НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Інститут прикладного системного аналізу

Кафедра математичних методів системного аналізу

До захисту допущено:

Завідувач кафедри

_____ Дмитро Вдовиченко

«___»_____20__ р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Системи та методи штучного інтелекту»

спеціальності 122 «Системи та методи штучного інтелекту»

на тему: «Розробка програмного забезпечення для аналізу ефективності серверів на платформі .NET Core»

Виконав:
студент IV курсу, групи КА-76
Вдовиченко Дмитро Юрійович _____

Керівник:
Асистент к.т.н
Гуськова Віра Генадіївна _____

Консультант з економічного розділу:
Доцент кафедри теоретичної і прикладної економіки ФММ
Рощина Надія Василівна _____

Консультант з нормоконтролю:
Доцент к.т.н.
Коваленко Анатолій Єпіфанович _____

Рецензент:
Посада, науковий ступінь, вчене звання,
Прізвище, ім’я, по батькові _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2021 року

Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Інститут прикладного системного аналізу

Кафедра математичних методів системного аналізу

Рівень вищої освіти – перший (бакалаврський)

Спеціальність – 122 "Комп’ютерні науки"

Освітньо-професійна програма «Системи та методи штучного інтелекту»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Оксана ТИМОЩУК
«___»_____20__ р.

ЗАВДАННЯ

на дипломну роботу студенту

Вдовиченко Дмитру Юрійовичу

1. Тема роботи «Розробка програмного забезпечення для аналізу ефективності серверів на платформі .NET Core», керівник роботи Густкова Віра Генадіївна, Асистент к.т.н, затверджені наказом по університету від «___»_____ 20__ р. №_____

2. Термін подання студентом роботи _____

3. Вихідні дані до роботи

Існуюче програмне забезпечення для створення вебсерверів на платформі .NET Core.

4. Зміст роботи

Дослідження предметної області, розробка програмного забезвечення на платформі .net core, аналіз розробленого програмного продукту, функціонально-вартісний аналіз програмного продукту.

5. Перелік ілюстративного матеріалу (із зазначенням плакатів, презентацій тощо)

Актуальність задачі, платформа .NET Core, середа розгортання додатку, асинхроні запити.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Економічний	Рощина Н.В. доцент к.е.н.		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітка
1	Вивчення літератури за темою роботи	13.04.2021	
2	Підготовка першого розділу	21.04.2021	
3	Підготовка другого розділу	1.05.2021	
4	Розробка програмного продукту	19.05.2021	
5	Підготовка третього розділу	25.05.2021	
6	Підготовка економічної частини	28.05.2021	
7	Оформлення розділів відповідно до нормоконтролю		
8	Підготовка презентації доповіді	30.05.2021	
9	Попередній захист дипломної роботи	01.06.2021	
10	Захист дипломної роботи		

Студент Дмитро ВДОВИЧЕНКО

Керівник Віра ГУСЬКОВА

РЕФЕРАТ

Дипломна робота містить: 91 с., 26 рис., 6 табл., 2 додатки, 9 джерела.

ПОРІВНЯЛЬНИЙ АНАЛІЗ, .NET CORE, ASP.NET CORE.

У роботі розглянуто та проаналізовано найбільш поширені методи для створення вебсерверів на платформі .NET Core. Досліджені методи для покращення ефективності вебсерверів на платформі .NET Core та був проведений порівняльний аналіз їх ефективності.

Мета проаналізувати та провести аналіз методів створення та оптимізації вебсерверів на платформі .NET Core.

Об’єктом дослідження роботи стали вебсервера, проблеми при їх створені та методи оцінки їх ефективності.

Предметом дослідження є інструменти для створення вебсерверів на платформі .NET Core та методи їх оптимізації.

ABSTRACT

Thesis contains: 31 pp., 26 fig., 6 table., 2 appendices, 9 sources.

COMPARATIVE ANALYSIS, .NET CORE, ASP.NET CORE.

In this work considers and analyzes the most common methods for creating web servers on the .NET Core platform. Methods for improving the efficiency of web servers on the .NET Core platform were investigated and a comparative analysis of their efficiency was performed.

The purpose is to analyze and analyze methods of creating and optimizing web servers on the .NET Core platform.

The object of the study were web servers, problems with their creation and methods for evaluating their effectiveness.

The subject of the research are tools for creating web servers on the .NET Core platform and methods of their optimization.

ЗМІСТ

ВСТУП

Потреба в автоматизації виникла ще в стародавні часи і неухильно зростала в процесі розвитку людських потреб. Винахід будь-яких нових засобів пересування чи виробництва тоді вимагав удосконалень, підвищення комфорту, безпеки для інших та, звичайно, простоти в експлуатації. Автоматизація у всі часи людства розвивала людину, давала змогу здобувати бажаного з меншими зусиллями й затратами. Завжди були й будуть задачі, процеси, які можливо автоматизувати, без цього наш світ не може змінюватися.

Поява й розповсюдження персональних комп’ютерів привели до діджиталізації у всіх сферах життя людини. Це дало неабиякий приріст швидкості розвитку людства. В наш час залишилось мало спеціальностей, які не використовують в своїх задачах комп’ютер. Теперішній людини навіть в повсякденному житті важко представити себе без комп’ютера, смартфона і так далі. Саме це створило нову еру автоматизації.

Тематика першого розділу присвячена огляду серверів та методів їх спілкування. Також, приділена увага і до оцінки їх ефективності.

У другому розділі присутній огляд платформи .NET Core та методів для створення вебсерверів на ній, були розглянуті, і методи оптимізації.

Третій розділ розглядає результати роботи обраних методів та порівняння їх.

У четвертому розділі був проведений функціонально-вартісний аналіз програмного продукту.

РОЗДІЛ 1 ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Теоретичні відомості про сервера

Сервером можна назвати відповідну програму або комп'ютер які надають функціонал, так званим, клієнтам. Під клієнтами, ми розуміємо, деякі програми або процеси яким надається певний функціонал, по типу, доступу до деяких данних або виконання обчислень. Сервер та клієнт можуть знаходитись на одному комп'ютері або спілкуватися в мережі, наприклад, інтернет або локальна сіть. Один сервер може обслуговувати декілька клієнтів та клієнт може використовувати декілька серверів. Клієнтом для сервера може бути програма яка сама є сервером.

Спікування між сервером та клієнтом, сьогоні, в більшості випадків, реалізовано за допомогою моделі запит-відповідь, у якій, клієнт відправляє запит до сервера, який в свою чергу, виконує певні дії та повертає відповідь клієнту, відповідь може містити в собі як результат запиту, наприклад, дані, так і свідчення про те що запит буде опрацьовано. Одним із прикладів, такого спілкування є протокол HTTP (Hypertext Transfer Protocol).

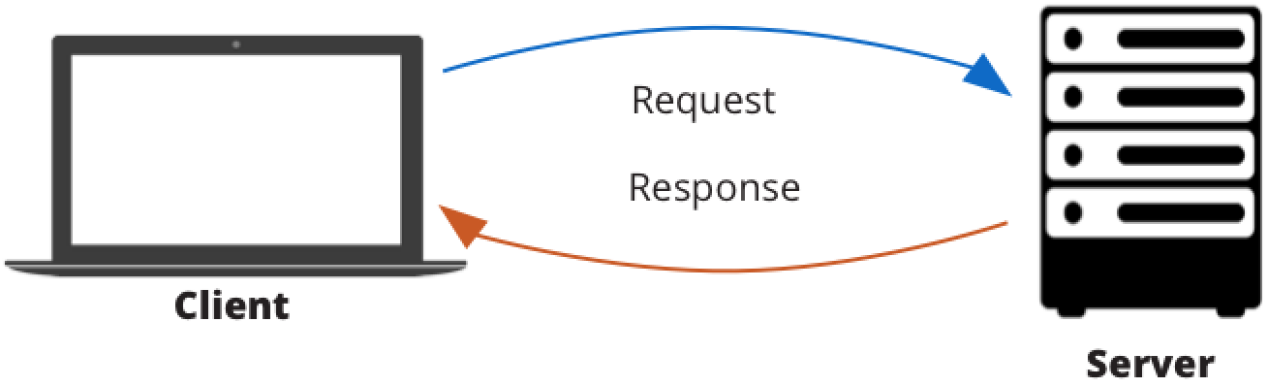


Рисунок 1.1 — Демонстація моделі запит-відповідь[1]

1.2 Типи серверів

Проанолізувавши цілі для яких використовують сервера, їх можна поділити на наступні категорії.

Сервер застосунків — сервер який виконує первні прикладні програми. Ці сервера дозволяють скліентам в мережі використовувати їх, без необхідності, встановлювати їх копію.

Сервер каталог — надає клієнтам можливість шукати інформацію в розподіленій мережі та представляє єдину точку доступу. Тобто, створюється індекс для базиданих, файлів та информації в певній мережі, що надалі дозволяє виконувати пошук в цій мережі.

Сервер комунікації — підтримує оточення яке необхідне для того щоб ондна кінцева точка могла знаходити та спілкуватися з іншими. В залежності від параметрів безпеки, цей сервер може містити сервіс для знаходження інших точок або вони будуть відомі заздалегіть.

Обчислювальний сервер — надає своїм клієнтам певні обчислювальні ресурси, в певній мережі. Часто, його також називають суперкомп'ютер.

Файловий сервер — надає доступ до файліт та папок своїм клієнтам в мережі.

Сервер баз даних — підтримує базуданих та надає клієнтам до неї доступ в мережі.

Медіасервер — надає доступ до цифрового відео або аудіо шляхом потокової медіа передачі(контент може бути переглянутий одразу після отримання, тобто, не має необхідності в повній передачі файлу).

Проксі-сервер — знаходиться посередині між кліентом та іншим сервером, при цему, приймає трафік клієнта та напрявляє його на інший сервер. Зазвичай, це роблять для фільтрації даних, покращення передачі трафіку або для передачі трафіку в складній мережі.

Вище були розглянуть декілька категорій, які частіше за все зустрічаються, сьогодні.

1.3 Спілкування клієнта та сервера

Для того щоб зрозуміти як працють сервера, важливо, розуміти як працює спілкування сервера з кліентом та клієнта з сервером. В свою чергу, для цього потрібно ознайомитись з термінами та понятями, які наведені нижче.

1.3.1 Комп’ютерана мережа

Комп’ютерна мережа — це набір ком’ютерів які з’єднані міє собою, тобто, можуть обмінюватись різними видами інформації, наприклад, текст, зображення або відео. Ком’ютерна мережа може складатися з групи підмереж. Комп’ютерні мережі розділяють на дві категорії — локальні та глобальні мережі. Їх, головна, відмінність в діапазоні з’єднання учасників мережі. Відповідно, глобальні мережі дозволять з’єднувани ком’ютери які знаходяться на великих відстраннях, навіть, на різних континентах. А локальні, в більшості випадків, в межах одної будівлі.

Важливо розуміти, яким чином члени мережі ідентифікують один одного, тобто, визначають хто отримає повідмлення. Для цього у кожному повідомлені вказано хто повинен його отримати, та, напевно обмінятися інформацією. Через це, для того щоб відправити повідомлення відправник повинен знати інформацію про одержувачя. Для цього, у кожного ком’ютера є унікальна MAC адреса. За допомогою цього, кожен пом’ютер отримує унікальну фізичку адресу. IP адресса, в свою чергу, є унікальним ідентифікатор ком’ютера в мережі. Вона складається з чотирьох 8ми бітних чисел які розділені крапками, наприклад 127.0.0.1, де кожне число приймає значення від 0 до 255.

1.3.2 Клієнт-серверна архітектура

Клієнт-серверна архітектура є одною із архітектурних шаблонів програмного забезпечення та дозволяє створювати розподілені мережеві застосунки, де, відповідні, робочі задачі розподіленні між сервісами або провайдерами ресурсів, які називають серверами. Споживачів цих ресурсів або даних називають, відповідно, клієнтами. Комп'ютерне забезпечення може хостити один або більше програм серверів, які надають їх ресурси клієнтам. Зазвичай, клієнти не надають ресурсів серверу, але споживають ресурси сервера. В свою чергу, клієнти створюють сесію спілкування з серверами, які чекають на, відповідний, запит. Для комунікації сервер та клієнт використовують мережевий протокол та прикладний програмний інтерфейс.

1.3.3 Мережеві протоколи

Мережевий протокол — це набір правил, які визначають, як інформація буде передана між застосунками, в нашому випадку, між клієнтом та сервером. Правила протоколу реалізовані в мережевому драйвері. Мережевий драйвер – це програмне забезпечення, яке форматує інформацію яку клієнт відправляє серверу та сервер клієнту.

Клієнти та сервер мають доступ до мережевого драйвера через мережевий прикладний програмний інтерфейс. Мережевий прикладний програмний інтерфейс складається з системних викликів або рутин програмних бібліотек які надають доступ до засобів мережевого зв'язку. В якості прикладу, засобу мережевого зв'язку, можна навести TLI (Transport Layer Interface) для сімейства операційних систем UNIX та WINSOCK для операційних систем Windows.

Ключова особливість, мережевих протоколів, полягає в можливості надання можливості спілкування сервера та клієнта, навіть, у випадках коли вони використовують різні операційні системи та програмні архітектури. Також, один сервер може використовувати декілька мережевих протоколів, у випадку, коли клієнти будуть їх підтримувати.

1.3.4 Мережевий прикладний програмний інтерфейс

Мережевий прикладний програмний інтерфейс — це прикладний програмний інтерфейс який складається з системних викликів або рутин цифрового спілкування. Програмне забеспечення, може, використовувати ці рутини для спілкування з іншими застосунками, які знаходяться на одному або різних ком'ютерах. В контексті, клієнт-серверної архітектури, сервер та клієнт використовують мережевий прикладний програмний інтерфейс для обміну інформацією згідно з, відповідним, мережевим протоколом. Для успішної комунікації, програмні оточення сервера та клієнта повині відповідати одному мережевому протоколу. Проте, деякі мережеві протоколи можуть бути реалізовані через декілька прикладних програмних інтерфейсів. Наприклад, мережевий протокол TCP/IP може бути досягнутий через сокет або TLI, в залежності від того який мережевий прикладний програмний інтерфейс доступний на, відповідній, операційній системі.

1.3.5 Мережева модель OSI

Модель OSI (EMBBC) (Open Systems Interconnection Basic Reference Model) в перекладі з англайської — базова еталонна модель взаємодії відкритих систем[1]. Представляє абстрактну мережеву модель для комунікацій та розробки нових протоколів. Ця модель складається з семи рівнів, що дозволяє

спростити тестування програмних продуктів які використовують цю модель для спілкування, тому що, ми можемо визначити на якому саме рівні виникла помилка. Інформація подоружує від одного рівня до іншого. Зазвичай, кожен рівень додає певні заголовки, які потріблі для коректної інтерпритації даних. Кожен рівень представлений певним протоколом. Протоколи можуть використовувати різне програмне або апаратне забезпечення. Рівні між собою не пов’язані, так як, виконують різні цілі. При відправлені повідомлення дані проходять з сьомого рівня до першого, а при отримані з першого до сьомого.

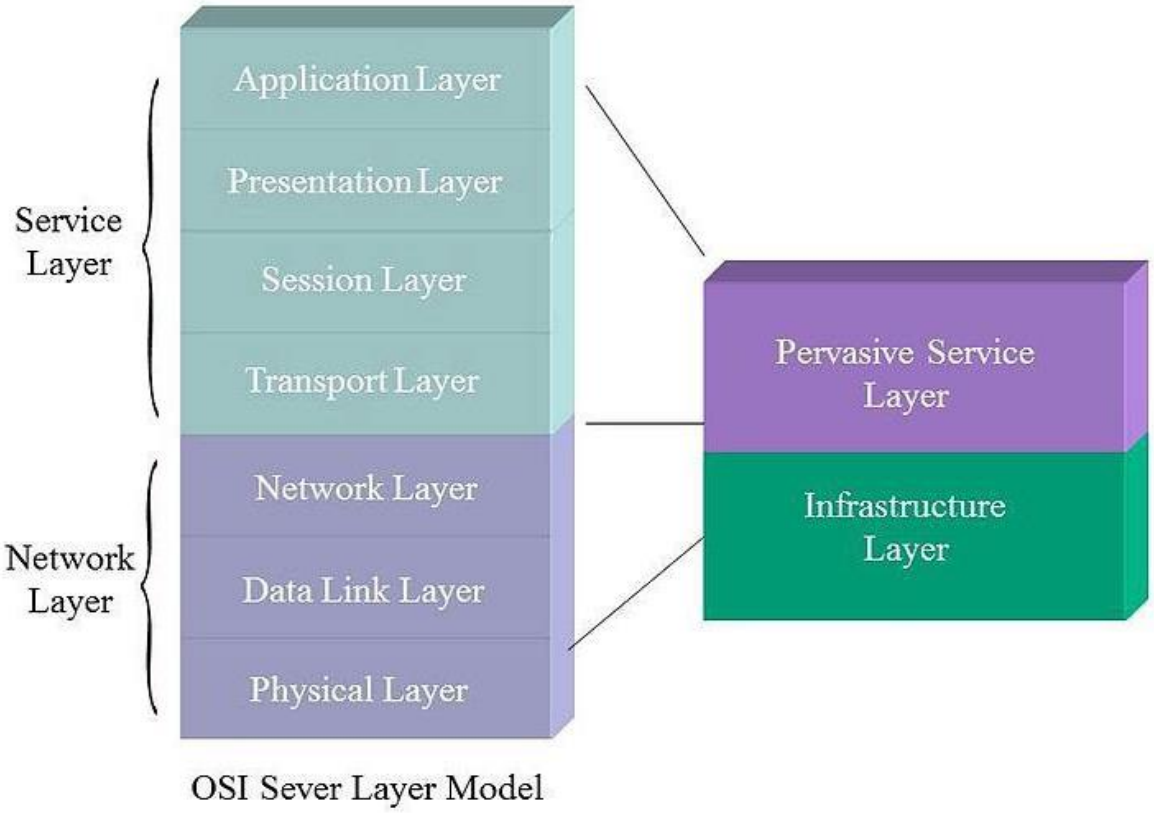


Рисунок 1.2 — Демонстрація моделі OSI[2]

Для кращого розуміння, мережевої моделі OSI, потрібно розібратися як інформація подоружує між рівнями. Для цього, розглянемо що відбувається з даними на кожному рівні.

На сьомому рівні, прикладний рівень (application layer), формуються дані які будуть відправленні, наприклад результата запиту до бази даних або елесктоний лист. Далі дані передаються на шостий рівень. Протоколи які відносяться до прокладного рівня: DNS (Domain Name System (Service) Protocol), FTP (File Transfer Protocol), HTTP (HyperText Transfer Protocol), NFS (Network File System).

На шостому, рівень представлення (presentation layer), дані перетворють згідно з форматом передачі по мережі або дані, отримані з мережі, перетворюють в формат який зрозумілий додаткам, які знаходяться на сьомому рівні, тобто, відбувається кодування або декодування. На цьому рівні, також, може проводитись стиснення або розпокування, що збільшує швидкість виконання рівнів нижче, та шифрування або дешифрування, що забезпечує таємність переданих даних. В якості прикладу протоколу, який знаходиться на шостому рівні і забезпечує, безпечне з’єднання, можна навести SSL (Secure Sockets Layer).

П’ятий, сеансовий рівень (session layer), відповідає за підтримання сеансу зв’язку, що дозволяє, клієнту та серверу взаємодіяти тривалий час. Рівень управляє створенням або завершенням сеансу, синхронізацією завдань, визначенням права на передачу даних, обміном інформацією і підтримкою сеансу в періоди неактивності програм. У потоці даних присутні контрольні точки, за допогою яких, забеспечується синхронізація передачі, тобто, при порушені взаємодії процес поновиться з останьої контрольної точки. Сеанси передачі складаються із запитів і відповідей, які здійснюються, у нашому випадку, клієнтом та сервером. Прикладами протоколів сеансового рівня є — X.235 (такоє відомий як ISO 8327), Zone Information Protocol (ZIP) та Session Control Protocol (SCP).

Четвертий, транспортний рівень (transport layer), відповідає за передачу даних без втрат, дублювання та в початковій послідовності. В залежності від

протоколу, дані розбиваються на блоки, тобто, короткі поєднуються в один а довгі розбиваються на декілька. Не всі протоколи гарантують цілісність, порядок та відсутність дуюлювання, наприклад, протокол UDP (User Datagram Protocol), але при цьому його швидкодія вища ніж у інших протоколів цього рівня які забеспечують, зазначені вище, умови. В якості приклада для протокола який гарантовано виконує умови, зазначені вище, можна навести протокол TCP (Transmission Control Protocol).

Трерій, мережний рівень (network layer), визначає шлях для передачі даних. Виконує, наступні, функції маршрутизація та комутація пакетів, визначення найкоротших маршрутів в мережі, відстеження неполадок в мережі. Багато мереж розділені на підмережі, тому для з’єднання з членами підмережі, використовують, так звані, роутери (маршрутизатори) для доставлення пакетів між мережами. Самий популярний протокол цього рівння — IP (Internet Protocol). Є два варіанти цього протоколу IPv4 та IPv6, головна відмість це розмір адреси учасника мережі, у IPv4 він складає чьотири байти, а у IPv6 шість байтів. IPv4 — це стара версія протоколу, але тим не менш, в більшості випадків використовують її.

Другий, канальний рівень або рівень каналів (data link layer), відповідає за передачу даних між учасниками, що перебувають в одному сегменті локальної мережі. Також може використовуватися для виявлення і, можливо, виправлення помилок, що виникли на фізичному рівні. Прикладами протоколів, що цьому рівні, є: Ethernet, Point-to-Point Protocol (PPP), HDLC та ADCCP.

Перший, фізичний рівень (physical layer), визначає метод передачі даних, представлених у двійковому вигляді, від одного пристрою (комп'ютера) до іншого[3]. До фізичного рівня відносяться наступні мережеві інтерфейси: RS-232, V.35, RS-485, RJ-45, RJ-11 та роз'єми BNC і AUI.

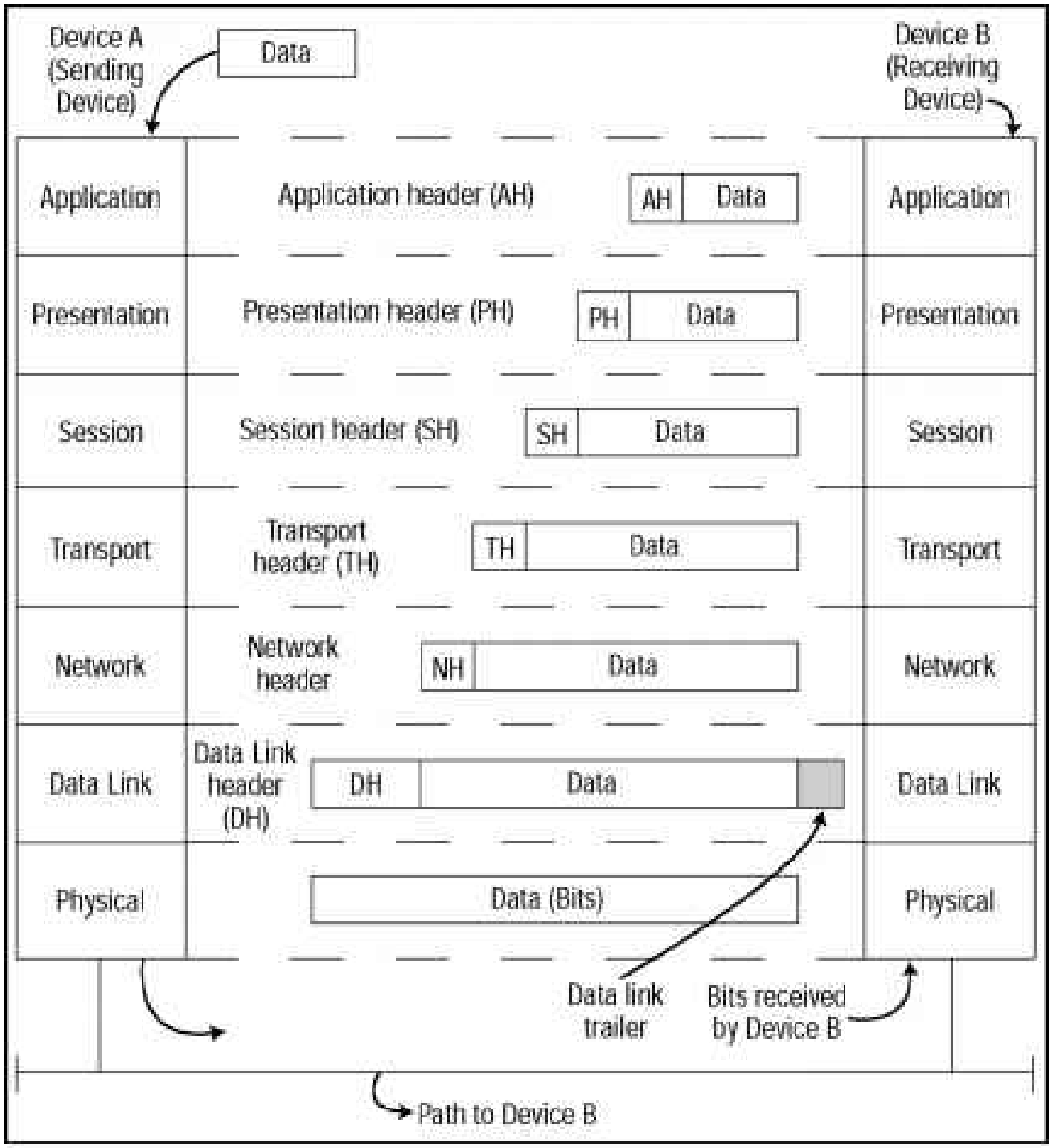


Рисунок 1.3 — Демонстрація роботи моделі OSI[3]

1.3.6 Локальні з’єднання

Сервер та клієнт з’єднані локально у випадку коли вони знаходяться на одному комп’ютері. У цьому випадку, вони створюють локальну мережу. Далі, будуть розглянуть різні види локальних з’єднань.

З’єднання через спільну ділянку пам’яті – це вид з’єднання який використовується на UNIX подібних операційних системах. Як можна зрозуміти з назви, спільна ділянка пам’ять використовується для спілкування сервера та клієнта. Клієнт не може мати більше одного з’єднання такого типу з сервером. Цей тип з’єднання надає високу швидкіть спілкування, але має деякі проблеми безпекою, шкідливі програми можуть переглянути або видалити повідомлення, у випадку якщо у них є доступ до цієї ділянки пам’яті.

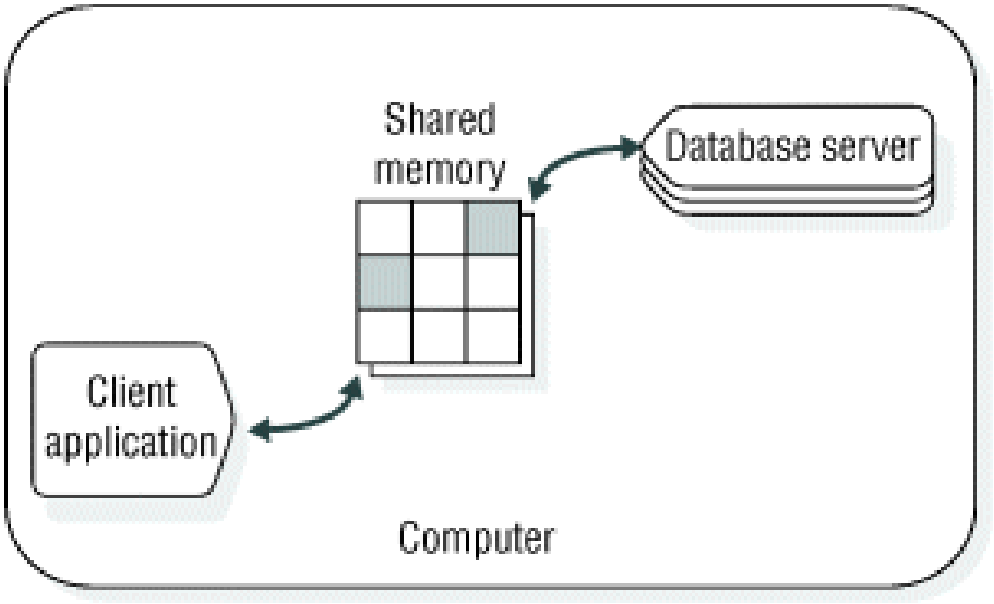


Рисунок 1.3 — Демонстрація з'єднання через спільну ділянку пам'яті, у випадку сервера базиданих[4]

Stream-pipe connection – це вид з’єднання який дозволяє процесам на одному ком’ютері спілкуватися між собою. Він реалізований на UNIX подібних операційних системах. Цей вид працює повільніше ніж з’єднання через спільну

ділянку пам’яті, але ситуація коли інший потік, який не бере участь в спілкувані, може прочитати або змінити вміст повідомлення, виключена.

1.4 Тесування ефективності(Performance Testing)

Тестування ефективності або продуктивності — це процес тестування програмного забезпечення, що використовується для тестування швидкості, часу відгуку, стабільності, надійності, масштабованості та використання ресурсів програмного додатку за певного навантаження. Основною метою тестування продуктивності є виявлення та усунення вузьких місць продуктивності програмного додатку. Основним завданням цього тестування є перевірка наступних параметвір програмного додатку, в нашому випадку сервера:

- швидкість — визначає, чи сервер достатньо швидко реагує;
- масштабованість — визначає максимальне завантаження клієнтами, яке може обробляти сервер;
- стабільність — визначає, чи стабільний сервер при різних навантаженнях.

1.5 Висновки до розділу 1

У розділі були розглянуті теоретичні відомості про сервера та вебсервера, методи оцінки їх ефективності та проблеми які можуть виникнути в процесі їх експлатації. Також, для кращого розуміння принципів їх роботи, були розглянуті різні методи комунікації сервера з клієнтом та навпаки.

РОЗДІЛ 2 РОЗРОБКА ПРОГРАМНОГО ЗАБЕЗВЕЧЕННЯ НА ПЛАТФОРМІ .NET CORE

2.1 Теоретичні відомості про .NET Framework

2.1.1 Означення .NET Framework та частини з яких він стладається

.NET Framework — це програмне забезпечення, розроблене корпорацією Microsoft, яке працює в основному на Microsoft Windows.[https://en.wikipedia.org/wiki/.NET_Framework] Він включає велику бібліотеку класів під назвою Framework Class Library (FCL) і забезпечує взаємодію мови (кожна мова може використовувати код, написаний іншими мовами) для декількох мов програмування. Програми, написані для .NET Framework, виконуються в програмному середовищі (на відміну від апаратного середовища) з назвою Common Language Runtime (CLR).

CLR — це віртуальна машина програми, яка надає такі послуги, як безпека, управління пам’яттю та обробка винятків. Таким чином, комп'ютерний код, написаний за допомогою .NET Framework, називається "керованим кодом"(managed code). FCL і CLR разом складають .NET Framework.

Проміжна мова (Intermediate Language (IL)) — це об'єктно-орієнтована мова програмування, призначена для використання компіляторами для .NET Framework перед статичним або динамічним компілюванням до машинного коду. Проміжна мова використовується .NET Framework для генерації незалежного від машини коду як вихід компіляції вихідного коду, написаного на будь-якій мові програмування .NET.

Проміжна мова — це мова збірок на основі стеку, яка перетворюється на байт-код під час виконання віртуальної машини. Він визначається специфікацією загальної мовної інфраструктури (Common Language Infrastructure (CLI)). Оскільки IL використовується для автоматичної генерації скомпільованого коду, немає необхідності вивчати його синтаксис.

Цей термін також відомий як проміжна мова Microsoft (Microsoft Intermediate Language (MSIL)) або загальна проміжна мова (Common Intermediate Language (CIL)).

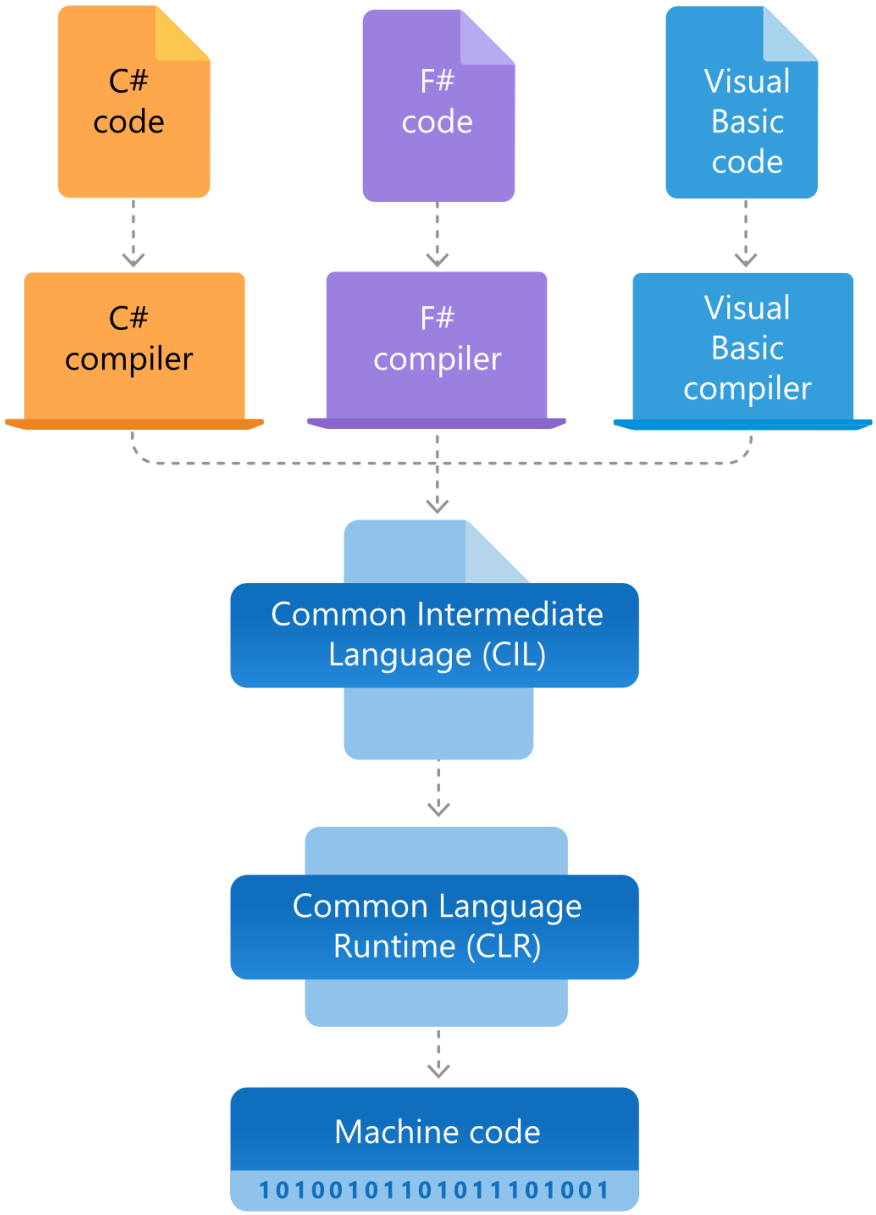


Рисунок 2.1 — Схема виконання коду на платформі .NET Framework[5]

FCL забезпечує користувальницький інтерфейс, доступ до даних, підключення до бази даних, криптографію, розробку веб-додатків, числові

алгоритми та мережеві комунікації. Програмісти виробляють програмне забезпечення, поєднуючи свій вихідний код із .NET Framework та іншими бібліотеками. Фреймворк призначений для використання більшістю нових програм, створених для платформи Windows. Microsoft також виробляє інтегроване середовище розробки програмного забезпечення .NET під назвою Visual Studio.

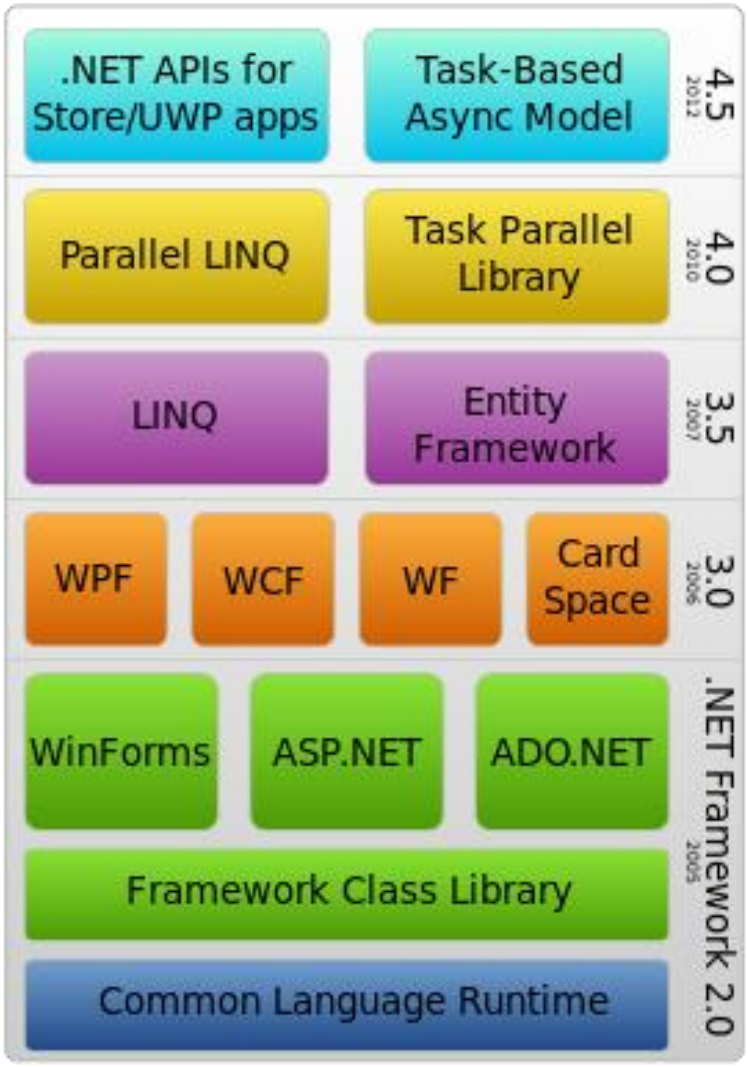


Рисунок 2.2 — Компонети FCL[6]

.NET Framework починався як власне програмне забезпечення, хоча фірма працювала над стандартизацією програмного пакета, майже відразу, навіть до його першого випуску. Незважаючи на зусилля, спрямовані на стандартизацію,

розробники, переважно ті, що належать до спільнот вільного програмного забезпечення та програм із відкритим кодом, висловили своє занепокоєння обраними умовами та перспективами будь-якого впровадження вільного та відкритого коду, особливо щодо патентів на програмне забезпечення. З тих пір Microsoft змінила розробку .NET, щоб більш точно слідувати сучасній моделі розробленого спільнотою проекту програмного забезпечення, включаючи випуск оновлення свого патенту, обіцяючи вирішити проблеми.

2.1.2 .NET Core, .NET Standard та Xamarin

.NET Core — це відкрита універсальна платформа розробки, яка підтримується корпорацією Майкрософт та спільнотою .NET на сайті GitHub. Вона є кроссплатформенною, підтримує Windows, Mac OS та Linux і може використовуватися на пристроях, на серверах, у вхідних системах та в сценаріях IoT (Інтернет-речовина). У цій основі лежать технології .NET Framework і Silverlight. Вона оптимізована для мобільних та серверних робочих навантажень.

.NET Standard — це офіційна специфікація .NET API, які доступні в декількох реалізаціях .NET. Мотивацією для створення.NET Standard було встановлення більшої однорідності екосистеми .NET. Однак .NET 5 застосовує інший підхід до встановлення єдиності, і цей новий підхід позбавляє потреби в .NET Standard у багатьох сценаріях.

Xamarin — це платформа з відкритим кодом для створення сучасних та продуктивних додатків для iOS, Android та Windows за допомогою .NET. Xamarin — це абстракційний рівень, який управляє зв'язком спільного коду з базовим кодом платформи. Xamarin працює в керованому середовищі, яке забезпечує такі зручності, як виділення пам'яті та збір сміття.

Xamarin дозволяє розробникам розподіляти в середньому 90% своїх програм між платформами. Цей шаблон дозволяє розробникам писати всю свою бізнес-логіку однією мовою (або використовувати код існуючуї програми повторно).

Програми на Xamarin можна писати на ПК або Mac і компілювати в рідні пакети програм, такі як файл .apk на Android або файл .ipa на iOS.

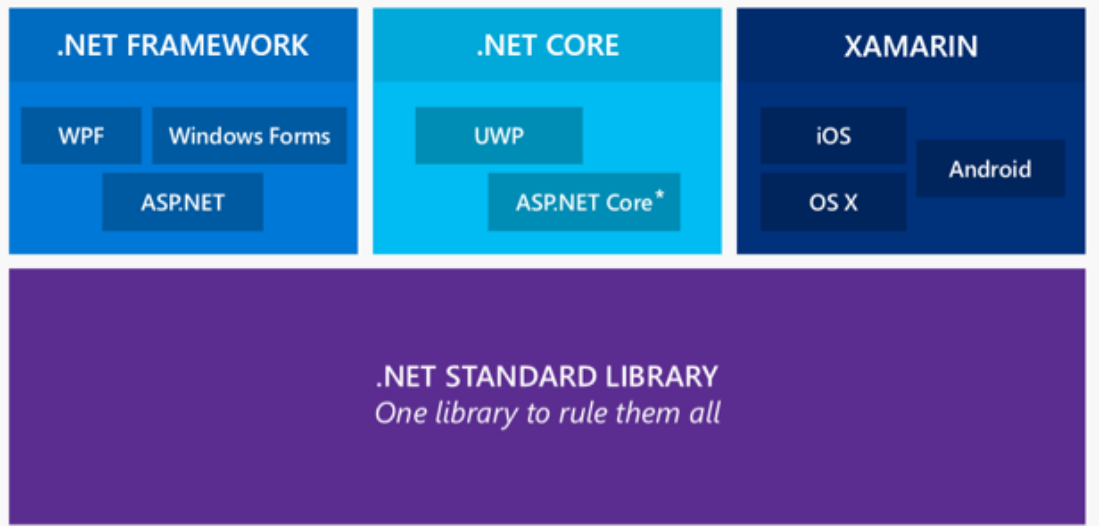


Рисунок 2.3 — Оточення .NET[7]

2.2 ASP.NET Core

ASP.NET Core - це платформа веб-розробки Microsoft. Оригінальний ASP.NET був представлений у 2002. ASP.NET Core складається з платформи для обробки HTTP-запитів, серії основних фреймворків для створення додатків, та вторинні утиліти, що забезпечують допоміжні функції, як показано на рисунку 2.4.

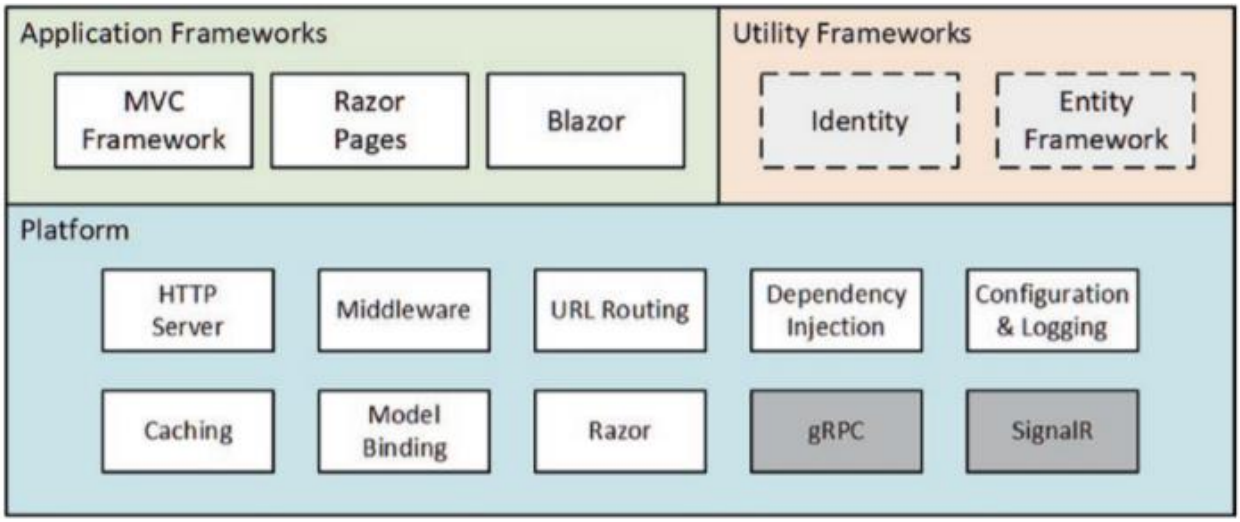


Рисунок 2.4 — Структура ASP.NET Core.[8, 30с]

2.2.1 MVC Framework

MVC Framework був представлений ще в основні періоди ASP.NET. Оригінальний ASP.NET спирався на модель розвитку під назвою Web Pages, яка відтворила досвід написання настільних програм, але призвела до громіздких веб-проектів, які не масштабувались. MVC Framework був представлений поряд із Web Pages з моделлю розробки, яка враховувала характер HTTP та HTML, а не намагатися приховати це.

MVC розшифровується як модель-представлення-контролер (Model-View-Controller), що являє собою шаблон дизайну, що описує форму програми. Шаблон MVC наголошує на розмежуванні проблем, де сфери функціональності визначаються незалежно, що було ефективним в порівняні з нечіткою архітектурою Web Pages.

Ранні версії MVC Framework були побудовані на основі ASP.NET, що спочатку була розроблена для Web Pages, що призвело до деяких незручних особливостей та обхідних шляхів. З переходом до .NET Core ASP.NET став

ASP.NET Core, а MVC Framework був перебудований на відкритому, розширюваному та міжплатформеному фундаменті.

MVC Framework залишається важливою частиною ASP.NET Core, але спосіб його використання змінився з ростом single-page applications (SPA). У SPA, браузер робить один запит HTTP і отримує HTML-документ, який доставляє

багатий клієнт, який зазвичай пишеться на клієнті JavaScript, наприклад Angular або React. Перехід до SPA означає, що чисте розділення яке присутнє в MVC Framework, не є настільки важливим, хоча MVC Framework залишається корисним і використовується для підтримки SPA через веб-сервіси.

2.2.2 Razor Pages

Одним недоліком MVC Framework є те, що він може потребувати великої підготовчої роботи, перш ніж програма зможе почати реалізовувати необхідну бізнес логіку. Незважаючи на структурні проблеми, однією перевагою Web Pages було те, що прості додатки можна було створити за пару годин.

Razor Pages приймає суть розробки Web Pages та реалізує їх, використовуючи функції платформи, розроблені для MVC Framework. Код і вміст змішуються, утворюючи самостійні сторінки; це відтворює швидкість розробки Web Pages без деяких основних технічних проблем (хоча питання масштабування складних проектів все ще може бути проблемою).

Сторінки Razor можна використовувати разом із MVC Framewor. Наприклад, основні частини додатоку, можна, створити використовуючи MVC Framework, і використати Razor Pages для додаткових функцій, таких як адміністрування або інструменти для звітування.

2.2.3 Blazor

Blazor дозволяє використовувати C# для написання програм на стороні клієнта. Існує дві версії Blazor: Blazor Server та Blazor WebAssembly.

Blazor Server — це стабільна та підтримувана частина ASP.NET Core, і вона працює за допомогою стійкого з'єднання через HTTP із сервером ASP.NET Core, де виконується код C # програми.

Blazor WebAssembly — це експериментальний випуск, який йде на крок далі і запускає код C # програми у браузері. Жодна з версій Blazor не підходить для будь-яких ситуацій, але вони дають представлення про напрямок майбутнього розвитку ASP.NET Core.

2.2.4 Utility Frameworks

Ці два фреймворки тісно пов'язані з ASP.NET Core, але не використовуються безпосередньо для генерації вмісту або даних HTML. Entity Framework Core — це об’єктно-реляційне відображення (object-relational mapping (ORM)) Microsoft, яке представляє дані, що зберігаються в реляційній базі даних як об'єкти .NET. Entity Framework Core можна використовувати в будь-якому додатку .NET Core, і він зазвичай використовується для доступу до баз даних у програмному забезпечені на ASP.NET Core.

ASP.NET Core Identity — це система автентифікації та авторизації Microsoft, і вона використовується для перевірки облікових даних користувачів у застосуваннях на ASP.NET Core та обмежує доступ до функцій сервера.

2.2.5 Платформа (Platform) ASP.NET Core

Платформа ASP.NET Core містить низькорівневі функції, необхідні для отримання та обробки HTTP-запитів та створення, відповідно HTTP-відповідей. Існує інтегрований HTTP-сервер, система компонентів проміжного програмного рівння для обробки запитів та ключові функції від яких залежать інші фреймворки, як наприклад, маршрутизація URL-адрес.

Більшу частину часу на розробку буде витрачено на Application Frameworks, але для ефективного використання ASP.NET Core потрібно розуміти можливості, які надає платформа ASP.NET Core, без яких не могли б функціонувати фрейворки вищого рівня.

Щоб, краще, зрозуміти ASP.NET Core, корисно зосередитись лише на ключових особливостях: pipeline запитів(request pipeline), посередники(middleware) та сервіси(services). Розуміння того, як ці функції поєднуються, навіть не вдаючись у подробиці, забезпечує корисний контекст для розуміння зміст проекту ASP.NET Core та платформи ASP.NET Core.

Призначення платформи ASP.NET Core — отримання HTTP-запитів та надсилання відповідей на них, які делегує ASP.NET Core, до посередників. Посередники розташовані в ланцюжку, який називається pipeline запитів. Коли надходить новий запит HTTP, платформа ASP.NET Core створює об’єкт, який його описує, та відповідний об’єкт що описує відповідь. Ці об'єкти передаються першому посереднику в ланцюжку, який аналізує запит і модифікує відповідь. Потім запит передається наступному посереднику в ланцюжку, при цьому кожен компонент аналізує запит і та, може, змінювати відповідь. Після того, як запит пробився по pipeline, платформа ASP.NET Core надсилає відповідь, як показано на малюнку[].

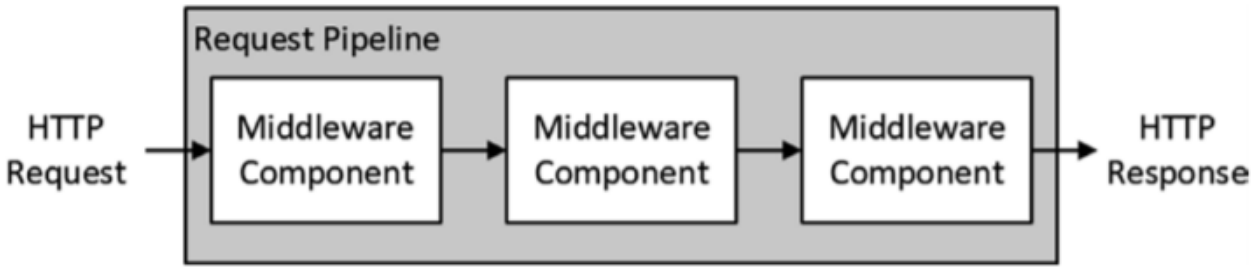


Рисунок 2.5 — The ASP.NET Core pipeline запиту[8, с255].

Деякі компоненти зосереджені на формуванні відповідей на запити, але існують інші для надання допоміжних функцій, таких як форматування певних типів даних або читання та запис файлів cookie. ASP.NET Core включає посередників, які вирішують загальні проблеми. Якщо жодний із посередників не створив відповідь, тоді ASP.NET Core поверне відповідь із HTTP 404 не знайдено (Not Found) кодом стану.

Сервіси — це об’єкти, що надають функції у веб-програмі. Будь-який клас можна використовувати як сервіс, також, немає обмежень на функції, які надають сервіси. Що робить сервіси особливими, так це те, що ними керує ASP.NET Core, і така функція як ін’єкція залежностей(dependency injection) дозволяє легко отримати доступ до сервісів у будь-якій точці програми, включаючи посередників.

Ін’єкція залежностей може бути складною для розуміння темою. Наразі достатньо знати, що існують об'єкти, якими керує платформа ASP.NET Core, якими можуть користуватися посередники, або координувати між компонентами або щоб уникнути дублювання загальних функцій, таких як логування або завантаження даних конфігурації, як показано на малюнку [].

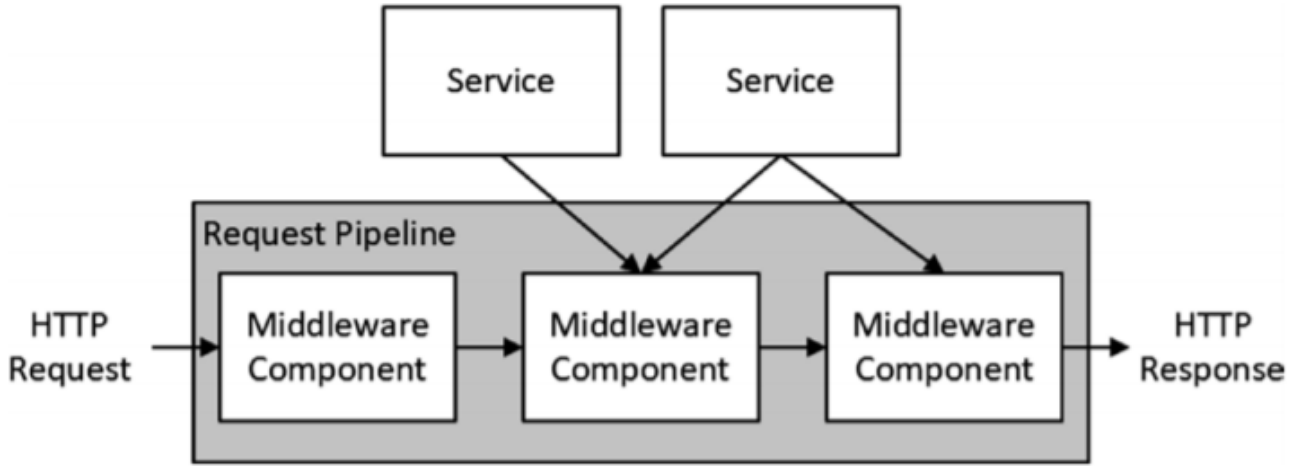


Рисунок 2.6 — Сервіси на платформі ASP.NET Core [8, с256].

Як видно з рисунка, посередники використовують лише ті сервіси, які потрібні для їх роботи. ASP.NET Core надає деякі основні сервіси, які можуть бути доповнені додатковими сервісами, специфічними для застосування.

Розглянемо дві важливі технології які надає платформа: SignalR та gRPC. SignalR використовується для створення низької затримки канали зв'язку між програмами. Він забезпечує основу для платформи Blazor Server, але SignalR рідко використовується безпосередньо, і є кращі альтернативи для тих декількох проектів, які потребують низької затримки обміну повідомленнями, наприклад, сітка подій Azure або службова шина Azure.

gRPC — це новий стандарт для міжплатформенних віддалених викликів процедур (RPC) через HTTP, який спочатку був створений Google (g у gRPC) та пропонує переваги ефективності та масштабованості. gRPC може бути майбутнім стандартом для веб-служб, але він не може використовувати у веб-додатках, оскільки для цього потрібен низькорівневий контроль над повідомленнями HTTP, які він надсилає, що не дозволяють браузери. (Існує бібліотека браузера, яка дозволяє використовувати gRPC через проксі-сервер, але це підриває переваги використання gRPC.) До gRPC можна використовувати в браузері, включення його в ASP.NET Core представляє інтерес лише для

проектів, які використовують його для зв'язку між серверами, для яких існує багато альтернативних протоколів.

2.3 Методи оптимізації

Є одна основна причина, чому ядро ASP.NET є улюбленим фреймворком для всіх — його продуктивність. Це забезпечує розробникам стабільну основу для створення веб-додатків. Однак важливо вчасно проводити перевірки, щоб переконатися, що продуктивність на потрібному рівні.

Далі описуються усі важливі поради щодо покращення продуктивності ваших програм ASP.NET Core.

2.3.1 Важливість використаної версії

Кожна версія має нове оновлення, розширені функції та кращу продуктивність. Наприклад, .NET Core 2.1 додав компілятор JIT, підтримку скомпільованих регулярних виразів та теги Spa <T>, тоді як версія 2.2 має підтримку HTTP2. ASP.NET Core 3.0 пропонує швидке читання та запис пам'яті, можливість розвантаження збірки тощо. Остання версія .NET, тобто .Net 5, планується випустити в листопаді 2020 року зі спрощеною обробкою та оптимізованими можливостями продуктивності.

Отже, створюючи програми з використанням ядра ASP.NET, завжди віддайте перевагу найновішій версії ASP.NET Core, оскільки Microsoft завжди покращує продуктивність останньої версії порівняно з попередньою.

2.3.2 Блокуючі запитів

Вам слід розробити програму ASP.NET Core для одночасного виконання декількох процесів. Цього досягти можна за допомогою асинхронного API, який обробляє тисячі запитів, не чекаючи блокуючих запитів. Замість того, щоб чекати завершення синхронного завдання, він може почати працювати над іншим потоком.

Проблема програм ASP.NET Core полягає у блокуванні асинхронного виконання за допомогою виклику Task.Wait або Task.Result. Вони блокують потік, поки завдання не буде завершено, і чекають його завершення. Крім того, уникайте виклику Task.Run, оскільки ASP.NET Core вже запускає потоки в звичайному пулі потоків, тому виклик Task.Run призведе до зайвого непотрібного планування в пулі потоків.

Натомість, бажано, зробити асинхронними «гарячий» код, тобто, операції вводу-виводу та довготривалих операцій, щоб операції могли виконуватися, не впливаючи на інші процеси.

2.3.3 Використання кешу

Для підвищення продуктивність програми, загальновідомий фокус полягає у зменшенні кількості запитів, що надсилаються на сервер. Як це працює: запит надсилається на сервер, а отримана відповідь зберігається. Отже, коли наступного разу здійснюється запит для подібної відповіді, замість запиту сервера, дані перевіряються із збереженими даними, і якщо вони збігаються, дані отримуються звідти, а не здійснюють запит на сервер.

Ось як кешування економить час і покращує загальну продуктивність. Існують три види кешування на стороні клієнта, сервера та клієнта/сервера. ASP.NET Core надає різні види кешування, таке як кешування в пам'яті, кешування відповідей, розподілене кешування тощо.

2.3.4 Оптимізація доступу до даних

Для підвищення продуктивності програми ми можемо оптимізувати логіку доступу до даних, у більшості випадків, до бази даних. Оскільки дані отримуються з бази даних, ця операція кожен раз вимагає багато часу. Ось декілька прийомів, які можна використовувати для підвищення продуктивності програми:

- зменште кількість HTTP запитів, тобто кількість мережевих зворотних викликів;
- замість того, щоб робити кілька запитів на сервер, краще, отримати дані в одному або двох запитах;
- встановіть кеш для незмінних даних;
- не отримуйте дані заздалегідь, якщо це не потрібно, це збільшує навантаження на відповідь і, отже, додаток сповільнюється.

2.4 Висновки до розділу номер 2

У даному розділі була розглянута платформа .NET Core, а саме, її придатність для створення вебсерверів, були розглянуті різні способи для реалізації цього. Приділена увага і тому через які компоненти ASP.NET Core проходить кожен запит та кожна відповідь. Також, в цьому розділі приділена увага методам, які можуть покращити ефективність вебсерверів на платформі .NET Core.

РОЗДІЛ 3 АНАЛІЗ РОЗРОБЛЕНОГО ПРОГРАМНОГО ПРОДУКТУ

3.1 Середовище виконання коду

3.1.1 Теоретичні відомості про Docker

В якості середовища для виконання програмних продуктів я вибрав Docker, тому що, ця технологія спрощює процес встановлення програмного забезпечення, надає можливість використовувати програмне забезпечення яке не підтримується на використовуємій операційній системі, значно спрощює процес публікації, відповідного, сервера та зводить до мінімуму різницю між середовищем розробки та робочим.

Docker — це відкрита платформа для розробки, доставки та запуску додатків. Docker дозволяє відокремити ваші програми від інфраструктури, щоб ви могли швидко доставити програмне забезпечення. За допомогою Docker ви можете керувати інфраструктурою так само, як і програмами. Скориставшись методологіями Docker для швидкої доставки, тестування та розгортання коду, ви можете значно скоротити затримку між написанням коду та його запуском у виробництво.

Docker надає можливість упаковувати та запускати додаток у вільно ізольованому середовищі, яке називається контейнером. Ізоляція та безпека дозволяють запускати багато контейнерів одночасно на одному хості. Контейнери легкі і містять усе необхідне для запуску програми, тому вам не потрібно покладатися на те, що зараз встановлено на хості. Ви можете легко ділитися контейнерами під час роботи та бути впевненими, що всі, з ким ви ділитесь, отримують той самий контейнер, який працює однаково.

Docker пропонує інструменти та платформу для управління життєвим циклом ваших контейнерів. Розробіть свою програму та допоміжні компоненти за допомогою контейнерів. Контейнер стає пристроєм для розповсюдження та тестування вашої програми. Коли будете готові, розгорніть свою програму у

виробничому середовищі, як контейнер або організовану службу. Це працює однаково, незалежно від того, чи є ваше виробниче середовище локальним центром обробки даних, хмарним постачальником чи гібридом двох.

Docker використовує архітектуру клієнт-сервер. Клієнт Docker розмовляє з демоном Docker, який виконує важкі роботи з будівництва, запуску та розподілу ваших контейнерів Docker. Клієнт Docker і демон можуть працювати в одній системі, або ви можете підключити клієнт Docker до віддаленого демона Docker. Клієнт Docker і демон взаємодіють за допомогою REST API, через сокети UNIX або мережевий інтерфейс. Іншим клієнтом Docker є Docker Compose, який дозволяє працювати з програмами, що складаються з набору контейнерів.

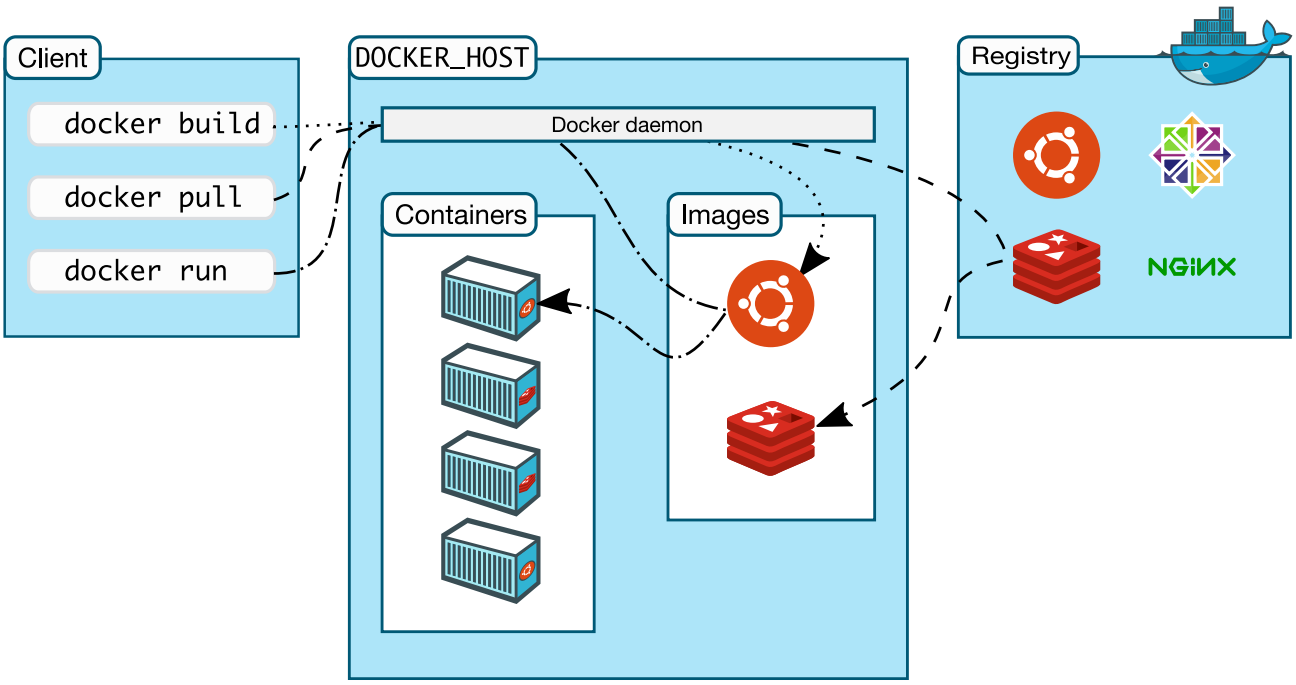


Рисунок 3.1 — Демонстрація архітектури Docker[9]

3.1.2 Dockerfile для ASP.NET Core

Docker може автоматично створювати образи, читаючи інструкції з Dockerfile. Dockerfile — це текстовий документ, який містить усі команди, які користувач може викликати в командному рядку для складання образу.

Розглянемо Dockerfile для ASP.NET Core, дивіться рисунок 3.2.

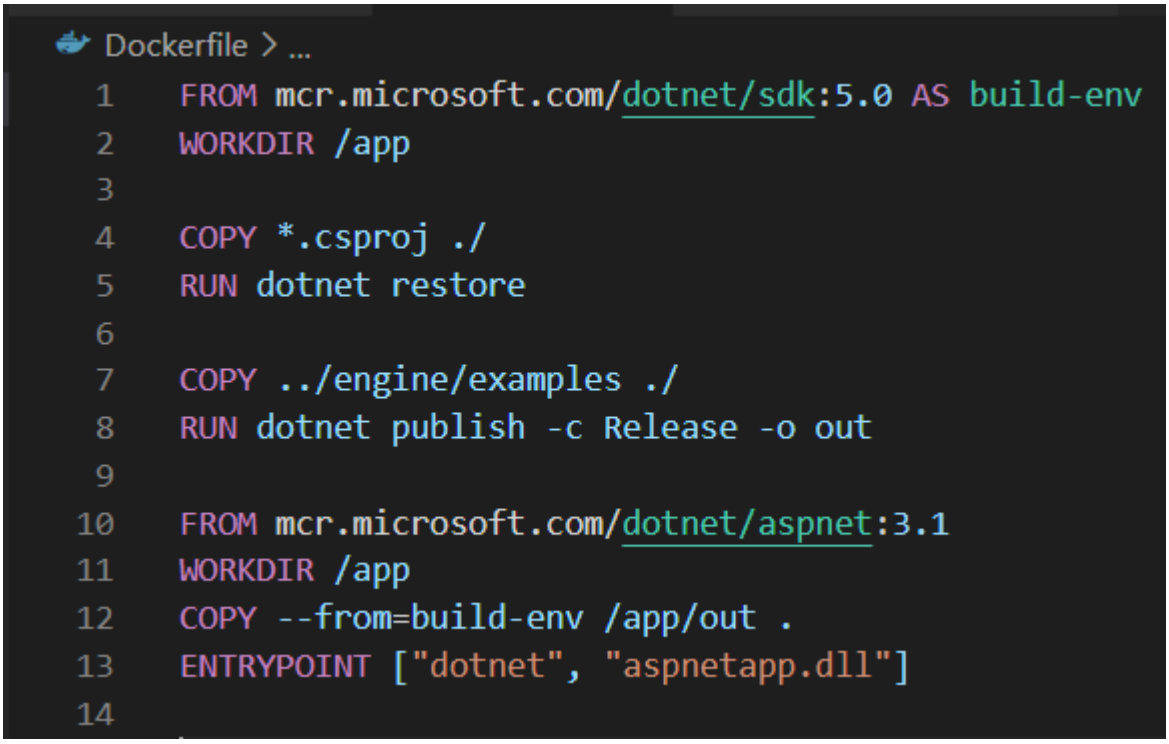


Рисунок 3.2 — Dockerfile для ASP.NET Core

За допомогою команди FROM, ми вказуємо базовий образ на основі якого буде створений необхідний нам образ. В цьому випадку це sdk (software development kit) для .NET, тобто, набір інстументів для розробки програмного забезпечення для платформи .NET. Далі за допомого команди WORKDIR ми вказуємо папку в якій будуть виконуватись всі подальші команди. Після цего, ми копіюємо всі .csproj файли в наш образ, ці файли представляють із себе файли проекту на .NET, з інформацією про залежності і не тільки. Далі ми встановлюємо всі залежності для кожного проекту. Останім кроком є створення dll файлу з проектом, який включає в себе процес компіляції. Кроки по

копіюваню файлів проекту та файлів програмного коду розділені на два різні етапи, це зроблено з цілью, зменшити час на створення нового образу та запуску контейнера на його основі, це викликано тим що, процес встановлення залежностей займає багато часу і його потрібно виконувати заново лише у випадку коли залежності для проекту змінилися, тобто, при зміні у файлі проекту, а не при кожній зміні програмного коду. Таким чином, про виконані процесу встановлення залежностей Docker використає результат одного з минулих виконань, якщо залежності не змінилися.

3.1.3 docker-compose для ASP.NET Core та PostgreSQL

PostgreSQL — це потужна об'єктно-реляційна система баз даних з відкритим кодом, яка активно працює понад 30 років, що забезпечило їй сильну репутацію надійності та продуктивності.

Compose — це інструмент для визначення та запуску багатоконтейнерних програм Docker. За допомогою Compose ви використовуєте файл YAML для налаштування служб програми. Потім за допомогою однієї команди ви створюєте та запускаєте всі служби з вашої конфігурації.

Розглянемо docker-compose файл для запуску ASP.NET Core та PostgreSQL на рисунку 3.3

docker-compose.yaml

```yaml
version: '3.7'

services:
  web:
    container_name: 'aspnetcoreapp'
    image: 'aspnetcoreapp'
    build:
      context: .
      dockerfile: aspnetcore.prod.dockerfile
    ports:
    - "5000:5000"
    depends_on:
    - "postgres"
    networks:
      - aspnetcoreapp-network

  postgres:
    container_name: 'postgres'
    image: postgres
    environment:
      POSTGRES_PASSWORD: sample
    networks:
      - aspnetcoreapp-network

networks:
  aspnetcoreapp-network:
    driver: bridge

```

Рисунок 3.3 — docker-compose для ASP.NET Core та PostgreSQL

1.2 Висновки до розділу номер 3

В ході виконання цього розділу, було проаналізоване програмне забезпечення на платформі .NET Core, яке виконує функції вебсервера. Була

приділена увага методам для розгортання додатків на цій платформі та принципам їх роботи. Були проаналізовані різні методи для покращення ефективності вебсерверів.

РОЗДІЛ 4 ФУНКЦІОНАЛЬНО-ВАРТІСНИЙ АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

У даному розділі проводиться оцінка основних характеристик програмного продукту, розробленого для дослідження ефективності вебсерверів на платформі .NET Core.

Нижче наведено аналіз різних варіантів реалізації вебсерверів з метою вибору оптимальної, з огляду при цьому як на економічні фактори, так і на характеристики продукту, що впливають на продуктивність роботи і на його сумісність з апаратним забезпеченням. Для цього було використано апарат функціонально-вартісного аналізу.

Функціонально-вартісний аналіз (ФВА) – це технологія, яка дозволяє оцінити реальну вартість продукту або послуги незалежно від організаційної структури компанії. ФВА проводиться з метою виявлення резервів зниження витрат за рахунок ефективніших варіантів виробництва, кращого співвідношення між споживчою вартістю виробу та витратами на його виготовлення. Для проведення аналізу використовується економічна, технічна та конструкторська інформація.

Алгоритм функціонально-вартісного аналізу включає в себе визначення послідовності етапів розробки продукту, визначення повних витрат (річних) та кількості робочих часів, визначення джерел витрат та кінцевий розрахунок вартості програмного продукту.

4.1 Постановка задачі проектування

У роботі застосовується метод ФВА для проведення техніко-економічного аналізу розробки вебсерверів. Оскільки рішення стосовно проектування та реалізації компонентів, що розробляється, впливають на всю

систему, кожна окрема підсистема має її задовольняти. Тому фактичний аналіз представляє собою аналіз функцій програмного продукту, який відповідає нашим вимогам.

Технічні вимоги до програмного продукту є наступні:

- стабільність та можливість відновлення у разі виникнення критичної помитлки;
- зручність та зрозумілість для користувача;
- швидкість обробки даних та доступ до інформації в реальному часі;
- можливість зручного масштабування та обслуговування;
- мінімальні витрати на впровадження програмного продукту.

4.2 Обґрунтування функцій програмного продукту

Головна функція F_0 – розробка програмного продукту, який реалізує необхідну бізнес-логіку та може одночасно працювати з декількома користувачами. Беручи за основу цю функцію, можна виділити наступні:

F_1 – вибір мови програмування;

F_2 – вибір фреймворку;

F_3 – вибір середовища розробки.

Кожна з цих функцій має декілька варіантів реалізації:

Функція F_1:

а) C#

б) C++

Функція F_2:

а) ASP.NET Core;

б) Wb

Функція F_3:

а) Jetbrains Rider;

б) Visual Studio.

Варіанти реалізації основних функцій наведені у морфологічній карті системи (рис. 4.1).

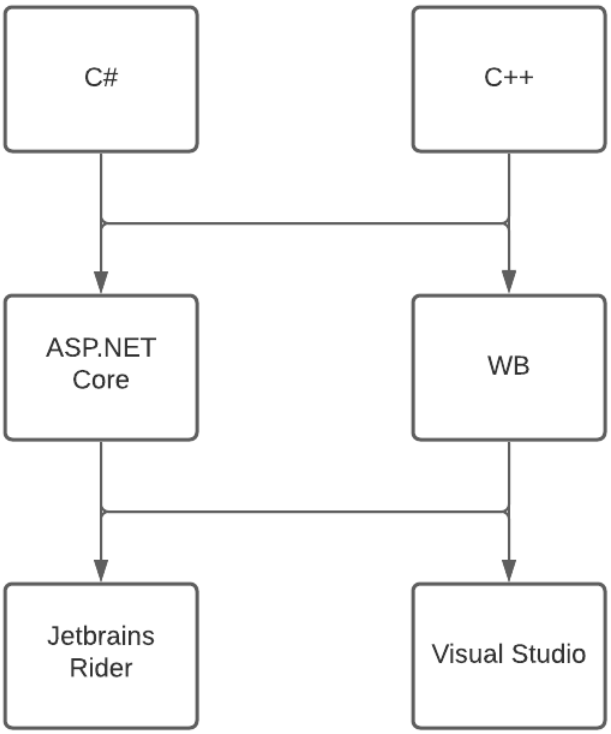


Рисунок 4.1 — Морфологічна карта

Морфологічна карта відображає множину всіх можливих варіанти основних функцій.

Таблиця 4.2 – Позитивно-негативна матриця

Функції	Варіанти реалізації	Переваги	Недоліки
F_1	A	Швидка розробка програми, доступність	Швидкість роботи трохи менша

		бібліотек, кросплатформеність	
	Б	Код швидко виконується	Іде багато часу на розробку програми
F_2	*А*	Надійно працює з складними проектами	Не підтримується багатьма мовами
	Б	Надійність	Додатковий час на інсталяцію та вивчення
F_3	*А*	Підтримується багатьма мовами програмування, легко запускається на будь-якому сервері	Додаткові витрати на лізензію
	Б	Багато інструментів, безпечна	Підтримує одночасно лише одну мову програмування

На основі цієї карти будуємо позитивно-негативну матрицю варіантів основних функцій (Таб.4.2). Робимо висновок, що при розробці програмного продукту деякі варіанти реалізації функцій варто відкинути, тому що вони не відповідають поставленим перед програмним продуктом задачам. Ці варіанти відзначені у морфологічній карті.

Функція F_1*:*

Перевагу віддаємо швидкості вивчення, простоті використання та наявності стандартних бібліотек для обчислення. Для спрощення роботи по написанню коду варіант Б має бути відкинутий.

Функція F_2*:*

Обидва варіанти можна використовувати в розробці.

Функція F_3:

Віддаємо перевагу варіанту А в разі вибору мови програмування C#.

Таким чином, будемо розглядати такий варіанти реалізації ПП:

F_1а – F_2а – F_3а

F_1а – F_2б – F_3а

Для оцінювання якості розглянутих функцій обрана система параметрів, описана нижче.

4.3 Обґрунтування системи параметрів ПП

На основі даних, розглянутих вище, визначаються основні параметри вибору, які будуть використані для розрахунку коефіцієнта технічного рівня.

Для того, щоб охарактеризувати програмний продукт, будемо використовувати наступні параметри:

- *X1* – швидкодія мови програмування;
- *X2* – об’єм пам’яті для обчислень та збереження даних;
- *X3* – час обробкт одного запиту;
- *X4* – потенційний об’єм програмного коду.

Гірші, середні і кращі значення параметрів вибираються на основі вимог замовника й умов, що характеризують експлуатацію ПП як показано у таблиці 4.4.

Таблиця 4.4 – Основні параметри ПП

Назва Параметра	Умовні позначення	Одиниці виміру	Значення параметра		
			гірші	середні	кращі
Швидкодія мови	X1	оп/мс	19889	25846	30073

програмування					
Об’єм пам’яті	X2	Мб	1479	1298	1114
Час обробкт одного запиту	X3	мс	65	46	33
Потенційний об’єм програмного коду	X4	кількість рядків коду	4943	3588	2386

За даними таблиці 4.3 будуються графічні характеристики параметрів – рис. 4.2 – рис. 4.5.

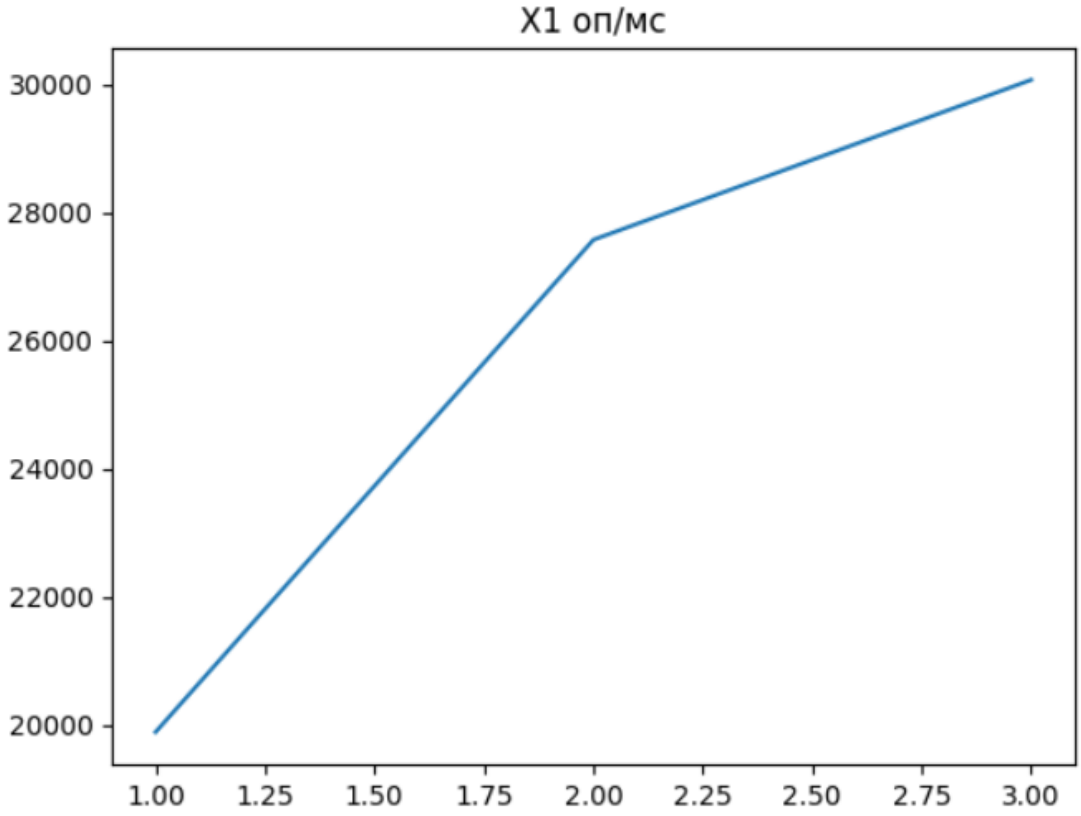


Рисунок 5.2 – X1, швидкодія мови програмування

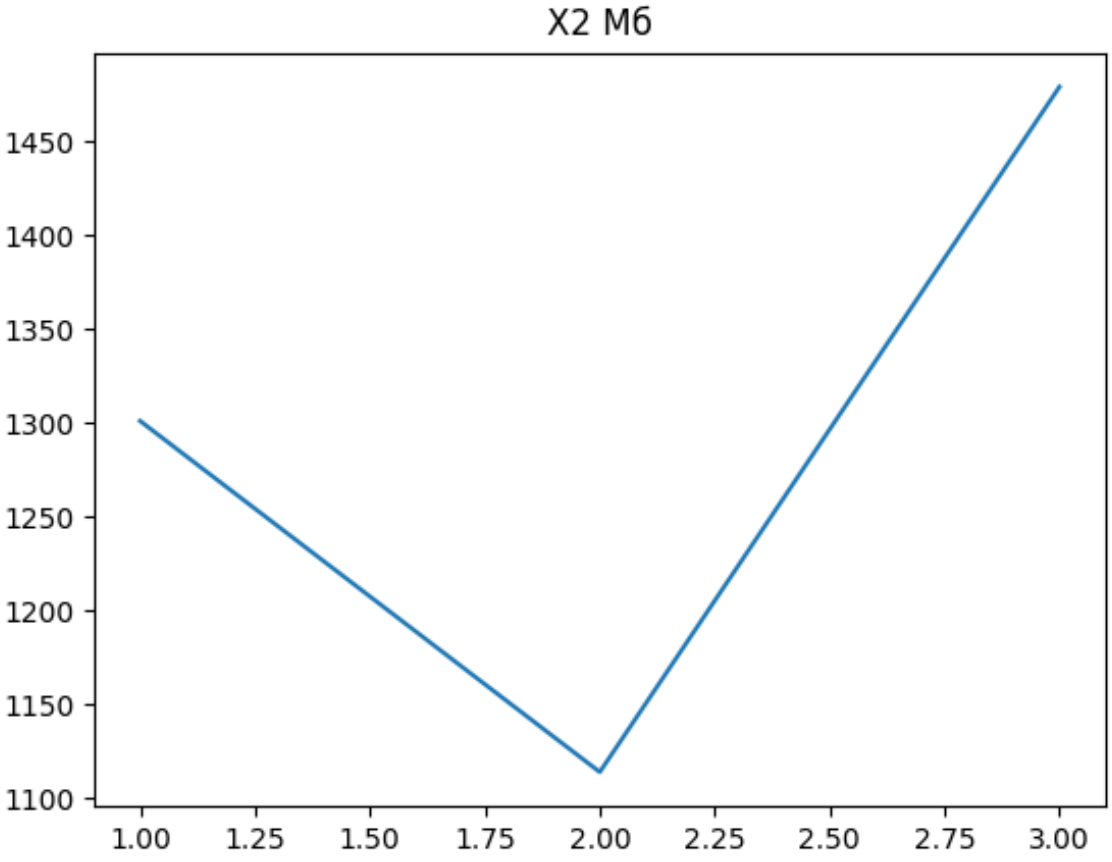


Рисунок 5.3 – X2, об’єм пам’яті

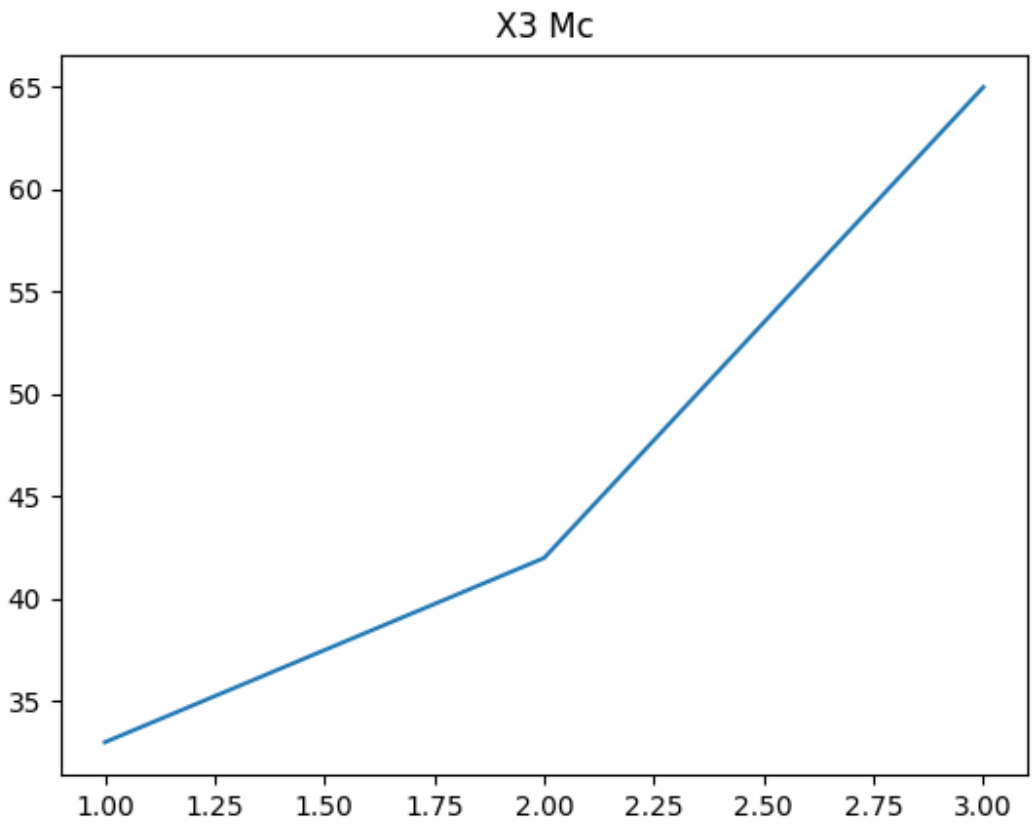


Рисунок 5.4 – X3, час на обробку одного запиту

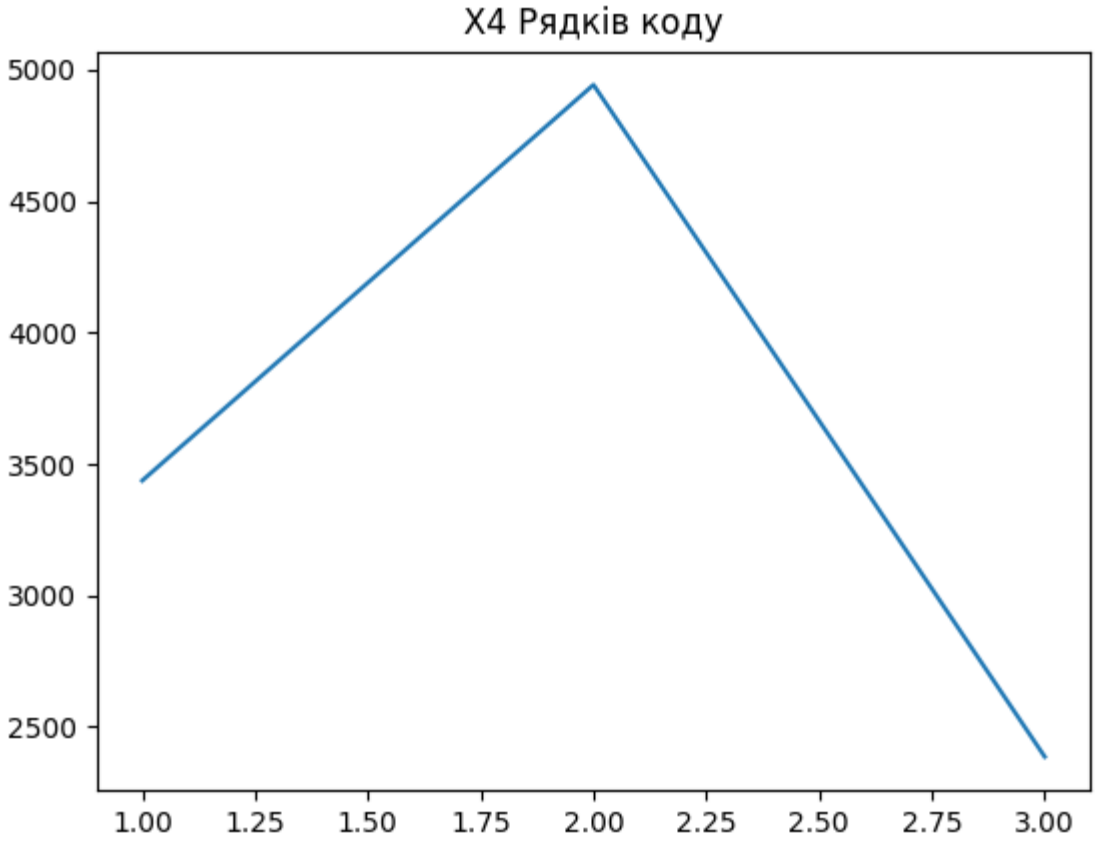


Рисунок 5.5 – X4, потенційний об’єм програмного коду

4.4 Аналіз експертного оцінювання параметрів

Після детального обговорення й аналізу кожний експерт оцінює ступінь важливості кожного параметру для конкретно поставленої цілі – розробка програмного продукту, який реалізує необхідну бізнес логіку та може одночасно працювати з декількома користувачами.

Значимість кожного параметра визначається методом попарного порівняння. Оцінку проводить експертна комісія із 7 людей. Визначення коефіцієнтів значимості передбачає:

– визначення рівня значимості параметра шляхом присвоєння різних рангів;
– перевірку придатності експертних оцінок для подальшого використання;
– визначення оцінки попарного пріоритету параметрів;
– обробку результатів та визначення коефіцієнту значимості.

Результати експертного ранжування наведені у таблиці 4.4.

Таблиця 4.4 – Результати ранжування параметрів

Позначення параметра	Назва параметра	Одиниці виміру	Ранг параметра за оцінкою експерта							Сума рангів R_i	Відхи-лення Δ_i	Δ_i^2
			1	2	3	4	5	6	7			
X1	Швидкодія мови програмування	Оп/мс	4	4	3	3	4	4	3	25	7,5	56,25
X2	Об’єм пам’яті	Mb	1	2	2	2	1	1	1	10	-7,5	56,25
X3	Час на обробку одного запиту	мс	3	3	4	4	3	3	4	24	6,5	42,25
X4	Потенційний об’єм програмного коду	Кількість рядків коду	2	1	1	1	2	2	2	11	-6,5	42,25
	Разом		10	10	10	10	10	10	10	70	0	41

Для перевірки степені достовірності експертних оцінок, визначимо наступні параметри:

а) сума рангів кожного з параметрів і загальна сума рангів:

$$R_i = \sum_{j=1}^{N} r_{ij} R_{ij} = \frac{Nn(n+1)}{2} = 70, \quad (4.1)$$

де N – число експертів,

n – кількість параметрів;

б) середня сума рангів:

$$T = \frac{1}{n} R_{ij} = 17{,}5 \quad (4.2)$$

в) відхилення суми рангів кожного параметра від середньої суми рангів:

$$\Delta_i = R_i - T. \quad (4.3)$$

Сума відхилень по всім параметрам повинна дорівнювати 0;

г)загальна сума квадратів відхилення:

$$S = \sum_{i=1}^{N} \Delta_i^2 = 197. \quad (4.4)$$

Порахуємо коефіцієнт узгодженості:

$$W = \frac{12S}{N^2(n^3 - n)} = \frac{12 \cdot 191}{7^2(4^3 - 4)} = 0{,}804081 > W_k = 0{,}67. \quad (4.5)$$

Ранжування можна вважати достовірним, тому що знайдений коефіцієнт узгодженості перевищує нормативний, котрий дорівнює 0,67.

Скориставшись результатами ранжирування, проведемо попарне порівняння всіх параметрів і результати занесемо у таблицю 4.5.

Таблиця 4.5 – Попарне порівняння параметрів.

Параметри	Експерти							Кінцева оцінка	Числове значення
	1	2	3	4	5	6	7		
X1 і X2	>	>	>	>	>	>	>	>	1,5
X1 і X3	>	>	<	<	>	>	<	>	1,5
X1 і X4	>	>	>	>	>	>	>	>	1,5
X2 і X3	<	<	<	<	<	<	<	<	0,5
X2 і X4	<	>	>	>	<	<	<	<	0,5
X3 і X4	>	>	>	>	>	>	>	>	1,5

Числове значення, що визначає ступінь переваги *i*–го параметра над *j*–тим, a_{ij} визначається по формулі:

$$a_{ij} = \begin{cases} 1.5 \text{ при } X_i > X_j \\ 1.0 \text{ при } X_i = X_j \\ 0.5 \text{ при } X_i < X_j \end{cases}. \quad (4.6)$$

З отриманих числових оцінок переваги складемо матрицю А=$\| a_{ij} \|$.

Для кожного параметра зробимо розрахунок вагомості $K_{вi}$ за наступними формулами:

$$K_{\text{вi}} = \frac{b_i}{\sum_{i=1}^{n} b_i} \quad (4.7)$$

$$b_i = \sum_{i=1}^{N} a_{ij} \tag{4.8}$$

Відносні оцінки розраховуються декілька разів доти, поки наступні значення не будуть незначно відрізнятися від попередніх (менше 2%). На другому і наступних кроках відносні оцінки розраховуються за наступними формулами:

$$K_{\text{ві}} = \frac{b_i'}{\sum_{i=1}^{n} b_i'}, \tag{4.9}$$

$$b_i' = \sum_{i=1}^{N} a_{ij} b_j \tag{4.10}$$

.Як видно з таблиці 4.6, різниця значень коефіцієнтів вагомості не перевищує 2%, тому більшої кількості ітерацій не потрібно.

Таблиця 4.6 – Розрахунок вагомості параметрів

Параметриx_i	Параметриx_j				Перша ітер.		Друга ітер.	
	X1	X2	X3	X4	b_i	$K_{\text{ві}}$	b_i^1	$K_{\text{ві}}^1$
X1	1,0	1,5	1,5	1,5	5,5	0,34	21,75	0,34
X2	0,5	1,0	0,5	0,5	2,5	0,16	9,25	0,16
X3	0,5	1,5	1,0	1,5	4,5	0,28	16,75	0,28
X4	0,5	1,5	0,5	1,0	4,5	0,28	16,75	0,28
Всього:					16	1	64,5	1

4.5 Аналіз рівня якості варіантів реалізації функцій

Визначаємо рівень якості кожного варіанту виконання основних функцій окремо.

Абсолютні значення параметрів *X2* (Об'єм пам'яті), *X3* (час обробки одного запиту) та *X4* (потенційний об'єм програмного коду) відповідають технічним вимогам умов функціонування даного ПП.

Абсолютне значення параметра *X1* (швидкість роботи мови програмування) обрано не найгіршим.

Коефіцієнт технічного рівня для кожного варіанта реалізації ПП розраховується так (таблиця 4.7):

$$K_K(j) = \sum_{i=1}^{n} K_{вi,j} B_{i,j}\,, \qquad (4.11)$$

де n – кількість параметрів;

$K_{вi}$– коефіцієнт вагомості i–го параметра;

B_i – оцінка i–го параметра в балах.

Таблиця 4.7 – Розрахунок показників рівня якості варіантів реалізації основних функцій ПП

Основні функції	Варіант реалізації функції	Параметри	Абсолютне значення параметра	Бальна оцінка параметра	Коефіцієнт вагомості параметра	Коефіцієнт рівня якості
F1	А	X1	10000	7	0,34	2,38
F2	А	X2	512	5	0,16	0,8
	Б	X2	1024	3	0,16	0,48
F3	А	X3	1500	8	0,28	2,24

За даними з таблиці 4.7 за формулою:

$$K_K = K_{\text{ТУ}}[F_{1k}] + K_{\text{ТУ}}[F_{2k}] + \ldots + K_{\text{ТУ}}[F_{zk}], \quad (4.12)$$

визначаємо рівень якості кожного з варіантів:

$$K_{K1} = 2{,}38 + 0{,}8 + 2{,}24 = 5{,}42,$$

$$K_{K2} = 2{,}38 + 0{,}48 + 2{,}24 = 5{,}1.$$

Як видно з розрахунків, кращим є перший варіант, для якого коефіцієнт технічного рівня має найбільше значення.

4.6 Економічний аналіз варіантів розробки ПП

Для визначення вартості розробки ПП спочатку проведемо розрахунок трудомісткості.

Всі варіанти включають в себе два окремих завдання:

1. Розробка проекту програмного продукту;

2. Розробка програмної оболонки;

Завдання 1 за ступенем новизни відноситься до групи А, завдання 2 – до групи Б. За складністю алгоритми, які використовуються в завданні 1 належать до групи 1; а в завданні 2 – до групи 3.

Для реалізації завдання 1 використовується довідкова інформація, а завдання 2 використовує інформацію у вигляді даних.

Проведемо розрахунок норм часу на розробку та програмування для кожного з завдань.

Загальна трудомісткість обчислюється як

$$T_О = T_Р \cdot К_П \cdot К_{СК} \cdot К_М \cdot К_{СТ} \cdot К_{СТ.М}, \quad (4.13)$$

де $T_Р$ – трудомісткість розробки ПП;

$К_П$ – поправочний коефіцієнт;

$К_{СК}$ – коефіцієнт на складність бізнес логіки;

$К_М$ – коефіцієнт рівня мови програмування;

$К_{СТ}$ – коефіцієнт використання стандартних модулів і прикладних програм;

$К_{СТ.М}$ – коефіцієнт стандартного математичного забезпечення

Для першого завдання, виходячи із норм часу для завдань розрахункового характеру степеню новизни А та групи складності алгоритму 1, трудомісткість дорівнює: $T_Р$ =90 людино-днів. Поправочний коефіцієнт, який враховує вид нормативно-довідкової інформації для першого завдання: $К_П$ = 1.7. Поправочний коефіцієнт, який враховує складність контролю вхідної та вихідної інформації для всіх семи завдань рівний 1: $К_{СК}$ = 1. Оскільки при розробці першого завдання використовуються стандартні модулі, врахуємо це за допомогою коефіцієнта $К_{СТ}$ = 0.8. Тоді загальна трудомісткість програмування першого завдання дорівнює:

$$T_1 = 90 \cdot 1.7 \cdot 0.8 = 122.4 \text{ людино-днів.}$$

Проведемо аналогічні розрахунки для подальших завдань.

Для другого завдання (використовується алгоритм третьої групи складності, степінь новизни Б), тобто $T_Р$ =27 людино-днів, $К_П$ =0.9,$К_{СК}$ = 1,$К_{СТ}$ =0.8:

$$T_2 = 27 \cdot 0.9 \cdot 0.8 = 19.44 \text{ людино-днів.}$$

Складаємо трудомісткість відповідних завдань для кожного з обраних варіантів реалізації програми, щоб отримати їх трудомісткість:

$$T_I = (122.4 + 19.44 + 6.7 + 19.44) \cdot 8 = 1343{,}84 \text{ людино-годин.}$$

$$T_{II} = (122.4 + 19.44 + 11.2 + 19.44) \cdot 8 = 1307.52 \text{ людино-годин.}$$

Найбільш високу трудомісткість має варіант II.

В розробці беруть участь два програмісти з окладом 34000 грн., один системний архітектор з окладом 45000. Визначимо середню зарплату за годину за формулою:

$$С_ч = \frac{М}{T_m \cdot t} \text{ грн.,} \quad (4.14)$$

де М – місячний оклад працівників;

T_m – кількість робочих днів в місяці;

t – кількість робочих годин в день.

$$С_ч = \frac{34000 + 34000 + 45000}{3 \cdot 21 \cdot 8} = 224{,}40 \text{ грн.} \quad (4.15)$$

Тоді, розрахуємо заробітну плату за формулою:

$$С_{ЗП} = С_ч \cdot Т_i \cdot К_Д\,, \quad (4.16)$$

де $С_ч$– величина погодинної оплати праці програміста;

$Т_i$ – трудомісткість відповідного завдання;

$К_Д$ – норматив, який враховує додаткову заробітну плату.

Зарплата розробників за варіантами становить:

4. $С_{ЗП} = 224{,}40 \cdot 1343{,}84 \cdot 1.2 = 361\,869{,}23$ грн.

II. $С_{ЗП} = 224{,}40 \cdot 1307.52 \cdot 1.2 = 383\,469{,}46$ грн.

Відрахування на єдиний соціальний внесок становить 22%:

4. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 102811{,}43 \cdot 0.22 = 79\,611{,}23$ грн.

II. $С_{ВІД} = С_{ЗП} \cdot 0.22 = 104117{,}62 \cdot 0.22 = 84\,363{,}28$ грн.

Тепер визначимо витрати на оплату однієї машино-години. ($С_М$)

Так як одна ЕОМ обслуговує одного програміста з окладом 340000 грн., з коефіцієнтом зайнятості 0,2 то для однієї машини отримаємо:

$$С_Г = 12 \cdot М \cdot К_З = 12 \cdot 34000 \cdot 0{,}2 = 81000 \text{ грн.}$$

З урахуванням додаткової заробітної плати:

$$С_{ЗП} = С_Г \cdot (1 + К_З) = 81000 \cdot (1 + 0.2) = 97\,920 \text{ грн.}$$

Відрахування на соціальний внесок:

$$С_{ВІД} = С_{ЗП} \cdot 0.22 = 97\,920 \cdot 0{,}22 = 21\,542{,}4 \text{ грн.}$$

Амортизаційні відрахування розраховуємо при амортизації 25% та вартості ЕОМ – 45000 грн.

$$С_А = К_{ТМ} \cdot К_А \cdot Ц_{ПР} = 1.15 \cdot 0.25 \cdot 45000 = 12\ 937{,}5 \text{ грн.},$$

де $К_{ТМ}$– коефіцієнт, який враховує витрати на транспортування та монтаж приладу у користувача;

$К_А$– річна норма амортизації;

$Ц_{ПР}$– договірна ціна приладу.

Витрати на ремонт та профілактику розраховуємо як:

$$С_Р = К_{ТМ} \cdot Ц_{ПР} \cdot К_Р = 1.15 \cdot 45000 \cdot 0.05 = 2\ 587{,}5 \text{ грн.},$$

де $К_Р$– відсоток витрат на поточні ремонти.

Ефективний годинний фонд часу ПК за рік розраховуємо за формулою:

$$Т_{ЕФ} = (Д_К – Д_В – Д_С – Д_Р) \cdot t_З \cdot К_В = (365 – 104 – 12 – 16) \cdot 8 \cdot 0.85 =$$

$$= 1\ 584.4 \text{ годин},$$

де $Д_К$ – календарна кількість днів у році;

$Д_В$, $Д_С$ – відповідно кількість вихідних та святкових днів;

$Д_Р$ – кількість днів планових ремонтів устаткування;

t –кількість робочих годин в день;

$К_В$– коефіцієнт використання приладу у часі протягом зміни.

Витрати на оплату електроенергії розраховуємо за формулою:

$$С_{ЕЛ} = Т_{ЕФ} \cdot N_С \cdot К_З \cdot Ц_{ЕН} = 1584{,}4 * 0.7 * 0.2 * 3.52 = 780{,}79 \text{ грн.},$$

де $N_С$ – середньо-споживча потужність приладу;

$К_З$– коефіцієнтом зайнятості приладу;

$Ц_{ЕН}$ – тариф за 1 КВт-годин електроенергії.

Накладні витрати розраховуємо за формулою:

$$С_Н = Ц_{ПР} \cdot 0.67 = 45000 \cdot 0{,}67 = 30\,150 \text{ грн.}$$

Тоді, річні експлуатаційні витрати будуть:

$$С_{ЕКС} = С_{ЗП} + С_{ВІД} + С_А + С_Р + С_{ЕЛ} + С_Н, \qquad (4.17)$$

$С_{ЕКС}$ = 97 920 + 21 542,4 + 12 937,5 + 2 587,5 + 780,79 + 30 150 = 138783,19 грн.

Собівартість однієї машино-години ЕОМ дорівнюватиме:

$$С_{М\text{-}Г} = С_{ЕКС} / Т_{ЕФ} = 138783{,}19 / 1584{,}4 = 87{,}59 \text{ грн/год.}$$

Оскільки в даному випадку всі роботи, які пов‘язані з розробкою програмного продукту ведуться на ЕОМ, витрати на оплату машинного часу, в залежності від обраного варіанта реалізації, складає:

$$C_М = C_{М-Г} \cdot T, \quad (4.18)$$

I. $C_М = 87,59 \cdot 1307.52 = 114\,525,67$ грн.

II. $C_М = 87,59 \cdot 1343,84 = 117\,706,94$ грн.

Накладні витрати складають 67% від заробітної плати:

$$C_Н = C_{ЗП} \cdot 0,67, \quad (4.19)$$

I. $C_Н = 114\,525,67 \cdot 0,67 = 76\,756,98$ грн.

II. $C_Н = 117\,706,94 \cdot 0,67 = 78\,863,64$ грн.

Отже, вартість розробки ПП за варіантами становить:

$$C_{ПП} = C_{ЗП} + C_{ВІД} + C_М + C_Н, \quad (4.20)$$

I. $C_{ПП} = 361\,869,23 + 79\,611,23 + 114\,525,67 + 76\,756,98 = 632763,11$ грн.

II. $C_{ПП} = 383\,469,46 + 84\,363,28 + 117\,706,94 + 78\,863,64 = 664403,32$ грн.

4.7 Вибір кращого варіанту ПП техніко-економічного рівня

Розрахуємо коефіцієнт техніко-економічного рівня за формулою:

$$К_{ТЕР j} = К_{Кj} / С_{Фj}, \quad (4.21)$$

$$К_{ТЕР1} = 5,42 / 632763,11 = 8,5 \cdot 10^{-6},$$

$$К_{ТЕР2} = 5,1 / 664403,32 = 7,6 \cdot 10^{-6}.$$

Як бачимо, найбільш ефективним є перший варіант реалізації програми з коефіцієнтом техніко-економічного рівня $К_{ТЕР1} = 8,5 \cdot 10^{-6}$.

4.8 Висновки до розділу 4

Проведено повний функціонально-вартісний аналіз програмного продукту. Визначено та проведено оцінку основних функцій програмного продукту. Визначено параметри, які характеризують програмний продукт. Проведено експертне оцінювання параметрів та аналіз якості варіантів реалізації функцій.

Проведено економічний аналіз варіантів розробки – трудомісткість, витрати на заробітну плату та інші витрати.

На основі аналізу вибрано варіант реалізації програмного продукту.

ВИСНОВКИ

У даній дипломній роботі було розглянуто методи для створення та оптимізації серверів на платформі .NET Core. У першому розділі була детально опрацьована предметна область зазначеного завдання.

В другому розділі було проаналізовано головні методи, що зазвичай використовуються для цього завдання. Також була приділена увага і методам оптимізації.

У третьому розділі були проаналізовані методи вивчені раніше.

ПЕРЕЛІК ПОСИЛАНЬ

1. Мережева модель OSI. Матеріал із Вікіпедії. URL: https://uk.wikipedia.org/wiki/%D0%9C%D0%B5%D1%80%D0%B5%D0%B6%D0%B5%D0%B2%D0%B0_%D0%BC%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_OSI
2. How do computers communicate with each other. URL: https://levelup.gitconnected.com/how-do-computers-communicate-with-each-other-50636acbeb4c
3. OSI Model and Protocols. URL: https://arunprashanth6.medium.com/osi-model-and-protocols-dd5795e6afb0
4. Shared-memory connections (UNIX). URL: https://www.ibm.com/docs/en/informix-servers/14.10?topic=connections-shared-memory-unix
5. What is .NET Framework. URL: https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet-framework
6. .NET Framework. Матеріал із Вікіпедії. URL: https://en.wikipedia.org/wiki/.NET_Framework
7. .NET Framework and .NET Standard. URL: https://www.chubbydeveloper.com/net-framework-v-net-core-net-standard/
8. Freeman A. Pro ASP.NET Core 3. Apress, 2020. 1400 p.
9. Docker overview. URL: https://docs.docker.com/get-started/overview/

ДОДАТОК А ЛІСТИНГ ПРОГРАМИ

Dockerfile

```
FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build-env
WORKDIR /app

COPY *.csproj ./
RUN dotnet restore

COPY ../engine/examples ./
RUN dotnet publish -c Release -o out

FROM mcr.microsoft.com/dotnet/aspnet:3.1
WORKDIR /app
COPY --from=build-env /app/out .
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

docker-compose.yaml

```
version: '3.7'

services:
  web:
    container_name: 'aspnetcoreapp'
    image: 'aspnetcoreapp'
    build:
      context: .
      dockerfile: aspnetcore.prod.dockerfile
    ports:
     - "5000:5000"
    depends_on:
     - "postgres"
    networks:
      - aspnetcoreapp-network

  postgres:
    container_name: 'postgres'
    image: postgres
    environment:
      POSTGRES_PASSWORD: sample
    networks:
      - aspnetcoreapp-network
```

```
networks:
  aspnetcoreapp-network:
    driver: bridge
```

Startup.cs

```
using Ardalis.ListStartupServices;
using BlazorAdmin;
using BlazorAdmin.Services;
using Blazored.LocalStorage;
using BlazorShared;
using Microsoft.AspNetCore.Authentication.Cookies;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.DataProtection;
using Microsoft.AspNetCore.Diagnostics.HealthChecks;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc.ApplicationModels;
using Microsoft.EntityFrameworkCore;
using Microsoft.eShopWeb.ApplicationCore.Interfaces;
using Microsoft.eShopWeb.Infrastructure.Data;
using Microsoft.eShopWeb.Infrastructure.Identity;
using Microsoft.eShopWeb.Web.Configuration;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Diagnostics.HealthChecks;
using Microsoft.Extensions.Hosting;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Net.Http;
using System.Net.Mime;

namespace Microsoft.eShopWeb.Web
{
    public class Startup
    {
        private IServiceCollection _services;

        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }
```

```
        public IConfiguration Configuration { get; }

        public void ConfigureDevelopmentServices(IServiceCollection
services)
        {
            ConfigureProductionServices(services);
        }

        public void ConfigureDockerServices(IServiceCollection services)
        {
            services.AddDataProtection()
                .SetApplicationName("eshopwebmvc")
                .PersistKeysToFileSystem(new DirectoryInfo(@"./"));

            ConfigureDevelopmentServices(services);
        }

        private void ConfigureInMemoryDatabases(IServiceCollection
services)
        {
            services.AddDbContext<AppIdentityDbContext>(options =>
                options.UseInMemoryDatabase("Identity"));

            ConfigureServices(services);
        }

        public void ConfigureProductionServices(IServiceCollection
services)
        {
            services.AddDbContext<CatalogContext>(c =>

c.UseSqlServer(Configuration.GetConnectionString("CatalogConnection")));
            services.AddDbContext<AppIdentityDbContext>(options =>

options.UseSqlServer(Configuration.GetConnectionString("IdentityConnectio
n")));

            ConfigureServices(services);
        }

        public void ConfigureTestingServices(IServiceCollection services)
        {
            ConfigureInMemoryDatabases(services);
        }

        public void ConfigureServices(IServiceCollection services)
```

```
        {
            services.AddCookieSettings();


services.AddAuthentication(CookieAuthenticationDefaults.AuthenticationSch
eme)
                .AddCookie(options =>
                {
                    options.Cookie.HttpOnly = true;
                    options.Cookie.SecurePolicy =
CookieSecurePolicy.Always;
                    options.Cookie.SameSite = SameSiteMode.Lax;
                });

            services.AddIdentity<ApplicationUser, IdentityRole>()
                       .AddDefaultUI()
                       .AddEntityFrameworkStores<AppIdentityDbContext>()
                                    .AddDefaultTokenProviders();

            services.AddScoped<ITokenClaimsService,
IdentityTokenClaimService>();

            services.AddCoreServices(Configuration);
            services.AddWebServices(Configuration);

            services.AddMemoryCache();
            services.AddRouting(options =>
            {
                options.ConstraintMap["slugify"] =
typeof(SlugifyParameterTransformer);
            });
            services.AddMvc(options =>
            {
                options.Conventions.Add(new
RouteTokenTransformerConvention(
                         new SlugifyParameterTransformer()));

            });
            services.AddControllersWithViews();
            services.AddRazorPages(options =>
            {
                options.Conventions.AuthorizePage("/Basket/Checkout");
            });
            services.AddHttpContextAccessor();
            services.AddHealthChecks();
            services.Configure<ServiceConfig>(config =>
```

```
            {
                config.Services = new List<ServiceDescriptor>(services);

                config.Path = "/allservices";
            });


            var baseUrlConfig = new BaseUrlConfiguration();
            Configuration.Bind(BaseUrlConfiguration.CONFIG_NAME,
baseUrlConfig);
            services.AddScoped<BaseUrlConfiguration>(sp =>
baseUrlConfig);
            services.AddScoped<HttpClient>(s => new HttpClient
            {
                BaseAddress = new Uri(baseUrlConfig.WebBase)
            });
            services.AddBlazoredLocalStorage();
            services.AddServerSideBlazor();

            services.AddScoped<HttpService>();
            services.AddBlazorServices();

            services.AddDatabaseDeveloperPageExceptionFilter();

            _services = services; // used to debug registered services
        }


        public void Configure(IApplicationBuilder app,
IWebHostEnvironment env)
        {
            var catalogBaseUrl = Configuration.GetValue(typeof(string),
"CatalogBaseUrl") as string;
            if (!string.IsNullOrEmpty(catalogBaseUrl))
            {
                app.Use((context, next) =>
                {
                    context.Request.PathBase = new
PathString(catalogBaseUrl);
                    return next();
                });
            }

            app.UseHealthChecks("/health",
                new HealthCheckOptions
                {
                    ResponseWriter = async (context, report) =>
```

```
                    {
                        var result = new
                        {
                            status = report.Status.ToString(),
                            errors = report.Entries.Select(e => new
                            {
                                key = e.Key,
                                value =
Enum.GetName(typeof(HealthStatus), e.Value.Status)
                            })
                        }.ToJson();
                        context.Response.ContentType =
MediaTypeNames.Application.Json;
                        await context.Response.WriteAsync(result);
                    }
                });
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
                app.UseShowAllServicesMiddleware();
                app.UseMigrationsEndPoint();
                app.UseWebAssemblyDebugging();
            }
            else
            {
                app.UseExceptionHandler("/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseBlazorFrameworkFiles();
            app.UseStaticFiles();
            app.UseRouting();

            app.UseCookiePolicy();
            app.UseAuthentication();
            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute("default",
"{controller:slugify=Home}/{action:slugify=Index}/{id?}");
                endpoints.MapRazorPages();
                endpoints.MapHealthChecks("home_page_health_check");
                endpoints.MapHealthChecks("api_health_check");
                //endpoints.MapBlazorHub("/admin");
                endpoints.MapFallbackToFile("index.html");
```

```
            });
    }}}
```

UserController.cs

```
using BlazorShared.Authorization;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.eShopWeb.ApplicationCore.Interfaces;
using System.Collections.Generic;
using System.Linq;
using System.Security.Claims;
using System.Threading.Tasks;

namespace Microsoft.eShopWeb.Web.Controllers
{
    [Route("[controller]")]
    [ApiController]
    public class UserController : ControllerBase
    {
        private readonly ITokenClaimsService _tokenClaimsService;

        public UserController(ITokenClaimsService tokenClaimsService)
        {
            _tokenClaimsService = tokenClaimsService;
        }

        [HttpGet]
        [Authorize]
        [AllowAnonymous]
        public async Task<IActionResult> GetCurrentUser() =>
            Ok(User.Identity.IsAuthenticated ? await CreateUserInfo(User)
: UserInfo.Anonymous);

        private async Task<UserInfo> CreateUserInfo(ClaimsPrincipal
claimsPrincipal)
        {
            if (!claimsPrincipal.Identity.IsAuthenticated)
            {
                return UserInfo.Anonymous;
            }

            var userInfo = new UserInfo
            {
                IsAuthenticated = true
            };
```

```
            if (claimsPrincipal.Identity is ClaimsIdentity
claimsIdentity)
            {
                userInfo.NameClaimType = claimsIdentity.NameClaimType;
                userInfo.RoleClaimType = claimsIdentity.RoleClaimType;
            }
            else
            {
                userInfo.NameClaimType = "name";
                userInfo.RoleClaimType = "role";
            }

            if (claimsPrincipal.Claims.Any())
            {
                var claims = new List<ClaimValue>();
                var nameClaims =
claimsPrincipal.FindAll(userInfo.NameClaimType);
                foreach (var claim in nameClaims)
                {
                    claims.Add(new ClaimValue(userInfo.NameClaimType,
claim.Value));
                }

                foreach (var claim in
claimsPrincipal.Claims.Except(nameClaims))
                {
                    claims.Add(new ClaimValue(claim.Type, claim.Value));
                }

                userInfo.Claims = claims;
            }

            var token = await
_tokenClaimsService.GetTokenAsync(claimsPrincipal.Identity.Name);
            userInfo.Token = token;

            return userInfo;
        }
    }
}
```

OrderController.cs

```
using MediatR;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.eShopWeb.Web.Features.MyOrders;
```

```
using Microsoft.eShopWeb.Web.Features.OrderDetails;
using System.Threading.Tasks;

namespace Microsoft.eShopWeb.Web.Controllers
{
    [ApiExplorerSettings(IgnoreApi = true)]
    [Authorize] // Controllers that mainly require Authorization still
use Controller/View; other pages use Pages
    [Route("[controller]/[action]")]
    public class OrderController : Controller
    {
        private readonly IMediator _mediator;

        public OrderController(IMediator mediator)
        {
            _mediator = mediator;
        }

        [HttpGet]
        public async Task<IActionResult> MyOrders()
        {
            var viewModel = await _mediator.Send(new
GetMyOrders(User.Identity.Name));

            return View(viewModel);
        }

        [HttpGet("{orderId}")]
        public async Task<IActionResult> Detail(int orderId)
        {
            var viewModel = await _mediator.Send(new
GetOrderDetails(User.Identity.Name, orderId));

            if (viewModel == null)
            {
                return BadRequest("No such order found for this user.");
            }

            return View(viewModel);
        }
    }
}
```

ManageController.cs

```
using Microsoft.AspNetCore.Authentication;
using Microsoft.AspNetCore.Authorization;
```

```
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using Microsoft.eShopWeb.ApplicationCore.Interfaces;
using Microsoft.eShopWeb.Infrastructure.Identity;
using Microsoft.eShopWeb.Web.Services;
using Microsoft.eShopWeb.Web.ViewModels.Manage;
using System;
using System.Linq;
using System.Text;
using System.Text.Encodings.Web;
using System.Threading.Tasks;

namespace Microsoft.eShopWeb.Web.Controllers
{
    [ApiExplorerSettings(IgnoreApi = true)]
    [Authorize] // Controllers that mainly require Authorization still
use Controller/View; other pages use Pages
    [Route("[controller]/[action]")]
    public class ManageController : Controller
    {
        private readonly UserManager<ApplicationUser> _userManager;
        private readonly SignInManager<ApplicationUser> _signInManager;
        private readonly IEmailSender _emailSender;
        private readonly IAppLogger<ManageController> _logger;
        private readonly UrlEncoder _urlEncoder;

        private const string AuthenticatorUriFormat =
"otpauth://totp/{0}:{1}?secret={2}&issuer={0}&digits=6";

        public ManageController(
          UserManager<ApplicationUser> userManager,
          SignInManager<ApplicationUser> signInManager,
          IEmailSender emailSender,
          IAppLogger<ManageController> logger,
          UrlEncoder urlEncoder)
        {
            _userManager = userManager;
            _signInManager = signInManager;
            _emailSender = emailSender;
            _logger = logger;
            _urlEncoder = urlEncoder;
        }

        [TempData]
        public string StatusMessage { get; set; }

        [HttpGet]
```

```
        public async Task<IActionResult> MyAccount()
        {
            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            var model = new IndexViewModel
            {
                Username = user.UserName,
                Email = user.Email,
                PhoneNumber = user.PhoneNumber,
                IsEmailConfirmed = user.EmailConfirmed,
                StatusMessage = StatusMessage
            };

            return View(model);
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Index(IndexViewModel model)
        {
            if (!ModelState.IsValid)
            {
                return View(model);
            }

            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            var email = user.Email;
            if (model.Email != email)
            {
                var setEmailResult = await
_userManager.SetEmailAsync(user, model.Email);
                if (!setEmailResult.Succeeded)
                {
                    throw new ApplicationException($"Unexpected error
occurred setting email for user with ID '{user.Id}'.");
                }
```

```
            }

            var phoneNumber = user.PhoneNumber;
            if (model.PhoneNumber != phoneNumber)
            {
                var setPhoneResult = await
_userManager.SetPhoneNumberAsync(user, model.PhoneNumber);
                if (!setPhoneResult.Succeeded)
                {
                    throw new ApplicationException($"Unexpected error
occurred setting phone number for user with ID '{user.Id}'.");
                }
            }

            StatusMessage = "Your profile has been updated";
            return RedirectToAction(nameof(Index));
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult>
SendVerificationEmail(IndexViewModel model)
        {
            if (!ModelState.IsValid)
            {
                return View(model);
            }

            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            var code = await
_userManager.GenerateEmailConfirmationTokenAsync(user);
            var callbackUrl = Url.EmailConfirmationLink(user.Id, code,
Request.Scheme);
            var email = user.Email;
            await _emailSender.SendEmailConfirmationAsync(email,
callbackUrl);

            StatusMessage = "Verification email sent. Please check your
email.";
            return RedirectToAction(nameof(Index));
        }
```

```
        [HttpGet]
        public async Task<IActionResult> ChangePassword()
        {
            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            var hasPassword = await _userManager.HasPasswordAsync(user);
            if (!hasPassword)
            {
                return RedirectToAction(nameof(SetPassword));
            }

            var model = new ChangePasswordViewModel { StatusMessage =
StatusMessage };
            return View(model);
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult>
ChangePassword(ChangePasswordViewModel model)
        {
            if (!ModelState.IsValid)
            {
                return View(model);
            }

            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            var changePasswordResult = await
_userManager.ChangePasswordAsync(user, model.OldPassword,
model.NewPassword);
            if (!changePasswordResult.Succeeded)
            {
                AddErrors(changePasswordResult);
                return View(model);
            }
```

```
            await _signInManager.SignInAsync(user, isPersistent: false);
            _logger.LogInformation("User changed their password
successfully.");
            StatusMessage = "Your password has been changed.";

            return RedirectToAction(nameof(ChangePassword));
        }

        [HttpGet]
        public async Task<IActionResult> SetPassword()
        {
            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            var hasPassword = await _userManager.HasPasswordAsync(user);

            if (hasPassword)
            {
                return RedirectToAction(nameof(ChangePassword));
            }

            var model = new SetPasswordViewModel { StatusMessage =
StatusMessage };
            return View(model);
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> SetPassword(SetPasswordViewModel
model)
        {
            if (!ModelState.IsValid)
            {
                return View(model);
            }

            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }
```

```
            var addPasswordResult = await
_userManager.AddPasswordAsync(user, model.NewPassword);
            if (!addPasswordResult.Succeeded)
            {
                AddErrors(addPasswordResult);
                return View(model);
            }

            await _signInManager.SignInAsync(user, isPersistent: false);
            StatusMessage = "Your password has been set.";

            return RedirectToAction(nameof(SetPassword));
        }

        [HttpGet]
        public async Task<IActionResult> ExternalLogins()
        {
            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            var model = new ExternalLoginsViewModel { CurrentLogins =
await _userManager.GetLoginsAsync(user) };
            model.OtherLogins = (await
_signInManager.GetExternalAuthenticationSchemesAsync())
                .Where(auth => model.CurrentLogins.All(ul => auth.Name !=
ul.LoginProvider))
                .ToList();
            model.ShowRemoveButton = await
_userManager.HasPasswordAsync(user) || model.CurrentLogins.Count > 1;
            model.StatusMessage = StatusMessage;

            return View(model);
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> LinkLogin(string provider)
        {
            // Clear the existing external cookie to ensure a clean login
process
            await
HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);
```

```
            // Request a redirect to the external login provider to link
a login for the current user
            var redirectUrl = Url.Action(nameof(LinkLoginCallback));
            var properties =
_signInManager.ConfigureExternalAuthenticationProperties(provider,
redirectUrl, _userManager.GetUserId(User));
            return new ChallengeResult(provider, properties);
        }

        [HttpGet]
        public async Task<IActionResult> LinkLoginCallback()
        {
            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            var info = await
_signInManager.GetExternalLoginInfoAsync(user.Id);
            if (info == null)
            {
                throw new ApplicationException($"Unexpected error
occurred loading external login info for user with ID '{user.Id}'.");
            }

            var result = await _userManager.AddLoginAsync(user, info);
            if (!result.Succeeded)
            {
                throw new ApplicationException($"Unexpected error
occurred adding external login for user with ID '{user.Id}'.");
            }

            // Clear the existing external cookie to ensure a clean login
process
            await
HttpContext.SignOutAsync(IdentityConstants.ExternalScheme);

            StatusMessage = "The external login was added.";
            return RedirectToAction(nameof(ExternalLogins));
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
```

```
        public async Task<IActionResult> RemoveLogin(RemoveLoginViewModel
model)
        {
            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            var result = await _userManager.RemoveLoginAsync(user,
model.LoginProvider, model.ProviderKey);
            if (!result.Succeeded)
            {
                throw new ApplicationException($"Unexpected error
occurred removing external login for user with ID '{user.Id}'.");
            }

            await _signInManager.SignInAsync(user, isPersistent: false);
            StatusMessage = "The external login was removed.";
            return RedirectToAction(nameof(ExternalLogins));
        }

        [HttpGet]
        public async Task<IActionResult> TwoFactorAuthentication()
        {
            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            var model = new TwoFactorAuthenticationViewModel
            {
                HasAuthenticator = await
_userManager.GetAuthenticatorKeyAsync(user) != null,
                Is2faEnabled = user.TwoFactorEnabled,
                RecoveryCodesLeft = await
_userManager.CountRecoveryCodesAsync(user),
            };

            return View(model);
        }

        [HttpGet]
        public async Task<IActionResult> Disable2faWarning()
```

```
        {
            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            if (!user.TwoFactorEnabled)
            {
                throw new ApplicationException($"Unexpected error occured
disabling 2FA for user with ID '{user.Id}'.");
            }

            return View(nameof(Disable2fa));
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Disable2fa()
        {
            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            var disable2faResult = await
_userManager.SetTwoFactorEnabledAsync(user, false);
            if (!disable2faResult.Succeeded)
            {
                throw new ApplicationException($"Unexpected error occured
disabling 2FA for user with ID '{user.Id}'.");
            }

            _logger.LogInformation("User with ID {UserId} has disabled
2fa.", user.Id);
            return RedirectToAction(nameof(TwoFactorAuthentication));
        }

        [HttpGet]
        public async Task<IActionResult> EnableAuthenticator()
        {
            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
```

```
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            var unformattedKey = await
_userManager.GetAuthenticatorKeyAsync(user);
            if (string.IsNullOrEmpty(unformattedKey))
            {
                await _userManager.ResetAuthenticatorKeyAsync(user);
                unformattedKey = await
_userManager.GetAuthenticatorKeyAsync(user);
            }

            var model = new EnableAuthenticatorViewModel
            {
                SharedKey = FormatKey(unformattedKey),
                AuthenticatorUri = GenerateQrCodeUri(user.Email,
unformattedKey)
            };

            return View(model);
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult>
EnableAuthenticator(EnableAuthenticatorViewModel model)
        {
            if (!ModelState.IsValid)
            {
                return View(model);
            }

            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            // Strip spaces and hypens
            var verificationCode = model.Code.Replace(" ",
string.Empty).Replace("-", string.Empty);

            var is2faTokenValid = await
_userManager.VerifyTwoFactorTokenAsync(
```

```
                user,
_userManager.Options.Tokens.AuthenticatorTokenProvider,
verificationCode);

            if (!is2faTokenValid)
            {
                ModelState.AddModelError("model.TwoFactorCode",
"Verification code is invalid.");
                return View(model);
            }

            await _userManager.SetTwoFactorEnabledAsync(user, true);
            _logger.LogInformation("User with ID {UserId} has enabled 2FA
with an authenticator app.", user.Id);
            return RedirectToAction(nameof(GenerateRecoveryCodes));
        }

        [HttpGet]
        public IActionResult ResetAuthenticatorWarning()
        {
            return View(nameof(ResetAuthenticator));
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> ResetAuthenticator()
        {
            var user = await _userManager.GetUserAsync(User);
            if (user == null)
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            await _userManager.SetTwoFactorEnabledAsync(user, false);
            await _userManager.ResetAuthenticatorKeyAsync(user);
            _logger.LogInformation("User with id '{UserId}' has reset
their authentication app key.", user.Id);

            return RedirectToAction(nameof(EnableAuthenticator));
        }

        [HttpGet]
        public async Task<IActionResult> GenerateRecoveryCodes()
        {
            var user = await _userManager.GetUserAsync(User);
            if (user == null)
```

```
            {
                throw new ApplicationException($"Unable to load user with
ID '{_userManager.GetUserId(User)}'.");
            }

            if (!user.TwoFactorEnabled)
            {
                throw new ApplicationException($"Cannot generate recovery
codes for user with ID '{user.Id}' as they do not have 2FA enabled.");
            }

            var recoveryCodes = await
_userManager.GenerateNewTwoFactorRecoveryCodesAsync(user, 10);
            var model = new GenerateRecoveryCodesViewModel {
RecoveryCodes = recoveryCodes.ToArray() };

            _logger.LogInformation("User with ID {UserId} has generated
new 2FA recovery codes.", user.Id);

            return View(model);
        }

        private void AddErrors(IdentityResult result)
        {
            foreach (var error in result.Errors)
            {
                ModelState.AddModelError(string.Empty,
error.Description);
            }
        }

        private string FormatKey(string unformattedKey)
        {
            var result = new StringBuilder();
            int currentPosition = 0;
            while (currentPosition + 4 < unformattedKey.Length)
            {
                result.Append(unformattedKey.Substring(currentPosition,
4)).Append(" ");
                currentPosition += 4;
            }
            if (currentPosition < unformattedKey.Length)
            {
                result.Append(unformattedKey.Substring(currentPosition));
            }

            return result.ToString().ToLowerInvariant();
```

```
        }

        private string GenerateQrCodeUri(string email, string
unformattedKey)
        {
            return string.Format(
                AuthenticatorUriFormat,
                _urlEncoder.Encode("eShopOnWeb"),
                _urlEncoder.Encode(email),
                unformattedKey);
        }
    }
}
```

ДОДАТОК Б ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

Слайд 1

Розробка програмного забезпечення для аналізу ефективності серверів на платформі .NET Core

Виконав: Вдовиченко Дмитро Юрійович
Науковий керівник: Асистент кафедри ММСА к.т.н
Гуськова Віра Геннадіївна

Слайд 2

Постановка задачі

- Дослідити механізми роботи .NET Core
- Ознайомитись з методами оцінки ефективності вебсерверів
- Дослідити методи оптимізації вевсерверів на платформі .NET Core
- Розробити програмний додаток для подальших експериментів
- Зрівняти результати з використанням методів оптимізації та без

Слайд 3

Мета Роботи	Об’єкт Дослідження	Предмет Дослідження
Аналіз існуючих методів створення вебсерверів на платформі .NET Core та методи їх опримізації	Сервера та вебсервера, аналіз їх ефективності та проблеми при створені	Інструменти для створення вебсерверів на платформі .NET Core та методи їх оптимізації

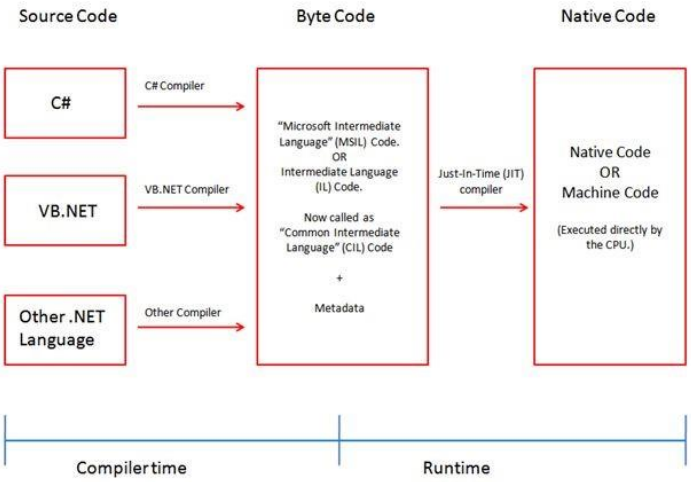
Слайд 4

Актуальність

- Потреби середнього та великого бізнесу в адмініструванні та автоматизації
- Кількість користувачів онлайн сервісів, щодено, збільшується
- Онлайн сервіси заміщують десктопні додатки

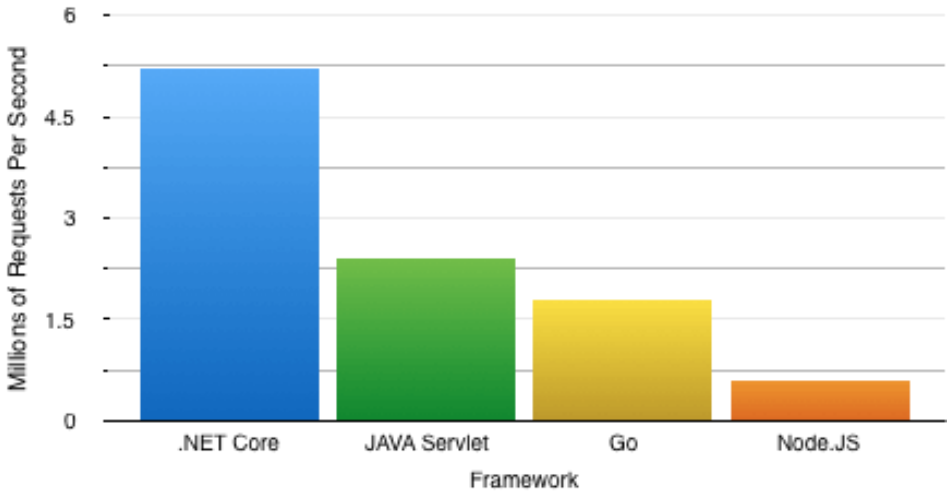
Слайд 5

Як .NET Core виконує код



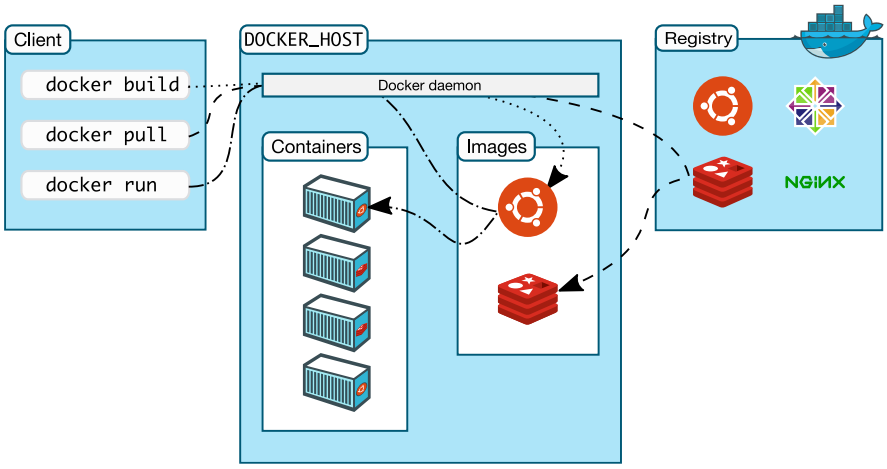
Слайд 6

Зрівняння .NET Core з іншими популярними технологіями



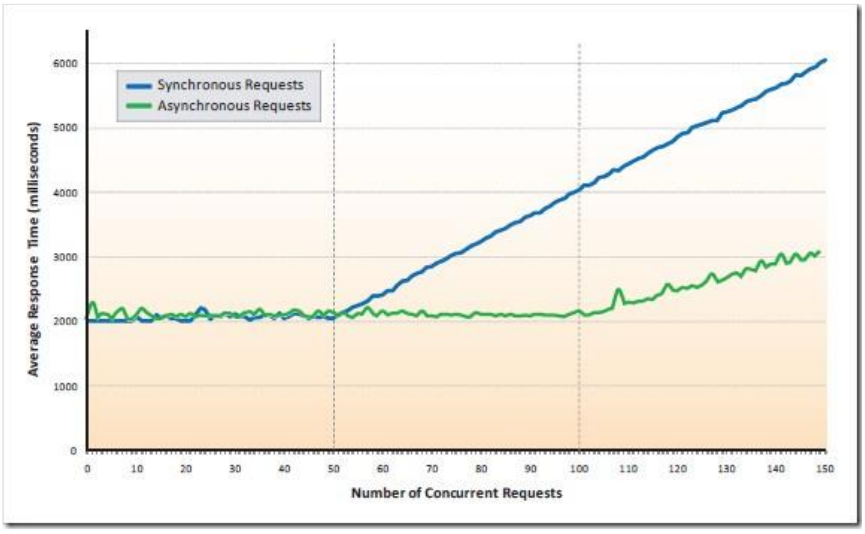
Слайд 7

Технології для розгортання додатку



Слайд 8

Використання асинхроних запитів



Слайд 9

Переваги кешування

- Кешування зменшує споживання пропускної здатності; отже, це зменшує мережевий трафік і зменшує перевантаження мережі.
- Кешування зменшує навантаження віддаленого веб -сервера, широко розподіляючи дані між кешами проксі через WAN.
- У сценарії, коли віддалений сервер недоступний через збій або розділення мережі, клієнт може отримати кешовану копію у проксі. Отже, підвищується надійність веб -служби.

Слайд 10

Дякую за увагу!