Microservice application

1. Normal Eureka client, server communication

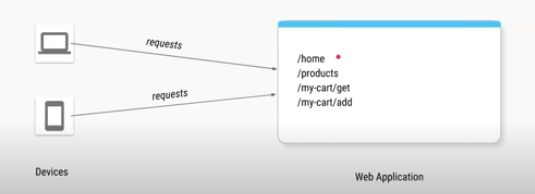
AccountConsumer, AccountProducer, DiscoveryService

2. demo-client,demo-eureka-server, hello-service

3. EmployeeConsumer, EmployeeProducer, Eureka-server

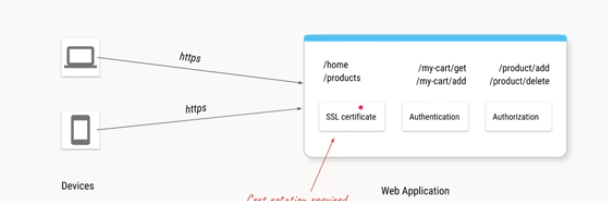
API GATEWAY - Features

1. Separate out cross cutting concens



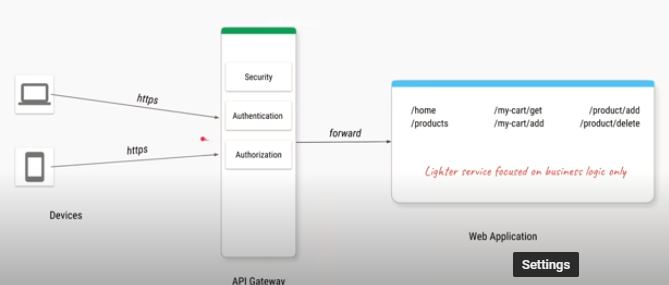
Lets say we created a ecommerce appl, for that we created a web appl which has these urls /home,/product,/my-cart/get, /my-cart/add. If u fire /home url it gives the static html or jsp so that we can see the home page. When we make next call /products it will give list of products in the form of json. If the user is logged in, the user can make request to get the items, and adding the items to the cart.

If the user fires /home or /products even if the user not logged in, we want to send the data back to the user. But if the user fires any of the operation of shopping cart which is used to get items from shopping cart or add items to cart, we want to ensure that the user should authenticated. Similarly, we have some url to add the products and delete the products which might require an administrative role, for that we want to authorize whether the user is signed in and try to access the url is actually valid administrator. We want to ensure that web appl is secured, so instead of HTTP we use HTTPS in our appl, for that we need to add SSL certificates



So basically ur web appl now has 3 added components like SSL certificates, authentication, authorization in addition to all the business logic and we want to try to separate these things which is not business logic related into a separate component. That component is called **API Gateway.**

So API Gateway is a component which acts as entry point for our API’S. In our case this API’s are hosted on web appl, so it becomes entry point for our web appl.



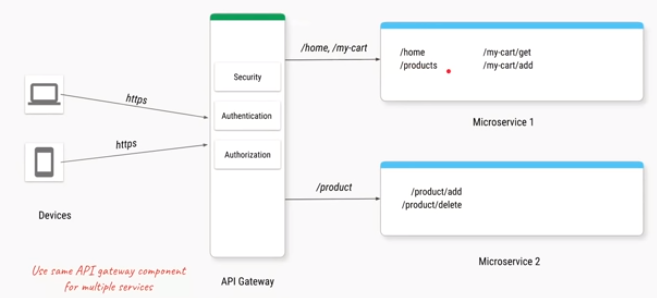
So everytime when client makes the request, first it goes to API Gateway, the gateway ensures the request are HTTPS using SSL certificate, then it will ensure the user is authenticated when user is calling /get or /add and also it ensure user is authorized if the user calls /add or /delete.

Only if this condition are satisfied then only the request is forwarded to ur web appl. That way API Gateway will act as a guard for any request which is coming to ur appl, so it will protect ur API from bad address.

1. Separate and consolidate cross cutting concern across Microservices

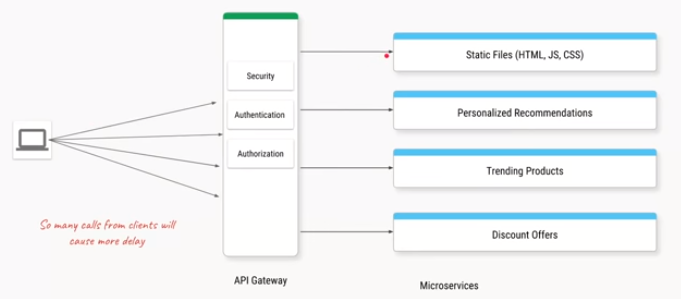
Consider now we convert single monolithic appl to multiple microservices . So if any url hit from the clients for /home or /my-cart then it goes to microservice1 and anything related to /product url, it goes to microservice2.

Since all ur request will always come from API Gateway which is a single component, where API Gateway can decides based on these paths where to route ur urls either to microservice1 or microservice2 through routing

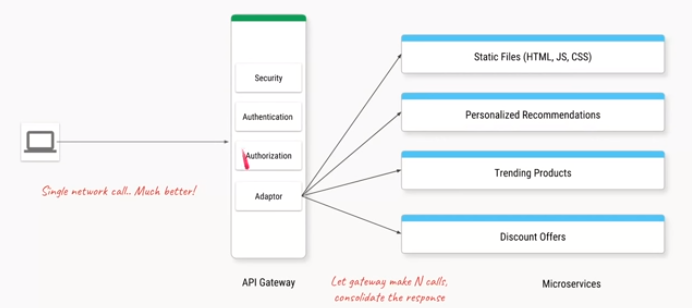


1. Replacing multiple client calls with single API calls

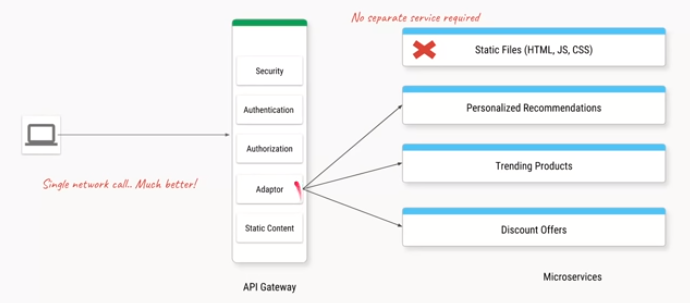
Now we want to add more features to our web application, like we want to give personalized recommendations for the logged in user, we give list of all trending products and also we have some discount offers and all these features are provided by microservice.



Now the client first request for home page and get all static files and we make AJAX call which is HTTP call for personalized recommendation, trending products and discount offers. Now we have 4 HTTP call made by the client to ensure that home page is rendered correctly. It is possible that if the appl grows further we can more services and client have to make more number of HTTP calls to render single home page.

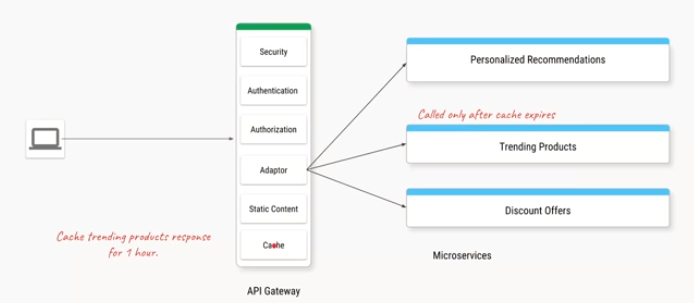


Instead the better options is the client will send single call to request a home page and will have some component in API Gateway called Adaptor which will make all these 4 calls on behalf of client and consolidate the response from these services and send back single response to the client.



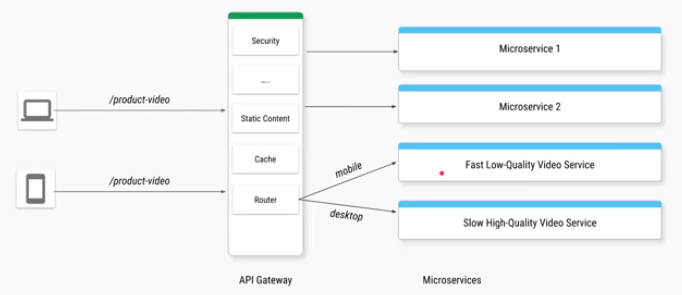
Also since there is one service serving only static files which is not best use case for microservice, there is no business logic, it is serving the same file again and again. We can have component within API Gateway where there is a request for static files, the API Gateway itself will return the response and that way we can remove the microservice completely.

Lets we have service to fetch trending products and trends of the product are recalculated only after 1 hr, so if there are 50 calls within 1 hr by the client, it always will be forwarded by API Gateway to this microservice and the microservice keep returning the same response again and again.



API Gateway has a feature called response cache where u give a url and threshold time where it needs to cache the response. So the first time when client makes the call, it forwards to trending product microservice and get the response and cache it for 1hr, and next time when client sends the same request instead of going back to service it will return the response from cache itself.

1. Routing based on headers, paths and params etc



Lets say we expose a url called /product-video and will have 2 services in backend. One service which responsible for low quality video which will much faster and other service for high quality video which is slow. In API Gateway we will configure such that if request comes from mobile, it routes that request to fast service and if request comes from desktop we route that service to high quality service.

Lets say one service have too many copies to that service, in that case API Gateway also takes responsibility of being load balancer.

**ZUUL PROXY AS API GATEWAY**

Zuul is like a front door to server instances, all request flows through Zuul and then goes to the services. Zuul can be used as authentication, monitoring, routing, filtering and load balancing component.

Zuul Filter is a class, that provides you different methods for filtering the requests. Zuul has 4 filtering mechanism like pre filter, post filter, error filter and route filter.

When the request comes to ur gateway, it first hits the pre filter, then route filter and then post filter. Error filter is something that runs in the background, if there is any error in filtering mechanism it immediately catches.

Zuul filter has 4 important characteristic like type, execution order, criteria and run()- which contains logic of filter.

1. Zuul is an edge service or an API Gateway that proxies' requests to multiple backing services.

2. Zuul provides a unified “front door” to your system, which allows a browser, mobile app, or other user interface to consume services from multiple hosts without managing cross-origin resource sharing (CORS) and authentication for each one.

3. In simple words, Zuul receives all the requests coming from the UI (either web or mobile) and then it delegates the requests to internal microservices.

4. Zuul is built to enable dynamic routing, monitoring, resiliency and security.

5. Zuul is a JVM-based router and server-side load balancer from Netflix.

6. Zuul is a series of Filters that can perform a range of actions during the routing of HTTP requests and responses.

7. Zuul provides a framework to dynamically read, compile, and run these Filters. Filters do not communicate with each other directly - instead, they share state through a Request Context which is unique to each request. ========================

Types of filters in Zuul

Pre – Filters execute before the request routing to the origin.

Routing – Filters handle during routing the request to an origin.

Post – Filters execute after the request has been routed to the origin.

Error – Filters execute when an error occurs during one of the other phases.

**Example 2 – Refer DoctorService, DiagnosisService, ZuulProxy2**

**https://github.com/Java-Techie-jt/spring-cloud-zuul**

1. Run all application
2. No eureka server
3. Run to access doctor service, <http://localhost:8089/doctor-api/getDS>
4. Run to access doctor service, <http://localhost:8089/doctor-api/welcome/sam>
5. Run to access diagnosis service, <http://localhost:8089/diagnosis-service>
6. Check in ZuulProxy application, all filters will be printed in console

**Example 3 - ZuulEurekaServer,ZuulConsumer1, ZuulConsumer2, ZuulApiGateway, ZuulFilter**

**Refer https://www.youtube.com/watch?v=Iaz2apI0Ago**

1. Run all the application
2. Run eureka in <http://localhost:8082> – it will register 3 instances
3. Access consumer1 application through zuul using

<http://localhost:8081/con1/consumer1/get/mahesh>

1. Access consumer2 through zuul using

<http://localhost:8081/con2/consumer2/get/sam>

1. Now stop ZuulApiGateway and run ZuulFilter appl just to see the filter information
2. Access consumer1 application through zuul using

<http://localhost:8081/con1/consumer1/get/mahesh>

Now we can see filters in ZuulFilter application

**SWAGGER -- used to document api**

Previously we executed Rest API using Postman, then we have to setup Postman for each and every API method for testing. Postman site dosent have visibility about the number of API methods which are exposed by Rest API until and unless we go and check into the code directly. In order to automate the process of API execution and Swagger exactly do the same thing.

Swagger automatically captures the details from API, it reads the various methods present in API, and put together in a flexible UI screen. If there is any change in the REST API, it automatically read the update code and update the swagger UI, just by restarting the application.

1. Swagger is a open source API documentation framework to help developers in design, document, and consume RESTFUL web services.
2. Swagger reads an API and extract in the form of interactive UI called Swagger UI
3. Swagger UI offers HTML view of API with JSON format.

**Steps: Refer SpringBoot-Swagger**

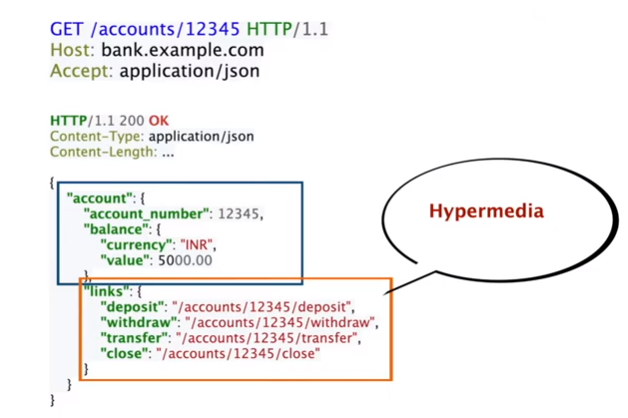
1. Configure dependency in pom.xml
2. Run the appl
3. Run <http://localhost:8080/swagger-ui.html> -- to see all REST endpoints

https://www.youtube.com/watch?v=HHyjWc0ASl8&t=468s

**HATEOAS – Hypermedia as the Engine of Application State**

It is representational model of REST where this particular architecture provides links to different REST endpoints where u can expose different hypermedia links in the REST endpoint itself.

Consider an example where GET request is invoked to query the account balance. The response will contain account info which request by REST Api, but along with account balance it consist of some rest endpoint links in the response which is called as Hybermedia which is responsible for driving the application state. There are 4 links where banking client can take now such as deposit, withdraw, transfer and close the account



Consider an another example where same GET request is invoked to query the account balance and the account has insufficient funds and hence links are placed dynamically by the server after considering the actions which can be taken forward. As funds are negative, the only action which can be carried out by this user is deposit action, the user cant withdraw, transfer or close the account as his balance is less. This how hypermedia helps in driving the state of application.



Example 1: Refer SpringBoot-Hateoas

1. Start spring boot application
2. In Postman with POST request, create a new account with body-raw-json with <http://localhost:8080/api/v1/create> with following data

{

    "currency":"INR",

    "amount":2000,

    "accountNumber":12345,

    "rateOfInterest":"3",

    "accountType":"SAVINGS",

    "accountStatus":"ACTIVE"

}

1. In Postman with GET request, retrieve the balance using <http://localhost:8080/api/v1/account/12345>

Which retrieves the account information

Here we want to create a link to deposit the amount, to transfer the amount and withdraw the amount on the basis of amount available. If the amount is positive value then user can transfer, withdraw or deposit the amount and if amount is 0, then he can only deposit the amount, user cant withdraw or transfer the amount.

1. In Postman with PUT request, deposit amount in the account with body-raw-json with <http://localhost:8080/api/v1/deposit> with following data

{

    "currency":null,

    "amount":25000,

    "accountNumber":12345,

    "rateOfInterest":null,

    "accountType":null,

    "accountStatus":null

}

The amount will be deposited into the account

1. In Postman with PUT request, deposit amount in the account with body-raw-json with http://localhost:8080/api/v1/withdraw with following data

{

    "currency":null,

    "amount":3000,

    "accountNumber":12345,

    "rateOfInterest":null,

    "accountType":null,

    "accountStatus":null

}

Now the amount will be withdraw from the account

1. In Postman with POST request we can transfer the amount from one account to another using <http://localhost:8080/api/v1/transfer> with body-raw-json

{

   "fromAccountNumber":12345,

    "toAccountNumber":12346,

    "transferAmount":2000,

    "mode":"NEFT"

}

Now it display “Invalid destination account number”, for that we need to create new account with 12346

Let say we have REST endpoint called getCustomers, we provide the customer information, along with that we provide the link for the customers specific information or some other link to the next REST url which is specific to that particular customer. That is called HATEOAS

HATEOAS cannot be used in every REST implementation,if u don’t have any hypermedia links which u want to link it with particular REST endpoint no need to use HATEOAS. But if u want to link different hypermedia links to ur REST points then u can use HATEOAS.

**Steps: Refer SpringBoot-Hateoas**

1. Create UserResource and exposed as REST endpoint, which contains getAll() which list all the Users, so we create Users class as model
2. //Step 1

@GetMapping("/all")

public List<Users> getAll() {

Users users1 = new Users("Peter",2300L);

Users users2 = new Users("Sam", 2400L);

return Arrays.asList(users1, users2);

}

This endpoint returns list of users and it dosent return any HATEOAS related data.

1. Inorder to returns HATEOAS specific data, we create another REST endpoint exactly similar one

@GetMapping("/hateoas/all")

public List<Users> getAll() {

Users users1 = new Users("Peter",2300L);

Users users2 = new Users("Sam", 2400L);

return Arrays.asList(users1, users2);

}

Which returns same list of users, but we need to create a link. In order to do that Users class has to extend ResourceSupport class which helps as to identify that this class needs to have HATEOAS link.

1. Now we need to add a link by, inorder to create Link we need to use ControllerLinkBuilder which link to controller class. We can add url in link using slash so name will be in url and specify as self url

@GetMapping(value = "/hateoas/all", produces = MediaTypes.***HAL\_JSON\_VALUE***)

**public** List<Users> getHateOASAll() {

Users users1 = **new** Users("Peter", 2300L);

Link link = ControllerLinkBuilder.*linkTo*(UsersResource.**class**)

.slash(users1.getName()).withSelfRel();

users1.add(link);

Users users2 = **new** Users("Sam", 2400L);

users2.add(link);

**return** Arrays.*asList*(users1, users2);

}

This will return list of users which will have hyperlinks.

1. Run the application
2. Go to browser and launch <http://localhost:8002/rest/users/all>

* Which returns the list of users but links wont be created, but by default it creates since we use ResourceSupport class

1. But now if we use hateoas url it will create the links

Go to browser and launch <http://localhost:8002/rest/users/hateoas/all>

* Now it will create the links, if we notice we added name in the link so we can see name in url as href
* If the client is using this particular url, he dosent have to know what url we need to use for accessing Peter. Instead he can hit /all and get hypermedia from the response itself
* If we see facebook graph API, twitter as well uses the same concept

1. To create multiple links, we create another link with relation with salary

**SPRING CLOUD SLEUTH AND ZIPKIN**

Log tracing is important from the perspective of log that we generate and analyzing the time information. Spring cloud sleuth and zipkin in combination provides a good way to analysis the log and timing information.

Consider a real life problem, suppose there is a bank ABC and we as a user perform some transaction logging and u find some issue where u cant perform the transaction.

So we contact the support people who are taking care of website, on seeing the issues they are taking the things and if they are not understanding and not able to resolve the issue, they go down the line to maintenance team. Maintenance team sees the issue and they also not able to resolve the issue and they further go down to QA team which performs actual testing of this website.

Quality team also analysis but unable to find the root cause, so QA team further goes down to developer who developed the website. Developer also not understand the things from the generated error messages. When ur website is running on the backend it generates the logs, so developer first point of contact to analysis the issue is logs that generated, from that we can get more information that is not appearing on the screen. That is important of using logs.

Consider we have different microservice, so if request is going from one microservice to another and fron that service to another, if some error happens, so we should have some ID in order to track. Spring Cloud sleuth is the library that gives the facility to add an ID, it is going to track the request from starting to ending. Zipkin is a UI which takes input from spring cloud sleuth and shows the request in proper user interface. It also provides info about how much time taking from one request to another request at each point of contact.

**SPRING CLOUD SLEUTH**

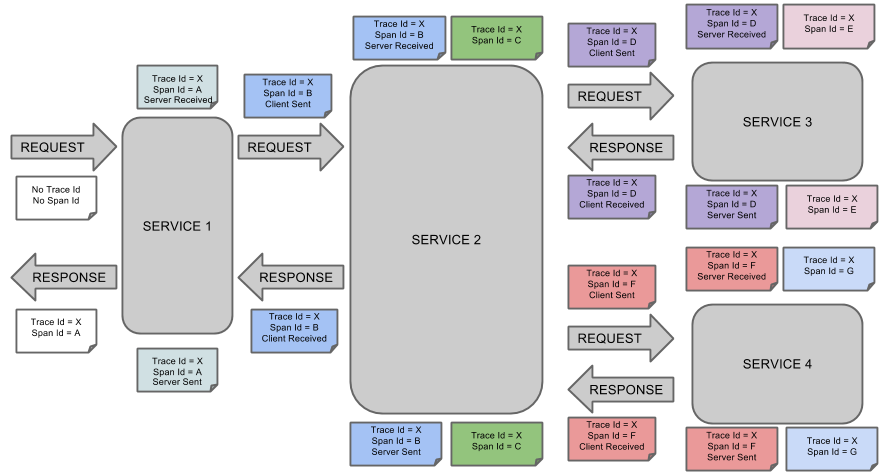
1. It implements a distributed tracing solution for Spring cloud

**Terminology**

1. Span

It is basic unit of work which has unique 64 bit ID. It also have metadata to keep the information like timing

1. Trace – A set of spans forming a tree like structure



**We have** 4 services, now the user is generating some request to Service1, initially it have no span and trace id. But once it process it creates span id A and trace id X. When it start processing the request, it got its span id.

Now the request travel to Service2, so when it leaves Service1 trace id is same, but span id changes to B because it takes care of different service.

Inside Service2, we have trace id as X, span id as B and server received because Service1 is processed something and now further it is going to travel so trace id as X and span id as C.

Inside Service2, it is going to perform 2 task, first task it has spanid B and for second task it has span id C. Once the task is finished it forward the request to next.

Now it goes to Service3, so span id changes to D because new work is coming and trace id is same because the same request is travelling. Inside Service3, span id is E because it is going to perform some action individually.

Now it generated some response and going back to user, so trace id X and span id E because it is same task, now it comes to first task of Service3 with trace id X and same span id D that it was performing.

Now the request come to Service4 with trace id X and span id F and it has two task, for the first task it has trace id X, span id F and for second task it has trace id X and span id G since it is new task. Now it is giving response back with same span id G and span id F.

Now the response comes to Service2 with same span id C and span id B and again it comes to Service1 with same span id A and finally response reach the user.

**STEPS:**

Consider we have REST application like Doctor portal, Doctor service, Patient service, Disease service and notification service with web and sleuth dependency. From Doctor portal we will call Doctor, Patient and Disease service, all these service are connect with notification service.

1. DoctorPortal-Sleuth with Web and Sleuth dependency
2. Create MainController.java

RestTemplateBuilder provided by Spring Boot when it is starting which creates the default instance of RestTemplateBuilder.

Using RestTemplateBuilder, we create RestTemplate which is used to call another REST services which running on port 9081 which has allDoctors endpoint and takes input parameter as hospital.

Like that we configured for Diseases and Patients also

1. DoctorService-Sleuth

Here the endpoint is /allDoctors with hospital as parameter. In DoctorServiceAPIImpl, with rest template we will call NotificationService

1. NotificationService-Sleuth

Here we create notification service

1. Run DoctorPortal-Sleuth, DoctorService-Sleuth, NotificationService-Sleuth
2. From DoctorPortal it calls Doctorservice and from their we call NotificationService
3. Run in browser, <http://localhost:7081/doctors> -- It will display List of doctors in Apollo
4. Now we see log info in DoctorsPortal, while printing logs Sleuth works

**INFO [doctor-portal,d6c67b1c5151ce6a,d6c67b1c5151ce6a,true] 16388 --- [nio-7081-exec-1] com.pack.MainController : Going to call doctors service**

doctor-portal – it is the appl that generates the log and it comes from spring application name is yml file.

d6c67b1c5151ce6a – trace id which is same throughout the distributed system

d6c67b1c5151ce6a – span id which is same as trace id because it is the place where it originated

false – if logs generated by application being imported to any third party like Jenkins, so if u import ur logs to any third party then it is true

1. Now see doctorservice logs, we can see the same trace id but different span id
2. Run patients and diseases and we can see the same trace id and different span id.
3. Now stop NotificationService and run <http://localhost:7081/doctors>, now it will show log message on error level

**ZIPKIN**

1. **ZIPKIN** is a distributed tracing system.
2. Gathering timing data
3. Helps in troubleshooting latency problems(which service is taking so much time) in service architecture

**Features**

1. Lookup of data logs
2. Storage of logs – The logs generated but different services S1,S2,S3 can be stored in Zipkin, if service is restarted later time we can see the past data.
3. We can query the data using trace id generated by sleuth, we can query based on service name (ie) the logs generated by particular service, even we can query by operation name, tags and duration
4. Dependency diagram – consider we S1,S2,S3,S4 service and we have request that connecting S1, S2 and S3. So in dependency diagram we can see how many service are involved in particular time.

**How to get data ?**

1. Applications need to be instrumented to report trace data to Zipkin, we should have some thing monitor the requestion going from one service to another for that we use Tracer or instrumentation library like Spring cloud Sleuth

**Installation**

1. Download Zipkin from zipkin.io/pages/quickstart --- click latest release and get the zipkin-server-2.19.2.jar file
2. Execute the jar file using >java –jar “filename”
3. Run Zipkin in browser as, <http://localhost:9411>

* Which starts Zipkin UI

Previously 5 Microservice is ready and zipkin is running, now we combine Zipkin into Sleuth.

1. Now we need one dependency **Zipkin-client,** previously we added Sleuth dependency. But one important point is Sleuth comes embedded inside Zipkin, so we change sleuth to zipkin in all pom.xml file
2. For configuration, we need to configure in application.properties file

sleuth:

sampler:

rate: 10 (default value)

Rate – for low traffic endpoints as it allows you surge protection. The point for sampler rate 10% is when u export all ur logs that is generated by application to Zipkin, generally zipkin has internal storage so that storage is very huge so we do not have any control of how many number of request ur application is going to have (ie) why this sampler rate is going to have control on that how much request, how much log info u export to zipkin. 10 means by default 10% of request export to zipkins

For testing purpose, all the request we want to export to zipkins so we give 100 as rate

sleuth:

sampler:

rate: 10

in all properties files.

1. Now we start all the application
2. Run endpoint in browser, <http://locahost:7081/doctors>

Now u can see the log info in the console with trace id, span id and also Boolean value to true because now we are export the log info to zipkin

1. In order to make my application know about zipkin url, we have configure url in doctor portal properties file as

Zipkin:

Base-url: <http://localhost:9411>

1. Now go to Zipkins url, we can see all service names available
2. Click Find traces, we can see trace information
3. If we click further we can how much time it is taking to execute
4. Now click each service and see detail information

Client Address: In real time, we may have more than 1 instance for single service, individual instance that creates log that IP address will be coming

1. Finally we have dependency diagram which display the interaction of each service
2. Now we hit patients end point <http://locahost:7081/patients> and refresh the Zipkin page and see the changes
3. Now stop the NotificationService and hit the doctor service and check the error info in Zipkin

**SAMPLING IN ZIPKIN**

Spring Cloud Sleuth has a Sampler strategy that you can implement to take control of the sampling algorithm. If you are exporting span data to Zipkin or Spring Cloud Stream, there is also an AlwaysSampler that exports everything and a PercentageBasedSampler that samples a fixed fraction of spans. he PercentageBasedSampler is the default if you are using spring-cloud-sleuth-zipkin or spring-cloud-sleuth-stream. You can configure the exports using spring.sleuth.sampler.percentage. The passed value needs to be a double from 0.0 to 1.0 so it’s not a percentage. For backwards compatibility reasons we’re not changing the property name.

**Spring.sleuth.sampler.percentage=1.0 //This will display only partly in zipkin portal**

@Bean

**public** Sampler defaultSampler() {

**return** Sampler.***ALWAYS\_SAMPLE***;

}

But this will display fully

**ACTUATORS – Refer SpringBoot-SimpleREST**

Actuators are concept created by Spring Boot where u can monitor ur application for any production support related information. For example in general what we do in production so we monitor the process, we go and check the health, we check whether the process is up and running, we check whether database connection is up and running, where database connection is successfully inside the process and heap space available in particular JVM. So all these are taken care by concept called Actuator.

Useful when u want to monitor ur production running application, when u want to debug certain things even in development phase. Actuator exposes certain REST endpoints and each endpoint has purpose and gives certain metrics.

1. We have basic Spring Boot application, refer SpringBoot2-SimpleRest project
2. To configure Actuator, configure the dependency

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId

</dependency>

1. In order to access the actuator, use <http://localhost:8111(project> portno)/actuator
2. If u want to access all actuator configure in application.properties files

**management.endpoints.web.exposure.include=\***

1. <http://localhost:8080/actuator/beans> -- will expose all the beans which are recognized by Spring Application Context
2. <http://localhost:8080/actuator/caches> -- which expose the information about cache
3. <http://localhost:8080/actuator/health> -- just check whether the service is up or not
4. <http://localhost:8080/actuator/configprops> -- displays all properties that used by spring boot or these are default properties that is configured by spring boot. Eg:port
5. <http://localhost:8080/actuator/env> -- exposes all the information related to environment, for ex: Java run time env, class path, library path
6. <http://localhost:8080/actuator/loggers> -- which contain all logger info
7. <http://localhost:8080/actuator/httptrace> -- give all the recent request to the server
8. <http://localhost:8080/actuator/mappings> -- when u debug http end points

**SPRING CLOUD OPENFEIGN**

It is used to call third party REST services, used to consume third party microservice, it is declarative style of calling microservice. Previously we have Spring boot REST template also there to invoke microservice.

Feign client is a declarative HTTP client developed by Netflix, when we are performing load balancing in Spring cloud and when we are integration Spring cloud eureka registry to invoke external service we should not use REST template in that case we can use Feign client, where it internally balance the load.

The advantage of Feign client over REST template, if we are using feign client we no need to write unit test case for REST client because there is no code only developer need to declare and annotate an interface, where actual implementation will be provided at runtime.

What is Spring cloud OpenFiegn?

1. Declarative REST client: Feign creates a dynamic implementation of an interface decorated with JAX-RS or Spring MVC annotations – it means we have an interface annotated with feign annotation and rest of things will automatically implement by Spring boot feign libraries itself by detecting those annotations
2. @EnableFeignClients – when u write this annotation on main application class, it scans for interface that declare as feign clients using

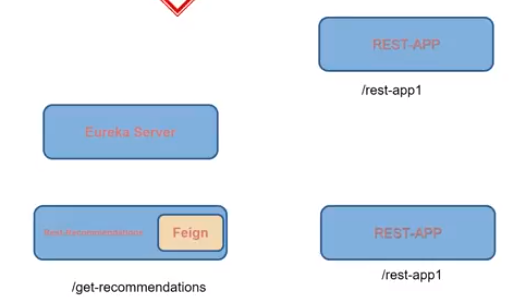
FeignClient @FeignClient;

So when u have @EnableFeignClients, it is going to scan all of ur project directory and look for @FeignClient annotation. If there any interface which is annotated with @FeignClient, it will consider that interface to implement internally as a service which will call third party.

1. @FeignClient – Annotation for interface declaring that a REST client with that interface should be created – which means it is going to call third party REST services

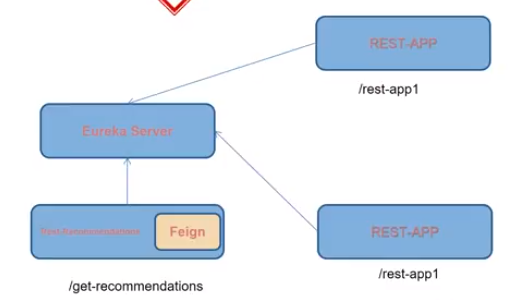
Program- SpringFeignDemo project

1. We have a cloud environment containing 4 microservices up and running. First is Eurekaserver which is microservice registry which maintains list of available microservice up and running

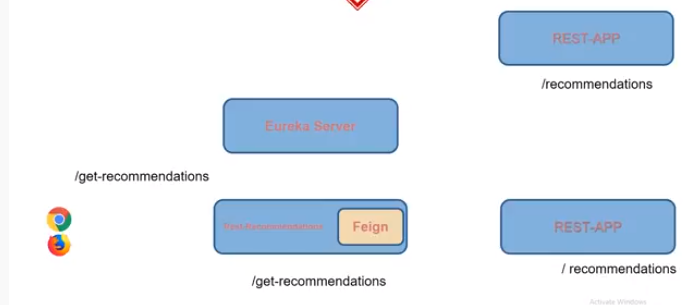
****

We have another microservice called Rest-Recommendations which has request mapped to /get-recommendations, we can see Feign inside because in an microservice architecture, one microservcie can invoke other microservice so indeed one microservice is depend on other microservice so Rest-recommendation microservice depend on REST-APP microservice.

Next we have two instance of same spring boot microservice appl called REST-APP. Now Rest-Recommendation microservice communicate to REST-APP through HTTP request , but since we have two instance to which one we are going to communicate, this decision is done by Feign client.



Eureka clients are registering themselves against Eureka server,so eureka knows which microservice are available in cloud environment. Now feign will fetch that list from Eureka server because feign has to know which microservice are available.



Let assume that one of client make a request to /get-recommendations, it perform that request through Rest-recommendation application, but this application depends on all microservcies specifically it depends on REST-APP microservice. Feign is REST client or REST consumer, we can use it to actually communicate to microservice that depending on (ie) to REST-APP microservice. But we have two REST-APP so which one we are going to communicate so Feign will use behind scenes Ribbon client side load balancer.

So Ribbon which used inside Feign itself has picked the first instance, so Feign as REST client will perform HTTP request to REST-APP, now that instance will respond something that will received by Rest-recommendations and it process with that response. But Rest-recommendation microservice has to send something back to user.

Now another user perform another request to /get-recommendations and Rest-recommendation microservice depends on REST-APP so we make HTTP request using feign. Actually no need to use Feign client we can use HTTP client using REST template but Rest tenmplate dosent suit for load balancing without any specific annotation.

Normally REST template will do load balancing with specific annotation defined by Hysterix. Feign is REST client and Rest-recommendation talk to REST-APP, but if we have multiple instances we have to apart from the request, we have to choose which instance to target that load balancing task is performed by Ribbon which is used by feign behind scene

1. Create Feign-EurekaServer with EurekaServer dependency and @EnableEurekaServer annotation
2. Create Feign-RestApp with web,eureka client dependency with @EnableDiscoveryClient
3. Create FeignClient with web,eureka client,open feign with @EnableDiscoveryClient, @EnableFeignClients
4. Run all application
5. Run in browser, <http://localhost:8001/rest-version>

When u trigger /rest-version, we are returning the result of call to getVersion() from the RestClient which is an interface, and retrieving the instance of restClient through Constructor Dependency injection.

RestClient interface is Feign client and indicate the target API is rest-app(it is name of microservice that u r trying to reach). Feign client is Rest consumer which consume another REST endpoint and it is a microservice called rest-app.

We are trying to target some handler inside Rest-app microservice and that handler is mapped to /version and it returns String

Example 2: Refer FeignEurekaServer, FeignEurekaClient-1, FeignEurekaClient-2, FeignEurekaClient-3

[4 - Spring Boot Microservices : How to integrate Feign Client using Spring Boot? - YouTube](https://www.youtube.com/watch?v=MehQOXGHhV0)

1. Run all application
2. Check eureka from <http://localhost:8761>
3. Run <http://localhost:8001/> - Now from EurekaClient-1 calls EurekaClient-2 and from client 2 calls EurekaClient-3 using RestTemplate
4. Now run <http://localhost:8001/> using FeignClient, prints same result

Feign Client

1. Feign client is declarative and just an annotation for an interface
2. Wellsuited for restful approach(we can declare all CRUD operations in a single feign client interface). We have service exposed three endpoints and we need to consume all those endpoints, in that we create an interface and add method that need to consume and add some annotation
3. Fault tolerance and client side load balancing is in build

RestTemplate

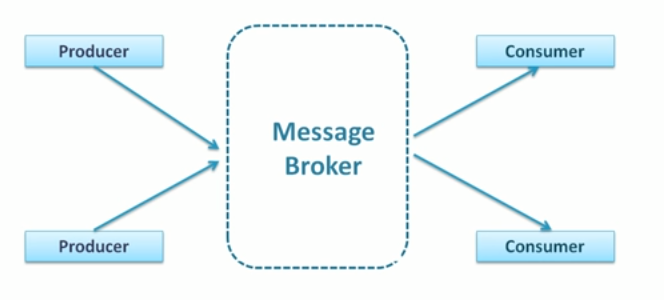
1. We ends up with writing many boiler plate code to make call from one service to another and we have to write many unit test cases. In Feign client it just interface no need to write test cases
2. Not suited for CRUD operation, we have to make calls to achieve it
3. Fault tolerance and load balancing is not in build

**APACHE KAFKA**

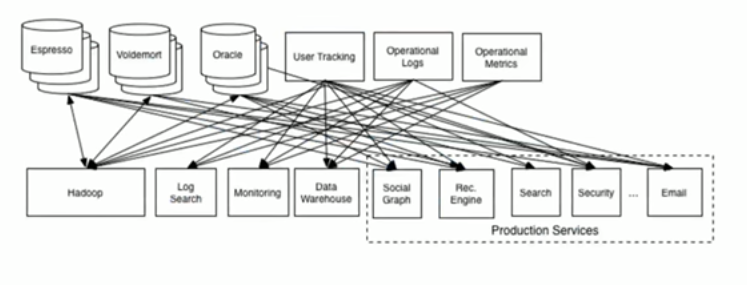
Kafka is open source project, originally developed at linkedin and later open source since 2011. Since then it has evolved and established itself as very popular tool for building real time data pipelines.

Kafka is a distributed streaming platform and similar to enterprise messaging system.

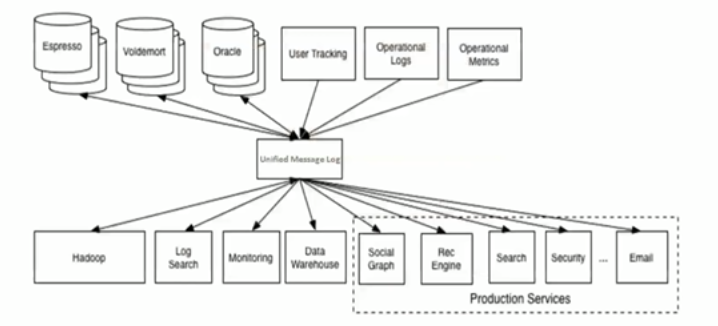
In a typical messaging system there are 3 components like producer, broker and a consumer. The producer are the client appl that sent some messages, the broker receives messages from publishers and stores these message. Consumers will read the messages from brokers.



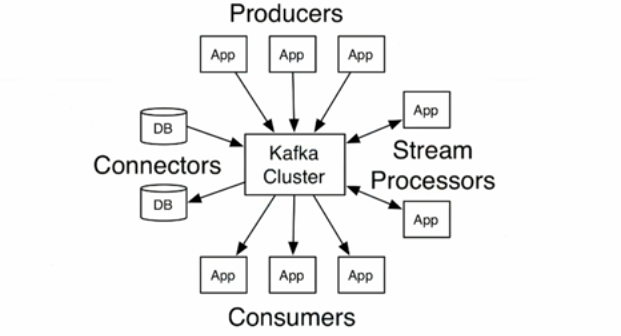
It is very simple but now we see data integration problem in large organization.



It shows data integration in large enterprise. There are many source system and multiple destination system and given a task to create data pipeline to move data among these systems. For growing organization, the number of source and destination system keep getting bigger and bigger, finally the data pipeline become mesh and some part of this pipeline keep breaking every day. So if we use any messaging system for solving this type of integration problem, the solution may be neater and cleaner like this



That the idea discovered by the team in LinkedIn, then they started evaluating existing messaging system but none of them meet their criteria to support the decided throughput and scale. Finally they end up creating Kafka, Kafka is highly scalable and fault tolerant enterprise messaging system.



Producer application sending messages to Kafka cluster which is nothing but a bunch of brokers running in a group of computers. They take message records from producers and store it in kafka message log. At the bottom these are consumer appl, they read messages from kafka clusters, process it and do whatever they want to do may be send them to Hadoop or Cassandra or hbase or pushing it back again into Kafka for someone else to read these modified or transform messages.

Kafka is so powerful regarding throughput and scalability so that it allows you to handle a continuous stream of messages , so if you can plugin some stream processing framework to Kafka, it could be ur back bone infrastructure to create a real time stream processing application. These are stream processing application which reads continuous streams of data from Kafka, process them and then either store in Kafka or send them directly to other systems.

Next is Kafka connectors, they are ready to use connectors to import data from database to Kafka or export data from Kafka to database.

Features:

1. Kafka used as an enterprise messaging system, it dosent mean a traditional messaging system, you can use it to simplify complex data pipelines that made up of vast number of consumers and producers.
2. Used for stream processing platform, there are two parts of stream processing, a stream and processing framework. Kafka gives u a stream and u can plugin a processing framework.
3. It also provides connectors to import and export bulk data from databases and other systems

Kafka – Core Concepts

The main terminology used in Kafka are

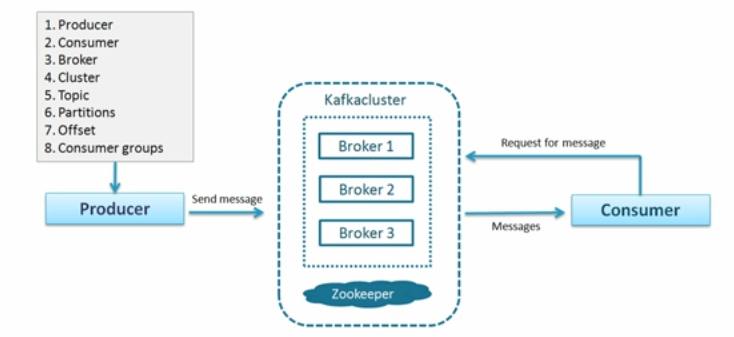
1. Producer – An application that sends messages or data or message record to Kafka. Ultimately it is small to medium sized piece of data. The message may have different meaning for us but for Kafka it is simple array of bytes.

For example, if I want to send a file to Kafka, we will create a producer application and send each line of file as a message. In this case message is one line of text but for Kafka it is array of bytes. Similarly, if we want to send all the records from the table, we will send each row as a message or if we want to send a result of query we will create a producer application, fire a query against the database, collect the result and start sending each row as message. So while working with Kafka if you want to send some data you have to create a producer application. It is very unlikely we get readymade producer application that fits our purpose.

1. Consumer – An application that receives the data. Producer don’t send data to recipient address, they send it to Kafka server and anyone who is interest in that data can take it from Kafka server. So an application that request data from Kafka server is a consumer, they request data from any producer provided they have permission to read it.

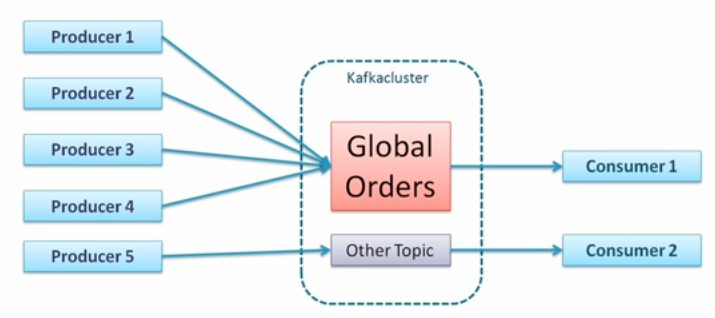
If you want to read the file sent by the producer, we will create a consumer application then we request Kafka for the data. The Kafka server will send me some messages in form of line and client application receives the line from Kafka server. Consumer will process them and again request for some more messages. The client keeps requesting data from Kafka and Kafka will give message records as long as new messages are coming from the producer.

1. Broker – Broker is Kafka server, the producer and consumer don’t interact directly, they use Kafka server as an agent or broker to exchange messages.
2. Cluster – A group of computers sharing workload for a common purpose. It is simply a group of computers each executing one instance of Kafka broker.



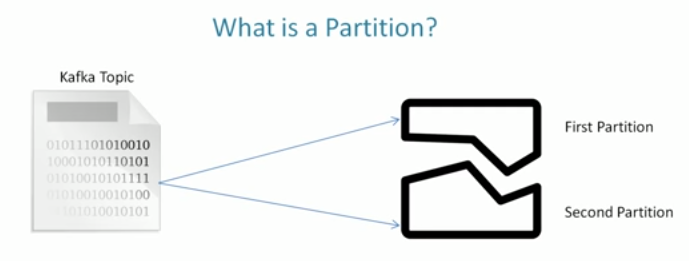
1. Topic -- Producer will send data to Kafka broker, then consumers can ask for data from Kafka broker, but which data. Topic is a arbitrary name given to data set or unique name for data stream

For example, we create a topic called Global Orders and every point of sale has producer. Each of them send their order detail as a message to single topic called Global orders and subscriber interest in orders can subscribe to same topic.

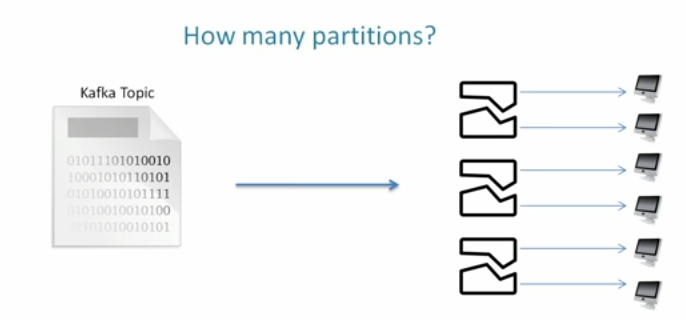


1. Partition – Broker will store the data for topic, it may be larger than storage of single computer in that case broker may have a challenge in storing that data.

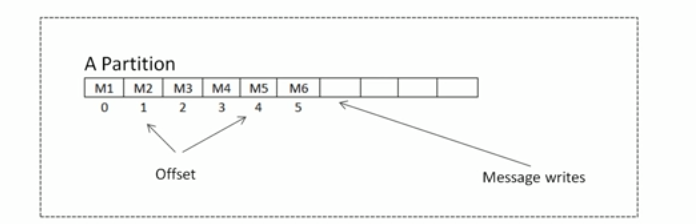
One solution is to break it into one or more parts and distribute to multiple computers. Kafka is distributed system that runs on cluster of computers. So kafka can break the topic into partition and store 1 partition on 1 computer.



We may think how kafka will decide on number of partitions (ie) some topics may be large some be small so how kafka knows 100 partition or 10 partitions. The answer is Kafka dosent take that decision, we have to take the decision. When we create topic we take that decision and Kafka broker will create that many partition for ur topic. But every topic sits on single computer so do some estimation and math to calculate the partition.



1. Offset – It is sequence id to a message in a partition. This id is assigned as message arrives in the partition and this id once assigned never changed. This sequencing means that Kafka stores the messages in the order of arrival within the partition. The first message gets offset 0 and secons message gets offset 1 and so on. But no global offset across partition. Offsets are local to partitions.



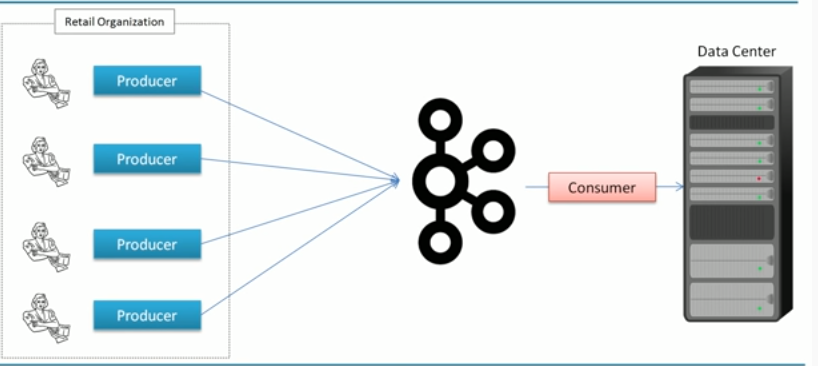
So if u want to locate the message then we should know topic name, partition number and offset.

1. Consumer Group – It is a group of consumer to share the work.

There is one large task and want to divide among multiple people, so u create a group and members of the same group share the work.

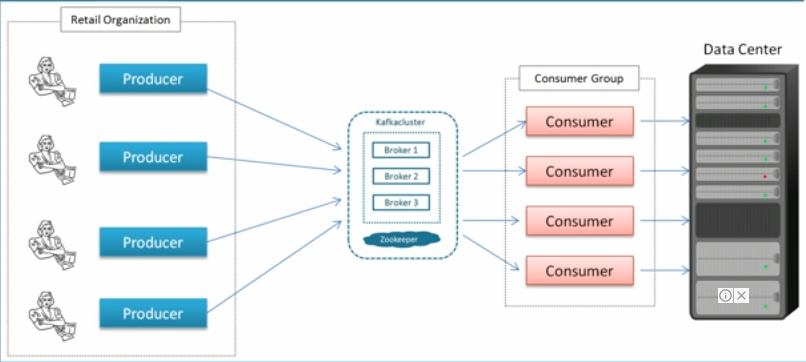
For example, we have a retail organization, in every store we have few billing counters and you want to bring all invoices from every billing center to data center. Kafka is a good solution to transport data from billing location to data centers.

First thing we decide to create producer in very billing location, these producers will send bill as messages to the Kafka topic. The next thing is to create a consumer, the consumer will read data from Kafka topic and write them to data center, it is perfect solution but a small problem.



Think scalability, we have 100 producers pushing data to single topic, how you will handle that volume and velocity. So u decided to create large Kafka cluster and partition ur topic. So ur topics is partitioned and distributed across clusters. So brokers are sharing the workload to receive and store data. From the source side we have several producers and brokers to share the work load. But in destination side we have only 1 consumer, there comes the consumer group, u create consumer group and start executing multiple consumers and tell them to divide the work.

How we divide the work? We have 600 partition, starting 100 consumer so each consumer will take 6 partition. If not we start some more consumer in same group, we can upto 600 consumer with 1 partition for each consumer. The maximum number of consumer in group is total number of partition u have on topic. Kafka dosent allow more than 2 consumer to read from same partition simulanteously.



**Installing Kafka**

1. Download Kafka from <https://kafka.apache.org/downloads>
2. Download Scala 2.12  - [kafka\_2.12-2.0.0.tgz](https://archive.apache.org/dist/kafka/2.0.0/kafka_2.12-2.0.0.tgz) ([asc](https://archive.apache.org/dist/kafka/2.0.0/kafka_2.12-2.0.0.tgz.asc), [sha512](https://archive.apache.org/dist/kafka/2.0.0/kafka_2.12-2.0.0.tgz.sha512))
3. Extract the kafka
4. Create folder zookeeper\_data inside Kafka folder
5. Go to config folder – edit Zookeeper.properties file with

dataDir=C:\Softwares\kafka\_2.12-2.0.0\zookeeper\_data

1. Create folder kafka-logs inside Kafka
2. Configure this inside server.properties file of Kafka

log.dirs=C:\Softwares\kafka\_2.12-2.0.0\kafka-logs

1. Configure few more property in server.properties file

offsets.topic.num.partitions=1

offsets.topic.replication.factor=1

transaction.state.log.replication.factor=1

transaction.state.log.min.isr=1

min.insync.replicas=1

default.replication.factor=1

port = 9092

advertised.host.name = localhost

1. Configure windows folder inside bin of Kafka to environment variable

C:\Softwares\kafka\_2.12-2.0.0\bin\windows in path

1. Kafka needs zookeeper, first we start zookeeper

C:\Softwares\kafka\_2.12-2.0.0\config>**zookeeper-server-start.bat zookeeper.properties**

It takes one parameter zookeeper.properties, and runs on port no 2181.

Zookeeper is another open source project that came from Hadoop project. It is used to provide some coordination service for a distributed system. Since Kafka is distributed system and we have multiple brokers so we need a system to coordinate various things among these brokers. So we need zookeeper.

1. In new command prompt start Kafka broker

C:\Softwares\kafka\_2.12-2.0.0\config>kafka-server-start.bat server.properties

It takes configuration information from server.properties

1. In another new command prompt, ask for list of brokers

C:\Users\senthil.kumart>zookeeper-shell.bat localhost:2181 ls /brokers/ids

We have 1 broker with id 0

1. Create kafka topic

C:\Users\senthil.kumart>kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test

It takes parameters like zookeeper info localhost and port

--create – to create a topic

--partitions – to give number of partition

test – topic name

1. Start the producer and send message

C:\Users\senthil.kumart>kafka-console-producer.bat --broker-list localhost:9092 --topic test

>Hello world

> Welcome to Kafka

Ctl+c to terminate

To send a message to kafka, we need a broker address. We had a broker running locally in port 9092

1. Start the consumer to receive the message

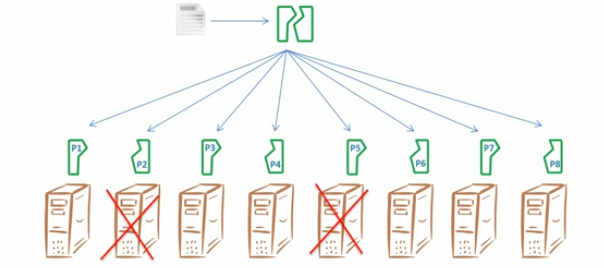
C:\Users\senthil.kumart>kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic test --from-beginning

Displays the message

--bootstrap-server have same meaning as broker-list

What is Fault tolerance?

Kafka is distributed system and it works on cluster of computers. Most of the time kafka will spread the data in partition over various system in cluster. If one or two system in the cluster fails what will happen to the data, will u be able to read it probably not, that’s the fault.



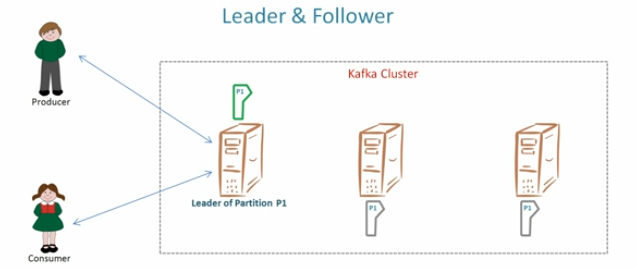
Fault tolerance is very common in distributed system, it means making the data available even in case of some failures. One solution is to make multiple copies of data and keep it on separate systems. So if u have 3 copies of partition and Kafka stores them on 3 different machines u should be able to avoid 2 failures. Since u have 3 copies on 3 different machine even if 2 of them fails, even we can read the data from 3rd system.

There is particular term to create multiple copies called as replication factor. If we say replication factor as 3 it means we maintain 3 copies of partition, if it is 2 means we have 2 copies of partition.

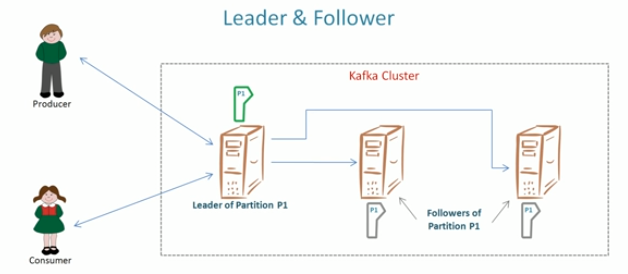
**So kafka implements fault tolerance by applying replication to the partititon. We can define replication factor at topic level, so we don’t set a replication factor for partition instead we set it for topic and it applies to all partition within the topic.**

How Kafka makes these copies?

Kafka implements Leader & Follower model, so for every partition one broker is elected as leader and the leader takes care all client interaction (ie) when producer send some data it connects to leader and start sending data, it is leaders responsibility to receive the message, store it in local disk and send back the acknowledgement to the producer. Similarly, when consumer is willing to read data, it sends the request to leader and it is leaders responsibility to send requested data back to consumer. For every partition we have leader and the leader takes care of all request and response.



We haven’t made any copy, that where the followers come into play, so if we create a topic with replication factor to set, a leader of topic is already maintaining the first copy, we need 2 more copies so kafka will identify two more brokers as followers to make those 2 copies. These followers copy the data from the leaders.



To demonstrate 1leader and 2 followers, we need 3 node Kafka cluster. In ideal cluster we install 1 broker on 1 computer, for demo we can start multiple brokers on single machine.

Now we start 3 brokers before that we make copy of broker config file and modify.

1. cp config/server.properties config/server-1.properties
2. cp config/server.properties config/server-2.properties

Now we have 3 properties file

1. Now we want to change some configuration in server-1.properties file

**broker.id – it is unique id for broker, the default value for first broker is 0 so we change to 1 for server-1 and 2 for server-2**

Next property is broker port where broker will bind itself, broker will use this port no to communicate with producer and consumer

For server-1

port = 9093

advertised.host.name = localhost

For server-2

port = 9094

advertised.host.name = localhost

**if we have different machine there is no need to change the ports, in single machine we need to change it.**

Next property is log.dirs which is main data directory of broker, we don’t want all of the broker to write into same directory. So create a new directory for logs and give that path

log.dirs=C:\Softwares\kafka\_2.12-2.0.0\kafka-logs1

log.dirs=C:\Softwares\kafka\_2.12-2.0.0\kafka-logs2

1. Start zookeeper server

C:\Softwares\kafka\_2.12-2.0.0\config>zookeeper-server-start.bat zookeeper.properties

1. Now start all 3 servers in separate command prompt

C:\Softwares\kafka\_2.12-2.0.0\config>kafka-server-start.bat server.properties

C:\Softwares\kafka\_2.12-2.0.0\config>kafka-server-start.bat server-1.properties

C:\Softwares\kafka\_2.12-2.0.0\config>kafka-server-start.bat server-2.properties

Now we have 3 node Kafka cluster up and running

1. We created a topic with replication factor 3 and show the leader and follower for each partition.
2. Create a topic

C:\Softwares\kafka\_2.12-2.0.0\config>kafka-topics.bat --zookeeper localhost:2181 --create --topic TestTopicXYZ --partitions 2 --replication-factor 3

1. C:\Softwares\kafka\_2.12-2.0.0\config>kafka-topics.bat --zookeeper localhost:2181 --describe --topic TestTopicXYZ

* Describe command will show topic name and number of partition in this topic and replication factor for the topic
* Since we have two partition on topic, it displays two row, one for each partition
* We have Leader 1 for partition 0 which means broker 1 will store and maintain the first copy of partition and fulfill all client request for this partition. Similarly Leader 2 for partition 1
* Next is replicas, for the partition 0 we have 3 copies 1,2,0 (ie) broker 1 maintains 1st copy, broker 2 maintains 2nd copy and broker 0 maintains 3rd copy, so Broker 1 is leader and broker 2 and broker 0 are followers.
* Isr is list of insync replica, you might have 3 copy but one of them does not link with leader so Isr shows list of replicas that are sync with leaders

**Broker Configuration**

Previously we created multiple node cluster on a single machine, we also saw some configuration like port, broker.id, log.dirs

Apache Kafka is highly configurable system, and it provides many configurable parameters, most of them have default values, we can some key broker configuration.

1.zookeeper.connect – This parameter takes zookeeper connection string, the connection string is simply a hostname with portno. This parameter is also necessary to form a cluster (ie) all brokers are running on different system, how do they about each other, if they don’t know about each other they are not part of cluster. So the zookeeper is connecting link among all brokers to form the cluster.

2. delete.topic.enable – If u want to delete a topic, we can use topic management tool(ie) cmd promt to delete the topic. But by default, deleting the topic is not allowed because default value for this parameter is false. It is protection in production env, but if u want to delete the topic in development and testing env, then set the parameter to true.

3. auto.create.topics.enable – If a producer send a message to non-existent topic, Kafka will create the topic automatically and accept the data. This behavior is suitable for development env but in production env we want to implement more control approach, so we set this parameter to false and Kafks will stop creating topic automatically. We can create topic manually using topic management tool and no one will able to send data to non-existent topic.

4. default.replication.factor

5. num.partitions

Both default value is 1 and they are effective when u have auto created topics. If kafka is creating topic automatically then new topic will have 1 partition and 1 copy. If we want other values, we can change default settings

6.log.retention.ms

7.log.retention.bytes

Whatever data we send to Kafka, it is not retained by kafka forever, Kafka is not database, u wont send data to Kafka for storage so that u can query it later. It is message broker, it should deliver the data to consumer and then clean it up.

Kafka gives 2 options to configure the retention period, the default option is retention by time and default retention period is 7 day. In this case, Kafka will clean up all the messages older than 7 day.

If u want to change the duration, you can specify value for log.retention.ms configuration. Kafka gives another option to define the retention period, specified as size in log.retention.bytes for partition size (ie) log.retention.bytes=1GB, Kafka will trigger a clean up activity when partition size reaches to 1GB.

**Producer API (Refer SimpleProducer.java)**

We can use Kafka to solve complex data integration problem, use to create series of validation, transformation and build complex data pipeline, use to record information for later consumption for example playing click history, use it to log transaction and create application to responsding real time, use to collect data from mobile phone, smart appliance, and sensors in IOT application.

If u look at any of these use cases, it is all about asynchronous communication among applications. So whatever we do with Kafka we must have producer that will send data to Kafka. You need to create a producer for ur appl to send data to kafka. The most common method to create Kafka producer is using Kafka API.

SimpleProducer.java

1. In this example, we want to send a String message to Kafka, it is simple java string, most of time Kafka message are key value pairs. So with every message you can send key, however the key is not mandatory, u can send message without key as well. In this example we send key and value and send to topic called SimpleProducerTopic

String topicName = "SimpleProducerTopic";

String key = "Key1";

String value = "Value-1";

1. Create object for KafkaProducer

**Producer<String, String> producer = new KafkaProducer <>(props);**

To create an object, u need a property object with atleast 3 mandatory configuration. These core configuration are bootstrap server, key serializer and value serializer. Bootstrap server is a list of Kafka servers, the producer object use this list to connect to Kafka cluster, you can specify one or more brokers in the list. The recommendation to provide 2 brokers, if one broker is down the producer can connect to other broker from the list.

props.put("bootstrap.servers", "localhost:9092,localhost:9093");

The next two property is about kafka message, it is just array of bytes, in this example we send String key and String value but kafka accepts only array of bytes, so we need a class to convert our message key and value into array of bytes. The activity of converting java objects into array of bytes is called serialization, so this two property is use to specify the appropriate Serializer class for key and value (ie) StringSerializer.class

Kafka also provides other Serializer like IntSerializer(used to send integer key),DoubleSerializer,JsonSerializer

We define 3 info and package them into Properties object, then we pass properties object to KafkaProducer constructor and instantiate the Producer

1. Now we have producer instantiated, now we want to send messages so we create ProducerRecord object. The ProducerRecord object requires 3 things like topic name,key and message value, we pass these things to ProducerRecord constructor and instantiate producerrecord object

This object is our message and it should be given to producer so that producer can send it to Kafka broker

**ProducerRecord<String, String> record = new ProducerRecord<>(topicName,key,value);**

1. To send the record to producer

**producer.send(record);**

We make a call to send method on Producer object and handover the record object. Now it is producer responsibility to deliver this message to the broker.

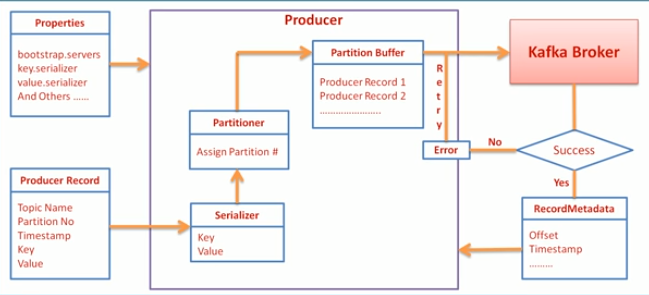
1. After sending all messages, we need to close the producer object, it is necessary to clean up all necessary resources that producer may be using in
2. Run zookeeper
3. Run kafka server
4. Run SimpleProducer.java
5. Run kafka consumer

C:\Users\senthil.kumart>kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic SimpleProducerTopic --from-beginning

It will display the message value-1 from producer

How a message goes from client application to a broker through Producer ?

1. Create a java properties object and package all the producer configurations that we want to set. These settings include 3 mandatory configuration like bootstrap.servers, key.serializer and value.serializer which is used in previous example.
2. We create a ProducerRecord and package 5 things in a ProducerRecord object like topic name, partition no, timestamp, key and value. Partition no, timestamp and key are optional depending upon use case. This object is infact the message that we want to send to Kafka broker.
3. So we instantiate Producer object using properties object, then we send producer record to producer object. When the message is handed over to producer, following thing happens
4. The producer will apply serializer to serialize ur key and value (ie) converting key and value object to array of bytes
5. Then it will send record to partitioner. The partitioner choose the partition for the message, the default partitioner will use ur message key to determine the appropriate partition. If a message key is specified kafka will hash the key for getting the partition number. If u specify same key for multiple message, all of them go to same partition. If message key is not specified, the default partitioner will try to evenly distributed the message to all available partition for topic, it uses round robin algorithm.
6. Once we have partition number, the producer is ready to send message to broker but instead of sending message immediately, the producer keep the message in partition buffer. So the producer maintains in-memory buffer for each partition and sends the record in badges.
7. Finally, the producer will send badge of records to the broker. If the broker can receive and save the message, it will send an acknowledgement in form of record metadata object. If anything goes wrong, the producer receives the error.
8. Some errors may be recoverable with a retry, for example, suppose the leader of partition is down if we retry few milliseconds we may have a new leader elected. So in case of recoverable errors, the producer will retry sending the badge before it throws an exception. We can configure the number of retry and time between two retry using configuration. The producer will not attempt to retry if the error is not recoverable error.



**Callback and Acknowledgement**

Basically there are 3 approaches to send messages to Kafka

1. Fire and Forget

It is most simple approach, in this method we send a message to broker and don’t care if it was successfully received or not. The example we created earlier follow this approach.

Kafka is distributed system, it comes with inbuilt fault tolerance features, that makes Kafka highly available system, so most of the time ur message will reach to the broker. We also know that producer will automatically retry in case of recoverable errors, so the probability of losing the messages is less. There are many use cases where we dealing with huge volumes and losing small portion of record is not critical.

It is important in fire and forget method you may lose some messages. So don’t use this method when u not afford to lose any messages

1. Synchronous send

In this case, we send message and wait until we get the response. In the case of success, we get record metadata object and in the event of failure we get exception. We don’t care about success record metadata we always care about exception because we want to log errors for later analysis and appropriate action.

It is used when ur messages are critical and not afford to lose any messages. This approach will slow u down, it will limit ur throughput because we are waiting for every message to get and acknowledge.

Each message will take some time to deliver in network, so after every message we wait for network delay.

**SynchronousProducer.java –follow same steps as before program**

1. Create Producer properties and instantiate Producer object
2. Create ProducerRecord object
3. Handover the message to producer by making call to send(), but this time we want to get response and handle an exception, so we surround the code in try, catch and finally block.
4. send() returns Java Future object and we call get() on Future object. The get() will wait till we get success or exception. In case of success we get RecordMetadata object and printing partition number and offset number from RecordMetadata
5. In case of exception we print the stacktrace, but in production code u may want to log the message and exception detail for later analysis. We can have finally block to close producer object.
6. Start zookeeper
7. Start kafka server
8. Start spring boot application
9. Start kafka console to receive message
10. Asynchronous Send

In this method, we send a message and provide callback function to receive acknowledgement, we don’t wait for success and failures. The producer will callback our function with record metadata and exception object. So in this approach we keep sending messages as fast as u can without waiting for response and handle failures later as they come using callback function.

You have a limit of inflight messages, this limit is control by configuration parameter called max.in.flight.requests.per.connection, this parameter controls how many messages you can send to the server without receiving response, the default is 5 but u can increase the value.

AsynchronousProducer.java

1. The only difference is that we have another parameter for send(), it is callback object
2. If we want to create callback class, then it should implement Callaback interface, then we override onCompletion(). The producer will call this method after receiving an acknowledgement or an exception.
3. If exception object is not null we have failure or we have success

**CUSTOM SERIALIZER**

We already know we need appropriate serializers for key and values. Previously we are sending the String, kafka also provide int, long, double serializers, these dosent suit most of use case

If u coming from database background, u can think of topic like table and each message sent to topic as record, those record are not single string or number. Normally we have multiple columns in a record, when working with Kafka we need to able to send record of multiple columns.

If u coming from OOPS background, u see kafka message as object, normally these object will have multiple fields and methods. We should able to send these objects to Kafka as a message. Sending simple string in Kafka fulfil some requirement but in complex requirement we need to send some objects.

For example, Supplier object, invoice object. If u want to send some custom object or row like structure, you need to implement custom Serializer and Deserializer.

**Example: Supplier, SupplierProducer, SupplierConsumer, SupplierSerializer, SupplierDeserializer**

1. Create Supplier class, we will serialize supplier class and send supplier object as message to Kafka

* Defines supplierId, supplierName, supplierStartDate, constructor and getter method.
* Used to instantiate Supplier object and send as Kafka message

1. Create SupplierSerailizer.java, to convert supplier object into byte array

* It should implement Serializer interface with generic type as Supplier and override the implemented methods configure() – for initialization, serialize(), close() – for cleanup
* Kafka Producer will call these methods once, it will configure() when we instantiate the producer and call close() when we close the producer
* The main action in serialize() (ie) if data is null, it return null because nothing to serialize
* We convert suppliername and supplierdate into UTF8 bytes, then we allocate ByteBuffer and put everything into ByteBuffer, since we need to know length of suppliername and supplierdate string at the time of deserialization we encode their sizes into ByteBuffer and return Byte array

1. Create SupplierDeserializer.java, used to convert byte array to supplier object.

* We deserialize very field, create a new Supplier object and return it .
* On new requirement when we add new field in Supplier class, then we need to change Serializer and Deserializer, may be Producer and Consumer too. After making new changes we cant read ur earlier messages because u change the format and modify the code to read new format, that where generic Avro serializer is used

1. Create SupplierProducer.java, this producer will send supplier object as Kafka record

* Same like other Producer problem
* We add value.Serializer as SupplierSerializer and generic type to Supplier
* We send two messages using Synchronous send

1. Create SupplierConsumer.java, used to read supplier object from Kafka and print in console

* We created property object and set 3 mandatory properties bootstrap server(list of kafka broker used to connect to Kafka cluster), key.serializer and value.serializer.
* In producer we used key and value serializer, but in consumer we use only deserializer. Here we use StringSerializer for key and custom deserializer for value.
* The next property is group\_id, so we can specify your consumer group name as value of this property. You dont need to worry about creating the group, participating in a group, who is group coordinator and group leader all that is taken care by API using groupname which is String. Group\_id property is not mandatory but when you are not part of any group that means we starting independent consumer and code will read all of data of topic and no sharing of work and your consumer will need to read all data and process alone.
* Create KafkaConsumer object and subscribe to one or more topics, the method takes list of topics so we can subscribe to multiple topics at a time. Subscribing to a topic means we are informing Kafka broker that u want to read data for these topics.
* After subscribing u want to fetch some data and process them for that we use while loop. Poll() will return some messages, you process them and again fetch for more messages, the parameter to poll() is timeout if there is no data to pull so this value is specifies how quickly you want the poll method to return with or without data. When u call to poll() for first time from a consumer it finds a group coordinator joins the group, receives partition assignment and fetches some records from those partition.
* For loop process each message where consumer receives from Kafka broker and display supplier fields on the console

1. Start Zookeeper
2. Start Kafka Server
3. Run SupplierProducer.java
4. Run SupplierConsumer.java
5. Now producer info will displayed in consumer part
6. Keeping properties in a separate file is more flexible and you will have configuration outside the code and load property values from external java properties files.

Create SupplierConsumer.properties will all property information

**KAFKA- CUSTOM PARTITIONER**

We also saw that Kafka partitioner is responsible for deciding partition number for each message, the default partitioner follows this rules

* If a partition is specified in the record it will use it, if u already know which partition u want to send the data u will hardcode and use 1st rule
* If no partition is specified but a key is present choose the partition based on hash value of key, if u want ur data to be distributed based on ur key, u will specify a key in ur message but there is catch with a key and that is because the way hashing works.

The hashing guarantee that key will always give same number but didn’t ensure that two different key will never give u same number.

* If no partition or key is present choose a partition in a round robin fashion, u don’t care about which partition ur data is landing, but u want the partitioner distributed ur data evenly, we use 3rd rule.

For example, if u have 3 table and want to send all rows from 3 tables to 3 different partition, we can send table name as key that will wrong, because table A and table B can give same number after hashing. So it is better in managing the translation of table name to a partition number in ur producer and hard code partition number with message.

Another alternative is to implement custom partitioner and use that partitioner instead of default partitioner.

There is another catch with key, the partition number is mod of hash value of key and total number of partition in topic, so if u r increasing the number of partition of topic, the default partitioner returns different number, that may be problem if u relay on key for achieving particular partition. With these two problem we don’t find key to good use in decide custom partitioner.

If u want a particular type of partitioning the only option to create algorithm and implement tin custom partitioner.

Assume we are collecting data from bunch of sensors, all sensors sending to a single topic,we planned 10 partition for topic. But we want 3 partition dedicated for a particular sensor called TSS and 7 partition for rest of sensors. How did u achieve this?

--- It can done by implementing custom partitioner

1. Create SensorProducer.java

* It is similar to other producer problem discussed earlier, here we set two extra property
* partitioner.class, since we are not using default partitioner we are set this property to class name of custom partitioner
* speed.sensor.name – it is not kafka configuration property, it is custom property we are using to supply the name of sensor that requires a special treatment. We don’t want to hardcode TSS in custom partitioner so we use custom configuration method to apss value to partitioner
* We send some messages for various sensor and then we send some messages for TSS sensor

1. Create SensorPartitioner.java

* Create class which implement Partitioner interface which 3 methods, configure() for initialization and close() for clean up purpose and called once at the time of initializing
* Inside config(), we want to find the sensor name that requires 3 partition, the producer is sending the name as custom configuration.

speedSensorName = configs.get("speed.sensor.name").toString();

The above line is extracting the configuration value and setting to private variable so we can use it later

* partition() is a place where producer will call this method for each message and provide all details with every call. So the input to method is topicname, key, value and cluster details. We have everything that is require to calculate the partition number and return an integer as partition number.
* This method is the place where we implement algorithm for partition, we apply the algorithm in 4 step

Step 1: To determine the number of partition and reserve 30% of it for TSS, assuming we have 10 partition for topic, this logic will reserve 3 partition for TSS

How u get the number of partition in topic ? we got Cluster object as input, and the method partitions for topic, it give as list of all partition, then we take size of list that will the number of partition in topic and sp is 30% of partition

Step2: If we don’t get a String key, throw an exception, we need key because key tells us sensor name, without knowing sensor name we cant decide the message should go to one of 3 reserved partition or other bucket of 7 partition.

Step3 and 4 : To determine partition number, if key equal to TSS, then we hash the message value, divide by 3 and take mod as partition number. Using mod will make sure that we will always get 0,1, or 2. This message belong to TSS will go to partition 0 or 1 or 2.

If key is not equal to TSS, we divide by 7 and take mod, the mod will be between 0 and 6 so we adding 3 to shift from 3.

In Step 3 we are hashing message value because everytime key is TSS so hashing TSS will give same number everytime and all TSS message will go to same partition but we want to distribute to first 3 partition so we are hashing message value to get different number every time and in step 4 we are hashing message key because we want to show that we should be careful if u want to use key for achieving particular partition, and different key can land in same partition.

1. Start zookeeper
2. Start kafka server
3. Create topic SensorTopic

C:\Softwares\kafka\_2.12-2.0.0\config>kafka-topics.bat --zookeeper localhost:2181 --describe --topic SensorTopic

**KAFKA PRODUCER CONFIGURATION**

Kafka producer is controlled by Kafka configuration, in our earliear examples we use 4 configuration parameter likes

1. bootstrap.servers – is a list of kafka broker URL and port no, since it is mandatory so we must have atleast one value is specified for this parameter, the value provide for this parameter is used by the producer to connect to Kafka cluster without this value the producer cannot reach to the cluster, you should provide atleast two addresses because if the first broker is down the producer will reach out second address, if you have large cluster you can provide more that two addresses.
2. Key.serializer – takes the name of class that you want to use for serializing your key
3. Value.serializer – takes the names of class that you plan to use as a value serializer, you can use the same class for both key and value, if your key and value both are strings you can use same serializer for both
4. Partitioner.class – if you are using custom partitioner, we should specify the class name for this parameter
5. Acks – Is to configure acknowledgement, when producer send some message to kafka broker they get a response back from the broker, the response is RecordMetadata object or exception. This parameter takes 3 parameter 0,1 or all.

* If acks=0 means the producer will not wait for the response it will send messages over the network and forget it. There 3 side effects like possible loss of messages, high throughput and no retry. Since producer is not waiting for response there is no guarantee that the server has received the record so u many lose some records. However the producer is not waiting for an acknowledgement it can send data as fast as the network can support and achieve high throughput, the third side effect the producer not even go for retry

Kafka is a highly available system so there is a possibility that you lose ur record, however understand that there no guarantee so use this setting when loss of few message is not an issue.

* If acks=1 the producer will wait for the response however the response is send by the leader, so this parameter will have an impact on when the leader will send the response, in this case the leader will response after recording the message in its local storage, if the leader is down and message delivery fails the producer will retry after few milliseconds this option appears to be a safe choice. However there is a catch, you still cant guarantee that you will not lose ur message, because we have a single copy of message we are not sure it is replicated.
* Acks=all, the leader will acknowledge after it receives an acknowledgement from all of the live replicas. It will give highest reliability but costs highest latency. It is slowest option because you will waiting for all replicas however you can achieve better throughput using asynchronous send

1. Retries

It defines how many times the producer will retry after getting an error. The default value is 0.

retry.backoff.ms will control the time between two retries, the default is 100ms

1. max.in.flight.requests.per.connection

If you are using asynchronous send with a callback function to check ur errors, you are not waiting for response but ultimately get response using callback function.

How many inflight requests are allowed that are still not acknowledged?

For that we define max.in.flight.requests.per.connection, setting this parameter to a high value will increase memory usage but at the same time it will increase throughput. If u have enough memory you may want to set it to a higher value to achieve better performance for asynchronous send.

**KAFKA – CONSUMER GROUPS**

If your producer pushing data to topic at a moderate speed, a single consumer is enough to read and process the data. However if u want to scale up ur system and read data from kafka in parallel u need multiple consumers. Many application may have clear need for multiple producer pushing data to a topic at one end and multiple consumers read and process the data on other end.

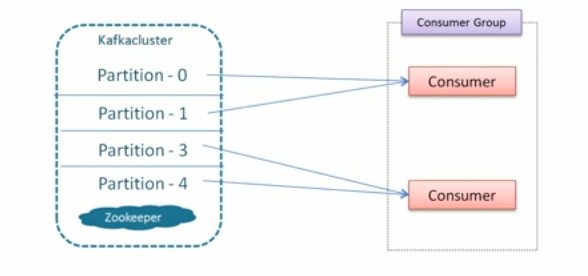
There is no coordination or sharing of data needed at producer side, but on the consumer side we have various factors.

1. How do we read messages in parallel in a single application?

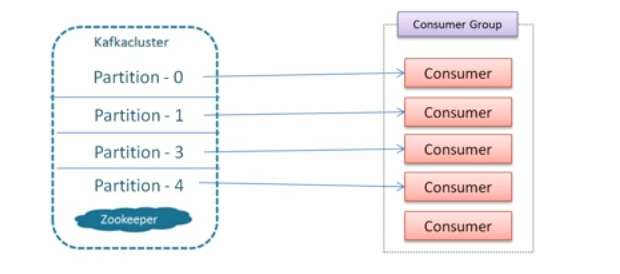
We can do that by creating a group and starting many consumers in the same group.

If we have multiple consumer reading data in parallel from same topic, all of them can read same messages. Only one consumer owns a partition at any point in time.

For example, we have 1 topic and 4 partition, so if we have only 1 consumer, it reads from all 4 partition, if we have 2 consumer each of them reads 2 partition,



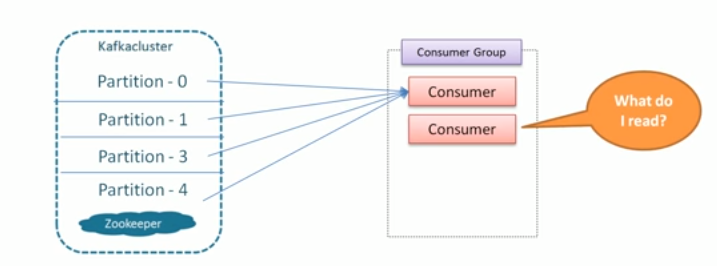
so the fundamental concept is the consumer dosent share the partition. There is no way we can read the same message more than once, however the solution also brings a limitation. The number of partitions on a topic is upper limit of consumer u can have in a group, so in our example if u have more than 4 consumer, one of them reads nothing, kafka won’t complain that u have 4 partition but starting 5 consumers



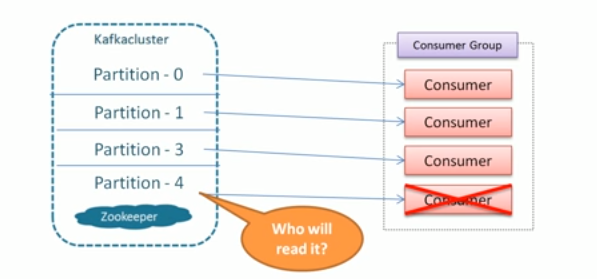
Simply 5th consumer will have nothing to read.

We have 4 partition and 4 consumer processes all reading parallel and no one reading each other data, so no duplicate. How do the consumer enter and exit in group?

We started with 1 consumer and wanted to scale up, so added 1 more, now we have 2 consumer, which partition does new consumer read and who should move some partition from 1st consumer and assign to 2nd consumer . The reassign problem dosent end.



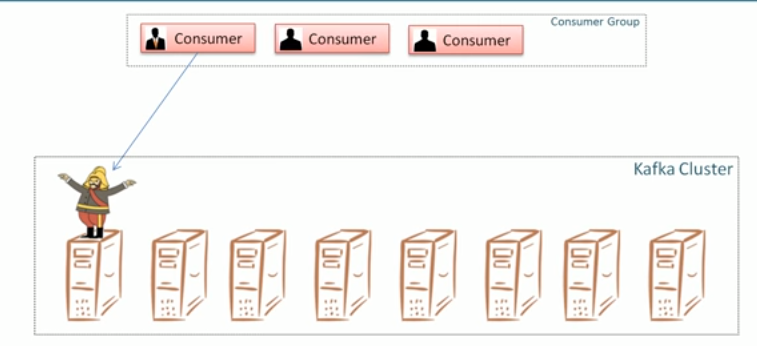
Assumer we have 4 consumer and 1 consumer crashed, we left with 3 and what happen to partition 4 and who will read it, after sometime the collapsed consumer is recovered so again we have 4 consumer so reassignment will be required once again.



In real application, consumers keep joining and exit, we don’t have control over that, how Kafka handles it?

When Consumer joins the group, how is partition assigned to it, what happens to partition when a consumer leaves the group, who manages all of these. Group corordinator manages this.

So one of kafka broker gets elected as group coordinator, when a consumer wants to join the group it sends the request to coordinator, the first corordinator to participate in a group becomes a leader, all other consumers joins later becomes the members of the group.



We have two actors coordinator and leader.

Coordinator responsible for managing list of group members, so everytime new consumer joins the group or existing member leaves the group, the coordinator modifies the list. On event of membership changes the coordinator realizes that it is time to rebalance the partition assignment because we may have new members and need to assign some partition or member left we need to reassign partition to some one else. So everytime the list is modified the coordinator initiates a rebalance activity

Leader is responsible for executing rebalancing activity, the leaders take a list of current members assign partition to them and send back to coordinator. The coordinator then communicates back to member about the new partition. During the rebalance activity none of consumers are allow to read any messages.

**KAFKA – OFFSET MANAGEMENT**

Kafka maintains two types of offset

1. Current Offset – used to avoid resending same record again to same consumer

When we call poll() method, Kafka sends some messages to us. Let us assume we have 100 records in partition, the initial position of current offset is 0, we made our first call and receive 20 messages. Now kafka will move this offset to 20, when we make our next request it will send some more messages starting from 20 and again move the offset forward. The offset is integer number that is used by Kafka to maintain current position of consumer. The conumser dosent get same record twice because of current offset.

1. Committed offset

Used to avoid resending to a new consumer in the vent of partition rebalancing.

This offset is the position that consumer has confirmed about processing (ie) after receiving the list of messages, we want to process it. This processing may be just storing in database, once we successfully process the record, we should commit the offset. So the committed offset is pointed to last record that the consumer successfully processed. The committed offset is critical in case of partition rebalance.

In the event of rebalancing when a new consumer is assigned same partition, where to start, what is already processed by previous owner, the answer is committed offset.

How to commit an offset?

1. Auto commit

* Easiest method, u can control the feature by setting 2 property

enable.auto.commit – by default true, we can turn it off by setting false

and auto.commit.interval.ms – defines the interval of autocommit, the default is 5sec. So in a default configuration when u make call to poll(), it will check if it is time to commit. If u past 5 sec since the previous call, the consumer will commit the last offset. So Kafka will commit ur current offset every 5 sec.

You have some messages in the partition, u made first code request, you receive 10 messages hence the consumer increase the offset to 10. You take 4 sec to process this 10 messages and make a new call, since u haven’t passed 5 sec, the consumer will not commit the offset.

You receive another set of records and some reason rebalance is triggered at this moment, first 10 records are already processed but nothing is committed yet, so the partition goes to different consumer. Since we don’t have committed offset, the new owner of the partition should start processing from beginning and process first 10 records once again. You can reduce the incident of commit by setting auto.commit.interval to lower value but u cant guarantee to eliminate repeated processes. The solution to particular problem using manual commit.

1. Manual commit

We can configure enable.auto.commit=false and manually commit after processing the records. Two approach for manual commit

1. Commit Sync – reliable method but it is blocking method, it will block ur call for completing a commit operation. It will also retry if there are recoverable errors
2. Commit Async – send the request and continue, the drawback is that commit async will not retry.

For example, we are trying to commit an offset as 75, it will fail for some recoverable reason and try to retry after few seconds, since it is async call without knowing that previous commit is waiting u initiate another commit. This time it is to commit 100 and it is successful, while commit 75 is waiting for retry.

Obviously we don’t want to commit 75 after commit 100, that make cause problem, so we design async commit not to retry, because we know that if one commit fails for a recoverable reason the next higher order commit will succeed.

The commit has significant impact on the client application, so we need to choose an appropriate method based on use case.

Refer ManualConsumer.java

1. In this example, we use Asynchronous commit (ie) commitAsync, in case of error we make sure that we commit before close and exit. So we use synchronous commit (ie) commitSync() before close our consumer.
2. Here we manually committing before pulling the next set of records.

**Spring Boot with Apache Kafka Producer (Refer SpringBoot-Kafka-Producer project)**

1. Create Spring boot project with Kafka (created by Spring boot so that easy to publish messages) and Web dependency
2. Create UserResource.java as Rest controller with mapping as Kafka. We create Get mapping /publish/{name}
3. We need to publish the message which we need to push using KafkaTemplate and autowired it

@Autowired

KafkaTemplate<String,String> kafkaTemplate;

1. We need to publish this on to a topic,

kafkaTemplate.send(TOPIC,message);

so we create a constant TOPIC

Private static final String TOPIC=”Kafka Example”;

1. Start zookeeper

C:\Softwares\kafka\_2.12-2.0.0\config>zookeeper-server-start.bat zookeeper.properties

1. Start Kafka server

C:\Softwares\kafka\_2.12-2.0.0\config>kafka-server-start.bat server.properties

1. Create the Kafka topic

C:\Users\senthil.kumart>kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic Kafka\_Example

Created topic “Kafka\_Example”

1. Once the topic is created we need to publish, before that we need to consume from this particular topic, so that we can see messages are getting posted or not. So we create a consumer which is connected to this topic and show what message are coming in

> kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic Kafka\_Example --from-beginning

1. To publish the message, we start spring boot application
2. Go to browser and run <http://localhost:8081/kafka/publish/Ram>, so in the browser it will print Published successfully and in consumer console it will print the message Ram
3. Now we published only String, but if u want to publish a JSON message, so we create a model class User.java with name, dept and salary
4. In UserResource.java we change String to User class

@Autowired

**private** KafkaTemplate<String, User> kafkaTemplate;

kafkaTemplate.send(TOPIC, new User(name,”Tech”,120000L));

1. However we need to serialize this particular User object in the time we want to publish. By default Kafka uses String, so we didn’t do anything for String message publishing part but now we publish JSON here, so we need to tell Kafka that we publish a JSON message
2. So we create KafkaConfiguration.java inside com.pack.config
3. This program has to be loaded during Spring boot initialization so we define @Configuration annotation.

Now we need to create a bean of type ProducerFactory and create a configuration and provide that configuration for creating DefaultKafkaProducerFactory. When we create DefaultKafkaProducerFactory we need to give some configuration. This configuration is of type HashMap and we put some values.

We need to put ProducerConfig which contain keywords like BOOTSTRAP\_SERVERS\_CONFIG where we need to provide server name (ie) localhost 127.0.0.1:9092

Next we put what type of key (ie) ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG and mention the serialization class which we going to use as key (ie) String.

Next for the value we need to provide ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG and for value we provide JSONSerializer.class

@Bean

**public** ProducerFactory<String, User> producerFactory() {

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

config.put(ProducerConfig.***BOOTSTRAP\_SERVERS\_CONFIG***, "127.0.0.1:9092");

config.put(ProducerConfig.***KEY\_SERIALIZER\_CLASS\_CONFIG***, StringSerializer.**class**);

config.put(ProducerConfig.***VALUE\_SERIALIZER\_CLASS\_CONFIG***, JsonSerializer.**class**);

**return** **new** DefaultKafkaProducerFactory<>(config);

}

1. Now who will tell Spring that this is configuration, now we have to create KafkaTemplate with String and User.

@Bean

**public** KafkaTemplate<String, User> kafkaTemplate() {

**return** **new** KafkaTemplate<>(producerFactory());

}

1. Restart spring boot application
2. Run in browser as, http://localhost:8081/kafka/publish/sam , so in browser it prints Published successfully and in Consumer console it prints user json format

**Spring Boot with Kafka Consumer (Refer SpringBoot-Kafka-Consumer project)**

1. Create Spring boot project with kafka and web dependency
2. Create Configuration class similar to Producer called KafkaConfiguration.java annotated with @Configuration. The bean required here is ConsumerFactory of type String and String which returns DefaultKafkaConsumerFactory.

It expects some configuration of type HashMap and add some values likes ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG in 127.0.0.1. Next we have to give group id using Consumer.Config.GROUP\_ID\_CONFIG.

Now we provide key for deserialization class as ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG and StringDeserializer.class

We provide the value as ConcumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG and StringDeserializer.class

@Bean

**public** ConsumerFactory<String, String> consumerFactory() {

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

config.put(ConsumerConfig.***BOOTSTRAP\_SERVERS\_CONFIG***, "127.0.0.1:9092");

config.put(ConsumerConfig.***GROUP\_ID\_CONFIG***, "group\_id");

config.put(ConsumerConfig.***KEY\_DESERIALIZER\_CLASS\_CONFIG***, StringDeserializer.**class**);

config.put(ConsumerConfig.***VALUE\_DESERIALIZER\_CLASS\_CONFIG***, StringDeserializer.**class**);

**return** **new** DefaultKafkaConsumerFactory<>(config);

}

1. Now we need to inject ConsumerFactory on to KafkaListenerFactory. We use ConcurrentKafkaListenerContainerFactory and we create object for it and set to ConsumerFactory.

@Bean

**public** ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerContainerFactory() {

ConcurrentKafkaListenerContainerFactory<String, String> factory = **new** ConcurrentKafkaListenerContainerFactory();

factory.setConsumerFactory(consumerFactory());

**return** factory;

}

1. Create Listener class as KafkaConsumer.java annotated with @Service. We need to provide a listener using @KafkaListener, whenever there is message inside Kafka topic, this particular listener will be called so that we can access the message.

How it know which topic to get it from so we have to provide topics=”Kafka\_Example” and provide groupId=”group\_id” which is provided in configuration

@Service

**public** **class** KafkaConsumer {

@KafkaListener(topics = "Kafka\_Example", groupId = "group\_id")

**public** **void** consume(String message) {

System.***out***.println("Consumed message: " + message);

}

}

1. Spring boot need to identify this KafkaListener, for that we use @EnableKafka in KafkaConfiguration.java, which scan for listeners
2. Start zookeeper
3. Start Kafka server
4. Start Spring boot application
5. We create producer using console

C:\Users\senthil.kumart>kafka-console-producer.bat --broker-list localhost:9092 --topic Kafka\_Example

>Hello world

>Welcome

Now whatever we typed in producer will present in Consumer

1. Now we try to JSON message, so we create User.java model class
2. Now we need to inject User into KafkaConfiguration, create another ConsumerFactory<String,User> with group id as group\_json and JsonDeserializer.class

@Bean

**public** ConsumerFactory<String, User> userConsumerFactory() {

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

config.put(ConsumerConfig.***BOOTSTRAP\_SERVERS\_CONFIG***, "127.0.0.1:9092");

config.put(ConsumerConfig.***GROUP\_ID\_CONFIG***, "group\_json");

config.put(ConsumerConfig.***KEY\_DESERIALIZER\_CLASS\_CONFIG***, StringDeserializer.**class**);

config.put(ConsumerConfig.***VALUE\_DESERIALIZER\_CLASS\_CONFIG***, JsonDeserializer.**class**);

**return** **new** DefaultKafkaConsumerFactory<>(config, **new** StringDeserializer(),

**new** JsonDeserializer<>(User.**class**));

}

Since we are getting custom message, we need to pass StringDeserializer and JsonDeserializer to DefaultKafkaConsumerFactory

1. Create new KafkaListenerFactory using ConcurrentKafkaListenerContainerFactory

@Bean

**public** ConcurrentKafkaListenerContainerFactory<String, User> userKafkaListenerFactory() {

ConcurrentKafkaListenerContainerFactory<String, User> factory = **new** ConcurrentKafkaListenerContainerFactory<>();

factory.setConsumerFactory(userConsumerFactory());

**return** factory;

}

1. Create one more KafkaListener in KafkaConsumer.java with another topic called Kafka\_Example\_json with groupId=group\_json.

There is a listener provided using containerFactory=userKafkaListenerFactory

@KafkaListener(topics = "Kafka\_Example\_json", groupId = "group\_json",

containerFactory = "userKafkaListenerFactory")

**public** **void** consumeJson(User user) {

System.***out***.println("Consumed JSON Message: " + user);

}

So that user message will directly deserialize and comes to particular method

1. Now we create new topic Kafka\_Example\_json

C:\Users\senthil.kumart>kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic Kafka\_Example\_json

1. Restart the spring boot application
2. Run the producer on this topic

C:\Users\senthil.kumart>kafka-console-producer.bat --broker-list localhost:9092 --topic Kafka\_Example\_json

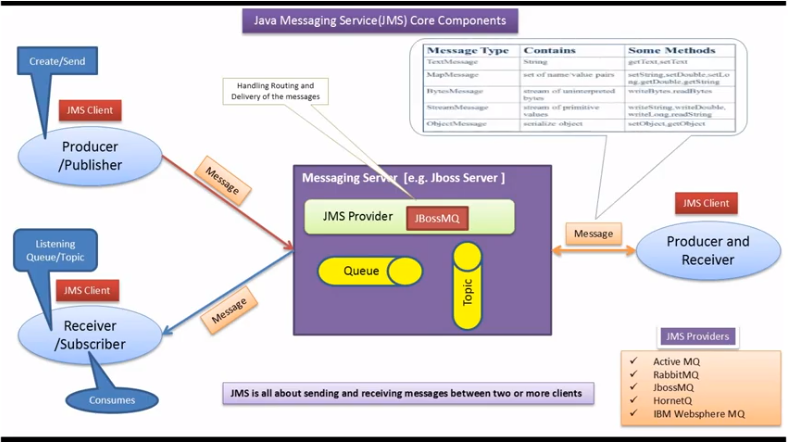
>{“name”:”sam”,”dept”:”Java”}

Now it will print json message on Consumer

**JAVA MESSAGING SYSTEM**

Using JMS we can send and receive messages between two or more clients. We can Messaging server Jboss server which support JMS providers (ie) JBossMQ which has Queue and Topic. There are different JMS providers are available like ActiveMQ, RabbitMQ, JBoss MQ, HornetMQ, IBM Websphere MQ.

JMS Provider is responsible for routing and delivery of the messages. We have Producer/publisher responsible for creating the message and sending the message to queue or topic, we have receiver or subscriber which will listen to queue/topic and if message are there it will consume. We can send different type of messages between producer and receiver like textmessage, mappmessage, BytesMessage, StreamMessage and ObjectMessage



Two types of messaging

1. Point to Point message

In PTP for each message there will be only one consumer, we have producer/publisher which sends the message to queue and consumer which listens the queue and get the message from queue. Once consumer consumes the message, it sends the acknowledgement back to messaging server. Queues are based on PTP

1. Publish/subscribe messaging

Each messages have multiple consumer, we have producer/publisher which creates a message and sends the message to topic, and we have multiple subscriber which subscribes to topic. Once message arrived to topic, the message will be copied and send to multiple subscribers.

For example, we take JDBC API which supports multiple databases like MySQL, Oracle and postgresql db like that JMS API which supports different JMS providers like ActiveMQ, RabbitMQ, IBM Websphere MQ etc

**Why Active MQ instead of database?**

Because there are appl where we can use database also, but ActiveMQ is messaging queue where data is stored and fetching that data.

We are using ActiveMQ instead of database because of reliable communication between the two distributed processes, we can also store messages in database but as soon as messages receives we should delete the message (ie) insert or delete for each messages. When we try to scale that upcoming 1000 of messages per second in database which tend to failover.

Message Oriented Middleware(MOM) like ActiveMQ are build to handle this type of use cases, they assume that message in system will be deleted quickly and can do optimization to avoid overhead. It can also push messages to consumer instead of consumer doing SQL query.

**INSTALLATION**

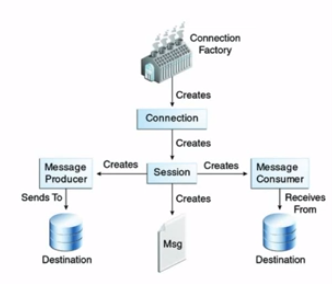
1. **Download ActiveMQ from** <https://activemq.apache.org/components/classic/download/> and extract it
2. **Go to bin folder and start**

**>activemq start**

**3. Open ActiveMQ page,** [**http://localhost:8161**](http://localhost:8161)

**4. Click Manage ActiveMQ broker, give both username and password as admin where we can see ActiveMQ console**

**JMS API Programming Model**

****

**The basic building blocks of a JMS application consists of**

**Using ConnectionFactory we can create connection, using Connection we create Session, using Session we can create message and create MessageProducer which sends message to destination and we can create MessageConsumer which receive message from destination. Using JMS, the application can communicate with each other by sending receiving messages.**

1. **Administered Objects: connection factories and destinations**

**ConnectionFactory is the object a client uses to create a connection to a provider. Each connection factory is an instance of ConnectionFactory, QueueConnectionFactory or TopicConnectionFactory interface.**

**There are two types of destinations like queue and topics. The message producer sends messages to the destination and message consumer consume message from destination**

1. **Connections**

**It encapsulates a virtual connection with JMS provider. You use connection to create one or more session**

1. **Sessions**

**It is a single threaded context for producing and consuming messages. You use session to create Message producers, Message consumers, Messages, Queue browsers, temporary queues and topics**

1. **Message producers**

**It is created by session and used for sending messages to a destination which implements MessageProducer interface**

1. **Message Consumers**

**It is created by a session used for receiving messages sent to a destination which implements MessageConsumer interface**

1. **Message Listeners**

**It acts as an asynchronous event handler for messages, which contain onMessage() which defines the actions to be taken when message arrives**

1. **JMS Message Selectors**

**Used to filter the messages it receives**

1. **JMS Messages**

**The ultimate purpose of JMS application is to produce and to consumer messages that can then be used by other software applications**

1. **JMS Queue brokers**

**You can create a QueueBrowser to inspect the messages in the queue. Messages sent to queue remain in the queue until the message consumer for that queue consumes them. So JMS API provides an object that allows you to browse the messages in queue**

**10.JMS Exception Handling**

**Catching JMSException provides a way to handle exception related to JMS API**

**Send and receive message from Queue (Refer JMS-MessageQueue project)**

1. Run MessageSender.java
2. Check in ActiveMQ browser, in Queue we have MESSAGEQUEUE
3. Click MESSAGEQUEUE, u can see the message
4. Run MessageSender.java 3 times, and see message in queue
5. Run MessageReceiver.java, to receive the message so that in browser Message dequeued changes as 1
6. Two more time again run it, messages dequeued will be change and no of pending messages will be 0, so we consumed 3 messages

**Send and receive message from Queue using Annotation**

1. MessageConfiguration.java, it is configuration class, Spring container reads this class and creates a ConnectionFactory object. To create ConnectionFactory object Spring container needs to know ActiveMQ broker url provided as DEFAULT\_BROKER\_URL.

Once the ConnectionFactory object is created, Spring Container creates JMSTemplate object, for which Spring needs connectionfactory object and MESSAGE\_QUEUE. Then Spring container creates MessageConverter

JMSTemplate provides an abstraction which hides all the complexities of JMS commication, it takes care of creating the connection, creating the session, creating the message producers and message consumers. It is also used to send and receive the message from queue

1. AppConfig.java, it is main configuration class, Spring container reads this configuration class and makes sure all the dependent objects are created and autowired properly.
2. MessageSender.java, it has sendMessage() which is responsible for sending message to the queue
3. MessageReceiver.java, it has receiveMessage() which is responsible for receiving message from queue. Here we have message converter which converts java object to JMS message and JMS message to java object
4. MessageProducerApp.java, we created context object, using context object we are getting a MessageSender object from Spring container, then we send message to a queue
5. Run MessageProducerApp.java
6. Goto ActiveMQ console, MESSAGE\_QUEUE created with 1 Message enqueued
7. MessageConsumerApp.java, we created context object, using that object we get MessageReceive object, then we receive message from queue
8. Run MessageConsumerApp.java
9. Goto ActiveMQ console, we see queue with 1 message dequeued
10. Like this we can send multiple messages, by running MessageProducerApp multiple times and receive by running MessageConsumerApp multiple times

**SEND AND RECEIVE ACTIVEMQ MESSAGE ASYNCHRONOUSLY**

1. Refer SpringJMSProducer project
2. Run MessageProducerApp.java
3. In SpringJMSConsumer project, run MessageConsumerApp.java

Here we create context object by reading Spring configuration class, it will keep on waiting for message, whenever the message comes, the application will consume the message asynchronously

1. Just run MessageProducerApp.java, each time then MessageConsumerApp will consume the message till that milliseconds

**LOADBALANCING IN MICROSERVICES**

When u r developing microservice oriented application, so load balancing is very much important.

Problem statement:

Consider an microservice application DoctorPortal that uses MicroService1, MicroService2 and MicroService3. MicroService1 is running in production environment, in production environm we have many instance of MicroService1 so remember the advantage of microservice that we can more than one instances. Consider we have 3 instances MS1, MS2, MS3. From DoctorPortal we have sending n number of request to MicroService1 to get number of services available. So how “n” number of request going to know which instance to call for the response. So how it is going to divide the load for these instance, so this is the problem when we have multiple instance of same service.

So we have same multiple instance for MicroService2 and MicroService3 and multiple request coming from DoctorPortal so how it is going to divide the request to multiple instances.

Solution – Loadbalancer

1. Load balancing improves the distribution of workloads across multiple microservices (ie) if we have multiple instance on single microservice and many request sending to that microservice loadbalance is used to divide the request to each instances.

Two types

1. Client side loadbalancer

Where loadbalancer is part of client (ie) when u r making the request to service we know which instance to call and it is responsibility of client to track that one

1. Server side loadbalancer

Where loadbalancer is part of server (ie) client is making a request and server have load balancer implementation and it knows about the number of instances of Microservice1 and it decides which instance to call

So basically loadbalancer is to divide the load among available services, client side means it is responsibility of client to call particular instance of that service and on server side it is responsibility of server itself.

**In client side load balancer is part of request itself, so when u r making request to individual instance, it knows which instance to call**

1. **Netflix Ribbon + Spring boot – Ribbon provides the facility of load balancing in microservice architecture**
2. **@LoadBalanced**
3. **RestTemplate for communication among services**
4. **Load balancer algorithms like Round robin, Least connection, Agent based adaptive load balancing**

**Example 1: Refer RibbonClientProject, RibbonMicroservice1**

1. Create REST based RibbonClientProject and RibbonMicroservice1, from RibbonClientProject we will call RibbonMicroservice1 using RestTemplate. We will create 3 instances of RibbonMicroservice1 and those 3 instance we will configure in RibbonClientProject and we see how the request is loadbalanced among those request
2. Create RibbonMicroservice1 with web dependency
3. Create RibbonClientProject which is client of RibbonMicroservice so we add the dependency web and

To make it act as load balance we have to add @RibbonClient with name so that it know which client.

We are creating without service registry so we have to know how many instance of particular service is running, so we configured using

Ribbon-service:

Ribbon:

listOfServers: localhost:9081, localhost:9082, localhost:9083

so we have different instance of microservice1 is running

To create the instance of RestTemplate,

@Bean

|  |
| --- |
|  |
|  | @LoadBalanced |
|  | public RestTemplate restTemplate() { |
|  | return new RestTemplate(); |
|  | } |
|  | } |

now RestTemplate has been created with @LoadBalanced, so RestTemplate coming with one more interceptor. When RestTemplate going to call any third party microservice it is going to call with one interceptor and that interceptor is provided by the Ribbon client

1. When we give request /services it will invoke /allService1, we have different algorithms in this case Round robin – when 1st request comes it pick first instance secondly goes to 9082 and thirdly goes to 9083.
2. Run RibbonClientProject
3. Run RibbonMicroservice1, which runs in 9081
4. Now we want to run RibbonMicroservice1 in 9082 and 9083
5. Right click RibbonMicroservice1Application.java – Run Configuration –change the name RibbonMicroservice1Application -9082– in env tab give server.port=9082
6. Once again right click RibbonMicroservice1Application.java – Run Configuration –change the name RibbonMicroservice1Application-9083 – in env tab give server.port=9083
7. Now run in browser, <http://localhost:7081/services>

We get response from 9081

Refresh the page, now we get response from 9082, once again refresh now we get from 9083. We have 3 request servicing all 3 instance.

If we refresh again, the counter will be 2 again starts from 9081 then 9082 and then 9083.

The counter is local to the instance that is running, it is not distributed and serving different instance and this counter is specific to that instance so only it is incrementing when endpoint on that particular instance is finished

1. Now stop the server running on 9083, but ribbon dosent know whether it is down or not, so now when we run for 9083 it will show the error

But u refresh again and again the ribbon client decomposition the service that is running on 9083, so when number of request that are failing, so if u r hitting that server two three times and it is not responding, so load balance in RibbonClientProject knows it is down and decompose that server. If u refresh few times again it will show error page because it may think after some time the service may up

So whenever we use @LoadBalanced internally it uses some interceptor which is called LoadBalancerInterceptor( u can check in the console error)

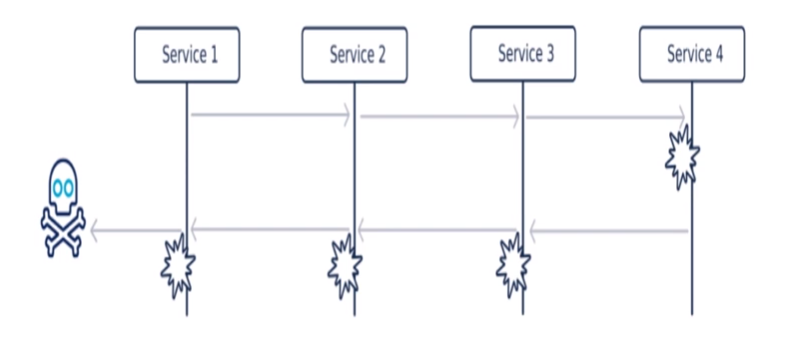
**Example 2: Refer ShoppingServer-Ribbon,ShoppingService-Ribbon,PaymentService-Ribbon**

1. Run ShoppingServer-Ribbon
2. Check Eureka server using <http://localhost:8761>
3. Run PaymentService-Ribbon with port no 8081
4. Change port no as 8082 in PaymentService application.yml and run it once again
5. Change port no as 8083 in PaymentService application.yml and run it once again, check whether all 3 services are updated in eureka
6. Run ShoppingService-Ribbon and check whether update in eureka which runs in 9999
7. Now run <http://localhost:9999/payment/20000>
8. Now it will display the output and if u refresh each time it will display the output from different port no (ie)8081 or 8082 or 8083

**CIRCUIT BREAKER IN MICROSERVICE (Hysterix, Spring Retry,Resilience 4J)**

**Circuit breaker is a design pattern used in modern software development. It is used to detect failures and encapsulates the logic of preventing a failure from constantly recurring, during maintenance, temporary external system failure or unexpected system difficulties.**

**To avoid such failures and to make application more resilient, to make appl more fault proof so only we have circuit breaker design pattern**

****

**In enterprise appl, we have multiple service layers, so one user request coming is pass through first 3 service layer and goes to service 4 and in service 4 there is failure happens, so that failure lead to service3 and from there to service 2 and service 1 respectively and all way goes to user, and user is impacted then our business will be impacted. To avoid this we have circuit breaker design pattern which will provide the solution to trace the critical problem.**

**Netflix created an implementation over circuit breaker design pattern as Hystrix which easily pluggable to Spring Boot appl. Whenever we try to hit the service which is down, in those case it is having a fallback in place. In case of main service is not working then fallback will provide required response, those failures will not reflect to users.**

Hystrix is a library created by Netflix used to handle failures and do some fallbacks when there is a failure. Hystrix is fault tolerant framework similar to catch block in a try catch exception. Hystrix have fallback mechanism with which we can do fallback, additionally we can configure this fallback using circuit breaker design pattern.

Circuit breaker design pattern is similar to how circuit works in electrical board, we have different states like open, close and half open. For example, if u have circuit open ur current dosent pass through ur circuit and light is always switched off, which means something is broken. If the circuit is closed (ie) electricity pass through the circuit and light glows so typically in successful scenario ur circuit is closed. Initially ur circuit will be closed and it open if there is failure. Hystrix consolidates all these circuit breaker design pattern and shows up in Hystrix dashboard.

We can control through configuration whether the circuit is opened or closed where we say there are 100 request coming in and 5 request then open the circuit and for next continuous 5 request don’t allow them and by default fail them. For example in IRCTC website if u refresh again and again, sometimes it immediately fails this happens because there is fault tolerant or circuit breaker which is implemented which will break the circuit, it will not send till the server, it will say the 1st request fails so we are going to fail for next 5 continuous request until the circuit is going to function.

Let take ecommerce application where the request comes from different source to different microservices, what happens if one service goes down, the related source crashes instead we have to create fault tolerant api where it returns empty response or error message to user, we should not crash UI if something crashes

Example: Refer HystrixApplication project

[Hystrix Dashboard | Fault Tolerant Circuit Breaker Design Pattern Example in Spring Boot - YouTube](https://www.youtube.com/watch?v=SxJ2uLZWOyw)

1. Create project with dependency web,actuator, hystrix, hystrix dashboard
2. Configure @EnableCircuitBreaker and @EnableHystrixDashboard
3. In RestController.java we created hello(), in this method we created some artificial failures by using RandomUtils nextBoolean() which randomly give true or false and based on that it throws Exception. In normal application if we don’t have Hystrix implementation enabled, it shows exception in UI however we don’t want to do that so we enable hystrix.

In order to make hello() to be fault tolerant we add @HystrixCommand with fallbackMethod which is nothing but failure scenario which will be invoked when there is exception in this particular method. We also give commandKey and groupKey which is used to identify the method in Hystrix dashboard

1. We implement fallback(), the signature should be exactly same where we added @HystrixCommand annotation, if there is failure there is fallback and if no error means it returns “Hello world”
2. Run the application and actuator will have 2 paths (ie) /hystrix – will open the dashboard

/hystrix.stream – which contains all streaming information

1. Open the hystrix using <http://localhost:8092/hystrix>
2. In that provide <http://localhost:8092/hystrix.stream>

Which contains the streaming information and it dosent contain anything

1. In another tab, run REST endpoint <http://localhost:8092/rest/hello>, now in hystrix dashboard it shows the status of circuit

Example 2: Refer HystrixServer, HystrixBackendService, HystrixBackendServiceBackup

1. We created an endpoint from HystrixServer, from that endpoint we try to access HystrixBackendService, so when HystrixBackendService is down we fallback HystrixBackendServiceBackup
2. In HystrixServer, we have /getAccounts endpoint which call getName() of HystrixImpl bean. From this method we try to access some backend service getName(). If any error occurs we call fallback method which calls backup service getName() which gives required response to user.
3. Run all 3 appl. Now execute

<http://localhost:8080/getAccounts>

Now it execute backend service

1. Now stop HystrixBackendService, then run <http://localhost:8080/getAccounts>, now it execute Backup service as backend service is stopped so it invoke fallback method

**SPRING BOOT TURBINE**

Turbine is an open-source tool from Netflix for aggregating multiple streams into a single stream.

Consider we have two microservice application running, and if we want to see the metrics of both application we have open separate hystrix dashboard and see the metrics. So for application with many microservice it is difficult to open separate hystrix dashboard, so if u want to combine all the metrics in single dashboard we can use Turbine.

@EnableTurbine internally uses Eureka server and access all the clients that are registered inside server.

Example: Refere Turbine-EurekaServer, Turbine-Hystrix, Turbine-Recommendationservice, Turbine-UserService

1. Run all application
2. Run eureka registry as <http://localhost:8761>
3. Run userservice as [localhost:8060/personalized/1](http://localhost:8060/personalized/1), then run the hystrix dashboard separately using <http://localhost:9090/hystrix> and give <http://localhost:8060/actuator/hystrix.stream>
4. Run recommendationservice as [localhost:8070/recommendations](http://localhost:8070/recommendations), then run the hystrix dashboard separately using <http://localhost:9090/hystrix> and give <http://localhost:8070/actuator/hystrix.stream>

Instead of running hystrix twice to see, use turbine to see all services at a time

1. Run Hystrix <http://localhost:9090/hystrix> in that use

<http://localhost:9090/turbine.stream?cluster=default>

It displays both user service and recommendation service.

If nothing display hit both service once again and try

1. If u want to access individual service in hystrix, define

turbine.aggregator.clusterConfig= USER-SERVICE,RECOMMENDATION-SERVICE in application.properties file of TurbineHystrix

and access like

<http://localhost:9090/turbine.stream?cluster=USER-SERVICE>

<http://localhost:9090/turbine.stream?cluster=RECOMMENDATION-SERVICE>

**SPRING CLOUD CONFIG SERVER**

As you know whole world is moving from monolithic to distributed system,when we talk about distributed system microservice play very important role.

**Why do we need config server?**

**1.Configuration in distributed system, configuration in microservices**

The configuration in distributed system is important when we talk about distributed system microservices comes into picture which means appl is divided into small pieces and they are combined together they make the application, so main application makes call to these services and perform the functions

2.Changes in properties in different environment like dev, test, prod

For example creating a microservice which is interacting with database and to connects with database it needs username and password so in the dev environment it should use a database which is entirely dedicated for development purpose. Once the development is done now it goes to test environment where it will have different username and password to connect with database, once testing is done now application goes live in production, in production env the database instance is different for example we use Oracle there will different oracle instance for production different from test and development separately

So when software is moving from one env to another env, the things are changing, so we cant do always code changes so when software is moved from dev to test and then to production then it is not accepted to do code changes.

1. Changes in properties files

We can change properties files or yml file. Consider we have microservice appl which has properties file, now we can change all db version in this file for dev or test or prod env.

**So how do we make changes to properties and how we can effectively do without changing the code this is the place where config server plays an important role.**

1. Need to restart to change the config

In monolithic application if we are not using config server and we want to make changes to configuration we have to restart your application to effect the changes made in properties file.

But when spring cloud config comes into picture we can make the changes in your config files and these changes will reflect into your appl.

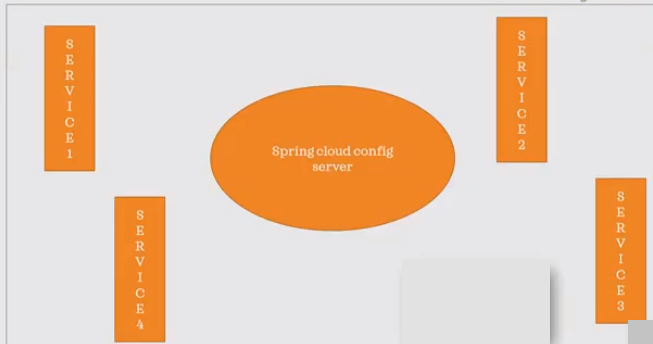
**What is Config server?**

1. Used to change the configuration at runtime and for that we don’t need to restart or redeploy ur application
2. Externalizing the configuration means we are changing the property configuration from outside without changing the code itself
3. Ideal for cloud native applications (ie) we are talking about microservices, distributed system so cloud env gives many ways to create the microservice by giving different dependencies.

What is Spring Cloud Config Server?

1. It provides server and client side support for externalized configuration in a distributed system
2. It provides an HTTP resource based API for external configuration

So when we have config server it is providing the REST endpoints, with the help of these endpoints we can get all the details about the configuration that you are making the changes



Here we have spring cloud config server with different services like service1, service2,service3 and service4 and all the services have different config which is changing when it is moving from dev to test and to prod. To make these changes we are using cloud config server, all of 4 applications are connected to spring cloud config server, consider we do changes in service2 then we do changes in the respective file mapped to config server and changes will take effect immediately.

Spring cloud config server maintains the configuration details in another repository which can be github or bitbucket or any local file system. Normally all the 4 service have username and password defined in different properties file, so you make the changes to this repository and this repository is looked by spring cloud config server and this cloud config server is pushing the information to these services.

Any changes made to this repository like github or bit bucket, these changes will affect config server and it will be notified to respective service.

**Example 1: Refer SpringConfigServer-1**

Step 1: SpringConfigServer-1, we will reading properties or yml file from Git repository using cloud config server

1. Create SpringConfigServer-1 project with actuator and configserver
2. Use @EnableConfigServer in main class, which makes Spring boot appl as config server
3. Create repository in GitHub and create different properties file

We created configprops with 2 properties files

1. Now we want to give the location of git repo inside applixcation.yml file

Label: master indicate that we take from master, u can specify the branches too

Config server library exposes some endpoints to access the properties like

/{application}/{profile}[/{label}]

/{application}-{profile}.yml

/{label}/{application}-{profile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

Using one of these we can see data in properties file

1. Run SpringConfigServer-1
2. Run as <http://localhost:8090/config-server(application> name)/default – default is profile which access default property

Now it will display all the info from default application.properties file

1. To access application-prod.properties file we use

<http://localhost:8090/config-server/prod>

1. Now we want change something in properties file and it has to update automatically, if u refresh the page it will automatically updated
2. Now we are accessing the master details if we want to access the branch then

<http://localhost:8090/config-server/default/branchname>

similarly

<http://localhost:8090/config-server/prod/branchname>

Step 2: Refer SpringConfigClient-1

1. Create microservice to communicate with Spring cloud config server
2. Create SpringConfigClient-1 with dependency web, config client and actuator
3. Create MainController.java
4. How config server comes into picture when config client is implemented and start the application, it will try to connect to config server. By default the address of config server is <http://localhost:8888>, if u want to override then u can configure in bootstrap.yml file. Always bootstrap.yml because we want to configure at the time of loading the server

server:

port: 8081

spring:

cloud:

config:

uri: http://localhost:8090/

Even if we not mention http://localhost:8090 , by default it invoke <http://localhost:8888>, the server starts but we don’t have default profile so it wont fetch, so always configure http://localhost:8090/

1. Start the application
2. Run <http://localhost:8081/info>

Fetch the default property from server to ur application

1. If we want to take profile specific properties like test or production, we need to configure in bootstrap.yml as

**profile: prod**

1. Run the application, run <http://localhost:8081/info>

Now it will load production related properties

9.Now we want to update the configuration without restarting the server (ie) whenever we change anything in properties file it automatically reflect in client side

So we have two ways to read the configuration.

1. When u have **@ConfigurationProperties,** these values will be refreshed when u call **/refresh[POST]** endpoint which is provided by Spring cloud config, this is the post request. When u call this, the changes made in git repo it will loaded into the system
2. When u have @Value annotation, we have to call /refresh[POST] in addition,whenever we have @Value we have to call @RefreshScope

So when u call /refresh[POST] only @ConfigurationProperties will be updated, @Value will not updated for that we have to use @RefreshScope too

1. Now we have to expose actuator info using <http://localhost:8081/actuator>

It will display only few actuators

1. To configure other endpoints we have configure in bootstrap.yml as

Management:

Endpoints:

Web:

Exposure:

Include:

* -health
* - refresh

So we enabled refresh end point

1. Restart both the application
2. Now we change values application-prod.properties in git repo, if we run <http://localhost:8081/info> ,it wont be updated
3. To update we have to give /refresh, post request in POSTMAN. So in Postman we give

<http://localhost:8081/actuator/refresh> in POST

[

    "config.client.version",

    "channel.url",

    "channel.name",

    "channel.source"

]

Will display the above saying the data is changed

1. Now if we run <http://localhost:8081/info>, now we can see the updated configuration from properties file

Each time if u change in properties files then u have to /refresh endpoint in actuator, then only it updates the data

At a time we can give only one profile

**Example 2:**

1. Create SpringConfigServer-2 project with config server dependency
2. Create bootstrap.properties file which will be loaded before all configuration is loaded

**spring.cloud.config.server.git.uri=C:/Microservice2/SpringConfigServer-2**

Now we are converting this application as git repository

Create config-client.properties inside the project with

message=Spring cloud config server without GIT

1. Open Git bash in SpringConfigServer-2

* Git init
* Git add .
* Git commit –m config-client.properties

1. Create SpringConfigClient-2
2. Start both the application
3. Now run <http://localhost:8981/rest/message>

Will return the value from properties file

1. Now if we change the value in config-client.properties file it wont change the value.

We change value in config-client.properties after that do commit

* Git add config-client.properties
* Git commit –m config-client.properties

1. If u want to change automatically then we have to give

<http://localhost:8981/actuator/refresh> with POST

9.In this example if we change anything in config-client.properties file we have to commit and then run actuator refresh

**SPRING CLOUD BUS**

Previously **Spring Cloud Config Server** as a centralized location for keeping the configuration properties related to the application services (microservices).  The application services will act as **Config Clients** who will communicate with **Config Server** to retrieve the properties related to them.

If any property is changed, the related service need to be notified by triggering a **refresh event** with Spring Boot Actuator (***/actuator/refresh***). The user will have to manually trigger this **refresh event**. Once the event is triggered, all the beans annotated with **@RefreshScope** will be reloaded (the configurations will be re-fetched) from the **Config Server**.

In a real microservice environment, there will be a large number of independent application services. Therefore is it not practical for the user to manually trigger the **refresh event** for all the related services whenever a property is changed.

The better approach is to trigger the refresh event for one service and broadcast the event through all other available services.

**Spring Cloud Bus** links the independent services in the microservices environment through a light weight message broker (e.g:- **RabbitMQ** or **Kafka**).  This message broker can be used to broadcast the configuration changes and events. In addition, it can be used as a communication channel among independent services.

**Spring Cloud Bus** links the independent application services (distributed nodes) through lightweight message broker. The message broker can be implemented with either **RabbitMQ** or **Kafka**.

Refer this site for program explantion

<https://springbootdev.com/2018/07/17/spring-cloud-config-refreshing-the-config-changes-with-spring-cloud-bus-part-2/>

Example:

Step 1: Install RabbitMQ

1. go to rabbitmq official website [https://www.rabbitmq.com/](https://www.youtube.com/redirect?q=https%3A%2F%2Fwww.rabbitmq.com%2F&redir_token=QUFFLUhqa2U2cndsUks5Wk53M0tjMzh5YVJJbTVBTGdBUXxBQ3Jtc0tsUGs1VDhlSmUxWVpOWFU3VHRWMVNwVEpZUmh0b0NYNTk1alhsRTJJRjlBOUU1ZVJkWlZ1MlpTTWc3dmpPYUZCVlphZFhMT0lqOEY4T1FfeEFsemdHZlNGVms5Z1pnMkd0d0hlY1BQSXp3c0t4MDhGNA%3D%3D&v=V9DWKbalbWQ&event=video_description)

2. Click on "Get Started" menu

3. then click on "Download + installation"

4. then click on windows installer recommended

5. then download erlang 64 bit from <https://www.erlang.org/downloads/22.2> - click [OTP 22.2 Windows 64-bit Binary File](http://erlang.org/download/otp_win64_22.2.exe)(94825052)

6. then download the rabbitmq3.8.9 .exe file \

7. First install erlang

8. then install rabbitmq

9. then go to start menu and search for rabbitmq command prompt

10. type command "rabbitmq-plugins enable rabbitmq\_management"

All set to go now go to [http://localhost:15672](https://www.youtube.com/redirect?q=http%3A%2F%2Flocalhost%3A15672&redir_token=QUFFLUhqbXM1TTMtUTEtaUE4R0RrbnRTVTBvdndCR0Z6UXxBQ3Jtc0tuTTktSk4xejJwVGQwNFVxUl9HSWlhQnF0ZlJubXIzUWFBTlhQQ1pMS2RMM1VzeWJaeHE2aWFVVEU2bFYtYm1iMVdFNXlweG5mRzA0QktSalYyb3F3b1BZaTBmeGFXSmJxMjRpUnFpZGgtTDg5OUFPMA%3D%3D&v=V9DWKbalbWQ&event=video_description)

username: guest

passowrd: guest

Step 2: Create gitlab repository called cloudbus-repository with configuration-properties with application.properties, development-service.properties and employee-service.properties

Step 3:

1. Create SpringCloudConfig-CloudBusServer project with dependency config server
2. Create application.properties file

**spring.cloud.config.server.git.uri** :- This specifies the **Git repository** location where the property files are stored.

**spring.cloud.config.server.git.searchPaths**:- If the property files are stored under the sub directory of the repository, the directory name should be specified here. if there are multiple sub directories, those can be declared with spaces. If the property files are stored in the root of the repository, then you can neglect this configuration. In this example, we have stored the property files under the “***configuration-properties***” directory of the repository. Therefore we have used the “**searchPaths**” configuration here.

Step 4:

1. Create DepartmentService-CloudBus with web,actuator,config client and rabbit mq dependency
2. Create EmployeeService-CloudBus with web,actuator,config client and rabbit mq dependency

Step 5:

1. Start all application
2. Run <http://localhost:8081/service>

It will fetch serviceName of DepartmentService

from gitlab configured in Config server

1. Run <http://localhost:8082/service>

It will fetch serviceName of EmployeeService

from gitlab configured in Config server

1. But if we change in properties files of Gitlab then we have to run /actuator/refresh in POST individually

we have to manually trigger the **/actuator/refresh**event for all **Config Clients** (application services) whenever a property is changed. Otherwise the change will not get reflected in all services. We found that it is not practical and easy approach.

Here we are going to use the **Spring Cloud Bus** to **broadcast** the **refresh event** across all services.  Therefore whenever the property is changed, we need to trigger the **refresh event** for the **Spring Cloud Bus**. This can be done with invoking  **/actuator/bus-refresh**  endpoint through any of the connected services (any service that is connected to the **Spring Cloud Bus**). Then the **Spring Cloud Bus** will broadcast the **refresh event** across all the connected services.

In this way, it is possible to **trigger** the **refresh event** in **one service** and get it **reflected** in **all other connected services**.

1. Now if we change the value in GITLAB in any one server it will affect the common values for all services using /actuator/bus-refresh

<http://localhost:8081/actuator/bus-refresh> using POST request

Now we call /actuator/bus-refresh for departmentservice now it will affect all the service through spring cloud bus

1. Now if we run <http://localhost:8081/service> it will be updated the properties information of both department service and employee service

**SPRING CLOUD STREAM**

It is a framework for building highly scalable event driven microservice which connected with shared messaging system.



We have two microservice producer which publish the event and consumer which subscribe the event, which is up and running. To perform the event driven architecture we need one of messaging system like Kafka and RabbitMQ.

As we are using Kafka we have add Kafka related properties to producer as well kafka properties to consumer. My requirement change from Kafka to RabbitMQ, now we will go to the code and add RabbitMQ configuration both in producer and consumer services.

So my application is tightly coupled with messaging system which is not good. To solve this problem we use Spring Cloud Stream where we don’t need to specify the implementation of messaging system which we are using, only we have specify the required binding dependency. If we use Kafka we have to specify Kafka binder dependency in Producer and consumer then how microservice communicate with Kafka will take care by Spring cloud Stream and if we use RabbitMQ then give binder dependency in producer and consumer, spring cloud stream will take care.

Example:

1. Start zookeeper

C:\softwares\kafka\config>zookeeper-server-start.bat zookeeper.properties

1. Start Kafka server

C:\softwares\kafka\config>kafka-server-start.bat server.properties

1. Create SpringCloudStreamPublisher with web, lombok, cloud stream and Spring for Apache Kafka
2. Create SpringCloudStreamConsumer with lombok, cloud stream and Spring for Apache Kafka
3. When we create Kafka we have to create Topic to communicate with producer and consumer. Publisher will push the message to topic and consumer will fetch message from topic
4. Create topic to communicate between producer and consumer

C:\Softwares\kafka\_2.12-2.0.0\config>kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 -topic messageinfo

1. List all topic in Kafka

C:\Softwares\kafka\_2.12-2.0.0\config>kafka-topics.bat --list --zookeeper localhost:2181

1. Now we have to give the name of topic so that the publisher can publish the event and consumer can consume the messages from the topic
2. In both application.yml we have to give the topic name. Now the publisher will publish the book object over the topic and consumer will get from the topic

10.Start SpringConfigStreamPublisher

11. Start SpringCloudStreamConsumer application

12. In Postman <http://localhost:9192/publish> in POST request with body

{

“id” : 123,

“name” : “Java”

}

Now we can see the book object in consumer appl

**SPRING CLOUD STREAM USING RABBITMQ**

In RabbitMQ, there are 4 types of exchanges,

1. Amq.direct – The routing key is same like queue name, it is one-to-one, what are the queue you want to publish the message it will going to queue alone
2. Amq.fanout – it is like one-to-many, it will not looking for the routing key, the message will be published to all queue that is connected to particular exchange
3. Amq.topic – Topic is based on routing key, for example we have exchangetext in which 3 queues are connected test1.queue, test2.queue and test3.queue. While creating the connection itself we will give the routing key so what are the routing key you are publishing, the message will be delivered to corresponding queue
4. Amq.headers – It is based on X-Match header value, it will not consider routing key. If any header is matched it will publish to that particular queue

By default when u create from Spring cloud stream, topic will be created exchanges

Step 1:

1. Create a exchange
2. Click Add a new exchange
3. Name: test
4. Type: direct, Durability: durable, Auto delete :No
5. Click Add Exchange
6. Create Queue
7. Click Add a new Queue
8. Name: test.queue
9. Click Add Queue
10. Now attach the queue to exchange
11. Click test in exchange
12. Click Bindings
13. In that give test.queue in TO QUEUE
14. Routing Key as # (so any value is given it will send to corresponding exchange)
15. Click Bind
16. Click Queues
17. Click test.queue, now u can see test is binded and ready 0 since no message is published
18. Now we publish one sample test message
19. In Exchange, Click test
20. In Publishing message under Routing key give as test
21. In Payload, some sample message like hello world
22. Click Publish
23. Click Queue tab, click test.queue. now u can see 1 message under ready
24. If we want to read the message, click Get Message, we can see the message published

Step 2:

1. Create SpringCloudStreamRabbitMQ with cloud stream and spring for rabbit dependency
2. To convert SpringBoot application to Spring cloud stream we use @EnableBinding in main class. It takes one or more interfaces as input. By default Spring cloud stream provides 3 interfaces like Sink (for input), Source (for publishing message from appl to message service), Processor (it contains both Source and Sink

Now go to Sink interface, it uses @Input which identifies this is input channel through which we are going to receive message from RabbitMQ to our application. So whatever name given in input channel it is the name expecting in RabbitMQ exchange, if not passed any name it will take method name of subcriberchannel

Source interface is for output, from appl we want to publish message to RabbitMQ we have to use Source interface. Source uses default channel name as output

Processor interface which extends both Source and Sink.

@EnableBinding provides the connection to RabbitMQ server.

1. Start the application. It will create an exchange called input and creates a queue because we have to read messages from the queue by taking exchange name + anonymous.64 bit unique number
2. In RabbitMQ, <http://localhost:15672>

We can exchange called “input” and if we click we can see group name input.anonymous.64 bit number. Go to queues tab and see the queue

1. Now stop the appl, in RabbitMQ the queue will be automatically deleted but exchange will be available. If we restart it will create another queue because it dynamically generate random queue
2. Now we publish one message and read from our application. So we create model class Employee
3. Now we want to receive messages from this input exchange to that queue whatever is generated so we create handle() with @StreamListener annotation through which we can consume the message and provide the exchange name called “input”, which is nothing but a constant in Sink interface
4. Start the application which create one more unique queue
5. From RabbitMQ we publish a message, go to Exchange input and in Publish message

Routing key : test1

Headers: content-type = application/json

Payload:

{ “name”:”sam”,”age”:24,”salary”:20000,”address”:”xyzstreet”}

Click Publish message

Now the message will published in my spring boot application and in queue, ready=0 means that the message is consumed by appl

1. Instead of 64 bit number for groupname we can give some name in application.properties file as

Spring.cloud.stream.bindings.input.group=queue

1. Start the application, now the queue name is input.queue

Previously we have used Sink.class which is provided by Spring cloud stream which has exchange name “input” and creates corresponding queue to the exchange. If u not giving group name it create as exchangename.anonymous.64 bit random number.

If we not give the queue name in properties file, it is non durable (ie) once u stop the server, the queue will be removed from RabbitMQ. In case we give queue name in properties file, even if the server is stopped the queue will still exist, because it is durable

Step 3: Create producer application

1. Create SpringCloudStreamProducerRabbitMQ with spring web, cloud stream and spring for rabbit dependency
2. Use @EnableBinding to convert Springboot appl to cloud stream with Source which is output (ie) publish the message from appl to RabbitMQ
3. Create Employee model class
4. Create Controller class to access message using REST endpoints. So whenever we hit endpoint /emp we want to publish message to RabbitMQ.

To publish message to RabbitMQ we need to have access to MessageChannel which is provided by Source interface. It has output() which publish message to RabbitMQ. RabbitMQ will send the message of type Message and we use MessageBuilder to construct message, even we can add some headers and finally build() the message

1. Start the application, it will create only exchange “output” it wont create any queue for publishing message
2. Goto postman in POST give <http://localhost:8080/emp>

{ “name”:”sam”,”age”:24,”salary”:20000,”address”:”xyzstreet”}

In the console, it create new connection factory and publish the message to exchange

1. To see the message, Click output exchange, in To Queue: slect any queue
2. Restart the appl, in postman give different value
3. Now come to that queue, click Get message, you can see the message
4. We have created Source interface, instead we can create our own interface.

Create Orders interface, normally Source interface for output with MessageChannel and Sink interface for input with SubscriberChannel.

Here we create MessageChannel orders() with @Output through this exchange we can publish message from appl to RabbitMQ

Next we have SubscribableChannel payments() with @Input used to read message from RabbitMQ to our application, here we are not providing name to input so it will take method name

1. Now we pass Orders to @EnableBinding, so that it will create my exchanges and queues for the appl
2. Start the application, now we have application.output and application.orders as 2 channels. Now in RabbitMQ we can see orders, payments exchange

Create separate endpoint to publish the orders

1. Start the application
2. In Postman, POST url as <http://localhost:8080/emp/orders> with { “name”:”sam”,”age”:24,”salary”:20000,”address”:”xyzstreet”}
3. In RabbitMQ, click Orders exchange and u can spike it indicates the message is published to orders. If it is bounded to some queue we can see the message otherwise it will lost
4. Now the same message is consumed in Consumer part, for that we have destination property in application.properties file

spring.cloud.stream.bindings.orders.destination=testOrder

spring.cloud.stream.bindings.payments.destination=testOrder

spring.cloud.stream.bindings.payments.group=queue

So we publish the message to testOrder and retrieve the message in this testOrder.queue.

1. Add a subscriber in main class with @StreamListener
2. Restart the application, now we publish message in Postman with POST request with

{ “name”:”sam”,”age”:24,”salary”:20000,”address”:”xyzstreet”}

1. Goto RabbitMQ, testOrder exchange, we got spike means we publish the message, once the message is published using @StreamListner we can consume the message

**SPRING BOOT SCHEDULER**

**Most of time we want to run few tasks or services at regular interval or particular moment of time, so we need to have some instruction so that it triggers automatically using**

* @Scheduled
* @EnableScheduling
* @Async
* @EnableAsync

**@Scheduled**

An annotation that marks a method to be scheduled. Exactly one of the below attributes must be specified.

fixedDelay() – if we mention fixed delay of 1 min that task will executed at every 1 min, this 1 min interval starts from end of execution. The first task is completed after that 1 min the next task starts

fixedRate(), - it does not wait for previous task to complete

initDelay(),cron(), -> <http://www.cronmaker.com/> fixedDelayString(), fixedRateString(), initDelayString()

Some conditions/rules for method that is annotated @Scheduled

* No arguments, otherwise system will throw an exception
* No return type, if not then return type will be ignored

Note: Processing of @Scheduled annotations is performed by registering a ScheduledAnnotationBeanPostProcessor.

**@EnableScheduling**

Enables Spring's scheduled task execution capability. This enables detection of @Scheduled annotations on any Spring-managed bean in the container.

**@EnableAsync**

Enables Spring's asynchronous method execution capability.

**@Async**

* Annotation that marks a method as a candidate for asynchronous execution.
* Can also be used at the type level, in which case all of the type's methods are considered as asynchronous.
* In terms of target method signatures, any parameter types are supported.
* However, the return type is constrained to either
  + void or
  + java.util.concurrent.Future.

Example 1: Refer SpringBootScheduler-1 (<https://www.youtube.com/watch?v=uxV-3947fiM>)

We run one scheduler in each 5 sec which add data to database, similarly one more scheduler which data from db every 15sec, while retrieve we get 3 object as 5sec it add a object into database.

1. Create Spring boot appl with web,spring data jpa, mysql, Lombok
2. Create service, dao and model
3. Start the appl

You can generate the cron expression from https://www.freeformatter.com/cron-expression-generator-quartz.html

**Spring Boot Batch Processing**

1. **It is open source framework for batch processing, lots of enterprise appl using Spring batch for creating robust batch processing system which are light weight and pojo based.**
2. **It also provides reusable function for processing large volumes of data**
3. **It also includes other features like logging/tracing, transaction management, job processing statistics, restarting the job, skipping message inside batch processing and resource management**

**Spring Batch Architecture**

1. **Consider scheduler that runs inside our JVM which is going to trigger our Spring batch processing, in general we use spring scheduler or quartz scheduler. The scheduler launch JobLauncher which is class in Spring batch framework, it is a starting point for any particular job to be started inside spring batch**
2. **Job Launcher immediately triggers Job Repository which holds all statistical info of how many batches where run and what is the status of each batch, how many messages are processed or how many are skipped etc**
3. **Once Job Launcher triggers a Job repository, the Job Launcher has also have Job registered with this Job Launcher. This particular Job has a step, a Step basically consist of 3 different component inside Spring framework called ItemReader, ItemProcessor, ItemWriter, all these are useful when you want to read something from particular source, process that particular message and then write back to some other source. Most of the case these sources are either database or file system or queuing system**
4. **For example, reading a file, we will read the file using ItemReader, we process the data inside the file basically each data can be converted into pojo or it can transfer to some other object using ItemProcessor and finally using ItemWriter we can write back to database or publish that to a queue.**
5. **You can configure multiple steps inside the job but ItemReader, processor and writer can be one instance per step. All steps are enclosed within Step Exceution, whereas job execution happens at job level so if there are multiple step inside the job that is consider as JobExecution and each step has its own step execution**
6. **Once all steps are completed, the step status is updated back into Job repository and we can also get some statistics on how many messages have read, how many processed and how many failed or skipped etc**

**Example 1: Refer SpringBoot-BatchExample**

**We are going to read CSV file and write csv file back to database, we use ItemReader to read data from CSV file, use ItemProcessor to process the data and finally ItemWriter to write inside database**

1. **We provide dependency like spring batch, jdbc, mysql, xstream(used to serialize and deserialize objects, if we comment spring boot by default adds 1.4.7 but we need 1.4.9 so we added explicitly)**
2. **Person is model class with 4 properties**
3. **PersonItemProcessor implements ItemProcessor and overrides process() which used to write code into batch jobs according to interface. It receive person object and transform into uppercase**
4. **BatchConfiguration class**

**@EnableBatchProcessing adds many critical bean that supports jobs and steps work.**

**reader() – creates a ItemReader, it looks for file called person.csv and parses each line item with enough information to turn into person object**

**processor() – creates an instance of PersonItemProcessor which will process the person object**

**writer() – creates an ItemWriter, this end at JDBC destination and automatically gets the copy of datasource created by @EnableBatchProcessing. It includes SQL statement needed to insert single person driven by java bean properties**

**importUserJob()- defines the job, in job definition we need incrementer because job use database to maintain job execution step, we then list each step of job which has one step.**

Spring Batch has the rule that a JobInstance can only be run once to completion. This means that for each combination of identifying job parameters, you can only have one JobExecution that results in COMPLETE. A RunIdIncrementer will append an additional, unique parameter to the list of parameters so that the resulting combination would be unique...giving you a new JobInstance each time you ran the job with the same combination of identifying parameters.

**step1() – defines a single step. Jobs are build from the step where each step can involve reader, processor and writer.**

**In step definition we define how much data to write at a time, in this we write upto 10 records at a time using chunk(10).chunk() is prefixed with generic type which represent input and output types of each chunk of processing**

1. **JobCompletionNotificationListener class, this code listens for when job is completed and use JdbcTemplate to inspect the result.**
2. **When we run the appl, it creates Person table apart from that many table it creates to maintain the state of executed jobs**

**batch\_job\_seq, batch\_step\_execution\_seq, batch\_job\_execution\_seq – which maintenance the sequence, normally mysql dose not support to create sequence so that’s why spring batch creates separate table to generate primary key. This primary key. Other tables have primary key and spring batch retrieves from the corresponding sequence table.**

[**https://docs.spring.io/spring-batch/docs/3.0.x/reference/html/metaDataSchema.html**](https://docs.spring.io/spring-batch/docs/3.0.x/reference/html/metaDataSchema.html)

 For example, JobInstance, JobExecution, JobParameters, and StepExecution map to BATCH\_JOB\_INSTANCE, BATCH\_JOB\_EXECUTION, BATCH\_JOB\_EXECUTION\_PARAMS, and BATCH\_STEP\_EXECUTION, respectively. ExecutionContext maps to both BATCH\_JOB\_EXECUTION\_CONTEXT and BATCH\_STEP\_EXECUTION\_CONTEXT. The JobRepository is responsible for saving and storing each Java object into its correct table. 

**In maven dependency, expand spring-batch-core.jar – expand org.springframework.batch.core – you can see so many schema-\*.sql. Since we use mysql, u can see schema-mysql.sql which will have ddl queries to create all extra tables.**

**Most of the table have a column called version, to avoid concurrency issues we use version column.** This column is important because Spring Batch employs an optimistic locking strategy when dealing with updates to the database. This means that each time a record is 'touched' (updated) the value in the version column is incremented by one. When the repository goes back to try and save the value, if the version number has change it will throw OptimisticLockingFailureException, indicating there has been an error with concurrent access. This check is necessary since, even though different batch jobs may be running in different machines, they are all using the same database tables.

batch\_job\_instance is used to holds all info about job instance, which contains job\_name which comes from our project in BatchConfiguration.class as importUserJob, JOB\_KEY: A serialization of the JobParameters that uniquely identifies separate instances of the same job from one another. (JobInstances with the same job name must have different JobParameters, and thus, different JOB\_KEY values).

**Example 2: Refer SpringBootBatchScheduler – used to run the job for every 5 second and convert from db to csv**

**Example 3: Refer SpringBatchDbToXml – include spring-oxm dependency for XML**

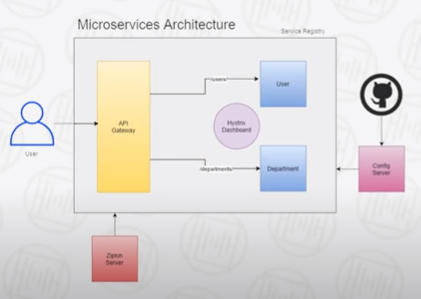
**Example 4: Refer MultiResourceItemReaderExample – used to read items from multiple CSV files**

**Example 5: Refer SpringBatchTasklet**

**Spring batch provides a Tasklet interface to process a single task like clean and slice resources before or after any step running.**

**MICROSERVICE ARCHITECTURE**

We will creating couple of microservice and we will create service registry and connect those microservice with service registry. We will create API Gateway so that all our request will be traverse through API Gateway. We will create microservice as resillent microservice using Hystrix circuit breaker, and also create distributed log tracing framework.

****

We will create 2 microservice as UserService and DepartmentService, every user will tag to department, so we save user from one service and department from another service and we create couple of methods to get department info and user info. We create methods to fetch user with its department, so we have a call from UserService to DepartmentService using REST API call.

Next we create service registry to register all the microservice created so we can get an idea what are the different microservice are available and the status of all microservices.

Next we create an API Gateway which act as an gateway for all API in our microservice, so all request should’nt be directly going to microservices, they should going from one API gateway and from there they should traverse according to microservice URL pattern.

Suppose if there is failure in any of the microservices (ie) DepartmentService is not working or UserService is not working and we are calling DepartmentService from UserService as well suppose if DepartmentService is not working we should get to know that DepartmentService is not working we should send message to user that DepartmentService is not working, for that we create timeout for particular API’s like if we are not getting response at particular time frame then we should call a fallback method. This fallback method will notify the user that particular microservice is not working. This is implement using Hystrix circuit breaker where we can also visualize how much my endpoints are failing and how much are getting successful.

Next we implement config server, considering we have multiple microservices and for all these microservices there might be some few configuration are repeated for all microservices, so its not best practice to all the configuration in all microservices. Suppose there is change in any one of them we need to go to all microservices and change it, so we create a cloud config server in which we create a GIT repository and we save all our configuration. We create cloud config server which will read that repository and give all the configuration to all the microservices.

Nest we create distributed logging, if we have multiple microservices we should able to identify which microservice is failing and where our request is traversing for that we use Zipkin and Sleuth library. Using Sleuth we can identify which are the services being called, what is trace id and span id.

**Step 1: DepartmentService Project**

1. Create DepartmentService with web,jpa,mysql dependency, created with controller,entity,repository and service, application.yml with just server port and application.properties with db info
2. Run the application
3. In postman with POST request run <http://localhost:9001/departments/>

With body – raw – JSON

{

    "departmentName":"IT",

    "departmentAddress":"Chennai",

    "departmentCode":"IT-306"

}

1. In Postman with GET request run <http://localhost:9001/departments/1> and check whether the data is fetched

**Step 2: UserService Project**

1. Create UserService with web,jpa,mysql dependency, created with application.yml with just server port and application.properties with db info
2. Create controller, entity, repository, service
3. Now we get the user along with department, so we create a wrapper object which contains both user and department. Here we create Department class over here and ResponseTemplateVO which contains both user and department object
4. Run the application
5. In Postman with POST request run <http://localhost:9002/users/> with body-raw-JSON

{

    "firstName":"Senthil",

    "lastName":"Kumar",

    "email":"senthil@gmail.com",

    "departmentId":1

}

1. In Postman with GET request run <http://localhost:9002/users/2> which returns the user with department

{

    "user": {

        "userId": 2,

        "firstName": "Senthil",

        "lastName": "Kumar",

        "email": "senthil@gmail.com",

        "departmentId": 1

    },

    "department": {

        "departmentId": 1,

        "departmentName": "IT",

        "departmentAddress": "Chennai",

        "departmentCode": "IT-306"

    }

}

So now user object and department object are related to each other but they have different microservice for their operations and calling each other using REST API call

**Step 3: ServiceRegistry Project**

We create a service registry to maintain all the microservice which is used to get status updates, port number and url information of all microservice which is running

1. Create ServiceRegistry project with eureka server dependency
2. In application.yml file configure

server:

port: 8761

eureka:

client:

register-with-eureka: false

fetch-registry: false

1. Define @EnableEurekaServer in main class
2. Now in UserService and DepartmentService configure Eureka Client dependency and configure @EnableEurekaClient in both main class
3. Configure application name and eureka registry information in UserService and DepartmentService application.yml
4. Change localhost to DEPARTMENT-SERVICE in UserService class to communicate from UserService to DepartmentService
5. Run ServiceRegistry, DepartmentService, UserService
6. Check all services are register in Eureka running on <http://localhost:8761>
7. In Postman with GET request check <http://localhost:9002/users/2>
8. You will be getting an error in order to avoid the error we use @LoadBalanced in UserServiceApplication
9. Now whenever we run <http://localhost:9002/users/2>, it will display the user and department information

**Step 4: API Gateway**

All the request coming from the user will come to API gateway first and from there all request will be traversed to related microservice

1. Create CloudGateway project with eureka client, actuator, cloud gateway routing
2. In application.yml we configure server port, spring application name, eureka client configuration and routing configuration (ie) all request have url pattern “/users” should be redirect to UserService and url with “/departments” redirect to DepartmentService
3. Run all the application
4. Now CloudGateway, DepartmentService, UserService will be registered in ServiceRegistry and check using <http://localhost:8761>
5. Now run the appl using 9191 through api gateway it traverse to user and department service
6. In Postman with POST request we add new department using <http://localhost:9191/departments/> with body-raw-json

{

  "departmentName":"HR",

    "departmentAddress":"Mumbai",

    "departmentCode":"HR-307"

}

1. Similary add new user in POST request with <http://localhost:9191/users/> with bdy-raw-json as

{

    "firstName":"Ram",

    "lastName":"Kumar",

   "email":"ram@gmail.com",

    "departmentId":2

}

1. Now get user with department with GET request using http://locahost:9191/users/3

**Step 5: Implementing Circuit Breaker**

Circuit breaker will identify which of the services are not running and will run the fallback methods available

1. In CloudGateway project add Netflix hystrix dependency, in main class enable hystrix using @EnableHystrix
2. Now we create FallbackController and whenever the service is down we will redirect to request to particular Rest services
3. Configure the fallback methods in application.yml using filters
4. So if we are not getting the response from either of the microservice within 4 sec then the request will be forwarded to either departmentservicefallback or userservicefallback method
5. We enable hystrix.stream so we can use this configuration in Hystrix dashboard
6. Create HystrixDashboard project with eureka client, hystrix dashboard
7. In application.yml file configure server port, spring appl name and eureka client configuration, hystrix dashboard configuration as proxy-stream-allow-list which allow all proxy stream in Hystrix dashboard
8. In main class configure @EnableEurekaClient, @EnableHystrixDashboard
9. Start all the 5 application and check <http://localhost:8761> whether all 4 services are registered in Eureka
10. Run <http://localhost:9295/hystrix> which opens hystrix dashboard in that provide <http://localhost:9191/actuator/hystrix.stream> which monitor all the streams
11. Now in Postman, add new department and new User in POST request using <http://localhost:9191/departments/> and <http://localhost:9191/users/> respectively. We can fetch the users using http://localhost:9191/users/2

You can check the hystrix dashboard where we get all info of user and department service

1. Now we stop department service, and in Postman if we hit the request <http://localhost:9191/users/1> it will invoke DepartmentService. Since it is down it will invoke the fallback method

If we hit department service using <http://localhost:9191/departments/1> in this case also it will invoke departmentservice fallback method since it is down

**Step 6: Implementing Cloud Config Server**

In all the microservice, we have common eureka client information which is configured in application.yml file. In case if there is any change in eureka client zone then we have to change in all microservices. So we need to create one configuration server and from there we can get all configuration and use in all microservices.

1. Create ConfigServer with eureka client, cloud config server dependency
2. In main class configure @EnableEurekaClient and @EnableConfigServer
3. In application.yml file define server port, spring application name
4. In GITHUB create a new repository called config-server which contains an application.yml which defines all common configuration used by all microservices (ie) in our case it is eureka client configuration
5. Now we add spring-cloud-starter-config in all microservices
6. Now we add configuration in application.yml so that all microservice will directly talk to the cloud server

But application.yml is used for application context, so whenever we start the application Spring boot will read all info from application.yml file. But for getting all the cloud configuration we use bootstrap.yml which is used to bootstrap the configuration and then starts the application context from application.yml

So we create bootstrap.yml in all microservice and put cloud configuration and remove eureka client info from all application.yml file

1. Start all the 6 application and check <http://localhost:8761> whether all 5 services are registered in Eureka
2. Now in Postman try to add new department and user and check whether it is working

**Step 7: Implementing Zipkin and Sleuth**

We have lot of microservice and one request will traverse to many application microservices, so trace them if any error, to trace all of those microservice is very difficult. For that we implement zipkin and sleuth logging which is used to handle all distributed log traces. So we can identify which services is been called and what is the trace id which is unique across all the microservices for one request. We get span id which will change according to each and every microservice.

So trace id will remain same and for that traceid whatever microservice we are traversing that span id will change accordingly. So we can get to know where the request starts, from where the request ends, where was failure and success we can get from distributed log trace.

1. Start Zipkin server using

* Java -jar zipkin.jar

Which started in http://127.0.0.1:9411/

1. Now we want to implement zipkin client and sleuth in UserService and DepartmentService because for that only we added logger. Add zipkin client and sleuth dependency in UserService and DepartmentService
2. Now we add zipkin url in UserService, DepartmentService application.yml

zipkin:

base-url: <http://127.0.0.1:9411/>

1. Start all the application and check whether it is configured in eureka
2. In Postman try to add new department and user in POST request
3. Go to <http://localhost:9411/zipkin> and u can see the flow of service with service name

Resilience 4J

- It is a lightweight fault tolerance library inspired by Netflix Hystrix, but designed for Functional Programming

- There are many transient dependency that comes with Netflix Hystrix, but Resilience 4J uses Vavr only.

- Vavr formerly called Javaslang is a functional library for Java 8+ that provides persistent data types and functional control structure

- Resilience4j provides higher order functions called decorators to enhance any functional interface, lambda expr or method reference with

1. Circuit breaker

2. Rate Limiter

3. Retry

4. Bulkhead

So Hysrix provide only circuit breaker but with resilience we have other features also.

Circuit Breaker

We have 2 services S1 and S2 and both are communicating with each other. We have circuit breaker, if it is closed state the communication is happening properly and if it is opened the communication is not happening

If we are calling 3rd party service we need to store the state so Resilience4j used Ring Bit Buffer in closed state to store the success or failure status of function calls

Success function call - 0

Failure function call – 1

BulkHead

When u r hitting 3rd party service there are limits of number of request that hitting 3rd party service so that is controlled by bulkhead. For eg, if we want 10 concurrent request happen to 3rd party service.

It is provided with 2 implementation

1. A SemaphoreBulkhead which uses Semaphores
2. A FixedThreadPoolBulkhead which uses bounded queue and a fixed thread pool

Ratelimiter

Consider we have service which is publically open and we have client which is hitting that service unlimitedly. So if we don’t any control over how many request it can receive per second. So we should have some feature to handle that limit which can be done using Resilience4j ratelimiter. So if we configure as 100 request/second then we provide 200 request, it will take only 100 request and remaining request will be discarded

So rate limiting is used to control the rate of traffic sent or received by a network interface controller. It is used to prevent DOS attack

Retry

Consider client is hitting the service and suppose circuit is in open state due to number of failures. Now circuit breaker makes the hit to 3rd party to retry to check up or not so that circuit can be closed. How many request should send per second to check whether health is good or not that can be configured in retry. For eg, if we give 2 request/sec then circuit breaker per sec it will hit 2 request to 3rd party to check if it is up or not. If it is up it will close the circuit

The number of times the call made to 3rd party to check if its health is good or not

Spring Boot + Resilience 4j

Circuit breaker has 3 states,

1. Closed state – when everything is normal so circuit breaker remains in closed state and all calls pass through the services
2. Open state – when failure rate goes above threshold so the breaker will goes to the open state. So in open state circuit breaker returns a error for all calls with out executing the function
3. Halfopen state – After timeout period the circuit switches to half open state to check underlying problem still exist or not. If failure rate is above threshold again from half open it goes to open state. If succeed in half open state then it send back to closed state

Fallback method should have same signature as ur registerSeller() has (ie)same return type, same arguments. It can take one extra parameter called Throwable, so if any exception occur in registerSeller() that can get inside fallbackmethod using Throwable t

@CircuitBreaker(name = "service1", fallbackMethod = "fallbackForRegisterSeller")

public String registerSeller(SellerDto sellerDto) {

String response = restTemplate.postForObject("/addSeller", sellerDto, String.class);

return response;

}

Inside the method we make 3rd party call (ie) any expensive operation, database operation or any thread call which takes too much time and above method we give @CircuitBreaker which provide with name and fallbackmethod. If we have circuitbreaker it have by default configuration what is time out, what are the no of calls etc will be by default provided by Resilience4j. If we want to do custom configuration in application.yml

resilience4j.circuitbreaker:

instances:

service1: -- circuit breaker name

registerHealthIndicator: true – If we put health indicator true, then we check the status of the appl all the details will be there

ringBufferSizeInClosedState: 5 – When there are total 5 number of calls then it will determine whether it is closed or not

ringBufferSizeInHalfOpenState: 3 – 3 calls will be tried, then the state will be changed

waitDurationInOpenState: 10s -If circuit state is changed from closed to open state so it will wait for 50 sec to change the state

failureRateThreshold: 50 – If there are 10 calls allowed so 5 calls are failing only then circuit breaker state will change from close to open

Step 1: Resilience CircuitBreaker

1. Create Resilience-OrderService with web,actuator,aop, resilience-spring-boot2 dependency
2. Create Resilience-ItemService with web dependency
3. Run both application
4. We give request from OrderService to ItemService
5. Run <http://localhost:8082/order>
6. Run <http://localhost:8082/actuator/health>

Will display the health of the application where the state is closed

1. Now hit <http://localhost:8082/order> for 5 times and when u check

<http://localhost:8082/actuator/health> now the state will half open because we specify minimum no of calls to 5

1. Now we start once again ItemService and run <http://localhost:8082/order>
2. Run <http://localhost:8082/actuator/health> again status is halfopen because we have buffercall as 1
3. Run <http://localhost:8082/order> 3 times
4. Run <http://localhost:8082/actuator/health> again state is changed to closed which means it allow all calls from Itemservice

Step 2: Spring Boot Resilience4j – Retry implementation

1. We call orderservice, this orderservice internally calls itemservice using rest endpoint. In service based architecture, if one service is down for particular time or slow response due to large number of request towards that resource

How we can reattempt automatically from orderservice to hit itemservice by limit number of times to hit this item service

1. In application.yml we provide

maxRetryAttempts:3 – 3 times it will retry

waitDuration : 11000 – fixed wait duration between retry attempts, so within 11000 ms, it has to attempt 3 times

1. Next we define @Retry with fallback method
2. Start OrderService application
3. Run <http://localhost:8082/order>, it will go to fallback method, since Itemservice is down
4. It will print “Item service is down” after 3 attempts is completed. You can see the console for attempt details
5. Now start ItemService application
6. Run <http://localhost:8082/order>, it will print “Item Selected”

Step 3: Bulkhead implementation

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

Consider we have ship which is divided into multiple compartments called bulkheads, the bulkheads are used to seal the part of ship to prevent the ship from sinking in case of flood. Similarly a failure is expected when we design a software so appl should be split into multiple components and resource should be isolated in such a way that so failure of one component is not affecting to the other

When to use bulkhead pattern ?

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

Bulkhead with Resilience4j – 2 types

1. Semaphore – In this approach, we limit the number of concurrent requests to the service. It will reject the incoming requests once the limit is hit
2. FixedThreadPoolBulkHead – In this approach, we isolate a set of thread pool from system resources, using only that thread pool for the service. We also use a waiting queue apart from the thread pool, if both the thread pool and queue are full then the request will get rejected with BulkHeadFullException

Using Semaphore

1. OrderService internally calls ItemService endpoint
2. Now we define @Bulkhead with name and fallbackMethod
3. In application.yml, we provide bulkhead details

maxWaitDuration: 5000 – max amount of time a thread would be blocked when attempting to enter a saturated bulkhead

maxConcurrentCalls: 5 – max amount of parallel execution allowed by bulkhead (ie) 5 calls

1. Start ItemService appl
2. Start OrderService appl, here we call concurrently 10 calls using Streams, so OrderService we are calling concurrently 10 times
3. Now in the console or OrderService, we will call ItemService 10 times successfully and we didn’t get any exception because in orderservice we provide 5000 msec and in Itemservice we provide thread sleep to 4000 msec
4. Now in maxWaitDuration we provide as 1000, maxConcurrentCalls: 2 and Thread.sleep(1000);
5. Start ItemService and then OrderService appl

This time it will throws exception, this time for 1000msec it allows only few calls and for other calls it should go to fallbackmethod

Using FixedThreadPoolBulkhead

1. Now in maxWaitDuration we provide as 5000, maxConcurrentCalls: 5 and Thread.sleep(4000);

|  |
| --- |
| resilience4j.thread-pool-bulkhead: |
|  | instances: |
|  | orderService: |
|  | maxThreadPoolSize: 1 |
|  | coreThreadPoolSize: 1 |
|  | queueCapacity: 1 |

It wont go to fallback method until maxThreadPoolSize and queue capacity is full

Step 4: Ratelimiter implementation

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

If the client exceeds this limit, further requests can either be declined, be processed in a later period or be serviced by allocating a smaller amount of resources or by providing only best effort guarantees.

Rate limiting specified in terms of

1. Requests per second
2. Requests per minute
3. Requests per hour

2 types of Rate limiter

1. Client side rate limiter – we use resilience4j-ratelimiter. If we want to use resilience4j-ratelimiter, we need to configure in application.yml file like

timeoutDuration: 5 – The default wait time a thread waits for a permission

limitRefreshPeriod: 500ns – The period of a limit refresh

limitForPeriod: 50 – The number of permissions available during one limit refresh period

It means we send 50 request to the services within 500 nanosec

1. Server side rate limiter – we use Kong API gateway and Response API filter
2. OrderService internally calls ItemService
3. In OrderController we add @RateLimiter which have name and fallbackmethod
4. In application.yml we need to provide ratelimiter configuration

limitRefreshPeriod : within 4sec it will allow only 2 calls

1. Start ItemService appl
2. Start OrderService appl
3. If we want to send concurrent request to the service we can use Apache Jmeter5.4.1
4. Download and Install Jmeter5.4.1, go to bin – click jmeter.bat
5. Jmeter dashboard opens
6. Right click Test Plan – Add – Thread(Users) – Thread Group

Number of Threads – 20 - how many threads concurrently hit to orderservice

1. Right click Thread Group – Add – Sampler – Http Request

Server Name: localhost port number: 8082

Path: /order

1. Right Http Request – Add – Listener – View Result tree
2. In Jmeter – click Start button – Click Yes for saving
3. Here u can see only 2 request will be successful and if click on the request u can see response data
4. If we click on failure request it will invoke the data from fallback method
5. Now application.yml we change timeoutDuration: 10s
6. Restart OrderService appl
7. Click start in Jmeter
8. We can see for each set of request it will wait for 10sec and after 10sec it will go to fallback method

Step 5: Resilience4j Time limiter implementation

Setting a limit on the amount of time we are willing to wait for an operation to complete is called time limiting

If the operation does not complete within the time specified, we want to notified about it with a timeout error

Resilience4j Timelimiter can be used to set time limits(timeouts) on asynchronous operations implemented with CompleteableFuture.

In real time appl, if we are invoking database services and if we are not getting response after certain amount of time in such time of scenarios we use timelimiter

Resilience4j Time Limiter Configuration

timeoutDuration: 2s

cancelRunningFuture: true – whether cancel should be called on the running future

1. In OrderController, we will create a method called createOrder() which calls SlowService method called slowMethod() and we define @TimeLimiter with name
2. In slowMethod() we specify some amount of time where thread should wait using Thread.sleep()

First comment try catch block

1. Start ItemService appl
2. Start OrderService appl
3. Download and Install Jmeter5.4.1, go to bin – click jmeter.bat
4. Jmeter dashboard opens
5. Right click Test Plan – Add – Thread(Users) – Thread Group

Number of Threads – 20 - how many threads concurrently hit to orderservice

1. Right click Thread Group – Add – Sampler – Http Request

Server Name: localhost port number: 8082

Path: /order

1. Right Http Request – Add – Listener – View Result tree
2. In Jmeter – click Start button – Click Yes for saving
3. Now we get timeout error in the orderservice console
4. Now we remove the comment of try catch block in SlowService class
5. Restart OrderService appl
6. Click Start button in JMeter once again
7. Now we can see few success calls and other calls are failured
8. Now we make Thread.sleep(1000) in SlowService class
9. Restart OrderService appl
10. Click Start button in JMeter once again
11. This time we can see more successful calls