Paweł Rajba
pawel@cs.uni.wroc.pl
http://www.itcourses.eu/

Microservices

Agenda

- Wprowadzenie, SOA vs. MSA
- Architektura
- Wady i zalety różnych podejść, kiedy używać
- Design Principles, przegląd wzorców
- Podział na Microserwisy
- Transakcje i spójność danych
- Realizacja zapytań, external API
- Technologie
- Kto tego używa?

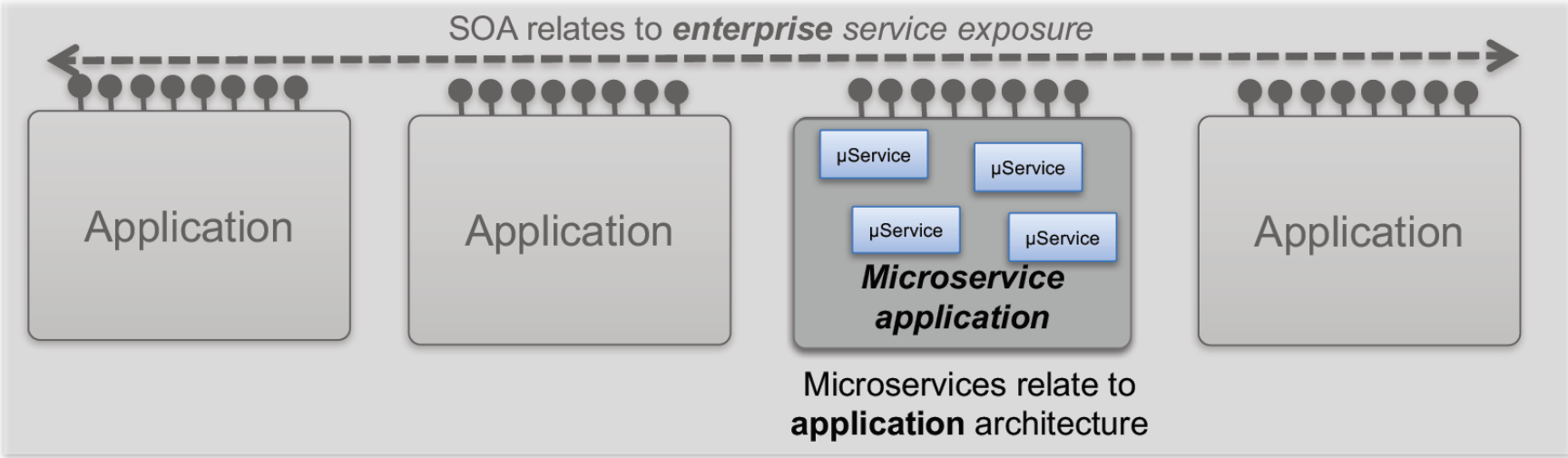
Wprowadzenie

- Service Oriented Architecture
  - Funkcjonalności udostępnione jako usługi
  - Określony kontrakt wymiany danych
  - Daje możliwość skorzystania z funkcjonalności wielu różnym klientom (web, mobile, desktop)
  - Dobrze napisane umożliwia łatwe skalowanie
  - Cykl życia usługi nie wymusza zmian u klientów (dopóki zachowany jest kontrakt)

Wprowadzenie

- Microservices Architecture (MSA)
  - Koncept SOA, ale w bardziej granularnej postaci
  - „Małe" usługi z pojedynczymi odpowiedzialnościami
  - Aplikacja zwykle bazuje na wielu usługach
  - Dostęp do usług nie powiązany z żadną technologią
  - Lekka komunikacja pomiędzy komponentami
  - Każda usługa ma własne repozytorium danych
  - Każda usługa ma własny cykl życia i publikacji
  - Transakcje najczęściej są rozproszone
    - … lub oparte o „eventual consistency"

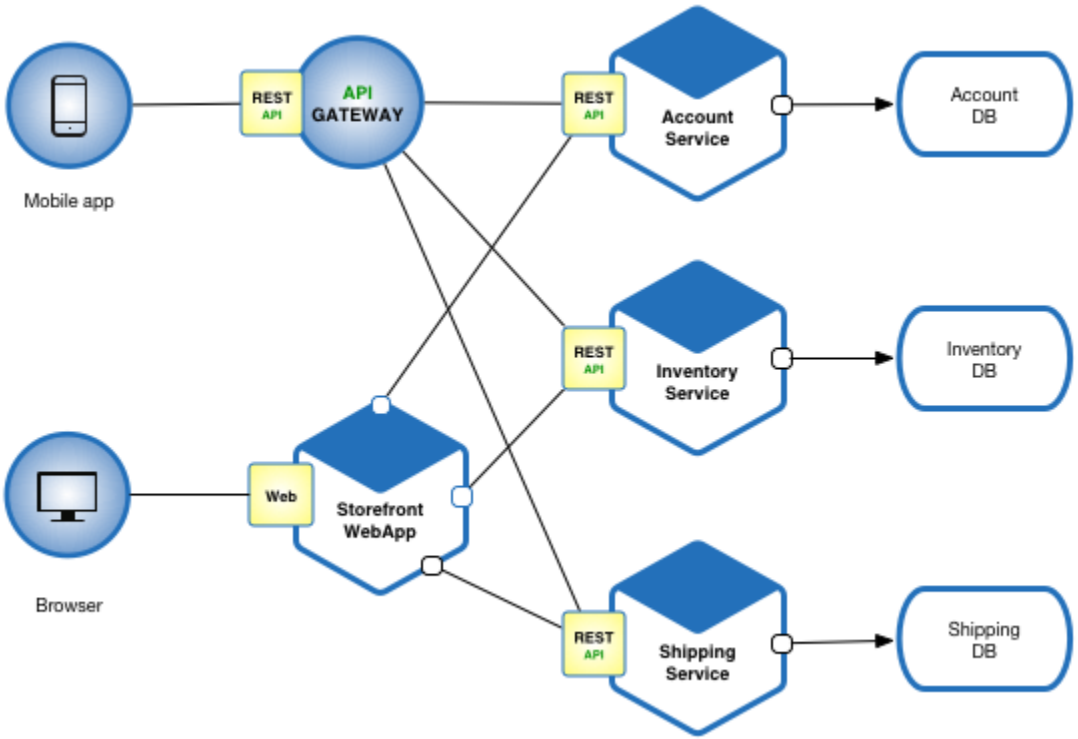
SOA vs. MSA



https://www.slideshare.net/kimjclark/3298-microservices-and-how-they-relate-to-esb-api-and-messaging-inter-connect-2017

Architektura

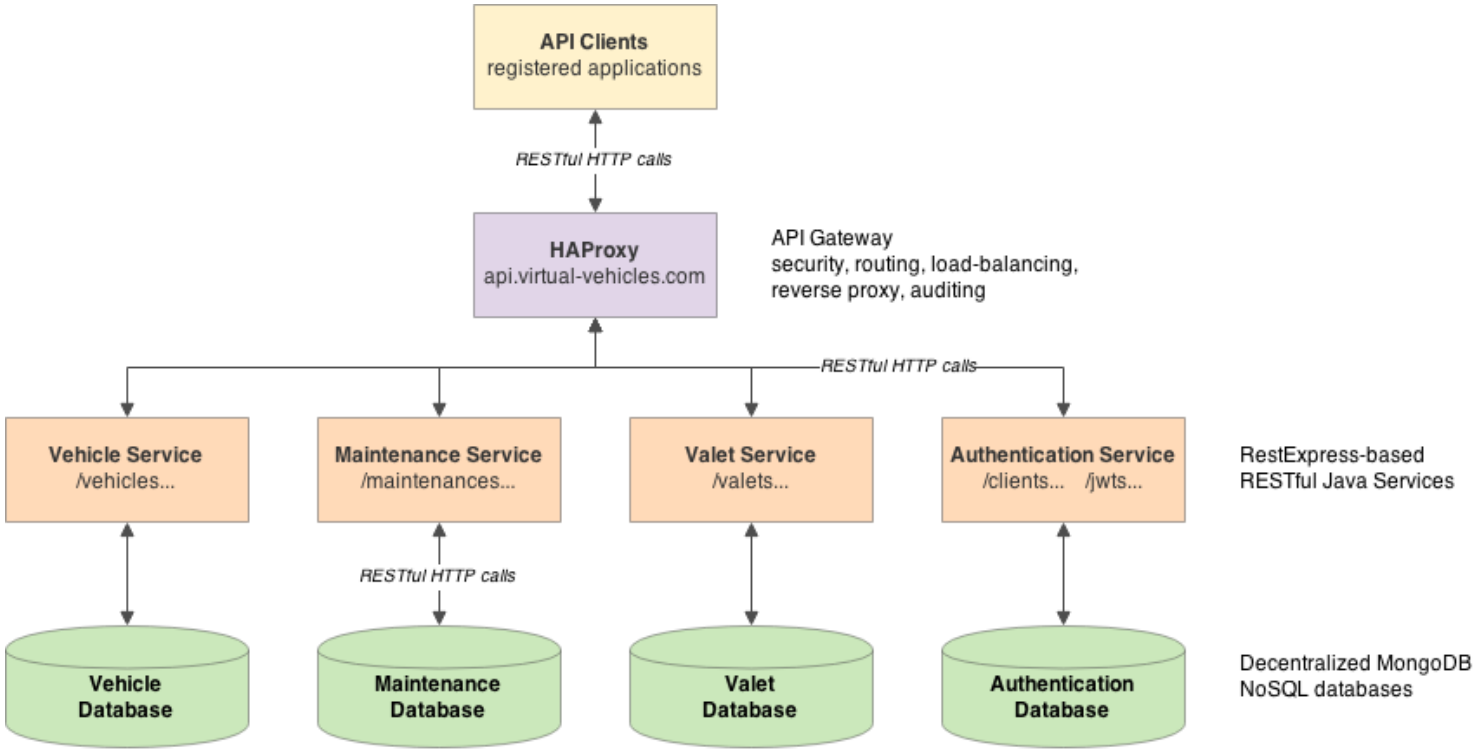
- MSA, przykład 1



http://microservices.io/patterns/microservices.html

Architektura

- MSA, przykład 2



https://programmaticponderings.files.wordpress.com/2015/05/virtual-vehicles-architecture-4.png

Wady monolitycznej architektury

- Czas publikacji dużego systemu może być długi
- Publikacja obarczona większym ryzykiem błędów
- Małe, szybkie zmiany muszą czekać na „release”
- Sztywny zbiór technologii (technology stack)
  - Brak możliwości łatwej adaptacji nowych technologii
- Często silne powiązania pomiędzy komponentami
  - Jeśli kod jest źle napisany, a najczęściej tak jest
- Przeciążone IDE z powodu dużej ilości kodu
- Skalować musimy cały system, a nie pojedyncze składowe, które potrzebują skalowania

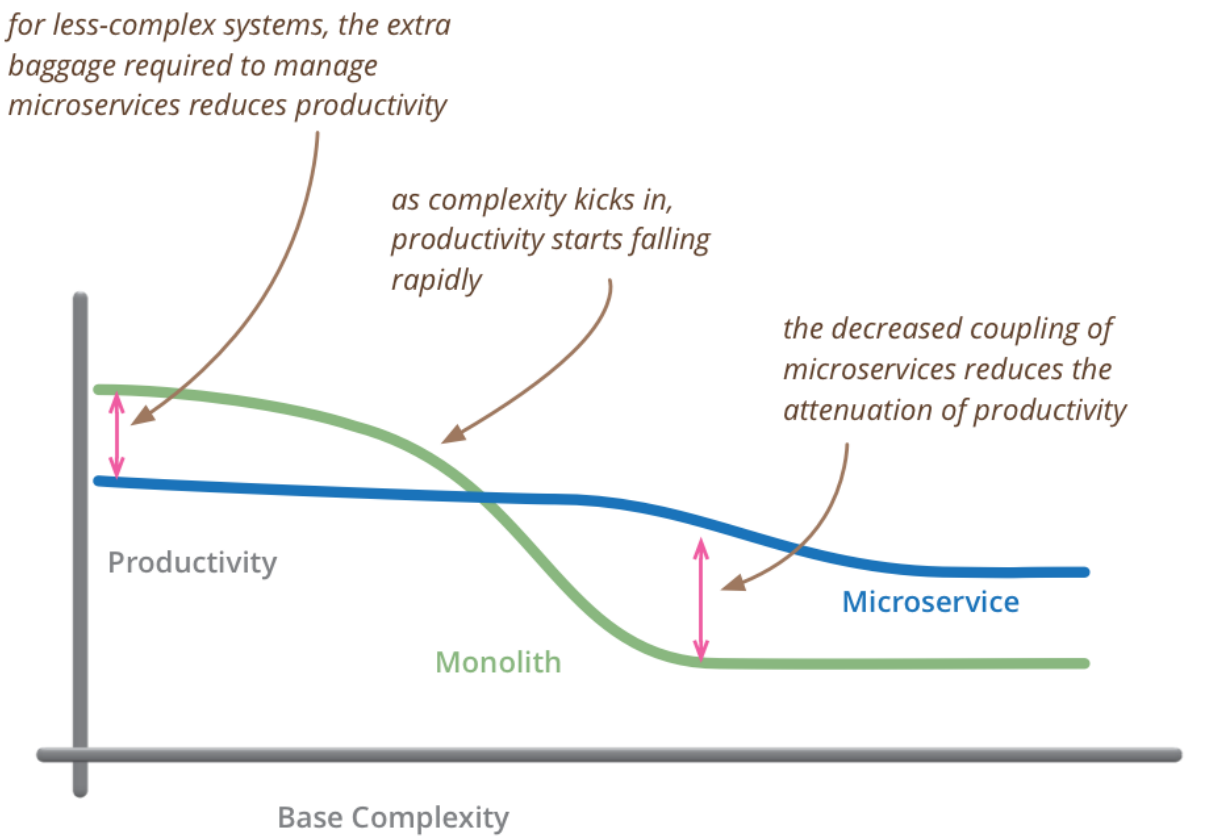
Zalety MSA

- Szybsza reakcja na zmiany w biznesie
- Niezależne i częstsze publikacje
  - Jeśli coś pójdzie nie tak, nie psuje to całej aplikacji
- Silna separacja komponentów
  - … i tym samym łatwiej je modyfikować i publikować
- Łatwiejsza skalowalność
- Większa odporność i bezpieczeństwo systemu
  - Awaria w jednej usłudze nie musi oznaczać awarii całego systemu
- Łatwiej odwzorować strukturę funkcjonalną organizacji (DDD)
- Potencjalna różnorodność technologii
  - Łatwiej uniknąć długu technologicznego, bo możemy przepisać jeden komponent bez konieczności przepisywania wszystkiego
- Sporo wsparcia w technologii
  - Lekkie protokoły
  - Usługi w chmurze, wirtualizacja
  - Wsparcie dla komunikacji asynchronicznej

Wady MSA

- Rozproszona architektura, której utrzymanie
  - więcej kosztuje
  - wymaga więcej kompetencji
- Testowanie jest trudniejsze
- Transakcje są często rozproszone po wielu komponentach
  - Trudno zrealizować spójność transakcyjną i zwykle kończy się na „eventual consistency”
  - Wymaga koordynacji pomiędzy zespołami
- Komunikacja pomiędzy komponentami
  - Jest wolniejsza
  - Bardziej podatna na błędy

Kiedy używać?



but remember the skill of the team will outweigh any monolith/microservice choice

https://martinfowler.com/bliki/MicroservicePremium.html

Design Principles

- Wysoka spójność
  - SOLID: Single Responsibility Principle
    - tylko jeden powód na zmianę usługi
  - Ten „powód” może wynikać z obszaru lub funkcji biznesowej
    - Ważne, żeby go jasno ustalić
  - Jeśli trzeba:
    - tworzymy nowe usługi
    - przebudowujemy istniejące

Design Principles

- Autonomiczność
  - Luźne powiązania oparte na kontraktach i interfejsach
  - Komunikacja
    - Synchroniczna
    - Asynchroniczna (np. przez publikację i obsługę zdarzeń)
  - Niezależność technologiczna
    - Unikanie publikacji bibliotek klienckich. Wyjątki?
  - Unikamy współdzielenia baz danych i bibliotek
  - Usługa ma własny cykl życia i zmiany w innych komponentach powinny być niezależne
  - Wsteczna kompatybilność (Open-Closed principle)
  - Strategia wersjonowania
    - Utrzymywanie dwóch wersji przez pewien czas
      - Warto przemyśleć odpowiednie endpointy (np. /V2/invoice/)
    - Semantic versioning: Major.Minor.Patch (np. 1.7.12)

Design Principles

- Business Domain Centric
  - Granice funkcjonalności powinny być wyznaczone przez funkcję biznesową
    - Związek z Bounded Context z DDD
    - Wsparcie dla Domain lub Sub Domain
    - Granice powinny się dostosowywać wraz z rozwojem domain lub subdomain
  - Jasny interfejs i kontrakt „na zewnątrz”
  - Aspekt techniczny
    - Niepożądany, ale czasami pragramatyczny
      - Problem z wydajnością
      - Osobna usługa do archiwizacji danych

Design Principles

- Odporność na awarie
  - Przygotować system na spodziewane problemy
    - Np. wynikające z interakcji z usługami zależnymi
    - Przykładowe strategie to
      - Wyłączyć ten rodzaj funkcjonalności np. pokazywanie promocji
      - Udostępnić podstawową lub ograniczoną funkcjonalność (np. ceny domyślne bez rabatów)
  - Fail fast
    - Jeśli coś idzie nie tak, szybki feedback do użytkownika
    - Nie przeciągać transakcji
    - Sterowanie przez timeouty (globalne, per transakcja, itp.)
  - Problemy mogą być na wielu poziomach: wyjątki, opóźnienia, niedostępność, problemy z siecią
    - Monitorować timeouty do późniejszej analizy

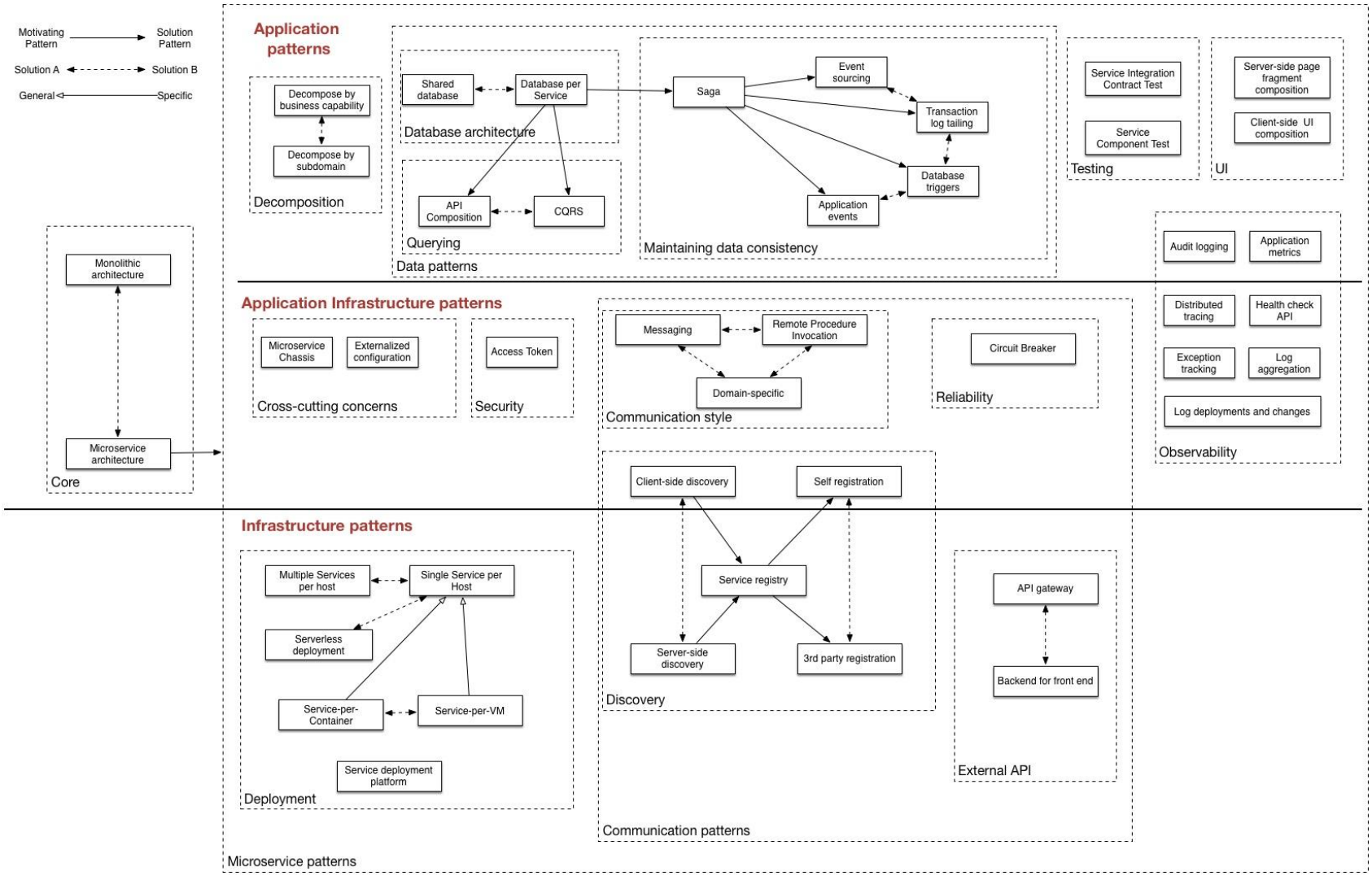
Design Principles

- Monitoring i logging
  - Status systemu (health check)
  - Zcentralizowany monitoring
    - Monitorowanie na różnych poziomach
      - Infrastruktury, transakcji biznesowych
    - Monitorowanie różnych aspektów
      - Timeouty, czasy odpowiedzi, dostępy do danych, procesy biznesowe (np. czas od zatwierdzenia koszyka do złożenia płatności)
    - Określenie SLA i generowanie powiadomień w przypadku jej niedotrzymania
    - Narzędzia do wizualizacji danych
    - Rozwiązywanie problemów i planowanie skalowania
  - Zcentralizowany logging
    - Rejestracja istotnych zdarzeń, aby sprawnie rozwiązywać problemy
    - Rejestracja odpowiednich danych: czasu, hosta, komunikatu, itd.
    - Rejestracja CorrelationID do śledzenia rozproszonych transakcji

Design Principles

- Automatyzacje
  - Continues Integration
    - Wykonanie unit i integration tests
  - Continues Deployment, DevOps
    - Daje bardzo szybką publikację zmian
    - Współpracuje zwykle z CI
    - Użytkownik bardziej zadowolony

Przegląd wzorców



Podział na microserwisy

Podział na Microserwisy

- Jak dzielić?
  - Business capability
  - Struktura organizacyjna
    - Conway Law
- Konsekwencje podziału i problemy

"Any organization that designs a system … will inevitably produce a design whose structure is a copy of the organization's communication structure."

-- Melvin Conway, 1968

Transakcje i spójność danych

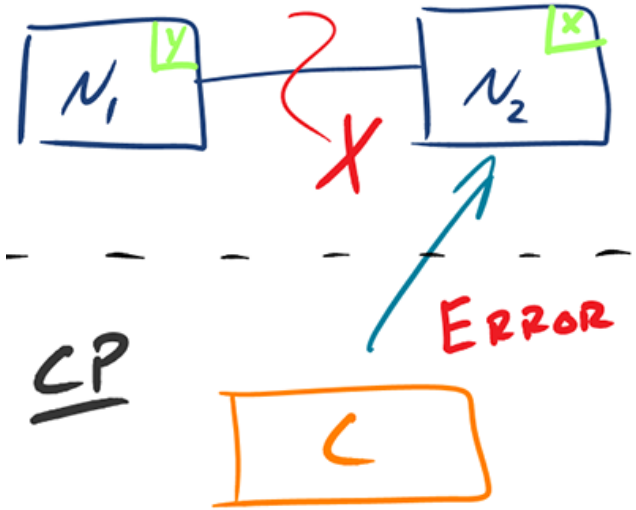
CAP Theorem

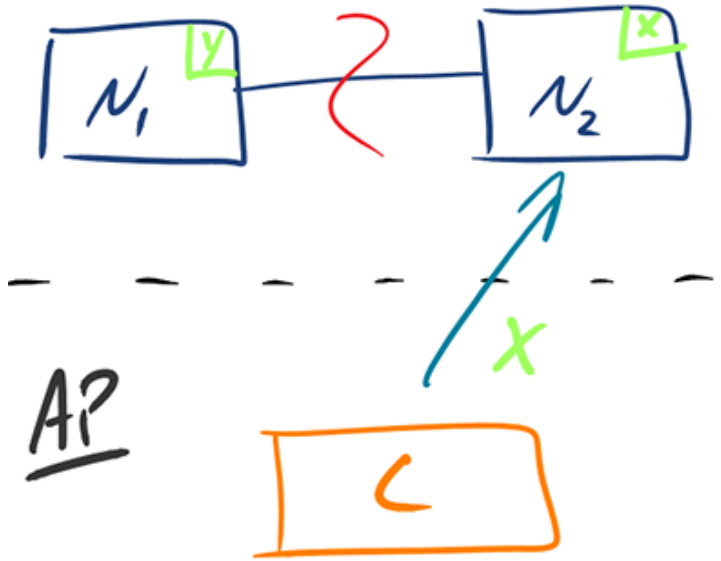
- CAP to
  - Consistency
    - Każdy węzeł udostępnia te same, najnowsze dane
    - Każdy klient ma widok na te same dane
  - Availability
    - Każdy aktywny węzeł odpowiada na wszystkie żądania danych w rozsądnym czasie
  - Partition Tolerance
    - System działa pomimo podziału sieci

https://dzone.com/articles/understanding-the-cap-theorem

CAP Theorem

- Teoria mówi, że nie można mieć wszystkiego, czyli albo CA, albo CP, albo AP
- Ponieważ MSA wymusza P, pozostaje nam





http://robertgreiner.com/2014/08/cap-theorem-revisited/

2PC jest kłopotliwy

- Gwarantuje spójność, ale…
  - 2PC coordinator – single point of failure
  - CAP theorem – 2PC mocno ogranicza dostępność
  - Blokady mocno ograniczają dostępność danych
  - Nie wspierane przez wiele systemów
    - np. bazy NoSQL

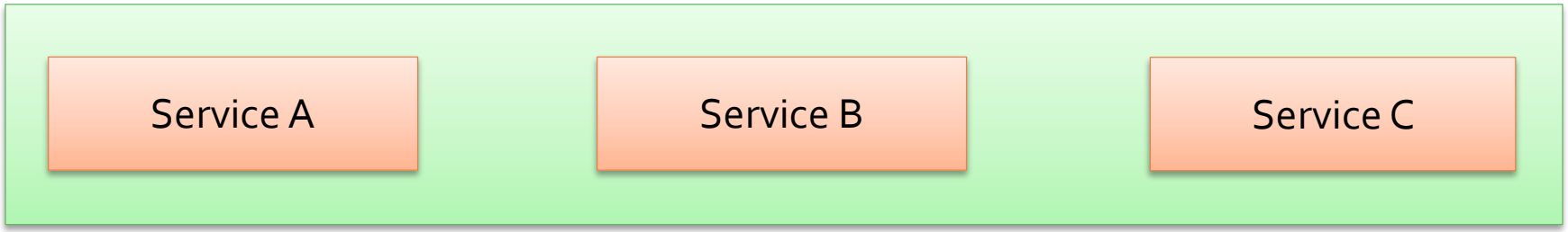
ACID vs. BASE

- A może zmieńmy oczekiwania…
- ACID → BASE
  - Basic Availability
    - The database appears to work most of the time.
  - Soft-state
    - Stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.
  - Eventual consistency
    - Stores exhibit consistency at some later point (e.g., lazily at read time).

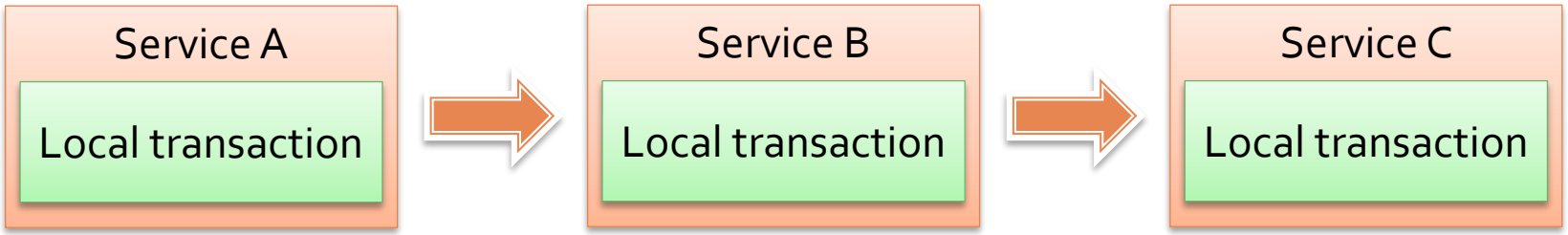
https://neo4j.com/blog/acid-vs-base-consistency-models-explained/

Saga

- Transakcja rozproszona



- Saga

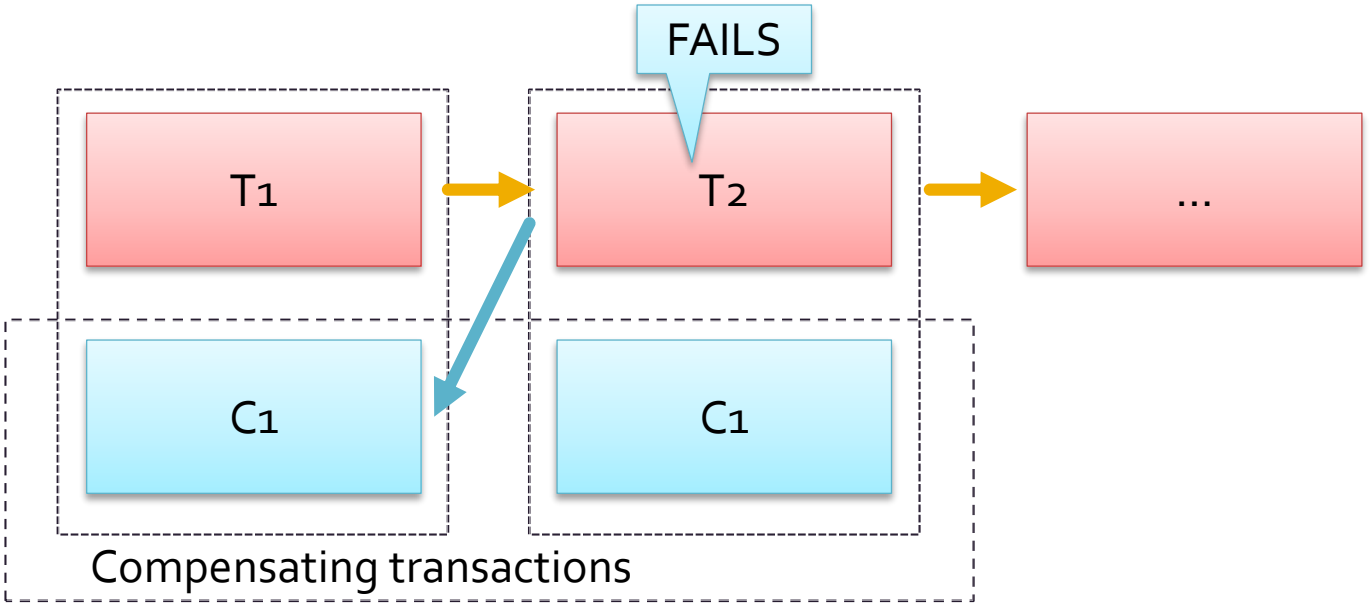


Saga

- Saga to sekwencja lokalnych transakcji
- Każda lokalna transakcja
  - Dokonuje lokalnych zmian
  - Publikuje zdarzenie (wysyła wiadomość) aby uruchomić kolejną lokalną transakcję

Wycofywanie transakcji

- Wprowadzamy „compensating transactions”
- Dla każdej transakcji „eventually consistent” należy napisać logikę „rollback’u”



Interakcja z UI

- Monitorowanie i pokazywanie stanu Sagi
  - Po zakończeniu
  - Stany pośrednie
- Anulowanie realizowanej Sagi
  - Uniemożliwić
  - Po prostu przerwać
    - Po wykonaniu bieżącej transakcji lokalnej vs. natychmiast
    - Konsekwencje biznesowe

Koordynacja Sagi

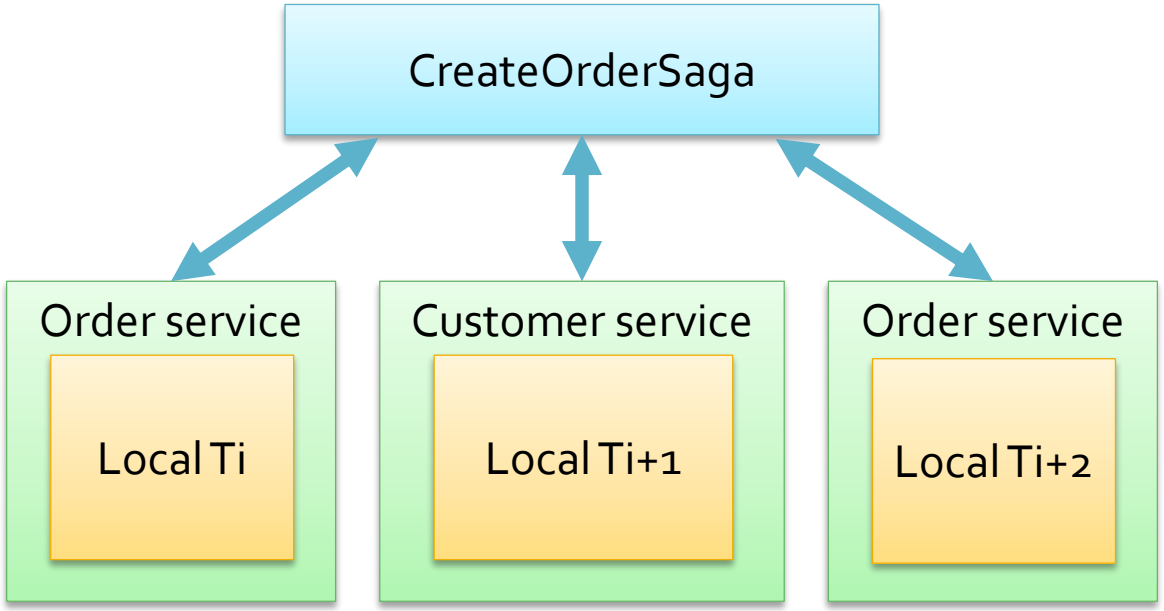
- Jak określić, co robić po skończeniu transakcji Ti?
- **Choreografia**: decyzja rozproszona

vs.

- **Orkiestracja**: decyzja scentralizowana

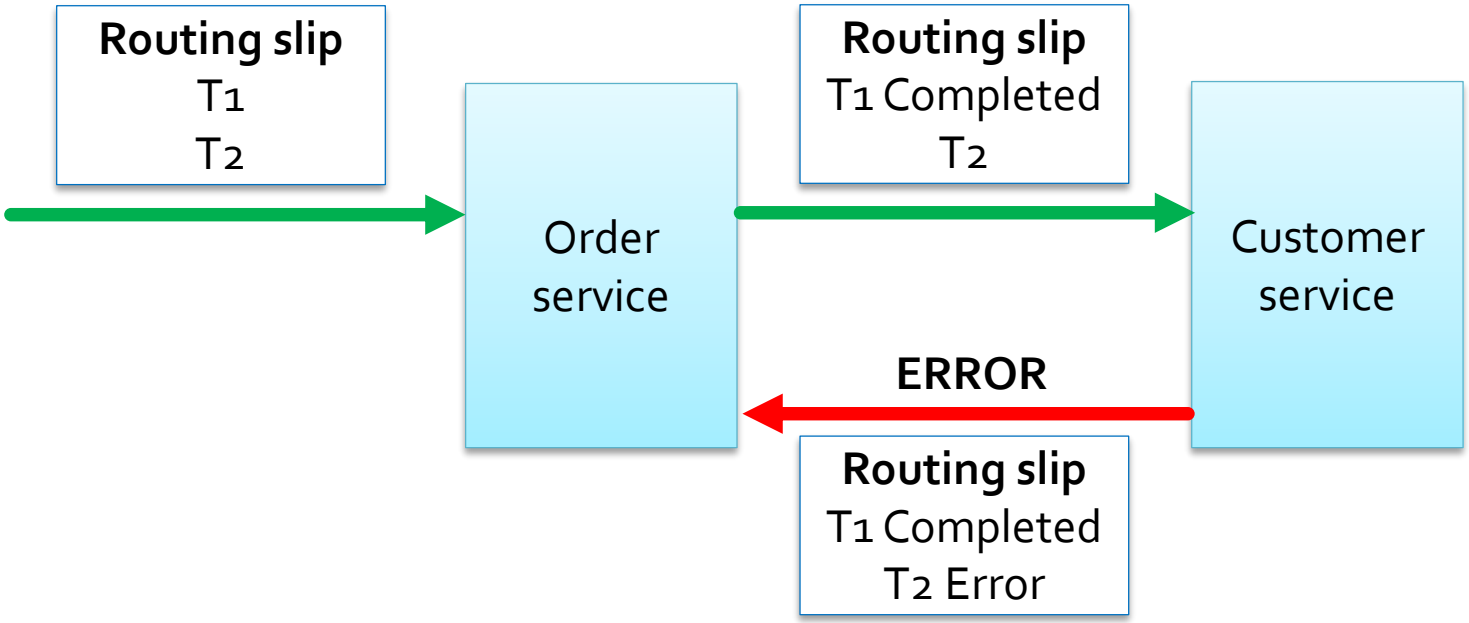
Orkiestracja

- Orkiestrator monitoruje stan i uruchomia kolejne transakcje lokalne (lub je wycofuje)



Choreografia

- Kolejne transakcje są uruchamiane przez publikację zdarzeń
- Dodatkowo może być przekazywany tzw. „routing slip”, które zawiera listę kolejnych kroków oraz stan bieżący



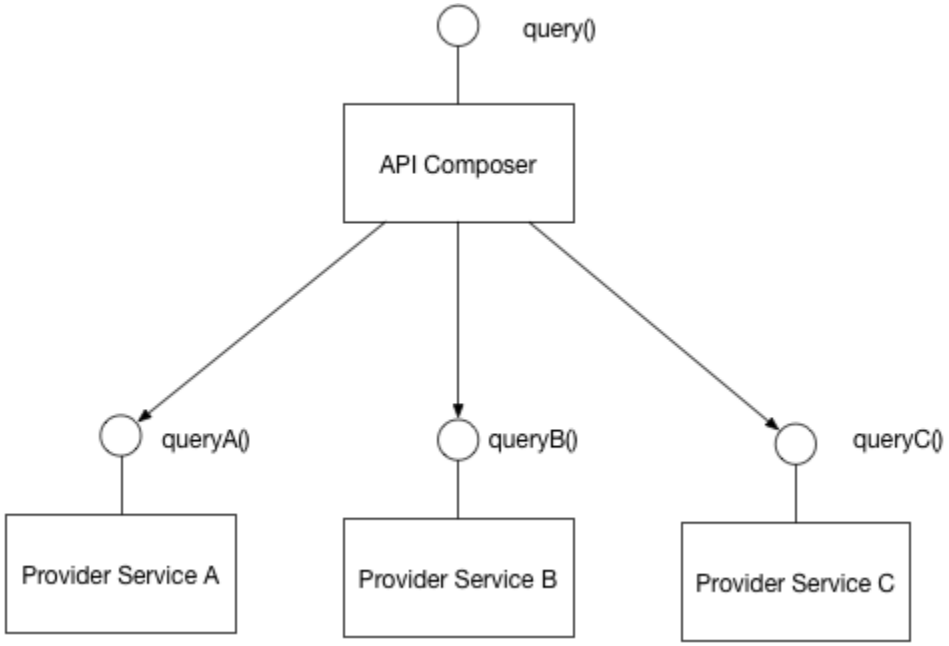
Saga podsumowanie

- Zagadnienie jest dużo szersze
  - Można rozważać zalety i wady różnych metod koordynacji
  - Można analizować różnego rodzaju anomalię i wprowadzać mechanizmy łagodzące skutki
- Więcej:
  - http://microservices.io/microservices/general/2018/03/22/microxchg-sagas.html
  - http://vasters.com/archive/Sagas.html
  - https://blog.bernd-ruecker.com/saga-how-to-implement-complex-business-transactions-without-two-phase-commit-e00aa41a1b1b

Realizacja zapytań

Realizacja zapytań

- CQRS
- Api Composition



External API

External API

- API Gateway
  - Główny punkt wejścia dla infrastruktury MSA
  - Może implementować
    - Load balancing, Caching, Security
  - Pozwala na
    - Jeden punkt dostępowy do wszystkich Microservices
    - Bardziej dynamiczne zarządzenia infrastrukturą MS
    - Lepszy routing, np. klientów z Europy kierować do odpowiednich serwerów

Technologie

Technologie

- Komunikacja
  - Synchroniczna
  - Asynchroniczna
    - Zdarzenia i kolejki, platforma integracyjna
  - Oparta o RPC, HTTP, REST
- Hosting
  - Wirtualizacja
  - Containers
    - Wirtualizacja w bardziej granularnym wymiarze np. Docker
  - Własne maszyny lub IaaS

Technologie

- Rejestracja i katalog
  - Pożądanym jest posiadanie repozytorium dostępnych usług
  - Usługa może rejestrować się sama, rejestracja może być też zapewniona przez infrastrukturę
  - Infrastruktura może monitorować usługi i wyrejestrowywać te, które nie odpowiadają
- Monitoring
  - Na poziomie sieci: Nagios, PRTG
  - Na poziomie aplikacji: New Relic, BMC APM
- Logging
  - Rozwiązania: Splunk, Kibana, ważne automatyczne analizy
  - Śledzenie transakcji poprzez CorrelationID
  - Kwestia formatu logów
  - Dla MicroServices: koniecznie zcentralizowany

Technologie

- Skalowanie
  - Więcej instancji usługi vs. więcej zasobów dla usługi
  - Automatyczne vs. na żądanie
  - Load balancers
  - Kiedy: problemy z wydajnością (bieżące lub spodziewane)
- Caching
  - Są dużym wsparciem dla poprawy wydajności
  - Implementowane na różnych poziomach (proxy, client)

Kto tego używa?

- Comcast Cable
- Uber
- Netflix
- Amazon
- Ebay
- Sound Cloud
- Karma
- Groupon
- Hailo
- Gilt
- Zalando
- Capital One Why Capital One is at Re:Invent and Keynote
- Lending Club
- AutoScout24

How we build microservices at Karma

"Microservices" and "Microservice Architecture" are hot buzz words in the development community right now, but concrete examples of microservices in production are still scarce. I thought it might help to give a brief overview of how we've utilized microservices for our backend API at Karma over the past couple of years. It's not exactly a "how-to," more like a "why-to," or a "wherefore-to," but hopefully it will give you some insight as to whether microservices are appropriate for your application, and how you can go about using them.

the UK Government Digital Service
Real Estate
Property & Homes For Sale
Forward
Twitter
PayPal
Bluemix
The Guardian

Literatura

- https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/
- https://github.com/dotnet-architecture/eShopOnContainers
- http://microservices.io/
- http://martinfowler.com/articles/microservices.html
- http://martinfowler.com/microservices/
- https://smartbear.com/learn/api-design/what-are-microservices/
- https://en.wikipedia.org/wiki/Microservices
- https://blog.karmawifi.com/how-we-build-microservices-at-karma-71497a89bfb4#.nz66s9wxk
- http://www.slideshare.net/InfoQ/from-a-monolith-to-microservices-rest-the-evolution-of-linkedins-service-architecture
- http://www.pwc.com/us/en/technology-forecast/2014/cloud-computing/features/microservices.html
- https://www.youtube.com/watch?v=PY9xSykods4
- https://www.youtube.com/watch?v=wgdBVIX9ifA