Architektura monolityczna i mikroserwisowa: analiza porównawcza i praktyczne zastosowania

Podejścia architektoniczne w projektowaniu współczesnych systemów informatycznych
Projektowanie Systemów Rozproszonych

mgr inż. Jakub Woźniak

Zakład Systemów Informatycznych
Instytut Informatyki Politechniki Poznańskiej

Plan wykładu

1. Wprowadzenie do architektur systemowych
2. Architektura monolityczna
3. Architektura mikroserwisowa
4. Porównanie architektur
5. Szczegółowa analiza mikroserwisów
6. Domain-Driven Design w mikroserwisach
7. Problemy z rozproszonymi transakcjami
8. Podsumowanie

Ewolucja architektur systemowych

Od systemów scentralizowanych do rozproszonych

Wyzwania współczesnych systemów:

- Skalowalność
- Odporność na awarie
- Szybki rozwój funkcjonalności
- Zróżnicowane obciążenie

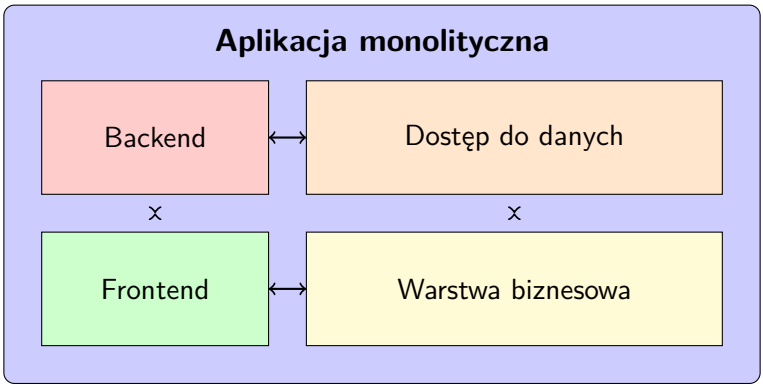
Dlaczego architektura ma znaczenie:

- Determinuje ograniczenia i możliwości systemu
- Wpływa na jakość, wydajność i utrzymywalność
- Decyduje o kosztach rozwoju w długim terminie

Architektura monolityczna - definicja

Definicja formalna

System zbudowany jako **pojedyncza, spójna jednostka**, gdzie wszystkie komponenty (frontend, backend, baza danych) są ściśle ze sobą zintegrowane i współdzielą ten sam kod oraz zasoby.



Charakterystyka architektury monolitycznej

Cechy strukturalne:

- Jeden kod bazowy
- Jedna baza danych
- Jeden proces/kontener
- Wspólna kompilacja i wdrożenie
- Silne zależności między modułami

Właściwości operacyjne:

- Proste wdrażanie (jeden plik)
- Scentralizowane zarządzanie
- Synchroniczne interakcje między komponentami
- Jednolity stack technologiczny
- Globalne skalowanie systemu

Przykłady monolitów z realnego świata

Amazon (przed 2001):

- Jednolity system obsługujący całą logikę e-commerce
- Trudności w skalowaniu wraz ze wzrostem ruchu
- Problemy z wprowadzaniem zmian przy rosnącej bazie kodu

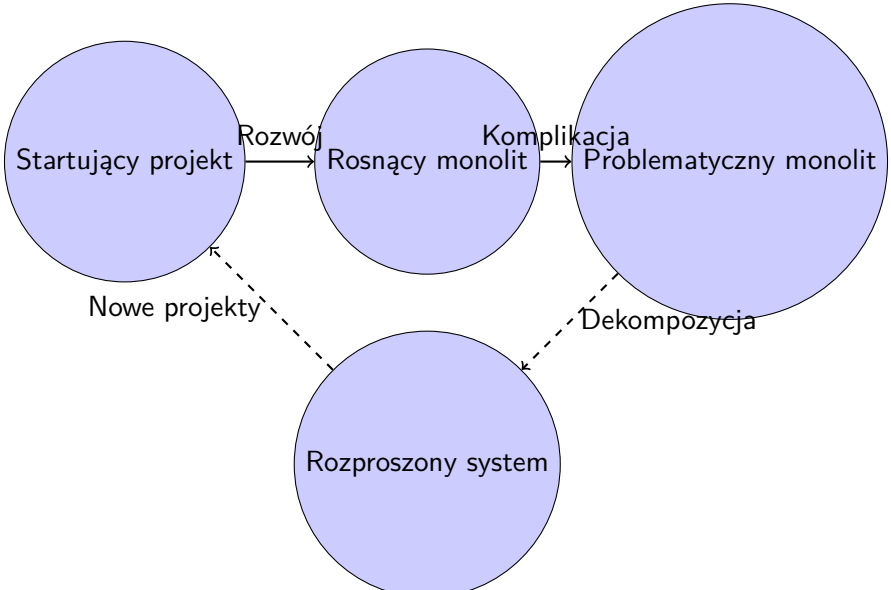
Netflix (wczesna wersja):

- Monolityczna aplikacja do zarządzania streamingiem
- Awarie bazodanowe przy wzroście ruchu
- Ograniczona elastyczność technologiczna

Uber (początkowo):

- Jeden kod odpowiedzialny za łączenie kierowców z pasażerami
- Ten sam system obsługiwał płatności, zarządzanie przejazdami
- Rosnące problemy z utrzymaniem i rozwojem

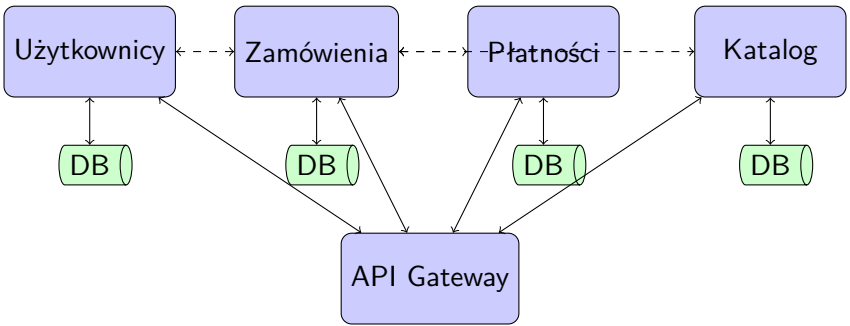
Cykl życia monolitu



Architektura mikroserwisowa - definicja

Definicja formalna

Zbiór niezależnych, samodzielnych usług komunikujących się przez API, z dedykowanymi bazami danych i możliwością niezależnego skalowania.



Charakterystyka architektury mikroserwisowej

Cechy strukturalne:

- Wiele niezależnych aplikacji
- Dedykowane bazy danych
- Luźne powiązania (loose coupling)
- Autonomia technologiczna
- Izolacja awarii (failure isolation)

Właściwości operacyjne:

- Niezależne wdrażanie usług
- Selektywne skalowanie
- Równoległy rozwój przez różne zespoły
- Komunikacja przez API (REST, gRPC, etc.)
- Rozproszony monitoring i logowanie

Przykłady mikroserwisów z realnego świata

Amazon:

- Ponad 1000 mikroserwisów po migracji w 2001 roku
- Dedykowane usługi dla koszyka, rekomendacji, płatności, wysyłki
- Każda strona produktu powstaje z kompozycji wielu usług

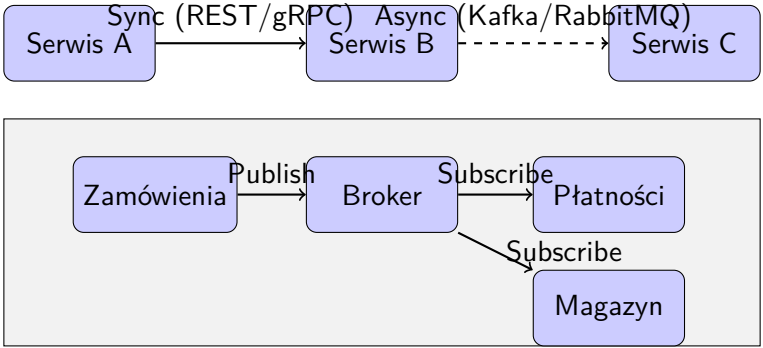
Netflix:

- Ponad 700 mikroserwisów w produkcji
- Usługi do zarządzania treścią, personalizacji, transkodowania wideo
- Architektura oparta na AWS z zaawansowaną instrumentacją

Uber:

- Ponad 500 niezależnych usług
- Usługi do zarządzania kierowcami, wyceny przejazdów, notyfikacji
- Wykorzystanie mikroserwisów do obsługi usług w różnych regionach

Wzorce komunikacji w mikroserwisach



Synchroniczna: bezpośrednia odpowiedź, blokująca, REST/gRPC

Asynchroniczna: kolejki wiadomości, event-driven, pub/sub

Hybryda: selektywne użycie obu wzorców zależnie od wymagań

Szczegółowe porównanie architektur

Aspekt	Monolit	Mikroserwisy
Rozwój i utrzymanie	Jeden kod źródłowy Prostsze debugowanie Trudna współpraca większych zespołów	Niezależne zespoły na usługach Większa złożoność całościowa Równoległy rozwój funkcjonalności
Skalowalność	Skalowanie całości systemu Nieefektywne przy nierównym obciążeniu Prostsze mechanizmy	Niezależne skalowanie usług Optymalizacja zasobów Złożone strategie skalowania

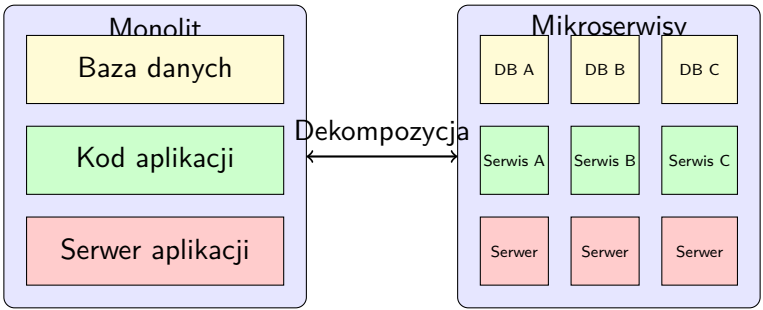
Porównanie architektur (kontynuacja)

Aspekt	Monolit	Mikroserwisy
Wydajność	Szybsza komunikacja wewnętrzna Brak narzutu sieciowego Mniejsze opóźnienia	Opóźnienia sieciowe Narzut serializacji/deserializacji Wyzwania z komunikacją
Elastyczność technologiczna	Ograniczona do jednego stosu Trudniejsza migracja całości Spójne biblioteki i narzędzia	Różne języki/frameworki per usługa Heterogeniczne bazy danych Łatwiejsza adopcja nowych technologii

Porównanie architektur (kontynuacja)

Aspekt	Monolit	Mikroserwisy
Koszty	Niższe koszty infrastruktury Mniejsze wymagania operacyjne Prostsze zarządzanie	Wyższe koszty infrastruktury Potrzeba zespołu DevOps Większe nakłady na monitoring
Odporność na awarie	Awaria jednej części wpływa na całość Prostsze mechanizmy odzyskiwania Single point of failure	Izolacja awarii pojedynczych usług Złożona obsługa błędów Graceful degradation

Wizualne porównanie architektury - wzorzec wdrożenia



Zalety architektury mikroserwisowej

a.) **Skalowalność**

Indywidualne skalowanie usług
Optymalny przydział zasobów
Możliwość skalowania w poziomie
Przykład: Amazon zwiększa moc dla usługi płatności w Black Friday

b.) **Izolacja błędów**

Awaria jednej usługi nie blokuje systemu
Circuit breaker pattern
Przykład: Netflix Hystrix dla obsługi awarii

c.) **Szybsze wdrażanie**

Częstsze aktualizacje
Równoległy rozwój przez wiele zespołów
Mniejsze ryzyko zmian
Przykład: Uber wdraża tysiące zmian dziennie

d.) **Elastyczność technologiczna**

Różne stosy technologiczne
Niezależny wybór języków i baz danych
Przykład: Python dla analityki, Java dla transakcji

Wady architektury mikroserwisowej

a.) **Złożoność zarządzania**

- Potrzeba narzędzi do orkiestracji (Kubernetes)
- Wymagania dla monitorowania (Prometheus)
- Centralizacja logowania (ELK)
- Większa złożoność diagnostyki

b.) **Koszty infrastruktury**

- Zwiększone zapotrzebowanie na zasoby
- Każda usługa wymaga osobnych zasobów
- Koszty rozproszonych baz danych

c.) **Komunikacja sieciowa**

- Opóźnienia w wywołaniach API
- Łańcuchowe zapytania (cascade calls)
- Obciążenie sieci wewnętrznej
- Narzut na serializację/deserializację

d.) **Spójność danych**

- Brak prostych transakcji
- Problemy z danymi rozproszonymi
- Złożoność obsługi akcji rozproszonych

Częste pułapki w architekturze mikroserwisowej

a.) **Niejasne granice usług**

Tworzenie zbyt drobnych usług (nano-services)
Silnie powiązane usługi (tight coupling)
Rozdzielanie funkcjonalności bez uzasadnienia biznesowego
Przykład: Rozdzielenie uwierzytelnienia i autoryzacji

b.) **Nadmierna inżynieria**

Przedwczesna mikroserwicyzacja
Implementacja dla prostych systemów bez potrzeby skalowania
Ignorowanie kosztów i złożoności
Przykład: Blog firmowy rozbity na mikroserwisy

Częste pułapki w architekturze mikroserwisowej

c.) **Brak automatyzacji**

- Niewystarczająca infrastruktura CI/CD
- Problemy z wersjonowaniem i deployem
- Brak zautomatyzowanych testów

d.) **Ignorowanie fundamentów DDD**

- Projektowanie oparte na technologii zamiast domeny
- Niespójne modele danych między usługami
- Arbitralne granice systemów

Fundamenty Domain-Driven Design

Wspólny język (Ubiquitous Language):

Jednolity język używany zarówno przez ekspertów domenowych, jak i zespół programistyczny.
Redukuje nieporozumienia i zwiększa spójność w komunikacji.

Konteksty ograniczone (Bounded Contexts):

Wyznaczanie granic mikroserwisów w oparciu o naturalne podziały domeny biznesowej.
Każdy kontekst ma własne modele i logikę, co minimalizuje zależności między serwisami.

Fundamenty Domain-Driven Design

Modelowanie strategiczne:

Identyfikacja subdomen: core domain (kluczowa domena), supporting domain (wspierająca), generic domain (ogólna).
Pozwala na priorytetyzację obszarów o największym wpływie na biznes.

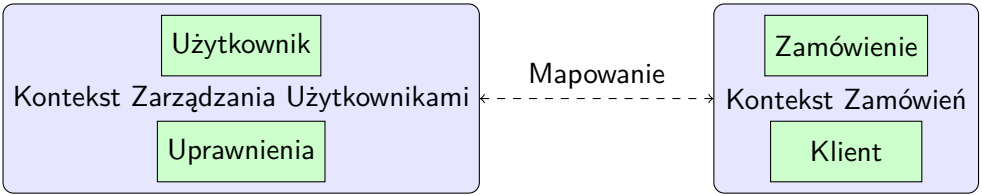
Modelowanie taktyczne:

Projektowanie szczegółowych elementów domeny, takich jak:

- Agregaty
- Encje
- Serwisy domenowe

Używane do implementacji logiki biznesowej w ramach kontekstu ograniczonego.

Kontekst ograniczony (Bounded Context)



Użytkownik = osoba z kontem

Klient = osoba składająca zamówienie

Kontekst ograniczony definiuje granice mikroserwisu w oparciu o domenę biznesową

Każdy kontekst posiada własny model i słownik pojęć

Ten sam termin może mieć różne znaczenie w różnych kontekstach

Implementacja: każdy mikroserwis = jeden bounded context

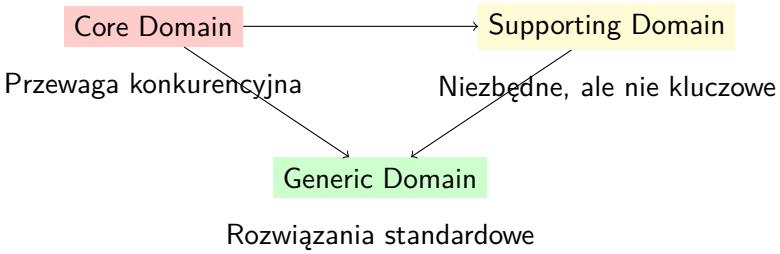
Mapowanie kontekstów w DDD

Wzorzec mapowania	Charakterystyka
Shared Kernel	Współdzielona część modelu między kontekstami, wymaga ścisłej koordynacji
Customer/Supplier	Relacja dostawca-odbiorca, zależność jednokierunkowa
Conformist	Jeden kontekst przyjmuje model drugiego bez możliwości wpływu
Anticorruption Layer	Warstwa translacji chroniąca własny model przed zewnętrznym
Open Host Service	Publiczny protokół dostępu do systemu
Published Language	Wspólny format wymiany danych (np. JSON Schema)

Mapowanie określa relację między kontekstami/mikroserwisami

Wybór wzorca zależy od relacji zespołów i priorytetów biznesowych

Strategic DDD

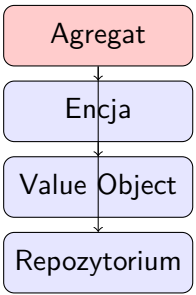


Core Domain: Najważniejsza część systemu, unikalna wartość biznesowa
Przykład: Algorytm matchowania Ubera, system rekomendacji Netflixa
Wymaga najlepszych zasobów, in-house development

Supporting Domain: Wspiera Core Domain, ale nie jest unikalna
Przykład: Profile użytkowników, zarządzanie kontem

Generic Domain: Standardowa funkcjonalność
Przykład: Uwierzytelnianie, powiadomienia email
Kandydaci do outsourcingu lub gotowych rozwiązań

Tactical DDD



Agregat: Logiczna grupa powiązanych obiektów traktowana jako całość

- Koszyk zakupowy z pozycjami
- Zamówienie z adresem dostawy

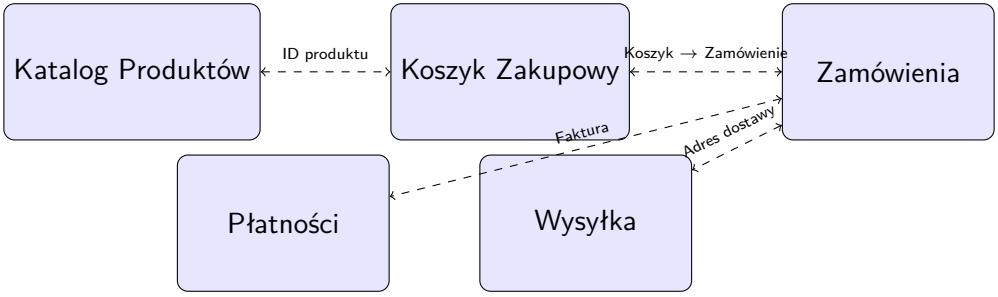
Encja: Obiekt posiada tożsamość i może zmieniać się w czasie

Value Object: Niemutowalny, bez identyfikatora

Repozytorium: Abstrakcja dostępu do danych

Serwis domenowy: Operacje przekraczające agregaty

DDD w praktyce - przykład e-commerce



Każdy kontekst = osobny mikroserwis z własnym modelem danych

Agregaty:

Katalog: Produkt (encja) z atrybutami (value objects)

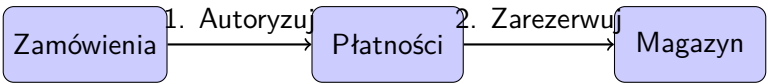
Koszyk: Koszyk (agregat) z pozycjami koszyka (encje)

Zamówienia: Zamówienie (agregat) z pozycjami i statusem

Serwisy domenowe:

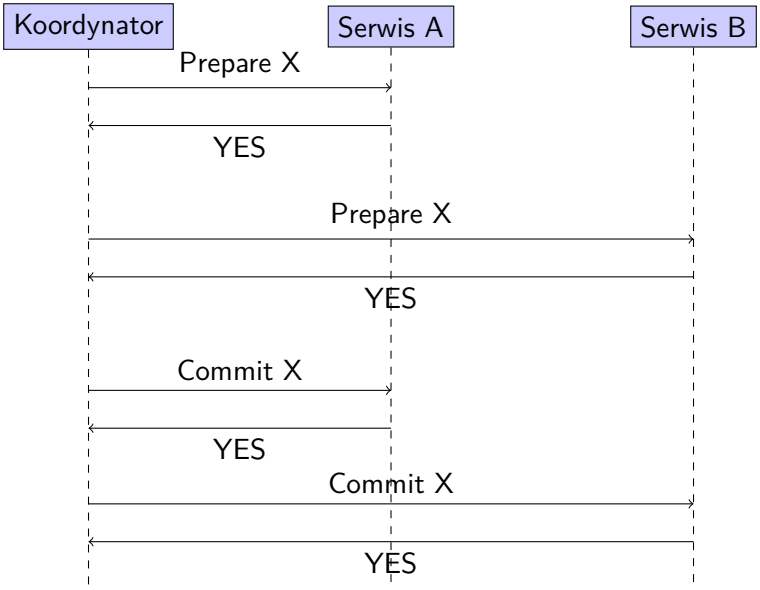
Konwersja koszyka na zamówienie, Obsługa procesu płatności

Wyzwanie atomowości w systemach rozproszonych



Problem: Brak gwarancji, że wszystkie usługi zatwierdzą swoje operacje.
Konsekwencja: Płatność została zautoryzowana, ale towar niedostępny - konieczność ręcznego wycofania.

Two-Phase Commit (2PC) - Diagram Sekwencji



Kompromisy spójności - Two-Phase Commit (2PC)

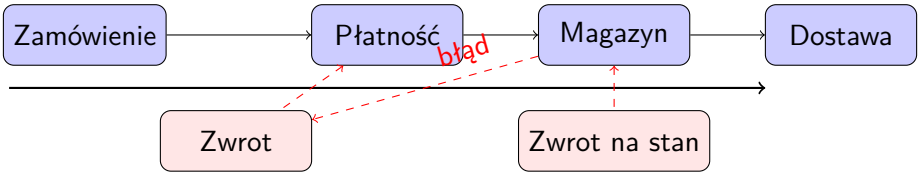
Zalety:

- Gwarancja atomowości
- Spójność danych

Wady:

- Blokowanie zasobów w fazie przygotowania
- Podatność na awarie koordynatora
- Problemy z wydajnością
- Trudne odzyskiwanie po awarii

Kompromisy spójności - Saga Pattern



Definicja: Sekwencja lokalnych transakcji z mechanizmem kompensacji

Zasada działania:

Każdy krok to osobna transakcja

Błędy obsługiwane przez transakcje kompensacyjne

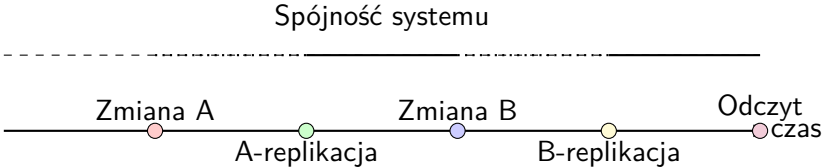
Przykład: anulowanie płatności, gdy magazyn zgłasza brak produktu

Implementacje:

Choreografia: sterowanie przez zdarzenia

Orchestracja: centralny koordynator

Eventual Consistency



Definicja: System osiąga spójność danych po pewnym czasie, dopuszczając przejściowe niespójności

Zastosowanie:

- Systemy gdzie szybkość ważniejsza niż absolutna spójność
- Mechanizmy synchronizacji oparte o zdarzenia
- Przykład: liczba produktów w magazynie, status dostawy

Kompromisy:

- Prostszy model programowania
- Lepsza wydajność i dostępność
- Konieczność obsługi niespójności przez aplikację

Strategie zarządzania rozproszonymi transakcjami

	2PC	**Saga**	**Eventual Consistency**
Spójność	Silna	Ostateczna	Ostateczna
Wydajność	Niska	Średnia	Wysoka
Złożoność	Wysoka	Wysoka	Średnia
Dostępność	Niska	Wysoka	Wysoka
Najlepsze zastosowania	Krytyczne operacje finansowe	Procesy biznesowe wieloetapowe	Dane niewymagające natychmiastowej spójności
Przykłady	Przelew z konta na konto	Zamawianie biletów lotniczych	Liczniki wyświetleń, statystyki

Wybór odpowiedniej architektury

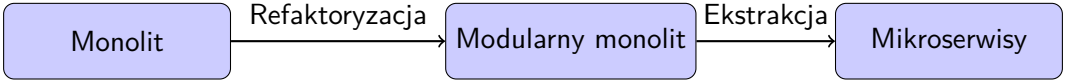
Kiedy stosować monolity?

- Małe i średnie projekty
- Ograniczone zasoby DevOps
- Proste domeny biznesowe
- Początkowa faza produktu
- Niskie wymagania skalowalności
- Brak potrzeby niezależnych wdrożeń

Kiedy stosować mikroserwisy?

- Duże, złożone systemy
- Wiele niezależnych zespołów
- Zróżnicowane wzorce obciążenia
- Wysoka skalowalność
- Elastyczność technologiczna
- Wdrożenia z wysoką częstotliwością

Ewolucja zamiast rewolucji



Modularny monolit: Etap pośredni
- Wewnętrzne bounded contexts
- Jasno określone granice modułów
- Zdefiniowane interfejsy między modułami
- Łatwiejsza migracja na mikroserwisy

Strategia ekstrakcji:
- Najpierw wydzielenie najbardziej niezależnych funkcjonalności
- Stopniowe zmniejszanie monolitu
- Kontrolowane wprowadzanie złożoności

Najważniejsze praktyki i zasady

a.) **Projektowanie z myślą o domenie biznesowej**

- DDD jako fundament podziału na usługi
- Granice usług zgodne z bounded contexts

b.) **Automatyzacja procesów**

- CI/CD jako konieczność dla mikroserwisów
- Infrastruktura jako kod (IaC)
- Testy automatyczne

c.) **Monitorowanie i obserwowanie**

- Centralne logowanie
- Distributed tracing
- Metryki wydajności

Najważniejsze praktyki i zasady

d.) **Przygotowanie na awarie**

- Circuit breakers
- Retries with backoff
- Fallbacks
- Chaos engineering

e.) **Przemyślana strategia danych**

- Świadome kompromisy spójności
- Odpowiednie wzorce dla transakcji rozproszonych
- Event sourcing jako alternatywa

Wnioski końcowe

Nie ma jednego, optymalnego podejścia dla wszystkich systemów

Mikroserwisy nie są celem, lecz środkiem do osiągnięcia konkretnych wymagań biznesowych

Złożoność rozproszonego systemu zawsze przewyższa złożoność monolitu

Stopniowa ewolucja często jest bezpieczniejsza niż rewolucyjna zmiana

Przemyślany projekt domeny (DDD) jest kluczem do sukcesu obu architektur

Zrozumienie kompromisów i kosztów jest niezbędne dla podjęcia właściwej decyzji

"Wybierz najprostszą architekturę, która spełnia twoje rzeczywiste wymagania - nie tę, która wygląda najlepiej na konferencji."

Bibliografia I

- Newman, S. (2015). *Building Microservices*. O'Reilly Media.
- Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley Professional.
- Richardson, C. (2019). *Microservices Patterns*. Manning Publications.
- Fowler, M. *Microservices*. https://martinfowler.com/articles/microservices.html
- Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional.
- Kleppmann, M. (2017). *Designing Data-Intensive Applications*. O'Reilly Media.
- Richardson, C. *Microservices.io*. https://microservices.io/
- Killalea, T. (2016). *The Hidden Dividends of Microservices*. Communications of the ACM, 59(8), 42-45.
- Mauro, T. (2015). *Adopting Microservices at Netflix: Lessons for Architectural Design*. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

Bibliografia II

 Reinhold, E. (2016). *The Microservices Dichotomy: Uber*, QCon San Francisco.

Dziękuję za uwagę

Pytania?