Microservices Design Patterns - Quick Guide

Microservices Design Patterns - Overview

Microservice is a service-based application development methodology. In this methodology, big applications will be divided into smallest independent service units. Microservice is the process of implementing Service-oriented Architecture (SOA) by dividing the entire application as a collection of interconnected services, where each service will serve only one business need.

The Concept of Going Micro

In a service-oriented architecture, entire software packages will be sub-divided into small, interconnected business units. Each of these small business units will communicate to each other using different protocols to deliver successful business to the client. Now the question is, how Microservice Architecture (MSA) differs from SOA? In one word, SOA is a designing pattern and Microservice is an implementation methodology to implement SOA or we can say Microservice is a type of SOA.

Following are some rules that we need to keep in mind while developing a Microservice-oriented application.

- **Independent** − Each microservice should be independently deployable.
- **Coupling** − All microservices should be loosely coupled with one another such that changes in one will not affect the other.
- **Business Goal** − Each service unit of the entire application should be the smallest and capable of delivering one specific business goal.

To apply these principles, there are certain challenges and issues which must be handled. Microservices Design Patterns discusses those common problems and provides solutions to them. In coming sections, we'll discuss the problems and the solution using the applicable design pattern.

Design Patterns relevant for Microservices are grouped into five major categories.

- **Decomposition Design Patterns** − A application is to be decomposed in smaller microservices. Decomposition design patterns provides insight on how to do it logically.
- **Integration Design Patterns** − Integration design patterns handles the application behavior in entirety. For example, how to get result of multiple services result in single call etc.

- **Database Design Patterns** − Database design patterns deals with how to define database architecture for microservices like each service should have a seperate database per service or use a shared database and so.
- **Observability Design Patterns** − Observability design patterns considers tracking of logging, performance metrices and so.
- **Cross Cutting Concern Design Patterns** − Cross Cutting Concern Design Patterns deals with service discovery, external configurations, deployment scenarios etc.

Decompose By Business Capability

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service should be developed independently in agile manner to enable continous delivery/deployment. When a large, complex application is to be built using microservice architecture, the major problem is how to design loosely coupled microservices or to break a large application into small loosely coupled services?

Solution

We can define a microservice corresponding to a particular business capability. A business capability refers to the business activity targetted to generate value. A business capability can be referred as business object. For Example −

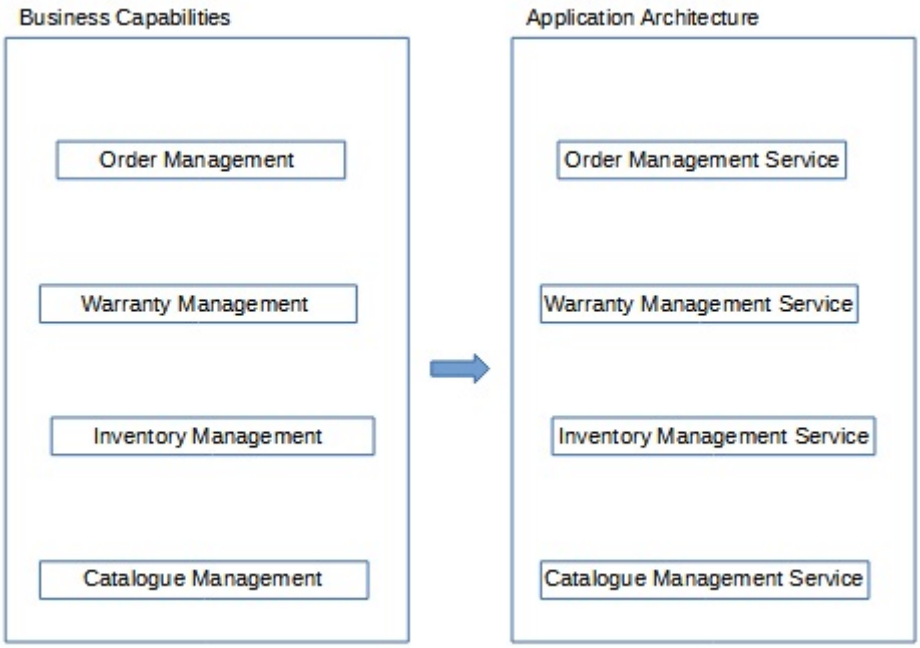
- **Order Management** − Order Management Business Capability refers to Orders.
- **Customer Management** − Customer Management Business Capability refers to Customers.

Business Capabilities can further be categorized into multi-level hiearchical structure. For example, Order Mangement can have delivery, inventory, service etc. as business capabilities.

Example

Consider an example of an Online Book Store. It can have following business capabilities and corresponding microservices −

- Books Catalog Management
- Inventory Management
- Order Management
- Warranty Management



Advantages

- **Stable Architecture** − As business capabilities are stable, this architecture is highly stable.
- **Cross-functional Teams** − Development Teams works independently, are cross-functional and are organized around functional features instead of technical features.
- **Loosely Coupled Services** − Developed services will be loosely coupled and cohesive.

Dis-advantages

- **Need good understand of Business** − Business capabilities needs be indentified after understanding the business. Understanding organizational structure can help as organizations are structured based on their capabilities.
- **High Level Domain Model needed** − Business domain objects required as they corresponds to business capabilities.

Decompose By Subdomain

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service should be developed independently in agile manner to enable continous delivery/deployment. When a large, complex application is to be built using microservice architecture, the major problem is how to design loosely coupled microservices or to break a large application into small loosely coupled services?

Solution

We can define a microservice corresponding to Domain-Driven Design(DDD) subdomains. DDD refers to business as a domain and a domain can have multiple subdomains. Now each subdomain refers to different areas. For Example −

- **Order Management** − Order Management subdomain refers to Orders.
- **Customer Management** − Customer Management subdomain refers to Customers.

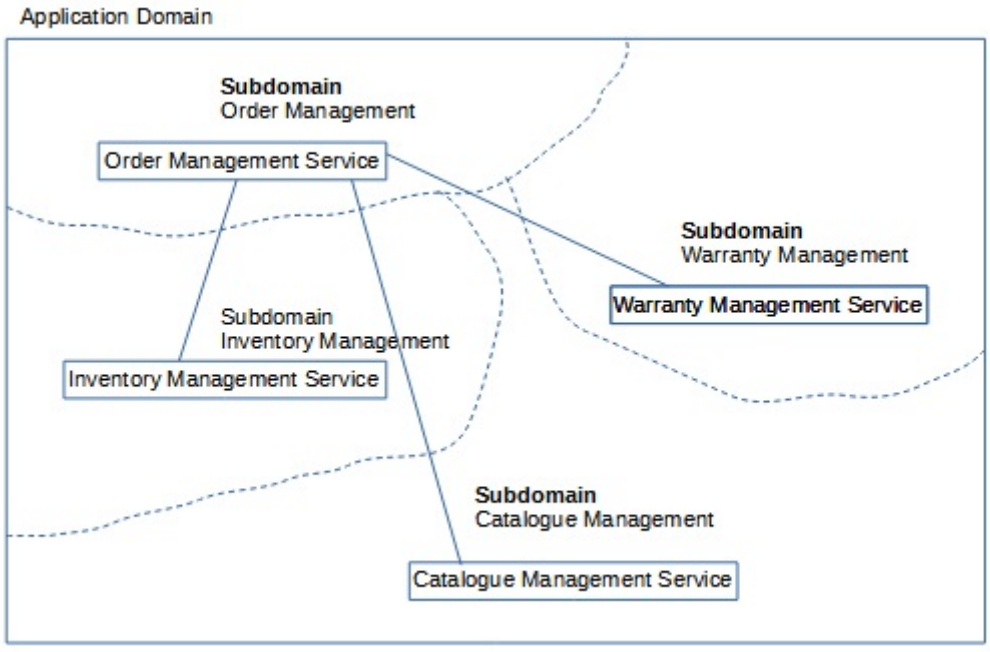
Subdomains can be further classified using following criterias −

- **Core** − Most important and key differentiator of an application.
- **Supporting** − Business related and are used to support the business activities.
- **Generic** − Not specific to business but are used to enhance the business operations.

Example

Consider an example of an Online Book Store. It can have following subdomains and corresponding microservices −

- Books Catalog Management
- Inventory Management
- Order Management
- Warranty Management



Advantages

- **Stable Architecture** − As business subdomains are stable, this architecture is highly stable.

- **Cross-functional Teams** − Development Teams works independently, are cross-functional and are organized around functional features instead of technical features.
- **Loosely Coupled Services** − Developed services will be loosely coupled and cohesive.

Dis-advantages

- **Need good understand of Business** − Business subdomains needs be indentified after understanding the business. Understanding organizational structure can help as organizations are structured based on their capabilities.
- **High Level Domain Model needed** − Business domain objects required as they corresponds to business subdomains.

Decompose By Strangler

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service should be developed independently in agile manner to enable continous delivery/deployment. When a large, complex application is to be built using microservice architecture, the major problem is how to design loosely coupled microservices or to break a large application into small loosely coupled services?

Solution

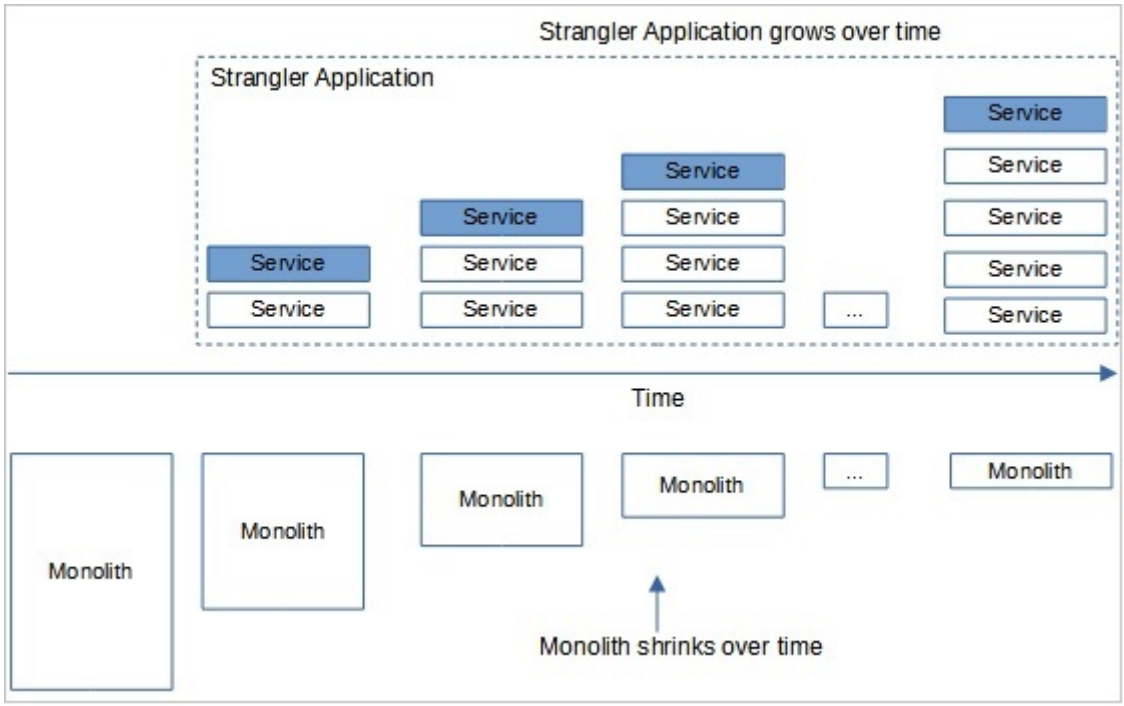
We can define a microservice using strangler pattern. A strangler application has two types of services −

- **Existing Behavior** − These services exhibits the behavior that previously resides in Monolith.
- **New Functionalities** − These services implements new behavior.

So over the time of development, microservices increases and monolith shrinks with features moving out from monolith to Strangler Application.

Example

Consider an example of an Online Book Store. Initially we have only developed Book Catalog management service and other services are supported in legacy monolith application. During the course of development, more and more services are developed and functionalities are moved away from a monolith.



So when a new service is developed, the monolith is strangled, the old component is decommissioned and new microservice is deployed and supports the new functionality. A strangler pattern can be implemented using three steps −

- **Transformation** − Develop the microservices independently to implement a particular functionality of a monolith.
- **Co-Exist** − Both Monolith and Microservices will work. User can access functionality from both components.
- **Elliminate** − Once the newly developed functionality is production ready, remove the functionality from the monolith.

Advantages

- **Test Driven Development** − As services are developed in chunks, we can use TDD for business logic and ensure the code quality.
- **Independent Teams** − Teams can work in parallel fashion on both monolith and microservices thus making a robust delivery mechanism.

Microservices Design Patterns - API Gateway

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. When a large, complex application is to be built using microservice architecture, microservices can use different protocols. For example, some microservices

using REST and some are following AMQP. Now problem is how to allow clients to access each microservice seemlessly without worrying about protocols and other intricacies.

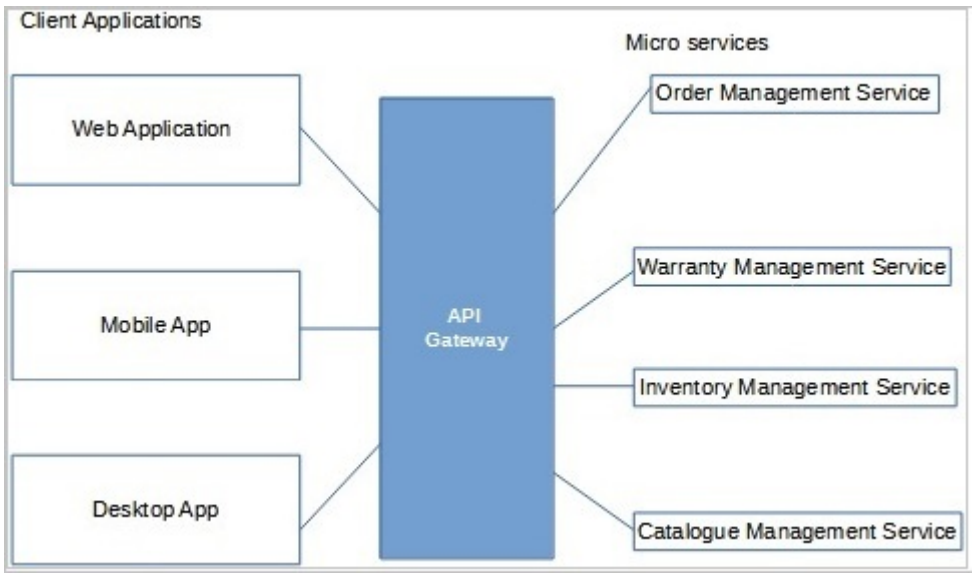
Solution

We can define an API Gateway which will acts as single entry point for all type of clients. Following are the other benefits of API Gateway −

- **Simple Proxy** − API Gateway can acts as simple proxy to some requests to redirects them to relevant service.
- **Multiple Services** − API Gateway can redirects call to multiple services.
- **Client Specific API** − API Gateway can provide client specific API as well, like a different API for Desktop version than a Mobile Application.
- **Protocol Handling** − API Gateway handles the communication protocols to each service call internally and clients are concerned only with request/response.
- **Security and Authentication** − API Gateway can implement a security that each request goes to service only after authentication and authorization.

Example

Consider an example of an Online Book Store. API Gateway allows to use the online Book store APIs on multiple devices seemlessly.



Advantages

- **Client Insulation** − Clients are insulated from knowing the location of microservices or how to call them.
- **Multiple Service Calls** − API Gateway can handle multiple services and give result as one and thus can reduce the round trips and increase the performance.

- **Standard Interface** − API Gateway provides a standard interface to Clients to get responses from microservices.

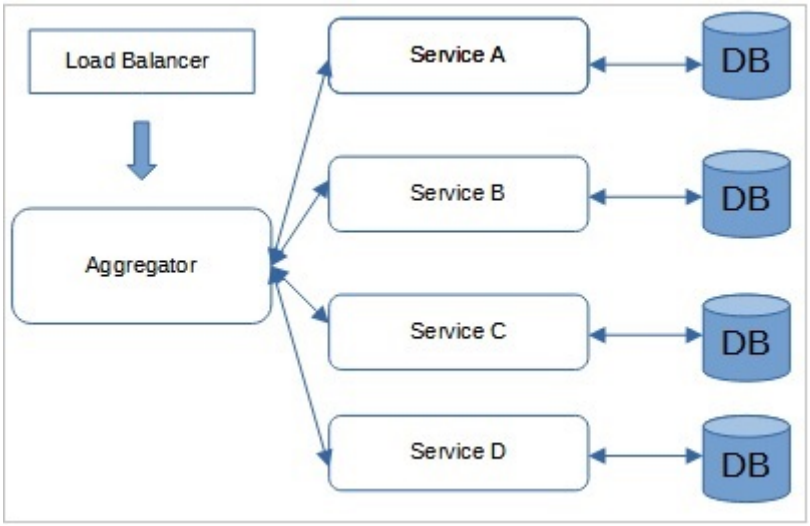
Microservices Design Patterns - Aggregator

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. When a large, complex application is to be built using microservice architecture, we often need to get the combined result of multiple microservices and show on the application.

Solution

We can define an Aggragator as a simple web module will act as a load balancer, which means it will call different services as per requirements. Following is a diagram depicting a simple microservice web app with aggregator design. As seen in the following image, the "Aggregator" is responsible for calling different services one by one. If we need to apply any business logic over the results of the service A, B and C, then we can implement the business logic in the aggregator itself.



An aggregator can be again exposed as another service to the outer world, which can be consumed by others whenever required. While developing aggregator pattern web service, we need to keep in mind that each of our services A, B and C should have its own caching layers and it should be full stack in nature.

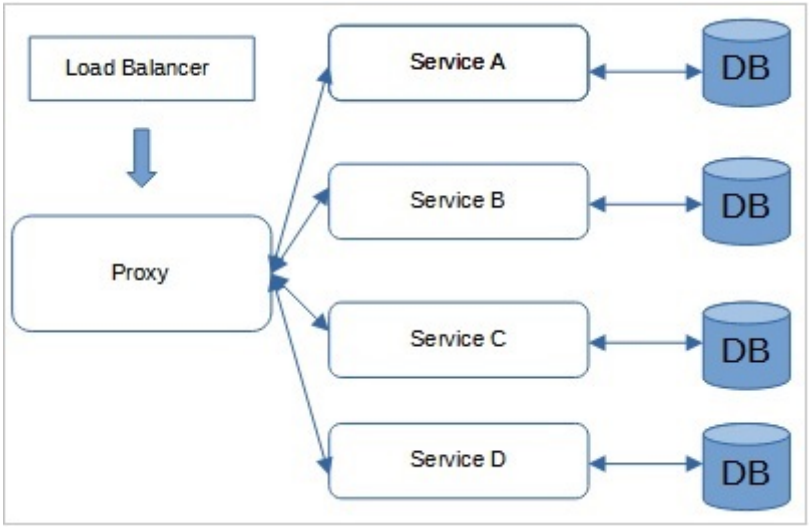
Microservices Design Patterns - Proxy

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. When a large, complex application is to be built using microservice architecture, we often need to prepare a unified interface which can do the common work like authentication and authorization before each service call.

Solution

Proxy microservice pattern is a variation of the aggregator model. In this model we will use proxy module instead of the aggregation module. Proxy service may call different services individually.



In Proxy pattern, we can build one level of extra security by providing a dump proxy layer. This layer acts similar to the interface.

Advantages

- Proxy pattern provides a uniform interface instead of different interface per microservice.
- Proxy pattern allows to apply uniform concerns like logging, security etc at one place.

Client Side UI Composition

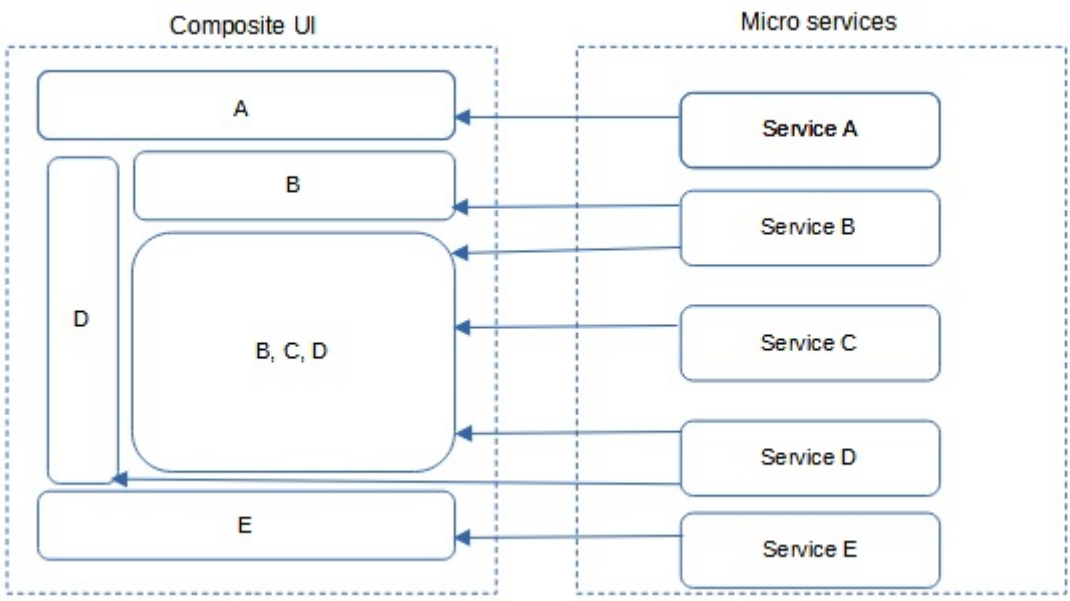
Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. Now how to develop a UI page/screen which can displays data from multiple services.

Solution

Each UI team can develop a client side UI component such as Angular Component which implements or corresponds to a particular microservice. For multiple services, UI team

responsible to prepare a skeleton UI or page skeletons by building pages which are composed of multiple service specific UI components.



Advantages

- **Indepedent UI Teams** − Each UI team can work once a microservice contract is available without any need for all microservices availability.
- **Managable UI development** − UI being developed in components becomes managable and efficient.
- **Easier Development** − UI development becomes easier and maintainable.

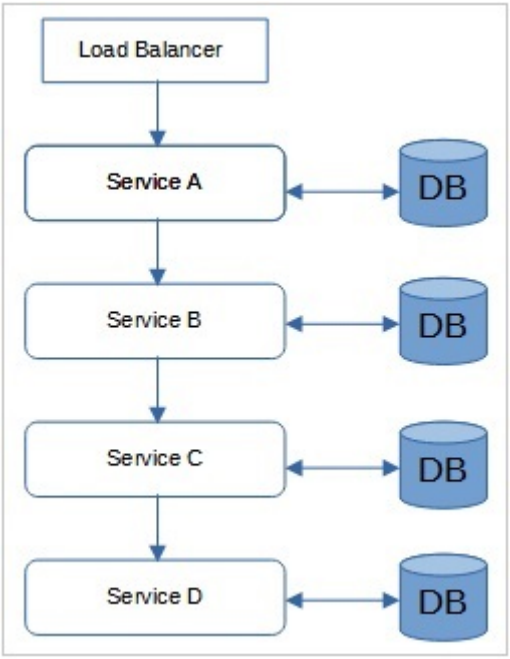
Chain of Responsibilities

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. Now if one service needs output of another service as dependency then how to handle such a case.

Solution

We can use Chain of Responsibilities Pattern. As the name suggests, this type of composition pattern will follow the chain structure. Here, we will not be using anything in between the client and service layer. Instead, we will allow the client to communicate directly with the services and all the services will be chained up in a such a manner that the output of one service will be the input of the next service. Following image shows a typical chained pattern microservice.



Disadvantage

One major drawback of this architecture is, the client will be blocked until the entire process is complete. Thus, it is highly recommendable to keep the length of the chain as short as possible.

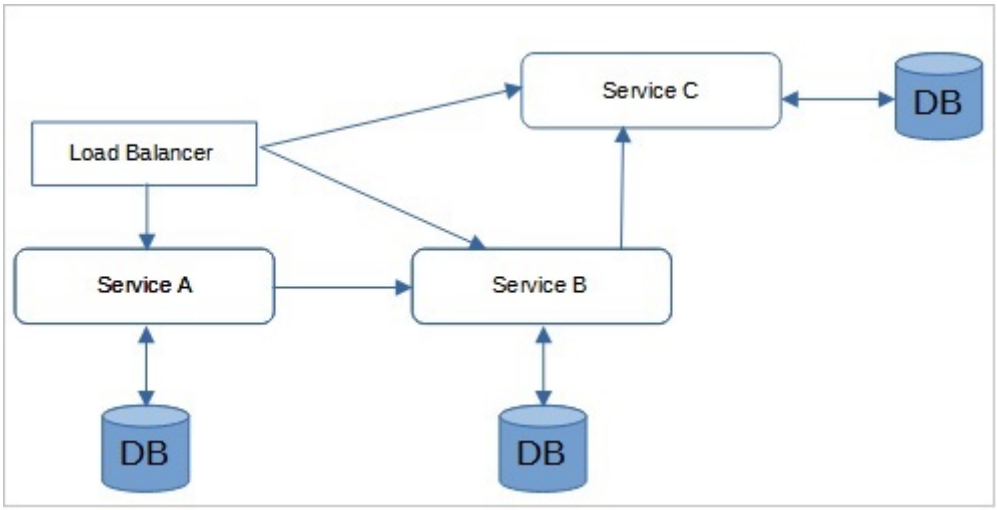
Microservices Design Patterns - Branch

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. Now consider a case where one service needs output of another service as dependency and client can call any service.

Solution

We can use Branch Microservices Design Pattern here. Branch microservice pattern is the extended version of aggregator pattern and chain pattern. In this design pattern, the client can directly communicate with the service. Also, one service can communicate with more than one services at a time. Following is the diagrammatic representation of Branch Microservices.



Advantages

Branch microservice pattern allows the developer to configure service calls dynamically. All service calls will happen in a concurrent manner, which means service A can call Service B and C simultaneously.

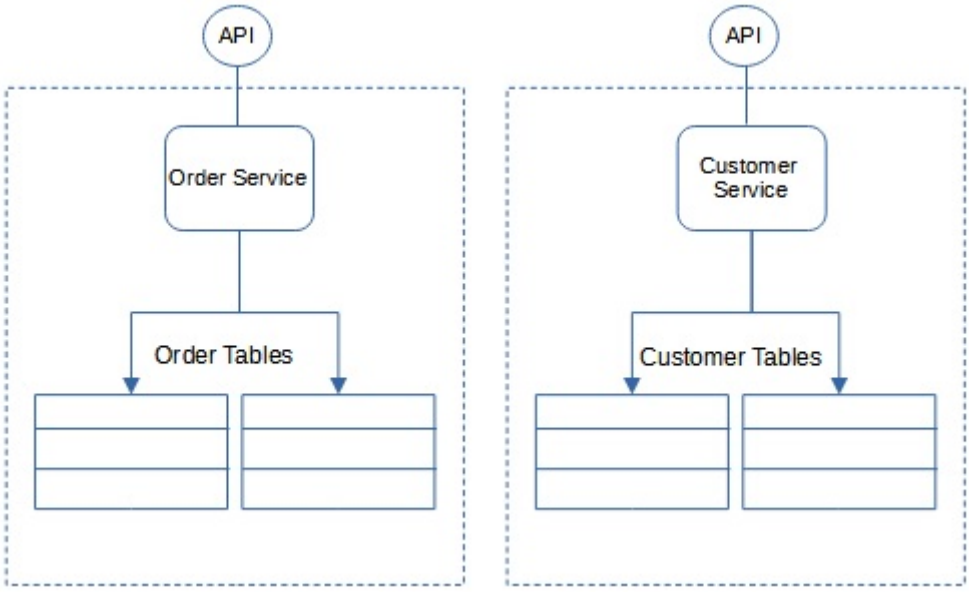
Database per Service

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. What should be the database structure/architecture in microservices based application.

Solution

We can keep each microservice data private to that microservice and this data will be accessible only via relevant microservice. The microservice will use its own database for transactions. Following diagram shows database per service design pattern implementation.



Database per Service does not always need to have seperate databases provisioned. We can implement the pattern using following ways considering a relational database.

- **Private tables per Service** − Each microservice can utilize a set of tables and these tables should be accessible only via their relevant microservice.
- **Schema per Service** − A seperate schema can be defined per microservice.
- **Database Server per Service** − Entire database server can be configured per microservice.

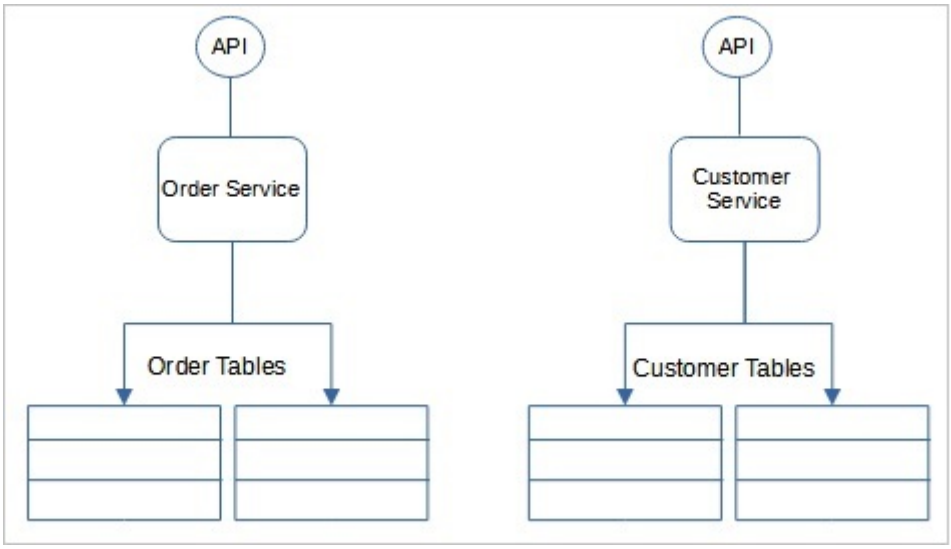
Shared Database per Service

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. What should be the database structure/architecture in microservices based application.

Solution

We can use a database which is shared among microservices. Each service is free to use data accessible to other services. Database will maintain the ACID transactions.



In this pattern, each service should use transaction management of underlying database so the ACID property of the database can be utilized. Conside the following pseudocode −

```
BEGIN TRANSACTION
…
SELECT * FROM ORDERS WHERE CUSTOMER_ID = ?
…
SELECT CREDIT_LIMIT FROM CUSTOMERS WHERE CUSTOMER_ID = ?
…
INSERT INTO ORDERS ... WHERE ORDER_LIMIT < CREDIT_LIMIT
…
COMMIT TRANSACTION
```

Here order service uses database transaction to ensure that during order, credit limit of the customer is checked.

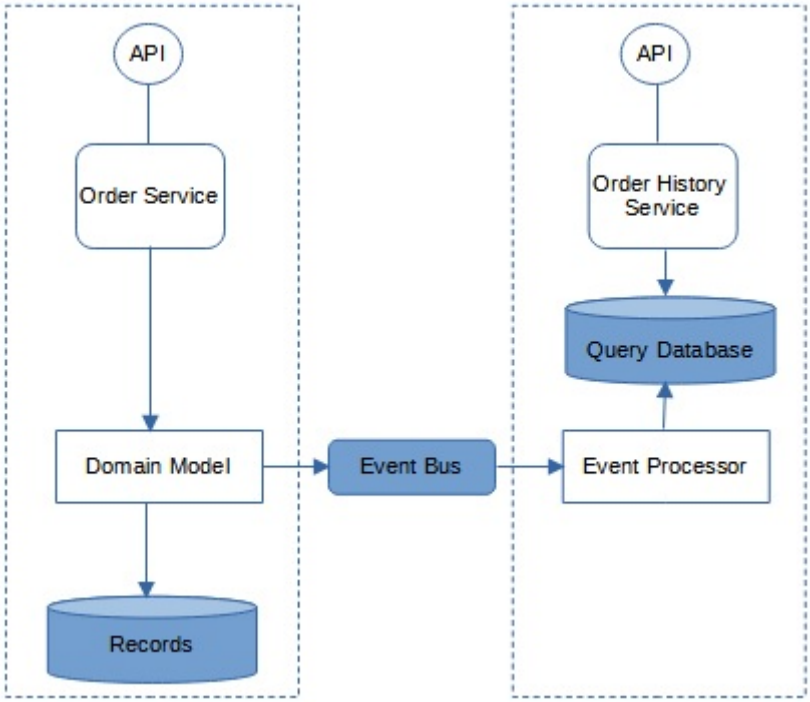
Command Query Responsibility Segregator

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment and if we've used a database per service design pattern then how to make query which needs data from multiple services.

Solution

We can define a view database which is a read-only data to support the required query. Application will keep the view database up to date by subscribing to the events raised by the services which owns the data. In this design pattern, we segregate the update and read operations. One service will only read the data and other services will update the data.



In order to implement this pattern, we often need to refactor the domain model to support seperate operations for querying data and to update data so that each operation can be handled by microservices independently. CQRS patterns ensures that operation that reads data is seperate from that which updates the data. So an operation can either read or write data but cannot perform both together.

Now multiple services can update the records and send events to application to update the view database. This helps the Query service to get the consistent data without any performance hit.

Microservices Design Patterns - Saga

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment and if we've used a database per service design pattern then how to implement a transaction which spans multiple services.

Solution

We can use Saga Pattern. A saga is a sequence of local transactions. In this pattern, each transaction updates the database and triggers an event or publishes a message for next transaction in saga. In case, any local transaction fails, saga will trigger series of transactions to undo the changes done so far by the local transactions.

Consider an example of order service and customer service. Order service can make an order and then ask customer service if credit limit is crossed or not. In case credit is crossed, the

customer service will raise an event to order service to cancel the order otherwise to place the order successfully.

In order to implement this pattern, we often need to Choreography based saga or Orchestrator based saga.

In choreography based saga, services handles the domain events during local transactions and either complete the transaction or undo the same while in orchestrator based saga, an orchestrator object handles events during local transactions and then tell coordinate which local transaction is to be executed.

Aynchronous Messaging

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. Microservices handle requests from clients and often need to communicate with other microservices to fulfill the requests. So there is a requirement for inter-process communication protocol.

Solution

We can use Aynchronous Messaging Pattern for inter service communication as using synchronous communication will result in tight coupling of services and also requires both client and service to be available during request.

Using Aynchronous Messaging, a service can communicate by exchanging messages over messaging channels. Following are some of the different ways of asynchronous messaging communications.

- **Request / Synchronous Response** − In this method, service makes a request to another service and expects a reply promptly.
- **Notifications** − In this method, service sends a message to a recipient but is not expecting any response. Recipient is not expected to send a response as well.

- **Request / Aynchronous Response** − In this method, service makes a request to another service and expects a reply within reasonable timeframe.
- **Publish / Subscribe** − In this method, service publishes a message to zero or more recipients. Services which subscribe the message will receive the same. No response needed.
- **Publish / Aynchronous Response** − In this method, service publishes a message to zero or more recipients. Services which subscribe the message will receive the same. Some of the them sends back an acknowledgement/reply.

Example

RabbitMQ and Apache Kafka are good examples of asynchronous messaging in microservices arena.

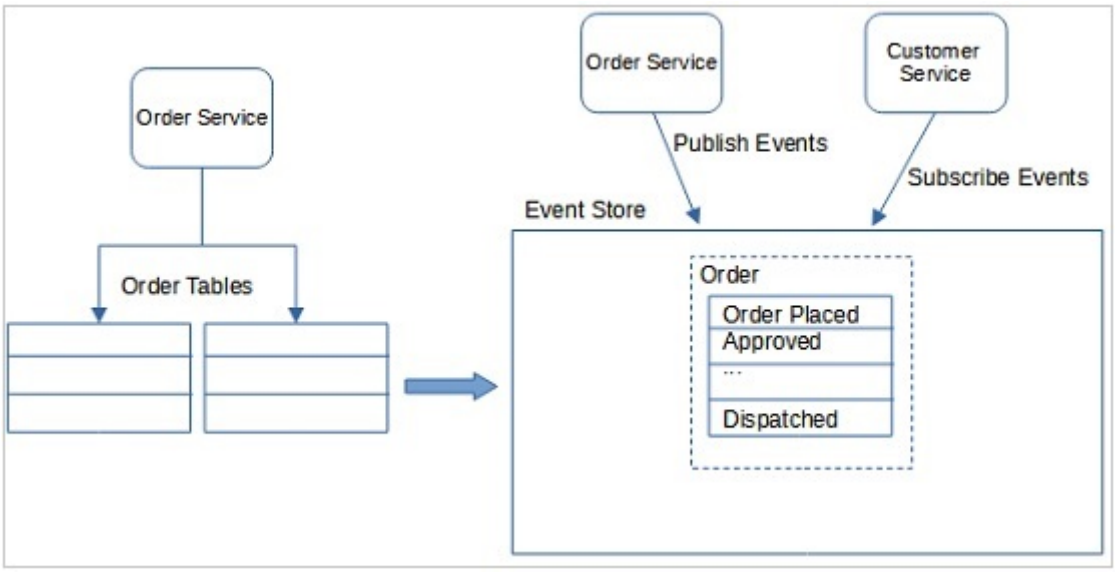
Event Sourcing

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment and if we've used a database per service design pattern then how to implement a transaction which spans multiple services.

Solution

We can use Event Sourcing Pattern for inter service communication. In this type of communication, each service persists the events in event store for every action taken. Each service can subscribe to these events and correspondingly updates its transaction status. Consider a case of Order Service vs Customer Service. A customer service can subscribe to events updated by order service and update its status accordingly.



Advantages

Following are the advantages of using event sourcing pattern −

- **Ideal for Event driven Architecture** − This pattern allows to reliably publish events whenever a state changes.
- **Persistent Events** − As events are persisted instead of domain objects, object-relational mismatch never happens.
- **Audit Log** − As events captured every change, so audit logs covers 100% changes.
- **Entity State identification** − We can create temporal queries on events database to check the current state of the entity at any point.
- **Monolith to Microservice architecture movement get easier** − Using event sourcing pattern, we can create loosely coupled microservices which communicates via events. Thus migration from a monolith to microservice based application development becomes easier.

Log Aggregation

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. Requests often span multiple services. Each service instance write some information in its log file in a standardized format. These logs can be info, error, warning or debug logs. How to analyze and troubleshoot application problems using these logs.

Solution

We can use a centralized logging service which aggregates the logs from each service. User should be able to search and analyze the logs provided by this logging service. User should be

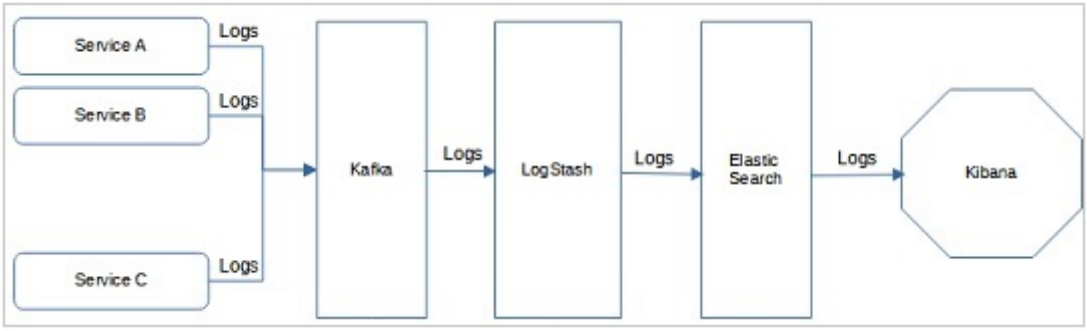
able to configure alerts when certain type of messages appear in logs.

Corelation ID

When first microservice receives a call, it should generate a corelation id which then can be passed to downstream services. This corelation id should be logged across all microservices. It will help to track the information spanning multiple services.

Searchable Logs

As logs should be placed at a centralized location, following diagram showcase how to use Kafka, LogStash and Kibana to aggregate logs and search the indexed logs using required filters.



Microservices generates logs, which are published using kafka log appender which then output the log messages to kafka cluster. LogStash ingests the messages from kafka, transforms the messages and publish to elastic search container. Now kibana provides a visual interface to search/read indexed logs from elastic search container and provides required filters.

Performance Metrics

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. How to analyze and troubleshoot application problems. How to track application performance and check bottlenecks. How to tracking mutiple services with minimum runtime overhead?

Solution

We can implement a instrumentation service which will be responsible to gather statistics about individual operations and a central metrics service which should aggregates metrics and provides the reporting and alerting. These services can collect the performance metrics in two ways −

- **Push** − A services pushes the metrics to central metrics service.

- **Pull** − The central metrics service pulls the metrics from the services.

Examples

Following are the examples of Instrumentation libraries −

- Java Metrics Library − A Java library to get insight into what code does in production.
- Prometheus client libraries − Prometheus libraries to monitor services.

Following are the examples of Metrics Aggregation libraries −

- Prometheus − An open-source systems monitoring and alerting toolkit.
- AWS Cloud Watch − AWS resources and service observability and monitoring service.

Distributed Tracing

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. Requests often span multiple services. Using external monitoring, we can check overall response time and no. of invocations but how to get insight on individual transactions/operations. A service may use databases, messaging queues, event sourcing etc. How to track scattered logs across multiple services?

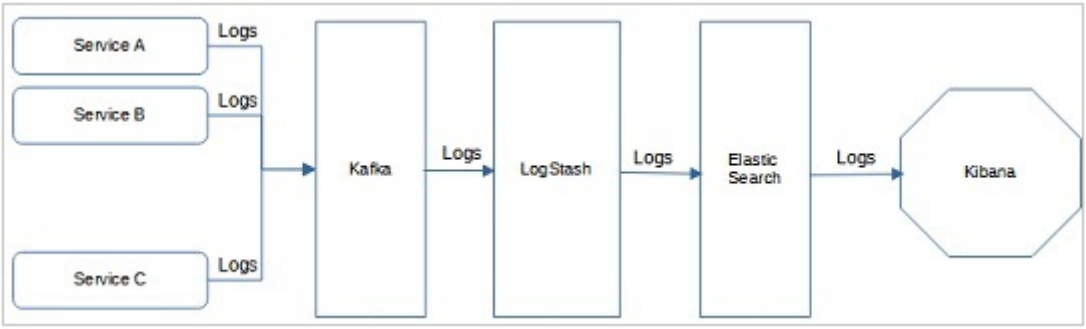
Solution

We can instrument a service which is designed to perform the following operations −

- **Corelation ID** − Generate a unique external request id per external request and pass this external id to each service involved in processing the request.
- **Log the Corelation ID** − Each log message generated by processing service should have this correlation id.
- **Record the Details** − Records the start/end time and other relevant details in logs when a request is processed by a service.

Searchable Logs

As logs should be placed at a centralized location, following diagram showcase how to use Kafka, LogStash and Kibana to aggregate logs and search the indexed logs using required filters.



Microservices generates logs, which are published using kafka log appender which then output the log messages to kafka cluster. LogStash ingests the messages from kafka, transforms the messages and publish to elastic search container. Now kibana provides a visual interface to search/read indexed logs from elastic search container and provides required filters.

Microservices Design Patterns - Health Check

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. Now in case a database is down or cannot afford more connections then a monitoring system should raise alerts. Loadbalancer/service registry/api gateway should not redirect any request to such failed service instances. So we need to detect if a running service instance is able to take request(s) or not.

Solution

We can add a health check point to each service e.g. HTTP /health which returns the status of service health. This endpoint can perform the following tasks to check a service health −

- **Connections Availablity** − Status of database connections or connections to infrastructure services used by current service.
- **Host Status** − Status of host like disk space, cpu usage, memory usage etc.
- **Application specific logic** − business logic determining service availablity.

Now a monitoring service e.g. load balancer, service registry periodically invokes the health check endpoint to check the health of the service instance.

External Configuration

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. These services often interacts with infrastructure services or 3rd party services.

Insfrastructure services may include a service registry, a message broker, database server. 3rd party services can be payment services, email services, messaging services. Apart from varying services, environments often vary. Consider the following cases −

- **Configuration Data** − Configurations to external/3rd party services should be provided to the micro services e.g. database credentials, network urls etc.
- **Multiple Environments** − There are often varying environments like dev, qa, test, staging or pre-prod and production. A service should be deployed on each environments without any code modifications.
- **Varying configuration data** − Configurations to external/3rd party services also varies from dev to production e.g. dev databse to production database, test payment processor vs original payment processor services.

Solution

We can externalize all configurations from database credentials to network urls. Service will read the configuration data during startup e.g. from a properties file/ system environment variables or using command line arguments. This pattern helps in deploying the microservices without any modification/recompilation needed.

Service Discovery

Problem Statement

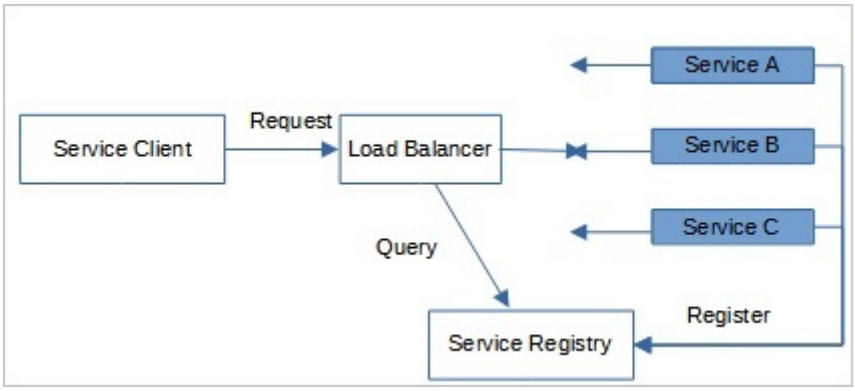
Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. These services often run in containerized/virtual environments and their number of instances and location changes dynamically.

So we require a mechanism to enable client of a microservice to make requests to dynamically changing service instances.

Solution

We can use Service Discovery pattern. To implement this pattern, we need a router/load balancer running at a particular fixed location and a service registry where all microservice instances are registered.

Now a client makes a service request, it will come to the load balancer which then inquires the service registry. If service instance is available, then the request is redirected to the available service instance.



Microservices Design Patterns - Circuit Breaker

Problem Statement

Microservice architecture structures an application as a set of loosely coupled microservices and each service can be developed independently in agile manner to enable continous delivery/deployment. These services often interacts with other microservices. Now there is always a possibility that a service is overloaded or unavailable. In such a case the caller service will also wait. If multiple services are getting blocked then it will hamper the performance and can cascade the impact on overall application.

Now, how to prevent a service failure or network failure from cascading to other services. If one service is down then it should not be given further requests.

Solution

We can use circuit breaker pattern where a proxy service acts as a circuit breaker. Each service should be invoked through proxy service. A proxy service maintains a timeout and failures count. In case of consecutive failures crosses the threshold failures count then proxy service trips the circuit breaker and starts a timeout period. During this timeout period, all requests will failed. Once this timeout period is over, proxy service allows a given limited number of test requests to pass to provider service. If requests succeed the proxy service resumes the operations otherwise, it agains trips the circuit breaker and starts a timeout period and no requests will be entertained during that period.

Blue Green Deployment

Microservice architecture structures an application as a set of loosely coupled microservices and each service should be developed independently in agile manner to enable continous delivery/deployment. When a large, complex application is to be built using microser\

architecture, the major problem is how to design loosely coupled microservices or to break a large application into small loosely coupled services while keeping both the system in production.

Solution

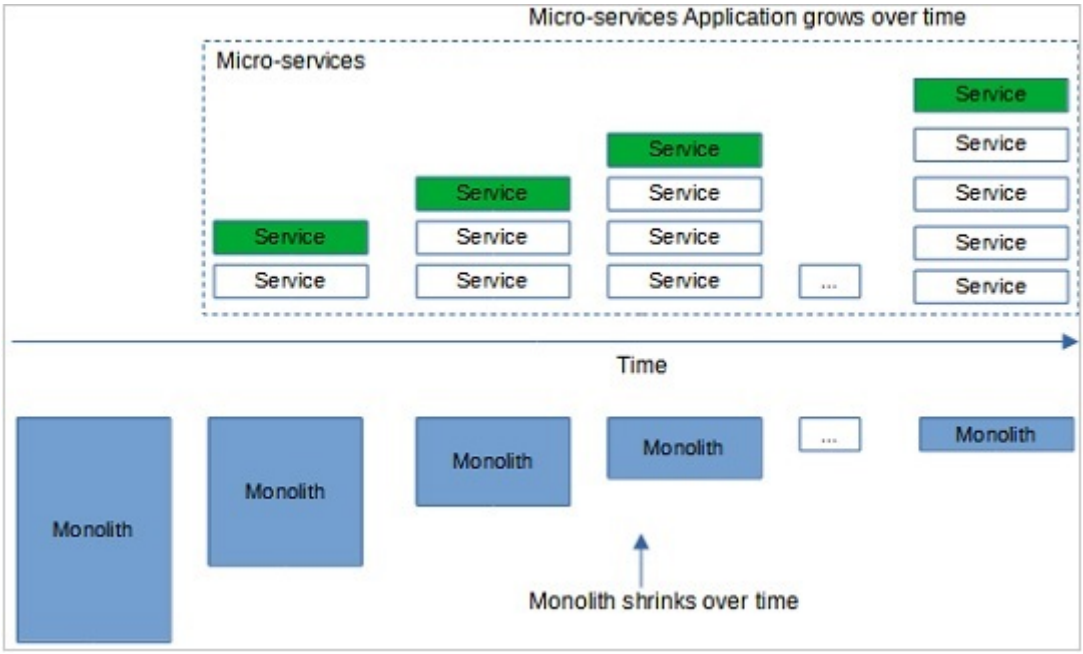
We can define deploy newly development microservices using blue-green deployment. In this model, user traffic is diverted from old application to new microservice application gradually. One a microservice is available in production, the load balancer redirects the request targetted for old application to the new microservice.

- **Blue Environment** − The old application running in the production is called blue environment.
- **Green Environment** − The new services deployed which replicates the given part of old application is called the green environment.

So over the time of development, microservices increases and monolith shrinks with features moving out from monolith to microservices Application.

Example

Consider an example of an Online Book Store. Initially we have only developed Book Catalog managment service and other services are supported in legacy monolith application. During the course of development, more and more services are developed and functionalities are moved away from a monolith.



This mode of deployment helps in reducing the downtime or even zero downtime while migrating from a monolith to microservices based application.