**Spring Boot**

First Spring Boot is not a framework, it is a way to ease to create stand-alone application with minimal or zero configurations. It is approach to develop spring based application with very less configuration. It provides defaults for code and annotation configuration to quick start new spring projects within no time

Spring Boot automatically configures required classes depending on the libraries on its classpath. Suppose your application want to interact with DB, if there are Spring Data libraries on class path then it automatically sets up connection to DB along with the Data Source class.

Spring boot enabled building production read based application quickly, and provides many non-functional features,

* Embedded servers, which are easy to deploy with the containers
* It helps in monitoring the multiples components.
* It helps in configuring the components externally.

Advantages:

* It is very easy to develop Spring Based applications with Java or Groovy.
* It reduces lots of development time and increases productivity.
* It avoids writing lots of boilerplate Code, Annotations and XML Configuration.
* It is very easy to integrate Spring Boot Application with its Spring Ecosystem like Spring JDBC, Spring ORM, Spring Data, Spring Security e
* Absolutely no code generation and no requirement for XML configuration, to avoid XML Configuration completely
* To avoid defining more Annotation Configuration(It combined some existing Spring Framework Annotations to a simple and single Annotation)
* Spring Boot avoid writing lots of import statements
* Spring Boot comes with inbuilt server, we no longer have to use any external servers like Tomcat, Glass-fish or anything else, so don’t need to deploy WAR files

**Disadvantage:**

It will be little tough to migrate existing spring enterprise applications to Spring Boot.

**Spring Boot Project Creation Types:**

There are multiple approaches to create Spring Boot project. We can use any of the following approach to create application.

* Spring Maven Project
* Spring Starter Project Wizard
* Spring Initializr
* Spring Boot CLI

|  |
| --- |
| <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>com.java4s</groupId>  <artifactId>SpringBootHelloWorld</artifactId>  <version>0.0.1-SNAPSHOT</version>  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.6.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  </dependencies>  <properties>  <java.version>1.8</java.version>  </properties>  </project> |

 I haven’t included version number for spring-boot-starter-web :-) but maven downloaded some jar files with some version(s) related to spring-boot-starter-web, how its possible? that’s because of Maven’s parent child relation. While adding spring boot parent project, I have included version as 1.5.6.RELEASE, so again we no need to add version numbers for the dependencies.  As I told you earlier, spring-boot-starter-parent contains configuration meta data, this means, it knows which version of dependency need to be downloaded.

spring-boot-starter-parent contains configuration meta data, this means, it knows which version of dependency need to be downloaded.

**Added server port and server context path:**

server.port=2017

server.servlet.context-path=/DemoBoot

Add Datasource Information in application.properties without XML file:

# Here 'test' is the database name

spring.datasource.url=jdbc:mysql://localhost/test

spring.datasource.username=java4s

spring.datasource.password=java4s

spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver

**YAML:**

YAML stands for YAML Ain't Markup Language.

YAML is human friendly data serialization standard for all programming languages.

It supports for all languages.

YAML is an indentation-based markup language which aims to be both easy to read and easy to write. Many projects use it because of its readability, simplicity and good support for many programming languages.

**Spring Cloud**

Microservices is a form of service-oriented architecture style and we can developing single, small, meaningful functional feature as single service, each service has it’s own process and communicate with lightweight mechanism, deployed in single or multiple servers.

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

Microservices focus on a single business domain that can be implemented as fully independent deployable services and implement them on different technology stacks.

**Advantages of microservice architecture?**

* Each micro service is small and focused on a specific feature / business requirement.
* Microservice can be developed independently by small team of developers (normally 2 to 5 developers).
* Microservice is loosely coupled, means services are independent, in terms of development and deployment both.
* Microservice can be developed using different programming language (Personally I don't suggest to do it).
* Microservice allows easy and flexible way to integrate automatic deployment with Continuous Integration tools (for e.g: [Jenkins](http://jenkins-ci.org/), [Hudson](http://hudson-ci.org/), [bamboo](https://www.atlassian.com/software/bamboo) etc..)
* The productivity of a new team member will be quick enough.
* Microservice is easy to understand, modify and maintain for a developer because separation of code,small team and focused work.
* Microservice allows you to take advantage of emerging and latest technologies (framework, programming language , programming practice, etc.).
* Microservice has code for business logic only, No mixup with HTML,CSS or other UI component.
* Microservice is easy to scale based on demand.
* Microservice can deploy on commodity hardware or low / medium configuration servers.
* Easy to integrate 3rd party service.
* Every microservice has it's own storage capability but it depends on the project’s requirement, you can have common database like MySQL or Oracle for all services.

**Benefits:**

* **Independent Development** – All microservices can be easily developed based on their individual functionality
* **Independent Deployment** – Based on their services, they can be individually deployed in any application
* **Fault Isolation** – Even if one service of the application does not work, the system still continues to function
* **Mixed Technology Stack** – Different languages and technologies can be used to build different services of the same application
* **Granular Scaling**–  Individual components can scale as per need, there is no need to scale all components together

**Microservices vs. Monolithic Architecture:**

|  |  |
| --- | --- |
| **Microservices** | **Monolithic Architecture** |
| Service Startup is relatively quick | Service startup takes more time |
| Fault isolation is easy. Even if one service goes down, other can continue to function. | Fault isolation is difficult. If any specific feature is not working, the complete system goes down. In order to handle this issue, the application needs to re-built, re-tested and also re-deployed. |
| All microservices should be loosely coupled so that changes made in one does not affect the other. | Monolithic architecture is tightly coupled. Changes in one module of code affect the other |
| Small Focused Teams. Parallel and faster development | Large team and considerable team management effort is required |
| Change in the data model of one Microservice does not affect other Microservices. | Change in data model affects the entire database |

**Spring Cloud:**

Spring Cloud helps you build cloud-native applications, cloud-native application means that your application was specifically built(made) and engineered(planed) for the cloud. It means your application fully utilizes all of the cloud computing paradigms (patterns). Spring Cloud itself is not actually a framework. Loosely speaking, Spring Cloud is used to describe a number of projects that all fall under the same umbrella

**Netflix** is the world’s leading subscription service for watching TV episodes and movies on favorite device.

The company's primary business is its subscription-based streaming [OTT](https://en.wikipedia.org/wiki/Over-the-top_media_services) service which offers online streaming of a library of films and television programs.

**What is Eureka?**

In simple word Eureka is a service Registry or we can say it is an embedded server provided by Netflix third party which integrate with spring framework.

**Main purpose to use Eureka** : Micro service Registration and Discovery with Spring Cloud and Netflix's Eureka

**Spring cloud tools**

* Service Discovery (Eureka)
* Api gateway (Zuul)
* Circuit Breaker (Hystrix)
* Client-Side Load Balancing (Ribbon)
* Config Server
* Spring bus
* Distributed Tracing (Sleuth with Zipkin), Spring Cloud Bus
* Rest Client (Feign)

**Eureka Server (Discovery Service)**

**Discovery service** is an application that holds the information about all client-service applications. Every Micro service will register into the Eureka server and Eureka server knows all the client applications running on each port and IP address. Eureka Server is also known as Discovery Server.

Service discovery provides a way to check the health of a service and remove any unhealthy instances

When a client micro service registers with Eureka it provides metadata such as host, port, and health indicator thus allowing for other microservices to discover it. The discovery server expects a regular heartbeat message from each microservice instance. If an instance begins to consistently fail to send a heartbeat, the discovery server will remove the instance from his registry.

The **@EnableEurekaServer** annotation is used to make your Spring Boot application acts as a Eureka Server.

eureka.client.registerWithEureka=false

eureka.client.fetchRegistry=false

server.port=8761

**eureka.client.registerWithEureka=false**

By default, the Eureka Server registers itself into the discovery, because our application should be acting as a server. o you’ll need to disable that, as well.

@EnableEurekaClient annotation in the main Spring Boot application class file. The @EnableEurekaClient annotation makes your Spring Boot application act as a Eureka client

Client side:

eureka.client.serviceUrl.defaultZone= http://localhost:8761/eureka

**Eureka Server Example:**

|  |
| --- |
| **import**org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;  @SpringBootApplication  @EnableEurekaServer  **publicclass**DiscoveryServiceApplication {  //http://localhost:8761/  **publicstaticvoid**main(String[] args) {  SpringApplication.*run*(DiscoveryServiceApplication.**class**, args);  }  }  **Properties file:**  server:  port: ${PORT:8761}  eureka:  instance:  hostname: localhost  client:  registerWithEureka: false  fetchRegistry: false  server:  enableSelfPreservation: false  spring:  application:  name: discovery-service  zipkin:  base-url: http://localhost:9411/  sleuth:  sampler:  probability: 1  logging:  pattern:  console: "%clr(%d{yyyy-MM-dd HH:mm:ss}){faint} %clr(${LOG\_LEVEL\_PATTERN:-%5p}) %clr([${springAppName:-},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-Span-Export:-}]){yellow} %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG\_EXCEPTION\_CONVERSION\_WORD:-%wEx}"  file: /home/nagendra/logs/discovery-service.log  level:  org.springframework: WARN  org.hibernate: WARN |

**Client Application:**

|  |
| --- |
| **import**org.springframework.cloud.client.circuitbreaker.EnableCircuitBreaker;  **import**org.springframework.cloud.client.discovery.EnableDiscoveryClient;  **import**org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;  @SpringBootApplication  @EnableDiscoveryClient  @EnableCircuitBreaker  @EnableHystrixDashboard  **publicclass**AccountServiceApplication {  //http://localhost:2222/accounts  //http://localhost:2222/  **publicstaticvoid**main(String[] args) {  SpringApplication.*run*(AccountServiceApplication.**class**, args);  }  }  **Properties File**  server:  port: ${PORT:2222}  spring:  application:  name: account-service  zipkin:  base-url: http://localhost:9411/  sleuth:  sampler:  probability: 1  logging:  pattern:  console: "%clr(%d{yyyy-MM-dd HH:mm:ss}){faint} %clr(${LOG\_LEVEL\_PATTERN:-%5p}) %clr([${springAppName:-},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-Span-Export:-}]){yellow} %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG\_EXCEPTION\_CONVERSION\_WORD:-%wEx}"  file: /home/nagendra/logs/account-service.log  level:  org.springframework: WARN  org.hibernate: WARN  eureka:  client:  serviceUrl:  defaultZone: http://localhost:8761/eureka/  instance:  leaseRenewalIntervalInSeconds: 1  leaseExpirationDurationInSeconds: 2  ribbon:  eureka:  enabled: true  management:  endpoints:  web:  exposure:  include: hystrix.stream, info, health |

**Api Gateway [Zuul]**

**Zuul** is the front door for all requests from devices and web sites to the backend of the Netflix streaming application. As an edge service application, Zuul is built to enable dynamic routing, monitoring, resiliency and security.

**Zuul Server** is a gateway application that handles all the requests and does the dynamic routing of microservice applications. The Zuul Server is also known as Edge Server.

For Example, /api/user is mapped to the user service and /api/products is mapped to the product service and Zuul Server dynamically routes the requests to the respective backend application

@EnableZuulProxy annotation on your main Spring Boot application. The @EnableZuulProxy annotation is used to make your Spring Boot application act as a Zuul Proxy server

Zuul has mainly four types of filters that enable us to intercept the traffic in different timeline of the request processing for any particular transaction. We can add any number of filters for a particular url pattern.

* **pre filters** – are invoked before the request is routed.
* **post filters** – are invoked after the request has been routed.
* **route filters** – are used to route the request.
* **error filters** – are invoked when an error occurs while handling the request.

|  |
| --- |
| import javax.servlet.http.HttpServletRequest;  import com.netflix.zuul.ZuulFilter;  import com.netflix.zuul.context.RequestContext;    public class PreFilter extends ZuulFilter {      @Override    public String filterType() {      return "pre";    }      @Override    public int filterOrder() {      return 1;    }      @Override    public boolean shouldFilter() {      return true;    }      @Override    public Object run() {      RequestContextctx = RequestContext.getCurrentContext();      HttpServletRequest request = ctx.getRequest();        System.out.println("Request Method : " + request.  RequestContext:  Request context holds request, response, and state and data information. It will be shared all filters  RequestContextctx = RequestContext.getCuurentcontext()  //get the servlet request  HttpServletRequest req = ctx.getRequest();  //Get the servlet response  HttpServletResponse res = ctx.getResponse()  //Get a variable  String foobar = (String) ctx.get(“foobar”);  () + " Request URL : " + request.getRequestURL().toString());      return null;    }  }  **Register all Filter in Bean Configuration file**  @Bean  public PreFilterpreFilter() {      return new PreFilter();  }  @Bean  public PostFilterpostFilter() {      return new PostFilter();  }  @Bean  public ErrorFiltererrorFilter() {      return new ErrorFilter();  }  @Bean  public RouteFilterrouteFilter() {      return new RouteFilter();  }  **Application:**  @SpringBootApplication  @EnableZuulProxy  **publicclass**GatewayServiceApplication {  **publicstaticvoid**main(String[] args) {  SpringApplication.*run*(GatewayServiceApplication.**class**, args);  }  }  **Properties file:**  info:  component: Edge Server  spring:  application:  name: gateway-service  zipkin:  base-url: http://localhost:9411/  sleuth:  sampler:  probability: 1  endpoints:  restart:  enabled: true  shutdown:  enabled: true  health:  sensitive: false  zuul:  prefix: /api  routes:  account:  path: /account/\*\*  serviceId: account-service  customer:  path: /customer/\*\*  serviceId: customer-service  ribbon:  eureka:  enabled: true  eureka:  client:  serviceUrl:  defaultZone: http://localhost:8761/eureka/  registerWithEureka: false  server:  port: 8765  logging:  pattern:  console: "%clr(%d{yyyy-MM-dd HH:mm:ss}){faint} %clr(${LOG\_LEVEL\_PATTERN:-%5p}) %clr([${springAppName:-},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-},%X{X-Span-Export:-}]){yellow} %clr(${PID:- }){magenta} %clr(---){faint} %clr([%15.15t]){faint} %clr(%-40.40logger{39}){cyan} %clr(:){faint} %m%n${LOG\_EXCEPTION\_CONVERSION\_WORD:-%wEx}"  file: /home/nagendra/logs/gateway-service.log  level:  org.springframework: WARN  org.hibernate: WARN |

**Circuit Breaker:**

In microservices-based architecture, one service might depend on another service, and if one service goes down, then failures may cascade to other services as well. Spring Cloud provides a Netflix Hystrix-based Circuit Breaker to handle these kinds of issues.

Netflix created Hystrix library implementing the [Circuit Breaker pattern](https://martinfowler.com/bliki/CircuitBreaker.html) to address these kinds of issues. We can use Spring Cloud Netflix Hystrix Circuit Breaker to protect microservices from cascading failures.

In this post, we are going to learn:

* Implementing Circuit Breaker pattern using @HystrixCommand
* How to propagate ThreadLocal variables
* Monitoring Circuit Breakers using Hystrix Dashboard

Hystrix can redirect the call to an internal fallback method in the service consumer. If a service repeatedly fails to respond, Hystrix will open the circuit and fast fail (i.e. call the internal fallback method without trying to call the service) on every subsequent call until the service is available again.

@EnableHystrix annotation into your main Spring Boot application class file. The @EnableHystrix annotation is used to enable the Hystrix functionalities into your Spring Boot application

**Hystrix configuration is done in four major steps.**

1. Add Hystrix starter and dashboard dependencies.

|  |
| --- |
| **<dependency>**  **<groupId>org.springframework.cloud</groupId>**  **<artifactId>spring-cloud-starter-hystrix</artifactId>**  **</dependency>**  **<dependency>**  **<groupId>org.springframework.cloud</groupId>**  **<artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>**  **</dependency>** |

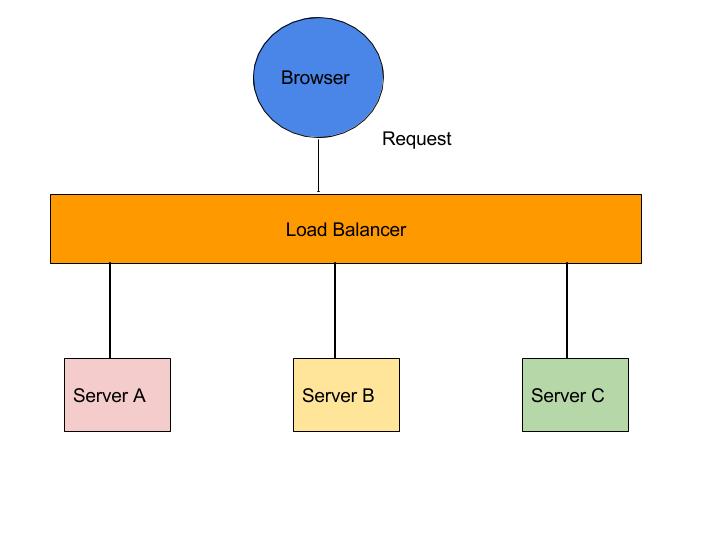
|  |
| --- |
| Main class  @SpringBootApplication  @EnableDiscoveryClient  @EnableFeignClients  @EnableCircuitBreaker  **publicclass**CustomerServiceApplication {  //http://localhost:3333/  **publicstaticvoid**main(String[] args) {  SpringApplication.*run*(CustomerServiceApplication.**class**, args);  }  }  F**eign client and circuit breaker and ribben client**  @FeignClient(name ="account-service", fallback = AccountFallback.**class**)  @RibbonClient("account-service")  **publicinterface**AccountClient {  @RequestMapping(method = RequestMethod.***GET***, value = "/accounts/customer/{customerId}")  **public** List<Account>getAccounts(@PathVariable("customerId") Integer customerId);  @RequestMapping(value = "/accounts/elkdemo")  **public** String helloWorld();  @RequestMapping(value = "/accounts/exception")  **public** String exception();  @RequestMapping("/accounts/ribbon")  **public** String backend();  }  Circute Breaker implementation  @Component  **publicclass**AccountFallback**implements**AccountClient {  @Override  **public** List<Account>getAccounts(Integer customerId) {  List<Account>acc = **new**ArrayList<Account>();  **return**acc;  }  @Override  **public** String helloWorld() {  **return**"fall back helloWorld() is executed due to account service down";  }  @Override  **public** String exception() {  **return**"fall back exception() is executed due to account service down";  }  @Override  **public** String backend() {  **return**"fall back backend() is executed due to account service down";  }  } |

**Load balancing**

**Load balancing** its simple like balaning the request and forwarded to appropriate server for handling the request

## **Server Side Load Balancing**

In Java EE architecture, we deploy our war/ear files into multiple application servers, then we create a pool of server and put a load balancer (Netscaler) in front of it, which has a public IP. The client makes a request using that public IP, and Netscaler decides in which internal application server it forwards the request by round robin or sticky session algorithm. We call it server side load balancing.

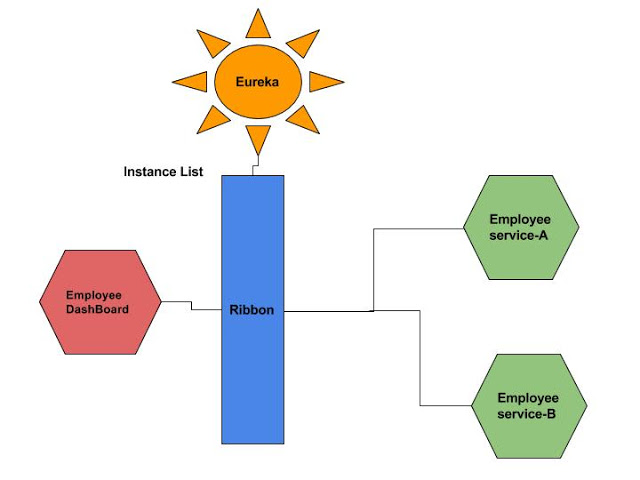


**Problem:** The problem of server side load balancing is if one or more servers stop responding, we have to manually remove those servers from the load balancer by updating the IP table of the load balancer.

## **Client Side Load Balancing**

To understand client side load balancing, let's recap microservices architecture. We generally create a service discovery like Eureka or Consul, where each service instance registers when bootstrapped. Eureka server maintains a service registry; it maintains all the instances of the service as a key/value map, where the {service id} of your microservice serves as the key and instances serve as the value. Now, if one microservice wants to communicate with another microservice, it generally looks up the service registry using DiscoveryClient and Eureka server returns all the instances of the calling microservice to the caller service. Then it was a caller service headache which instance it calls. Here, client side load balancing stepped in. Client side load balancing maintains an algorithm like round robin or zone specific, by which it can invoke instances of calling services. The advantage is s service registry always updates itself; if one instance goes down, it removes it from its registry, so when the client side load balancer talks to the Eureka server, it always updates itself, so there is no manual intervention- unlike server side load balancing- to remove an instance.

Another advantage is, as the load balancer is in the client side, you can control its load balancing algorithm programmatically. Ribbon provides this facility, so we will use Ribbon for client side load balancing.

[](http://2.bp.blogspot.com/-Zwwt2dm41Z4/WY3cD5TbMCI/AAAAAAAAFs4/1vif-nIoHdowD_C4ZLJSoOy2qvMiDmo2wCK4BGAYYCw/s1600/Microservices+Communication_+Ribbon+as+a+Load+balancer(1).jpg)

**FiegnClient**

**What Is a Feign Client?**

Netflix provides Feign as an abstraction over REST-based calls, by which microservices can communicate with each other, but developers don't have to bother about REST internal details.

Feign, as a client, is an important tool for microservice developers to communicate with other microservices via Rest API.

Feign uses interfaces annotated with @FeignClient to generate API requests and map responses to Java classes. Let's first compile a list of API calls:

**Action HTTP Method URL**

(C) Create a user POST https://jsonplaceholder.typicode.com/users

(R) Get a list of users GET https://jsonplaceholder.typicode.com/users

(R) Get one user GET https://jsonplaceholder.typicode.com/users/{id}

(U) Update user details PUT or PATCH https://jsonplaceholder.typicode.com/users/{id}

(D) Delete a user DELETE [https://jsonplaceholder.typicode.com/users/{id}](https://jsonplaceholder.typicode.com/users/%7bid%7d)

|  |
| --- |
| **Implementation steps :**  Creating a interface adding abstract methods and add the @FeignClientannotations ,  Fegin client internally uses ribben for client side load balancing.  Ribben is get the all services information from eureka server, ribben forward the requests to appropriate server  @FeignClient(name="UserService" )//Service Id of EmployeeSerach service  **publicinterface**UserClient {  @RequestMapping(method = RequestMethod.POST, value = "/users")  UsercreateUser(Useruser);  @RequestMapping(method = RequestMethod.GET, value = "/users")  List<user>getUsers();  @RequestMapping(method = RequestMethod.GET, value = "/users/{userId}")  UsergetUser(@PathVariable("userId") Long userId);  @RequestMapping(method = RequestMethod.PUT, value = "/users/{userId}")  UserupdateUser(@PathVariable("userId") Long userId, Useruser);  @RequestMapping(method = RequestMethod.DELETE, value = "/users/{userId}")  **void**deleteUser(@PathVariable("userId") Long userId);  } |

**Config Server:**

Spring Cloud Configuration Server is a centralized application that manages all the application related configuration properties.

the @EnableConfigServer annotation in your main Spring Boot application class file. The @EnableConfigServer annotation makes your Spring Boot application act as a Configuration Server

**Spring Cloud Bus**

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.

The better approach is to trigger the refresh event for one service and broadcast the event through all other available services.  This sounds good, we are going to explore a way to trigger the **refresh event** for only one service and that event is automatically propagated (broadcasted) through all the other services. This can be achieved with **Spring Cloud Bus**.

Once the user **triggers** the**refresh event** for any service (here it is **department-service**), the **Spring Cloud Bus** will receive the **refresh event**. Then it will broadcastthe **refresh event** across **all the connected clients** through the underlying message broker (e.g:- **RabbitMQ**).

The **refresh event** is triggered by invoking the endpoint  */actuator/bus-refresh*of the **department-service**.

http://localhost:8081/actuator/bus-refresh

**The property file naming rule**

Since all the property files related to different services are placed in a single location, there should be a way to distinguish them among services. Otherwise the Spring Cloud Config Server will have to face for a problem when picking up the correct configuration file for the service. This can be achieved with name of the property file. The property file should be named based on following rules.

* The name should start with the application name as declared in the relevant service.

**e.g:-  spring.application.name = department-service**

* If there different profiles, the profile name should comes after the application name.

**e.g:- department-service-uat.properties , department-service-qa.properties  (or .yml)**

* The spring cloud config server will pick the correct property file based on the application name and the activated profile name. If no profile is activated/mentioned explicitly, it will pick the property file with no profile suffix.

**e.g:- department-service.properties (or .yml)**

**Profile can be activated throw vmorguments with running application**

**-Dspring.profiles.active=qa**

**What is the purpose of application.properties?**

It is used to declare the properties that should be shared among all services.

### **@RefreshScope  and   “*/actuator/refresh*”  event**

By default, the configuration values are read on the **client’s startup**, and not again. You can force a bean to *refresh* its configuration – to pull updated values from the **Config Server** – by annotating the **WelcomeController** with the Spring Cloud Config **@RefreshScope**and then by triggering a*/actuator/refresh* event.

Once the */***actuator/refresh** event is triggered for the service, all the beans that are annotated with **@RefreshScope** will be refreshed (that means the values will be retrieved from the Config Server and updates the bean)

**Why spring-cloud-config-server related properties are placed in bootstrap.properties (bootstrap.yml) instead of application.properties (or application.yml)**

In microservice environment, the properties related to the service may be stored in the **Spring Cloud Config Server** and it is identified as a external property source. Therefore those external properties should be fetched and made available to the application **before**the **main application context** is loaded.  If we put those **spring-cloud-config-server**(fetching related) configuration properties in the **application.properties** file, the remote properties will not be loaded before the**main application context** loads. This may lead for some runtime errors saying that the targeted properties does not exist in the main application context. The solution is to load those external properties before the main application context started.

we should place the spring-cloud-config related properties in the **bootstrap.properties** (**bootstrap.yml**). It will be loaded before the  **application.properties** (**application.yml**) is loaded. It will helps to create a “**bootstrap**” context which is a parent context of the main application context and it is responsible for fetching the related properties from the external configuration sources (such as Spring Cloud Config Server)

### **How to share properties (common properties) across all services?**

As i have described above, there are some properties which are common to all or few services. e.g:- database and security related properties etc..

we will look at how those properties are managed with spring cloud config server.

Any property that is added in to the **application.properties** (or **.yml)** will be shared among all services.  Therefore if you want some properties to be shared among all or few services, add them into a file called **application.properties** (or**application.yml**) and commit the file. Then the **Spring Cloud Config Server** will enable those properties available for all services (as shared properties).

**Spring cloud sleuth**

Spring Cloud Sleuth provides the distributed tracing capabilities and we can also export these trace information to [Zipkin](https://zipkin.io/" \t "_blank) to visualize the call traces.

**In this post, we are going to learn:**

* Tracing Distributed Service Calls
* Using Spring Cloud Sleuth for Distributed Tracing
* Distributed Tracing with Zipkin Server

Sleuth makes it possible to trace the requests by adding unique ids to logs.

A trace id (1st) is used for tracking across the microservices; represents the whole journey of a request across all the microservices, while span id (2nd) is used for tracking within the individual microservice

To use Sleuth, add dependency to pom.xml …

Example: the catalog-service endpoint http://localhost:8181/api/products endpoint which internally invokes inventory-service endpoint http://localhost:8282/api/inventory.

In catalog-service logs you can find log statements something like:

2018-03-20 10:54:29.625 INFO [catalog-service,0335da07260d3d6f,0335da07260d3d6f,false] 53617 --- [io-8181-exec-10] ...

And, check logs in inventory-service, you can find log statements something like:

2018-03-20 10:54:29.662 INFO [inventory-service,0335da07260d3d6f,1af68249ac3a6902,false] 53685 --- [oryController-6] ...

Observe that TraceID **0335da07260d3d6f** is same in both catalog-service and inventory-service for the same REST API call. **This way we can easily track correlate the logs across services.**

|  |
| --- |
| spring.zipkin.base-url=http://localhost:9411/  spring.sleuth.sampler.probability=1 |

By default, spring.sleuth.sampler.probability=0.1 which means only 10% of tracing information will be exported to Zipkin. Make it to your desired percentage.

**Add dependencies both services:**

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-sleuth</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-sleuth-zipkin</artifactId>  </dependency> |

## Zipkin Server

Zipkin is an application that monitors and manages the Spring Cloud Sleuth logs of your Spring Boot application. To build a Zipkin server, we need to add the Zipkin UI and Zipkin Server dependencies in our build configuration file.

[application-name,traceid,spanid,zipkin-export]

* Application-name = Name of the application
* Traceid = each request and response traceid is same when calling same service or one service to another service.

Trace id is used for tracking across the micro service.

* Spanid = Span Id is printed along with Trace Id. Span Id is different every request and response calling one service to another service.

Span is used for tracking within the microservice.

* Zipkin-export = By default it is false. If it is true, logs will be exported to the Zipkin server.

**Steps to create Zipkin server**

**Step 1:** create spring boot application or docker image

Step 2: add @EnableZipkinServer in spring boole application class

|  |
| --- |
| **import**zipkin.server.~~EnableZipkinServer~~;  @SpringBootApplication  @~~EnableZipkinServer~~  **publicclass**ZipkinServerApplication {  //http://localhost:9411/zipkin/  **publicstaticvoid**main(String[] args) {  SpringApplication.*run*(ZipkinServerApplication.**class**, args);  }  }  Docker compose file:  We can create a Zipkin server backed by in-memory datastore using the following **docker-compose-mem.yml file**.  version: '2'  services:  # The zipkin process services the UI, and also exposes a POST endpoint that  # instrumentation can send trace data to. Scribe is enabled by default.  zipkin:  image: openzipkin/zipkin  container\_name: zipkin  # Environment settings are defined here https://github.com/openzipkin/zipkin/tree/1.19.0/zipkin-server#environment-variables  environment:  - STORAGE\_TYPE=mem  # Uncomment to disable scribe  # - SCRIBE\_ENABLED=false  # Uncomment to enable self-tracing  # - SELF\_TRACING\_ENABLED=true  # Uncomment to enable debug logging  # - JAVA\_OPTS=-Dlogging.level.zipkin=DEBUG  ports:  # Port used for the Zipkin UI and HTTP Api  - 9411:9411 |

**Spring Boot Monitor**

**Actuator:**

Spring Boot Actuator module helps you monitor and manage your Spring Boot application by providing production-ready features like health check-up, auditing, metrics gathering, HTTP tracing etc. All of these features can be accessed over JMX or HTTP endpoints.

Actuator also integrates with external application monitoring systems like [Prometheus](https://prometheus.io/), [Graphite](https://graphiteapp.org/), [DataDog](https://www.datadoghq.com/), [Influx](https://www.influxdata.com/), [Wavefront](https://www.wavefront.com/), [New Relic](https://newrelic.com/) and many more. These systems provide you with excellent dashboards, graphs, analytics, and alarms to help you monitor and manage your application from one unified interface.

Actuator uses [Micrometer](http://micrometer.io/), an application metrics facade to integrate with these external application monitoring systems. This makes it super easy to plug-in any application monitoring system with very little configuration.

**Micrometer:**

Micrometer provides a simple faced to integrate actuator metrics with external monitoring systems.

It supports several monitoring systems like Netflix Atlas, AWS Cloudwatch, Datadog, InfluxData, SignalFx, Graphite, Wavefront, Prometheus etc.

To integrate actuator with Prometheus, you need to add the micrometer-registry-prometheusdependency .

<!-- Micrometer Prometheus registry -->

<dependency>

<groupId>io.micrometer</groupId>

<artifactId>micrometer-registry-prometheus</artifactId>

</dependency>

Once you add the above dependency, Spring Boot will automatically configure a [PrometheusMeterRegistry](https://github.com/micrometer-metrics/micrometer/blob/master/implementations/micrometer-registry-prometheus/src/main/java/io/micrometer/prometheus/PrometheusMeterRegistry.java) and a [CollectorRegistry](https://github.com/prometheus/client_java/blob/master/simpleclient/src/main/java/io/prometheus/client/CollectorRegistry.java) to collect and export metrics data in a format that can be scraped by a Prometheus server.

All the application metrics data are made available at an actuator endpoint called /prometheus. The Prometheus server can scrape this endpoint to get metrics data periodically.

**Prometheus (ప్రోమేతియస్):**

Prometheus is an open-source monitoring system that was originally built by SoundCloud. It consists of the following core components -

* A data scraper that pulls metrics data over HTTP periodically at a configured interval.
* A time-series database to store all the metrics data.
* A simple user interface where you can visualize, query, and monitor all the metrics.

## Grafana:

Grafana allows you to bring data from various data sources like Elasticsearch, Prometheus, Graphite, InfluxDBetc, and visualize them with beautiful graphs.

It also lets you set alert rules based on your metrics data. When an alert changes state, it can notify you over email, slack, or various other channels.

*Note that,* ***Prometheus dashboard also has simple graphs. But Grafana’s graphs are way better. That’s why, in this post, we’ll integrate Grafana with Prometheus to import and visualize our metrics data.***

**Downloading Prometheus**

You can download the Prometheus docker image using docker pull command like so

docker pull prom/Prometheus

Running Prometheus using Docker:

sudo docker run -d --name=prometheus -p 9090:9090 -v /home/nagendra/micro-meter/prometheus.yml prom/Prometheus

<http://192.168.95.150:9090>

**Prometheus Configuration (prometheus.yml):**

|  |
| --- |
| # my global config  global:  scrape\_interval: 15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.  evaluation\_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.  # scrape\_timeout is set to the global default (10s).  # Load rules once and periodically evaluate them according to the global 'evaluation\_interval'.  rule\_files:  # - "first\_rules.yml"  # - "second\_rules.yml"  # A scrape configuration containing exactly one endpoint to scrape:  # Here it's Prometheus itself.  scrape\_configs:  # The job name is added as a label `job=<job\_name>` to any timeseries scraped from this config.  - job\_name: 'prometheus'  # metrics\_path defaults to '/metrics'  # scheme defaults to 'http'.  static\_configs:  - targets: ['192.168.95.150:9090']  - job\_name: 'spring-actuator'  metrics\_path: '/actuator/prometheus'  scrape\_interval: 5s  static\_configs:  - targets: ['192.168.95.150:9595'] |

## Downloading and running Grafana using Docker

Type the following command to download and run Grafana using Docker

docker run -d --name=grafana -p 3000:3000 grafana/grafana

**Commands for execution:**

sudo docker build --tag=micro-meter:latest --rm=true .

sudo docker run --name=micro-meter --publish=9091:9091 micro-meter:latest

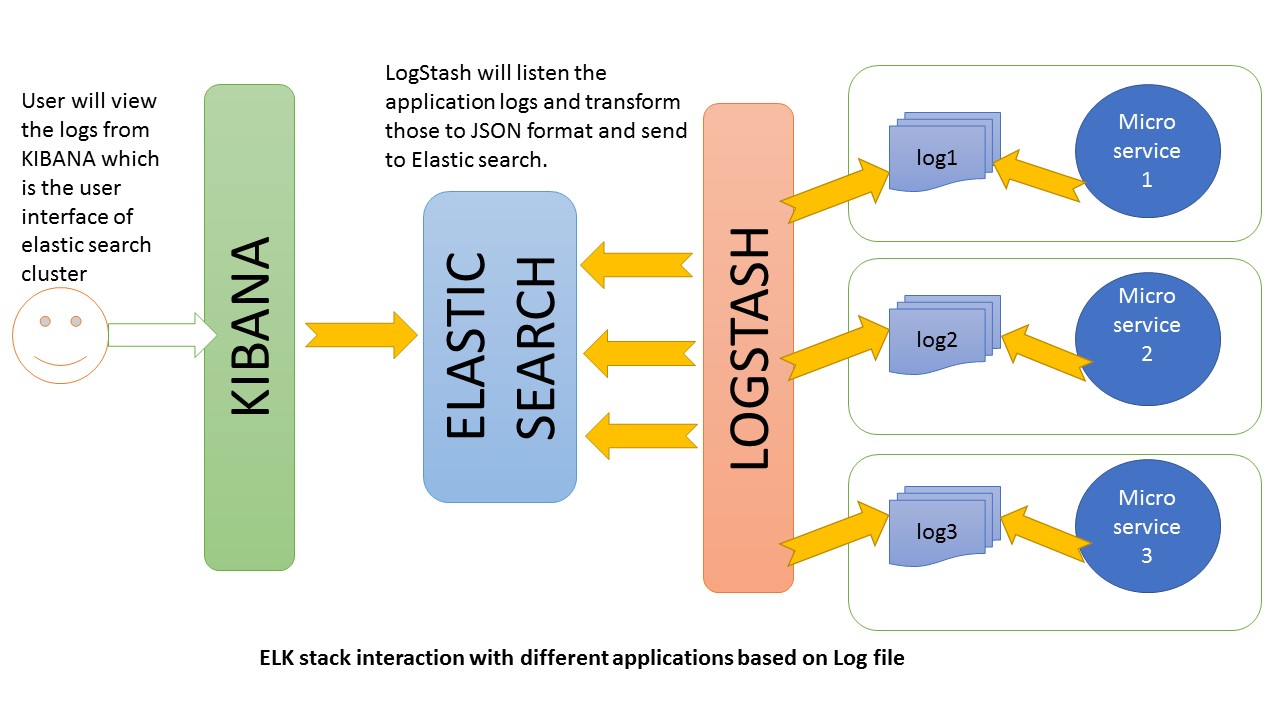
**Metrics Reports:**

JVM, report utilization of:

* Various memory and buffer pools
* Statistics related to garbage collection
* Thread utilization
* Number of classes loaded/unloaded
* CPU usage

**ELK**

**ELK**on the other hand is a combination of three open source tools(Elastic Search-Logstash-Kibana). Similar to Splunk, ELK can be installed on-premise as well as setup on the cloud. Their cloud platform is called Elastic Cloud. If you are an AWS user, then you have another option: AWS Elastic Search. In October last year, AWS released this as a hosted solution for ELK.



**Logstash:**

Logstash is the data collection pipeline tool. It the first component of ELK Stack which collects data inputs and feeds it to the Elasticsearch. It collects various types of data from different sources, all at once and makes it available immediately for further use.

 **Elasticsearch:**

Elasticsearch is a NoSQL database which is based on Lucene search engine and is built with RESTful APIs. It is a highly flexible and distributed search and analytics engine. Also, it provides simple deployment, maximum reliability, and easy management through horizontal scalability. It provides advanced queries to perform detailed analysis and stores all the data centrally for quick search of the documents.

**Kibana**

Kibana is a data visualization tool. It is used for visualizing the Elasticsearch documents and helps the developers to have an immediate insight into it. Kibana dashboard provides various interactive diagrams, geospatial data, timelines, and graphs to visualize the complex queries done using Elasticsearch. Using Kibana you can create and save custom graphs according to your specific needs.

**The ELK stack from**[**Elastic**](https://www.elastic.co/)**consist of:**

* [**Logstash**](https://www.elastic.co/products/logstash)Logstash can collect log events from multiple types of sources using [*input*](https://www.elastic.co/guide/en/logstash/current/input-plugins.html) plug-ins, transform it to a format you prefer using [*filter*](https://www.elastic.co/guide/en/logstash/current/filter-plugins.html) and [*codec*](https://www.elastic.co/guide/en/logstash/current/codec-plugins.html) plug-ins and send it to a number of destinations using [*output*](https://www.elastic.co/guide/en/logstash/current/output-plugins.html) plug-ins.
* [**Elasticsearch**](https://www.elastic.co/products/elasticsearch)  
  Elasticsearch is a distributed and scalable full-text search database, that allows you to store and search large volumes of log events.

[**Kibana**](https://www.elastic.co/products/kibana)  
Kibana lets you visualize and analyze your log events stored in Elasticsearch