МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХЕРСОНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ
Факультет комп'ютерних наук, фізики та математики
Кафедра інформатики, програмної інженерії та економічної кібернетики

РОЗРОБЛЕННЯ СЕРВЕРНОГО API З АРХІТЕКТУРОЮ МІКРОСЕРВІСІВ ДЛЯ СИСТЕМИ БРОНЮВАННЯ

Кваліфікаційна робота (проєкт)

на здобуття ступеня вищої освіти «магістр»

Виконав: студент 2 курсу
Спеціальності 121 Інженерія програмного забезпечення
Освітньо-професійної програми «Інженерія програмного забезпечення» другого (магістерського) рівня вищої освіти Альохін Антон Вікторович
Керівник доктор педагогічних наук, професор Круглик Владислав Сергійович
Рецензент кандидат фізико-математиних наук, доцент Котова Ольга Володимирівна

Херсон – 2020

ЗМІСТ

ВСТУП

Актуальність дослідження.

Проблема проектування та створення якісного програмного забезпечення є надзвичайно важливою у сучасному інформаційному світі. З розвитком IT індустрії було знайдено багато різних підходів та концепцій до побудови складних програмних систем [31]. Показником гарно побудованої програми є, звісно, її архітектура, яка правильно описує предметну область та є формальною моделлю системи. Архітектурою можна вважати набір певних структурних компонентів зв’язаних між собою, які задають поведінку всієї системи. Основною задачею архітектури є управління складністю, елегантне та доцільне відображення предметної області. Довгий час провідне місце займала так звана “монолітна архітектура”. При даному підході вся система являє собою моноліт, який фізично розташовується на єдиній машині, запускається в одному процесі та виконує всі бізнес-операції системи. Монолітний додаток піддається лише горизонтальному масштабуванню шляхом запуску декількох окремих серверів із кожним окремим монолітом [34].

Але з плином часу знаходилися інші ідеї та підходи, саме таким стала мікросервісна архітектура. За допомогою мікросервісів ви можете розбити великі системи на менші компоненти, щоб відновити контроль над архітектурою. Ви можете впровадити невеликі зміни та додати окремі функції за допомогою мікросервісів, розширивши один або набір компонентів та розгорнувши їх. Завдяки архітектурі мікросервісів, ваш додаток буде запускати різні сервіси як незалежні блоки, кожен із власним середовищем виконання, кодовою базою, потоками тощо.

Деякі з найбільш інноваційних та найвигідніших підприємств у світі - такі як Amazon, Netflix, Uber та Etsy - пояснюють величезний успіх своїх IT-ініціатив, зокрема, впровадженням мікросервісів. З часом ці

підприємства демонтували свої монолітні програми та переробили їх на архітектури на основі мікропослуг. Це допомогло швидко досягти масштабних переваг, більшої спритності бізнесу та немислимих прибутків.

Об’єкт дослідження – мікросервісна архітектурна програмного забезпечення.

Предмет дослідження – серверний API з мікросервісною архітектурою.

Мета дослідження – реалізація серверного API з архітектурою мікросервісів для системи бронювання.

У зв’язку з поставленою метою було визначено **завдання дослідження:**

1. Виконати огляд та аналіз мікросервісів як сучасного рішення для архітектури ПЗ.
2. Розглянути особливості застосування технології .NET Core для реалізації мікросервісної архітектури.
3. Проаналізувати та описати вимоги до сервісу системи бронювання.
4. Спроєктувати мікросервіси та бази даних системи.
5. Розробити серверний API з мікросервісною архітектурою для системи бронювання.

Методи дослідження. Для вирішення поставлених завдань були використані наступні методи:

- теоретичні: аналіз, синтез, систематизація, зіставлення, класифікація наукових джерел інформації з програмування веб додатків;
- практичні: проєктування та створення додатку здійснювалось за допомогою: .NET Core, ASP.NET Core, Entity Framework, PostgreSQL.

Зв'язок роботи з науковими програмами, планами, темами. Кваліфікаційна робота пов’язана з напрямом наукових досліджень кафедри інформатики, програмної інженерії та економічної кібернетики

факультету комп’ютерних наук, фізики та математики Херсонського державного університету.

Наукова новизна одержаних результатів. Результати кваліфікаційної роботи є новими і полягають у використанні технології .NET Core для розробки мікросервісної архітектури системи бронювання басейну університету.

Практична цінність роботи полягає у тому, що представлені результати дозволяють застосовувати розроблену сервісну частину додатку для забезпечення функціонування системи бронювання басейну Херсонського державного університету.

Апробація результатів роботи. Основні положення та результати роботи були обговорені на засіданні кафедри інформатики, програмної інженерії та економічної кібернетики. Також здійснено їх оприлюднення шляхом публікації у збірнику наукових праць.

Структура роботи. Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків та додатків.

РОЗДІЛ 1
МІКРОСЕРВІСИ ЯК СУЧАСНА АРХІТЕКТУРА СТВОРЕННЯ ДОДАТКІВ

1.1 Генезис архітектури програмного забезпечення

Архітектура програмного забезпечення – це спосіб структурування програмної або обчислювальної системи [26], абстракція елементів системи на певній фазі її роботи. Система може складатись з кількох рівнів абстракції і мати багато фаз роботи, кожна з яких може мати окрему архітектуру.

Також, архітектура – це набір значущих рішень з приводу організації системи програмного забезпечення, набір структурних елементів і їх інтерфейсів, за допомогою яких компонується система, разом з їх поведінкою, обумовленим у взаємодії між цими елементами, компонування елементів в поступово укрупнюються підсистеми, а також стиль архітектури який направляє цю організацію - елементи та їх інтерфейси, взаємодії і компонування [16].

Дослідження архітектури програмного забезпечення намагається визначити як краще розбити систему на частини, як ці частини визначають та як взаємодіють один з одним, як між ними передається інформація, як ці частини розвиваються по одному і як все описане найкраще записати, використовуючи формальну чи неформальну нотацію [7]. Архітектуру необхідно будувати, щоб вона найкраще відповідала вимогам до системи, що створюється, згідно до принципу "Form follows function".

Терміном "Архітектура" також називають документування архітектури програмного забезпечення. Документування архітектури спрощує процес комунікації між зацікавленими особами та командою розробки, дозволяє на ранніх етапах проектування зафіксувати прийняті рішення про дизайн системи високого рівня. Також дозволяє

використовувати компоненти цього дизайну, шаблони проектування та інші розробки повторно в інших проектах [8].

Область комп'ютерних наук з моменту свого існування і створення зіткнулася з проблемами, що пов'язані зі складністю та багатокомпонентністю програмних систем. Раніше проблеми складності вирішувалися розробниками шляхом правильного вибору структур даних, розробки алгоритмів і розмежування повноважень [17]. Хоча термін "архітектура програмного забезпечення" і є відносно новим для індустрії розробки ПЗ, фундаментальні принципи цієї області невпорядковано застосовувалися першими в цій області розробниками ПЗ починаючи з середини 1980-х років. Перші спроби зрозуміти і пояснити програмного архітектуру системи мали проблеми, зв’язані з неточностями і страждали від нестачі організованості. Часто обмежувалися прикладом діаграми з блоків, що з'єднані лініями. У 1990-ті роки спостерігається спроба визначити і систематизувати основні аспекти даної дисципліни. Початковий набір шаблонів проектування, стилів проектування, передового досвіду, мов опису і формальна логіка були розроблені протягом цього часу [21].

Основною ідеєю дисципліни програмної архітектури є ідея зниження складності системи шляхом її абстракції і розмежування повноважень. На сьогоднішній день немає єдиного чіткого визначення терміну архітектура програмного забезпечення.

Початок архітектурі програмного забезпечення як концепції було покладено в науково-дослідницькій роботі Едсгера Дейкстри в 1968 році і Девіда Парнаса на початку 1970-х. Популярність вивчення цієї області зросла з початку 1990-х років разом з науково-дослідницькою роботою по дослідженню архітектурних стилів шаблонів, мов описання архітектури, документування архітектури і формальних методів [38].

Науково-дослідницькі установи відішрають важливу роль у розвитку дисципліни архітектури ПЗ. Мері Шоу і Девід Герлан з університету

Карнегі-Меллон написали книгу під назвою "Архітектура програмного забезпечення: перспективи нової дисципліни" в 1996 році, в якій описали концепції архітектури програмного забезпечення, такі як компоненти, коннектори, стилі і так далі. У Каліфорнійському університеті в Ірвайні, інститут по дослідженню ПЗ насамперед досліджує архітектурні стилі, мови опису архітектури і динамічні архітектури [37].

Першим стандартом опису програмної архітектури став IEEE 1471[en]: ANSI / IEEE 1471—2000: Рекомендації по опису переважно програмних систем. Стандарт був прийнятий в 2007 році, під назвою ISO ISO / IEC 42010:2007.

У той час як в IEEE 1471 архітектура програмного забезпечення представляла архітектуру «програмно-інтенсивних систем», які визначаються як «будь-яка система, в якій програмне забезпечення робить істотний вплив на проектування, побудова, розгортання і розвиток системи в цілому», видання 2011 р йде ще далі, включаючи визначення системи ISO / IEC 15288 та ISO / IEC 12207, які охоплюють не тільки апаратне і програмне забезпечення, але також «людей, процеси, процедури, засоби, матеріали та природні об'єкти». Це відображає взаємозв'язок між програмної архітектури, архітектури підприємства і архітектури рішення.

Проектування архітектури ПЗ – це процес розробки, що виконується після етапу аналізу і формулювання вимог. Завдання проектування архітектури – перетворення системних вимог у вимоги до ПЗ та побудувати на їхній основі архітектуру системи [39]. Побудова архітектури системи здійснюється шляхом визначення цілей системи, її вхідних і вихідних даних, декомпозиції системи на підсистеми, компоненти або модулі та розроблення її загальної структури [30].

Проектування архітектури системи може проводитися різними методами: стандартизованим, об’єктно-орієнтованим, компонентним та ін, кожний з яких пропонує свій шлях побудови архітектури, а саме,

визначення концептуальної, об’єктної й інших моделей за допомогою відповідних конструктивних елементів, таких як блок-схеми, графи, структурні діаграми [25]. Розглянемо детальніше поняття архітектурних стилів.

Архітектурний стиль, або архітектурний шаблон – це набір принципів, високорівнева схема, що забезпечує абстрактну інфраструктуру для зібрання систем. Архітектурний стиль покращує секціонування і сприяє повторному використанню дизайну завдяки забезпеченню рішень проблемам, що часто зустрічаються. Архітектурні стилі і шаблони можна розглядати як набір принципів, які формують додаток [30].

Автономна (standalone) архітектура

Додатками можуть бути, як правило, сервісні програми, системні утиліти, текстові та графічні редактори, компілятори, досить прості корпоративні програми. Розвинена корпоративна інформаційна система, як правило, не може складатися з окремих, не пов'язаних між собою компонентів [24].

На кожному робочому місці (комп'ютері) реалізуються всі функції додатків (інтерфейс користувача, бізнес-логіка і управління даними) та використовується однокористувальницький режим роботи системи.

Перевагами даної архітектури є автономність роботи кожного комп'ютера системи та розвинений, зручний інтерфейс користувача.

Недоліки даної архітектури є обмежена обчислювальна потужність, дублювання інформації на різних комп'ютерах, складність її передачі і синхронізації.

Дані системи набули поширення з появою персональних комп'ютерів в 80-х рр. У таких системах можуть функціонувати текстові та графічні редактори, компілятори, досить прості корпоративні програми. Сучасна корпоративна інформаційна система зазвичай не може складатися з окремих, не пов'язаних між собою компонентів.

Клієнт-серверна архітектура описує розподілені системи, що складаються з окремих клієнта і сервера і з'єднує їх мережі. Найпростіша форма системи клієнт-сервер, що називають дворівневою архітектурою – це серверний додаток, до якого безпосередньо звертаються безліч клієнтів [24].

Історично архітектура клієнт-сервер являє собою настільний додаток з графічним UI, обмінюватися даними з сервером бази даних, на якому у формі процедур розташовується основна частина бізнес-логіки, або з виділеним файловим сервером. Якщо розглядати більш узагальнено, архітектурний стиль клієнт-сервер описує відносини між клієнтом і одним або більше серверами, де клієнт ініціює один або більше запитів, очікує відповіді і обробляє їх при отриманні. Зазвичай сервер авторизує користувача і потім проводить обробку, необхідну для отримання результату. Для зв'язку з клієнтом сервер може використовувати широкий діапазон протоколів і форматів даних.

На сьогоднішній день прикладами архітектурного стилю клієнт-сервер можуть бути веб-додатки, що виконуються в Інтернеті або у внутрішніх мережах організацій. Також настільні додатки для операційної системи Microsoft Windows, що виконують доступ до мережевих сервісів даних, додатки, що виконують доступ до віддалених сховищ даних (такі як програми читання електронної пошти, FTP-клієнти і засоби доступу до баз даних) та інструменти та утиліти для роботи з віддаленими системами (такі як засоби управління системою і засоби моніторингу мережі).

До різновидів стилю клієнт-сервер відносяться:

- Системи клієнт-черга-клієнт. Цей підхід дозволяє клієнтам обмінюватися даними з іншими клієнтами через чергу на сервері. Клієнти можуть читати дані з і відправляти дані на сервер, який виступає в ролі простої черги для зберігання даних. Завдяки цьому клієнти можуть розподіляти і синхронізувати файли і відомості. Іноді таку архітектуру

називають пасивною чергою.

- Однорангові (Peer-to-Peer, P2P). Створений на базі клієнт-чергу-клієнт, стиль P2P дозволяє клієнту і серверу обмінюватися ролями з метою розподілу і синхронізації файлів і даних між безліччю клієнтів. Ця схема розширює стиль клієнт-сервер, додаючи множинні відповіді на запити, спільно використовувані дані, виявлення ресурсів і стійкість при видаленні учасників мережі [23].

- Сервери додатків. Спеціалізований архітектурний стиль, при якому додатки і сервіси розміщуються і виконуються на сервері, і тонкий клієнт виконує доступ до них через браузер або спеціальне встановлене на клієнті ПЗ. Прикладом є клієнт, який працює з додатком, що виконується на сервері, через таке середовище як Terminal Services (Служби терміналів).

Основні переваги архітектурного стилю клієнт -сервер:

- Великий рівень безпеки. Всі дані зберігаються на сервері, який зазвичай забезпечує більший контроль безпеки, ніж клієнтські комп'ютери.

- Централізований доступ до даних. Оскільки дані зберігаються лише на сервері, адміністрування доступу до даних набагато простіше, ніж в будь-яких інших архітектурних стилях.

- Простота обслуговування. Ролі та відповідальність обчислювальної системи розподілені між декількома серверами, спілкуються один з одним по мережі. Завдяки цьому клієнт гарантовано залишається необізнаним і не схильним до впливу подій, що відбуваються з сервером (ремонт, оновлення або переміщення).

Товстий клієнт архітектурі клієнт-сервер – це додаток, що забезпечує (на противагу тонкому клієнтові) повну функціональність і незалежність від центрального сервера. Часто сервер в цьому випадку є лише сховищем даних, а вся робота по обробці та поданням цих даних переноситься на машину клієнта.

Товстий клієнт має повну функціональність роботи з даними сервера, забезпечує режим роботи багатьох користувачів, надає можливість роботи навіть при обривах зв'язку з сервером, має можливість підключення до банків даних без використання мережі Інтернет, володіє високою швидкодією.

Однак широкі функціональні можливості "товстого клієнта" часто несумісні з політикою безпеки інформаційної системи і вартість його надмірно висока. При роботі з ним виникають проблеми з віддаленим доступом до даних, що виражаються в складності поновлення даних, узгодження їх з іншими клієнтами і пов'язаної з цим неактуальністю даних.

Тонкий клієнт в комп'ютерних технологіях – це комп'ютер або програма-клієнт в мережах з клієнт-серверної або термінальній архітектурою, де велика частина завдань по обробці інформації перенесена на сервер і права доступу клієнта строго обмежені. Прикладом тонкого клієнта може служити комп'ютер з браузером, який використовується для роботи з веб-додатками [23].

Монолітна архітектура програмного забезпечення полягає в тому, що різні компоненти програми об'єднуються в одну програму на одній платформі. Зазвичай монолітне додаток складається з бази даних, клієнтського призначеного для користувача інтерфейсу і серверного додатка. Всі частини програмного забезпечення уніфіковані, і все його функції управляються в одному місці.

Компоненти монолітного програмного забезпечення взаємопов'язані і взаємозалежні, що допомагає програмному забезпеченню бути самодостатнім. Ця архітектура є традиційним рішенням для створення додатків, але деякі розробники вважають її застарілою.

Основні переваги монолітної архітектури

Спрощена розробка та розгортання. Є багато інструментів, які ви можете інтегрувати для полегшення розробки. Крім того, всі дії

виконуються з одним каталогом, що спрощує розгортання. Завдяки монолітному ядру розробникам не потрібно розгортати зміни або оновлення окремо, оскільки вони можуть зробити це відразу і заощадити багато часу.

Менше наскрізних проблем. Більшість додатків залежать від безлічі міжкомпонентних завдань, таких як контрольні журнали, ведення логів, обмеження швидкості і т. Д. Монолітні додатки набагато легше враховують ці питання завдяки своїй єдиній кодової базі. До цих завдань простіше підключати компоненти, коли все працює в одному додатку.

Краща продуктивність. При правильній збірці монолітні додатки зазвичай більш продуктивні, ніж додатки на основі мікросервісів. Наприклад, з додатком на мікросервісній архітектурі може знадобитися виконати 40 викликів API для 40 різних мікросервісів, щоб завантажити кожен екран, що, призводить до зниження продуктивності. Монолітні додатки, в свою чергу, забезпечують більш швидкий зв'язок між програмними компонентами завдяки загальному коду і пам'яті [23].

Недоліки монолітної архітектури

Кодова база з часом стає громіздкою. З плином часу більшість продуктів продовжують розроблятися і збільшуються в обсязі, а їх структура стає розмитою. Кодова база починає виглядати дійсно громіздко і стає важкою для розуміння і зміни, особливо для нових розробників. Також стає все важче знаходити побічні ефекти і залежності. З ростом кодової бази погіршується якість і перевантажується IDE.

Складно впроваджувати нові технології. Якщо в додаток необхідно додати якусь нову технологію, розробники можуть зіткнутися з перешкодами для на шляху впровадження. Додавання нової технології означає переписування всього програми, що є дорогим і вимагає багато часу.

Обмежена гнучкість. У монолітних додатках кожне невелике оновлення вимагає повного повторного розгортання. Таким чином, всі

розробники повинні чекати, поки це не буде зроблено. Коли кілька команд працюють над одним проектом, гнучкість може бути значно знижена.

Сервісно-орієнтована архітектура (SOA) - це стиль архітектури програмного забезпечення, який передбачає модульне додаток, що складається з дискретних і слабосв’язанних програмних агентів, які виконують конкретні функції [23]. SOA розділяє компоненти за двома основними ролями: постачальник і споживач сервісів. Обидві ці ролі можуть грати програмні агенти. Концепція SOA полягає в наступному: додаток може бути спроектовано і побудовано таким чином, що його модулі легко інтегруються і можуть бути легко використані повторно.

Переваги SOA

Повторне використання сервісів. Через автономної і слабо пов'язаної природи функціональних компонентів в сервіс-орієнтованих додатках ці компоненти можна повторно використовувати в декількох додатках, без впливу на інші сервіси.

Легкість в супроводі. Оскільки кожна служба програмного забезпечення є незалежною одиницею, її легко оновлювати і підтримувати, не зачіпаючи інші служби. Наприклад, великими корпоративними додатками легше управляти, коли вони розбиті на служби.

Більш висока надійність. Служби легше налагоджувати і тестувати, ніж величезні шматки коду, як в монолітах. Це, в свою чергу, робить продукти на основі SOA більш надійними.

Паралельна розробка. Оскільки сервіс-орієнтована архітектура розбита на прошарку, вона підтримує паралелізм в процесі розробки. Незалежні сервіси можуть розроблятися паралельно і бути завершені одночасно.

Недоліки SOA

Складність в управлінні. Основним недоліком сервіс-орієнтованої

архітектури є її складність. Кожен сервіс повинен забезпечувати своєчасну доставку повідомлень. Кількість цих повідомлень може перевищувати мільйон за один раз, що ускладнює управління всіма службами.

Високі інвестиційні витрати. Розробка SOA вимагає значних попередніх інвестицій в людські ресурси, технології та розробку.

Додаткове навантаження. У SOA все вхідні дані перевіряються до того, як один сервіс взаємодіє з іншим сервісом. При використанні декількох сервісів це збільшує час відгуку і знижує загальну продуктивність.

SOA найкраще підходить для складних корпоративних систем, наприклад банківських. Банківську систему надзвичайно складно розділити на мікросервіси. Але монолітний підхід також не годиться для банківської системи, так як одна частина може пошкодити все додаток. Краще рішення - використовувати підхід SOA і організувати складні додатки в ізольовані незалежні сервіси.

Мікросервісна архітектура

Мікросервіси - це тип сервісно-орієнтованої архітектури програмного забезпечення, орієнтований на створення ряду автономних компонентів, складових додаток. На відміну від монолітних додатків, створених як єдине ціле, мікросервісні додатки складаються з декількох незалежних компонентів, які склеєні разом за допомогою API.

Підхід на основі мікросервісів орієнтований головним чином на бізнес-пріоритети і можливості, тоді як монолітний підхід організований навколо технологічних рівнів, призначених для користувача інтерфейсів і баз даних. Мікросервісний підхід став тенденцією в останні роки, так як все більше і більше підприємств стають гнучкими і переходять на DevOps. Мікросервіси додають унікальну споживчу цінність шляхом спрощення систем. Розбиваючи вашу систему або додаток на безліч більш дрібних частин, ви реалізуєте спосіб зменшення дублювання, підвищення

узгодженості та зменшення зв'язку між частинами, що робить ваші загальні частини системи більш зрозумілими, більш масштабованими і більш легкими для зміни.

Переваги мікросервісів

Легко розробляти, тестувати і розгортати. Найбільша перевага мікросервісів перед іншими архітектурами полягає в тому, що невеликі окремі сервіси можуть створюватися, тестуватися і розгортатися незалежно. Оскільки одиниця розгортання невелика, це полегшує і прискорює розробку і реліз [33]. Крім того, реліз однієї одиниці не обмежений випуском інший, яка ще не завершена. І останній плюс тут полягає в тому, що ризики розгортання знижуються в міру того, як розробники релізі частини програмного забезпечення, а не ціле додаток.

Підвищена гнучкість. За допомогою мікросервісів кілька команд можуть працювати над своїми сервісами незалежно і швидко. Кожна окрема частина програми може бути побудована незалежно через ізольованість компонентів мікросервісів. Підвищена гнучкість дозволяє розробникам оновлювати компоненти системи, що не відключаючи додаток. Крім того, гнучкість забезпечує більш безпечний процес розгортання і підвищений час безвідмовної роботи. Нові функції можуть бути додані в міру необхідності, не чекаючи запуску всього програми.

Можливість масштабування по горизонталі. Вертикальне масштабування може бути обмежено пропускною спроможністю кожного сервісу. Але горизонтальне масштабування (створення більшої кількості сервісів в одному пулі) не обмежене і може працювати з мікросервісами динамічно. Крім того, горизонтальне масштабування може бути повністю автоматизовано.

Недоліки мікросервісів

Складність реалізації. Поділ програми на незалежні мікросервіси тягне за собою більше артефактів управління. Цей тип архітектури вимагає ретельного планування, величезних зусиль, командних ресурсів і

навичок.

Проблеми безпеки. У мікросервісному додатку кожна функціональність, яка взаємодіє через API ззовні, збільшує ймовірність атак. Ці атаки можуть відбутися тільки в тому випадку, якщо при створенні програми не будуть виконані належні заходи безпеки.

Різні мови програмування. Можливість вибирати різні мови програмування - це палиця з двома кінцями. Використання різних мов ускладнює розгортання. Крім того, важче перемикати програмістів між етапами розробки, коли кожен сервіс написаний іншою мовою.

Безсерверна архітектура - це підхід застосування хмарних обчислень для створення і запуску додатків і сервісів без необхідності управління інфраструктурою. У безсерверних додатках виконання коду управляється платформою, що дозволяє розробникам розгортати код, не турбуючись про обслуговування та забезпеченні сервера. Додаток все ще працює на серверах, але стороння хмарна служба, така як AWS, несе повну відповідальність за ці сервери. Безсерверна архітектура усуває необхідність в додаткових ресурсах, масштабування додатків, обслуговуванні серверів, а також системах зберігання і баз даних. Безсерверна архітектура включає в себе дві концепції:

- FaaS (Function as a Service - функція як послуга) - модель хмарних обчислень, яка дозволяє розробникам завантажувати частини функціоналу в хмару і дозволяє цим частинам виконуватися незалежно.
- BaaS (Backend as a Service - бекенда як послуга) - модель хмарних обчислень, яка дозволяє розробникам передавати на аутсорсинг аспекти бекенда (управління базою даних, хмарне сховище, хостинг, аутентифікація користувачів).

Переваги безсерверной архітектури

Простота розгортання. У безсерверних додатках розробникам не потрібно турбуватися про інфраструктуру. Це дозволяє їм зосередитися

на самому коді.

Зниження вартості. Перехід до безсерверної архітектурі знижує витрати. Оскільки вам не потрібно обробляти бази даних, деяку логіку і сервери, ви можете не тільки створювати більш якісний код, але і скорочувати витрати.

Покращена масштабованість. Безсерверні додатки можуть обробляти величезну кількість запитів, тоді як традиційні додатки будуть перевантажені раптовим їх збільшенням.

Недоліки безсерверної архітектури

Прив'язка до постачальника. Прив'язка до постачальника описує ситуацію, коли ви надаєте постачальнику повний контроль над своїми операціями. В результаті зміни в бізнес-логікою обмежені, і міграція від одного постачальника до іншого може викликати труднощі.

1.2 Переваги та недоліки мікросервісів

Вибір Мікросервіси існують вже більше десяти років. Навіть незважаючи на це, багато компаній досі не впевнені, чи цей підхід в кінцевому підсумку вийде з ладу або їм потрібно прийняти його, щоб не система стала застарілою.

Архітектура мікросервісів з’явилася для вирішення обмежень монолітних архітектур - масштабування, гнучкості та продуктивності. Крім того, до цього процесу постійно впроваджуються нові інструменти, такі як Docker, Kubernetes, шаблони та методи [34]. У порівнянні з більш монолітними конструкціями, мікросервіси мають такі переваги:

Низький поріг входження

Оскільки мікросервіси мають модульний дизайн, розробники виконують більш цілеспрямовану роль, ніж могли б зробити в організації, яка використовує монолітний додаток.

Кожний сервіс оснащен всім, від зберігання даних до комунікації. Як результат, невеликі команди можуть бути призначені для розробки,

тестування та розгортання одного сервісу.

Цей менший обсяг робить інтернатів нових розробників більш коротким і простим процесом.

Нові розробники можуть розпочати роботу одразу, оскільки їм не потрібно розуміти функцію кожного сервісу, а також те, як пов’язана вся система. Це означає, що мікросервіси можуть допомогти компаніям зменшити витрати на оплату роботи та швидше залучити нових працівників.

Зниження рисків

Розробка, тестування та розгортання монолітних додатків та програм SOA може зайняти багато часу. Це тому, що навіть одна невелика помилка, яка впливає на одну функцію, може затримати розгортання для всієї платформи [11].

На відміну від цього, архітектура мікросервісів дозволяє розробляти та застосовувати кожну функцію незалежно.

Цей поділ зменшує ризик, оскільки відмова одного сервісу не призведе до відмови інших сервісів. Розробники можуть вносити зміни в одний сервіс або відкочувати оновлення помилок, не перерозподіляючи цілу програму.

Гнучке зберігання даних

Однією з найбільших переваг архітектури мікросервісів є те, що вона дозволяє організаціям зберігати дані в декількох місцях.

Навпаки, монолітна архітектура та архітектура SOA обмежують можливості зберігання даних лише в одному місці. Хоча такий підхід чудово працює для невеликих однорідних наборів даних, це не зовсім масштабоване рішення [15].

Коли компанії починають мати справу з більшими наборами даних, і ці дані починають набувати різних характеристик, вони можуть почати стикатися з деякими проблемами. Завдяки архітектурі мікросервісів розробники можуть вільно вибирати тип сховища, який найкраще

відповідає потребам кожної послуги.

Поліглот - використання різних мов програмування

Організації можуть не тільки вибрати різне рішення для зберігання даних для кожного сервісу, але розробники також можуть вибрати мову програмування, яка найкраще підходить для роботи.

Хоча може мати сенс підтримувати уніфікованість речей з однаковою мовою, що використовується повсюдно, правда, різні мови мають різні функції.

Справа в тому, що поліглот мікросервісів надає розробникам свободу вибору, які інструменти найкраще підходять для них, без узгодження основних зусиль з розробки з іншими командами.

Зменшений безлад

Оскільки технології з часом знецінюються, вони з часом замінюються новими інструментами. У монолітній програмі це означає додавання нових інструментів до існуючого стеку.

У довгостроковій перспективі це означає, що монолітні програми можуть стати дуже великими. Роз’єднання монолітного додатка з мікросервісами представляє можливість зменшити розмір вашої кодової бази, видаливши невикористану функціональність [13]..

Швидкість і простота подальшого розгортання спрощують захист програми від невикористаних функціональних можливостей.

Толерантність до відмов та ізоляцію несправностей

Ще однією ключовою перевагою мікросервісів є покращена стійкість до відмов та ізоляція несправностей. Оскільки мікросервіси слабо пов'язані між собою, вони можуть бути надзвичайно стійкими до несправностей. Це означає, що якщо один сервіс вийде з ладу, це не призведе до компрометації інших сервісів у програмі.

Кожен мікросервіс містить усе необхідне для виконання призначеної функції. Монолітні програми існують як єдиний блок коду, де різні модулі тісно пов'язані.

У цьому випадку, якщо одна функція вийде з ладу, це може означати простої всієї системи. Мікросервіси також полегшують пошук та ізоляцію проблем, обмежуючи пошук лише одним модулем і тим самим зменшуючи час роздільної здатності.

Швидкість розгортання

Архітектура мікросервісів може легко узгоджуватися з Agile, DevOps та CI / CD. Оскільки компанії сприймають ці ідеології, що трансформують внутрішню культуру та робочі процеси та автоматизують ручні процеси, швидкість розгортання різко зросте.

Різні команди можуть одночасно працювати над різними модулями, не чекаючи, поки інші групи завершать проекти, перш ніж вони зможуть рухатися вперед.

Їх також легше знаходити, модифікувати та проводити тестування якості. Крім того, невеликі команди можуть одночасно розробляти, тестувати та розгортати кілька модулів, що дозволяє збільшити обсяг виробництва, не наймаючи більше людей.

Масштабованість та гнучкість

Серед ключових переваг мікросервісів є гнучкість та масштабованість, яку може забезпечити цей підхід.

Гнучкість, яку надають мікросервіси, поширюється також на зобов'язання постачальників та технологій. Організації можуть вільно застосовувати стеки нових технологій на одному сервісі, що означає, що ключовим фактором є пошук правильних інструментів для роботи.

Слабо пов'язані мікросервіси дозволяють командам масштабувати кожну послугу горизонтально та незалежно.

Моніторинг безпеки

Сучасні рішення мікросервісів можуть ідентифікувати вразливі місця в усій системі з майже нульовим рівнем помилкових спрацьовувань, які заважають продуктивності.

Окрім того, оскільки кожен сервіс ізольований від інших модулів

програми, легше виявити першопричину проблеми безпеки. Порівняно з монолітними додатками, де все підключено, мікросервіси захищають організації від збитків, спричинених тривалим простоєм [10].

Більше того, така ізоляція запобігає загрозам безпеці компрометувати інші модулі у вашій програмі, що призводить до збільшення продуктивності та економії коштів для організацій.

Незважаючи на безліч переваг мікросервісів, вони не завжди є найкращим вибором для кожного проекту.

Першим кроком у визначенні того, чи правильно підходять мікросервіси, є оцінка проблем, з якими ви, мабуть, зіткнетеся під час трансформації. Звідти вам потрібно буде з'ясувати, чи готова ваша команда з ними. Хоча більша частина процесу розробки спрощується за допомогою мікросервісів, є декілька областей, де мікросервіси насправді можуть викликати нові складності, з чим повязані такі недоліки архітектури:

Складності проектування

По-перше, спілкування між сервісами може бути складним. Додаток може включати десятки, а то й сотні різних сервісів, і всі вони повинні безпечно спілкуватися.

По-друге, налагодження стає складнішим завданням для мікросервісів. Якщо програма, що складається з декількох мікросервісів, і кожен мікросервіс має власний набір логів, відстеження джерела проблеми може бути важким.

І по-третє, хоча модульне тестування може бути простішим за допомогою мікросервісів, інтеграційне тестування - ні. Компоненти розподіляються, і розробники не можуть перевірити всю систему на своїх окремих машинах.

Критична залежність від API.

Кожен мікросервіс має свій власний API, на який програми покладаються, щоб бути послідовними. Хоча ви можете легко вносити

зміни в мікросервіс, не впливаючи на зовнішні системи, що взаємодіють з ним, але якщо ви зміните API, будь-яка програма, що використовує цю мікросервіс, зазнає впливу, якщо зміна не є зворотно сумісною.

Модель архітектури мікросервісів призводить до великої кількості API, що є вкрай важливим для функціонування підприємства, тому управління API стає критично важливим.

Високі витрати

Щоб архітектура мікросервісів працювала для вашого додатку, вам потрібна достатня інфраструктура хостингу з підтримкою безпеки та обслуговування, а також потрібні кваліфіковані команди розробників, які розуміють і керують усіма послугами.

Якщо у вас вже є ці речі, витрати, пов’язані з переходом до мікросервісів, можуть бути нижчими, але більшості підприємств, які в даний час працюють з монолітною архітектурою, доведеться інвестувати в нову інфраструктуру та ресурси розробників, щоб зробити це.

1.3 Огляд технології .NET Core для розробки мікросервісної архітектури

ASP .NET Core сприяє розробникам у створенні міжплатформенних мікросервісів із .NET на різних операційних системах з легким розгортанням у хмарному середовищі.

Існує багато причин, чому ASP .NET Core має певну популярність при розробці додатків - його природа з відкритим кодом добре допомагає у впровадженні високопродуктивних рішень, крос-платформенної допомоги під час виконання, швидшого розвитку, конфігурації середовища на базі хмари тощо.

Основна точка зору, на якій базуються Microservices та ASP .NET Core, цілком порівнянна між собою. Оскільки .NET був одним із перших, хто створював додатки на основі Simple Object Access Protocol, його методологія вважається сучасною архітектурою мікросервісів [1].

ASP .NET Core має вбудовану допомогу для створення мікросервісів із контейнерами на основі Docker у формі API, які швидко споживають мікросервіси з будь-якого типу додатків.

Давайте тепер детально розглянемо ключові атрибути розробки ASP .NET Core та його переваги щодо створення мікросервісів:

Основні якості ASP .NET Core для побудови мікросервісів:

Краща та швидша контейнерізація.

Будь-яку програму ASP .NET Core можна помістити всередину контейнера Docker. Навіть традиційне програмне забезпечення підтримувало контейнерну технологію. Але остання версія того ж самого дозволяє отримати більш якісні образи Docker, і це також швидше.

Сумісність з Kubernetes та Docker.

Його сумісність з Kubernetes є додатковою функцією в шапці ASP .NET Core, тому вона може використовувати всі функції Kubernetes та ефективно розвивати мікросервіси. Навіть створення мікросервісів за допомогою Docker є ефективним варіантом завдяки всій доступній детальній документації [1].

Сумісність із крос-платформними програмами.

ASP .NET Core підтримує кілька операційних систем і сумісний з крос-платформною технологією. Раніша версія була обмежена, але тепер остання підтримує крос-платформність. Для мікросервісів важливо, щоб вони не залежали від будь-якої платформи чи архітектури [2].

Швидкість і вища сталість.

З часом та новішими версіями останні випуски ASP .NET Core ставали швидшими, кращими та безпечнішими. Останні показники вказують на той факт, що результати його роботи стають все кращими та кращими. Збільшилася незмінність щодо випусків, які відбувалися за останні кілька років [13].

Гнучкість для вибору IDE.

Створюючи програму в середовищі мікросервісів за допомогою ASP

.NET Core, розробники мають вибір вибрати IDE, яка їм найбільше підходить. Необов’язково взагалі вибирати Microsoft Visual Studio. Розробники мають можливість вибору безкоштовного програмного забезпечення для створення своїх програм у Microservices.

Сумісність з хмарними технологіями.

Основна природа ASP .NET Core підтримує хмарну масштабованість. Будь-який тип мікросервісів, побудований за допомогою цієї технології, безперечно працюватиме на всіх важливих хмарних послугах. Наприклад, Azure - це одна з хмарних технологій, яку можна рекомендувати, оскільки вона була розроблена спеціально для спільноти .NET.

Модульний та легкий

Дві основні властивості ASP .NET Core - його модульна природа та легка вагомість є ключовими для вибору його для розробки контейнерного додатку Microservice.

ASP .NET Core доводить свою ціну на ринку IT і успішно вважається придатним для створення мікросервісів. Існує багато мов програмування, які можна вибрати для побудови ідеальної архітектури мікросервісів, але ASP .NET Core виявився ідеальним вибором.

Висновки до розділу 1

Отже, нами були розглянуто мікросервісна архітектура для створення додаткві. Мікросервіси - це тип сервісно-орієнтованої архітектури програмного забезпечення, орієнтований на створення ряду автономних компонентів, складових додаток.

Розглянуто генезис та порівняно архітектури програмного забезпечення, зокрема таких як автономна, клієнт-серверна, монолітна, сервісно-орієнтована та мікросервісна архіктектури.

Було проаналізовано переваги та недоліки мікросервісної архіктектури.

Також було виконано огляд технології .NET Core для розробки мікросервісної архітектури.

РОЗДІЛ 2
ПРОЄКТУВАННЯ МІКРОСЕРВІСІВ ДЛЯ СИСТЕМИ БРОНЮВАННЯ

2.1 Вимоги до сервісу системи бронювання

Використання системи бронювання дозволяє значною мірою покращити якість обслуговування клієнтів, оскільки, забезпечуючи їх бронювання в автоматичному режимі, скорочуються затрати часу на оформлення квитків чи абонементу, а також підвищується якість та ефективність роботи персоналу.

Сьогодні більшість людей віддають перевагу бронюванню послуг у мережі Інтернет, оскільки такий спосіб є більш зручним для клієнтів та має ряд переваг.

З-поміж переваг упровадження системи бронювання в мережі Інтернет для клієнтів можна виділити:

- швидке отримання підтвердження резервування;
- необхідно затратити мінімум часу, щоб оформити резерв;
- є можливість отримання повної інформації стосовно своєї заяви у будь-який момент часу.

Компанії також отримують переваги при впровадженні онлайн-системи бронюванн:

- в першу чергу, це скорочення витрат через автоматизацію робочого процесу;
- можливість отримати актуальну інформацію про поточний стан завантаженості системи у будь-який момент часу;
- поліпшується якість обслуговування клієнтів, а саме – заявки клієнтів обробляються більш швидко.

Одним із факторів, який визначає популярність компанії на ринку, виступає час, витрачений на обслуговування клієнтів. У цьому аспекті беззаперечна перевага віддається обслуговуванню в онлайн-режимі. Саме

тому бронювання виступає центральною ланкою у загальній схемі обслуговування клієнтів.

На практиці велика кількість сайтів компаній надають можливість здійснювати резервування по запиту, тобто в оф-лайн режимі. Також таку схему роботи називають псевдоонлайн-бронюванням. Власне, такий принцип роботи – це форма попередньої заявки. Тобто, користувач обирає час, кількість місць, і при цьому вказує контактну інформацію.

Заява, відправлена користувачем, підлягає обробці (тобто затвердженню) з боку співробітника компанії. Як є така потреба, далі він зв’язується з клієнтом, уточнює деталі та здійснює фактичне розміщення броні.

Системи бронювання такого типу з-поміж основних своїх переваг мають легкість (простоту) у реалізації та низьку вартість розробки та втілення. Однак при цьому варто враховувати, що процес оформлення резерву за таких умов багато в чому нагадує прийом заявок по телефону.

Автоматизована система бронювання дозволяє вирішити ряд завдань, які зазвичай виникають у процесі резервування:

- уникнення «людського фактору», тобто найрозповсюдженіших помилок персоналу;
- перенавантаження чи, навпаки, простій ресурсів;
- зниження навантаження на співробітників компанії;
- полегшення процесу відслідковування роботи, формування звітів за певний проміжок часу,
- можливість надавати або позбавляти повноважень співробітників для роботи з системою, за необхідності.

Отже, для підтримки популярності сервісу, варто враховувати, що досвід взаємодії з ресурсом є надзвичайно важливим для успіху компанії.

Тобто, щоб клієнти із задоволенням користувалися послугами, які надаються сайтом компанії, зокрема, функцією бронювання як основною, необхідно, щоб сервіс був простий у використанні, а інтерфейс сайту –

зручним та інтуїтивно зрозумілим.

Система бронювання повинна відображати реальну інформацію про:

- поточні тарифи,
- наявність вільних місць,
- можливість здійснення резервування.

Сервіс системи бронювання басейну полягає в розробці та впровадженні програмного забезпечення для продажу і автоматичного аудиту наявності квитків та абонементів.

З огляду на загальні вимоги до системи бронювання, можна сформулювати наступні твердження. Система бронювання повинна відображати реальну інформацію щодо:

- Адреси басейну, розкладу роботи, способів та маршрутів добирання;
- Видів, типів доріжок;
- Інших розважальних та спортивних комплексів на території закладу;
- Можливості перебування дітей (з визначенням віку);
- Наявності сервісних умов (кафе, роздягальні, камери схову, сейфи для зберігання цінних речей та документів, інше)
- Систем тренувань для відвідувачів;
- Часу проведення спеціалізованих тренувань;
- Плану використання басейну, в якому по годинах вказано кількість вільних доріжок для плавців-любителів;
- Типів абонементів та їх наявності;
- Програми підтримки постійних клієнтів;
- Цінової інформації;
- Системи знижок та інших акцій;
- Можливості здійснення резервування станом на час звернення до системи;
- Санітарних вимог для відвідування закладу;

- Правил безпеки для можливості відвідування та перебування на території закладу;

- Обмежень пов’язаних з перебуванням в басейні та інших приміщеннях;

- Контактних даних менеджерів (ПІБ, спосіб зв’язку - телефон, E-mail, інше);

- Різновидності видів та способів проведення оплати;

- Підтвердження здійснення оплати та проведення бронювання в автоматичному режимі з визначенням даних про час здійснення оплати та проведення бронювання конкретним замовником.

Згідно з наведеними вимогами можно скласти Use Case діаграмму до системи (рис 2.1).

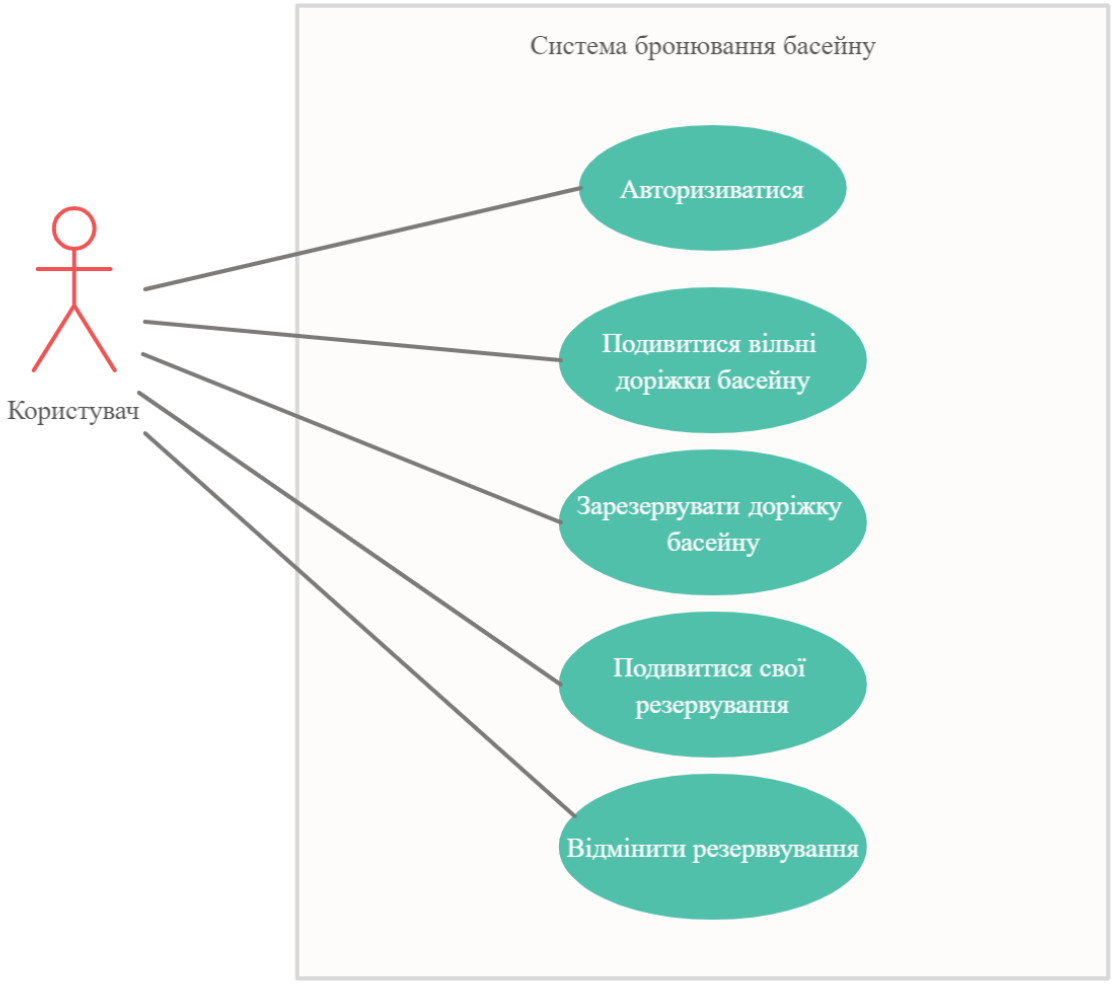

Рисунок 2.1 - Use Case діграма Системи бронювання басейну «Розклад»

Використання системи має спростити та покращити якість управління всіма процесами, пов’язаними з прийманням та обробкою замовлень, надати можливість менеджеру вчасно отримати потрібну інформацію для опрацювання.

Онлайн-бронювання покликане ефективно використати відвідувачами час проведення відпочинку та занять, скоротити та спростити процес замовлення.

Існує доцільність працювання системи бронювання в автономному режимі, цілодобово 24 години, 7 днів на тиждень.

2.2 Проєктування мікросервісів системи

Система бронювання є гнучкою до вибори об’єктів доменої області. Тобто для часткового випадку системи бронювання басейну використовується доріжки басейну, але система може працювати з іншими об’єкта для гнучкості, наприклад бронювання квитків кінотеатру, тощо.

Всю систему можна умовно розподілити на 2 основні частини: набір допоміжних сервісів, таких як точка для єдиного входу (API Gateway), а також незалежні самостійні мікросервіси. Система повина бути побудована за шаблонами:

Агрегатор. Розбиваючи бізнес-функціонал на кілька менших логічних фрагментів коду, стає необхідним подумати про те, як співпрацювати дані, що повертаються кожним сервісом [22]. Цю відповідальність не можна покласти на споживача, оскільки тоді йому може знадобитися зрозуміти внутрішню реалізацію програми виробника. Паттерн агретор допомагає вирішити цю проблему. У ньому йдеться про те, як ми можемо узагальнити дані різних служб, а потім надіслати остаточну відповідь споживачеві. Для цього будемо використовувати API Gateway який розділить запит на декілька мікросервісів та буде агрегувати дані перед тим, як відправити їх споживачеві.

База даних на сервіс. Повинна бути розроблена одна база даних на мікросервіс; вона повинна бути приватною лише для цього сервісу. До нього повинен мати доступ лише API мікросервісу. До нього не можуть отримати доступ інші сервіси безпосередньо. Наприклад, для реляційних баз даних ми можемо використовувати приватні таблиці на сервіси, схему на послугу або сервер баз даних на послугу. Кожнен мікросервіс повинен мати окремий ідентифікатор бази даних, щоб можна було надати окремий доступ для встановлення бар'єру та запобігання користуванню іншими таблицями сервісу.

Всього в системі чотири мікросервіси: Pool Service, User Service, Calendar Service , Booking Service.

Pool Service є доменим сервісом системи бронювання, тобто цей сервіс відповідає за те у якій конкретній предметній області працює система бронювання. Сервіс надає можливість отримати дані про басейну, його контакти та місцезнаходженя та інформаціїю беспосередньо про доріжки басейну, які є доменим об’єктом для системи бронювання.

User Service оперує даними користувачів та співпрацює із іншими сервісами в системі. Наприклад, при відображенні списку всіх замовлень користувача, ми йдемо до Booking Service. Також User Service представляє собою сервіс аутентифікації та авторизації, він імплементує Oauth2 та OpenId Connect протокол. Він розроблений, щоб надати загальний спосіб автентифікації запитів для всіх ваших програм, незалежно від того, чи є вони веб-сайтами, власними, мобільними або кінцевими точками API. Сервіс можна використовувати для реалізації єдиного входу (SSO) для декількох програм. Він може бути використаний для автентифікації фактичних користувачів за допомогою форм для входу та подібних користувацьких інтерфейсів, а також автентифікації на основі сервісу, який зазвичай включає видачу, перевірку та оновлення токенів без будь-якого інтерфейсу користувача.

Calendar Service оперує календарним графіком об’єкта (доріжки

басейну). Зберігає в собі дані про розклад та календарні записи. Надає можливість запису календарної події різних видів (бронювання, технічне обслуговування, вихідний день, тощо) для об’єкта. Також сервіс використовується клієнтом для відображення свободних сеансів відвідування певних доріжок басейну.

Booking Service забезпечує систему функціоналом для бронювання об'єктів. Якщо можливо сервіс назначає резервацію певного об’єкту на певний час роблячи при цьому відповідний запис в Calendar Service. В майбутньому у цьому сервісі також плануються інтеграція з платіжною системою.

Також для успішної реалізації взаємодії мікросервісів та всієї внутрішньої роботи необхідно використовувати додаткові інструменти:

API Gateway. При розробці та створенні великих або складних програм на базі мікросервісів із кількома клієнтськими програмами хорошим підходом може бути API Gateway. Це сервіс, який надає єдину точку доступу для певних груп мікросервісів. Це схоже на паттрн "Фасад" з об'єктно-орієнтованого дизайну, але в цьому випадку це частина розподіленої системи. Шаблон API Gateway також іноді називають "серверним інтерфейсом", оскільки ви створюєте його, думаючи про потреби клієнтської програми [22].

Тому API Gateway знаходиться між клієнтськими програмами та мікросервісами. Він діє як зворотний проксі-сервер, перенаправляючи запити від клієнтів до служб. Він також може надати додаткові наскрізні функції, такі як аутентифікація, припинення SSL та кеш-пам'ять.

Програми підключаються до однієї кінцевої точки, API Gateway, яка налаштована на пересилання запитів до окремих мікросервісів.

Потрібно бути обережним при реалізації шаблону API Gateway. Зазвичай не є гарною ідеєю мати єдиний API Gateway, який об’єднує всі

внутрішні мікросервіси Вашої програми. Якщо це так, він діє як монолітний агрегатор або оркестратор і порушує автономію мікросервісу, поєднуючи всі мікросервіси. Тому API Gateway повинні бути розділені на основі ділових кордонів та клієнтських програм, а не діяти як єдиний агрегатор для всіх внутрішніх мікросервісів.

В системі бронювання API Gateway буде реалізований як власна служба ASP.NET Core WebHost, що працює як контейнер.

Docker. Це провідна платформа для контейнеризації програмного забезпечення у світі. Він інкапсулює вашу мікросервіс у те, що ми називаємо контейнером Docker, який потім можна самостійно підтримувати та застосовувати. Кожен із цих контейнерів нестиме відповідальність за певний бізнес-функціонал.

За допомогою Docker можно зробити свою програму незалежною від хост-середовища. Оскільки система бронювання має архітектуру мікросервісів, то за допомогою Docker ми можете інкапсулювати кожен з них у контейнери Docker. Контейнери Docker - це легке середовище, ізольоване від ресурсів, за допомогою якого ви можете створювати, підтримувати, доставляти та розгортати свою програму [35].

Переваги:

- Docker - це популярне програмне забезпечення, що розвивається, з чудовою підтримкою спільноти та розроблене для мікросервісів
- Він легкий у порівнянні з віртуальними машинами, що робить його економічним та економічним
- Це забезпечує однорідність середовищ розробки та виробництва, що робить його придатним для створення власних хмарних додатків
- Він забезпечує можливості для постійної інтеграції та розгортання
- Докер нікуди не йде; він забезпечує інтеграцію з популярними

інструментами та сервісами, такими як AWS, Microsoft Azure, Ansible, Kubernetes, Istio та багато іншого.

2.3 Структура баз даних сервісів системи

База даних Pool Service зберігає домені дані системи бронювання - дані про басейн. Має дві головні таблиці Pool та PoolTrack (рис. 2.1).
Pool зберігає в собі основну інформацію про басейн: назву, місцезнаходження та контактну інформацію. Також таблиця має відношення один до багатьох с таблицею PoolTrack. Тобто басейн має кілька доріжок кожна р яких має своє ім'я (Name) там унікальний номер (PoolTrackNumber)

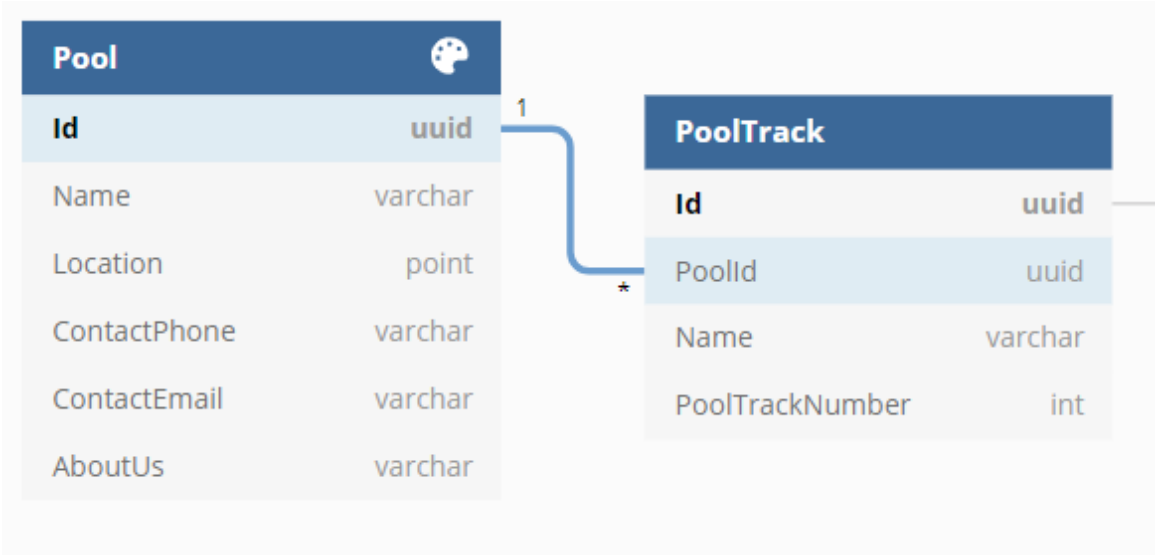

Рисунок 2.2 - Структура бази даних сервісу Pool Service

Для реалізації User Service використано систему ідентифікації ASP.NET Identity, зберігає всю інформацію про користувача у базі даних. ASP.NET Identity використовує Entity Framework Code First для реалізації всього свого механізму збереження. Тому проектування бази даних основано на структурі бази даних яка постачається використанням ASP.NET Identity. Містить в собі такі типові таблиці (рис. 2.3):

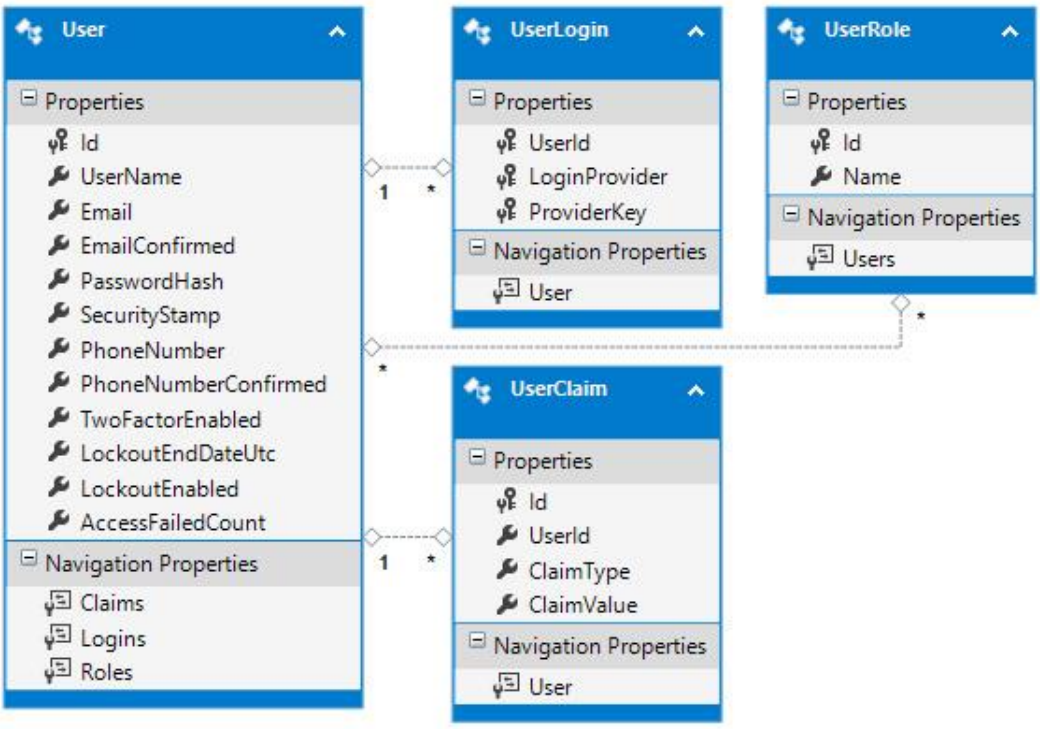

Рисунок 2.3 - Структура бази даних сервісу User Service

User - Зареєстровані користувачі вашого системи. Включає ідентифікатор користувача та ім’я користувача. Може містити хешований пароль, якщо користувачі входять з обліковими даними, які є специфічними для сайту (а не використовують облікові дані із зовнішнього сайту, такого як Facebook), та штамп безпеки (SecurityStamp), щоб вказати, чи щось змінилося в облікових даних користувача. Також може містити адресу електронної пошти, номер телефону, увімкнено двофакторну аутентифікацію, поточну кількість невдалих входів та чи заблоковано обліковий запис.

UserRole - Група юзерів з своїми правами. Включає ідентифікатор ролі та ім'я ролі (наприклад, "Адміністратор" або "Користувач").

UserClaim - Набір тверджень (або претензій) щодо користувача, які представляють ідентичність користувача. Може забезпечити більший вираз особистості користувача, ніж можна досягти за допомогою ролей.

UserLogin - Інформація про зовнішнього постачальника автентифікації (наприклад, Facebook), який слід використовувати під час

входу в систему користувача.

База даних Calendar Service має такі основні таблиці (рис. 2.4):

Schedule - розклад для об'єкту (доріжки басейну), означає час старту та час кінця роботи об’єкту для певного дня. Розклад наперед створюється засобами Calendar Service.

CalendarRecord - означає что для певного об'єкта (доріжки басейну) була зроблена запис в календарі з якоїсь причини (має відповідний стовбец Reason) - може бути забронъований (Booked), знаходитися на технічному обслуговуванні (Maintenance), тощо. CreatorId позначає ким був зроблений запис, значення якого дорівнює ідентифікатору відповідного юзера в User Service.

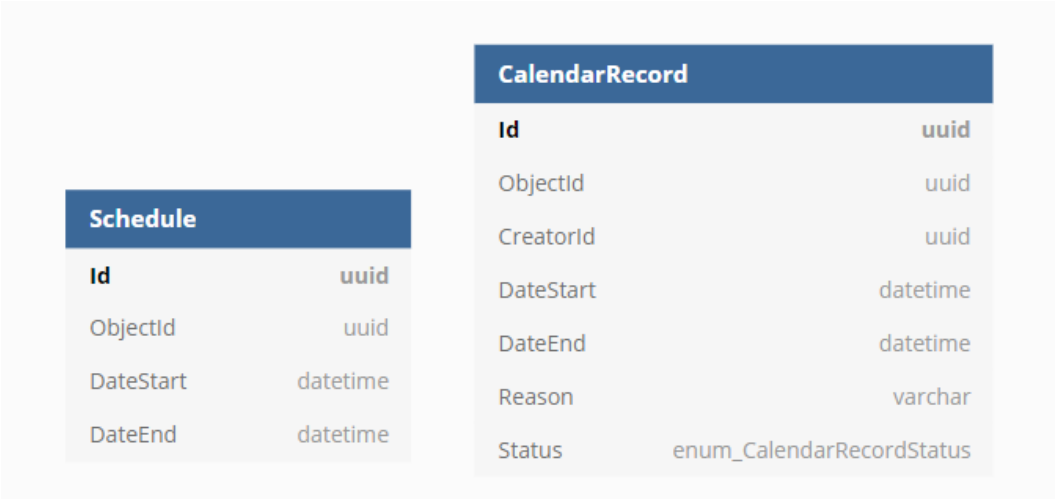

Рисунок 2.4 - Структура бази даних сервісу Calendar Service

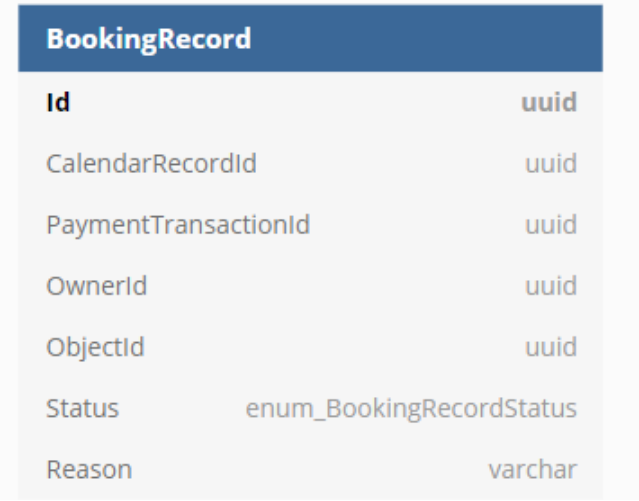

Рисунок 2.5 - Структура бази даних сервісу Booking Service

Booking Service відповідно зберігає дані про зроблені бронювання в системі. Найголовніша таблиця в базі сервісу BookingRecord (рис. 2.5) пов’язує об’єкт бронювання (ObjectId), календарний запис (CalendarRecordId) та власника бронювання (OwnerId). Також закладена можливість у майбутньому запровадити платежі для бронювання (PaymentTransactionId).

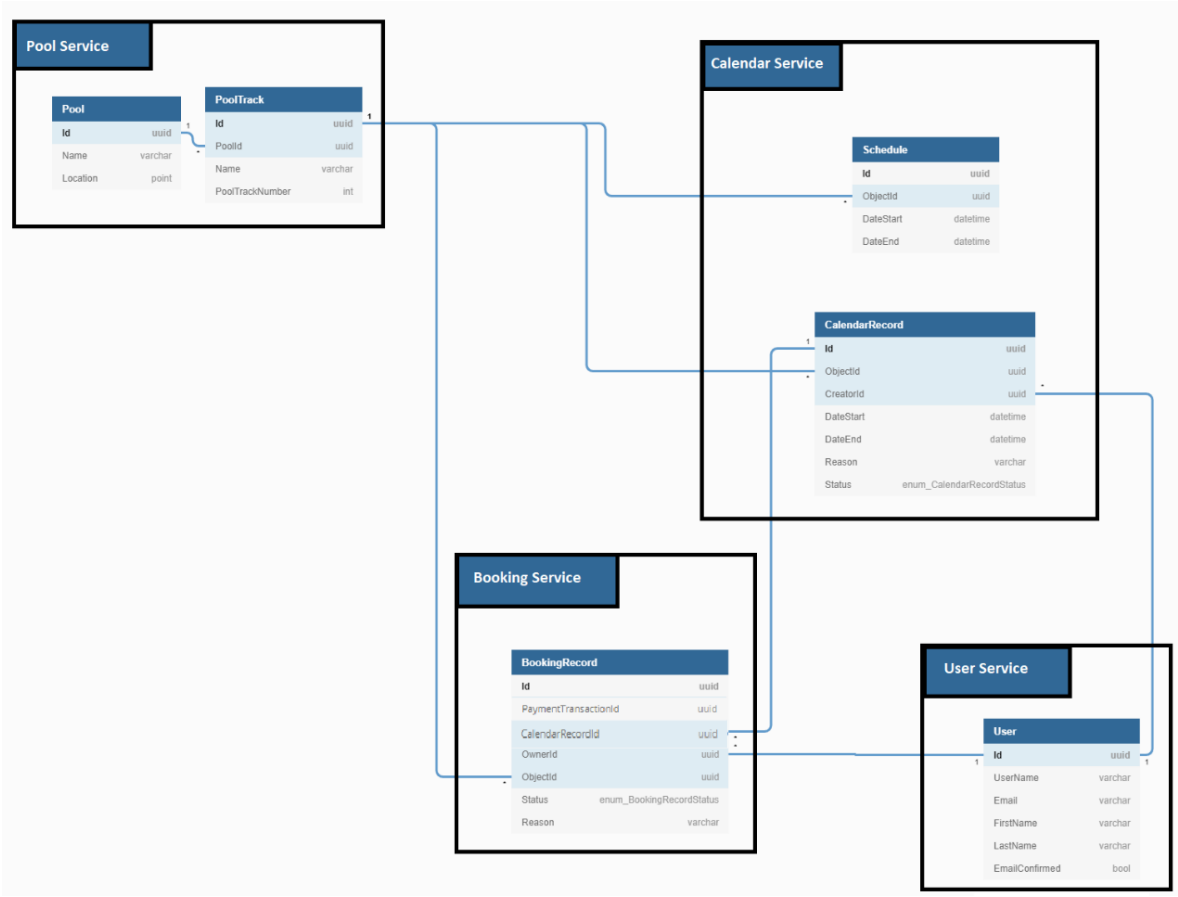

Рисунок 2.6 - Схема взамозвязку даних між базами даних сервісів

Як було зазначено раніше кожен сервіс має свою окрему базу даних. Дані в сервісах слабо пов’язані між собою. Це означає що не має звязків, які забезпечують цілістність, між таблицями разних сервісів. Проте один сервіс может оперувати індетифікатор деякої сутніості яка зберігається в іншому сервісі. Це ми можемо побачити на схемі взамозвязку даних між базами даних сервісів (рис. 2.6). Schedule, CalendarRecord та BookingRecord таблиці мають стовбець ObjectId, який обозначає домений об’єкт в системі бронювання, в нашому випаду – доріжку басейну та

дорівнює PoolTrack.Id. Також CalendarRecord та BookingRecord посилаються на User властивостями CreatorId та OwnerId відповідно. А BookingRecord посилається на CalendarRecord відповідно через CalendarRecordId

Висновки до розділу 2

Отже, нами було проаналізовано та виділено основні вимог до системи онлайл бронювання басейну. Сервіс системи бронювання басейну полягає в розробці та впровадженні програмного забезпечення для продажу і автоматичного аудиту наявності квитків та абонементів. Автоматизована система бронювання дозволяє вирішити ряд завдань, які зазвичай виникають у процесі резервування:

- уникнення «людського фактору», тобто найрозповсюдженіших помилок персоналу;
- перенавантаження чи, навпаки, простій ресурсів;
- зниження навантаження на співробітників компанії;
- полегшення процесу відслідковування роботи, формування звітів за певний проміжок часу,
- можливість надавати або позбавляти повноважень співробітників для роботи з системою, за необхідності.

Виділено та спроєктовано такі мікросервіси системи як Pool Service, User Service, Calendar Service , Booking Service. Також наведена структура баз даних для відповідних сервісів.

Також описано взаємодія сервісів в рамках системи з допомогою технологій API Gateway та Docker.

РОЗДІЛ 3
РОЗРОБКА СИСТЕМИ БРОНЮВАННЯ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

3.1 Особливості реалізації мікросервісів системи бронювання на платформі .NET

Під час реалізації мікросервісів було використано такі технології на платформі .NET Core як ASP.NET Core, Entity Framework Core, IdentityServer. Розглянемо особливості цих технологій детальніше.

ASP.NET Core - це безкоштовна платформа з відкритим кодом та міжплатформна платформа для створення хмарних додатків, таких як веб-програми, програми IoT та мобільні серверні системи. Він призначений для роботи як у хмарі, так і в локальній мережі [13].

Так само, як .NET Core, він був побудований модульним з мінімальними накладними витратами, а потім інші більш розширені функції можуть бути додані як пакети NuGet відповідно до вимог програми. Це призводить до високої продуктивності, вимагає менше пам'яті, менше розміру розгортання та простоти обслуговування.

ASP.NET Core - це фреймворк з відкритим кодом, що підтримується корпорацією Microsoft та спільнотою, тому ви також можете внести або завантажити вихідний код із сховища ASP.NET Core на Github.

ASP.NET 3.x працює лише на .NET Core 3.x, тоді як ASP.NET Core 2.x працює на .NET Core 2.x, а також .NET Framework.

ASP.NET Core має такі переваги:

Кросплатформеність: Програми ASP.NET Core можуть працювати на Windows, Linux та Mac. Тому вам не потрібно створювати різні програми для різних платформ, використовуючи різні фреймворки [1].

Швидкість: ASP.NET Core більше не залежить від System.Web.dll для зв’язку браузер-сервер. ASP.NET Core дозволяє нам включати пакети, які нам потрібні для нашої програми. Це зменшує конвеєр запитів та

покращує продуктивність та масштабованість.

Inversion of Control (IoC) Container: Він включає вбудований IoC-контейнер для автоматичного введення залежностей, що робить його ремонтопридатним і перевіряється.

Інтеграція із сучасними фреймворками інтерфейсу користувача: це дозволяє використовувати та керувати сучасними фреймворками інтерфейсу, такими як Angular, ReactJS, Umber, Bootstrap тощо, використовуючи Bower (веб-менеджер пакетів).

Хостинг: Веб-програма ASP.NET Core може розміщуватися на декількох платформах з будь-яким веб-сервером, таким як IIS, Apache тощо. Це не залежить лише від IIS як стандартної .NET Framework.

Спільний доступ до коду: це дозволяє створювати бібліотеку класів, яку можна використовувати з іншими платформами .NET, такими як .NET Framework 4.x або Mono. Таким чином, одна база коду може бути спільно використана між фреймворками.

Побічне керування версіями програм: ASP.NET Core працює на .NET Core, який підтримує одночасний запуск декількох версій програм.

Менший розмір розгортання: Додаток ASP.NET Core працює на .NET Core, який менше, ніж повний .NET Framework. Отже, програма, яка використовує лише частину .NET CoreFX, матиме менший розмір розгортання. Це зменшує розміщення розгортання.

Entity Framework Core - це об’єктно-реляційний фреймворк (ORM) фреймворк із відкритим кодом для програм .NET, що підтримуються корпорацією Майкрософт. Це дозволяє розробникам працювати з даними, використовуючи об'єкти класів конкретного домену, не зосереджуючись на основних таблицях та стовпцях бази даних, де ці дані зберігаються [12]. За допомогою Entity Framework розробники можуть працювати на вищому рівні абстракції, коли мають справу з даними, а також можуть створювати та підтримувати орієнтовані на дані програми з меншим кодом порівняно з традиційними програмами. Це позбавляє від

необхідності більшості коду доступу до даних, який зазвичай потрібно писати розробникам ".

Entity Framework розміщується між бізнес рівнем додатку (класами доменів) та базою даних. Він зберігає дані, що зберігаються у властивостях суб'єктів господарювання, а також отримує дані з бази даних і автоматично перетворює їх на об'єкти бізнес рівня.

Особливості Entity Framework:

Кроссплатформенність: EF Core - це міжплатформний фреймворк, який може працювати на Windows, Linux та Mac.

Моделювання: EF (Entity Framework) створює EDM (Entity Data Model) на основі сутностей POCO (Plain Old CLR Object) із властивостями get / set різних типів даних. Ця модель використовується під час запитів або збереження даних сутності до базової бази даних.

Запит: EF дозволяє нам використовувати запити LINQ (C # / VB.NET) для отримання даних з базової бази даних. Постачальник баз даних перекладе ці запити LINQ на мову запитів, що стосується бази даних (наприклад, SQL для реляційної бази даних). EF також дозволяє нам виконувати необроблені SQL-запити безпосередньо до бази даних.

Відстеження змін: EF відстежує зміни, що відбулись у екземплярах ваших об’єктів (значення властивостей), які потрібно надіслати до бази даних.

Збереження: EF виконує команди INSERT, UPDATE та DELETE в базу даних на основі змін, що відбулися у ваших об'єктах під час виклику методу SaveChanges (). EF також забезпечує асинхронний метод SaveChangesAsync ().

Паралельність: EF використовує Optimistic Concurrency за замовчуванням для захисту перезапису змін, внесених іншим користувачем, оскільки дані були отримані з бази даних.

Транзакції: EF виконує автоматичне управління транзакціями під час запиту або збереження даних. Він також надає параметри

налаштування управління транзакціями.

Кешування: EF включає перший рівень кешування з коробки. Отже, повторний запит поверне дані з кешу, а не потрапить у базу даних.

Вбудовані конвенції: EF дотримується конвенцій за шаблоном програмування конфігурації та включає набір правил за замовчуванням, які автоматично налаштовують модель EF.

Конфігурації: EF дозволяє нам налаштувати модель EF, використовуючи атрибути анотації даних або API Fluent, щоб замінити правила за замовчуванням.

Міграції: EF надає набір команд міграції, які можна виконувати на консолі NuGet Package Manager або інтерфейсі командного рядка для створення або управління базовою схемою бази даних.

IdentityServer - це сервер автентифікації з відкритим кодом, який реалізує стандарти OpenID Connect (OIDC) та OAuth 2.0 для ASP.NET Core. Він розроблений, щоб надати загальний спосіб автентифікації запитів для всіх ваших програм, незалежно від того, чи є вони веб-сайтами, власними, мобільними або кінцевими точками API. IdentityServer можна використовувати для реалізації єдиного входу (SSO) для декількох програм та типів програм. Він може бути використаний для автентифікації фактичних користувачів за допомогою форм для входу та подібних користувацьких інтерфейсів, а також автентифікації на основі послуг, яка зазвичай включає видачу, перевірку та оновлення токенів без будь-якого інтерфейсу користувача. IdentityServer розроблений як настроюване рішення. Кожен екземпляр, як правило, налаштовується відповідно до потреб окремої організації та / або набору програм.

IdentityServer надає проміжне програмне забезпечення, яке працює в програмі ASP.NET Core, і додає підтримку OpenID Connect та OAuth2. Організації повинні створити власну програму ASP.NET Core, використовуючи проміжне програмне забезпечення IdentityServer, щоб виступати в якості Secure Token Service (STS) для всіх своїх протоколів

безпеки на основі токенів. Проміжне програмне забезпечення IdentityServer надає кінцеві точки для підтримки стандартних функціональних можливостей.

Identity Server надає можливість програмам підтримувати наступні сценарії:

- Людські користувачі, що отримують доступ до веб-програм за допомогою браузера.
- Людські користувачі, які отримують доступ до внутрішніх веб-API через додатки на основі браузера.
- Людські користувачі мобільних / власних клієнтів, які отримують доступ до внутрішніх веб-API.
- Інші програми, що мають доступ до внутрішніх веб-API (без активного користувача або користувальницького інтерфейсу).
- Будь-якій програмі може знадобитися взаємодіяти з іншими веб-API, використовуючи власну ідентифікацію або делегуючи особу користувача.

У кожному з цих сценаріїв відкриту функціональність потрібно захищати від несанкціонованого використання. Як мінімум, це зазвичай вимагає автентифікації користувача чи принципала, який робить запит на ресурс. Ця автентифікація може використовувати один із декількох загальних протоколів, таких як SAML2p, WS-Fed або OpenID Connect. Взаємодія з API зазвичай використовує протокол OAuth2 та його підтримку маркерів безпеки. Відокремлення цих найважливіших наскрізних проблем безпеки та деталей їх реалізації від самих додатків забезпечує узгодженість та покращує безпеку та ремонтопридатність. Передання цих проблем на спеціальний продукт, такий як IdentityServer, допомагає кожній програмі самостійно вирішити ці проблеми.

3.2 Тестування мікросервісів системи

Тестування мікросервісів може бути складною роботою, оскільки

воно відрізняється від того, як ми тестуємо програми, побудовані з використанням традиційного архітектурного стилю. У монолітній програмі .NET тестування дещо простіше порівняно з мікросервісами, що забезпечує незалежність від реалізації та короткі цикли доставки [21].

В контексті монолітного додатка .NET, де не використовується постійна інтеграцію та розгортання. Це стає більш складним, коли тестування поєднується з постійною інтеграцією та розгортанням. У мікросервісах нам потрібно буде зрозуміти тести для кожного мікросервіса та те, як ці тести відрізняються один від одного. Також зауважте, що автоматизоване тестування не означає, що ми взагалі не будемо проводити жодного тестування вручну.

Ось кілька речей, які роблять тестування мікросервісу складним та складним завданням:

Мікросервіси можуть мати кілька сервісів, які працюють разом або окремо для корпоративної системи, тому вони можуть бути складними.

Мікросервіси призначені для націлювання на кількох клієнтів, отже вони включають більш складні випадки використання.

Кожен компонент архітектурного стилю мікросервісу є ізольованими та незалежними, тому протестувати їх трохи складно, оскільки їх потрібно протестувати індивідуально та як цілісну систему [21].

Можуть існувати незалежні групи, які працюють над окремими компонентами, які можуть знадобитися для взаємодії між собою; тому їх слід тестувати таким чином, щоб тести охоплювали не лише внутрішні, а й зовнішні сервіси. Це робить роботу з тестування мікросервісів більш складною та складною.

У мікросервісах кожен компонент призначений для незалежної роботи, але вони можуть бути загальними для різних інших сервісів вимагати доступу до спільних даних. Але в мікросервісах кожен сервіс відповідає за модифікацію власної бази даних. Отже, тестування

мікросервісів буде більш складним, оскільки сервісам потрібно буде отримувати доступ до даних за допомогою викликів API з іншими сервісами, що додатково додає залежності до інших сервісів. Цей тип тестування доведеться проводити за допомогою фіктивних тестів.

Інтеграційне тестування гарантує, що компоненти програми функціонують належним чином у зібраному вигляді. ASP.NET Core підтримує інтеграційне тестування за допомогою модульних тестових фреймворків та вбудованого тестового веб-хоста, який можна використовувати для обробки запитів без накладних витрат на мережу.

На відміну від модульного тестування, інтеграційні тести часто стосуються проблем інфраструктури програм, таких як база даних, файлова система, мережеві ресурси або веб-запити та відповіді. У юніт-тестах замість цих проблем використовуються підробки або вигадки. Але мета інтеграційних тестів - підтвердити, що система працює належним чином із цими системами, тому для інтеграційного тестування ви не використовуєте підробку чи макет об’єктів. Натомість ви включаєте інфраструктуру, таку як доступ до бази даних або виклик сервісу з інших сервісів.

Оскільки інтеграційні тести використовують більші сегменти коду, ніж модульні тести, і оскільки інтеграційні тести покладаються на елементи інфраструктури, вони, як правило, на порядок повільніші, ніж модульні тести. Таким чином, корисно обмежити кількість тестів інтеграції, які ви пишете та запускаєте.

ASP.NET Core включає вбудований тестовий веб-хост, який можна використовувати для обробки HTTP-запитів без накладних витрат на мережу, що означає, що ви можете запускати ці тести швидше, ніж при використанні реального веб-хоста. Тестовий веб-хост (TestServer) доступний у компоненті NuGet як Microsoft.AspNetCore.TestHost. Його можна додавати до тестових проектів інтеграції та використовувати для розміщення програм ASP.NET Core.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи було досягнуто поставленої мети, яка полягала у розробці сервісного API для системи з архітектурою мікросервісів.

У ході виконання роботи були виконані наступні завдання:

1. Виконано огляд та аналіз мікросервісів як сучасного рішення для архітектури ПЗ. Мікросервіси - це тип сервісно-орієнтованої архітектури програмного забезпечення, орієнтований на створення ряду автономних компонентів, складових додаток. Основні переваги гнучкість, масштабованість, швидкість розробки та незалежність мови програмування. Основні недоліки складність проєктування та висока вартість розробки.

2. Розглянуто особливості застосування технології .NET Core для реалізації мікросервісної архітектури. Виділено що ASP.NET Core добре підходить для впровадження мікросервісів завдяки модульності сумісністі з хмарними технологіями та технологіями контеренізаціїї.

3. Проаналізувано та описано вимоги до сервісу системи бронювання. Сервіс системи бронювання басейну полягає в розробці та впровадженні програмного забезпечення для продажу і автоматичного аудиту наявності квитків та абонементів.

4. Спроєктувано мікросервіси та бази даних системи. Виділено та спроєктовано такі мікросервіси системи як Pool Service, User Service, Calendar Service, Booking Service. Також наведена структура їх баз даних сервісів та описана взаємодія в рамках системи.

5. Розроблено серверний API з мікросервісною архітектурою для системи бронювання. Перераховано, які технології були використані для реалізації API сервісів системи бронювання басейну. Реалізована бізнес логіка системи бронювання відповідає загальним вимогам до систем бронювання.

Результатом роботи є створення серверного API системи бронювання басейну ХДУ з використанням технологій .NET Core. Представлені результати дозволяють застосовувати розроблену систему для спрощення та автоматизування процесу бронювання доріжок для користувачів басейном ХДУ.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ASP.NET Core License". GitHub. .NET Foundation. July 5, 2017. Retrieved April 14, 2018.
2. ASP.NET MVC, Web API and Web Pages (Razor)". dotnetfoundation.org. .NET Foundation. Archived from the original on 17 February 2015. Retrieved 17 February 2015.
3. Architectural Styles and the Design of Network-based Software Architectures [Электронный ресурс] // UCI: [сайт]. [2000]. URL: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (дата обращения: 15.05.2016).
4. Chris Richardson. From Design to Deployment / Chris Richardson, Floyd Smith,2016. – 74 p.
5. E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software /E. Evans – Addison-Wesley, 2003 – 560 p.
6. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions / Gregor Hohpe, Bobby Woolf – Addison-Wesley, 2004 – 736 p.
7. Fielding Roy. Architectural Styles and the Design of Network-based Software Architectures – Каліфорнійський університет в Ірвайні, 2000. – с. 1.
8. Fielding Roy. Architectural Styles and the Design of Network-based Software Architectures, 2000. – с. 8.
9. HOW BIG SHOULD A MICRO-SERVICE BE? [Электронный ресурс] // The Bovine Synchrotron, James Lewis's blog: [сайт]. [2013]. URL: http://bovon.org/archives/350 (дата обращения: 02.05.2016).
10. I. Nadareishvili. Microservice Architecture: Aligning Principles, Practices, and Culture / I. Nadareishvili, R. Mitra, M. McLarty, M. Amundsen – O'Reilly Media, 2016 – 146 p.
11. Introduction to microservices. – Режим доступу:

https://nginx.com/blog/introduction-to-microservices/ – Дата доступу: 21.10.2020

12. Julia Lerman. Programming Entity Framework. — 2nd Edition. — O'Reilly, 2010. — 920 p.
13. Landwerth, Immo (November 12, 2014). ".NET Core is Open Source". .NET Framework Blog. Microsoft. Retrieved December 30, 2014.
14. Martin Fowler – Microservices – Режим доступу:http://martinfowler.com/articles/microservices.html – Дата доступу: 21.10.2020
15. Microservices [Электронный ресурс] // Martin Fowler: [сайт].[2014]. URL: http://martinfowler.com/articles/microservices.html (дата обращения: 02.05.2016).
16. Philippe Kruchten, The Rational Unified Process: An Introduction, Third Edition (Addison-Wesley Professional), 2003. – c. 5.
17. Separation of concerns article https://en.wikipedia.org/wiki/Separation_of_concerns
18. Service Discovery. – Режим доступу: https://nginx.com/blog/service-discovery-ina-microservices-architecture/ – Дата доступу: 02.11.2020
19. Service-Oriented Architecture (SOA) [Электронный ресурс] // urlIntegration: [сайт]. URL: http://www.urlintegration.com/?page_id=752 (дата обращения: 03.05.2016).
20. ServiceOrientedAmbiguity [Электронный ресурс] // Martin Fowler: [сайт]. [2005]. URL: http://martinfowler.com/bliki/ServiceOrientedAmbiguity.html (дата обращения: 02.05.2016).
21. Testing ASP.NET Core services and web apps [Электронный ресурс] // [сайт]. URL: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/test-aspnet-core-services-web-apps (Дата доступу:

03.11.2020).

22. Using an API Gateway. – Режим доступу: https://nginx.com/blog/buildingmicroservices-using-an-api-gateway/ – Дата доступу: 02.11.2020
23. Белл, Майкл (2008). Белл, Майкл (ред.). Сервис-ориентированное моделирование: анализ, проектирование и архитектура сервисов . Вайли
24. Валерий Коржов. Многоуровневые системы клиент-сервер. - Издательство "Открытые системы", 1997. – 10 с.
25. Гарсас, Хавьер; Пиаттини, Марио (2005). «Онтология знаний в области микроархитектурного проектирования». Программное обеспечение IEEE
26. Е. М. Пройдаков, Л. А. Теплицький. Англо-Український тлумачний словник з обчислювальної техніки, Інтернету і програмування. – СофтПрес. – с. 552.
27. Казман, Рик (май 2003). «Архитектура, дизайн, реализация» (PDF) . Институт программной инженерии . - О различии архитектурного проектирования и рабочего проекта.
28. Клементс, Пол (2010). Документирование программных архитектур: взгляды и не только (2-е изд.). Эддисон-Уэсли Профессионал
29. Кристиан Нейгел, Билл Ивьен и др. C# 4.0 и платформа .NET 4 для профессионалов = Professional C# 4 and .NET 4. — М.: «Диалектика», 2010. — С. 1440.
30. Крухтен, Филипп (1995). «Архитектурные чертежи - модель архитектуры программного обеспечения« 4 + 1 »» (PDF) . Программное обеспечение IEEE
31. Лен, Басс (2012). Архитектура программного обеспечения на практике (3-е изд.). Эддисон-Уэсли Профессионал
32. М. Фаулер. Архитектура корпоративных программных приложений

/ М. Фаулер. – Издательский дом Вильямс, 2006 – 544 с.

33. Микросервисы [Электронный ресурс] // Хабрахбар: [сайт]. [2015]. URL: https://habrahabr.ru/post/249183/ (дата обращения: 02.05.2016).
34. Ньюмен С. Создание микросервисов / Ньюмен С. – СПб.: Питер, 2016 – 304 с.
35. Офіційний сайт Docker. – Режим доступу: https://docker.com/ – Дата доступу: 02.11.2020
36. Ричардс, Марк (2020). Основы архитектуры программного обеспечения: инженерный подход . O'Reilly Media
37. Фаулер М. Архитектура корпоративных программных приложений. Москва: "Вильяме", 2006. 544 с.
38. Фаулер, Мартин (сентябрь 2003 г.). "Кому нужен архитектор?" (PDF) . Программное обеспечение IEEE . 20 (5
39. Чернецький К., Айзенекер У. Порождающее программирование. Методы, инструменты, применение. - Видавничий дім "Пітер", 2005. - 730 с.
40. Шан, Тони; Хуа, Винни (октябрь 2006 г.). «Механизм построения решения». Материалы 10-й Международной конференции по корпоративным вычислениям IEEE EDOC
41. Эндрю Троелсен. Язык программирования C# 2010 и платформа .NET 4.0 = Pro C# 2010 and the .NET 4.0 Platform, 5ed. — М.: «Вильямс», 2010. — С. 1392

ДОДАТКИ

Додаток А

КОДЕКС АКАДЕМІЧНОЇ ДОБРОЧЕСНОСТІ
ЗДОБУВАЧА ВИЩОЇ ОСВІТИ ХЕРСОНСЬКОГО
ДЕРЖАВНОГО УНІВЕРСИТЕТУ

Я, Алоскін Антон Вікторович,
учасник(ця) освітнього процесу Херсонського державного університету, **УСВІДОМЛЮЮ**, що академічна доброчесність – це фундаментальна етична цінність усієї академічної спільноти світу.

ЗАЯВЛЯЮ, що у своїй освітній і науковій діяльності **ЗОБОВ'ЯЗУЮСЯ**:

– дотримуватися:
- вимог законодавства України та внутрішніх нормативних документів університету, зокрема Статуту Університету;
- принципів та правил академічної доброчесності;
- нульової толерантності до академічного плагіату;
- моральних норм та правил етичної поведінки;
- толерантного ставлення до інших;
- дотримуватися високого рівня культури спілкування;

– надавати згоду на:
- безпосередню перевірку курсових, кваліфікаційних робіт тощо на ознаки наявності академічного плагіату за допомогою спеціалізованих програмних продуктів;
- оброблення, збереження й розміщення кваліфікаційних робіт у відкритому доступі в інституційному репозитарії;
- використання робіт для перевірки на ознаки наявності академічного плагіату в інших роботах виключно з метою виявлення можливих ознак академічного плагіату;

– самостійно виконувати навчальні завдання, завдання поточного й підсумкового контролю результатів навчання;

– надавати достовірну інформацію щодо результатів власної навчальної (наукової, творчої) діяльності, використаних методик досліджень та джерел інформації;

– не використовувати результати досліджень інших авторів без використання покликань на їхню роботу;

– своєю діяльністю сприяти збереженню та примноженню традицій університету, формуванню його позитивного іміджу;

– не чинити правопорушень і не сприяти їхньому скоєнню іншими особами;

– підтримувати атмосферу довіри, взаємної відповідальності та співпраці в освітньому середовищі;

– поважати честь, гідність та особисту недоторканність особи, незважаючи на її стать, вік, матеріальний стан, соціальне становище, расову належність, релігійні й політичні переконання;

– не дискримінувати людей на підставі академічного статусу, а також за національною, расовою, статевою чи іншою належністю;

– відповідально ставитися до своїх обов'язків, вчасно та сумлінно виконувати необхідні навчальні та науково-дослідницькі завдання;

– запобігати виникненню у своїй діяльності конфлікту інтересів, зокрема не використовувати службових і родинних зв'язків з метою отримання нечесної переваги в навчальній, науковій і трудовій діяльності;

– не брати участі в будь-якій діяльності, пов'язаній із обманом, нечесністю, списуванням, фабрикацією;

– не підроблювати документи;

– не поширювати неправдиву та компрометуючу інформацію про інших здобувачів вищої освіти, викладачів і співробітників;

– не отримувати і не пропонувати винагород за несправедливе отримання будь-яких переваг або здійснення впливу на зміну отриманої академічної оцінки;

– не залякувати й не проявляти агресії та насильства проти інших, сексуальні домагання;

– не завдавати шкоди матеріальним цінностям, матеріально-технічній базі університету та особистій власності інших студентів та/або працівників;

– не використовувати без дозволу ректорату (деканату) символіки університету в заходах, не пов'язаних з діяльністю університету;

– не здійснювати і не заохочувати будь-яких спроб, спрямованих на те, щоб за допомогою нечесних і негідних методів досягати власних корисних цілей;

– не завдавати загрози власному здоров'ю або безпеці іншим студентам та/або працівникам.

УСВІДОМЛЮЮ, що відповідно до чинного законодавства у разі недотримання Кодексу академічної доброчесності буду нести академічну та/або інші види відповідальності й до мене можуть бути застосовані заходи дисциплінарного характеру за порушення принципів академічної доброчесності.

05.05.2020	[підпис]	Антон Алоскін
(дата)	(підпис)	(ім'я, прізвище)