Patrycja Kozak

200831

Analiza porównawcza architektury monolitycznej oraz mikroserwisów na przykładzie aplikacji webowej

**Comparative Analysis of Monolithic Architecture and Microservices Architecture Based on a Web Application Example**

Praca dyplomowa na kierunku:

Informatyka

Praca wykonana pod kierunkiem   
dr inż. Artur Krupa

Instytut Informatyki Technicznej

Katedra Sztucznej Inteligencji

Warszawa, rok 2024

**Oświadczenie Promotora pracy**

Oświadczam, że niniejsza praca została przygotowana pod moim kierunkiem   
i stwierdzam, że spełnia warunki do przedstawienia tej pracy w postępowaniu   
o nadanie tytułu zawodowego.

Data .................................... Podpis promotora ...................................................

**Oświadczenie autora pracy**

Świadom odpowiedzialności prawnej, w tym odpowiedzialności karnej za złożenie fałszywego oświadczenia, oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami prawa, w szczególności ustawą z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (Dz.U. 2019 poz. 1231 z późn. zm.).

Oświadczam, że przedstawiona praca nie była wcześniej podstawą żadnej procedury związanej z nadaniem dyplomu lub uzyskaniem tytułu zawodowego.

Oświadczam, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną. Przyjmuję do wiadomości, że praca dyplomowa poddana zostanie procedurze antyplagiatowej.

Data ..................................... Podpis autora pracy .............................................

**Streszczenie**

**Tytuł Pracy dyplomowej**

Praca dyplomowa poświęcona jest analizie i porównaniu dwóch popularnych architektur systemowych – monolitycznej oraz mikroserwisowej – pod kątem ich implementacji, wydajności oraz kosztów wdrożenia. W pierwszej części pracy przedstawiono przegląd literatury, charakteryzując obie architektury, ich cechy, zalety i wady, a także dokonano ich porównania. Następnie opisano wykorzystane rozwiązania technologiczne, w tym języki programowania, frameworki, bazy danych, narzędzia do konteneryzacji . W części praktycznej stworzono dwa systemy: pierwszy w architekturze monolitycznej, drugi w mikroserwisowej. Szczegółowo omówiono proces projektowania, implementacji, komunikacji między komponentami oraz konfiguracji systemów. W ramach analizy porównano metryki wydajnościowe obu rozwiązań oraz oszacowano koszty wdrożenia aplikacji w środowisku chmurowy. Ostatni rozdział zawiera wnioski końcowej oraz rekomendacje.

Słowa kluczowe – monolit, mikroserwisy, analiza porównawcza, wydajność, architektura monolityczna, architektura mikroserwisowa, konteneryzacja

**Summary**

**Tytuł Pracy w języku ang.**

This thesis is dedicated to the analysis and comparison of two popular system architectures – monolithic and microservices – in terms of their implementation, performance, and deployment costs. In the first part of the thesis, a literature review is presented, characterizing both architectures, their features, advantages, and disadvantages, as well as providing a comparison between them. Next, the utilized technological solutions are described, including programming languages, frameworks, databases, and containerization tools.

In the practical part, two systems were developed: the first using the **monolithic architecture**, and the second using the **microservices architecture**. The process of designing, implementing, component communication, and configuring the systems is discussed in detail. As part of the analysis, the performance metrics of both solutions were compared, and the costs of deploying the applications in the **Microsoft Azure** cloud environment were estimated. The final chapter includes conclusions and recommendations.

Keywords – monoliht, microservices, comparative analisys, performance metrics, monolithic architecture, microservices architecture, containerization

**Spis treści**

[**Wstęp** 9](#_Toc185075225)

[**1** **Przegląd literatury** 11](#_Toc185075226)

[**1.1** **Charakterystyka architektury monolitycznej** 11](#_Toc185075227)

[**1.2** **Charakterystyka architektury mikroserwis** 11](#_Toc185075228)

[**1.3** **Porównanie architektury monolitycznej oraz mikroserwisów** 12](#_Toc185075229)

[**2** **Wykorzystane rozwiązania** 15](#_Toc185075230)

[**2.1** **Języki programowania** 15](#_Toc185075231)

[**2.2** **Frameworki oraz biblioteki** 15](#_Toc185075232)

[**2.3** **Bazy danych i messaging** 15](#_Toc185075233)

[**2.4** **Konteneryzacja i orkiestracja** 16](#_Toc185075234)

[**2.5** **API Gateway** 17](#_Toc185075235)

[**3** **Implementacja** 19](#_Toc185075236)

[**3.1** **System w architekturze monolitycznej** 19](#_Toc185075237)

[**3.1.1** **Projekt systemu** 19](#_Toc185075238)

[**3.1.2** **Szczegóły Implementacji** 21](#_Toc185075239)

[**3.1.3** **Wnioski z implementacji systemu w architekturze monolitycznej** 28](#_Toc185075240)

[**3.2** **System w architekturze mikroserwisów** 29](#_Toc185075241)

[**3.2.1** **Dekompozycja na mikroserwisy** 29](#_Toc185075242)

[**3.2.2** **Komunikacja miedzy serwisami** 30](#_Toc185075243)

[**3.2.3** **Wnioski implementacji** 37](#_Toc185075244)

[**4** **Analiza architektur** 39](#_Toc185075245)

[**4.1** Metryki wydajnościowe 39](#_Toc185075246)

[**4.2** **Koszt wdrożenia aplikacji w chmurze Azure** 39](#_Toc185075247)

[5 Wnioski 42](#_Toc185075248)

[**Bibliografia** 43](#_Toc185075249)

[Spis Fragmentu Kodów 45](#_Toc185075250)

# **Wstęp**

W ciągu ostatnich lat na znaczeniu zyskały głównie dwie koncepcje architektoniczne: architektura monolityczna oraz mikroserwisowa. Wybór odpowiedniego podejścia może mieć kluczowe znaczenie dla sukcesu aplikacji, wpływając na elastyczność, skalowalność oraz łatwość utrzymania systemu.

Architektura monolityczna, będąca tradycyjnym podejściem do projektowania aplikacji, zakłada, że wszystkie komponenty systemu są zintegrowane w jedną całość. Architektura mikroserwisowa proponuje podział systemu na niezależne, współpracujące ze sobą serwisy. Każdy mikroserwis realizuje określoną funkcjonalność i może być rozwijany, wdrażany oraz skalowany niezależnie od innych

Wiele znanych firm rozpoczynało tworzenie swoich systemów od architektury monolitycznej, co w początkowych etapach rozwoju było rozwiązaniem prostym, logicznym oraz dającym zadowalający efekt. Przykładami takich firm są Netflix, Amazon, GitHub, Atlassian czy Basecamp [1]. Firma Netflix była jednym z pierwszych przedsiębiorstw, które zdało sobie sprawę, że architektura monolityczna może być niewystarczająca, i może nie być najlepszym rozwiązaniem w przypadku złożonych aplikacji. Architekci Netflixa, przez pojedynczy błąd z 2008 roku, który spowodował masową utratę danych i doprowadził do kilkudniowego przestoju, zdecydowali się przenieść swoją aplikację z architektury monolitycznej do mikroserwisów opartych na chmurze AWS [2]. Uznano, że migracja architektury, usprawni dostępność oraz skalowalność systemu. Podzielenie systemu na 700 mikroserwisów, z których każdy odpowiedzialny był za jedną funkcjonalność, umożliwiało dokonywanie zmian w dowolnej części systemu, mając pewność, że aplikacja nie zostanie wyłączona. Pomogło to w zminimalizowaniu braków dostępności do serwisu.

Wraz z dynamicznym rozwojem technologii oraz rosnącymi wymaganiami i potrzebami współczesnych aplikacji, pojawia się konieczność przeanalizowania, a także porównania architektury mikroserwisów oraz monolitycznej. Tradycyjne już podejście monolityczne, mimo swojej prostoty, często może nie sprostać wyzwaniom jakie może mieć aplikacja. Może ona okazać się niewystarczająca pod względem skalowalności, elastyczności czy szybkości wdrażania nowych, potrzebnych funkcji. Użycie architektury mikroserwisów, która pozwala na dekompozycję aplikacji na niezależnie rozwijalne komponenty, wymaga dokładnego zrozumienia, kiedy i dlaczego warto użyć właśnie jej, jako rozwiązania.

Analiza ta może być szczególnie istotna dla podmiotów stojących przed decyzją o modernizacji swojej architektury. Porównanie tych dwóch podejść pozwoli również zidentyfikować najlepsze praktyki oraz potencjalne pułapki, co jest kluczowe dla skutecznego wdrożenia nowoczesnych rozwiązań technologicznych.

Celem niniejszej pracy magisterskiej jest przeprowadzenie analizy porównawczej architektury monolitycznej oraz mikroserwisowej w kontekście ich odpowiedniości w różnych scenariuszach wdrożenia aplikacji. Proces badawczy będzie obejmował aspekty techniczne oraz funkcjonalne obu podejść, wraz z uwzględnieniem ich wpływu na skalowalność, wydajność, elastyczność, łatwość zarządzania oraz utrzymania systemu. Dodatkowymi aspektami porównawczymi będzie utylizacja zasobów oraz bezpieczeństwo. Praca ta będzie również skupiać się na identyfikacji sytuacji, w których każda z analizowanych architektur może być preferowana. Dodatkowo praca ta ma na celu dostarczenie praktycznych wskazówek dla specjalistów zajmującymi się architekturą oprogramowania w celu wyboru tej odpowiedniej w zależności od kontekstu projektu oraz jego wymagań.

Praca składa się z pięciu głównych rozdziałów, które szczegółowo analizują implementację systemu w dwóch architekturach: monolitycznej oraz mikroserwisowej. Rozdział 1 to przegląd literatury, w którym scharakteryzowano obie architektury oraz dokonano ich porównania. Rozdział 2 przedstawia wykorzystane rozwiązania technologiczne, takie jak języki programowania, frameworki, bazy danych, systemy messaging, narzędzia do konteneryzacji i API Gateway. Rozdział 3 skupia się na implementacji systemu w obu architekturach, omawiając szczegóły projektowe i wnioski wynikające z ich realizacji. W Rozdziale 4 przeprowadzono analizę obu podejść, porównując je pod względem wydajności oraz kosztów wdrożenia w chmurze Azure. Całość podsumowano w Rozdziale 5, gdzie przedstawiono wnioski końcowe dotyczące obu architektur.

# **Przegląd literatury**

## **Charakterystyka architektury monolitycznej**

Architektura monolityczna jest tradycyjnym podejściem do projektowania i budowania systemów informatycznych, w którym cała aplikacja funkcjonuje jako jednolity i nierozerwalny blok kodu. [3] W tego rodzaju architekturze wszystkie moduły systemu, takie jak interfejs użytkownika, logika biznesowa i warstwa danych, są ze sobą ściśle powiązane i uruchamiane wspólnie w ramach jednego procesu. Chris Richardson w swojej książce [4] wskazuje, że ta cecha wynika z fundamentalnego założenia monolitu - wszystkie funkcjonalności są kompilowane do jednego wykonywalnego pliku lub pakietu. Zintegrowana struktura monolitu sprawia, że aplikacja jest budowana, testowana i wdrażana jako jedna całość, co przekłada się na prostotę w początkowej fazie projektu.

Jedną z kluczowych zalet architektury monolitycznej jest jej prostota koncepcyjna i implementacyjna. Vaughn Vernon podkreśla, że cała aplikacja funkcjonująca w ramach jednego środowiska znacząco upraszcza procesy projektowania i implementacji, szczególnie dla małych zespołów programistycznych. Programiści mogą skupić się na rozwoju funkcjonalności bez konieczności zarządzania złożoną infrastrukturą czy komunikacją między komponentami. [5]

Architektura monolityczna ma jednak swoje ograniczenia, które stają się widoczne wraz ze wzrostem złożoności aplikacji. Jej skalowanie jest trudne i kosztowne, ponieważ wymaga skalowania całej aplikacji, nawet jeśli tylko jedna jej część wymaga większej wydajności. Utrzymanie staje się coraz bardziej problematyczne, ponieważ zależności między modułami sprawiają, że każda zmiana może wywołać nieprzewidziane problemy w innych częściach systemu. Dodatkowo, awaria jednego modułu może unieruchomić całą aplikację, co obniża niezawodność rozwiązania. Architektura monolityczna najlepiej sprawdza się w mniejszych projektach, które nie wymagają dużej skalowalności, takich jak wewnętrzne systemy ERP [6] czy proste aplikacje e-commerce.

## **Charakterystyka architektury mikroserwis**

Architektura mikroserwisowa to nowoczesne podejście do projektowania aplikacji, które zakłada podział systemu na wiele niezależnych usług, z których każda odpowiada za realizację jednej konkretnej funkcjonalności. Mikroserwisy działają jako odrębne jednostki, które mogą być rozwijane, wdrażane i skalowane niezależnie od siebie. Komunikacja między nimi odbywa się za pomocą lekkich protokołów, takich jak HTTP, gRPC czy komunikaty w kolejkach wiadomości. [7]

Główną zaletą architektury mikroserwisowej jest jej elastyczność i skalowalność. Dzięki niezależności poszczególnych usług można je skalować w zależności od obciążenia, co pozwala na efektywniejsze wykorzystanie zasobów. Dodatkowo, możliwość niezależnego wdrażania poszczególnych mikroserwisów przyspiesza wprowadzanie zmian i ułatwia dodawanie nowych funkcjonalności bez ryzyka destabilizacji całego systemu. Awaria jednego mikroserwisu zazwyczaj nie wpływa na działanie pozostałych, co zwiększa niezawodność aplikacji jako całości.

Architektura mikroserwisowa wymaga jednak większych nakładów na zarządzanie i utrzymanie. Rozproszona natura systemu sprawia, że niezbędne są zaawansowane narzędzia do monitorowania, orkiestracji (np. Kubernetes) i zarządzania komunikacją między usługami. Wdrożenie mikroserwisów wymaga także bardziej złożonej infrastruktury, co wiąże się z wyższymi kosztami początkowymi. Kolejnym wyzwaniem jest testowanie aplikacji, ponieważ integracja niezależnych usług może generować nieoczekiwane problemy. [8]

Architektura mikroserwisowa znajduje zastosowanie głównie w dużych, złożonych systemach, które wymagają skalowalności, niezawodności i elastyczności. Jest to popularne rozwiązanie w platformach streamingowych, takich jak Netflix, czy dużych serwisach e-commerce, takich jak Amazon, które muszą obsługiwać miliony użytkowników w sposób płynny i niezawodny. [9]

Wybór architektury mikroserwisowej pozwala na większą adaptacyjność do dynamicznych wymagań, jednak wiąże się z wyższymi kosztami i większą złożonością zarządzania.

## **Porównanie architektury monolitycznej oraz mikroserwisów**

W artykule [10] autor zgłębia się w długo toczącą się debatę pomiędzy architekturami monolitów oraz mikroserwisów. Omówione zostają powszechnie błędne przekonania na temat architektury mikroserwisów, twierdząc że nie są one cudownym środkiem ale raczej narzędziem do zarządzania złożonością nowoczesnych aplikacji. Autor podkreśla, że wszystkie aplikacje nieuchronnie stają się złożone, a mikroserwisy oferują sposób na złagodzenie zaistniałej złożoności poprzez możliwość podzielenia jej na mniejsze, oraz łatwiejsze do zarządzania fragmenty. Jednym z kluczowych argumentów autor uznał to, że rozmiar mikroserwisu ma mniejsze znaczenie niż jego skuteczność w zaspokajaniu konkretnych potrzeb aplikacji. Zamiast skupiać wyłącznie na tworzeniu jak najmniejszych usług, sugeruje on tworzenie serwisów które zapewniają równowagę pomiędzy złożonością a łatwością zarządzania. Ponadto w artykule podważono koncepcję dychotomii[[1]](#footnote-1) pomiędzy monolitami a mikroserwisami, przez co zaproponowany został model hybrydowy, łączący elementy obu architektur. Autor sugeruje, że taki model odzwierciedla rzeczywistość wielu aplikacji, w których współistnieje mieszanka komponentów opartych na podejściu monolitycznym oraz mikroserwisowym. Omówiona została także ewolucja mikroserwisów na przestrzeni czasu, gdzie wywnioskowane zostało, że postęp w narzędziach i platformach sprawił, że ten rodzaj architektury stał się dużo bardziej dostępny i przyjazny.

W ramach pracy [11] autorzy porównali wydajność samodzielnie opracowanego systemu opartego na mikroserwisach z jego monolitycznym odpowiednikiem, zbadali także wpływ konteneryzacji na wydajność systemu. Korzystając z JMeter [12] do symulacji użytkowników, stwierdzono, że system monolityczny miał krótszy czas reakcji oraz większą przepustowość, zużywając mniej pamięci RAM oraz procesora w porównaniu do systemu mikroserwisów. Konteneryzacja systemu, przy użyciu Dockera [13] nie zmniejszyła wydajności, wręcz wykazała poprawę zużycia pamięci RAM, procesora, przepustowości oraz zauważono mniejsze opóźnienia. Wykorzystanie Dockera na systemie opartym o mikroserwisy skutkowało podobnymi opóźnieniami, ale zwiększyło przepustowość nawet o 15% przy dużym obciążeniu systemu, jednakże kosztem użycia większej ilości pamięci RAM i procesora.

Kolejny artykuł [14] ocenia takie wskaźniki jak czas reakcji oraz przepustowość zarówno przy wysokim jak i niskim obciążeniu. Dodatkowo autorzy porównali skuteczność różnych technologii wykrywania usług (Service discovery technologies) [15] Eureka oraz Consul, dla architektur mikroserwisowych. Ich ustalenia wskazują, że przy typowych obciążeniach obie te architektury wykazują podobny poziom wydajności. Jednakże różnice stają się widoczne wraz ze wzrostem obciążenia, gdzie aplikacje monolityczne wykazują tendencję do wykazywania większej wydajności. W artykule tym podkreślono, że użycie architektury monolitycznej jest zalecane w przypadku, gdy celem programisty jest to by aplikacja obsługiwała żądania szybciej. Warto zauważyć, że pod względem przepustowości oraz liczby obsługiwanych żądań na sekundę, mikroserwisy wykorzystujące Consul do wykrywania usług, były o 4% lepsze niż te używające Eureki.

Autorzy [16] porównując architketury monolitów oraz mikroserwisów w swojej pracy zauważają, że architektura monolityczna zazwyczaj oferuje lepszą wydajność na pojedynczej maszynie niż mikroserwisy, ze względu na dodatkowe obciążenie związane z przekazywaniem żądań między komponentami.

W pracy [17] wykonano porównanie obu architektur przy użyciu narzędzia do testowania obciążenia Gatling [18]. Badania te zostały wykonane na maszynie z systemem operacyjnym Ubuntu 18.04.2 LTS, a aplikacje wdrożone zostały przy pomocy Dockera. Praca ta porównuje te architektury pod względem takim parametrów jak: wydajność i czas odpowiedzi. Testy obciążeniowe wykazały ze mikroserwisy są bardziej wydajne, gdy aplikacja musi przetworzyć więcej requestów. Natomiast monolity mogą być dobrym rozwiązaniem, w przypadku aplikacji o mniejszym obciążeniu, przy czym dodatkowo są prostsze do stworzenia. Autorzy wspominają, że wybór architektury powinien być zdefiniowany przez wymagania biznesowe, tak aby spełniało ono wymagania inwestorów.

## 

# **Wykorzystane rozwiązania**

Wybór opisanych poniżej technologii został podyktowany zarówno zainteresowaniami autora, jak i jego doświadczeniem w pracy z nimi. W celu zapewnienia wysokiej jakości kodu, opierano się na wskazówkach i rekomendacjach zawartych w książkach [19] [20]

## **Języki programowania**

**.Net oraz C#**

Platforma .NET to darmowe, otwarte środowisko programistyczne, które umożliwia tworzenie i uruchamianie aplikacji na różnych, popularnych platformach. W momencie rozpoczęcia pracy nad aplikacją najnowszą dostępną wersją .NET była wersja 8. [21] Język C# jest nowoczesnym, obiektowym językiem programowania, w którym kod jest kompilowany do CIL (Common Intermediate Language).

**Typescript**

Typescript to silnie typowany język programowania, który rozszerza możliwości JavaScript poprzez wprowadzenie statycznych typów. Kompiluje się do standardowego JavaScript, dzięki czemu może być używany we wszystkich nowoczesnych przeglądarkach oraz środowiskach uruchomieniowych. [22]

## **Frameworki oraz biblioteki**

**MassTransit**

MassTransit jest oprogramowaniem typu open-source dla platformy .NET, służącym do implementacji wzorców komunikacji między usługami. Framework ten umożliwia asynchroniczną komunikację oraz integrację między systemami przy użyciu wiadomości. Rozwiązanie to jest stosowanie w tworzeniu rozwiązań opartych na mikrousługach. [23]

**Vue.js 3**

Vue to progresywny framework JavaScript do budowania interfejsów użytkownika oraz aplikacji jednostronicowych (SPA – Single Page Applications). Cechuje się wydajnością, łatwością użycia oraz elastycznością. [24]

## **Bazy danych i messaging**

**Microsoft SQL Server**

Microsoft SQL Server relacyjny system zarządzania bazą danych (RDBMS) stworzony przez Microsoft. Charakteryzuje się wysoką wydajnością, niezawodnością i skalowalnością. Wykorzystuje język T-SQL do zarządzania danymi i zapytań. SQL Server doskonale integruje się z technologiami Microsoft, co czyni go popularnym rozwiązaniem w aplikacjach biznesowych i korporacyjnych [25]

**RabbitMQ**

RabbitMQ jest systemem kolejkowania wiadomości (message broker) opartym na protokole AMQP, czyli standardowym protokole komunikacyjnym, służącym do niezawodnego przesyłania wiadomości między systemami. [26] Umożliwia asynchroniczną komunikację między usługami, zapewniając niezawodne przesyłanie, kolejkowanie i odbieranie wiadomości. Charakteryzuje się wysoką wydajnością, skalowalnością i wsparciem dla wielu języków programowania, dzięki czemu jest często stosowany w systemach rozproszonych i mikroserwisach. [27]

## **Konteneryzacja i orkiestracja**

**Docker**

Docker jest platformą do tworzenia, uruchamiania i zarządzania kontenerami, które izolują aplikacje wraz ze wszystkimi ich zależnościami. Dzięki temu zapewnia spójność działania aplikacji w różnych środowiskach, takich jak lokalne maszyny, serwery czy chmura. Docker jest lekki, szybki i umożliwia efektywne zarządzanie zasobami. [28]

**Docker Compose**

Docker Compose jest narzędziem umożliwiającym definiowanie i uruchamianie aplikacji składających się z wielu kontenerów za pomocą jednego pliku konfiguracyjnego YAML. Ułatwia to zarządzanie skomplikowanymi środowiskami, w których poszczególne usługi działają w oddzielnych kontenerach. Narzędzie to pozwala na szybkie wdrażanie, skalowanie i testowanie aplikacji wielokontenerowych. [29]

## **API Gateway**

API Gateway jest pośrednikiem zarządzającym ruchem między klientami, a usługami w systemach opartych na mikroserwisach. Służy jako pojedynczy punkt wejścia dla żądań API, umożliwiając tym przekierowanie do odpowiednich usług. [30]

**Ocelot**

Ocelot jest to API Gateway dla platformy .NET, który służy jako brama wejściowa do mikroserwisów, zarządzając routingiem żądań, agregacją, loadbalancingiem oraz dostarczając funkcje takie jak cache'owanie czy rate limiting. Jest to narzędzie open-source, które pozwala na łatwe konfigurowanie tras za pomocą pliku JSON, dzięki czemu można w prosty sposób przekierowywać ruch z jednego punktu wejścia do różnych mikroserwisów. [31]

# **Implementacja**

Implementacja została zrealizowana etapowo. Pierwszym krokiem została stworzona wersja monolityczna, która pozwoliła na szybkie opracowanie podstawowej funkcjonalności oraz przetestowanie głównych założeń systemu. Model ten umożliwił na łatwiejsze zrozumienie logiki biznesowej oraz eliminację potencjalnych błędów na wczesnym etapie projektu. Następnie, zgodnie z podejściem rekomendowanym w literaturze [32], przeprowadzona została dekompozycja monolitu na niezależne mikroserwisy. Proces dekompozycji polegał na identyfikacji kluczowych komponentów systemu, które następnie zostały wydzielone jako odrębne serwisy.

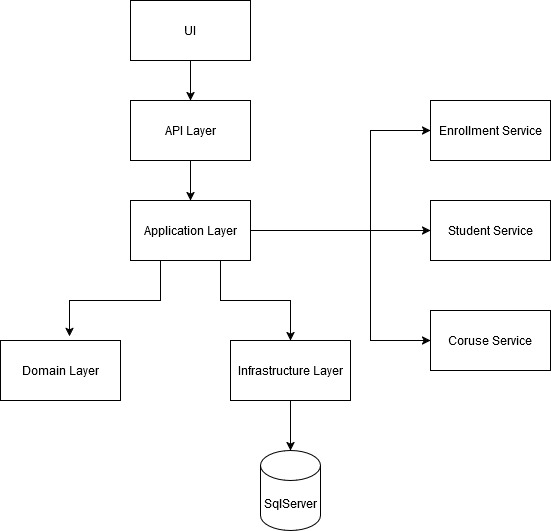
## **System w architekturze monolitycznej**

### **Projekt systemu**

Projekt systemu w architekturze monolitycznej został zrealizowany w technologii .NET 8.0, wykorzystując wzorzec warstwowy (Rys.1). System składa się z czterech głównych projektów:

* **LMS.Core** - zawiera modele domenowe i interfejsy
* **LMS.Infrastructure** - zawiera implementację dostępu do danych
* **LMS.Application** - zawiera logikę biznesową i serwisy
* **LMS.API** - warstwa prezentacji, kontrolery API

Każda warstwa pełni określoną rolę i ma dostęp tylko do warstw niższego poziomu. Taki podział pozwala na zachowanie separacji zagadnieńi ułatwia utrzymanie kodu.



Rys. 1 Diagram architektury aplikacji monolitycznej

### **Szczegóły Implementacji**

**Warstwa dostępu do danych**

Przy projektowaniu systemu przyjęto podejście "database-first", gdzie najpierwzaprojektowano strukturę bazy danych zawierającą tabele:

* Courses - przechowującą informacje o kursach
* Students - zawierającą dane studentów
* Enrollments - przechowującą zapisy studentów na kursy
* CoursePrerequisites - definiującą wymagania wstępne dla kursów

W ramach implementacji warstwy dostępu do danych wykorzystano Entity Framework Core jako ORM. Zdefiniowano kontekst bazy danych ApplicationContext (Kod 1), który zawiera konfigurację wszystkich encji oraz ich relacji.

1. public class ApplicationContext : DbContext

2. {

3. public ApplicationContext(DbContextOptions<ApplicationContext> options) : base(options)

4. {

5. }

6.

7. public DbSet<Course> Courses { get; set; }

8. public DbSet<Student> Students { get; set; }

9. public DbSet<Enrollment> Enrollments { get; set; }

10. public DbSet<CoursePrerequisite> CoursePrerequisites { get; set; }

11.

12. protected override void OnModelCreating(ModelBuilder modelBuilder)

13. {

14. // Course Configuration

15. modelBuilder.Entity<Course>(entity =>

16. {

17. entity.HasKey(e => e.Id);

18. entity.Property(e => e.Title).IsRequired().HasMaxLength(100);

19. entity.Property(e => e.Description).HasMaxLength(500);

20. entity.Property(e => e.MaxEnrollment).IsRequired();

21. entity.Property(e => e.CurrentEnrollment).IsRequired();

22.

23. // Course - Prerequisites relationship

24. entity.HasMany(e => e.Prerequisites)

25. .WithOne(p => p.Course)

26. .HasForeignKey(p => p.CourseId)

27. .OnDelete(DeleteBehavior.Restrict);

28. });

29.

30. // Student Configuration

31. modelBuilder.Entity<Student>(entity =>

32. {

33. entity.HasKey(e => e.Id);

34. entity.Property(e => e.FirstName).IsRequired().HasMaxLength(50);

35. entity.Property(e => e.LastName).IsRequired().HasMaxLength(50);

36. entity.Property(e => e.Email).IsRequired().HasMaxLength(100);

37. entity.HasIndex(e => e.Email).IsUnique();

38. });

39.

40. // Enrollment Configuration

41. modelBuilder.Entity<Enrollment>(entity =>

42. {

43. entity.HasKey(e => e.Id);

44. entity.Property(e => e.EnrollmentDate).IsRequired();

45. entity.Property(e => e.Status).IsRequired();

46.

47. // Relationships

48. entity.HasOne(e => e.Student)

49. .WithMany(s => s.Enrollments)

50. .HasForeignKey(e => e.StudentId)

51. .OnDelete(DeleteBehavior.Restrict);

52.

53. entity.HasOne(e => e.Course)

54. .WithMany(c => c.Enrollments)

55. .HasForeignKey(e => e.CourseId)

56. .OnDelete(DeleteBehavior.Restrict);

57.

58. // Unique constraint for student-course combination

59. entity.HasIndex(e => new { e.StudentId, e.CourseId }).IsUnique();

60. });

61.

62. // CoursePrerequisite Configuration

63. modelBuilder.Entity<CoursePrerequisite>(entity =>

64. {

65. entity.HasKey(e => e.Id);

66. entity.Property(e => e.IsMandatory).IsRequired();

67.

68. // Relationships

69. entity.HasOne(p => p.PrerequisiteCourse)

70. .WithMany()

71. .HasForeignKey(p => p.PrerequisiteCourseId)

72. .OnDelete(DeleteBehavior.Restrict);

73.

74. // Unique constraint for course-prerequisite combination

75. entity.HasIndex(p => new { p.CourseId, p.PrerequisiteCourseId }).IsUnique();

76. });

77. }

78. }

Kod 1 Application Context aplikacji monolitycznej

Szczególną uwagę poświęcono mapowaniu relacji między kursami a ich wymaganiami wstępnymi, gdzie zastosowano relację wiele-do-wielu z dodatkową tabelą *CoursePrerequisites*. Zaimplementowano również mechanizm migracji, który pozwala na wersjonowanie schematu bazy danych.

1. public class CoursePrerequisite

2. {

3. public int Id { get; set; }

4. public int CourseId { get; set; }

5. public int PrerequisiteCourseId { get; set; }

6. public bool IsMandatory { get; set; }

7. public Course Course { get; set; }

8. public Course PrerequisiteCourse { get; set; }

9. }

10.

Kod 2 Klasa CoursePrerequisite

**Modele domenowe**

Zdefiniowano kluczowe modele systemu: Course, Student oraz Enrollment. Każdy z nich posiada własne właściwości oraz logikę biznesową. W modelu Course znajdują się informacje na temat kursów (Kod 2). Model Student (Kod 3) zawiera podstawowe informacje o uczestnikach kursów, natomiast Enrollment (Kod 4) reprezentuje zapis na kurs wraz ze statusem i datą zapisu.

1. public class Course

2. {

3. public int Id { get; set; }

4. public string Title { get; set; }

5. public string Description { get; set; }

6. public int MaxEnrollment { get; set; }

7. public int CurrentEnrollment { get; set; }

8. public DateTime CreatedDate { get; set; }

9. public DateTime? UpdatedDate { get; set; }

10. public ICollection<CoursePrerequisite> Prerequisites { get; set; }

11. public ICollection<Enrollment> Enrollments { get; set; }

12. }

13.

Kod 3 Klasa Course

1. public class Student

2. {

3. public int Id { get; set; }

4. public string FirstName { get; set; }

5. public string LastName { get; set; }

6. public string Email { get; set; }

7. public DateTime DateOfBirth { get; set; }

8. public DateTime CreatedDate { get; set; }

9. public DateTime? UpdatedDate { get; set; }

10. public ICollection<Enrollment> Enrollments { get; set; }

11. }

12.

Kod 4 Klasa Student

1. public class Enrollment

2. {

3. public int Id { get; set; }

4. public int StudentId { get; set; }

5. public int CourseId { get; set; }

6. public DateTime EnrollmentDate { get; set; }

7. public EnrollmentStatus Status { get; set; }

8. public Student Student { get; set; }

9. public Course Course { get; set; }

10. }

11.

Kod 5 Klasa Enrollment

**Serwisy aplikacyjne**

Warstwa serwisów zawiera główną logikę biznesową systemu. CourseService (Kod 6) odpowiada za zarządzanie kursami, StudentService (Kod 8) obsługuje operacje związane ze studentami, a EnrollmentService (Kod 7) implementuje złożoną logikę zapisów na kursy, włącznie z weryfikacją wymagań wstępnych i dostępności miejsc. W serwisach zaimplementowano obsługę transakcji, zapewniając spójność danych przy operacjach modyfikujących wiele encji.

1. public class CourseService : ICourseService

2. {

3. private readonly ApplicationContext \_context;

4. private readonly IMapper \_mapper;

5.

6. public CourseService(ApplicationContext context, IMapper mapper)

7. {

8. \_context = context;

9. \_mapper = mapper;

10. }

11. // Wybrane metody w CourseService

12. public async Task<CourseDto> CreateCourseAsync(CreateCourseDto courseDto)

13. {

14. var course = \_mapper.Map<Course>(courseDto);

15. \_context.Courses.Add(course);

16. await \_context.SaveChangesAsync();

17. return \_mapper.Map<CourseDto>(course);

18. }

19. public async Task<IEnumerable<CourseDto>> GetAllCoursesAsync()

20. {

21. var courses = await \_context.Courses.ToListAsync();

22. return \_mapper.Map<IEnumerable<CourseDto>>(courses);

23. }

24. public async Task<IEnumerable<PrerequisiteDto>> GetPrerequisitesAsync(int courseId)

25. {

26. return await \_context.CoursePrerequisites

27. .Where(p => p.CourseId == courseId)

28. .Select(p => new PrerequisiteDto

29. {

30. CourseId = p.CourseId,

31. PrerequisiteCourseId = p.PrerequisiteCourseId,

32. PrerequisiteCourseName = p.PrerequisiteCourse.Title,

33. IsMandatory = p.IsMandatory

34. })

35. .ToListAsync();

36. }

37. }

38.

Kod 6 Wybrane metody w CourseService

1. public class EnrollmentService : IEnrollmentService

2. {

3. private readonly ApplicationContext \_context;

4. private readonly IMapper \_mapper;

5.

6. public EnrollmentService(ApplicationContext context, IMapper mapper)

7. {

8. \_context = context;

9. \_mapper = mapper;

10. }

11.

12.

13. public async Task<EnrollmentDto> CreateEnrollmentAsync(CreateEnrollmentDto enrollmentDto)

14. {

15. var prerequisites = await \_context.CoursePrerequisites

16. .Where(p => p.CourseId == enrollmentDto.CourseId)

17. .ToListAsync();

18.

19. var studentEnrollments = await \_context.Enrollments

20. .Where(e => e.StudentId == enrollmentDto.StudentId && e.Status == EnrollmentStatus.Completed)

21. .Select(e => e.CourseId)

22. .ToListAsync();

23.

24. foreach (var prerequisite in prerequisites.Where(p => p.IsMandatory))

25. {

26. if (!studentEnrollments.Contains(prerequisite.PrerequisiteCourseId))

27. {

28. throw new ApplicationException("Prerequisites not met");

29. }

30. }

31.

32. var enrollment = new Enrollment

33. {

34. StudentId = enrollmentDto.StudentId,

35. CourseId = enrollmentDto.CourseId,

36. EnrollmentDate = DateTime.UtcNow,

37. Status = EnrollmentStatus.Active

38. };

39.

40. \_context.Enrollments.Add(enrollment);

41. await \_context.SaveChangesAsync();

42.

43. return await GetEnrollmentAsync(enrollment.Id);

44. }

45.

46. public async Task<IEnumerable<EnrollmentDto>> GetAllEnrollmentsAsync()

47. {

48. var enrollments = await \_context.Enrollments.ToListAsync();

49.

50. return \_mapper.Map<IEnumerable<EnrollmentDto>>(enrollments);

51. }

52. }

53.

Kod 7 Wybrane metody EnrollmentService

1. public class StudentService : IStudentService

2. {

3. private readonly ApplicationContext \_context;

4. private readonly IMapper \_mapper;

5. public StudentService(ApplicationContext context, IMapper mapper)

6. {

7. \_context = context;

8. \_mapper = mapper;

9. }

10. public async Task<StudentDto> CreateStudentAsync(CreateStudentDto studentDto)

11. {

12. var student = \_mapper.Map<Student>(studentDto);

13.

14. \_context.Students.Add(student);

15. await \_context.SaveChangesAsync();

16.

17. return \_mapper.Map<StudentDto>(student);

18. }

19. public async Task<StudentDto> GetStudentByIdAsync(int id)

20. {

21. var student = await \_context.Students.FindAsync(id);

22. return \_mapper.Map<StudentDto>(student);

23. }

24.

25. public async Task<IEnumerable<EnrollmentDto>> GetStudentEnrollmentsAsync(int studentId)

26. {

27. var enrollments = await \_context.Enrollments

28. .Include(e => e.Course) // Include course details

29. .Include(e => e.Student) // Include student details

30. .Where(e => e.StudentId == studentId)

31. .Select(e => new EnrollmentDto

32. {

33. Id = e.Id,

34. StudentId = e.StudentId,

35. CourseId = e.CourseId,

36. StudentName = $"{e.Student.FirstName} {e.Student.LastName}",

37. CourseName = e.Course.Title,

38. EnrollmentDate = e.EnrollmentDate,

39. Status = e.Status.ToString()

40. })

41. .ToListAsync();

42.

43. return enrollments;

44. }

45. }

46.

Kod 8 Wybrane metody StudentService

**Kontrolery API**

Zaimplementowano kontrolery REST API dla każdego głównego modułu systemu. Kontrolery odpowiadają za przyjmowanie żądań HTTP, walidację danych wejściowych oraz przekazywanie ich do odpowiednich serwisów. Dla przykładu na figurze Kod 9 znajduje się implementacja kontrolera dla kursów.

1.

2. [Route("api/[controller]")]

3. [ApiController]

4. public class CourseController : ControllerBase

5. {

6. private readonly ICourseService \_courseService;

7. private readonly ILogger<CourseController> \_logger;

8.

9. public CourseController(ICourseService courseService, ILogger<CourseController> logger)

10. {

11. \_courseService = courseService;

12. \_logger = logger;

13. }

14.

15. [HttpGet]

16. public async Task<ActionResult<IEnumerable<CourseDto>>> GetCourses()

17. {

18. var courses = await \_courseService.GetAllCoursesAsync();

19. return Ok(courses);

20. }

21.

22. [HttpGet("{id}")]

23. public async Task<ActionResult<CourseDto>> GetCourse(int id)

24. {

25. var course = await \_courseService.GetCourseByIdAsync(id);

26. if (course == null) return NotFound();

27. return Ok(course);

28. }

29.

30. [HttpPost]

31. public async Task<ActionResult<CourseDto>> CreateCourse(CreateCourseDto courseDto)

32. {

33. try

34. {

35. var course = await \_courseService.CreateCourseAsync(courseDto);

36. return CreatedAtAction(nameof(GetCourse), new { id = course.Id }, course);

37. }

38. catch (Exception ex)

39. {

40. \_logger.LogError(ex, "Error creating course");

41. return BadRequest(ex.Message);

42. }

43. }

44.

45. [HttpGet("{id}/prerequisites")]

46. public async Task<ActionResult<IEnumerable<PrerequisiteDto>>> GetPrerequisites(int id)

47. {

48. var prerequisites = await \_courseService.GetPrerequisitesAsync(id);

49. return Ok(prerequisites);

50. }

51.

52. [HttpPost("{id}/prerequisites")]

53. public async Task<ActionResult<PrerequisiteDto>> AddPrerequisite(int id, AddPrerequisiteDto prerequisiteDto)

54. {

55. try

56. {

57. var prerequisite = await \_courseService.AddPrerequisiteAsync(id, prerequisiteDto);

58. return Ok(prerequisite);

59. }

60. catch (Exception ex)

61. {

62. \_logger.LogError(ex, "Error adding prerequisite");

63. return BadRequest(ex.Message);

64. }

65. }

66. }

67.

Kod 9 Implementacja kontrolera CourseController

Zastosowano wzorzec DTO (Data Transfer Objects) do bezpiecznej komunikacji między klientem a serwerem. Przykładowy model DTO znajduje się na Kod 10.

1. public class CourseDto

2. {

3. public int Id { get; set; }

4. public string Title { get; set; }

5. public string Description { get; set; }

6. public int MaxEnrollment { get; set; }

7. public int CurrentEnrollment { get; set; }

8. }

9.

Kod 10 CourseDTO

**Konfiguracja aplikacji**

W ramach konfiguracji aplikacji zdefiniowano wszystkie zależności w kontenerze DI. (Kod 11, Kod 12)

1. builder.Services.AddInfrastructure(builder.Configuration);

2. builder.Services.AddApplication();

Kod 11 Rejestracja zależności DI

1. public static IServiceCollection AddApplication(this IServiceCollection services)

2. {

3. services.AddAutoMapper(typeof(MappingProfile).Assembly);

4. services.AddScoped<ICourseService, CourseService>();

5. services.AddScoped<IStudentService, StudentService>();

6. services.AddScoped<IEnrollmentService, EnrollmentService>();

7.

8. return services;

9. }

10.

Kod 12 Rejestracja zależnosci DI serwisów aplikacji

Skonfigurowano middleware pipeline. Polega to na dodaniu komponentów middleware do przetwarzania żądań (Kod 13).

1. // Configure the HTTP request pipeline.

2. if (app.Environment.IsDevelopment())

3. {

4. app.UseSwagger();

5. app.UseSwaggerUI();

6. }

7. app.UseCors("AllowVueApp");

8. app.UseHttpsRedirection();

9. app.UseAuthorization();

10. app.MapControllers();

11.

Kod 13 Konfiguracja middleware pipeline

Do ustawienia połączenia z bazą danych zastosowano wzorzec Options Pattern do zarządzania konfiguracją aplikacji. (Kod 14, Kod 15)

1. public static IServiceCollection AddInfrastructure(this IServiceCollection services,IConfiguration configuration)

2. {

3. services.AddDbContext<ApplicationContext>(options =>

4. options.UseSqlServer(

5. configuration.GetConnectionString("DefaultConnection"),

6. b => b.MigrationsAssembly(typeof(ApplicationContext).Assembly.FullName)

7. )

8. );

9.

10. services.AddScoped<DataSeeder>();

11. return services;

12. }

13.

Kod 14 Wstrzykniecie zależnosci bazodanowej

1. "ConnectionStrings": {

2. "DefaultConnection": "Server=(localdb)\\MSSQLLocalDB;Database=LMS.Monolith;Trusted\_Connection=True;MultipleActiveResultSets=true"

3. },

4.

Kod 15 Fragment konfiguracji do bazy danych

### **Wnioski z implementacji systemu w architekturze monolitycznej**

**Prosta implementacja i zarządzanie**

W architekturze monolitycznej cała logika biznesowa, interfejs użytkownika i warstwa danych są zawarte w jednej, spójnej aplikacji. To znacząco upraszcza proces tworzenia, testowania i wdrażania. Brak konieczności koordynowania komunikacji między wieloma usługami sprawia, że monitorowanie i debugowanie błędów są prostsze i bardziej bezpośrednie. Wymagania dotyczące wdrożenia są znacznie mniejsze – najczęściej wystarczy pojedynczy serwer lub maszyna wirtualna, co redukuje koszty związane z infrastrukturą.

**Spójność danych**

Monolityczne systemy korzystają z jednej, centralnej bazy danych, co ułatwia zarządzanie transakcjami. Brak potrzeby synchronizacji danych między rozproszonymi bazami redukuje ryzyko niespójności.

**Szybsze wdrażanie początkowej wersji**

W architekturze monolitycznej zespoły programistyczne mogą szybciej stworzyć prototyp systemu i wdrożyć pierwszą wersję aplikacji, ponieważ wszystkie komponenty są zintegrowane w jednej aplikacji. Brak potrzeby koordynacji pracy między wieloma serwisami sprzyja krótszemu czasowi dostarczenia.

**Wydajność w przypadku mniejszych aplikacji**

Monolit często może oferować lepszą wydajność w przypadku mniejszych systemów, ponieważ komunikacja między komponentami odbywa się lokalnie (w pamięci) i nie wymaga dodatkowych protokołów sieciowych. Brak narzutu związanego z sieciową komunikacją między usługami sprzyja szybszemu przetwarzaniu operacji.

**Złożoność i wyzwania związane z rozwojem dużych systemów**

W przypadku dużych aplikacji monolitycznych skalowanie systemu staje się trudne, ponieważ cała aplikacja musi być uruchomiona w całości. Skalowanie pojedynczych komponentów nie jest możliwe. W miarę rozwoju aplikacji kod źródłowy staje się coraz większy i bardziej skomplikowany, co może prowadzić do powstawania "Big Ball of Mud" [33] – chaotycznej, trudnej do zrozumienia i rozwijania struktury kodu. W monolicie wszystkie komponenty są związane z jedną technologią i środowiskiem, co ogranicza możliwość wprowadzania nowych narzędzi czy rozwiązań. Każda zmiana w kodzie wymaga ponownego wdrożenia całej aplikacji, co może wydłużyć proces deploymentu i zwiększyć ryzyko awarii.

**Mniejsza odporność na błędy**

Ponieważ wszystkie komponenty działają w ramach jednej aplikacji, awaria jednego modułu może wpłynąć na działanie całego systemu, co zmniejsza odporność na błędy.

W przypadku dużych aplikacji trudno jest także odizolować problematyczny fragment systemu i wdrożyć szybkie poprawki.

## **System w architekturze mikroserwisów**

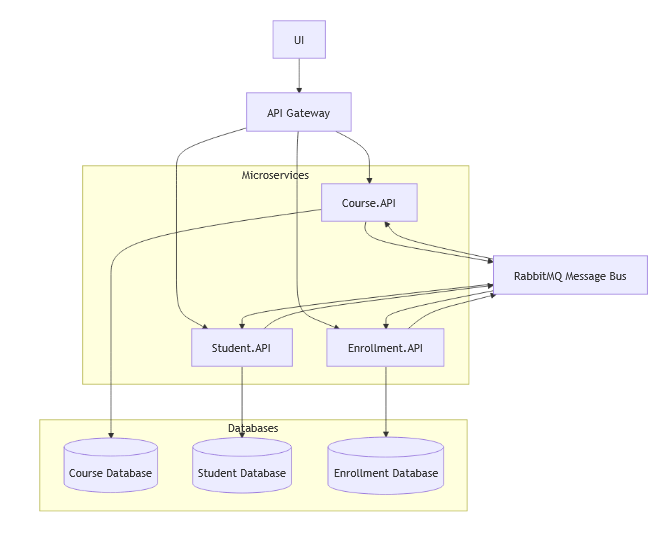
### **Dekompozycja na mikroserwisy**

Proces dekompozycji aplikacji monolitycznej na aplikacje mikroserwisową stanowił kluczowy etap w ewolucji systemu. Dekompozycja ta wymagała dokładnej analizy istniejącego monolitu oraz zaplanowania strategii podziału na niezależne komponenty. Głównym celem było przekształcenie pojedynczej aplikacji, na zestaw autonomicznych serwisów, z których każdy odpowiada za konkretną domenę biznesową.

Analiza dekompozycji, poza aspektami technicznymi, obejmowała przede wszystkim logike biznesową oraz wzajemne zależności między komponentami. Każdy wydzielony serwis został zaprojektowany tak, aby mógł działać niezależnie, posiadając własną bazę danych oraz jasno zdefiniowany interfejs do komunikacji.

Efektem tego procesu było wydzielenie trzech niezależnych serwisów:

Course Serivce, Student Service oraz Enrollment Service. Każdy z tych serwisów otrzymał własną bazę danych. W celu komunikacji między serwisami należało zastosować RabbitMQ, aby serwisy te były w stanie informować się o zdarzeniach wzajemnie na siebie wpływających. Diagram aplikacji mikroserwisowej został przedstawiony na Rys. 2.



Rys. 2 Diagram aplikacji mikroserwisowej

### **Komunikacja miedzy serwisami**

Komunikacja między komponentami w monolicie była prosta oraz bezpośrednia przede wszystkim dzięki wspólnej bazie danych. W tym przypadku każdy z serwisów miał bezpośredni dostęp do całej bazy, przez co istniała możliwość tworzenia złożonych zapytań łączących wiele tabel. (Kod 16)

1. public async Task<CourseDetails> GetCourseDetails(int courseId) {

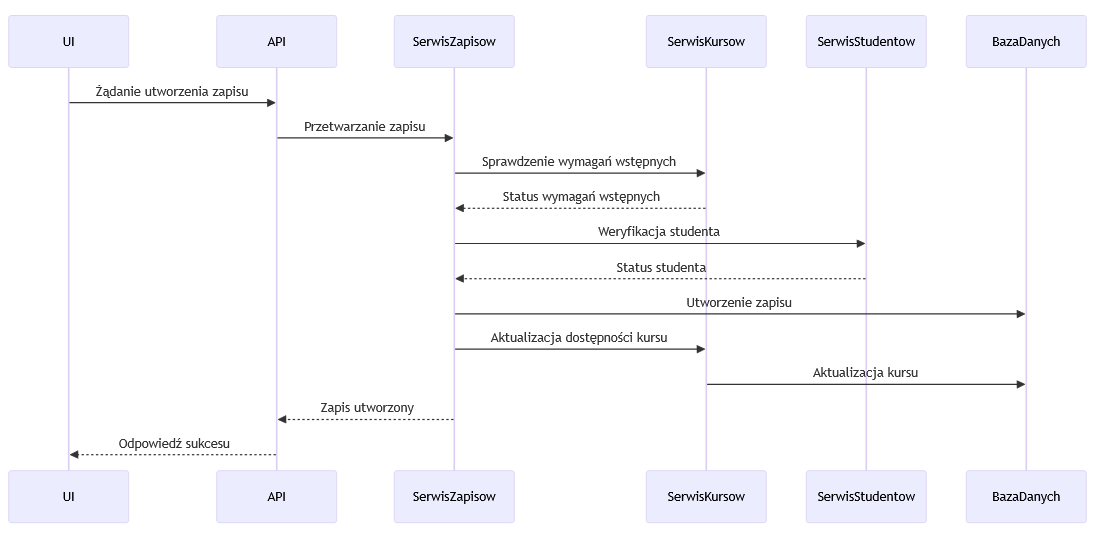
2. return await \_context.Courses .Include(c => c.Students) .Include(c => c.Enrollments) .FirstOrDefaultAsync(c => c.Id == courseId);

3. }

4.

Kod 16 Przykład zapytania łączonego wiele tabel

Przykładem tego jest proces zapisu studenta na kurs przedstawiony na Rys. 3 Przepływ danych dla procesu zapisu studenta na kurs w aplikacji monolitycznej.



Rys. 3 Przepływ danych dla procesu zapisu studenta na kurs w aplikacji monolitycznej

Diagram przedstawia proces zapisu studenta na kurs w ramach aplikacji monolitycznej, ukazując interakcje pomiędzy klientem, API, poszczególnymi warstwami serwisowymi oraz bazą danych.

1. Żądanie od UI

Proces rozpoczyna się od wysłania przez UI żądania zapisu studenta na kurs do warstwy API. Żądanie to zawiera dane dotyczące studenta oraz kursu, na który ma być zapisany.

1. Przetwarzanie zapisu w serwisie zapisów

API deleguje żądanie do Serwisu Zapisów (EnrollmentService), który pełni główną rolę w zarządzaniu logiką zapisu.

1. Sprawdzenie wymagań wstępnych

Serwis Zapisów komunikuje się z Serwisem Kursów (CourseService), aby sprawdzić, czy student spełnia wymagania wstępne dla wybranego kursu.  
Serwis Kursów zwraca status wymagań wstępnych do Serwisu Zapisów

1. Weryfikacja studenta

Jeśli wymagania wstępne są spełnione, Serwis Zapisów odwołuje się do Serwisu Studentów (StudentService) w celu weryfikacji statusu studenta (np. czy student istnieje i ma uprawnienia).

Serwis Studentów zwraca wynik weryfikacji do Serwisu Zapisów.

1. Utworzenie zapisu w bazie danych:

Po pozytywnej weryfikacji studentów i wymagań wstępnych, Serwis Zapisów tworzy nowy zapis w Bazie Danych. Informacje o zapisach są tam zapisywane jako nowy rekord.

1. Aktualizacja dostępności kursu

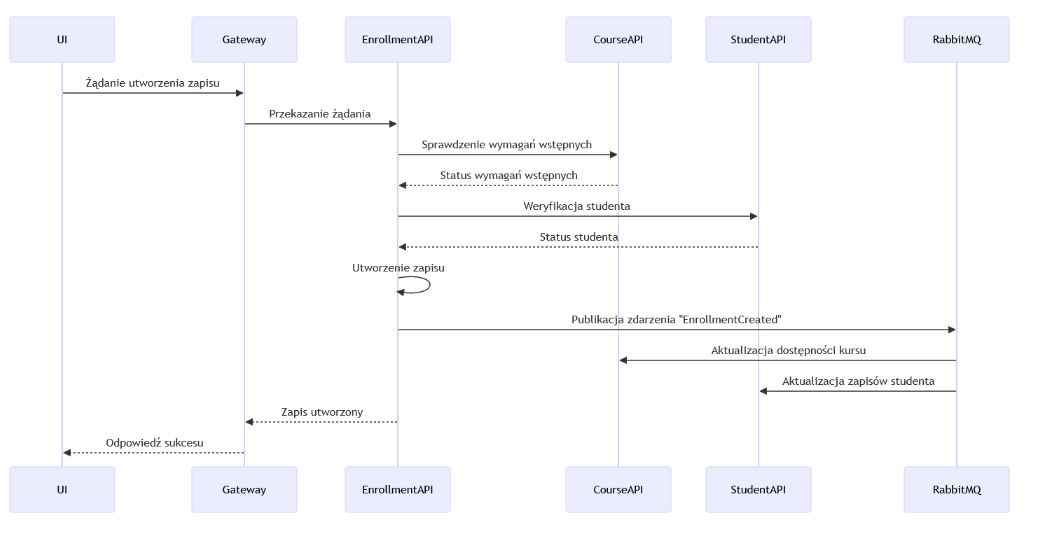
Następnie Serwis Zapisów informuje Serwis Kursów o konieczności zmniejszenia dostępnej liczby miejsc na kursie.

Serwis Kursów aktualizuje te dane w Bazie Danych, zmniejszając liczbę dostępnych miejsc.

1. Odpowiedź

Gdy proces zakończy się Serwis Zapisów wysyła potwierdzenie do API, które następnie przekazuje odpowiedź sukcesu do Klienta

Proces ten musiał znacznie się zmienić gdy aplikacja została zmieniona na architekturę mikroserwisową. (Rys. 4)



Rys. 4 Przepływ danych zapisu studenta na kurs w aplikacji mikroserwisowej

1. Żądanie utworzenia zapisu przez klienta

Proces rozpoczyna się, gdy Klient (interfejs użytkownika) wysyła żądanie utworzenia nowego zapisu na kurs do Gateway. Gateway pełni rolę punktu wejściowego (API Gateway), który odbiera żądanie i przekazuje je do odpowiedniego serwisu backendowego.

1. Przekazanie żądania do EnrollmentAPI

Gateway przekazuje żądanie do mikroserwisu EnrollmentAPI. Serwis ten jest odpowiedzialny za logikę zapisu studenta na kursSprawdzenie wymagań wstępnych.

1. Sprawdzenie wymagań wstępnych

EnrollmentAPI komunikuje się z CourseAPI za pomocą HTTP, aby sprawdzić, czy student spełnia wymagania wstępne dla wybranego kursu. CourseAPI zwraca status wymagań wstępnych do EnrollmentAPI.

1. Weryfikacja studenta:

Po uzyskaniu statusu wymagań, EnrollmentAPI wysyła zapytanie do StudentAPI w celu weryfikacji studenta (czy student istnieje w systemie i ma prawo do zapisu).

1. Utworzenie zapisu:

Gdy wszystkie weryfikacje przebiegną pomyślnie (wymagania i status studenta są poprawne), EnrollmentAPI tworzy zapis we własnym kontekście

1. Publikacja zdarzenia do RabbitMQ

EnrollmentAPI publikuje zdarzenie EnrollmentCreated do systemu kolejkowego RabbitMQ. RabbitMQ umożliwia asynchroniczną komunikację między mikroserwisami, zapewniając luźne powiązania między nimi

1. Aktualizacja stanu w innych serwisach

RabbitMQ przesyła zdarzenie do CourseAPI, który aktualizuje dostępność miejsc na kursie (zmniejsza liczbę dostępnych miejsc). Przesłane jest także zdarzenie do StudentAPI, który aktualizuje informacje o zapisach studenta.

1. Odpowiedz sukcesu

Po zakończeniu wszystkich operacji, EnrollmentAPI przesyła odpowiedź do Gateway, który przekazuje komunikat sukcesu z powrotem do UI.

Kluczową zmianą był sposób informowania innych serwisów o zmianach. Poprzednio z powodu jednej bazy danych, aktualizacja informacji była bezpośrednia. W obecnym rozwiązaniu wykorzystano dwa sposoby komunikowania się z innymi mikroserwisami:

1. **Komunikacja synchroniczna (REST API)**

Kod 17 ilustruje komunikację między mikroserwisami przy użyciu protokołu HTTP. W architekturze mikroserwisowej usługi takie jak Course.API i Student.API działają jako niezależne serwisy, które udostępniają swoje funkcjonalności za pośrednictwem endpointów HTTP. Metoda CheckPrerequisites sprawdza, czy student spełnia wymagania wstępne dla danego kursu, korzystając z dwóch zewnętrznych API: Course.API i Student.API. Za pomocą klienta HTTP metoda wysyła żądanie GET do Course.API i pobiera listę wymagań wstępnych kursu. Kolejne żądanie GET do Student.API zwraca listę identyfikatorów kursów ukończonych przez studenta.

Przy użyciu LINQ sprawdzane jest, czy wszystkie obowiązkowe wymagania wstępne znajdują się na liście ukończonych kursów. W przypadku problemów z komunikacją z API rzucany jest wyjątek ServiceUnavailableException. Metoda zwraca true, jeśli wymagania są spełnione, lub false, jeśli nie.

1. private async Task<bool> CheckPrerequisites(int studentId, int courseId)

2. {

3. try

4. {

5. // Get prerequisites from Course.API

6. var courseClient = \_clientFactory.CreateClient("CourseAPI");

7. var prerequisitesResponse = await courseClient

8. .GetAsync($"/api/course/{courseId}/prerequisites");

9.

10. if (!prerequisitesResponse.IsSuccessStatusCode)

11. throw new ServiceUnavailableException("Could not get prerequisites");

12.

13. var prerequisites = await prerequisitesResponse.Content

14. .ReadFromJsonAsync<List<PrerequisiteDto>>();

15.

16. // Get completed courses from Student.API

17. var studentClient = \_clientFactory.CreateClient("StudentAPI");

18. var completedCoursesResponse = await studentClient

19. .GetAsync($"/api/student/{studentId}/completed-courses");

20.

21. if (!completedCoursesResponse.IsSuccessStatusCode)

22. throw new ServiceUnavailableException("Could not get completed courses");

23.

24. var completedCourses = await completedCoursesResponse.Content

25. .ReadFromJsonAsync<List<int>>();

26.

27. // Check if all mandatory prerequisites are met

28. return prerequisites

29. .Where(p => p.IsMandatory)

30. .All(p => completedCourses.Contains(p.PrerequisiteCourseId));

31. }

32. catch (Exception ex)

33. {

34. \_logger.LogError(ex, "Error checking prerequisites");

35. throw;

36. }

37. }

38.

Kod 17 Przykład wywołania innych serwisów za pomocą HTTP

1. **RabbitMQ**

W mikroserwisowej wersji systemu LMS, wysyłanie zdarzeń do RabbitMQ jest realizowane poprzez publikowanie oraz konsumpcję zdarzeń. Zdarzenia są zdefiniowane w postaci obiektu z potrzebnymi danymi (Kod 18)

1. public class EnrollmentCreatedEvent

2. {

3. public int Id { get; set; }

4. public int StudentId { get; set; }

5. public int CourseId { get; set; }

6. public DateTime EnrollmentDate { get; set; }

7. }

8.

Kod 18 Zdarzenie stworzenia zapisu

Po tworzenia zapisu oraz po uzyskaniu oraz zweryfikowaniu wstępnych informacji od innych serwisów, metoda CreateEnrollmentAsync (Kod 19) publikuje zdarzenie EnrollmentCreatedEvent za pomocą \_publishEndpoints.Publish, która wysyła to zdarzenie do kolejki RabbitMQ. Wzorzec ten nazywa się Architektura Sterowana Zdarzeniami (eng. Event Driven Architecture), umożliwia on innym mikroserwisom asynchroniczne reagowanie na to zdarzenie.

1. public async Task<EnrollmentDto> CreateEnrollmentAsync(CreateEnrollmentDto dto)

2. {

3. \_logger.LogInformation("Creating enrollment for student {StudentId} in course {CourseId}",

4. dto.StudentId, dto.CourseId);

5. // 1. Check prerequisites via HTTP (need immediate response)

6. var prerequisitesMet = await CheckPrerequisites(dto.StudentId, dto.CourseId);

7. if (!prerequisitesMet)

8. throw new PrerequisitesNotMetException("Not all prerequisites are met");

9.

10. // 2. Create enrollment

11. var enrollment = new Models.Enrollment

12. {

13. StudentId = dto.StudentId,

14. CourseId = dto.CourseId,

15. EnrollmentDate = DateTime.UtcNow,

16. Status = EnrollmentStatus.Pending

17. };

18.

19. \_context.Enrollments.Add(enrollment);

20. await \_context.SaveChangesAsync();

21.

22. \_logger.LogInformation("Creating enrollment for student {StudentId} in course {CourseId}",

23. enrollment.StudentId, enrollment.CourseId);

24. await \_publishEndpoint.Publish(new EnrollmentCreatedEvent

25. {

26. Id = enrollment.Id,

27. StudentId = enrollment.StudentId,

28. CourseId = enrollment.CourseId,

29. EnrollmentDate = enrollment.EnrollmentDate

30. });

31. \_logger.LogInformation("Published EnrollmentCreated event for enrollment {EnrollmentId}", enrollment.Id);

32.

33. return \_mapper.Map<EnrollmentDto>(enrollment);

34. }

Kod 19 Implementacja publikacji eventu zapisu przez MassTransit

Konsument EnrollmentCreatedConsumer nasłuchuje na kolejce RabbitMQ zdarzeń typu EnrollmentCreatedEvent i przetwarza je asynchronicznie. Po otrzymaniu zdarzenia, sprawdza w bazie danych, czy istnieje student o podanym StudentId, a następnie dodaje wpis o nowym zapisie studenta na kurs do tabeli StudentEnrollments. Proces kończy się zapisaniem zmian w bazie danych i zalogowaniem informacji o przetworzeniu zdarzenia. Proces ten został przedstawiony w Kod 20.

1. public class EnrollmentCreatedConsumer : IConsumer<EnrollmentCreatedEvent>

2. {

3. private readonly StudentContext \_context;

4. private readonly ILogger<EnrollmentCreatedConsumer> \_logger;

5.

6. public EnrollmentCreatedConsumer(StudentContext context, ILogger<EnrollmentCreatedConsumer> logger)

7. {

8. \_context = context;

9. \_logger = logger;

10. }

11.

12. public async Task Consume(ConsumeContext<EnrollmentCreatedEvent> context)

13. {

14. \_logger.LogInformation("Consuming EnrollmentCreated event: StudentId {StudentId}", context.Message.StudentId);

15.

16. var student = await \_context.Students.FindAsync(context.Message.StudentId);

17. if (student != null)

18. {

19. var enrollment = new StudentEnrollment

20. {

21. StudentId = context.Message.StudentId,

22. CourseId = context.Message.CourseId,

23. EnrollmentDate = context.Message.EnrollmentDate,

24. Status = EnrollmentStatus.Active

25. };

26.

27. \_context.StudentEnrollments.Add(enrollment);

28. await \_context.SaveChangesAsync();

29.

30. \_logger.LogInformation("Added enrollment for student {StudentId} in course {CourseId}",

31. student.Id, context.Message.CourseId);

32. }

33. }

34. }

35.

Kod 20 Implementacja konsumowania zdarzenia zapisu przez StudentService

Dzięki użyciu RabbitMQ i biblioteki MassTransit, konsument automatycznie reaguje na pojawienie się nowego zdarzenia w kolejce poprzez odpowiednią konfigurację. (Kod 21)

1. builder.Services.AddMassTransit(x =>

2. {

3. // Add consumers

4. x.AddConsumer<EnrollmentCreatedConsumer>();

5. x.AddConsumer<EnrollmentCancelledConsumer>();

6.

7. x.UsingRabbitMq((context, cfg) =>

8. {

9. cfg.Host(builder.Configuration["EventBusSettings:HostAddress"]);

10.

11. // Configure consumers

12. cfg.ConfigureEndpoints(context);

13. });

14. });

15.

Kod 21 Konfiguracja MassTransit

**API Gateway**

W architekturze monolitycznej API Gateway nie było potrzebne, ponieważ cała logika i dane były obsługiwane w ramach jednego procesu, a wszystkie żądania trafiały bezpośrednio do jednego punktu wejścia. W przypadku mikroserwisów Gateway pełni rolę pośrednika między klientem (UI) a mikroserwisami i jest zrealizowane za pomocą biblioteki Ocelot w .NET (Kod 22).

1. // Add Ocelot configuration file

2. builder.Configuration.SetBasePath(builder.Environment.ContentRootPath)

3. .AddJsonFile("ocelot.json", optional: false, reloadOnChange: true)

4. .AddEnvironmentVariables();

5.

6. // Add Ocelot

7. builder.Services.AddOcelot(builder.Configuration);

8.

Kod 22 Konfiguracjia Ocelot w .Net

W przypadku mikroserwisów mamy wiele niezależnych usług, które realizują różne funkcjonalności systemu. Bez API Gateway klient musiałby:

* Znać adresy poszczególnych mikroserwisów (CourseAPI, StudentAPI, EnrollmentAPI).
* Wysyłać żądania bezpośrednio do każdego serwisu.
* Obsługiwać różne formaty odpowiedzi

API Gateway rozwiązuje te problemy, pełniąc funkcję centralnego punktu wejścia do systemu mikroserwisowego. API Gateway odbiera żądania od klienta i przekazuje je do odpowiedniego mikroserwisu na podstawie zdefiniowanych reguł w pliku ocelot.json (Kod 23).

1. {

2. "Routes": [

3. {

4. "DownstreamPathTemplate": "/api/{everything}",

5. "DownstreamScheme": "http",

6. "DownstreamHostAndPorts": [

7. {

8. "Host": "course.api",

9. "Port": 80

10. }

11. ],

12. "UpstreamPathTemplate": "/course-api/{everything}",

13. "UpstreamHttpMethod": ["Get", "Post", "Put", "Delete"]

14. },

15. {

16. "DownstreamPathTemplate": "/api/{everything}",

17. "DownstreamScheme": "http",

18. "DownstreamHostAndPorts": [

19. {

20. "Host": "student.api",

21. "Port": 80

22. }

23. ],

24. "UpstreamPathTemplate": "/student-api/{everything}",

25. "UpstreamHttpMethod": ["Get", "Post", "Put", "Delete"]

26. },

27. {

28. "DownstreamPathTemplate": "/api/{everything}",

29. "DownstreamScheme": "http",

30. "DownstreamHostAndPorts": [

31. {

32. "Host": "enrollment.api",

33. "Port": 80

34. }

35. ],

36. "UpstreamPathTemplate": "/enrollment-api/{everything}",

37. "UpstreamHttpMethod": ["Get", "Post", "Put", "Delete"]

38. }

39. ],

40. "GlobalConfiguration": {

41. "BaseUrl": "http://gateway.api"

42. }

43. }

44.

Kod 23 Plik ocelot.json

### **Wnioski implementacji**

**Niezależność komponentów**

Każdy mikroserwis działa jako niezależna jednostka, która realizuje konkretne zadanie lub odpowiada za określoną funkcjonalność systemu. Dzięki temu możliwa jest **izolacja błędów** – awaria jednego mikroserwisu nie musi prowadzić do całkowitego zatrzymania systemu. **Skalowalność** staje się bardziej elastyczna – tylko te komponenty, które wymagają większej wydajności, mogą być skalowane niezależnie.

**Wygoda pisania i wdrażania**

Mikroserwisy można tworzyć przy użyciu **różnych technologii i języków programowania**, co pozwala zespołom programistycznym wybierać najbardziej odpowiednie narzędzia dla danego komponentu. Zespół odpowiedzialny za dany mikroserwis może rozwijać, testować i wdrażać swoje rozwiązanie niezależnie od innych zespołów, co **skraca czas dostarczania funkcjonalności**.

**Łatwiejsza konserwacja i rozwój**

Mikroserwisy można **aktualizować niezależnie**, bez potrzeby ingerencji w cały system. Dzięki modularnej budowie łatwiej jest **zarządzać kodem** oraz utrzymywać i rozwijać poszczególne komponenty. Każdy mikroserwis ma **jasno zdefiniowaną granicę odpowiedzialności**, co ułatwia zarówno debugowanie, jak i dodawanie nowych funkcji.

**Przejrzystość i podział odpowiedzialności**

W architekturze mikroserwisowej system jest podzielony na mniejsze, **łatwiejsze do zrozumienia komponenty**, co sprawia, że złożone systemy są bardziej przejrzyste. Wysoka spójność i niski stopień powiązań między komponentami wspierają **zasadę SRP (Single Responsibility Principle).**

**Zwiększona elastyczność biznesowa**

Umożliwia szybkie wdrażanie nowych funkcjonalności i łatwe testowanie **eksperymentalnych zmian** na małych fragmentach systemu. Mikroserwisy pozwalają na **szybszą adaptację** do zmieniających się wymagań biznesowych i rynkowych.

**Wyzwania**

Należy jednak pamiętać o potencjalnych wyzwaniach związanych z mikroserwisami. W architekturze mikroserwisowej system składa się z wielu małych, niezależnych komponentów. Każdy mikroserwis wymaga zarządzania cyklem życia, wdrażania, konfiguracji oraz aktualizacji. W miarę wzrostu liczby usług złożoność zarządzania znacząco wzrasta. Mikroserwisy komunikują się ze sobą za pomocą sieci (najczęściej REST API lub komunikatów przez brokery wiadomości, np. RabbitMQ). Zarządzanie komunikacją, szczególnie w przypadku błędów sieciowych, opóźnień czy przeciążeń, staje się bardziej wymagające. Choć mikroserwisy są niezależne, niektóre funkcje wymagają współpracy kilku usług. Koordynowanie tych zależności wymaga odpowiedniego projektowania i synchronizacji, aby uniknąć tzw. efektu domina w przypadku awarii jednej z usług. W architekturze mikroserwisów każdy komponent często posiada własną bazę danych. Choć to sprzyja niezależności, utrzymanie spójności danych staje się wyzwaniem, szczególnie w scenariuszach wymagających transakcji między różnymi usługami.

# **Analiza architektur**

## Metryki wydajnościowe

## **Koszt wdrożenia aplikacji w chmurze Azure**

Z pomocą narzędzia Azure Pricing Calculator [33] wyestymowane zostały koszty wdrożenia aplikacji w chmurze. Aby to zrobić należało przeprowadzić analizę potrzebnych zasobów oraz zrozumieć różnice pomiędzy architekturami.

Wspólne serwisy zostały wybrane jako kluczowe elementy infrastruktury, aby zapewnić podstawowe działanie aplikacji. Obejmują one:

* **Azure App Service** na poziomie Basic Tier (B1) z 1 rdzeniem, 1,75 GB RAM i 10 GB Storage przy pełnym obciążeniu 730 godzin miesięcznie.
* **Azure SQL Database** w modelu DTU[[2]](#footnote-2) Purchase Model na poziomie Basic Tier (B), z 5 DTUs oraz 2 GB wbudowanego storage'u dla pojedynczej bazy danych działającej przez 730 godzin miesięcznie.

W przypadku **architektury mikroserwisowej**, oprócz powyższych serwisów, dodano również **Service Bus** – usługę komunikacyjną umożliwiającą wymianę wiadomości między niezależnymi mikroserwisami. Usługa ta zapewnia niezawodną komunikację i usprawnia przepływ danych pomiędzy komponentami aplikacji, co jest kluczowe dla aplikacji opartej na mikroserwisach. Podobnie jak w przypadku innych usług, wybrano Basic Tier, aby zoptymalizować koszty.

**Monolit**

W przypadku aplikacji monolitycznej, aplikacja rozwijana i wdrażana jest jako jedność, co prowadzi do mniejszej liczby zasobów oraz prostszej konfiguracji infrastruktury.

Tab. 1 Tabela cen wdrożenia aplikacji monolitycznej na platformę Azure

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Service category** | **Service type** | **Region** | **Description** | **Monthly cost** |
| Compute | App Service | West US | Basic Tier; 1 B1 | €52,23 |
| Databases | Azure SQL Database | East US | Single Database, DTU Purchase Model | €4,67 |
|  |  | **Total** |  | **€56,90** |

**Mikroserwisy**

Architektura mikroserwisów zakłada podział aplikacji na niezależne, mniejsze usługi, które mogą działać autonomicznie. Wymaga to rozbudowania infrastruktury, ponieważ każdy Mikroserwis musi być hostowany oraz zarządzany osobno. Wpływa to na wzrost kosztów związany z wykorzystaniem wielu instancji maszyn wirtualnych, kontenerów oraz wszelkich innych usług.

Tab. 2 Tabela cen wdrożenia aplikacji mikroserwisowej na platforme Azure.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Service category** | **Service type** | **Region** | **Description** | **Monthly cost** |
| Compute | App Service | West US | Basic Tier; 1 B1 | €52,23 |
| Compute | App Service | West US | Basic Tier; 1 B1 | €52,23 |
| Compute | App Service | West US | Basic Tier; 1 B1 | €52,23 |
| Compute | App Service | West US | Basic Tier; 1 B1 | €52,23 |
| Databases | Azure SQL Database | East US | Single Database, DTU Purchase Model, Basic Tier, B: | €4,67 |
| Databases | Azure SQL Database | East US | Single Database, DTU Purchase Model, Basic Tier, B: | €4,67 |
| Databases | Azure SQL Database | East US | Single Database, DTU Purchase Model, Basic Tier, B | €4,67 |
| Integration | Service Bus | East US | Basic tier: 10 million messaging operations | €0.48 |
|  |  | **Total** |  | **€** **223.42** |

**Podsumowanie**

Wdrożenie aplikacji monolitycznej charakteryzuje się znacznie niższymi kosztami wdrożenia w wysokości €56,90 miesięcznie [Tab. 1]. Wynika to z dużo prostrzej infrastruktury oraz ograniczonej liczby wymaganych zasobów. Z kolei aplikacja mikroserwisowa generuje znacznie wyższe koszty €223,42 miesięcznie [Tab. 2]. Wzrost ten wynika z konieczności wykorzystania wielu instancji Azure App Service oraz Azure SQL Database, a także dodatkowych usług, takich jak Service Bus, które umożliwiają komunikację pomiędzy mikroserwisami. Koszt wdrożenia jednego mikroserwisu został oszacowany na €57,02 miesięcznie. Obejmuje on:

* Azure App Service: €52,23
* Azure SQL Database: €4,67
* Proporcjonalny koszt Service Bus: €0,12

W związku z tym, wdrożenie każdego kolejnego mikroserwisu będzie najprawdopodobniej zwiększać miesięczne koszty o około €57, co w dłuższym okresie czasu znacząco wpłynąć na budżet projektu.

# Wnioski

# **Bibliografia**

|  |  |
| --- | --- |
| [1] | „https://medium.com/@sadhnajuhirdeen/what-are-the-big-companies-still-uses-monolithic-architecture-2555a93345f1,” [Online]. [Data uzyskania dostępu: 1 05 2024]. |
| [2] | „https://www.techtarget.com/searchaws/definition/Amazon-Web-Services,” [Online]. [Data uzyskania dostępu: 05 2024]. |
| [3] | „https://beecommerce.pl/blog/co-to-jest-aplikacja-monolityczna,” 05 12 2024. [Online]. |
| [4] | C. Richardson, Microservices Pattern ith examples in Java. |
| [5] | V. Vernon, Implementing Domain-Driven Design. |
| [6] | „https://www.sap.com/poland/products/erp/what-is-erp.html,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [7] | „https://www.atlassian.com/pl/microservices/microservices-architecture,” [Online]. |
| [8] | B. Brendan, Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services. |
| [9] | „https://keen.io/blog/architecture-of-giants-data-stacks-at-facebook-netflix-airbnb-and-pinterest/,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [10] | A. Davis, „https://www.infoq.com/articles/monolith-versus-microservices/,” 05 2024. [Online]. |
| [11] | R. Flygare i A. Holmqvist, „Performance characteristics between monolithic and microservice-based systems,” [Online]. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1119785&dswid=-457. |
| [12] | „https://jmeter.apache.org/,” [Online]. [Data uzyskania dostępu: 05 2024]. |
| [13] | „https://docs.docker.com/get-started/overview/,” [Online]. [Data uzyskania dostępu: 05 2024]. |
| [14] | O. Al-Debagy i P. Martinek, „A Comparative Review of Microservices and Monolithic Architectures”. |
| [15] | „https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/,” [Online]. [Data uzyskania dostępu: 05 2024]. |
| [16] | A. O. A. P. G. Blinowski, „https://www.semanticscholar.org/paper/Monolithic-vs.-Microservice-Architecture%3A-A-and-Blinowski-Ojdowska/31a9d5d7286b24d5d2a99af005dca7a814640aec,” [Online]. [Data uzyskania dostępu: 05 2024]. |
| [17] | W. Z. Konrad Gos, „https://scholar.google.pl/citations?view\_op=view\_citation&hl=pl&user=h0ikHxkAAAAJ&citation\_for\_view=h0ikHxkAAAAJ:bnK-pcrLprsC,” [Online]. [Data uzyskania dostępu: 05 2024]. |
| [18] | „https://docs.gatling.io/,” [Online]. [Data uzyskania dostępu: 05 2024]. |
| [19] | R. C. Martin, „Czysty kod. Podręcznik dobrego programisty”. |
| [20] | R. C. Martin, „Czysta architektura”. |
| [21] | „https://learn.microsoft.com/pl-pl/dotnet/core/whats-new/dotnet-8/overview,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [22] | „https://www.typescriptlang.org/,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [23] | „https://masstransit.io/introduction,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [24] | „https://vuejs.org/,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [25] | „https://learn.microsoft.com/en-us/sql/sql-server/what-is-sql-server?view=sql-server-ver16,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [26] | „https://boringowl.io/blog/wprowadzenie-do-amqp-advanced-message-queuing-protocol,” [Online]. Available: 05. |
| [27] | „https://www.rabbitmq.com/,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [28] | „https://britenet.eu/pl/blog-post/czym-jest-docker-i-jakie-sa-korzysci-z-jego-uzywania/,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [29] | „https://docs.docker.com/compose/,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [30] | „https://learn.microsoft.com/pl-pl/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [31] | „https://ocelot.readthedocs.io/en/latest/introduction/bigpicture.html,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [32] | „Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |
| [33] | „https://microservices.io/patterns/monolithic.html,” [Online]. Available: https://microservices.io/patterns/monolithic.html. [Data uzyskania dostępu: 10 12 2024]. |
| [34] | „https://dzone.com/articles/the-evolution-of-distributed-systems,” [Online]. [Data uzyskania dostępu: 05 12 2024]. |

# Spis Fragmentu Kodów

[Kod 1 Application Context aplikacji monolitycznej 22](#_Toc185074945)

[Kod 2 Klasa CoursePrerequisite 22](#_Toc185074946)

[Kod 3 Klasa Course 23](#_Toc185074947)

[Kod 4 Klasa Student 23](#_Toc185074948)

[Kod 5 Klasa Enrollment 23](#_Toc185074949)

[Kod 6 Wybrane metody w CourseService 24](#_Toc185074950)

[Kod 7 Wybrane metody EnrollmentService 25](#_Toc185074951)

[Kod 8 Wybrane metody StudentService 25](#_Toc185074952)

[Kod 9 Implementacja kontrolera CourseController 27](#_Toc185074953)

[Kod 10 CourseDTO 27](#_Toc185074954)

[Kod 11 Rejestracja zależności DI 27](#_Toc185074955)

[Kod 12 Rejestracja zależnosci DI serwisów aplikacji 27](#_Toc185074956)

[Kod 13 Konfiguracja middleware pipeline 28](#_Toc185074957)

[Kod 14 Wstrzykniecie zależnosci bazodanowej 28](#_Toc185074958)

[Kod 15 Fragment konfiguracji do bazy danych 28](#_Toc185074959)

[Kod 16 Przykład zapytania łączonego wiele tabel 30](#_Toc185074960)

[Kod 17 Przykład wywołania innych serwisów za pomocą HTTP 34](#_Toc185074961)

[Kod 18 Zdarzenie stworzenia zapisu 34](#_Toc185074962)

[Kod 19 Implementacja publikacji eventu zapisu przez MassTransit 35](#_Toc185074963)

[Kod 20 Implementacja konsumowania zdarzenia zapisu przez StudentService 36](#_Toc185074964)

[Kod 21 Konfiguracja MassTransit 36](#_Toc185074965)

[Kod 22 Konfiguracjia Ocelot w .Net 36](#_Toc185074966)

[Kod 23 Plik ocelot.json 37](#_Toc185074967)

Wyrażam zgodę na udostępnienie mojej pracy w czytelniach Biblioteki SGGW   
w tym w Archiwum Prac Dyplomowych SGGW

.................................................................

*(czytelny podpis autora pracy)*

1. Dychotomia - podział jakiejś całości na dwie różniące się zasadniczo części; też: ta różnica [↑](#footnote-ref-1)
2. DTU – Jednostka transakcji bazy danych. Reprezentuje połączoną miarę procesora, pamięci, odczytów i zapisów. [36] [↑](#footnote-ref-2)