## Brief Introduction to Microservices

Microservices is not an api/libraries or it doesnt belongs to any specific programming language. Its an methodology or an architectural style of building an enterprise-large scale complex business system. Microservices methodology or architectural style has provided bunch of recommendations and guidelines along with set of design patterns help us in building large scale complex systems.

## As per the microservices architecture:

A complex business system is broken down into multiple smaller services which has the below characteristics

1. **loosely coupled**
2. **independently deployable**
3. **high scalable**
4. **collobarative services**
5. **resillient**

that can be developmend by multiple smaller mutually collaborating teams independently.

developer has to understand the architecture and various patterns related to it and facilitate himself in building the application based on microservices architecture.

It becomes quite difficult for the developers building a solution from the scratch based on microservices architecture.

To help us in quickly building applications and deploying based on microservices architecture we need support of programming languages.

but when it comes to development of microservices we just not only need support interms of technology by providing apis, we need infrastructure tools in facilitating the collaboration, discovery, deployment, monitoring the micro services as well.

1. **technology api**
2. **tools**

unless we have both together we cannot deliver a solution on microservices

So a single programming language cannot address/fulfill by providing all the necessary tools/api in building the microservices based application. So unfortunately we dont have a single language providing the whole solution yet, which makes implementation of microservices quite complex in the market.

There are several players in the market offer tools and apis in building microservices

1. **Netflix =** one of the early innovator and adopter of microservices architecture. they build several tools

in facilitating the deployment and delivery of microservices like

**eureka server**

**ribbon**

**hystix**

**zipkin**

**zuul**

**sleuth**

2. **apache =** has provided few tools in facilitating the deployment of microservices

zookeeper

alternate to them spring framework as apart of microservice module has comeup with their own set of tools and apis in developing and delivering microservices application

**Microservices is an methodology** or an architectural style of building software applications, it has provided guidelines, design patterns and recommendations that people has to adopt in building large scale complex business systems

We decompose the large-scale business systems into multiple smaller services which has the below characteristics

1. ***loosely coupled***
2. ***independently deployable***

3. ***highly scalable***

4. ***resilient***

5. ***Collaborative***

that can be developed by a smaller collaborative team of developers independently

Developer has to adopt the guidelines, recommendations and have to implement design patterns in building the software systems which seems to quite complex. So to help us in quickly building applications based on microservices architecture we need support of programming languages.

To build and manage the microservices we not only require api/libraries along with that we need support of tools for deploying, discovering, monitoring and managing the microservices. A programming language like java can provide api/library support which will not be sufficient to build a complete system.

There are third-party vendors provided tools support in enabling the deployment, discovery, monitoring and managing the microservices.

From this we can understand there are 2 parts of the world are there

1. **developing the software solution based on microservices**

2. **delivering the microservices to real-world using tools stack**

## #1 how can we develop the software solution based on microservices

Java has provided api for building microservices component based on Rest principles using Jaxrs api

## #2 how to delivery the microservices solution using tools stack

There are various providers offered tools

1. **The early adopter of the microservices architecture is netflix.** He has invented several tools that helps us in discovering and integrating the microservices across the applications as below.

    **1.1 Eureka Server = discovery server**

    **1.2 Feign client = declarative client api for consuming restful services**

    **1.3 Hystrix = circuit breaker for ressilience**

    **1.4 Ribbon = client-side load balancer**

    **1.5 Zuul = api gateway for routing the microservices**

netflix has provided all the above tools as opensource and freely distributable

2. **apache has also provided several tools interms of discovery and integrating microservices**

**2.1 Consul = discovery server**

**2.2 Zookeeper = distributed configuration management**

**2.3 Camel Rest DSL = Gateway**

**2.4 Zipkin = Tracing server for troubleshooting the latency problems in microservices**

The above tools are available from the perspective of java platform in discovering and integrating the microservices

In addition to this there are otherways of discovering and integrating the microservices from an non-java platform as well using devops tools

1. **docker =** containerizing the microservices application
2. **kubernetes** = container cluster manager

    pod = containerizing the application

    services (load balancer) = exposing & discovering

    deployments = roling updates

by adopting devops tools like docker and kubernetes service discovery and load balancing are taken care by kubernetes itself. For integrating the services at client side we still need

1. **Feign client**

2. **Hystrix**
3. **Zuul**

## How to delivery a complete microservices solution?

It requires a mixture of multiple techologies and tools stack. A developer has to be very good at api/library and even should have knowledge on the third-party vendor provided tools in delivery a complete microservices based application. which seems to quite complex in building the application.

Even though the third-party vendors provided enough tools in discovering and integrating microservices consuming these tools aspart of the application is quite complex and has to write lot of boiler-plate logic

To help us quickly delivering the microservices based application in Spring platform, spring has introduced 3 modules

1. **Spring Cloud**
2. **Spring Microservices**
3. **Spring Boot**

Initially Spring has adopted the strategy of integrating these third-party popular tools into Spring Framework as part of MicroServices module rather than building these tools from scratch.

develper can quickly build microservices solutions using spring microservices module through third-party tool integrations offered by spring

For eg..

developer can quickly bring up eureka server by writing little boiler-plate logic through spring microservices autoconfigurations

similarly all the rest of the tools follows the same patterns of integrations

now consuming third-party tools and delivery the application becomes quite easy because spring microservices (integration) module

over the time spring framework has started building all of these tools natively as part spring platform and has offered directory from Spring Microservices module

**What is the support working with microservices as apart of the java platform?**

Java language has not provided any tools in discovering and integrating microservices, it has only provided jaxrs api to help us in building service components.

In order to intergrate, discover and consume these microservices there are third-party vendors provided various different tools

**#1 Netflix**

**1.1 Eureka Server**

**1.2 Ribbon**

**1.3 Zuul**

**1.4 Hystrix**

**1.5 Feign Client**

**#2 Apache**

**2.1 Camel DSL**

**2.2 Consul**

**2.3 Zipkins**

**2.4 Zookeeper**

these vendors has provided the above tools on java platform and made them open source and freely distributable as well.

Now developer can build microservices and can leverage the above java tool in deploying, discovering and integrating microservices. even though tools are offered by the third-party organizations adopting and implementing them requires huge amount of code and configuration to be written, which seems quite complex in using them.

To help us easily adopting those third-party tools Spring Framework has provided Spring Cloud + Spring Microservices module (built on top of Spring Boot)

Initially Strategy of Spring Framework developers is to quickly bring support of developing + delivering microservices based application on Spring Platform. In order to achieve their goal the spring team has provided integrations in working with Netflix and Apache tools discussed above.

Anyone can quickly configure and bring up the Eureka Server or use Ribbon or Zuul tools using Spring Microservices module with minimal code where most of the configurations required are taken care by autoconfigurations.

looks like the third-party vendors has not provided complete solution addressing all the kinds of requirements that typically exists for an microservices based application. So Spring developers by themself started building the discovery, integration and various other tools native in Spring Framework that helps in building microservices based applications on the Spring platform itself.

**What is the support of Spring Framework interms of microservice development?**

As part of Spring Cloud + Spring Microservices (latest) has provided below set of tools

1. **Eureka Server integration**
2. **Spring Cloud Config Server/Config Client (Replacing Zookeeper)**

3. **Spring Circuit breaker (Replacing Hystrix)**
4. **Spring Api Gateway (Replacing Zuul)**
5. **Spring Load balancer (Replacing Ribbon)**
6. **Feign Client Api**

---

## Why & How to develop applications based on Microservices architecture?

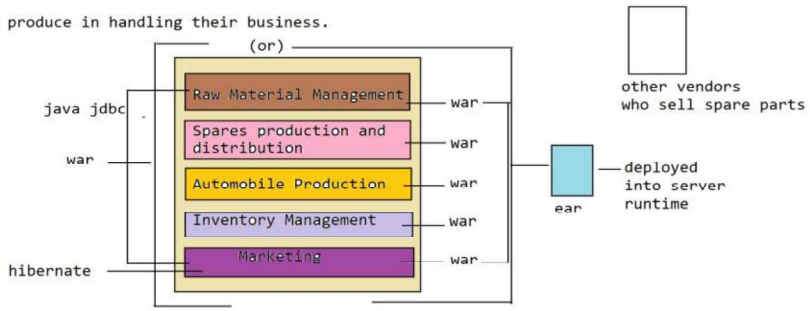
To better understand the world of Microservices, let us explore how to develop applications without the microservices.

In general when we are developing enterprise large applications we build the applications based on Monolithic application Architecture.

goal/vision:
build an software system that help in automating the business process of **production & delivery** of automobiles by an automobile business system.

What are the different modules we need to produce in handling their business.

1. procurement of raw material
2. inventory management
3. spares
   - production
   - distribution
   - consumption
4. production of automobiles
   - assembly
   - qa check
   - labelling and approvals
5. marketing
   - distribution chain




Your application is built out of
#1 either one single project which comprises the source code pertaining to all the modules
(or)
#2 each module as a war project having a parent project as ear sharing common libraries across the wars, which has to be assembled together to deploy as a complete solution.

```
nextgenautomobile (single project)
|-src
  |-main
    |-java
      [modules are build in package hierarchies]
    |-resources
  |-webapp
    |-WEB-INF
|-pom.xml
```

```
nextgenautomobileEAR
|-lib
  |-[common modules]
  |-third-party libraries
|-rawmaterial (war project)
|-sparepartdistribution (war project)
|-inventorymanagement (war project)
|-automobileproduction (war project)
|-marketing (war project)
|-nextgenautomobileservice (common functionality)
```

- modularity (broke down the source code so that we can easily develop and manage)
- reusability of the libraries

Monolithic Application Architecture:
Application is built and delivered as one single deployable artifact.

In monolithic application architecture all the components of the system across the modules co-exist together referring each of them through classpath and be part of single jvm only.

we cannot independently deploy a part or piece of the system leaving the other components which will breaks the dependencies or refferences to the other classes.

---

Spring Framework as part of

**Spring Cloud**

**Spring MicroServices**

has provided richer set of tools in discovering, integrating, managing and monitoring the microservices based application

1. Eureka Server Integration with Spring Microservices
2. Spring Cloud Config Server and Config Client (Replacing Zookeeper)

3. Spring Load balancer (Replacing Ribbon)
4. Spring Circuit Breaker (Replacing Hystrix)
5. Spring Api Gateway (Replacing Zuul)
6. Feign Client Integration with Spring Microservices

To better understand microservices architecture, let us explore monolithic application architecture and how does applications are being built and delivered in monolithic application architecture.

## What is Monolithic Application architecutre?

An application is built out of single source code project where the components across the modules of the application are referenced directly (through the classpath), and the entire application is being deployed and running under one jvm.

The application is delivered as one single deployable artifact which is deployed on an Server Runtime is called "Monolithic" application development.

**- single sourcecode project**

**- directly referenced/integrated the module components via classpath references**

**- deployed on one single jvm as a single deployable artifact**

---
---

## Monilithic Application Architecture

---

**Problem:**

We want to develop an enterprise large application, which contains several modules belongs to various functional areas that are logically related to each other and should work together

Our application should support different types of clients like desktop browsers, mobile browsers and native mobile apps to use the functionality of our system. In addition we want third-parties to consume and integrate our business system into their business.

The application should be integratable with other applications through web services or via messaging brokers in addition should be able to support exchanging data on multiple data representation stands like XML, JSON, HTML etc

**Problem Context:-**

1. **modules catered to different functionalities should be connected logically and work together**
2. **various different types of clients (mobile/desktop)**
3. **integrable through third-parties either through web services / messaging brokers**

4. should serve multiple data-representation formats support various different platforms

We can build such an enterprise application with wide-variety of requirements addressing different aspects of users either using monolithic application architecture or microservices architecture

There few circumstances or context under which we can build the application based on Monolithic Application Architecture as described below.

1. **We want a team of developers to work on development of the application**

2. **As the whole application has been built out of single source code, the new team members can understand the holistic view of the application and the module dependencies across the modules, so they can be quickly productive and contribute across any of the modules of the system**

3. **we want to run multiple instances of the application across multiple machines to meet the scalability aspects and high availability aspects of the system.**

If we want to achieve scalability and high availability requirements easily we can use monolithic application architecture, because across all the env we need to deploy the same artifact which requires identical env, that can be created from cloned env

4. the developers should be able to understand the overall architecture of the application easily and how do they work together

**Solution:**

use monolithic application architecture in developing the software system

**What are the advantages of building application based on monolithic application architecture?**

**1. Easy to develop**

The IDEs and tools available are designed in support of monolithic application development, so developer can quickly build the application

**2. Easy to deploy**

The entire application is packaged as a single deployable war/ear, we just need to copy to the target runtime to run it

**3. Easy to Scale**

We can create quickly the cloned environments to run the application on multiple machines

Along with advantages we have quite a number of dis-advantages as well with Monolithic application development.

Over the time the application becomes large due to increasing modules or increasing business requirement. To meet the demands of the application we need to increase the number of resources working in a team, where it brings up more complexity and adds challenges in handling larger team, due to which we run into several **drawbacks** as below.

### 1. Overloaded IDE

The larger the code base the slower IDE, because for everychange we make in the source code, IDE has to rebuild the entire project (across all modules), which takes too much amount of time and kills the developers productivity

### 2. Overloaded Server Runtime

The larger the application, it takes more time for deploying and startup of the Server. During development, developer in order to debug the code, he has to make changes and redeploy the application where he has to wait lot of time waiting for servers to comeup, which kills developers productivity. In addition deployment of the application on to the production also takes lot of time.

### 3. Modularity and Re-usability is difficult to achieve

The entire application is being built on one single source, identifying the modules and reusing the components across the modules becomes quite complex for the developer, most of the time the developers will endup in creating new components rather than identifying and reusing the existing components of the system. Due to which code duplication and maintainability issues will creep up

**4.** Large monolithic applications makes the developer feel afraid of working on the application, he often find it complex in understand the modules of the system to work on, due to lack of complete knowledge, the quality of code degrades to a great extent.

## 5. Scaling the application becomes very difficult

**5.1** In a monolithic application we can achieve scalability only in one dimension, which is called horizontal scaling, we cannot scale an module independent of the entire system.

**5.2** The cost of scaling the application is very high when it comes to monolithic application. Even though the traffic source or load is high on a single module of the system, we need to scale up the entire application to make the module scaled, which requires more computing resources and more cost in scaling one module (because the entire system has to be scaled)

5.3 different modules has different resource requirement

few modules are cpu bounded requires more processing capacity to perform operation

few modules are memory bounded consumes more memory in performing operation.

In monolithic deployment even though we can scaleup the resources of the system, we cannot cater the resources to a specific module based on its needs.

## 6. Continous Integration and Continous deployment is tough to achieve

The frequent deployment of the application becomes very difficult, since the application is very huge it

takes lot of time in shutting down the application, and there could lot of schedule jobs which might running as part of the application. When we trigger a deployment, the existing jobs might fail in stopping due to which the redeployment of the application might results in failure of starting up few of the components, which leaves the system in in-consistent state.

The developer has to jump in and should cleanup the environment in redeploying the application which takes more amount of time than usual

### 7. Scaling up the development team is very difficult

When the application grows bigger we cannot manage the development with small team, we need to increase the no of resources working within the team. If the team grows bigger managing them will become quite difficult job.

Even then also we cannot break the team into multiple smaller teams either

1. ***based on technology***
2. ***functional area***

because there are several challenges in developing the problem by multiple teams as below.

**7.1** If teams are working independently on their respective functional modules end of the day the changes made by one team may not be in compatibile with other module which breaks the other module, it requires huge coordination between teams to ensure their deliverables are compatible and intact without breaking each

**7.2** Even one of the module has been finished for release, it cannot be updated into production independent of other modules as all of the code changes across the modules are made into one single source, deploying the module makes the incomplete development of other modules to be released.

There is no way apart from teams sitting idle waiting for others to complete

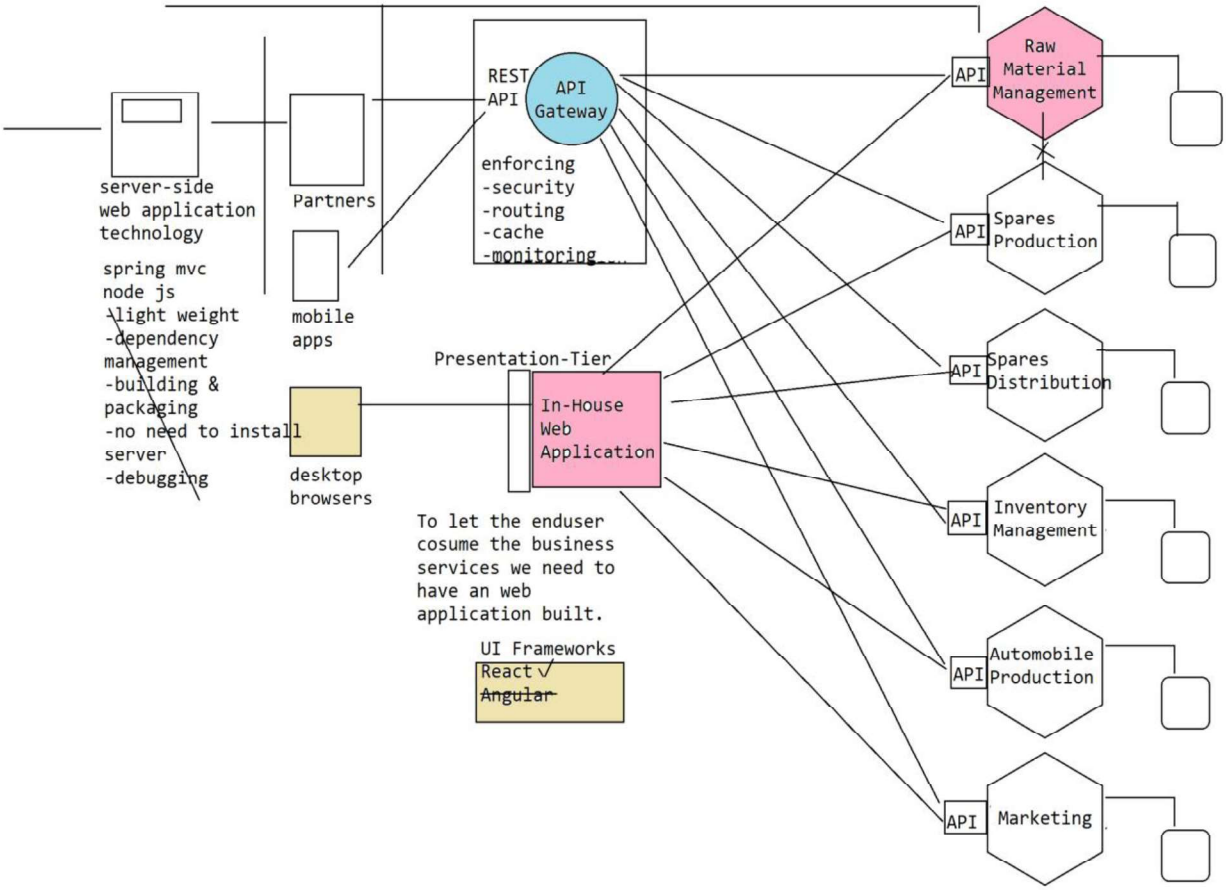
exceess planning and coordination is required to ensure all the teams can work together and relase it together which is highly difficult

## 8. Long-term commitment to the technology Stack

Monolithic applications forces us to stick to an techonology stack. we need to choose the technology stack in which we want to build the application initially itself and should be commited to it. In future if there are new frameworks or latest technologies emerges, we cannot migrate the application incrementally, as the modules are directly dependent on each other chaning a technology of a module effects other modules of the system.

The only possibility in adopting the new technologies is rebuilding the application from the scratch.

---
---




**what are the dis-advantages or drawbacks in developing an application based on monolithic application architecture?**

1. **overloaded ide**
2. **overloaded server runtime**
3. **difficult to achieve re-usability through modularity**

4. due to the complexity in understand the sourcecode, the developer will result in poor implementation of the code which will degrades the quality of source code

5. monolithic applications are not easy to be scaled

5.1 only supports one dimensional scaling (horizontal scaling)

5.2 the cost of scaling is very high, because we need to scale always the entire application rather than a single module which requires huge computing capacity

5.3 different modules has different resource requirement, few modules are cpu bounded and few other modules are memory bounded. we cannot cater the resources based on nature of the module in meeting the resource requirements.

6. we cannot implement continous integration and continous deployment/delivery practices easily, because the monolithic applications has several cron/scheduler jobs which may not be terminated properly due to which a redeployment might leave the system in an in-consistent state and requires a cleanup from scratch

7. we cannot scaleup the development team

if the application grows bigger we need more resources to develop the application. we cannot manage teams of larger size so we need to broke down the teams into smaller groups based on technology or functional area. There are several problems we run into by breaking down the development into multiple teams in Monolithic architecture

7.1 If the teams are working independent of each other, the changes made by one team might be in-compatible with other modules of the system, it

requires through coordination and analysis in performing the development by individual teams

7.2 A team cannot plan its features and cannot go for release independently, as the project is built out of single source the team has to wait for others to complete their features for release

from the above we cannot scale up the development teams and cannot do parallel development and releases

8. long-term commitment to the technology stack = we cannot adopt or uptake the latest technologies that are emerging in the market, because if we implement the latest technology stack in one module the other components will be broken, the only way to uptake new technologies adopt it across the whole system from scratch which is quite complex and required re-investment

---

**How does Microservices application architecture looks like?**

The whole application is broken down into smaller services which are

- **loosely coupled**
- **independently deployable**
- **highly scalable**
- **resilent**
- **mutually collaborative services**

that can be developed by independent collaborating teams

***Note:*** (Architecture: read this along with architecture)

**1. The application is not broken down into microservices based on number of modules there are various design considerations to be taken into account in identifying and breaking down the system into Microservices**

## 2. How can we have a microservices independently deployable of others?

## 2.1 How can we make the microservices independently deployable?

**No two microservices of the application should talk to each other by holding the concreate references via the classpath, because we cannot make them independently deployable. The each microservice can be deployed independent of other service because we dont need other services as dependencies since we are not referring their components.**

## 2.2 How can we reuse functionality across the services if they are independent?

These services are exposed based on REST Endpoints, each of them can reuse the others through api calls.

## 2.3 Why we want the services to be independently deployable?

There are plently of advantages we get by having each of them to be loosely coupled and independently deployable.

**2.3.1** We can organize and conduct development of the application by breaking down the development team into smaller groups each independent of others

**2.3.2** Each microservices can be built out of their own source code, without having the dependency of others

2.3.3 Can be independent in planning and release the modules without bothering about the other modules.

2.3.4 Each module can be deployed independently without affecting the availability of the others

2.3.5 Each module can be scaled up independent of others

2.3.6 Failures can be isolated due to independent deployable

## 2.4 Why do we want the microservices completely loosely coupled?

If there exists coupling between the services a change on a service will effect other micro services of the system. To manage these changes we need

1. **huge coordination**
2. **change and impact analysis for every requirement**

which will degrades the productivity and delivery of the system

if we can make the microservices loosely coupled from each other, then a change in one services is isolated from other services

due to which development teams are having higher independence

no coordination

no need to think about impact analysis etc

**How can we make microservices completely loosely coupled?**

If no 2 microservices are holding the concrete reference others via classpath in communicating or reusing the functionality we can achieve loosely coupling (integrating through REST endpoints), but this is not sufficient.

Across the services if there are sharing the same schema a change in table or schema for one of the microservice will effect other micorservices so that loosely coupling will be broken down.

To achieve highest level of loose-coupling every Microservices should have their own schema/database. Which is called Database per Service pattern.

## Problem Context:

Want to build an enterprise large application which contains several modules of various different functionality

There are different types of clients like Desktop Browser, Mobile Brower and Native Mobile Application users want to consume the business services offered by our business

There are third-parties want to integrate their business system through Web Services and Messaging Gateways

Our application should serve the data in various data representation standards like XML, JSON/ YAML

## What circumstances or forces make you build the above application requirements on Microservices architecture:

1. We want independent development teams working in parallel across various functional modules of the application

2. New Member should be quickly productive and should contribute to the module of the system

3. The application should be easy to understand and quick to modify

4. We want to adopt continuous integration and delivery practices

5. We want to scale by running multiple instances on different machines independently

## What are the advantages in building and delivering an application based on Microservices Architecture

1. Each Microservices is relatively small and has its own source code independent of other services

  1.1 A developer can easily understand the source code of the application, there by he contribute the best in building the application which increases the quality of the system

  1.2 IDE will faster in building the application

  1.3 Server Runtimes will quickly deploy the Microservices, so that development team can quickly debug their application

  1.4 Every developer can easily demorcate the boundaries of the modules he is working on, so he can encourage modularity through which we can achieve reusablity

2. Services can be deployed independently

3. Parallel development is promoted

4. testability of the application becomes very easy

5. Independently Scalable, here we can achieve vertical scaling

  5.1 Rather than scaling the system as a whole, we can identify the Microservices for whom the load is high and can scale up only the part of the system it belongs to rather than the whole

    due to which low cost of scaling

we cater right set of computing resources based on the nature of the service

6. Fault Isolation

7. Eliminates long-term comitmment to the technology stack

Along with advantages, there are several dis-advantages in choosing and developing an application on Microservices architecture

**1. Developer has to deal with huge complexity in designing and creating an distributed system**

**1.1 Developer must implement inter-service communication mechanism and deal with communication failures**

**1.2 There are times where we need write archestration logic in communicating with multiple microservices in collborating their responses in building the functionality which is quite complex job to deal with**

**1.3 Testing the interactions between the microservices is very difficult, because we need to simulate the failures and see the behaviour of the system**

**1.4 IDEs/Tools are in favour of Monolithic appliction development no support for developer in building microservices appliction**

**1.5 Deployment complexities in production and operation challenges in running is very high**

**1.6 The more the number of services deployed the increased consumption of jvm memory**

---
---

**What is the support jee platform in developing and delivering an Monolithic application architecture solution?**

Can we leverage the same jee platform tools in delivering Microservices or not? If not Why?

Then what are the alternates?

**What is Microservices application architecture?**

The enterprise application is decomposed into multiple smaller services that are

1. **loosely coupled**
2. **independently deployable**
3. **highly scalable**
4. **resilient**
5. **Collaborative Services**

that are being developed by colloborative independent teams

**advantages:-**

1. Each service is built out of its own source code project

1.1 Independent teams can work in parallel in building and delivering the services which promotes parallel application development

1.2 New developers can quickly understand the application and can become productive

1.3 The developers can understand architecture of the application, so that they produce quality of code and encourages modularity through which we can achieve reusablity and thereby application is easily maintainable

1.4 IDE are not overloaded

1.5 Server Runtimes can quick start due to smaller application size, so developer time will not be wasted during debugging the application

2. Fault Isolation

3. Independently Scalable

4. Can implement CI/CD practices easily

5. Testability of the services are easy as those are small in size

6. We can adopt quickly the new emerging technologies for a part of system rather than whole

**dis-advantages:-**

There are several drawbacks and complexities involved in building applications based on microservices architecture

**1. developer has to deal with complexity interms of building distributed services**

2. existing IDEs doesnt have support for building distributed solutions based on microservices architecture so developer find it very difficult being productive during development

3. more complexity interms of inter-service communication

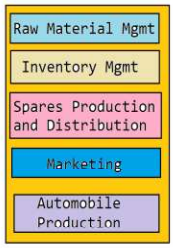
4. has to deal with intermediate failures between the services and has to write the exception management logic or retry logic in dealing with such service failures

5. the more the number of service deployments the more consumption of jvm memory and the cost of deployment increases

6. testing the inter-service communication is highly complex

---

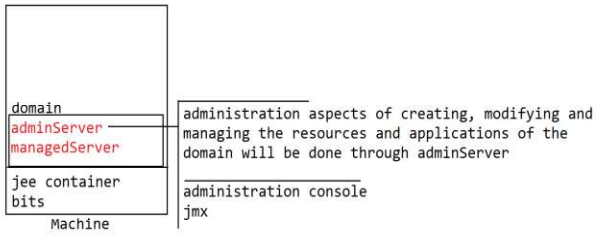
**How to develop and deploy an Monolithic architecture based application? What is the support of jee interms of development and delivering the monolithic applications?**



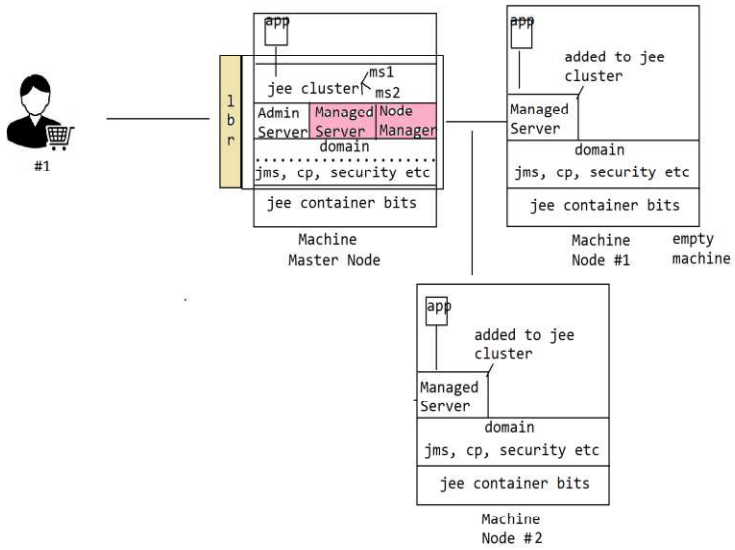

Enterprise Large Application **[Monolithic Application Architecture]**

Typical requirements in developing and deploying an enterprise large application are
1. Security
2. Resource Pooling (Connection Pooling)
3. Memory Management
4. Global Transactionality
5. Application logging
6. Monitoring
7. JMS Messaging
8. Schedulers
9. Clustering
10. Caching
11. Load Balancing

These are the common services that are required for a typical enterprise class level application. Rather than the developer building all the above services by his own he can take help of Jee Containers which takes of providing all these cross platform services to the applications.




domain = will be created for a group of applications that shares the common resources/services

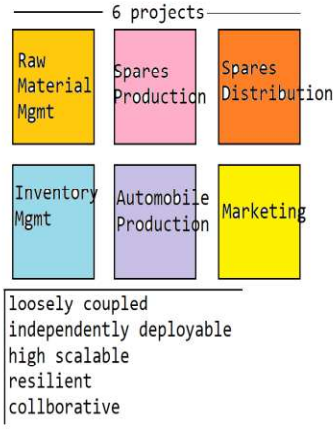



AdminServer
- propagating domain configurations and resources across all the nodes of the cluster
- app deployments are propagated across all the nodes of the cluster

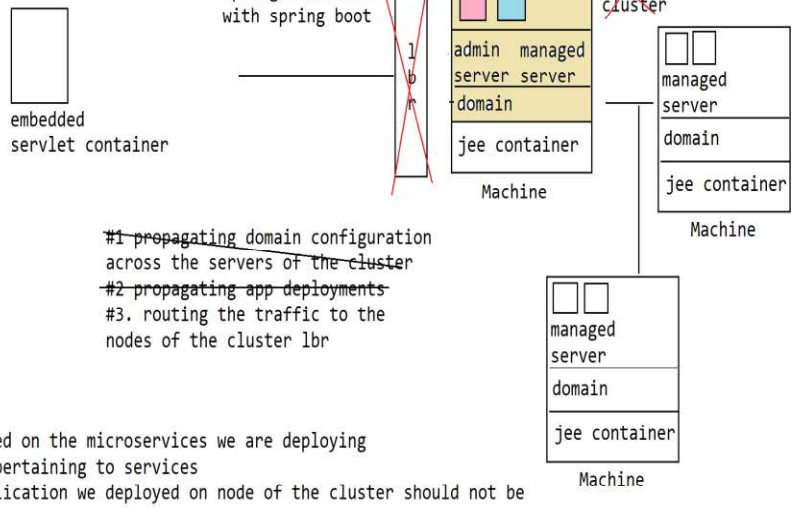
jee cluster resources
distributed jms resources
distributed caching
distributed transactionality
load balancer
session replication (HA)
sticky session

Node Manager
keeps track of or monitors the servers on the cluster

How does MicroServices based applications on delivered and deployed in the realworld?




The Microservices should be deployed by distributing them across the nodes of the cluster few Microservices on 1 node few on another node, it is infeasible to run every microservice on all the nodes of the cluster due to computing capacity limitation.




Each Node of the Cluster should have
1. different domain configuration based on the microservices we are deploying
2. different resource configurations pertaining to services
3. The applications/microservices application we deployed on node of the cluster should not be propagated to other nodes of the cluster

**But in the world of JEE Container managing the application/resource deployment across the nodes of the cluster is the main feature which we dont need**

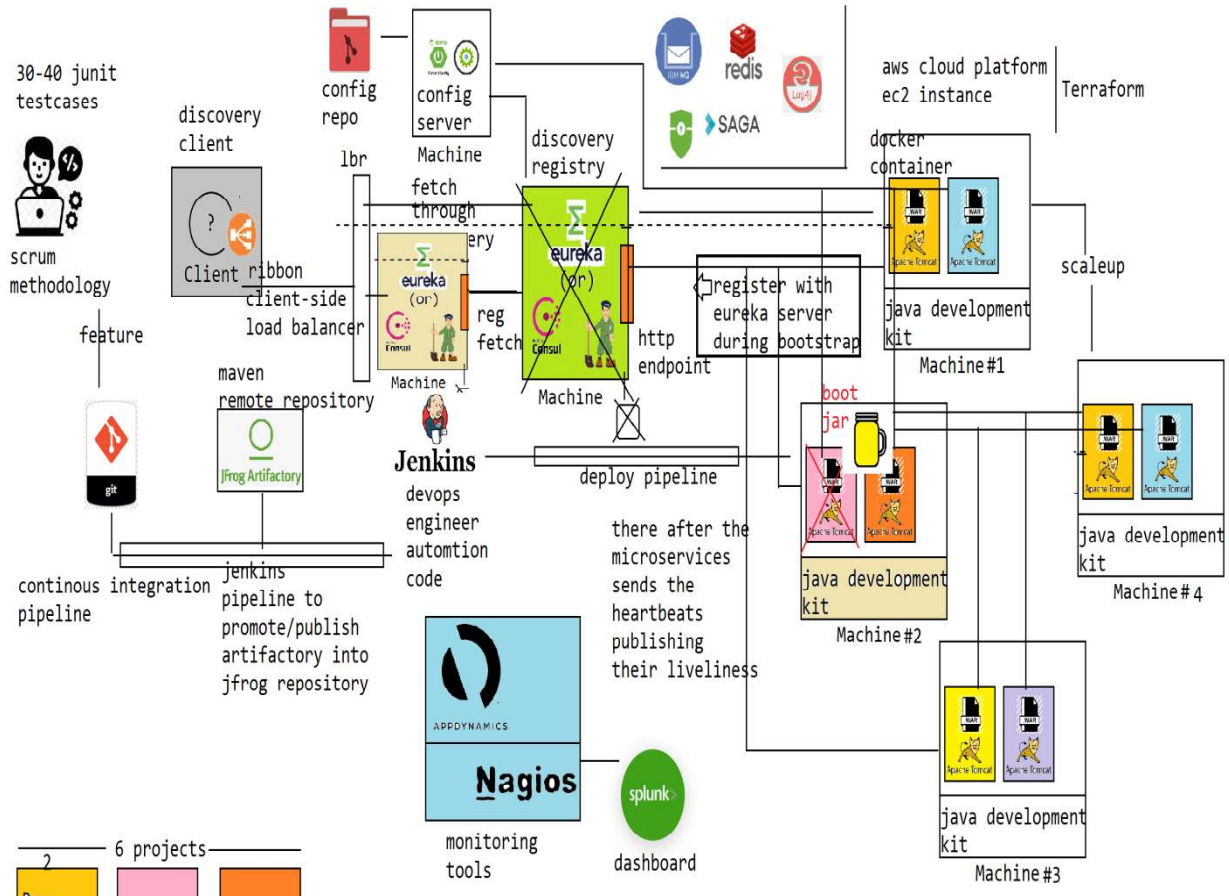
we can get rid of cluster management but we can use cross-platform services provided by jee container like
1. connection pooling
2. jms resources
3. security
4. transactionality
5. cache
6. logging

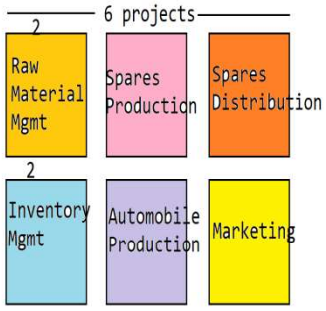
Not all the microservices requires all the cross-platform services provided by the jee container, few microservices uses connection pooling, few other microservices requires jms resources etc.
If we deploy microservices on jee container, it becomes heavy weight solution in running the application where most of the services provided by jee container are not used but we need still use high computing capacity in running the applications (because of all the features/services bootstrapped as part of jee container)

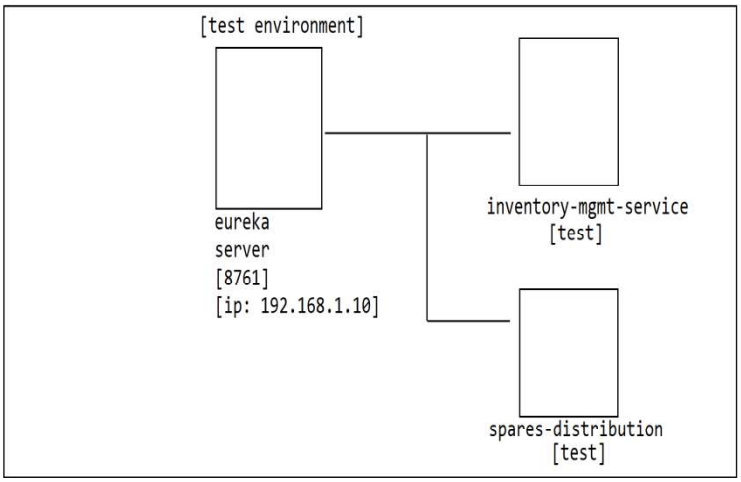
Instead of it we can use the cross-platform services through many of the third-party libraries which are pluggable and configurable based on our requirement
1. Connection Pooling = dbcp, c3p0, proxool, hikari etc
2. jms resources = JMS MQ Server, Kafka, Active MQ, Rabit MQ etc
3. Security = Spring Security
4. Transactionality = delcarative transaction management libraries are available like Spring Transactionality
5. Caching = Distributed caching libraries are available like Coherence, EHCahe, JCache, SwarnCache, RedisCache, MemCache
6. Logging = log4j, sl4j, common-logging, logback

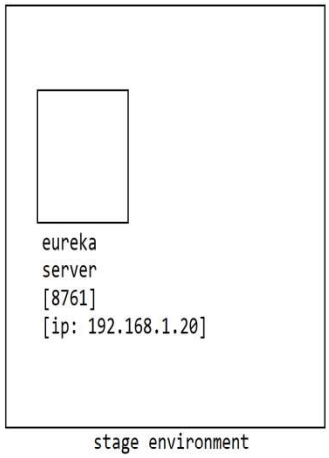
Based on our Microservice requirement we can choose any of these third-parties and integrate into our application using Spring Framework with Spring Boot so that huge amount of efforts in leveraging the third-party apis are reduced.

different nodes has different microservices running, the lbr is not an intelligent component which doesnt have information about which nodes are running with which microservices, upon receiving the request the lbr simply routes the request to the node of the cluster based on load/routing algorithm we configured which doesnt suits for microservices deployment


30-40 junit testcases
scrum methodology
feature
git
continous integration pipeline
discovery client
Client ribbon
client-side load balancer
config repo
config server
Machine
lbr
fetch through
ery
eureka (or)
reg fetch
Machine
maven remote repository
JFrog Artifactory
jenkins pipeline to promote/publish artifactory into jfrog repository
Jenkins
devops engineer automtion code
discovery registry
eureka (or)
consul
Machine
http endpoint
deploy pipeline
there after the microservices sends the heartbeats publishing their liveliness
redis
SAGA
aws cloud platform ec2 instance
Terraform
docker container
register with eureka server during bootstrap
java development kit
Machine#1
scaleup
boot jar
java development kit
Machine#2
java development kit
Machine#4
java development kit
Machine#3
APPDYNAMICS
Nagios
monitoring tools
splunk
dashboard


6 projects
2
Raw Material Mgmt
Spares Production
Spares Distribution
2
Inventory Mgmt
Automobile Production
Marketing


[test environment]
eureka server [8761] [ip: 192.168.1.10]
inventory-mgmt-service [test]
spares-distribution [test]


eureka server [8761] [ip: 192.168.1.20]
stage environment

## Why dont we use JEE Container for deploying and deliverying an Microservice application?

1. Not all the Microservices are running on all the nodes of the cluster, as those are loosely coupled and independently deployable. We scatter Microservices across the nodes of the cluster where each node may run 1 or more microservices based on computing capacity.

In this case we dont need cluster management services like

1.1 application deployment

1.2 resource propagation

1.3 domain configuration management etc

as different microservices requires different resources to run and with different domain configurations, so each node of the cluster is unique from another.

2. Cross-Platform Services

Each Microservice requires mostly few of the enterprise class-level services offered by the jee container. As the microservices itself is small and may not use all the features provided the jee container using an jee container in running an microservices seems to be an heavy weight solution, because the features offered by jee container or not configurable or pluggable, those are bootedup/loaded by default during server startup which increases the utilization of computing resources

## 3. Load Balancer

**Since not all the microservices runs on all the nodes of the cluster, routing the incoming request for a microservices requires service discovery in identifying where the microservices is running on which node of the cluster.**

**A load balancer is not intelligent enough in tracking or identifying the services and their availabiblity on the cluster due to which we cannot use load balancer component provided as apart of jee container**

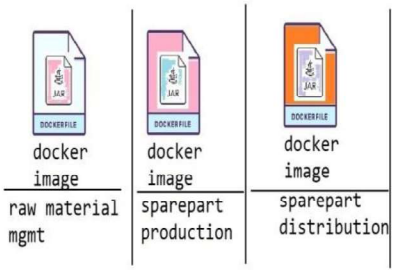
**By considering all the above we can conclude there is no need of jee container for deploying and delivering the microservices application.**

**Due to the above reason people use embedded servlet containers or standlone servlet containers for deploying and delivering microservices application.**

---

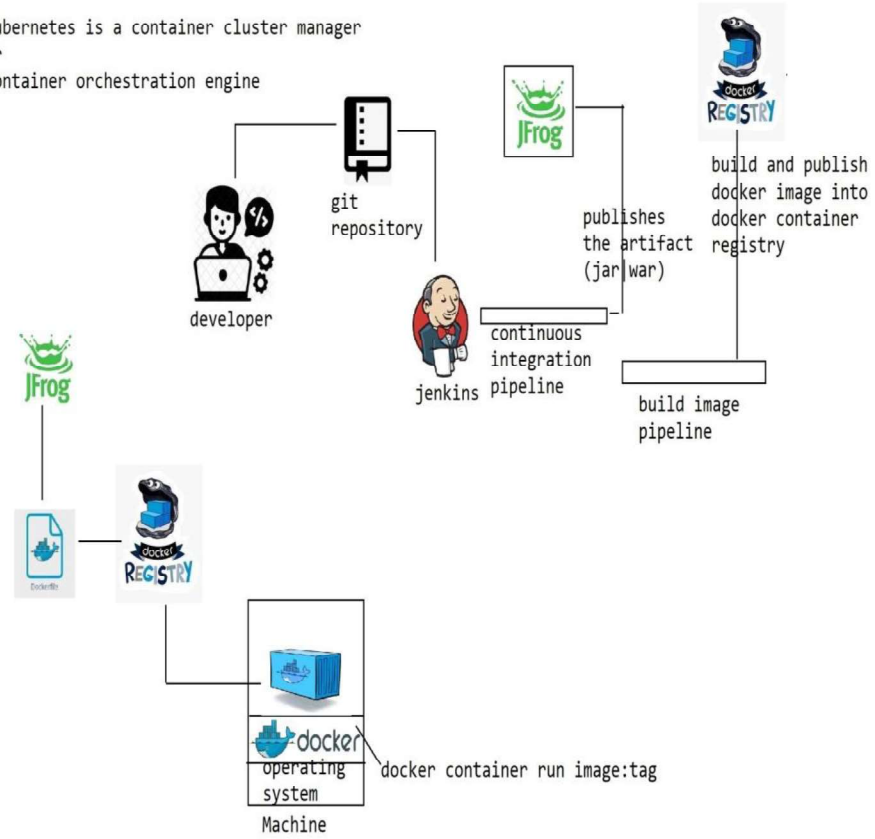
docker is an containerization technology
we can package an application along with softwares/tools and configurations required to run the application inside an container image and we delivery the application as an image. We can create an container out of the image build which runs/executes an application in isolated environment independent of other applications or containers running on the same environment.

advantages:-
1. Each application can be executed isolated from another application on the same env, so that we can easily run multiple applications on same machine by effectively utilizing the computing capacity of the machine.
2. The application deployer dont need to know what is the underlying application that is packaged and he dont need know the instructions related to running the application. docker abstracts the details of the underlying application and provides docker CLI commands that automates the workflow execution of packaging and running any application using docker.

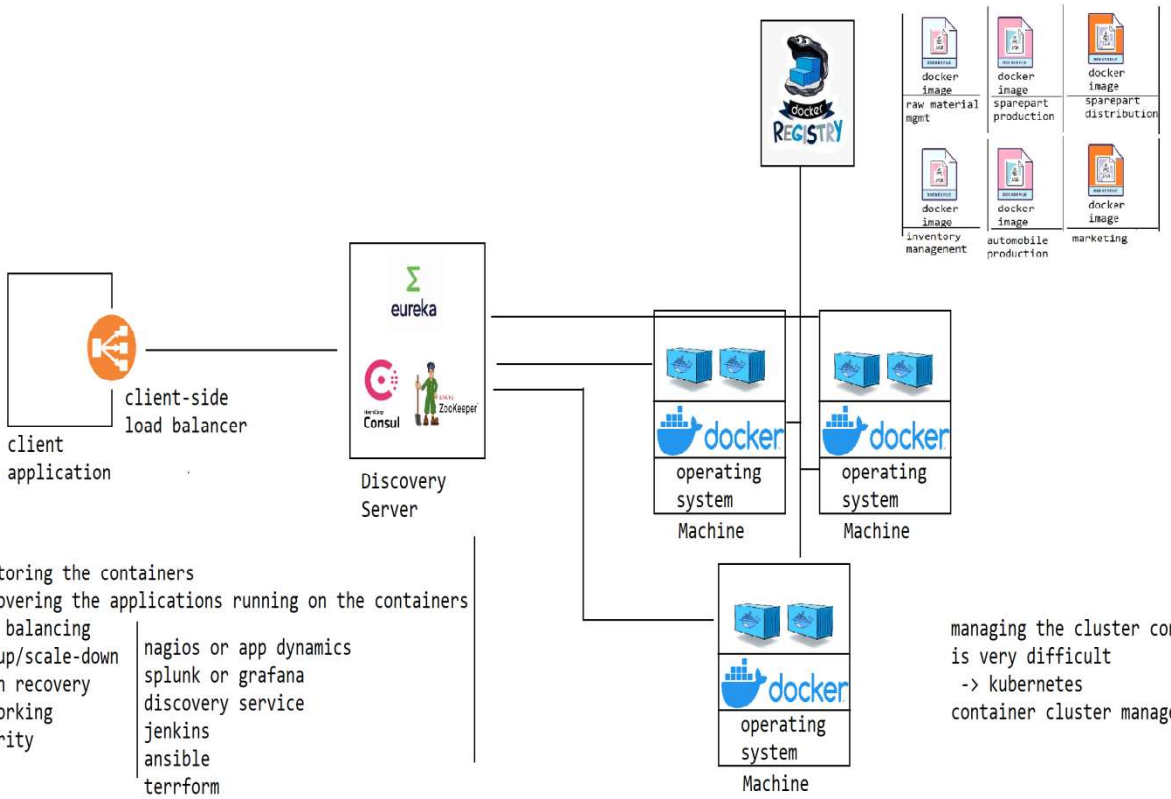



containerization technology
- application can be executed in an isolated manner independent of other applications running on the same env
- abstracts the underlying application, where people dont need know the details of the application to run it, docker provides cli through which we can achieve workflow automation in running the application
- quickly achieve application delivery

Kubernetes is a container cluster manager
or
Container orchestration engine




docker image = application packaged with software utilities/tools required for running the application in an isolated manner is called an image
docker container = instance or a application under executed created from docker image




1. monitoring the containers
2. discovering the applications running on the containers
3. load balancing
4. scalup/scale-down
5. crash recovery
6. networking
7. security

- nagios or app dynamics
- splunk or grafana
- discovery service
- jenkins
- ansible
- terrform
- ribbon

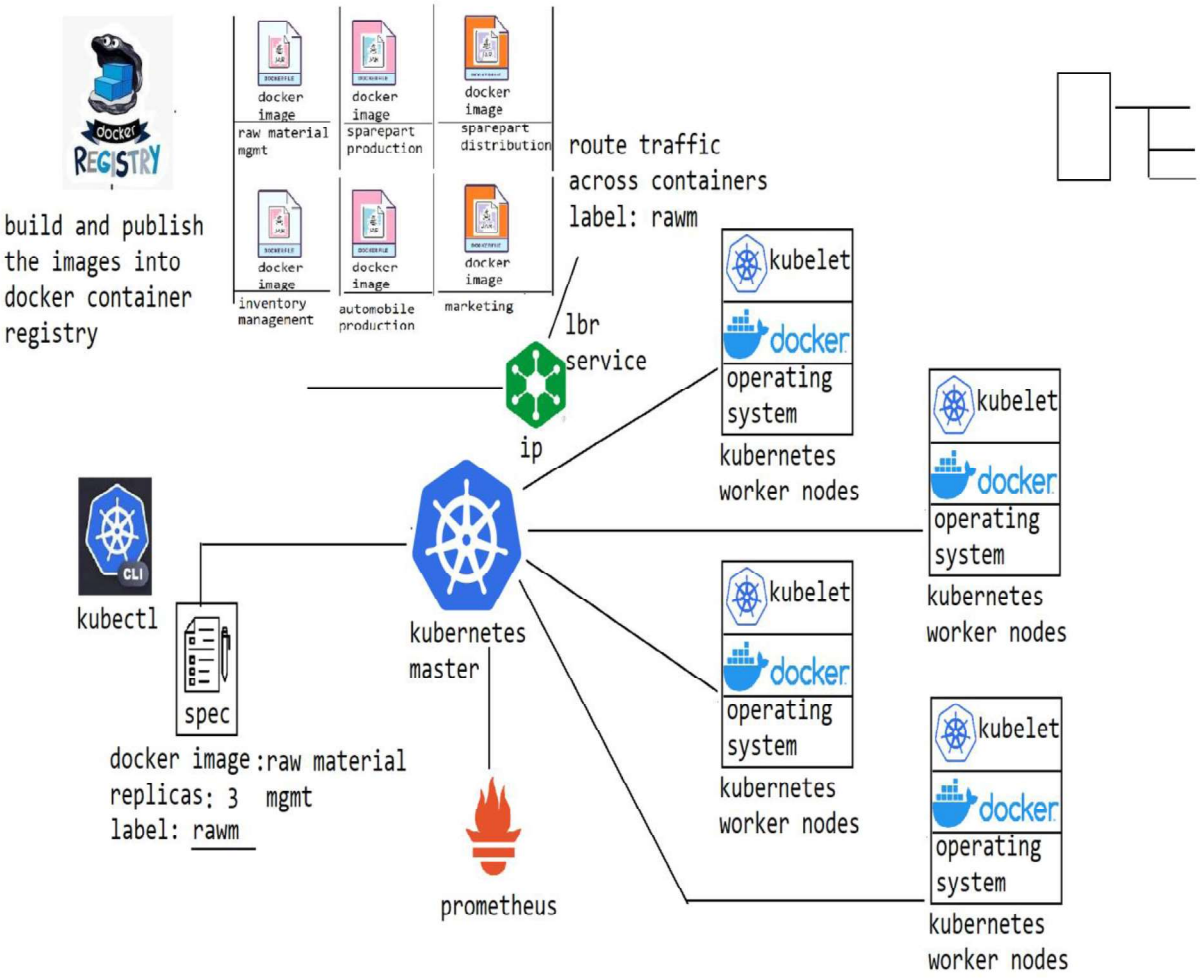
managing the cluster containers
is very difficult
-> kubernetes
container cluster manager

eureka server
ribbon
spring cloud
- config server
- config client
api gateway
spring circuit breaker
feign

-

basic of kafka
[saga design pattern]

design patterns of microservices
- cqrs
- api composer
- saga




## Why do we need Spring Cloud Config Server? What is purpose of it?

**When we are working on Microservices based application development, if we package the application configuration within the application by placing it as**

part of application.yml (or any other configuration file) and if we ship the application, we run into several problems as described below.

1. every change in the configuration requires a rebuild/repackage of the application and should produce a new version of the artifact.

2. if we are running the microservice application in a containerized environment, we need to rebuild the container image with the latest artifact version produced out of our application which requires significant amount of efforts in baking the changes

3. We should identify the instances of the microservices that are running across the nodes of the cluster and should stop the exiting instances and kickoff the never version of the microservices which we have build. identifying the nodes and rolling the newer version of Service is a tedious job

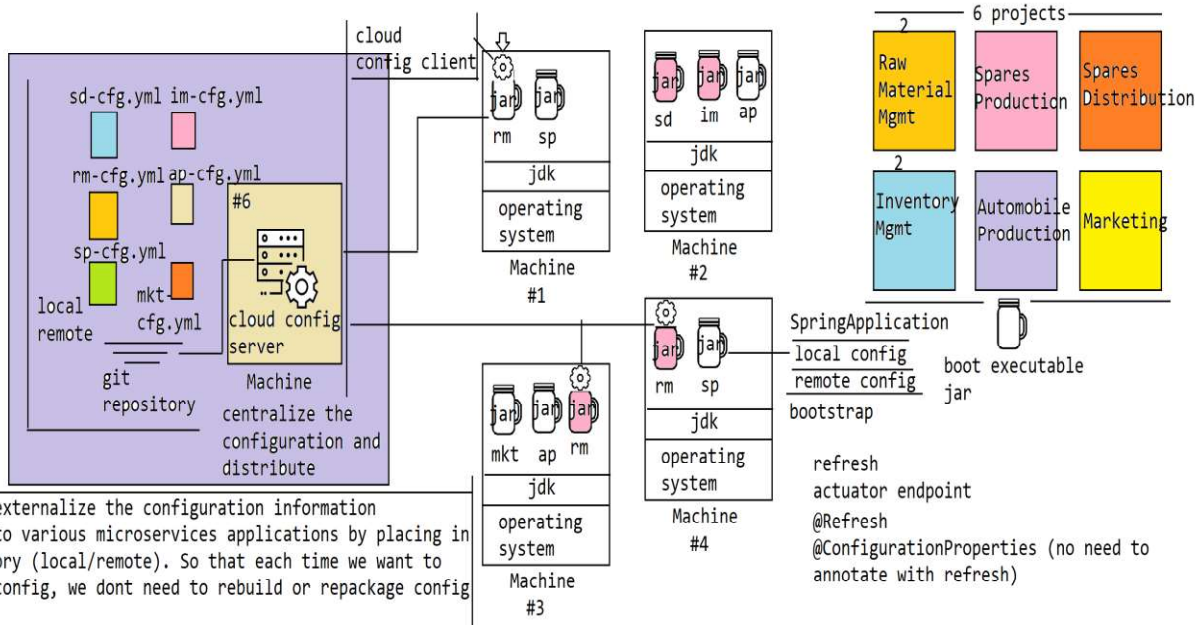
4. If it is an containerized environment we need to take the help of devops engineer in identifying and rolling out the new versions of the container which requires huge coordination across the teams in rolling a simple configuration change

5. during the time of rolling the newer versions of the service with configuration changes, there will be a higher chance of impact in service availability and performance of that microservice

By considering all of the above looks like a small configuration change requires huge time, and team coordiation, cost in rolling the changes and impact of

**availability and performance of the service, how to overcome this problem?**

**Spring Cloud module has provided Spring Cloud Config Server using which we can externalize the configuration of a microservice application and distribute across the instances of the microservices centrally.**




we need to externalize the configuration information pertaining to various microservices applications by placing in git repository (local/remote). So that each time we want to change the config, we dont need to rebuild or repackage config server.

during the time of booting up the config server it pulls the application configurations from the git repository and loads into the memory of config server

how to modify the configuration changes of an application?
goto git repository modify the configuration pertaining to the application in corresponding application configuration file and commit and push the changes to the repository.

we dont need to restart or reboot the config server, rather config server periodically polls the git repository verifying any configuration changes in the files by checking against commit hash, if there is change the commit hash it will reloads that specific configuration file into the config server automatically.

```
@RestController
@RequestMapping("/configserver")
class ConfigServer {
  private Map<String, List<PropertySource>> configRegistry;
  @Value("git.url")
  private String gitRepositoryUrl;

  public ConfigServer() {
    // pull the git repository and load the configurations
into the PropertySources
    // file-name key and List<PropertySource> yaml file
contents
  }

  @GetMapping("/{serviceName}")
  public byte[] readConfig(String serviceName) {
    // returns the yaml file as part of the response
  }
}
deployed and run on an embedded servlet container running on
a port no
```

In general we package configuration information pertaining to an application within the application itself by writing it as apart of application.yml in spring boot application. but when it comes to microservices application packaging the configuration within the application brings lot of problems as the microservices are distributed and ran across the nodes of the cluster of machines.

**problems:**

1. every change within the configuration requires recompilation, repackaging and redeployment/restart of the application which requires significant rework to be done by the developer.

2. if we are containerizing the application then we need to repackage the newer version of the artifact into containerized image again and upload into container repository for distribution

3. identify the nodes on which the microservice application is running is very difficult to roleout the new version of the services with configuration changes

4. if the services is running out of a docker container we need coordination across the teams in rolling out the configuration changes or newer version of the container image

5. during the time of rolling the newer version of the service we might encounter service loss or unavailablity of service and even performance issues which will impacts the business temporarily

From all the above we can understand packaging the configuration information within the microservices

application requires significant efforts, more time and higher cost in rolling the changes.

To help us in distributing the configuration information in a microservices based cloud deployments spring framework has provided spring cloud config server and config client as part of Spring cloud module.

#1 Spring Cloud

ConfigServer:

---

inventory-mgmt-service

```
|-src
    |-main
        |-java
        |-resources
            |-application.yml
|-pom.xml
```

stock

stock_no  stock_nm  description  quantity  unit_price

```
package com.ims.entities;
@Entity
```

```
@Table(name="stock")
class Stock {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    int stockNo;
    String stockName;
    String description;
    int quantity;
    double unitPrice;
    // accessors
}

package com.ims.dto;
class StockDto {
    int stockNo;
    String stockName;
    int quantity;
    double unitPrice;
    // accessors
}

package com.ims.repositories;
```

```
interface StockRepository extends JpaRepository<Stock,
Integer> {
    List<Stock> findByStockName(String stockName);
}


package com.ims.service;
@Service
@ConfigurationProperties(prefix = "stock")
class StockService {
    @Autowired
    private StockRepository stockRepository;

    private String transportType;
    private int slaDays;

    public List<StockDto> getStocks(String stockName)
{}
}


@RestController
@RequestMapping("/stock")
class StockApiController {
    @GetMapping(value="/{stockName}", produces =
{MediaType.APPLICATION_JSON})
```

```
    public List<StockDto>
getStocks(@PathVariable("stockName") String stockName)
{


    }

}


@SpringBootApplication

class InventoryMgmtApplication {

    public static void main(String[] args) {


    SpringApplication.run(InventoryMgmtApplication.clas
s, args);

    }

}
```

**application.yml**

---

```
server:

    port: 8081
```

In the above application we should externalize the db configuration and even application component specific configuration (transportType, slaDays) and place them

as part of cloud config server since those are going to change over the time.

---

**#1**

**for our microservice application create an config repository in git which is global across all the services our application. we are building automobile application which has been broken down into 6 microservices; Now to hold configuration information pertaining to all the microservices of our automobile application let us create a config repository**

**1.1 create a directory under the d:**

**d:\automobile-config-repo**

**1.2 goto the directory and do git init .**

**this will initialize an empty git local repository**

1.3 create an configuration file to hold configuration information for inventory management service

d:\automobile-config-repo

|-inventory-mgmt-service.yml

|-rawmaterial-mgmt-service.yml

|-automobile-production-service.yml

|-spares-production-service.yml

**|-spares-distribution-service.yml**

**|-marketing-service.yml**

**inventory-mgmt-service.yml**

```
spring:
    datasource:
        url:
jdbc:mysql://localhost:3306/inventorymgmtdb
        driverClassname: com.mysql.jdbc.Driver
        username: root
        password: root
    jpa:
        show-sql: true
        generate-ddl: true
stock:
    transportType: air
    slaDays: 2
```

and write the relevant configuration we want to externalize for each of the microservices in the respective service configurations files

1.4 git add -all

1.5 git commit -m "added configurations"

---
---

## CloudConfig Server:

In order to distribute the microservice application configuration across the n/w we need to build an ConfigServer HttpEndpoint, which will takes the request from microservice application and dispatches the service configuration yml as a response.

We need to configure CloudConfig HttpEndpoint with git config repo url, so that during the bootup it loads the yml configuration into the local cache and upon receiving the request with a service name as an input the ConfigServer Endpoint can locate service specific configuration and distribute as a response back to microservice application

Since it is a common requirement of distributing configuration information across the microservices over the n/w, instead of we writing the ConfigServer Endpoint, the spring cloud module has provided the ConfigServer component to us, which we can directly use.

All we have to do is configure the ConfigServer HttpEndpoint as a bean definition within the ioc container of our application, so that upon receiving the request, the DispatcherServlet by taking the help

of RequestMappingHandlerMapping/Adapter can dispatch the request to ConfigServer Endpoint.

While configuring the ConfigServer as a bean definition we need to inject git uri of the config repo to load and distribute the configuration.

**#1 create a spring webmvc application**

**#2 configure or programmatically register DispatcherServlet / ContextLoaderListener to the Servlet Container**

**#3 configure HandlerMapping, MessageConverters and ConfigServer Endpoint as a bean definitions within dispatcher-servlet.xml or WebMvcConfig.class**

**#4 package and deploy the application on the Servlet Container**

The ConfigServer HttpEndpoint has been provided to us as apart of SpringCloud Config Server module.

Instead of it we can take the advantage of Spring Boot and AutoConfigurations in building and running the ConfigServer endpoint

If we are using Spring boot, all of the components of Spring WebMvc like

- DispatcherServlet/ContextLoaderListener will be configured to the ServletContainer by AnnotationConfigServletWebServerApplicationContext

- HandlerMapping, MessageConverters are autoconfigured through AutoConfiguration classes.

- The application is packaged as spring boot executable jar and deployed on embedded servlet container

The only thing we need to is we need configure CloudConfig HttpEndpoint as a bean definition in SpringApplication class

```
@SpringBootApplication
@EnableConfigServer
class CloudConfigServerApplication {
    @Autowired
    private Environment env;


    public static void main(String[] args) {

    SpringApplication.run(CloudConfigServerApplication.
class, args);
    }
}
```

**application.yml**

```
server:
```

```
    port: 8888
spring:
    cloud:
        config:
            server:
                git:
                    uri:
                        -
file://${user.home}/automobile-configrepo
```

We want to externalize the microservices configuration and want to access the configuration distributedly across the network. Todo so we need an ConfigServer (REST) HttpEndpoint which will load the configuration and cache it and distributes to all overs the instances of microservices across the cluster.

Since it is a common requirement of any microservice application to externalize, centralize and distribute config info, instead of we writing the endpoint the spring cloud module has provided cloud config server httpendpoint to us.

**Even though the Spring Cloud module has provided Http(REST) endpoint, in order to expose we need to do the following**

**1. create web mvc application**

**2. configure DispatcherServlet and ContextLoaderListener programmatically to Servlet Container**

**3. configure HandlerMapping, HandlerAdapter, MessageConverters and ConfigServer (Endpoint class provided by CloudConfig Module) as bean definitions through WebMvcConfigurer class**

**4. configure ConfigServer with git repository URI**

**5. package the application as war file**

**6. deploy on tomcat/servlet container and start it, then the endpoint will become accessible.**

**Instead of building and exposing it as a RestController using webmvc, we can use Spring Boot module along with Spring Cloud module.**

**#1 Create an SpringBoot WebMvc Application**

**#2 add starter dependencies as**

- **spring-boot-starter-web**
- **spring-cloud**

#3 we dont need to configure DispatcherServlet and ContextLoaderListener programmatically to Servlet Container, rather SpringApplication class will create AnnotationConfigServletWebServerApplicationContext, which will internally takes care of instantiating DispatcherServlet, ContextLoaderListener and register

them with the EmbeddedServlet container during the startup of the Server

#4 The rest of the spring mvc components like HanderMapping, HandlerAdapter, MessageConverters are configured as bean definitions in ioc container through auto-configurations.

#5 we need to configure ConfigServer (REST) endpoint class as bean definition in Java Configuration class and inject git uri into the class

**application.yml**

```
configrepo:
    git:
        uri:
```

```
@SpringBootApplication
class ConfigConfigServerApplication {
    @Autowired
    private Environment env;

    @Bean
    public ConfigServerEndpoint configServerEndpoint()
{
        String gitUri =
env.getProperty("configrepo.git.uri");
```

```
        ConfigServerEndpoint endpoint = new
ConfigServerEndpoint();
        endpoint.setGitRepoUrl(gitUri);
        return endpoint;
    }

    public static void main(String[] args) {

    SpringApplication.run(ConfigConfigServerApplication
.class, args);
    }
}
```

The SpringCloud module has provided ConfigServer Endpoint class, but we have to write the code for configuring it as bean definition within our application, instead of we configuring framework components as bean definitions the spring cloud should provide autoconfiguration classes for configuring Spring Cloud components as bean definitions automatically.

So to help us in quickly brining up the ConfigServer endpoint and other spring cloud components as bean definitins spring cloud module has provided autoconfigurations classes and distributed them as starter dependencies.

**spring-boot-starter-cloud**

**Now how to run Cloud Config Server to centralize and distribute configuration information of microservices?**

```
#1 create spring boot application
#2 add starter dependencies
    - spring-boot-starter-web
    - spring-boot-starter-cloud-config-server
#3 write application.yml
server:
    port: 8888
spring:
    cloud:
        config:
            server:
                git:
                    uri:
                        -
file://${user.home}/repositoryname
#4 write application class

@EnableConfigServer
@SpringBootApplication
```

```
class CloudConfigServerApplication {

    public static void main(String[] args) {

    SpringApplication.run(CloudConfigServerApplication.
class, args);

    }

}
```

Dgfdgfgfgfhfhgfh

## How do we create a cloud config server to centralize and distribute the microservices configuration?

In order to work with Cloud Config Server, the spring cloud module has provided REST HttpEndpoint, which we need to expose it as apart of Config Server Application. We dont need configure Cloud Config Server HttpEndpoint as a bean definition to be exposed, if we create a spring boot web mvc application and enable Spring Cloud Config Server autoconfigurations it automatically takes care of configuring the HttpEndpoint as a bean definition by injecting gitURI and exposes it as an HttpEndpoint.

## #1 create a git config repository

place all of the microservices configurations in their respective yml|properties files and commit/push into the git repository

## #2 create a spring boot application

configure git repository url in application.yml and enable autoconfiguration for CloudConfigServer

**application.yml**

---

```
server:
    port: 8889
spring:
    cloud:
        config:
            server:
                git:
                    uri: "file://d:/work/automobiles-config-repo"
```

```
@SpringBootApplication
@EnableConfigServer
class CloudConfigServerApplication {
    public static void main(String[] args) {

    SpringApplication.run(CloudConfigServerApplication.class, args);
    }
}
```

d:\work\automobile-configrepo [local git repository]

inventory-management-service.yml

```
spring:
    datasource:
        url:
        driverClassname:
        username:
        password:
    jpa:
        show-sql: true
        generate-ddl: false
stock:
    transportType: air
    slaDays: 3
```

**How to access the service configuration we placed in Spring Cloud Config Server?**

send an http get request with URI of the configserver appended with yml filename.

GET http://localhost:8889/inventory-management-service.yml

---

[Microservice]

```
inventory-mgmt-service
|-src
    |-main
        |-java
        |-resources
            |-application.yml
        |-webapp
            |-WEB-INF
|-pom.xml
```

here we dont want to write the service configuration, that is going to change over the course of time in application.yml and package it as part of the application, rather we want to externalize the service configuration in cloud config server and pull over the http request with which we want to boot our microservice application.

The Microservice application is basically an Spring Boot application which will be kicked off from SpringApplication.run(Config.class, args); Now the SpringApplication.run() method will does the below things.

1. creates an empty environment object
2. detects and loads the external configuration into the env object

3. print banner

4. detects the WebApplicationType

    if mvc jars are found it treats as WebApplicationType=WEB and instantiates

    AnnotationConfigServletWebServerApplicationContext

    if webflux jars are found it treats WebApplicationType=REACTIVE and instantiates

    AnnotationConfigReactiveWebServerApplicationContext

    else WebApplicationType=NONE

    AnnotationConfigApplicationContext

5. detects and instantiates and registers spring factories (autoconfigurations)

6. invokes ApplicationContextInitializer

7. Prepare Context

8. Refresh Context

9. invokes CommandLineRunners and ApplicationRunners

10. during the above stages it will publish the different types of events and calls the listeners to perform actions

By default every spring boot application will tries to detect and loads the configuration from local filesystem like application.properties|yml or SystemProperties or Environment Variables of the machine on which we are running.

But in our case along with local configuration, we have our microservices application configuration is placed in ConfigServer also, so we wanted to load Microservice configuration from ConfigServer as well which requires.

1. an REST/HTTP Client program which will invokes the ConfigServer with our MicroService application name and pulls the application configuration with which it should create ioc container

2. The SpringApplication class should not invoke the REST/HTTP Client program in loading the configuration of the MicroService from Cloud Config Server, because this is an additional process required only incase of MicroService application only.

That is where we need One more guy/person who should take the responsibility of invoking Cloud Config Server with Microservice Name by taking the help of REST/HTTP Client program and get the configuration and pass to Spring Application class

When we run the application before SpringApplication class begins execution the process of invoking the CloudConfigServer and accessing service configuration should takes place and pass to SpringApplication class.

**How to pass configuration information to SpringApplication class?**

SpringApplication class during the time of instantiating the ioc container it checks for parent

ioc container within ServletContext (nested bean factories). if the other person can get the Configuration from CloudConfig Server and place the configuration in ioc container and places it in ServletContext.

We need to write an WebApplicationInitializer or ServletContainerInitializer in which we need to write the logic for invoking ConfigServer get the configuration with which create ioc container and place in ServletContext.

There after SpringApplication class will create child ioc container nesting parent container we created.

To summarize we need to write below components:

1. REST/HTTP Endpoint for invoking Cloud Config Server

2. We need a BootStrap Initializer class which will executes during the Server startup, it invokes Cloud Config Server gets the configuration with which create ioc container and place it as parent container in ServletContext, so that SpringApplication can use it in creating the Child.

We dont have to write this code, since it is an common / boiler-plate logic it has been provided to us as part of

spring-cloud-config-client = REST/HTTP Endpoint

spring-cloud-bootstrapping = BootStrap Initializer

**automobile-configserver**

```
|-src
    |-main
        |-java
            |-com.acs
                |-
AutomobileConfigServerApplication.java
        |-resources
            |-application.yml
|-pom.xml
```

```
application.yml
server:
    port: 8889
spring:
    cloud:
        config:
            server:
                git:
                    uri: "file://d:/work/automobile-
configrepo"
```

```
@SpringBootApplication
@EnableConfigServer
```

```
class AutoMobileConfigServerApplication {}
```

How to access the microservice application configuration from the cloud config server?

we need to send an HTTP GET Request pointing to URI of config server followed by yml configuration filename.

```
GET http://localhost:8889/inventory-mgmt-service.yml
```

local git repository

```
d:\work

|-automobile-configrepo

    |-inventory-mgmt-service.yml (service configuration)
```

---

**Microservice integration with ConfigServer**

**inventory-mgmt-service**

```
|-src

    |-main

        |-java

        |-resources

            |-application.yml

        |-webapp
```

|-WEB-INF

|-pom.xml

we dont want to package the microservice configuration which is going to change over the course of time inside the application.yml, rather than we want to pull the configuration from cloud config server.

**#1**

**To pull the configuration from cloud config server we need?**

We need HTTP/REST Client Program that will takes the service name as an input and sends a GET request to config server and pulls the service configuration and returns. Now to the HTTP/REST Client program we need to populate the ConfigServer Base URL, to invoke with service name.

**#2 After getting the Configuration we want to load and create ioc container, how?**

during the time of bootstrapping the application we want to invoke the ConfigServer through the help of HTTP Client program and pull the microservice configuration and load into ioc container.

Microservices mean we are building Spring Boot Application, wherein the SpringApplication.run() will

takes the responsiblity of bootstrapping the application.

Since the SpringApplication is load the configuration information of the application into environment object before creating the ioc container, can we ask **SpringApplication class to invoke ConfigServer and load microservice configuration?**

No, since pulling the configuration from ConfigServer is a specific requirement of a cloud/microservice application, SpringApplication class should not invoke ConfigServer and pull the service configuration as it is not applicable for every type of application.

We need Initializer/Bootstrapper who should execute during the Servlet Container startup after ServletContext has been created, the initializer should pull the Microservice configuration from Cloud Config Server, should create ioc container and place in ServletContext object as a parent ioc container

So that SpringApplication class during the time of creating the ioc container will checks to see the parent container within ServletContext if exists will nest the SpringApplication ioc container as a child with the parent, so that all the cloud config server service configuration will be available to our microservice application through nested ioc containers.

We need to write on Bootstrapper or Initializer class hookup to Servlet Container so that it would be invoked during startup.

Looks like every microservice application during the bootup has to invoke ConfigServer and load the microservice configuration in a parent ioc container, which seems to common/boiler-plate logic, instead of we writing these components Spring Cloud has provided these 2 components as 2 dependencies

spring-cloud-config-client = HTTP/REST Client program

spring-cloud-bootstrap = Initializer

application.yml

---

```
server:
    port: 8081

spring:
    cloud:
        config:
            uri: http://localhost:8889
```

To facilitate in loading the configuration from external sources like a cloud config server into environment object of ioc container, the spring cloud has provided bootstrap dependency.

The spring cloud bootstrap dependency creates an parent context by loading the external source configuration provided by cloud config server

when we add spring-cloud-bootstrap dependency into our project, it creates an bootstrap context by reading bootstrap.yml|properties in case if the file is not available as part of your project, then it looks into application.yml|properties file and reads the config config server uri.

Now the boostrap context reads cloud config server uri from bootstrap.yml|properties and tries to invoke config server endpoint, reads the microservice configuration and creates an env and loads into ioc container and places it as parent context. then hands overs to SpringApplication.

So if we want some properties to be available during the time of bootstrapping the application then place them in boostrap.yml|properties

if we have some properties which are application specific and will not change over the time then place them in application.yml|properties

if we have application specific properties but those will change over the time, then place them in cloud config server

To summarize:

**application.yml**

---

```
spring:
    application:
        name: inventory-mgmt-service
server:
    port: 8081
```

**boostrap.yml**

---

```
spring:
    cloud:
        config:
            uri: http://localhost:8889
```

**How to enable the microservice in pulling the configuration from config server to bootstrap the application?**

```
inventory-mgmt-service
|-src
    |-main
        |-java
        |-resources
            |-application.yml
            |-bootstrap.yml
        |-webapp
            |-WEB-INF
|-pom.xml
```

we need to add 2 dependencies

spring-cloud-config-client

spring-cloud-bootstrap

**note:**

in the previous version of spring-cloud-config-client, it has transtitive dependency as spring-cloud-bootstrap so we dont used to include bootstrap dependency.

in spring-cloud-config-client 2.2.8.RELEASE they removed bootstrap as a transitive dependency, so we need to add both othese dependencies manually

in spring-cloud 3.x the provided spring-cloud-starter-config which is a spring boot starter dependencies that pulls spring-cloud-config-client dependency along with that we need to add spring-cloud-starter-bootstrap as well.

spring-cloud-boostrap will tries to look for bootstrap.yml, if not found it looks for properties in application.properties|yml

in bootstrap.yml we need to add below properties for pulling the configuration from cloud config server

```
spring:
    config:
        import:
            - configserver:http://localhost:8889/
```

in addition to config.import as configserver we can write file: and classpath: also which will loads external configuration during bootstrap from local file system location or classpath location.

in addition to the config server uri specified we need to add spring.application.name in bootstrap.yml as it is required to pull the configuration before the Spring Boot application begins

**bootstrap.yml**

```
spring:
    application:
        name: inventory-mgmt-service
    config:
        import:
            - configserver:http://localhost:8889/
```

if we dont specify the spring.application.name, the it tries to look for default configuration with name application.yml

by default Cloud Config server for every hit in pulling configuration from it goes to git repository fetching the latest configuration file.

**How to configure the micorservice application to pull the configuration from config server?**

**#1 add starter dependencies**

spring-cloud-starter

spring-cloud-starter-config

spring-cloud-starter-boostrap

**#2**

**bootstrap.yml**

---

```
spring:
    application:
        name: inventory-mgmt-service
    config:
        import:
            - configserver:http://localhost:8889/
```

if we dont specify the spring.application.name, by default the boostrap context tries to pull the config from config server with name "application".

there are many ways the spring cloud config server supports reading and loading the configuration into boostrap context like

- classpath: = loads the yml|properties from classpath location of the project

- file: = loads the yml|properties from absolute location of the Filesystem

- configserver: = loads the configuration by hitting config server

application.yml

---

```
server:
    port: 8081
```

it is recommended to use both bootstrap.yml and application.yml in a microservice application.

use bootstrap.yml = for especial wanted to load external from external sources into boostrap context

use application.yml = to load service specific configuration that will not change over the runtime of the application.

---

**What configuration we need to write to bootstrap a cloud config server?**

**application.yml**

```
spring:
    application:
```

```
        name: "automobile-configserver"
    cloud:
        config:
            server:
                git:
                    uri: "file://d:/work/automobile-
configrepo"
                    refresh-rate: 180
```

if we dont specify the refresh-rate it default to "0", so that per each hit to cloud config server it tries to pull the configuration from git repository. by specifying the refresh-rate it caches the configuration for specified number of seconds and there after it reloads from config repository.

---

We dont want everyone to access the Cloud Config Server and access the microservices configuration, inorder to secure it lets us enable spring security and basic authentication.

**#1 add spring-boot-starter-security**

by default during the server startup it generates a default username : admin and password as random number.

now to provide specific username/password with which we want to enable access to cloud config server we need to below security configuration in application.yml

```
security:
    user:
        name: configadmin
        password: welcome1
```

**Config Server**

1. public git repository how to use

2. private git repository with personal access token

https://username:pat@gituri

3. how to secure config server with basic authentication

4. refresh-rate

If there are any configuration changes on cloud config server, after the microservice application has been started. how do we need to reload the configurations changes into microservice without restarting the application.

1. enable refresh actuator endpoint on microservice.

#1. add spring-boot-starter-actuator

#2. enable/expose refresh endpoint on microservice

```
management:
    endpoint:
        refresh:
            enabled: true
management:
    endpoints:
        web:
            exposure:
                include:
                    - refresh
```

we need to send a post request to the microservice refresh actuator endpoint

[POST] http://localhost:8081/actuator/refresh

then all the bean definitions which are annotated with @Refresh or injected with @ConfigurationProperties will be reloaded automatically.

## Spring Cloud Config Server

---

### #1 how to configure public git repository instead of a local git repo

```
spring:
    cloud:
        config:
            server:
                git:
                    uri:
http://github.com/techsriman/automobile-configrepo.git
```

### #2 how to configure private git repository with personal access token?

2.1 goto git hub account and navigate to developer settings and generate an personal access token and copy it

2.2 configure config.import as configserver:http://username:password@repoUrl

```
spring:
    cloud:
        config:
            server:
```

```
                git:
                    uri:
http://techsriman:pattoken@github.com/techsriman/automo
bile-configrepo.git
```

**#3 how to configure refresh-rate for config server?**

if we are using local maven repository the refresh-rate is not applied, every access to the config server will pull the config information from local maven repository.

In case if we are using remote git repository then refresh-rate will allows the config server to cache the config information for specified number of seconds and returns the same copy of the configuration to the microservices even the underlying config in repository has been changed.

```
spring:
    cloud:
        config:
            server:
                git:
                    uri:
http://github.com/techsriman/automobile-configrepo.git
```

refresh-rate: 180

## #4 how to secure the access to the config server?

we can enable spring security and basic authentication enforcing the clients to provide username/password in accessing the config server.

### 4.1 add spring-security-starter

it autoconfigures springSecurityFilterChain, http basic authentication, and authenticationManager with default username: admin, generated password in the console.

we can pass username/password in configuration file as below instead of using generated password.

```
security:
    username: configuser
    password: welcome1
```

instead of the above we can store username/password in database and can write UserDetailsService to enable db based authentication

---
---

## Spring Cloud Config Client

### #1 how to enable pulling configuration from Config Server during bootstrap

bootstrap.yml

```
spring:
    application:
        name: inventory-mgmt-service
    config:
        import:
            - configserver:http://localhost:8889/
[pointing to configserver uri]
```

### #2 how to pull configuration from config server with basic authentication

bootstrap.yml

```
spring:
    application:
        name: inventory-mgmt-service
    config:
        import:
            - configserver:http://localhost:8889/
[pointing to configserver uri]
```

```
    cloud:
        config:
            username: configadmin
            password: welcome1
```

**#3 how to reload the configuration changes into the microservice application?**

1. we need to enable refresh actuator endpoint

2. we need to annotation the bean definitions with @Refresh or @ConfigurationProperties annotation

application.yml

```
management:
    endpoint:
        refresh:
            enabled: true
    endpoints:
        web:
            exposure:
                include:
                    - refresh
```

now we need to send post request to the microservice actuator refresh endpoint to reload the configuration from cloud config server

---
---

**Working with profiles:**

---

we dont need to host multiple config servers for dev, test and prod etc. each microservice by themself requires different configurations based on the environment on which there are running.

So the microservice defines variables profile based configurations by writing multiple properties files per profile or multi-document yml with profiles or multiple yml documents with profile name suffix as shown below.

for eg..

**automobile-configrepo (git repository)**

|-inventory-mgmt-service-dev.properties

|-inventory-mgmt-service-test.properties

|-inventory-mgmt-service-prod.properties

(or)

**automobile-configrepo (git repository)**

|-inventory-mgmt-service-dev.yml

|-inventory-mgmt-service-test.yml

|-inventory-mgmt-service-prod.yml

(or)

**automobile-configrepo (git repository)**

|-inventory-mgmt-service.yml (with multi-doc support)

inventory-mgmt-service.yml

```
---
spring:
  config:
   activate:
    on-profile: dev
store:
  slaDays: 100
  transportType: roadways
...
---
spring:
  config:
   activate:
    on-profile: test
store:
```

```
  slaDays: 200
    transportType: air
...
```

From the above we can understand cloud config server hosts microservice configurations which are profile enabled. when the request comes from microservice for pulling service configuration, the cloud config server checks for active profile and reads the relevant configuration of the services of that profile and returns to the microservice

Now the microservice has to activate the profiles in bootstrap.yml

bootstrap.yml

```
--------------
spring:
  profiles:
   active:
    - dev
-------------------------------------------------------
-----------------------------------------
```

## Discovery Service/Registry/Server

---

In a microservices environment, the services are scatter across the various nodes of the cluster wherein different nodes are running different instances of the services.

Note: The nodes of the cluster are not identical (the cluster doesnt have replicated nodes)

Now in such case how does the client application knows the service he wanted to consume is available on which nodes across the cluster and would be able to consume. To help the clients in discovering the services running across nodes of the cluster we need discovery service/registry.

There are several third-parties comeup with different discovery engines as below.

1. Eureka server
2. Hashicorp Consul
3. Apache Zookeeper

out of the above Netflix eureka is the most popular discovery registry in the market.

A typical discovery registry should support the below aspects of managing and distributing the microservices information

1. every service should be able to register by themself with the discovery service/engine

2. discovery service should keep track of replica of instances on where service is running on. for e.g.. a microservice is running across multiple nodes of the cluster, the discovery engine should register the microservice with multiple replicas of instances rather than multiple microservices. So that loadbalancing across the replicas is possible

3. a discovery server should support discovering the microservice instances based on the service name and availability.

4. the discovery engine should support healthchecks or healthbeats in monitoring the service instance availability, so that it can discover and return instances which are healthy to the client.

5. In addition a discovery server should support slaves/replicas of their own for high availability in production

We need to use one of the discovery service third-party libraries in hosting and running the discovery services on the microservices cluster. To run eureka server or any of the discovery services we need to configure instantiate and host them as part of servlet containers.

In order to help us in quickly working on the Third-party libraries spring boot + spring cloud has provided direct integrations into these engines through auto-configurations.

Now we are going to use Eureka Server as a service discovery engine in hosting and discovering the microservices using spring boot + spring cloud

## Eureka Server

The eureka server has been built on jersey rest api implementation and runs on tomcat server by default. now we need to configure the jersey components, build and package the application on a standalone tomcat server which requires huge amount of efforts in configuring and running the application.

instead of that spring boot + spring cloud has provided autoconfigurations where most of the configurations required to run the eureka server are configured with defaults and allows us to populate external configuration through yml files, so that the rest endpoints will be registered and deployed on embedded servlet containers.

**The Eureka Server is a discovery engine/service, where the microservices during the startup/bootup will register themself with eureka discover server. The eureka server will keeps hold of the services information/endpoints in-memory,** since the service availability information will change periodicaly.

To help the Eureka Server to properly discover and route the clients through active service instances

only, the eureka server supports heartbeats. Each service by themself has to send periodical heartbeats letting the server knows the availability of the service.

So that when the client send a discovery request to the eureka server, the server looks in in-memory registry to identify the active endpoints and returns to the client.

---

**How to host the Eureka Service Discovery engine using Spring Boot + Spring Cloud?**

**pre-requisite:-**

Setup eureka server configuration in configrepo and host it as part of config server, so that we can distribute config server configuration across the replicas.

#1 setup an spring boot + spring cloud application

#2 add the below starter dependencies

- spring-cloud-starter
- spring-cloud-starter-config
- spring-cloud-eureka-server

#3 in the boostrap.yml write the cloud config in pulling the eureka server configuration from config server during bootstrap

```
spring:
    application:
        name: automobile-eurekaservice
    config:
        import:
            - configserver: http://localhost:8889/
```

application.yml

```
server:
    port: 8761 (default port)
```

#4

every eureka server comes up with discovery client libraries inside it.

In general, discovery client helps in talking to eureka server in discovering the microservices.

**why does eureka server comes up with discovery client?**

In production deployments we are going to bringup an standby eureka server replica of the master so that incase of crash all the clients can make use of the standby server for service discovery.

The slave or replica eureka server during the startup/bootup will discover its master and pull the service registry through the help of discovery client libraries, so that discovery client library is packaged as part of eureka server by default.

Note: by default as part of eureka server the discovery client is enabled.

#5 automobile-eurekaservice.yml

```
eureka:
    client:
        register-with-eureka: false
        fetch-registry: false
```

with the above configuration we turned off the discovery client in the eureka server

**register-with-eureka = true (default)**, which means the eureka server should discover and register with other eureka servers on the cluster. since we are running only one instance of eureka server we need to set this property false, otherwise we will be poluted with logs of failure discovery.

**fetch-registry = true (default)** which indicates pull the service information from other eureka server and

populate which is applicable for standby eureka server, as we are running master instance we need to turnoff

**#5**

```
@SpringBootApplication

@EnableEurekaServer // kickoff autoconfigurations in enabling the endpoints

class EurekaServerApplication {}
```

**Eureka Server**

---

Eureka Server acts as a Service Discovery Registry, each microservice during the boostrap will register themself with the eureka server by publishing their endpoint information for discovery. The client applications would discover the Microservice and access them through the help of Eureka Server.

To ensure or keep track of liveliness of the services, each microservice will send the heartbeats to the eureka server publishing their availability, based on which eureka server will route the clients to the available services only.

---

**How to work with eureka server?**

**#1__**

**automobile-configserver**

```
|-src
    |-main
        |-java
            |-AutomobileConfigServerApplication.java
        |-resources
            |-application.yml
|-pom.xml
```

```
server:
    port: 8889
spring:
    cloud:
        config:
            server:
                git:
                    uri:
http://github.com/techsriman/automobile-configrepo.git
                    refresh-rate: 180
```

automobile-configrepo [git repository]

inventory-mgmt-service.yml

automobile-eurekaserver.yml

automobile-eurekaserver.yml

---

In the Eureka Server configuration, we dont need to write any configuration pertaining to eureka server, rather we need write configuration for turning off the client discovery on the eureka server.

```
eureka:
    client:
        register-with-eureka: false
        fetch-registry: false
```

#2

automobile-eurekaserver

```
|-src
    |-main
        |-java
            |-AutomobileEurekaServerApplication.java
        |-resources
            |-application.yml
            |-bootstrap.yml
```

```
|-pom.xml
    -spring-cloud-starter
    -spring-cloud-starter-config
    -spring-cloud-starter-boostrap
    ------------------------------ for pulling eureka server configuration from cloud config server
    -spring-cloud-starter-netflix-eureka-server
```

application.yml

```
server:
    port: 8761
```

bootstrap.yml

```
spring:
    application:
        name: automobile-eurekaserver
    config:
        import:
            - configserver:http://localhost:8889/
    cloud:
        config:
            username: configadmin
            password: welcome1
```

```
@SpringBootApplication
@EnableEurekaServer
class AutomobileEurekaServerApplication  {
    public static void main(String[] args) {

    SpringApplication.run(AutomobileEurekaServerApplica
tion.class, args);
    }
}
```

---

**How to register Microservice with eureka server?**

Now its the time where we need to register microservice with eureka server during its bootstrap and publish heartbeats to eureka server.

In order to do this within the microservice we need to add eureka discovery client library.

```
inventory-mgmt-service
|-src
    |-main
        |-java
        |-resources
```

```
|-pom.xml
-spring-cloud-starter-netflix-eureka-client
```

netflix eureka client provides libraries to register microservice with the eureka server. we need to configure the information about eurekaserver to register with eureka in microservice application.

**inventory-mgmt-service.yml**

```
---
spring:
    config:
        activate:
            on-profile: dev
  datasource:
    driver-class-name:
    url:
    username:
    password:
stock:
  transportType: roadways
    slaDays: 10
eureka:
  client:
```

```
        register-with-eureka: true
        fetch-registry: false
        service-url:
            defaultZone: http://localhost:8761/eureka
...
---
spring:
    config:
        activate:
            on-profile: test
  datasource:
    driver-class-name:
    url:
    username:
    password:
stock:
  transportType: airways
    slaDays: 1
eureka:
  client:
        register-with-eureka: true
        fetch-registry: false
        service-url:
```

```
            defaultZone: http://localhost:8762/eureka

...
```

---

**How to enable HA for eureka server?**

For high availability we need to run 2 instances of eureka server on 2 different computers, so that if one of it crashes the other machine will be available for discovering services.

when we run multiple eureka servers for high availability one of the server becomes master, and other eureka servers becomes replicas

**#1 The microservices registers/publishes** their information onto the master eureka discovery server only, and even periodical heartbeats are send to master server only.

The microservices are not aware of the replicas, because if microservice has to register with replicas as well, along with master and send heartbeats to replicas, the performance of the microservice will degrade.

**#2 The Replica Eureka Server will register itself with Eureka Master as a Replica** and fetch service registry information from Master with which it will be hosted.

**#3 Now to the client applications will discover the microservice endpoints through LBR configured on eureka server cluster.**

#1

automobile-eurekaserver

```
|-src
     |-main
          |-java
          |-resources
               |-application.yml
               |-boostrap.yml
|-pom.xml
```

application.yml

---

```
spring:
     config:
          activate:
               on-profile: master
```

```
server:
    port: 8761
spring:
    config:
        activate:
            on-profile: replica
server:
    port: 8762
```

boostrap.yml

```
spring:
    application:
        name: automobile-eurekaserver
    config:
        import:
            - configserver:http://localhost:8889
```

automobile-eurekaserver.yml

```
spring:
    config:
        activate:
            on-profile: master
eureka:
```

```
    client:
        register-with-eureka: false
        fetch-registry: false

spring:
    config:
        activate:
            on-profile: replica
eureka:
    client:
        register-with-eureka: true
        fetch-registry: true
        service-url:
            defaultZone: http://localhost:8761/eureka
```

#1

```
automobile-configserver
|-src
  |-main
    |-java
      |-CSApp.java
    |-resources
      |-application.yml
|-pom.xml
```

```
application.yml
server:
  port: 8889
spring:
  cloud:
   config:
    server:
      git:
       uri: github-repouri
```

@EnableConfigServer

spring-cloud-starter-config
spring-cloud-starter
spring-boot-starter-security

[centralizes and distributes the configuration across all the microservices]

[git]

```
automobile-configrepo
|-inventory-mgmt-service.yml
```

```
spring:
  datasource:
    url:
    driver-class-name:
    username:
    password:
  jpa:
    show-sql: true
stock:
  transportType: air
  slaDays: 2
eureka:
  client:
   register-with-eureka: true
   fetch-registry: false
   service url:
    defaultZone:
http://localhost:8761/eureka
```

#2

@EnableEurekaServer

```
automobile-eurekaserver
|-src
  |-main
    |-java
    |-resources
      |-application.yml
|-pom.xml
```

```
application.yml
server:
  port: 8761
eureka:
  client:
   register-with-eureka: false
   fetch-registry: false
```

spring-cloud-starter-netflix-eureka-server

#3

```
inventory-mgmt-service
|-src
  |-main
    |-java
    |-resources
      |-application.yml [static]
      |-bootstrap.yml [pull ccs]
|-pom.xml
```

```
application.yml
server:
  port: 8082
```

```
bootstrap.yml
spring:
  application:
    name: inventory-mgmt-service
  config:
   import:
     -configserver:http://localhost:8889
```

spring-cloud-starter
spring-cloud-starter-config
spring-cloud-starter-bootstrap
spring-cloud-starter-netflix-eureka-client

add the netflix-eureka-client library to the microservice, the default behaviour of discovery client is to register with eureka server and will fetch registry during the startup.

microservice configuration we need to add discovery client related configuration enabling him to register with eureka server and dont fetch registry

```
eureka:
  client:
    register-with-eureka: true
    fetch-registry: false
    service-url:
      defaultZone: http://locahost:8761/eureka
```

## discovery client

---

The client application/program inorder to access the microservices running across the nodes of the cluster they should know the information about which microservice is running on which node of the cluster. it would be very difficult for the client byhimself in tracking and accessing the microservice so he need to take the help of eureka server.

eureka server has exposed HTTP Endpoints that enables the client applications to discover or search for microservices and their location to access. so in our client applicationion we need to write HTTP Client program in invoking eureka server endpoints to identify the microservice information to access.

instead of we writing the HTTP Client program in invoking the eureka server in discovering the microservices the eureka has provided eureka discovery client api/library, which can be used directly in accessing the information about the microservice

The eureka discovery client library has been marked as deprecated in recent versions of spring cloud and has provided spring cloud discovery client library as an alternate to it.

The discovery client api is used for communicating with eureka server in identifying the information about where the microservices instances are running. once we get the information we need use RestTemplate (api) for invoking the microservice

**automobile-dealer**

```
|-src
    |-main
        |-java
        |-resources
            |-application.yml
|-pom.xml
    |-spring-boot-starter-web
```

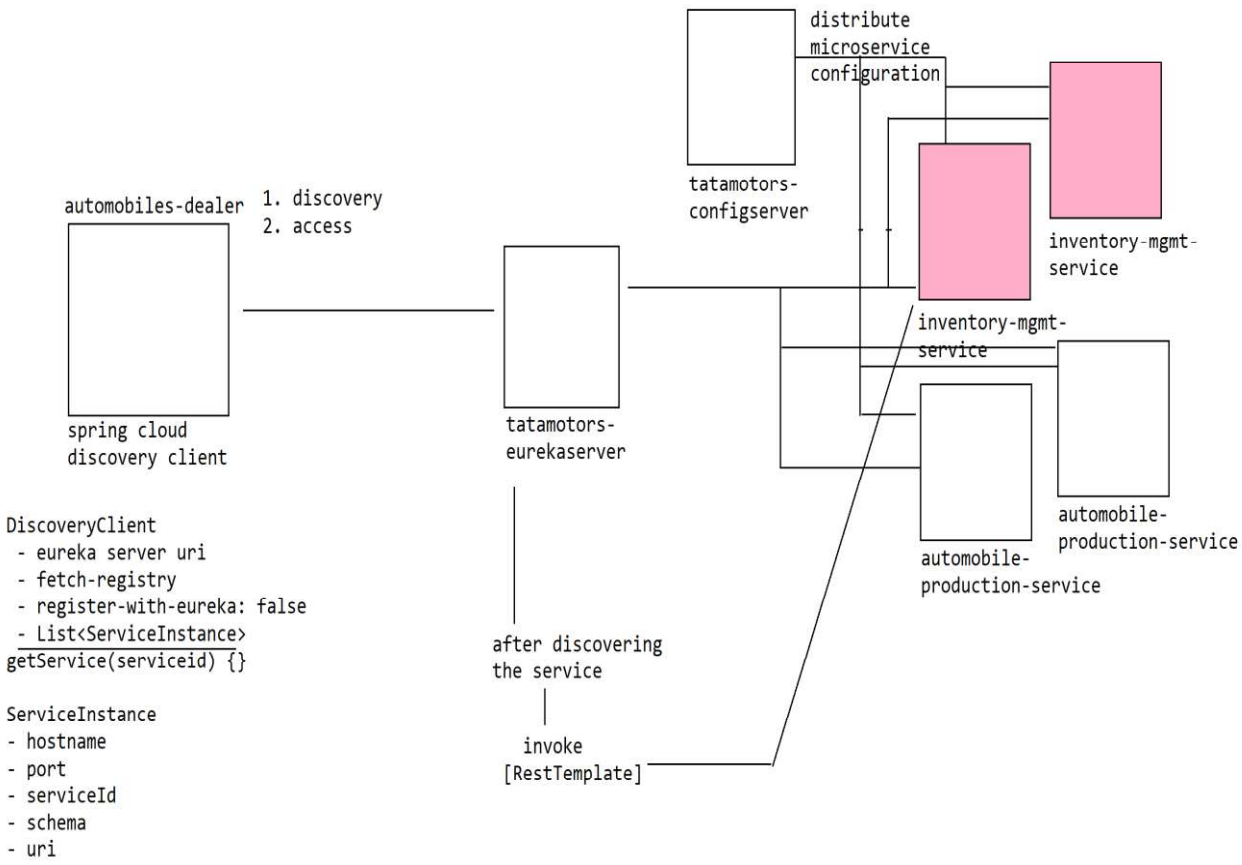
**application.yml**

```
------------------
eureka:
  client:
        register-with-eureka: false
        fetch-registry: true
        service-url:
            defaultZone: http://localhost:8761/eureka
```

```
@Component
class InventoryApiDiscoveryManager {
    @Autowired
    private DiscoveryClient discoveryClient;

    public List<ServiceInstance>
getStockApiServiceInstances(String serviceId) {
        return
discoveryClient.getServiceInstances(serviceId);
    }
}

@Component
class InventoryManager {
    @Autowired
    private InventoryApiDiscoveryManager
inventoryApiDiscoveryManager;
    @Autowired
    private RestTemplate;

    public void showAvailableStock(String stockName) {
        List<ServiceInstance> serviceInstances = null;
        String stockApiServiceUri = null;
```

```
        serviceInstances =
inventoryApiDiscoveryManager.getStockApiServiceInstance
s("INVENTORY-MGMT-SERVICE");
        // 2 nodes


        stockApiServiceUrl =
"http://"+serviceInstances[0].getHost() +
":"+serviceInstances[0].getPort()+"/stock/"+stockName+"
/available";


        List<StockDto> stockdtos =
restTemplate.getForObject(stockApiServiceUrl,
List.class);
        // iterate and print dtos
    }
}


@SpringBootApplication
@EnableDiscoveryClient
class AutomobileDealerApplication {


    public static void main(String[] args) {
        ApplicationContext context =
SpringApplication.run(AutomobileDealerApplication.class
, args);
```

```
        InventoryManager inventoryManager =
context.getBean(InventoryManager.class);

    inventoryManager.showAvailableStock("television");
    }
}
```




In microservices all the instances of the services are registered with eureka discovery server, so in order to access the microservice the client programs has to discover and locate the microservices.

So eureka to help the clients in discovering the microservices has provided HTTP Endpoints.

Now we need to write in the Client program invoking the HttpEndpoints exposed by Eureka Server and the service instances where those are running and use that information in accessing the actual service.

Instead of we writing the code in invoking the eureka server for discovering the microservices, the eureka has provided discovery client library which has predefined set of classes provided with logic for invoking and discovering the Service Instances running on Eureka.

DiscoveryClient is a component provided as part of eureka-discovery-client library into which we need to populate the eureka server uri.

and invoke an method getServices(String serviceId) which returns List<ServiceInstance>

There after getting the instances we need to use the service details in invoking the actual service.

## Spring Cloud Load balancer

---

By using discovery client api we can identify the service instances and we can invoke the microservice as well. but we need to implement at the client-side our own traffic distribution algorithm like round-robin or LRU etc to route the traffic across the instances of the service.

By the above we endup in writing more amount logic and need to handle lot of complexity in accessing the microservices, instead netflix has provided ribbon client-side load balancer.

**Why do we need to have a load-balancer at client-side, why not we can have load balancer on the server-side?**

**#1 solution (infeasible)**

The loadbalancer is not an intelligent component to keep track of the service instances and the nodes on which those are running. it only registers the nodes of the cluster and distributes the traffic across the nodes based on algorithm we applied with no knowledge of of the service running across the nodes.

So having an central load balancer to distribute traffic across the services instances of the cluster is not possible.

**#2 solution (complex/high cost in implementation)**

per service type we can group the nodes on which a service is running and register with an LBR Per ServiceType = 1 LBR

In this way if there 6 microservices running across various nodes of the cluster we need to have 6 LBR instances each per a microservice registered with the nodes of the services where those are running.

In turn to discover these microservice specific LBR we need to register them with eureka server.

**There are several downsides or dis-advantages with this architecture of deploying and running microservices**

1. The developer has to manually configure the LBR each time a microservice has been scaled-out or scaled-in, keeping track of the nodes and their corresponding LBR and manually reconfiguring the LBR with service instances is the most complex job and difficult to manage.

2. If the LBR itself has been crashed then services becomes un-available, to ensure the high-availability of the services we need to run LBR on multiple nodes which is very costly effort and adds maintainability overhead.

3. The client inorder to access the microservice he has to go through multiple discoveries which will increases the latency and degrades the performance in accessing the services

4. through out architecture the LBR has to keep track of the health/availablity of the microservices through heartbearts or healthchecks, and in turn the eureka has to keep track of the LBR availablities through heart beats which increases the overall cpu/resource consumption and network bandwith in keeping track of the instances and their availability

5. the cost of implementing this architechture seems to be quite high and looks to be not an effective solution.

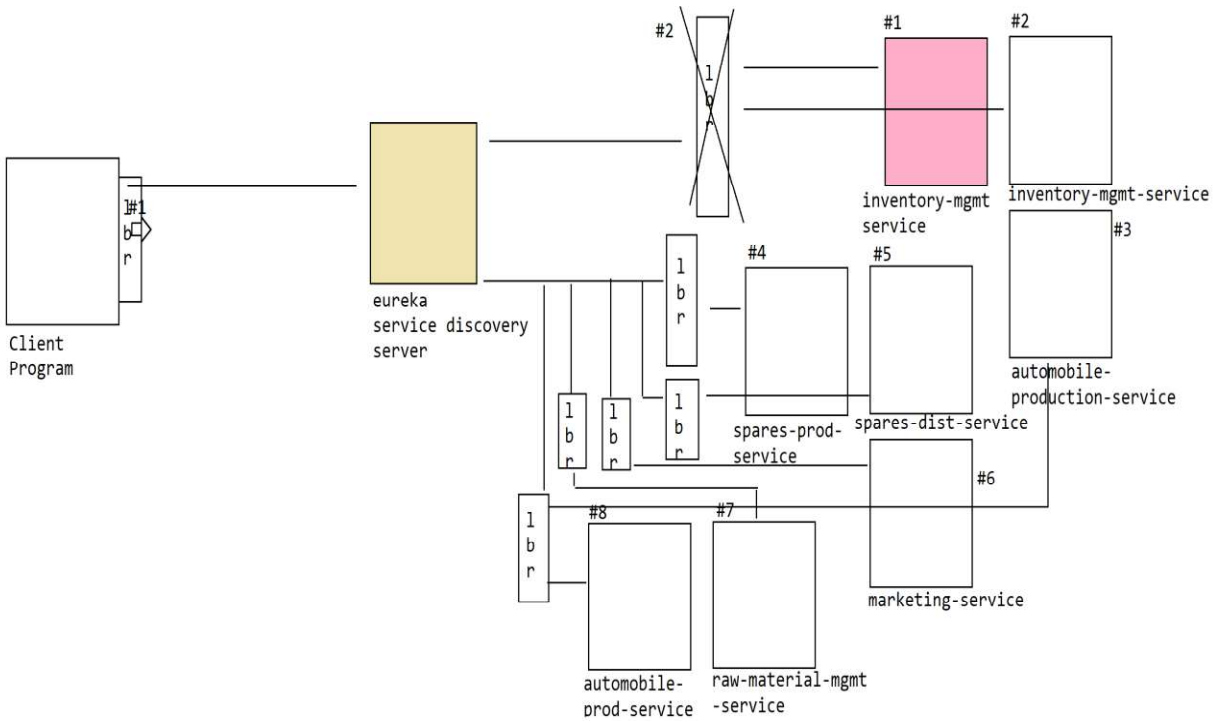
**So implementing an Server-Side load balancer from the above seems to be not an ideal solution in microservices architecture**.

So we need to use client-side loadbalancer in distributing the traffic across the nodes of the microservice.

There are several client-side loadbalancer libraries are available

1. Ribbon from netflix

The Spring has deprecated the Ribbon load balancer and has replaced with Spring Cloud Loadbalancer.




We cannot use server-side loadbalancer as there are many challenges involved in using them, instead each client while accessing the microservice through discovery registry has to implement client-side loadbalancer inorder to distribute the traffic multiple instances of the service.

Netflix has provided ribbon as a client side loadbalancer api/library, in the recent versions of the spring cloud, the ribbon has been marked as deprecated and replaced with Spring Cloud Load balancer.

## How to we need to access the microservice in a loadbalanced manner?

while invoking any of the methods of any of the microservices across the cluster we need to perform the below things.

#1 discover the nodes on which the microservice is running across the cluster

#2 apply round robin algorithm in choosing the node we want to access

#3 construct the Path pointing the node/api which we want to access

#4 invoke the api/microservice endpoint

the above code has to be written not for one method invocation of an api, rather we should apply the same logic accross all the api methods we are accessing on microservice cluster

```
@SpringBootApplication
class AutomobileDealerApplication {

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
```

```
    }


    public static void main(String[] args) {
        ApplicationContext context =
SpringApplication.run(AutomobileDealerApplication.class
, args);
    }
}


@Service
class StoreService {
    @Autowired
    RestTemplate restTemplate;


    public List<StockDto> getStocksAvailable(String
stocknName) {
        // uriComponentBuilder
        //url
        return restTemplate.getForObject(url,
List<StockDto>.class);
    }
}
```

```
class RoundRobinLBRAdvice implements MethodBeforeAdvice
{
    @Autowired
    private DiscoverClient discoverClient;

    public void before(Method method[], Object[] args,
Object proxy) {
        http://INVENTORY-MGMT-
SERVICE/stock/tv/available

        List<ServiceInstance> serviceInstances =
discoveryClient.getInstances("INVENTORY-MGMT-SERVICE");
        // apply round robin algorithm
        constructor original url

        url = new url
    }
}
```

application.yml

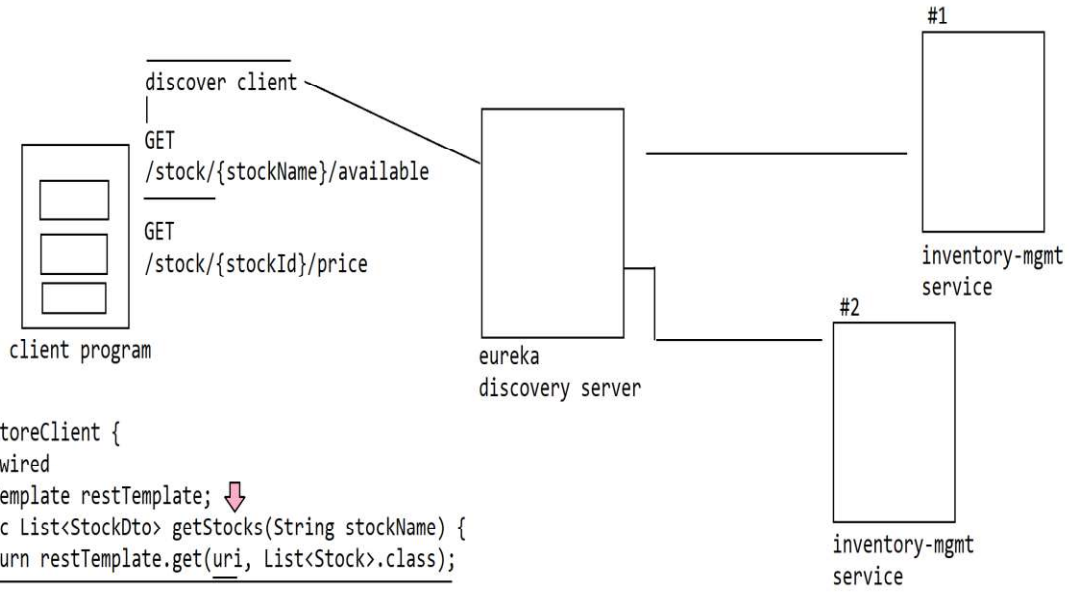
```
eureka:
    client:
        register-with-eureka: false
        fetch-registry: true
```

```
        service-url:
            defaultZone: http://localhost:8761/eureka
```

Instead of it **spring cloud loadbalancer has provided an MethodBeforeAdvice internally which contains the logic for discovering the service instances available on the cluster by talking through eureka server** and apply round-robbin algorithm in choosing the service construct the url with node details.

and the above advice has to be applied on RestTemplate so that every method call on the RestTemplate class will gothrough Advice and discovery takes place and replaces the URL with which the RestTemplate will invoke the service.

To let the Spring Cloud know we want to apply Loadbalancer advice on RestTemplate we need annotate the RestTemplate bean definition with @LoadBalanced so that while creating the object of RestTemplate it applies the advice and creates a proxy and stores the proxied RestTemplate object within the ioc container.




```
class StoreClient {
  @Autowired
  RestTemplate restTemplate;
  public List<StockDto> getStocks(String stockName) {
    return restTemplate.get(uri, List<Stock>.class);
  }
}
```

http://**INVENTORY-MGMT-SERVICE**/stock/tv/available

```
class RoundRobinLoadbalancerAdvice implements
MethodBeforeAdvice {
  public void before(MethodInvocation methodInvocation) {
    DiscoveryClient
  }
}
```

## OpenFeign

---

Spring Cloud OpenFeign is a declarative REST Client library that can be used for accessing microservices without writing any client code aspart of our application.

There are many features of feign client which makes it more powerful

1. it is a declarative client library where we are going to annotate the interface methods with spring mvc annotations or openfeign annotations using which we are going to access the microservice

2. the openfeign client is highly customizable through a set of components like

- **decoders**
- **encoders**
- **loggers**
- **LoadBalancers**
- **FeignBuilders**

```
class StockClient {

    private HttpInvoker httpInvoker;

    public double getStockPrice(String stockName) {
```

```
        Map<String, String> queryParamMap = new
HashMap<>();

        queryParamMap.put("stockName", stockName);

        httpInvoker.invokeGET(url, queryParamMap,
Double.class);

    }


    public List<StockDto> getStocksAvailable(String
stockName, String location) {

        Map<String, String> queryParamMap = new
HashMap<>();

        queryParamMap.put("location", location);

        Map<String, Object> uriVariables = new
HashMap<>();

        uriVariables = new HashMap<>();

        uriVariables.put("stockName", stockName);

        String baseUrl =
"http://localhost:8080/stock/{stockName}/available";

        List<StockDto> stock =
httpInvoker.invokeGET(baseUrl, uriVariables,
queryParams, List<StockDto>.class);

    }

}
```

```
class HttpInvoker {


    public Object invokeGET(String url, Map<String,
String> queryParams, Class<?> responseType) {

        URIComponentBuilder builder =
URIComponentBuilder.fromUriString(url);

        for(String paramName : queryParams.keySet()) {

            builder.queryParam(paramName,
queryParams.get(paramName));

        }

        String url = builder.build().toUriString();

        return restTemplate.getForObject(url,
responseType);

    }


    public Object invokeGET(String url, Map<String,
Object> uriVariables, Map<String, String> queryParams,
Class<T> responseType) {

        URIComponentBuilder builder =
URIComponentBuilder.fromUriString(url);

        if(queryParams != null) {

            for(String paramName :
queryParams.keySet()) {

                builder.queryParam(paramName,
queryParams.get(paramName));

            }
```

```
        }
        builder.uriVariables(uriVariables);
        String url = builder.build().toUriString();
        return restTemplate.getForObject(url,
responseType);
    }

    public Object invokeGET(String url, Map<String,
String> queryParams, Map<String, Object> uriVariables,
Map<String, String> headerParams, Class<T>
responseType) {

    }
    public Object invokePOST(String url, Object entity,
Class<?> responseType) {
        return restTemplate.postForObject(url, entity,
responseType);
    }
}
```

If we are writing the HttpEndpoint invocation logic in every HTTPClient program, the way we invoke a

GET request method

- query params
- path params

- cookie params

- matrix params

- header params

would be same irrespective the HttpEndpoint we are calling only the inputs values will differ and the output response will differ.

It would similar for any type of HttpMethod which is POST, PUT, DELETE as well.

Instead of redundantly writing the Http Invocation logic in all methods of the client class we can write a generic invoker which would take the input data and invoke the HttpEndpoint and returns the response.

```
@Component
class HttpInvoker {
    @Autowired
    @LoadBalanced
    private RestTemplate restTemplate;

    public Object invokeGET(String url, Map<String,
String> queryParams, Class<?> responseType){
        String requestURL = null;
        URIComponentBuilder builder =
URIComponentBuilder.fromUriString(url);
        for(String paramName : queryParams.keySet()) {
```

```
            builder.queryParam(paramName,
queryParams.get(paramName));
        }
        requestURL = builder.build().toUriString();
        return restTemplate.getForObject(requestURL,
responseType);
    }

    public Object invokeGET(String url, Map<String,
Object> uriVariables,

    Class<?> responseType) {
        String requestURL = null;
        URIComponentBuilder builder =
URIComponentBuilder.fromUriString(url).uriVariables(uri
Variables);
        requestURL = builder.build().toUriString();
        return restTemplate.getForObject(requestURL,
responseType);

    }

    public Object invokePOST(String url, Map<String,
Object> uriVariables, Object entity,

Class<?> responseType) {
```

```
        String requestURL = null;

        URIComponentBuilder builder =
URIComponentBuilder.fromUriString(url).uriVariables(uri
Variables);

        requestURL = builder.build().toUriString();

        return restTemplate.postForObject(url, entity,
responseType);

    }

    // we will have with various combinations different
methods for all HTTP Methods to be used in invoking the
HttpEndpoint

}
```

Now within our HTTPClient program instead of we building the logic in invoking the Endpoint we can prepare data and invoke HttpInvoker which takes care of calling Endpoint as below.

```
@Component

class StoreClient {

    @Autowired

    private HttpInvoker httpInvoker;


    // here productName should be sent as uriVariable
in GET request

    public double getProductPrice(String productName) {
```

```
        String baseURI = null;
        Map<String, Object> uriVariables = null;

        baseURI =
"http://localhost:8081/inventorymgmtservice/stock/{prod
uctName}/price";
        uriVariables = new HashMap<>();
        uriVariables.put("productName", productName);
        return httpInvoker.invokeGET(baseURI,
uriVariables, Double.class);
    }

    // invoke Endpoint with productName as uriVariable
and location as queryParameter
    public List<StockDto> getAvailableStocks(String
productName, String location) {
        String baseURI = null;
        Map<String, Object> uriVariables = null;
        Map<String, String> queryParams = null;

        baseURI =
"http://localhost:8081/inventorymgmtservice/stock/{prod
uctName}/available";
        uriVariables = new HashMap<>();
        uriVariables.put("productName", productName);
```

```
        queryParams = new HashMap<>();
        queryParams.put("location", location);
        return httpInvoker.invokeGET(baseURI,
uriVariables, queryParams, List<StockDto>.class);
    }
}
```

In the above client program we are ending up in repeatedly writing the logic in cooking up the data and invoking the HttpInvoker which is seems to redundant across all the Client programs.

Let us try to resolve the problem:

```
@Component
interface StockClient {

    public Double getProductPrice(String productName);
}

@Aspect
@Component
class HttpInvokerAdvice implements MethodInterceptor {

    @Autowired
```

```
    private HttpInvoker httpInvoker;


    @AroundAdvice("within(com.automobiles.dealer.client
.*)")
    public Object invoke(MethodInvocation
methodInvocation) {
        Method method = null;
        Object[] args = null;
        Class<?> targetClass = null;
        Map<String, Object> uriVariables = null;
        Map<String, String> queryParams = null;


        targetClass = methodInvocation.getTarget();
        method = methodInvocation.getMethod();
        args = methodInvocation.getArguments();


        // read endpoint-config.xml
        // identify httpClient based on className
(StockClient)
        // within the StockClient httpClient search for
httpInvocation element based on methodName
        requestURL = httpClient.getBaseURI() +
httpInvocation.subResourceURI();


        if(httpInvocation.hasURIVariables) {
```

```
            uriVariables = new HashMap();

    uriVariables.put(httClient.getUriVariables()[0].nam
e, args[0]);
        }
        return httpInvoker.invokeGET(requestURL,
uriVariables, httpClient.getResponseType());
    }
}
```

**endpoint-config.xml**

```
----------------------
<httpClients>
    <httpClient className="StockClient"
baseURI="http://localhost:8081/inventorymgmtservice/sto
ck">
        <httpInvocation name="getProductPrice"
subResourceURI="/{productName}/price"
responseType="Double">
            <parameter name="productName"
type="uriVariable"/>
        </httpInvocation>
    </httpClient>
</httpClients>
```

Instead of we writing the HttpInvoker and HttpInvokerAdvice which will automate the process of invoking the HttpEndpoint based on metadata declaration, the OpenFeign api has provided the above components, which we can make use of invoking the REST/HTTP Endpoints.

```
@FeignClient(url="http://localhost:8081/inventorymgmtservice/stock/")
interface StoreService {
    @GetMapping(value="/{productName}/price", produces = {"text/plain"})
    public double getProductPrice(@PathVariable("productName") String productName);
}

@SpringBootApplication
@EnableFeignClients
class FeignClientApplication {
    public static void main(String[] args) {
        ApplicationContext context = SpringApplication.run(FeignClientApplication.class, args);
        StoreService storeService = context.getBean(StoreService.class);
```

```
        double price =
storeService.getProductPrice("television");
    }
}
```

## What is circuit breaker?

---

Circuit breaker is one of the integration-tier design pattern which is used for protecting the client applications while accessing the remote services.

The circuit breaker is not an spring framework feature or is not only applied for microservices, it an independent design pattern of its own and can be applied anywhere at the client-side when we are accessing remote services in case of

- soap services
- http endpoints
- ejb / rmi invocations

ofcourse it is not limited by language, it is adopted and implemented by many programming languages as it is a design pattern

When the client application is trying to access a remote service for consuming/reusing the functionality of the remote service/application, there could be a chance where due to several reason the remote service might become un-responsive.

It could be due to heavy load on the remote service or it might be accessing a remote system which seems to slow or due to drained system resources etc.

At this time if our client application is allowed to access the remote service, it creates several problems at the client side as described below

1. The client application would be sending the request to the un-responsive remote service blocking the threads and consuming system resources which will waste the system resources of the client application.

2. due to the longer wait time, the client application might eventually run into timeout even after keeping hold of the system resources for long waiting for the response from the service.

3. during the time of waiting for response from un-responsive remote service, if the client application is hitting up with more numbers of requests for accessing the un-responsive remote service, the more number of threads will be kept under blocked state and increases the consumption system resources which will eventually crashes the client application and makes the complete completely loss of service

4. The post effect of accessing an un-responsive remote service will not only affect the client application it shows an cascading affect on the clients who are accessing the client application as well.

5. at this point where the remote service became slow/non-responsive if more number of clients are accessing the remote service it will leads to crash as well.

***How to resolve the above problem?***

If the client application can identify which remote services are becoming unresponsive and running into

exception (timeout) while accessing the service and can block the requests to the remote service for certain interval time rejecting the requests at the client side, so that we can save the client application from crashing and allowing rest of the features of the client to be available for accessing. This can be achieved by employing circuit breaker design pattern.

**How does circuit breaker design pattern works?**

*#1 invoke the remote service in a controlled environment where we can monitor request/response.*

```
interface Invocation {
    public Object invoke();
}

class GetStockPrice implements Invocation {
    @Autowired
    private RestTemplate restTemplate;

    public Object invoke() {
        return restTemplate.getForObject(url,
Double.class);
    }
}
```

```
class CircuitBreaker {

    int failureThreadshold;

    int openStateInMilliseconds; // 30 seconds


    int failureCount=0;

    long openStateInterval=-1;

    boolean openState = false;


    public Object invoke(Invocation invocation) {

        try {

            if(openState == true) {

                long ctMilliSeconds =
Calendar.getTimeInMilliSeconds(); //1010

                if((ctMilliSeconds -
openStateInterval) < openStateInMilliseconds) {

                    throw new
CircuitOpenedException("remote service is non-
responsive, so circuit got opened");

                }else {

                    openState = false;

                    failureCount = 0;

                    openStateInterval = -1;

                }
```

```
            }


            if(failureCount == failureThreadShold) {
                openStateInterval =
Calendar.getTimeInMilliseconds(); //1000
                openState = true;
                throw new
CircuitOpenedException("remote service is non-
responsive, so circuit got opened");
            }else {
                Object response = invocation.invoke();
                failureCount = 0;
                openState = false;
                return response;
            }
        }catch(Throwable t) {
            failureCount++;
            throw t;
        }
    }
}


<bean id="inventoryManagementCB"
class="CircuitBreaker">
```

```
    <property name="failureThreadshold" value="5"/>
    <property name="openStateInMilliseconds"
value="40000"/>
</bean>

<bean id="sparePartsDistributionCB"
class="CircuitBreaker"/>

@Component
class StoreClient {
    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    @Qualifier("inventoryManagementCB")
    private CircuitBreaker circuitBreaker;

    public double getStockPrice(final String stockName)
{
        return circuitBreaker.invoke(new Invocation() {
            public Object invoke() {
                String baseURI =
"http://localhost:8081/INVENTORY-MGMT-
SERVICE/stock/{stockName}/price";
                Map uriVariables = new HashMap<>();
```

```
                uriVairables.put("stockName",
stockName);


                String url =
URIComponentBuilder.fromUriString(baseURI).uriVariables
(uriVariables).build().toUriString();
                return restTemplate.getForObject(url,
Double.class);
            }
        });
    }
}



class Test {
    public static void main(String[] args) {
        ApplicationContext context =
SpringApplication.run(Test.class, args);
        StockClient sc =
context.getBean(StockClient.class);


        //20 times
        sc.getStockPrice("television");
    }
}
```

Instead of we implementing the above circuit breaker design pattern there are lot of open source circuit breaker third-party implements are available.

1. netflix hystrix
2. resilience4j
3. spring retry
4. sentinel

Instead of working with individual circuit breaker implementations directly spring cloud circuit breaker is a wrapper that allows you to work with any of the circuit breaker implementations without changing the code.

```
// 100%
@SpringBootApplication
class CircuitBreakerApplication {


    @Bean("inventoryManagementCB")
    public HystrixCircuitBreaker circuitBreaker() {
        HystrixCircuitBreaker circuitBreaker = new
HystrixCircuitBreaker();
        // populate threshold limits
        return circuitBreaker;
```

```
    }


    public static void main(String[] args) {
        ApplicationContext context =
SpringApplication.run(CircuitBreakerApplication.class,
args);


    }
}


@Component
class StoreClient {
    @Autowired
    private RestTemplate restTemplate;


    @Autowired
    private HystrixCircuitBreaker circuitBreaker;


    public double getProductPrice(String productName) {
        return circuitBreaker.run(new
HystrixInvocation() {
            public Object invoke() {
                // write the logic for calling remote
service
```

```
            }
        });
    }
}
```

In the above classes we have used Hystrix provided api classes in implementing CircuitBreaker functionality for our application. Let us say we want to move away from Hystrix to Resilience4j then now we need to rewrite all over the code in our application classes whereever we are using Hystrix to Resilience4j which is a huge effort.

Instead of working with individual libraries in implementing CircuitBreaker functionality, spring cloud has provided SpringCloudCircuitBreaker wrapper using which we can work with any of the Circuit Breaker third-party implementations.

```
interface CircuitBreaker {
    Object run(Runnable, T fallback) {}
}

class HystrixCircuitBreaker implements CircuitBreaker {
    com.netflix.hystrix.CircuitBreaker breaker = null;

    Object run(Runnable T, Function<T> fallback) {
```

```
        breaker.run(T);
    }
}

class Resilience4jCircuitBreaker implements
CircuitBreaker {
    com.resilience4j.CircuitBreaker breaker;
}

HystrixCircuitBreaker
Resilience4JCircuitBreaker
SentinelCircuitBreaker



@Component
class StoreClient {
    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private CircuitBreaker circuitBreaker;

    public double getProductPrice(String productName) {
```

```
        return circuitBreaker.run(new
HystrixInvocation() {
            public Object invoke() {
                // write the logic for calling remote
service
            }
        });
    }
}
```

Spring Cloud Circuit Breaker has provided their own interface with each vendor specific implementation class wrapping the vendor api. Now depends the vendor api we want to use we need to instantiate and configure Spring provided implementation for Vendor api.

To instantiate the Spring Circuit Breaker implementation class, Spring cloud has provided as Factory class

```
interface CircuitBreaker {
    T run(Supplier T, Function<Throwable> fallback);
}
```

For this interface Spring Cloud has provided one implementation per one CircuitBreaker vendor library Spring supports.

```
class HystrixCircuitBreaker implements CircuitBreaker {
    com.netflix.hystrix.CircuitBreaker cb;

    public T run(Supplier<T> s, Function<Throwable> t)
{
        return cb.run(s);
    }
}
```

The above class is a wrapper on top of HystrixCircuitBreaker library, which invokes hystrix api classes

**class Resilience4JCircuitBreaker implements CircuitBreaker {}**

The above class is a wrapper on top of Resilience4J library

```
abstract class CircuitBreakerFactory {
    public CircuitBreaker create(String id); // creates an object of CircuitBreaker and places in ioc container with the given id
}
```

for the above factory class spring cloud has provided 1 implementation for one vendor

```
class HystrixCircuitBreakerFactory extends CircuitBreakerFactory {

    public CircuitBreaker create(String id) {

        the create method will instantiate the vendor circuitbreaker object and wraps into Spring provided vendor CircuitBreaker implementation and returns to us.

    }

}
```

```
class Resilience4JCircuitBreakerFactory extends CircuitBreakerFactory {

    public CircuitBreaker create(String id) {

        the create method instantiates Resilience4J vendor api and wraps into Spring provided vendor implementation class Resilience4JCircuitBreaker object

    }

}
```

```
@Component

class StoreClient {

    @Autowired
```

```
    private RestTemplate restTemplate;


    public double getProductPrice(final String
productName) {
        CircuitBreaker cb = null;
        CircuitBreakerFactory cbf = new
Resilience4JCircuitBreakerFactory();


        cb = cbf.create("ims"); // vendor
implementation of cb object
        return cb.run(()->{


            String baseURI =
"http://localhost:8081/inventory-mgmt-
service/stock/{productName}/price";
            // uriVariables created
            String url =
URIComponentBuilder.fromURIString(baseURI).uriVariables
(uriVariables).build().toUriString();
            return restTemplate.getForObject(url,
Double.class);
        });
    }
}
```

In the above code to switch from one vendor to another vendor again we need to modify the AbstractFactory implementation class. Instead of we creating the object of vendor factory implemntation, The Spring Cloud autoconfigurations takes care of creating the vendor factory object and places in ioc container as a bean definition.

for eg.. if we are using spring-cloud-resilience4j-circuit-breaker starter library, the Resilience4JCircuitBreakerFactoryAutoConfiguration configures Resilience4JCircuitBreakerFactory as a bean definition.

```
@Component
class StoreClient {
    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private CircuitBreakerFactory cbf;

    public double getProductPrice(final String
productName) {
        CircuitBreaker cb = null;
```

```
        cb = cbf.create("ims"); // vendor
implementation of cb object

        return cb.run(()->{

            String baseURI =
"http://localhost:8081/inventory-mgmt-
service/stock/{productName}/price";

            // uriVariables created

            String url =
URIComponentBuilder.fromURIString(baseURI).uriVariables
(uriVariables).build().toUriString();

            return restTemplate.getForObject(url,
Double.class);

        });

    }

}
```

The CircuitBreakerFactory instantiates the CircuitBreaker object with default configurations.

## Spring Cloud Circuit Breaker

is an wrapper that is provided on top of various different third-party implementations of Circuit breaker libraries. by using Spring Cloud Circuit breaker we can switch from any of the third-party libraries of the Circuit Breaker like

- **Hystrix**
- **Resilience4J**
- **Spring Retry**
- **Sentinel**

Spring cloud circuit breaker has provided an CircuitBreaker interface with one implementation per one third-party library, where the implementation wraps the Third-party library api inside it.

```
interface CircuitBreaker {

        Object run(Supplier S, Function<Throwable> f);

}
```

to the above interface the spring cloud circuit breaker has provided 1 implementation class per one third-party api wrapping third-party library implementation CircuitBreaker

**class HystrixCircuitBreakerImpl implements CircuitBreaker {**

```
    com.netflix.hystrix.HystrixCircuitBreakerImpl circuitBreaker;
```

```
}
```

To instantiate the vendor specific CircuitBreaker implementation the Spring has provided CircuitBreakerFactory which is an Abstract Factory.

```
abstract class CircuitBreakerFactory {
    CircuitBreaker create(String id);
}
```

For each vendor implementation they provided an implementation of the Abstract Factory which takes care of not only instantiating the Spring implementation even the vendor object wrapped inside the Spring implementation also will be instantiated by the implementation factory.

```
class HystricCircuitBreakerFactoryImpl extends CircuitBreakerFactory {
    CircuitBreaker create(String id) {
        // it has the logic for instantiating Spring provided HystrixCircuitBreaker object and even wrapped object of com.netflix.hystrix.HystrixCircuitBreakerImpl as well
    }
}
```

by default the Factory takes care of instantiating the CircuitBreaker object with default configuration.

**How to instantiate the the CircuitBreaker object?**

We need to use the Vendor implementation Factory object to instantiate the CircuitBreaker object of that vendor.

```
CircuitBreakerFactory cbf  = new
HystricCircuitBreakerFactoryImpl();

CircuitBreaker cb = cbf.create("slow");
```

even though we need to vendor implementation factory to instantiate the CircuitBreaker object we dont need to create the object of the factory, because when we include the relevant starter of the vendor for e.g.. spring-cloud-starter-neflix-hystrix the autoconfigurations corresponding to the starter takes care of instantiating the object of the implementation factory and places as bean definition within ioc container.

So in our Client Class we simply autowire CircuitBreakerFactory using which we can create the object of CircuitBreaker implementation class

```
class StoreClient {

    @Autowired

    CircuitBreakerFactory circuitBreakerFactory;
```

```
    public double getStockPrice(String stockName) {
        CircuitBreaker cb =
circuitBreakerFactory.create("default");
    }
}
```

**if we want to switch from one vendor implementation to another vendor we just only need to change the starter dependency, we dont have to even change one line of code in our application.**

We need to create multiple CircuitBreaker objects to wrap microservices calls within them based on the nature of the microservice we are invoking.

For eg.. if we have group/multiple microservices which has similar type of traffic and processing capacity, then we can create on CircuitBreaker object and can wrap the service calls to them through the same object.

spares-production-service

spares-distribution-service - has similar nature of traffic patterns - 1 circuit breaker

by default CircuitBreakerFactory will instantiate the object of CircuitBreaker with default threadshold

values, but from the above we can understand we need CircuitBreaker to be instantiated with different values based on the microservices are invoking.

we need to tell the CircuitBreakerFactory, which CircuitBreaker should be instantiated with which configuration so that it can instantite the object by populating the values.

Now we need to provide CircuitBreakerConfiguration to CircuitBreakerFactory asking it to use the Configuration to instantiate the corresponding CircuitBreaker.

So for this Spring Cloud has provided CircuitBreakerConfiguration class, but different CircuitBreaker vendors has different configurations So spring cloud has provided one CircuitBreakerConfiguration class per one vendor implementation.

Instead of we instantiating the vendor CircutiBreakerConfiguration class Spring cloud has provided an Builder class per vendor which will faclitate in easily instantating the Configuration object.

The CircuitBreakerFactory takes care of instantiating the object of CircuitBreaker with default configuration values, but while accessing the microservices the default configurations values may not applicable.

Few microservices could be "slow" running services which requires higher timeout interval and few microservices by nature would fail frequently which requires higher failure threashold before the circuit goes to open state, so **how to customize the CircuitBreaker object specific to the service we are accessing?**

Since the CircuitBreakerFactory is creating the CircuitBreaker object, to customize the CircuitBreaker creation we need to provide configuration to CircuitBreakerFactory asking to customize the configuration with which it instantiates the object of CircuitBreaker

Across the third-party vendors the CircuitBreaker configuration will differ, so spring cloud has provided CircuitBreaker configuration class each per one vendor.

For eg..

**Resilience4J third-party library**, Spring cloud has provided Resilience4JCircuitBreakerConfiguration, Hystrix third-party library, it has provided

HystrixCircuitBreakerConfiguration since the configuration options differs from vendor to vendor.

Now configure the parameters and populate into CircuitBreakerConfiguation of the vendor. To easily instantiate and populate the values into respective vendor CircuitBreakerConfiguration SpringCloud has provided builder classes which are "ConfigBuilder"

for eg.. when we are working with Resilience4J library to create Resilience4CircuitBreakerConfiguration, we need to use Resilience4JConfigBuilder

```
Resilience4JConfigBuilder builder = new
Resilience4JConfigBuilder("slow");

// populate the configuration into builder

Resilience4JConfiguration configuration =
builder.build();
```

```
CircuitBreaker breaker =
circuitBreakerFactory.create("slow");
```

```
class Resilience4CircuitBreakerConfiguration {

    int id;

    CircuitBreakerConfiguration
circuitBreakerConfiguration;
```

```
    TimeLimiterConfig timeLimiterConfig;


}


class Resilience4JConfigBuilder {

    int id;

    CircuitBreakerConfiguration
circuitBreakerConfiguration;

    TimeLimiterConfig timeLimiterConfig;


    public Resilience4JConfigBuilder(String id) {}

    public Resilience4JConfigBuilder
circuitBreakerConfiguration(CircuitBreakerConfiguration
configuration) {}

    public Resilience4JConfigBuilder
timeLimiterConfig(TimeLimiterConfig config) {}

    Resilience4CircuitBreakerConfiguration build();

}
```

By using the above ConfigBuilder of each vendor, create the object of Configuration of that vendor and pass the Configuration object to CircuitBreakerFactory

We are not configuring the CircuitBreakerFactory as a bean definition to populate configuration objects into it, rather it is autoconfigured as a bean definition by AutoConfiguration class, then **how to customize the bean definition that is instantiated by the AutoConfiguration class?**

```
interface Customizer<T> {
    public void customize(T t);
}

@Component
class CircuitBreakerFactoryCustomizer implements
    Customizer<Resilience4JCircuitBreakerFactory> {
    public void
customize(Resilience4JCircuitBreakerFactory cbf) {
        create Resilience4JConfiguration and add to cbf

    }
}
```

There are muliple vendor libraries are available for CircuitBreaker pattern like

**Resilience4J**

**Hystrix etc**

out of the above Resilience4J has got too many number of features is very popular in the market. it supports below features

**1. CircuitBreakerConfiguration** = we can configure threadshold limits in managing the CircuitBreaker.

The circuitbreaker can be in one of the three states

1. **OPEN** = requests are no more allowed to the remote service
2. **CLOSED** = allows the requests to the remote service
3. **HALF-OPEN** = the number of requests that can be allowed to the remote service before the circuit is fully closed.

**2. RateLimitter =** RateLimiter configuration helps us in limiting the number of requests to an Remote Service

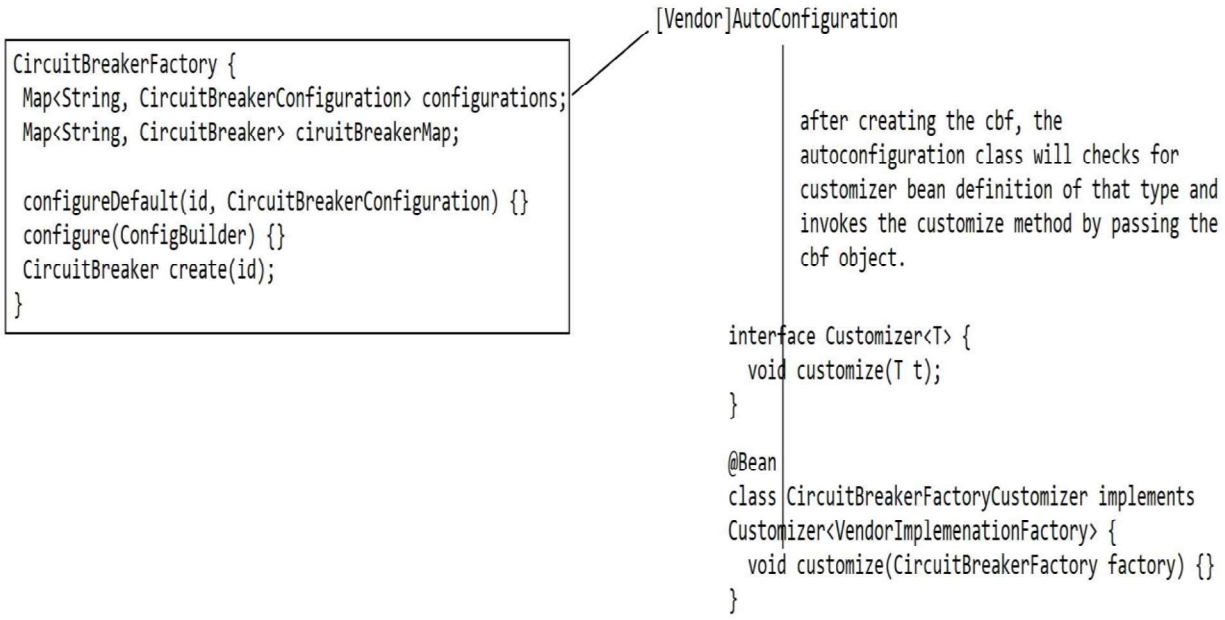
**3. TimeLimitter =** how long the client has to wait for a response from the remote service can be configured through TimeLimitter

**4. BuilkHeadProvider =** Number of concurrent requests allowed to the remote service can be managed through BulkHeadProvider

**5. Retry =** In case of failure in accessing an RemoteService, we can specify retry automatically.

The Resilience4J project was built from the motivation of Hystrix and we can consider Resilience4J as a next version of Hystrix

Now how to work with Spring Cloud Circuit Breaker using Resilience4J Third-party library in applying customizations



## How to customize the CircuitBreaker configuration?

We need to write Customizer interface implementation on CircuitBreakerFactory, so that the autoconfiguration class will invoke the Customizer bean definition by passing CirucitBreakerFactory to us, into which we can populate the CircuitBreakerConfiguration.

```
@Component
class CircuitBreakerFactoryCustomizer implements
Customizer<CircuitBreakerFactory> {
    public void customize(CircuitBreakerFactory cbf) {


    }
}
```

in the above method we need to write the logic for creating the object of CirucitBreakerConfiguration through the help of CircuitBreakerConfigBuilder and apply on CircuitBreakerFactory

We need to apply the customizations based on the Third-Party Library we are working with. Resilience4J is a very popular third-party library available in Java for implementation CiruitBreaker pattern for our application, it has lot of features

1. CircuitBreaker
2. TimeLimiter
3. RateLimiter

4. BulkHead

5. Retry

```
@RestController
class StoreService {
    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private CircuitBreakerFactory
circuitBreakerFactory;

    @GetMapping(value="/store/{stockName}")
    public List<StockDto>
getStocks(@PathVariable("stockName") String stockName)
{
        CircuitBreaker cb = null;

        cb = circuitBreakerFactory.create("slow");
        return cb.run(()->{
            return restTemplate.getForObject(url,
List<StockDto>.class);
        });
    }
```

```
}


@SpringBootApplication
class CircuitBreakerApplication {


    @Bean
    public
Customizer<Resilience4JCircuitBreakerFactory>
defaultCircuitBreaker() {
        return new
Customizer<Resilience4JCircuitBreakerFactory>() {
            public void
customize(Resilience4JCircuitBreakerFactory factory) {
                factory.configureDefault(new
Function<String,
Resilience4JConfigBuilder.Resilience4JCircuitBreakerCon
figuration>() {
                    public
Resilience4JCircuitBreakerConfiguration apply(String
id) {
                        // populate
CircuitBreakerConfig and TimeLimiterConfiguration
                        Resilience4JConfigBuilder
builder = new Resilience4JConfigBuilder(id);


                        CircuitBreakerConfig cbConfig
=
```

```
CircuitBreakerConfig.custom().failureThreshold(5).waitD
urationInOpenState(Duration.ofMillSeconds(300)).sliding
WindowSize(2).build();


                        TimeLimiterConfig tlConfig =
TimeLimiterConfig.custom().timeOutDuration(Duration.ofS
econds(1)).build();


    Resilience4JCircuitBreakerConfiguration config =
builder.timeLimitConfig(tlConfig).circuitBreakerConfig(
cbConfig).build();
                            return config;
                        }
                    });
                }
            };


            @Bean
            public
Customizer<Resilience4JCircuitBreakerFactory>
slowConfig() {
                return new
Customizer<Resilience4JCircuitBreakerFactory>() {
                    public void
customize(Resilience4JCircuitBreakerFactory factory) {
                        factory.configure(new
Consumer<Resilience4JConfigBuilder>() {
```

```
                            public void
accept(Resilience4JConfigBuilder builder) {

                                CircuitBreakerConfig
cbConfig =
CircuitBreakerConfig.custom().failureThreshold(5).waitD
urationInOpenState(Duration.ofMillSeconds(300)).sliding
WindowSize(2).build();



                                TimeLimiterConfig
tlConfig =
TimeLimiterConfig.custom().timeOutDuration(Duration.ofS
econds(1)).build();


    Resilience4JCircuitBreakerConfiguration config =
builder.timeLimitConfig(tlConfig).circuitBreakerConfig(
cbConfig).build();

                            }

                        }, "slow");

                    }

                };

            }

    }


    public static void main(String[] args) {


    SpringApplication.run(CircuitBreakerApplication.cla
ss, args);
```

```
    }


}


@SpringBootApplication
class CircuitBreakerApplication {


    @Bean
    public
Customizer<Resilience4JCircuitBreakerFactory>
slowCircuitBreakerConfig() {
        return new
Customizer<Resilience4JCircuitBreakerFactory>() {
            public void
customize(Resilience4JCircuitBreakerFactory
tocustomize) {


                tocustomize.configure(new
Consumer<Resilience4JConfigBuilder>(){
                    public void
accept(Resilience4JConfigBuilder builder) {


    builder.circuitBreakerConfig(CircuitBreakerConfig.c
ustom().build()).timeLimiterConfig(TimeLimiterConfig.cu
stom().build());
                    }
```

```
                }, "slow");
        };
    }

    public static void main(String args[]) {

    SpringApplication.run(CircuitBreakerApplication.cla
ss, args);
    }
}

class Resilience4JCircuitBreakerFactory {

    public
configure(Consumer<Resilience4JConfigBuilder> consumer,
String... ids) {
        Resilience4JConfigBuilder builder = new
Resilience4JConfigBuilder(id);
        consumer.accept(builder);
        Resilience4JCircuitBreakerConfiguration
configuration = builder.build();

    }
}
```

```
interface Consumer<T> {
    void accept(T t) {
        // operation
    }
}
```

we want to create
Resilience4JCircuitBreakerConfiguration

Resilience4JConfigBuilder
- CircuitBreakerConfig
- TimeLimiterConfig
- build();

```
@Component
class StoreClient {
    @Autowired
    private RestTemplate restTemplate;
    @Autowired
```

```
    private CircuitBreakerFactory cbf;


    public double getStockPrice(String stockName) {
        CircuitBreaker cb = null;


        cb = cbf.create("default");
        return cb.run(()->{
            String baseURI = "http://INVENTORY-MGMT-
SERVICE/stock/{stockName}/price";
            Map<String, Object> uriVariables = new
HashMap();
            uriVariables.put("stockName", stockName);
            String resourceURL =
UriComponentBuilder.fromUriString(baseURI).uriVariables
(uriVariables).build().toUriString();
            return
restTemplate.getForObject(resourceURL, Double.class);
        });
    }
}



class A {
    public long add(List<Integer> l) {
        long sum = 0;
```

```
        for(Integer n: l) {
            sum += n;
        }
        return sum;
    }
}
```

Test.java

---

```
A a = new A();
List<Integer> l = new ArrayList<>();
l.add(10);
l.add(20);
l.add(30);
int sum = a.add(l);

class A {
    long add(Consumer<List<Integer>> consumer) {
        List<Integer> l = new ArrayList();
        Long sum = 0;
        consumer.accept(l);
        for(Integer n : l) {
            sum += n;
        }
```

```
            return sum;
        }
    }

    A a = new A();
    long sum = a.add(new Consumer<List<Integer>() {
        void accept(List<Integer> l) {
            l.add(10);
            l.add(20);
            l.add(30);
        }
    });
```

## Spring Cloud Gateway

---

There are few common services/requirements we want to enforce across the microservices of our application like

1. **security**
2. **routing**
3. **transformation**
4. **data aggregation**

we can implement these common requirements in each individual service, but we endup duplicating the code across all the microservices and takes lot of time in enforcing such requirements.

To rescue us from above problems cloud gateway has been introduced. cloudgateway acts as an single entry point or gateway of receiving microservice requests from the client.

### 1. Security

So that we can apply security at the gateway level, which indirectly secure all endpoints of the microservices as it acts as an entry point

### 2. routing

We can configure routes with predicates/conditions based on the request received we can route to a different versions of the microservices

### 3. transformations

we can modify both request body and response body during the time of request and response by attaching filters to the gateway asking him to apply

### 4. aggregation

we can write the logic at gateway in calling multiple microservices and aggregate the data and return to the client application

There are lot of api gateway libraries are available in market

**1. zuul**

**2. apigee**

in addition to the third-party libraries **spring has provided spring-cloud-gateway module to implementation api gateway functionality which works based on reactive streams**

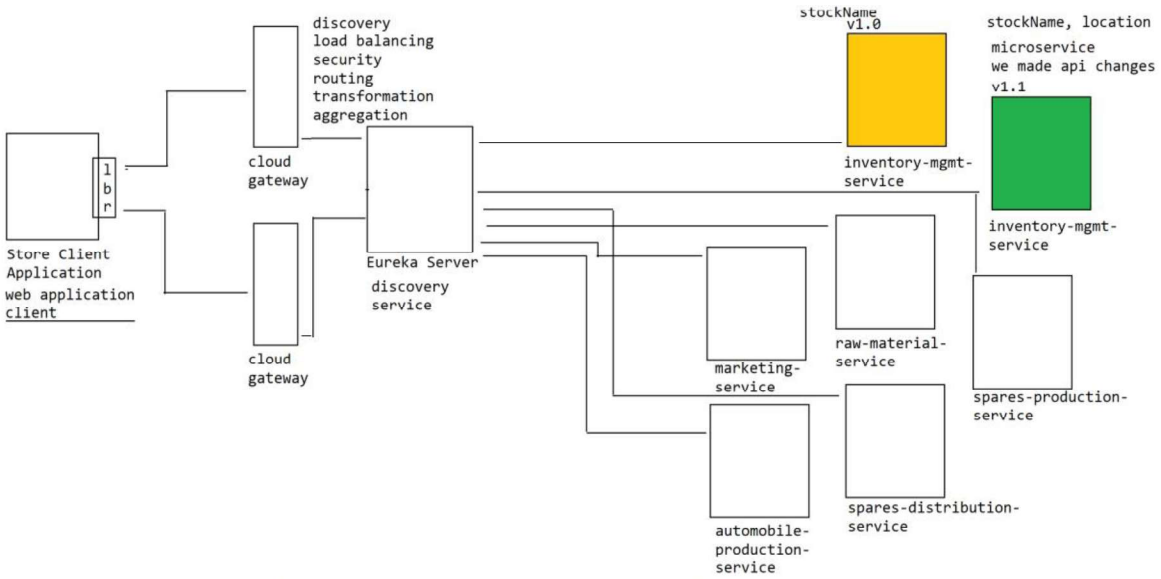
The spring-cloud-gateway has been implemented on reactive stack so we achieve non-blocking i/o through which we can achive maximum throughput.

From gateway library provider to provider the features will differ few vendors supports transformations, few other vendors supports aggregation etc.

Spring Cloud Gateway supports

1. **routing based on predicates**
2. **transformations through filters**
3. **any how using spring security we can enfore security at api gateway level**

but spring cloud **gateway doesnt support aggregations**.




1. want to secure the microservices and only want to allow authenticated and authorized users to access?

2. based on the type of the client accessing the microservices, we want to route to different copies/versions of the microservice?
How to implementing routing?

3. we want to apply transformations of the request/response to make it compatible with the client

4. we want to aggregate the data and return the combined data of the multiple microservices.

## Spring Cloud Gateway

---

There are several common requirements that has to be applied across multiple microservices of our application like

- **security**
- **transformation**
- **routing**
- **data aggregation**
- **logging**
- **auditing**
- **monitoring**

etc

These are all the common requirements that has to be applied for each microservice of our application, having them implemented individually on each microservice will make the code duplicated across all of the services and becomes difficult to manage.

Instead of it we can have an common gateway / entrypoint in receiving the requests into our application, discover and load balance across the multiple instances of the microservice through api gateway.

There are lot of third-party vendor libraries are available for working with api gateway like

1. zuul
2. apigee
3. kong

in addition to the above spring has provided cloud gateway which is implemented on reactive api, which will provide more throughput and performance in routing the requests.

Different api gateway vendors offers different features, the spring cloud gateway provides 3 features

1. routing
2. transformation
3. security through spring security module.

**How does the architecture of spring cloud gateway?**

There are 3 major components are there in api gateway

**1. routing =** routing is used for mapping an incoming request to the appropriate microservice

**2. predicates =** conditions to be evaluated based on different parameters of the http request to map the request to the corresponding microservice

**3. filters =** used for applying pre/post process of the request/response while invoking the microservice

There are 2 ways of working with api gateway

**1. through configuration approach**

**2. programmatic api**

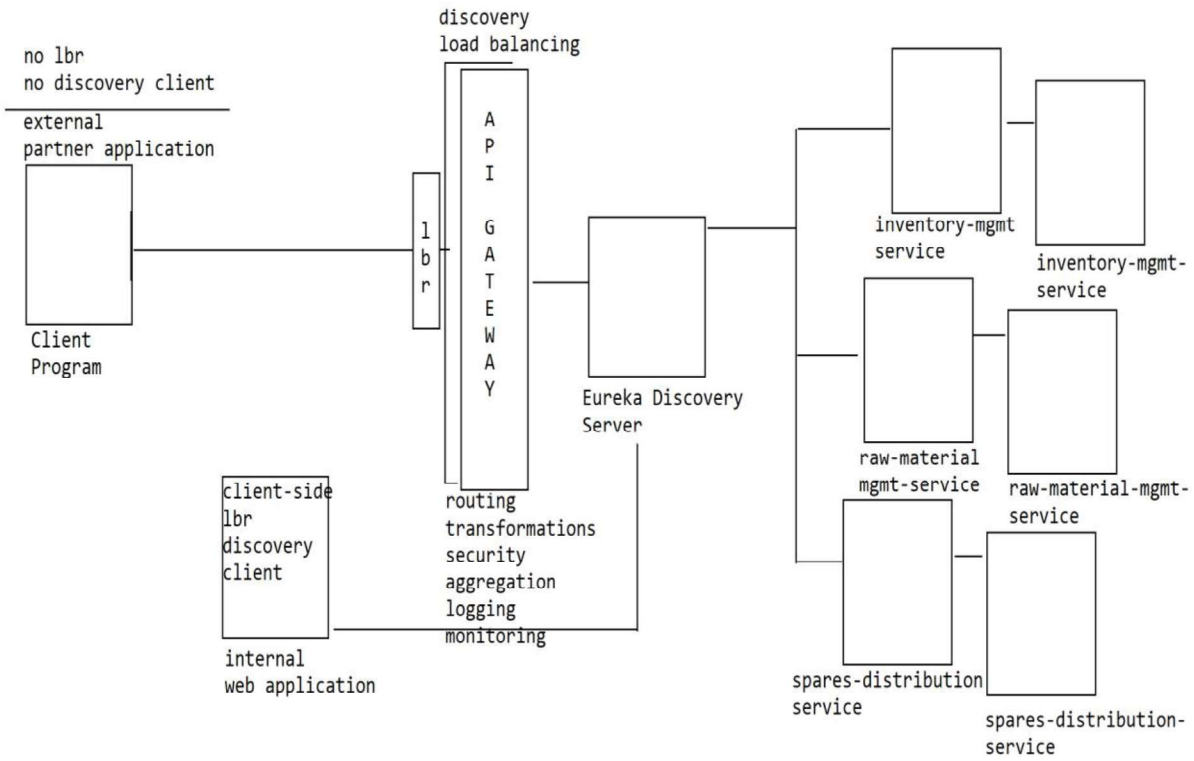
```
RouteLocatorBuilder
 |- routes (mandatory)
    |-predicates (optional)
              |-filters (optional)

tatamotors-api-gateway
|-eureka-discover-client
|-spring-cloud-loadbalancer
|-spring-cloud-gateway
|-spring-web
|-lombok

@SpringBootApplication
@EnableDiscoveryClient
class TatamotorsCloudGatewayApplication {

    @Bean
```

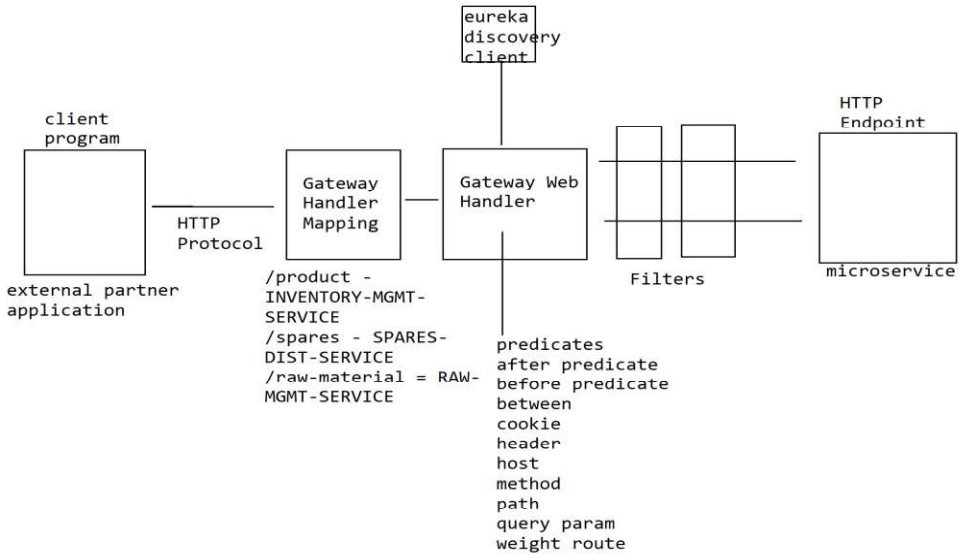
```
    public RouteLocator
routeLocator(RouteLocatorBuilder builder) {
        return builder.routes().route("stock list", r -
> r.path("/stock/**").uri("lb:/INVENTORY-MGMT-
SERVICE")).build();
    }

    public static void main(String[] args) {

    SpringApplication.run(TatamotorsCloudGatewayApplica
tion.class, args);
    }
}
```

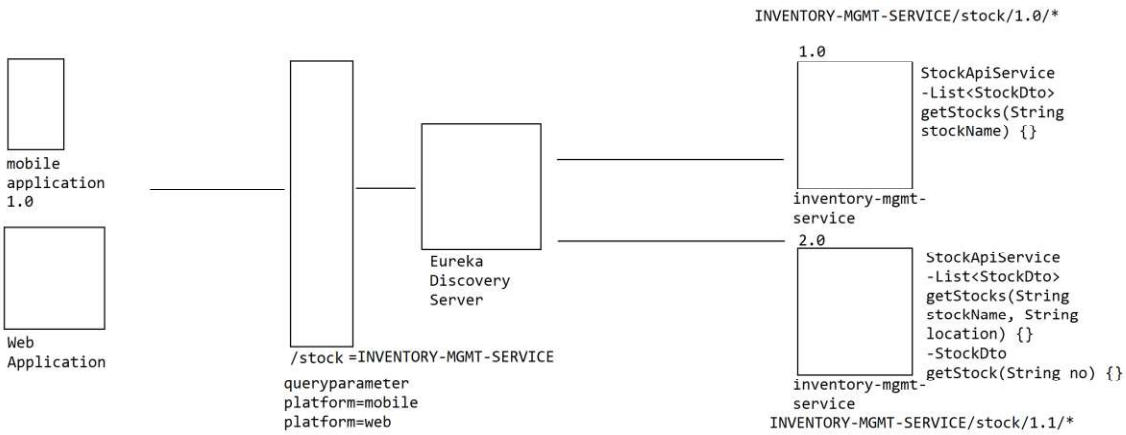
application.yml

---

```
server:
    port: 8088
eureka:
    client:
        register-with-eureka: false
        fetch-registry: true
        service-url:
            defaultZone: http://localhost:8761/eureka
```


no lbr
no discovery client
external
partner application
Client
Program
discovery
load balancing
lbr
API
GATEWAY
routing
transformations
security
aggregation
logging
monitoring
Eureka Discovery
Server
client-side
lbr
discovery
client
internal
web application
inventory-mgmt
service
inventory-mgmt-
service
raw-material
mgmt-service
raw-material-mgmt-
service
spares-distribution
service
spares-distribution-
service


eureka
discovery
client
client
program
HTTP
Protocol
external partner
application
Gateway
Handler
Mapping
/product -
INVENTORY-MGMT-
SERVICE
/spares - SPARES-
DIST-SERVICE
/raw-material = RAW-
MGMT-SERVICE
Gateway Web
Handler
predicates
after predicate
before predicate
between
cookie
header
host
method
path
query param
weight route
Filters
HTTP
Endpoint
microservice




## spring cloud gateway

spring cloud gateway acts as an gateway of receiving requests in an microservices based application, all the common functionality/logic that has to applied across multiple microservices of our application like

- security
- transformations
- routing
- logging
- monitoring
- aggregation

can be applied at api gateway level easily. In addition to the above api gateway handles the responsibility of load balancing and discovering the microservices through discovery registry.

upon receiving the request api gateway applies routings with predicates in determining the request to be routed to which microservice of the application. There are lot of predicates we can apply on a request based on http protocol like

- **time-based predicate**
    1. before
    2. after
    3. between
- http header
- cookie
- path
- http method
- query parameter
- path parameter

For each the perdicates there are supported PredicateRouteFactory are there which takes care instantiating the GatewayPredicate implementation object with appropriate predicate logic within them.

Based on predicate evaluates to true/false, matching route will be executed and the request will be forwarded to the appropriate microservice.

**There are 2 ways we can work on spring cloud gateway.**

1. either through configuration approach
2. through programmatic approach

## #2. programmatic approach

```
tatamotors-cloud-gateway
|-src
    |-main
        |-java
        |-resources
|-pom.xml
    |-spring-cloud-gateway
    |-spring-boot-starter
    |-spring-cloud-starter
    |-spring-cloud-load-balancer
    |-spring-cloud-circuitbreaker-resilience4j
    |-lombook
    |-spring-webflux
```

netty server: reactive

```
@SpringBootApplication
@EnableDiscoveryClient
```

```
class TataMotorsCloudGatewayApplication {


    @Bean

    public RouteLocator
routeLocator(RouteLocatorBuilder builder) {

        return builder.routes().route("stockavailable",
r -> r.path("/stock/**").uri("lb://INVENTORY-MGMT-
SERVICE")).build();

    }

    public static void main(String[] args) {


    SpringApplication.run(TataMotorsCloudGatewayApplica
tion.class, args);

    }

}


application.yml

server:

    port: 9999

eureka:

    client:

        register-with-eureka: false

        fetch-registry: true

        service-url

            defaultZone: http://localhost:8761/eureka
```

```
@RestController

@RequestMapping("/stock")

class StockApiController {

    @GetMapping(value="/{stockName}/available",
produces = {MediaType.APPLICATION_JSON_VALUE})

    public List<StockDto> getStocks(String stockName) {

        return stockDtos;

    }

}
```

```
#example1

forward the request to v1 or v2 service based on query
parameter platform=web/mobile

spring:

  cloud:

    gateway:

      routes:

      - id: stockv1

        uri: lb://INVENTORY-MGMT-SERVICE1

        predicates:
```

```
        - Path=/stock/**
        - name: Query
          args:
            param: platform
            regexp: mobile
      - id: stockv2
        uri: lb://INVENTORY-MGMT-SERVICE2
        predicates:
        - Path=/stock/**
        - name: Query
          args:
            param: platform
            regexp: web
```

programmatic java configuration:

```
//@Bean
    public RouteLocator
routeLocator(RouteLocatorBuilder builder) {
        return builder.routes()
                .route("stockv1",
                        r ->
r.path("/stock/**").and().query("platform",
"mobile").uri("lb://INVENTORY-MGMT-SERVICE1"))
                .route("stockv2",
```

```
                                r ->
r.path("/stock/**").and().query("platform",
"web").uri("lb://INVENTORY-MGMT-SERVICE2"))

                    .build();

    }
```

---

eureka server

cloud config

cloud loadbalancer

circuit breaker

feign client

gateway

---------- programming

ha eureka server

eg kafka microservices

oauth: syllabus

**design patterns:**

---

#1 how to break down an application into microservices?

#2 how to design database schema in microservices?

#3 aggregate the responses from multiple microservices in returning to the client using api gateway

#4 cqrs (command-query and responsibility segregation)

#5 saga design pattern

- coreography
- orchestrator

config server = Yes

eureka server = Yes

spring cloud load balancer = Yes

- 2 types are there

client-side loadbalancer

server-side loadbalancer

1. client-side load balancer = bi-passing the api gateway (internal/in-house applications)

2. server-side load balancer

- 2 levels load balancer

2.1 to loadbalance the requests between the api gateways (platform/cloud load balancer)

2.2 api gateway to distribute the traffic across the instances of microservices (spring cloud gateway/discover client)

feign client api = Yes

spring cloud circuit breaker= Yes

spring cloud gateway = Yes

router = is a path to a microservice

predicate = conditions to be evaluated to match the request

filter = modifying request/response (transformations)

## Microservices design patterns

---

## #1 how do you break down your application into microservices?

**or**

## how do you decompose your application into microservices?

**or**

## how do you identify the microservices of your application?

---

There are multiple technics in decomposing the application into microservices.

## 1. decompose based on business capabilities

business capabilities are the functionalities that generates a value to the business. For eg.. in a typical automobile management system there are different functionalities/capabilities the system will offer like

- raw material management
- spares production and distribution
- inventory management
- automobile production
- marketing

so we can divide our application into 5 microservices pertaining the capabilities of our system. we can derive capabilities out of requirements of the business system.

## 2. decompose based on subdomain

we can decompose based on subdomains, where we can identify the subdomains of the system through study of business requirements. in an automobile management application we can broadly divide the system based on subdomains as below.

#1. procurement

- raw material management
- inventory management

#2. order management service

- spares distribution
- automobile distribution

#3. production service

- automobile production
- spares production

#4. marketing

## 3. single-responsibility principle

each microservice should serve only one functionality, so that there should be only one reason for modifying a microservices, that makes the microservices loosely coupled. for eg.. in automobile management system, we can break the system into 6 microservices

1. raw material management service
2. spares production
3. spares distribution
4. inventory management
5. automobile production
6. marketing

## 4. service per team

depends on the number of teams working we breakdown them into microservices. lets say we can divide and distribute the project into 4 teams as, where each team is responsible for developing and delivering the microservice as well

1. inventory management team
2. production and marketing team
3. spares production and distribution
4. raw material management

---

## How do we design the database for a microservice based application?

There are 2 patterns in designing the database for microservice application

### 1. database per service

it is recommended to design the database schemas based on service per db pattern so that the microservices will be completely loosely coupled and maintainable

### 2. shared database

there are reasons why we need to go for single database / shared database across the microservices

2.1 we want to implement acid transactions across the microservices

2.2 we want to access the data across the microservices more oftenly

2.3 if we are migrating an legacy application where a shared database is already being used then we can continue use the shared database rather than decomposing

2.4 simple to operate

---
---

## How does microservices communicates with each other?

Let us say we have requirement where a microservice in order to fullfil the functionality it has to gather the data by talking to multiple microservices how can we aggregate the data and communicate with multiple microservices in our application?

There are 2 design patterns we need to apply in intra-communication of the microservices.

### 1. api composer

implement an api composer service in which we invoke the microservices and performs data aggregation in-memory and returns

#### advantages:

1. microservices are loosely-coupled

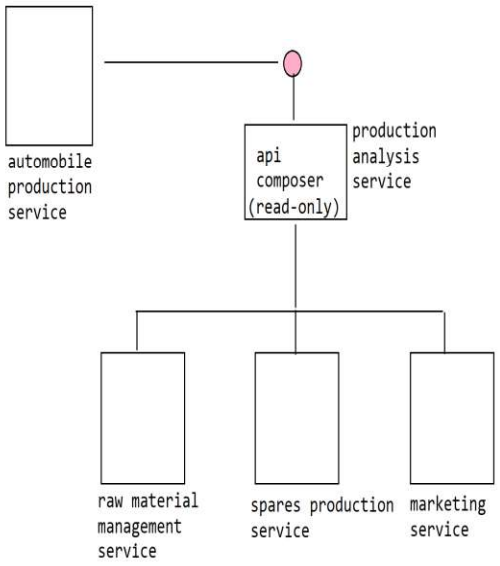
2. resuse the service callouts and data aggregation logic across the services whoever requires the same information

#### dis-advantages:

1. in-memory data aggregation is not suitable for large volumes of data

2. results in performance problems, if the data has to be access more frequently

api composer




1. each of the microservices are loosely coupled from each other through api composer service
2. if multiple microservices requires the same information, then each them has to implement service callout logics and data aggregation logic across which will results in duplication of the functionality, instead of it if we introduce the api composer, each of the otherservices can pull the data by talking to api composer service.

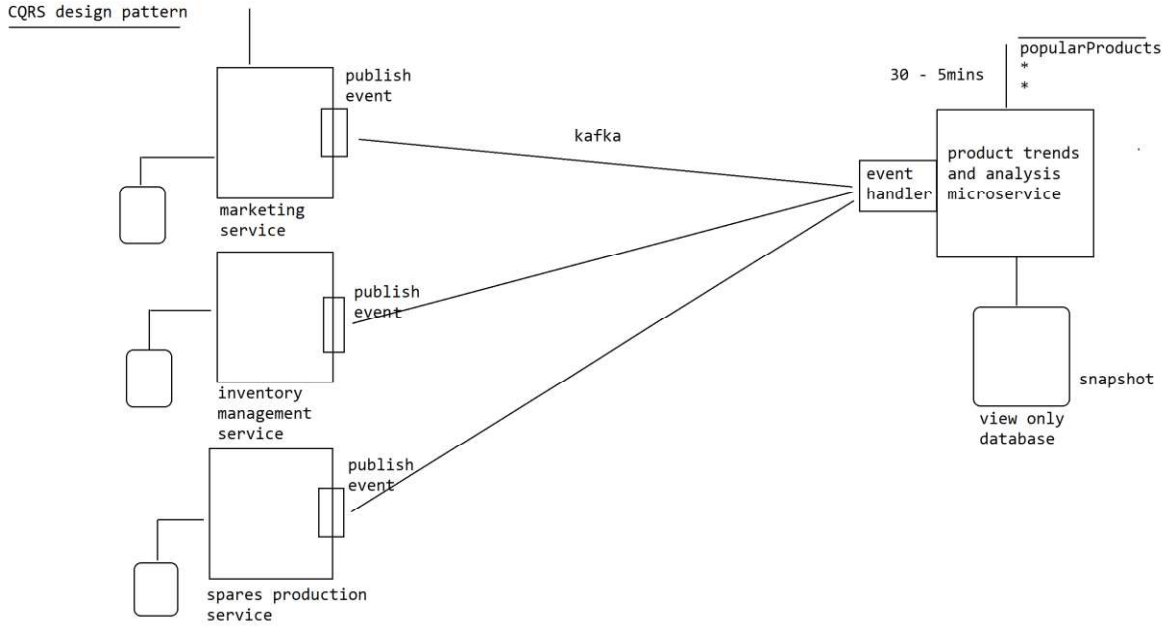
need:-
we need to implement api composer when we are using database per service

drawbacks:-
1. if the individual service callouts are resulting in large datasets of data, then aggregating the data in-memory becomes in-efficient and huge memory will be wasted
2. frequent access to api composer results in huge network calls across the microservices

---

## 2. cqrs (command query responsibility seggregation)******

CQRS design pattern




---

## #1. how to decompose the business system into microservices?

There are 4 ways are there based on which we can identify or break down the system into microservices

1. based on business capabilities
2. based on sub-domain
3. single responsibility principle
4. service per team

## #2. how to design the database for an microservice application?

There are 2 ways we design the database schema in an microservice application

### 1. database per service pattern

Per each microservice we need to create a separate schema pertaining to that microservice.
advantage:

1. is we achieve loose-coupling

dis-advantage:

1. if a microservice requires the data that is part of another microservice schema, inorder to fetch the data we need to make a service

callout which results in lot of disadvantages like

more service callouts

huge network congession

more bandwidth consumption

## 2. shared database

let all the microservices uses one shared database in storing and accessing the data.

**advantage:-**

1. better suitable for accessing the data of any other microservices directly through database queries rather than service callouts

2. we can implement 2 phase commit/global transactions across the services easily as there is a shared database

3. simpler to implement

**dis-advantage:**

1. the microservices are tightly coupled with each other through he shared schema. a change in the underlying database table would affect multiple microservices

---

## How can a microservice can communicate and aggregate the data from multiple microservices?

There are 2 design patterns are there

1. api composer
2. command query responsibility seggregation (cqrs)

## #1. api composer

if we are using database per service model and if a microservice wants to query the data from multiple other microservices and aggregate the data, instead of each service querying the data from multiple microservices and aggregating we can an api composer service.

api composer is responsible for querying the data from multiple microservices and aggregates the data in-memory and returns. So that all the microservices who requires the same data can go through the same api composer thus making all the microservices loosely coupled.

## advantages:-

1. all the microservices are loosely-coupled

2. the logic for service callouts and data aggregation can be reused across all the services

**dis-advantage:-**

1. querying large volumes of data and aggregating the data in memory will results in higher usage of memory

2. the more the calls to the api composer results in more number of networks call to the microservices which results in poor performance

**#2. command query responsibility segreggation (cqrs)*****

each microservice upon performing an action or an operation will trigger or publish event with the data containing or describing the change.

we register listener for all the events published by the individual services, receives the event performs the processing and stores the data in read-only database.

So the aggregator service will receive the request and fetch the data from read-only

database and returns rather than querying the data across all the services. With this all the dis-advantages we have discussed or resolved.

***advantages:-***

1. each microservice is loosely coupled

2. no repeated network calls in fetching the same data from the other microservices

3. no in-memory data aggregation so efficient usage of the memory

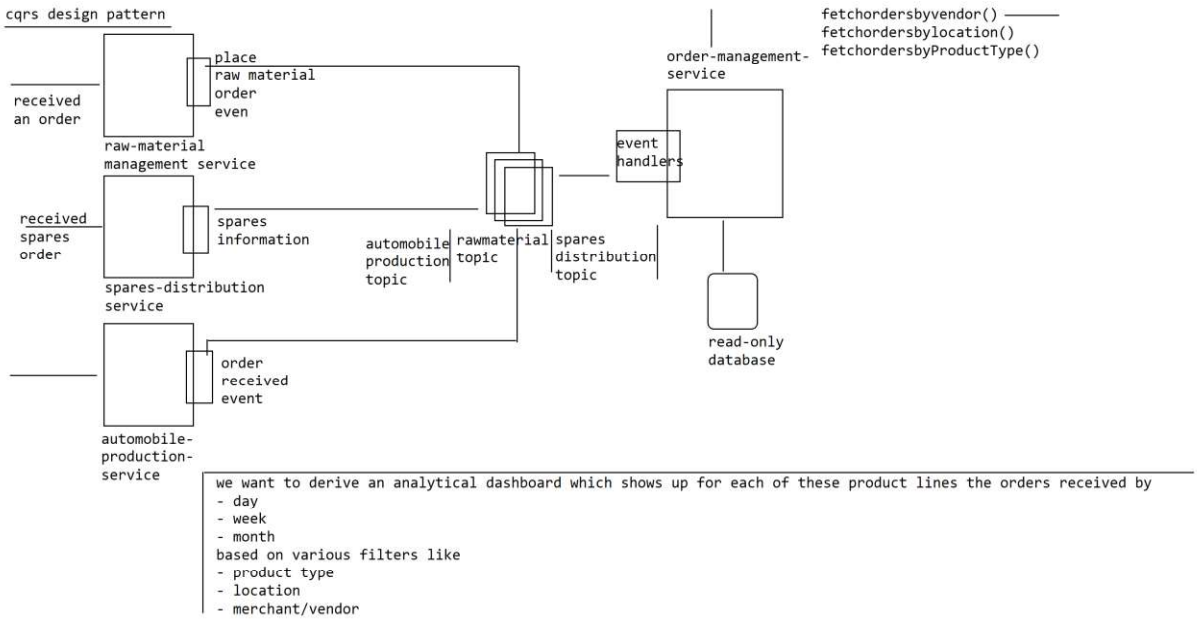
4. no network congession

5. no consumption of network bandwidth which results in better performance of the microservice

**dis-advantage:**

1. highly complex to design and implement

---
---




we want to derive an analytical dashboard which shows up for each of these product lines the orders received by
- day
- week
- month

based on various filters like
- product type
- location
- merchant/vendor

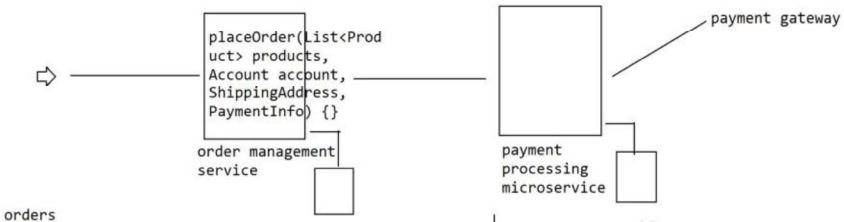
## How to manage transactionality when it comes to microservices?

To **apply transactionality** across the microservices we need to **implement saga design pattern.**

There are 2 ways/types of implementing the saga design pattern

1. choreography-based saga
2. orchestration-based saga

---



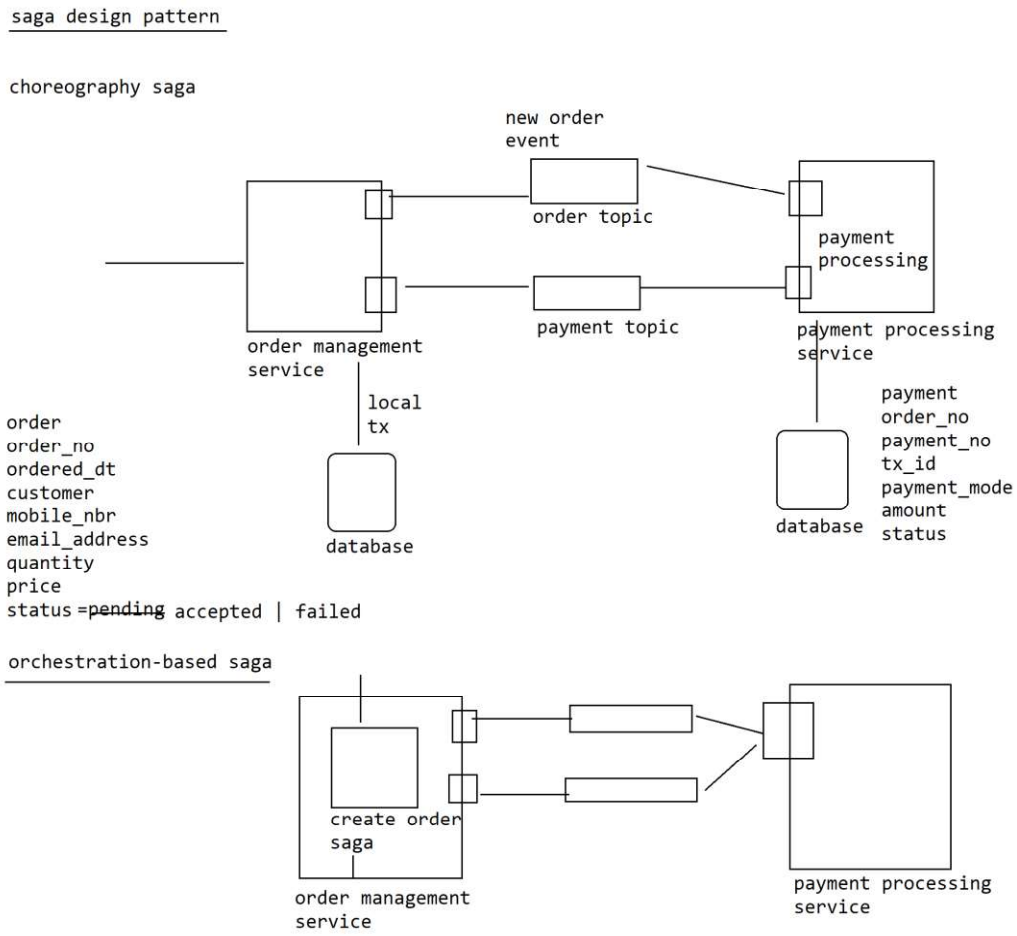

orders
order_no order_placed_dt customer mobile email address quantity price shipping_address_id billing_address_id

address
address_id address_line1 address_line2 city state zip country

order_line_items
order_id product_id quantity unit_price

payment_table
order_id payment_id payment_method transaction_ref_id date amount status

There are several limitations or drawbacks in implementing global transactions in microservices world.
1. we cannot achieve highest level of concurrency in processing the requests by application global transactions
2. if the services are implemented using different technology or language feasibility of application global transactions is not possible.




## How to implement transactionality across the microservices?

When we choose database per service design pattern, they could be a business transaction that might be spanning across multiple microservice calls in such case 1 solution we have in applying the transactionality across the microservices is using global / 2pc transactions.

we can use 2-phase commit or global transactions in managing the transactionality that spans across multiple databases in microservices, but there are lot of limitations or drawbacks are there

1. in case of 2-phase commit when a microservice has begin the transaction and perform the operation on the database, within the active transaction if it propagates the call to another microservices then the current active transaction locks the database and underlying schema util all the participating resources are commited/rollback due to which it results in poor concurrency levels.

we can consider global transactions as a anti-pattern for microservices, since we use microservice architecture for concurrency and scalability.

2. In case of microservices each service might have different resources (persistence stores) where few supports global transactions other may not. similarly different microservices in the application may have been implemented in different technologies due to which implementing global transactions could be in-compatible

by considering the above reasons the global transactions is not an solution in managing the transactions across the microservices.

To overcome the above problem, we use saga design pattern to manage transactionality across the microservices.

## Saga design pattern

---

In saga design pattern each microservice performs the operations on their database in local transaction by coordinating and applying an compensating operation incase of a failure with the other.

**There are 2 types of saga design pattern**

**1. choreography-based saga**

**2. orchestration-based saga**

## #1. choreography-based saga.

upon receiving an request by a microservice to carry an operation, the microservice performs the business operation and stores the data in its database under local transaction, and inorder to propagate perform an related operation by the other microservice to complete

business operation, it publishes an event in the topic asking the other microservice to complete the business operation.

The other microservice receives the event through listener and performs the operation and based on the outcome of the operation it again publishes an event in reply back topic to the initiator/callee service.

In case of failure the callee microservice executes an compensating transaction to bring the system into consistent state which is called saga design pattern

1. order service (received the order)

2. let the order service create the order and insert the data into database table, but to indicate the order is still pending for payment processing it stores the status of the order as pending and commits the local transaction to immediately allow other requests to be processed.

3. but in order to complete the payment processing, the order service has to communicate with payment processing service.

4. now instead of making an synchronous call to the payment processing service

    1. if it is a synchronous call, resource utilization will increase

    2. the services will become tightly coupled

the order service has to publish an event to an event topic, where in the payment processing service as a message listener, who reads the order information and proceeds for payment processing.

5. based on the outcome of the payment processing, the payment processing service will places a event/message back into reply back topic, now the order service will recieves the ack message from payment processing service and based on the outcome it applies an success or compensating operation either updating the order record to approved / failed

so that both the services and business operations are consistent in nature at any point of time, thus we are able to achieve the similar affect of applying global transactions by

executing each local transaction in a coordinated manner

## #2. orchestration-based saga

in case of choreography saga, the initiator service has to be written with lot of logic in coordinating and communicating by exchange messages and apply compensating operations to achieve data consistence and transactionality.

So that the logic in the microservice has been polluted along with business logic even cooridination and transaction managment logic as well, which can be separated out and can be written in another component called orchestrator.

upon receiving the request by the order service, it pass down the request to the orchestrator component to carryout the operation and enforce transactionality through message exchange. So that the original service will have only business logic being implemented and transaction coordination logic is taken care by orchestrator.

For each business operation that spans across the multiple microservices we are going to write one orchestrator component.

## How to achieve high-availability in eureka server?

```
tatamotors-eureka-server
|-src
    |-main
        |-java
        |-resources
            |-application.yml
|-pom.xml
```

Now we want to run multiple eureka servers as peers to ensure high availability of the discovery servers.

In real-time we run each peer on different machine which has different host/port number through which an eureka server will identify the peer.

if the peer is also running the same host of the primary eureka server, it will not be recognized as peer since there is no way to achieve ha by running both eureka instances on same computer.

Right now for our experimentation we have only 1 computer to make the eureka treat 1 machine as 3 we are going to configure the ip of our computer with 3 different local domain names under hosts file.

windows:-

goto C:\Windows\System32\drivers\etc

open hosts file under administrator rights

linux:

/etc/hosts

under sudo or root user

make an entry into the hosts file with 3 domain names pointing the same ip address as shown below.

127.0.0.1    discoveryserver1.com

127.0.0.1    discoveryserver2.com

127.0.0.1    discoveryserver3.com

now we made it look like 3 computer differents since those are 3 domain names but all of them are referring to same machine.

To run 3 eureka servers pointing each of the others as peers we need to create 3 projects, instead of it we can make use of profiles.

when we run 3 projects

#1 project refer -> in configuration 2, 3 as peers

#2 project refer -> in configuration 1, 3 as peers

#3 project refer -> in configuration 1, 2 as peers

that means the only difference between three projects is configuration only. just to run 3 eureka servers

with 3 configurations we dont have to create 3 projects, rather in one project we can create 3 profiles.

application.yml

---

```
---
server:
    port: 8761
spring:
    application:
        name: tatamotors-eureka-cluster
  config:
        activate:
            on-profile: discoverserver1
eureka:
    instance:
        hostname: discoveryserver1.com
    client:
        register-with-eureka: true
        fetch-registry: true
        service-url:
```

```
            defaultZone:
http://discoveryserver2.com:8762/eureka,
http://discoverserver3.com:8763/eureka

...

---

server:

    port: 8762

spring:

    application:

        name: tatamotors-eureka-cluster

  config:

        activate:

            on-profile: discoverserver2

eureka:

    instance:

        hostname: discoveryserver2.com

    client:

        register-with-eureka: true

        fetch-registry: true

        service-url:

            defaultZone:
http://discoveryserver1.com:8761/eureka,
http://discoverserver3.com:8763/eureka

...
```

```
---
server:
    port: 8763
spring:
    application:
        name: tatamotors-eureka-cluster
  config:
        activate:
            on-profile: discoverserver3
eureka:
    instance:
        hostname: discoveryserver3.com
    client:
        register-with-eureka: true
        fetch-registry: true
        service-url:
            defaultZone:
http://discoveryserver1.com:8761/eureka,
http://discoverserver2.com:8762/eureka
...


java -jar -Dspring.profiles.active=discoveryserver1.com
target\tatamotors-eureka-server.jar
```

```
java -jar -Dspring.profiles.active=discoveryserver2.com
target\tatamotors-eureka-server.jar

java -jar -Dspring.profiles.active=discoveryserver3.com
target\tatamotors-eureka-server.jar

-------------------------------------------------------
--------------------------------------

inventory-mgmt-service

|-src

    |-main

        |-java

        |-resources

            |-application.yml

|-pom.xml

    eureka-discovery-client
```

the inventory management service has to register itself to eureka server to get it discovered by the clients.

now we need to populate the information about eureka server with whom it has to register to get it discovered.

```
application.yml

-----------------

server:

    port: 8082
```

```
eureka:
    client:
        register-with-eureka: true
        fetch-registry: false
        service-url:
            defaultZone:
http://discoveryserver1.com:8761/eureka,http://discover
yserver2.com:8762/eureka,
http://discoveryserver3.com:8763/eureka
-------------------------------------------------------
---------------------------------------
tatamotors-dealer
|-src
    |-main
        |-java
        |-resources
            |-application.yml
|-pom.xml
    -eureka-discovery-client
    -spring-cloud-load-balancer


@Service
class StoreService {
    @Autowired
```

```
    private RestTemplate restTemplate;


    public List<StockDto> getStocks(String stockName) {
        http://inventory-mgmt-
service/stock/stockName/available
        // invoking the microservice
        return stockDtos;
    }
}


@SpringBootApplication
class TataMotorsDealersApplication {
    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
application.yml
----------------
spring:
    application:
        web-application: none
```

```
eureka:
    client:
        register-with-eureka: false
        fetch-registry: true
        service-url:
            defaultZone:
http://discoveryserver1.com:8761/eureka,
http://discoveryserver2.com:8762/eureka,
http://discoveryserver3.com:8763/eureka
```

## OAuth2

---

OAuth2 is a new specification than its predecessor OAuth1.0, these 2 are completely different and even not backward compatible as well.

In a traditional security implementations, we use username/password based authentication to gain access to an secured resource

adhoc group of people of engineering graduates has innovated and comeup with their own algorithmic approach of granting the access to an Http service, onbehalf of the resource owner to third-party without providing the username/password and published an white papper to the world. by looking at their efforts and algorithm looks like most of the enterprises has uptaken the security standard which is OAuth 1.0

Due to wide acceptancy of OAuth 1.0 the team has decided to enhance the framework and make it compatible with all the types of applications/clients and with broaded scope of acceptancy. That is where OAuth2 comes into picture which is more compatible to all the types of clients and IETF standard has approved.

## What is OAuth2 what is the purpose of it?

OAuth2 is not an authentication technic to protect the access to the application, rather it is an authorization layer attached in between to grant access to the HTTP service to an third-party.

## How can an Third-Party application can gain the access to the HttpService or a Resource on behalf of the Resource Owner?

Earlier to OAuth standard or by using traditional security mechanisms the Resource Owner has to pass the credentials in accessing the HttpService/Resource to the Third-Party application, so that Third-Party can produce the Credentails to the Resource in gaining the access and perform the operation.

For eg.. A customer has to provide the paytm wallet credentails to an E-Commerce website like an amazon/flipkart, so that the third-party application can access the wallet of the customer in and can bill the money to complete the order.

## This approach has lot of drawbacks let us explore

1. The Resource owner has to provide the credentails information of the Resource/HttpService to the third-party, hence the third-party might store the credentails with them permanently in a clear-text format for future use which poses a security risk.

In short = third-party has a permanent access to our resource/http service

2. The third-party gets an overly boarded access onbehalf of us to the Resource/HTTP Service. the resource owner doesnt have any control over what operations can a third-party is allowed to perform the Resource.

3. The Resource owner cannot keep track of the third-parties to whom he has granted the access to the resource to easily manage. so a fraud detection and elimination becomes tough

4. The Resource owner cannot revoke the access to an specific third-party application. Since the only way to revoke the access is by changing the password, which will revokes the access from all the third-parties rather than one.

5. if the third-party application has been compromised, it results in compromising the resource owner credentials as well and an hacker gains the access to the Resource of the Resource owner.

**How to solve the above problem in gaining the access to the Resource on behalf of the Resource owner to the third-party application.** that is where

**OAuth2 specification** comes into picture

OAuth2 is not an authentication mechanics, we cannot secure an application access using OAuth. It is used for granting a temporary access to a resources for a third-party application which is called Authorization Standard.

Restful Service

**How to secure a restful service?**

To secure a Http Service we need to use Http Security Standards like Basic Authentication or Form/Digest authentication

if we want our restful service to be granted access to a third-party onbehalf of resource owner temporarily then only we need to use OAuth.

---

## What is OAuth2, what is the purpose of it?

OAuth2 is an authorization standard through which we can grant access to a Resource on behalf of the Resource Owner to a third-party application.

In OAuth2 there are total four roles are there

### 1. Resource Owner

an entity capable of granting the access to a protected resource by authenticating and granting an access token.

### 2. Third-Party Application

An application requesting the access to a protected resource on behalf of the resource owner

## 3. Resource Server

The Resource Server hosts the services and grants the access the service by accepting accept tokens

## 4. Authorization Server

Server issues access tokens to the third-party applications upon successful authentication of the resource owner.

## The OAuth2 supports 4 types of grant

1. Authorization Code
2. Implicit

3. Resource Owner Password Credentials

We directly takes the resource owner credentails as authorization grant in getting the access token for accessing the resource.

For eg.. paytm payments bank application is a third-party wants to access paytm wallet services. as the client application is a trusted application for PayTm they can grant access tokens directly to paytm payment bank by using resource owner credentails.

## 4. Client Credentials

Client credentials are used as an authorization grant, when the client is acting on its own behalf (the client is the resource owner)

**Spring Security has provided OAuth2 module**, that helps us in quickly building the Authorization Server, Resource Server and Authentication in generating grants and access tokens allowing the third-parties to access the resources onbehalf of the Resource Owner.

```
paytm-oauth2
|-src
    |-main
        |-java
        |-resources
            |-application.yml
|-pom.xml
    - spring-starter-web
    - spring-starter-security
    - spring-cloud-oauth2
    - spring-cloud-autoconfigure-resource-server
```

```
@RestController
@RequestMapping("/api/wallet")
class PayTMWalletResource {
    @PostMapping(value="/pay", consumes =
{"application/json"}, produces={"application"})
    public PaymentStatus pay(PaymentRequest request) {
```

```
        // deducts the money from customer wallet and
credits to merchant
        return status;
    }
}
```

PaymentRequest

- orderNo
- orderedDate
- merchantId
- amount

PaymentStatus

- orderNo
- txId
- transactionDate
- status

---

**#1 Resource Server**

Resource Server = is configured with default url pattern ("/*")

```
@Configuration
@EnableResourceServer
```

```
class PayTMResourceServer extends
ResourceServerConfigurerAdapter {

    public void configure(HttpSecurity http) {


    http.authorizeRequests().antMatches("/api**").authe
nticated().anyRequest().permitAll();

    }

}
```

**Resource Server =** REST/Http Endpoint that contains logic for reading access token and verify it is valid or not, based on which forwards the request to the Resource.

From the above we are telling ResourceServer validate every request which is coming with url pattern "/api**" for access_token. for other url patterns simply ignore

---

## #2 Web Security (Basic Authentication)

paytm-login.jsp

---

```
<h2>PayTM Login</h2>

<form
action="${pageContext.request.contextPath}/authenticate
" method="post">
```

```
    username: <input type="text" name="j_username"/>
    password: <input type="password"
name="j_password"/>
    <input type="submit" value="login"/>
</form>

@Controller
class PayTMUserLoginController {
    @GetMapping("/login")
    public String showLoginPage() {
        return "paytm-login";
    }
}

@Configuration
@EnableWebSecurity
public class PayTMWebSecurityConfigurer extends
WebSecurityConfigurerAdapter {
    @Autowired
    private PasswordEncoder passwordEncoder;

    public void configure(HttpSecurity http) {
    http.csrf().disabled().formLogin().loginUrl("/login
").loginProcessingUrl("/authenticate").username_paramet
er("j_username").password_parameter("j_password").and()
```

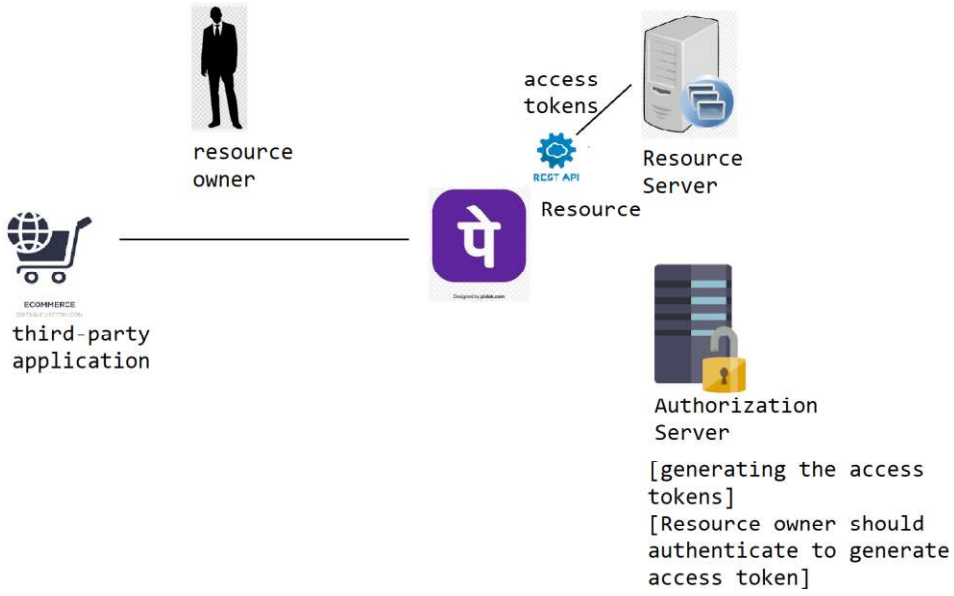
```
.authorizeRequests().antMatches("/login**",
"/oauth/authorize**").permitAll().anyRequests().authent
icated();

    }


    public void configure(AuthenticationManagerBuilder
builder) {
    builder.inMemory().withUser("joe").password(passwor
dEncoder.encode("welcome1")).athorities("customer");

    }


    @Bean

    public AuthenticationManager
authenticationManagerBean() {

        return super.authenticationManagerBean();

    }

}
```

**#3 Authorization Server**

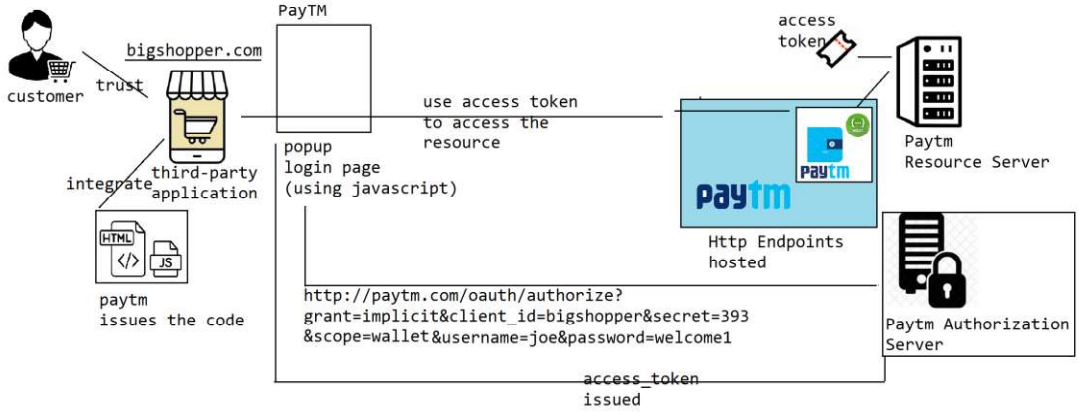
---

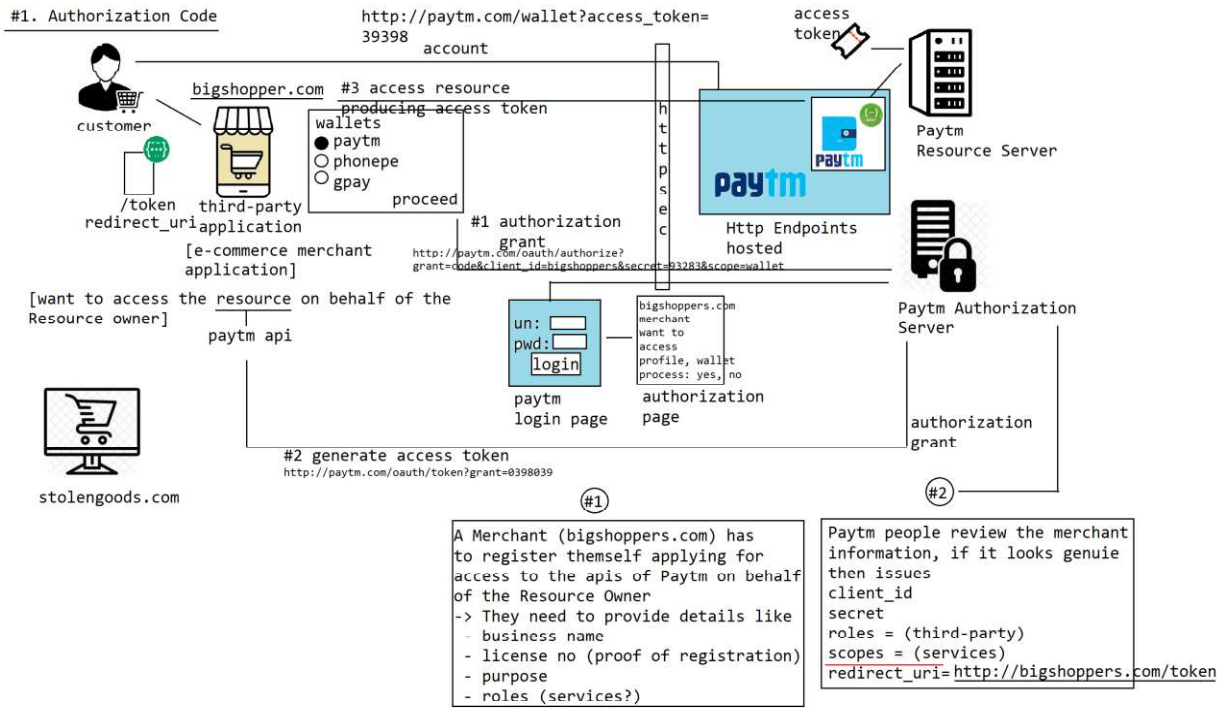
```
@Configuration

@EnableAuthorizationServer

public class PayTMAuthorizationServer extends
AuthorizationServerConfigurerAdapter {

  @Autowired
```

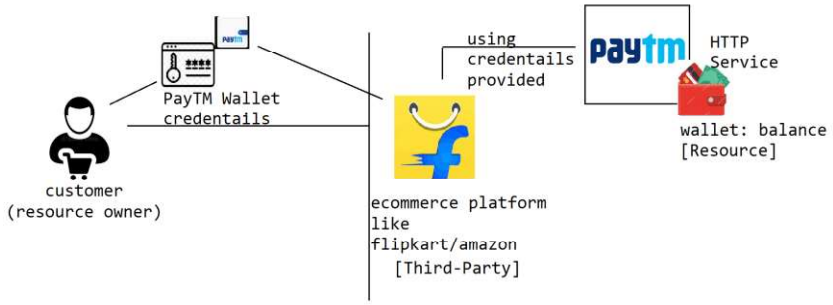
```
    private PasswordEncoder passwordEncoder;


    @Autowired
    private AuthenticationManager
authenticationManager;


    public void configure(AuthorizationServerConfigurer
security) {
    security.tokenKeyAccess("permitAll()").checkAccessT
oken("isAuthenticated()").allowFormAuthenticationForCli
ents();
    }


    public void configure(ClientServiceConfigurer
clients) {


    clients.inMemory().withClient("bigshopper").secret(
passwordEncoder.encode("9383")).authorizedGrantTypes("a
uthorization_code,
password").scopes("wallet").accessTokenValidityInSecond
s(60 * 60 *
2).authorities("READ_WRITE").redirectUris("http://local
host:8082/token");
    }
}
```




#2 Implicit grant






The Resource owner wants to grant the access to the Resource onbehalf of him to the third-party how this can be achieved?

## Apache Kafka

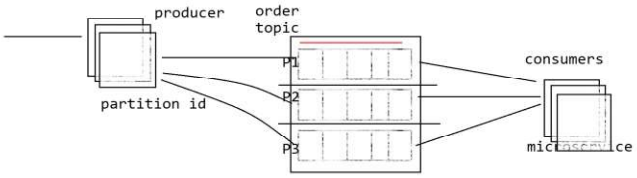
---

Apache Kafka is an opensource distributed event platform that works on Streaming api. Apache Kafka is majorly used for building microservices based applications.

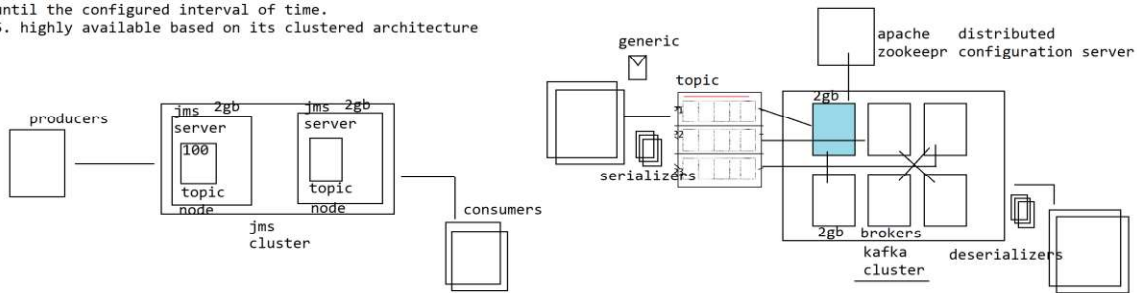
Using Spring Boot Kafka Integeration api we can easily build consumers and producers in reading and processing events on Kafka cluster

The spring boot kafka module has provided KafkaTemplate which helps in publishing an message to the specific kafka topic we specified.

Similar we can build Kafka Consumers through Listener api where listerner looks for a message in the kafka topic and when new messages arrives automatically triggers and invokes listener by passing the message for consumption




1. highest level of concurrency by distributing the events across the kafka cluster and allowing multiple consumers/producers to publish and subscribe events concurrently
2. the performce in delivering the messages in kafka clusted is limited by the network bandwidth/throughput, an average latency in delivering the messages is less than 2ms
3. highly scalable by handling trillions of messages per day, petabytes of data
4. unlike jms or other messaging systems, the kafka topics are durable, which means the data can be stored permanently in a kafka topic until the configured interval of time.
5. highly available based on its clustered architecture




```
ManagerOrderResource
- OrderStatus placeOrder(CreateOrderRequest
orderRequest) {}

OrderStatus
- orderNo
- orderedDate
- accountHolderName
- amount
- status

CreateOrderRequest
- uniqueAccountNo
- Map<String, Integer> items;
- UPIId
```

createorder topic

```
{
  "orderNo" : 9393
  "orderedDate: 303
  "upiId" :
  "amount":
}
```

CreateOrderMessage

```
ManagePaymentService
  @KafkaListener
 public void
processCreateOrderMessage(String
createOrderMessage) {
  Jackson
  ObjectMapper

}
```

# The End

# MicroServices