### MICRO SERVICES

**Monolithic Application**

**All the functionalities are implemented and enabled as part of the single application.**

Disadvantage

1**. Difficult to deploy big application.**

**Even small changes, we need to test entire application.**

**2. Scalability - Deployed as single application. When scale up, we need to do entire application.**

**When traffic spikes up, number of application instance needs to be increased.**

**When traffic comes to normal, number of application instance needs to be decreased.**

**Service Oriented Architecture (SOA)**

SOA is essentially a collection of services. These services communicate with each other. The communication can involve either simple data processing or could involve two or more services co-ordinating some activity.

It uses Enterprise Service Bus for communication.

Maximizes application reusability

**Microservice**

Rather than full application, why don't you split the application into mini application. Each mini application called as micro services.

One large application divided into multiple collaborating process.

Each Microservices are communicated through REST API's.

Advantages

1. Deployment flexibility

2. Technology flexibility

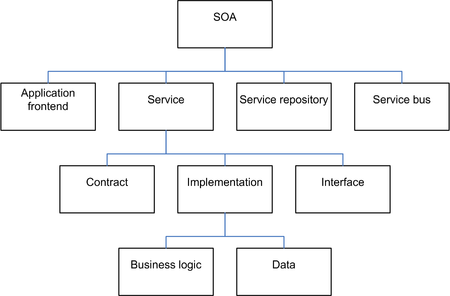
3. can be scaled separately

Disadvantages

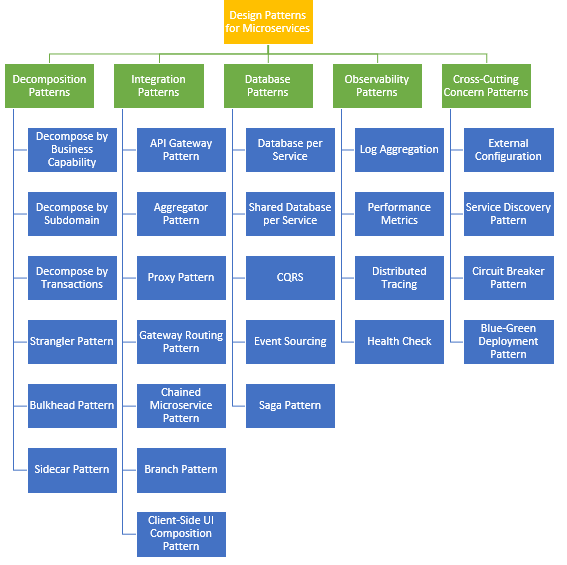
- Deployment/ architecture complexity

- Service discovery. Which endpoints needs to be called.

Microservice architecture solves the above problem. Microservice Architecture is divided into set of applications called Microservice.



The crux of the microservices pattern is to create an independent service which can be scaled and deployed independently.



# Decomposition Patterns

**Decompose by Business Capability**

[Decompose by business capability](https://microservices.io/patterns/decomposition/decompose-by-business-capability.html)  define services corresponding to business capabilities.

The microservice architecture structures an application as a set of loosely coupled services. The goal of the microservice architecture is to accelerate software development by enabling continuous delivery/deployment.

A service must be small enough to be developed by a small team and to be easily tested. It make sense to apply the SRP to service design as well and design services that are cohesive and implement a small set of strongly related functions.

How to decompose an application into services?

**Solution**

Define services corresponding to business capabilities. A business capability is a concept from business architecture modelling. It is something that a business does in order to generate value. A business capability often corresponds to a business object, e.g.

* *Order Management* is responsible for orders
* *Customer Management* is responsible for customers

**Decompose by sub domain**

Decomposing an application using business capabilities might be a good start, but you will come across so-called “God Classes” which will not be easy to decompose. These classes will be common among multiple services.

How to decompose an application into services?

**Solution**

Define services corresponding to Domain-Driven Design (DDD) subdomains.

DDD refers to the application’s problem space - the business - as the domain.

A domain consists of multiple subdomains. Each subdomain corresponds to a different part of the business.

Subdomains can be classified as follows:

* Core - key differentiator for the business and the most valuable part of the application
* Supporting - related to what the business does but not a differentiator. These can be implemented in-house or outsourced.
* Generic - not specific to the business and are ideally implemented using off the shelf software

The subdomains of an Order management include:

Product catalog service, Inventory management services, Order management services Delivery management services.

The corresponding microservice architecture would have services corresponding to each of these subdomains.

**Decompose by Transactions / Two-phase commit (2pc) pattern**

This pattern can decompose services over the transactions. Then there will be multiple transactions in the system. One of the important participants in a distributed transaction is the transaction coordinator. The distributed transaction consists of two steps:

* **Prepare phase** — during this phase, all participants of the transaction prepare for commit and notify the coordinator that they are ready to complete the transaction
* **Commit or Rollback phase**— during this phase, either a commit or a rollback command is issued by the transaction coordinator to all participants

The problem with 2PC is that it is quite slow compared to the time for operation of a single microservice. Coordinating the transaction between microservices, even if they are on the same network, can really slow the system down; so, this approach isn’t usually used in a high load scenario.

**Strangler Pattern - Refactoring to microservicesnew**

## **Problem**

How do you migrate a legacy monolithic application to a microservice architecture?

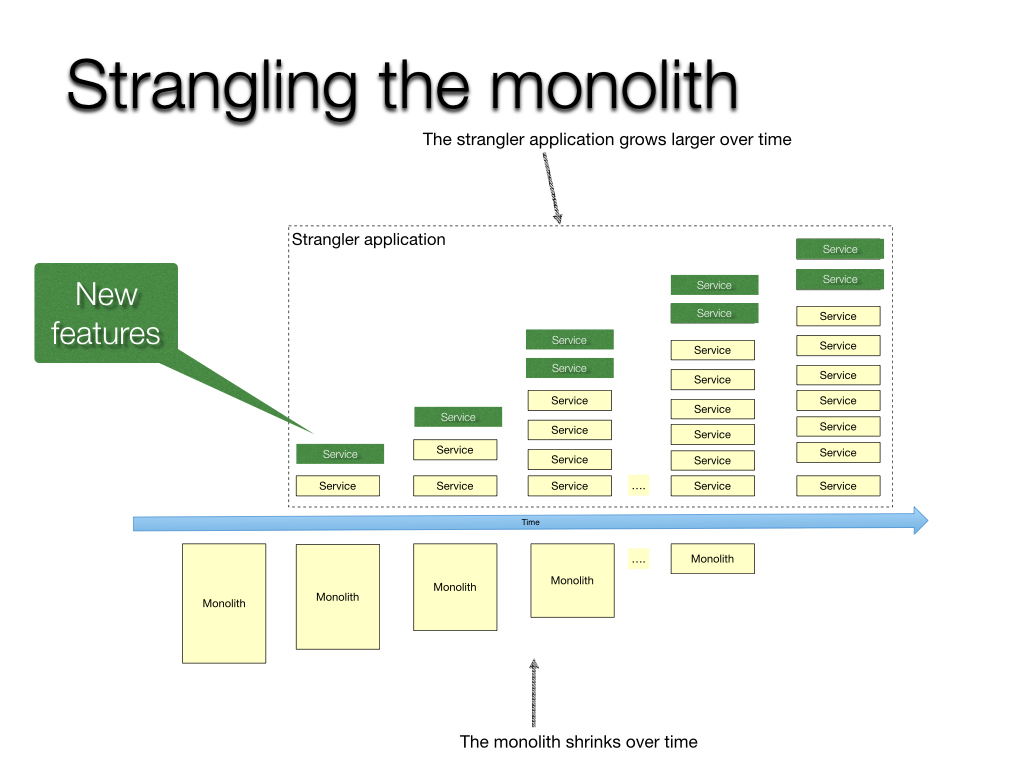
## **Solution**

Modernize an application by incrementally developing a new (strangler) application around the legacy application. In this scenario, the strangler application has a [microservice architecture](https://microservices.io/patterns/microservices.html).

The strangler application consists of two types of services.

First, there are services that implement functionality that previously resided in the monolith.

Second, there are services that implement new features. The latter are particularly useful since they demonstrate to the business the value of using microservices.



This creates two separate applications that live side by side in the same URI space. Over time, the newly refactored application “strangles” or replaces the original application until finally, you can shut off the monolithic application. The Strangler Application steps are transform, coexist, and eliminate:

* Transform — Create a parallel new site with modern approaches.
* Coexist — Leave the existing site where it is for a time. Redirect from the existing site to the new one so the functionality is implemented incrementally.
* Eliminate — Remove the old functionality from the existing site.

## **Bulkhead Pattern**

This pattern isolates elements of an application into pools so that if one fails, the others will continue to function. This pattern is named Bulkhead because it resembles the sectioned partitions of a ship’s hull.

Partition service instances into different groups, based on consumer load and availability requirements.

This design helps to isolate failures and allows you to sustain service functionality for some consumers, even during a failure.

## **Sidecar Pattern**

This deploys components of an application into a separate processor container to provide isolation and encapsulation.

This pattern can also enable applications to be composed of heterogeneous components and technologies.

This pattern is named Sidecar because it resembles a sidecar attached to a motorcycle.

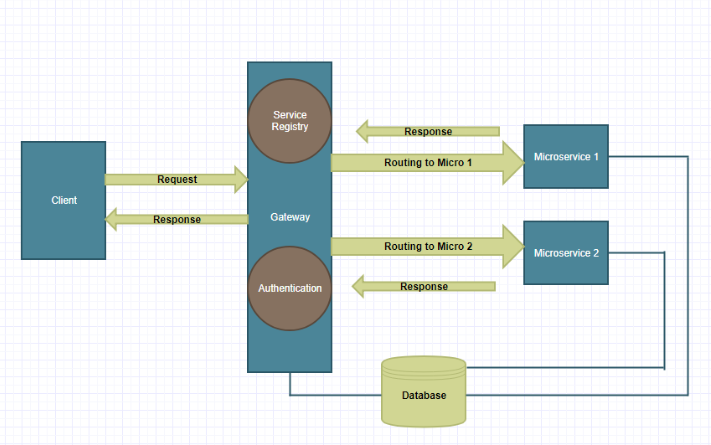
In the pattern, the sidecar is attached to a parent application and provides supporting features for the application. The sidecar also shares the same lifecycle as the parent application, is created and retired alongside the parent.

The sidecar pattern is sometimes referred to as the sidekick pattern

# Cross Cutting Concerns Patterns

**API gate way**

* The [API Gateway pattern](https://microservices.io/patterns/apigateway.html) defines how clients access the services in a microservice architecture.
* API gateway is single entry point for all clients.
* The API gateway handles requests in one of two ways.
  + Some requests are simply proxied/routed to the appropriate service.
  + It handles other requests by fanning out (spread) to multiple services.
* The API gateway might also implement security, e.g. verify that the client is authorized to perform the request.
* The API Gateway is a server. It is a single-entry point into a system. API Gateway encapsulates the internal system architecture. It provides an API that is tailored to each client. It also has other responsibilities such as **authentication, monitoring, load balancing, caching, request shaping and management,**and **static response handling**.
* All the requests made by the client go through the API Gateway. After that, the API Gateway routes requests to the appropriate microservice.
* The API Gateway can provide each client with a custom API. It also translates between two protocols, such as **HTTP,** **WebSocket,** and **Web-Unfriendly** protocols that are used internally.



## **Working of API Gateway**

In microservices, we route all the requests through an API. We can implement common features like **authentication, routing, caching, versioning, auditing,**and **logging** in the API Gateway.

## **Advantages of API Gateway**

* + The most important advantage of API Gateway is that it encapsulates the internal structure of the application.
  + Rather than invoking the specific service, the client directly talks to the API Gateway.
  + It reduces the number of round trips between client and application.
  + It simplifies the client code.
  + It reduces coding efforts, makes the application more efficient, decreases errors all at the same time.
  + It provides each kind of client with a specific API.

## **Disadvantages**

* + It requires routing rules.
  + There is a possibility of a single point of failure.
  + Risk of complexity due to all the API rules are in one place.

**Aggregator Microservice Design Pattern:**

It talks about how we can aggregate the data from different services and then send the final response to the consumer. This can be done in two ways:

1. A **composite microservice** will make calls to all the required microservices, consolidate the data, and transform the data before sending back.

2. An **API Gateway** can also partition the request to multiple microservices and aggregate the data before sending it to the consumer.

It is recommended if any business logic is to be applied, then choose a composite microservice. Otherwise, the API Gateway is the established solution.

Aggregator would be a simple web page that invokes multiple services to achieve the functionality required by the application.

In the Aggregator pattern, a service will invoke multiple microservices to get the data from multiple microservices. Microservice can aggregate the data collected and apply any processing if required. Each microservice can have individual caching & database, in the same way, an aggregator can have it’s own caching to improve the efficiency.

The user makes a single call to the Aggregator, and the aggregator then calls each relevant microservice and collects the data, apply business logic to it, and further publish is as a REST Endpoint. More variations of the aggregator are:

* Proxy Microservice Design Pattern: A different microservice is called upon the business need.
* Chained Microservice Design Pattern: In this case each microservice is dependent/ chained to a series of other microservices.

<https://github.com/iluwatar/java-design-patterns/tree/master/aggregator-microservices>

**Gateway Routing Pattern**

* The API gateway is responsible for request routing.
* An API gateway implements some API operations by routing requests to the corresponding service.
* When it receives a request, the API gateway consults a routing map that specifies which service to route the request to.
* A routing map might, for example, map an HTTP method and path to the HTTP URL of service.

**Chained Microservice Pattern**

* There will be multiple dependencies of for single services or microservice

e.g.: Sale microservice has dependency products microservice and order microservice.

* Chained microservice design pattern will help you to provide the consolidated outcome to your request. The request received by a microservice-1, which is then communicating with microservice-2 and it may be communicating with microservice-3. All these services are synchronous calls.

**Branch Pattern**

* A microservice may need to get the data from multiple sources including other microservices.
* Branch microservice pattern is a mix of Aggregator & Chain design patterns and allows simultaneous request/response processing from two or more microservices. The invoked microservice can be chains of microservices.
* Brach pattern can also be used to invoke different chains of microservices, or a single chain, based your business needs.

# Cross Cutting Concerns Patterns

**External Configuration**

**Problem**

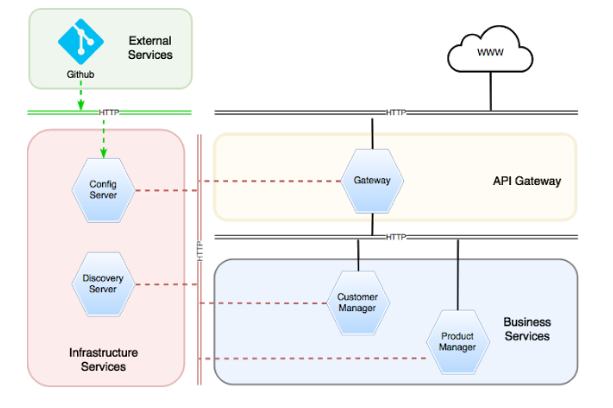
A service typically calls other services and databases as well. For each environment like dev, QA, UAT, prod, the endpoint URL or some configuration properties might be different. A change in any of those properties might require a re-build and re-deploy of the service.

**Solution**

To avoid code modification configuration can be used. Externalize all the configuration, including endpoint URLs and credentials. The application should load them either at start-up or on the fly.

Spring Cloud config server provides the option to externalize the properties to GitHub and load them as environment properties. These can be accessed by the application on start up or can be refreshed without a server restart.

Config Server is now the central part of our infrastructure that is supposed to both store and serve configuration for all other microservices. Thus, you should always scale it properly, so that your system stays resilient.



<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.config.server.EnableConfigServer;

@EnableConfigServer

@SpringBootApplication

public class SpringConfigServerApplication {

public static void main(String[] args) {

SpringApplication.run(SpringConfigServerApplication.class, args);

}

}

**Server Configuration**

src/main/resources/bootstrap.properties

#Server port

server.port = 8888

#Git repo location.

spring.cloud.config.server.git.uri=E:\\devsetup\\gitworkspace\\spring-cloud\\config-git-repo

spring.cloud.config.server.git.cloneOnStart=true

**Client Configuration**

src/main/resources/bootstrap.properties

spring.application.name=client-config

#Active Profile - will relate to development properties file in the server.

#If this property is absent then,default profile will be activated which is

#the property file without any environment name at the end.

spring.profiles.active=development

# N.B. this is the default:

spring.cloud.config.uri=http://localhost:8888

management.security.enabled=false

<https://dzone.com/articles/dynamic-configuration-management-in-microservice-a>

**Service Discovery Pattern**

**Problem**

When microservices come into the picture, we need to address a few issues in terms of calling services. With container technology, IP addresses are dynamically allocated to the service instances. Every time the address changes, a consumer service can break and need manual changes. Each service URL has to be remembered by the consumer and become tightly coupled.

**Solution**

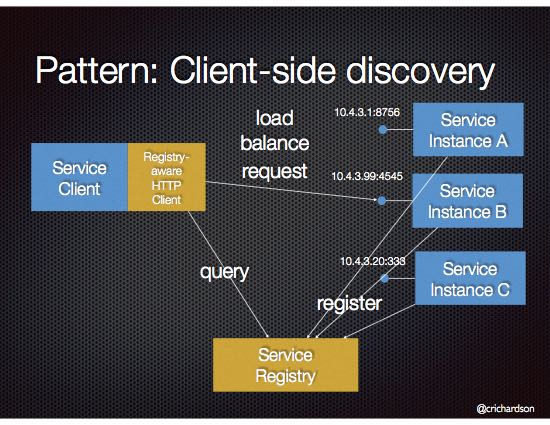
A service registry needs to be created which will keep the metadata of each producer service and specification for each. A service instance should register to the registry when starting and should de-register when shutting down.

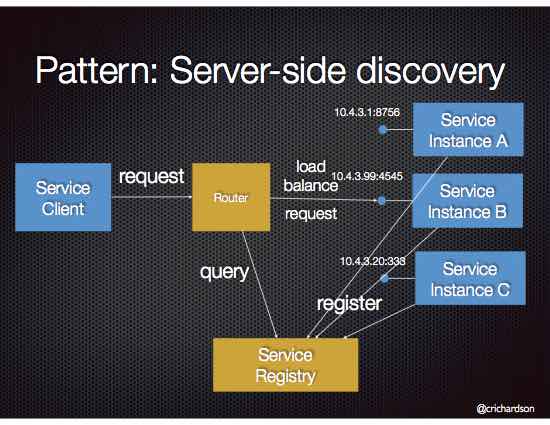
The registry also needs to do a health check of the producer service to ensure that only working instances of the services are available to be consumed through it.

There are two types of service discovery:

**client-side : eg: Netflix Eureka.**

**Server-side : eg: AWS ALB.**





**Circuit Breaker design pattern**

The Circuit Breaker design pattern is used to stop the process of request and response if a service is not working.

**Problem**:

A service generally calls other services to retrieve data, there is always the possibility that the other service is unavailable or may be down.

There are two problem with this.

1. First, the request will keep going to the down service, exhausting network resources, and slowing performance.

2. Second, the user experience will be bad and unpredictable.

**Solution**:

The consumer should invoke a remote service via a proxy that behaves in a similar fashion to an electrical circuit breaker.

When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period, all attempts to invoke the remote service will fail immediately.

After the timeout expires the circuit breaker allows a limited number of test requests to pass through.

If those requests succeed the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

<https://medium.com/@madhukaudantha/microservice-architecture-and-design-patterns-for-microservices-e0e5013fd58a>

<https://microservices.io/patterns/reliability/circuit-breaker.html>

**Blue-Green Deployment Pattern**

**Problem**

With microservice architecture, one application can have many microservices. If we stop all the services then deploy an enhanced version, the downtime will be huge and can impact the business. Also, the rollback will be a nightmare. How do we avoid or reduce downtime of the services during deployment?

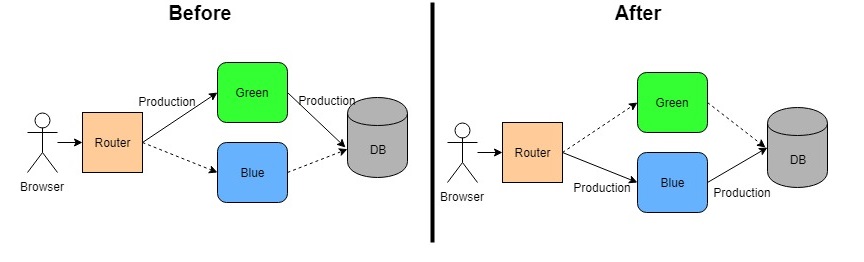
**Solution**

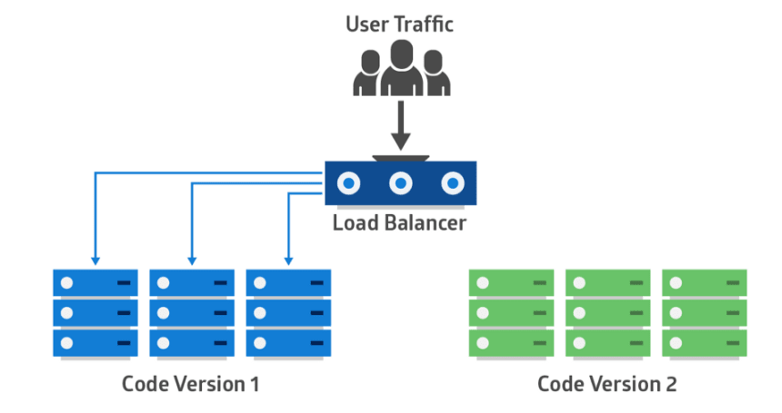
The blue-green deployment strategy can be implemented to reduce or remove downtime. It achieves this by running two identical production environments, Blue and Green. Let's assume Green is the existing live instance and Blue is the new version of the application. At any time, only one of the environments is live, with the live environment serving all production traffic.

All cloud platforms provide options for implementing a blue-green deployment.

**What Is Blue-Green Deployment?**

Blue-green deployment is a technique that enables continuous delivery to production with reduced downtime and risk. It achieves this by running two identical production environments called Blue and Green. Let's assume, Green is the existing live instance and Blue is the new version of the application. At any time, only one of the environments is live, with the live environment serving all production traffic.





**Benefits**

* It helps to reduce the downtime and even reduces it to zero depending on the application design and deployment approach.
* It gives a rapid way of rollback of the application in case of production issue.
* It helps to build confidence to business users as testing of new version can be done in Production in isolation before rollout.

# Observability Patterns

**Log aggregation**

**Problem**

The application consists of multiple services and service instances that are running on multiple machines. Requests often span multiple service instances.

Each service instance generates writes information about what it is doing to a log file in a standardized format. The log file contains errors, warnings, information and debug messages.

**Solution**

Use a centralized logging service that aggregates logs from each service instance. The users can search and analyse the logs. They can configure alerts that are triggered when certain messages appear in the logs.

**Performance Metrics**

Metrics is measurements that provide insight into what the application is doing and how it is performing

**Problem**

When the service portfolio increases due to microservice architecture, it becomes critical to keep a watch on the transactions so that patterns can be monitored, and alerts sent when an issue happens.

How should we collect metrics to monitor application performance?

**Solution**

A metrics service is required to gather statistics about individual operations. It should aggregate the metrics of an application service, which provides reporting and alerting. There are two models for aggregating metrics:

Push — the service pushes metrics to the metrics service e.g. NewRelic, AppDynamics

Pull — the metrics services pulls metrics from the service e.g. Prometheus

**Distributed Tracing**

**Problem**

In a microservice architecture, requests often span multiple services. Each service handles a request by performing one or more operations across multiple services e.g. database queries, publishes messages, etc. Then, how do we trace a request end-to-end to troubleshoot the problem?

While in troubleshoot it is worth to have trace ID, we trace a request end-to-end.

**Solution**

The solution is to introduce a transaction ID.

We need a service which

- each external request a unique external request id.

- Passes the external request id to all services.

- Includes the external request id in all log messages.

- Records information (e.g. start time, end time) about the requests and operations performed when handling an external request in a centralized service

**Spring Cloud Slueth, along with Zipkin server, is a common implementation**.

**Health Check**

**Problem**

Sometimes a service instance can be incapable of handling requests yet still be running. For example, it might have ran out of database connections. When this occurs, the monitoring system should generate a alert. Also, the load balancer or service registry should not route requests to the failed service instance.

How to detect that a running service instance is unable to handle requests?

**Solution**

A service has an health check API endpoint (e.g. HTTP /health) that returns the health of the service. The API endpoint handler performs various checks, such as

the status of the connections to the infrastructure services used by the service instance

the status of the host, e.g. disk space

application specific logic

A health check client - a monitoring service, service registry or load balancer - periodically invokes the endpoint to check the health of the service instance.

**Spring Boot Actuator does implement a /health endpoint and the implementation can be customized, as well**.

To enable a /health endpoint, we need to add the dependency of **spring-boot-starter-actuator**. It provide various capabilities including a health check endpoint.

# Database Patterns

Defining the database architecture for microservices we need to consider the below points.

* Services must be loosely coupled. They can be developed, deployed, and scaled independently.
* Business transactions may enforce invariants that span multiple services.
* Some business transactions need to query data that is owned by multiple services.
* Databases must sometimes be replicated and shared in order to scale.
* Different services have different data storage requirements.

## **Database per Service**

To solve the above concerns, one database per microservice must be designed; it must be private to that service only. It should be accessed by the microservice API only. It cannot be accessed by other services directly.

For example, for relational databases, we can use private-tables-per-service, schema-per-service, or database-server-per-service.

**Command Query Responsibility Segregation (CQRS)**

**Problem**

Once we implement database-per-service, there is a requirement to query, which requires joint data from multiple services — it's not possible.

Then, how do we implement queries in microservice architecture?

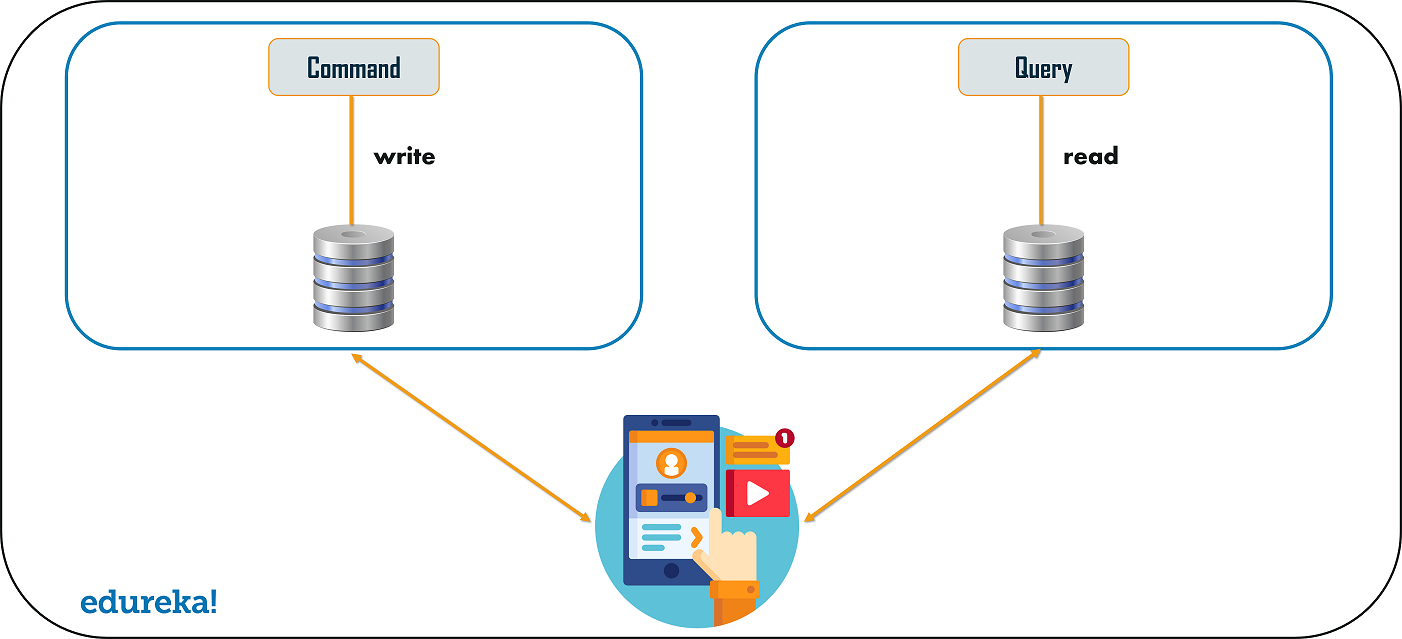
**Solution**

CQRS suggests splitting the application into two parts — the command side and the query side.

The **command side** handles the Create, Update, and Delete requests.

The **query side** handles the query part by using the materialized views.

The event sourcing pattern is generally used along with it to create events for any data change. Materialized views are kept updated by subscribing to the stream of events.



**Event Sourcing**

The event sourcing design pattern creates events regarding the changes in the application state.

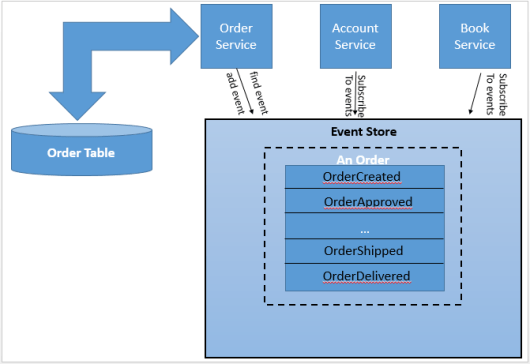
Also, these events are stored as a sequence of events to help the developers track which change was made when.

So, with the help of this, you can always adjust the application state to cope up with the past changes.

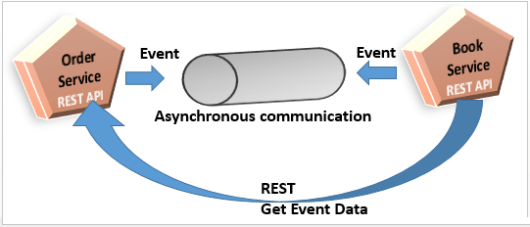
You can also query these events, for any data change and simultaneously publish these events from the event store.

Once the events are published, you can see the changes in the application state on the presentation layer.

The Event Sourcing pattern defines an approach to handling operations on data that’s driven by a sequence of events, each of which is recorded in an append-only store.



Event-based communication is very similar to messaging. Instead of sending messages, the service instead generates events. Take a look at the following diagram:



As you can see, the Order Service generates an event. This is a signal that something has happened, such as an order for a book being generated. The services that are interested in this type of event, order generated, send a call to the Order Service.

In event-based communication, there is no need for a particular message structure from the message broker. We can also use this approach with transactional messaging to avoid Two-Phase Commit.

The online store verifies the customer’s credit limit when creating an order using a sequence of steps:

* An Order aggregate, which is created with a NEW status, publishes an OrderCreated event
* The Customer aggregate consumes the OrderCreated event, reserves credit for the order and publishes an CreditReserved event
* The Order aggregate consumes the CreditReserved event, and changes its status to APPROVED
* If the credit check fails due to insufficient funds, the Customer aggregate publishes a CreditLimitExceeded event. This event does not correspond to a state change but instead represents a failed attempt to violate a business rule. The Order aggregate consumes this event and changes its state to CANCELLED.

**Saga Pattern**

The Saga Pattern is as microservices architectural pattern to implement a transaction that spans multiple services.

A saga is a sequence of local transactions.

**Problem**

Each service has its own database. Some business transactions, however, span multiple service so you need a mechanism to ensure data consistency across services.

For example, Since Orders and Customers are in different databases the application cannot simply use a local ACID transaction.

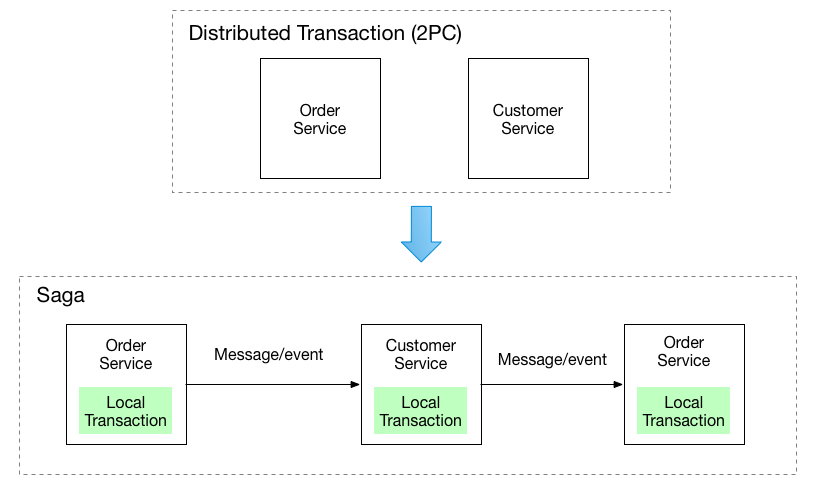
How to maintain data consistency across services?

**Solution**

Implement each business transaction that spans multiple services as a saga.

A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga.

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.



There are two ways of coordination sagas:

**Events/Choreography** - each local transaction publishes domain events that trigger local transactions in other services.

When there is no central coordination, each service produces and listens to the other service's events and decides if an action should be taken or not.

In this approach, there is no central orchestrator. Each service participating in the Saga performs their transaction and publish events. The other services act upon those events and perform their transactions. Also, they may or not publish other events based on the situation.

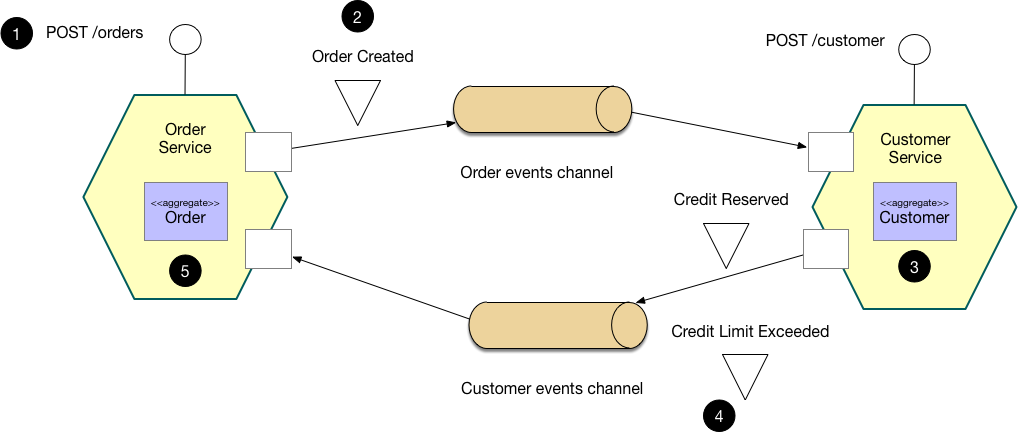
**Command/Orchestration** - an orchestrator (object) tells the participants what local transactions to execute.

When a coordinator service is responsible for centralizing the saga's decision making and sequencing business logic.

In this approach, there is a Saga orchestrator that manages all the transactions and directs the participant services to execute local transactions based on events. This orchestrator can also be though of as a Saga Manager.

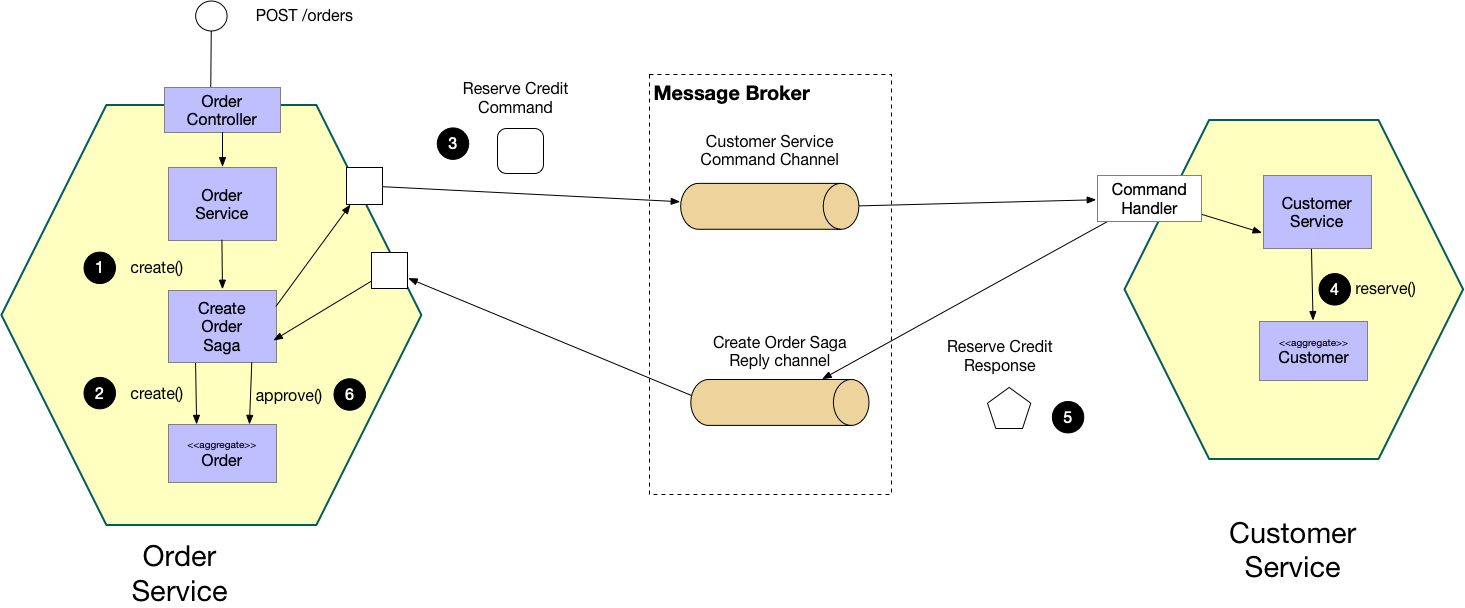
In the orchestration approach, we define a new service with the sole responsibility of telling each participant what to do and when. The saga orchestrator communicates with each service in a command/reply style telling them what operation should be performed.

**Example: Choreography-based saga**



|  |  |
| --- | --- |
| **Choreography based SAGA**   1. The Order Service receives the POST /orders request and creates an Order in a PENDING state 2. It then emits an Order Created event 3. The Customer Service’s event handler attempts to reserve credit 4. It then emits an event indicating the outcome 5. The OrderService’s event handler either approves or rejects the Order | **Choreography based SAGA**   1. The Order Service receives the POST /orders request and creates the Create Order saga orchestrator 2. The saga orchestrator creates an Order in the PENDING state 3. It then sends a Reserve Credit command to the Customer Service 4. The Customer Service attempts to reserve credit 5. It then sends back a reply message indicating the outcome 6. The saga orchestrator either approves or rejects the Order |

## **Example: Orchestration-based saga**



This pattern has the following benefits:

* It enables an application to maintain data consistency across multiple services without using distributed transactions

This solution has the following drawbacks:

* The programming model is more complex. For example, a developer must design compensating transactions that explicitly undo changes made earlier in a saga.

Transactions are an essential part of applications. Without them, it would be impossible to maintain data consistency.

One of the most powerful types of transactions is called a Two-Phase Commit, which is in summary when the commit of a first transactions depends on the completion of a second.

**Implementation**: How easy is to implement proposed solution?

**Testability**: How easy is it to test the solution?

**Complexity**: How many technologies are needed to build the solution?

**Consistency**: Is the query side immediately consistent after a successful command? Or is it or eventually consistent?

**Portability**: Is it easy to port a solution to a different vendor? For instance, some of the solutions may rely on specific databases. That creates vendor lock-in.

**Scalability**: Is it easy to horizontally scale the one of the sides, independent of the another?

**Distribution**: Is the solution distributed? Are the use-cases handled by different processes?

**Extensibility**: Is it easy to add new model?