Style architektoniczne

Monolit · Mikroserwisy · Modularny monolit

mgr inż. Jakub Woźniak

Politechnika Poznańska · Instytut Informatyki · Semestr letni 2025/26

Plan wykładu

Agenda

- Monolit — kiedy wystarczy?
- Modularny monolit — kompromis
- Mikroserwisy — korzyści i koszty
- Domain-Driven Design i Bounded Contexts
- Distributed Monolith — anti-pattern
- Case study: Segment.io
- ☁ Cloud-native vs cloud-hosted

Monolit

Klasyczny monolit

Definicja

Jedna aplikacja, jeden proces, jedna baza danych, jeden deployment.

Zalety:

- Prosty development i deploy
- Proste debugowanie
- Brak opóźnień sieciowych
- Transakcje ACID „za darmo“

Wady:

- Cały system = jeden punkt awarii
- Skaluje się *cały*, nie części
- Duży zespół = konflikty scalania
- Długie buildy, wolne testy

DHH (Basecamp / 37signals)

„Mikroserwisy to architektura dla dużych organizacji z dużymi problemami. Większość firm nie ma tych problemów.“

Basecamp / HEY.com — miliony użytkowników, monolit w Rails.

- Mały zespół (20 osób) — niska koordynacja
- Dobrze zdefiniowane warstwy w kodzie
- Agresywne cachowanie
- Pionowe skalowanie (potężne serwery)

Modularny monolit

Idea

Modularny monolit

Monolit podzielony na **moduły z jasnymi granicami**. Komunikacja przez publiczne interfejsy, nie przez współdzieloną bazę.

- Granice modułów = przyszłe granice serwisów
- Jeden deploy, ale niezależne zespoły
- Brak kosztu sieci między modułami

Case study: Shopify

Shopify Engineering — „Deconstructing the Monolith“

Monolit w Rails, 3 mln LoC. Zamiast migracji do mikroserwisów:

1. Zdefiniowali **bounded contexts** w istniejącym kodzie
2. Wymusili granice narzędziem *Packwerk*
3. Komunikacja między modułami przez **zdarzenia**
4. Rezultat: szybkość dev jak w mikroserwisach, prostota ops monolitu

Uwaga

Modularny monolit wymaga **dyscypliny** — bez niej granice się rozmywają.

Mikroserwisy

Definicja

Mikroserwisy

System składający się z **małych, niezależnie wdrażanych serwisów**, każdy realizuje jedną zdolność biznesową.

Kluczowe cechy (Sam Newman):

- **Niezależne wdrożenie** — zmiana jednego nie wymaga wdrożenia innych
- **Modelowanie wokół domeny** — serwis = bounded context
- **Własność danych** — każdy serwis ma swoją bazę

Korzyści mikroserwisów

1. **Niezależne skalowanie** — skaluj tylko wąskie gardło
2. **Izolacja awarii** — awaria jednego nie zabija całości
3. **Technologiczna różnorodność** — Python do ML, Go do API
4. **Szybsze cykle wydawnicze** — mniejszy zakres zmian
5. **Autonomia zespołu** — team per service (prawo Conwaya)

Koszty mikroserwisów

Ukryte koszty

- **Opóźnienia sieciowe** — 1ms+ zamiast nanosekund
- **Rozproszona diagnostyka** — request przechodzi 10 serwisów
- **Spójność danych** — koniec z ACID między serwisami
- **Narzut operacyjny** — 100 serwisów × monitoring, deploys, dyżury

Kiedy mikroserwisy mają sens?

TAK, gdy...

- Duży zespół (50+ devów)
- Różne komponenty skalują się inaczej
- Dojrzały DevOps
- Dobrze zrozumiana domena

NIE, gdy...

- Mały zespół (< 10 devów)
- Niezrozumiana domena
- Brak dojrzałości operacyjnej
- „Bo tak robią w Netflixie“

Domain-Driven Design

Bounded Contexts

Bounded Context (Eric Evans, 2003)

Granica, w której model domeny jest spójny. Ten sam termin (np. „klient“) może znaczyć co innego w kontekście sprzedaży i supportu.

- Bounded Context ⟶ naturalny kandydat na granicę serwisu
- **Context Map** — diagram relacji między kontekstami
- **Ubiquitous Language** obowiązuje *w obrębie* kontekstu

Context Map — relacje

Relacja	Opis
Partnership	Dwa zespoły razem ewoluują oba konteksty
Customer-Supplier	Odbiorca zależy od dostawcy, negocjują kontrakt
Anti-Corruption Layer	Odbiorca tłumaczy model dostawcy na swój
Shared Kernel	Współdzielony fragment modelu (ryzyko!)
Open Host Service	Dostawca publikuje API dla wszystkich

Anti-patterns

Distributed Monolith

Najgorsze z dwóch światów

Mikroserwisy, które muszą być wdrażane **razem**, dzielą **jedną bazę** i synchronicznie się wywołują. Wszystkie koszty, żadne korzyści.

Sygnały ostrzegawcze:

- Deploy jednego wymaga deployu trzech innych
- Zmiana schematu DB = koordynacja wielu zespołów
- Jedna zmiana biznesowa = modyfikacja 5+ serwisów

Jak unikać?

1. **Jasne granice danych** — każdy serwis = własna baza
2. **Asynchroniczna komunikacja** gdzie to możliwe
3. **Unikanie współdzielonych bibliotek z logiką biznesową**
4. **Test niezależności**: czy mogę wdrożyć ten serwis *sam*?

Case study: Segment

Segment.io — podróż tam i z powrotem

Goodbye Microservices (2018)

Faza 1: Monolith — prosty pipeline, wszystko działa.

Faza 2: 100 mikroserwisów (1 per integracja) — powtórzony kod, N × te same bugi, koszmar operacyjny.

Faza 3: Powrót do zunifikowanego pipeline'u.

Lekcje z Segment

1. **Granice wzdłuż domen**, nie wzdłuż integracji
2. Wspólna logika = biblioteka, nie 100 kopii
3. Mikroserwisy nie rozwiązują **złej dekompozycji**
4. **Poznaj domenę**, zanim pokroisz system

Porównanie

Monolit vs Modularny monolit vs Mikroserwisy

Aspekt	Monolit	Modularny	Mikroserwisy
Złożoność deployu	Niska	Niska	Wysoka
Niezależne skalowanie	Nie	Nie	Tak
Izolacja awarii	Brak	Częściowa	Wysoka
Opóźnienie wewnętrzne	ns	ns	ms
Spójność danych	ACID	ACID	Eventual
Narzut operacyjny	Niski	Niski	Wysoki
Cloud deploy	PaaS / VM	PaaS / kontener	K8s / ECS

Cloud-native vs Cloud-hosted

Dwa podejścia do chmury

Cloud-hosted

Przenieś aplikację „jak jest“ na VM / kontener w chmurze.

- Lift & shift
- Minimum zmian w kodzie
- Nie korzystasz z platformy

Cloud-native

Projektuj z myślą o chmurze od dnia pierwszego.

- Stateless, kontenery, CI/CD
- Managed services zamiast self-hosted
- Autoscaling, resilience by design

Heroku / Adam Wiggins, 2011

Metodologia budowania aplikacji SaaS — naturalnie pasuje do chmury i kontenerów.

Czynnik	Zasada
Codebase	Jedno repo, wiele deployów
Config	Konfiguracja w zmiennych środowiskowych
Backing services	Bazy, kolejki jako podpinane zasoby
Processes	Bezstanowe procesy (stateless)
Port binding	Aplikacja eksportuje HTTP jako usługę
Disposability	Szybki start i graceful shutdown

12factor.net — Adam Wiggins (Heroku)

Architektury w chmurze — PaaS i kontenery

Styl	Deployment w chmurze	Przykłady serwisów
Monolit	1 kontener / VM, PaaS	App Service, Cloud Run, Elastic Beanstalk
Modularny monolit	1 kontener, managed DB	ECS / Cloud Run + RDS / Cloud SQL
Mikroserwisy	N kontenerów, orkiestracja	EKS / GKE / AKS, ECS, Cloud Run per svc

Ważne

Chmura nie wymusza mikroserwisów — monolit na Cloud Run to **valid architecture**.

Startup z 5-osobowym zespołem chce zacząć od mikroserwisów, „żeby się *później łatwiej skalować*“.

Jakie argumenty im przedstawisz?

Kiedy byłby dobry moment na pierwszą dekompozycję?

Podsumowanie

Kluczowe wnioski

1. **Monolit to nie wstyd** — dla wielu firm optymalna architektura
2. **Modularny monolit** — korzyści organizacyjne, brak kosztów sieciowych
3. **Mikroserwisy** wymagają dojrzałości i zrozumienia domeny
4. **Bounded Contexts** — najlepsze narzędzie do definiowania granic
5. **Poznaj domenę, zanim pokroisz system**
6. **Cloud-native ≠ mikroserwisy** — 12-Factor App stosuj niezależnie od stylu

Źródła i lektury

- S. Newman — *Building Microservices*, 2nd ed., rozdz. 1–3
- E. Evans — *Domain-Driven Design*, rozdz. 14–15
- Segment — *Goodbye Microservices* (blog)
- Shopify — *Deconstructing the Monolith* (blog)
- M. Fowler — *Monolith First*
- 12factor.net — Adam Wiggins (Heroku)

Następny wykład

Wykład 3: Komunikacja w systemach rozproszonych

REST · gRPC · Message Brokers · API Gateway