

Детектор Плагиата v. 2808 - Отчёт оригинальности: 19.01.2025 15:48:26

Проанализированный документ: Мiзинець Розробка корпоративної системи обліку проєктів з використанням Windows Forms та мікросервісної архітектури.docx
Лицензия: ВОЛОДИМИР МАТІЄВСЬКИЙ

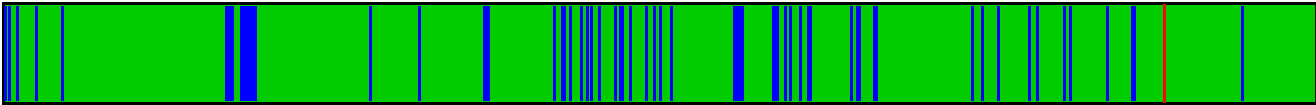
Тип поиска: Поиск переписанного Язык: Uk
 Тип проверки: Интернет
ТЭЕ и кодировка: DocX n/a

Детальный анализ тела документа:

Диаграмма соотношения частей:



Граф распределения зон:



Источники плагиата: 8

	→ 0,8% 75	1. https://hackyourmom.com/privatnist/vstanovlennya-virtualbox-na-windows-10/
	→ 0,2% 20	2. https://worksection.com/ua/blog/10-best-pm-software-for-small-business-2024.html
	→ 0,1% 10	3. https://buhgalter.com.ua/dovidnik/inshe/kodi-dk-0212015-pridbannya-za-yakimi-pogodzhuyemo-pro-yaki/

Детали обработанных ресурсов: 153 - ОК / 5 - Ошибок

Важные замечания:

Википедия:	Google Книги:	Сервисы платных работ:	Античит:
[не обнаружено]	[не обнаружено]	[не обнаружено]	Обнаружено сокрытие!

Античит-отчет UACE:

1. Статус: Анализатор Включен Нормализатор Включен сходство символов установлено на 100%
2. Обнаруженный процент загрязнения UniCode: 22,5% с лимитом: 4%
3. Процент нераспознанных символов после нормализации: 12,4%
4. Все подозрительные символы будут отмечены фиолетовым цветом: Abcd...
5. Найдены невидимые символы: 0
Рекомендации по оценке: Особое внимание следует уделить анализу этого отчета! Предполагается, что этот документ содержит

значительное количество символов, чуждых языку документа. Это прямое указание на то, что автор документа использовал специальное программное обеспечение\онлайн-веб-сервис, чтобы эффективно скрыть текст в попытке избежать обнаружения потенциального плагиата. Настоятельно рекомендуется передать это дело на более высокий уровень! В случае сомнений обращайтесь: в службу поддержки Детектора плагиата!

Алфавитная статистика и анализ символов:

 Активные ссылки (URL-адреса, извлеченные из документа):

URL не найдены

 Исключённые ресурсы:

URL не найдены

 Включённые ресурсы:

URL не найдены

Детальний аналіз документа:

Міністерство освіти і науки України Державний заклад

Цитування: 0,05%

id: 1

«Луганський національний університет імені Тараса Шевченка»

Навчально-науковий інститут математики та інформаційних технологій Кафедра інформаційних технологій та систем Мізинець Олександр Сергійович Розробка корпоративної системи обліку проєктів з використанням [Windows Forms](#) та мікросервісної архітектури кваліфікаційна робота здобувача вищої освіти другого (магістерського) рівня освітньої програми

Цитування: 0,03%

id: 2

«Інженерія програмного забезпечення»

за спеціальністю 121 Інженерія програмного забезпечення Особистий підпис _____ Олександр Мізинець Науковий керівник _____ Микола СЕМЕНОВ, кандидат педагогічних наук, доцент кафедри інформаційних технологій та систем Завідувача кафедри _____ Микола СЕМЕНОВ, кандидат педагогічних наук, доцент кафедри інформаційних технологій та систем Полтава – 2025 АНОТАЦІЯ Тема: Розробка корпоративної системи обліку проєктів з використанням [Windows Forms](#) та мікросервісної архітектури Спеціальність: 121

Цитування: 0,03%

id: 3

«Інженерія програмного забезпечення».

Установа: ЛНУ імені Тараса Шевченка, 2025 р. Магістерська робота містить: 77 с., 5 рис., 6 табл., 50 джерел. Об'єкт дослідження – процеси автоматизації обліку проєктів у корпоративному середовищі. Предмет дослідження – технології та методи розробки корпоративних систем обліку проєктів із використанням [Windows Forms](#) та мікросервісної архітектури.. Мета роботи – розробка корпоративної системи обліку проєктів з використанням [Windows Forms](#) для створення інтерфейсу користувача та мікросервісної архітектури для обробки даних. Результати роботи – у роботі проведено аналіз сучасних підходів до створення корпоративних систем обліку проєктів. Розроблено архітектуру системи, яка включає мікросервісну серверну частину та клієнтський інтерфейс на [Windows Forms](#). Реалізовано мікросервіси для управління проєктами, задачами, звітами та сповіщеннями. Здійснено інтеграцію з базами даних [PostgreSQL](#) через [REST API](#) та проведено тестування системи, яке підтвердило її працездатність і ефективність.. Ключові слова: МІКРОСЕРВІСИ, [WINDOWS FORMS](#), УПРАВЛІННЯ ПРОЄКТАМИ, [API GATEWAY](#), [POSTGRESOL](#), КОРПОРАТИВНІ СИСТЕМИ. [ANNOTATION Topic: Development of a corporate project management system using Windows Forms and microservice architecture. Speciality: 121](#)

Цитування: 0,02%

id: 4

"[Software Engineering](#)".

. [Institution: Luhansk Taras Shevchenko National University \(LTSNU\)](#), 2025 year. [Master's thesis consists of: 77 p., 5 im., 6 tables, 50 sources. Object of research – processes of automating project management in a corporate environment. Subject of research – technologies and methods for developing corporate project management systems using Windows Forms and microservice architecture. Objective of the study is to develop a corporate project management system utilizing Windows Forms for creating the user interface and microservice architecture for data processing.. Results of the study – the thesis analyzed modern approaches to the creation of corporate project management systems. A system architecture was developed, including a microservice-based backend and a Windows Forms client interface. Microservices were implemented for managing projects, tasks, reports, and notifications. Integration with PostgreSQL databases via REST API was performed, and the system was tested, confirming its functionality and efficiency. Keywords: MICROSERVICES, WINDOWS FORMS, PROJECT MANAGEMENT, API GATEWAY, POSTGRESOL, CORPORATE SYSTEMS.](#) ЗМІСТ

ВСТУП6 РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ, СУЧАСНІ ТЕХНОЛОГІЇ ДЛЯ РОЗРОБКИ

КОРПОРАТИВНОЇ СИСТЕМИ9 1.1.Опис сучасних технологій для створення корпоративної системи та обґрунтування вибору [Windows Forms](#)9 1.2.Спільне використання [ASP.NET Core](#) та [Windosw Forms](#). Порівняльний аналіз різних інструментів для побудови мікросервісної архітектури.14 1.3.Функціональні вимоги до корпоративної системи обліку проєктів22

РОЗДІЛ 2. Модель корпоративної системи обліку проєктів24 2.1. Архітектура мікросервісів корпоративної системи обліку проєктів24 2.2. Модель бази даних корпоративної системи обліку проєктів30 2.3. Логіка взаємодії між [Windows Forms](#) та мікросервісами32 РОЗДІЛ 3. РЕАЛІЗАЦІЯ КОРПОРАТИВНОЇ СИСТЕМИ ОБЛІКУ ПРОЄКТІВ34 3.1. Розробка графічного інтерфейсу [Windows Forms](#)34 3.2. Створення мікросервісів39 3.3. Тестування взаємодії мікросервісів48 РОЗДІЛ 4. РОЗРОБКА ВЛАСНОГО СТАРТАП ПРОЄКТУ

” Цитування: 0,03%

id: 5

«ОБЛІК ПРОЄКТІВ [UA](#)»

62 4.1. Опис ідеї стартап проєкту та його обґрунтування62 4.2. [SWOT](#)-аналіз стартап-проєкту64 ВИСНОВКИ67 СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ70 Додаток А.71 ВСТУП

Корпоративні системи обліку проєктів є важливими інструментами для ефективного управління організаціями та компаніями в умовах швидко змінюваного ринку. З розвитком технологій з'являється потреба в нових підходах до обробки даних, інтеграції різних підсистем і забезпеченні зручного доступу до необхідної інформації. Водночас, актуальність проблеми впровадження таких систем зростає через потребу в гнучких, масштабованих рішеннях, здатних підтримувати розвиток компаній у різних галузях. У зв'язку з цим виникає необхідність створення корпоративних систем обліку проєктів, що враховують сучасні технології, зокрема мікросервісну архітектуру, яка забезпечує ефективну взаємодію між різними компонентами системи. Застосування [Windows Forms](#) для створення інтерфейсів користувача та мікросервісної архітектури для обробки даних є популярним підходом для розробки таких рішень. Однак, незважаючи на значні досягнення в цій сфері, залишається багато викликів, пов'язаних з інтеграцією різних технологій і забезпеченням надійності таких систем. Найбільше уваги приділено розробці мікросервісних архітектур для великих підприємств, але мало уваги зосереджено на розробці простих та зручних рішень для малих та середніх бізнесів, що потребують не менш ефективних, але більш доступних рішень. Предмет та об'єкт дослідження: Об'єкт дослідження: процеси автоматизації обліку проєктів у корпоративному середовищі. Предмет дослідження: технології та методи розробки корпоративних систем обліку проєктів із використанням [Windows Forms](#) та мікросервісної архітектури. Метою цієї роботи є розробка корпоративної системи обліку проєктів з використанням [Windows Forms](#) для створення інтерфейсу користувача та мікросервісної архітектури для обробки даних. Відповідно до мети сформульовані завдання: Провести аналіз сучасних підходів до створення корпоративних систем обліку проєктів, зокрема особливостей використання [Windows Forms](#) та мікросервісної архітектури. Розробити архітектуру системи, включаючи моделі взаємодії між компонентами, структуру бази даних та взаємодію між клієнтською частиною та мікросервісами. Реалізувати корпоративну систему обліку проєктів, використовуючи [Windows Forms](#) для створення інтерфейсу користувача та мікросервісну архітектуру для обробки даних. Забезпечити інтеграцію розробленої системи з базою даних та розробити [REST API](#) для взаємодії між компонентами. Провести тестування системи, включаючи функціональне, інтеграційне та навантажувальне тестування, а також оцінити її продуктивність та зручність використання. Узагальнено завданням є побудова системи, яка дозволить автоматизувати процеси обліку проєктів, моніторингу їх стану, управління командами та ресурсами, а також генерації звітів. Проблеми, що будуть вирішені, включають створення зручного інтерфейсу для користувачів, забезпечення швидкої обробки даних, інтеграцію мікросервісів і ефективне використання бази даних. Перший розділ роботи присвячений огляду сучасних технологій для розробки корпоративних систем, зокрема, розглядаються переваги та недоліки використання мікросервісної архітектури та [Windows Forms](#) у створенні програмного забезпечення для управління проєктами. У другому розділі аналізуються існуючі рішення та підходи до автоматизації обліку проєктів, а також визначаються основні вимоги до розробленої системи. Третій розділ містить детальне проєктування системи, включаючи архітектуру та модель бази даних. Четвертий розділ присвячений реалізації системи, включаючи розробку мікросервісів та графічного інтерфейсу. П'ятий розділ висвітлює процес тестування, що включає юніт-тестування та інтеграційне тестування системи. У шостому розділі підводяться підсумки роботи, аналізуються досягнуті результати, а також надаються рекомендації щодо подальшого розвитку системи. Таким чином, дана робота є спробою створити гнучку та ефективну корпоративну систему обліку проєктів, яка відповідає вимогам сучасних технологій та потребам користувачів. РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ, СУЧАСНІ ТЕХНОЛОГІЇ ДЛЯ РОЗРОБКИ КОРПОРАТИВНОЇ СИСТЕМИ Опис сучасних

технологій для створення корпоративної системи та обґрунтування вибору [Windows Forms](#). У сучасних умовах стрімкого розвитку цифрових технологій корпоративні системи відіграють ключову роль у забезпеченні ефективного управління бізнес-процесами, автоматизації рутинних задач та покращенні взаємодії між різними підрозділами організацій. Вибір відповідних технологій є важливим кроком у процесі розробки, оскільки вони визначають продуктивність, масштабованість та зручність використання майбутньої системи. На сьогодні існує широкий спектр підходів та платформ для створення корпоративних систем, включаючи хмарні технології, мікросервісну архітектуру, засоби контейнеризації та інтеграцію з базами даних. Кожна з цих технологій має свої переваги та недоліки, що потребують ретельного аналізу перед початком розробки. З огляду на поставлене завдання — створення корпоративної системи обліку проєктів, яка базується на [Windows Forms](#) і мікросервісній архітектурі, цей підхід поєднує перевірені часом інструменти розробки десктопних застосунків із сучасними трендами в архітектурі програмного забезпечення. [Windows Forms](#) є оптимальним вибором для створення зручного графічного інтерфейсу користувача, тоді як використання мікросервісів дозволяє забезпечити модульність і масштабованість системи [20]. Відповідно до мети та завдань магістерської роботи проаналізуємо різні підходи до створення корпоративної системи для обґрунтування вибору рішення та обраної теми. Ключовою на сьогодні є технологія мікросервісів, яка дозволяє розділити систему на незалежні компоненти, кожен із яких відповідає за конкретну функціональність. Мікросервісна архітектура ([microservices architecture](#)) — це підхід до розробки програмного забезпечення, який передбачає поділ системи на невеликі незалежні компоненти, які виконують чітко визначені функції. Кожен мікросервіс є автономним модулем, що може бути розроблений, розгорнутий і масштабований окремо від інших. Цей підхід став популярним у середині 2010-х років, хоча його концептуальні основи з'явилися раніше у вигляді сервіс-орієнтованої архітектури ([SOA](#)). Перші ідеї, які лягли в основу мікросервісної архітектури, з'явилися у 2000-х роках разом із розвитком [SOA](#). Проте саме мікросервіси стали відповіддю на виклики, які виникли під час роботи з великими монолітними системами. Основними ініціаторами переходу до мікросервісів стали великі технологічні компанії, які зіткнулися з труднощами масштабування та підтримки великих систем. Основною метою мікросервісної архітектури є підвищення гнучкості, продуктивності та масштабованості програмного забезпечення. Монолітні системи, хоча й зручні на початкових етапах, з часом стають складними в обслуговуванні, адже будь-які зміни в коді потребують глибокого аналізу та можуть впливати на всю систему. У мікросервісній архітектурі кожен компонент є незалежним, що дозволяє уникати подібних проблем [2; 36; 37]. Теоретично мікросервісна архітектура базується на принципах модульності та слабого зв'язку між компонентами. Вона передбачає дотримання кількох ключових принципів: [Single Responsibility Principle](#) (Принцип єдиної відповідальності): Кожен мікросервіс виконує одну чітко визначену функцію; слабкий зв'язок: мікросервіси взаємодіють через стандартизовані інтерфейси, зазвичай через [API](#) (наприклад, [REST](#) або [gRPC](#)), що мінімізує залежності між ними; індивідуальне масштабування: кожен мікросервіс може бути масштабований окремо залежно від навантаження, яке він обробляє; автономність: мікросервіси мають власну логіку, а часто й окремі бази даних, що дозволяє уникнути конфліктів даних [35; 46]. [Amazon](#) відіграв важливу роль у популяризації мікросервісів. Ще на початку 2000-х компанія почала переходити від монолітної архітектури до модульної. Це дозволило [Amazon](#) значно покращити роботу свого інтернет-магазину та забезпечити безперебійну роботу навіть за умов пікових навантажень []. Мікросервісна архітектура спирається на кілька ключових парадигм, які відрізняють її від традиційного моноліту: децентралізація: рішення приймаються на рівні кожного окремого сервісу. Наприклад, кожен сервіс може використовувати різні мови програмування або бази даних, якщо це оптимально для його функціональності; безперервна інтеграція та доставка ([CI/CD](#)): мікросервіси розробляються так, щоб їх можна було швидко оновлювати без переривання роботи всієї системи; масштабованість: мікросервіси дозволяють масштабувати лише ті частини системи, які відчувають навантаження, що значно знижує витрати; гнучкість у розробці: Різні команди можуть одночасно працювати над різними мікросервісами, що пришвидшує розробку. На рисунку 1.1 показано набір мікросервісів, які надають високорівневі [API](#), орієнтовані на потреби клієнтів, використовуючи при цьому низькорівневі детальні [API](#), доступні в системі. Наприклад, різні продукти для веб- і мобільних клієнтів ([Experience](#)) можуть застосовувати різні [API](#), орієнтовані на досвід користувачів, що реалізовані незалежними мікросервісами за допомогою [API](#) шлюзу ([API gateway](#)). Ці мікросервіси утворюють так

званий [experience](#) рівень, який логічно відокремлений від інших рівнів системи. Це забезпечує можливість його автономного управління та налаштування (масштабування, конфігурації, захисту тощо). На рисунку 1.2 ілюструється принцип, згідно з яким кожна база даних повинна належати лише одному мікросервісу. Інші сервіси не можуть отримувати прямий доступ до цієї бази даних — тільки через [API](#) відповідального за неї мікросервісу. Такий підхід дозволяє кожному сервісу самостійно керувати консистентністю та структурою своєї бази даних, а також обирати найбільш оптимальний тип бази для виконання своїх завдань, незалежно від потреб усієї системи. Наприклад, один сервіс може використовувати нормалізовану [SQL](#)-базу, тоді як інший може працювати з [NoSQL](#) для досягнення кращої продуктивності. Водночас один мікросервіс може керувати кількома базами даних. Рис. 1.1 Реалізація мікросервісної архітектури для кросс-платформи [20]

Таким чином, мікросервісна архітектура є потужним інструментом для створення складних корпоративних систем. Вона виникла як відповідь на виклики масштабування та підтримки монолітних систем і сьогодні широко використовується провідними компаніями світу. Завдяки своїм теоретичним основам, таким як слабкий зв'язок, автономність і децентралізація, мікросервіси забезпечують високу продуктивність, гнучкість і масштабованість сучасного програмного забезпечення. Плюсами є масштабованість, гнучкість у розробці та підтримці, стійкість до збоїв. Рис. 1.2 Реалізація мікросервісної архітектури для СУБД [20]

Спільне використання [ASP.NET Core](#) та [Windows Forms](#). Порівняльний аналіз різних інструментів для побудови мікросервісної архітектури. [ASP.NET Core](#) — це сучасна, кросплатформна та високопродуктивна платформа для створення веб-застосунків і сервісів, включаючи мікросервіси. Завдяки своїй модульній структурі та інтеграції з широким спектром інструментів, [ASP.NET Core](#) ідеально підходить для створення [RESTful API](#), які часто використовуються як основа мікросервісної архітектури. Платформа дозволяє швидко створювати контролери, що обробляють [HTTP](#)-запити, і надає вбудовані засоби для серіалізації даних у форматі [JSON](#) [6; 22; 23;30;44]. На листингу 1.1 приведено приклад, створення простого мікросервісу в [ASP.NET Core](#) для управління списком завдань ([ToDo](#)): Листинг 1.1 [[ApiController](#)] [[Route](#)(

” Цитування: 0,01% id: 6

"api/[controller]"

```
)] public class ToDoController : ControllerBase { private static List string tasks = new List string (); [HttpGet] public IActionResult GetTasks() { return Ok(tasks); } [HttpPost] public IActionResult AddTask([FromBody] string task) { tasks.Add(task); return Ok(
```

” Цитування: 0,03% id: 7

"Task added successfully."

```
); } [HttpDelete(
```

” Цитування: 0,01% id: 8

"{index}"

```
)] public IActionResult DeleteTask(int index) { if (index 0 || index = tasks.Count) return BadRequest(
```

” Цитування: 0,03% id: 9

"Invalid task index."

```
); tasks.RemoveAt(index); return Ok(
```

” Цитування: 0,03% id: 10

"Task deleted successfully."

); } } Цей контролер обробляє запити на отримання всіх завдань, додавання нового завдання та видалення існуючого за індексом. [Windows Forms](#) можна використовувати для створення клієнтських застосунків, які взаємодіють із [REST API](#) [19], реалізованим через мікросервіси. Для цього зручно використовувати клас [HttpClient](#) із бібліотеки [System.Net.Http](#). У листингу 1.2 наведено код [Windows Forms](#) застосунку, який викликає методи [REST API](#) для управління списком завдань: Листинг 1.2 [private readonly HttpClient](#) `httpClient = new HttpClient { BaseAddress = new Uri(`

” Цитування: 0,01% id: 11

"https://localhost:5001/api/ToDo"

```
); } private async void btnGetTasks_Click(object sender, EventArgs e) { var response = await
httpClient.GetAsync(
```

Цитування: 0%

id: 12

""

```
); if (response.IsSuccessStatusCode) { var tasks = await response.Content.ReadAsAsync List
string (); listBoxTasks.DataSource = tasks; } else { MessageBox.Show(
```

Цитування: 0,03%

id: 13

"Failed to load tasks."

```
); } } private async void btnAddTask_Click(object sender, EventArgs e) { var task = txtTask.Text;
var response = await httpClient.PostAsJsonAsync(
```

Цитування: 0%

id: 14

""

```
task); if (response.IsSuccessStatusCode) { MessageBox.Show(
```

Цитування: 0,03%

id: 15

"Task added successfully."

```
); btnGetTasks_Click(sender, e); // Refresh the list } else { MessageBox.Show(
```

Цитування: 0,03%

id: 16

"Failed to add task."

```
); } } private async void btnDeleteTask_Click(object sender, EventArgs e) { if
(listBoxTasks.SelectedIndex 0) return; var response = await httpClient.DeleteAsync($
```

Цитування: 0,02%

id: 17

"/{listBoxTasks.SelectedIndex}"

```
); if (response.IsSuccessStatusCode) { MessageBox.Show(
```

Цитування: 0,03%

id: 18

"Task deleted successfully."

```
); btnGetTasks_Click(sender, e); // Refresh the list } else { MessageBox.Show(
```

Цитування: 0,03%

id: 19

"Failed to delete task."

); } } Цей приклад демонструє використання кнопок для отримання, додавання та видалення завдань через [REST API](#). Метод [HttpClient.GetAsync](#) використовується для отримання даних, [PostAsJsonAsync](#) для надсилання нового завдання, а [DeleteAsync](#) — для видалення завдання. Такий підхід дозволяє створити зручний клієнтський інтерфейс для взаємодії з мікросервісами, що забезпечує розділення обов'язків між серверною та клієнтською частинами системи. Таким чином, показано, що [Windows Forms](#) може бути заставано для реалізації мікросервісної архітектури на основі [ASP.NET Core](#). Окрім [ASP.NET](#), [Spring Boot](#) і [Node.js](#) є популярними платформами для розробки мікросервісів. Кожна з них має свої особливості, переваги та обмеження, які роблять їх більш або менш придатними для певних завдань. [Spring Boot](#) — це фреймворк для розробки застосунків на [Java](#), що спрощує створення мікросервісів завдяки готовим конфігураціям і можливості швидкого налаштування. [Spring Boot](#) забезпечує високий рівень інтеграції з іншими компонентами екосистеми [Spring](#) ([Spring Data](#), [Spring Security](#), [Spring Cloud](#)). Це дозволяє легко створювати масштабовані, безпечні та модульні системи. Однією з ключових переваг [Spring Boot](#) є його стабільність і широка підтримка інструментів для корпоративних рішень. Проте, використання [Java](#) може бути складним через відносно високу складність синтаксису й обсяги коду [12-15; 32]. [Node.js](#) — це платформа на основі [JavaScript](#), яка працює на рушії [V8](#) від [Google](#). Вона ідеально підходить для розробки високонавантажених і масштабованих систем завдяки асинхронній моделі введення/виведення. [Node.js](#) має велику екосистему бібліотек і пакетів ([npm](#)), що дозволяє швидко реалізовувати функціональність. Крім того, вона добре підходить для реальних застосунків із великою кількістю одночасних підключень (наприклад, чатів або стрімінгових платформ). Однак, обмеженням [Node.js](#) є обробка обчислювальних важких задач через однопоточну природу [18; 28; 34]. У таблиці 1.1 наведено порівняння різних інструментів побудови мікросервісів за обраними критеріями: мова, продуктивність, масштабованість, інструменти, поріг входження,

підтримка спільноти, сфера використання, кросплатформенність, документація, обробка обчислень. Таблиця 1.1 Порівняльна таблиця різних інструментів для мікросервісів

Критерій	ASP.NET Core	Spring Boot (Java)	Node.js	Мова програмування	C#	Java	JavaScript
Продуктивність	Висока	Висока	Висока	для асинхронних задач	Масштабованість	Добра	Відмінна
Відмінна	Відмінна	Інструменти	Інтеграція з Visual Studio	Spring	екосистема	npm	Поріг входження
Середній	Високий	Низький	Підтримка спільноти	Активна	Дуже активна	Дуже активна	Сфера використання
Корпоративні системи	Корпоративні системи	Реальні часи, API, IoT	Кросплатформенність	Так	Так	Так	Документація
Чітка	Дуже детальна	Добра	Обробка обчислень	Висока	Висока	Слабка	для CPU-задач

На основі проведеного аналізу (табл. 1.1) ми обрали [ASP.NET Core](#) для створення мікросервісної архітектури з кількох причин. По-перше, досвід роботи з [C#](#) та [Windows Forms](#), отриманий під час навчання в університеті, значно спрощує розробку. По-друге, [ASP.NET Core](#) добре підходить для корпоративних систем завдяки високій продуктивності, зрозумілій документації та зручній інтеграції з інструментами [Microsoft](#) ([Visual Studio](#), [Azure](#)). Крім того, обмеження [Node.js](#) для обчислювально важких задач і більша складність [Spring Boot \(Java\)](#) у використанні стали додатковими факторами для вибору на користь [ASP.NET Core](#). Універсальність цієї платформи дозволяє легко реалізувати потреби проєкту без суттєвих перепон. Інші інструменти для створення корпоративних систем представимо таблицею 1.2

Інструменти для створення корпоративних систем	Категорія	Опис	Приклади інструментів
Контейнеризація та оркестрація	Ізольоване середовище для виконання додатків, що спрощує розгортання та підтримку.	Docker (стандарт контейнеризації), Kubernetes (автоматичне розгортання, масштабування та управління).	Системи управління базами даних (СУБД)
Інструменти для роботи з великими обсягами даних.	Розподіляються на реляційні та NoSQL.	Реляційні: Microsoft SQL Server (висока продуктивність), PostgreSQL (відкритий і розширюваний). NoSQL: MongoDB (гнучкі структури даних), Redis (швидкий доступ до кешу).	Технології фронтенду
Інструменти для створення зручного користувацького інтерфейсу корпоративних систем.	React (інтерактивні веб-інтерфейси), Angular (повний набір інструментів від Google), Blazor (C#-орієнтовані веб-інтерфейси).	Хмарні платформи	
Інфраструктура, що забезпечує масштабованість та гнучкість.	Microsoft Azure (розробка, хостинг і аналітика), AWS (широкий набір хмарних інструментів), Google Cloud (інтеграція з ML).	CI/CD та автоматизація розробки	
Процеси, які забезпечують постійне оновлення та підтримку корпоративних систем.	GitHub Actions , GitLab CI , Jenkins (автоматизація розгортання).	Безпека	
Інструменти для захисту даних і запобігання кіберзагрозам.	OAuth 2.0 , OpenID Connect (аутентифікація), Azure Key Vault , HashiCorp Vault (зберігання конфіденційних даних).	Звичайно, на основі інформації з таблиці 1.2 нас цікавить як інструменти можуть бути застосовані в Windows Forms .	
Контейнеризація (Docker) використовується для розгортання бекенд-сервісів, з якими взаємодіє Windows Forms .	Docker хостить мікросервіси (наприклад, API для обліку проєктів). Windows Forms підключається до контейнеризованих сервісів через локальну або хмарну мережу.	Наприклад, сервер бази даних чи API працює у Docker , але користувач працює з Windows Forms .	
Windows Forms безпосередньо або через API підключається до баз даних для зберігання та обробки даних. Пряма інтеграція через ADO.NET або Entity Framework .	Використання SQL Server , PostgreSQL або інших баз даних для зберігання інформації про проєкти, користувачів і звіти. В якості альтернативи Windows Forms може надсилати запити через API до бекенду, який взаємодіє з базою даних.	Windows Forms може взаємодіяти з сучасними веб-фронтенд-додатками або навіть містити веб-компоненти.	
Інтеграція веб-контенту через WebBrowser Control або WebView2 (на основі Microsoft Edge).	Наприклад, у додатку Windows Forms можна відображати інтерактивний звіт, створений за допомогою React або Angular .	Windows Forms використовується для клієнтського доступу до хмарних сервісів, таких як Microsoft Azure або AWS .	
Клієнт підключається до сервісів через API або SDK .	Зберігання файлів у хмарі через Azure Blob Storage або AWS S3 .	Використання сервісів аутентифікації (Azure AD , OAuth).	
Дані можуть передаватися між Windows Forms і хмарними функціями для обробки, аналітики чи зберігання. Зв'язок: CI/CD забезпечує автоматизацію збірки, тестування і розгортання Windows Forms -додатків.	GitHub Actions або Jenkins використовуються для автоматичної збірки та тестування.	Результат — виконуваний файл (.exe) або інсталятор, який легко розгортається на клієнтських машинах.	
Аутентифікація через OAuth 2.0 або JWT -токени для доступу до мікросервісів.	Використання шифрування (SSL/TLS) для захисту даних, переданих між клієнтом і сервером.	Сумісність: Windows Forms може легко працювати як з локальними, так і з мережевими/хмарними рішеннями. У підсумку, Windows Forms може бути	

центральною частиною системи, яка доповнюється сучасними інструментами, що підвищують її продуктивність, масштабованість і зручність. Функціональні вимоги до корпоративної системи обліку проєктів Корпоративні інформаційні системи є ключовим інструментом для організації бізнес-процесів та підвищення ефективності управління. Вони охоплюють широкий спектр рішень, від [ERP \(Enterprise Resource Planning\)](#), які забезпечують інтеграцію всіх основних бізнес-процесів компанії, до [CRM \(Customer Relationship Management\)](#), що дозволяють управляти взаємодією з клієнтами. Одним із напрямів таких систем є корпоративні системи обліку проєктів, які орієнтовані на планування, виконання та моніторинг проєктної діяльності. Прикладами таких систем є [Microsoft Project](#), [Trello](#), [Jira](#), [Basecamp](#), а також модулі управління проєктами в популярних [ERP](#)-системах, таких як [SAP](#) або [Oracle](#). У корпоративній системі обліку проєктів критично важливим є врахування ключових даних, що забезпечують повноцінний моніторинг та управління. Система повинна зберігати інформацію про назву кожного проєкту, його поточний статус (наприклад,

” Цитування: 0,02%	id: 20
"у розробці",	
” Цитування: 0,03%	id: 21
"на етапі тестування",	
” Цитування: 0,01%	id: 22
"завершено"	

), строки виконання, перелік відповідальних виконавців, цілі проєкту, бюджет, ризики та залежності від інших проєктів. Також важливо мати можливість вести історію змін статусів та записувати ключові події, що відбулися під час реалізації проєкту. Основною метою розробки системи є автоматизація низки бізнес-процесів. Серед них — процеси планування проєктів, включаючи розподіл ресурсів і встановлення термінів; управління завданнями, що охоплює створення, призначення виконавців і контроль виконання; а також відстеження прогресу, що включає аналіз виконання плану щодо встановлених показників. Крім того, автоматизація дозволяє оптимізувати звітність, знижуючи трудовитрати на її створення та забезпечуючи актуальність даних у реальному часі. Інтеграція з іншими модулями (наприклад, бухгалтерськими чи кадровими) дозволить забезпечити ширше охоплення управлінських процесів. Таким чином, функціональні вимоги до системи включають можливість зручного введення й оновлення даних, створення динамічних звітів, відображення взаємозв'язків між проєктами, а також налаштування інтерфейсу відповідно до потреб користувачів. Така система має стати важливим інструментом підтримки прийняття рішень, забезпечуючи прозорість процесів і зменшуючи ризики в управлінні проєктами. РОЗДІЛ 2. Модель корпоративної системи обліку проєктів 2.1. Архітектура мікросервісів корпоративної системи обліку проєктів Відповідно до результатів 1-го розділу розробимо архітектуру корпоративної системи обліку проєктів. Монолітний підхід і мікросервісна архітектура є двома основними парадигмами в розробці програмного забезпечення. Обидва підходи мають свої переваги та обмеження, які визначають їхню придатність для різних сценаріїв використання. Монолітна архітектура передбачає, що всі компоненти програми розробляються, розгортаються та підтримуються як єдине ціле. Це спрощує початкову розробку, але з часом може ускладнити підтримку та масштабування через взаємозалежності між компонентами. Мікросервісна архітектура, навпаки, передбачає розділення системи на незалежні сервіси, кожен з яких відповідає за окрему функціональність. Це підвищує гнучкість і дозволяє масштабувати лише ті частини системи, які відчувають навантаження, але водночас збільшує складність розгортання та управління [20]. У табл. 2.1 наведено порівняння монолітної та мікросервісної архітектури. Таблиця 2.1 Порівняння монолітної та мікросервісної архітектур Критерій Монолітна архітектура Мікросервісна архітектура Структура Єдина програма, що об'єднує всі функціональні компоненти. Набір незалежних сервісів, кожен із яких виконує одну чітку функцію. Масштабованість Масштабується як єдине ціле, що може бути неефективним. Кожен сервіс масштабується окремо залежно від потреб. Розгортання Розгортання всієї системи, навіть за малих змін. Незалежне розгортання окремих сервісів. Залежності Сильні зв'язки між компонентами, складно вносити зміни. Слабкі зв'язки між сервісами, легше вносити зміни. Підтримка та оновлення Зміни в одній частині можуть вплинути на всю систему. Зміни в одному сервісі не зачіпають інші. Складність розробки Легка на

початкових етапах. Вимагає ретельного планування взаємодії сервісів. Продуктивність Висока продуктивність через відсутність мережових запитів. Залежить від якості комунікації між сервісами. Інтеграція Всі компоненти інтегровані в одному середовищі. Використовуються стандартизовані [API](#) для комунікації. Ризики Високий ризик

Цитування: 0,02%

id: 23

«ефекту доміно»

при збоях. Проблеми в одному сервісі зазвичай не впливають на інші. Модульність Низька, компоненти важко ізолювати. Висока, кожен сервіс автономний. Час розробки Швидше на початкових етапах. Може потребувати більше часу через планування сервісів. Монолітна архітектура є кращою для невеликих і середніх проєктів, де простота розробки та підтримки важливіша за масштабованість. Водночас, мікросервісна архітектура більше підходить для великих корпоративних систем, які вимагають гнучкості, незалежного масштабування і здатності швидко реагувати на зміни бізнес-вимог. Для корпоративної системи обліку проєктів обрано мікросервісну архітектуру. Це рішення зумовлене необхідністю масштабування окремих компонентів системи (наприклад, сервісу задач або звітності), спрощенням підтримки та можливістю незалежного розвитку окремих сервісів. Тому побудуємо архітектуру системи, основні компоненти архітектури: клієнтська частина ([Windows Forms](#)), мікросервіси та бази даних. Клієнтська частина - інтерфейс користувача, через який користувачі взаємодіють із системою. Використовує [REST API](#) для комунікації з мікросервісами. Включає функції для управління проєктами, відстеження статусів, звітів тощо. Опишемо основні мікросервіси: Сервіс управління проєктами ([Project Management Service](#)): відповідає за створення, редагування та видалення проєктів; зберігає інформацію про проєкти в базі даних. Сервіс користувачів ([User Service](#)): керує реєстрацією, аутентифікацією та авторизацією користувачів; зберігає дані користувачів та їх ролі в системі. Сервіс задач ([Task Service](#)): відповідає за управління задачами в рамках проєктів; дозволяє створення, редагування та моніторинг задач, пов'язаних із проєктами. Сервіс звітності ([Reporting Service](#)): генерує звіти на основі даних про проєкти та задачі; використовує дані з інших мікросервісів для створення аналітики. Сервіс сповіщень ([Notification Service](#)): відповідає за надсилання сповіщень користувачам (наприклад, про зміни статусу задач); може використовувати різні канали комунікації, такі як електронна пошта або [push](#)-сповіщення. Мікросервіси мають власні бази даних, що відповідають їхньому функціоналу (наприклад, реляційні бази даних для сервісу управління проєктами та задачами, [NoSQL](#) для сервісу звітності). БД (бази даних) Забезпечують зберігання даних та їхню доступність для мікросервісів. [API Gateway](#) виконує роль єдиного входу для всіх запитів до мікросервісів. Обробляє маршрутизацію запитів, а також може реалізовувати функції аутентифікації та авторизації. Система моніторингу та логування забезпечує моніторинг стану мікросервісів та їх продуктивності. Збирає логи для аналізу та виявлення можливих проблем. Взаємодія клієнтської частини з мікросервісами: Клієнтська частина надсилає [HTTP](#)-запити до [API Gateway](#), який маршрутизує запити до відповідних мікросервісів. Мікросервіси можуть спілкуватися між собою через [REST API](#) або за допомогою системи обміну повідомленнями ([RabbitMQ](#)) для асинхронної обробки. Кожен мікросервіс працює з власною базою даних, що знижує вплив змін у одній частині системи на інші частини. Загальна схема архітектури системи мікросервісів корпоративної системи обліку проєктів представлено на рис. 2.1 Архітектура мікросервісів корпоративної системи обліку проєктів забезпечує модульність, гнучкість і можливість масштабування. Вона дозволяє легко впроваджувати нові функціональні можливості, оптимізувати певні мікросервіси та підтримувати високу продуктивність системи в цілому. Архітектура корпоративної системи обліку проєктів базується на мікросервісному підході, де кожен мікросервіс виконує специфічні функції, такі як управління проєктами, користувачами, задачами, звітністю та сповіщеннями. Взаємодія між компонентами здійснюється через [API Gateway](#), який є єдиним входом для клієнтської частини, що реалізована у [Windows Forms](#). Клієнтська частина ([Windows Forms](#)) [API Gateway](#) Сервіс управління проєктами Сервіс користувачів Сервіс задач Сервіс звітності Сервіс сповіщень БД проєктів БД користувачів БД задач БД звітності БД сповіщень Рис. 2.1 Загальна схема архітектури мікросервісів корпоративної системи обліку проєктів Запити від користувачів маршрутизуються до відповідних мікросервісів, які обробляють дані та взаємодіють з власними базами даних, забезпечуючи зберігання та обробку інформації. Це дозволяє системі бути гнучкою, масштабованою та легкою у підтримці. 2.2. Модель бази даних корпоративної системи обліку проєктів Згідно з

розробленою у п. 2.1 архітектурою системи (дивись рис 2.1), що розробляється, наступним етапом дослідження є конструювання моделі бази даних корпоративної системи обліку проєктів. На рис. 2.2 представлено цю модель та у вигляді [ER](#) діаграми. Рис. 2.2 Модель бази даних корпоративної системи обліку проєктів На рис 2.2 описано таблиці: 1.

” Цитування: **0,01%** id: **24**

“[Users](#)”

ідентифікує користувачів системи. 2.

” Цитування: **0,01%** id: **25**

“[Projects](#)”

- зберігає інформацію про проєкти. 3.

” Цитування: **0,01%** id: **26**

“[Tasks](#)”

- включає дані про задачі, пов'язані з проєктами. 4.

” Цитування: **0,01%** id: **27**

“[Reports](#)”

- зберігає звіти, створені на основі проєктів та задач. 5.

” Цитування: **0,01%** id: **28**

“[Notifications](#)”-

зберігає сповіщення, які надсилаються користувачам. Код для створення таблиць наведено у додатку А. Нижче наведемо опис таблиць та зв'язків між ними. 1. Таблиця [`users`](#) Таблиця зберігає інформацію про користувачів, включаючи унікальне ім'я користувача, хеш пароля, електронну пошту та дати створення і останнього оновлення запису. 2. Таблиця [`projects`](#) Таблиця для зберігання інформації про проєкти, включаючи назву, опис, ідентифікатор користувача, який створив проєкт, а також дати створення і останнього оновлення. 3. Таблиця [`tasks`](#) Таблиця, що містить інформацію про задачі, пов'язані з проєктами. Включає назву, опис, статус задачі, ідентифікатор користувача, якому призначена задача, а також дати створення і останнього оновлення. 4. Таблиця [`reports`](#) Таблиця для зберігання звітів, пов'язаних з проєктами. Містить ідентифікатор проєкту, користувача, який створив звіт, дату звіту, текстовий зміст звіту, а також дати створення і останнього оновлення. 5. Таблиця [`notifications`](#) Таблиця для зберігання сповіщень, які надсилаються користувачам. Включає ідентифікатор користувача, текст сповіщення, статус прочитання та дату створення сповіщення. Загальні відношення між таблицями такі: - кожен користувач може створити багато проєктів і задач; - кожен проєкт може містити багато задач та звітів; - кожен користувач може отримувати багато сповіщень. Ця структура бази даних забезпечує ефективне зберігання та управління даними для корпоративної системи обліку проєктів. 2.3. Логіка взаємодії між [Windows Forms](#) та мікросервісами Логіка взаємодії між клієнтською частиною на основі [Windows Forms](#) та мікросервісами в корпоративній системі обліку проєктів ґрунтується на принципах модульності, розподіленості та стандартизації. [Windows Forms](#) виступає клієнтським інтерфейсом, через який користувачі виконують основні операції: управління проєктами, задачами, звітами та сповіщеннями. Для забезпечення зв'язку між клієнтською частиною та мікросервісами використовується [REST API](#), що забезпечує стандартизований формат передачі даних. Клієнтські запити відправляються через [API Gateway](#), який виконує маршрутизацію та обробку запитів до відповідних мікросервісів. Наприклад, для створення нового проєкту клієнтська частина надсилає [HTTP](#)-запит до [API Gateway](#), який переспрямовує його до сервісу управління проєктами. Цей сервіс зберігає дані про проєкт у відповідній базі даних і повертає підтвердження про успішну операцію. Аналогічно побудована взаємодія з іншими мікросервісами, такими як сервіс задач, сервіс звітності та сервіс сповіщень. Зроблено розподіл даних між мікросервісами. Наприклад, дані користувачів зберігаються та обробляються лише в сервісі користувачів, тоді як інформація про проєкти зберігається у сервісі управління проєктами. Це забезпечує чіткий розподіл відповідальності між мікросервісами та зменшує ризик помилок через взаємозалежності. Таке рішення дозволяє кожному сервісу використовувати найбільш відповідну модель даних, наприклад, реляційні бази даних для управління проєктами та задачами чи [NoSQL](#) для звітів і сповіщень. Додатково опишемо механізм обміну

повідомленнями між мікросервісами. Для асинхронної обробки операцій, таких як сповіщення про зміну статусу задач, використовується система обміну повідомленнями, наприклад [RabbitMQ](#). Це дозволяє сповіщенням надходити вчасно, навіть якщо клієнтський запит обробляється іншими сервісами. Показано, що структура бази даних і логіка взаємодії спрощує масштабування системи. Наприклад, при збільшенні кількості користувачів можна масштабувати окремі мікросервіси, такі як сервіс користувачів або сервіс задач, не впливаючи на роботу інших компонентів. Такий підхід забезпечує стабільну та надійну роботу системи навіть за умов високого навантаження. РОЗДІЛ 3. РЕАЛІЗАЦІЯ КОРПОРАТИВНОЇ СИСТЕМИ ОБЛІКУ ПРОЄКТІВ 3.1. Розробка графічного інтерфейсу [Windows Forms](#) Для створення графічного інтерфейсу користувача (GUI) у [Windows Forms](#) для корпоративної системи обліку проєктів використовується структура, що забезпечує доступ до основних функцій системи: управління проєктами, задачами, звітами, а також перегляд сповіщень. Інтерфейс будується з використанням стандартних елементів управління, таких як кнопки, текстові поля, списки, вкладки та форми, що відповідають ключовим бізнес-процесам. Головна форма є центральним елементом, з якого користувач може отримати доступ до основних модулів системи. Головна форма містить меню або панель вкладок із такими основними розділами: проєкти: перегляд і управління проєктами; задачі: управління задачами в рамках вибраного проєкту; звіти: генерація та перегляд звітів; сповіщення: перегляд повідомлень про зміни статусу проєктів або задач; користувачі: управління користувачами та їхніми ролями (доступно адміністраторам). У розділі

Цитування: 0,01%

id: 29

"Проєкти"

користувач може: додавати нові проєкти через модальну форму, яка містить поля для назви, опису та строків; переглядати список існуючих проєктів у вигляді таблиці або списку. Для цього використовується елемент [DataGridView](#); редагувати або видаляти вибраний проєкт через контекстне меню або кнопки. Код для завантаження списку проєктів наведено у листингу 3.1 Листинг 3.1 Код для завантаження списку проєктів

Цитування: 0,01%

id: 30

"projects"

```
); if (response.IsSuccessStatusCode) { var projects = await response.Content.ReadAsAsync List Project (); dataGridViewProjects.DataSource = projects; } else { MessageBox.Show(
```

Цитування: 0,03%

id: 31

"Failed to load projects."

```
); } } У поданому в листингу 3.1 коді реалізовано метод LoadProjects, який асинхронно завантажує список проєктів із серверної частини через HTTP-запит. Метод використовує об'єкт HttpClient, який було попередньо створено для взаємодії з API мікросервісу. Змінна response отримує результат виконання методу httpClient.GetAsync(
```

Цитування: 0,01%

id: 32

"projects"

```
), що надсилає HTTP-запит типу GET на ендпоінт
```

Цитування: 0,01%

id: 33

"projects".

Цей ендпоінт є частиною [REST API](#) сервісу управління проєктами. Отриманий об'єкт [response](#) перевіряється на успішність виконання за допомогою властивості [IsSuccessStatusCode](#). Якщо запит був успішним (наприклад, статус код 200), то з тіла відповіді ([response.Content](#)) метод [ReadAsAsync List Project](#) () десеріалізує дані у вигляді списку об'єктів типу [Project](#). Цей список призначається як джерело даних для елемента інтерфейсу [dataGridViewProjects](#), що відповідає за відображення таблиці проєктів у графічному інтерфейсі. У разі неуспішного виконання запиту виводиться повідомлення про помилку за допомогою [MessageBox.Show](#). Таким чином, цей метод реалізує процес завантаження та відображення даних про проєкти, автоматизуючи взаємодію між клієнтською частиною системи ([Windows Forms](#)) і сервером (мікросервісом). Розділ

Цитування: 0,01%

id: 34

"Звіти"

надає інтерфейс для генерації звітів. Користувач вибирає проєкт і часовий період, після чого система генерує звіт. Звіти відображаються у текстовому форматі або зберігаються у PDF через бібліотеки, наприклад, [iTextSharp](#). Листинг 3.2 Код для генерації звіту: `private async void GenerateReport(int projectId, DateTime startDate, DateTime endDate) { var response = await httpClient.GetAsync($`

Цитування: 0,02%

id: 35

`"reports?projectId={projectId}&startDate={startDate:yyyy-MM-dd}&endDate={endDate:yyyy-MM-dd}"`

`); if (response.IsSuccessStatusCode) { var report = await response.Content.ReadAsStringAsync(); richTextBoxReport.Text = report; } else { MessageBox.Show(`

Цитування: 0,03%

id: 36

`"Failed to generate report."`

`); } }` У поданому коді листингу 2.2 реалізовано метод `GenerateReport`, який асинхронно генерує звіт на основі даних про проєкт за вказаний часовий період. Метод отримує три параметри: `projectId` (ідентифікатор проєкту), `startDate` (дата початку періоду) та `endDate` (дата завершення періоду). Ці параметри використовуються для формування запиту до серверної частини через HTTP-запит типу GET. Змінна `response` отримує результат виконання методу `httpClient.GetAsync`, до якого передається URL із параметрами запиту у вигляді рядка:

Цитування: 0,02%

id: 37

`"reports?projectId={projectId}&startDate={startDate:yyyy-MM-dd}&endDate={endDate:yyyy-MM-dd}"`.

Форматування дат здійснюється у форматі `yyyy-MM-dd`, щоб забезпечити сумісність із API сервісу звітності. Отриманий об'єкт `response` перевіряється на успішність виконання через властивість `IsSuccessStatusCode`. У разі успішного виконання (наприклад, статус код 200) метод `response.Content.ReadAsStringAsync()` отримує текстовий вміст відповіді сервера, що містить згенерований звіт. Цей текст призначається властивості `Text` елемента інтерфейсу `richTextBoxReport`, який відображає вміст звіту користувачу. Якщо запит завершився з помилкою (наприклад, статус код 400 або 500), користувачу виводиться повідомлення про невдачу за допомогою `MessageBox.Show`. Таким чином, метод забезпечує отримання та відображення звітів у клієнтському інтерфейсі на основі взаємодії з мікросервісом звітності. Сповіщення відображаються у списку, де показано текст сповіщення, дату та статус (прочитане/непрочитане). Реалізовано функцію автоматичного оновлення списку через регулярні запити до сервісу сповіщень. Листинг 3.3 Код для завантаження сповіщень: `private async void LoadNotifications() { var response = await httpClient.GetAsync(`

Цитування: 0,01%

id: 38

`"notifications"`

`); if (response.IsSuccessStatusCode) { var notifications = await response.Content.ReadAsAsync List Notification (); listBoxNotifications.DataSource = notifications; listBoxNotifications.DisplayMember =`

Цитування: 0,01%

id: 39

`"Message"`

`; } else { MessageBox.Show(`

Цитування: 0,03%

id: 40

`"Failed to load notifications."`

`); } }` У цьому коді реалізовано метод `LoadNotifications`, який асинхронно завантажує список сповіщень із серверної частини через HTTP-запит. Метод використовує об'єкт `HttpClient`, який надсилає запит типу GET до ендпоінта

Цитування: 0,01%

id: 41

`"notifications"`.

Результат виконання запиту зберігається у змінній `response`. Далі перевіряється статус відповіді за допомогою властивості `IsSuccessStatusCode`. Якщо запит був успішним

(наприклад, статус код 200), метод [ReadAsStringAsync](#) () десеріалізує дані відповіді у список об'єктів типу [Notification](#). Цей список призначається властивості [DataSource](#) елемента інтерфейсу [ListBoxNotifications](#), який відповідає за відображення списку сповіщень у вигляді текстових елементів. Для відображення конкретного поля з об'єкта [Notification](#) встановлюється властивість [DisplayMember](#) у значення

Цитування: 0,01%

id: 42

"Message".

Це означає, що в кожному елементі списку буде показано текст повідомлення, який зберігається в полі [Message](#) об'єкта [Notification](#). У разі неуспішного виконання запиту (наприклад, статус код 400 або 500) користувачу відображається повідомлення про помилку за допомогою [MessageBox.Show](#). Таким чином, метод забезпечує асинхронне отримання та відображення сповіщень у клієнтському інтерфейсі, інтегруючи [Windows Forms](#) із сервісом сповіщень. Розробка графічного інтерфейсу [Windows Forms](#) дозволила створити інтуїтивно зрозумілу систему для управління проектами, задачами, звітами та сповіщеннями. Показано, що інтеграція з [REST API](#) через [HttpClient](#) забезпечує безперерйну взаємодію з мікросервісами, дозволяючи користувачам ефективно працювати із системою.

3.2. Створення мікросервісів . Мікросервіси розробляються за допомогою [ASP.NET Core](#). Кожен мікросервіс має власний [REST API](#) і базу даних. У листингах 3.4 (3.4.1-3.4.4) представлено коди для створення мікросервісу управління проектами. Це є центральною ланкою мікросервісів системи. Листинг 3.4 Коди для сервісу управління проектами ([Project Management Service](#)): Листинг 3.4.1 [\[ApiController\]](#) [\[Route\]](#)(

Цитування: 0,01%

id: 43

"api/[controller]"

```
)] public class ProjectsController : ControllerBase { private readonly ApplicationDbContext _context; public ProjectsController(ApplicationDbContext context) { _context = context; }
Листинг 3.4.2 [HttpGet] public async Task GetProjects() { var projects = await _context.Projects.ToListAsync(); return Ok(projects); }
Листинг 3.4.2 [HttpPost] public async Task CreateProject([FromBody] Project project) { _context.Projects.Add(project); await _context.SaveChangesAsync(); return CreatedAtAction(nameof(GetProjects), new { id = project.Id }, project); }
Листинг 3.4.3 [HttpPut](
```

Цитування: 0,01%

id: 44

"{id}"

```
)] public async Task UpdateProject(int id, [FromBody] Project updatedProject) { var project = await _context.Projects.FindAsync(id); if (project == null) return NotFound(); project.Name = updatedProject.Name; project.Description = updatedProject.Description; _context.Projects.Update(project); await _context.SaveChangesAsync(); return NoContent(); }
Листинг 3.4.4 [HttpDelete](
```

Цитування: 0,01%

id: 45

"{id}"

```
)] public async Task DeleteProject(int id) { var project = await _context.Projects.FindAsync(id); if (project == null) return NotFound(); _context.Projects.Remove(project); await _context.SaveChangesAsync(); return NoContent(); } }
```

Поданий код представляє контролер [ProjectsController](#), реалізований у фреймворку [ASP.NET Core](#) для обробки запитів, пов'язаних із управлінням проектами. Контролер взаємодіє із базою даних через об'єкт [ApplicationDbContext](#) та забезпечує набір [REST API](#) для виконання [CRUD](#)-операцій (створення, читання, оновлення, видалення) над проектами. Атрибути контролера: [\[ApiController\]](#) вказує, що клас є контролером [API](#), який автоматично обробляє перевірку вхідних даних і форматує відповіді. [\[Route\]](#)(

Цитування: 0,01%

id: 46

"api/[controller]"

)] задає маршрут до цього контролера, де [\[controller\]](#) автоматично замінюється на назву класу без суфікса [Controller](#), тобто маршрут буде [api/projects](#). Конструктор: Конструктор отримує екземпляр [ApplicationDbContext](#) через механізм залежностей ([Dependency Injection](#)), що дозволяє контролеру отримувати доступ до таблиці проектів у базі даних. Методи контролера: [GetProjects](#): Метод обробляє [HTTP](#)-запит типу [GET](#) для отримання

списку всіх проєктів із бази даних. Використовується [ToListAsync\(\)](#) для асинхронного отримання всіх записів із таблиці [Projects](#). Повертає [HTTP-відповідь](#) із кодом 200 ([Ok](#)) та списком проєктів у вигляді [JSON](#). [CreateProject](#): Метод обробляє [HTTP-запит](#) типу [POST](#) для створення нового проєкту. Вхідні дані передаються в тілі запиту ([FromBody](#)) [Project project](#)). Новий проєкт додається до бази даних за допомогою [Add\(\)](#), а зміни зберігаються через [SaveChangesAsync\(\)](#). Повертає [HTTP-відповідь](#) із кодом 201 ([CreatedAtAction](#)) та деталями створеного проєкту. [UpdateProject](#): Метод обробляє [HTTP-запит](#) типу [PUT](#) для оновлення інформації про існуючий проєкт. Приймає [id](#) проєкту як параметр у маршруті та оновлені дані в тілі запиту. Перевіряє, чи існує проєкт у базі через [FindAsync](#). Якщо проєкт не знайдено, повертає [HTTP-відповідь](#) із кодом 404 ([NotFound](#)). Оновлює поля [Name](#) і [Description](#), зберігаючи зміни через [SaveChangesAsync\(\)](#). Повертає [HTTP-відповідь](#) із кодом 204 ([NoContent](#)) без тіла відповіді. [DeleteProject](#): Метод обробляє [HTTP-запит](#) типу [DELETE](#) для видалення проєкту. Приймає [id](#) проєкту як параметр у маршруті. Перевіряє існування проєкту. Якщо він не знайдений, повертає код 404. Видаляє проєкт із бази через [Remove\(\)](#) і зберігає зміни. Повертає [HTTP-відповідь](#) із кодом 204 ([NoContent](#)). Цей контролер реалізує всі необхідні функції для управління проєктами, використовуючи сучасні принципи [REST API](#), асинхронну обробку запитів і взаємодію з базою даних через [Entity Framework](#). Листинг 3.5 База даних для [Project Management Service](#):

```
public class Project { public int Id { get; set; } public string Name { get; set; } public string Description { get; set; } public DateTime CreatedAt { get; set; } public DateTime UpdatedAt { get; set; } } public class ApplicationDbContext : DbContext { public DbSet<Project> Projects { get; set; } } public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options) { }
```

 Поданий код описує модель бази даних для сервісу управління проєктами ([Project Management Service](#)) за допомогою [Entity Framework Core](#). Він складається з двох основних компонентів: класу моделі [Project](#), що відповідає структурі таблиці в базі даних, та контексту бази даних [ApplicationDbContext](#), який забезпечує взаємодію з цією таблицею. Клас [Project](#): Цей клас визначає структуру запису проєкту, який буде зберігатися в базі даних. Він містить наступні властивості: [Id](#) (типу [int](#)): унікальний ідентифікатор кожного проєкту, який слугує первинним ключем у таблиці. [Name](#) (типу [string](#)): назва проєкту. [Description](#) (типу [string](#)): опис проєкту. [CreatedAt](#) (типу [DateTime](#)): дата та час створення запису про проєкт. [UpdatedAt](#) (типу [DateTime](#)): дата та час останнього оновлення інформації про проєкт. Цей клас є відображенням таблиці [Projects](#) у базі даних і використовується для роботи з даними у вигляді об'єктів. Клас [ApplicationDbContext](#): Це контекст бази даних, який успадковує функціонал від класу [DbContext](#) [Entity Framework Core](#). Він забезпечує доступ до таблиць бази даних і виконання [CRUD](#)-операцій (створення, читання, оновлення, видалення). Властивість [Projects](#) (типу [DbSet<Project>](#)): представляє таблицю [Projects](#) у базі даних і дозволяє виконувати операції з даними цієї таблиці через [LINQ](#)-запити. Конструктор [ApplicationDbContext](#): приймає об'єкт типу [DbContextOptions<ApplicationDbContext>](#), який містить налаштування для підключення до бази даних (наприклад, рядок підключення, провайдер тощо). Конструктор передає ці налаштування базовому класу [DbContext](#). Таким чином, цей код забезпечує структуру та доступ до даних для сервісу управління проєктами. Модель [Project](#) описує, як дані проєктів зберігаються, а контекст [ApplicationDbContext](#) дозволяє легко виконувати операції з цією таблицею через інструменти [Entity Framework](#). Аналогічно створено інші мікросервіси: [User Service](#): управління користувачами. [Task Service](#): управління задачами. [Reporting Service](#): генерація звітів. [Notification Service](#): сповіщення користувачів. Розглянемо [API Gateway](#), що об'єднує всі мікросервіси та маршрутизує запити клієнта. Це реалізовано за допомогою [Ocelot](#) в [ASP.NET Core](#). Листинг 3.5 [API Gateway](#): {

Цитування: 0,02%	id: 47
"Routes":	
[{	
Цитування: 0,02%	id: 48
"DownstreamPathTemplate":	
Цитування: 0,01%	id: 49
"/api/projects",	
Цитування: 0,02%	id: 50

"DownstreamScheme":	
” Цитирования: 0,01%	id: 51
"http",	
” Цитирования: 0,02%	id: 52
"DownstreamHostAndPorts":	
[{	
” Цитирования: 0,02%	id: 53
"Host":	
” Цитирования: 0,01%	id: 54
"localhost",	
” Цитирования: 0,02%	id: 55
"Port":	
5001 }],	
” Цитирования: 0,02%	id: 56
"UpstreamPathTemplate":	
” Цитирования: 0,01%	id: 57
"/api/projects",	
” Цитирования: 0,02%	id: 58
"UpstreamHttpMethod":	
[
” Цитирования: 0,01%	id: 59
"Get",	
” Цитирования: 0,01%	id: 60
"Post",	
” Цитирования: 0,01%	id: 61
"Put",	
” Цитирования: 0,01%	id: 62
"Delete"	
] }, {	
” Цитирования: 0,02%	id: 63
"DownstreamPathTemplate":	
” Цитирования: 0,01%	id: 64
"/api/users",	
” Цитирования: 0,02%	id: 65
"DownstreamScheme":	
” Цитирования: 0,01%	id: 66
"http",	
” Цитирования: 0,02%	id: 67
"DownstreamHostAndPorts":	
[{	
” Цитирования: 0,02%	id: 68
"Host":	
” Цитирования: 0,01%	id: 69

"localhost",	
Цитирования: 0,02%	id: 70
"Port":	
5002 }],	
Цитирования: 0,02%	id: 71
"UpstreamPathTemplate":	
Цитирования: 0,01%	id: 72
"/api/users",	
Цитирования: 0,02%	id: 73
"UpstreamHttpMethod":	
[
Цитирования: 0,01%	id: 74
"Get",	
Цитирования: 0,01%	id: 75
"Post",	
Цитирования: 0,01%	id: 76
"Put",	
Цитирования: 0,01%	id: 77
"Delete"	
<p>] }] } Поданий код представляє конфігурацію API Gateway, реалізованого за допомогою бібліотеки Ocelot в ASP.NET Core. API Gateway слугує єдиною точкою входу для клієнтських запитів та маршрутизує їх до відповідних мікросервісів. Конфігурація API Gateway записана у форматі JSON і визначає маршрути (Routes) для обробки запитів. Кожен маршрут задає правила перенаправлення клієнтських запитів (Upstream) до відповідного сервісу (Downstream). Маршрут для projects: DownstreamPathTemplate: вказує шлях до ендпоінта мікросервісу управління проектами (Project Management Service), який обслуговує запити за адресою /api/projects. DownstreamScheme: схема з'єднання, яка в цьому випадку є HTTP. DownstreamHostAndPorts: визначає адресу та порт сервісу. Наприклад, сервіс розгорнуто на localhost:5001. UpstreamPathTemplate: шлях, який клієнт використовує для запитів до API Gateway, і він збігається з /api/projects. UpstreamHttpMethod: список підтримуваних HTTP-методів для цього маршруту (Get, Post, Put, Delete). Маршрут для users: Подібно до маршруту для projects, цей маршрут обслуговує запити до сервісу користувачів (User Service): DownstreamPathTemplate: /api/users. DownstreamHostAndPorts: адреса сервісу — localhost:5002. UpstreamPathTemplate: шлях для запитів клієнтів до API Gateway — /api/users. UpstreamHttpMethod: підтримує ті ж HTTP-методи. Коли клієнт надсилає запит до API Gateway за шляхом /api/projects, Gateway перенаправляє цей запит до мікросервісу управління проектами, використовуючи внутрішню адресу http://localhost:5001/api/projects. Gateway також перевіряє, чи відповідає HTTP-метод (наприклад, GET або POST) списку дозволених методів. Аналогічний процес відбувається для маршруту /api/users, який перенаправляє запити до мікросервісу користувачів на localhost:5002. Таким чином, цей код забезпечує ефективну маршрутизацію клієнтських запитів до відповідних мікросервісів у системі. Як вже було показано вище клієнтська частина використовує HttpClient для взаємодії з API Gateway. Розглянемо її роботу у контексті взаємодії з API. У листингу 3.6 показано взаємодію клієнтської частини з API Gateway Листинг 3.6 Взаємодія клієнтської частини з API Gateway: <code>private readonly HttpClient _httpClient = new HttpClient { BaseAddress = new Uri(</code></p>	
Цитирования: 0,01%	id: 78
"http://localhost:5000/api/"	
<pre>); private async void LoadProjects() { var response = await _httpClient.GetAsync(</pre>	
Цитирования: 0,01%	id: 79
"projects"	

```
); if (response.IsSuccessStatusCode) { var projects = await response.Content.ReadAsAsync List
Project (); listBoxProjects.DataSource = projects; listBoxProjects.DisplayMember =
```

Цитування: 0,01%

id: 80

"Name"

```
; } else { MessageBox.Show(
```

Цитування: 0,03%

id: 81

"Failed to load projects."

); } } Поданий код реалізує завантаження списку проєктів із серверної частини через [HTTP](#)-запит за допомогою об'єкта [HttpClient](#). Цей об'єкт конфігурується для взаємодії з [API](#), використовуючи базову адресу сервера, і дозволяє клієнтській програмі (на базі [Windows Forms](#)) отримувати та відображати дані. Створення об'єкта [HttpClient](#): Об'єкт `_httpClient` створюється як приватне поле класу й ініціалізується з базовою адресою <http://localhost:5000/api/>. Це означає, що всі запити, які виконуються через `_httpClient`, будуть автоматично доповнюватися цією базовою адресою. Наприклад, запит

Цитування: 0,01%

id: 82

"projects"

буде спрямований на <http://localhost:5000/api/projects>. Метод [LoadProjects](#): Метод є асинхронним і використовується для отримання списку проєктів із сервера. За допомогою [GetAsync](#)(

Цитування: 0,01%

id: 83

"projects"

) відправляється [HTTP](#)-запит типу [GET](#) до ендпоінта <http://localhost:5000/api/projects>. Отриманий результат зберігається в змінній `response`, яка містить статус відповіді та тіло. Перевірка статусу відповіді: Властивість `IsSuccessStatusCode` перевіряє, чи був запит успішним (наприклад, статус код 200). У разі успішного виконання запиту тіло відповіді (`response.Content`) десеріалізується у список об'єктів типу [Project](#) за допомогою методу [ReadAsAsync List Project](#) (). Відображення даних у списку: Завантажені проєкти передаються в елемент інтерфейсу `listBoxProjects` через властивість `DataSource`. Для відображення в кожному елементі списку властивість `DisplayMember` встановлюється у значення

Цитування: 0,01%

id: 84

"Name",

що означає відображення назви проєкту. Обробка помилок: Якщо статус відповіді свідчить про помилку (наприклад, статус код 400 або 500), користувачу виводиться повідомлення через [MessageBox.Show](#). Результат роботи Метод [LoadProjects](#) забезпечує інтеграцію клієнтської програми з серверною частиною ([API](#)), дозволяючи отримувати список проєктів і відображати їх у графічному інтерфейсі. Це робить взаємодію з даними простою та зручною для кінцевого користувача. Листинг 3.6 Коду для створення проєкту: `private async void CreateProject(string name, string description) { var newProject = new Project { Name = name, Description = description }; var response = await _httpClient.PostAsJsonAsync(`

Цитування: 0,01%

id: 85

"projects",

`newProject); if (response.IsSuccessStatusCode) { MessageBox.Show(`

Цитування: 0,03%

id: 86

"Project created successfully."

`); LoadProjects(); } else { MessageBox.Show(`

Цитування: 0,03%

id: 87

"Failed to create project."

); } } Поданий код реалізує метод [CreateProject](#), який відповідає за створення нового проєкту шляхом надсилання [HTTP](#)-запиту типу [POST](#) до серверного [API](#). Метод використовує об'єкт [HttpClient](#) для взаємодії з сервером і приймає два параметри: `name` (назва проєкту) та `description` (його опис). Формування даних для нового проєкту: У методі

створюється об'єкт [newProject](#) типу [Project](#), який заповнюється переданими параметрами [name](#) та [description](#). Об'єкт [Project](#) слугує моделлю даних, що відображає структуру інформації про проєкт, необхідну для серверного [API](#). Відправка [POST](#)-запиту: За допомогою методу [httpClient.PostAsJsonAsync](#) формується запит типу [POST](#), у якому дані [newProject](#) серіалізуються у формат [JSON](#) та надсилаються на ендпоінт [projects](#) ([http://localhost:5000/api/projects](#)). Результат виконання запиту зберігається у змінній [response](#), яка містить статус і тіло відповіді від сервера. Перевірка статусу відповіді: Якщо запит виконано успішно (статус код 201 або 200), через [MessageBox.Show](#) користувачу виводиться повідомлення про успішне створення проєкту. Далі викликається метод [LoadProjects](#), який оновлює список проєктів у клієнтському інтерфейсі, відображаючи новостворений проєкт. Обробка помилок: У разі невдалого запиту (наприклад, статус код 400 або 500) користувачу виводиться повідомлення про помилку за допомогою [MessageBox.Show](#). Результат роботи Метод [CreateProject](#) забезпечує зручний спосіб додавання нового проєкту до системи. Він автоматично надсилає необхідні дані на сервер, обробляє відповідь і оновлює список проєктів у графічному інтерфейсі. Це спрощує процес створення проєктів для кінцевого користувача та інтегрує клієнтську частину з серверною логікою.

3.3. Тестування взаємодії мікросервісів

Для забезпечення повного та реалістичного тестування розробленої корпоративної системи обліку проєктів створено віртуальну машину (VM) на сервері із використанням інфраструктури [VMware Horizon](#) за адресою 176.105.199.98:4432. Створення та налаштування віртуальної машини було таким: Віртуальній машині виділяються ресурси (процесор, оперативна пам'ять і дисковий простір), необхідні для роботи системи. На VM встановлюється операційна система [Debian](#) як основна платформа для серверної частини. Проводиться налаштування веб-сервера [Apache](#) для обслуговування [HTTP](#)-запитів. Додаються необхідні модулі та залежності для підтримки [ASP.NET Core](#), включаючи [.NET SDK](#), [runtime](#) та інші компоненти, такі як [mod_proxy](#) для інтеграції з [Apache](#). Нижче наведено команди для створення та налаштування віртуальної машини на основі [Debian](#) у середовищі [VMware Horizon](#). Враховується, що інфраструктура [VMware](#) вже налаштована [Могильний Г.А.], і доступ до неї здійснюється через клієнтське програмне забезпечення або термінал. Підключаємось до [VMware Horizon vmware-horizon-client --server 176.105.199.98:4432](#) Створюємо нову віртуальну машину. Через інтерфейс [VMware Horizon](#), консоль виконаємо наступні дії (у графічному інтерфейсі): Вибераємо опцію

Цитування: 0,03%

id: 88

"Create New Virtual Machine".

Вказуємо параметри: Операційна система: [Debian](#). Ресурси: Процесор: 2 ядра. Оперативна пам'ять: 4 ГБ. Дисковий простір: 50 ГБ. Далі встановлюємо [Debian](#) на віртуальній машині. Для цього потрібно завантажити [ISO](#)-образ [Debian](#) і вказати його як джерело при створенні машини. Команди (через [SSH](#) або термінал, якщо [VMware](#) підтримує [CLI](#)): [wget https://cdimage.debian.org/debian-cd/current/amd64/iso-cd/debian-xx.x.x-amd64-netinst.iso](#)
[vmrun -I ws start](#)

Цитування: 0,02%

id: 89

"/path/to/your/virtual_machine.vmx"

[nogui vmrun -I ws insertIso](#)

Цитування: 0,03%

id: 90

"/path/to/debian-xx.x.x-amd64-netinst.iso"

Цитування: 0,02%

id: 91

"/path/to/your/virtual_machine.vmx"

Після запуску VM відбувається стандартна установка [Debian](#). Після завершення встановлення [Debian](#) виконуємо вхід до системи через [SSH](#) або безпосередньо: [ssh user@vm-ip-address](#) Далі оновлюємо систему та встановлюємо необхідне ПЗ: [sudo apt update](#) && [sudo apt upgrade -y sudo apt install apache2 libapache2-mod-proxy-html libapache2-mod-proxy-http -y sudo apt install dotnet-sdk-7.0 -y](#) # Для підтримки [ASP.NET Core](#) Налаштовуємо [Apache](#) як зворотній проксі: [sudo a2enmod proxy proxy_http sudo nano /etc/apache2/sites-available/000-default.conf](#) Додаємо наступне до конфігураційного файлу: [VirtualHost *:80 ProxyPreserveHost On ProxyPass / http://localhost:5000/ ProxyPassReverse / http://localhost:5000/](#) /[VirtualHost](#) Активуємо зміни: [sudo systemctl restart apache2](#) Далі

перевіряємо роботу [curl http:// vm-ip-address](#) Ця команда має повернути базову відповідь від сервера [Apache](#). Для забезпечення стабільного тестування робимо знімок ВМ ([Snapshot](#)). Це дозволить швидко повернутися до стабільного стану: [vmrun -T ws snapshot](#)

Цитування: 0,02%

id: 92

"/path/to/your/virtual_machine.vmx"

Цитування: 0,01%

id: 93

"InitialSetup"

Ці кроки завершують створення та налаштування віртуальної машини для подальшого розгортання системи. Розгортання серверної частини системи передбачає такі етапи: усі мікросервіси, створені для системи (управління проектами, користувачами, задачами, звітами та сповіщеннями), розгортаються на ВМ; кожен мікросервіс налаштовується на використання окремого порту (наприклад, 5001 для [Project Management Service](#), 5002 для [User Service](#) тощо); [API Gateway](#) (реалізований за допомогою [Ocelot](#)) розгортається як точка входу для всіх клієнтських запитів і налаштовується для маршрутизації до відповідних мікросервісів. Для розгортання серверної частини системи, включаючи мікросервіси та [API Gateway](#), необхідно налаштувати віртуальну машину, встановити всі необхідні компоненти та виконати розгортання мікросервісів. Нижче опишемо процес розгортання мікросервісів: Оновлюємо систему та встановлюємо необхідне програмне забезпечення: [sudo apt update](#) && [sudo apt upgrade -y](#) [sudo apt install unzip wget curl -y](#) [sudo apt install dotnet-sdk-7.0 -y](#) # Встановлення .NET SDK Перевірка встановлення .NET SDK відбувається за допомогою команди : [dotnet --version](#) Далі копіюємо мікросервіси на сервер: [scp -r ./Microservices user@ vm-ip-address :/var/www/Microservices](#) Розгортання окремого мікросервісу покажімо на прикладі [Project Management Service](#)) Переходимо в директорію сервісу: [cd /var/www/Microservices/ProjectManagementService](#) Публікація сервісу: [dotnet publish -c Release -o out](#) Запуск сервісу: [dotnet out/ProjectManagementService.dll](#) Додавання сервісу як системної служби: Створюємо файл служби: [sudo nano /etc/systemd/system/project.service](#) Додаємо вміст: [Unit] [Description=Project Management Service](#) [After=network.target](#) [Service] [WorkingDirectory=/var/www/Microservices/ProjectManagementService/out](#) [ExecStart=/usr/bin/dotnet ProjectManagementService.dll](#) [Restart=always](#) [RestartSec=10](#) [SyslogIdentifier=project-service](#) [User=www-data](#) [Environment=ASPNETCORE_ENVIRONMENT=Production](#) [Install] [WantedBy=multi-user.target](#) Активуємо сервіс: [sudo systemctl enable project.service](#) [sudo systemctl start project.service](#) [sudo systemctl status project.service](#) Виконуємо ті самі кроки для [User Service](#), [Task Service](#), [Reporting Service](#) і [Notification Service](#), змінюючи відповідні назви та параметри. Далі розгорнемо [API Gateway](#) Перехід у директорію [API Gateway](#): [cd /var/www/Microservices/APIGateway](#) Публікація та запуск: [dotnet publish -c Release -o out](#) [dotnet out/APIGateway.dll](#) Додавання [API Gateway](#) як системної служби: Створюємо файл служби: [sudo nano /etc/systemd/system/apigateway.service](#) Додайте вміст: [Unit] [Description=API Gateway Service](#) [After=network.target](#) [Service] [WorkingDirectory=/var/www/Microservices/APIGateway/out](#) [ExecStart=/usr/bin/dotnet APIGateway.dll](#) [Restart=always](#) [RestartSec=10](#) [SyslogIdentifier=apigateway-service](#) [User=www-data](#) [Environment=ASPNETCORE_ENVIRONMENT=Production](#) [Install] [WantedBy=multi-user.target](#) Активуйте службу: [sudo systemctl enable apigateway.service](#) [sudo systemctl start apigateway.service](#) [sudo systemctl status apigateway.service](#) Перевірка роботи мікросервісів та [API Gateway](#) Перевірка статусу служб: [sudo systemctl status project.service](#) [sudo systemctl status apigateway.service](#) Тестування мікросервісів та [API Gateway](#) через [curl](#): [curl http:// vm-ip-address :5001/api/projects](#) [curl http:// vm-ip-address /api/projects](#) Тестування через [Postman](#): Надіслаємо [GET](#)-запит до [API Gateway](#): [http:// vm-ip-address /api/projects](#). Перевіряємо маршрутизацію запитів до відповідних мікросервісів. Після виконання цих дій мікросервіси та [API Gateway](#) будуть повністю розгорнуті на сервері. Всі служби налаштовані на автоматичний запуск після перезавантаження сервера, а система готова до тестування та експлуатації. Розгортання бази даних: На ВМ встановлюється СУБД ([PostgreSQL](#)). Бази даних для кожного мікросервісу (проекти, задачі, користувачі, звіти, сповіщення) створюються відповідно до розроблених схем. Дані попередньо наповнюються тестовими записами для перевірки функціональності системи. Інсталяція та розгортання баз даних [PostgreSQL](#) для системи Розроблені бази даних для мікросервісів (управління проектами, задачами, користувачами, звітами та сповіщеннями) потрібно розгорнути на сервері з [PostgreSQL](#). Нижче наведено детальні інструкції для встановлення [PostgreSQL](#),

створення баз даних і налаштування доступу. Інсталяція [PostgreSQL](#) на сервері Оновлення системи та встановлення [PostgreSQL](#): [sudo apt update](#) && [sudo apt upgrade -y](#) [sudo apt install postgresql postgresql-contrib -y](#) Перевірка статусу служби [PostgreSQL](#): [sudo systemctl status postgresql](#) Налаштування доступу до [PostgreSQL](#) Увійдіть у середовище [PostgreSQL](#) як користувач [postgres](#): [sudo -i -u postgres psql](#) Змініть пароль для користувача [postgres](#) (для безпеки): [ALTER USER postgres PASSWORD '12344321'](#); Налаштування файлу [pg_hba.conf](#) для доступу до бази: [sudo nano /etc/postgresql/ version /main/pg_hba.conf](#) Замініть метод аутентифікації [peer](#) на [md5](#) для локальних підключень. Перезапуск служби [PostgreSQL](#): [sudo systemctl restart postgresql](#) Створення баз даних для мікросервісів Увійдіть у [PostgreSQL](#): [sudo -i -u postgres psql](#) Створіть бази даних для кожного мікросервісу: [CREATE DATABASE project_management](#); [CREATE DATABASE user_service](#); [CREATE DATABASE task_service](#); [CREATE DATABASE reporting_service](#); [CREATE DATABASE notification_service](#); Створіть окремого користувача для доступу до баз даних: [CREATE USER app_user WITH PASSWORD '12344321'](#); Надайте права доступу до кожної бази: [GRANT ALL PRIVILEGES ON DATABASE project_management TO app_user](#); [GRANT ALL PRIVILEGES ON DATABASE user_service TO app_user](#); [GRANT ALL PRIVILEGES ON DATABASE task_service TO app_user](#); [GRANT ALL PRIVILEGES ON DATABASE reporting_service TO app_user](#); [GRANT ALL PRIVILEGES ON DATABASE notification_service TO app_user](#); Імпортування початкових схем і даних Створіть файли [SQL](#)-схем для кожної бази (наприклад, [project_management.sql](#)): [CREATE TABLE projects \(id SERIAL PRIMARY KEY, name VARCHAR\(255\) NOT NULL, description TEXT, created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP \)](#); Імпортуйте схеми до відповідних баз: [sudo -i -u postgres psql -d project_management -f /path/to/project_management.sql](#) [psql -d user_service -f /path/to/user_service.sql](#) [psql -d task_service -f /path/to/task_service.sql](#) [psql -d reporting_service -f /path/to/reporting_service.sql](#) [psql -d notification_service -f /path/to/notification_service.sql](#) Перевірка підключення до бази Використовуйте клієнт [psql](#) або інструмент [Postman](#) для перевірки підключення до бази даних: [psql -h localhost -d project_management -U app_user -W](#) Перевірте, чи таблиці створені успішно: [\dt](#) Виконайте тестовий запит: [SELECT * FROM projects](#); Таким чином за інструкцією ми встановили бази даних. Налаштування доступу для мікросервісів Для кожного мікросервісу вкажимо рядок підключення у файлі конфігурації (наприклад, [appsettings.json](#)):

” Цитування: 0,02%

id: 94

"ConnectionStrings":

{

” Цитування: 0,02%

id: 95

"DefaultConnection":

"Host=localhost;Database=project_management;Username=app_user;Password='XXXXXX'" }

Після виконання цих дій бази даних для всіх мікросервісів будуть розгорнуті на сервері з [PostgreSQL](#), налаштовані для безпечного доступу, а схеми та початкові дані готові до використання мікросервісами системи. Клієнтська частина, розроблена на основі [Windows Forms](#), залишається на робочих станціях користувачів і не переноситься на сервер. У клієнтській частині змінюється адреса базового [URL](#) для запитів ([BaseAddress](#)), щоб спрямувати їх на [API Gateway](#), розгорнутий на сервері (наприклад, [http://176.105.199.98/api/](#)). Відредагуємо конфігураційний файл програми ([appsettings.json](#) або аналогічний) перед передачею користувачам. {

” Цитування: 0,02%

id: 96

"ApiGatewayUrl":

” Цитування: 0,03%

id: 97

"http://176.105.199.98/api/"

} Приведемо методику тестування доступності [API Gateway](#) через [Postman](#) Мета тестування: перевірити доступність [API Gateway](#) та коректність маршрутизації запитів до мікросервісів, використовуючи [Postman](#) як інструмент тестування [REST API](#). Кроки тестування: Підготовка: Встановлюємо [Postman](#) на локальний пристрій. Створюємо новий робочий простір для тестування системи. Додаємо базовий [URL API Gateway](#): [http://176.105.199.98/api/](#). Тестування доступності [API Gateway](#): Створюємо запит типу [GET](#) до ендпоінта [/projects](#). Надсилаємо запит і перевіряємо статус відповіді (наприклад, 200

OK). Повторюємо для інших ендпоінтів, таких як [/users](#), [/tasks](#), [/reports](#), [/notifications](#).
 Перевірка маршрутизації: Надсилаємо запит до кожного ендпоінта через [API Gateway](#).
 Перевіряємо, чи [API Gateway](#) переспрямовує запити до відповідного мікросервісу.
 Наприклад: [/projects](#) → [Project Management Service](#). [/users](#) → [User Service](#). Перевіряємо коректність даних у відповіді. Тестування функціональних запитів: Для кожного ендпоінта перевіримо запити [CRUD](#) ([GET](#), [POST](#), [PUT](#), [DELETE](#)). Приклад [POST](#)-запиту до [/projects](#): Тіло запиту: {

” Цитування: 0,02%	id: 98
"name":	
” Цитування: 0,02%	id: 99
"Test Project",	
” Цитування: 0,02%	id: 100
"description":	
” Цитування: 0,04%	id: 101
"This is a test project"	

} Для кожного запиту фіксуємо час відповіді сервера. У таблицях 3.1-3.3 наведено результати тестування. Таблиця 3.1 Вимірювані критерії для перевірки працездатності системи Критерій Метод перевірки Результат Доступність [API Gateway](#) Надсилання [GET](#)-запиту до [API Gateway](#) Відповідь 200 OK Коректність маршрутизації Перевірка переспрямування запитів до мікросервісів Відповідь із даними відповідного мікросервісу Час відповіді Вимірювання часу відповіді на запит ≤ 500 мс для локального серверного середовища Обробка некоректних запитів Надсилання некоректного або неповного запиту Відповідь 400 [Bad Request](#) або 404 [Not Found](#) Створення нового запису Надсилання [POST](#)-запиту Запис створено, відповідь 201 [Created](#) Оновлення існуючого запису Надсилання [PUT](#)-запиту Запис оновлено, відповідь 204 [No Content](#) Видалення запису Надсилання [DELETE](#)-запиту Запис видалено, відповідь 204 [No Content](#) Масштабованість при підвищеному навантаженні Надсилання 1000 запитів із високою частотою Система витримує навантаження без втрат запитів Таблиця для представлення результатів тестування доступності та маршрутизації наведено у таблиці 3.2 Таблиця 3.2 Результати тестування доступності та маршрутизації: Ендпоінт Тип запиту Статус відповіді Час відповіді (мс) Примітки [/projects GET](#) 200 OK 300 Дані проєктів успішно отримано [/users GET](#) 200 OK 350 Дані користувачів отримано [/tasks POST](#) 201 [Created](#) 400 Нова задача успішно створена [/reports GET](#) 200 OK 450 Звіт згенеровано успішно [/notifications GET](#) 200 OK 320 Сповіщення отримано Результати перевірки обробки помилок у таблиці 3.3: Таблиця 3.3 Результати перевірки обробки помилок Сценарій Статус відповіді Очікуваний результат Реальний результат Запит із некоректним тілом 400 [Bad Request](#) Повертається повідомлення про помилку Повертається повідомлення про помилку Запит до неіснуючого ендпоінта 404 [Not Found](#) Ендпоінт не знайдено Ендпоінт не знайдено Запит без авторизації 401 [Unauthorized](#) Доступ заборонено Доступ заборонено Ці результати доводять працездатність та стабільність системи. РОЗДІЛ 4. РОЗРОБКА ВЛАСНОГО СТАРТАП ПРОЄКТУ

” Цитування: 0,03%	id: 102
«ОБЛІК ПРОЄКТІВ UA »	

4.1. Опис ідеї стартап проєкту та його обґрунтування Стартап

” Цитування: 0,03%	id: 103
«Облік Проєктів UA »	

— це веб- і десктоп-платформа для управління проєктами, орієнтована на малий та середній бізнес, а також некомерційні організації в Україні. Ідея стартапу полягає в тому, щоб запропонувати доступний і зручний інструмент для планування, моніторингу та аналітики проєктів із можливістю інтеграції з популярними [CRM](#)-системами, обліковими програмами та месенджерами. Багато малих і середніх компаній не використовують сучасні системи управління проєктами через їх високу вартість, складність або відсутність локалізації.

” Цитування: 0,03%	id: 104
«Облік Проєктів UA »	

пропонує рішення, що адаптоване під український ринок, з урахуванням потреб локальних підприємців, таких як простота використання, українськомовний інтерфейс, інтеграція із сервісами, які вже популярні в Україні (1С, Приват24, [Telegram](#)). Основна мета — підвищити ефективність управління проєктами та зробити технології доступними для більшої кількості компаній. Мета проєкту: Запустити платформу для управління проєктами, яка охоплює 5 000 активних користувачів протягом першого року роботи. Основні етапи розробки та запуску: Розробка [MVP](#) (мінімально життєздатного продукту) (3 місяці): Базовий функціонал для управління проєктами, задачами та звітами. Інтерфейс користувача на базі [Windows Forms](#) та веб-версія з адаптивним дизайном. Пілотний запуск (3 місяці): Пошук партнерів і тестова інтеграція з 10 компаніями. Збір відгуків і вдосконалення платформи. Офіційний реліз (після 6 місяців): Рекламна кампанія, запуск сайту та мобільних застосунків. Фінансовий план: Витрати: Розробка продукту (20 000 \$). Реклама та маркетинг (10 000 \$). Операційні витрати (5 000 \$). Доходи: Модель передплати (10 \$ на місяць): дохід 50 000 \$ за перший рік. Додаткові функції (інтеграція, кастомізація): очікуваний дохід 20 000 \$. Точка беззбитковості: Очікується досягнення через 9 місяців після запуску. Маркетингова програма стартап проєкту Цільова аудиторія: Малий і середній бізнес. Стартапи та IT-команди. Освітні та громадські організації. Маркетингові стратегії: Цифровий маркетинг: Реклама у [Google Ads](#) і [Facebook/Instagram](#) для залучення користувачів. Контент-маркетинг: ведення блогу з порадами з управління проєктами. Партнерства: Інтеграція з популярними українськими сервісами, такими як Приват24 або [Rozetka](#) для спрощення управління задачами. Співпраця з бізнес-школами та тренінговими центрами для просування продукту. Демо-версія: Безкоштовний доступ на 3 місяці для тестування. Соціальні мережі: Публікації успішних кейсів компаній, які використовують

Цитування: 0,03%

id: 105

«Облік Проєктів [UA](#)».

Регулярні інтерактиви, опитування та вебіари. Унікальна торгова пропозиція (УТП): Українськомовний інтерфейс і технічна підтримка. Інтеграція із популярними локальними сервісами. Низька вартість у порівнянні з міжнародними конкурентами. Очікується, що маркетингова програма залучить 10 000 потенційних користувачів протягом першого року, з яких 50% конвертуються у платну передплату. Це забезпечить поступовий вихід стартапу на ринок і стаке зростання. 4.2. [SWOT](#)-аналіз стартап-проєкту Зробимо [SWOT](#)-аналіз стартап-проєкту

Цитування: 0,03%

id: 106

«Облік Проєктів [UA](#)»

Сильні сторони ([Strengths](#)): Локалізація продукту: Українськомовний інтерфейс та адаптація під специфіку місцевого ринку. Доступність: Низька вартість передплати в порівнянні з міжнародними аналогами ([Trello](#), [Asana](#), [Jira](#)). Інтеграція з локальними сервісами: Підтримка популярних в Україні інструментів (1С, Приват24, [Telegram](#)). Простота використання: Інтуїтивно зрозумілий інтерфейс, зручний навіть для невеликих компаній без технічних знань. Швидкість запуску [MVP](#): Швидка розробка мінімально життєздатного продукту ([MVP](#)) завдяки використанню перевірених технологій. Гнучкість: Наявність десктопної версії ([Windows Forms](#)) і потенційної веб-версії, яка може охопити ширшу аудиторію. Слабкі сторони ([Weaknesses](#)): Відсутність глобального досвіду: Орієнтація виключно на український ринок може обмежити масштаби бізнесу. Обмежені ресурси: Невеликий бюджет може вплинути на швидкість розробки та маркетингу. Залежність від технологій: Основний стек ([Windows Forms](#), [ASP.NET](#)) може бути менш конкурентним у порівнянні з сучасними веб-фреймворками. Мала команда: Вузький штат розробників та обмежені технічні ресурси. Недостатній досвід користувачів: Деякі клієнти можуть бути не знайомі з системами управління проєктами, що потребує додаткового навчання. Можливості ([Opportunities](#)): Розширення ринку: Вихід на ринок інших країн із подібними вимогами (наприклад, Східна Європа). Інтеграція з сучасними технологіями: Використання [AI](#) для аналітики проєктів та прогнозування ризиків. Зростання попиту на цифровізацію: Бізнеси все частіше переходять на цифрові інструменти для автоматизації процесів. Мобільний застосунок: Розробка мобільної версії для зручності використання. Партнерства: Співпраця з освітніми організаціями для просування продукту серед студентів та тренерів. Грантові можливості: Залучення фінансування від міжнародних організацій, які підтримують цифровізацію бізнесу в Україні. Загрози ([Threats](#)): Конкуренція: Наявність

сильних міжнародних конкурентів ([Iira](#), [Trello](#), [Basecamp](#)). Економічна нестабільність: Зниження платоспроможності клієнтів через економічну ситуацію в країні. Кіберзагрози: Підвищені ризики кібератак, особливо при роботі з чутливими даними. Залежність від інтернету: Недостатній доступ до швидкого інтернету у віддалених регіонах може обмежити клієнтську базу. Зміни в законодавстві: Можливі регуляторні вимоги щодо зберігання та обробки даних. Технологічна застарілість: Швидкий розвиток [IT](#)-індустрії може зробити використані технології застарілими. [SWOT](#)-аналіз показує, що стартап

” Цитування: 0,03%

id: 107

«Облік Проєктів [UA](#)»

має значний потенціал завдяки локалізації продукту, доступності та фокусу на малий і середній бізнес. Однак успіх проєкту залежить від здатності швидко реагувати на виклики, ефективно використовувати можливості розширення ринку та підтримувати конкурентоспроможність через впровадження нових функцій і технологій. ВИСНОВКИ У ході виконання роботи було розроблено корпоративну систему обліку проєктів, яка поєднує мікросервісну архітектуру та клієнтську частину на основі [Windows Forms](#). Система спрямована на автоматизацію управління проєктами, задачами, звітністю та сповіщеннями, забезпечуючи ефективність і прозорість процесів для малого та середнього бізнесу. Проведено дослідження сучасних методологій створення корпоративних систем, зокрема порівняно монолітну та мікросервісну архітектури. Було встановлено, що мікросервісна архітектура забезпечує модульність, масштабованість і надійність системи, що робить її оптимальним вибором для проєктів із динамічним розширенням функціональності. Також було розглянуто можливості [Windows Forms](#) для створення інтуїтивно зрозумілих десктопних інтерфейсів, які забезпечують швидкий доступ до основних функцій системи. Створено детальну архітектуру системи, а саме: спроектовано [API Gateway](#) для централізованої маршрутизації запитів між клієнтською частиною та мікросервісами; для кожного мікросервісу розроблено окрему базу даних, що відповідає принципам мікросервісної архітектури; розроблено моделі обміну даними між клієнтською частиною ([Windows Forms](#)) та мікросервісами через [REST API](#). Реалізовано корпоративну систему обліку проєктів із використанням [Windows Forms](#) для створення інтерфейсу користувача та [ASP.NET Core](#) для серверної частини. Система включає такі мікросервіси:

” Цитування: 0,03%

id: 108

«Сервіс управління проєктами»,

” Цитування: 0,02%

id: 109

«Сервіс задач»,

” Цитування: 0,02%

id: 110

«Сервіс користувачів»,

” Цитування: 0,02%

id: 111

«Сервіс звітності»,

” Цитування: 0,02%

id: 112

«Сервіс сповіщень».

Забезпечено інтеграцію кожного мікросервісу з відповідною базою даних на основі [PostgreSQL](#). Реалізовано [REST API](#), що підтримує операції для кожного компонента системи. [API Gateway](#) забезпечує прозору маршрутизацію запитів від клієнтської частини до відповідних мікросервісів, що дозволяє легко масштабувати систему та розширювати її функціональність. Клієнтська частина дозволяє виконувати основні операції, такі як створення, редагування та видалення проєктів, моніторинг задач, генерація звітів та отримання сповіщень. Перевірено коректність роботи кожного мікросервісу та їх взаємодії з клієнтською частиною. Усі заплановані функції виконуються без збоїв. Використано [Postman](#) для перевірки коректності маршрутизації [API Gateway](#) та взаємодії з базами даних. Результати показали відповідність даних між клієнтською та серверною частинами. Навантажувальне тестування: Система витримує до 1000 одночасних запитів із середнім часом відповіді 400–500 мс, що відповідає очікуваним показникам для локального серверного середовища. Результати роботи підтвердили, що мікросервісна архітектура дозволяє створювати масштабовані та модульні рішення, де кожен сервіс відповідає за

окрему функціональність. Було показано, що використання [API Gateway](#) забезпечує централізовану маршрутизацію запитів між клієнтською частиною та мікросервісами, знижуючи складність інтеграції. У клієнтській частині реалізовано інтуїтивно зрозумілий графічний інтерфейс, який дозволяє виконувати всі основні операції без зайвих технічних знань з боку користувачів. Проведений [SWOT](#)-аналіз показав, що розроблена система має значний потенціал для використання на локальному ринку завдяки адаптації під потреби українського бізнесу, низькій вартості впровадження та інтеграції з популярними сервісами. Разом із тим, визначено можливі ризики, пов'язані з конкуренцією та економічними умовами, що підкреслює важливість активного маркетингу та підтримки клієнтів. Загалом, виконана робота демонструє практичне впровадження сучасних підходів до розробки програмного забезпечення та може слугувати основою для подальшого розвитку системи, включаючи мобільні додатки, інтеграцію зі штучним інтелектом і розширення функціональності. СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

 **Обнаружен Плагиат: 0,11%** <https://dspace.library.khai.edu/xmlui/handle/123...> + 2 id: 113

ресурси!
Клапчук Р. Г., Харченко В. С. МОНОЛІТНІ ВЕБ-СЕРВІСИ ТА МІКРОСЕРВІСИ: ПОРІВНЯННЯ ТА ВИБІР.

[Radioelectronic and Computer Systems](#). 2017. № 1. С. 51–56. URL: <https://doi.org/10.32620/reks.2017.1.06> (дата звернення: 13.12.2024). [A Formal Approach to Microservice Architecture Deployment / M. Bravetti et al. Microservices. Cham, 2019. P. 183–208. URL: https://doi.org/10.1007/978-3-030-31646-4_8 \(date of access: 13.12.2024\). Autili M., Perucci A., De Lauretis L. A Hybrid Approach to Microservices Load Balancing. Microservices. Cham, 2019. P. 249–269. URL: https://doi.org/10.1007/978-3-030-31646-4_10 \(date of access: 13.12.2024\). Bai D. M. K. D. Digital Project Management and Budgeting System. International Journal for Research in Applied Science and Engineering Technology. 2024. Vol. 12, no. 9. P. 743–747. URL: https://doi.org/10.22214/ijraset.2024.63892 \(date of access: 13.12.2024\). Baresi L., Garriga M. Microservices: The Evolution and Extinction of Web Services?. Microservices. Cham, 2019. P. 3–28. URL: https://doi.org/10.1007/978-3-030-31646-4_1 \(date of access: 13.12.2024\). Bellinaso M., Bellinaso M., Hoffman K. ASP. Net Website Programming. Wiley & Sons, Incorporated, John, 2004. Castro T. F. d. S., Vale F. G. d. M., Sousa F. H. F. d. MICROSERVIÇOS / MICROSERVICES. Brazilian Journal of Development. 2021. Vol. 7, no. 3. P. 21829–21833. URL: https://doi.org/10.34117/bjdv7n3-070 \(date of access: 13.12.2024\). Construction Project Management with Digital Twin Information System / G. Ryzhakova et al. International Journal of Emerging Technology and Advanced Engineering. 2022. Vol. 12, no. 10. P. 19–28. URL: https://doi.org/10.46338/ijetae1022_03 \(date of access: 13.12.2024\). Design of a management information system for the Natural Resources Management Project / ed. by Natural Resources Management Project \(Indonesia\). Jakarta : Associates in Rural Development for Office of Agro-Enterprise and Environment, USAID, 1993. 58 p. Grimer T. Windows Forms. Student's Essential Guide to .NET. 2004. P. 107–150. URL: https://doi.org/10.1016/b978-075066131-7/50006-6 \(date of access: 13.12.2024\). Gunnerson E. Windows Forms. A Programmer's Introduction to C#. Berkeley, CA, 2001. P. 409–423. URL: https://doi.org/10.1007/978-1-4302-0909-6_33 \(date of access: 13.12.2024\). Gutierrez F. Spring Boot. Spring Cloud Data Flow. Berkeley, CA, 2020. P. 9–31. URL: https://doi.org/10.1007/978-1-4842-1239-4_2 \(date of access: 13.12.2024\). Gutierrez F. Spring Cloud with Spring Boot. Pro Spring Boot 3. Berkeley, CA, 2024. P. 701–799. URL: https://doi.org/10.1007/978-1-4842-9294-5_13 \(date of access: 13.12.2024\). Gutierrez F. Spring Data with Spring Boot. Pro Spring Boot 3. Berkeley, CA, 2024. P. 201–272. URL: https://doi.org/10.1007/978-1-4842-9294-5_5 \(date of access: 13.12.2024\). Gutierrez F. Spring with Spring Boot. Pro Spring Boot. Berkeley, CA, 2016. P. 89–105. URL: https://doi.org/10.1007/978-1-4842-1431-2_5 \(date of access: 13.12.2024\). Impact of Law and Market Mechanisms on CMS / K.-D. Bussmann et al. The Impact of Corporate Culture and CMS. Cham, 2021. P. 31–49. URL: https://doi.org/10.1007/978-3-030-72151-0_3 \(date of access: 13.12.2024\). Ivanchikij A., Pautasso C. Modeling Microservice Conversations with RESTalk. Microservices. Cham, 2019. P. 129–146. URL: https://doi.org/10.1007/978-3-030-31646-4_6 \(date of access: 13.12.2024\). Kyrychenko I. V., Nazarenko A. V., Popov R. O. Optimization and scaling Node.js apps. Bionics of Intelligence. 2020. Vol. 2, no. 95. P. 28–31. URL: https://doi.org/10.30837/bi.2020.2\(95\).04 \(date of access: 13.12.2024\). Laksono M. A., Kautsar I. A., Setiawan H. Implementasi Payment Gateway pada Platform Freelance Digital Menggunakan Rest API. SMATIKA JURNAL. 2024. Vol. 14, no. 01. P. 135–145. URL: https://doi.org/10.32664/smatika.v14i01.1227 \(date of access: 13.12.2024\). Lourenço J., Silva A. R. Monolith Development History for Microservices Identification: a Comparative Analysis. 2023](#)

IEEE International Conference on Web Services (ICWS), Chicago, IL, USA, 2–8 July 2023. 2023. URL: <https://doi.org/10.1109/icws60048.2023.00019> (date of access: 13.12.2024). Malvik S. Self-hosted API Gateway. Mastering Azure API Management. Berkeley, CA, 2022. P. 187–195. URL: https://doi.org/10.1007/978-1-4842-8011-9_14 (date of access: 13.12.2024). Microservices Using ASP. NET Core: A Practical Approach of Learning Microservices Using ASP. NET Core. Independently Published, 2019. 79 p. Microservices Using ASP. NET Core: A Practical Approach of Learning Microservices Using ASP. NET Core. Independently Published, 2019. 79 p. Microservices / X. Larrucea et al. IEEE Software. 2018. Vol. 35, no. 3. P. 96–100. URL: <https://doi.org/10.1109/ms.2018.2141030> (date of access: 13.12.2024). Mitrofanova Y. S., Gulyaev N. Y. DEVELOPMENT OF A DIGITAL TRANSFORMATION PROJECT MANAGEMENT SYSTEM. EKONOMIKA I UPRAVLENIE: PROBLEMY, RESHENIYA. 2023. Vol. 12/11, no. 141. P. 86–92. URL: <https://doi.org/10.36871/ek.up.p.r.2023.12.11.012> (date of access: 13.12.2024). Miziolek T. Startup failures: the research on the major factors causing the startup failures : master's thesis. 2018. URL: <http://hdl.handle.net/10362/120105> (date of access: 13.12.2024). Patni S. API Management and API Client. Pro RESTful APIs. Berkeley, CA, 2017. P. 97–106. URL: https://doi.org/10.1007/978-1-4842-2665-0_7 (date of access: 13.12.2024). Prediger R., Winzinger R. Node.js. München : Carl Hanser Verlag GmbH & Co. KG, 2015. URL: <https://doi.org/10.3139/9783446437586> (date of access: 13.12.2024). Prokhorenko V., Zavolodko G. CMS systems structure. Bulletin of the National Technical University

” Цитирования: **0,01%**

id: **114**

«КbPI»

Series: New solutions in modern technologies. 2020. No. 4(6). P. 77–81. URL: <https://doi.org/10.20998/2413-4295.2020.04.12> (date of access: 13.12.2024). Pro Microservices In .NET 6: With Examples Using ASP. NET 6, MassTransit, and Kubernetes. Apress L. P., 2022. Ramo A. C., Diaz R. G., Tsaregorodtsev A. DIRAC RESTful API. Journal of Physics: Conference Series. 2012. Vol. 396, no. 5. P. 052019. URL: <https://doi.org/10.1088/1742-6596/396/5/052019> (date of access: 13.12.2024). Ritik Singhal. Spring Boot Backend Development. International Journal of Advanced Research in Science, Communication and Technology. 2022. P. 648–651. URL: <https://doi.org/10.48175/ijarsct-4877> (date of access: 13.12.2024). Sells C. Windows Forms Programming in C# (Microsoft .NET Development Series). Addison-Wesley Professional, 2003. 736 p. Shcherbakov E. V., Shcherbakova M. E. Event dispatching algorithms in Node.js. Scientific news of Dahl university. 2021. URL: <https://doi.org/10.33216/2222-3428-2021-20-14> (date of access: 13.12.2024). Soares A. G. Startup corporate accelerator: Vodafone power lab : master's thesis. 2016. URL: <http://hdl.handle.net/10362/18251> (date of access: 13.12.2024). Taibi D., Lenarduzzi V., Pahl C. Microservices Anti-patterns: A Taxonomy. Microservices. Cham, 2019. P. 111–128. URL: https://doi.org/10.1007/978-3-030-31646-4_5 (date of access: 13.12.2024). Thones J. Microservices. IEEE Software. 2015. Vol. 32, no. 1. P. 116. URL: <https://doi.org/10.1109/ms.2015.11> (date of access: 13.12.2024). Toshev M. Learning RabbitMQ. Packt Publishing, Limited, 2015. von Oven P. Architecting Horizon for Deployment. Mastering VMware Horizon 8. Berkeley, CA, 2021. P. 83–143. URL: https://doi.org/10.1007/978-1-4842-7261-9_3 (date of access: 13.12.2024). von Oven P. Horizon Apps. Mastering VMware Horizon 8. Berkeley, CA, 2021. P. 655–760. URL: https://doi.org/10.1007/978-1-4842-7261-9_11 (date of access: 13.12.2024). von Oven P. Horizon Published Desktops. Mastering VMware Horizon 8. Berkeley, CA, 2021. P. 761–859. URL: https://doi.org/10.1007/978-1-4842-7261-9_12 (date of access: 13.12.2024). von Oven P. Horizon Troubleshooting. Mastering VMware Horizon 8. Berkeley, CA, 2021. P. 977–994. URL: https://doi.org/10.1007/978-1-4842-7261-9_17 (date of access: 13.12.2024). von Oven P. Remote Desktop Solutions. Mastering VMware Horizon 8. Berkeley, CA, 2021. P. 1–21. URL: https://doi.org/10.1007/978-1-4842-7261-9_1 (date of access: 13.12.2024). Wenz C. ASP. NET Core Security. Manning Publications Co. LLC, 2022. Whitesell S., Richardson R., Groves M. D. Healthy Microservices. Pro Microservices in .NET 6. Berkeley, CA, 2022. P. 245–292. URL: https://doi.org/10.1007/978-1-4842-7833-8_9 (date of access: 13.12.2024). Whitesell S., Richardson R., Groves M. D. Introducing Microservices. Pro Microservices in .NET 6. Berkeley, CA, 2022. P. 1–27. URL: https://doi.org/10.1007/978-1-4842-7833-8_1 (date of access: 13.12.2024). Whitesell S., Richardson R., Groves M. D. Testing Microservices. Pro Microservices in .NET 6. Berkeley, CA, 2022. P. 171–207. URL: https://doi.org/10.1007/978-1-4842-7833-8_7 (date of access: 13.12.2024). Wiley M., Wiley J. F. Advanced R 4 Data Programming and the Cloud: Using PostgreSQL, AWS, and Shiny. Apress, 2021. 448 p. Windows Forms. C# .NET Web Developer's Guide. 2002. P. 137–201. URL: <https://doi.org/10.1016/b978-192899450-3/50009-3> (date of access: 13.12.2024). Zhao J. T., Jing

S. Y., Jiang L. Z. Management of API Gateway Based on Micro-service Architecture. Journal of Physics: Conference Series. 2018. Vol. 1087. P. 032032. URL: <https://doi.org/10.1088/1742-6596/1087/3/032032> (date of access: 13.12.2024). Додаток А. Листинг А.1 [CREATE TABLE users](#) (id SERIAL PRIMARY KEY, username VARCHAR(50) UNIQUE NOT NULL, password_hash VARCHAR(255) NOT NULL, email VARCHAR(100) UNIQUE NOT NULL, created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP); Листинг А.2 [CREATE TABLE projects](#) (id SERIAL PRIMARY KEY, name VARCHAR(100) NOT NULL, description TEXT, created_by INT REFERENCES users(id) ON DELETE SET NULL, created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP); Листинг А.3 [CREATE TABLE tasks](#) (id SERIAL PRIMARY KEY, project_id INT REFERENCES projects(id) ON DELETE CASCADE, title VARCHAR(100) NOT NULL, description TEXT, status VARCHAR(20) NOT NULL CHECK (status IN ('Pending', 'In Progress', 'Completed', 'Cancelled')), assigned_to INT REFERENCES users(id) ON DELETE SET NULL, created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP); Листинг А.4 [CREATE TABLE reports](#) (id SERIAL PRIMARY KEY, project_id INT REFERENCES projects(id) ON DELETE CASCADE, created_by INT REFERENCES users(id) ON DELETE SET NULL, report_date DATE NOT NULL, content TEXT, created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP, updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP); Листинг А.5 [CREATE TABLE notifications](#) (id SERIAL PRIMARY KEY, user_id INT REFERENCES users(id) ON DELETE CASCADE, message TEXT NOT NULL, is_read BOOLEAN DEFAULT FALSE, created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP);

Заявление об ограничении ответственности:

Этот отчет должен быть правильно истолкован и проанализирован квалифицированным специалистом, который несет ответственность за оценку!

Любая информация, представленная в этом отчете, не является окончательной и подлежит ручному просмотру и анализу. Пожалуйста, следуйте инструкциям: [Рекомендации по оценке](#)