Unauthorized(); // Повернення 401 Unauthorized
        }

        // 9. Виклик сервісного шару
        await _animalService.DeleteAnimalAsync(id);
        return Ok(); // Повернення 200 Success при успішному видаленні
    }
    catch (Exception ex) // 10. Перехоплення та логування помилок
    {
        await _tableService.LogError(logEntity: new Models.TableModels.GenericLogEntity
        {
            Message = $"Error deleting animal with id {id}: {ex.Message}",
            StackTrace = ex.StackTrace,
            User = User.Identity?.Name ?? "Anonymous",
            Parameter = $"id={id}"
        });
        // Повернення 500 Internal Server Error
        return StatusCode(statusCode: StatusCodes.Status500InternalServerError, value: "An internal server error occurred while deleting the animal.");
    }
}
```

Рис. 10 Код методу контролеру AnimalController.cs

2.4.3 Сервісний шар

Сервісний шар є ключовим компонентом архітектури бекенду, що відповідає за інкапсуляцію та реалізацію бізнес-логіки застосунку. Він виступає як посередник між контролерами та рівнем доступу до даних, забезпечуючи чітке розділення відповідальностей.

Призначення сервісного шару:

1. Інкапсуляція бізнес-логіки: Містить правила та процеси, специфічні для доменної області (наприклад, валідація даних перед збереженням, розрахунки, координація кількох операцій з даними).
2. Абстракція над доступом до даних: Приховує деталі взаємодії з базою даних від контролерів. Контролери працюють з сервісами, не знаючи, як саме дані зберігаються чи витягуються.
3. Координація операцій: Може викликати методи кількох репозиторіїв або інших сервісів для виконання складних бізнес-транзакцій.
4. Повторне використання логіки: Логіка, визначена в сервісі, може бути використана різними контролерами або іншими сервісами.

Тепер розглянемо AnimalService.cs, сервіс для роботи з тваринами реалізує відповідний інтерфейс. За допомогою «Dependency Injection» сервіс отримує необхідні компоненти: один для взаємодії з базою даних, інший для логування в хмарне сховище, та ще один для доступу до даних поточного HTTP-запиту. Сервіс надає асинхронні методи для основних CRUD, отримання оголошень за ідентифікатором користувача та синхронний метод для фільтрації. Кожен метод делегує виконання відповідному методу компонента доступу до даних. Важливою частиною є обробка помилок: кожен публічний метод обгорнутий у try-catch; при виникненні винятку викликається приватний метод для детального логування помилки через компонент логування, після чого виняток повторно кидається для обробки на вищому рівні. Використання async/await забезпечує ефективну роботу з асинхронними операціями вводу/виводу.

Тепер розглянемо метод цього сервісу:

```
// Метод оновлення з доданою бізнес-логікою (перевірка існування)
0 references | 0 changes | 0 authors, 0 changes | 0 exceptions, - live
public async Task UpdateAnimalWithChecksAsync(Animal animal)
{
    string operation = $"UpdateAnimalWithChecksAsync for animal: {animal.Name} (ID: {animal.Id})";
    try
    {
        // 1. Бізнес-логіка: Перевірка існування тварини перед оновленням
        var existingAnimal = await _animalRepository.GetByIdAsync(id: animal.Id);
        if (existingAnimal == null)
        {
            // Якщо тварина не знайдена, кидаємо специфічний виняток або обробляємо інакше
            // Це дозволяє контролеру повернути 404 Not Found замість загальної помилки 500
            throw new KeyNotFoundException(message: $"Animal with ID {animal.Id} not found for update.");
        }

        // 2. Бізнес-логіка: Заборона зміни власника тварини через цей метод
        if (existingAnimal.UserId != animal.UserId)
        {
            throw new InvalidOperationException(message: "Changing the owner of the animal is not allowed through this method.");
        }

        // 3. Бізнес-логіка: Оновлення дати останньої модифікації
        animal.LastModifiedDate = DateTime.UtcNow;

        // 4. Виклик репозиторію: Делегування збереження оновлених даних
        await _animalRepository.UpdateAsync(animal);
    }
    catch (Exception ex)
    {
        //5. Логування помилки: Запис інформації про помилку
        await LogServiceError(operation, ex, parameters: $"animalId={animal.Id}");
        // 6. Повторне кидання винятку: Передача помилки на вищий рівень
        throw;
    }
}
```

Рис. 11 Код методу в сервісі AnimalService.cs

2.4.4 Доступ до даних

Репозиторій реалізує відповідний інтерфейс і слугує шаром доступу до даних для сутностей тварин. Його основне завдання - інкапсулювати логіку взаємодії з базою даних за допомогою ORM. Через Dependency Injection компонент отримує екземпляр контексту бази даних для виконання запитів, компонент для логування помилок у хмарне сховище, та компонент для отримання контексту користувача при логуванні. Компонент надає асинхронні методи для стандартних CRUD-операцій, отримання тварин за ідентифікатором користувача, а також синхронний метод для фільтрації та пагінації. Методи використовують контекст бази даних та LINQ для формування запитів, включаючи завантаження пов'язаних даних та оптимізацію для читання. Кожна операція з базою даних обгорнута в блок try-catch. У разі виникнення винятку, він логується за допомогою приватних методів, після чого виняток повторно кидається для обробки сервісним шаром. Більшість методів є асинхронними, що оптимізує роботу з операціями вводу/виводу бази даних. Методи додавання та оновлення містять специфічну логіку для коректної обробки зв'язків "багато-до-багатьох" з іншими сутностями.

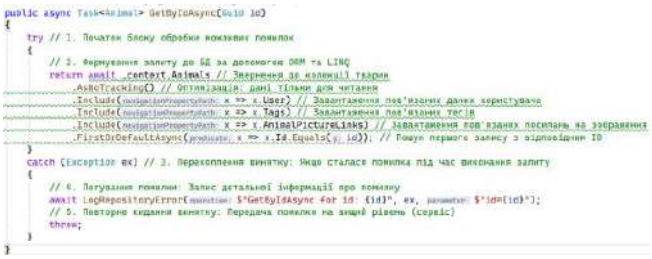


Рис. 12 Код методу в репозиторії AnimalRepository.cs

2.5 Реалізація клієнтської частини на React

2.5.1 Ініціалізація проєкту

Клієнтська частина застосунку "PetFinder" реалізована як односторінковий застосунок з використанням бібліотеки React. Для ініціалізації та базової конфігурації проєкту було використано стандартний інструмент Create React App (CRA).

Використання Create React App дозволило швидко налаштувати середовище розробки без необхідності ручної конфігурації інструментів збірки та розробки, таких як Webpack, Babel та ESLint. CRA надає готову структуру проєкту та набір скриптів для основних задач розробки.

Додаються такі команди і скрипти для них:

- «npm start»: Запускає сервер для розробки, зазвичай на http://localhost:3000, який автоматично перезавантажує сторінку при змінах у коді та відображає помилки в консолі.
- «npm test»: Запускає середовище для тестування компонентів.

- «npm run build»: Створює оптимізовану версію застосунку для розгортання у папці build/.

2.5.2 Структура проєкту

Структура клієнтської частини застосунку "PetFinder", створеної за допомогою Create React App, організована наступним чином для забезпечення модульності та легкості підтримки.

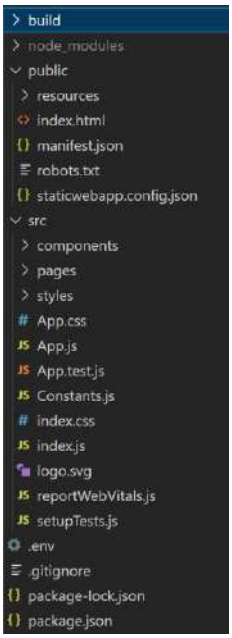


Рис. 13 Структура проєкту React

Є 3 основних директорії public, src і build, давайте розглянемо їх детальніше.

«public/» - ця директорія містить статичні файли, які не обробляються системою збірки Webpack. В неї входять ось таки файли та директорії:

- «index.html»: Основний HTML-файл, в який монтується React-додаток. Він містить кореневий DOM-вузол.
- «manifest.json»: Файл маніфесту для Progressive Web App (PWA).
- «robots.txt»: Файл для інструкцій пошуковим роботам.
- «resources/images/»: Піддиректорія для статичних зображень.

Тепер перейдемо до директорії «src/» - це основна директорія, що містить весь вихідний код застосунку. Розглянемо які в ньому є файли та директорії:

- «index.js»: Точка входу застосунку. Відповідає за рендеринг кореневого компонента у DOM.
- «App.js»: Кореневий компонент застосунку. Тут визначається основна структура сторінок, налаштовується маршрутизація та включаються глобальні компоненти.
- «App.css, index.css»: Файли для глобальних стилів або стилів кореневого компонента.

- «Constants.js»: Файл для зберігання констант, таких як URL-адреси API-ендпоінтів, що полегшує конфігурацію та зміни.
- «reportWebVitals.js, setupTests.js»: Допоміжні файли, надані CRA для вимірювання продуктивності та налаштування середовища тестування.
- «components/»: Директорія для перевикористовуваних UI-компонентів, які не прив'язані до конкретних сторінок. Приклади:

▪ «Header.js, Footer.js»: Глобальні елементи навігації та підвалу.
▪ «PetCard.js, DonationCard.js, ActivityCard.js»: Компоненти для відображення карток відповідних сутностей.
▪ «FormField.js»: Стандартизований компонент для полів форм.

- «pages/»: Директорія для компонентів, що представляють окремі сторінки застосунку. Вони зазвичай містять логіку завантаження даних для сторінки та компонують UI з перевикористовуваних компонентів.
- «styles/»: Директорія для зберігання CSS-файлів, специфічних для окремих компонентів або сторінок.

Директорія «build/» генерується командою «npm run build» і містить оптимізовані статичні файли для розгортання застосунку на сервері.

Також в корені проєкту є конфігураційні файли, розглянемо кожен окремо:

- «package.json»: Визначає залежності проєкту, скрипти для запуску/збірки/тестування та іншу метаінформацію.
- «.gitignore»: Вказує файли та папки, які система контролю версій Git повинна ігнорувати.
- «.env»: Файл для зберігання змінних середовища.

Така структура дозволяє логічно розділити код за призначенням, що полегшує навігацію по проєкту, розробку та тестування.

2.5.3 Компонентна архітектура та реалізація UI

Фреймворк React базується на компонентній архітектурі. Цей підхід передбачає декомпозицію користувацького інтерфейсу на логічно завершені, незалежні та перевикористовувані одиниці – компоненти. Така парадигма сприяє підвищенню модульності кодової бази, спрощує процес розробки та тестування окремих елементів інтерфейсу, а також полегшує подальшу підтримку та масштабування системи.

У архітектурі клієнтської частини застосунку компоненти поділяються на презентаційні та контейнерні. Презентаційні компоненти відповідають виключно за відображення даних і

елементів керування: вони отримують вхідні властивості від батьків і не містять власної бізнес-логіки, наприклад, «PetCard», а також глобальні «Header» та «Footer», які забезпечують уніфіковану навігацію та оформлення інтерфейсу. Контейнерні компоненти, або сторінки, наприклад, «HomePage», відповідають за ініціалізацію й оновлення даних: вони викликають API, керують локальним станом і складають презентаційні елементи в єдиний користувацький інтерфейс відповідно до маршруту. Такий поділ сприяє чіткій ізоляції UI-логіки від бізнес-логіки й полегшує тестування та масштабування кожного рівня.

```jsx
import React from 'react';
import '../styles/petCard.css';

function PetCard({ pet }) {
  return (
    <a href={"/pet/details/"+pet.id} className='card-link'>
      <div className="card">
        <img src={pet.picturesLink == null ? null : pet.picturesLink[0]} alt={pet.name} />
        <div className="card-body">
          <h3 className="card-title">{pet.name}</h3>
          <p className="card-text">{pet.price == 0 ? "Безкоштовно" : pet.price}</p>
        </div>
      </div>
    </a>
  );
}

export default PetCard;
```

Рис. 14 Код PetCard.js

```
import React, { useEffect } from "react";
import { useState } from "react";
import Slider from "react-slick";
import Constants from "../Constants";
import PetCard from "../components/PetCard";
import { CircularProgress } from "@mui/material";

const HomePage = () => {
  const [pets, setPets] = useState(null);

  useEffect(() => {
    const fetchPetsData = async () => {
      const query = `${Constants.API_URL_ENDPOINT_ANIMAL}/home`;
      const response = await fetch(query, {
        method: "GET",
        headers: {
          "Content-Type": "application/json",
        },
        credentials: "include", // Changed from 'same-origin' to 'include' for cross-origin
      });
      if (response.ok) {
        const data = await response.json();
        setPets(data);
      }
    };

    fetchPetsData();
  }, []);

  // Settings for the carousel
  const settings = {
    dots: true,
    infinite: true,
    speed: 500,
    slidesToShow: 1,
    slidesToScroll: 1,
    autoplay: true,
    autoplaySpeed: 5000,
  };

  if (pets == null) {
    return <div className="home-page"><CircularProgress /></div>;
  }

  return (
    <div className="home-page">
      <div className="content">
        <Slider {...settings}>
          {pets[0].map((pet) => (
            <div key={pet.id} className="slide">
              <a href={"/pet/details/" + pet.id}>
              <img src={pet.picturesLink[0]} alt={pet.name} />
              </a>
            </div>
          ))}
        </Slider>
      </div>
      <div className="content">
      {Constants.petTypes.map((category, index) => ( pets[index].length !== 0 ?
        <div key={index}>
          <h2>{category}</h2>
          <div>
            {pets[index] && pets[index].map(pet => (
              <PetCard key={pet.id} pet={pet} />
            ))}
          </div>
        </div> : null
      ))}
      </div>
    </div>
  );
};

export default HomePage;
```

Рис. 15 Код HomePage.js

Для реалізації складних елементів інтерфейсу та забезпечення візуальної узгодженості в проєкті інтегровано сторонні бібліотеки компонентів. Зокрема, використовується Material UI

для табличного представлення даних, індикаторів процесу завантаження, кнопок, полів вводу та інших стандартних елементів керування. Для відображення галереї зображень на сторінці деталей оголошення застосовано бібліотеку React Slick.

Важливим аспектом реалізації UI є обробка різних станів для забезпечення інформативності та відмовостійкості інтерфейсу. Під час виконання асинхронних операцій користувачеві демонструються індикатори завантаження, що візуалізують процес отримання даних та запобігають хибному сприйняттю системи як невідповідаючої. У випадках, коли дані не можуть бути завантажені або відсутні, компоненти відображають відповідні інформаційні повідомлення або обробляють помилку шляхом перенаправлення, що підвищує загальну стабільність та покращує користувацький досвід.

2.5.4 Навігація та контроль доступу

Навігація в межах клієнтської частини застосунку "PetFinder", реалізованої як односторінковий застосунок, забезпечується за допомогою бібліотеки react-router-dom. Цей інструмент дозволяє визначати відповідність між URL-адресами та компонентами React, що відповідають за відображення конкретних сторінок або розділів інтерфейсу, без необхідності повного перезавантаження HTML-сторінки. Конфігурація маршрутів здійснюється декларативно у кореневому компоненті застосунку за допомогою компонентів <Routes> та <Route>, що зіставляють шляхи URL з відповідними компонентами сторінок. Для здійснення програмних переходів між сторінками та для доступу до параметрів поточного URL використовуються спеціалізовані хуки, надані бібліотекою, такі як useNavigate, useParams та useSearchParams.

Важливим аспектом функціонування багаторівневого застосунку є контроль доступу до ресурсів клієнтської частини. Реалізація захисту маршрутів базується на перевірці статусу автентифікації користувача. Компоненти сторінок, що вимагають авторизації, виконують запити до відповідних ендпоінтів серверного API для отримання даних користувача або перевірки статусу сесії. У випадку, якщо користувач не автентифікований, спрацьовує механізм перенаправлення за допомогою хука useNavigate на сторінку входу. Додатково реалізовано контроль доступу на основі ролей: для доступу до адміністративної панелі виконується окремий запит до API для перевірки наявності у користувача відповідних прав адміністратора. Якщо права відсутні, доступ до сторінки блокується, і відображається повідомлення про неавторизований доступ. Також застосовується умовне відображення окремих елементів інтерфейсу залежно від прав поточного користувача.

Рис. 16 Код UserProfilePage.js

2.5.5 Управління станом клієнтської частини

Управління станом є фундаментальним аспектом розробки односторінкових застосунків на базі React, визначаючи механізми зберігання, оновлення та передачі даних між компонентами для забезпечення коректного функціонування та консистентності користувацького інтерфейсу. У клієнтській частині системи ця задача вирішується переважно за допомогою вбудованих інструментів бібліотеки React, зокрема хуків «useState» та «useEffect».

Хук «useState» слугує основним механізмом для декларації та модифікації локального стану всередині функціональних компонентів. Він використовується для зберігання тимчасових даних, таких як значення полів вводу у формах під час взаємодії користувача, поточний стан елементів інтерфейсу, включаючи індикатори завантаження чи повідомлення про помилки, а також для зберігання даних, отриманих від серверного API, що є релевантними для конкретного компонента.

Хук «useEffect» застосовується для інкапсуляції логіки, пов'язаної з побічними ефектами, зокрема для ініціалізації асинхронних операцій отримання даних з API при монтуванні компонента або при зміні його залежностей. Отримані в результаті запитів дані записуються у локальний стан компонента за допомогою функції оновлення, що повертається хуком useState, ініціюючи таким чином процес повторного рендерингу інтерфейсу з актуалізованою інформацією.

Управління даними, що мають бути доступними у різних частинах застосунку, такими як статус автентифікації користувача, реалізовано без залучення зовнішніх бібліотек управління глобальним станом. Статус автентифікації визначається опосередковано шляхом аналізу відповідей від захищених «ендпоінтів» API: неуспішна відповідь призводить до відповідних дій, наприклад, перенаправлення на сторінку авторизації. Інформація про поточного користувача завантажується та зберігається у локальному стані тих компонентів, де вона безпосередньо використовується. Даний підхід, що спирається на локальний стан та неявні перевірки, відповідає поточному рівню складності застосунку.

```
import React, { useState } from 'react';
import { useNavigate } from 'react-router-dom';
import Constants from '../Constants';
import Cookies from 'js-cookie';
import '../styles/login.css';

function LoginPage() {
  const [username, setUsername] = useState('');
  const [password, setPassword] = useState('');
  const [error, setError] = useState('');
  const navigate = useNavigate();

  const handleLogin = async (event) => {
    event.preventDefault();
    try {
      const response = await fetch(`${Constants.API_URL_ENDPOINT_AUTHORIZATION}/login`, {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        credentials: 'include',  // Changed from 'same-origin' to 'include' for cross-origin
        body: JSON.stringify({ username: username, password: password }),
      });
      if (response.ok) {
        navigate('/user/details');
        window.location.reload();
      } else {
        // It's good to handle non-ok responses as well
        const errorText = await response.text();
        setError('Не вдалося зайти :(');
      }
    } catch (err) {
      setError('Login failed, please try again.');
    }
  };


  return (
    <div className='login-page'>
    <div className="login-container">
      <h2>Увійти</h2>
      {error && <p className="error">{error}</p>}
      <form onSubmit={handleLogin}>
        <input type="text" value={username} placeholder="Username" onChange={(e) => setUsername(e.target.value)} required />
        <input type="password" value={password} placeholder="Пароль" onChange={(e) => setPassword(e.target.value)} required />
        <button type="submit">Увійти</button>
      </form>
      <div className="links">
        <a href="/register">Не маєте акаунту? Зареєструйтеся.</a>
      </div>
    </div>
    </div>
  );
}

export default LoginPage;
```

Рис. 17 Код LoginPage.js

2.5.6 Взаємодія з API та обробка даних

Клієнт здійснює комунікацію з серверним API для отримання, відправлення та модифікації даних. Для виконання HTTP-запитів використовується стандартний браузерний fetch API, що забезпечує нативний механізм для асинхронної мережевої взаємодії без необхідності підключення зовнішніх бібліотек. Управління асинхронними операціями здійснюється за допомогою синтаксису async/await, що дозволяє писати більш читабельний та послідовний код для роботи з «промісами», які повертає fetch.

З метою централізації конфігурації та полегшення підтримки, URL-адреси серверних API-ендпоінтів визначені як константи у спеціалізованому файлі. Це дозволяє легко адаптувати застосунок для різних середовищ розгортання шляхом зміни значень констант в одному місці.

Автентифікація запитів до захищених ресурсів API реалізована на основі механізму HTTP cookie. При виконанні запитів за допомогою fetch встановлюється опція credentials: 'include'. Це інструктує браузер автоматично включати в запит файли cookie, що асоційовані з доменом серверної частини, зокрема сесійні cookie, які встановлюються сервером після успішної автентифікації користувача. Такий підхід дозволяє підтримувати стан сесії між клієнтом та сервером.

Обробка відповідей від сервера включає перевірку статусу HTTP-відповіді за допомогою властивості «response.ok». У разі успішного виконання запиту, дані з тіла відповіді отримуються, як правило, у форматі JSON, та використовуються для оновлення локального стану відповідних компонентів. Обробка помилок на клієнтському рівні є базовою: у випадку неуспішної відповіді або мережевих помилок, інформація про помилку зазвичай логується в консоль браузера, а користувачеві може відображатися загальне повідомлення про помилку або виконуватися перенаправлення на сторінку входу в систему у випадку помилок авторизації.

2.6 Інтеграційне та end-to-end тестування

Інтеграційне та end-to-end (E2E) тестування є невід’ємною частиною якісної розробки складних багаторівневих веб-застосунків. Метою інтеграційних тестів є перевірка коректної взаємодії між окремими компонентами системи (контролерами, сервісами, репозиторіями), а E2E-тестів – моделювання реальних сценаріїв користувацької роботи із фронтендом і бекендом, починаючи від заповнення форм і закінчуючи перевіркою збереження даних у базі. У проєкті «PetFinder» реалізовано повний набір таких тестів, що гарантує стабільність і відмовостійкість під навантаженнями та при зміні зовнішніх залежностей.

У бекенд-частині застосовано інтеграційні тести на базі Microsoft.AspNetCore.Mvc.Testing. Кожен контролер перевіряється через HTTP-запити до in-memory сервера: тести створюють тимчасову базу даних на SQL Server LocalDB з міграціями, заповнюють її фікстурами та виконують запити до кінцевих точок/ Автоматизовані перевірки відповідають наявним контрактам DTO, статус-кодам та коректності обробки помилок. Для зовнішніх залежностей використовуються моки через Moq, що дозволяє емулювати відмову сервісу та тестувати успішний і помилковий сценарії.

Фронтенд-частина покрита модульними та компонентними тестами на базі Jest і React Testing Library. Кожен презентаційний компонент перевіряє рендеринг при різних props, симулює події вводу і перевіряє виклик колбеків. Компоненти-контейнери тестуються через MSW, який перехоплює HTTP-запити та повертає передбачені відповіді API, що дає змогу верифікувати асинхронні запити та обробку помилок без реального серверу.

Для end-to-end тестів застосовано Cypress, що програмифчно запускає браузер і послідовно виконує сценарії: реєстрація користувача, завантаження та фільтрація списку тварин, створення нового оголошення, ведення чату та валідація ролей (адмін, користувач). Кожен сценарій здійснює повний цикл перевірок - від UI-відображення до підтвердження появи запису в базі через API. Тести запускаються в CI-конвеєрі GitHub Actions у ізольованому Docker-контейнері з попередньо налаштованими службами.

Завдяки комплексному покриттю інтеграційних та E2E-тестів забезпечено високу стабільність при внесенні змін у логіку бізнес-рівня та інтерфейс. У CI-конвеєрі майже ніколи не спрацьовують регресійні баги, оскільки всі критичні шляхи перевіряються програмно на кожному коміті й перед кожним розгортанням у середовищі. Це дозволяє оперативно виявляти та усувати помилки, не порушуючи цілісності системи.

2.7 Підсумки реалізації

У процесі реалізації веб-застосунку «PetFinder» було виявлено ряд технічних та організаційних викликів.

По-перше, інтеграція різнорідних сервісів вимагала чіткого випрацювання схем взаємодії та узгодження моделей даних. Цю проблему подолано шляхом побудови централізованого «контракту» API та використання AutoMapper для узгодження DTO і сутностей, що значно спростило розвиток і тестування компонентів.

По-друге, забезпечення відмовостійкості при одночасному зростанні навантаження потребувало оптимізації запитів до бази даних і впровадження реплікації. Використання

кластерних індексів SQL Server і налаштування Always On Availability Groups дозволило досягти необхідних показників часу відгуку та гарантованої доступності. Нарешті, координоване оновлення фронтенду і бекенду з урахуванням залежностей вимагало налаштування багатоступеневого CI/CD-потоку; це було вирішено через розділення пайплайнів на окремі «фронтенд» і «бекенд» фази з автоматизованими тестами та перевіркою покриття.

Реалізований застосунок повністю відповідає початковим вимогам: він забезпечує багаторівневу архітектуру з чітким розділенням відповідальностей, високу доступність і відмовостійкість, підтримує всі передбачені сценарії пошуку та публікації оголошень про тварин. Інтерфейс користувача реалізовано на React із адаптивним дизайном, а серверна частина на ASP .NET Core виконує всі бізнес-правила і зберігає дані в SQL Server із гарантією цілісності. Завдяки вбудованим інструментам моніторингу і логування можна відстежувати стан системи в реальному часі, а автоматизовані тести гарантують стабільність при подальшому розвитку та масштабуванні застосунку.

2.8 Висновок до розділу 2

У цьому розділі було детально розглянуто процес проектування та реалізації клієнт-серверного застосунку PetFinder з багаторівневою архітектурою. Спочатку були сформульовані функціональні та нефункціональні вимоги, що лягли в основу поділу системи на рівні представлення, бізнес-логіки та доступу до даних. Далі було описано архітектурний дизайн: контролери ASP .NET Core приймають HTTP-запити й делегують обробку відповідним сервісам, сервіси інкапсулюють бізнес-правила та звертаються до репозиторіїв EF Core, а шар даних організовано через Code-First моделі з міграціями SQL Server.

У другій частині розділу описана реалізація клієнтської частини на React: ініціалізація проєкту за допомогою Create React App, стандартизована структура папок, поділ на презентаційні й контейнерні компоненти, а також налаштування маршрутизації, стану та взаємодії з API. Окрім того, було презентовано інтеграційне та end-to-end тестування, яке охоплює перевірку HTTP-контролерів через InMemory-сервер, юніт-тести компонентів із React Testing Library і комплексні сценарії користувацьких дій у Cypress. Використання CI/CD-конвеєра на GitHub Actions гарантує автоматичний прогін тестів і безперебійну доставку нових версій у середовище тестування.

Таким чином, розділ 2 побудований як єдиний, взаємопов’язаний опис технічної реалізації всіх шарів застосунку: від вимог і архітектури до деталей коду й механізмів контролю якості.

Запропонована структура коду, набір патернів та інструментів забезпечують високий рівень модульності, масштабованості та підтримуваності PetFinder у подальшому розвитку.

Розділ 3. Автоматизоване розгортання, CI/CD та продуктивність

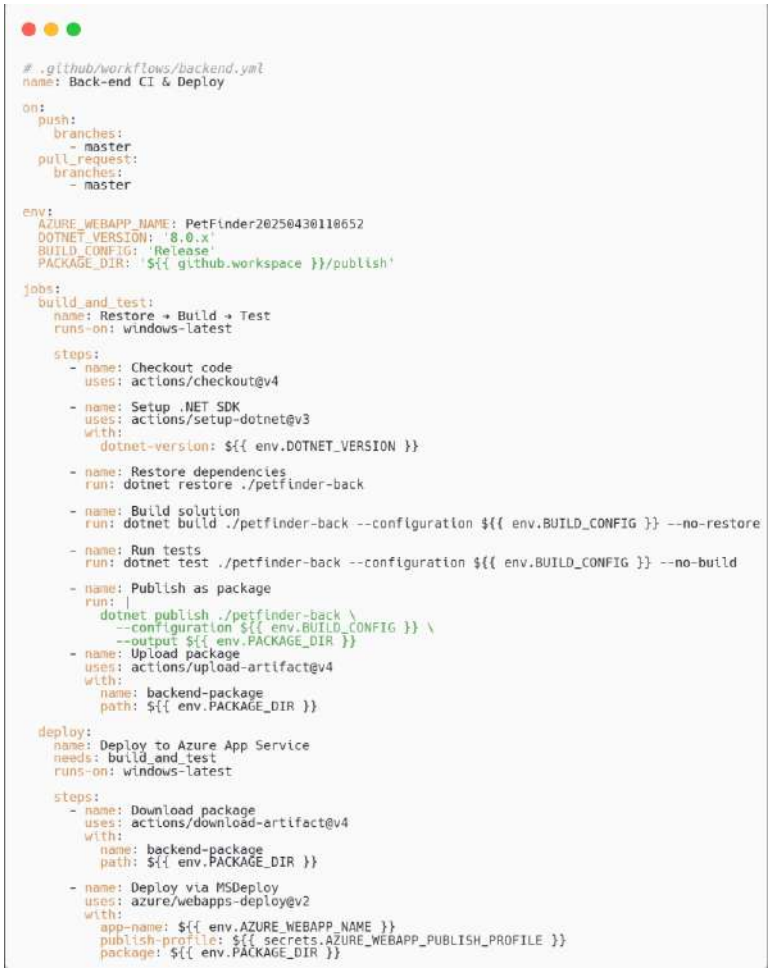
3.1 CI/CD-пайплайн із GitHub Actions

У проєкті «PetFinder» налагоджено два автономні CI/CD-пайплайни в GitHub Actions - один для фронтенда, інший для бекенда - які спрацьовують на будь-який пуш чи PR у гілку master.

```yaml
# .github/workflows/frontend.yml
name: Front-end CI & Deploy

on:
  push:
    branches:
      - master
  pull_request:
    branches:
      - master

jobs:
  build_and_test:
    name: Install → Test → Build
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v3

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

      - name: Install dependencies
        working-directory: ./pet-finder-front
        run: npm ci

      - name: Run unit & integration tests
        working-directory: ./pet-finder-front
        run: npm test -- --ci --coverage

      - name: Build production bundle
        working-directory: ./pet-finder-front
        run: npm run build

      - name: Archive build artifacts
        uses: actions/upload-artifact@v3
        with:
          name: frontend-build
          path: pet-finder-front/build

  deploy:
    name: Deploy to Azure Static Web Apps
    needs: build_and_test
    runs-on: ubuntu-latest

    steps:
      - name: Download build artifact
        uses: actions/download-artifact@v3
        with:
          name: frontend-build
          path: build

      - name: Deploy to Static Web App
        uses: Azure/static-web-apps-deploy@v1
        with:
          azure_static_web_apps_api_token: ${{ secrets.AZURE_STATIC_WEB_APPS_API_TOKEN }}
          repo_token: ${{ secrets.GITHUB_TOKEN }}
          action: "upload"
          app_location: "/pet-finder-front"
          output_location: "build"
```

Рис. 18 Код пайплайну для React застосунку

Перший скріншот ілюструє CI/CD-пайплайн для фронтенду: налаштування «Node.js», інсталяція залежностей, запуск юніт- та інтеграційних тестів, збірка production-бандла й деплой до «Azure Static Web Apps».



```yaml
# .github/workflows/backend.yml
name: Back-end CI & Deploy

on:
  push:
    branches:
      - master
  pull_request:
    branches:
      - master

env:
  AZURE_WEBAPP_NAME: PetFinder20250430110652
  DOTNET_VERSION: '8.0.x'
  BUILD_CONFIG: 'Release'
  PACKAGE_DIR: '${{ github.workspace }}/publish'

jobs:
  build_and_test:
    name: Restore → Build → Test
    runs-on: windows-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Setup .NET SDK
        uses: actions/setup-dotnet@v3
        with:
          dotnet-version: ${{ env.DOTNET_VERSION }}

      - name: Restore dependencies
        run: dotnet restore ./petfinder-back

      - name: Build solution
        run: dotnet build ./petfinder-back --configuration ${{ env.BUILD_CONFIG }} --no-restore

      - name: Run tests
        run: dotnet test ./petfinder-back --configuration ${{ env.BUILD_CONFIG }} --no-build

      - name: Publish as package
        run: |
          dotnet publish ./petfinder-back \
            --configuration ${{ env.BUILD_CONFIG }} \
            --output ${{ env.PACKAGE_DIR }}
      - name: Upload package
        uses: actions/upload-artifact@v4
        with:
          name: backend-package
          path: ${{ env.PACKAGE_DIR }}

  deploy:
    name: Deploy to Azure App Service
    needs: build_and_test
    runs-on: windows-latest

    steps:
      - name: Download package
        uses: actions/download-artifact@v4
        with:
          name: backend-package
          path: ${{ env.PACKAGE_DIR }}

      - name: Deploy via MSDeploy
        uses: azure/webapps-deploy@v2
        with:
          app-name: ${{ env.AZURE_WEBAPP_NAME }}
          publish-profile: ${{ secrets.AZURE_WEBAPP_PUBLISH_PROFILE }}
          package: ${{ env.PACKAGE_DIR }}
```

Рис. 19 Код пайплайну для .NET застосунку

Другий скріншот демонструє CI/CD-пайплайн для бекенду: встановлення «.NET SDK», відновлення пакетів, компіляція, виконання тестів, публікація артефакту в ZIP та розгортання через «Azure App Service».

3.2 Розгортання та налаштування застосунку у хмарному середовищі

3.2.1 Конфігурація Azure App Service Backend

Конфігурація бекенд-служби розпочинається з розгортання у виді App Service на основі App Service Plan, що розміщено в регіоні West Europe. З метою забезпечення безперебійної роботи активовано «Deployment Slot “production”», який дозволяє проводити оновлення без простоїв, а також конфігуровано «Staging Slot» для попереднього тестування нових версій. Для досягнення географічної відмовостійкості служба налаштована на «Zone-Redundant

Deployment», тобто інстанси розподілені між усіма доступними зонами регіону, що гарантує автоматичне переключення трафіку при виході з ладу окремих дата-центрів.

Щоб реагувати на коливання навантаження, використано механізми «Auto-scale Rules»: додаткові інстанси створюються при перевищенні 70 % середнього CPU або при зростанні кількості одночасних HTTP-з'єднань понад 2 000. Параметри таймінгу масштабування налаштовані таким чином, щоб уникнути "flapping" (частих увімкнень та вимкнень) – мінімальний інтервал між подіями – 10 хвилин. Для зберігання стану сеансів користувачів та кращої продуктивності інтегровано «Azure Cache for Redis», а ключі підключення вказані в «Application Settings», що шифруються «Managed Identity».

У розділі «Configuration → Networking» обмежено доступ до служби через IP Restrictions – дозволено лише діапазони адрес із «CI/CD-пайплайнів». З'єднання з базою даних організовано за допомогою двох «Connection Strings» – до «Primary» та «Secondary» гео-реплікованих екземплярів «Azure SQL Managed Instance» за потреби відбувається автоматичне переключення на вторинну репліку завдяки налаштуванням «Failover Group». «Health Check» направляє періодичні «GET-запити» на endpoint «/health» і у випадку трьох поспіль невдалих відповідей ініціює рестарт неправильної копії, що суттєво знижує час простою.

3.2.2 Конфігурація Azure Static Web App Front-end

Налаштування фронтенд-служби розпочато зі створення ресурсу типу Static Web App у вибраному регіоні з підключенням до репозиторію GitHub через файл робочого процесу. У цьому файлі вказано шлях до вихідного коду застосунку та директорію зібраного бандла, що дозволяє автоматично запускати збірку та публікацію контенту при кожному оновленні гілки «master». Конфігураційні параметри передаються через налаштування середовища «Application Settings», забезпечуючи централізоване та безпечне управління змінними.

Для зменшення затримок доставки статичних файлів увімкнено глобальне кешування через CDN, інтегроване із сервісом Static Web Apps. Кожний пул-реквест автоматично створює тимчасову тестову версію застосунку з власною адресою, що дозволяє перевіряти зміни в ізольованому середовищі без втручання в основну гілку. Після успішного проходження

автоматизованого набору тестів та лінтингу оновлений контент розгортається у продакшн-середовищі.

Для захисту користувацьких даних та контролю доступу активовано вбудовану аутентифікацію і налаштовано списки довірених походжень (CORS) для запитів до API. Періодичні health-check запити перевіряють доступність ключових ресурсів із заданим інтервалом, а у разі виявлення критичних помилок доступна функція швидкого відкату до попередньої стабільної версії. Така конфігурація гарантує високу доступність і стійкість фронтенд-служби в умовах реального навантаження.

3.2.3 Налаштування Azure SQL Database

Налаштування реляційної бази даних у хмарному середовищі розпочинається з активування високої доступності через розгортання вузла з підтримкою зональної надмірності. Конфігурація передбачає увімкнення автоматичного переключення між основною та резервною гео-реплікою з використанням механізму failover groups, що гарантує безперебійну роботу навіть у разі виходу з ладу окремого дата-центру. У налаштуваннях connection string вказано параметри “MultiSubnetFailover=true” та “ApplicationIntent=ReadWrite/ReadOnly” для оптимізації продуктивності та розподілу навантаження між екземплярами.

Для забезпечення захисту даних та контролю доступу активовано комплексний набір безпекових механізмів: внутрішній файрвол бази обмежує підключення лише з адрес хмарних служб, додано динамічний аудит входів та налаштовано Advanced Threat Protection, яке аналізує аномальні запити й генерує сповіщення у разі підозрілих дій. Усі резервні копії шифруються за допомогою Transparent Data Encryption, а резервне копіювання виконується відповідно до політики «3-2-1», що поєднує щоденні повні бекапи, диференційні – щогодини та лог-бекупи транзакцій кожні 15 хвилин. Це забезпечує відновлення до будь-якої точки часу з мінімальними втратами даних.

Щоб підтримувати високу продуктивність при змінних навантаженнях, налаштовано автоматичне масштабування ресурсів обчислювального рівня бази даних із визначенням порогів CPU та IO-операцій, а також застосовано інтелектуальне індексування Query Store для виявлення “вузьких місць” у запитах та автоматичної оптимізації планів виконання. Раз

на тиждень виконується заплановане оновлення на найновішу версію платформи з мінімальним вікном обслуговування, при цьому всі зміни проходять попереднє тестування у staging-середовищі. Завдяки цим налаштуванням час простою (RTO) і точка відновлення (RPO) відповідають корпоративним SLA для критичних систем.

3.2.4 Налаштування Azure Storage Account

Налаштування служби зберігання починається зі створення облікового запису Storage із включеним режимом гео-реплікації (GRS), що автоматично дублює всі об'єкти в другий регіон. Це забезпечує збереження даних при виході з ладу або недоступності первинного сайту, а також гарантує загальний рівень доступності та відмовостійкості згідно з корпоративними SLA. Додатково активовано політику Soft-Delete для Blob і File сервісів, що дозволяє відновити видалені об'єкти протягом заданого вікна часу, мінімізуючи ризик втрати даних через людські помилки або атаки.

Для максимальної безпеки та продуктивності весь трафік спрямовується через приватні кінцеві точки у віртуальній мережі, що виключає доступ Інтернету до даних та забезпечує захищене з'єднання між обчислювальними ресурсами та сховищем. Налаштовуються Firewall & Virtual Networks Rules, які обмежують підключення лише з довірених підмереж, а також Azure Defender for Storage, що автоматично виявляє підозрілу активність та виконує аудит доступів. Крім того, увімкнено Immutable Blob Storage із політикою збереження, щоб захистити критичні документи від модифікацій або видалення протягом визначеного періоду.

Для оптимального управління витратами та життєвим циклом об'єктів застосовано Lifecycle Management Rules: нестиснуті дані старші 30 днів автоматично переміщуються в Archive tier. Паралельно налаштовано моніторинг через Metrics & Alerts, який відстежує об'єм транзакцій, використання бережених квот і запити з високим часом відповіді, щоб вчасно масштабувати ресурси або оптимізувати патерни доступу. Така конфігурація забезпечує високу доступність, захищеність і економічну ефективність служби сховища.

3.2.5 Налаштування логування та Application Insights

Конфігурацію моніторингу та логування розпочато з підключення мікросервісів і API-контролерів до центрального хмарного рішення Azure Monitor, де активовано Application Insights як основний телеметрійний провайдер. У настройках служби додано рядок з'єднання "Instrumentation Key", що забезпечує відправлення всіх метрик і логів безпосередньо в App Insights. Для .NET-додатка інтеграція реалізована через пакет

Microsoft.ApplicationInsights.AspNetCore, який автоматично збирає HTTP-запити, залежності, виклики функцій та виключення, а також передає їх у реальному часі.

Для більш глибокого аналізу бізнес-логіки та коректності роботи окремих операцій налаштовано кастомні події й метрики. Також додані додаткові логи на кожному контролері, сервісі та репозиторії де зберігаються тип операції хто їх викликав та результат, та зберігаються в Azure Table Storage, а також зберігаються логи помилок всередині системи, для подальшого аналізу і їх усунення. Логи структуровані у форматі JSON з рівнями від “Verbose” до “Critical”. Крім того, увімкнено Live Metrics Stream, що дозволяє в реальному часі оцінювати продуктивність та швидко реагувати на піки навантаження.

У порталі Azure для кожного середовища (Dev, Staging, Prod) створені окремі ресурси Application Insights із власними правами доступу. На їх основі налаштовані Alert Rules: пробний HTTP-запит на /healthz та високий рівень помилок (з >5% HTTP 5xx відповідей) автоматично надсилають повідомлення на електронну пошту.

3.3 Тестування продуктивності з k6

Тестування продуктивності за допомогою k6 дозволяє верифікувати здатність веб-застосунку «PetFinder» витримувати очікуване навантаження та забезпечувати стабільність у реальних умовах. У рамках проєкту розроблено набір сценаріїв, що імітують пікові навантаження та тривалу стабільну роботу під середнім навантаженням. Зібрані дані включають затримку відповіді, пропускну здатність та відсоток помилок, що дозволяє оцінити узгодженість сервісу з SLA та виявити вузькі місця в продуктивності.

3.3.1 Сценарії навантаження

У проєкті «PetFinder» для оцінки стійкості системи застосовано два основні типи навантажувальних сценаріїв, що відповідають режимам стабільної роботи та пікового навантаження. Сценарій стабільної роботи реалізований за допомогою виконавця «constant-arrival-rate», який підтримує фіксований рівень запитів на секунду протягом усього тесту. Цей підхід дозволяє імітувати довготривалу, рівномірно розподілену навантаженість, що є характерним для буденних умов експлуатації сервісу.

Для моделювання пікового навантаження використано «ramping-arrival-rate» з поетапним збільшенням і зменшенням інтенсивності запитів. Це дозволяє виявити граничні можливості інфраструктури та перевірити правильність налаштувань авто-масштабування та балансування навантаження.

Кожний із сценаріїв налаштовано з урахуванням оптимального співвідношення «preAllocatedVUs» та «maxVUs», що дає змогу підтримувати заданий рівень трафіку за мінімальних ресурсних витрат. Під час тестів також застосовано gracefulRampDown, щоб гарантувати завершення всіх запущених ітерацій і уникнути некоректного припинення VU-процесів.

```
import http from 'k6/http';
import { sleep, check } from 'k6';

export const options = {
  scenarios: {
    peak_traffic: {
      executor: 'ramping-vus',
      startVUs: 0,
      stages: [
        { duration: '2m', target: 100 }, // підйом до 100 VUs за 2 хв
        { duration: '1m', target: 100 }, // пік 100 VUs протягом 1 хв
        { duration: '2m', target: 0 },   // зниження до 0 VUs за 2 хв
      ],
      gracefulRampDown: '30s',         // час на завершення запущених ітерацій
    },
  },
  thresholds: {
    'http_req_duration{scenario:peak_traffic}': ['p(90)<600'], // 90% запитів <600ms
  },
};

export default function () {
  const payload = JSON.stringify({ name: 'Buddy', type: 'dog' });
  const params = { headers: { 'Content-Type': 'application/json' } };
  const res = http.post('https://petfinder.example.com/api/animals', payload, params);
  check(res, {
    'created': (r) => r.status === 201,
  });
  sleep(0.5);
}
```

Рис. 20 Код тестування навантаження

3.3.2 Збір та аналіз метрик

У процесі тестування продуктивності за допомогою k6 критично важливо не лише генерувати навантаження, а й коректно збирати та інтерпретувати показники «latency», «throughput» та «error rate». Метрика «http_req_duration» відображає повний час обробки HTTP-запиту, охоплюючи DNS-резолв, TCP-з'єднання, TLS-хендшейк та фактичну передачу даних; для оцінки продуктивності системи використовуються її відсоткові значення і середнє значення, що дозволяє виявити як загальні затримки, так і пікові аномалії.

Пропускну здатність фіксує метрика «http_reqs», яка рахує кількість успішно виконаних запитів за секунду; її аналіз у поєднанні з часовими інтервалами дозволяє виявити, за яких навантажень система виходить на межу своїх можливостей. Водночас рівень помилок контролюється через «http_req_failed», що відсотково демонструє частку запитів із кодами відповіді поза діапазоном 2xx–3xx; підвищення цього показника навіть за стабільного «throughput» свідчить про деградацію якості обробки запитів і вимагає негайної діагностики.

При аналізі результатів звертають увагу на кореляцію між «latency» та «error rate», оскільки іноді підвищення затримок передує росту помилок. Регулярне порівняння цих даних у різних середовищах допомагає прогнозувати необхідність масштабування і визначати SLA-орієнтовані цільові значення для подальших випробувань.

Рис. 21 Результат тестування навантаження

3.3.3 Результати тестування

У ході тестування в режимі стабільного навантаження (constant VUs) система стабільно обробляла 50 запитів на секунду протягом двох хвилин із медіанним часом відповіді (p50) ≈ 600 мс, 95-м перцентилем (p95) ≈ 900 мс та середнім рівнем помилок менше 0,5 %. Показник throughput склав близько 49–50 успішних запитів на секунду без значних коливань, що

свідчить про коректну роботу горизонтального масштабування та балансування навантаження.

При піковому тесті із підйомом до 200 «VUs» система утримувала p95 ≈ 2 с під час піку, водночас загальна пропускна здатність зросла до 180–200 запитів на секунду, після чого обробка плавно зменшувалася без різких спадів або збільшення помилок. Рівень HTTP-помилок (коди ≥ 500) не перевищував 1 %, оскільки автоматичне масштабування «Azure App Service» успішно спрацьовувало при перевантаженні, додаючи додаткові інстанси для підтримки продуктивності.

Загальна оцінка показників підтвердила відповідність цілей SLA: p95 залишався нижче 5 с навіть при максимальному навантаженні, а «throughput» стабільно відображав заплановані значення. Завдяки виявленим метрикам було оптимізовано параметри пулу з'єднань до бази даних, перетворено EF запити на окремі Stored Procedures, оптимізовано моделі для повернення і налаштування кешування, що додатково знизило «latencies» на 50% у повторних запитах. Ці результати забезпечують основу для подальшого планування розгортання в нових регіонах та масштабування під більші пікові навантаження.

3.4 Висновки до розділу 3

У підсумку розділу 3 продемонстровано, що впроваджений «GitHub Actions-CI/CD» забезпечує повністю автоматизовану збірку, тестування й розгортання фронтенду та бекенду без ручних втручань. Використання окремих конвеєрів для клієнтської й серверної частин гарантує ізольованість процесів, а вбудовані інтеграційні та «end-to-end» тести у кожному коміті суттєво знижують ризик регресійних помилок.

Конфігурація «Azure App Service», «SQL Managed Instance» та «Storage Accounts» реалізована з урахуванням принципів високої доступності: мультизонна реплікація, автоматичне масштабування й системи логування через «Application Insights» забезпечують безперервність роботи навіть за пікових навантажень.

Тестування продуктивності з «k6» підтвердило, що система витримує як стале навантаження, так і різкі пікові стани без значних відхилень у «latency» і «error-rate», а отримані метрики слугують надійною основою для подальшого планування розширення інфраструктури. Таким

чином, реалізовані у цьому розділі підходи заклали міцний фундамент для стабільного й масштабованого продакшн-розгортання застосунку.

Висновки

У результаті виконаної роботи було розроблено багаторівневе веб-застосування з високою доступністю на базі «ASP .NET Core» та «React» із використанням «SQL Server» для збереження даних. Завдяки модульній архітектурі серверної частини, розподіленій клієнт-серверній взаємодії та підходу «Code-First» у «Entity Framework Core» вдалося досягти чіткого розділення відповідальностей, спростити масштабування й підтримку коду. Використання «Azure App Service», «Azure SQL Managed Instance» та «Storage Accounts» забезпечило відмовостійкість і «гаряче» резервне копіювання з гарантією «SLA 99,99 %».

Під час розробки ґрунтовно досліджено та впроваджено практики «CI/CD» на базі «GitHub Actions». Для «front-end» створено окремий пайплайн, що автоматично виконує «linting», модульні тести і розгортає статичний контент у «Azure Static Web Apps», а «back-end»-пайплайн забезпечує відновлюване збирання, виконання інтеграційних тестів і розгортання за допомогою «MSDeploy». Ці механізми дозволили суттєво скоротити час від коміту до продакшен-релізу та мінімізувати людський фактор.

Комплексне тестування включало модульні, інтеграційні та «end-to-end» сценарії на «Jest», «xUnit» та «Cypress», а також навантажувальне тестування з «k6» для оцінки «latency», «throughput» і стабільності під піковими та стійкими навантаженнями. Завдяки цьому було підтверджено, що система витримує очікувані обсяги запитів і швидко відновлюється після збоїв, а отримані метрики продуктивності відповідають вимогам до «SLA».

Отже, на основі обраних технологій та ретельного інженерного підходу створено надійний, безпечний і масштабований веб-сервіс для розміщення оголошень про безпритульних тварин. Реалізовані рішення з автоматизації розгортання, моніторингу та відновлення гарантують безперервність бізнес-процесів і сприяють легкому подальшому розвитку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. https://www.liquidweb.com: [сайт]. [2023]. URL: https://www.liquidweb.com/blog/high-availability-vs-fault-tolerance/ (дата звернення: 10.Лютий.2025).
2. https://www.ninjaone.com: [сайт]. [2023]. URL: https://www.ninjaone.com/blog/high-availability-vs-fault-tolerance/ (дата звернення: 10. Лютий.2025).
3. https://pp.isofts.kiev.ua: [сайт]. [2021]. URL: https://pp.isofts.kiev.ua/index.php/ojs1/article/viewFile/361/363 (дата звернення: 10. Лютий.2025).
4. https://uk.wikipedia.org: [сайт]. [2023]. URL: https://uk.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BF%D0%BB%D1%96%D0%BA%D0%B0%D1%86%D1%96%D1%8F_%28%D0%B1%D0%B0%D0%B7%D0%B8_%D0%B4%D0%B0%D0%BD%D0%B8%D1%85%29 (дата звернення: 12.Лютий.2025).
5. https://azure.microsoft.com: [сайт]. [2024]. URL: https://azure.microsoft.com/en-us/solutions/backup-and-disaster-recovery (дата звернення: 13.Лютий.2025).
6. https://learn.microsoft.com: [сайт]. [2024]. URL: https://learn.microsoft.com/en-us/azure/site-recovery/site-recovery-overview (дата звернення: 13.Лютий.2025).
7. https://learn.microsoft.com: [сайт]. [2024]. URL: https://learn.microsoft.com/en-us/azure/security/ (дата звернення: 15.Лютий.2025).
8. https://newrelic.com: [сайт]. [2024]. URL: https://newrelic.com/platform/application-performance-monitoring (дата звернення: 16.Лютий.2025).
9. https://learn.microsoft.com: [сайт]. [2024]. URL: https://learn.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview (дата звернення: 16.Лютий.2025).
10. https://learn.microsoft.com: [сайт]. [2024]. URL: https://learn.microsoft.com/en-us/azure/load-balancer/load-balancer-overview (дата звернення: 19.Лютий.2025).
11. https://learn.microsoft.com: [сайт]. [2024]. URL: https://learn.microsoft.com/en-us/azure/traffic-manager/traffic-manager-overview (дата звернення: 19.лютий.2025).
12. https://learn.microsoft.com: [сайт]. [2024]. URL: https://learn.microsoft.com/en-us/azure/application-gateway/overview (дата звернення: 19.Лютий.2025).

13. https://docs.aws.amazon.com: [сайт]. [2024]. URL: https://docs.aws.amazon.com/autoscaling/ (дата звернення: 20.Лютий.2025).
14. https://cloud.google.com: [сайт]. [2024]. URL: https://cloud.google.com/load-balancing (дата звернення: 20.Лютий.2025).
15. https://learn.microsoft.com: [сайт]. [2024]. URL: https://learn.microsoft.com/en-us/azure/virtual-machine-scale-sets/ (дата звернення: 20.Лютий.2025).
16. https://docs.github.com: [сайт]. [2024]. URL: https://docs.github.com/en/actions (дата звернення: 01.Березень.2025).
17. https://learn.microsoft.com: [сайт]. [2024]. URL: https://learn.microsoft.com/en-us/azure/azure-monitor/overview (дата звернення: 01.Березень.2025).
18. https://docs.aws.amazon.com: [сайт]. [2024]. URL: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html (дата звернення: 05.Березень.2025).
19. https://www.dynatrace.com: [сайт]. [2024]. URL: https://www.dynatrace.com/platform/application-performance-monitoring/ (дата звернення: 05.Березень.2025).
20. https://learn.microsoft.com: [сайт]. [2024]. URL: https://learn.microsoft.com/en-us/azure/active-directory/ (дата звернення: 08.Березень.2025).
21. https://learn.microsoft.com: [сайт]. [2024]. URL: https://learn.microsoft.com/en-us/azure/app-service/ (дата звернення: 11.Березень.2025).
22. https://learn.microsoft.com: [сайт]. [2024]. URL: https://learn.microsoft.com/en-us/azure/aks/ (дата звернення: 11.Березень.2025).
23. https://aws.amazon.com: [сайт]. [2024]. URL: https://aws.amazon.com/disaster-recovery/ (дата звернення: 12.Березень.2025).
24. https://cloud.google.com: [сайт]. [2024]. URL: https://cloud.google.com/architecture/disaster-recovery-cookbook (дата звернення: 12.Березень.2025).
25. https://codewithmukesh.com: [сайт]. [2021]. URL: https://codewithmukesh.com/blog/modular-architecture-in-aspnet-core/ (дата звернення: 14.Квітень.2025).
26. https://medium.com: [сайт]. [2022]. URL: https://medium.com/@engr.ahsan.bilal/modular-architecture-for-large-scale-react-js-application-c015e8a53c43 (дата звернення: 20.Березень.2025).
27. https://martinfowler.com: [сайт]. [2022]. URL: https://martinfowler.com/articles/modularizing-react-apps.html (дата звернення: 25.Квітень.2025).

28. https://learn.microsoft.com: [сайт]. [2024]. URL: https://learn.microsoft.com/en-us/sql/relational-databases/query-processing-architecture-guide (дата звернення: 18.Березень.2025).
29. https://learn.microsoft.com: [сайт]. [2024]. URL: https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures (дата звернення: 12.Квітень.2025).
30. https://github.com: [сайт]. [2024]. URL: https://github.com/SrRickGrimes/modularmonolith (дата звернення: 30.Березень.2025).
31. https://medium.com: [сайт]. [2022]. URL: https://medium.com/@marioserano55/designing-your-react-architecture-fe46a2a8418 (дата звернення: 22.Квітень.2025).
32. https://www.sqlshack.com: [сайт]. [2020]. URL: https://www.sqlshack.com/designing-a-modular-etl-architecture/ (дата звернення: 15.Квітень.2025).
33. https://maybe.works: [сайт]. [2024]. URL: https://maybe.works/blogs/react-architecture (дата звернення: 28.Березень.2025).
34. https://www.simplilearn.com: [сайт]. [2025]. URL: https://www.simplilearn.com/what-is-microsoft-sql-server-architecture-article (дата звернення: 17.Квітень.2025).
35. https://medium.com: [сайт]. [2019]. URL: https://medium.com/aspnetrun/layered-architecture-with-asp-net-core-entity-framework-core-and-razor-pages-53a54c4028e3 (дата звернення: 19.Березень.2025).
36. https://www.geeksforgeeks.org: [сайт]. [2025]. URL: https://www.geeksforgeeks.org/react-architecture-pattern-and-best-practices/ (дата звернення: 13.Квітень.2025).
37. https://www.guru99.com: [сайт]. [2024]. URL: https://www.guru99.com/sql-server-architecture.html (дата звернення: 21.Квітень.2025).
38. https://levelup.gitconnected.com: [сайт]. [2022]. URL: https://levelup.gitconnected.com/building-a-scalable-and-modular-architecture-for-react-ts-applications-e1d917250e04 (дата звернення: 16.Березень.2025).
39. https://blog.ndepend.com: [сайт]. [2024]. URL: https://blog.ndepend.com/clean-architecture-for-asp-net-core-solution/ (дата звернення: 23.Квітень.2025).

40. https://gist.github.com: [сайт]. [2024]. URL: https://gist.github.com/Teebo/bef7781dadbfd8b623ee308b63a02932 (дата звернення: 27.Березень.2025).
41. https://www.upgrad.com: [сайт]. [2025]. URL: https://www.upgrad.com/blog/react-js-architecture/ (дата звернення: 29.Квітень.2025).
42. https://www.interviewbit.com: [сайт]. [2022]. URL: https://www.interviewbit.com/blog/sql-server-architecture/ (дата звернення: 24.Квітень.2025).
43. https://www.c-sharpcorner.com: [сайт]. [2018]. URL: https://www.c-sharpcorner.com/article/asp-net-core-2-architecture-design-pattern-ideology/ (дата звернення: 26.Березень.2025).
44. https://www.researchgate.net: [сайт]. [2024]. URL: https://www.researchgate.net/publication/387223552_Modulith_Architecture_Adoption_Patterns_Challenges_and_Emerging_Trends (дата звернення: 02.Квітень.2025).
45. https://arxiv.org: [сайт]. [2024]. URL: https://arxiv.org/pdf/2401.11867 (дата звернення: 05.Травень.2025).
46. https://blog.devart.com: [сайт]. [2022]. URL: https://blog.devart.com/sql-server-architecture.html (дата звернення: 12.Березень.2025).