Wzorce architektoniczne

EDA · CQRS · Event Sourcing · Saga

mgr inż. Jakub Woźniak

Politechnika Poznańska · Instytut Informatyki · Semestr letni 2025/26

Plan wykładu

Agenda

- Event-Driven Architecture — trzy oblicza
- CQRS — oddzielne modele odczytu i zapisu
- Event Sourcing — log zdarzeń jako źródło prawdy
- Orkiestracja vs Choreografia
- Wzorzec Sagi — transakcje rozproszone
- ☁ Managed event/workflow services

Event-Driven Architecture

Czym jest EDA?

Event-Driven Architecture

Architektura, w której komponenty komunikują się przez **zdarzenia**. Producent nie wie, kto konsumuje — i odwrotnie.

Kluczowe cechy:

- **Decoupling** — producent i konsument niezależni
- **Asynchroniczność** — brak blokującego oczekiwania
- **Skalowalność** — łatwe dodawanie konsumentów
- Kafka, RabbitMQ, EventBridge jako **event backbone**

Trzy oblicza zdarzeń (M. Fowler)

Wzorzec	Opis	Przykład
Event Notification	„Coś się stało“ — minimalna informacja	`OrderPlaced { orderId }`
Event-Carried State Transfer	Zdarzenie niesie pełne dane	`OrderPlaced { id, items, total }`
Event Sourcing	Log zdarzeń = źródło prawdy	Pełna historia zmian konta

M. Fowler — „What do you mean by ‘Event-Driven’?“ · martinfowler.com/articles/201701-event-driven.html

Kiedy EDA?

TAK, gdy...

- Wielu konsumentów tego samego zdarzenia
- Potrzebujesz loose coupling
- Operacje mogą być asynchroniczne
- System event-heavy (logi, metryki, notyfikacje)

NIE, gdy...

- Potrzebujesz natychmiastowej odpowiedzi
- Prosty CRUD bez fan-outu
- Mały system, 1-2 serwisy
- Trudno debugować eventual consistency

CQRS

Command Query Responsibility Segregation

CQRS

Oddzielne modele dla **zapisów** (Command) i **odczytów** (Query). Zamiast jednego modelu ORM — dwa, zoptymalizowane pod różne potrzeby.

- **Command model** — walidacja, reguły biznesowe, zapis
- **Query model** — zdenormalizowane widoki, szybkie odczyty
- Modele mogą mieć **różne bazy danych**

M. Fowler — „CQRS“ · martinfowler.com/bliki/CQRS.html

Kiedy stosować CQRS?

TAK, gdy...

- Odczyty i zapisy mają **różne wymagania**
- Read-heavy system (100:1 read/write)
- Potrzebujesz wielu widoków tych samych danych
- Łączysz z Event Sourcing

NIE, gdy...

- Prosty CRUD — overhead się nie opłaci
- Mały zespół bez doświadczenia
- Dane są proste i jednomodelowe
- Silna potrzeba strong consistency

Jak się łączą?

Write model emituje **zdarzenia** → zdarzenia zapisywane w **event store** → **projekcje** budują read model → eventual consistency między modelami.

Korzyści połączenia:

- Write model: append-only event log — szybki zapis
- Read model: zdenormalizowane widoki — szybki odczyt
- Można zbudować **wiele projekcji** z tego samego strumienia

Uwaga

Eventual consistency = użytkownik może przez chwilę widzieć stare dane. Projektuj UI z tym na uwadze.

Event Sourcing

Log zdarzeń jako źródło prawdy

Event Sourcing

Zamiast przechowywać **aktualny stan**, przechowujesz **sekwencję zdarzeń**, które do tego stanu doprowadziły. Stan = replay wszystkich eventów.

Przykład: konto bankowe

```
AccountOpened { balance: 0 }
MoneyDeposited { amount: 1000 }
MoneyWithdrawn { amount: 200 }
```

Stan aktualny: balance = 800

Snapshoty i odtwarzanie

Problem: replay milionów eventów = wolny start.

Snapshot

Co N eventów zapisz pełny stan. Odtwarzanie: załaduj snapshot + replay eventów od snapshotu.

Podejście	Czas odtwarzania
Full replay (10K eventów)	200ms
Full replay (1M eventów)	20s
Snapshot + 100 eventów	5ms

Problemy Event Sourcing

Realne wyzwania

- **GDPR** — prawo do zapomnienia vs append-only log (crypto-shredding)
- **Schema evolution** — format eventów zmienia się w czasie
- **Rozmiar logu** — archiwizacja, retencja, snapshoty
- **Złożoność** — replay, projekcje, debugging
- **Eventual consistency** — UI musi to obsługiwać

Greg Young — „CQRS and Event Sourcing" (DDD Europe 2016)

Orkiestracja vs Choreografia

Dwa podejścia do koordynacji

Orkiestracja

Centralny **koordynator** steruje przepływem.

- Łatwe śledzenie i debugowanie
- Single point of failure
- Np. Temporal, Step Functions

Choreografia

Serwisy **reagują na zdarzenia** autonomicznie.

- Loose coupling
- Trudniejsze debugowanie
- Np. Kafka + eventy

Porównanie

Aspekt	Orkiestracja	Choreografia
Kontrola	Centralna	Rozproszona
Coupling	Wyższy	Niższy
Debugowanie	Łatwiejsze	Trudniejsze
Elastyczność	Mniejsza	Większa
Narzędzia	Temporal, Step Functions	Kafka, EventBridge
Złożony przepływ	Preferowane	Ryzykowne

Wzorzec Sagi

Saga — transakcje rozproszone bez locków

Saga (Garcia-Molina & Salem, 1987)

Sekwencja **lokalnych transakcji**. Każdy krok ma **kompensację** — operację cofającą jego efekt. Jeśli krok N padnie ⟶ wykonaj kompensacje N-1 … 1.

Dlaczego nie 2PC (Two-Phase Commit)?

- 2PC = **blokujący** — wszystkie zasoby czekają na koordynatora
- 2PC = **single point of failure** koordynatora
- 2PC nie skaluje się w mikroserwisach

H. Garcia-Molina, K. Salem — „Sagas" (ACM SIGMOD, 1987)

Choreograficzna vs orkiestracyjna saga

Saga choreograficzna

Serwisy emitują eventy — kolejny krok reaguje na event poprzedniego.

- **1.** Brak centralnego punktu awarii
- – Trudne śledzenie przepływu
- – Cykliczne zależności

Saga orkiestracyjna

Saga Orchestrator koordynuje kroki i kompensacje.

- **1.** Jasny przepływ, łatwy monitoring
- **1.** Centralne zarządzanie kompensacjami
- – Orchestrator = dodatkowy serwis

C. Richardson — Microservices Patterns, *rozdz. 4*

Przepływ

Zamówienie → Rezerwacja stocku → Płatność → Wysyłka

Co jeśli płatność się nie powiedzie?

Kompensacja: **zwolnij zarezerwowany stock**.

Zamówienie zmienia status na „anulowane".

Cecha	2PC	Saga
Blokowanie zasobów	Tak	Nie
Spójność	Strong	Eventual
Skalowanie	Słabe	Dobre
Kompensacje	Nie potrzebne	Wymagane
Złożoność kodu	Niska	Wyższa

Managed event/workflow services

Managed event routing

Dostawca	Usługa	Model
AWS	EventBridge	Event bus + reguły routingu
Google Cloud	Eventarc	Event routing do Cloud Run / Functions
Azure	Event Grid	Pub/Sub zdarzeniowy, reaktywny
CNCF	CloudEvents	Standard formatu zdarzeń (portability)

CloudEvents

Otwarty standard opisu zdarzeń (CNCF). Ujednolica format między dostawcami: `source`, `type`, `data`. Wspierany przez EventBridge, Event Grid, Eventarc.

Managed workflow orchestration

Dostawca	Usługa	Cechy
AWS	Step Functions	JSON state machine, Lambda integration, saga pattern
Google	Workflows	YAML DSL, connector library, HTTP-based
Azure	Durable Functions	Code-first (C#/JS/Python), fan-out/fan-in
Temporal	Temporal Cloud	Code-first, replay, managed Temporal

Zasada

Nie buduj własnego orchestratora — chyba że masz **bardzo** dobry powód.

Event Sourcing brzmi świetnie w teorii — pełna historia zmian, audit trail za darmo, możliwość odtworzenia stanu.

Ale jakie są *realne* problemy?

Kiedy Event Sourcing jest overkillem?

Podsumowanie

Kluczowe wnioski

1. **EDA = decoupling** — ale 3 różne wzorce, nie mylić
2. **CQRS** = oddzielne modele — nie zawsze potrzebny
3. **Event Sourcing** = audit trail + replay, ale GDPR i złożoność
4. **Saga > 2PC** w mikroserwisach — kompensacje zamiast locków
5. **Orkiestracja vs choreografia** — dobierz do złożoności przepływu
6. **Managed workflow** — Step Functions / Temporal zamiast DIY

Źródła i lektury

- C. Richardson — *Microservices Patterns*, rozdz. 4, 7
- S. Newman — *Building Microservices*, rozdz. 6
- M. Fowler — *Event Sourcing*, *CQRS*, *What do you mean by Event-Driven?*
- Greg Young — *CQRS and Event Sourcing* (DDD Europe 2016)
- Netflix — *Conductor: a microservices orchestrator* (blog)
- Temporal.io — docs.temporal.io
- H. Garcia-Molina, K. Salem — *Sagas* (ACM SIGMOD, 1987)

Następny wykład

Wykład 5: Dane w systemach rozproszonych

Database per Service · Replikacja · Partycjonowanie · CAP/PACELC