МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ІВАНА ФРАНКА

Кафедра теорії
оптимальних процесів

Курсова робота

“Мікросервісна система MyUniversity”

Виконав:
студент 4 курсу групи ПМА-41,
спеціальності 124 – “системний аналіз”
Байцар Р. М.
Керівник:
к. т. н., доцент кафедри теорії оптимальних процесів, Мельничин А. В.
Національна шкала_____
Кількість балів:___ Оцінка ECTS___

Члени комісії

_____ _____
(підпис) (прізвище та ініціали)
_____ _____
(підпис) (прізвище та ініціали)
_____ _____
(підпис) (прізвище та ініціали)

Львів – 2021

ЗМІСТ

ВСТУП

В сучасних реаліях інтернету кількість послуг і можливостей сягає такого об’єму, що звичайному користувачеві стає складно обрати кращий відеохостинг, новинний сайт, платіжну систему, тощо серед великої кількості кращих чи гірших веб ресурсів.

Необхідна стандартизація.

Мені, як студенту університету, що вчиться дистанційно було б зручно мати тестову систему (як e-learning.lnu.edu.ua), журнал з оцінками (як dekanat.lnu.edu.ua), розклад зі сповіщеннями, начальні матеріали та тому подібне в одному місці, та що важливо моя сутність користувача (студента) була єдиною для всіх цих ресурсів.

Завдання полягає в тому щоб об’єднати, стандартизувати всі університетські ресурси та надати максимально приємний та зручний досвід користування.

ОСНОВНІ ВИЗНАЧЕННЯ ТА ТЕРМІНОЛОГІЯ

Оскільки основне завдання, як і в моїй курсовій роботі «проектування та розробка серверної частини web-ресурсу для організації заходів», не змінилося, тобто я знову пишу «back», уся термінологія все ще актуальна і згодиться тут.

Нові терміни:

Мікросервіси — архітектурний стиль, за яким єдиний застосунок будується як сукупність невеличких сервісів, кожен з яких працює у своєму власному процесі та спілкується з рештою, використовуючи прості та швидкі протоколи передачі даних. Ці сервіси будуються навколо бізнес-потреб і розгортаються незалежно один від одного з використанням зазвичай повністю автоматизованого середовища. Існує абсолютний мінімум централізованого керування цими сервісами. Самі по собі вони можуть бути написані з використанням різних мов програмування і технологій зберігання даних. Мікросервісна архітектура зручна для реалізації процесу безперервної поставки програмного продукту, оскільки кожен мікросервіс є слабо залежним від іншого – це дозволяє працювати з різними версіями одного і того самого мікросервіса.

Брокер повідомлень – це посередник, який дозволяє абстрагуватися від конкретних протоколів джерела на приймача повідомлення та забезпечує безпосередню розсилку повідомлень між сторонами комунікації. Брокери часто розгораються разом з іншими мікросервісами в тому ж середовищі, тому ззовні вони схожі на ті ж мікросервіси, проте це, як правило, готові програмні рішення.

Виклик віддалених процедур (англ. Remote procedure call, RPC) – протокол, що дозволяє програмі, запущеній на одному комп'ютері, звертатись до функцій програми, що виконується на іншому комп'ютері, подібно до того, як програма звертається до власних локальних функцій.

Деякі терміни з попередньої роботи:

Сервер – сервером може виступати будь який комп'ютер який виконує роботу, команда на виконання якої, прийшла ззовні. Такий комп'ютер має публічне ім'я в інтернеті (звичне нам ip, наприклад 127.0.0.1, або красивий адрес www.my-site.com, який насправді також переводиться у машинний номер 0.0.0.0) та встановлений на цей комп'ютер софт що власне і опрацьовує запит який прийшов ззовні.

Клієнт – це програма, що здатна надсилати запит та отримувати відповідь від сервера. Зазвичай володіє деякою бізнес логікою. Наприклад коли ми заходимо на сайт facebook – ми завантажуємо програму клієнт прямо у свій браузер, але кожен клік по самому сайту – це запита до сервера і як результат ми бачимо стрічку новин або фотоальбом.

Бізнес логіка – формально це просто набір методів та функцій що розміщені в програмі.

.NET Core – це платформа, розроблена компанією Майкрософт, що дозволяє запускати, створений з її використанням, програмний продукт на операційній системі Windows, Linux або macOS.

ASP.Net core – програмне забезпечення (framework), що являє собою каркас веб застосунка. Розробляється компанією Майкрософт, але знаходиться у вільному доступі на github де кожен може долучитися до розробки.

Web API – принцип побудови веб аплікації, що повністю розбиває клієнтську на серверну частину на дві незалежні програми які спілкуються між собою через так звані endpoints (ендпоінти) за допомогою http протоколу.

Endpoint – кінцева точка зв'язку. Власне це і є той інтерфейс, що забезпечує зв'язок клієнта і сервера.

Http – протокол передачі даних між клієнтом і сервером. Оскільки дві програми є незалежними один від одного – запит що йде до сервера має

бути певним чином оформлений. Http складається з двох частин Header і Body. У хедері розміщена інформація про запит та його тип (Get, Post, Put…), в тілі інформація яку ми відправляємо з клієнта (паролі, повідомлення…). Відповідь сервера характеризується ще наявністю статусного коду:

2xx – запит оброблено правильно.

3xx – переадресація (зустрічаються рідко).

4xx – помилка клієнта у формуванні запиту.

5xx – помилка сервера в обробці запиту (неопрацьований exception у самій програмі).

Git – система контролю версій що дозволяє фіксувати кожен крок написання програми та зберігати його на віддаленому репозиторію. Зручний інструмент без якого не можна уявити собі сучасний процес розробки програм.

РОЗДІЛ 1: ПОСТАНОВКА ЗАДАЧІ

1.1. Вибір назви

Оскільки одна з цілей цього проекту – абстрагуватися від конкретного університету, тобто реалізувати систему організацій (Tenant system), назву потрібно обирати не прив’язуючись до конкретного університету, але щоб кожен користувач розумів, до якого університету він належить.

MyUniversity – проста назва, що чітко і однозначно описує ідею проекту.

1.2. Постановка задачі

Розробити єдину систему для організації університетської освіти, що враховує всі аспекти діяльності вишів та об’єднує всі університети в єдину систему. Попри те жоден університет ніяк не впливає на інший, та навіть не підозрює про його існування. Головним постачальним даного продукту може виступити умовне Міністерство освіти, оскільки воно зацікавлене в такого роду стандартизації, та збору інформації що усіх вишів централізованим шляхом.

Як Студент я хочу:

- систему для тестування (як e-learning.lnu.edu.ua)
- електронний журнал (як dekanat.lnu.edu.ua)
- електронний план навчання та навчальні матеріали
- афіша новин та подій мого університету
- зручна навігація між підсистемами

Як викладач я хочу:

- систему тестування для студентів
- електронний журнал успішності моїх студентів

- план навчання, який я можу редагувати, оновлювати
- можливість розсилки новин видимих різним, зацікавленим, підгрупам користувачів (цілій кафедрі, студентській групі, чи конкретним користувачам)
- створювати студентів, студентські групи та керувати ними

Як менеджер університету я хочу:

- керувати всіма системами свого університету
- створювати нових користувачів, керувати існуючими

Як головний адміністратор я хочу:

- керувати всіма існуючими університетами
- переглядати статистику успішності студентів різних університетів, різних факультетів, спеціальностей, груп
- керувати всіма існуючими користувачами

Користувач може мати декілька ролей (наприклад менеджер університету є також викладачем)

1.3. Вибір бази даних

Кожен мікросервіс має власну спеціалізацію (домен), відповідно власну базу даних якою він користується. В даному проекті я використовую EntityFramework з його зручними міграціями та підходом Code First (коли програма – користувач декларує вигляд бази даних).

1.4. Вибір архітектури

Останні роки коли питання стандартизації звучить часто, високою популярністю користується контейнеризація програмних продуктів та Kubernetes – побудова аплікацій за мікросервісним підходом вважається хорошою практикою. Такий підхід дає широкий список переваг:

- високий рівень незалежності: незалежна розробка, незалежне розгортання
- незалежне масштабування (кількість мікросервісів на які йде велике навантаження можна збільшити)
- невелика кодова база кожного мікросервіса
- простота заміни однієї реалізації мікросервісу іншою
- простота додавання нового функціоналу в систему
- ефективне використання ресурсів (мікросервіс що використовується не часто, не займає цінну оперативну пам’ять та процесор)
- Еластичність: вихід з ладу одного сервісу зазвичай не призводить до виходу з ладу всієї системи (якщо електронний журнал тимчасово не працює – це не заважає мені написати тест, проте виникає проблема: як записати результат мого тесту в журнал?)
- Кожен мікросервіс незалежно від інших може бути реалізований за допомогою будь-якої мови програмування, СУБД, та ін.
- Архітектурно побудовані за симетричним принципом (виробник-споживач)

РОЗДІЛ 2: МІКРОСЕРВІСНА АРХІТЕКТУРА

2.1 Опис підходу

При побудові системи такого масштабу варто дотримуватися деяких правил, що значно спробують процес розробки та обслуговування системи:

- Організаційна культура повинна охоплювати автоматизацію розгортання та тестування (автоматизувати розгортання дозволяє kubernetes та налаштовані процеси CI/CD (зазвичай таким займаються DevOps інженери), розгортати нові версії на окремих кластерах для розробників та тестувальників). Наприклад коли розробник написав нову версію якогось мікросервісу та зафіксував зміну в git – піде автоматичний процес розгортання на dev.my-university.edu.ua та test.my-university.edu.ua. Сумісність нової версії його мікросервісу з існуючими повинна забезпечуватися безпосередньо розробником
- Культура і принципи проектування повинні реалізувати перехоплення та опрацювання відмов і дефектів та перебоїв у середовищі виконання. Тобто якщо користувацький запит пройшов через декілька мікросервісів поспіль і один з них зіткнувся з помилкою – користувач повинен цю помилку побачити.
- Кожен мікросервіс гнучкий, стійкий до відмов, легко компонується з іншими мікросервісами, функціонально мінімальний та закінчений. Тобто кожен мікросервіс може делегувати частину роботи іншому мікросервісу.

2.2 Структура проекту

Перш ніж почати безпосередню імплементацію технічний вимог потрібно схематично зобразити заплановану систему, на яку можна опиратися в процесі розробки.

В моєму випадку схема має вигляд:

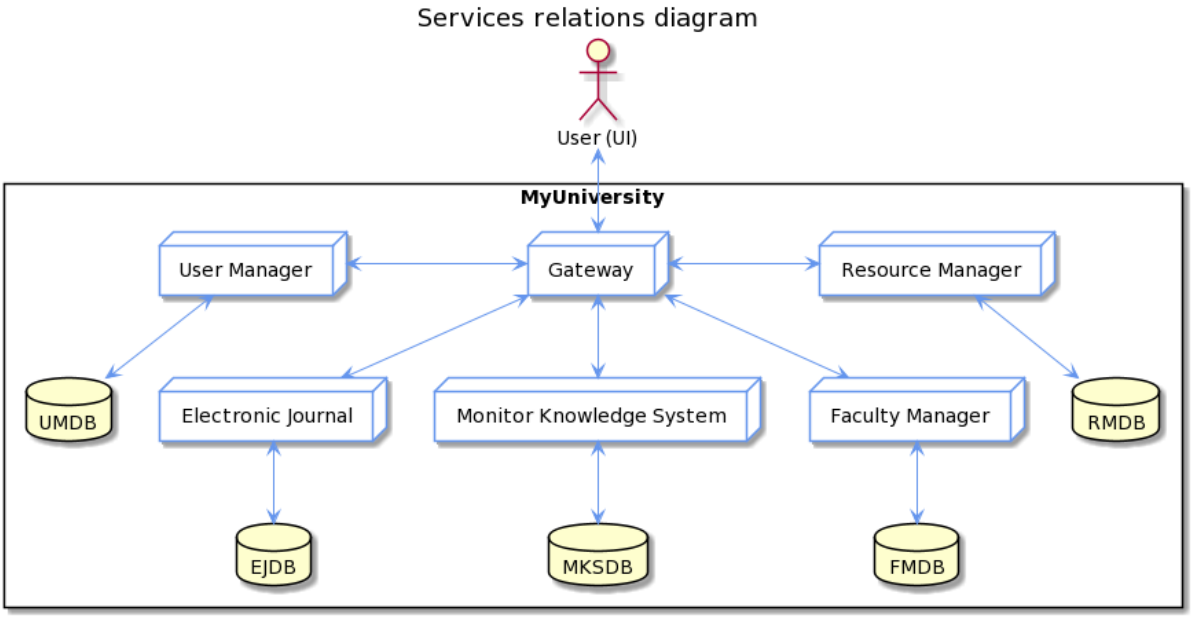

Рис. 2.1. Схематичне зображення системи MyUniversity

На схемі присутні основні логічні блоки системи, а саме:

Gateway – окремий мікросервіс єдина роль якого – приймати HTTP запити користувача (робити первинну валідацію вхідних моделей, перевіряти доступ користувача) та перенаправляти їх через засоби внутрішньої комунікації на мікросервіс який безпосередньо здійснить обробку запиту. Gateway – єдиний видимий ззовні мікросервіс, тому вважається вхідною точкою у всю систему.

User Manager – мікросервіс головна роль якого – керувати користувачами. Оскільки користувач належить до певного університету (tenant) та має певні ролі в системі, то інформація про ці університети та ролі також розміщена тут.

Resource Manager – мікросервіс для реєстрації ресурсів у системі.

Будь яка сутність (окрім сутностей User Manager’а) повинна мати область видимості. Наприклад журнал успішності певної дисципліни, певної групи студентів повинен бути видимий для студентів, що записані в ньому (read access), викладачу що веде предмет у цій дисципліні (full access) та університетському адміністратору, оскільки той має доступ до всіх ресурсів свого університету. Недоліком даного підходу є наслідування кожною сутністю інтерфейсу IResource, це сповільнить час виконання запиту, система постійно запитуватиме себе “чи має користувач доступ до цього ресурсу?”, “до яких ресурсів користувач має доступ?”.

Electronic Journal - електронний журнал успішності студентів.

Monitor Knowledge System – система для проведення тестування. Хорошим прикладом здешевлення проекту може слугувати інтеграція e-learning.

Faculty Manager – мікросервіс для відображення життєдіяльності факультетів, кафедр, управління студентських груп, поширенням наукових та навчальних матеріалів, тощо.

Кожен мікросервіс зберігає свій стан у власній базі даних, доступ до якої має лише він та його клони (наприклад під час сесії є логічним збільшити кількість сутностей MKS щоб кожен студент міг безперешкодно здати свій екзамен). Якщо, наприклад, Electronic Journal потрібно дізнатися інформацію, що міститься в базі даних Resource Manager’а, він повинен робити це через API того ж Resource Manager’а.

Уся комунікація між сервісами здійснюється через меседж брокера.

2.3 Проблема підходу

Попри ряд переваг, що дає такий підхід побудови аплікацій, яскраво виражається ряд недоліків, а саме:

- Мікросервіси успадковують усі проблеми розподілених систем (складність розподілених транзакцій, остаточна узгодженість, CAP теорема).
- Значні накладні витрати на інфраструктуру, моніторинг і операційні дії. Тобто витрати на Cloud провайдера, система логування, тощо.
- Ускладнене налагодження, відслідковування помилок в робочому сервісі, трасування. Grafana (Prometheus, loki, jaeger) частково усувають проблему.
- Обмеження типу «одна команда — один сервіс» викликає бар'єри: коли одна команда для розробки свого сервісу заблокована відсутністю необхідного їм функціоналу сервісу що розробляється іншою командою. У моєму прикладі проблема відсутня, оскільки я єдиний розробник.
- Незалежність сервісів призводить до дублювання коду (утиліти, робота з БД, об'єкти транспортування даних, тощо). В середовищі .NET частково нівелюється винесенням повторюваного коду в NuGet package.
- Проблеми зі стабільністю мережевого зв'язку між сервісами, мережеві затримки. Завжди є мала ймовірність, що відмовить сам меседж брокер і повідомлення буде втрачене.
- Ускладнене тестування і розгортання. Автоматизація і досвідчений DevOps частково нівелюють дану проблему.
- Ускладнене забезпечення безпеки, пов’язане зі складністю тестування системи.

РОЗДІЛ 3: МЕСЕДЖ БРОКЕР

3.1. Навіщо потрібен меседж брокер

Брокер повідомлень (англ. Message broker) - архітектурний підхід в розподілених системах типу мікросервісів, що дозволяє обмінюватися повідомленнями двом сторонам, абстрагуючись від їх реалізацій. Таким чином мікросервіс написаний на Python легко взаємодіє з мікросервісом на C++, важливо лише щоб поширити протокол на них обох (наприклад модель користувача має бути відомою обом). Крім перетворення повідомлень з одного формату в інший, в завдання брокера повідомлень також входить:

- перевірка повідомлення на помилки
- маршрутизація повідомлення між конкретними приймачами
- збереження повідомлень в базі даних (гарантує успішне пересилання повідомлення)
- виклик веб-сервісів
- поширення повідомлень передплатникам, якщо використовуються шаблони типу видавець-передплатник (publisher - subscriber патерн)

Використання брокерів повідомлень дозволяє розвантажити веб-сервіси в розподіленої системі, так як при відправці повідомлень їм не потрібно витрачати час на маршрутизацію, пошуку приймачів, тощо. Крім того, брокер повідомлень для підвищення ефективності може реалізовувати стратегії впорядкованої розсилки і визначення пріоритетності, балансувати навантаження та інше.

Популярними представниками меседж брокерів є Kafka, RabbitMQ, ActiveMQ, тощо.

3.2. Проблема вибору брокера для MyUniversity. Метод аналізу ієрархій

Під час вивчення дисципліни «системний аналіз» однією з проектних робіт було вивчення та застосування «методу аналізу ієрархій».

Командою з п’яти експертів, а саме: Байцар Ростислав, Довган Олена, Опацький Микола, Карпій Володимир, Цупанич Олександр, було оцінено 5 можливих альтернатив, які порівнювалися незалежно та окремо кожним експертом на основі 6 найважливіших критеріїв:

Рівень 1: вибрати меседж брокет для проекту мікросервісної архітектури					
Рівень 2: критерії					
Q1 максимально можливе навантаження		Q2 підтримка патерну publisher/subscriber		Q3 Підтримка RPC	
Q4 Підтримка стрімінгу		Q5 надійність (чи дійде повідомлення до адресанта)		Q6 Ціна ($ і складність використання)	
Рівень 3: альтернативи					
A RabbitMQ	B Kafka	C Azure service bus	D Amazon SQS	E gRPC	

Рис. 3.1. Постановка задачі MAI

В кінцевому результаті за результатом аграгованої думки (вага кожного експерта рівна) було прийнято рішення вибрати gRPC як меседж брокер для даного проекту.

0,184	RabbitMQ
0,230	Kafka
0,150	Azure service bus
0,155	Amazon SQS
0,281	gRPC

Рис. 3.2. Результат прийняття рішення за допомогою MAI

3.3. gRPC

gRPC – це система віддаленого виклику процедур із відкритим кодом (RPC), розробляється компанією Google з 2015 році. Вона використовує HTTP2 для транспорту, буфери протоколів як мову опису інтерфейсу та забезпечує функції такі як аутентифікація, двосторонній стрімінг та контроль потоку , блокування чи неблокування прив'язок, а також скасування та очікування (cancellation та timeouts). Він генерує прив’язки міжплатформенних клієнтів та серверів для багатьох мов програмування. Найбільш поширений для написання мікросервісних систем або підключення клієнтів мобільних пристроїв до серверних сервісів.

Комплексне використання HTTP2 в gRPC унеможливлює реалізацію клієнта gRPC у браузері, замість цього потрібен проксі роль якого виконує мікросервіс Gateway.

Декларація і поширення gRPC ендпоінтів та моделей здійснюється за допомогою protobuf файлів.

Protocol Buffers (protobuf) — формат серіалізації даних, запропонований корпорацією Google, як альтернатива XML. Зручний протокол, що в поєднанні з gRPC бібліотекою здійснює багато рутинної генерації коду. Можливо не нейкращий варіант декларації моделей для середовища .NET, але швидкість обміну повідомленнями, та зручність використання цього варті.

```
1   syntax = "proto3";
2
3   option csharp_namespace = "MyUniversity.UserManager.Role";
4
5   service Role {
6       rpc GetRoles (RoleRequest) returns (RolesReply);
7   }
8
9   message RoleRequest {}
10
11  message RolesReply {
12      repeated RoleReply Roles = 1;
13  }
14
15  message RoleReply {
16      int32 id = 1;
17      string role = 2;
18  }
```

Рис. 3.3. Приклад protobuf декларації деяких ресурсів в UserManager

РОЗДІЛ 4: КОРИСТУВАЧІ ТА МОЖЛИВОСТІ

4.1. Ролі

За замовчуванням в системі існує 5 ролей:

- SuperAdmin – головний адміністратор всієї системи, що має доступ до будь яких ресурсі. Єдиний хто може реєструвати нові університети, та не зобов’язаний належати до якого небуть з них.
- Service – зарезервована службова роль, має ті ж права що й SuperAdmin але використовується для сервіс – акаунтів, які в майбутньому можуть здійснювати якісь дії без участі користувача. Наприклад активація запланованого тесту з математики в Monitor Knowledge System.
- UniversityAdmin – адміністратор власного університету. Має ті ж права що й SuperAdmin, але в межах власного університету. Не може створювати SuperAdmin чи Service акаунти.
- Teacher – безпосередній користувач що працює зі студентами та керує навчальним процесом. Може керувати, створювати та редагувати студентів.
- Student – користувач системи, що споживає надану для нього викладачем інформацію. Може редагувати свою власну сутність.

4.2. Проблема масштабованості рольової системи

Між ролями існує неявна ієрархія: SuperAdmin > Service > UniversityAdmin > Teacher > Student, але вона не виражена явно і змінюється залежно від ситуації та місця використання. Система завжди дивиться на найвищу роль користувача і в своїй поведінці відштовхується від неї. Таким чином єдиний ендпоінт “Get /api/roles” поверне різні значення для UniversityAdmin та Teacher, а для Student взагалі відмовить в доступі.

Створення нової ролі потребує безпосереднього втручання розробника та оновлення тіла бізнес логіки. Навіть попри те що встановлення дозволів користувача винесено в окремий локальний сервіс IPermissionResolver.

Дизайн системи не розрахований на створення нових ролей, оскільки потреби в цьому немає, але це сильно збільшує ціну створення нових ролей в майбутньому, якщо в цьому буде потреба.

РОЗДІЛ 5: БАЗА ДАНИХ

5.1. Вибір бази даних

Будь яку сутність, що відображає реальний світ (або частину реального світу) можна програмно описати за та відобразити за допомогою реляційної моделі бази даних, не зважаючи наскільки ця сутність складна.

Вибір SQL бази даних був для мене очевидним, оскільки популярність такого підходу не спадає десятиліттями, а кількість якісних готових програмних рішень надзвичайно велика.

Враховуючи мій досвід роботи з нереляційними базами даними (mongodb) – перевага у вигляді документного, деревовидного збереження даних не настільки суттєва. Швидкість і гнучкість запитів мови SQL в десятки разів вища за mongodb. Відсутність чіткої схеми, валідації полів, відсутність інструментів мігрування значно підвищує ціну обслуговування такої бази даних. Це може призвести до того, що з часом в системі з'являться об'єкти не валідної форми з якими неможливо працювати. Це, в свою чергу, призведе до виходу з ладу всієї системі і систем залежних від неї.

5.2. Деталі реалізації

Оскільки я розробляю орієнтовану на орендаря (Tenant specific) аплікацію – необхідність розділяти дані різних університетів стоїть особливо гостро. Тому кожна сутність, яку є необхідність чітко розділяти між орендарями наслідує інтерфейс **ITenantSpecificEntity**. Це означає кожна така сутність в базі даних має поле TenantId, таким чином дані різних університетів можна чітко і однозначно розділяти між користувачами різних університетів. Такий розділ відбувається на рівні бізнес логіки і

розробнику потрібно докладати більших зусиль для забезпечення безпеки і конфіденційності даних.

Бувають ситуації коли деякі дані необхідно видаляти, але в залежності від рівня доступу поведінка видалення може відрізнятися. Інтерфейс **ISoftDeletableEntity** реалізує принцип м’якого видалення. Користувачі, наприклад викладачі, можуть помилково створювати студентів, яких вони пізніше видаляють і дня них стають невидимими. Проте SuperAdmin може бачити м’яко видалені сутності, відновлювати їх, або повністю видаляти з системи. Сутність можна видалити повністю лише якщо вона була м’яко видалена.

Інтерфейс **IResourceEntity** – надає сутності властивість ресурсу, видимість якого встановлюється через ресурс менеджер. Проблемою цього рішення є те, що така сутність унаслідується від сутності, що розміщена не просто в окремій таблиці, а в окремому мікросервісі. Це призводить до збільшення часу очікування користувача (на практиці лише долі секунд) та до проблеми коли ResourceManager тимчасово недоступний, тоді користувач не може отримати свою відповідь.

ВИКОРИСТАНІ ТЕХНОЛОГІЇ

Git – найважливіша складова написання будь якого програмного забезпечення це система контролю версій. Вона дозволяє зберігати зроблений прогрес на віддаленому сервері без страху втратити написаний код. Також git дозволяє швидко відкотити зроблені зміни якщо вони виявилися не робочими або містять іншу помилку.

Trello – організація проекту такого масштабу є не простим завданням. На допомогу приходить Trello де можна розбити процес розробки на окремі завдання та виставляючи порядок їх виконання. Не потрібно тримати все в голові. Це допомагає на стадії проектування. Моя дошка складається з колонок to do, in progress, done а також окрема колонка для багів які були виявлені в процесі тестування.

Microsoft Visual Studio 2019 – потужне IDE для розробки програмних продуктів на базі .NET. Версія 2019 є необхідною для використання .NET Core 3.0 і вище.

.NET 5 – наступний крок в екосистемі Microsoft. Наслідує всі можливості .NET Core 3.1, але несе в собі ідею об’єднати .NET Core та .NET Framework.

Entity Framework Core – ORM технологія розроблена Microsoft. Дозволяє робити відображення бази даних з її таблицями і користуватися ними наче звичайними колекціями C#. У деякій мірі це є реалізацією патерну репозиторій який розміщений на Data access layer.

LINQ to entity – проміжна мова запитів дозволяє будувати схожі до SQL скриптів команди які відправившись до сервера бази даних перетворюються у SQL команду, якщо її неможливо виконати то повернуться усі необхідні дані для того щоб виконати її вже на стороні програми засобами C#.

Swagger – це програмне забезпечення що допомагає в процесі розробки імітуючи програму – клієнта. На основі написаних моделей та точок входу swagger допомагає будувати http запити та тестувати поведінку програми.

AutoMapper – бібліотека, що дозволяє перетворювати об’єкти одного типу в об’єкти іншого типу тим самим створюючи мости передачі даних з одного рівня на інший.

Migrations – є компонентом Entity Framework дозволяючи будувати базу даних автоматично при першому запуску програми, спрощуючи роботу програміста і нівелюючи ймовірність зробити помилку в процесі створення бази даних.

ВИСНОВКИ

В безладі всесвітнього павутиння, звичайному користувачеві, легко заблудитися. Необхідно об’єднувати системи та сервіси в мережі та екосистеми, щоб покращувати якість сучасного софту та полегшити життя користувача.

Світові гіганти такі як Google, Microsoft, Amazon вже зробили це. Їх продукції користується шаленою популярністю та приносить значні прибутки компаніям.

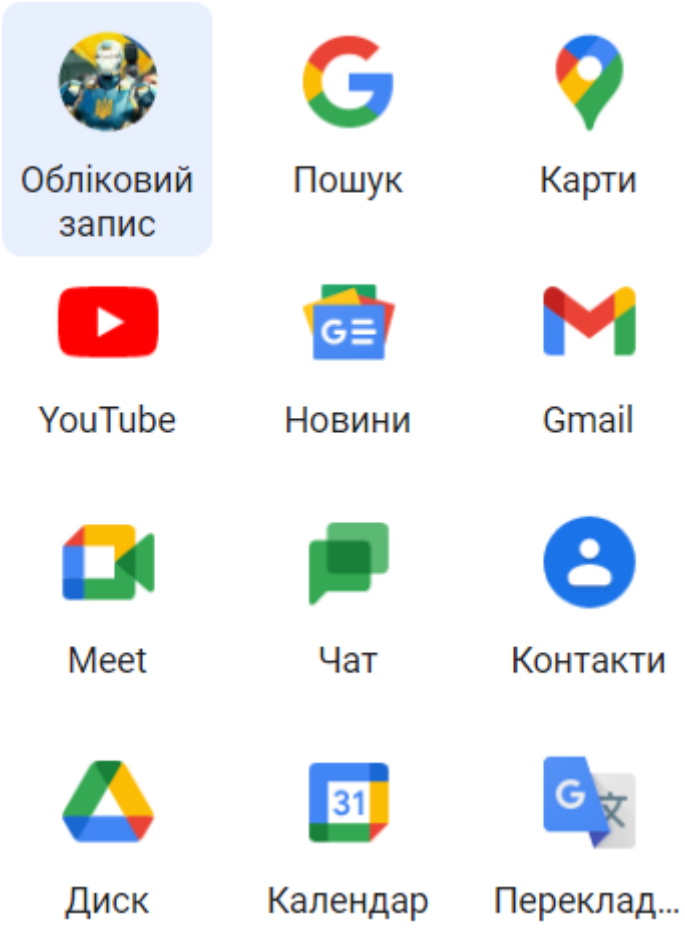

Рис. 6.1. Приклад екосистеми Google: всі сервіси об’єднані в одному зручному вікні та зв’язані єдиним обліковим записом

Використання сучасних практик необхідне для створення якісного продукту. Мікросервісний підхід дозволяє реалізувати гігантських розмірів ідеї, дати нове життя старим забутим продуктам та принести власникам

високий дохід в коротко і довгостроковій перспективі, скільки не обов’язково чекати повного релізу всіх задуманих проектів, а можна випускати їх один за одним по мірі їх завершеності.

Вихідний код зробленої роботи знаходиться у візьному використанні за посиланнями:

https://github.com/Rostik18/MyUniversity.Gateway

https://github.com/Rostik18/MyUniversity.UserManager

ДЖЕРЕЛА ТА ЛІТЕРАТУРА

1. Власний досвід розробки мікросервісних систем в компанії Levi9 [Електронний ресурс] - режим доступу URL: https://www.levi9.com/
2. Pro C# 7 - With .NET and .NET Core, Andrew Troelsen, Eighth Edition.
3. CLR via C#, Jeffrey Richter, fourth edition.
4. Мікросервіси, стаття Володимира Хорікова [Електронний ресурс] – режим доступу URL: https://habr.com/ru/post/249183/
5. Розуміння меседж брокерів, стаття [Електронний ресурс] – режим доступу URL: https://habr.com/ru/post/466385/
6. Entity Framework Core [Електронний ресурс] – режим доступу URL: https://habr.com/ru/company/otus/blog/500012/
7. Офіційна документація Майкрософт [Електронний ресурс] – режим доступу URL : https://docs.microsoft.com/ru-ru/aspnet/core/web-api/?view=aspnetcore-5.0