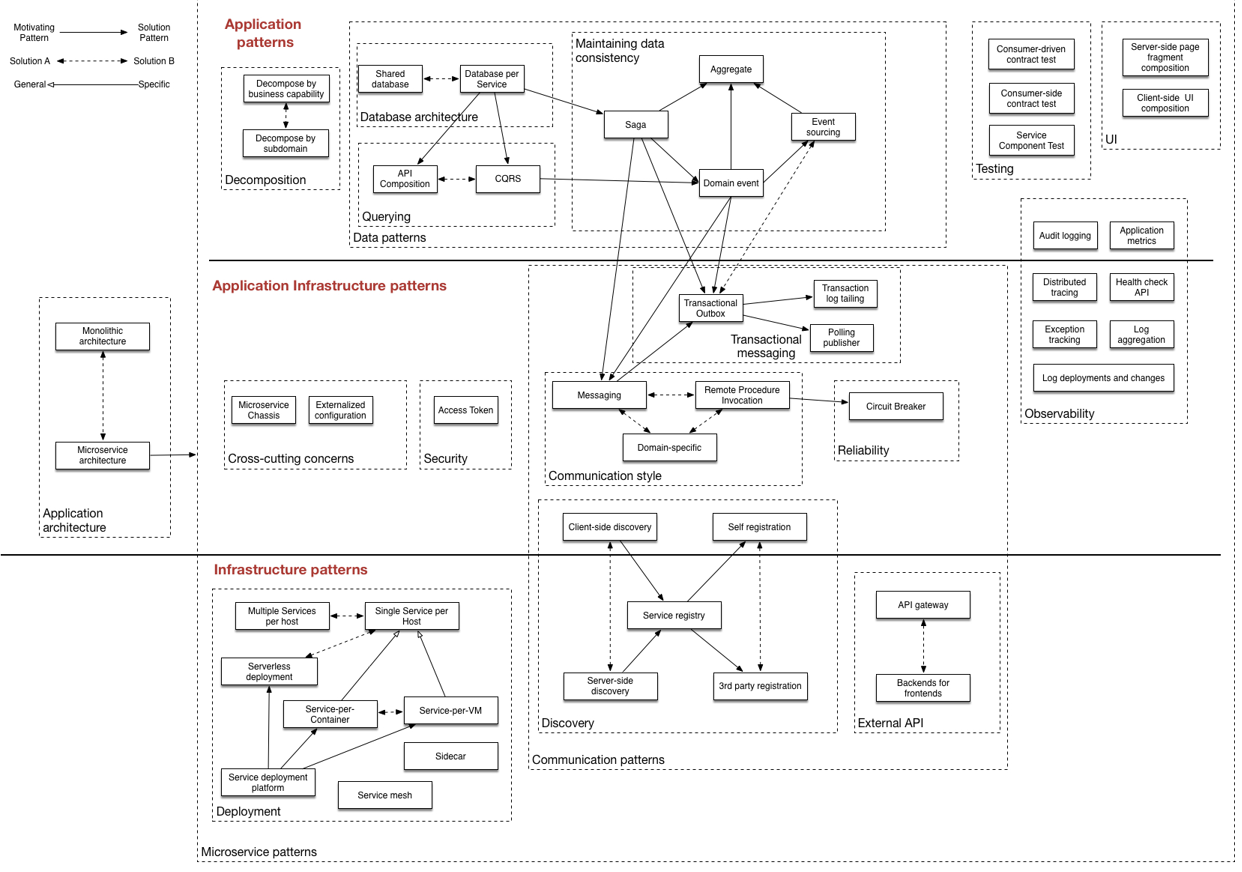
# Microservice overview

* **goal** microservice architecture is to accelerate software development by enabling continuous delivery/deployment
* **Single Responsibility Principle** – class should only have one reason to change
* **Common Closure Principle** - classes that change for the same reason should be in the same package.

<https://microservices.io/patterns/>



**Application architecture patterns**

* [Monolithic architecture](https://microservices.io/patterns/monolithic.html)
* [Microservice architecture](https://microservices.io/patterns/microservices.html)

**Decomposition**

* [Decompose by business capability](https://microservices.io/patterns/decomposition/decompose-by-business-capability.html)
* [Decompose by subdomain](https://microservices.io/patterns/decomposition/decompose-by-subdomain.html) – classified as {**Core, Supporting, Generic**}
  + EX: online store domain: Product catalog, Inventory, Order and Delivery management

**Deployment patterns**

* [Multiple service instances per host](https://microservices.io/patterns/deployment/multiple-services-per-host.html)
  + benefits
    - more efficient resource utilization compared to Single Service per Host
  + drawbacks
    - opposite of per host
  + lack of isolation, resource conflict, harder to monitor, manage and redeploy
* [Service instance per host](https://microservices.io/patterns/deployment/single-service-per-host.html)
  + benefits
    - Services instances are isolated from one another
    - There is no possibility of conflicting resource requirements or dependency versions
    - A service instance can only consume at most the resources of a single host
    - Its straightforward to monitor, manage, and redeploy each service instance
  + drawbacks
    - Potentially less efficient resource utilization compared to Multiple Services per Host
  + because more hosts
* [Service instance per VM](https://microservices.io/patterns/deployment/service-per-vm.html)
  + benefits
    - IaaS solutions such as AWS provide a mature and feature rich infrastructure
  + Straight forward scaling AWS scaling
    - VM encapsulates the details of the technology used to build the service
    - Each service instance is isolate
    - VM imposes limits on the CPU and memory consumed by a service instance
  + drawbacks
    - Building a VM image is slow and time consuming
* [Service instance per Container](https://microservices.io/patterns/deployment/service-per-container.html)
  + benefits
    - straightforward to scale up and down by changing number of container instances
    - container encapsulates details of technology to build the service.
  + All services are, for example, started and stopped in exactly the same way.
    - Each service instance is isolated
    - container imposes limits on CPU and memory consumed by a service instance
    - Containers are extremely fast to build and start
  + EX: 100x faster to package an application as a Docker container than it is to package it as an AMI
  + Docker containers start much faster than VM, only app process starts rather than entire OS.
  + drawbacks
    - infrastructure for deploying containers is not as rich as the infrastructure for Vm’s
* [Serverless deployment](https://microservices.io/patterns/deployment/serverless-deployment.html)
  + benefits
    - eliminates need to spend time on undifferentiated heavy lifting of managing low-level infrastructure
  + Instead, you can focus on your code.
    - The serverless deployment infrastructure is extremely elastic
  + automatically scales to handle load
    - pay for each request rather than provisioning what might be under utilized virtual machines or containers.
  + drawbacks
    - Significant limitation and constraints
  + lambdas can only respond to requests from a limited set of input sources
  + not intended to run services. EX: subscribe to message broker such as RabbitMQ
    - Applications must startup quickly
    - Risk of high latency
  + Can only react to increases in load. Can’t proactively pre-provision capacity
* [Service deployment platform](https://microservices.io/patterns/deployment/service-deployment-platform.html)
  + Examples
    - Docker orchestration frameworks including Docker swarm mode and Kubernetes
    - Serverless platforms such as AWS Lambda
    - PaaS including Cloud Foundry and AWS Elastic Beanstalk

**Cross cutting concerns**

* [Microservice chassis](https://microservices.io/patterns/microservice-chassis.html)
* [Externalized configuration](https://microservices.io/patterns/externalized-configuration.html)

**Communication style**

* [Remote Procedure Invocation](https://microservices.io/patterns/communication-style/rpi.html)
* [Messaging](https://microservices.io/patterns/communication-style/messaging.html)
* [Domain-specific protocol](https://microservices.io/patterns/communication-style/domain-specific.html)

**External API**

* [API gateway](https://microservices.io/patterns/apigateway.html)
* [Backend for front-end](https://microservices.io/patterns/apigateway.html)

**Transactional messaging**

* [Transactional outbox](https://microservices.io/patterns/data/application-events.html)
* [Transaction log tailing](https://microservices.io/patterns/data/transaction-log-tailing.html)
* [Polling publisher](https://microservices.io/patterns/data/polling-publisher.html)

**Service discovery**

* [Client-side discovery](https://microservices.io/patterns/client-side-discovery.html)
* [Server-side discovery](https://microservices.io/patterns/server-side-discovery.html)
* [Service registry](https://microservices.io/patterns/service-registry.html)
* [Self registration](https://microservices.io/patterns/self-registration.html)
* [3rd party registration](https://microservices.io/patterns/3rd-party-registration.html)

**Reliability**

* [Circuit Breaker](https://microservices.io/patterns/reliability/circuit-breaker.html)
  + benefits
    - Services handle failure of services they invoke
  + drawbacks
    - Challenging to choose timeout values without creating false positives or introducing excessive latency

**Data management**

* [Database per Service](https://microservices.io/patterns/data/database-per-service.html)
* [Shared database](https://microservices.io/patterns/data/shared-database.html)
* [Saga](https://microservices.io/patterns/data/saga.html) new

Given: applied the [Database per Service](https://microservices.io/patterns/data/database-per-service.html) pattern

Problem: How to maintain data consistency across services?

Forces: 2PC (Two phase commit) is not an option

Solution

1. Implement each business transaction that spans multiple services as a saga
2. A saga is a sequence of local transactions.
3. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga.
4. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

Example

1. The Order Service creates an Order in a *pending* state and **publishes** an OrderCreated event
2. The Customer Service receives the event attempts to reserve credit for that Order.
   1. It **publishes** either a Credit Reserved event or a CreditLimitExceeded event.
3. The Order Service receives the event and changes order state to either *approved* or *cancelled*

Pros

1. Enables data consistency across multiple services without using distributed transactions

Cons

1. programming model is more complex
2. must design compensating transactions that explicitly undo changes made earlier in a saga

* [API Composition](https://microservices.io/patterns/data/api-composition.html) new
* [CQRS](https://microservices.io/patterns/data/cqrs.html)

**Pattern: CQRS - Command Query Responsibility Segregation**

<https://microservices.io/patterns/data/cqrs.html>

Given: your applied the [Microservices architecture pattern](https://microservices.io/patterns/microservices.html) and the [Database per service pattern](https://microservices.io/patterns/data/database-per-service.html)

result, it is no longer straightforward to implement queries that join data from multiple services

if applied the [Event sourcing pattern](https://microservices.io/patterns/data/event-sourcing.html) then the data is no longer easily queried

Problem: How to implement a query that retrieves data from multiple services in a microservice architecture

Forces:

Solution

* 1. Define a view database, which is a read-only replica that is designed to support that query
  2. app keeps replica up to data by subscribing to Domain events published by service that own the data

Example

Pros

1. Supports multiple denormalized views that are scalable and performant
2. Improved separation of concerns = simpler command and query models
3. an event sourced architecture is required

Cons

1. Increased complexity
2. Potential code duplication
3. Replication lag/eventually consistent views

Related patterns

* The [Database per Service pattern](https://microservices.io/patterns/data/database-per-service.html) creates the need for this pattern
* The [API Composition pattern](https://microservices.io/patterns/data/api-composition.html) is an alternative solution
* The [Domain event](https://microservices.io/patterns/data/domain-event.html) pattern generates the events
* CQRS is often used with [Event sourcing](https://microservices.io/patterns/data/event-sourcing.html)
* [Event sourcing](https://microservices.io/patterns/data/event-sourcing.html)
* [Application events](https://microservices.io/patterns/data/application-events.html)

**Security**

* [Access Token](https://microservices.io/patterns/security/access-token.html)

**Testing**

* [Service Component Test](https://microservices.io/patterns/testing/service-component-test.html)
* [Consumer-driven contract test](https://microservices.io/patterns/testing/service-integration-contract-test.html)
* [Consumer-side contract test](https://microservices.io/patterns/testing/consumer-side-contract-test.html)

**Observability**

* [Log aggregation](https://microservices.io/patterns/observability/application-logging.html)
* [Application metrics](https://microservices.io/patterns/observability/application-metrics.html)
* [Audit logging](https://microservices.io/patterns/observability/audit-logging.html)
* [Distributed tracing](https://microservices.io/patterns/observability/distributed-tracing.html)
* [Exception tracking](https://microservices.io/patterns/observability/exception-tracking.html)
* [Health check API](https://microservices.io/patterns/observability/health-check-api.html)
* [Log deployments and changes](https://microservices.io/patterns/observability/log-deployments-and-changes.html)new

**UI patterns**

* [Server-side page fragment composition](https://microservices.io/patterns/ui/server-side-page-fragment-composition.html)
* [Client-side UI composition](https://microservices.io/patterns/ui/client-side-ui-composition.html)

# Rabbit MQ

* <https://www.rabbitmq.com>
* Asynchronous Messaging
  + Supports **AMQP** Advanced Message Queuing Protocol
  + Supports multiple messaging protocols
  + message queuing
  + delivery acknowledgement
  + flexible routing to queues, multiple exchange type
    - Direct exchange (Empty string) amq.direct
      * + If name is empty, the default queue is used
    - Fanout exchange amq.fanout
      * + routes messages to all bound queues
    - Topic exchange amq.topic
      * + messages routed to one or many queues
        + based on matching message routing key and the pattern
    - Headers exchange amq.match (amq.headers in RabbitMQ)
      * + routing on multiple attributes expressed as message headers rather than a routing key
        + Headers exchanges ignore routing key attribute
* Enterprise & Cloud Ready
  + Pluggable authentication, authorization, supports TLS and LDAP
  + Lightweight and easy to deploy in public and private clouds

# JWT token

* JWT token authentication (JSON Web Token)
  + Overview
    - An open standard (RFC 7519) that defines compact, self-contained way
      * for securely transmitting information between parties as a JSON object
    - info can be verified and trusted because it is digitally signed
    - **structure**: Header, Payload, Signature
* Access Token
* Refresh Token
* Black Listed Token.