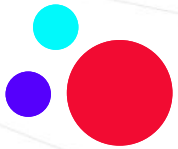
Architektura i narzędzia w systemach mikroserwisowych

infoShare Academy

Oleksii Tsyganov

infoShareAcademy.com

01. Agenda



Agenda

Welcome
History of Microservices
Problems of Monolith & SOA
Microservices Architecture
Problems Solved by Microservices
Designing Microservices Architecture
Deploying Microservices
Testing Microservices
Service Mesh
Logging And Monitoring
When NOT to use Microservices
Microservices and the Organization
Anti-Patterns and Common Mistakes
Breaking Monolith to Microservices
Case Study
Conclusion

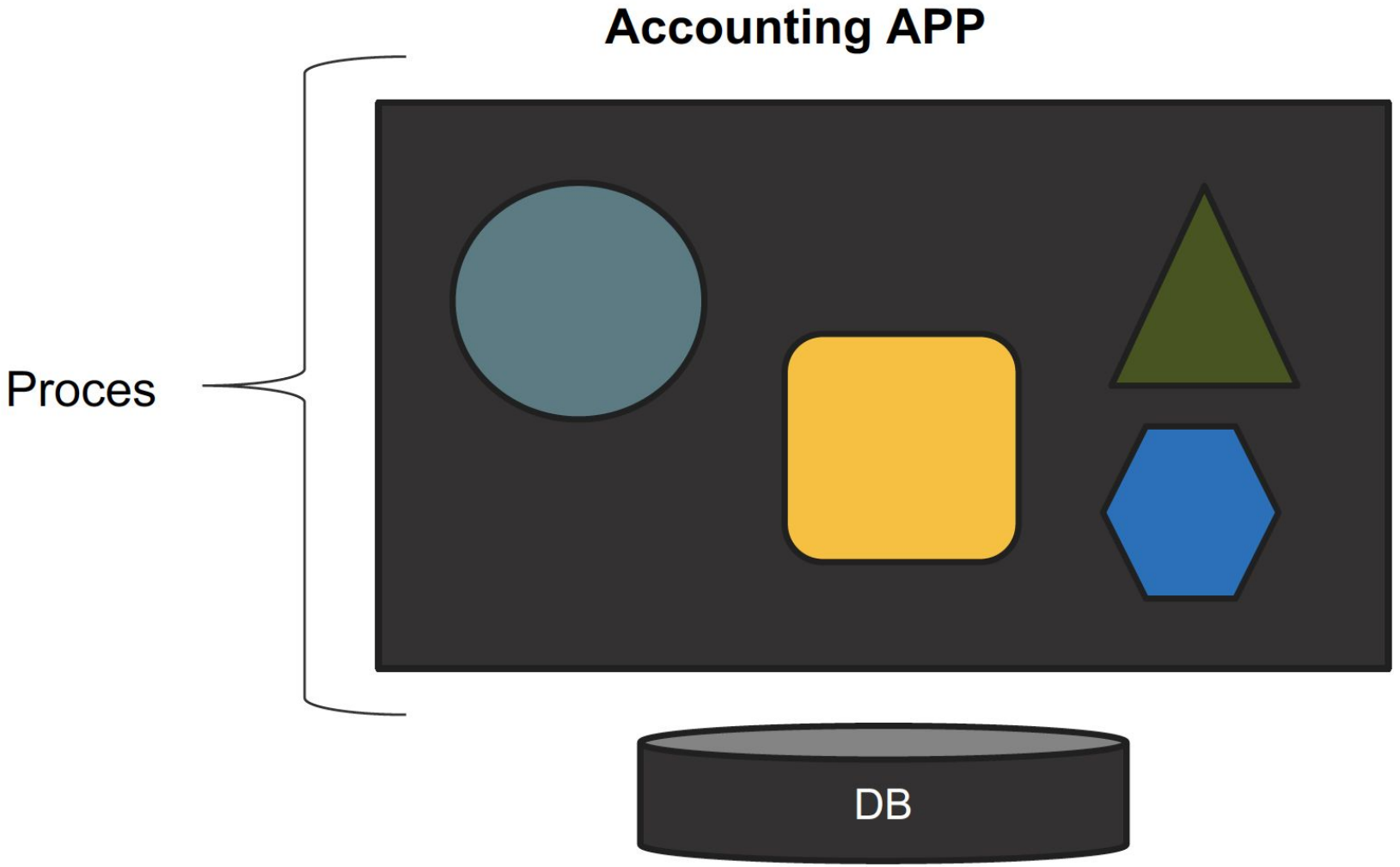
Before microservices. Monolith.

Jedyny proces

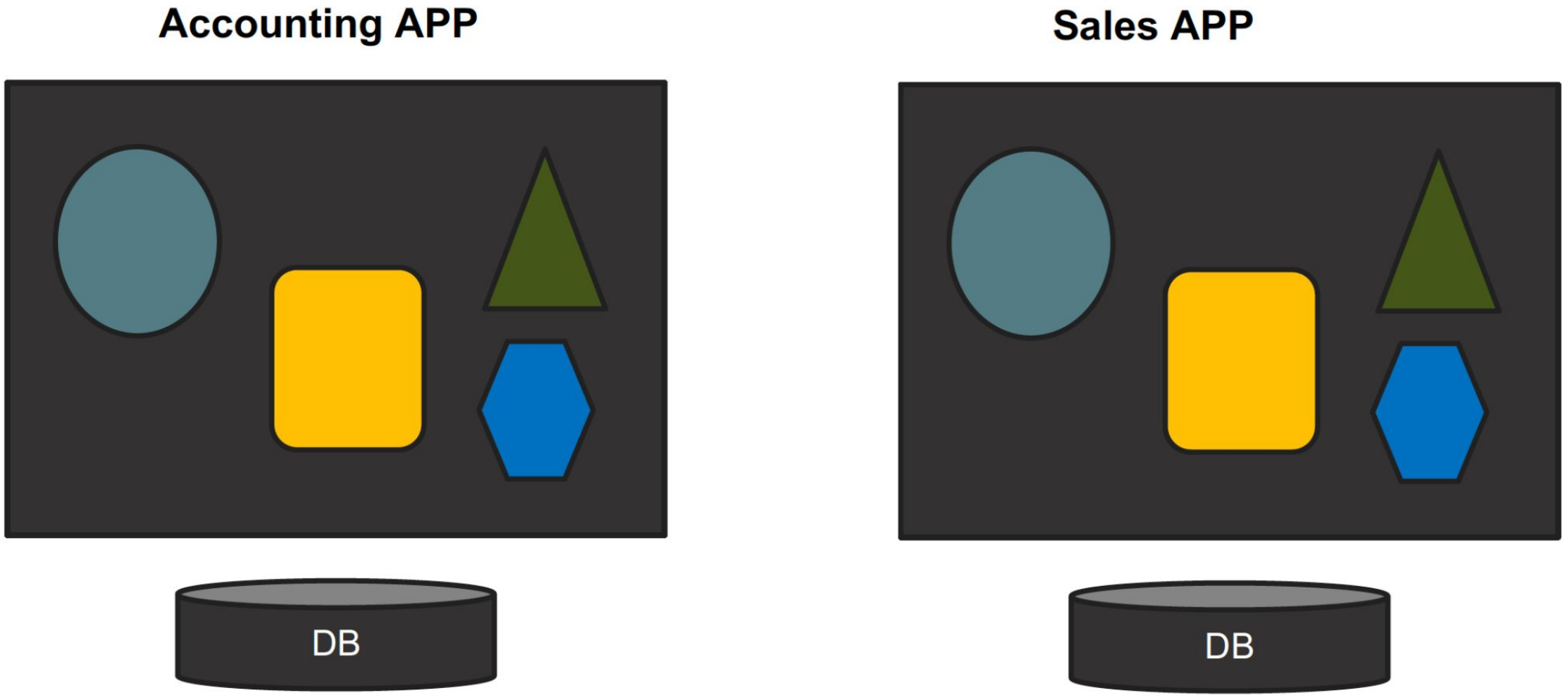
Związki pomiędzy wszystkimi klasami

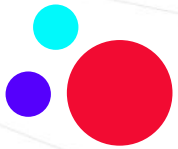
Implementowana jako “silos”

Before microservices. Monolith.



Before microservices. Monolith.



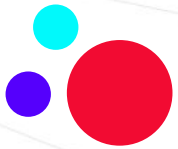


Monolith. Pros.

Łatwo projektować

Wydajność?

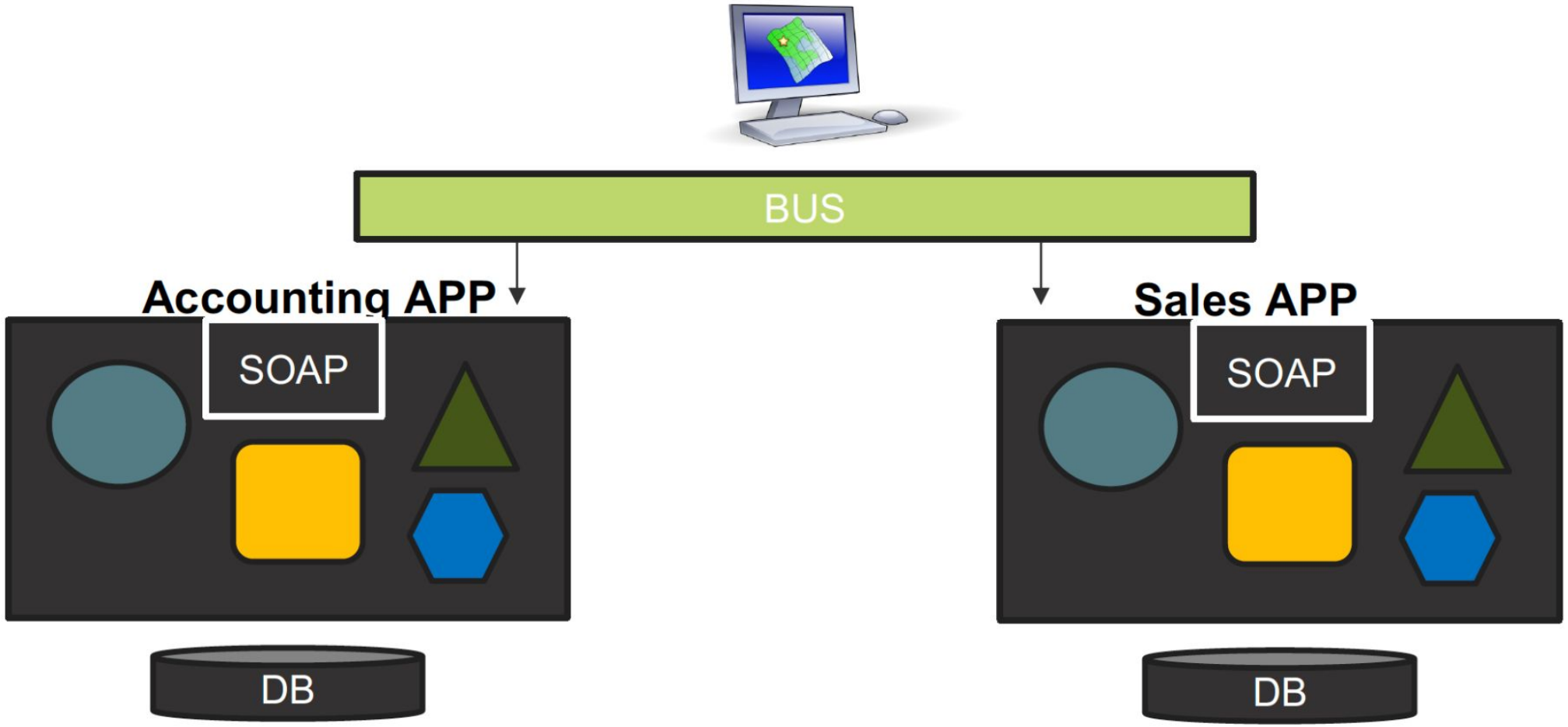
infoShare ACADEMY



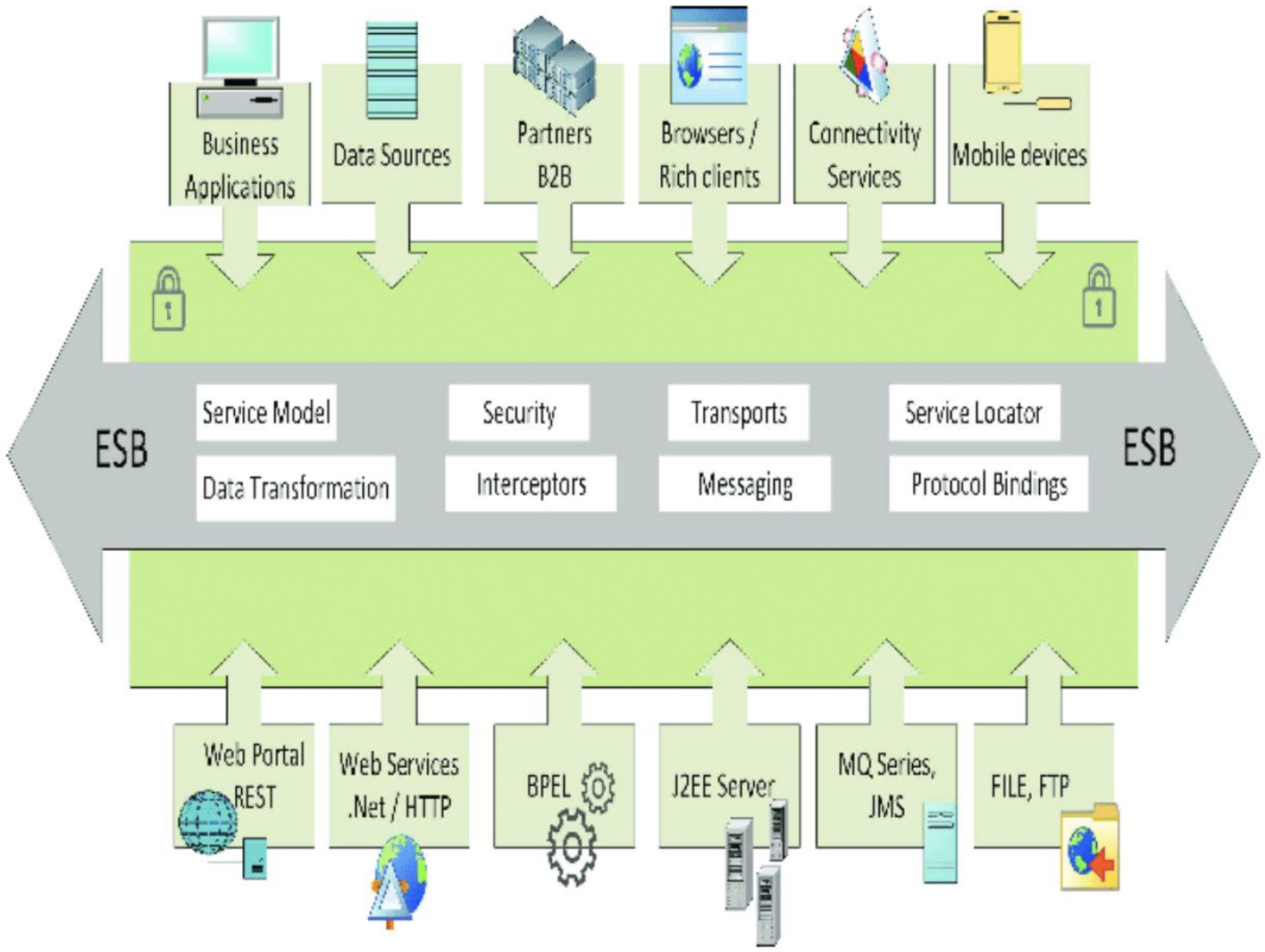
Before microservices. Service Oriented Architecture

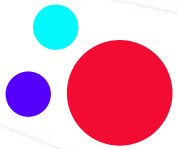
- Aplikację są serwisami, które wystawiają swoją funkcjonalność “w świat”
- Dzielenie się i dawanie
- Serwisy wystawiają metadane żeby zaznaczyć swoją funkcjonalność
- Zwykle implementowane za pomocą SOAP i WSDL
- Implementowane za pomocą Enterprise Service Bus (ESB)

Before microservices. Service Oriented Architecture



Before microservices. Service Oriented Architecture

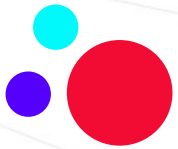




SOA. Pros.

Udostępnianie danych i funkcjonalności

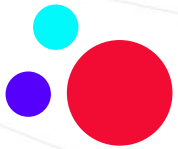
Polyglotowe podejście



Problems with Monolith and SOA

Jedyna platforma technologiczna:

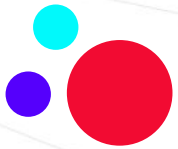
- wszystkie komponenty muszą być dewelopowane na tej samej platformie
- nie zawsze technologia jest lepsza dla zadania
- upgrade jest problemem



Problems with Monolith and SOA

Nieelastyczne wdrażanie:

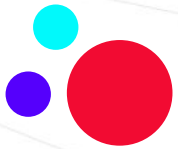
- Zawsze deployujemy całą aplikacje
- Nie ma możliwości deployu częściowego
- Update tylko jednego komponentu wymaga deploymentu całości “codebase”
- Wymuszanie uciążliwego procesu testów całości
- Długie cykle deploymentowe



Problems with Monolith and SOA

Nieelastyczne wdrażanie:

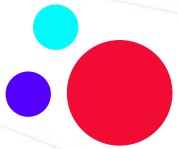
- W monolicie CPU i RAM dzielone pomiędzy wszystkimi komponentami
- Nie ma możliwości przydzielenia większej liczby zasobów dla komponenta systemu



Problems with Monolith and SOA

Skala i skomplikowaność:

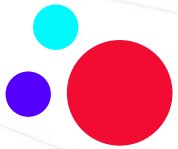
- “Codebase” jest skomplikowany i duży
- Mała zmiana może spowodować problemy z innymi komponentami
- Testowanie nie zawsze wykrywa bugi
- Wsparcie dla produktu jest bardzo trudne
- Bardzo trudne utrzymywanie
- Przestarzały system



Problems with Monolith and SOA

Skomplikowana i droga ESB:

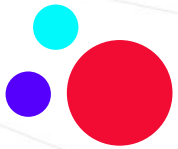
- ESB jest głównym komponentem
- Szybko staje przeciążony i drogi
- Próbuje robić „wszystko”
- Bardzo skomplikowany w utrzymaniu



Problems with Monolith and SOA

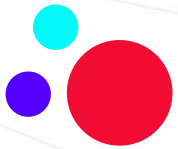
Brak narzędzi:

- dla SOA potrzebne krótkie cykle developmentu
- potrzeba w szybkim testowaniu
- nie ma narzędzi wspierających powyżej wskazane
- nie osiągnięto żadnego oszczędzania czasu w porównaniu z monolitem



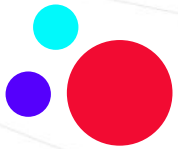
Architektura mikroserwisowa

- Problemy z monolitem i SOA przeprowadziły do nowego paradygmatu
- Musi być modularny, z prostym API
- Pojawił się w 2011, ale realnie otrzymał życie w 2014 roku
- Martin Fowler “Microservices” - standard “de-facto”



Cechy mikroserwisów

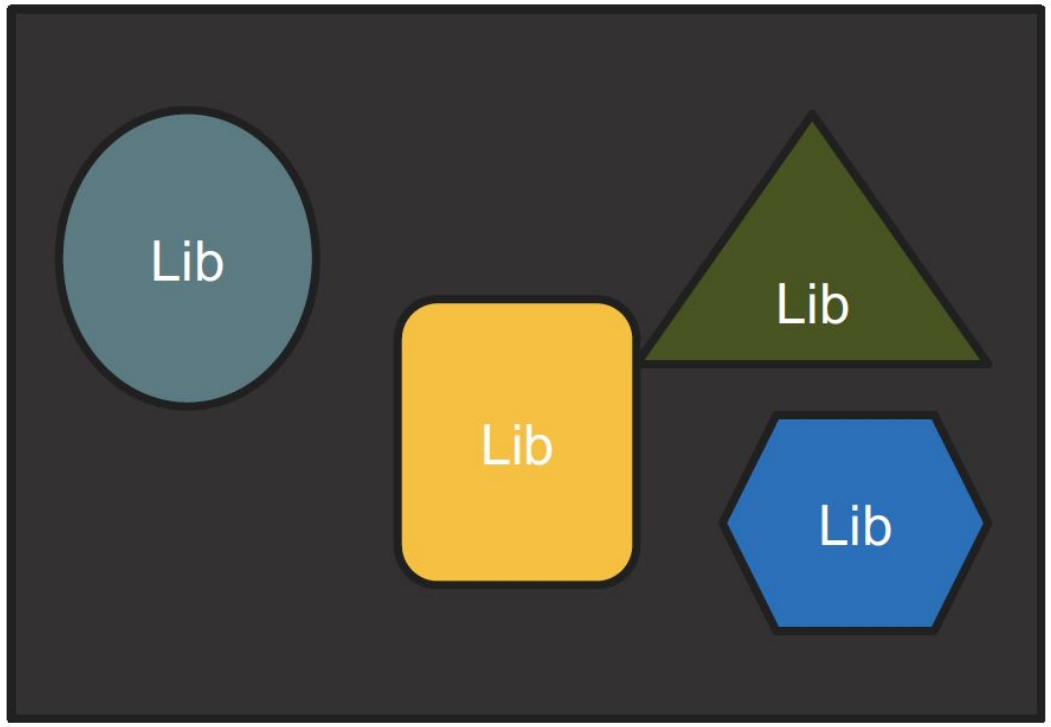
- Componentization via Services
- Organizes Around Business Capabilities
- Products not Projects
- Smart Endpoints and Dumb Pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for Failure
- Evolutionary Design

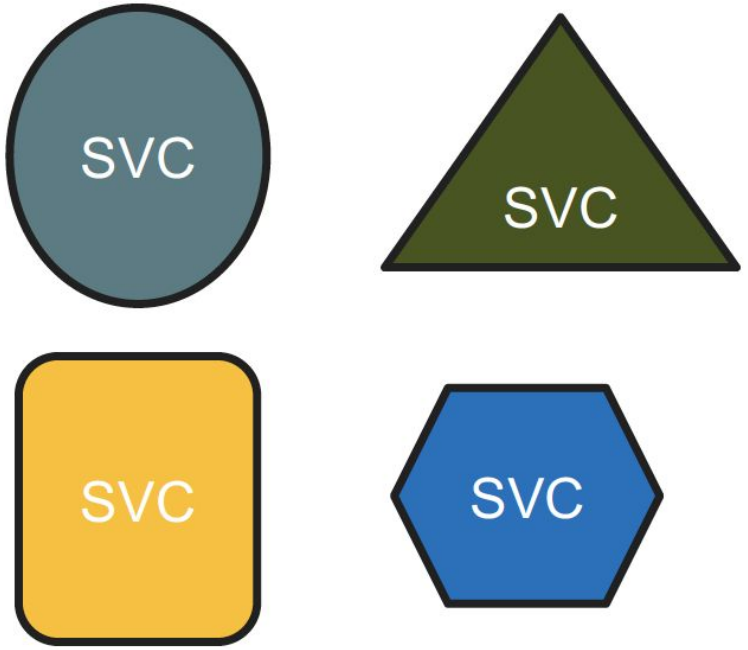


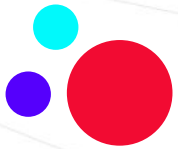
Componentization via Services

- Design modułowy zawsze jest dobrym pomysłem
- Komponenty są częściami które tworzą produkty
- Modularność może być osiągnięta poprzez:
  - biblioteki – wywołane bezpośrednio w procesie
  - serwisy – wywołane poza procesem (API, RPC)
- W mikroserwisach oczywiście preferujemy serwisy a nie biblioteki

Componentization via Services

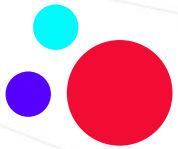






Organized Around Business Capabilities

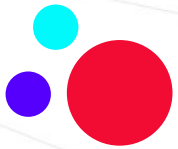
- **Tradycyjne projekty**:
- Mają zespoły z poziomową odpowiedzialnością: UI, DB, API
- Bardzo powolna komunikacja pomiędzy zespołami
- Nie używają tej wspólnej terminologii
- **Nie mają wspólnych celów**



Organized Around Business Capabilities

Zalety:

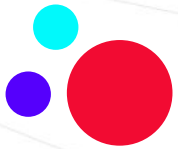
- Szybki development
- Dobrze oznaczone granice pomiędzy serwisami; również ludzie wiedzą, co trzeba robić



Products not Projects

Tradycyjne projekty:

- Celem jest dostarczenie pracującego kodu
- Nie ma trwałych relacji z klientem
- Często w ogóle nie znają klienta
- Po dostarczeniu kodu zespół przechodzi do kolejnego projektu

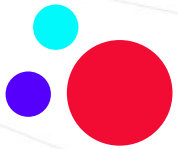


Products not Projects

Mikroserwisowe projekty:

- Celem jest dostarczenie pracującego produktu
- Produkt wymaga stałego wsparcia i bliskiej współpracy z klientem
- Zespół odpowiedzialny po dostarczeniu serwisu

“You build it, you run it”
W. Vogels, AWS CTO

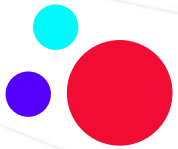


Smart Endpoints and Dumb Pipes

SOA projekty używają:

- ESB
- WS-* protokoły: WS-security, WS-messaging, WS-discovery

Pomiędzy serwisowa komunikacja stała zbyt skomplikowana i trudna do utrzymania

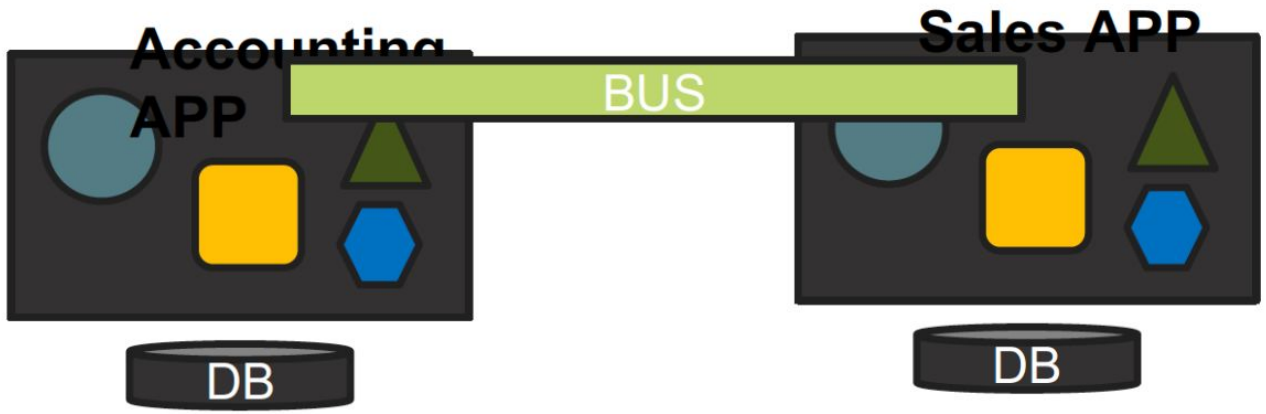


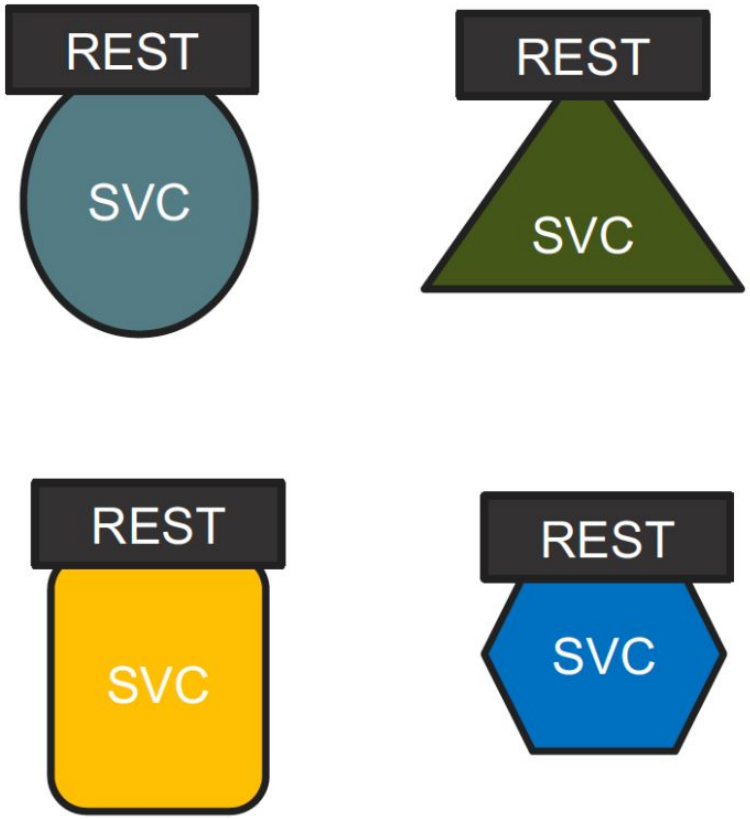
Smart Endpoints and Dumb Pipes

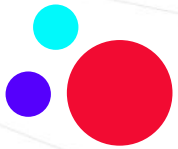
Systemy mikroserwisowe:

- Używają dumb pipes - prostych protokołów
- Nie wymyślają nowego, używają to co WEB proponuje
- Zwykłe REST API

Smart Endpoints and Dumb Pipes







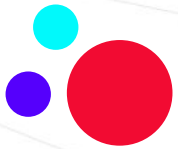
Smart Endpoints and Dumb Pipes

Ważne:

- Bezpośrednie połączenie pomiędzy serwisami nie jest dobrą opcją
- Lepsze rozwiązanie – używanie service discovery lub gateway
- Więcej protokołów w ostatnie lata (GraphQL, gRPC) – są skomplikowane

Co wygrywamy używając Smart Endpoint oraz Dumb Pipes:

- Przyspieszamy development aplikacji
- Robimy aplikacje prostą do obsługi



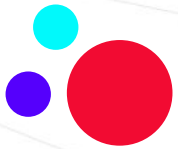
Decentralized Governance

Klasyczne projekty:

- Jest standard do prawie wszystkiego

Mikroserwisy:

- Każdy zespół jest odpowiedzialny za swoje decyzje
- każdy zespół jest odpowiedzialny za swój serwis
- System poliglota



Decentralized Data Management

Klasyczne projekty:

- Jedna baza danych - przechowuje wszystkie dane komponentów systemu

Mikroserwisy:

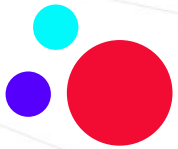
- Każdy serwis może mieć swoją bazę danych

Infrastructure Automation

Azure DevOps

GitLab

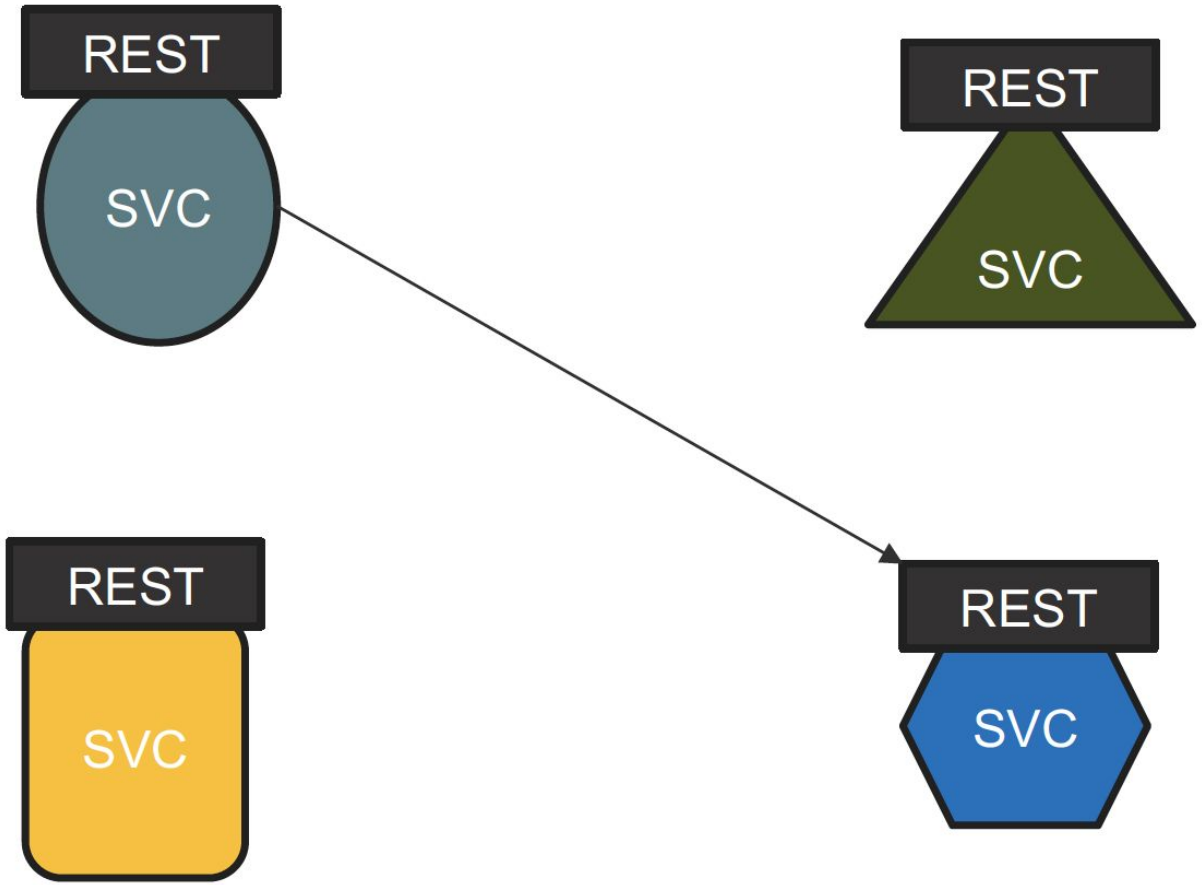
Jenkins



Design for Failure

- W architekturze mikroserwisowej dużo procesów I dużo ruchu sieciowego
- Dużo może pójść nie tak
- Kod musi ogarnąć takie błędy w prawidłowy sposób
- Trzeba logować i monitorować

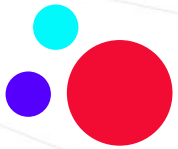
Design for Failure



Catch Exception

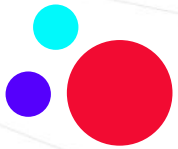
Retry

Log exception



Evolutionary Design

- Przejście na mikroserwisy powinno być postępowe
- Nie trzeba rozwalać wszystkiego i zaczynać od nowa
- Zmieniamy każdą część postępowo



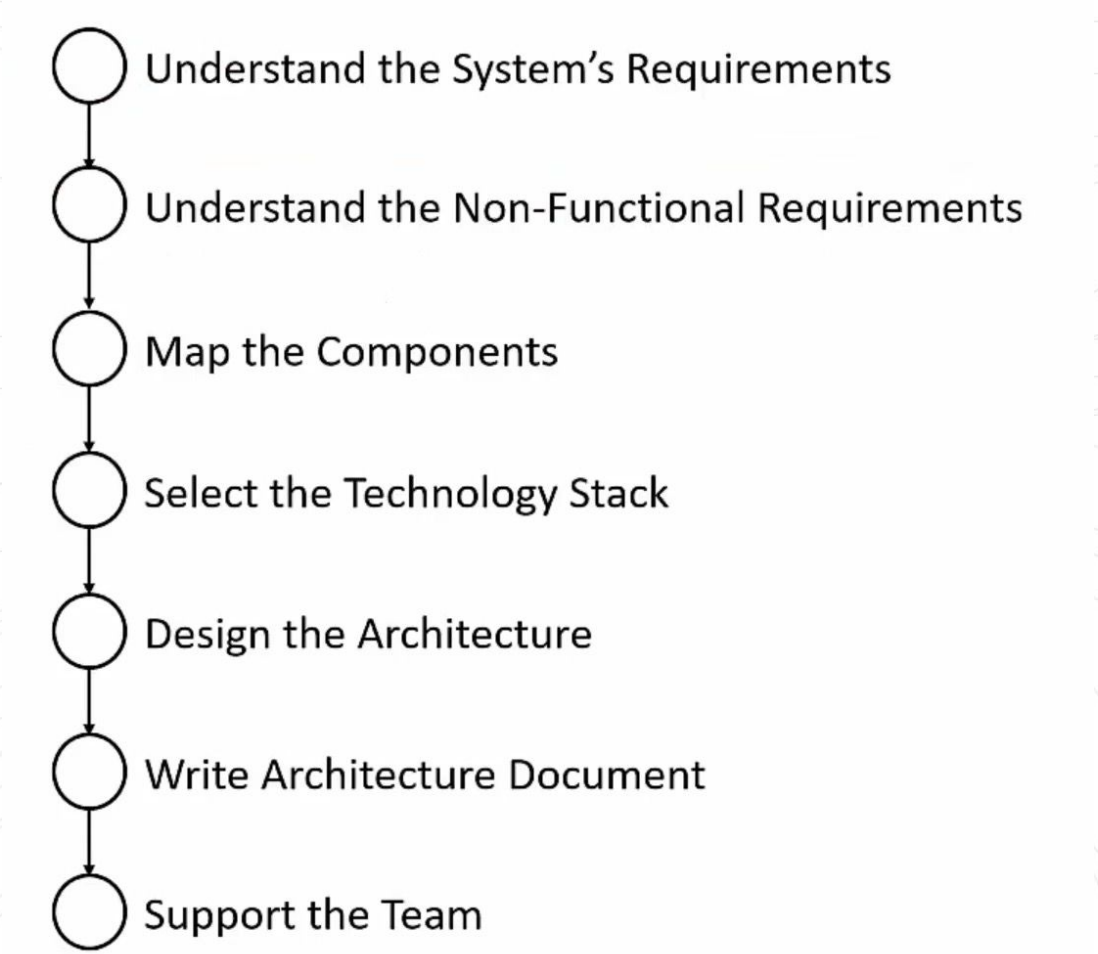
Problemy rozwiązywane za pomocą mikroserwisów

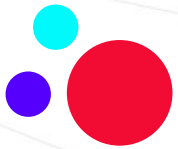
- Jedyna platforma technologiczna
- Nieelastyczny deployment
- Nieskuteczne zużycie zasobów
- Duża skala i skomplikowaność
- Drogie szyny serwisowe (ESB)
- Brak narzędzi

Projektowanie architektury mikroserwisowej

- Trzeba użyć metodologii
- Do not rush
- Więcej planuj, koduj mniej

Projektowanie architektury mikroserwisowej



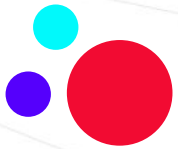


Projektowanie architektury mikroserwisowej. Mapowanie komponentów

- Najbardziej ważny krok całego procesu
- Definiuje jak system będzie wyglądał w przyszłości
- Ustalony - niełatwo zmienić

Mapowanie - definiowanie elementów systemu, z których tworzy się cały system.

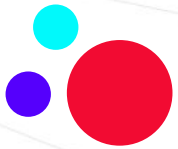
Komponenty = Serwisy



Projektowanie architektury mikroserwisowej. Mapowanie komponentów

Mapowanie musi bazować na:

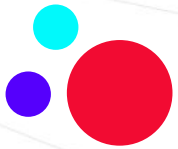
- **Wymaganiach biznesowych**
- Autonomia funkcjonalna
- Jednostki danych
- Autonomia danych



Projektowanie architektury mikroserwisowej. Mapowanie komponentów

Mapowanie musi bazować na:

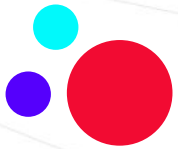
- Wymaganiach biznesowych
- **Autonomia funkcjonalna**
- Jednostki danych
- Autonomia danych



Projektowanie architektury mikroserwisowej. Mapowanie komponentów

Mapowanie musi bazować na:

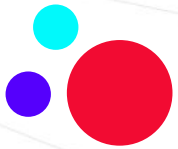
- Wymaganiach biznesowych
- Autonomia funkcjonalna
- **Jednostki danych**
- Autonomia danych



Projektowanie architektury mikroserwisowej. Mapowanie komponentów

Mapowanie musi bazować na:

- Wymaganiach biznesowych
- Autonomia funkcjonalna
- Jednostki danych (zamówienia, towary)
- **Autonomia danych**



Projektowanie architektury mikroserwisowej. Wybór stosu technologicznego

	App Types	Type System	Cross Platform	Community	Performance	Learning Curve
.NET	All	Static	No	Large	OK	Long
.NET Core	Web Apps, Web API, Console, Service	Static	Yes	Medium and growing rapidly	Great	Long
Java	All	Static	Yes	Huge	OK	Long
node.js	Web Apps, Web API	Dynamic	Yes	Large	Great	Medium
PHP	Web Apps, Web API	Dynamic	Yes	Large	OK -	Medium
Python	All	Dynamic	Yes	Huge	OK -	Short