Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мультимедійних систем факультету інформатики

Розробка системи автоматизації бізнес-процесів з використанням Worker Services на платформі .NET

Текстова частина до курсової роботи

за спеціальністю “Інженерія програмного забезпечення” 121

Керівник курсової роботи

Старший викладач Борозенний С. О.

_____

(підпис)

“____” _____ 2025 р.

Виконав студент

Шандрик А.В

Київ – 20___

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мультимедійних систем факультету інформатики

ЗАТВЕРДЖУЮ

Зав. кафедри мультимедійних систем,

доц., канд. наук

Жежерун О.П.

(підпис)

“_____” _____ 2025 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту факультету інформатики 3-го курсу Шандрику Андрію В’ячеславовичу

ТЕМА «Розробка системи автоматизації бізнес-процесів з використанням Worker Services на платформі .NET»

Зміст ТЧ до курсової роботи:

Анотація

Вступ

1. Теоретичні основи бізнес-процесів та їх автоматизації

2. Технологія Worker Services на платформі .NET

3. Переваги використання Worker Services у порівнянні з іншими підходами.

4. Реалізація системи автоматизації з використанням Worker Services

5. Перспективи розвитку та масштабування системи

Загальні висновки

Дата видачі: “____ ” ____ 2025 р.

Керівник _____ (підпис)

Завдання отримав _____ (підпис)

КАЛЕНДАРНИЙ ПЛАН ВИКОНАННЯ КУРСОВОЇ РОБОТИ

Тема: Розробка системи автоматизації бізнес-процесів з використанням Worker Services на платформі .NET

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	09.10.2024	
2.	Огляд технічної літератури за темою роботи	25.12.2024	
3.	Розробка програмного застосунку	28.03.2025	
4.	Написання теоретичної частини роботи	18.04.2025	
5.	Створення презентації до теоретичної частини роботи	22.04.2025	
6.	Захист курсової роботи		

ЗМІСТ

Зміст

АНОТАЦІЯ

Роботу присвячено дослідженню та демонстрації можливостей технології Worker Services на платформі .NET для автоматизації різноманітних бізнес-процесів. Детально розглянуто теоретичні основи Worker Services та їх переваги у порівнянні з іншими підходами. В рамках роботи реалізовано чотири демонстраційні проекти, що ілюструють різні сценарії використання Worker Services: Scoped, Timer, Queue та EventBus.

Проект Scoped демонструє реалізацію Worker Services з використанням обмеженої області видимості для сервісів.

Проект Timer демонструє використання Worker Services для виконання періодичних задач за допомогою таймера.

Проект Queue показує обробку фонових задач з використанням черги. У цьому проекті демонструється реалізація системи черг на основі Channel API для передачі та обробки робочих елементів.

Проект EventBus ілюструє використання Worker Services для роботи з шиною повідомлень.

Розглянуто практичні аспекти конфігурації та інтеграції Worker Services з іншими компонентами системи. Реалізовані проекти показують можливості використання Worker Services для різних задач автоматизації.

Вступ

Традиційно, автоматизація фонових задач на платформі .NET базувалась на використанні Windows Services, що часто виявлялось складним у налаштуванні та розгортанні. Альтернативні ж підходи, такі як використання таймерів чи планувальників задач, мали обмежені можливості та не завжди задовольняли потреби складних застосунків.

З розвитком .NET платформи з'явилась потреба в більш зручному та ефективному інструменті для роботи з фоновими задачами. Worker Services були розроблені саме для вирішення цих проблем. Вони надають простий та гнучкий спосіб створення довготривалих фонових сервісів, які легко інтегруються з іншими компонентами системи та мають вбудовані механізми для управління життєвим циклом, обробки помилок та масштабування.

У зв'язку з цим, Worker Services стали популярним вибором для автоматизації різноманітних бізнес-процесів, таких як обробка даних, взаємодія з зовнішніми системами, періодичне виконання операцій тощо. Однак, для ефективного використання Worker Services необхідно розуміти їх особливості та можливості.

Дана робота присвячена дослідженню та демонстрації різних сценаріїв використання Worker Services на платформі .NET. Метою роботи є продемонструвати гнучкість та ефективність Worker Services для вирішення широкого спектру задач автоматизації, а також порівняти різні підходи до їх реалізації. В роботі розглянуто чотири типи Worker Services: Scoped, Timer, Queue та EventBus.

Дослідження поділяється на дві основні частини: теоретичний огляд та практичну реалізацію. В теоретичній частині будуть розглянуті основні принципи роботи кожного типу Worker Service, їх переваги та недоліки. В практичній частині будуть представлені демонстраційні проекти, що ілюструють різні сценарії використання та інтеграції Worker Services з іншими компонентами системи.

1. Теоретичні основи бізнес-процесів та їх автоматизації

1.1. Поняття та характеристики бізнес-процесів

Бізнес-процес — це сукупність взаємопов'язаних задач і заходів, спрямованих на досягнення певної мети або результату в рамках організації. Ці процеси є основою ефективного функціонування підприємства та можуть включати як виробничі, так і адміністративні функції.

Структурно бізнес-процес складається з наступних основних елементів:

- **Вхід**: ресурси або інформація, необхідні для виконання процесу
- **Вихід**: результат або продукт, створений процесом
- **Процес**: сукупність дій або кроків, що перетворюють вхід у вихід
- **Учасники**: люди або системи, які беруть участь у виконанні процесу

За своєю природою бізнес-процеси класифікуються на:

- **Основні процеси**: створюють основну цінність для клієнта (виробництво продукту, надання послуги)
- **Допоміжні процеси**: підтримують основні процеси (управління персоналом, бухгалтерія)
- **Управлінські процеси**: спрямовані на координацію та контроль інших процесів (стратегічне планування, управління якістю)

1.2. Автоматизація бізнес-процесів

Автоматизація бізнес-процесів передбачає використання інформаційних технологій для виконання рутинних завдань і підвищення ефективності процесів. Це інструмент, що дозволяє організаціям досягати таких покращень:

- Підвищення ефективності використання ресурсів і зменшення витрат
- Збільшення гнучкості та можливості швидкого реагування на зміни
- Забезпечення прозорості організації для керівництва і співробітників
- Стандартизація та покращення якості виконання процесів

Управління бізнес-процесами (BPM, Business Process Management) як методологія включає такі етапи:

1. Ідентифікація — визначення та документування поточних процесів

2. Аналіз — оцінка ефективності та виявлення проблемних областей
3. Моделювання — створення моделей поточних і майбутніх процесів
4. Впровадження — реалізація покращених процесів
5. Моніторинг — постійний контроль і оцінка ефективності

1.3. Роль фонових процесів у системах автоматизації

При автоматизації бізнес-процесів важливу роль відіграє моделювання, яке допомагає візуалізувати та аналізувати як поточні, так і майбутні процеси. Моделювання бізнес-процесів відбувається з використанням різних нотацій:

- BPMN (Business Process Model and Notation) — для відображення функціональної послідовності робіт
- IDEF0 — для моделювання логічної послідовності робіт
- UML (Unified Modeling Language) — для детального опису взаємодії компонентів системи

Аналіз бізнес-процесів передбачає створення моделей:

- Модель "як є" (as-is) — відображає поточний стан процесів
- Модель "як буде" (to-be) — визначає бажаний стан після оптимізації

Для автоматизації процесів використовуються різні технологічні рішення:

- Спеціалізоване програмне забезпечення (ERP, CRM системи)
- Фонові сервіси для обробки даних
- Інструменти інтеграції та оркестрації процесів

Правильна організація фонових процесів дозволяє забезпечити безперебійне виконання бізнес-операцій, мінімізувати втручання користувачів у рутинні операції та підвищити загальну ефективність автоматизованих систем.

2. Технологія Worker Services на платформі .NET

2.1 Призначення Worker Services

Worker Services — це технологія для створення довготривалих, фонових служб в .NET, які працюють без взаємодії з користувачем. Ця технологія виникла як крос-платформна альтернатива класичним Windows Services і дозволяє виконувати різноманітні завдання, що вимагають тривалого виконання.

Обробка CPU-інтенсивних даних — Worker Services ідеально підходять для виконання ресурсомістких обчислень, аналізу даних, генерації звітів та інших задач, які могли б блокувати основний потік програми в разі синхронного виконання. Такі завдання можуть включати обробку зображень, математичні обчислення, або аналіз великих обсягів інформації.

Обробка черг завдань — однією з найпоширеніших галузей застосування є обробка завдань з черги. Worker може постійно моніторити чергу повідомлень (наприклад, RabbitMQ, Azure Service Bus, Kafka) і виконувати необхідні дії при надходженні нових завдань. Це дозволяє відокремити прийом завдань від їх виконання і забезпечити масштабованість.

Виконання операцій за розкладом — Worker Services чудово підходять для виконання періодичних задач, таких як резервне копіювання даних, оновлення індексів пошуку, очищення тимчасових файлів, генерація періодичних звітів або відправка нагадувань. Замість використання зовнішніх планувальників, ці функції можна вбудувати безпосередньо в програму.

Довготривалі процеси — будь-які операції, що вимагають постійного моніторингу або безперервної роботи, можуть бути реалізовані як Worker Services. Це можуть бути агенти моніторингу, процеси синхронізації даних між системами, або служби, що підтримують постійне з'єднання з зовнішніми ресурсами.

2.2. Крос-платформні можливості

На відміну від традиційних Windows Services, Worker Services в .NET забезпечують повну крос-платформну сумісність, що значно розширює можливості їх застосування.

Незалежність від операційної системи — Worker Services можуть працювати на будь-якій платформі, де підтримується .NET, включаючи Windows, різні дистрибутиви Linux та macOS. Це дозволяє розробникам створювати уніфіковану кодову базу для фонових служб, незалежно від цільової ОС.

Еволюція від Windows Services — Worker Services можна розглядати як наступний еволюційний крок після класичних Windows Services. Вони зберігають усі ключові можливості традиційних служб, але розширюють їх крос-платформною підтримкою та сучасними патернами розробки. Існуючі Windows Services можна відносно легко мігрувати до Worker Services, використовуючи сумісні інтерфейси та адаптери.

Використання на різних апаратних платформах — завдяки крос-платформності .NET, Worker Services можуть виконуватися на різноманітному обладнанні: від потужних серверів для обробки великих даних до пристроїв IoT з обмеженими ресурсами. Ця універсальність робить їх ідеальними для створення рішень, що працюють у гетерогенних середовищах.

Контейнеризація — Worker Services відмінно інтегруються з Docker та іншими технологіями контейнеризації, що спрощує їх розгортання, масштабування та управління в сучасних мікросервісних архітектурах. Контейнеризовані Worker Services можна легко розгортати в кластерах Kubernetes або інших оркестраторах контейнерів.

2.3. .NET Generic Host у Worker Services

.NET Generic Host є фундаментом для Worker Services, забезпечуючи уніфікований підхід до організації ресурсів та керування життєвим циклом програми.

Управління ресурсами та життєвим циклом — Generic Host інкапсулює всі базові компоненти програми та керує їхнім життєвим циклом, забезпечуючи правильне створення, ініціалізацію та коректне завершення роботи. Це спрощує архітектуру додатку, оскільки відповідальність за управління ресурсами централізована.

Dependency Injection (DI) — Generic Host надає вбудовану систему впровадження залежностей, що спрощує управління залежностями та тестування компонентів. Розробникам не потрібно створювати власну реалізацію DI або інтегрувати зовнішні бібліотеки, оскільки все необхідне вже включено в інфраструктуру хоста.

Конфігурація — Host забезпечує гнучку систему конфігурації, яка підтримує різні джерела налаштувань: файли appsettings.json, змінні середовища, командний рядок та інші. Це дозволяє легко адаптувати поведінку Worker Service до різних середовищ без зміни коду.

Логування — інтегрована система логування надає уніфікований інтерфейс для запису подій та помилок, з можливістю налаштування різних провайдерів логування (файли, консоль, бази даних, сторонні сервіси). Це спрощує діагностику проблем та моніторинг роботи служби.

Контрольоване завершення роботи — одна з ключових переваг Generic Host — механізми graceful shutdown, які забезпечують коректне завершення операцій перед зупинкою служби. Хост обробляє системні сигнали (SIGTERM, CTRL+C) і дає службам час на завершення поточних операцій та звільнення ресурсів.

2.4. Архітектура IHostedService

IHostedService є основним інтерфейсом для реалізації фонових служб в .NET, забезпечуючи стандартизований підхід до інтеграції з Generic Host.

Інтерфейс IHostedService — цей інтерфейс визначає базовий контракт для всіх служб, які працюють у рамках хоста. Він простий і містить лише два методи для управління життєвим циклом служби. Будь-який клас, що реалізує цей інтерфейс, може бути зареєстрований як фонова служба в контейнері DI.

Методи StartAsync та StopAsync — ці методи викликаються хостом при запуску та зупинці програми відповідно. StartAsync ініціалізує ресурси та починає виконання фонових завдань, а StopAsync відповідає за коректне завершення операцій та звільнення ресурсів. Обидва методи асинхронні та отримують CancellationToken для коректної обробки скасування операцій.

Реєстрація служб у контейнері DI — служби, що реалізують IHostedService, реєструються в DI-контейнері за допомогою методу AddHostedService<>. Це дозволяє хосту автоматично створювати і керувати екземплярами цих служб, а також забезпечує їм доступ до інших зареєстрованих сервісів.

Множинні реалізації IHostedService — в одному додатку можна зареєструвати декілька різних реалізацій IHostedService, які будуть працювати паралельно. Це дозволяє розділити відповідальність між різними компонентами та реалізувати модульну архітектуру. Хост автоматично запускає всі зареєстровані служби під час старту програми.

2.5. BackgroundService як основа Worker Services

BackgroundService є абстрактним класом, який значно спрощує реалізацію фонових служб, забезпечуючи базову інфраструктуру для тривалого виконання асинхронних операцій.

Наслідування від абстрактного класу BackgroundService — цей клас реалізує IHostedService та надає шаблонну реалізацію його методів, що

дозволяє розробникам зосередитися лише на бізнес-логіці своєї служби. BackgroundService автоматично керує запуском та зупинкою внутрішніх завдань, спрощуючи розробку.

Метод ExecuteAsync — це основний метод, який повинні реалізувати дочірні класи. Він викликається після запуску служби і зазвичай містить нескінченний цикл або довготривалу операцію, що виконується до моменту скасування. Саме в цьому методі міститься основна логіка фонової служби.

Робота з CancellationToken — BackgroundService автоматично передає токен скасування в метод ExecuteAsync, що дозволяє реагувати на запити зупинки. Правильна обробка цього токена забезпечує коректне завершення роботи служби без втрати даних або залишення незавершених операцій.

2.6. Управління пам'яттю та продуктивність

Довготривалі фонові служби мають специфічні вимоги до управління пам'яттю та оптимізації продуктивності, які відрізняються від звичайних короткострокових програм.

ServerGarbageCollection — на відміну від стандартних налаштувань, довготривалі Worker Services часто виграють від використання серверного збирача сміття, який оптимізований для багатопотокових сценаріїв з високою продуктивністю. Він використовує окремі купи для різних процесорів, що зменшує конкуренцію за ресурси та покращує паралельну обробку.

Компроміси між продуктивністю та використанням ресурсів — при налаштуванні Worker Services важливо знайти баланс між максимальною продуктивністю та ефективним використанням ресурсів. Наприклад, серверний GC може покращити продуктивність, але збільшує споживання пам'яті, що важливо враховувати в обмежених середовищах.

Стабільність довготривалого виконання — особливу увагу слід приділяти запобіганню витокам пам'яті та іншим проблемам, які можуть проявлятися лише після тривалої роботи програми. Важливо коректно реалізувати звільнення ресурсів, використовувати слабкі посилання для кешів та уникати накопичення об'єктів у статичних колекціях.

Оптимізація реального часу — для сценаріїв, де важлива швидкість реакції, може знадобитися налаштування збирача сміття для мінімізації пауз.

Використання concurrent GC і коректне виділення об'єктів може значно зменшити латентність під час виконання критичних операцій.

2.7. Розгортання Worker Services

Worker Services підтримують різноманітні сценарії розгортання, від традиційних системних служб до сучасних контейнеризованих середовищ.

Windows Service — Worker Services можна зареєструвати як традиційні Windows Services за допомогою пакету Microsoft.Extensions.Hosting.WindowsServices. Це дозволяє інтегруватися з системою управління службами Windows, автоматично запускатися при старті системи та взаємодіяти з іншими системними компонентами.

Systemd у Linux — на Linux-системах Worker Services можна зареєструвати як служби systemd, використовуючи пакет Microsoft.Extensions.Hosting.Systemd. Це забезпечує аналогічні переваги: автозапуск, моніторинг процесів, управління залежностями між службами та логування через journald.

Запуск у контейнерах — Worker Services ідеально підходять для контейнеризації з Docker. Вони не вимагають зовнішніх залежностей від графічного інтерфейсу, можуть працювати в безголовому режимі та легко масштабуватися. Такі контейнери можуть бути розгорнуті в оркестраторах, таких як Aspire, Kubernetes, з використанням Deployments або StatefulSets.

Хмарні середовища — сучасні хмарні платформи надають спеціалізовані сервіси для запуску фонових завдань, які можна використовувати для хостингу Worker Services. Наприклад, Azure Web Jobs, Azure Container Instances, AWS ECS можуть бути використані для запуску та масштабування Worker Services без необхідності управління базовою інфраструктурою.

3. Переваги використання Worker Services у порівнянні з іншими підходами.

Worker Services пропонують унікальний набір переваг у порівнянні з іншими технологіями фонової обробки у екосистемі .NET:

У порівнянні з Background Services в ASP.NET: Хоча обидві технології використовують однаковий базовий інтерфейс IHostedService, Worker Services створюють окремий, незалежний процес, у той час як Background Services працюють у межах веб-додатку ASP.NET Core. Worker Services ідеальні для сценаріїв, де фонова обробка повинна бути повністю відокремлена від веб-інфраструктури, уникаючи проблем із перезапуском веб-застосунку та конкуренцією за ресурси. Це забезпечує кращу ізоляцію, незалежне масштабування та вищу надійність для критичних фонових операцій.

У порівнянні з Quartz.NET в ASP.NET: Quartz.NET часто інтегрується як частина ASP.NET додатку через проміжне ПЗ (middleware). Worker Services, на відміну від цього підходу, уникають зайвих залежностей від веб-стеку, що робить їх більш легковагими та менш вразливими до проблем пов'язаних з веб-середовищем. Це особливо важливо для довготривалих завдань, які не повинні перериватися через перезапуски веб-сервера чи зміну конфігурації веб-застосунку.

У порівнянні з Hangfire в ASP.NET: Хоча Hangfire надає зручну інтеграцію з ASP.NET через панель моніторингу, Worker Services забезпечують чистіший архітектурний розподіл між веб-функціональністю та фоновими процесами. Цей розподіл запобігає сценаріям, де проблеми продуктивності фонових завдань впливають на відгук веб-API, або де інтенсивний веб-трафік сповільнює виконання критичних фонових операцій.

В контексті мікросервісної архітектури: Worker Services представляють собою окремі мікросервіси, що відповідає принципу "одна відповідальність — один сервіс", на відміну від Background Services, Quartz.NET чи Hangfire в ASP.NET, які об'єднують веб- та фонову функціональність в одному сервісі. Це забезпечує чіткіший розподіл відповідальностей, спрощує моніторинг, налагодження та незалежне масштабування конкретних функціональних блоків.

З погляду операційної підтримки: Відокремлення Worker Services від ASP.NET додатків спрощує діагностику проблем, оскільки логи та метрики фонових процесів не змішуються з веб-запитами. Також це дозволяє застосовувати різні політики розгортання, масштабування та відновлення для веб- та фонових компонентів, що підвищує загальну надійність системи.

В екосистемі .NET Aspire: Worker Services мають потужну інтеграцію з .NET Aspire, що дозволяє їм органічно вписуватися в архітектуру хмарно-нативних розподілених додатків. На відміну від інших підходів, Worker Services в .NET Aspire отримують вбудовану підтримку оркестрації контейнерів, автоматичну телеметрію через OpenTelemetry, централізоване логування та моніторинг через Aspire Dashboard. Aspire також забезпечує прозору конфігурацію сервісного відкриття та комунікації між сервісами, що особливо цінно для Worker Services, які потребують взаємодії з іншими компонентами системи. Ця інтеграція створює унікальну перевагу перед іншими технологіями фонової обробки, які вимагають додаткових зусиль для досягнення подібного рівня спостережуваності та керованості в розподілених середовищах.

4. Реалізація системи автоматизації з використанням Worker Services

У цьому розділі представлено практичну реалізацію системи автоматизації, побудованої на основі різних типів Worker Services платформи .NET. Спираючись на теоретичні положення, розглянуті в попередніх розділах, було розроблено чотири варіанти Worker Services, що демонструють різні методи автоматизації процесів: Scoped, Timer, Queue та EventBus. Кожен підхід має свої особливості застосування та дозволяє ефективно вирішувати специфічні завдання автоматизації.

4.1 Timer-Based Worker Service

Timer-Based Worker Service представляє підхід до автоматизації завдань за розкладом з використанням системного таймера. Основна ідея полягає у виконанні певного завдання через регулярні проміжки часу без необхідності зовнішнього втручання.

Реалізація TimerService базується на інтерфейсі IHostedService, що дозволяє інтегрувати сервіс у загальний життєвий цикл додатка. У методі StartAsync створюється екземпляр System.Threading.Timer, налаштований на виклик методу DoWork кожні 5 секунд. Це забезпечує регулярне виконання завдань за визначеним розкладом.

```
public Task StartAsync(CancellationToken stoppingToken)
{
    _logger.LogInformation("{Service} is running.", nameof(TimerService));
    _timer = new System.Threading.Timer(DoWork, state: null, TimeSpan.Zero, period: TimeSpan.FromSeconds(5));

    return _completedTask;
}
```

Рисунок 1 Код ініціалізації таймера в Timer-Based Worker Service (метод StartAsync)

Для забезпечення потокобезпечності при роботі з лічильником виконань використовується атомарна операція Interlocked.Increment. Це дозволяє уникнути проблем, пов'язаних з конкурентним доступом з різних потоків

```
private void DoWork(object? state)
{
    var count = Interlocked.Increment(ref _executionCount);

    _logger.LogInformation(
        "{Service} is working, execution count: {Count:#,0}",
        nameof(TimerService),
        count);
}
```

Рисунок 2 Приклад коду для потокобезпечного лічильника в Timer-Based Worker Service

При зупинці сервісу в методі StopAsync таймер переводиться в режим нескінченного очікування, що ефективно зупиняє його роботу:

```
public Task StopAsync(CancellationToken stoppingToken)
{
    _logger.LogInformation(
        "{Service} is stopping.", nameof(TimerService));

    _timer?.Change(Timeout.Infinite, 0);

    return _completedTask;
}
```

Рисунок 3Метод зупинки таймера у Timer-Based Worker Service

Для коректного звільнення ресурсів використовується реалізація інтерфейсу IAsyncDisposable:

```
public async ValueTask DisposeAsync()
{
    if (_timer is IAsyncDisposable timer) await timer.DisposeAsync();

    _timer = null;
}
```

Рисунок 4Реалізація інтерфейсу IAsyncDisposable для звільнення ресурсів

Хоча демонстраційна реалізація використовує просту операцію інкременту лічильника, цей підхід може бути легко адаптований для виконання будь-яких періодичних завдань, таких як:

- Регулярна перевірка стану системи або моніторинг ресурсів
- Оновлення кешу даних або збір статистики

- Створення резервних копій або архівування даних
- Очищення тимчасових файлів або застарілих записів у базі даних
- Надсилання сповіщень або звітів за розкладом

Timer-Based Worker Service особливо ефективний для завдань, які повинні виконуватися за чітким розкладом незалежно від інших подій у системі.

4.2 Scoped-Based Worker Service

Scoped-Based Worker Service демонструє підхід, при якому для кожного циклу виконання завдання створюється окрема область видимості (scope). Це дозволяє ізолювати ресурси між різними виконаннями та ефективно працювати з залежностями, що потребують певного життєвого циклу.

На відміну від Timer-Based підходу, Scoped-Based сервіс використовує BackgroundService як базовий клас, що забезпечує більш високорівневу абстракцію для тривалих фонових завдань. У методі ExecuteAsync створюється цикл, який періодично створює нову область видимості:

```
protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    _logger.LogInformation("{Service} is running.", nameof(ScopedService));

    while (!stoppingToken.IsCancellationRequested)
    {
        // Створення нової області видимості для кожної ітерації
        using (var scope = _serviceScopeFactory.CreateScope())
        {
            // Отримання сервісів з поточної області видимості
            var scopedService = scope.ServiceProvider.GetRequiredService<IScopedWorker>();

            // Виконання роботи у поточній області видимості
            await scopedService.DoWorkAsync(stoppingToken);
        }

        await Task.Delay(TimeSpan.FromSeconds(5), stoppingToken);
    }
}
```

Рисунок 5 Цикл обробки завдань у Scoped-Based Worker Service

Цей підхід особливо корисний для завдань, що працюють з:

- Контекстами баз даних, які потребують коректного відстеження змін
- Транзакціями, що повинні бути ізольовані між різними виконаннями
- Сервісами з обмеженим часом життя, які вимагають правильної ініціалізації та утилізації
- Ресурсами, що повинні бути звільнені після завершення кожного циклу

Використання областей видимості також дозволяє уникнути потенційних витоків пам'яті та ресурсів у довготривалих процесах, забезпечуючи їх правильне звільнення після кожного циклу роботи.

4.3 Queue-Based Worker Service

Queue-Based Worker Service демонструє підхід до обробки завдань через чергу. Замість виконання завдань за розкладом, цей тип сервісу реагує на появу нових елементів у черзі та обробляє їх асинхронно у порядку надходження.

Реалізація базується на трьох ключових компонентах:

- Інтерфейс черги завдань (IBackgroundTaskQueue)
- Сервіс, що обробляє завдання з черги (QueuedHostedService)
- Компонент, що додає завдання до черги (MonitorLoop)

4.3.1 Інтерфейс та реалізація черги завдань

```
public sealed class DefaultBackgroundTaskQueue : IBackgroundTaskQueue
{
    private readonly Channel<Func<CancellationToken, ValueTask>> _queue;

    public DefaultBackgroundTaskQueue(int capacity)
    {
        BoundedChannelOptions options = new(capacity)
        {
            FullMode = BoundedChannelFullMode.Wait
        };
        _queue = Channel.CreateBounded<Func<CancellationToken, ValueTask>>(options);
    }

    public async ValueTask QueueBackgroundWorkItemAsync(
        Func<CancellationToken, ValueTask> workItem)
    {
        ArgumentNullException.ThrowIfNull(workItem);

        await _queue.Writer.WriteAsync(workItem);
    }

    public async ValueTask<Func<CancellationToken, ValueTask>> DequeueAsync(
        CancellationToken cancellationToken)
    {
        Func<CancellationToken, ValueTask>? workItem =
            await _queue.Reader.ReadAsync(cancellationToken);

        return workItem;
    }
}
```

Рисунок 6 Інтерфейс IBackgroundTaskQueue та його реалізація

У цій реалізації використовується обмежений канал (BoundedChannel), що дозволяє контролювати максимальну кількість завдань у черзі. Режим BoundedChannelFullMode.Wait забезпечує блокування операції додавання нового завдання, якщо черга заповнена, до появи вільного місця.

Черга зберігає делегати Func<CancellationToken, ValueTask>, що дозволяє передавати будь-які асинхронні операції з підтримкою скасування для виконання

4.3.2 Обробник черги завдань

Основна логіка обробки реалізована в класі QueuedHostedService, який успадковує від BackgroundService:

```
public sealed class QueuedHostedService(
    IBackgroundTaskQueue taskQueue,
    ILogger<QueuedHostedService> logger) : BackgroundService
{
    protected override Task ExecuteAsync(CancellationToken stoppingToken)
    {
        logger.LogInformation("""
                              {Name} is running.
                              Tap W to add a work item to the
                              background queue.
                              """,
            nameof(QueuedHostedService));

        return ProcessTaskQueueAsync(stoppingToken);
    }

    private async Task ProcessTaskQueueAsync(CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            try
            {
                Func<CancellationToken, ValueTask>? workItem =
                    await taskQueue.DequeueAsync(stoppingToken);

                await workItem(stoppingToken);
            }
            catch (OperationCanceledException)
            {
                // Prevent throwing if stoppingToken was signaled
            }
            catch (Exception ex)
            {
                logger.LogError(ex,  message: "Error occurred executing task work item.");
            }
        }
    }

    public override async Task StopAsync(CancellationToken stoppingToken)
    {
        logger.LogInformation(
            $"{nameof(QueuedHostedService)} is stopping.");

        await base.StopAsync(stoppingToken);
    }
}
```

Рисунок 7 Логіка обробки черги завдань у QueuedHostedService

Метод ProcessTaskQueueAsync постійно очікує на нові завдання в черзі. Коли завдання з'являється, він отримує його за допомогою DequeueAsync та виконує, передаючи токен скасування. Важливо відзначити, що:

1. Обробка виконується у захищеному блоці try-catch, що забезпечує ізоляцію помилок між різними завданнями
2. Спеціальна обробка OperationCanceledException запобігає небажаним помилкам при зупинці сервісу
3. Всі інші винятки логуються, але не призводять до зупинки обробника

4.3.3 Додавання завдань до черги

Клас MonitorLoop демонструє додавання завдань до черги:

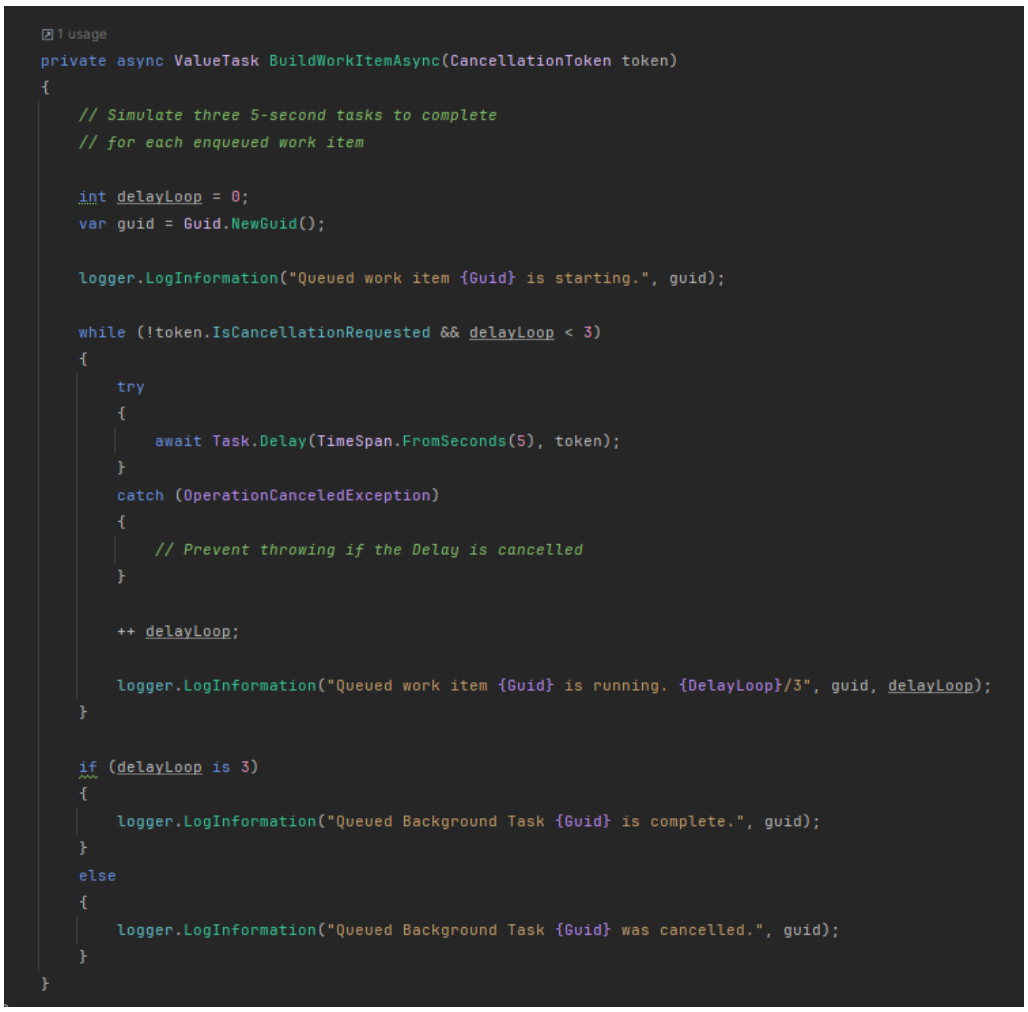
```
public sealed class MonitorLoop(
    IBackgroundTaskQueue taskQueue,
    ILogger<MonitorLoop> logger,
    IHostApplicationLifetime applicationLifetime)
{
    private readonly CancellationToken _cancellationToken = applicationLifetime.ApplicationStopping;

    1 usage
    public void StartMonitorLoop()
    {
        logger.LogInformation($"{nameof(MonitorAsync)} loop is starting.");

        // Run a console user input loop in a background thread
        Task.Run(async () => await MonitorAsync());
    }

    2 usages
    private async ValueTask MonitorAsync()
    {
        while (!_cancellationToken.IsCancellationRequested)
        {
            var keyStroke = Console.ReadKey();
            if (keyStroke.Key == ConsoleKey.W)
            {
                // Enqueue a background work item
                await taskQueue.QueueBackgroundWorkItemAsync(BuildWorkItemAsync);
            }
        }
    }
```

Рисунок 8 Код додавання завдань до черги через MonitorLoop



```
1 usage
private async ValueTask BuildWorkItemAsync(CancellationToken token)
{
    // Simulate three 5-second tasks to complete
    // for each enqueued work item

    int delayLoop = 0;
    var guid = Guid.NewGuid();

    logger.LogInformation("Queued work item {Guid} is starting.", guid);

    while (!token.IsCancellationRequested && delayLoop < 3)
    {
        try
        {
            await Task.Delay(TimeSpan.FromSeconds(5), token);
        }
        catch (OperationCanceledException)
        {
            // Prevent throwing if the Delay is cancelled
        }

        ++ delayLoop;

        logger.LogInformation("Queued work item {Guid} is running. {DelayLoop}/3", guid, delayLoop);
    }

    if (delayLoop is 3)
    {
        logger.LogInformation("Queued Background Task {Guid} is complete.", guid);
    }
    else
    {
        logger.LogInformation("Queued Background Task {Guid} was cancelled.", guid);
    }
}
```

Рисунок 9 Приклад реалізації завдання для черги

У цьому демонстраційному прикладі завдання додаються до черги при натисканні клавіші "W". У реальних системах завдання можуть додаватися різними способами:

- Як реакція на HTTP-запити
- При отриманні повідомлень із зовнішніх джерел (наприклад, RabbitMQ або Kafka)
- При виникненні подій у системі
- За розкладом через Timer-Based Worker Service

Кожне завдання представлене методом BuildWorkItemAsync, який симулює тривалий процес з трьох етапів по 5 секунд кожен. Важливо, що метод правильно обробляє сигнали скасування, що дозволяє коректно зупинити виконання при відключенні сервісу.

4.4 EventBus-Based Worker Service

EventBus-Based Worker Service представляє собою тип фонового сервісу, який базується на архітектурі повідомлень (message-driven architecture) та шаблоні проектування Publish-Subscribe. Це більш складний, але й більш гнучкий підхід до обробки асинхронних завдань порівняно з Queue-Based сервісами.

4.4.1 Концепція Event Bus

Event Bus (Шина подій) — це архітектурний патерн, що забезпечує комунікацію між різними компонентами системи без прямої залежності між ними. Він працює за принципом:

1. **Publishers (Видавці)** — компоненти, що генерують події
2. **Subscribers (Підписники)** — компоненти, що реагують на події
3. **EventBus (Шина подій)** — посередник, який забезпечує доставку повідомлень від видавців до підписників

Цей підхід особливо ефективний для:

- Мікросервісної архітектури
- Систем з розподіленими компонентами
- Систем з високими вимогами до масштабування
- Систем, де потрібна слабка зв'язаність (loose coupling) між компонентами

4.4.2 Реалізація на базі MassTransit і RabbitMQ

У представленому прикладі використовується MassTransit — бібліотека, що спрощує реалізацію шаблонів обміну повідомленнями в .NET, та RabbitMQ як транспортний рівень для передачі повідомлень.

```
public class OrderCreatedNotification : IConsumer<OrderCreated>
{
    public async Task Consume(ConsumeContext<OrderCreated> context)
    {
        await Task.Delay(1000);
        Console.WriteLine(context.ReceiveContext.InputAddress);
        Console.WriteLine($"Admin Notification -I just consumed a message with OrderId {context.Message.OrderId}, " +
                          $"that was created at:{context.Message.CreatedAt}");
    }
}
```

Рисунок 10 Реалізація споживача подій (IConsumer) у EventBus-Based Worker Service

Споживач реалізує інтерфейс IConsumer<T>, де T — тип повідомлення, яке він обробляє. У цьому випадку OrderCreated — це структура даних, що містить інформацію про створене замовлення. Метод Consume викликається автоматично при надходженні відповідного повідомлення, і в ньому виконується логіка обробки.

4.4.2.1 Налаштування Event Bus

```
public static IHostBuilder CreateHostBuilder(string[] args)
{
    return Host.CreateDefaultBuilder(args)
        .ConfigureServices((hostContext, services) =>
        {
            services.AddMassTransit(x =>
            {
                x.SetKebabCaseEndpointNameFormatter();

                var entryAssembly = Assembly.GetEntryAssembly();
                x.AddConsumers(entryAssembly);

                x.UsingRabbitMq((context, cfg) =>
                {
                    cfg.ReceiveEndpoint("order-created",
                        e => { e.ConfigureConsumer<OrderCreatedNotification>(context); });

                    cfg.ConfigureEndpoints(context);
                });
            });
        });
}
```

Рисунок 11 Конфігурація Event Bus з використанням MassTransit і RabbitMQ

У цьому блоці коду:

1. services.AddMassTransit() додає до сервісів DI-контейнеру необхідні компоненти MassTransit
2. x.SetKebabCaseEndpointNameFormatter() налаштовує формат іменування точок обміну повідомленнями (наприклад, OrderCreated → order-created)
3. x.AddConsumers(entryAssembly) автоматично реєструє всі класи-споживачі з поточної збірки
4. x.UsingRabbitMq() конфігурує RabbitMQ як транспорт для передачі повідомлень
5. cfg.ReceiveEndpoint("order-created", ...) явно налаштовує точку входу для повідомлень про створені замовлення та прив'язує до неї споживача OrderCreatedNotification

4.4.3 Архітектурні особливості Event Bus

EventBus-Based Worker Service відрізняється від Queue-Based принципово іншим підходом до організації обробки завдань:

Характеристика	Queue-Based	EventBus-Based
Зв'язаність компонентів	Пряма залежність	Слабка зв'язаність
Маршрутизація	Один отримувач	Багато отримувачів
Масштабування	В межах однієї системи	Розподілене
Стійкість до відмов	Обмежена	Висока
Типовий транспорт	In-memory, Entity Framework	RabbitMQ, Kafka, Azure Service Bus

EventBus-Based Worker Service ідеально підходить для:

1. **Інтеграції мікросервісів** — коли потрібно забезпечити обмін даними між незалежними сервісами
2. **Реалізації довгострокових бізнес-процесів** — коли процес складається з декількох етапів, які виконуються різними сервісами
3. **Розподілених систем** — коли компоненти системи працюють на різних серверах або в різних контейнерах
4. **Систем з високими вимогами до стійкості** — коли необхідно гарантувати обробку повідомлень навіть при тимчасових збоях

5. Перспективи розвитку та масштабування системи

Розроблена система автоматизації бізнес-процесів на базі Worker Services має значний потенціал для подальшого вдосконалення, масштабування та адаптації до нових вимог. Нижче наведено ключові напрями розвитку, які дозволять забезпечити стійкість, гнучкість та конкурентоспроможність системи в умовах динамічного середовища.

5.1. Горизонтальне масштабування за допомогою оркестраторів

Для обробки зростаючого навантаження система може бути масштабована шляхом додавання нових екземплярів Worker Services. Використання оркестраторів, таких як **Aspire, Kubernetes** або **Azure Container Apps**, дозволить автоматизувати розподіл завдань, балансування навантаження та відновлення після збоїв. Наприклад, для Queue-Based Worker Services можна динамічно збільшувати кількість споживачів черги залежно від обсягу повідомлень.

5.2. Інтеграція з хмарними сервісами

Використання хмарних платформ (**Azure**, **AWS**, **Google Cloud**) відкриває можливості для:

- **Глобального розгортання**: розміщення Worker Services у різних географічних регіонах для зменшення затримок.
- **Сервісів керованої інфраструктури**: заміна власних черг на Azure Service Bus або AWS SQS, що спрощує обробку повідомлень і знижує операційні витрати.
- **Serverless-архітектури**: комбінація Worker Services з Azure Functions для обробки пікових навантажень.

5.3. Покращення спостережуваності

Для ефективного моніторингу та аналізу роботи системи можна впровадити:

- **Інтеграцію з OpenTelemetry**: збір метрик, трасів та логів для аналізу продуктивності.
- **Дашборди (Grafana, Azure Monitor)**: візуалізація ключових показників (час обробки завдань, кількість помилок, використання ресурсів).
- **Автоматичні сповіщення**: налаштування алертів у разі критичних збоїв або перевищення лімітів.

5.4. Розширення функціоналу

- **Підтримка нових шаблонів обробки**: додавання Workflow-Based Worker Services для керування складними бізнес-процесами з використанням DSL-мови (наприклад, через Temporal.io).
- **Інтеграція з AI/ML**: автоматизація аналізу даних за допомогою машинного навчання (наприклад, класифікація завдань, прогнозування навантаження).
- **Підтримка IoT**: обробка подій від пристроїв IoT через EventBus, що дозволить інтегрувати систему з промисловими сценаріями.

5.5. Оптимізація продуктивності

- **Кешування даних**: використання Redis або Memcached для зменшення навантаження на бази даних.
- **Паралельна обробка**: впровадження шаблону Fan-Out/Fan-In для розподілу завдань між кількома Worker Services.
- **Профілювання пам’яті**: застосування інструментів на кшталт dotMemory для виявлення витоків та оптимізації алокації об’єктів.

5.6. Підвищення рівня безпеки

- **Шифрування повідомлень**: інтеграція з Azure Key Vault або AWS KMS для керування криптографічними ключами.
- **RBAC-механізми**: обмеження доступу до критичних операцій через ролі та політики.
- **Захист від DDoS**: використання хмарних сервісів (наприклад, Azure DDoS Protection) для фільтрації трафіку.

5.7. Адаптація до нових версій .NET

Оновлення до .NET 8+ дозволить використовувати:

- **Вдосконалені GC-налаштування**: для роботи з великими даними в режимі реального часу.
- **Нативну AOT-компіляцію**: зменшення часу запуску та споживання ресурсів.
- **Нові API для асинхронності**: покращена продуктивність при обробці тисячі одночасних завдань.

5.8. Мікросервісна архітектура

Розділення системи на спеціалізовані мікросервіси (наприклад, окремі служби для обробки платежів, генерації звітів, нотифікацій) дозволить:

- Незалежно масштабувати компоненти.
- Використовувати різні технології для різних задач (наприклад, F# для аналітичних модулів).
- Зменшити ризики впливу збоїв однієї частини системи на інші

5.9 Висновок

Запропоновані напрями розвитку забезпечують довгострокову життєздатність системи, дозволяючи їй адаптуватися до змін бізнес-вимог і технологічних трендів. Використання сучасних інструментів оркестрації, хмарних сервісів та модульного підходу робить систему гнучкою, масштабованою та готовою до інтеграції з інноваційними рішеннями.

Загальні висновки

У межах курсової роботи було проведено всебічне дослідження можливостей автоматизації бізнес-процесів за допомогою технології Worker Services на платформі .NET. Робота охоплює як теоретичний аналіз архітектурних особливостей та переваг Worker Services, так і практичну реалізацію, що дозволяє оцінити їх ефективність у реальних сценаріях застосування.

- У практичній частині проєкту було розроблено та протестовано чотири окремі сервіси, які реалізують різні підходи до фонової обробки задач:
- Timer-Based Worker Service — для періодичного виконання завдань за розкладом,
- Scoped-Based Worker Service — для ізольованої обробки з дотриманням життєвого циклу залежностей,
- Queue-Based Worker Service — для асинхронної обробки задач з черги,
- EventBus-Based Worker Service — для роботи в рамках архітектури подій із застосуванням RabbitMQ та MassTransit.

Кожен із цих підходів було детально досліджено з технічної точки зору, включаючи механізми запуску, обробки, зупинки та масштабування, а також практично реалізовано у вигляді відповідного робочого сервісу. Це дозволило не лише перевірити функціональність, а й виявити особливості та доцільність застосування кожного з них для різних типів задач.

Результати курсової роботи підтверджують, що Worker Services у .NET є потужним інструментом для створення ефективних, стійких і масштабованих рішень в області автоматизації. Розроблені сервіси демонструють практичну користь таких рішень для реального бізнесу — зниження витрат на ручну обробку, мінімізація людських помилок, прискорення реагування на події, а також можливість інтеграції з хмарними платформами та мікросервісною інфраструктурою.

Таким чином, використання Worker Services у системах автоматизації бізнес-процесів можна вважати не лише технічно обґрунтованим, а й економічно доцільним кроком на шляху до цифрової трансформації підприємств. Подальші дослідження можуть бути зосереджені на інтеграції з системами штучного інтелекту, оптимізації продуктивності у високонавантажених середовищах та впровадженні інструментів самовідновлення й самоналаштування для підвищення рівня автономності сервісів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Microsoft learn. Worker services in .NET. URL : https://learn.microsoft.com/en-us/dotnet/core/extensions/workers
2. .NET Aspire documentation. URL: https://learn.microsoft.com/uk-ua/dotnet/aspire/
3. MassTransit Documentation. URL: https://masstransit.io/
4. Вікіпедія. Бізнес-процес. URL: https://uk.wikipedia.org/wiki/Бізнес-процес