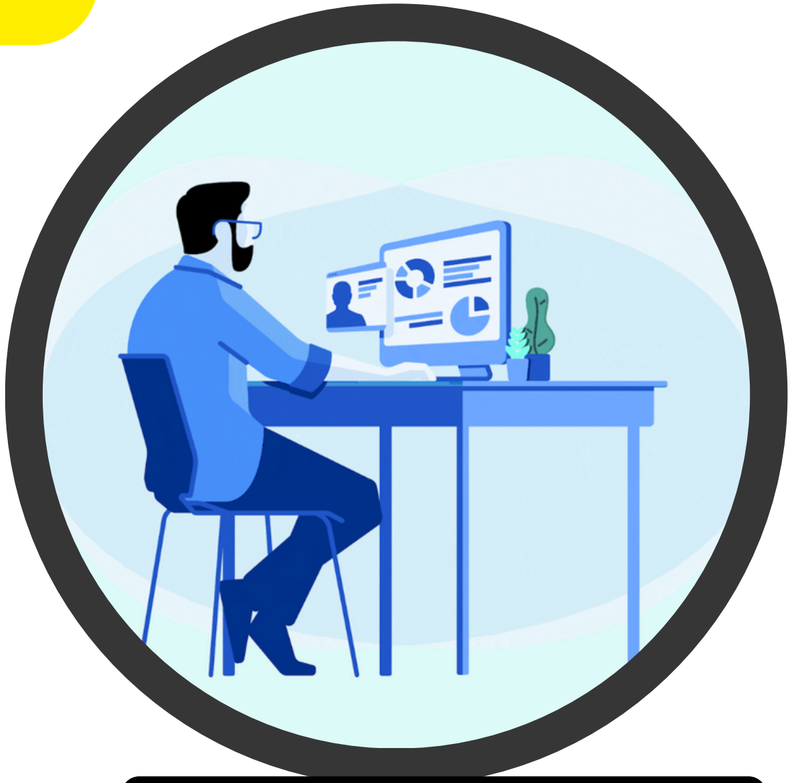
Spring Boot Microservice Master Course

Rajeev Gupta

rgupta.mtech@gmail.com

https://www.linkedin.com/in/rajeevguptajavatrainer



rgupta.mtech@gmail.com

Trainer's Profile

- Expert Java trainer MTech (Computer Science) with 18+ years experience in expertise in Java , OOAD, Design Patterns core, Spring Framework, Spring Boot, Spring Boot Microservice, Spring REST, Spring Data, Spring Security, Spring WS, Spring Mongo DB, DevOps with Java, Jenkin, Docker, Kubernetes, Messaging with Rabbit MQ, Kafka, Cloud AWS, GCP, Hibernate 5, EJB 3, Struts 1/2
- Helping technology organizations by training their fresh and senior engineers in key technologies and processes.
- Taught graduate and post-graduate academic courses to students with professional degrees.
- Conducted about 100 batches including fresher and lateral engineers.

Corporate Client

- Bank Of America
- MakeMyTrip
- GreatLearning
- Deloitte
- Kronos
- Yamaha Moters
- IBM
- Sapient
- Accenture
- Airtel
- Gemalto
- Cyient Ltd
- Fidelity Investment Ltd
- Blackrock
- Mahindra Comviva
- Iris Software
- harman
- Infosys
- Espire
- Steria
- Incedo
- Capgemini
- HCL
- CenturyLink
- Nucleus
- Ericsson
- Ivy Global
- Avaya
- NEC Technologies
- A.T. Kearney
- UST Global
- TCS
- North Shore Technologies
- Incedo
- Genpact
- Torry Harris
- Indian Air force
- Indian railways

rgupta.mtech@gmail.com

Training Etiquette

Punctuality

Join the session 5 minutes prior to the session start time. We start on time and conclude on time!

Feedback

Make sure to submit a constructive feedback for all sessions as it is very helpful for the presenter.

Silent Mode

Keep your mobile devices in silent mode, feel free to move out of session in case you need to attend an urgent call.

Avoid Disturbance

Avoid unwanted chit chat during the session.

Spring Boot Microservice Master Course

Module 1: Introduction to Microservices

Module 2: Twelve factor Rules

Module 3: Microservice design patterns

Module 4: Introduction to Spring cloud

Module 5: Eureka: Service discovery

Module 6: Feign: Declerative REST client

Module 7: Resilence 4j: Ciruit breaker

Module 8: Spring cloud config server

Module 9: Sluth and Zipkin

Module 10: Grafana Prometheus

Module 11: Spring boot ELK

Module 12: Rabbit MQ

Module 13: Kafka

Module 15:Project work

rgupta.mtech@gmail.com

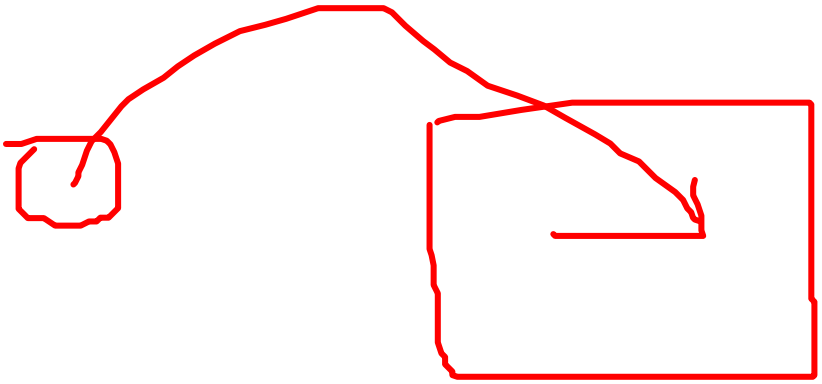
Module 1: Introduction to Microservices

rgupta.mtech@gmail.com

Spring boot : embedded tomcat

war : i would have external tomcat ... what i want that when i create the war file i dont want to kee the embeded tomcat along with that

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
    <scope>provided</scope>
</dependency>
```



JSP:

jasper : jsp engine
jsp --> eq servlet
missing from embedded tomcat

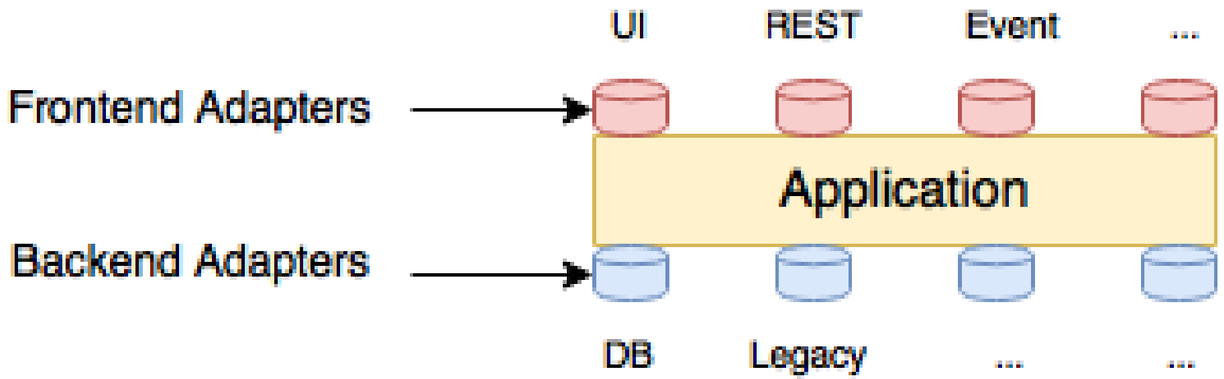
hence we have to put the jar file for it

What is microservice?

Microservices are an architectural style used by many organizations today as a game changer to achieve high degrees of agility, speed of delivery, and scale.
Microservices gives us a way to develop physically separated modular applications.

Microservices originated from the idea of Hexagonal Architecture(2005)

Hexagonal architecture advocates to encapsulate business functions from the rest of the world. These encapsulated business functions are unaware of their surroundings.



rgupta.mtech@gmail.com

What is microservice?

These services are built around business capabilities and independently deployable by fully automated deployment machinery.
There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies."

RestTemplate

"Microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API."

Martin Fowler, ThoughtWorks

Classical monolith design

Web Layer

(controllers, exception handlers, filters, view templates, and so on)

Service Layer

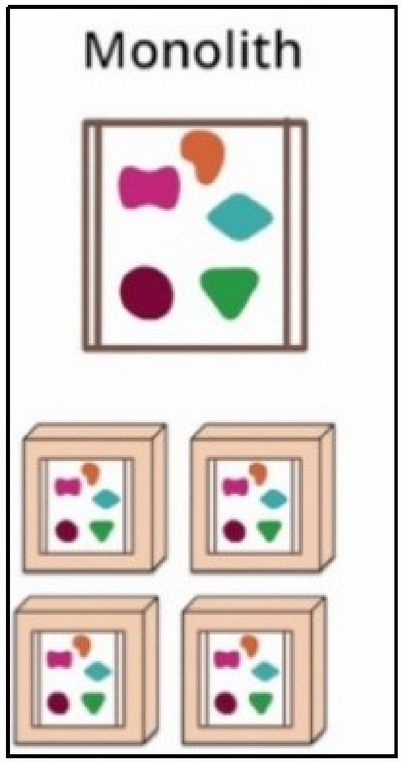
(application services and infrastructure services)

Repository Layer

(repository interfaces and their implementations)

rgupta.mtech@gmail.com

Classical monolith design vs Microservice



Classical architecture.

Application is built as a single unit.

Typical 3 layer EA:

- client-side UI (Browser, HTML + JS)
- a database (RDBMS, NoSql ..)
- server-side application

(Java, .NET, Ruby, Python, PHP ..)

monolith

1. team size
2. tech adaptoviltiy is slow

Spring boot + mysql _+ spring data

3. ci/cd

4. scablity is a issue

Why not monolith ?

Any changes to the system involve building and deploying a new version of the application.

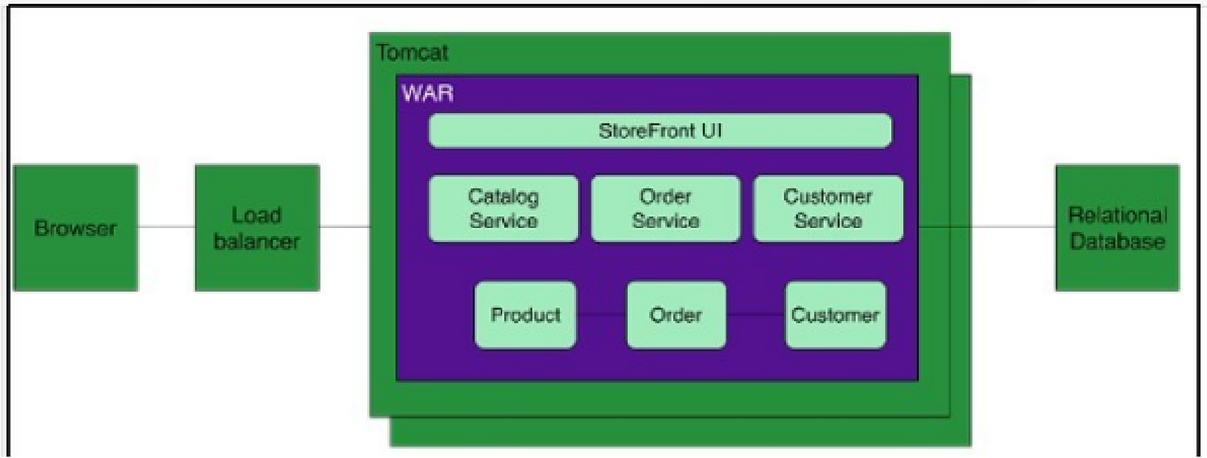
Changes require good planning and coordination.

Changes are expensive.

It is hard to keep a good modular structure.

Scaling requires scaling of the entire application rather than parts of it that require greater resource.

Long release cycles.



rgupta.mtech@gmail.com

Classical monolith Attributes

The server-side application

- will handle HTTP requests
- execute domain logic
- retrieve and update data from the database
- select and populate HTML views

This server-side application is a

Monolith - Single logical executable.

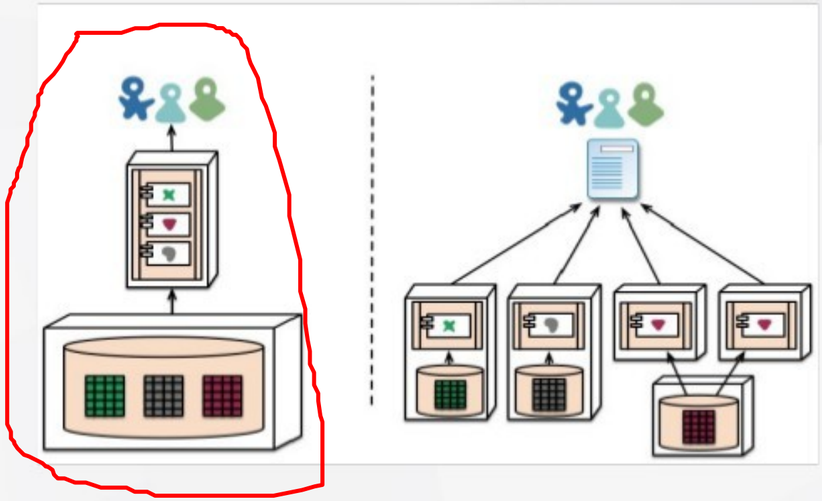
Monolithic server - natural approach

All logic for handling a request runs in a single process, divided and organized into classes, functions, and namespaces.

Application is developed on a developer's laptop, deployed to a testing environment and after that to production environment.

Monolith is horizontally scaled by running many instances behind a load-balancer.

Monolith vs Microservice attributes



Decentralized data management.

Decentralized governance.

They are built around business capabilities.

Each component runs as a separate application,

clustered to as many nodes as required.

rgupta.mtech@gmail.com

Monolith vs SOA

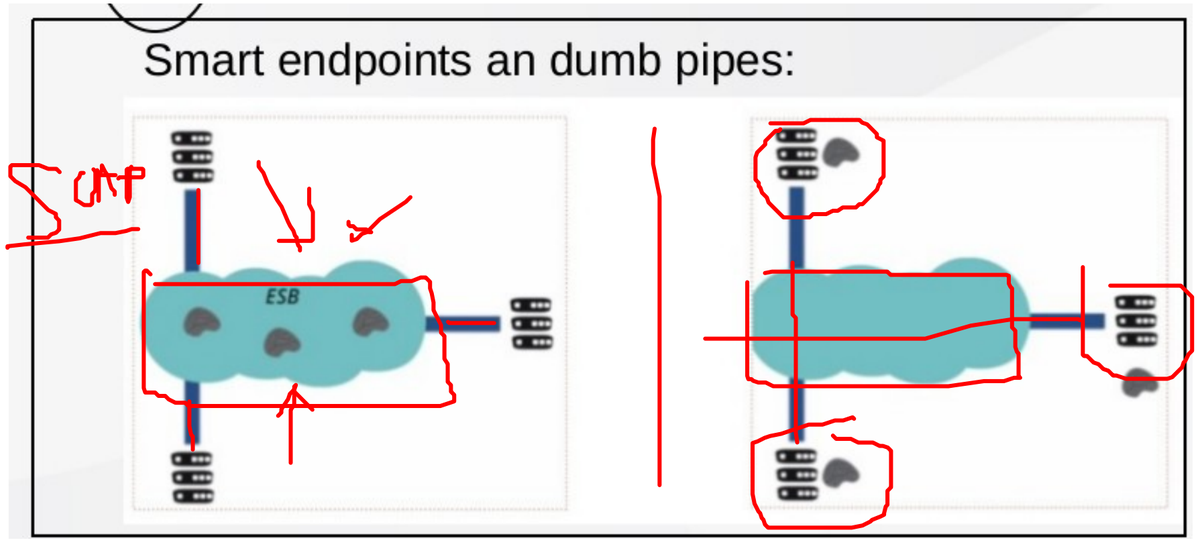
Service oriented arch

Applications aim to be as decoupled and as cohesive as possible.

Components usually communicate through REST.

Components sometime use lightweight messaging (RabbitMQ, Zero MQ).

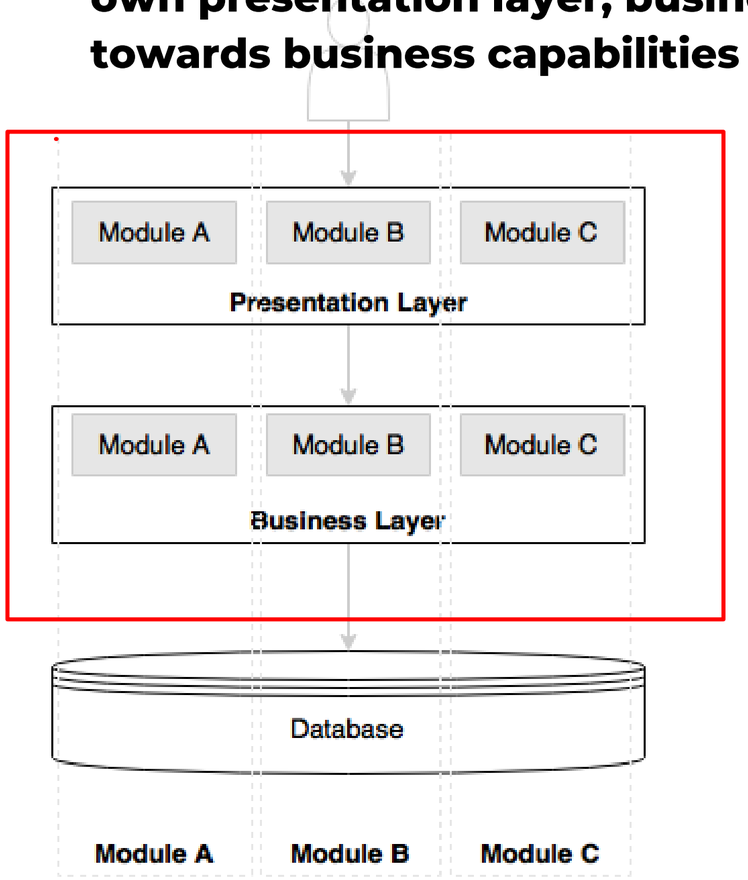
There is NO ESB or any other form of central communication management.

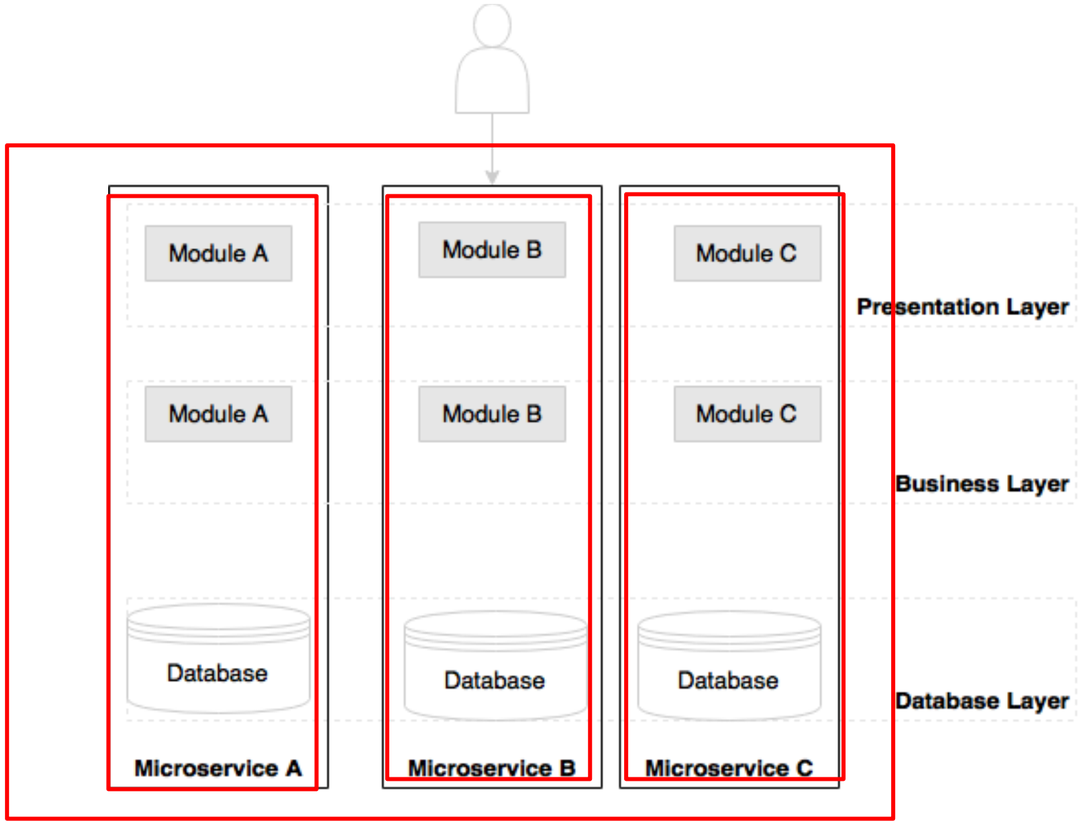


SOA	Microservices
XML	JSON
Complex to integrate	Easy to integrate
Heavy	Lightweight
Requires tooling	Light tooling
HTTP/SOAP	HTTP/REST

Classical monolith design vs Microservice

Each vertical slice represents a microservice. Each microservice will have its own presentation layer, business layer, and database layer. Microservices are aligned towards business capabilities

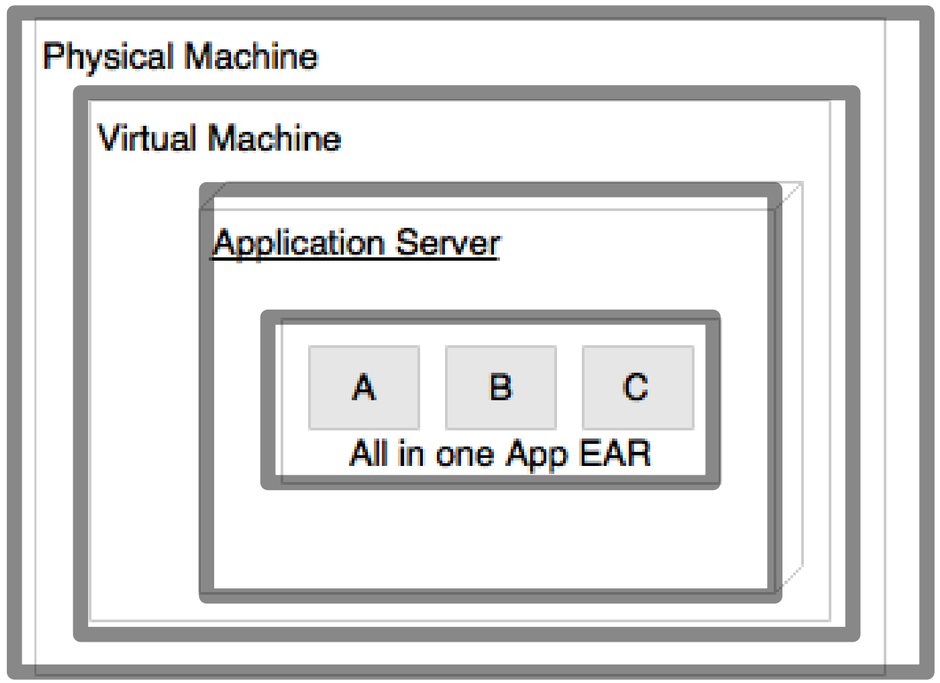




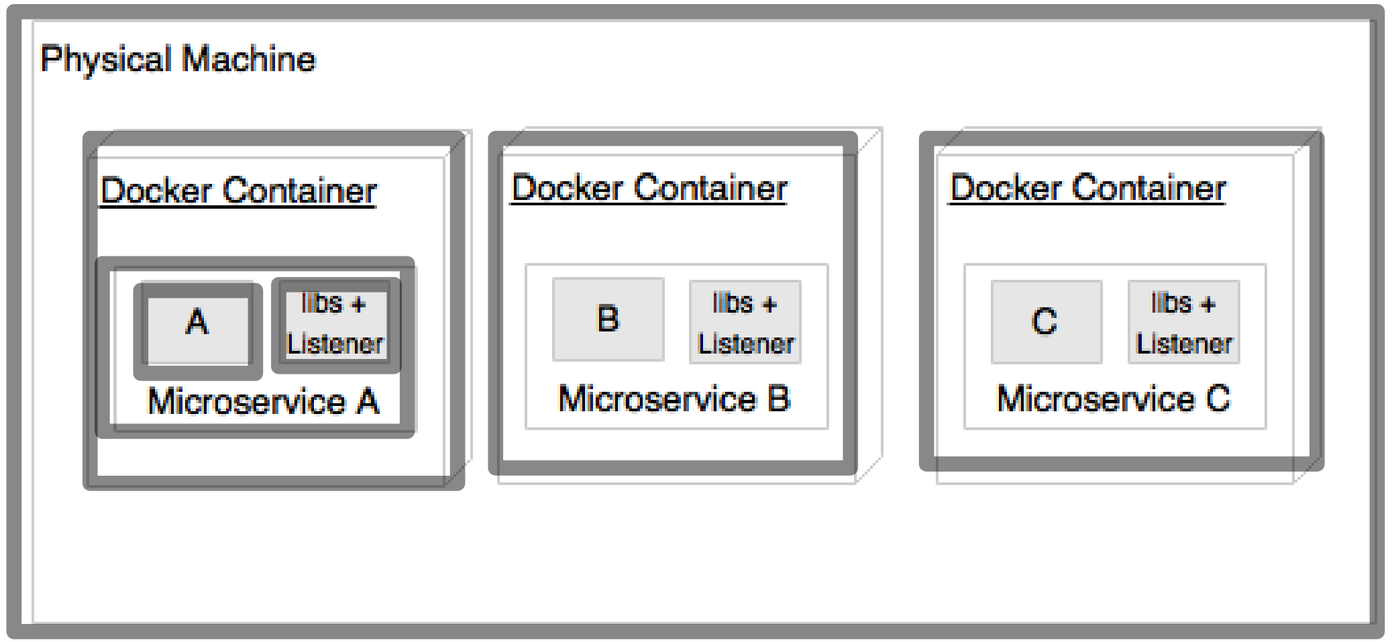
rgupta.mtech@gmail.com

Classical monolith design vs Microservice

Well-designed microservices are aligned to a single business capability; therefore, they perform only one function. As a result, one of the common characteristics we see in most of the implementations are microservices with smaller footprints



Traditonal Deployment

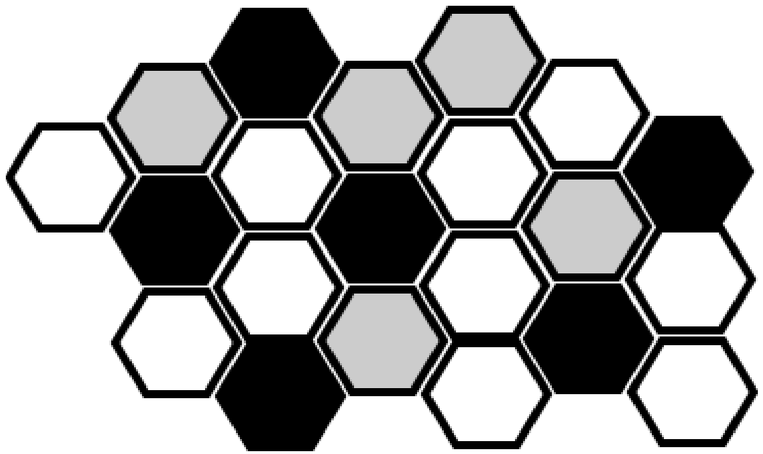


Microservices Deployment

rgupta.mtech@gmail.com

Microservices - The honeycomb analogy

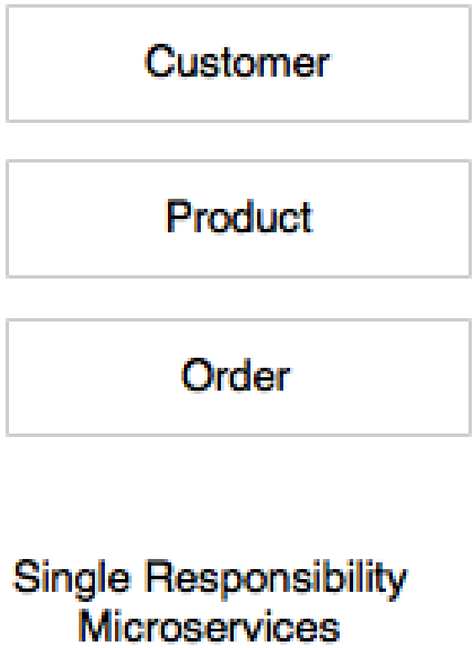
In the real world, bees build a honeycomb by aligning hexagonal wax cells. They start small, using different materials to build the cells.
Construction is based on what is available at the time of building. Repetitive cells form a pattern, and result in a strong fabric structure.
Each cell in the honeycomb is independent, but also integrated with other cells.
By adding new cells, the honeycomb grows organically to a big, solid structure.
The content inside the cell is abstracted and is not visible outside. Damage to one cell does not damage other cells, and bees can reconstruct those cells without impacting the overall honeycomb



rgupta.mtech@gmail.com

Single responsibility per service

It implies that a unit, either a class, a function, or a service, should have only one responsibility. At no point do two units share one responsibility, or one unit perform more than one responsibility. A unit with more than one responsibility indicates tight coupling

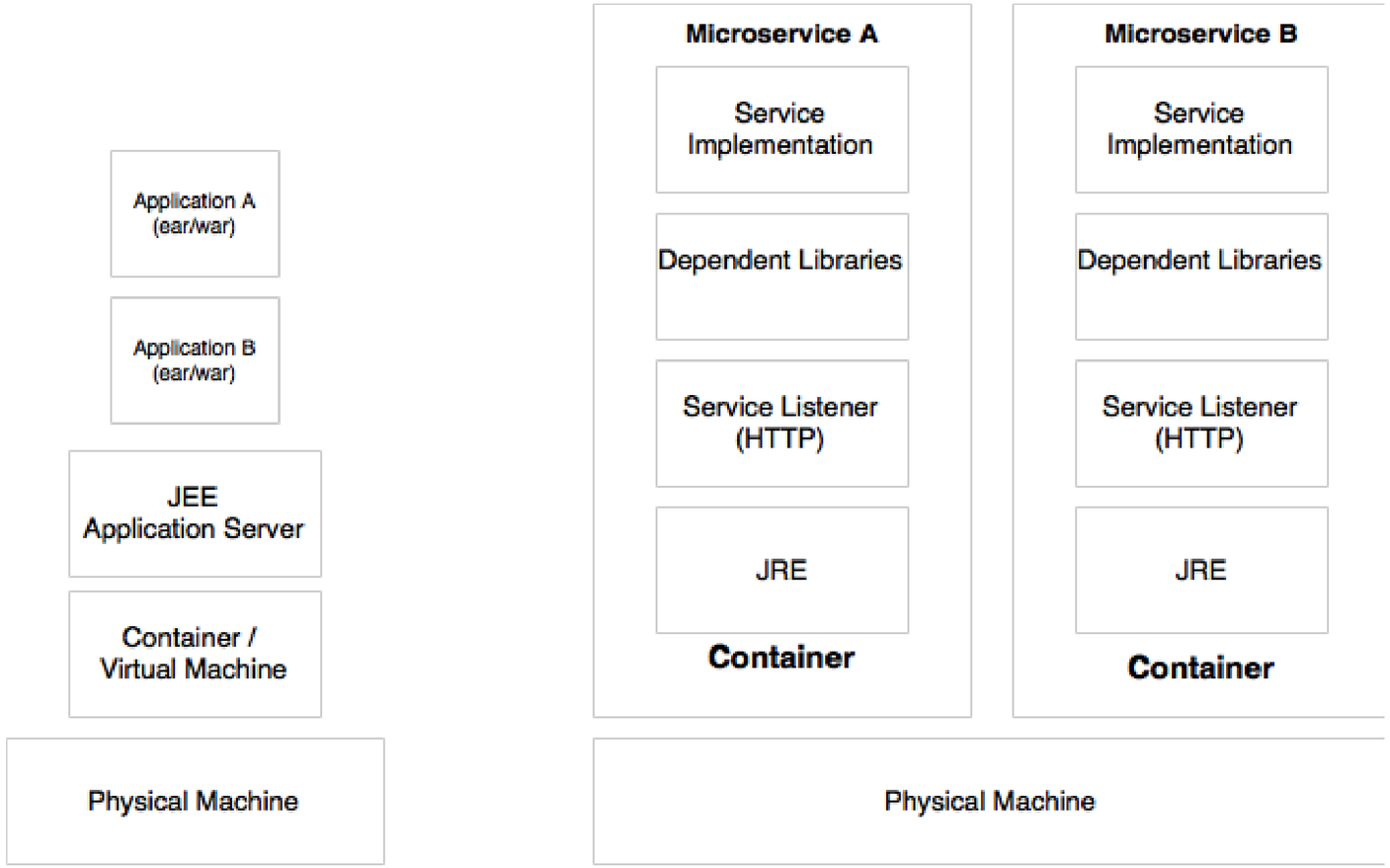


rgupta.mtech@gmail.com

Principles of microservices

Microservices are autonomous

Microservices are self-contained, independently deployable, and autonomous services that take full responsibility of a business capability and its execution. They bundle all dependencies including the library dependencies; execution environments, such as web servers and containers; or virtual machines that abstract the physical resources.



Characteristics of microservices

Services are first class citizens

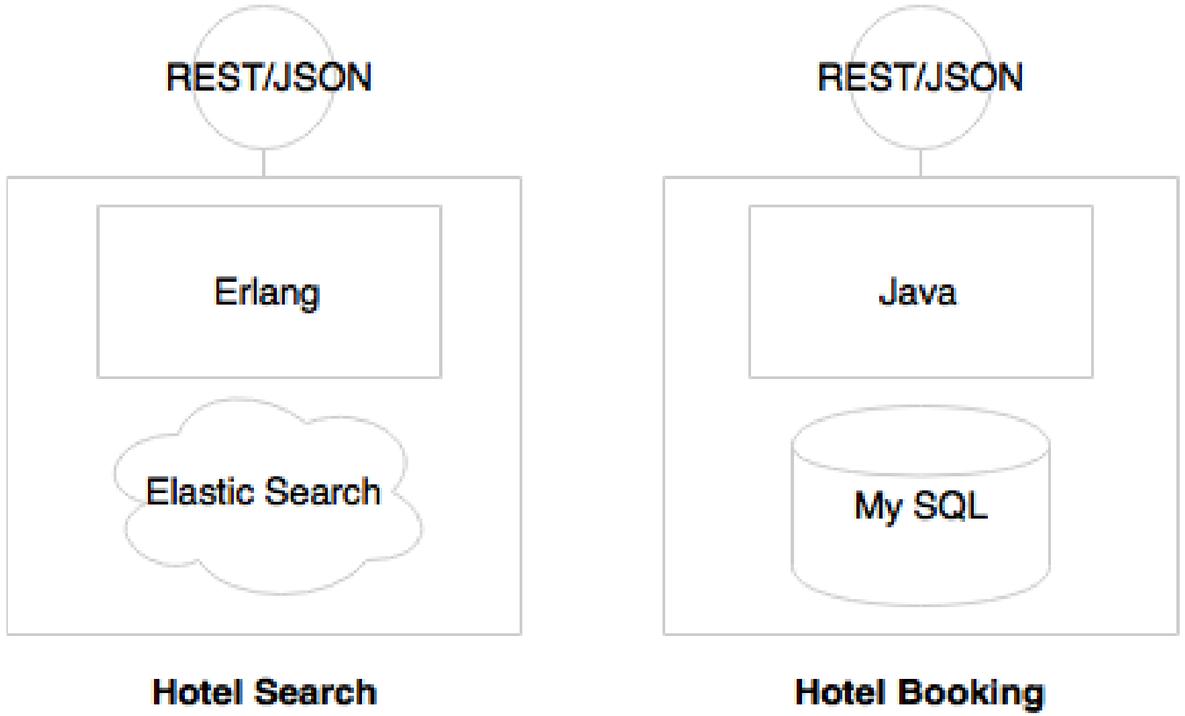
Characteristics of service in a microservice

- **Service contract**
- **Loose coupling**
- **Service abstraction**
- **Service reuse**
- **Statelessness**
- **Services are discoverable**
- **Service interoperability**
- **Service Composeability**

rgupta.mtech@gmail.com

Microservices with polyglot architecture

Since microservices are autonomous and abstract everything behind the service APIs, it is possible to have different architectures for different microservices



rgupta.mtech@gmail.com

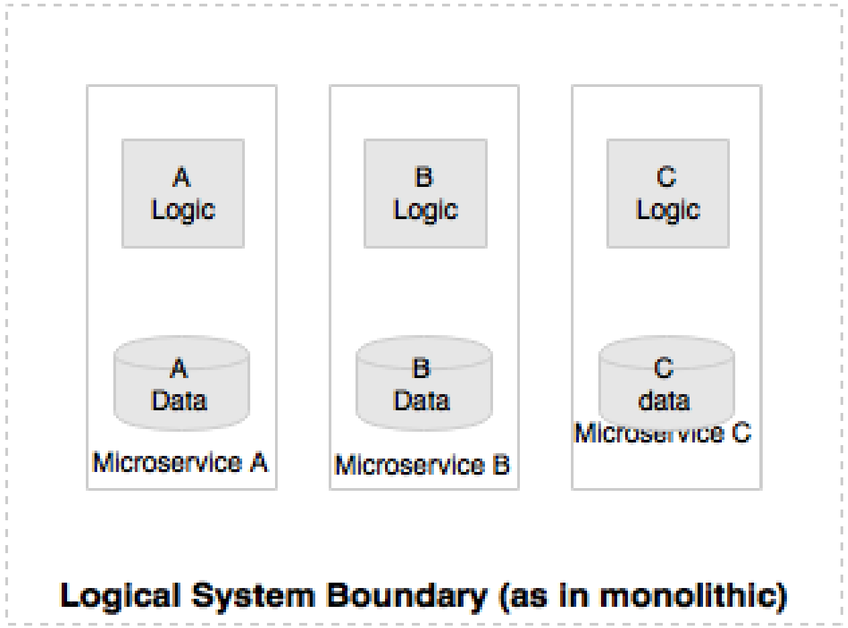
Microservices are distributed and dynamic

Single responsibility per service

Successful microservices implementations encapsulate logic and data within the service.
<u>This results in two unconventional situations:</u>

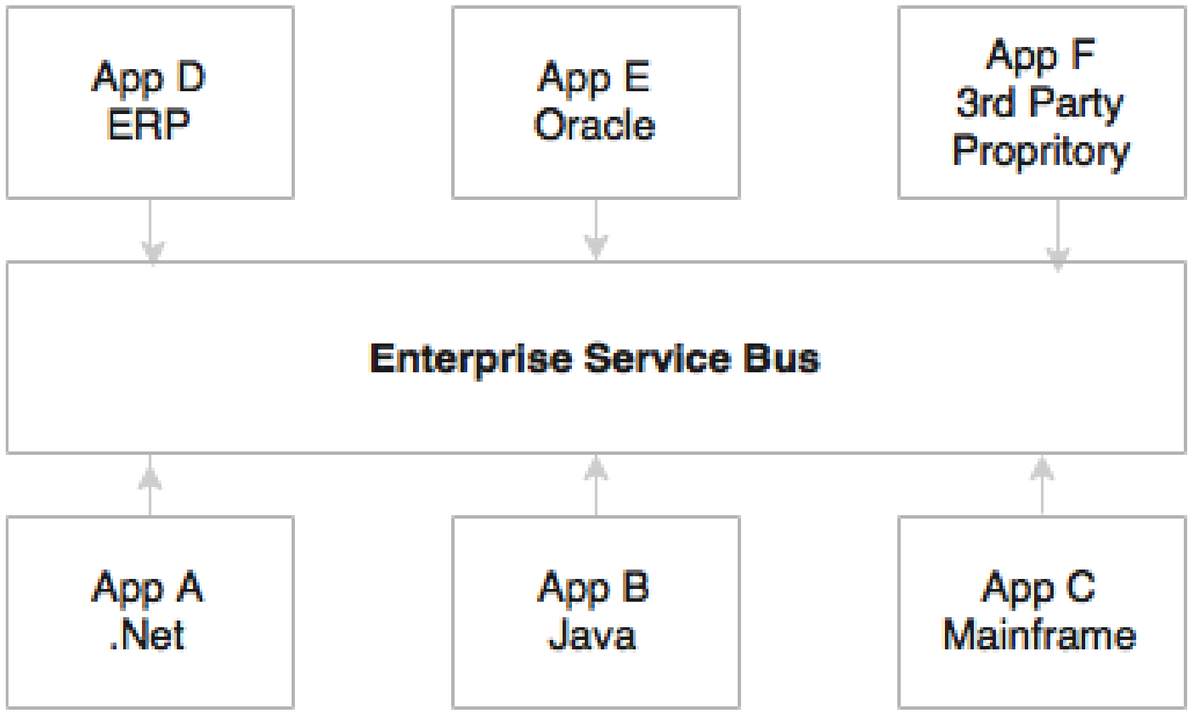
- **Distributed data and logic**
- **Decentralized governance**

Compared to traditional applications, which consolidate all logic and data into one application boundary, microservices decentralize data and logic. Each service, aligned to a specific business capability, owns its own data and logic



Microservices are Not same as SOA

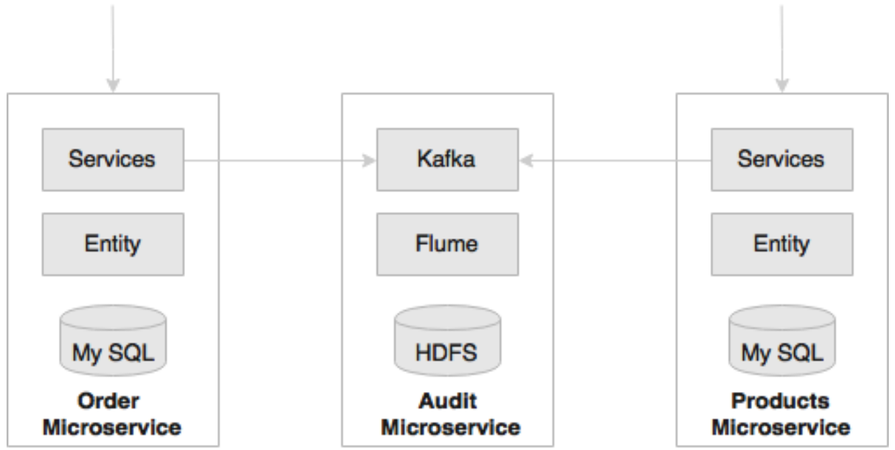
Microservices don't typically use centralized governance mechanisms the way they are used in SOA. One of the common characteristics of microservices implementations are that they are not relaying on heavyweight enterprise-level products, such as an Enterprise Service Bus (ESB). Instead, the business logic and intelligence are embedded as a part of the services themselves.



rgupta.mtech@gmail.com

Microservices benefits

Supports polyglot architecture



Enables experimentation and innovation

Elastically and selectively scalable

Allows substitution

Enables to build organic systems

Helps managing technology debt

Allowing co-existence of different versions

Supporting building self-organizing systems

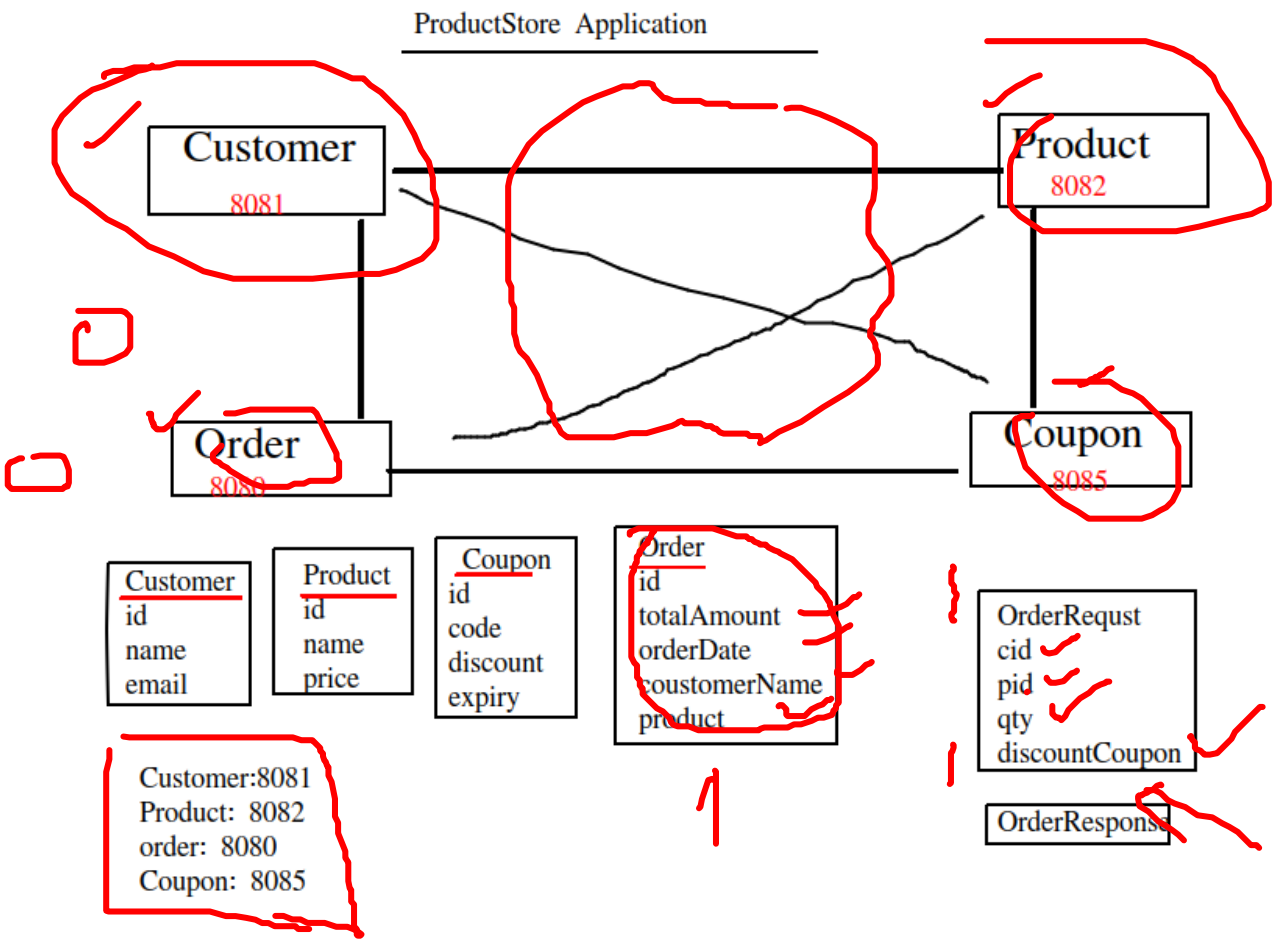
Supporting event-driven architecture

Enables DevOps

rgupta.mtech@gmail.com

Module 2: 12 Factor Rules

rgupta.mtech@gmail.com


ProductStore Application
Customer
8081
Product
8082
Order
8080
Coupon
8085
Customer
id
name
email
Product
id
name
price
Coupon
id
code
discount
expiry
Order
id
totalAmount
orderDate
coustomerName
product
OrderRequst
cid
pid
qty
discountCoupon
OrderRespons
Customer:8081
Product: 8082
order: 8080
Coupon: 8085

Twelve-Factor Apps

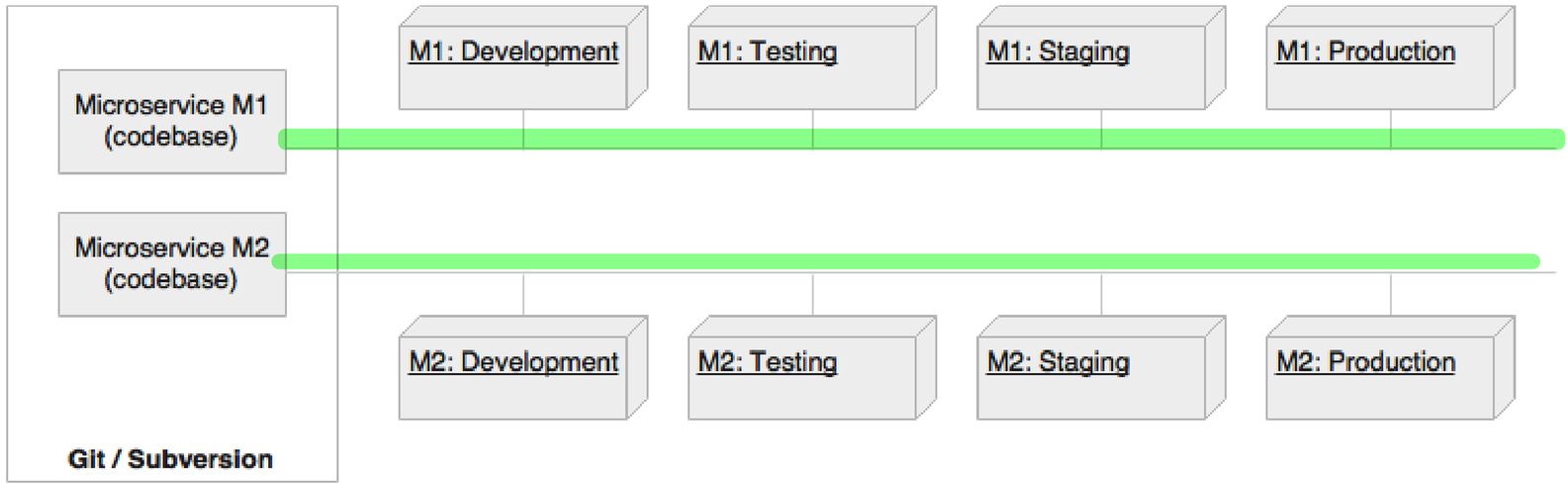
It is important to follow certain principles while developing cloud-native applications. Cloud native is a term used to develop applications that can work efficiently in a cloud environment, and understand and utilize cloud behaviors, such as elasticity, utilization-based charging, fail aware, and so on.

Twelve-Factor App, forwarded by Heroku, is a methodology describing characteristics expected from a modern cloud-ready application. These twelve factors are equally applicable for microservices as well. Hence, it is important to understand the Twelve-Factors.

rgupta.mtech@gmail.com

Single code base

The code base principle advices that each application should have a single code base. There can be multiple instances of deployment of the same code base, such as development, testing, or production. The code is typically managed in a source-control system such as Git, Subversion, and so on



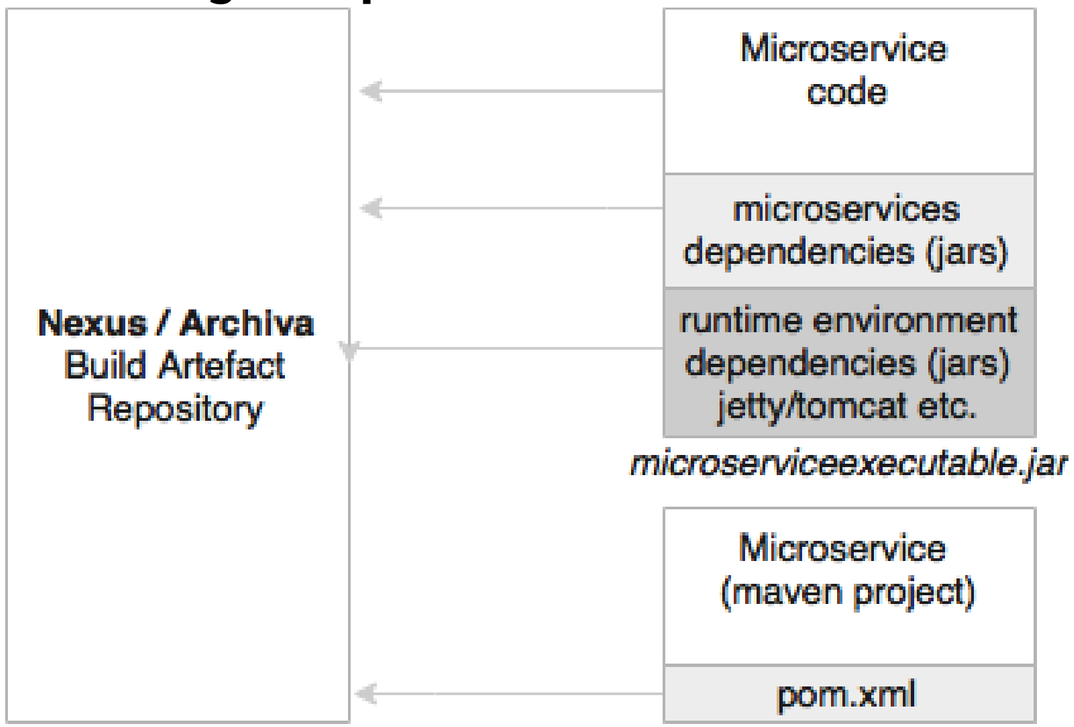
rgupta.mtech@gmail.com

Twelve-Factor Apps

Bundle dependencies

As per this principle, all applications should bundle their dependencies along with the application bundle. With build tools such as Maven and Gradle, we explicitly manage dependencies in a Project Object Model (POM) or gradle file, and link them using a central build artifact repository such as Nexus or Archiva.

This will ensure that the versions are managed correctly. The final executables will be packaged as a war file or an executable jar file embedding all dependencies

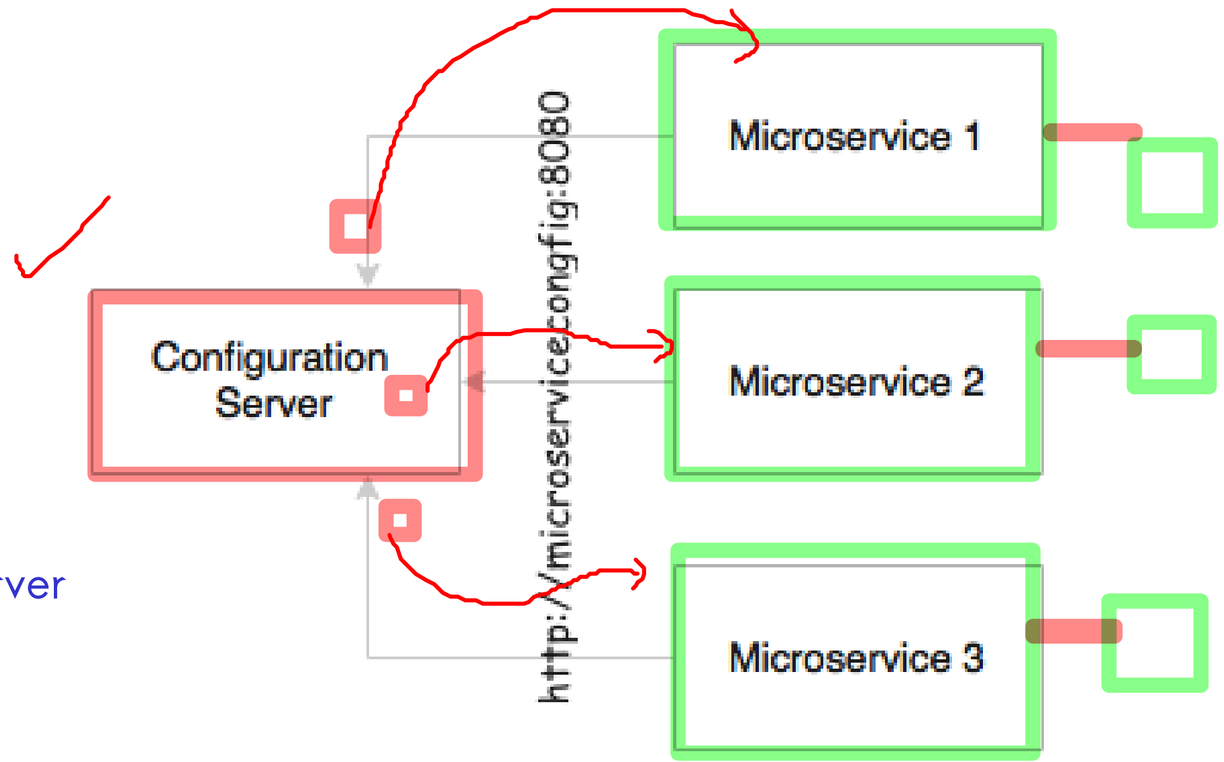


rgupta.mtech@gmail.com

Twelve-Factor Apps

Externalizing configurations

Microservices configuration parameters should be loaded from an external source.
This will also help you automate the release and deployment process as the only change between these environments are the configuration parameters

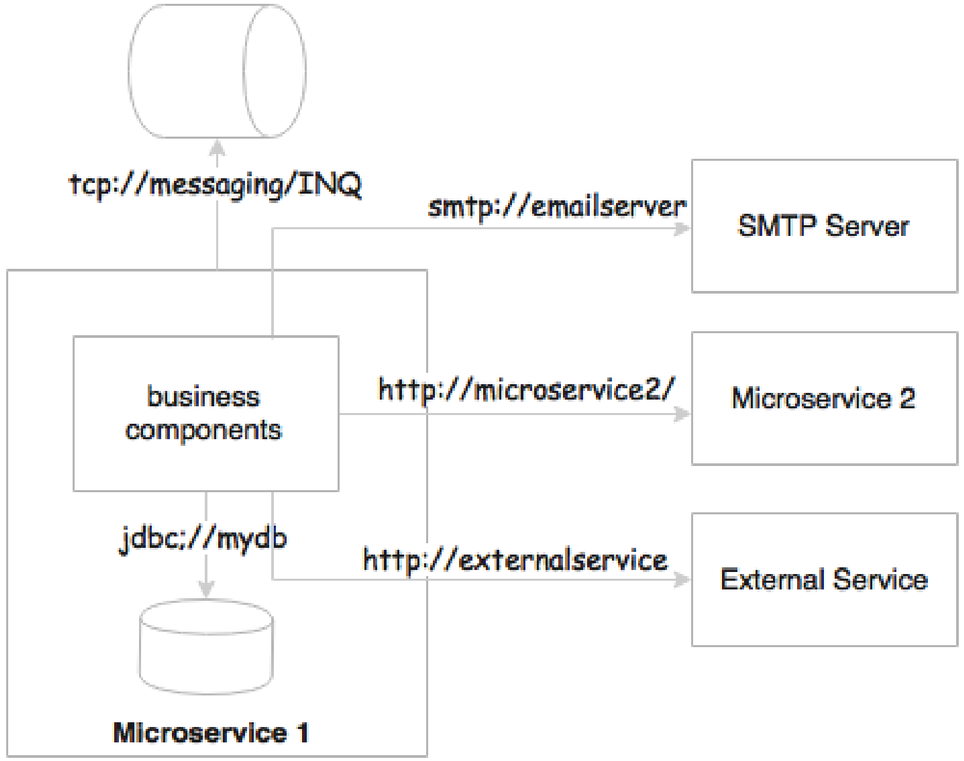


Spring cloud config server

rgupta.mtech@gmail.com

Backing services are addressable

All backing services should be accessible through an addressable URL. All services need to talk to some external resources during the life cycle of their execution. For example, they could be listening to or sending messages to a messaging system, sending an email, or persisting data to a database. All these services should be reachable through a URL without complex communication requirements



rgupta.mtech@gmail.com

Twelve-Factor Apps

Isolation between build, release, and run

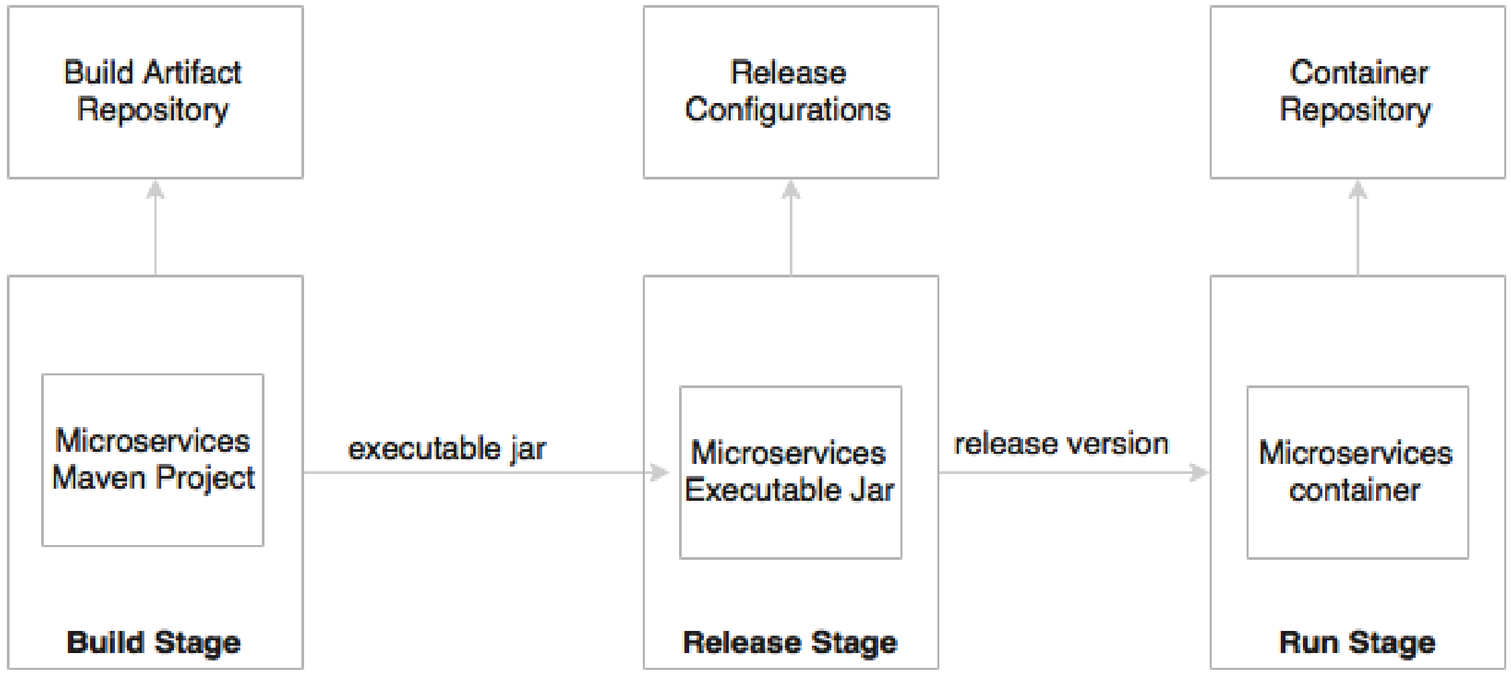
This principle advocates strong isolation between the build stage, the release stage, and the run stage.
The build stage refers to compiling and producing binaries by including all assets required.
The release stage refers to combining binaries with environment-specific configuration parameters.
The run stage refers to running applications on a specific execution environment.

The pipeline is unidirectional. Hence, it is not possible to propagate changes from run stages back to the build stage.
Essentially, it also means that it is not recommended to do specific builds for production; rather, it has to go through the pipeline

rgupta.mtech@gmail.com

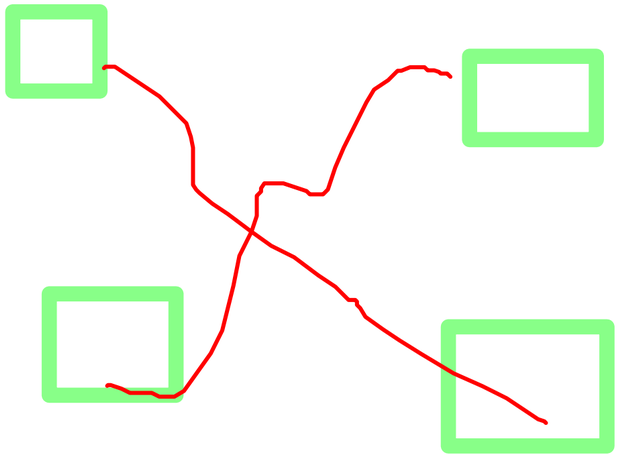
Isolation between build, release, and run

In microservices, the build will create executable jar files, including the service runtime, such as the HTTP listener. During the release phase, these executables will be combined with release configurations, such as production URLs, and so on, and create a release version, most probably as a container like Docker. In the run stage, these containers will be deployed on production via a container scheduler.



Stateless, shared nothing processes

This principle suggests that processes should be stateless and share nothing. If the application is stateless, then it is fault tolerant, and can be scaled out easily.
All microservices should be designed as stateless functions. If there is any requirement to store a state, it should be done with a backing database or in an in-memory cache



rgupta.mtech@gmail.com

Expose services through port bindings

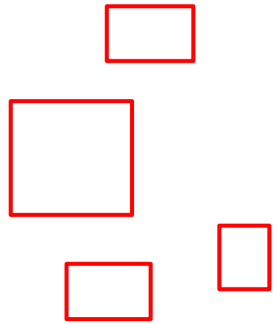
A Twelve-Factor App is expected to be self-contained or standalone. Traditionally, applications are deployed into a server--a web server or an application server, such as Apache Tomcat or JBoss, respectively.

A Twelve-Factor App ideally does not relay on an external web server. A HTTP listener, such as Tomcat, Jetty, and more, has to be embedded in the service or application itself.

Port binding is one of the fundamental requirements for microservices to be autonomous and self-contained. Microservice embeds the service listeners as a part of the service itself.

rgupta.mtech@gmail.com

Twelve-Factor Apps



Concurrency for scale out

The concurrency for scale out principle states that processes should be designed to scale out by replicating the processes. This is in addition to the use of threads within the process

Disposability, with minimal overhead

In a cloud environment targeting auto scaling, we should be able to spin up a new instance quickly. This is also applicable when promoting new versions of services.

Development, production parity

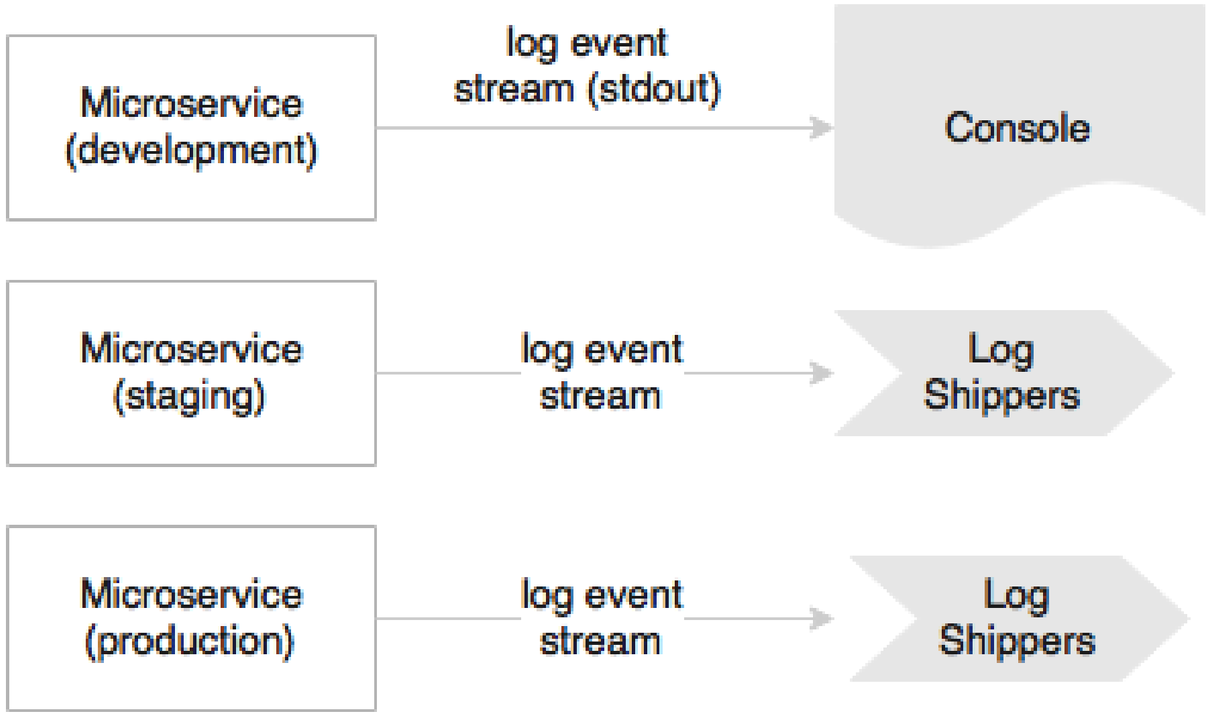
importance of keeping the development and production environments as identical as possible

rgupta.mtech@gmail.com

Externalizing logs

ELK

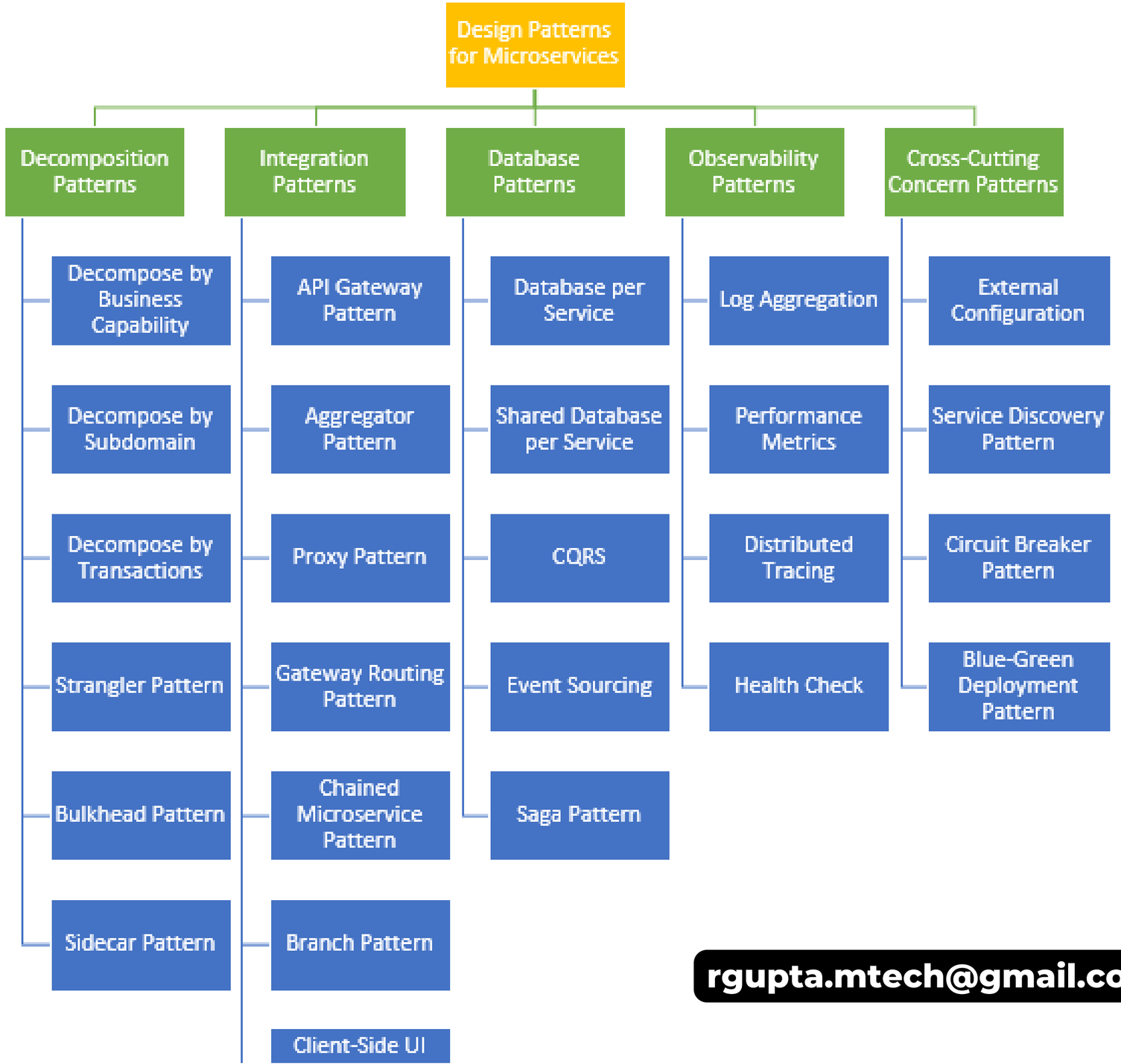
A Twelve-Factor App never attempts to store or ship log files. In a cloud, it is better to avoid local I/Os or file systems. If the I/Os are not fast enough in a given infrastructure, they could create a bottleneck. The solution to this is to use a centralized logging framework. Splunk, greylog, Logstash, Logplex, Loggly are some examples of log shipping and analysis tools



Module 3: Introduction to Microservice Design patterns

rgupta.mtech@gmail.com

Microservice Design patterns



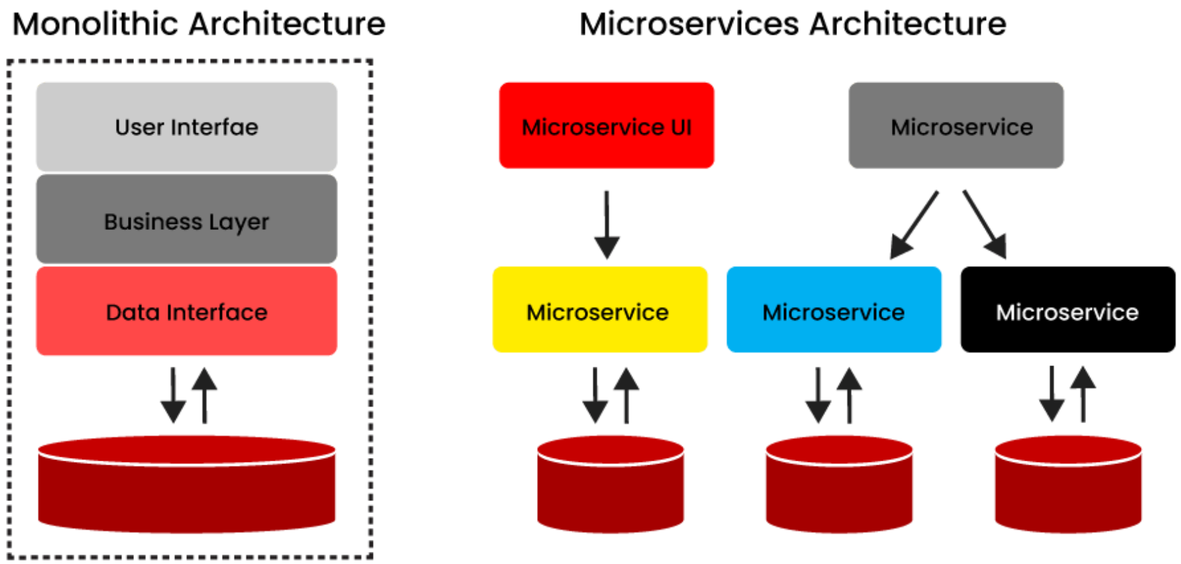
rgupta.mtech@gmail.com

Need of Microservice Design patterns

microservices are application are distributed applications, MSA creates challenges such as managing

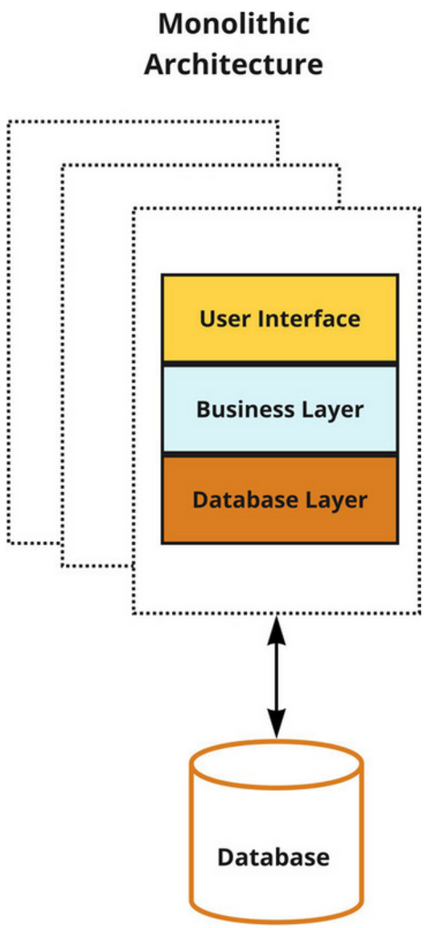
multiple instances of microservices, communication between microservices, transactional databases, monitoring, debugging, tracing and testing challenges, and so on.

Microservice design patterns help to solve these real-problem statements and use cases.



rgupta.mtech@gmail.com

Monolith Problems

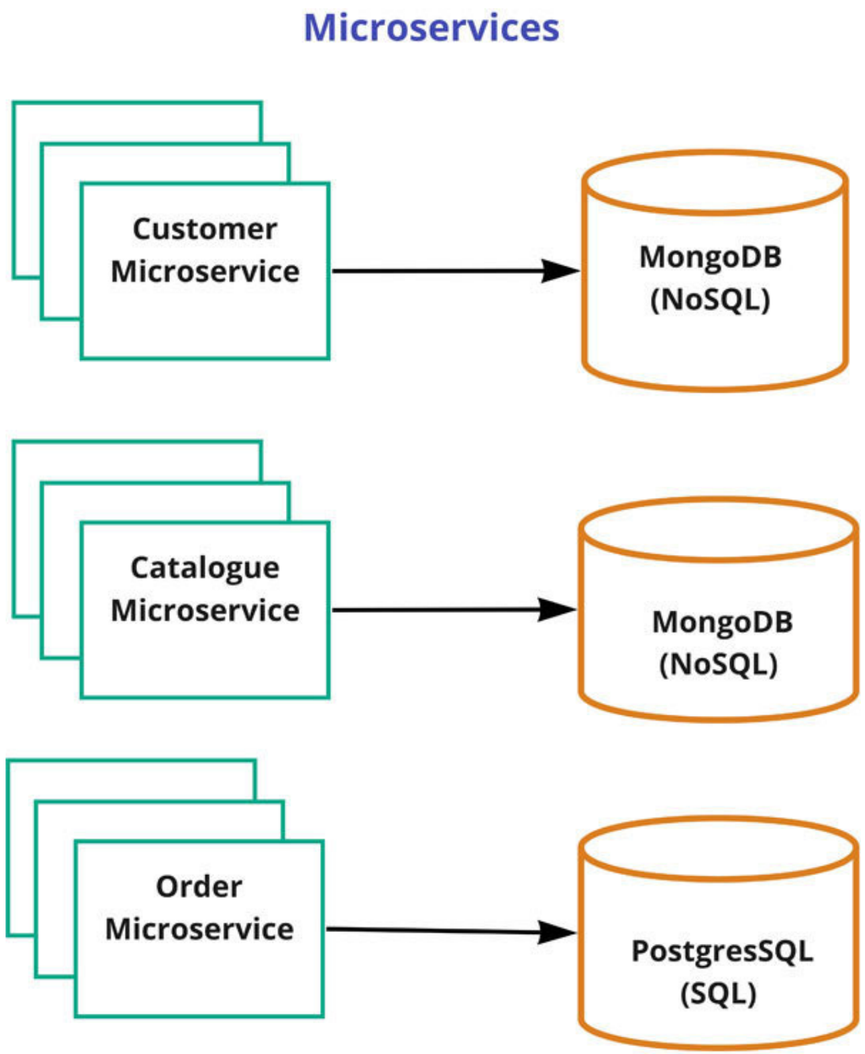


- Hard to change
- Integration issues
- Long testing and release cycle
- Hard to scale infrastructure
- Complex DevOps CI/CD cycle
- Not reliable
- Not highly available
- Not resilient
- Hard to adopt new technologies trends
- Difficult to achieve operational agility

Big ball of a black box, hard to understand, and debug the codeThere are workarounds and solutions to these challenges; however,
this model doesn't comply with the twelve-factor/fifteen-factor cloud native modern applications for containers.

rgupta.mtech@gmail.com

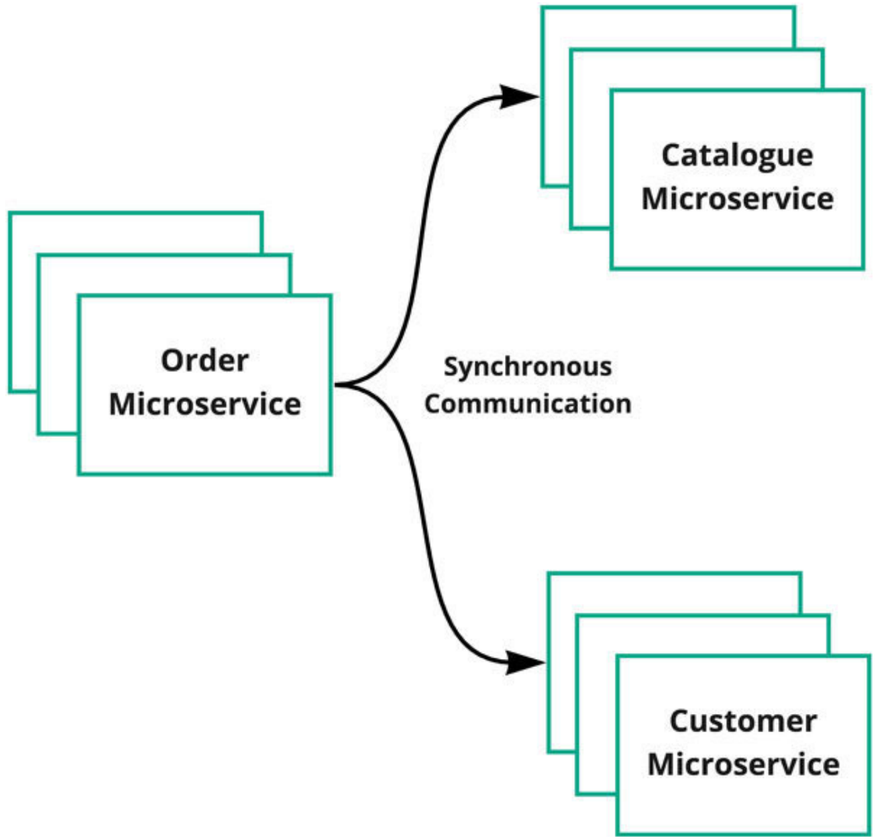
Microservices



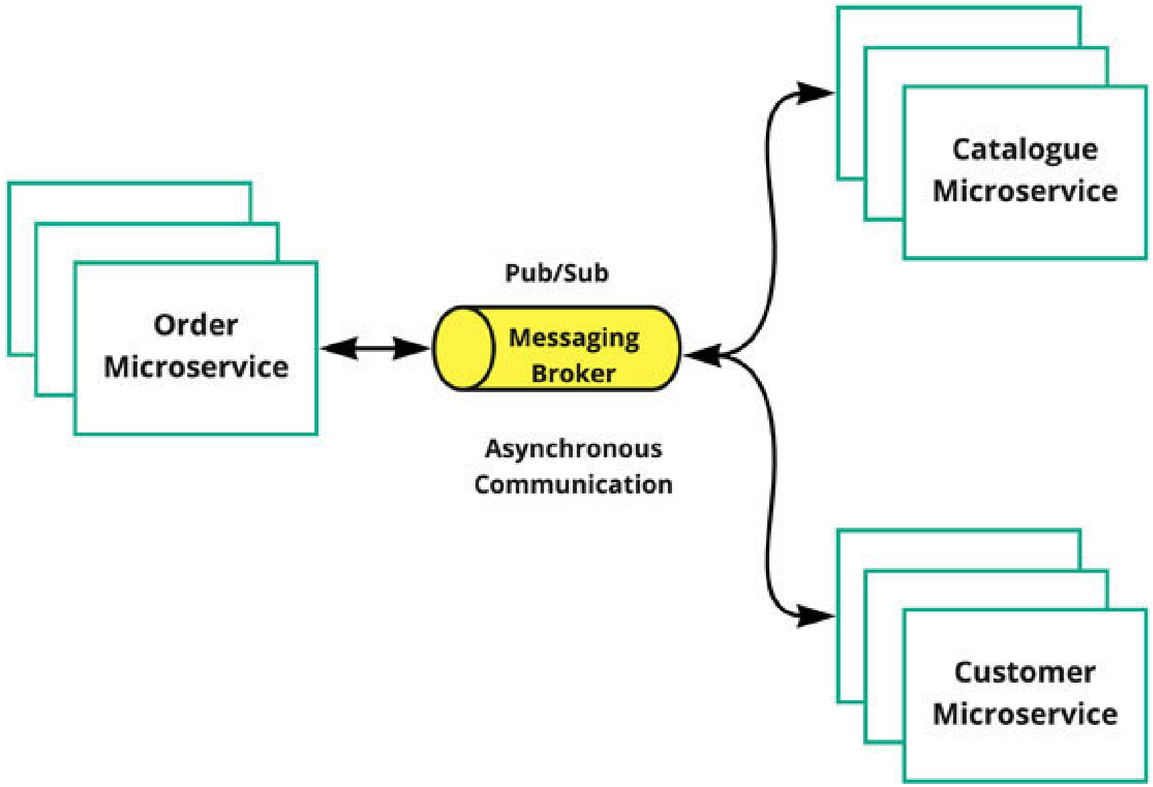
rgupta.mtech@gmail.com

Microservices Communications

Synchronous



Asynchronous

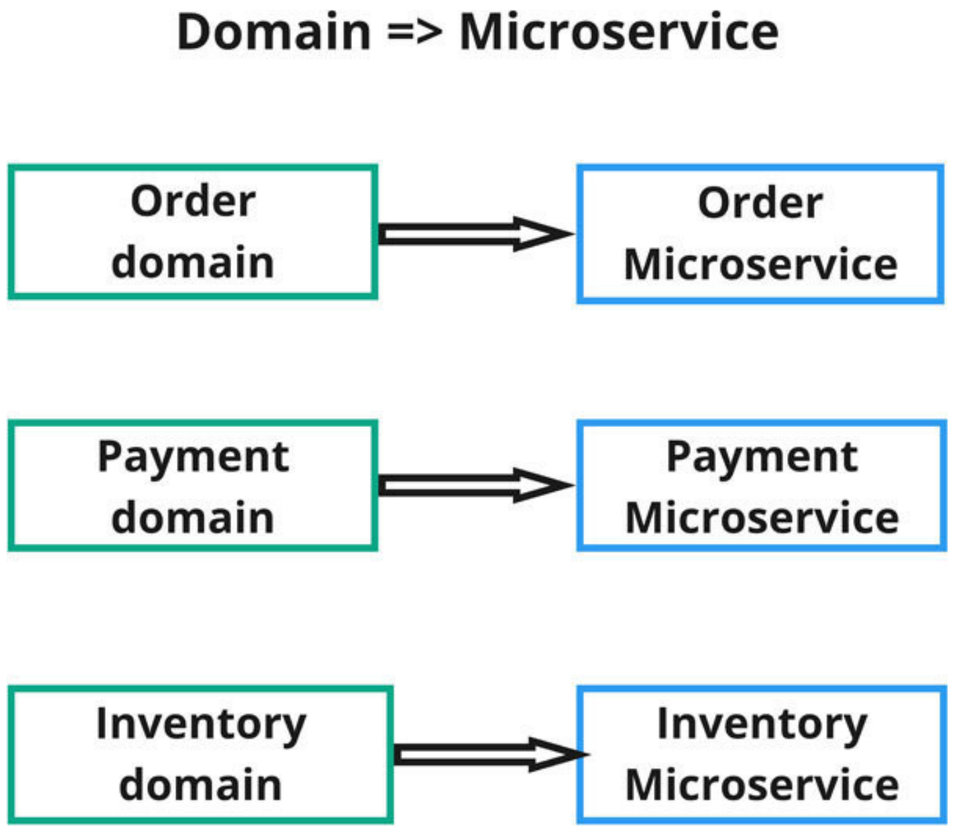


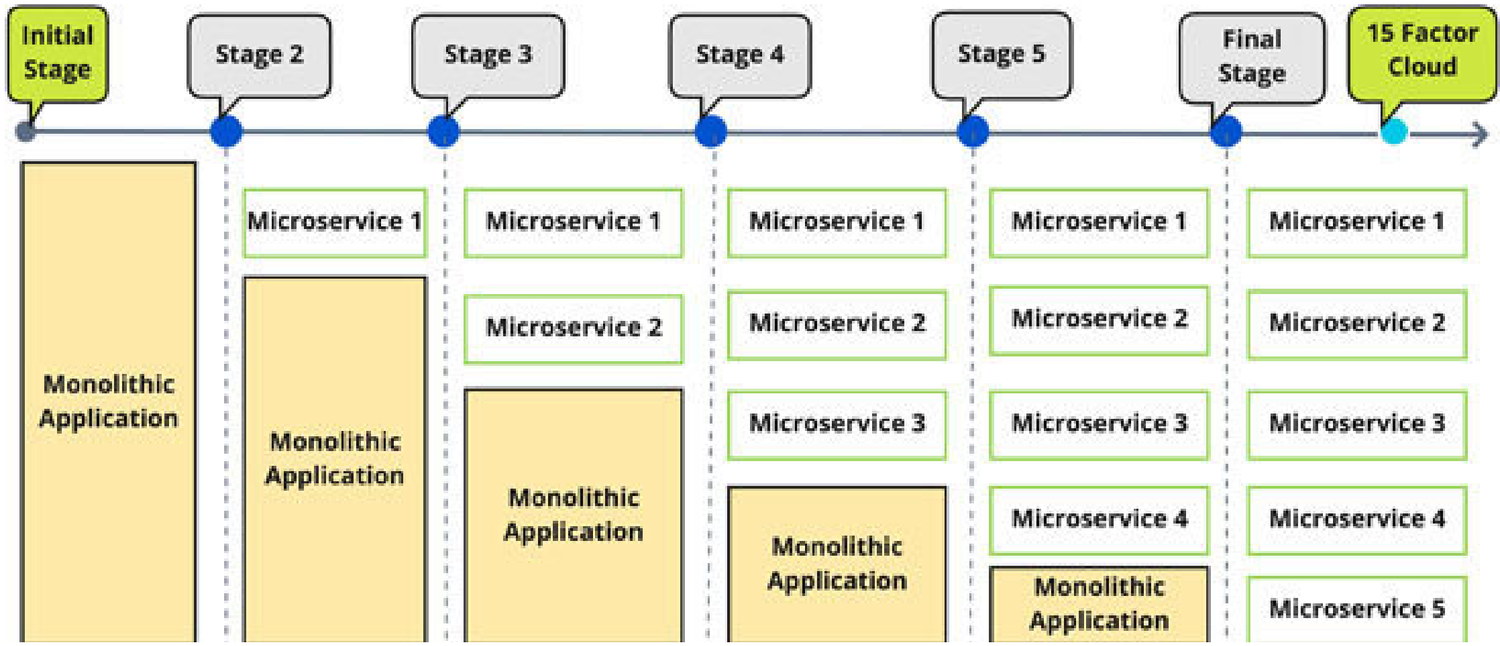
rgupta.mtech@gmail.com

Decomposition of microservices

These are some ways of decomposing microservices:

- Domain Driven Design
- Strangler pattern





rgupta.mtech@gmail.com

Integration patterns

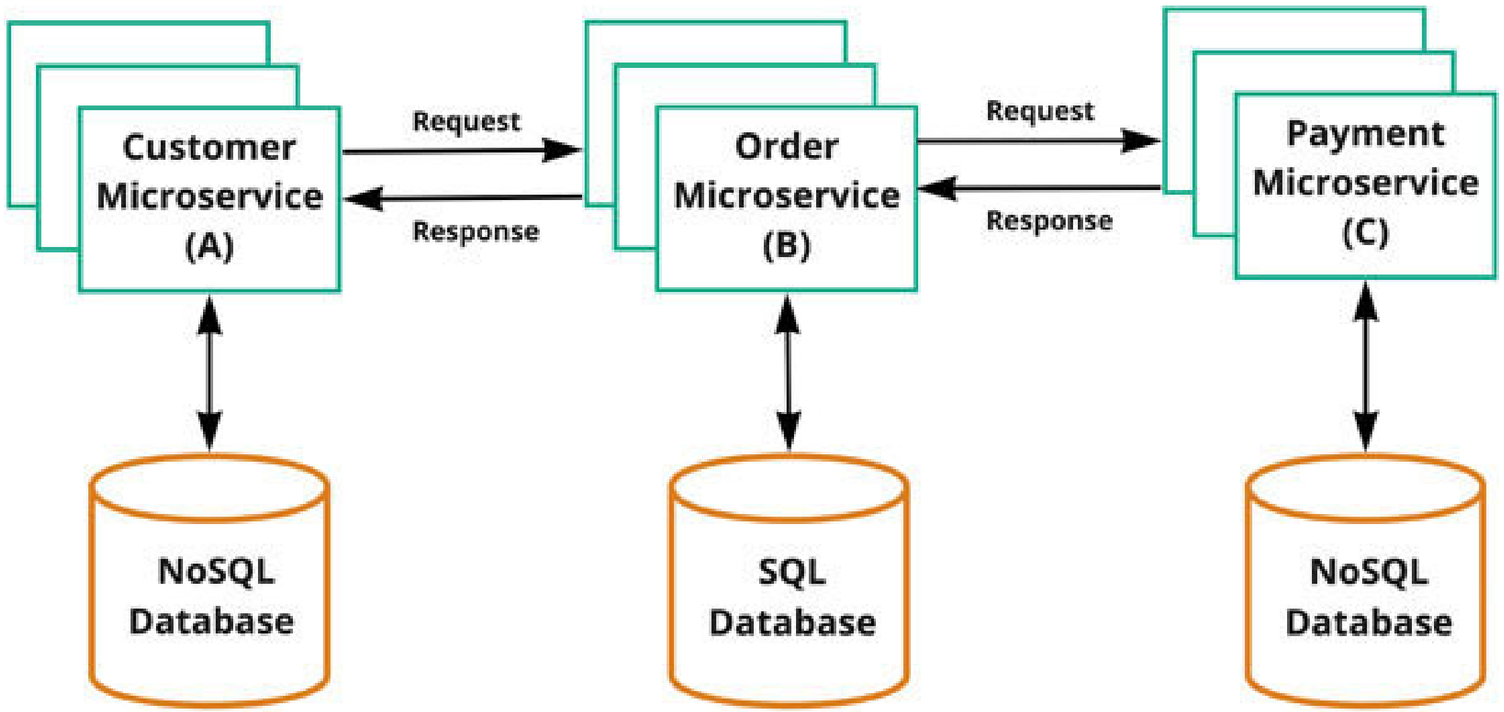
In the microservices architecture, all services are distributed and deployed on different servers and containers. They all communicate with each other. We will discuss a few important integration design patterns:

- Chain of responsibility
- Aggregator and branch design patterns
- API gateway design pattern
- Micro frontends (UI composition) pattern

rgupta.mtech@gmail.com

Chain of responsibility

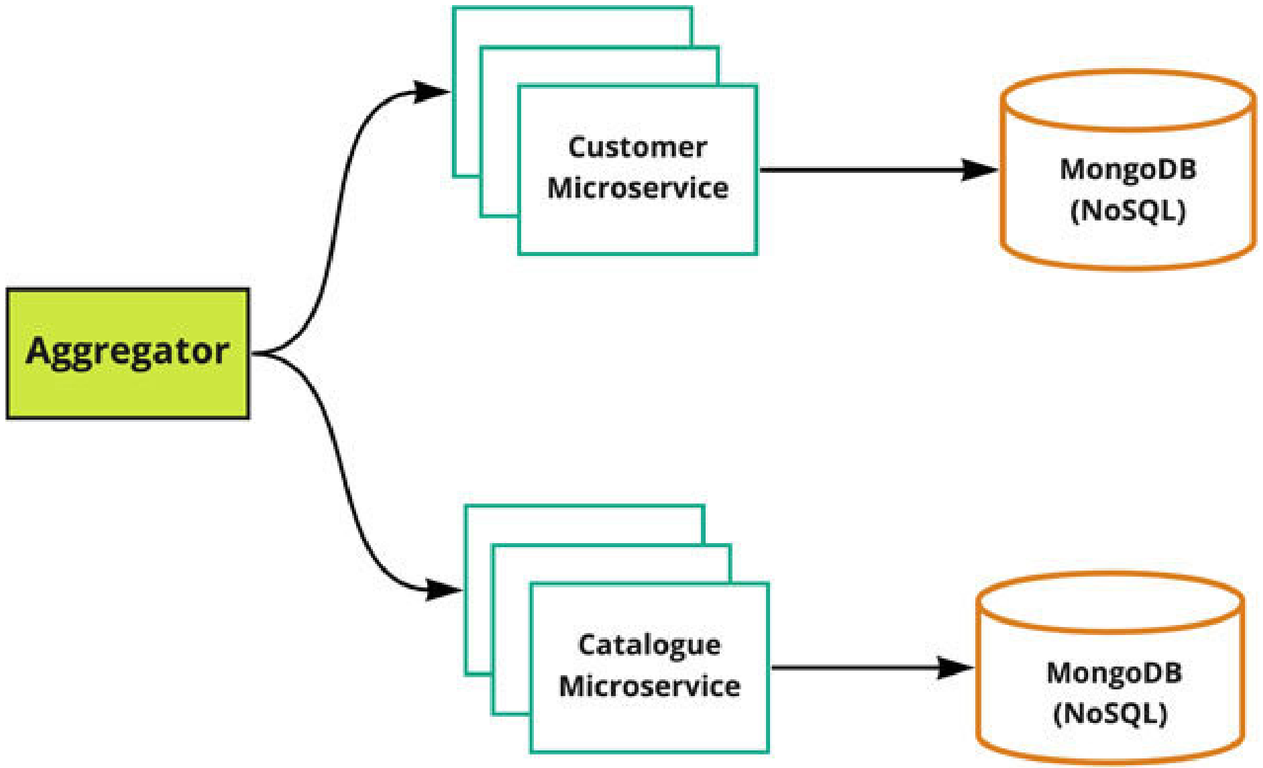
All microservices call each other in a chain or a given sequence to complete a use case end to end and produce a single response



rgupta.mtech@gmail.com

Aggregator and branch design patterns

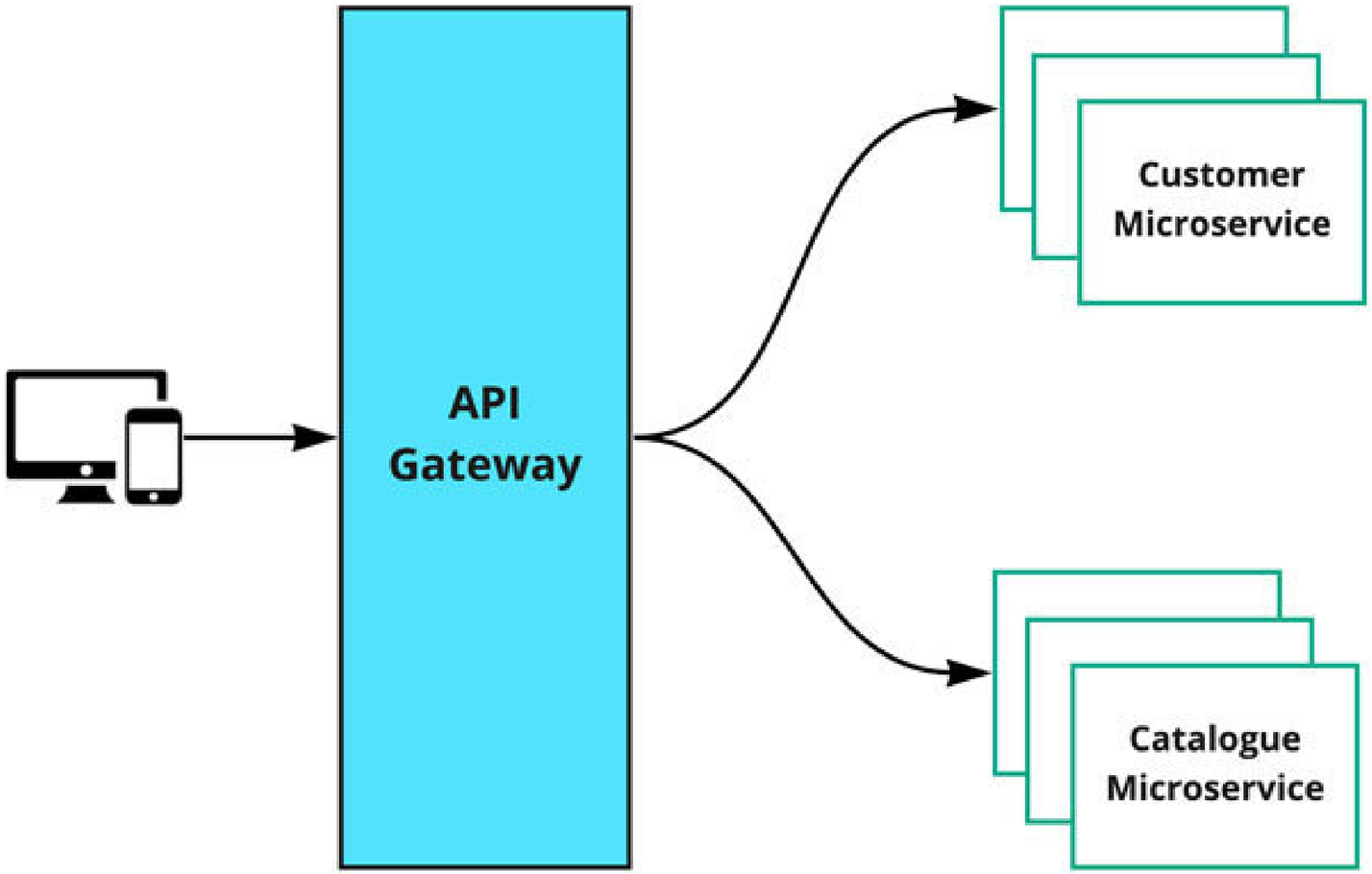
In this design pattern, two or more services work in parallel and combine their requests and responses at the end. So, unlike chain of responsibility design patterns this is not sequential and works in a sequence/chain. Parallel processing is possible, which creates advantages on top of the chain of responsibility design pattern



rgupta.mtech@gmail.com

API gateway design pattern

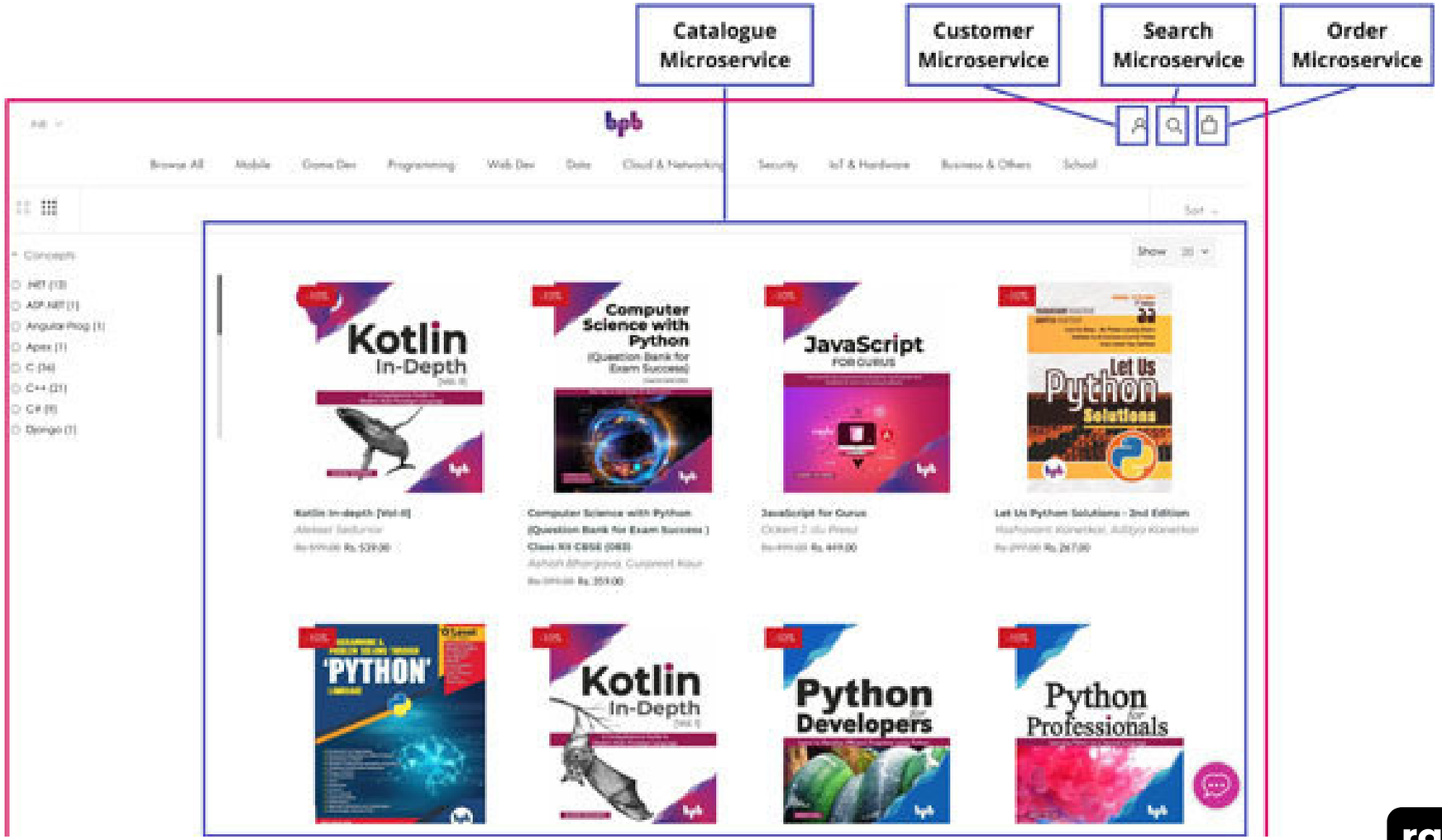
It’s deployed separately as a separate orchestrator application.Clients and other microservices first interact with the API gateway; it has all routing and filtering business logic which routes the client request to the appropriate microservice REST API for further processing.
It also manages multiple workloads from different kinds of clients like web, mobile, IoT, and so on.



rgupta.mtech@gmail.com

Micro frontends (UI composition) pattern

This design pattern is designed for front end/UI applications to display the API response data from different microservices. In this design approach, a single page application application is split into smaller tiles/sections which shows data from different backend microservices



rgupta.mtech@gmail.com

Integration patterns

When we plan about cloud native applications, it doesn't only mean modernize applications. It's also meant to modernize legacy databases.
At present, there are multiple SQL and NoSQL-based applications, which can be chosen based on the business use cases and type of structured and unstructured data.
Earlier, we had only VM supported databases. Now, databases can be also installed on containers which are scalable and highly available
There are multiple database patterns which are suitable for the microservice architecture:

There are multiple database patterns which are suitable for the microservice architecture:

- Database per service
- Shared database
- Command Query Responsibility Segregation
- SAGA design patterns
- Orchestrating microservices
- Choreography microservices
- Hybrid communication

rgupta.mtech@gmail.com

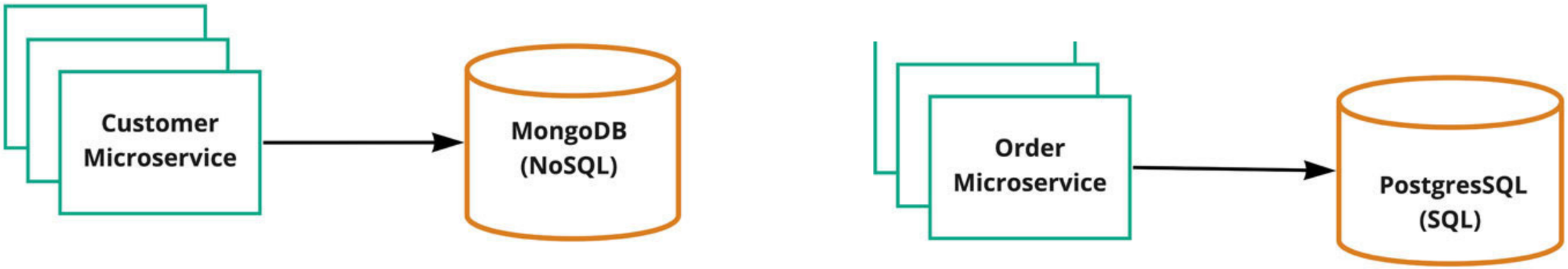
Database per service

In the microservice architecture, every microservice should have its own database and they work in isolation; however, during migration of brownfield legacy apps, monolithic and microservices might share the same database.

Every database has a separate requirement of memory, CPU speed, avoid duplication of data, and so on.

It's advisable to have a separate database per microservice to only store relevant data for that service, which makes it easy to manage, deploy, and upgrade in a container environment for the lightweight microservices.

The following diagram depicts that Customer and Order microservices have their own databases:



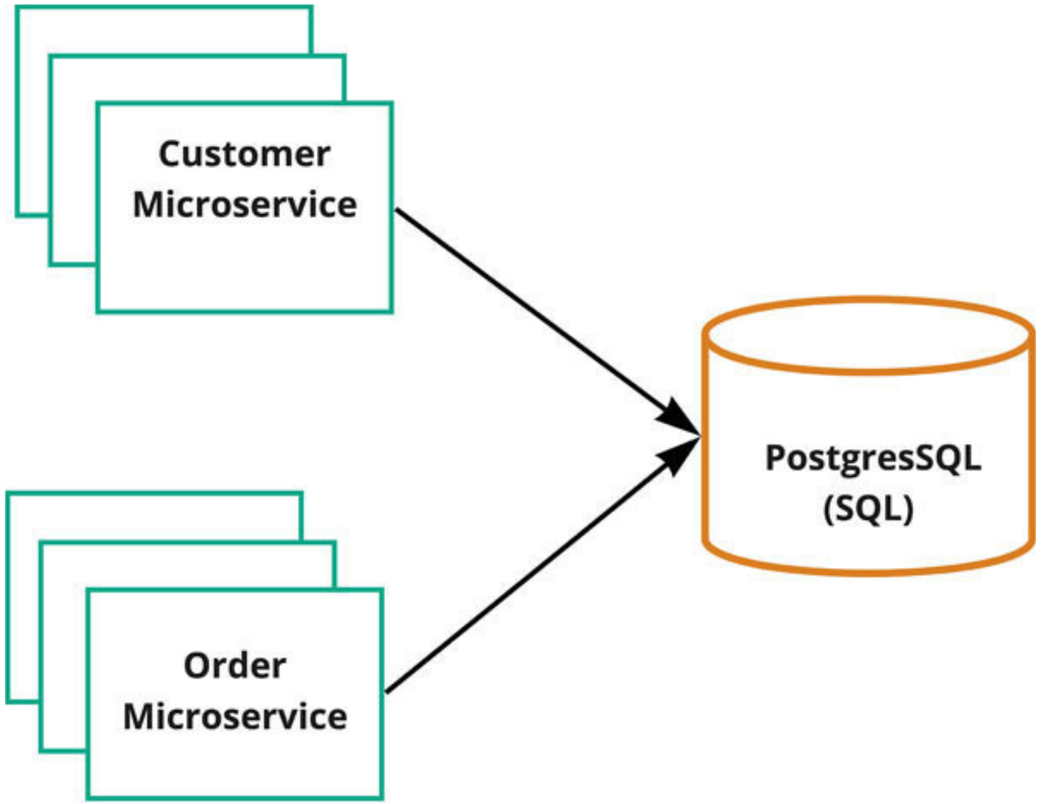
It doesn't support transactions, where commit happens in two-phases or where multiple cross tables are interconnected.

Doesn't fully support the CAP theorem for ACID transactions.

rgupta.mtech@gmail.com

Shared database

In some common use cases, where the data type is also the same, multiple microservices share the same database.
This design pattern is good for a smaller number of microservices from two to four, where they share common database tables from a single database with similar structural data type.
The following diagram depicts Customer and Order microservices sharing the same database:



Use cases

To maintain the ACID statement among a few similar microservices.Some monolithic applications can't directly split their database per service during the migration phase.

Initially, monolithic apps migrated to microservices, but they share the same database.

In later phases, shared databases will be broken to a single database per service.

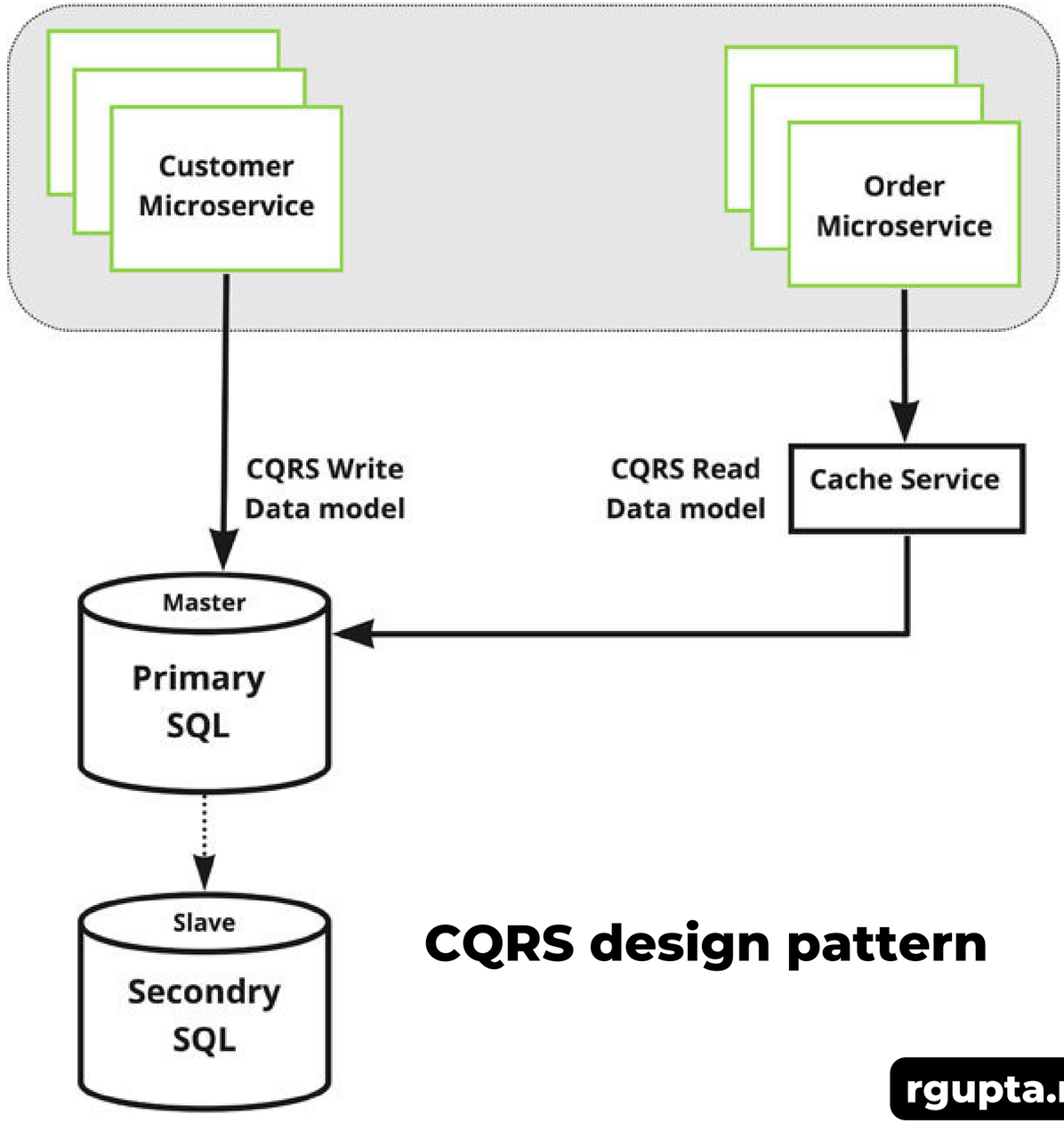
rgupta.mtech@gmail.com

Command Query Responsibility Segregation (CQRS)

It's a database pattern where the command (write) and query (read) have their separate databases and separate responsibilities.

CQRS is divided into two parts:

- delete
- Read query



CQRS design pattern

rgupta.mtech@gmail.com

Command Query Responsibility Segregation (CQRS)

- It makes separation of read and write for complex business use cases.
- In the cloud native microservice architecture, database tables are mapped with services based on one database table per data domain model like order service has order database tables.
- There could be many one-to-one database tables per service. Inmost of the business use cases, query can be built by joining many database tables
- In the CQRS pattern, the model will have the responsibility of inserting, updating, and deleting data from the database.
- Command has the sole responsibility of the database Create,Update, Delete action and queries are methods that are able to read and return data without any modification.
- CQRS defines a new standard for microservices to have separate read and write data models.
- These two different models are managed independently.
- It solves many complex business use cases where intensive query on multiple database tables is required

rgupta.mtech@gmail.com

SAGA design patterns

This pattern guarantees data consistency across multiple microservices in a distributed environment. Traditional monolithic applications supported two-phase commit of RDBMS databases.

- **2PC doesn’t support and guarantee data consistency across multiple databases for multiple microservices.**
- **The SAGA design pattern is about implementing transaction rollback and compensation controls to guarantee consistency in a series of atomic transactions that constitute a business process.**

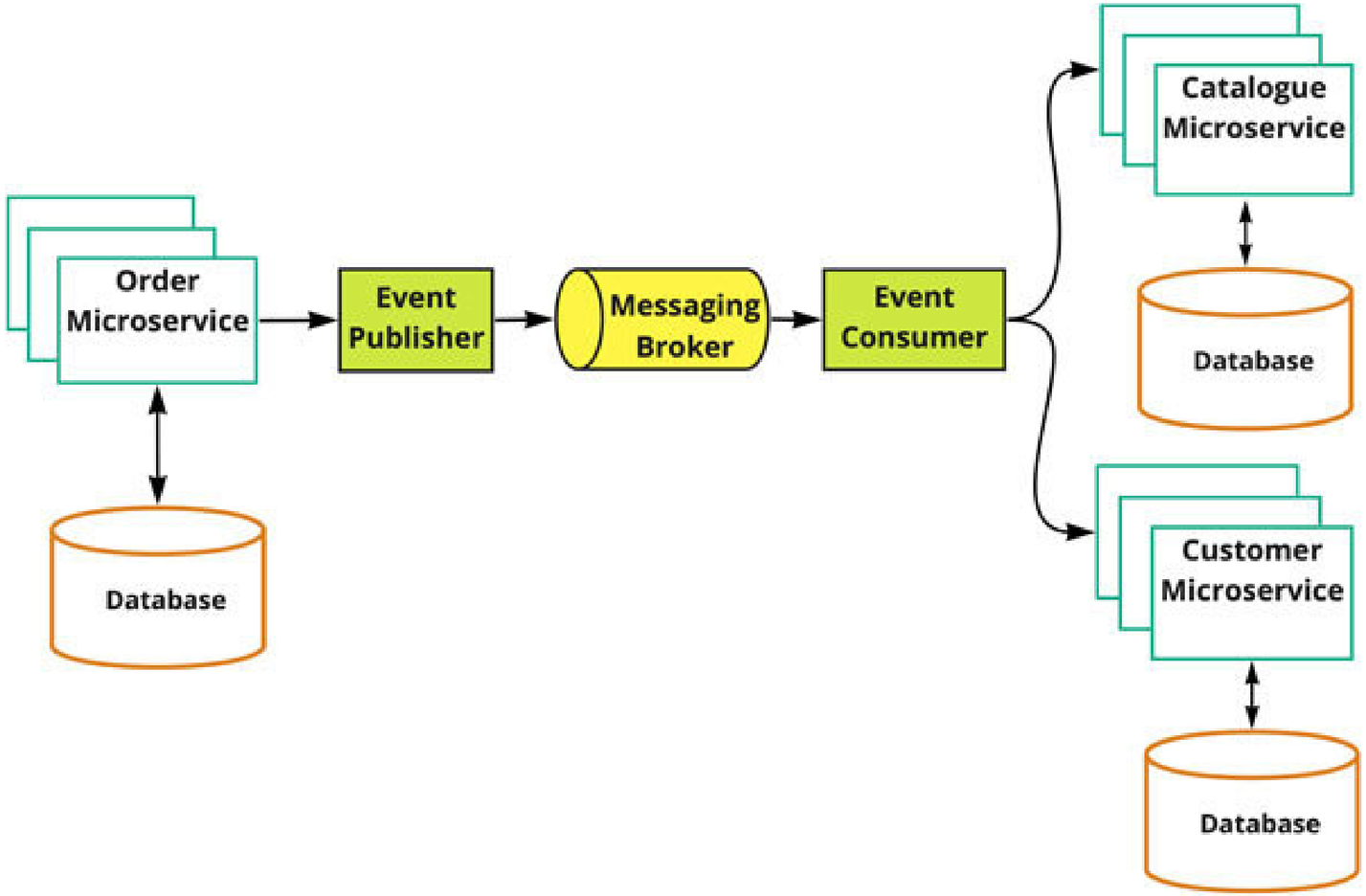
rgupta.mtech@gmail.com

Event-driven architecture

An event is an occurrence or change or a short-lived task in a microservice application/system. It's like push notification services, or alerts, and so on.

Some of the examples of events are as follows:

- Order is added in cart
- Order is submitted
- Payment is confirmed
- order is delivered
- cab is booked



rgupta.mtech@gmail.com

Observability and monitoring patterns

In a cloud native microservices environment, it’s very tough and tedious to monitor hundreds and thousands of microservices and their scaled containers.
There are so many moving parts, so we need observability and monitoring patterns to manage these humongous microservices traffic and their inter-communications.

Some of the examples of events are as follows:

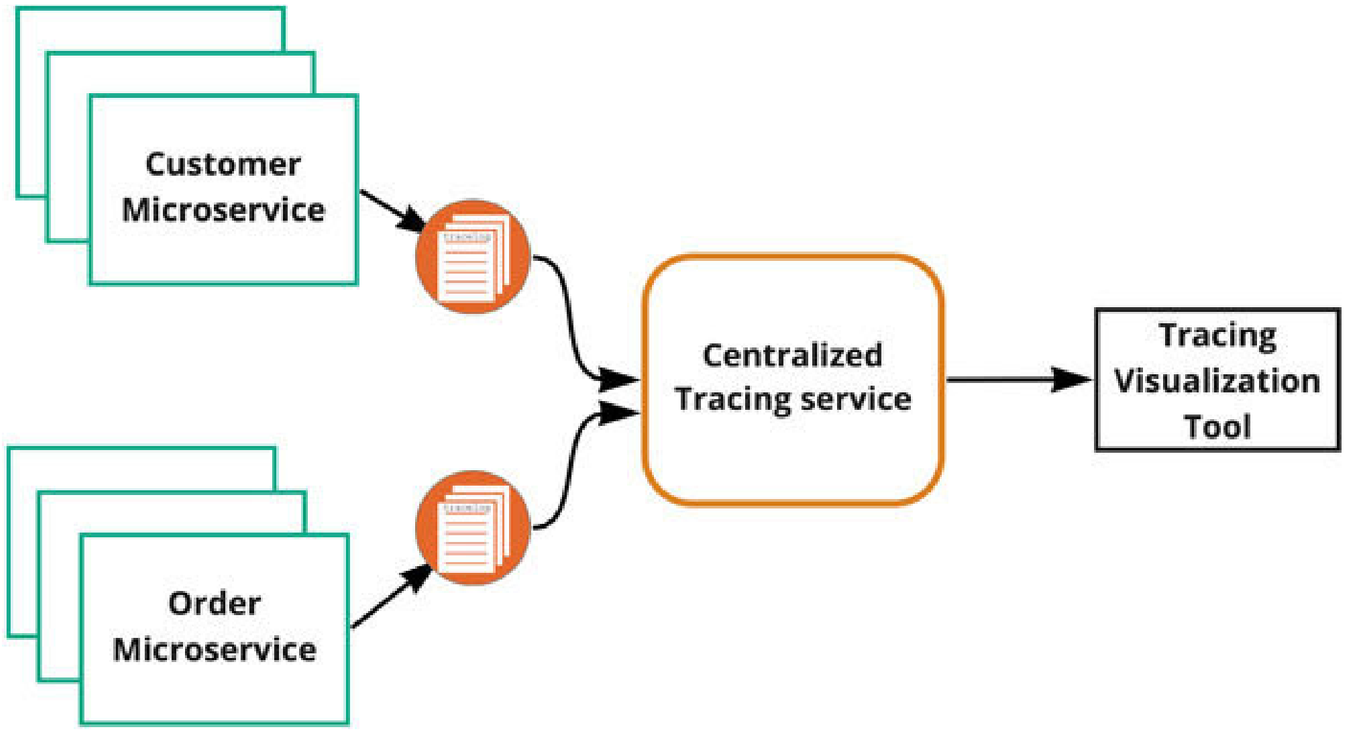
- Distributed tracing
- Log aggregation
- Health check API
- Application metrics
- Service discovery design patterns
- Client-side discovery
- Server-side discovery

rgupta.mtech@gmail.com

Distributed tracing

When multiple microservices interact with each other for various business use cases, there may be a possibility of failure of services during this inter-communication, which can break the business flow and make it complex and tedious to identify and debug the issue.
Logging is just not enough; it's a very tedious process to read logs and identify issues when we have thousands of lines of logs of multiple microservices on different containers in a multi-cloud environment. In some cases, the same microservice is deployed on multiple clusters and data centers.

The following diagram has two microservices such as Customer and Order which create and persist tracing logs and push to the centralized tracing service:



Distributed tracing

We need a mechanism to manage, monitor, and debug the production issues.
This pattern is based on the REST API tracing to quickly track the issue of any culprit and buggy services.

In this pattern, client request and response logs for the microservices REST API are recorded and monitored asynchronously.

This tracing can be done by various techniques; common correlation ID is one the popular techniques where end-to-end API calls from the client/UI to backend services are done by a unique correlation ID.

Every external client request assigns a unique request/correlation ID which can be passed to all subsequent services. In this way, tracking can be done easily by persisting all request-response payload.

- **Microservices tracing for debugging production issues.**
- **Application performance monitoring using APM tools..**
- **Log tracing for apps, databases, message broker, Kubernetes container solutions, and other infrastructure systems.**

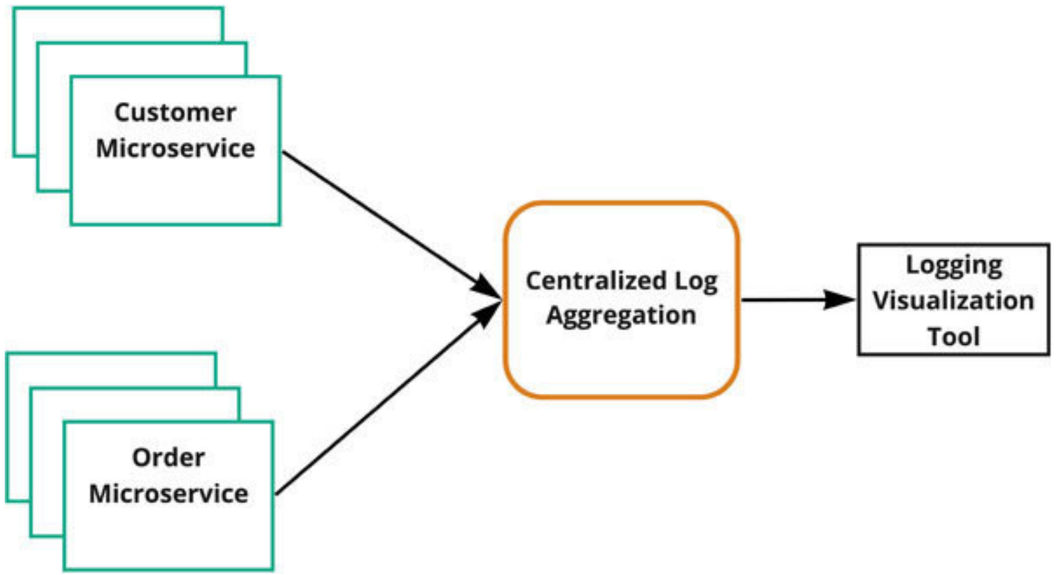
rgupta.mtech@gmail.com

Log aggregation

This is a pattern to aggregate logs at a centralized location. It’s a technique to collect logs from various microservices and other apps and persist at another location for query and visualization on log monitoring UI dashboards.

To complete business use cases, multiple microservices interact with each other on different containers and servers. During this interaction, they also write thousands of lines of log messages on different containers and servers.

It's difficult to analyse humongous end-to-end logs of business use cases on different servers for developers and DevOps operators.
Sometimes, it takes a couple of tedious days and nights to analyze logs and identify production issues, which may cause loss of revenue and customer’s trust, which is MOST important. Log aggregation and analysis are very important for any organization. Outage of applications can drop company's shares down.



There are multiple log aggregation solutions like ELF, EFK, Splunk, APM, and some other enterprise APM tools.

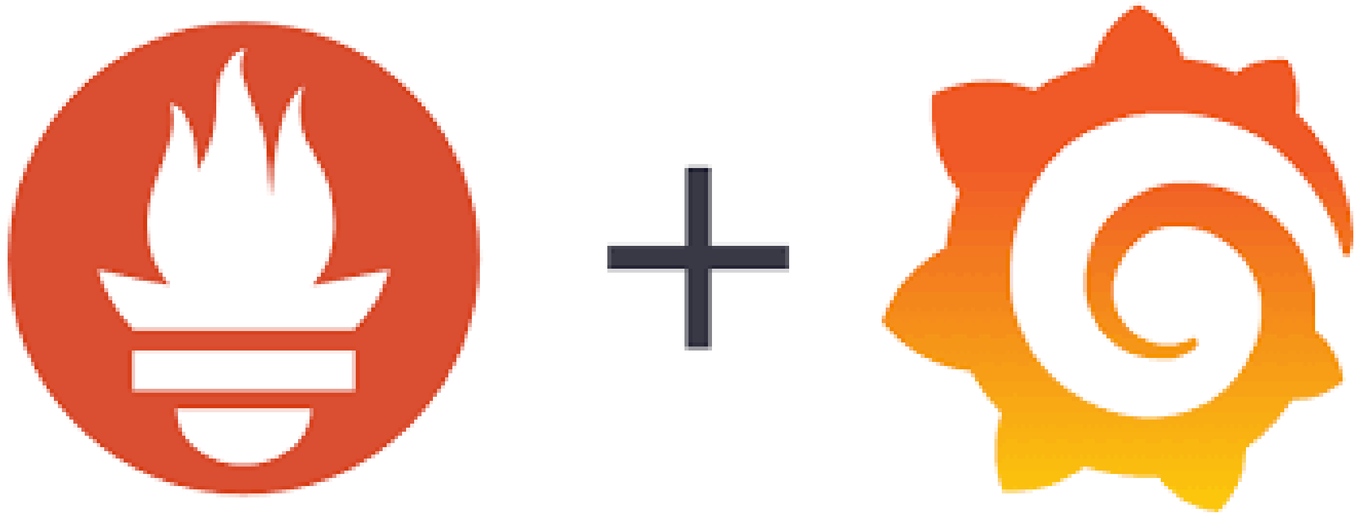
rgupta.mtech@gmail.com

Application metrics

This is a way to check application metrics, their performance,audit logs, and so on.

It’s a measure of microservices applications/REST APIs characteristics which are quantifiable or countable.
It helps to check the performance of the REST API like how many requests an API is handling per second and what’s the response time. It helps to scale the application and provide faster applications to web/mobile clients.
There are many tools available to check matrices of applications like Spring Boot Micrometer, they work either with push or pull models using REST APIs.

For example, Grafana pulls metrics of the applications by using the integrated Prometheus REST API handler and visualizes on the Grafana dashboard.



rgupta.mtech@gmail.com

Service discovery design pattern

It’s easy to inter-communicate with monolithic application services because they are all bundled in the same application and deployed as a bundled service on the same server. So, discovery of services is easy in monolithic applications.
In the microservice architecture, when many microservices are running in a distributed environment on multiple containers/servers.
It’s a complex process to discover and communicate with each other because they may have different dynamic IP addresses, when they scale and create multiple containers for the same app.

Sometimes, the same microservice application is running at multiple places in multiple containers, clusters, or VMs servers. There should be a way to register all services at one centralized place, which helps client requests to discover and redirect to REST API services in a distributed environment

This pattern is divided into two parts:

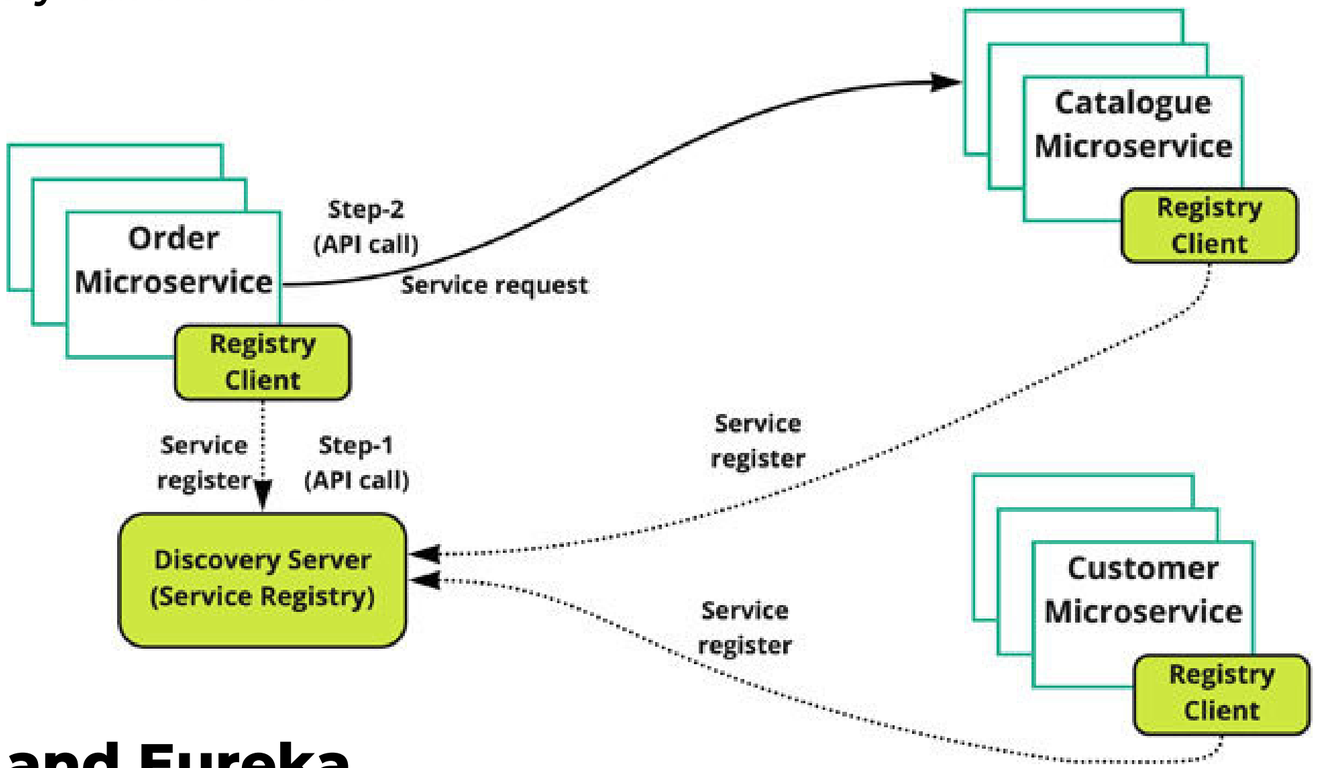
- **Client-side discovery**
- **Server-side discovery**

rgupta.mtech@gmail.com

Client-side discovery

This discovery happens at the client side. In this pattern, all microservices are registered first with the discovery server, where all services are listed with server details. In this pattern, there will be two API calls.
First, the client request will connect to the discovery server, get server details to the server which it wants to be connected to, and then call the actual API service directly in the second call.
In the following diagram, all three microservices have their own registry clients. They all are registered to the discovery server. The order microservice will check the Catalogue microservice server details with the first API call to the discovery server and then make a second API call to the Catalogue service.

This discovery is at client side:

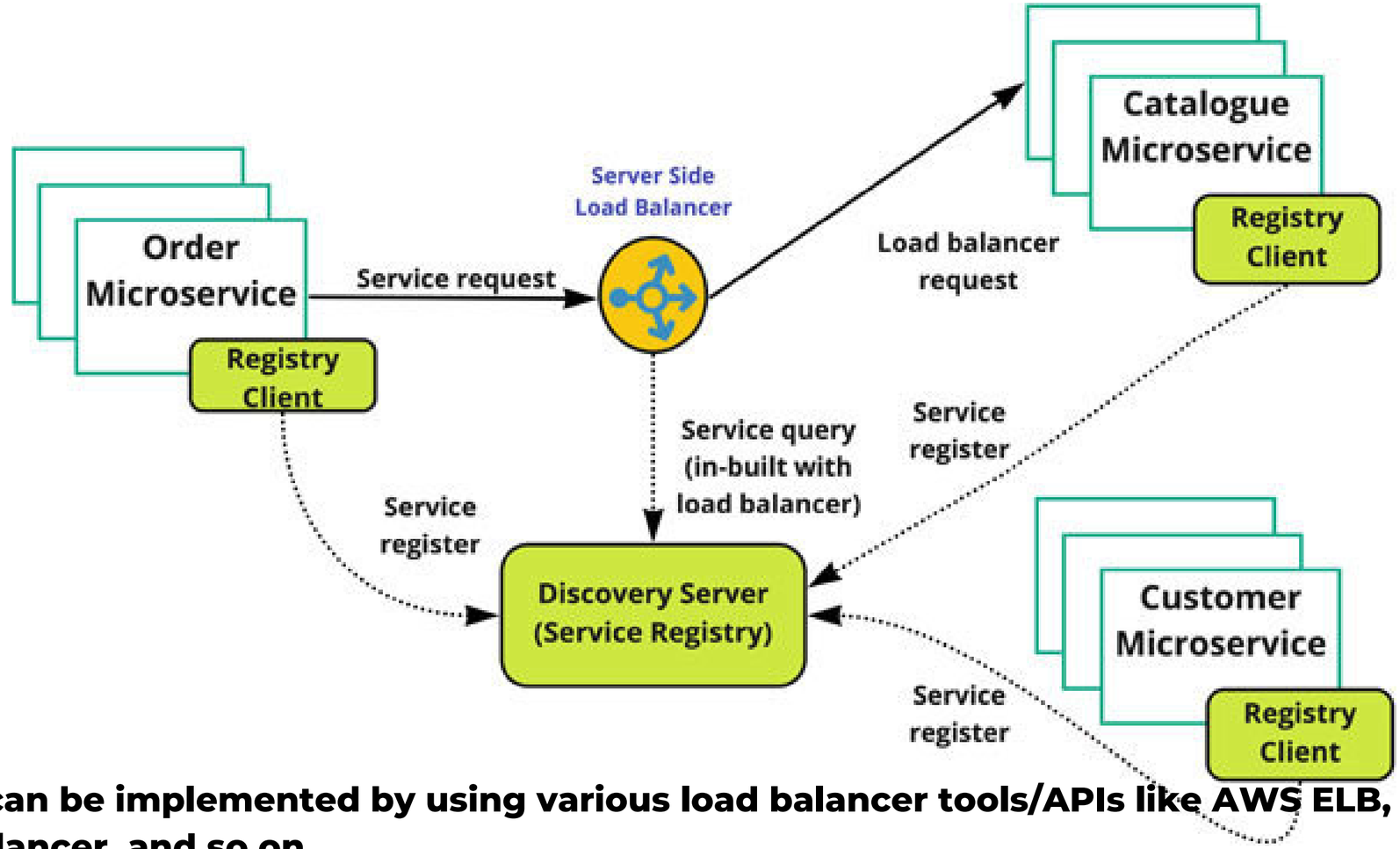


Consul and Eureka

rgupta.mtech@gmail.com

Server-side discovery

This server-side discovery happens at the backend infrastructure. In this case, all microservices are registered at the server-side registry. The load balancer is a good example which forwards client's requests to register services with it. Microservices can be easily connected with each other by using a single API call.



It can be implemented by using various load balancer tools/APIs like AWS ELB, Google Load Balancer, and so on.

There are other server-side container orchestration tools like Kubernetes, which manage this with their own load balancers at the containers end.

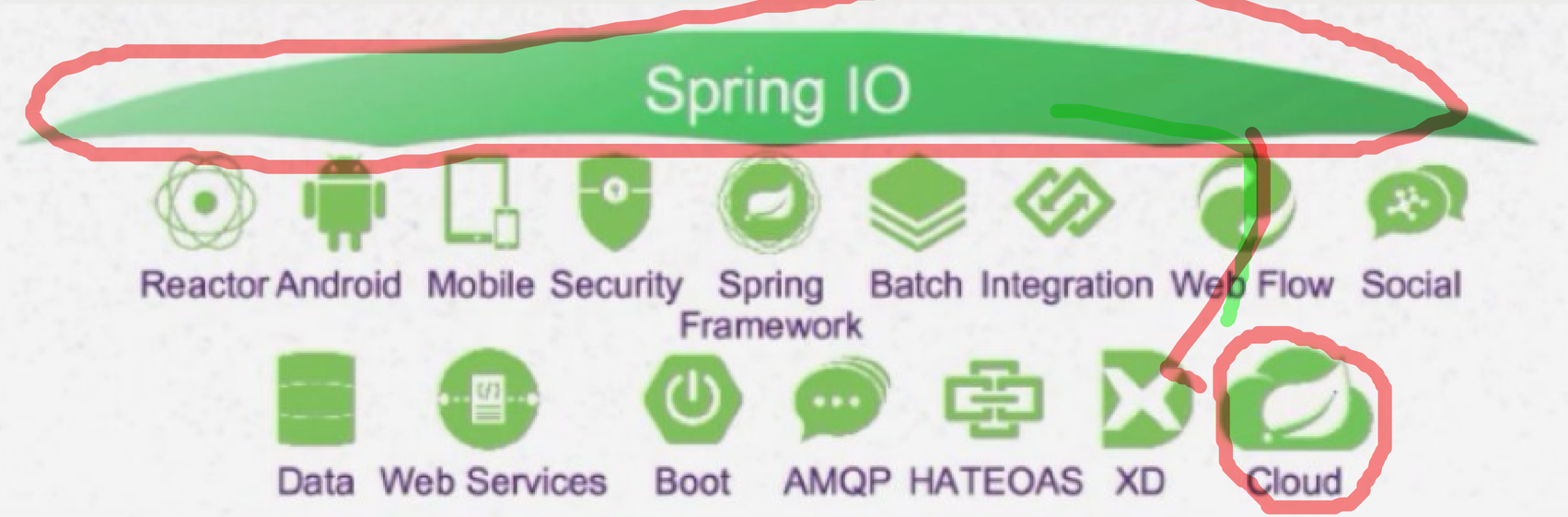
rgupta.mtech@gmail.com

Module 4: Introduction to Spring cloud

rgupta.mtech@gmail.com

Introduction to Spring cloud

Spring is a platform built for developing web applications in the **Java** language. It was first introduced in 2004. In 2006, sub-projects appeared. Each sub-project focuses on a different field. Till now, you can see the sub-projects listed as the following illustration.

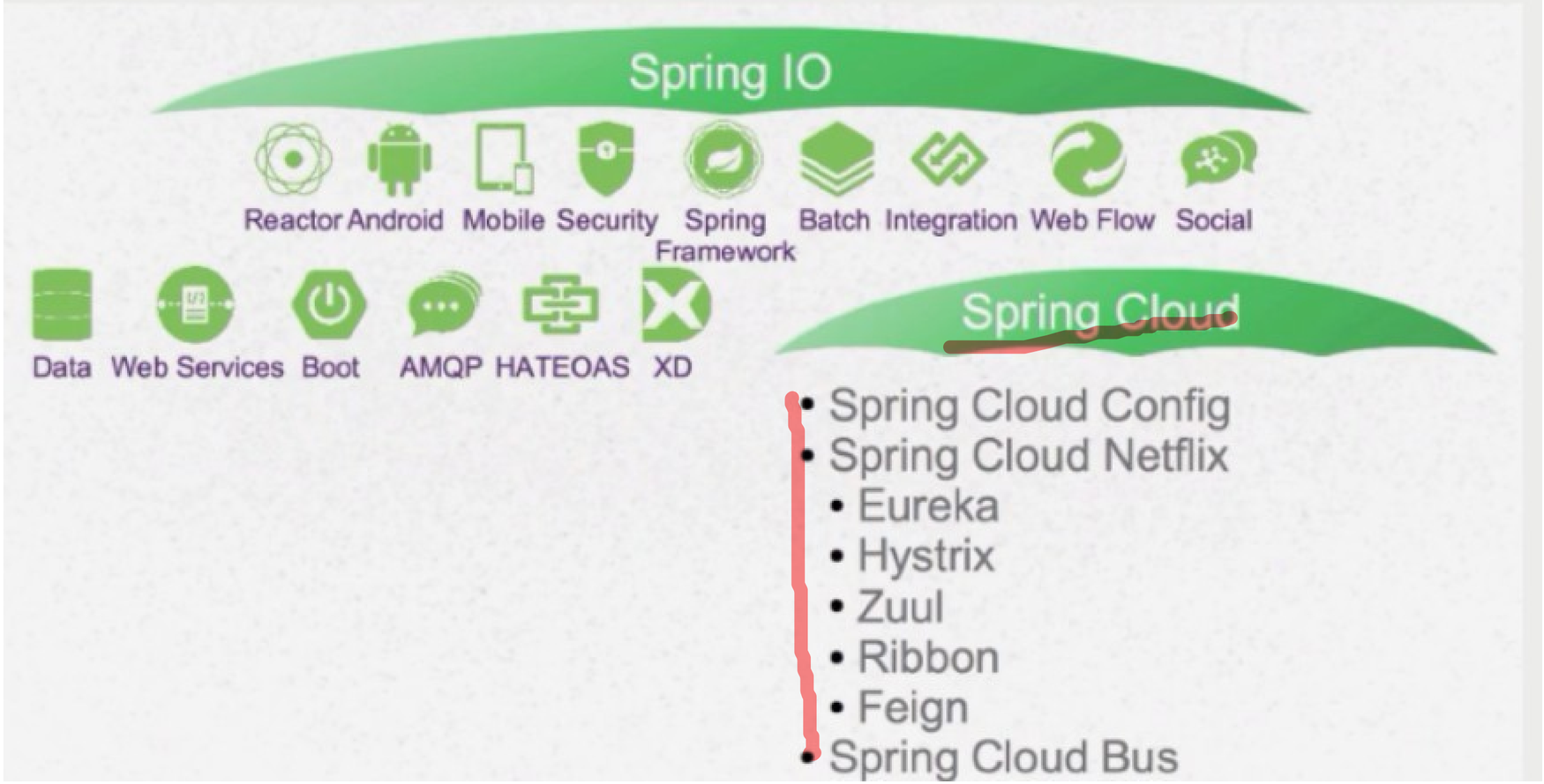


Spring IO (Spring Integration Objects) is the name used for the family of sub-projects of the **Spring**. It is considered as an umbrella and sub-projects are located below such umbrella.

Spring Cloud is a sub-project located in the **Spring IO** Umbrella and it itself is an umbrella, a sub-umbrella.

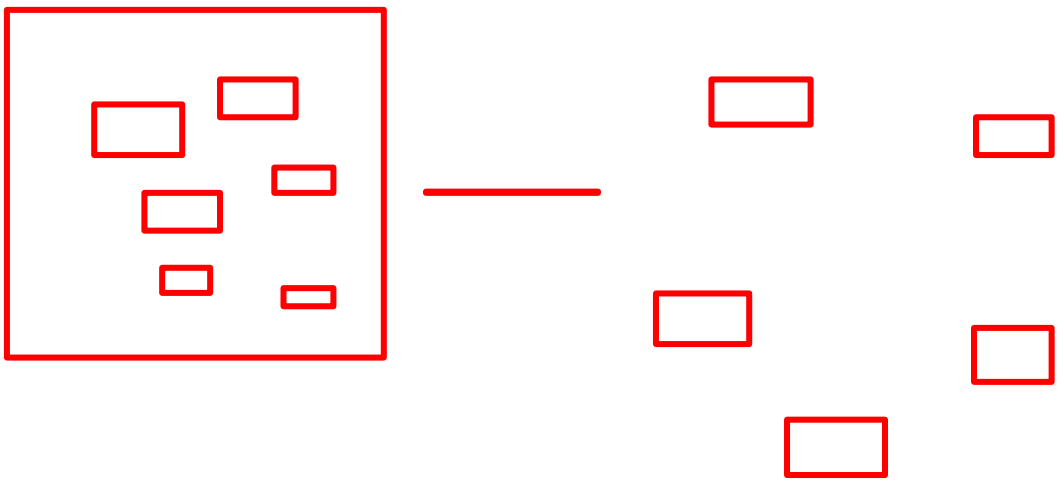
Introduction to Spring cloud

Spring Cloud is a sub-project located in the **Spring IO** Umbrella and it itself is an umbrella, a sub-umbrella.



What is spring cloud
is umberalla of many tools used to creat
ms as per 12 factor rule

netflex



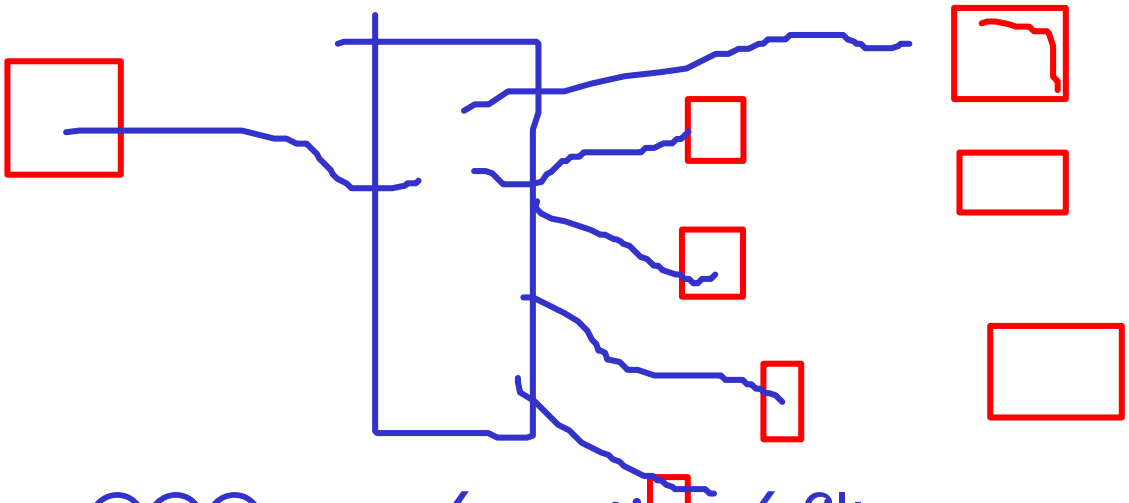
1. Eureka server: service discovery and load balancaing

spring cloud load labancer/ Ribbon

2. Resilance4J : many design pattern imp
circuit breakder/ bulkhead ...etc

Hystric

3. Spring cloud gateway (Zuul api gateway)



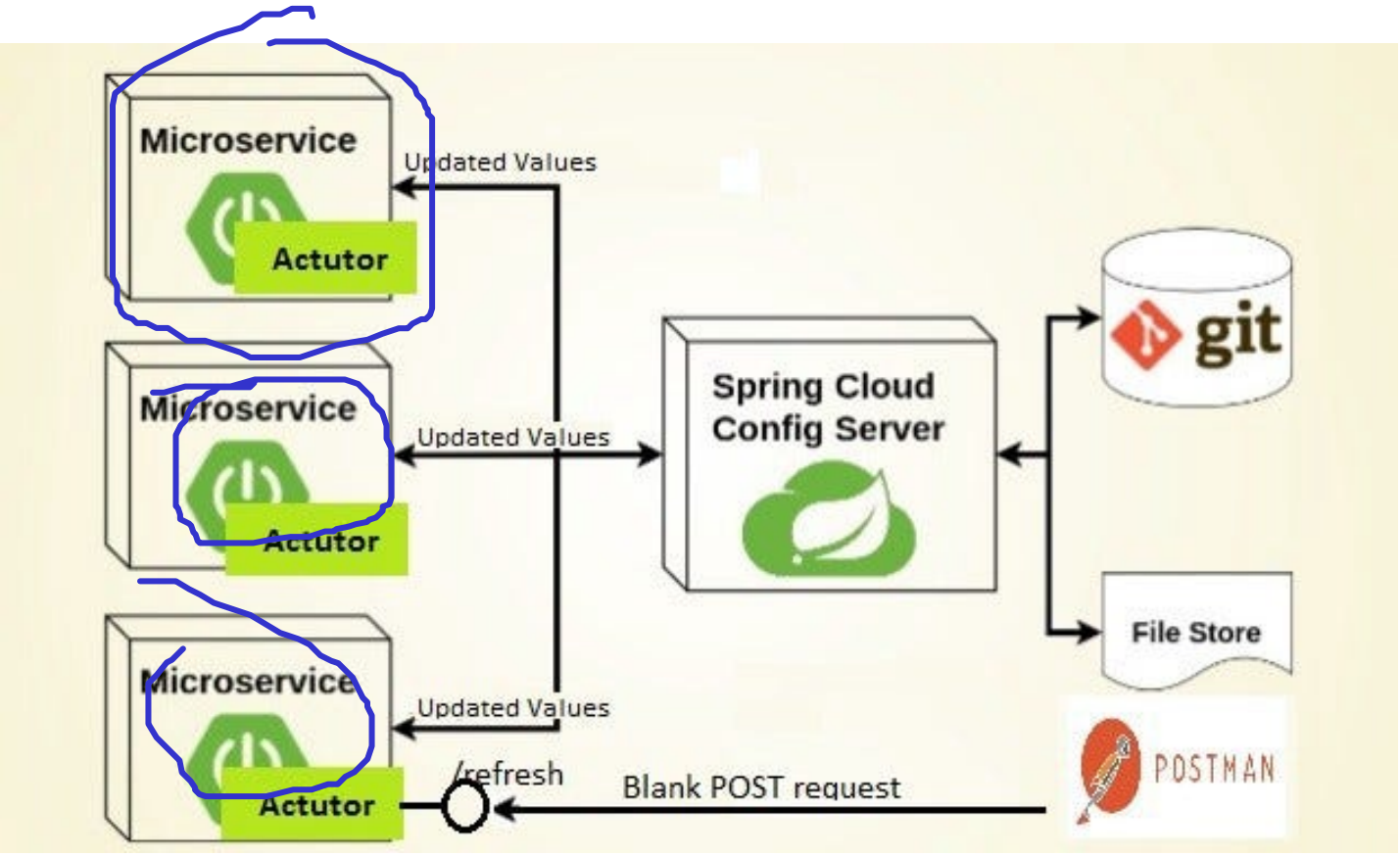
CCC: sec/ routing/ filters

JWT , Oauth

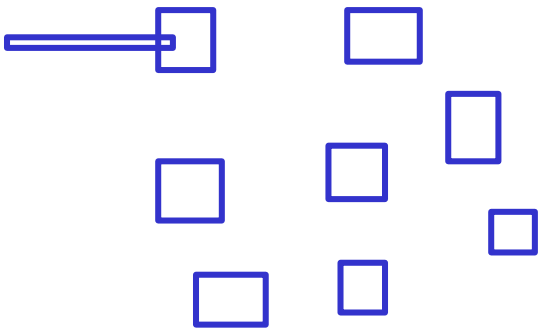
4. consule server: alterative to eureka server

Eureka server: service discovery and load balancaing

5. spring clound config sever



6. spring sluth and ziplin



7. grafana / promotheus

scrap the matrix

Dashboad

8. ELK

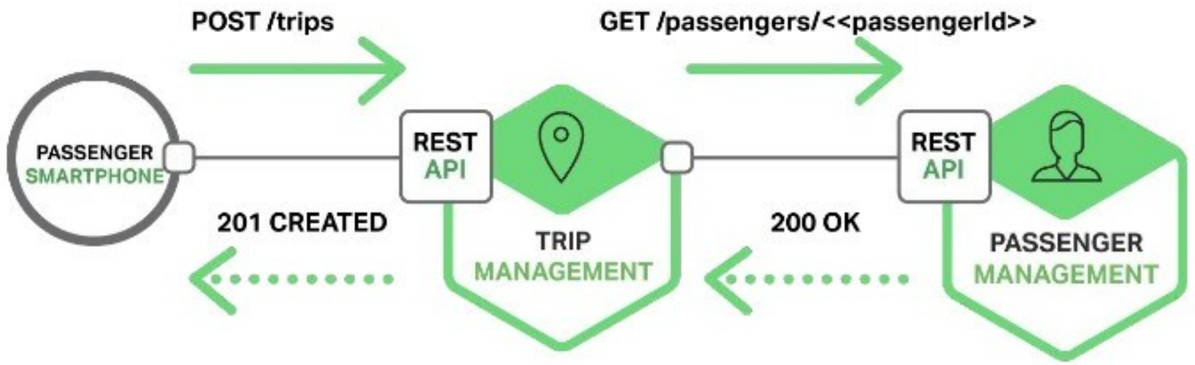
Module 4: Introduction to Eureka services discovery

How do services find each other?
What happens if we run multiple instances for a service
AKA yellow pages*
Eureka created by Netflix

rgupta.mtech@gmail.com

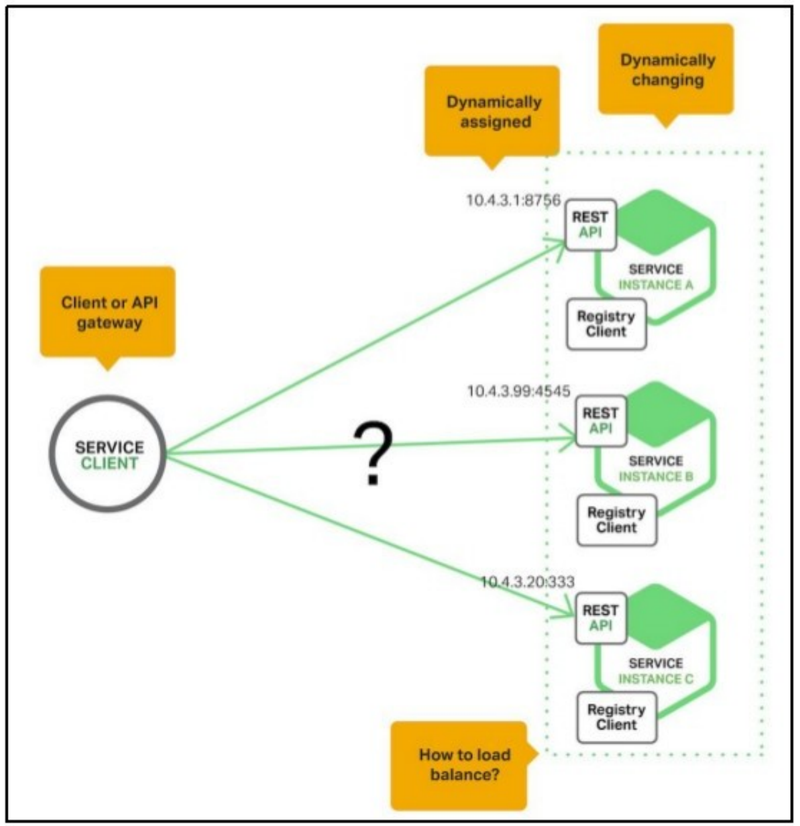
Need of Service discovery

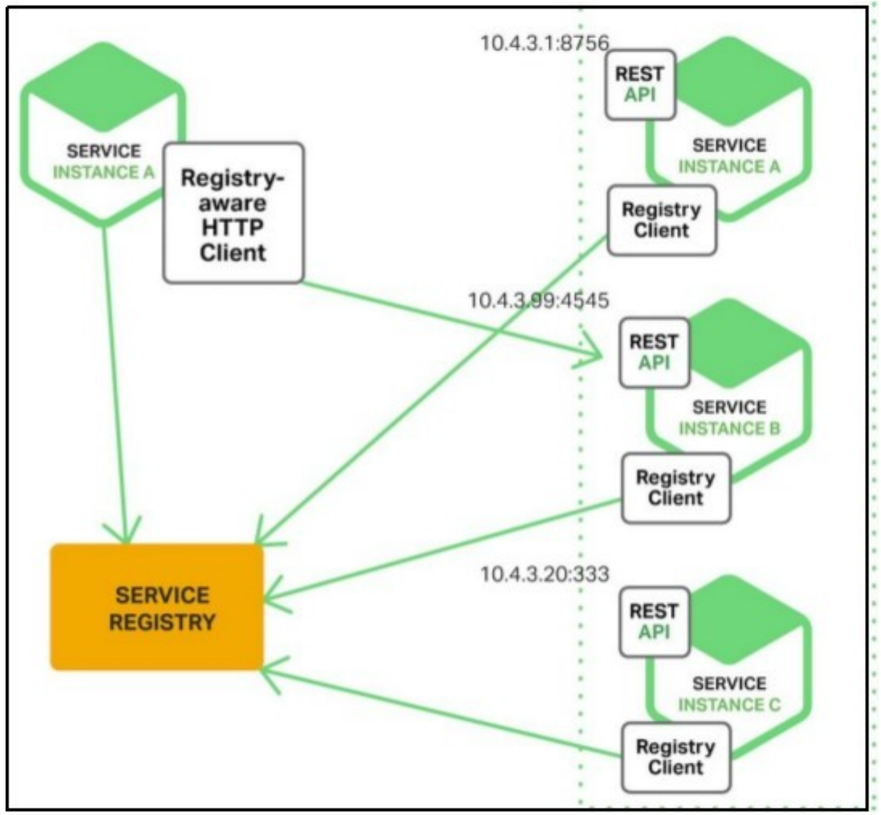
To perform communication between services we need know the location of the service(port, host). In traditional applications it's a simple task because services run in a fixed and known location.



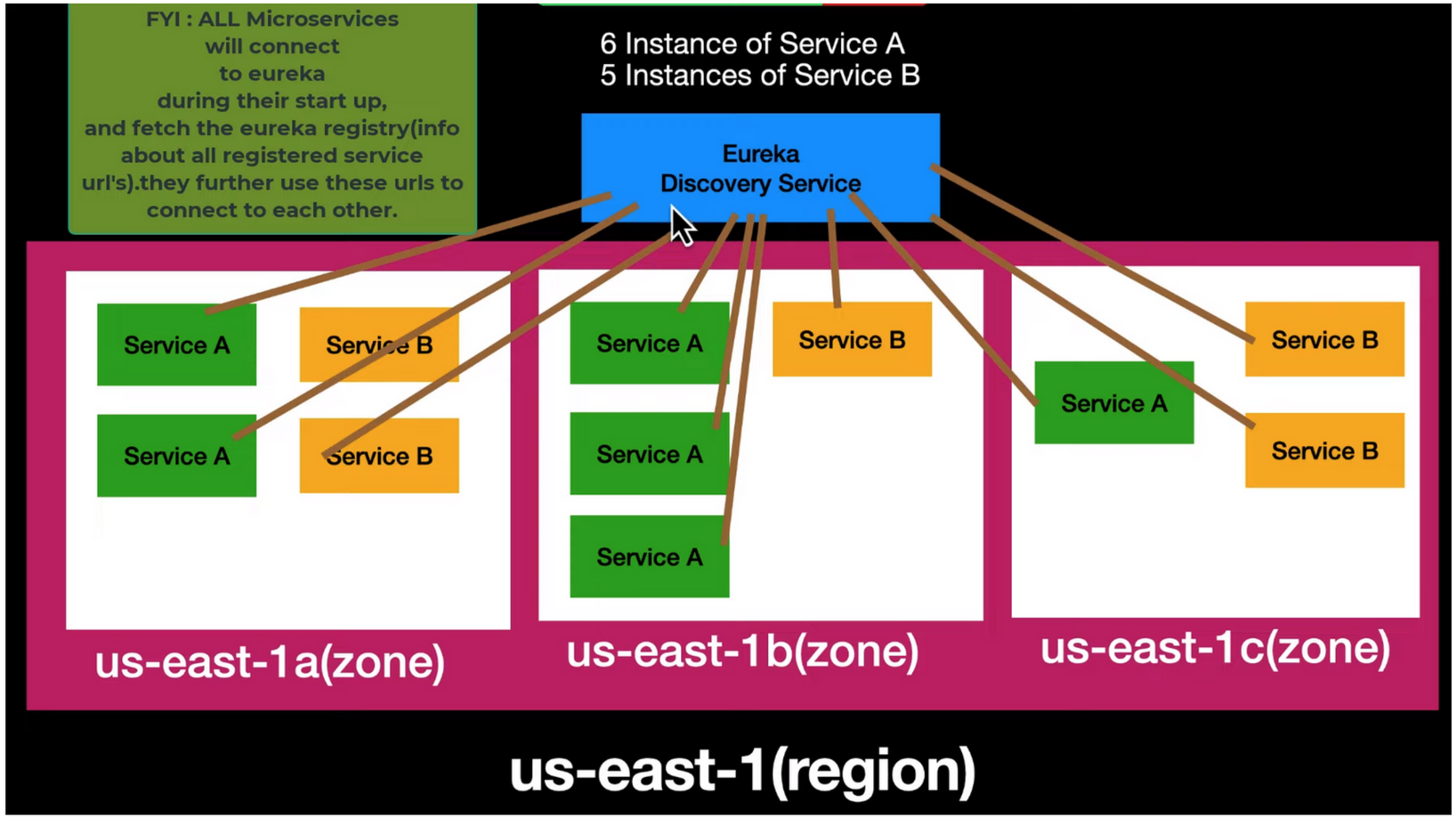
In modern applications the services are running in a dynamic enviroment. A service can have N instances running in N different machines. In this case, to know host and port of each service is very painful.

Need of Service discovery



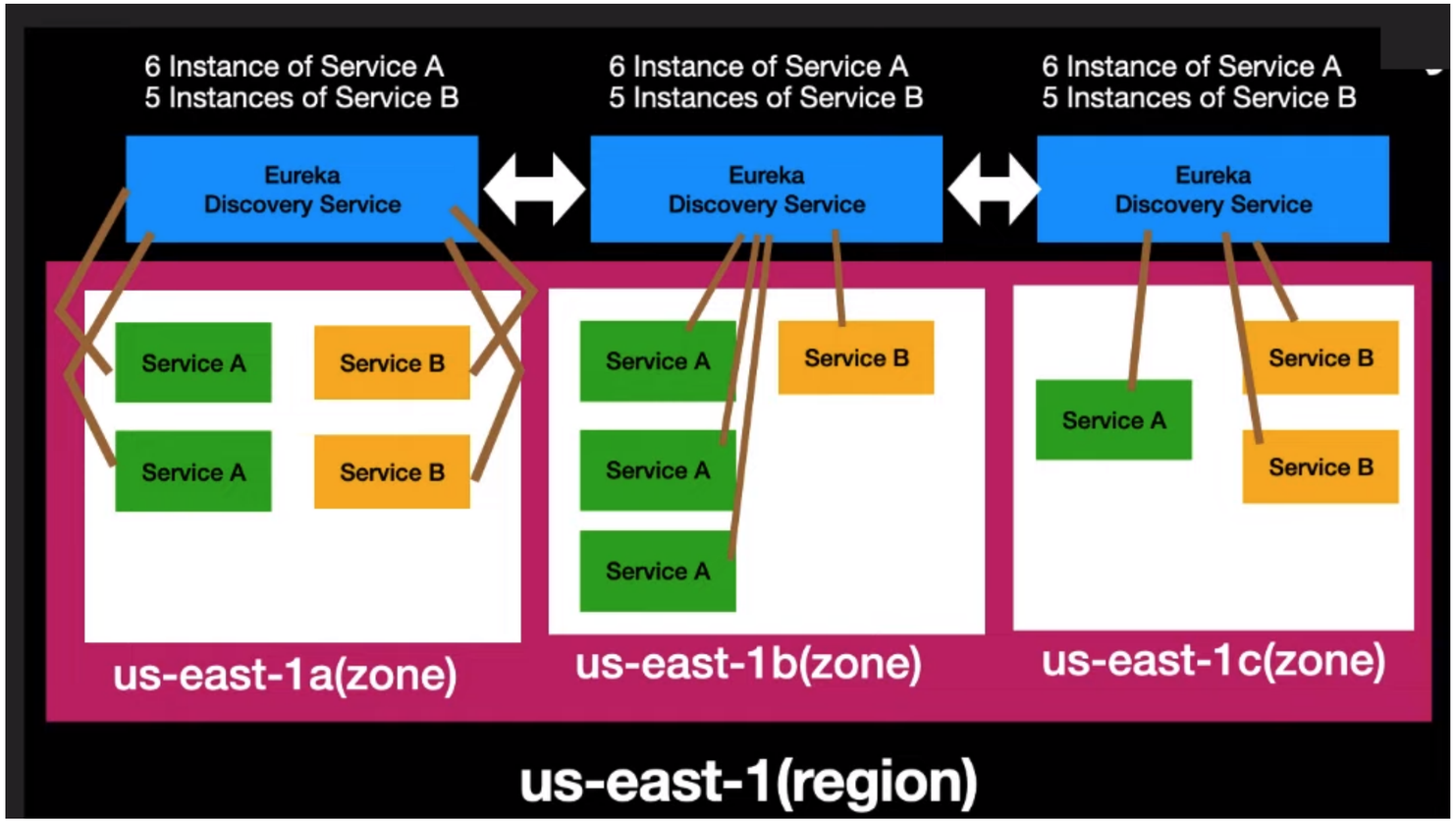


Understanding Configuration of eureka server

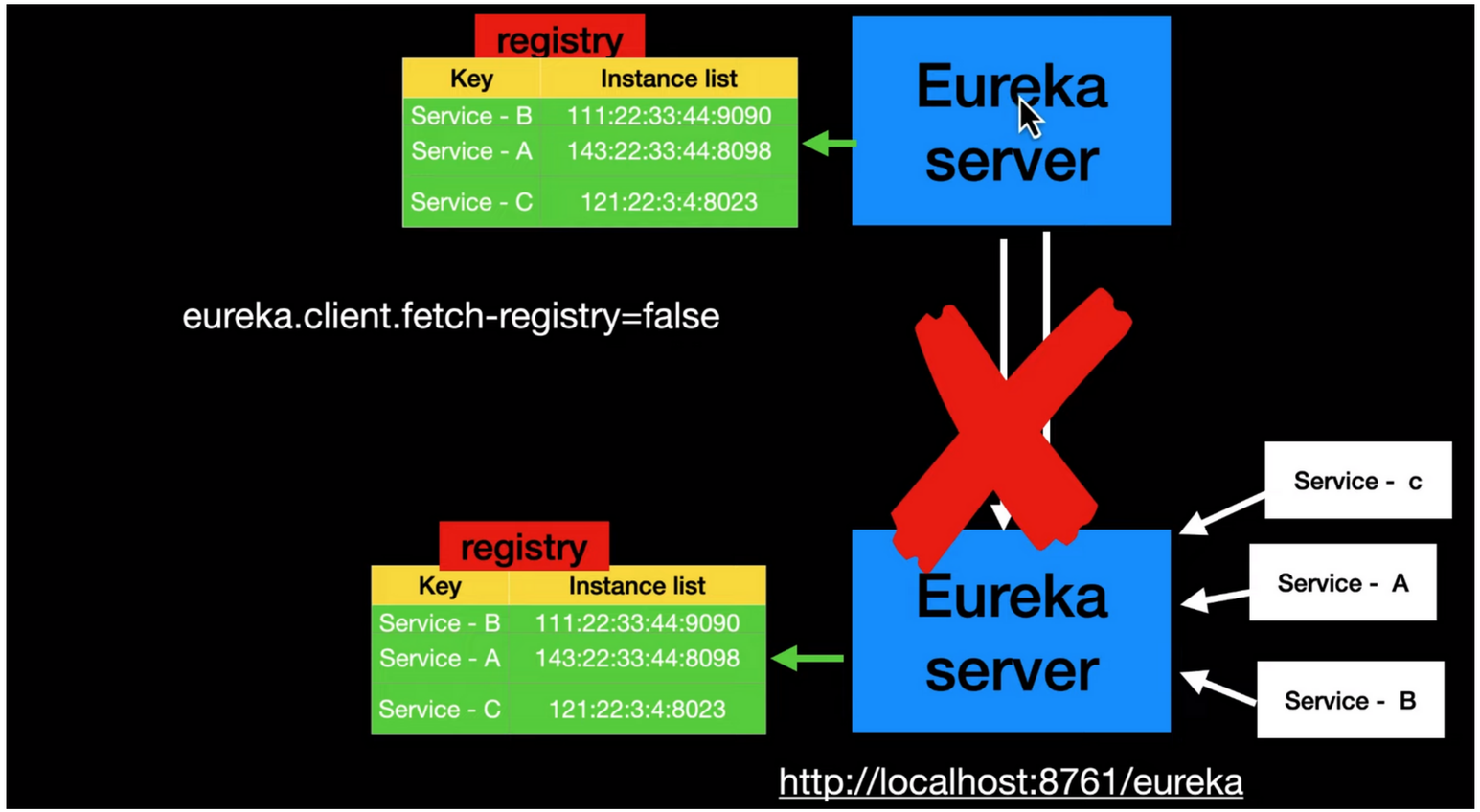


rgupta.mtech@gmail.com

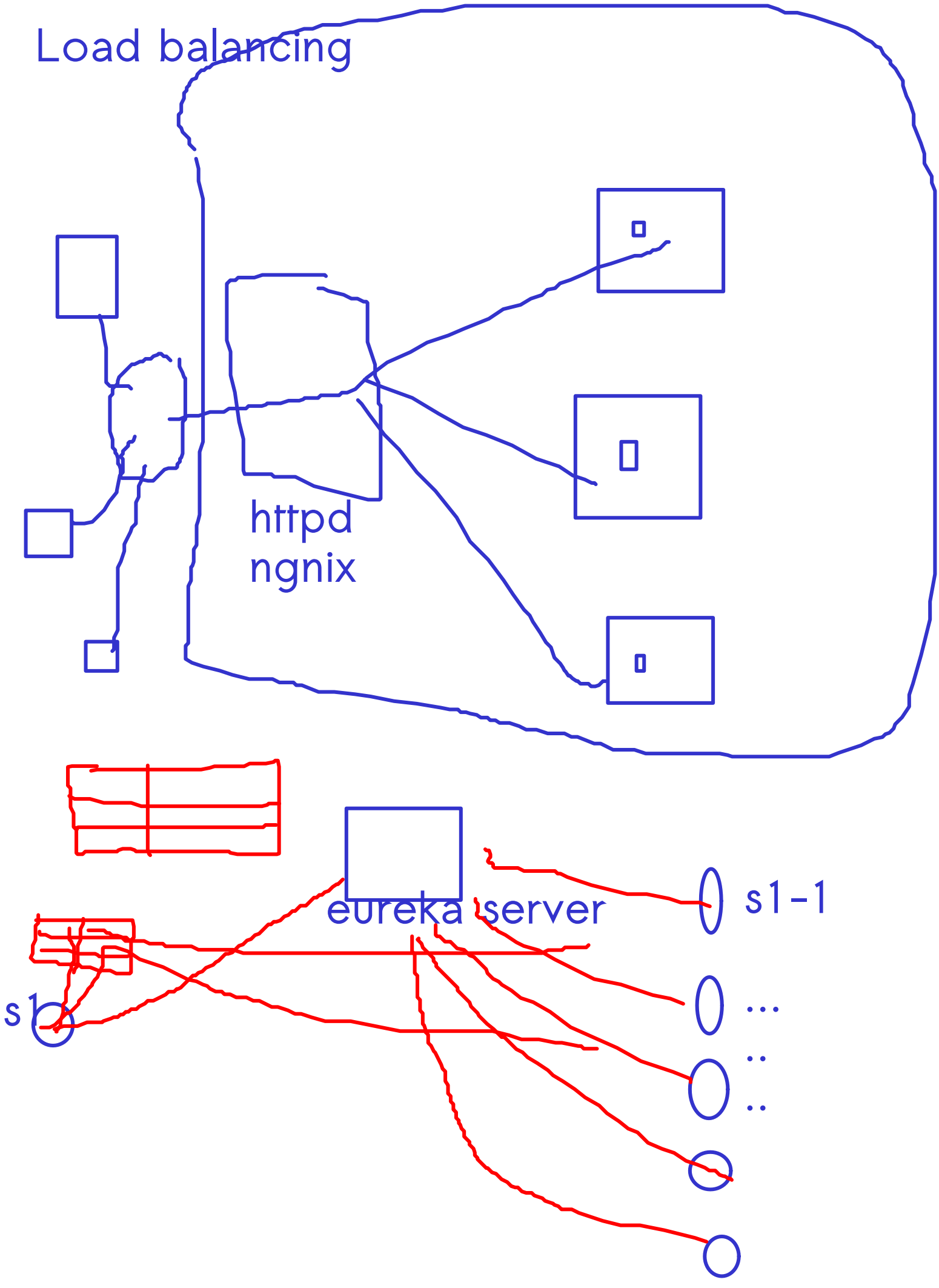
Understanding Configuration of eureka server



Understanding Configuration of eureka server



rgupta.mtech@gmail.com


Load balancing
httpd
ngnix
eureka server
s1-1
s1

Introduction to Cliend side load balancing

rgupta.mtech@gmail.com

Client-side Load Balancing

Each service typically deployed as multiple instances for fault tolerance and load sharing.
But there is problem how to decide which instance to use?

Spring Cloud loadbalancer (Old tool Netflix Ribbon)

- **it provide several algorithm for Client-Side Load Balancing.**
- **Spring provide smart RestTemplate for service discovery and load balancing by using @LoadBalanced annotation with RestTemplate instance.**

rgupta.mtech@gmail.com

Module 5:Feign Declaritive REST client

rgupta.mtech@gmail.com

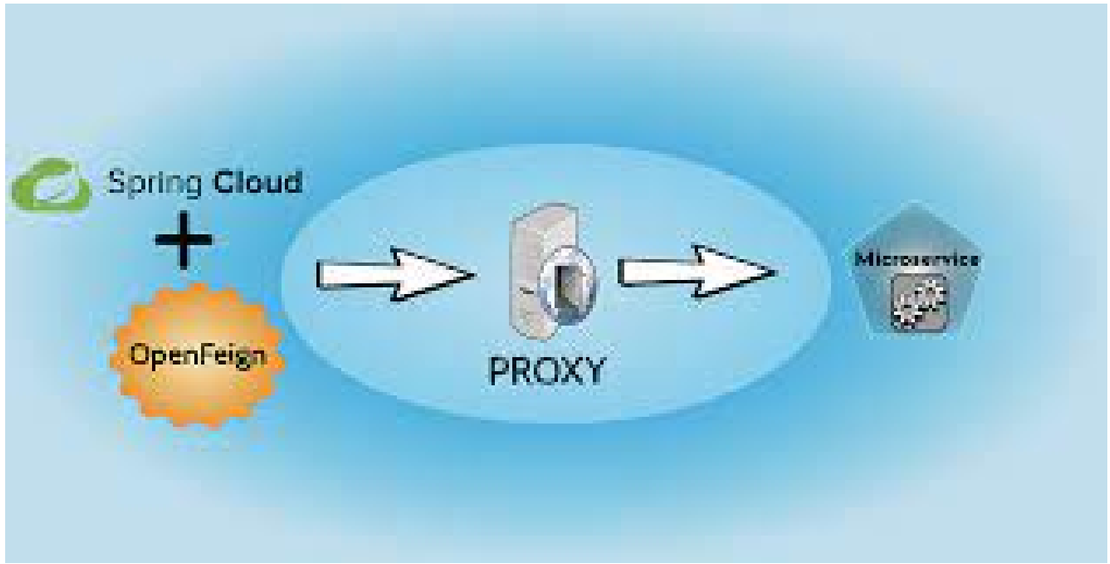
Spring Cloud OpenFeign

Spring Cloud OpenFeign — a declarative REST client for Spring Boot apps.

Feign makes writing web service clients easier with pluggable annotation support, which includes Feign annotations and JAX-RS annotations.

Also, Spring Cloud adds support for Spring MVC annotations and for using the same HttpMessageConverters as used in Spring Web.

One great thing about using Feign is that we don't have to write any code for calling the service, other than an interface definition.



rgupta.mtech@gmail.com

Module 6: Introduction to Resilience4j

rgupta.mtech@gmail.com

Agenda

What is Resilience4j?
Why Resilience4j?
Resilience4j Modules
How to use Resilience4j in your project?

rgupta.mtech@gmail.com

What is Resilience4j?

Resilience4j is an easy-to-use fault tolerance library inspired by Netflix Hystrix, but designed for Java 8 and functional programming.
In Resilience4j you don't have to go all-in, you can pick what you need.

Why Resilience4j?

- **This library is capable of handling the asynchronous calls Resilience4j is designed for Java8 and functional programming**
- **Resilience4j enables the resilience in complex distributed systems where failure might take place**
- **Resilience4j is lightweight because it only uses Vavr, which does not have any other external library dependencies.**

rgupta.mtech@gmail.com

How to use Resilience4j in your projects?

You can simply use the following dependency in your maven project and get started with the Resilience4j To add the Resilience4j dependency to your project, you need to add the following dependency to your pom.xml file

Resilience4j Module

- **Circuit Breaker**
- **Retry**
- **Rate Limiter**
- **BulkHead**
- **Time Limiter**
- **Cache**

rgupta.mtech@gmail.com

Circuit Breaker Module

Its basic function is to interrupt current flow after a fault is detected.

Unlike a fuse, which operates once and then must be replaced, a circuit breaker can be reset (either manually or automatically) to resume normal operation.

Circuit breaker pattern

- Detect something is wrong
- Take temporary steps to avoid the situation getting worse
- Deactivate the “problem” component so that it doesn’t affect downstream components

rgupta.mtech@gmail.com

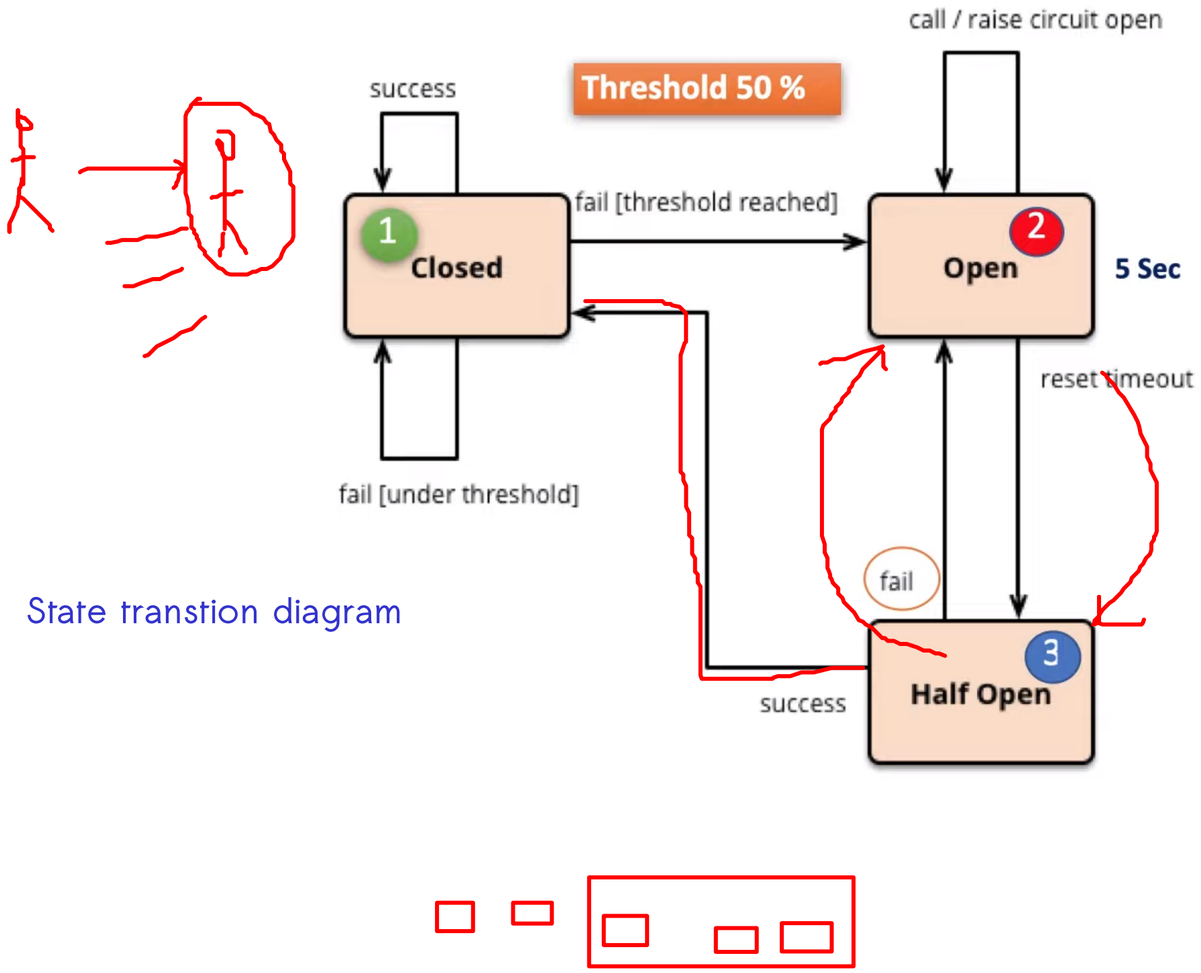
Circuit Breaker Module

- The circuit breaker is essentially a pattern that helps to prevent cascading failures in a system
- The circuit breaker pattern allows you to build a fault-tolerant and resilient system that can survive gracefully when key services are either unavailable or have high latency.
- Circuit breaker pattern is generally used in microservices architecture where there are multiple services involved but it can be used otherwise as well.

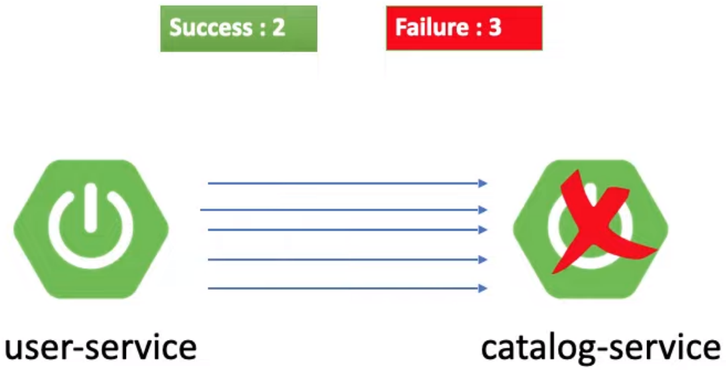
- **The circuit breaker has the following 3 states**
- **Closed – Closed is when everything is normal, in the beginning, it will be in the closed state and if failures exceed the threshold value decided at the time of creating circuit breaker, the circuit will trip and go into an open state.**
- **Open – Open is the state when the calls start to fail without any latency i.e calls will start to fail fast without even executing the function calls.**
- **Half-open – In half-open state what will happen is, the very first call will not fail fast and all other calls will fail fast just as in the open state. If the first call succeeds then the circuit will go inthe closed state again and otherwise, it will go into the open state again waiting for the reset timeout.**

rgupta.mtech@gmail.com

Circuit Breaker Module



State transtion diagram

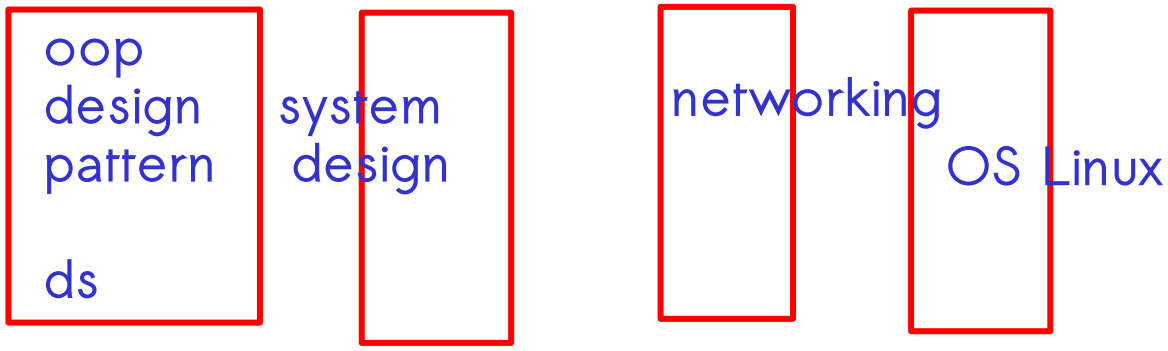



```
resilience4j:
  circuitbreaker:
    instances:
      couponservice:
        register-health-indicator: true
        event-consumer-buffer-size: 10
        failure-rate-threshold: 50
        minimum-number-of-calls: 5
        automatic-transition-from-open-to-half-open-enabled: true
        wait-duration-in-open-state: 5s
        permitted-number-of-calls-in-half-open-state: 3
        sliding-window-size: 10
        sliding-window-type: COUNT_BASED
```

rgupta.mtech@gmail.com

Learning path for cloud engg:

solid foundatiaon



core java --> java 8 streams-> jdbc -> design pattern
GOF pattern
--> servlet/jsp --> jpa --> spring core
di : xml/java/annotation
AOP
SpringJDBC
Spring MVC

--> spring boot ---> Bascic Auth-> JWT->Oauth->Keyclock
spring boot ms ---> AWS/gcp/oracle
RMM
-> Linux command----> docker--> k8s--> adv design
pattern
SAGA pattern

saml
openid

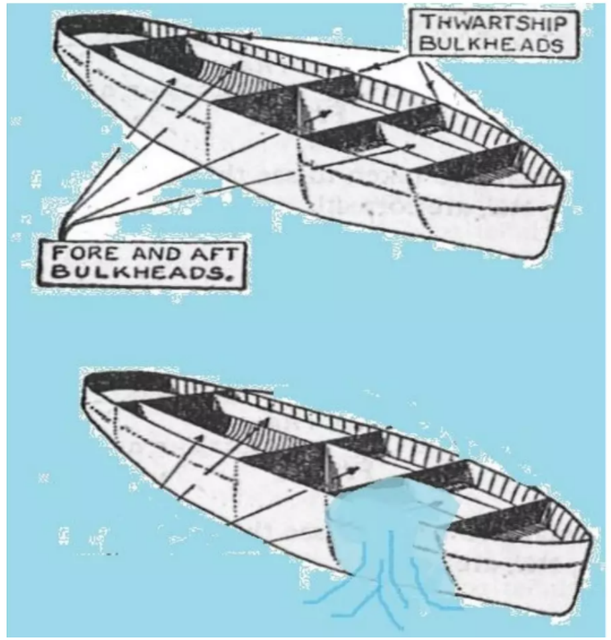
BulkHead Pattern

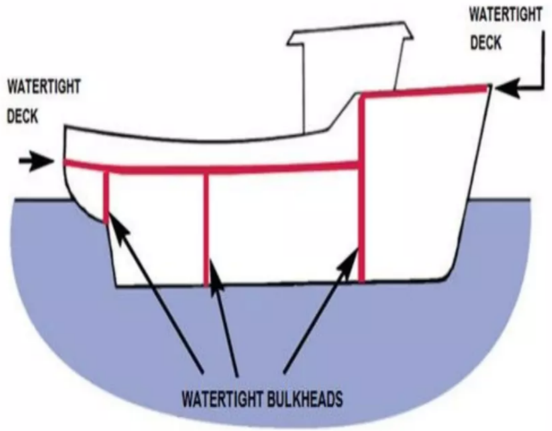
The BulkHead Pattern is a type of application design that is tolerant of failure. In a bulkhead architecture, elements of an application isolated into pools so that if one fails, the other will continue of function.

BulkHead Implementation framework

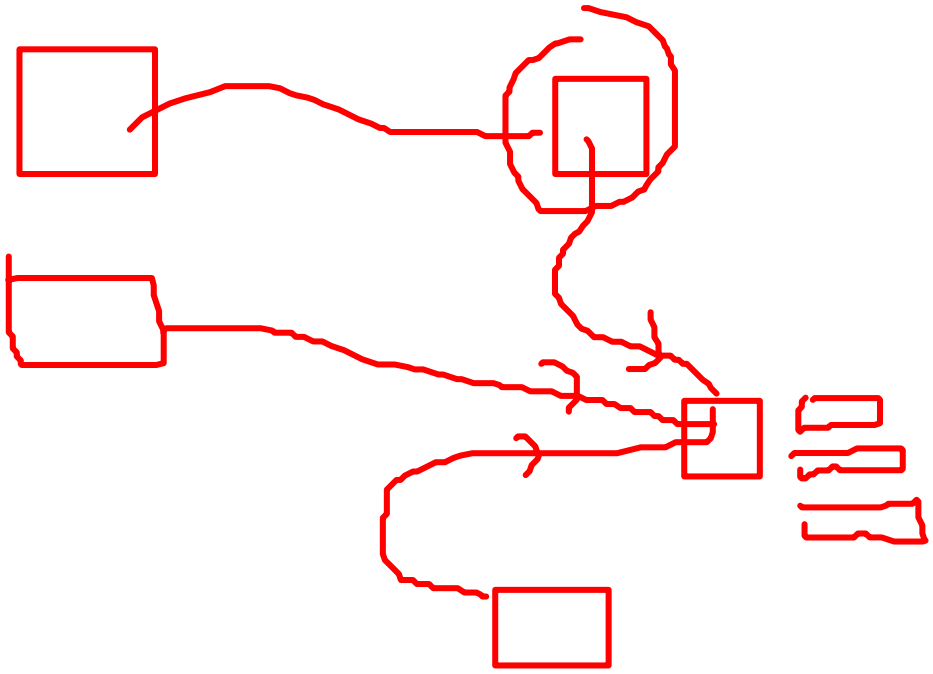
1. Hystrix

2. Resilience4j





rgupta.mtech@gmail.com



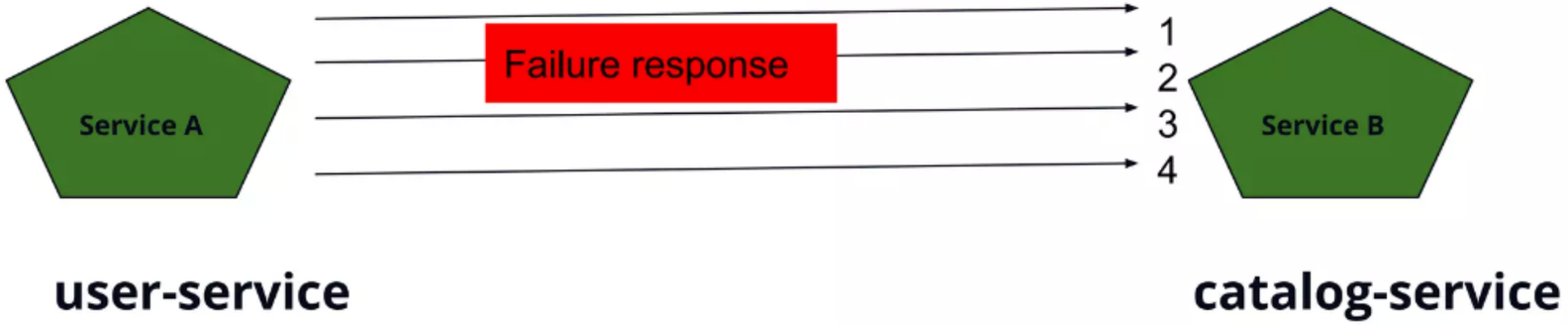
When to apply BulkHead Pattern?

1. Apply the bulkhead pattern whenever you want to scale a service independent of other services.
2. Isolate resources used to consume a set of backed services, especially if the application can provide some level of functionality even when one of the service is not responding.
3. Apply the bulkhead pattern to fault isolate components of varying risk or availability requirements.
4. Protect the application from cascading failure.

rgupta.mtech@gmail.com

Retry Module

- **There can be scenarios when there are intermittent network connectivity errors causing your service to be unavailable.**
- **These issues are generally self-correcting and if you retry the operation after a small delay its most probably going to succeed.**
- **Retry pattern help to resolve transient failures in distributed architecture by automatically hit the target resource again after some specific time.**




```
Resilience4j:
    retry:
      instances:
        userService:
          maxRetryAttempts: 4
          waitDuration: 5s
```

rgupta.mtech@gmail.com

When to use retry?

The faults are expected to be short lived, and repeating request that failed previously could succeed on a subsequent attempt.

- **Calling an HTTP service from another REST endpoint.**
- **Calling a web service.**
- **No or slow responses due to a large number of requests towards the resource(database or service).**

rgupta.mtech@gmail.com

Time Limiter

1. Setting a limit on the amount of time we are willing to wait for an operation to complete is called time limiting.
2. If the operation does not complete within the time specified, we want to notified about it with a timeout error.

Resilience4j's Time Limiter

TimeLimiter can be used to set limits (timeouts) on asynchronous operations implemented with completableFuture.

Configuration

```
timeoutDuration:2s

cancelRunningFuture:true
```

rgupta.mtech@gmail.com

Rate limiter pattern

Rate limiter pattern helps us to make services highly available just by limiting the number of calls we could process in a specific window.

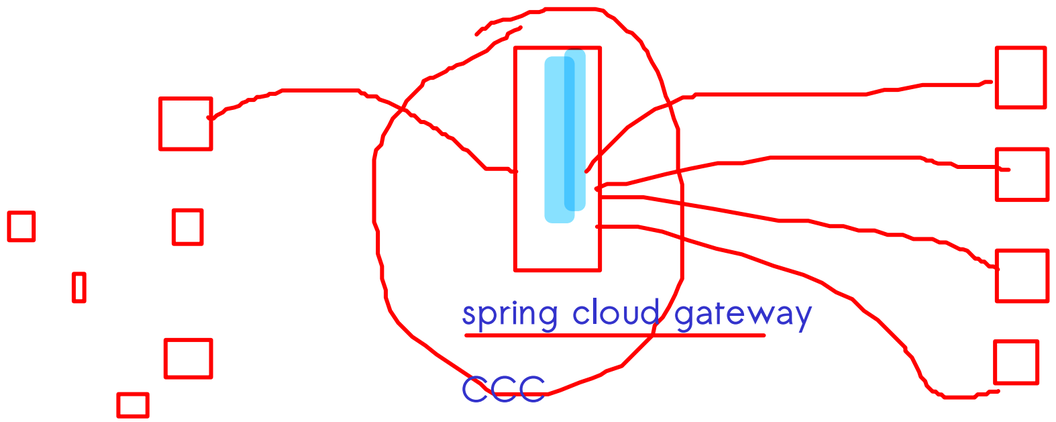
Rate limiting specified in terms of :-
1. Requests per second(rpc)
2. Requests per minute(rpm)
3. Requests per hour(rph) Rate Limiter Pattern

Resilience4j-ratelimiter

Config property	Default value	Description
timeoutDuration	5 [s]	The default wait time a thread waits for a permission
limitRefreshPeriod	500 [ns]	The period of a limit refresh. After each period the rate limiter sets its permissions count back to the limitForPeriod value
limitForPeriod	50	The number of permissions available during one limit refresh period

rgupta.mtech@gmail.com

Module 7: Spring Cloud Gateway

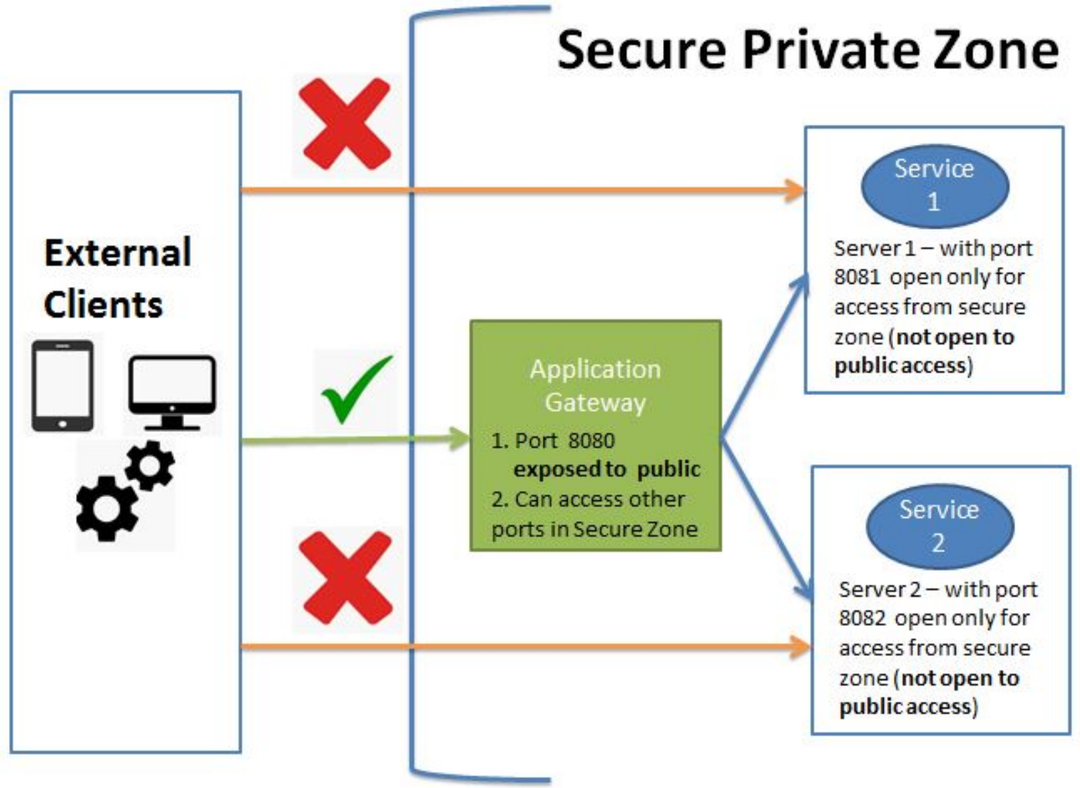


rgupta.mtech@gmail.com

What is an API Gateway? Why do we need it?

An API Gateway acts as a single entry point for a collection of microservices. Any external client cannot access the microservices directly but can access them only through the application gateway

In a real world scenario an external client can be any one of the three-
Mobile Application
Desktop Application
External Services or third party Apps



rgupta.mtech@gmail.com

Zuul vs API Gateway

Zuul is a blocking API. A blocking gateway api makes use of as many threads as the number of incoming requests. So this approach is more resource intensive.

If no threads are available to process incoming request then the request has to wait in queue.

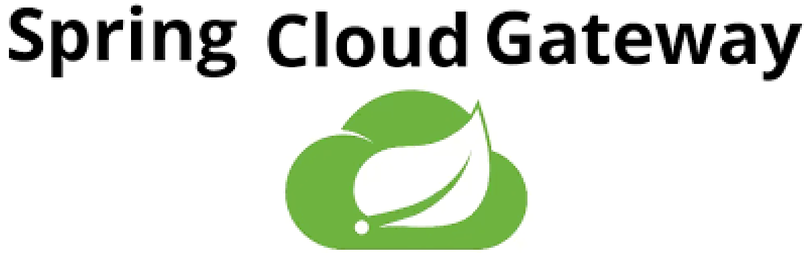
Spring Cloud Gateway is a non blocking API. When using non blocking API, a thread is always available to process the incoming request.

These request are then processed asynchronously in the background and once completed the response is returned. So no incoming request never gets blocked when using Spring Cloud Gateway.

rgupta.mtech@gmail.com

Advantage of API Gateway

1. This improves the security of the microservices as we limit the access of external calls to all our services.
2. The cross cutting concerns like authentication, monitoring/metrics, and resiliency will be needed to be implemented only in the API Gateway as all our calls will be routed through it.
3. The client does not know about the internal architecture of our microservices system. Client will not be able to determine the location of the microservice instances.
4. Simplifies client interaction as he will need to access only a single service for all the requirements.
5. Spring Cloud Gateway Architecture



rgupta.mtech@gmail.com

Component of API Gateway

Spring Cloud Gateway is API Gateway implementation by Spring Cloud team on top of Spring reactive ecosystem.
It consists of the following building blocks-
Route:

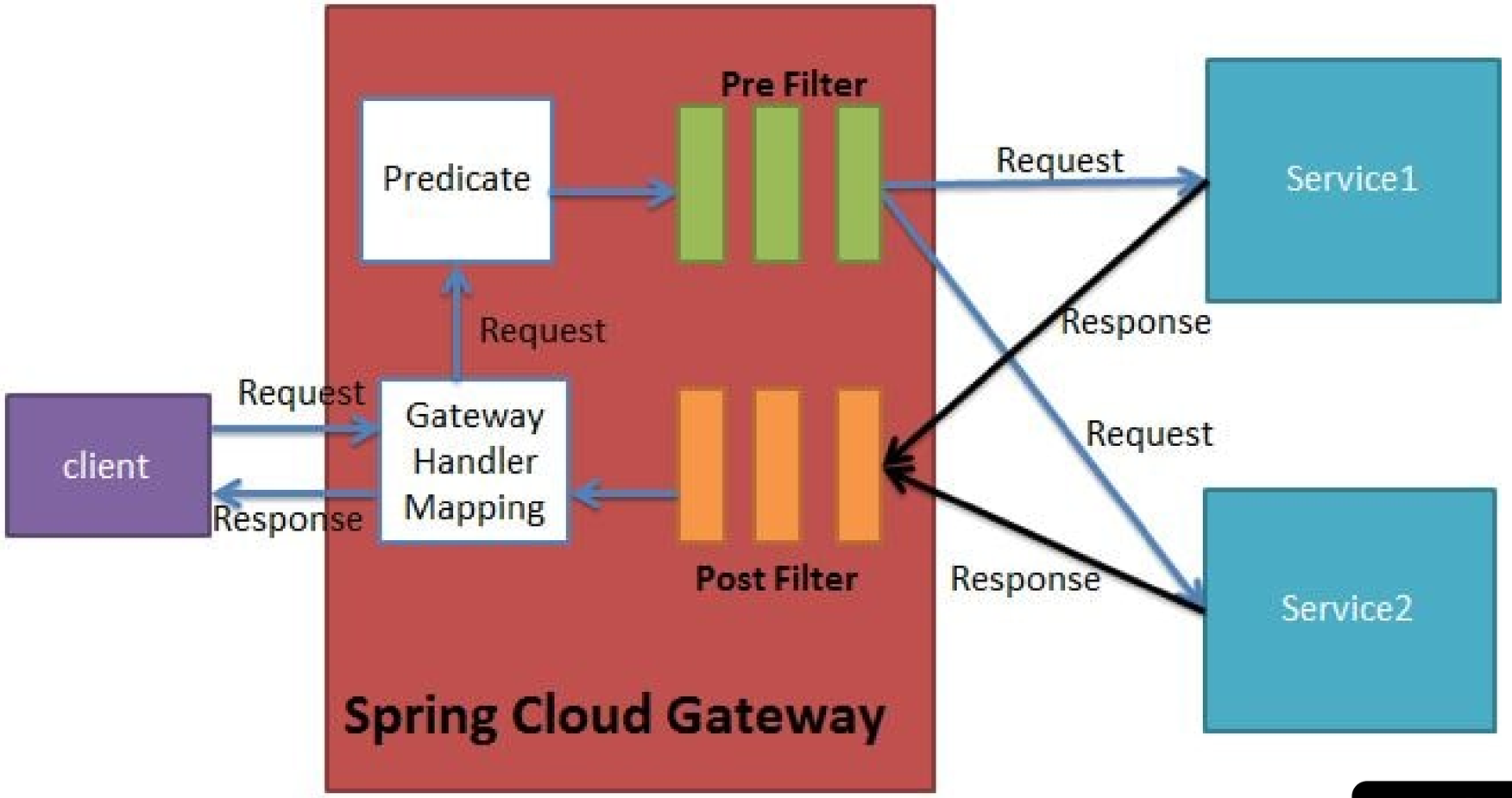
- **Route the basic building block of the gateway.**
- **It consists ofID destination URI Collection of predicates and a collection of filters**
- **A route is matched if aggregate predicate is true.**
-

Predicate:
This is similar to Java 8 Function Predicate. Using this functionality we can match HTTP request, such as headers , url, cookies or parameters.
Filter:
These are instances Spring Framework GatewayFilter. Using this we can modify the request or response as per the requirement.

rgupta.mtech@gmail.com

spring cloud gateway architecture

When the client makes a request to the Spring Cloud Gateway, the Gateway Handler Mapping first checks if the request matches a route. This matching is done using the predicates. If it matches the predicate then the request is sent to the filters.



rgupta.mtech@gmail.com

Implementing Spring Cloud Gateway

Using Spring Cloud Gateway we can create routes in either of the two ways -

1. **Use java based configuration to programmatically create routes**
2. **Use property based configuration(i.e application.properties or application.yml) to create routes.**

```java
@Configuration
public class SpringCloudConfig {
    @Bean
    public RouteLocator gatewayRoutes(RouteLocatorBuilder builder) {
        return builder.routes()
                .route(r -> r.path( ...patterns: "/greet/**").uri("http://localhost:8181/"))
                .route(r -> r.path( ...patterns: "/client/**").uri("http://localhost:8282/"))
                .build();
    }
}
```

```yaml
server:
  port: 8080
spring:
  cloud:
    gateway:
      routes:
        - id: greetservice
          uri: http://localhost:8181/
          predicates:
            - Path=/greet/**
        - id: clientservice
          uri: http://localhost:8282/
          predicates:
            - Path=/client/**
```

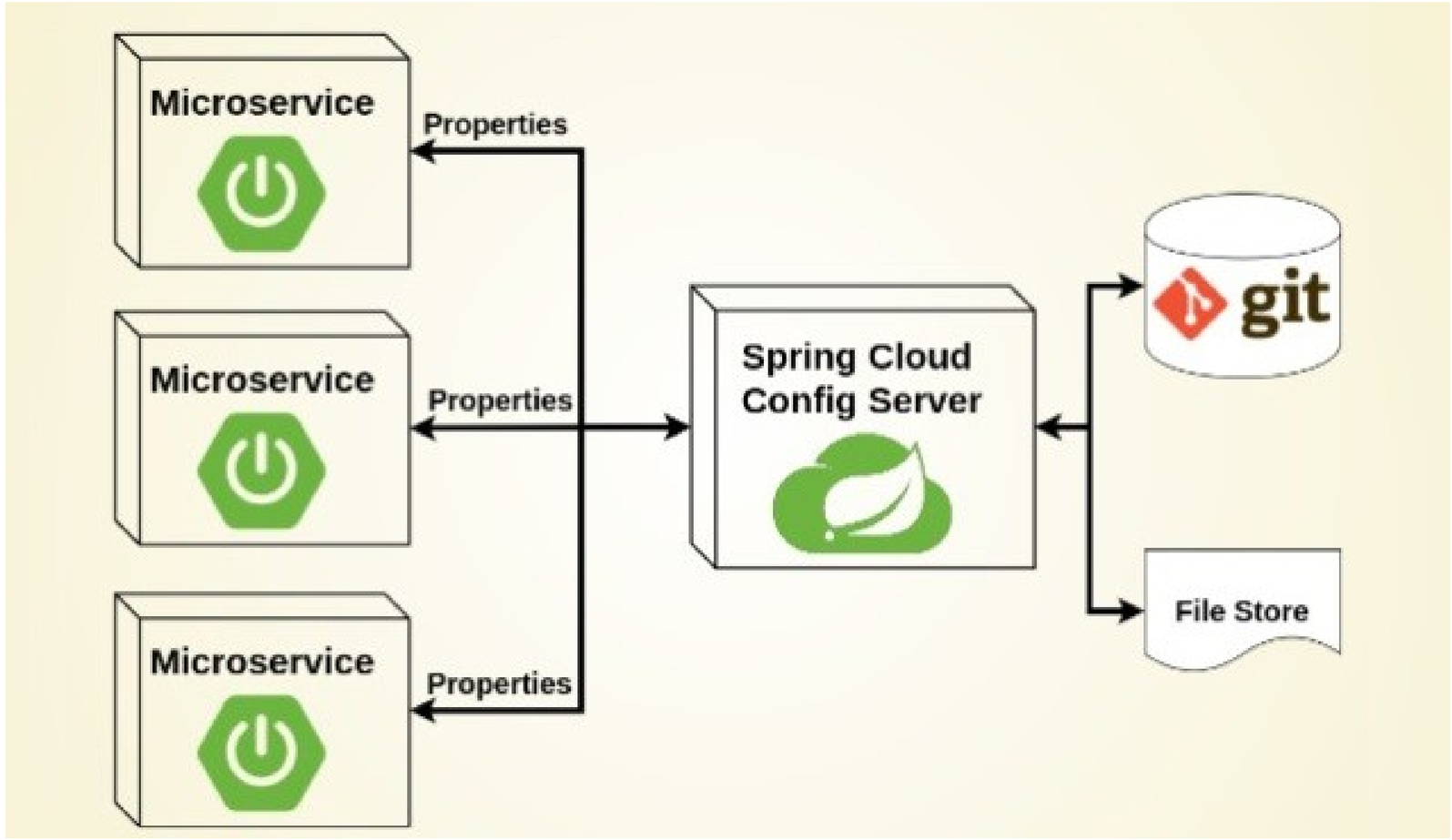
rgupta.mtech@gmail.com

Module 8: Spring Cloud Config Server

rgupta.mtech@gmail.com

Spring Cloud Config Server

Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system.
With the Config Server you have a central place to manage external properties for applications across all environments.



rgupta.mtech@gmail.com

Need of Spring Cloud Config Server

Modules can have common global properties which are repeated in all the modules. For example we have have properties related to Database, Messaging Queues etc. For example in our employee-consumer and employee-producer we are having the following property for

registering to Eureka Server.
eureka.client.serviceUrl.defaultZone=http://localhost:8090/eureka

We can externalize this property using Spring Cloud Config.

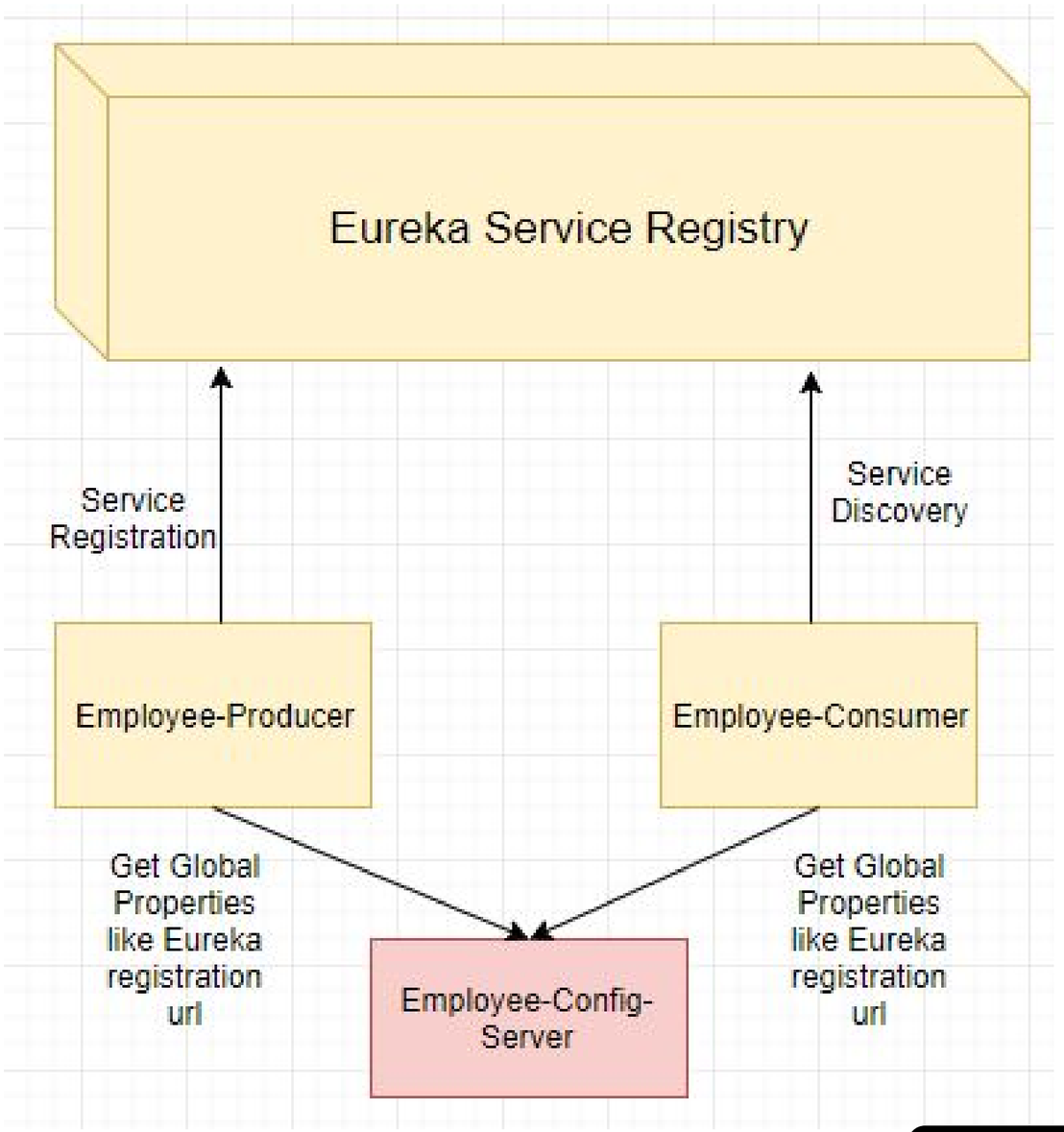
rgupta.mtech@gmail.com

Need of Spring Cloud Config Server

Modules can have common global properties which are repeated in all the modules. For example we have have properties related to Database, Messaging Queues etc. For example in our employee-consumer and employee-producer we are having the following property for

registering to Eureka Server.
eureka.client.serviceUrl.defaultZone=
http://localhost:8090/eureka

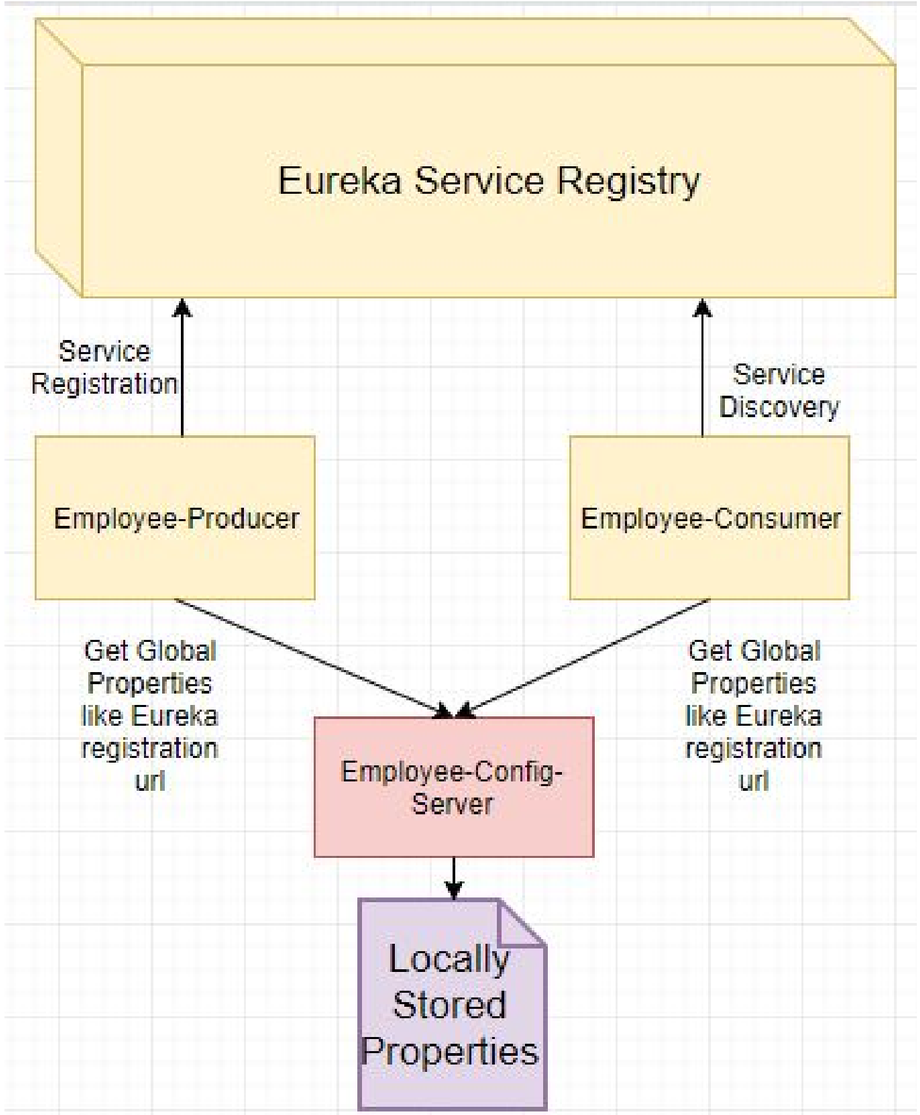
We can externalize this property using Spring Cloud Config.



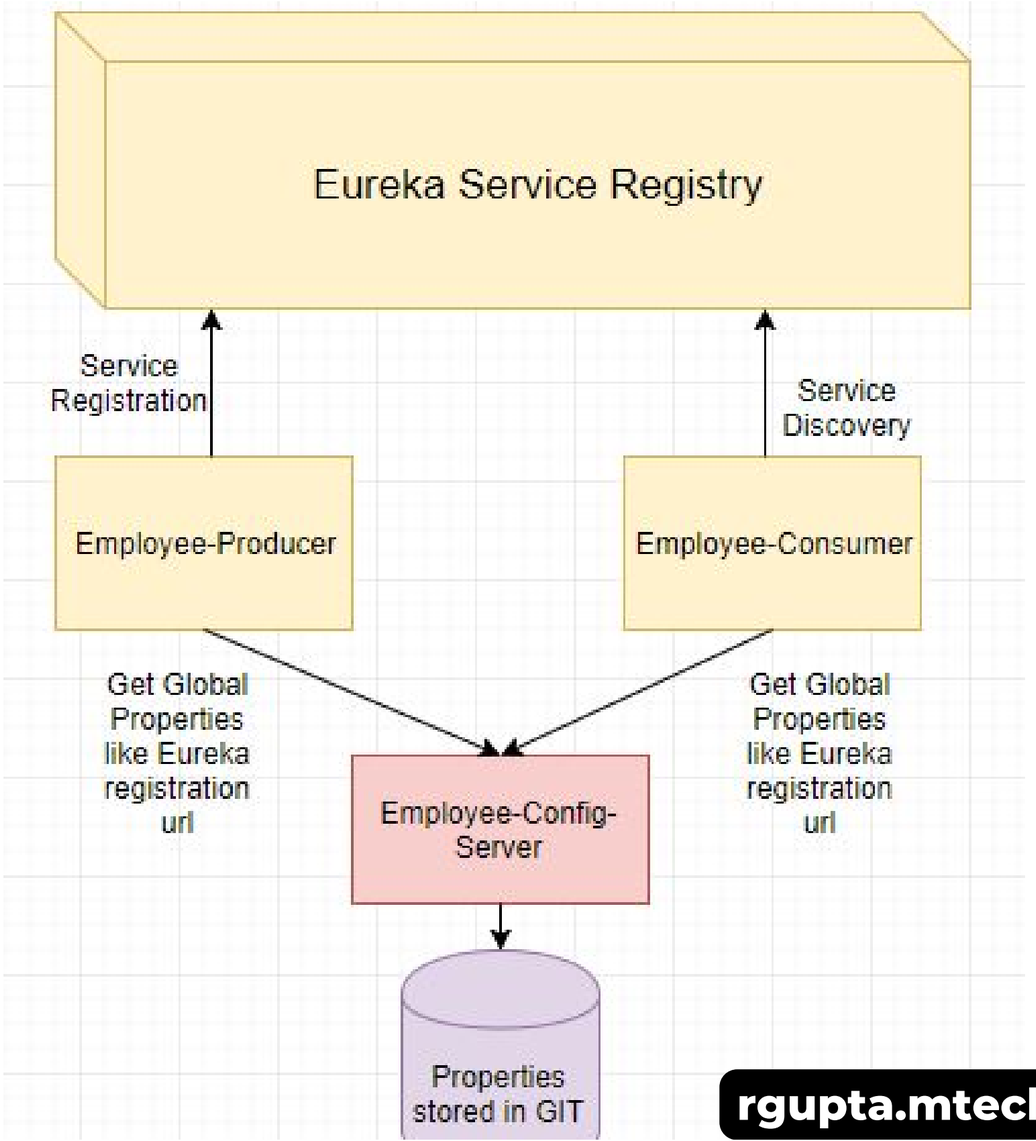
rgupta.mtech@gmail.com

How Cloud Config Server Works?

Local configuration



GIT configuration



rgupta.mtech@gmail.com

Module 9: Sluth and Zipkin

rgupta.mtech@gmail.com

Need of Distributed Log Tracing

- **MSA involve multiple services which interact with each other.**
- **So a functionality may involve call to multiple microservices. Usually for systems developed using Microservices architecture, there are many microservices involved.**
- **These microservices collaborate with each other.**

rgupta.mtech@gmail.com

Need of Distributed Log Tracing

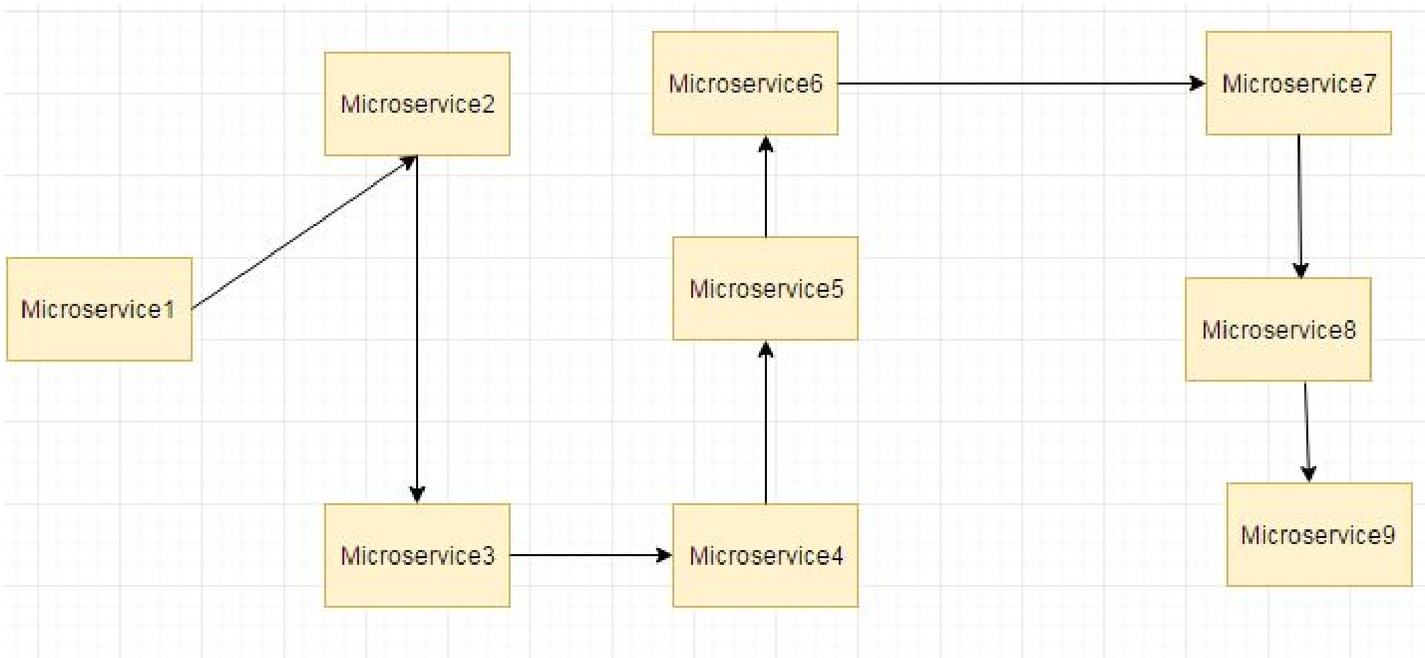
- **One of the major challenges in microservices is the ability to debug issues and monitor them. A simple action can trigger a chain of microservice calls and it would be tedious to trace these actions across the invoked microservices. This is because each microservice runs in an environment isolated from other microservices so they don’t share resources such as databases or log files. In addition to that, we might also want to track down why a certain microservice call is taking so much time in a given business flow.**

- **The Distributed Tracing pattern addresses the above challenges developers face while building microservices. There are some helpful open-source tools that can be used for distributed tracing, when creating microservices with Spring Boot and Spring Cloud frameworks.**

rgupta.mtech@gmail.com

Need of Distributed Log Tracing

- **Consider the following microservices-**
  - **If suppose during such calls there are some issues like exception has occurred.**
  - **Or may be there are latency issues due to a particular service taking more than expected time.**
- **How do we identify where the issue is occurring.**
  - **In regular project we would have used logging to analyze the logs to know more about occurred exceptions and also performance timing.**
  - **<u>But since microservices involves multiple services we cannot use regular logging</u>. Each Service will be having its own separate logs. So we will need to go through the logs of each service. Also how do we correlate the logs to a request call chain i.e which logs of microservices are related to Request1, which are related to Request2.**



rgupta.mtech@gmail.com

Need of Distributed Log Tracing: Tools

- **Spring Cloud Sleuth: A Spring Cloud library that lets you track the progress of subsequent microservices by adding trace and span id's on the appropriate HTTP request headers.**
- **It is used to generate trace id, span id, and this information to service calls in the headers and Mapping Diagnostic Context. So that it can be used by tools like Zipkin, ELK, etc to store indexes and process logs file.**

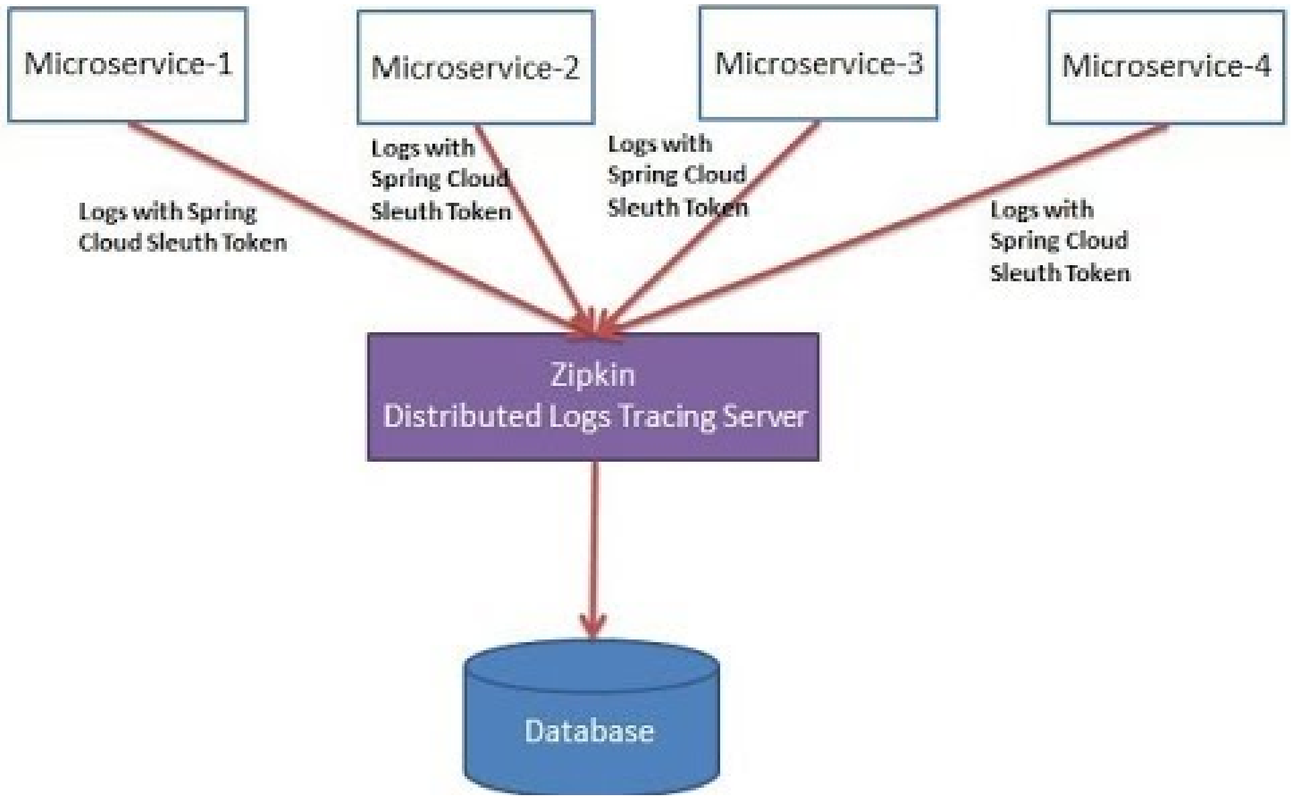
- **Zipkin: A Java-based distributed tracing application that helps gather timing data for every request propagated between independent services. It has a simple management console where we can find a visualization of the time statistics generated by subsequent services.**
- **Zipkin is a distributed tracing system. It helps gather timing data needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data.**
- **This is very useful during debugging when lots of microservices are implemented and the application becomes slow in any particular situation. In such a case, we first need to identify to see which underlying service is actually slow. Once the slow service is identified, we can work to fix that issue. Distributed tracing helps in identifying that slow component among in the ecosystem.**

rgupta.mtech@gmail.com

Need of Distributed Log Tracing

- **Spring Cloud Sleuth** is used to generate and attach the trace id, span id to the logs so that these can then be used by tools like Zipkin and ELK for storage and analysis

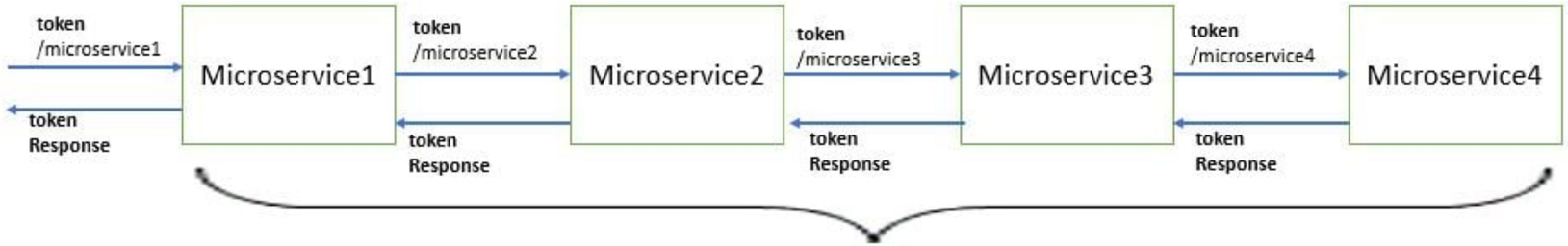
- **Zipkin** is an open-source project that provides mechanisms for sending, receiving, storing, and visualizing traces. This allows us to correlate activity between servers and get a much clearer picture of exactly what is happening in our application. Zipkin is a distributed tracing system



rgupta.mtech@gmail.com

Implementing Distributed Log Tracing with Spring boot

Spring Cloud Sleuth we will be adding a unique token to all requests. Spring Cloud Sleuth is used to generate and attach the trace id, span id to the logs so that these can then be used by tools like Zipkin and ELK for storage and analysis.



One Request Chain

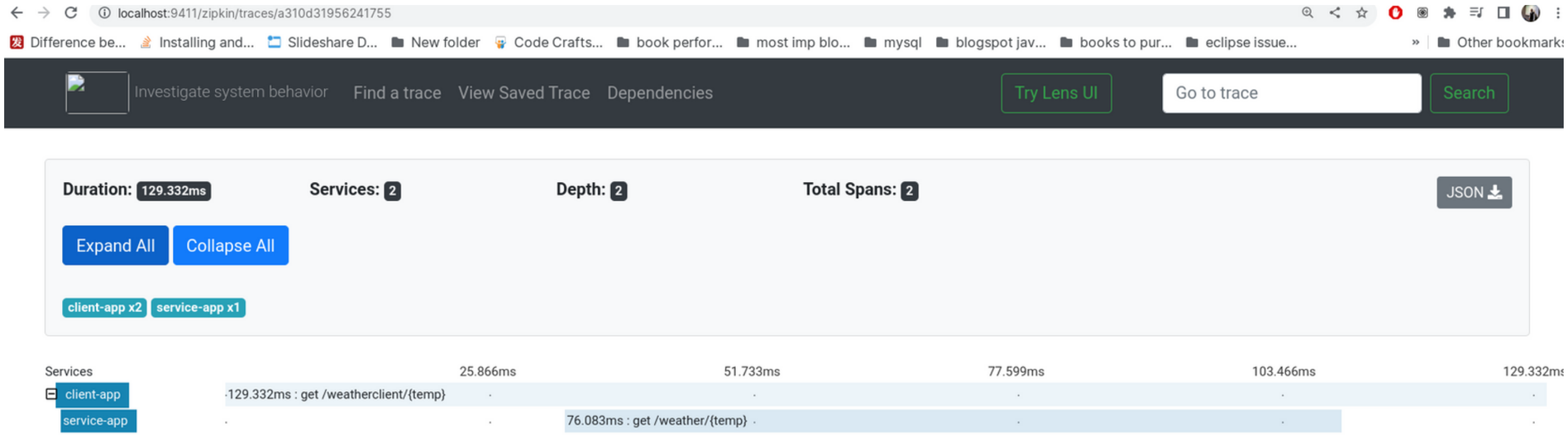
```
[microservice2,92bb6a46d4c8c452,640756ff93630bc1,true]
[microservice2,92bb6a46d4c8c452,640756ff93630bc1,true]
[microservice3,92bb6a46d4c8c452,d7a5a2098395f621,true]
[microservice3,92bb6a46d4c8c452,d7a5a2098395f621,true]
```

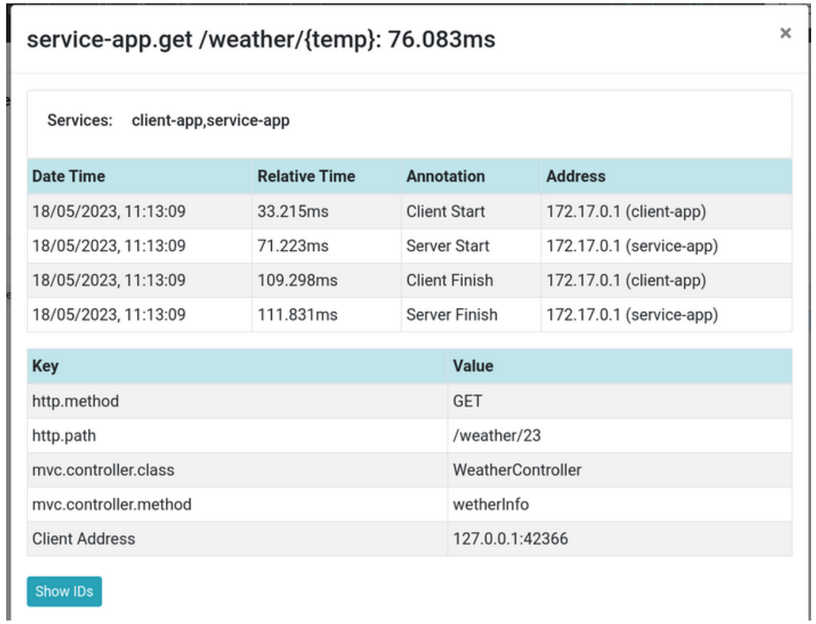
Application Name + TraceId + SpanId + Zipkin Export Flag

Application Name	TraceId	SpanId	Zipkin Export Flag
Name of the Application. Defined in application.properties	The Trace Id added by Sleuth. This Id is same in all services for a given request	The Span Id added by Sleuth. This Id is same in same unit of work (in our case same in same method) but different for different services for a given request	This Boolean value indicates whether the span should be exported to Zipkin

rgupta.mtech@gmail.com

Implementing Distributed Log Tracing with Spring boot





rgupta.mtech@gmail.com

Module 10: Grafana and Prometheus

rgupta.mtech@gmail.com

Introduction to application monitoring

- **Once we deploy our application in production env we want to provide best experience to the client**
- **We should get alter if something going wring in our application before client is going to be effected.**

- **Promethues**
  - **Prometheus is tool for server health monitoring + alter**
  - **can trigger customized alters**
  - **Differents matrix for measuring application healths such as**
    - **% of RAM consumption**
    - **% of CUP consumption**
    - **Heap memory usages**
    - **GC**
    - **Datasource ...**

- **Garfana**
  - **Grafana is a multi-platform open source analytics and interactive visualization software.**
  - **It provides charts, graphs, and alerts for the web when connected to supported data sources.**

rgupta.mtech@gmail.com

Introduction to application monitoring

- **A system monitor is a hardware or software component used to monitor system resources and performance in a computer. Its main goal is to analyze the computer's operation and performance, and to detect and alert about possible errors. Such systems give us the ability to find out what is happening on our machine at any given time.**
- **Whether it's the percentage of system's resources currently used, what commands are being run, or who is logged on.**
- **Running complex applications on actual servers is complicated and things can go haywire for several reasons such as :**
- **Disk Full : No new data can be store**
- **Software Bug : Request Errors**
- **High Temperature : Hardware Failure**
- **Network Outage : Services cannot communicate**
- **Low Memory Utilization : Money wasted**

- **Being able to monitor the performance of your system is essential. If system resources become too low it can cause a lot of problems. The ability to know what is happening can help determine whether system upgrades are needed, or if some services need to be moved to another machine.**

rgupta.mtech@gmail.com

Introduction to application monitoring

- **Promethues**
  - **Prometheus is a free software application used for event monitoring and alerting.**
  - **It records real-time metrics in a time series database built using a HTTP pull model, with flexible queries and real-time alerting.**
- **Garfana**
  - **Grafana is a multi-platform open source analytics and interactive visualization software.**
  - **It provides charts, graphs, and alerts for the web when connected to supported data sources.**

When a performance issue arises, there are 4 main areas to consider:
CPU, Memory, Disk I/O, and Network.

The ability to determine where the bottleneck is can save you a lot of time.

rgupta.mtech@gmail.com

What is Prometheus?

Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud.

Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community.

It is now a standalone open source project and maintained independently of any company.

To emphasize this, and to clarify the project's governance structure, Prometheus joined the Cloud Native Computing Foundation in 2016 as the second hosted project, after Kubernetes.

— prometheus.io

rgupta.mtech@gmail.com

What is Prometheus?

Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud.

Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community.

It is now a standalone open source project and maintained independently of any company.

To emphasize this, and to clarify the project's governance structure, Prometheus joined the Cloud Native Computing Foundation in 2016 as the second hosted project, after Kubernetes.

— prometheus.io

How it works?

it pulls (scrapes) metrics from a client (target) over http and places the data into its local time series database that you can query using its own DSL.

rgupta.mtech@gmail.com

What is Prometheus?

Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud.

Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community.

It is now a standalone open source project and maintained independently of any company.

To emphasize this, and to clarify the project's governance structure, Prometheus joined the Cloud Native Computing Foundation in 2016 as the second hosted project, after Kubernetes.

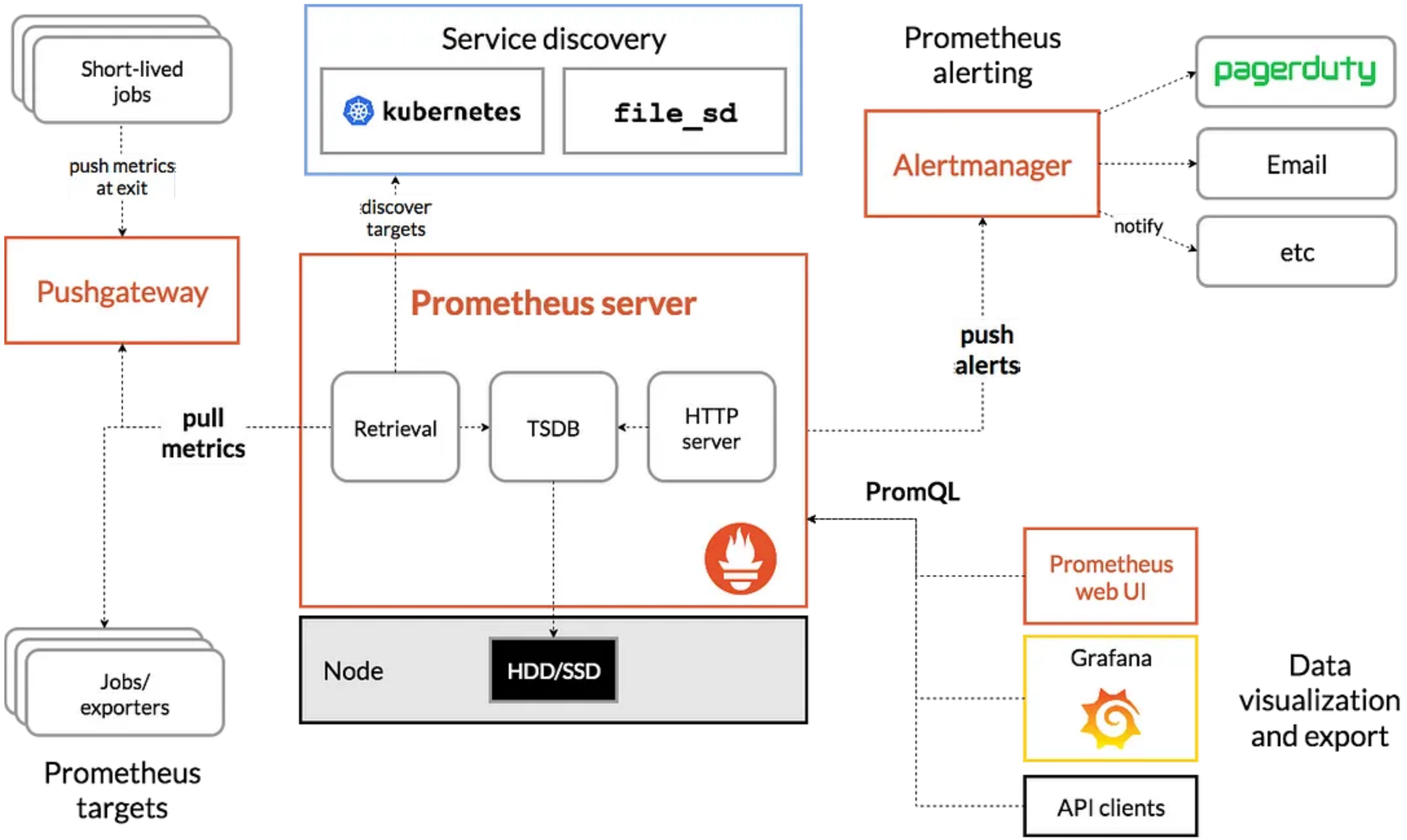
— prometheus.io

How it works?

it pulls (scrapes) metrics from a client (target) over http and places the data into its local time series database that you can query using its own DSL.

rgupta.mtech@gmail.com

Prometheus architecture



Component of Prometheus?

- **Prometheus Server : The main server that scrapes and stores the scraped metrics in a time series DB.**
- **Scrape : Prometheus server uses a pulling method to retrieve metrics.**
- **Target : The Prometheus servers clients that it retrieves info from.**
- **Exporter : Target libraries that convert and export existing metrics into Prometheus format.**
- **Alert Manager : Component responsible for handling alerts.**

rgupta.mtech@gmail.com

Module 11: Spring boot ELK

Elasticsearch, Logstash, and Kibana

rgupta.mtech@gmail.com

What is ELK? Need for it?

The ELK Stack consists of three open-source products

- **Elasticsearch**
- **Logstash**
- **Kibana**

Elasticsearch is a NoSQL database that is based on the Lucene search engine.

Logstash is a log pipeline tool that accepts inputs from various sources, executes different transformations, and exports the data to various targets. It is a dynamic data collection pipeline with an extensible plugin ecosystem and strong Elasticsearch synergy

Kibana is a visualization UI layer that works on top of Elasticsearch.

- **These three projects are used together for log analysis in various environments. So Logstash collects and parses logs, Elastic search indexes and store this information while Kibana provides a UI layer that provide actionable insights.**

rgupta.mtech@gmail.com

Use Cases for ELK?

Use Cases-

- **Consider you have a single application running and it produces logs. Now suppose you want analyze the logs generated. One option is to manually analyze them. But suppose these logs are large, then manually analyzing them is not feasible.**

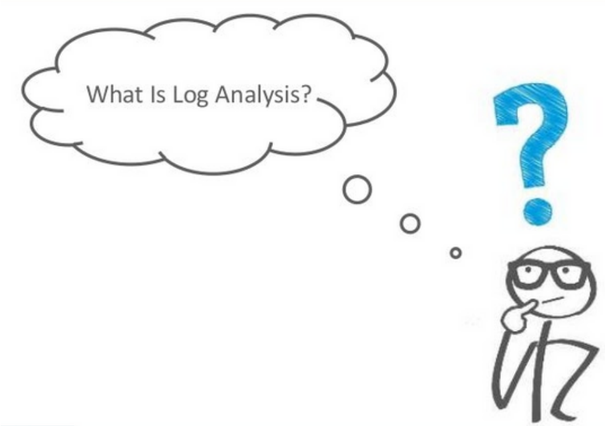
- **Suppose we have multiple Application running and all these applications produce logs. If we have to analyze the logs manually we will need to go through all the log files. These may run into hundreds.**

We can use ELK here to analyze the logs more efficiently and also using more complex search criterias. It provides log aggregation and efficient searching.

-

rgupta.mtech@gmail.com

Understanding Log analysis basics



rgupta.mtech@gmail.com

What is log analysis?

Log Analysis Is The Process Of Analyzing The Computer/ Machine Generated Data

rgupta.mtech@gmail.com

Need of Log analysis?

Issue Debugging

Predictive Analysis

Security Analysis

Performance Analysis

Internet of Things & Debugging

rgupta.mtech@gmail.com

Problem with log analysis ?

1. Non-consistent log format
2. Non-consistent time format
3. Decentralized logs
4. Expert knowledge requirement

rgupta.mtech@gmail.com

Problem with log analysis ?

1. Non-consistent log format
2. Non-consistent time format
3. Decentralized logs
4. Expert knowledge requirement

Tomcat Logs

```
ffMay 24, 2015 3:56:26 PM org.apache.catalina.startup.HostConfig deployWAR
INFO: Deployment of web application archive \soft\apache-tomcat-7.0.62\webapps\sample.war
has finished in 253 ms
```

Apache Access Logs

```
127.0.0.1 - - [24/May/2015:15:54:59 +0530] "GET /favicon.ico HTTP/1.1" 200 21630
```

IIS Logs

```
2012-05-02 17:42:15 172.24.255.255 - 172.20.255.255 80 GET /images/favicon.ico - 200
Mozilla/4.0+(compatible;MSIE+5.5;+Windows+2000+Server)
```

Problem with log analysis ?

1. Non-consistent log format
2. Non-consistent time format
3. Decentralized logs
4. Expert knowledge requirement

- 142920788
- Oct 12 23:21:45
- [5/May/2015:08:09:10 +0000]
- Tue 01-01-2009 6:00
- 2015-05-30 T 05:45 UTC
- Sat Jul 23 02:16:57 2014
- 07:38, 11 December 2012 (UTC)

rgupta.mtech@gmail.com

Problem with log analysis ?

1. Non-consistent log format
2. Non-consistent time format
3. Decentralized Logs
4. Expert knowledge requirement

rgupta.mtech@gmail.com

Problem with log analysis ?

1. Non-consistent log format
2. Non-consistent time format
3. Decentralized logs
4. **Expert Knowledge Requirement**

- Everyone do not have access to the logs
- General people might not have technical expertise to understand the information
- This can slow down the analysis process

rgupta.mtech@gmail.com

Log Management Tools ?

rgupta.mtech@gmail.com

What is ELK?

ELK Stack is a combination of **three** open source tools which forms a **log management tool/ platform**, that helps in deep ***searching***, ***analyzing*** and ***visualizing*** the log generated from different machines

What is ELK Stack: ElasticSearch

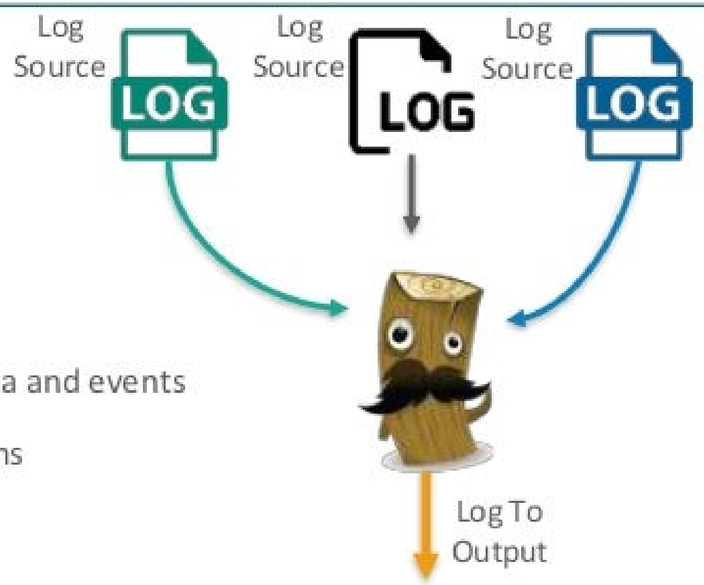
Features

- ✓ search engine/ search server
- ✓ NoSQL database i.e. can't use SQL for queries.
- ✓ Based on Apache Lucene and provides RESTful API
- ✓ Provides horizontal scalability, reliability and multenant capability for real time search
- ✓ Uses indexes to search which makes it faster

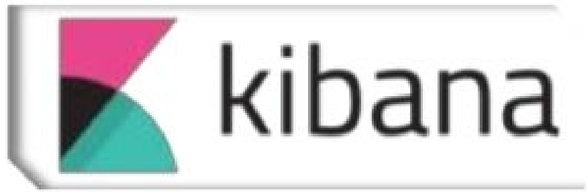
What is ELK Stack: LogStash

Features

- ✓ Data pipeline tool
- ✓ Centralizes the data processing
- ✓ Collects, parses and analyzes large variety of structured/ unstructured data and events
- ✓ Provides plugins to connect to various types of input sources and platforms



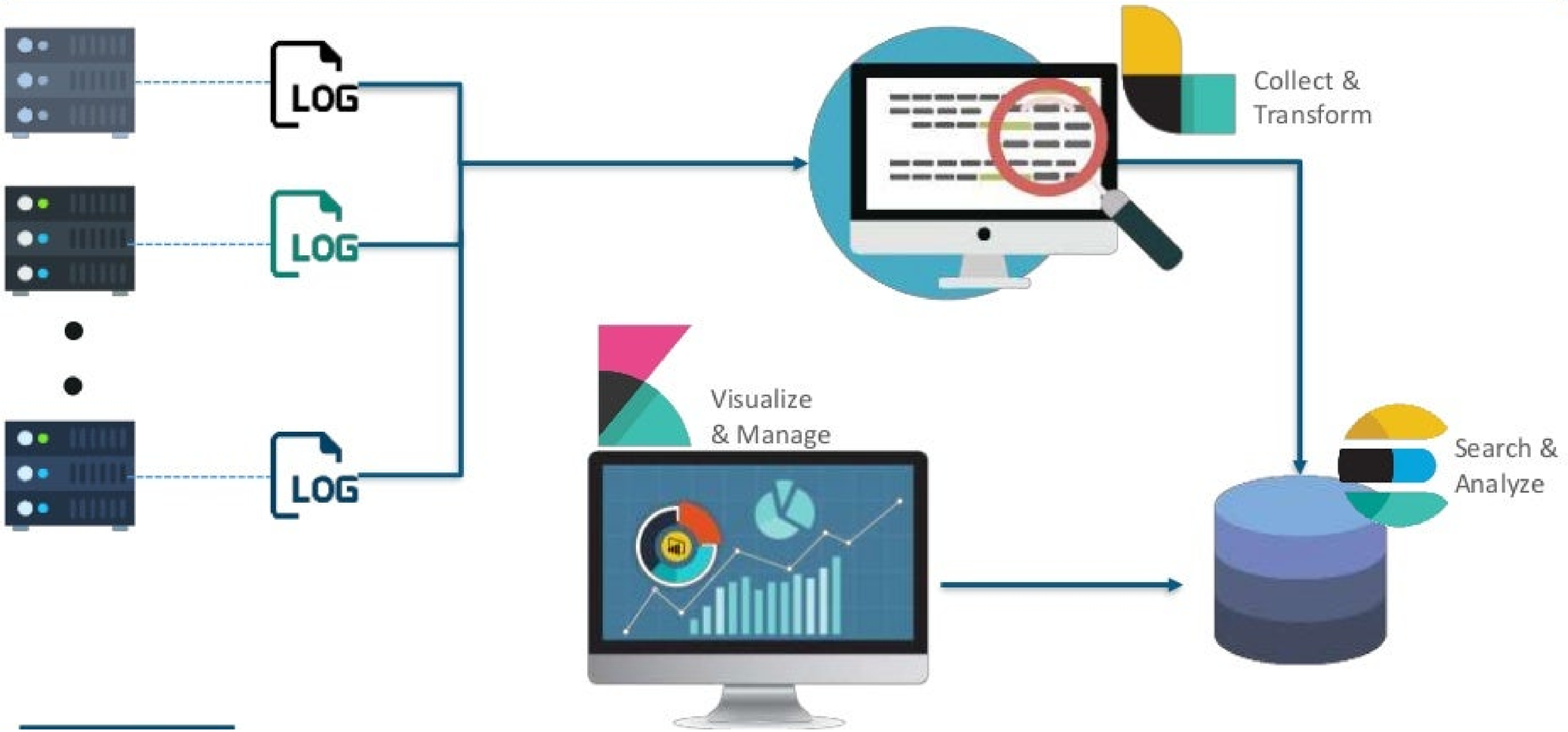
What is ELK Stack: Kibana



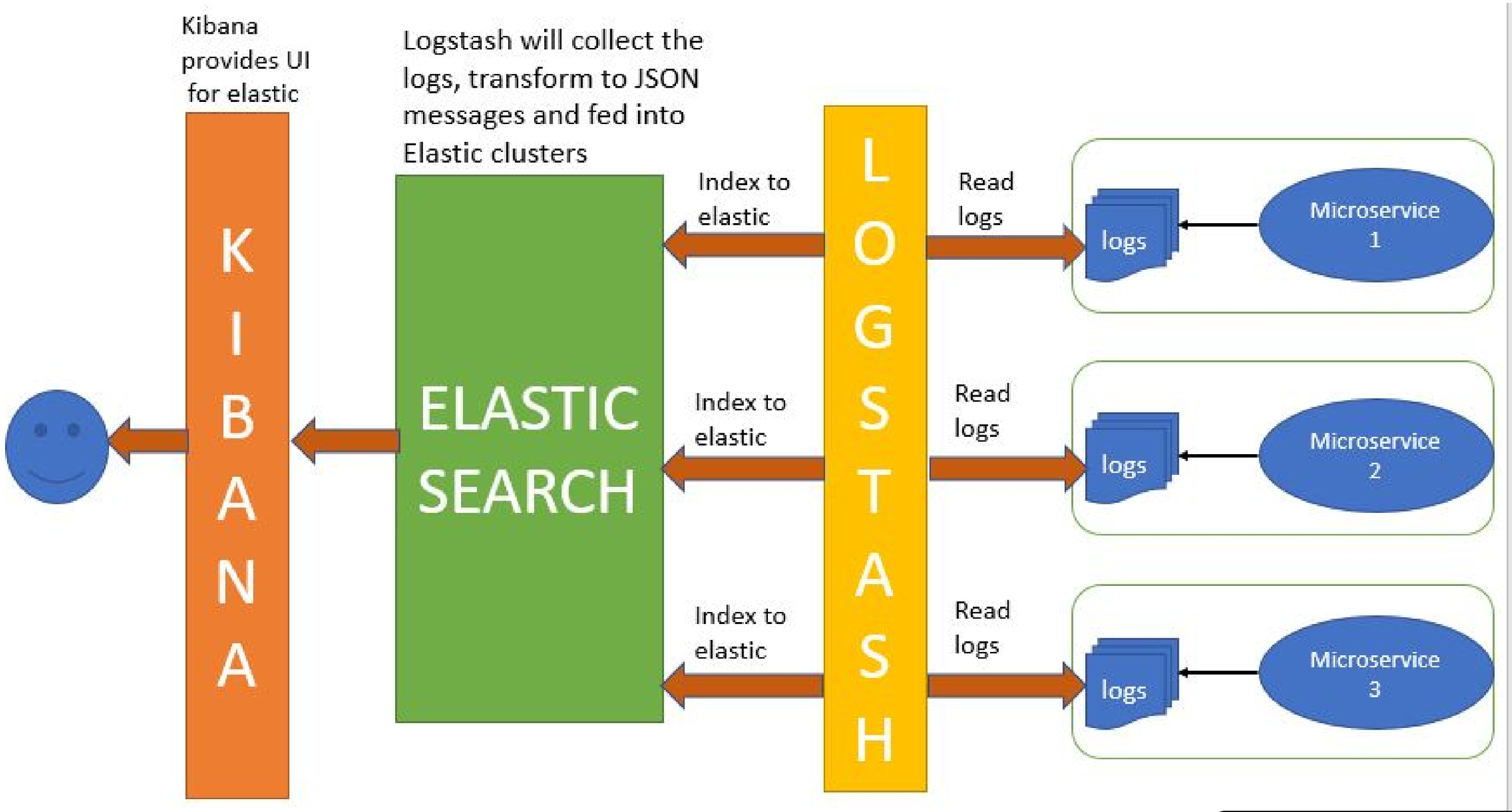
Features

- ✓ Visualization tool
- ✓ provides real-time analysis, summarization, charting, and debugging capabilities.
- ✓ Provides instinctive and user friendly interface
- ✓ Allows sharing of snapshots of the logs searched through.
- ✓ Permits saving the dashboard and managing multiple dashboards

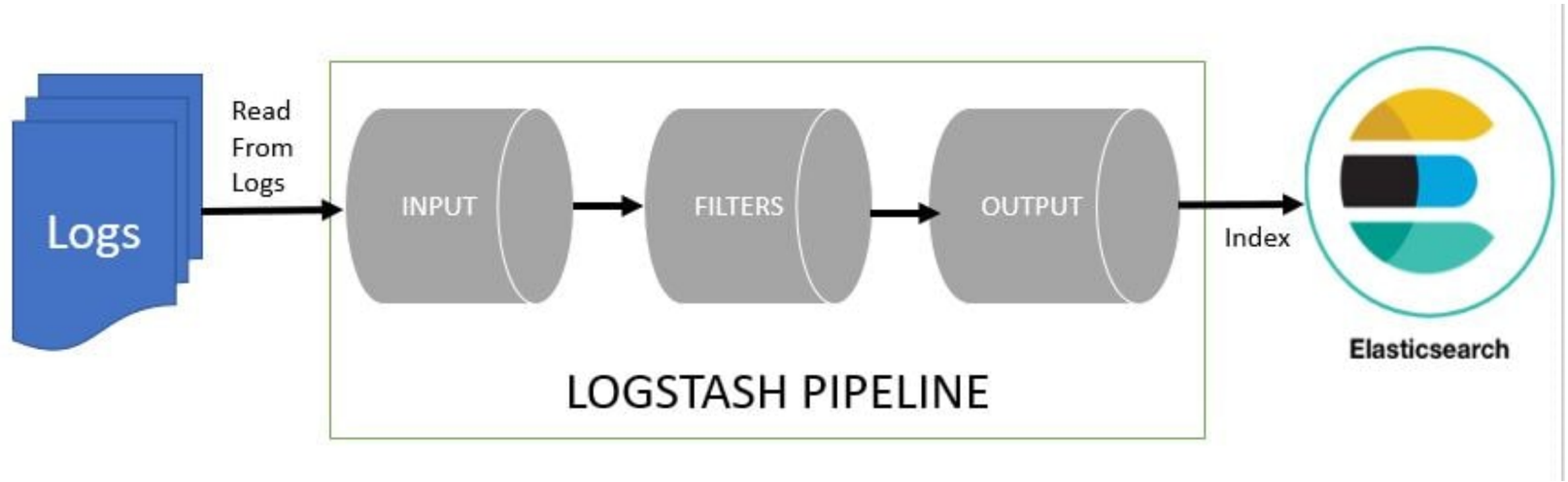
How ELK Stack Works?



Use Cases for ELK?



Spring Boot ELK



Module 12: Messaging with RabitMQ

rgupta.mtech@gmail.com

Agenda

- What is Messaging Queue, How it works and its uses
- What is Rabbit MQ
- Different types of Exchanges in Rabbit, MQ
- What is Messaging Queue, How it works and its uses
- Different types of Exchanges in Rabbit MQ .
- Direct Exchanges
- Fanout Exchanges
- Topic Exchanges
- Header Exchanges
- Retry and Error Handling Example

rgupta.mtech@gmail.com

What is RabbitMQ ?

What is RabbitMQ ?

- **RabbitMQ is a message broker that originally implements the Advance Message Queuing Protocol (AMQP)**

AMQP

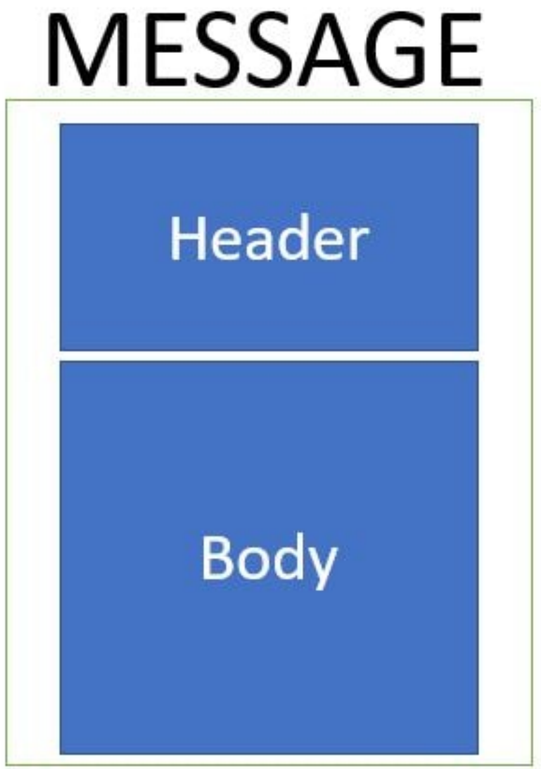
- **AMQP standardizes messaging using Producers, Broker and Consumers.**
- **AMQP standards was designed with the following main characteristics Security, Reliability, Interoperability**
- **Reliability confirms the message was successfully delivered to the message broker and confirms that the message was successfully processed by the consumer**

rgupta.mtech@gmail.com

What is Messaging?

Messaging is a communication mechanism used for system interactions. In software development messaging enables distributed communication that is loosely coupled.

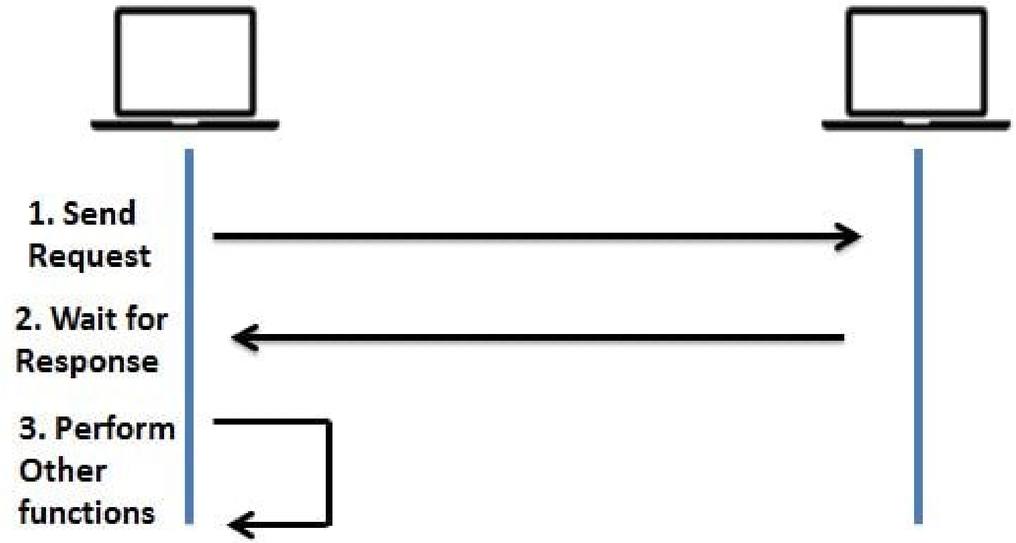
A messaging client can send messages to, and receive messages from, any other client. The structure of message can be defined as follows



rgupta.mtech@gmail.com

Synchronous Messaging

- **Synchronous Messaging is implemented when such that the messaging client sends a message and expects the response immediately. So the sender client waits for the response before he can execute the next task. So until and unless the message is recieved the sender is blocked.**

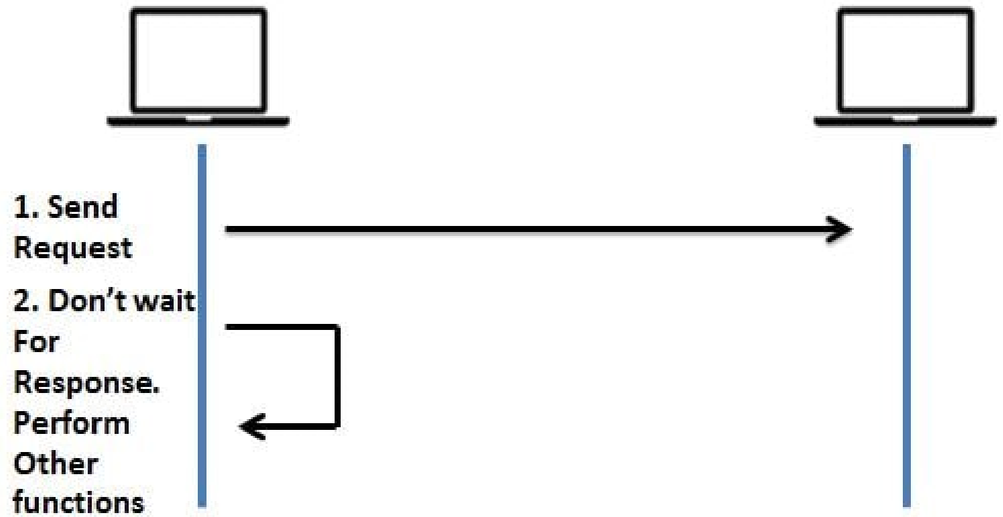


Synchronous Communication

rgupta.mtech@gmail.com

Asynchronous Messaging

Asynchronous Messaging is implemented such that the messaging client sends a message and does not expect the response immediately. So the sender client does not for the response before he can execute the next task. So the sender is not blocked

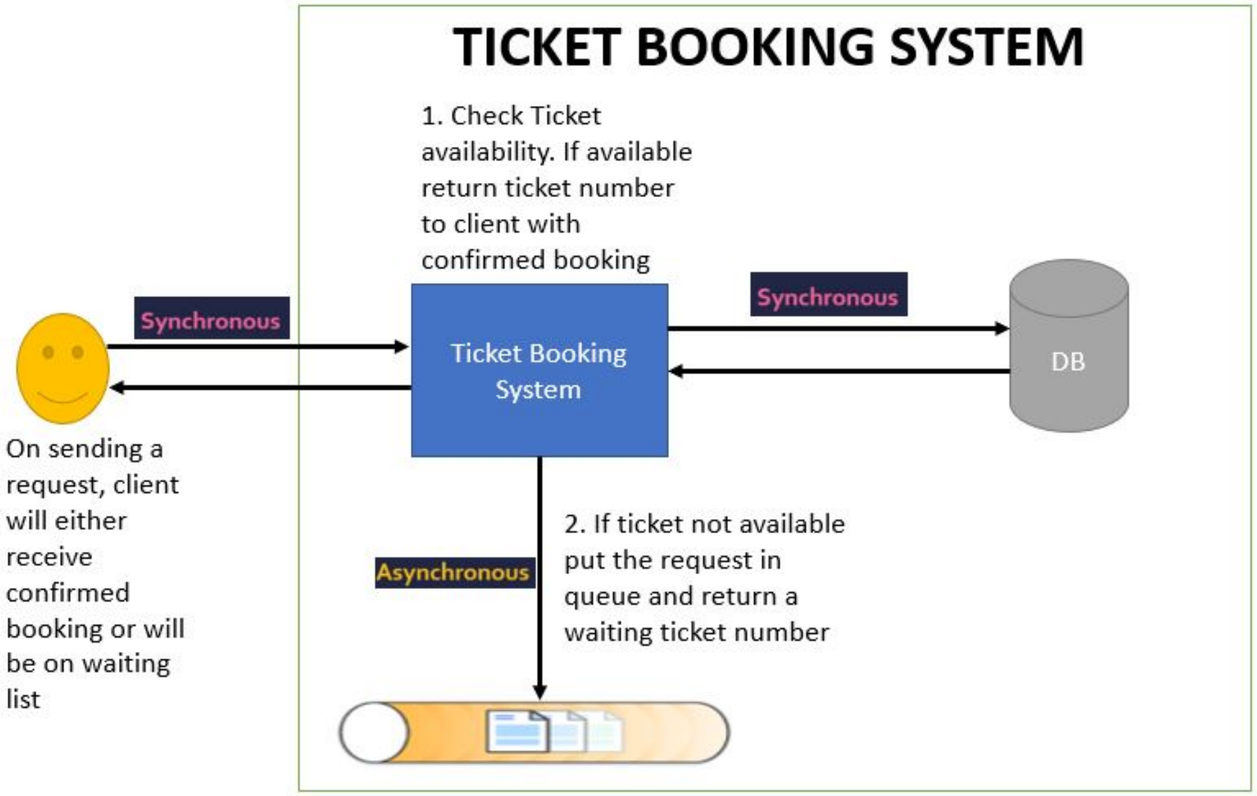


Asynchronous Communication

rgupta.mtech@gmail.com

Real time systems

- **Real time systems usually have a combination of syschronous and ascynchronous communication**

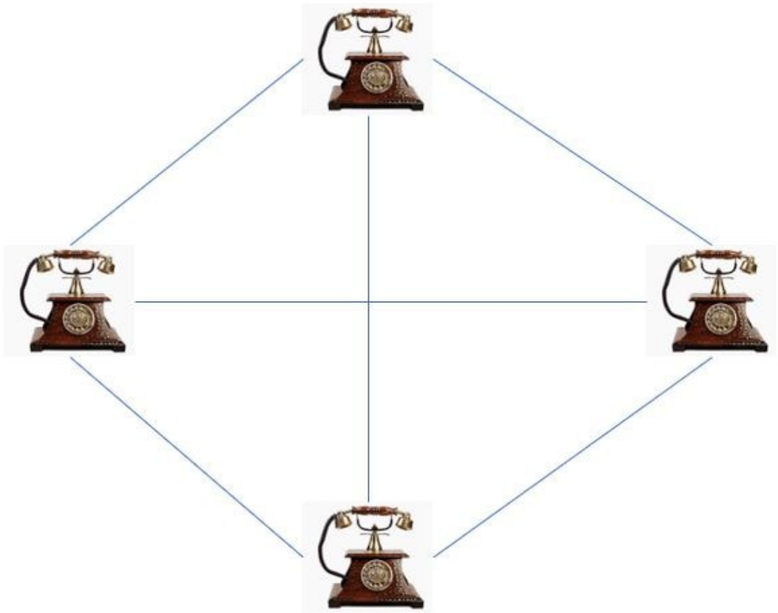


rgupta.mtech@gmail.com

Message Broker

Message Broker - are responsible for establishing connections with various client systems.

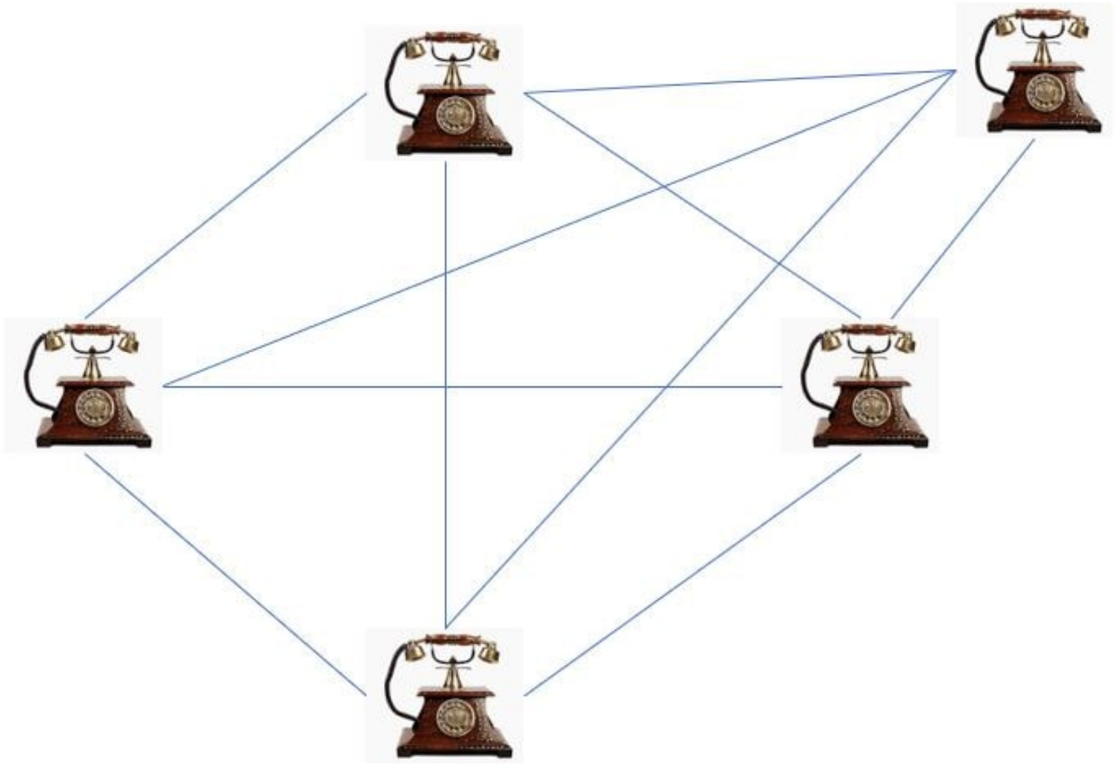
- **Let us consider role of Message Broker in a telecom system. Suppose initially there is no message broker. Then each telephone connection will have a direct line with all other telephone connections.**



rgupta.mtech@gmail.com

Message Broker

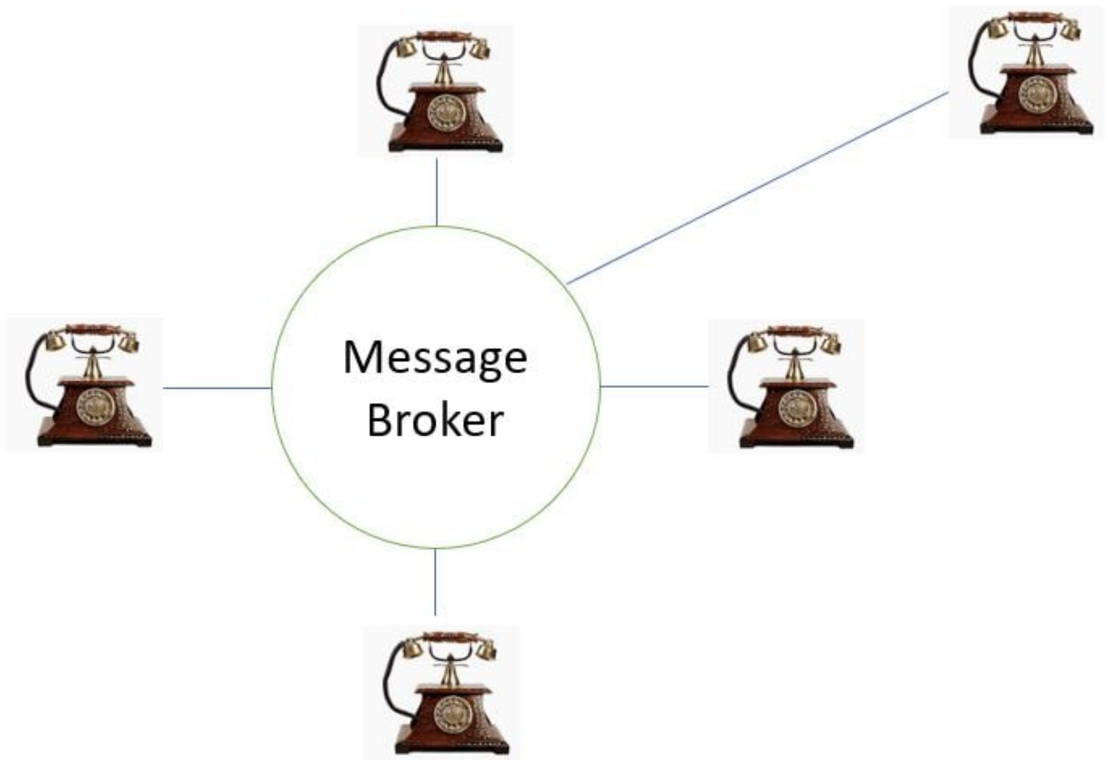
- **Suppose tomorrow if another telephone connection needs to be added, then all existing telephone connections will need to get a direct line with this new telephone connection. As more connections are added this will get more complicated.**



rgupta.mtech@gmail.com

Message Broker

- **With Message broker all connections are registered with Message Broker. So all connections only need to connect to the message broker. It will automatically route the message to the correct client based on some message configuration.**



rgupta.mtech@gmail.com

Rabbit mq installation ubuntu

```
echo "deb http://www.rabbitmq.com/debian/ testing main" | sudo tee -a
/etc/apt/sources.list
echo "deb http://packages.erlang-solutions.com/ubuntu wheezy contrib" | sudo tee -a
/etc/apt/sources.list
wget http://packages.erlang-solutions.com/ubuntu/erlang_solutions.asc
sudo apt-key add erlang_solutions.asc
sudo apt-get update
sudo apt-get -y install erlang erlang-nox
sudo apt-get -y --force-yes install rabbitmq-server
```

rgupta.mtech@gmail.com

Rabbit mq installation ubuntu

 # Enable the web interface
sudo rabbitmq-plugins enable rabbitmq_management
sudo service rabbitmq-server restart
ss -tunelp | grep 1567

sudo service rabbitmq-server start
sudo service rabbitmq-server stop
http://localhost:15672

https://stackoverflow.com/questions/8808909/simple-way-to-install-rabbitmq-in-ubuntu

rgupta.mtech@gmail.com

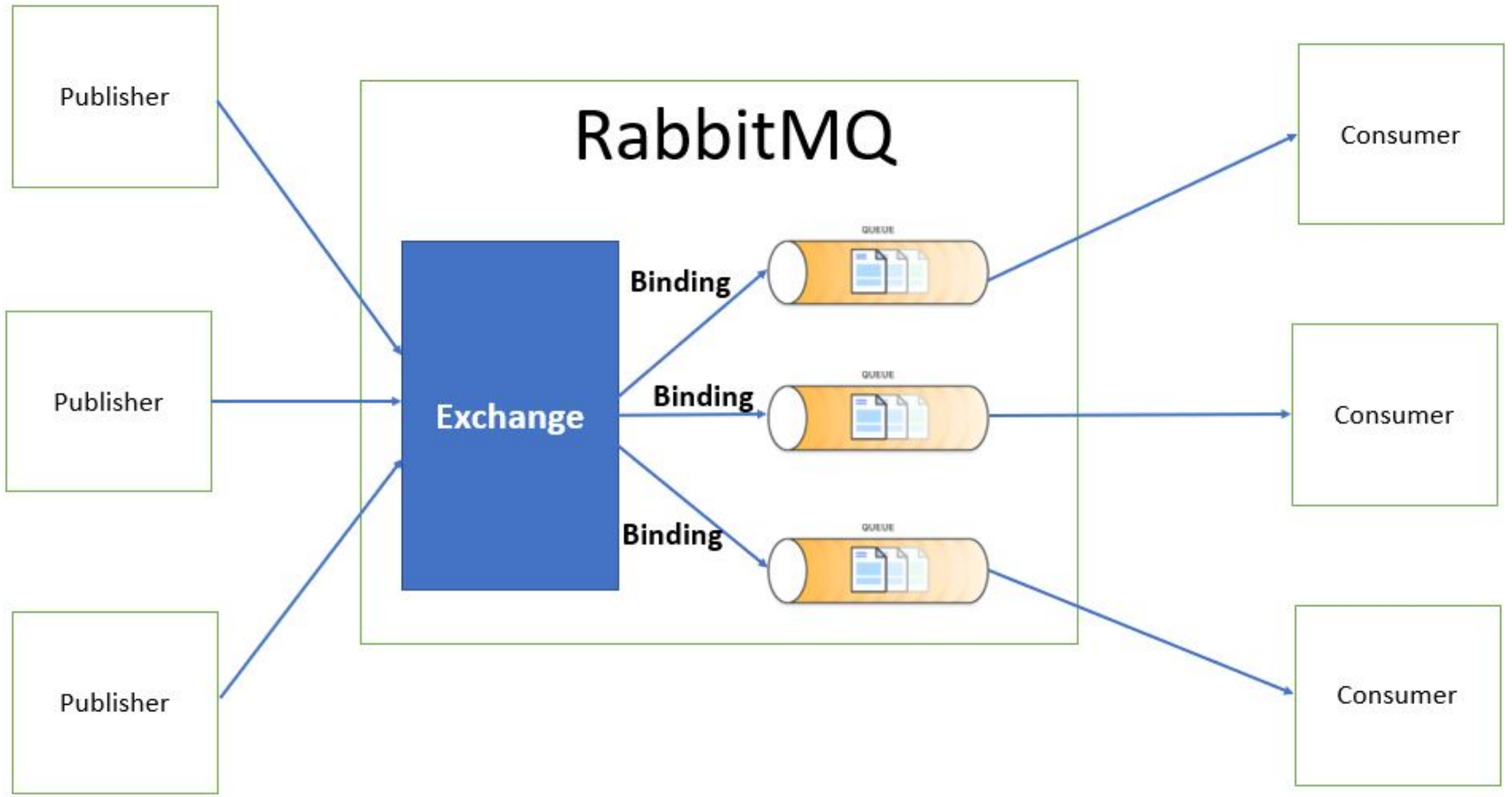
RabbitMQ Messaging Flow

When using RabbitMQ the publisher never directly sends a message to a queue. Instead, the publisher sends messages to an exchange.

Exchange is responsible for sending the message to an appropriate queue based on routing keys, bindings and header attributes.

Exchanges are message routing agents which we can define and bindings are what connects the exchanges to queues.

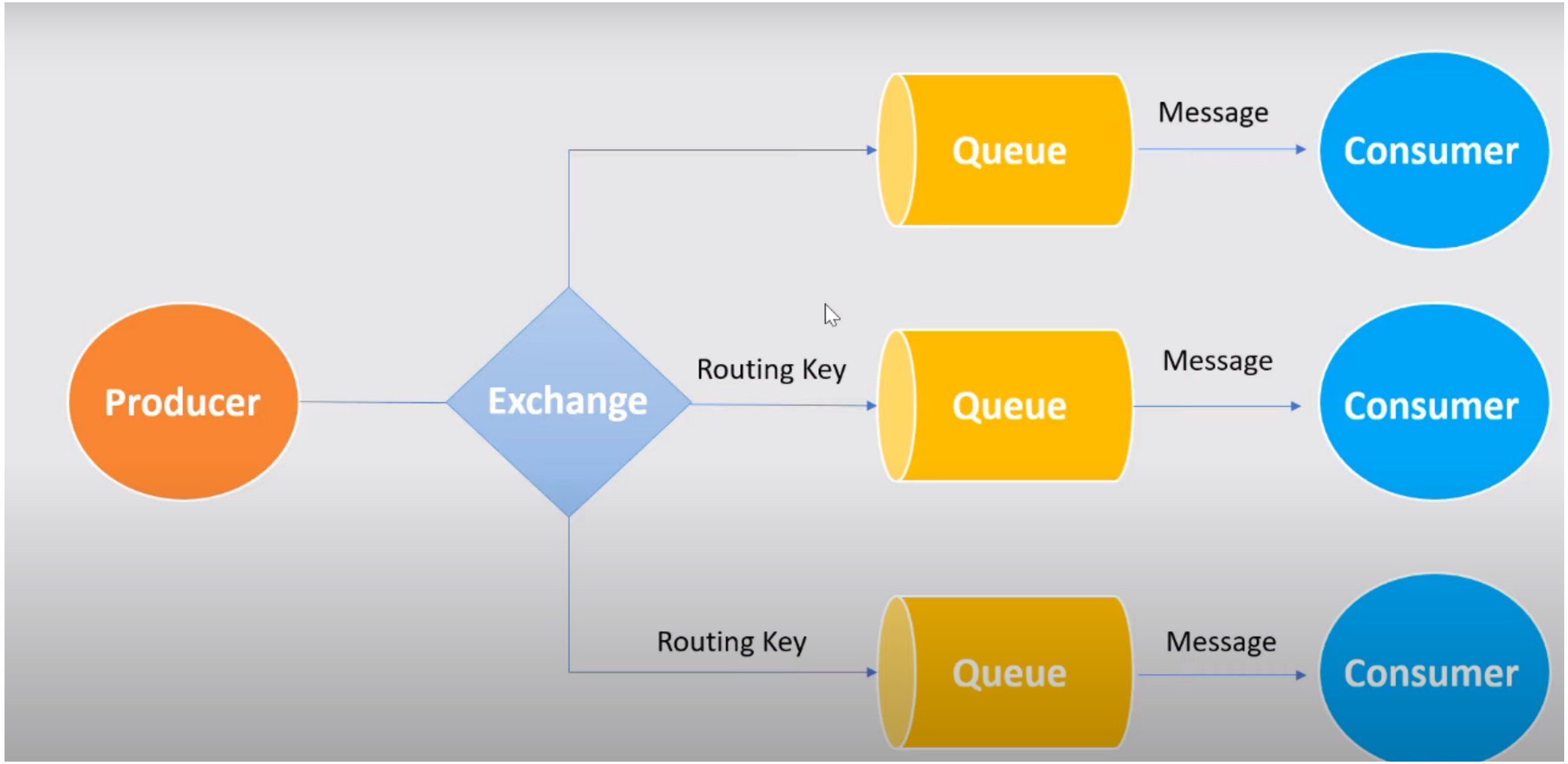
So in all our examples we will be creating first a Queue and Exchange, then bind them together.



rgupta.mtech@gmail.com

Spring boot rabbitMQ hello world

Create spring boot project with web, ampq dependencies



rgupta.mtech@gmail.com

Configuration rabbitmq

```java
@Configuration
public class MessagingConfig {
    public static final String QUEUE = "javademo_queue";
    public static final String EXCHANGE = "javademo_exchange";
    public static final String ROUTING_KEY = "javademo_routingKey";

    @Bean
    public Queue queue() {
        return new Queue(QUEUE);
    }
    @Bean
    public TopicExchange exchange() {
        return new TopicExchange(EXCHANGE);
    }

    @Bean
    public Binding binding(Queue queue, TopicExchange exchange) {
        return BindingBuilder.bind(queue).to(exchange).with(ROUTING_KEY);
    }

    @Bean
    public MessageConverter converter() {
        return new Jackson2JsonMessageConverter();
    }
    @Bean
    public AmqpTemplate template(ConnectionFactory connectionFactory) {
        final RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
        rabbitTemplate.setMessageConverter(converter());
        return rabbitTemplate;
    }
```

Request and response objects

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class Order {

    private String orderId;
    private String name;
    private int qty;
    private double price;
}
```

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class OrderStatus {

    private Order order;
    private String status;//progress,completed
    private String message;
}
```

rgupta.mtech@gmail.com

Order Producer

```java
@RestController
@RequestMapping("/order")
public class OrderPublisher {

    @Autowired
    private RabbitTemplate template;

    @PostMapping("/{restaurantName}")
    public String bookOrder(@RequestBody Order order, @PathVariable String restaurantName) {
        order.setOrderId(UUID.randomUUID().toString());
        //restaurantservice
        //payment service
        OrderStatus orderStatus = new OrderStatus(order, "PROCESS", "order placed succesfully in " + restaurantName);
        template.convertAndSend(MessagingConfig.EXCHANGE, MessagingConfig.ROUTING_KEY, orderStatus);
        return "Success !!";
    }
}
```

rgupta.mtech@gmail.com

Order Consumer

```
@Component
public class OrderConsumer {

    @RabbitListener(queues = MessagingConfig.QUEUE)
    public void consumeMessageFromQueue(OrderStatus orderStatus) {
        System.out.println("Message recieved from queue : " + orderStatus);
    }
}
```

rgupta.mtech@gmail.com

RabbitMQ types of Exchanges

With RabbitMQ we have the following types of Exchanges-

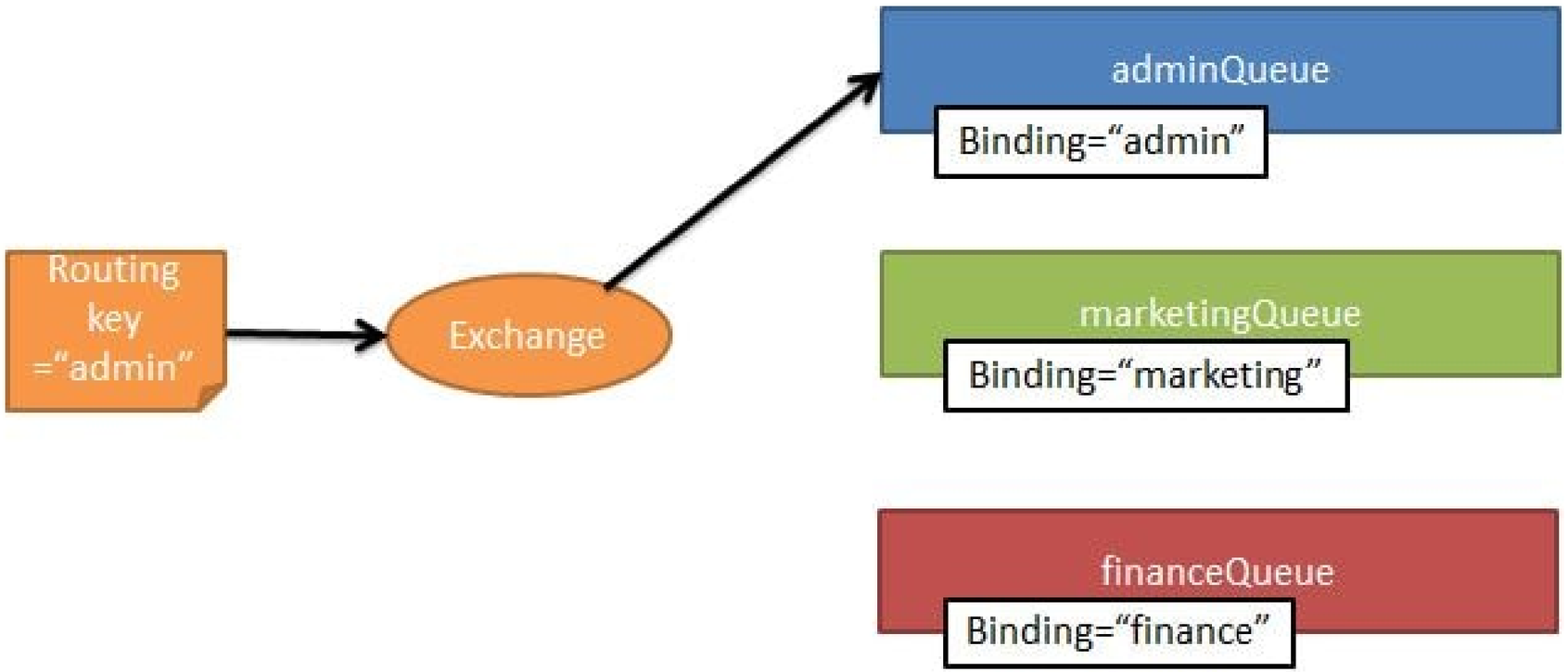
Direct Exchange
Fanout Exchange
Topic Exchange
Header Exchange

rgupta.mtech@gmail.com

Direct Exchange

Based on the routing key a message is sent to the queue having the same routing key specified in the binding rule.

The routing key of exchange and the binding queue have to be an exact match. A message is sent to exactly one queue.



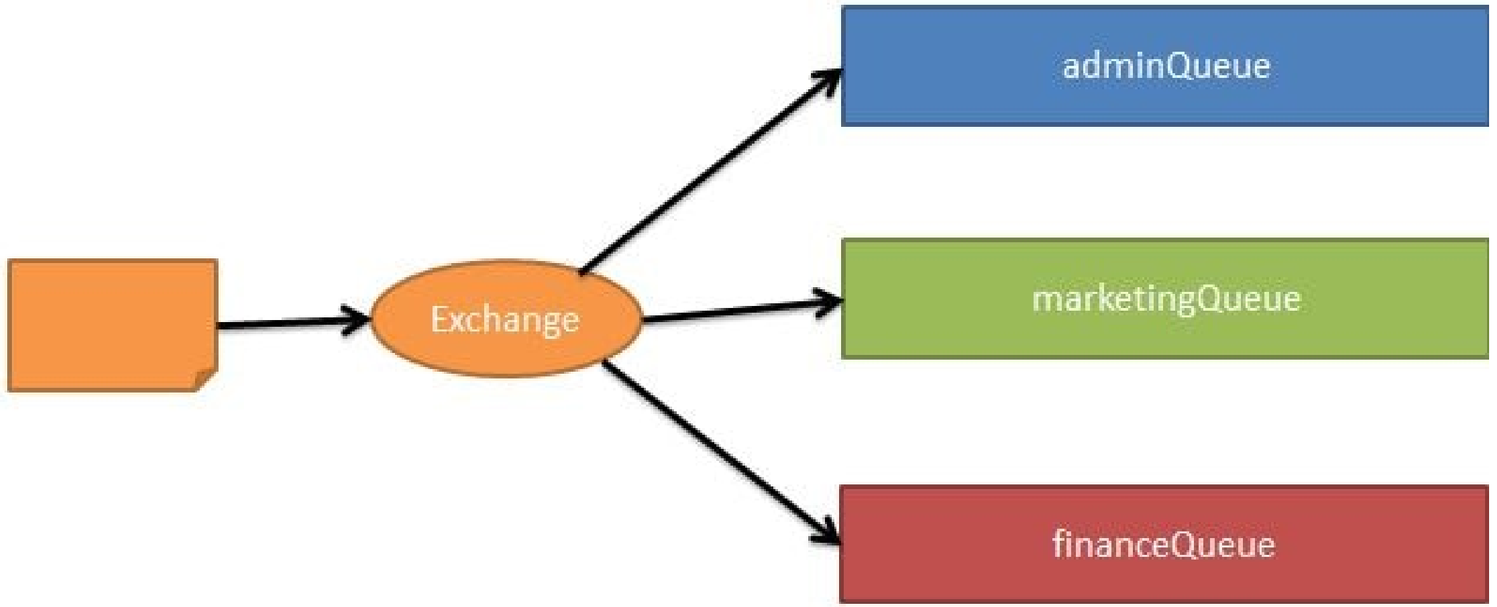
rgupta.mtech@gmail.com

Fanout Exchange

The message is routed to all the available bounded queues.

The routing key if provided is completely ignored.

So this is a kind of publish-subscribe design pattern.



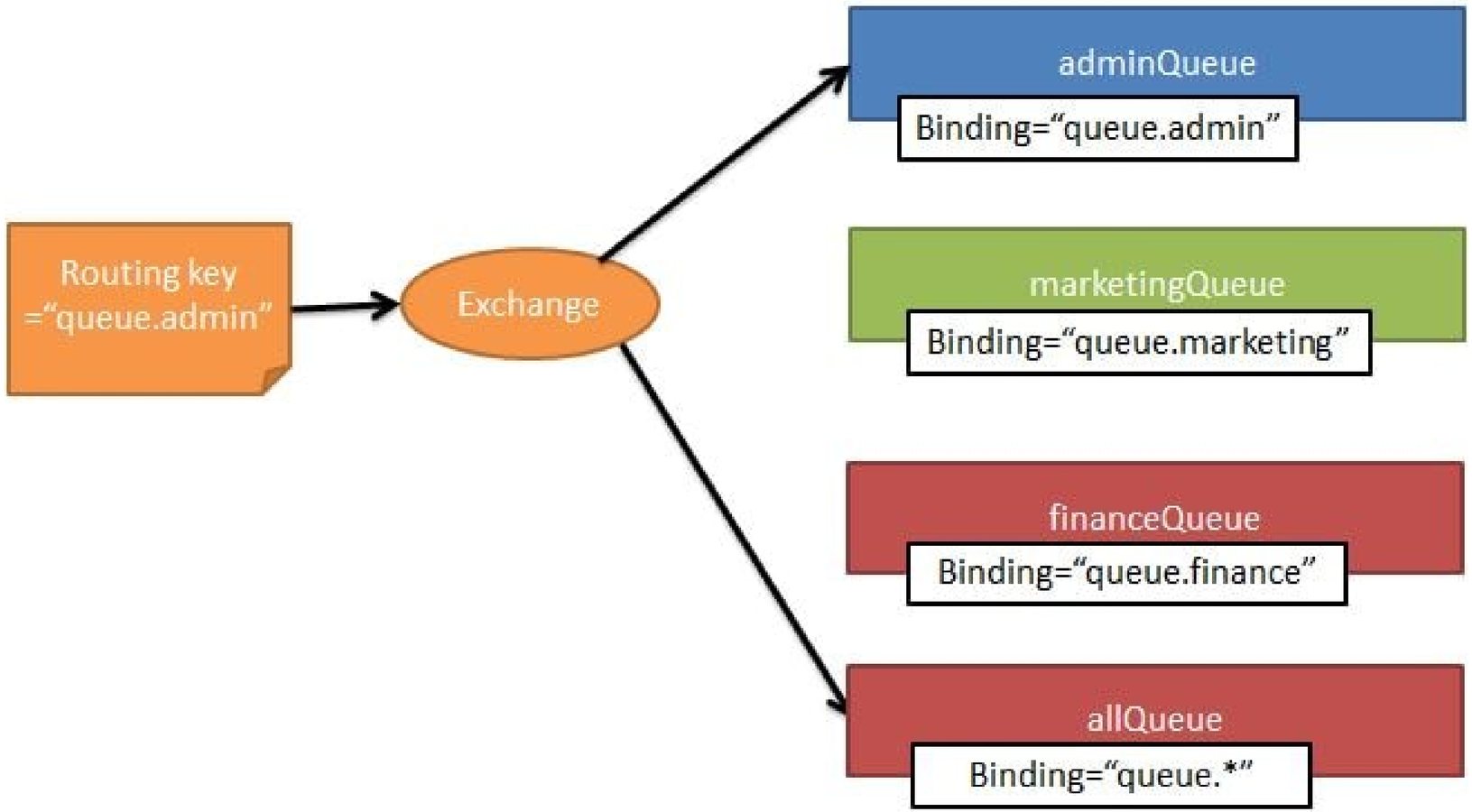
rgupta.mtech@gmail.com

Topic Exchange

Here again the routing key is made use of.

But unlike in direct exchange type, here the routing key of the exchange and the bound queues should not necessarily be an exact match.

Using regular expressions like wildcard we can send the exchange to multiple bound queues.

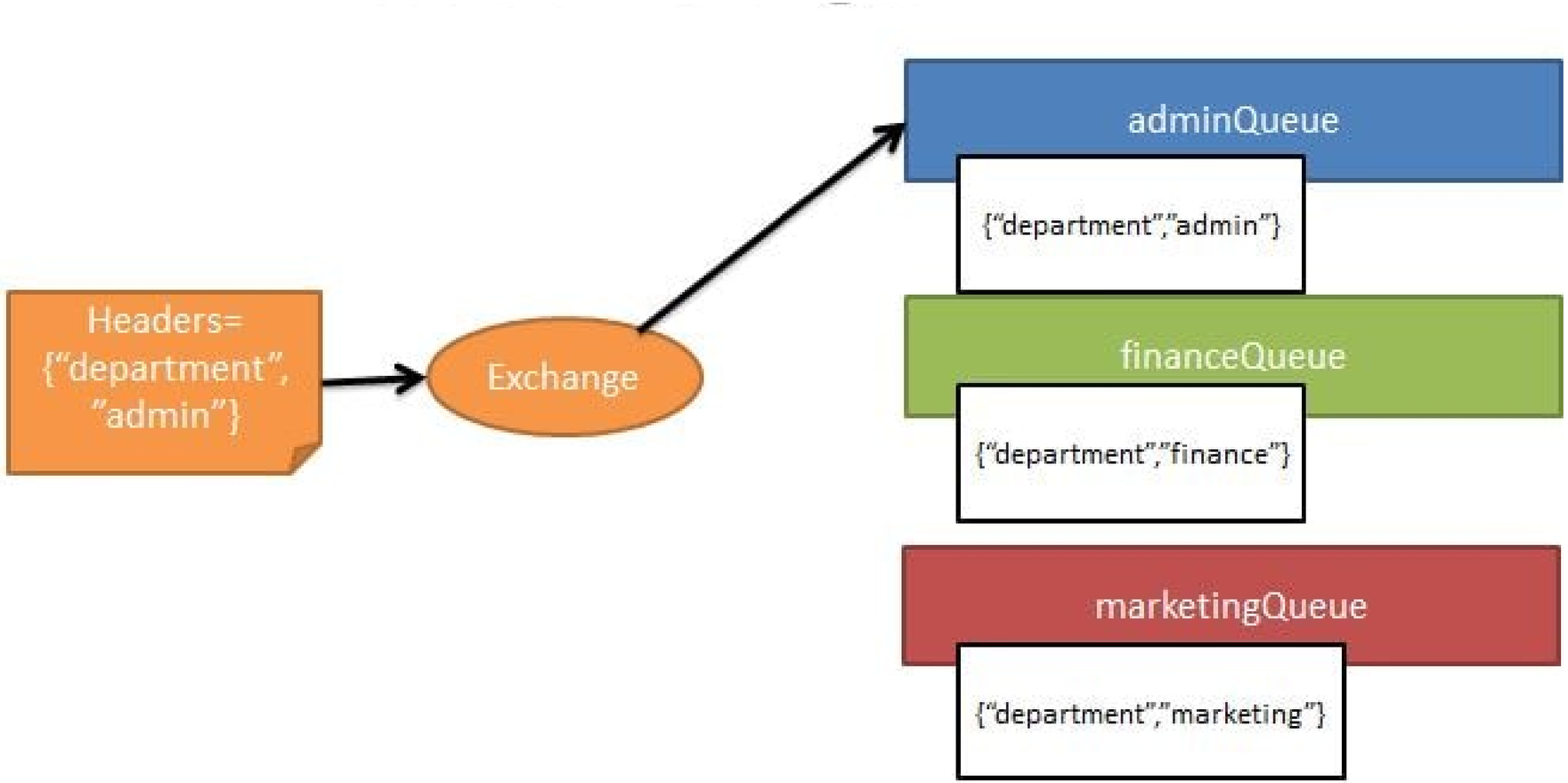


rgupta.mtech@gmail.com

Header Exchange

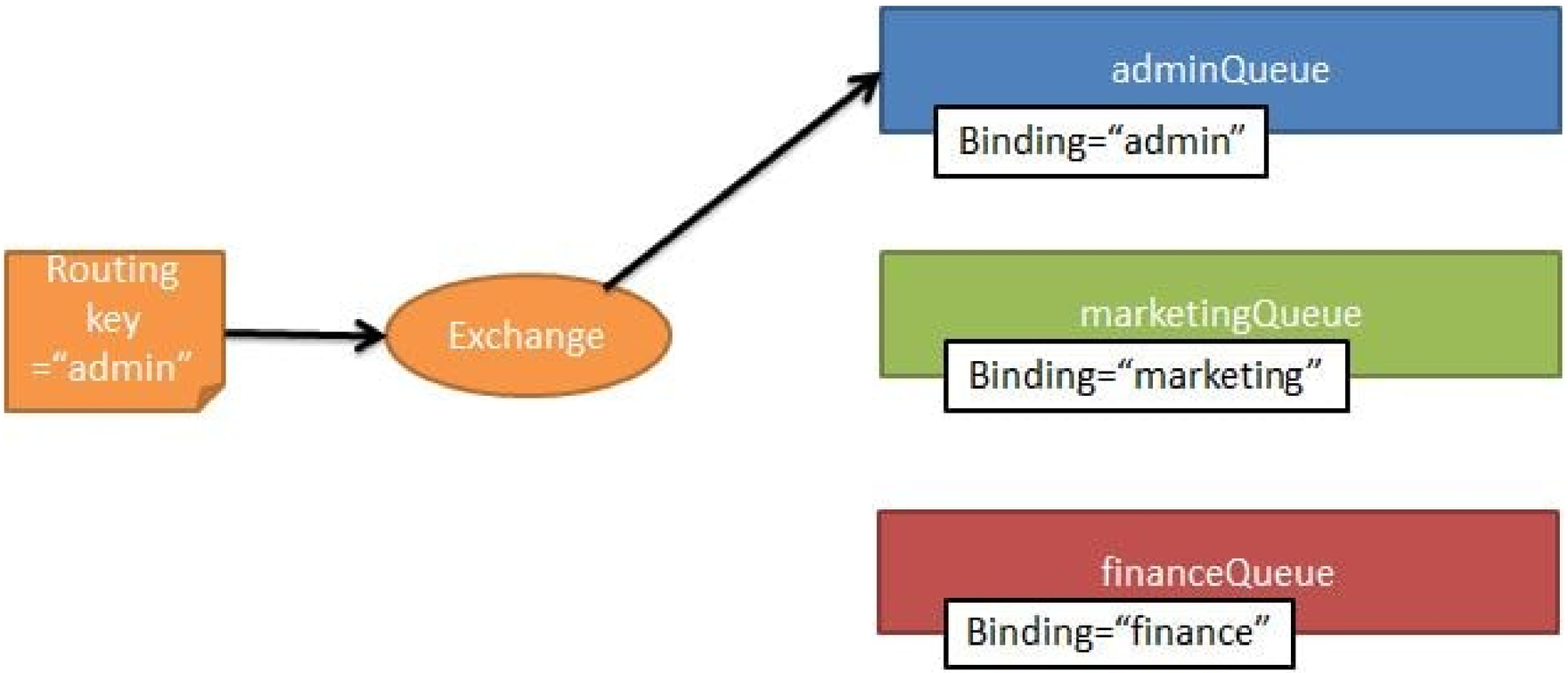
In this type of exchange the routing queue is selected based on the criteria specified in the headers instead of the routing key.

This is similar to topic exchange type, but here we can specify complex criteria for selecting routing queues.



rgupta.mtech@gmail.com

Direct exchange in details



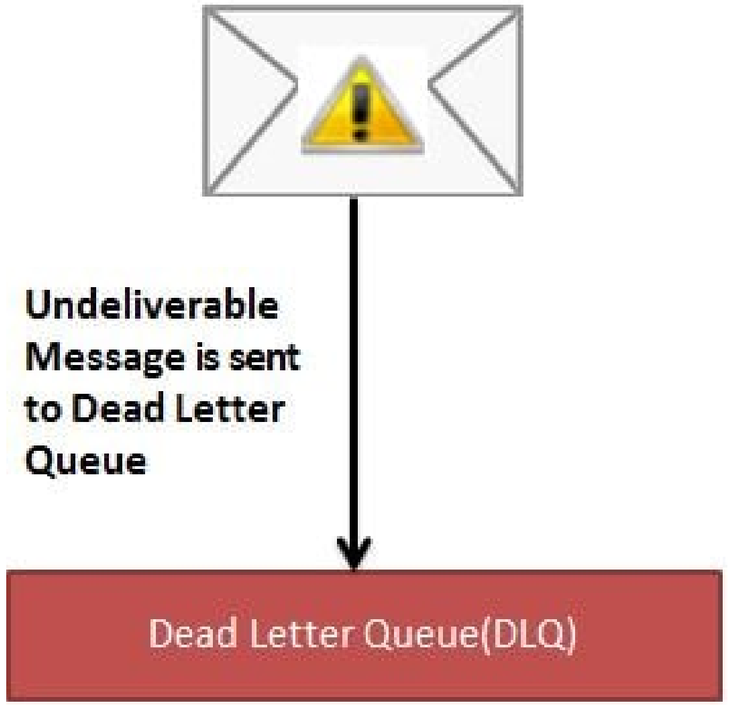
Retry and Error Handling

If exception exists after maximum retries then put message in a dead letter queue where it can be analyzed and corrected later.

What is a Dead Letter Queue?

In English vocabulary Dead letter mail is an undeliverable mail that cannot be delivered to the addressee.

A dead-letter queue (DLQ), sometimes which is also known as an undelivered-message queue, is a holding queue for messages that cannot be delivered to their destinations due to some reason or other.



rgupta.mtech@gmail.com

Retry and Error Handling

In message queueing the dead letter queue is a service implementation to store messages that meet

one or more of the following failure criteria:

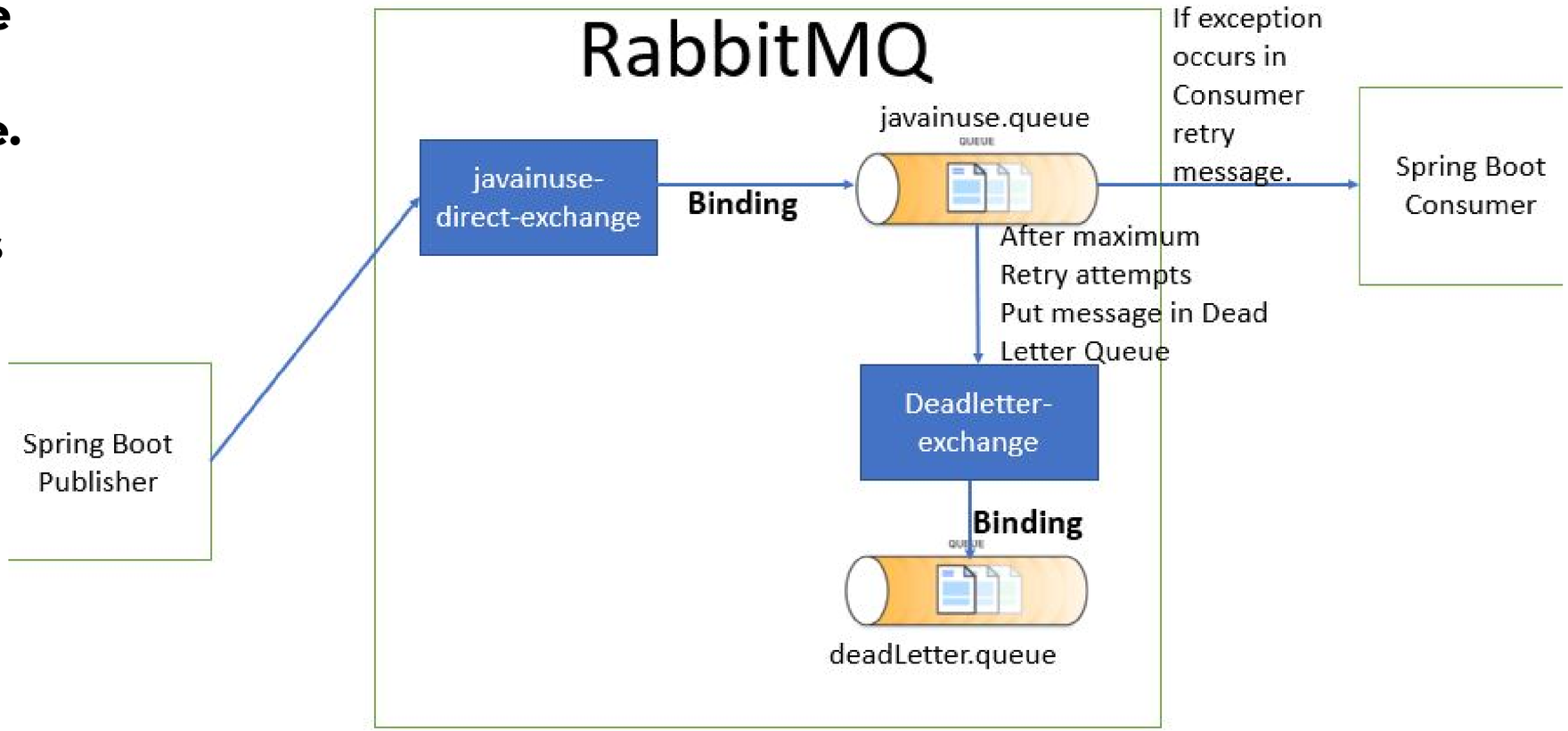
- **Message that is sent to a queue that does not exist.**
- **Queue length limit exceeded.**
- **Message length limit exceeded.**
- **Message is rejected by another queue exchange.**
- **Message reaches a threshold read counter number, because it is not consumed. Sometimes this is called a "back out queue".**

rgupta.mtech@gmail.com

Retry and Error Handling

Spring Boot Producer Module - It will produce a message and put it in RabbitMQ queue. It will also be responsible for creating the required queues including the dead letter queue.

Spring Boot Consumer Module - It will consume a message from RabbitMQ queue. We will be throwing an exception and then retrying the message. After maximum retries it will then be put in dead letter queue.



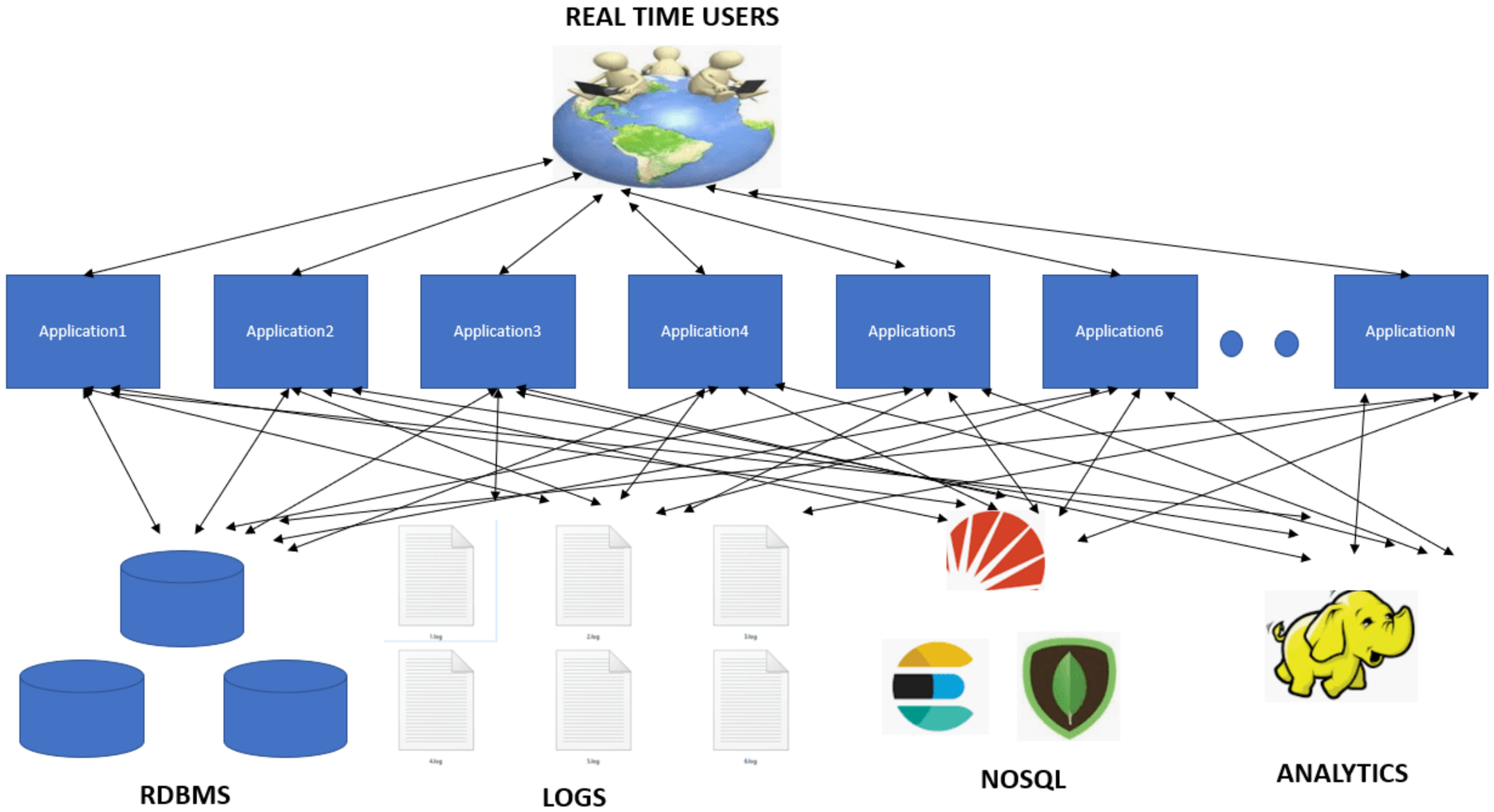
rgupta.mtech@gmail.com

Module 13: Messaging with Kafka

rgupta.mtech@gmail.com

What is the Need of Apache Kafka?

Complex web of applications involving point to point data movement. This involves moving large amount of data from one point to another.



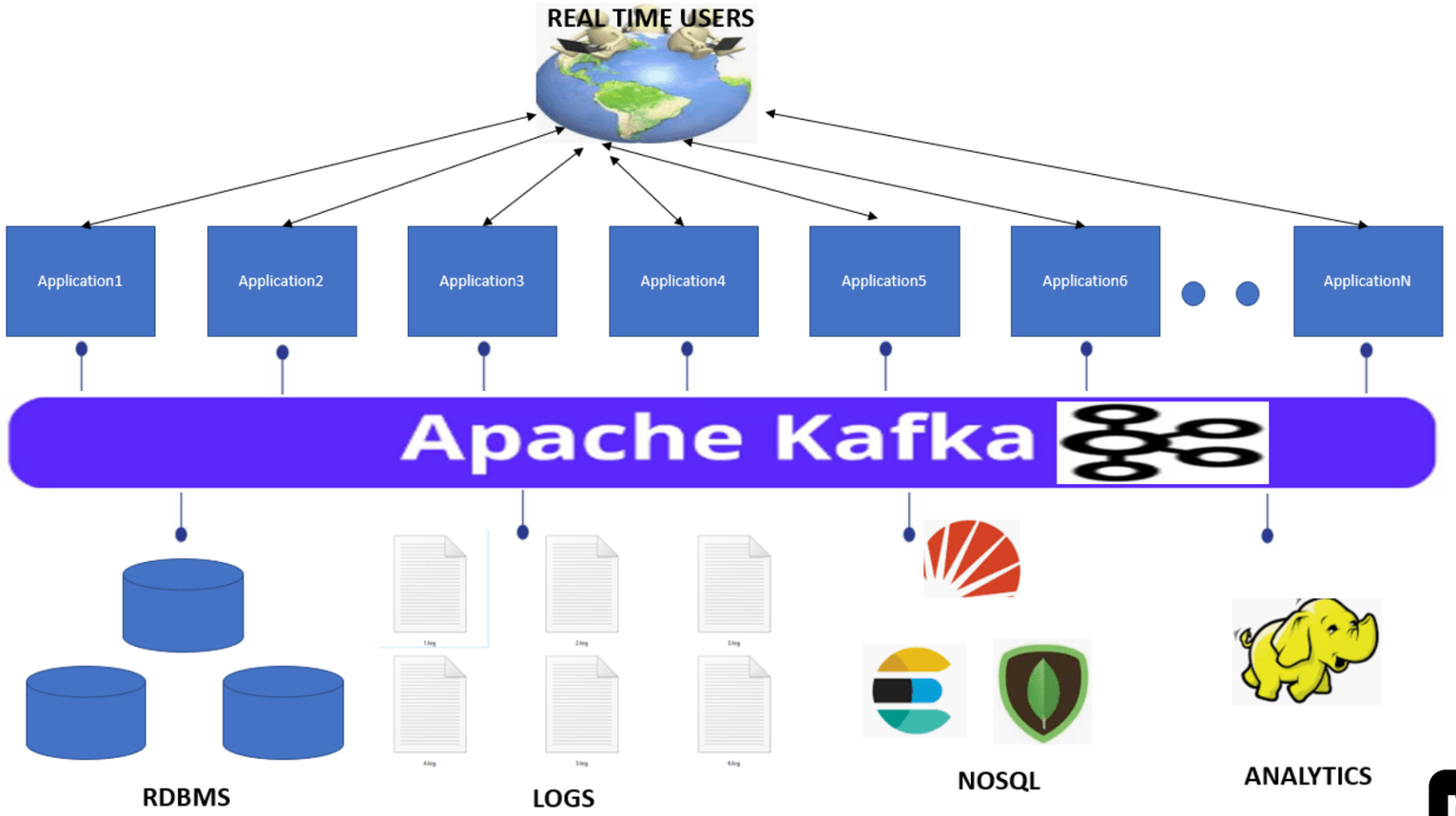
rgupta.mtech@gmail.com

What is Apache Kafka?

Apache Kafka is an open-source stream-processing software platform developed by LinkedIn and donated to the Apache Software Foundation.

It has been developed using Java and Scala.

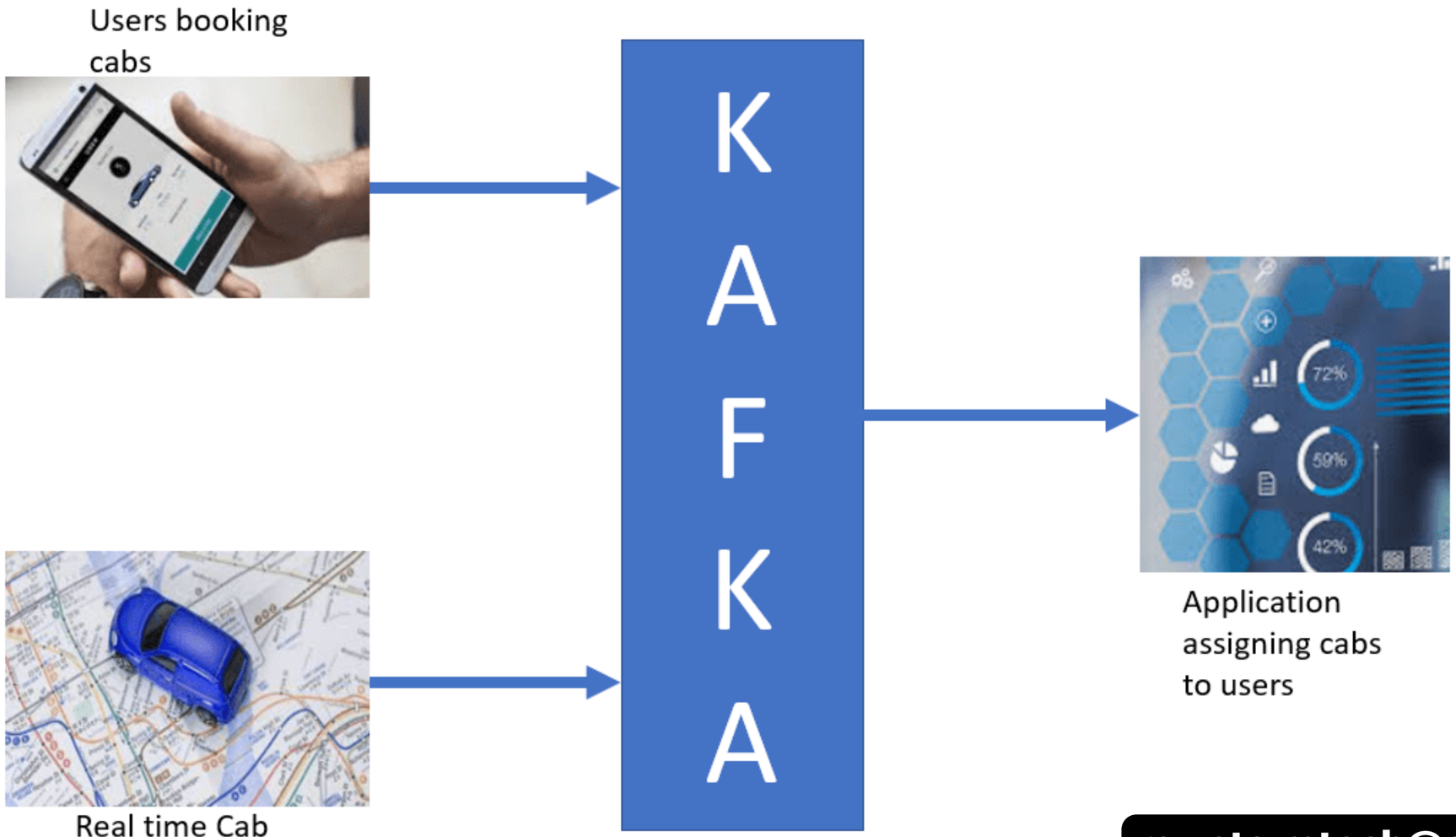
Apache Kafka is a high throughput distributed messaging system for handling real-time data feeds.



rgupta.mtech@gmail.com

Apache Kafka Usages

Real time example of Apache Kafka is Uber cab booking service. Uber makes use of Kafka to send User and Cab information to Uber Cab Booking System.

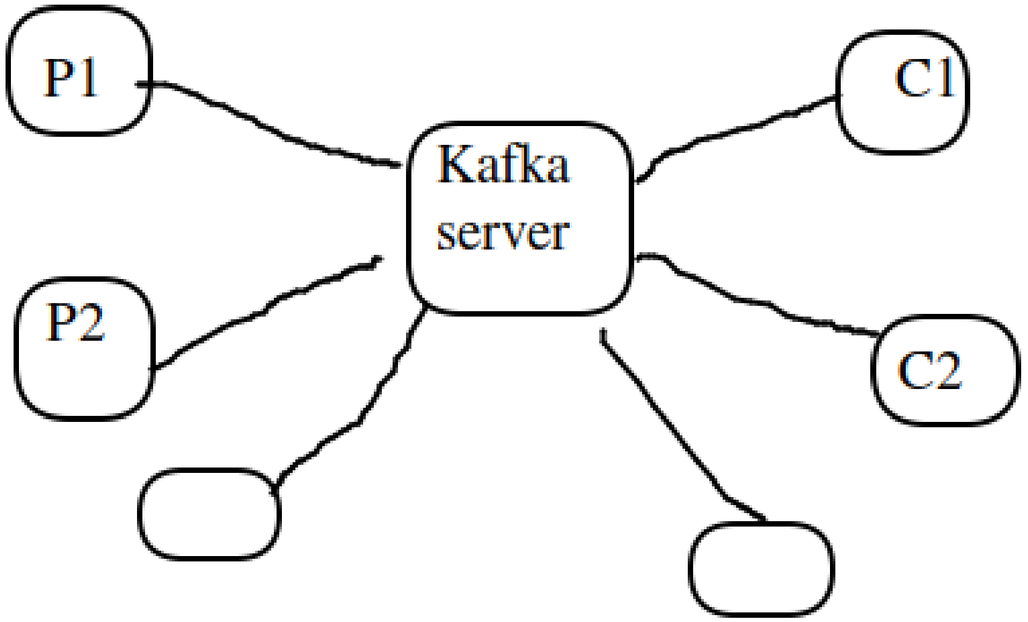


rgupta.mtech@gmail.com

What is Apache Kafka?

Event driven arch, millions of messages can be process per second

multiple producer

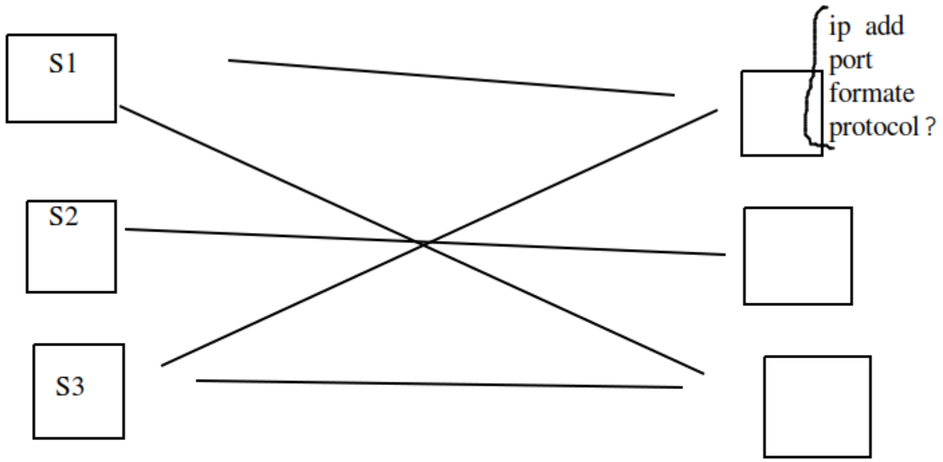


multiple broker and consumer

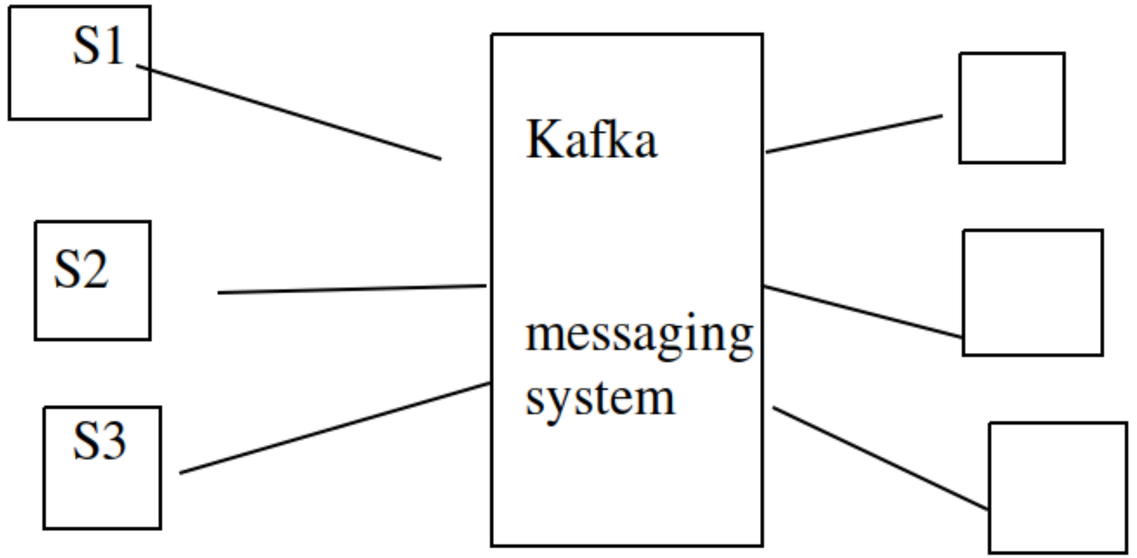
Apache Kafka is an open-source stream-processing software platform

In distributed env. kafka is reffered as kafka cluster made of more then one kafka server

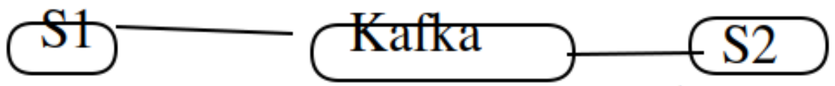
Without Kafka



Decoupling data processing pipeline



It solve Complex communication problem
it solve speed mismach problem

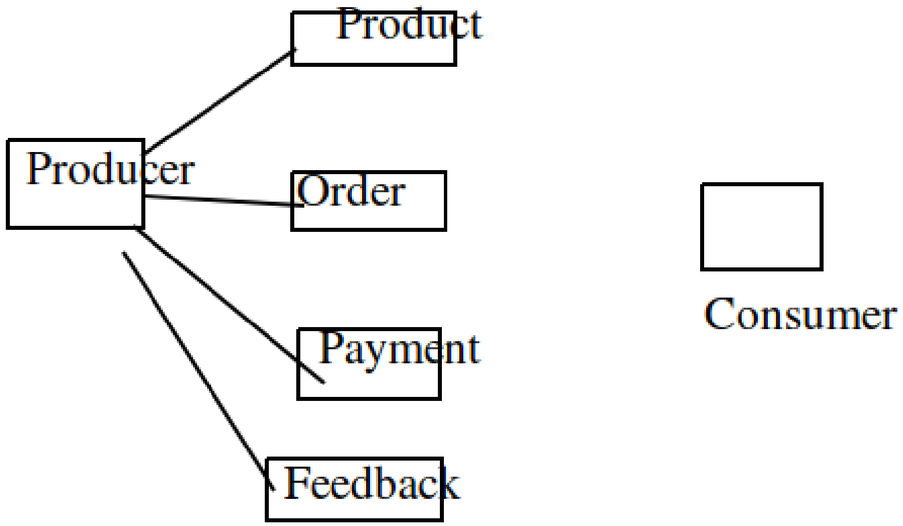


high speed

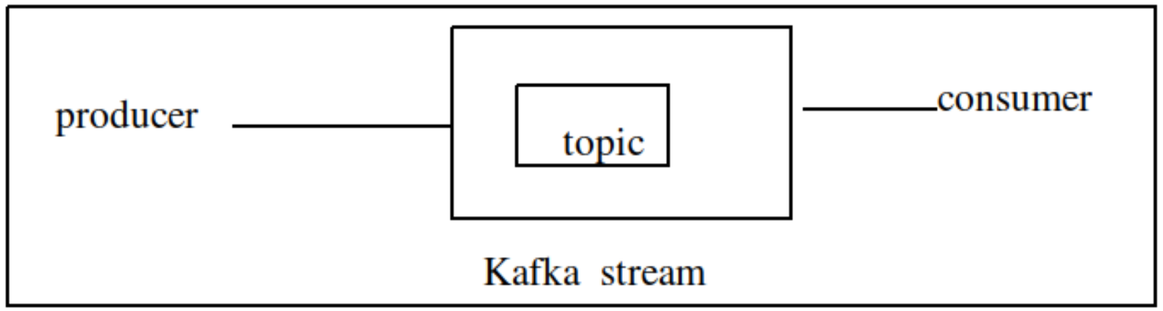
low speed

What is topic

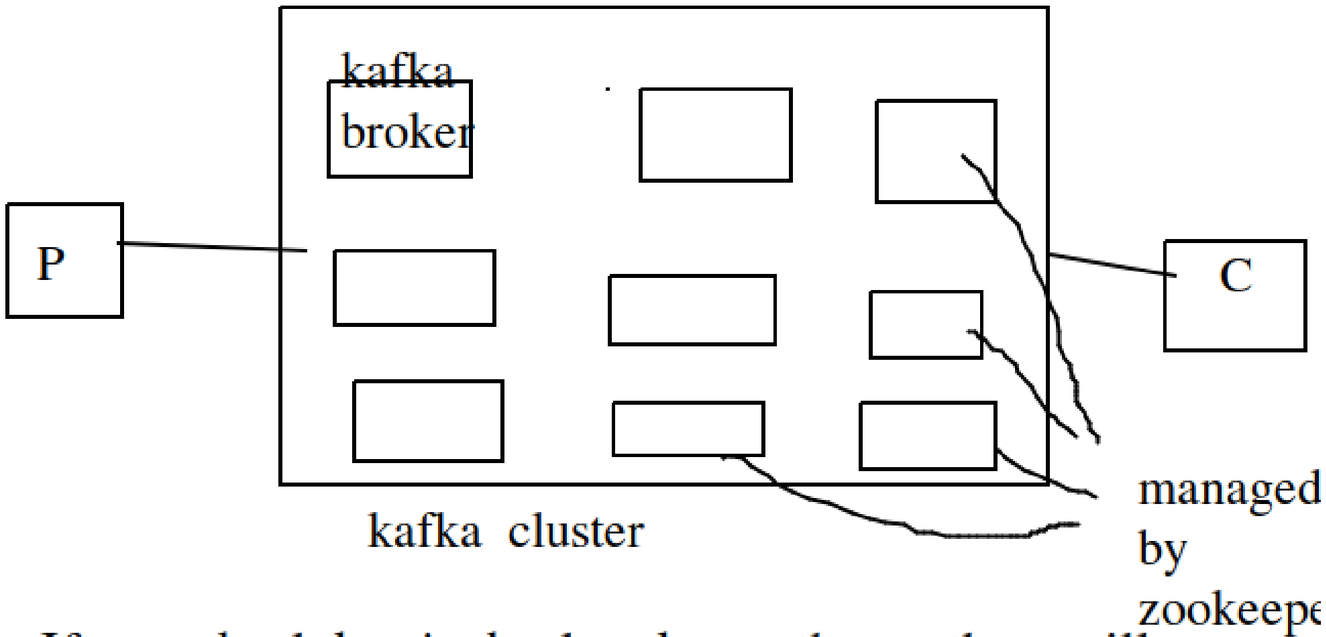
What if producer is sending 4 type of data?
Consumer get confused if he is only intrested on product data



Solution: segrate the data streams=> 4 topics for each category of data
similer to database table: related to one type of data



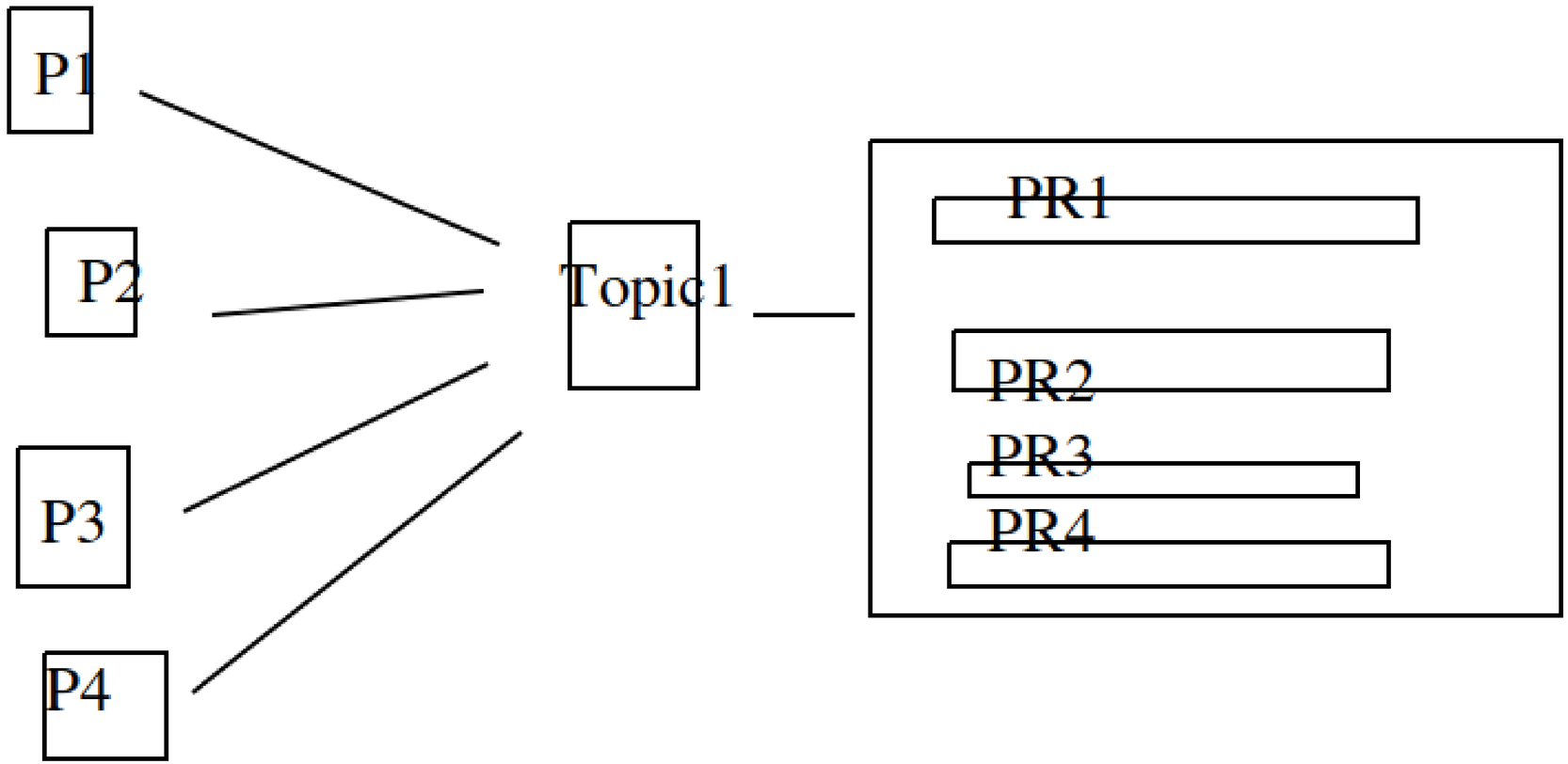
Scalability and fault tolerence(Zookeeper)



If any brokder is broke down then other will take care

How to manage it?
if one broker is broken down then who coordinate
Distributed service to manage lare amout of host
Focus on BL and not on distribution of logic
First we have to start the zookeeper and then kafka broker

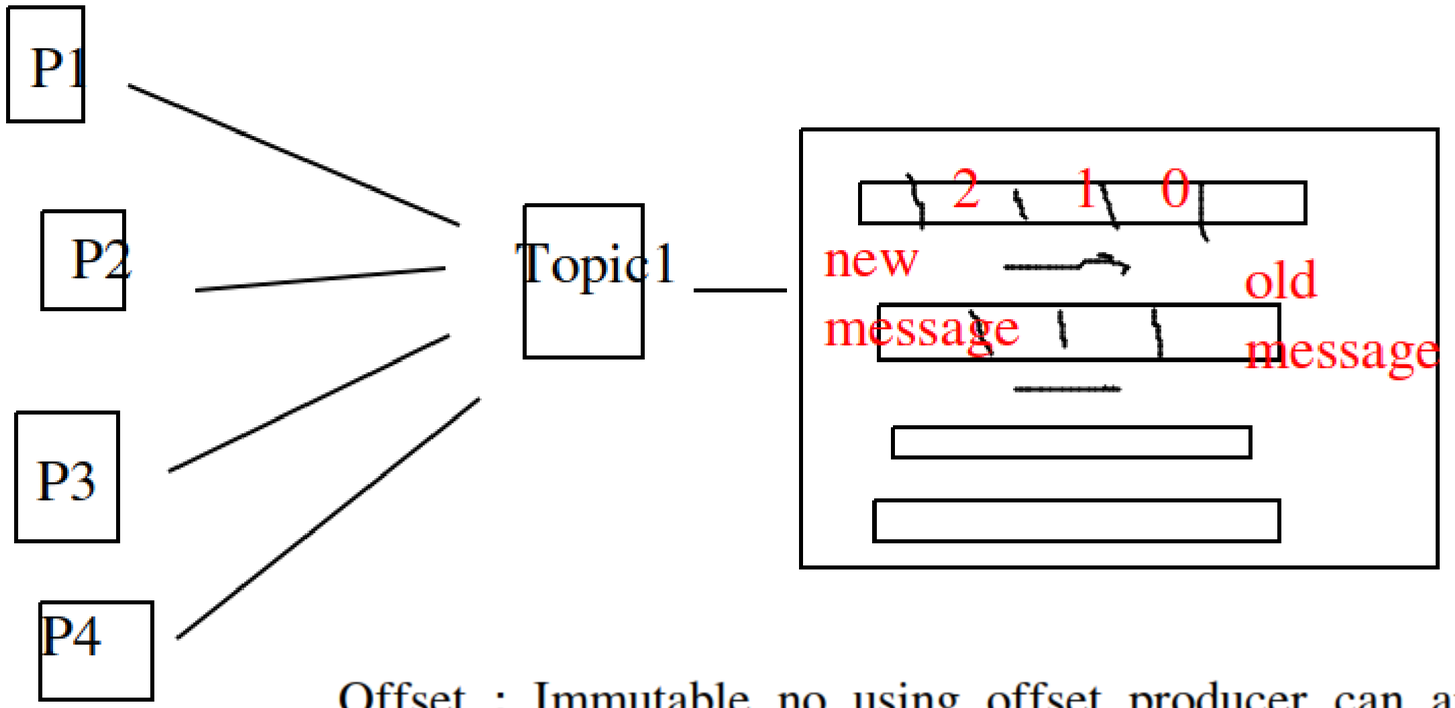
Problem of parallelism



We need to decide not of partitions while creating topics
we need to tell no of partitions

Partitions offset

Producer send the data into message offset system

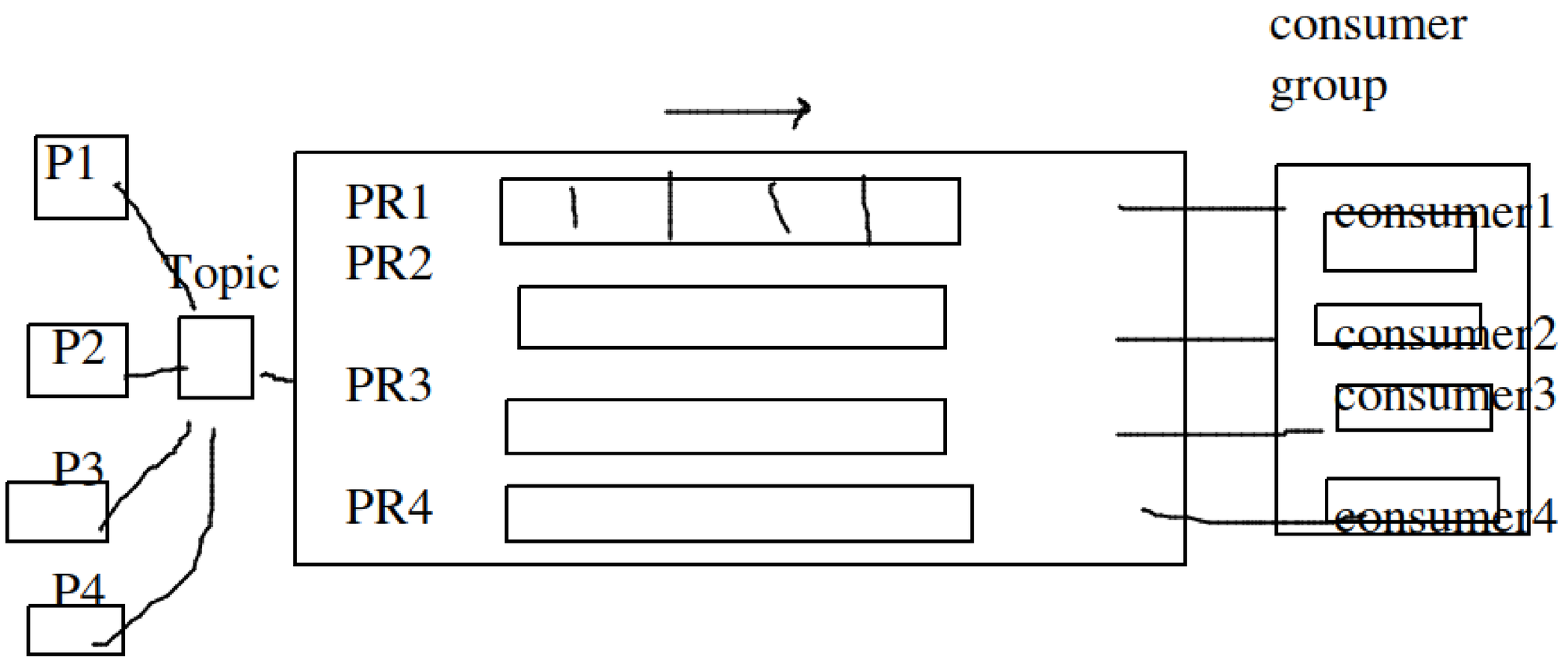


Offset : Immutable no using offset producer can arrange data into accending /decending order

Partition 1 data is not same as partion 2 data ⇒ To recognize message
which topic id?
which partition id?
which offset id?

Consumer Group

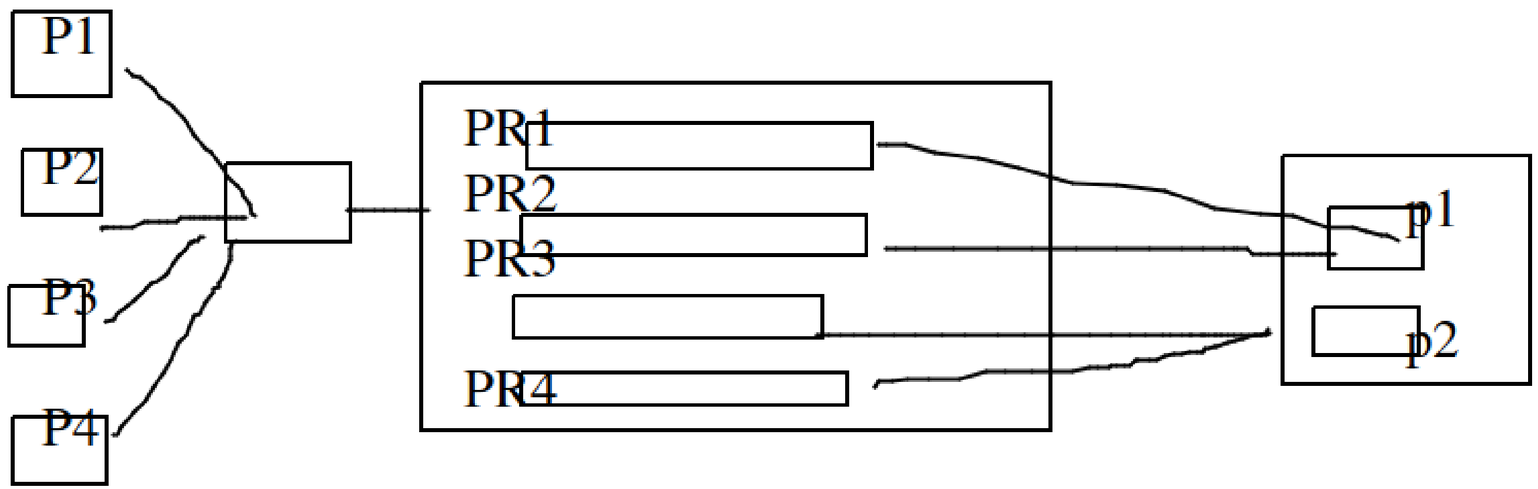
- We have only one consumer how data processing happens
- You create many consumer and connect one partion
- We can get all the data into one shot from 4 partitions, that is called consumer group
- Consumer group: single logical using they can share the work



rgupta.mtech@gmail.com

Consumer Group

- **Case I: if only 2 consumer is there in consumer group**

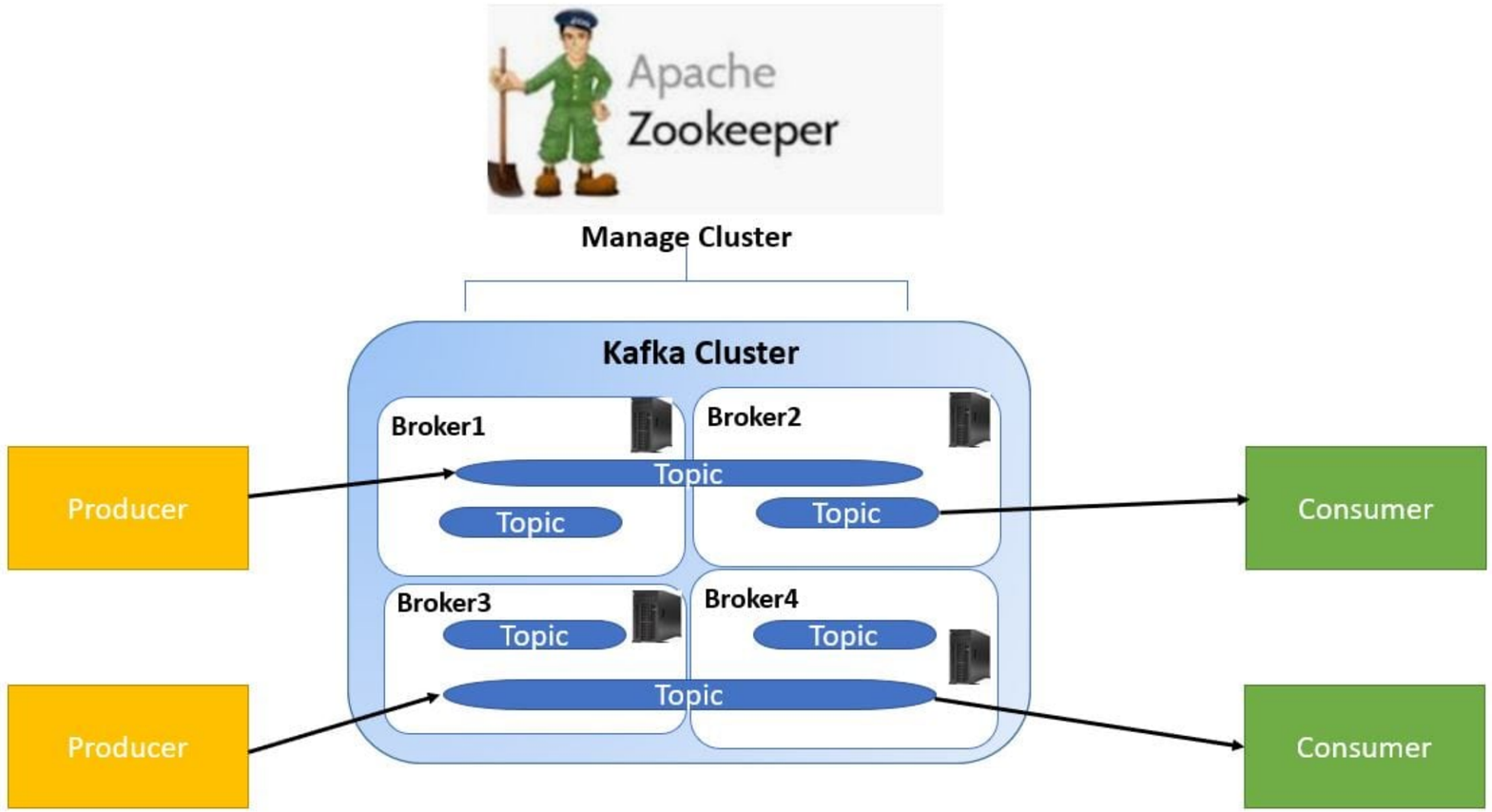


- **Case 2: if 4 consumer is there , each one take data from each partition**
- **Case 3: if we have 5 consumer group, 1 will be idel**
- **Case 4: if one consumer then all data send to that**

rgupta.mtech@gmail.com

Replication factor

- **Consider topic1 with 4 partition, then not all partition go into the broker**
- **Partition will be distribtued into multiple broker**
- **What if broker 2 is gone?**
- **How to solve the issue => replication copy**
- **If RF =3 then T1P1 should be replicated to 3 places in different brokers**
- **Although T1P1 is at 3 places one of them is called leader**
- **Kakfa zooker send data to the leader and then leader distributed/ replicate to others**



rgupta.mtech@gmail.com

Zookeeper

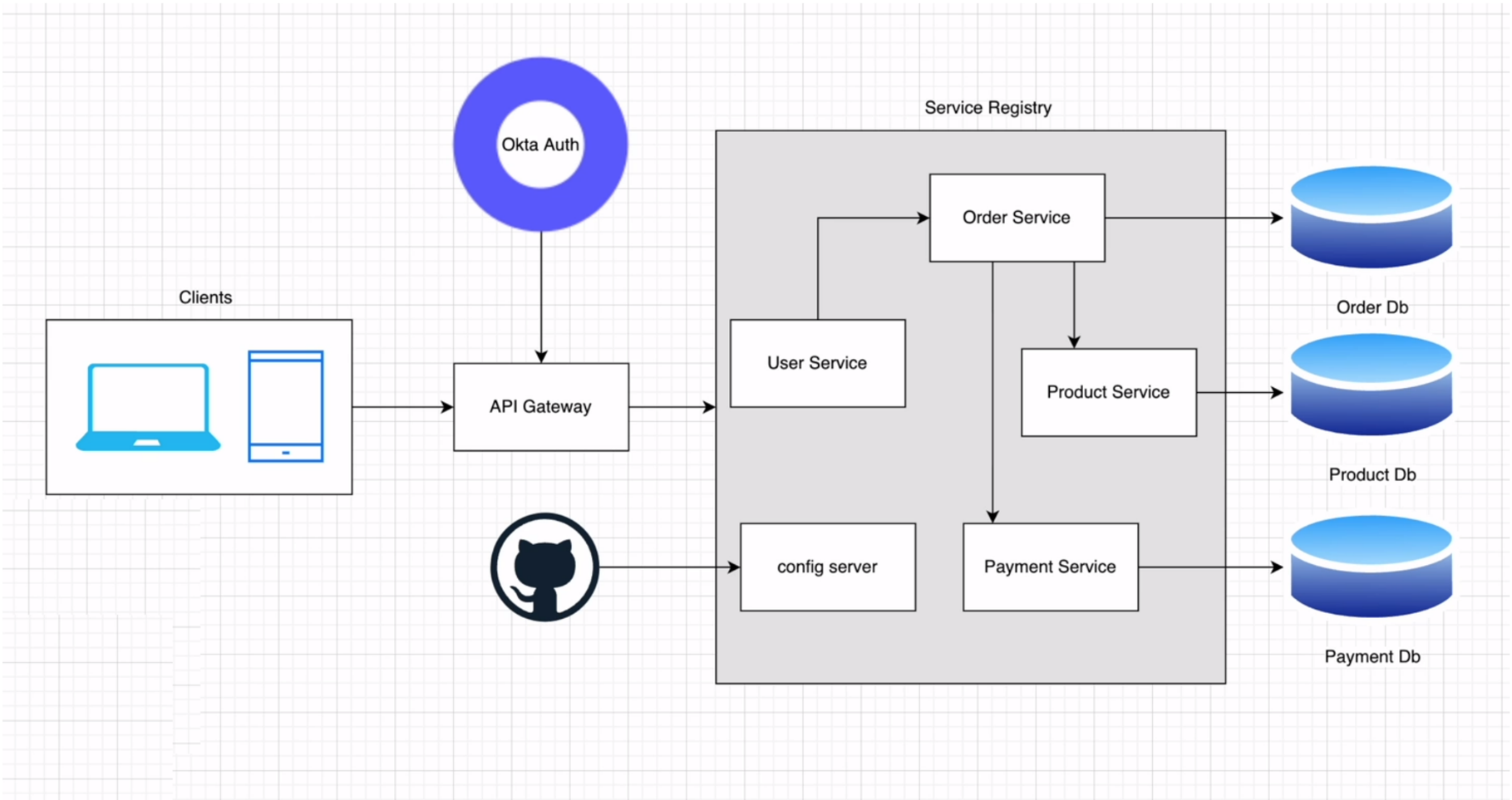
- **To manage the cluster we make use of Apache Zookeeper. Apache Zookeeper is a coordination service for distributed application that enables synchronization across a cluster.**
- **Zookeeper can be viewed as centralized repository where distributed applications can put data and get data out of it.**
- **It is used to keep the distributed system functioning together as a single unit, using its synchronization, serialization and coordination goals, selecting leader node.**

rgupta.mtech@gmail.com

Microservice Project work

rgupta.mtech@gmail.com

Microservice Example



rgupta.mtech@gmail.com