Microservices, PCF and Spring Cloud

By

Praveen Oruganti

Blog: https://praveenoruganti.blogspot.com

Facebook Group: https://www.facebook.com/groups/2340886582907696/

Github repo: https://github.com/praveenoruganti

Introduction to MicroServices

Microservices are built around the capabilities and independently deployable by fully automated deployment machinery like usage of SpringBoot, Jenkins and PCF.

Each Microservice should be able to provide unique business context.

Generally, it is implemented as a REST service on HTTP protocol, with technology-agnostic APIs.
Ideally, it does not share database with any other service.

Microservices are

- ✓ REST
- ✓ and Small Well Chosen deployable Units
- ✓ and Cloud Enabled

Challenges of Microservices

- ✓ Bounded Context
- ✓ Configuration Management
- ✓ Dynamic Scale up and Scale down – we will use Eureka,Ribbon.
- ✓ Visibility – we will use Zipkin(using sleuth for generated id for each request) and Zuul
- ✓ Pack of Cards – Not well designed – Fault tolerance i.e. using Hystrix.

Spring Cloud solves the above challenges of microservices by using **Netflix Eureka** for service registry and discovery which is known as Naming Server ,**Spring Cloud config** which provides central GIT location for all the configuration files, **Spring cloud sleuth** for distributed tracing, **Netflix Hystrix** for fault tolerance which acts as circuit breaker, **Netflix Ribbon** for Client side load balancing and **Netflix Zuul** is a gateway service that provides dynamic routing, monitoring, resiliency, security.

What is Bounded Context?

A bounded context is like a specific responsibility that is developed within a boundary. In a domain there can be multiple bounded contexts that are internally implemented. Eg. A hospital system can have bounded contexts like- Emergency Ward handling, Regular vaccination, Out patient treatment etc. Within each bounded context, each sub-system can be independently designed and implemented.

Advantages of Microservices

Advantage	Description
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

Why Microservices?

To build reliable business system

Is it a good idea for Microservices to share a common database?

Sharing a common database between multiple Microservices increases coupling between them. One service can start accessing data tables of another service. This can defeat the purpose of bounded context. So it is not a good idea to share a common database between Microservices.

What are the disadvantages of using Shared libraries approach to decompose a monolith application?

- ✓ You can create shared libraries to increase reuse and sharing of features among teams. But there are some downsides to it.
- ✓ Since shared libraries are implemented in same language, it constrains you from using multiple types of technologies.
- ✓ It does not help you with scaling the parts of system that need better performance.
- ✓ Deployment of shared libraries is same as deployment of Monolith application, so it comes with same deployment issues.
- ✓ Shared libraries introduce shared code that can increase coupling in software.

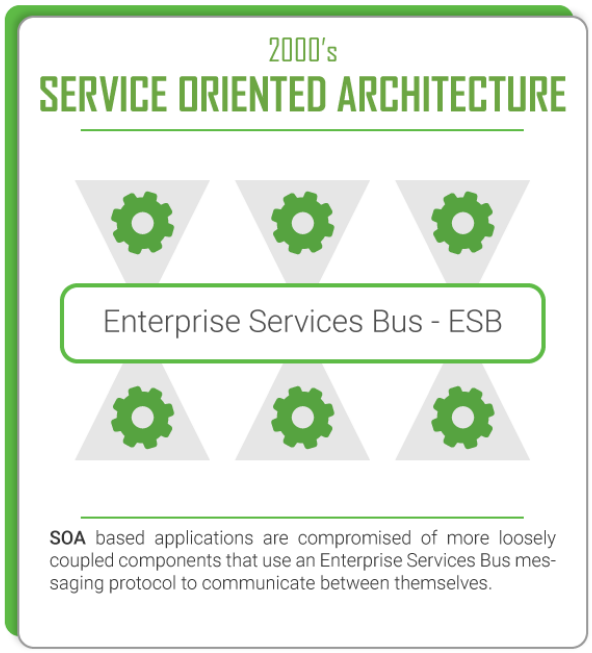
What is the preferred type of communication between Microservices? Synchronous or Asynchronous?

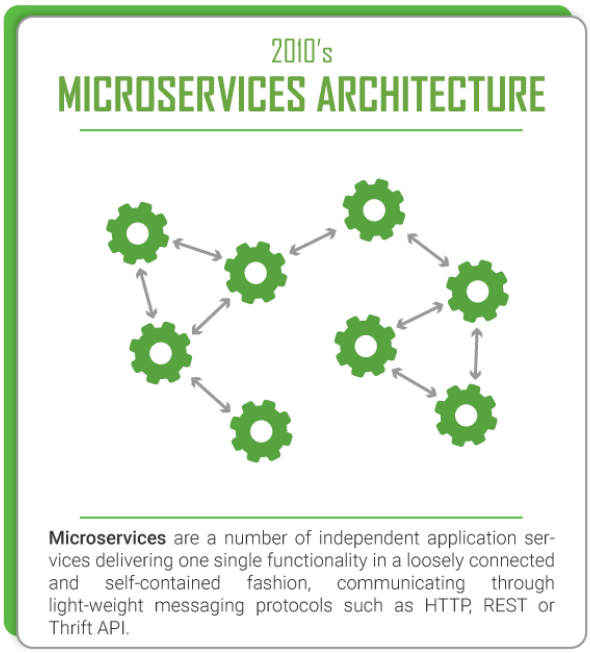
- ✓ Synchronous communication is a blocking call in which client blocks itself from doing anything else, till the response comes back.In Asynchronous

communication, client can move ahead with its work after making an asynchronous call. Therefore client is not blocked.

- ✓ In synchronous communication, a Microservice can provide instant response about success or failure. In real-time systems, synchronous service is very useful. In Asynchronous communication, a service has to react based on the response received in future.
- ✓ Synchronous systems are also known as request/response based. Asynchronous systems are event-based.
- ✓ Synchronous Microservices are not loosely coupled.
- ✓ Depending on the need and critical nature of business domain, Microservices can choose synchronous or asynchronous form of communication.

SOA vs Microservices





Service Oriented Architecture (SOA) is an approach to develop software by creating multiple services. It creates small parts of services and promotes reusability of software. But SOA development can be slow due to use of things like communication protocols SOAP, middleware and lack of principles.

On the other hand, **Microservices** are agnostic to most of these things. You can use any technology stack, any hardware/middleware, any protocol etc. as long as you follow the principles of Microservices. Microservices architecture also provides more flexibility, stability and speed of development over SOA architecture.

What is the difference between Orchestration and Choreography in Microservices architecture?

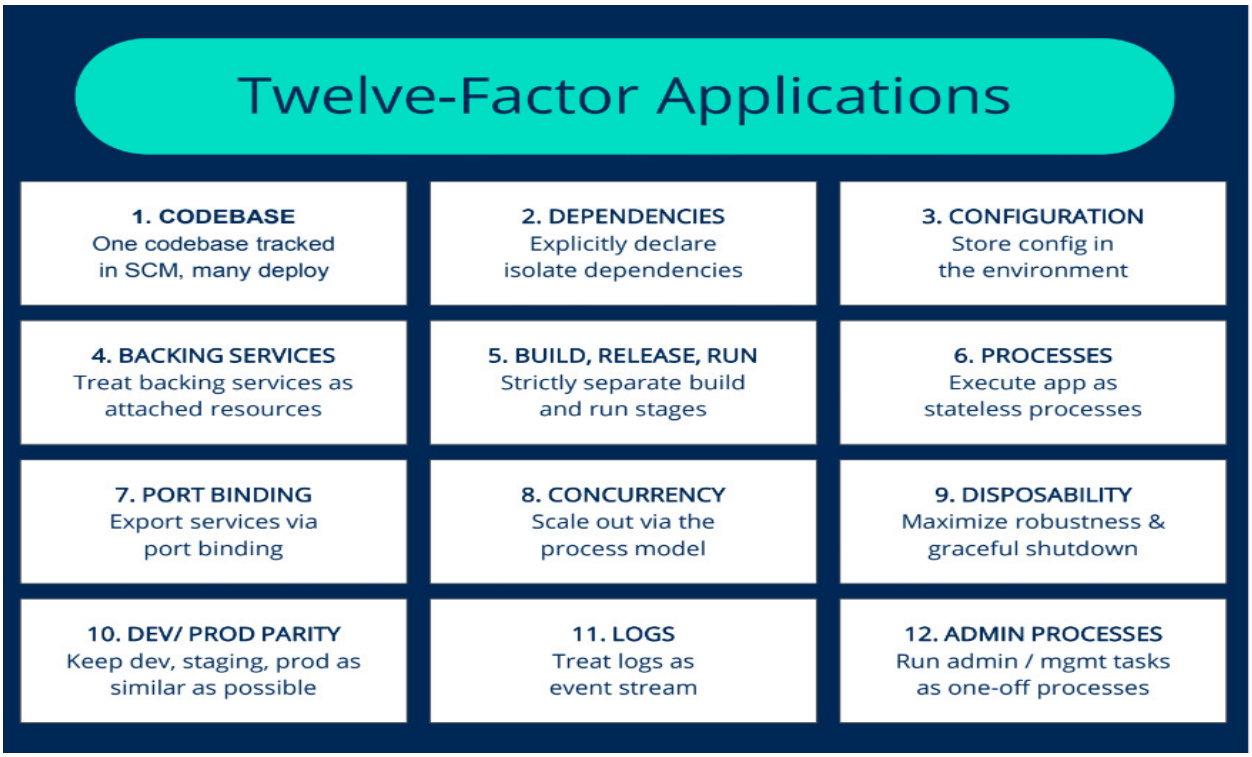
- ✓ In Orchestration, we rely on a central system to control and call various Microservices to complete a task. In Choreography, each Microservice works like a State Machine and reacts based on the input from other parts.
- ✓ Orchestration is a tightly coupled approach for integrating Microservices. But Choreography introduces loose coupling. Also, Choreography based systems are more flexible and easy to change than Orchestration based systems.
- ✓ Orchestration is often done by synchronous calls. But choreography is done by asynchronous calls. The synchronous calls are much simpler compared to asynchronous communication.

Can we create Microservices as State Machines?

Yes, Microservices are independent entities that serve a specific context. For that context, the Microservice can work as a State Machine. In a State Machine, there are lifecycle events that cause change in the state of the system.

For Example, In a Library service, there is a book that changes state based on different events like- issue a book, return a book, lose a book, late return of a book, add a new book to catalog etc. These events and book can form a state machine for Library Microservice.

12 Factor App



12 factor app (twelve-factor app) is a methodology for building distributed applications that run in the cloud and are delivered as a service. The approach was developed by Adam Wiggins, the co-founder of Horoku, a platform-as-a-service which is now part of Salesforce.com: The Customer Success Platform To Grow Your Business. Wiggin's goal was to synthesize best practices for deploying an app on Horoku and provide developers who are new to the cloud with a framework for discussing the challenges of native cloud applications.

Although some factors may seem self-evident to developers today, interest in developing apps that adhere to common best practices continues to grow with the rise of micro-services and applications that are composed of loosely-coupled web services.

<u>12 factors developers should think about when building native cloud apps:</u>

1. Code base
Use one codebase, even when building cross-platform apps. Address the needs of specific devices with version controls.

2. Dependencies
Explicitly declare and isolate all dependencies.

3. Configuration
Don't store config as constants in code. Design the app to read its config from the environment.

4. Backing Services
Treat back-end services as attached resources to be accessed with a URL or other locator stored in config.

5. Build, Release, Run
Strictly separate build and run stages.

6. Processes
Execute the app as one or more stateless processes. Data that must be persistent should be stored in a stateful backing service.

7. Port binding
Use port binding to export services.

8. Concurrency
Scale out apps horizontally, not vertically.

9. Disposability
Use fast startups and graceful shutdowns to maximize robustness.

10. Parity
Facilitate continuous deployment by making development, staging, and production environments as similar as possible.

11. Logs
Treat logs as event streams. Logs should not be concerned with routing or storing the app's output.

12. Admin processes
Run admin tasks as one-off processes from a machine in the production environment that's running the latest production code.

The 12-factor basics

When a developer uses the twelve-factor app DevOps methodology, applications will have certain characteristics in common that address a variety of scenarios as an app scales. For example, the methodology recommends that apps use declarative formats for setup automation to assist new developers that enter the project at a later time.

Apps should also be written to have maximum portability between execution environments and scale easily without significant reworking. Twelve-factor apps can be

written in any programming language and in combination with any back-end service, such as a database.

The goal of the twelve-factor framework is to help developers build apps that use an architecture that ensures speed, reliability, agility, portability and ultimately results in a robust and reliable application.

Any developer building cloud-based applications, most of which run as a service, should familiar with the 12 factors.

Benefits of 12-factor app methodology

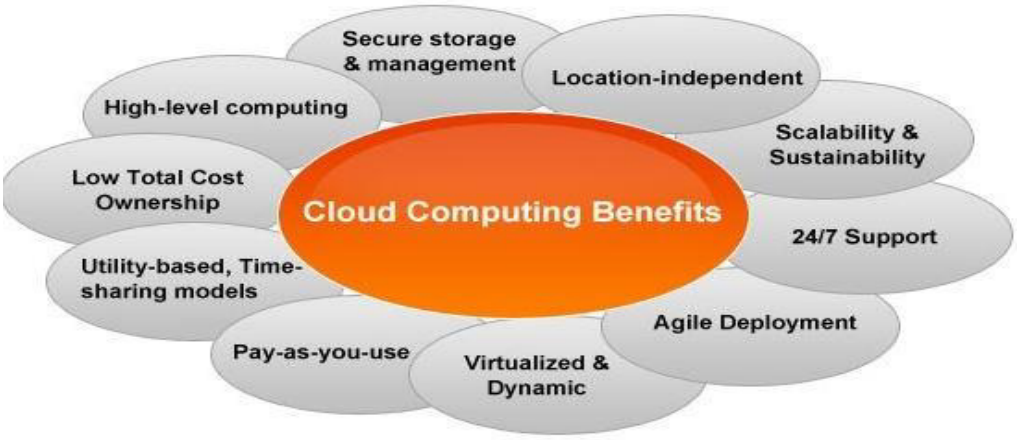
- ✓ This method shows a path to follow for the development.
- ✓ It avoids the confusion and saves time.
- ✓ The project can be handled more efficiently by adopting this method.
- ✓ Time and resource management can be done appropriately with this method.
- ✓ Deployment date is handled properly.
- ✓ End minute changes can be done easily.
- ✓ The method suits perfectly with cloud platforms and other operating systems as well.

To implement the above 12 Factor App we need a Cloud.

PCF (Pivotal Cloud Foundary)

Now a days Cloud Computing and Microservice have become very popular concept and almost all the organizations are investing and adapting it very fast. Currently there are only few popular cloud providers in the market and Cloud Foundry is one of them. It is a PaaS service where we can easily deploy and manage our applications and the Cloud Foundry will take care of the rest of the cloud based offerings like scalability, high availability etc.

Cloud Foundry is an open source cloud computing program basically generated in-house by VMware. It is presently allowed via Pivotal Software, which is a collective venture built up of VMware, General Electric, and EMC.



Cloud Foundry is optimized to perform…

- ✓ Fast application improvement and deployment.
- ✓ Extremely scalable and accessible architecture.
- ✓ DevOps-friendly workflows.
- ✓ Lowered chance of human failure.
- ✓ Multi-tenant calculate capabilities.

Not only can Cloud Foundry lighten developer workloads, however, considering Cloud Foundry manages so generous of an application's source administration, but it can also further hugely reduce the above burden on your employment team.

Essential advantages of Cloud Foundry:

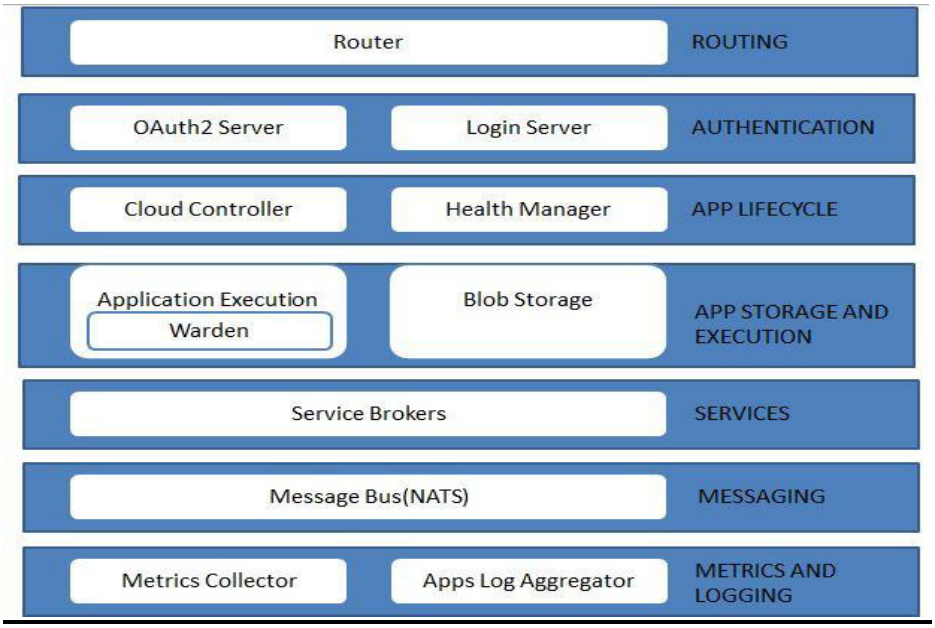
- ✓ Employment portability.
- ✓ Applying auto-scaling.
- ✓ Centralized principles administration.
- ✓ Centralized logging.
- ✓ Powerful routing.
- ✓ Application health management.
- ✓ Alliance with external logging elements like Logstash and Elasticsearch.
- ✓ A role-based path for expanded applications.
- ✓ Provision for horizontal and vertical scaling.
- ✓ Base security. Pivotal Training
- ✓ Assistance for multiple IaaS providers.

It is a service (PaaS) on which developers can build, deploy, run and scale applications.

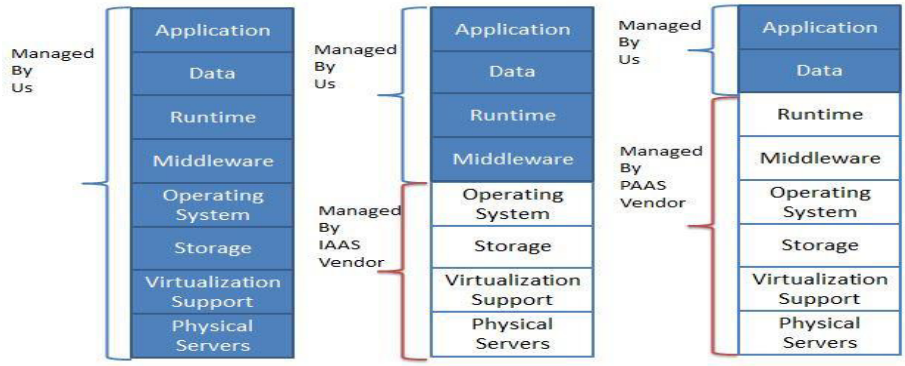
Many Organizations provide the cloud foundry platform separately. For example following are some cloud foundry providers

- ✓ Pivotal Cloud Foundry
- ✓ IBM Bluemix
- ✓ HPE Helion Stackato 4.0
- ✓ Atos Canopy
- ✓ CenturyLink App Fog
- ✓ Huawei FusionStage
- ✓ SAP Cloud Platform
- ✓ Swisscom Application Cloud

CloudFoundary Architecture



Pivotal Cloud Foundry (PCF) is just a multi-cloud platform for the deployment, management, and continuous delivery of applications, containers, and functions. PCF is just a distribution of the open source Cloud Foundry developed and maintained by Pivotal Software, Inc.



Advantages of PCF

- ✓ Fast application development and deployment.
- ✓ Highly scalable and available architecture.
- ✓ DevOps-friendly workflows.
- ✓ Reduced chance of human error.
- ✓ Multi-tenant compute efficiencies.

PCF Development Environment Setup

Downloading and installing Cloud Foundry Command Line Interface (CLI)

The Cloud Foundry (cf) command line interface (CLI) provides a set of commands for managing your apps. We will need to download and install this interface for our windows machine.

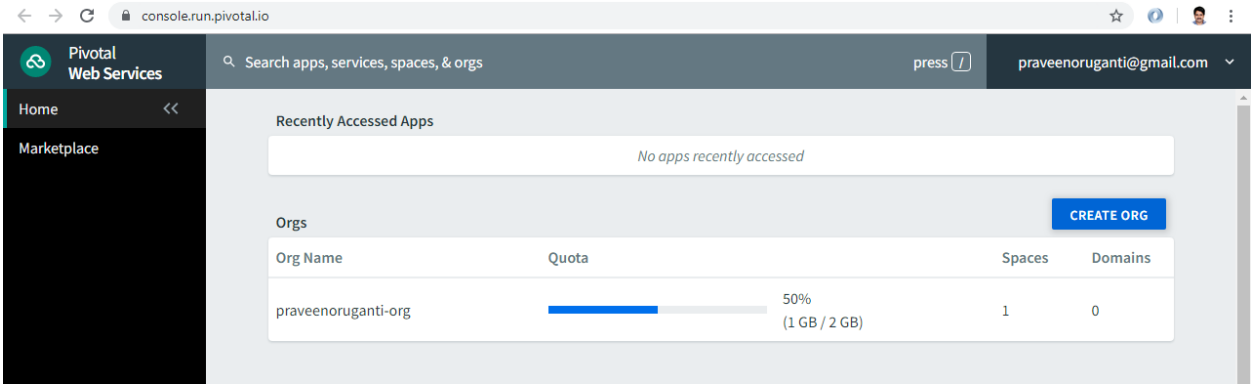
Download the installer from https://cli.run.pivotal.io/stable?release=windows64&source=github and unzip it. After that double click on the CF CLI executable and install it.

To list all of the cf commands and associated help information, use cf help. Use cf command_name -h to view detailed help information for a particular command.

Creating Pivotal Cloud Foundry Account

Go to https://account.run.pivotal.io/sign-up and register for free account by providing your personal details which includes mobile number as well.

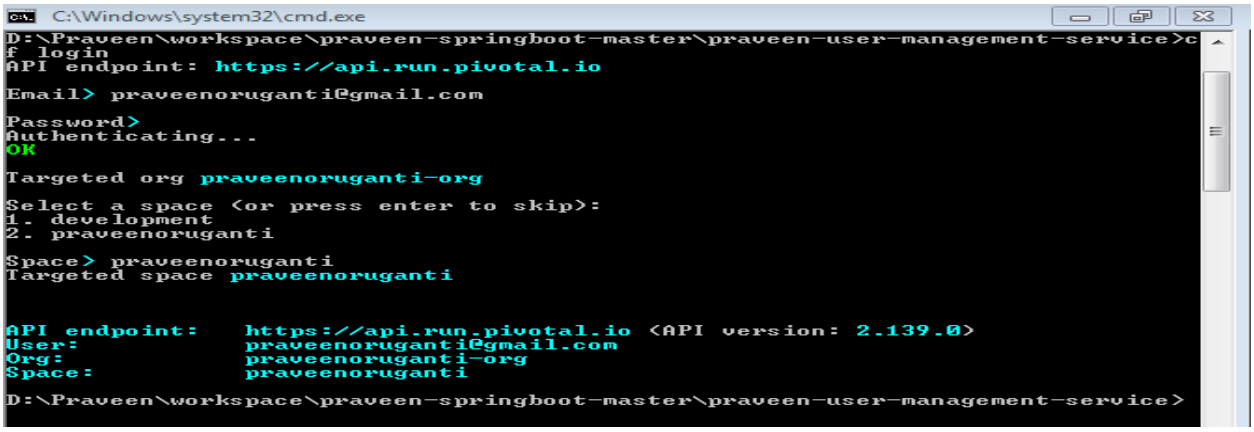
After that click on Pivotal Web Services



In the above step, I have created an Organization named praveenorugant-org. A default development space was assigned to us.

Organization (org) is a development account that encompasses computing resources, apps, and services. It can be owned and used by an individual or by multiple collaborators. Set the org name to be the name of the project you'll be working on or the name of your team. Don't worry - you can change this name at any time.

Login into your pcf account using CLI command <u>cf login</u> by providing your creditionals



<u>Now let's see how we can deploy the springboot application in pcf</u>

First you need to create manifest.yml for your springboot application shown as below.

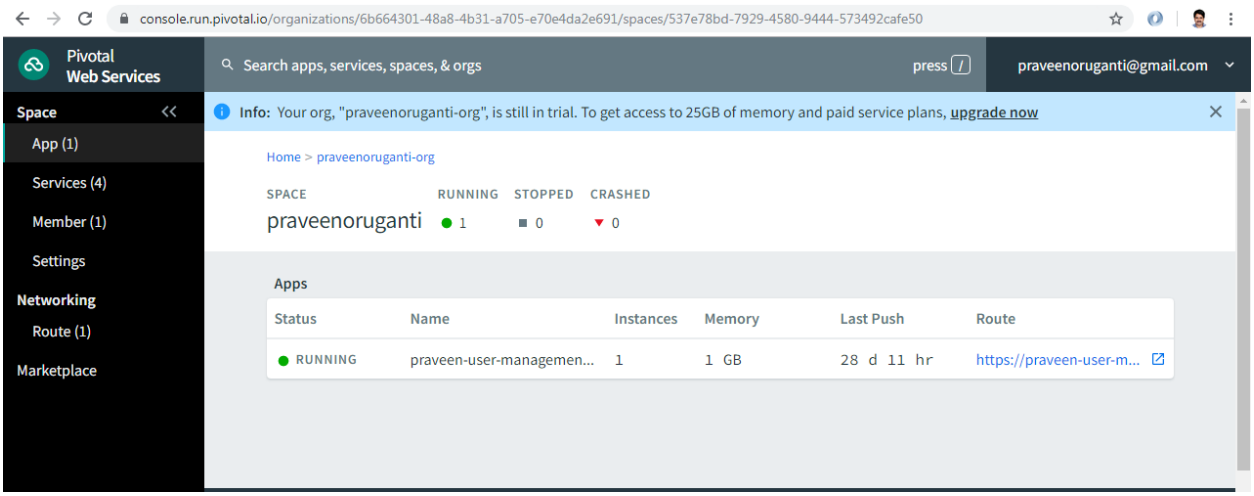
For example,
cd D:\Praveen\workspace\praveen-springboot-master\praveen-user-management-service

Example manifest.yml for spring boot application praveen-user-management-service

manifest.yml

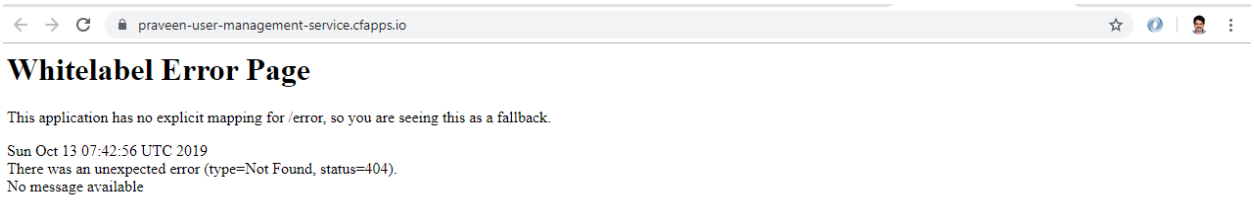
```yaml
---
applications:
 - name: praveen-user-management-service
   buildpack: java_buildpack
   path: target/praveen-user-management-service-1.0.jar
   services:
       - praveen-spring-config-server
       - praveen-service-registry
       - praveen-mysql
       - praveen-rabbitmq
       - praveen-redis
   domain: cfapps.io
   memory: 1G
   instances: 1
```

Now for deploying the application in PCF you need to run the command **cf push**



Open the Route URL mentioned in above screenshot for accessing the application.

https://praveen-user-management-service.cfapps.io/



Now modify the URL as https://praveen-user-management-service.cfapps.io/rest/users


```
[
  - {
        userid: 5,
        username: "PraveenOruganti",
        firstname: "Praveen",
        lastname: "Oruganti",
        email: "praveenoruganti@gmail.com",
        role: "Senior Technical Lead",
        ssn: "149903",
        address: "Hyderabad"
    }
]
```


For deleting app in PCF

cf delete -r app-name

for example, cf delete -r praveen-user-management-service

What cf push does?

- ✓ Upload package
- ✓ Staging -creation of droplet
    - determine the right build pack
    - Run the build packs to create droplet
- ✓ Deployment- deployment of droplet into a cell

Package (Source Code) + Build Pack(s) = Droplet

What is Cloud Foundry BuildPack?

Buildpacks provide framework and runtime support for apps. Buildpacks typically examine your apps to determine what dependencies to download and how to configure the apps to communicate with bound services. When you push an app, Cloud Foundry automatically detects an appropriate buildpack for it. This buildpack is used to compile or prepare your app for launch.

How to scale an microservice in PCF?

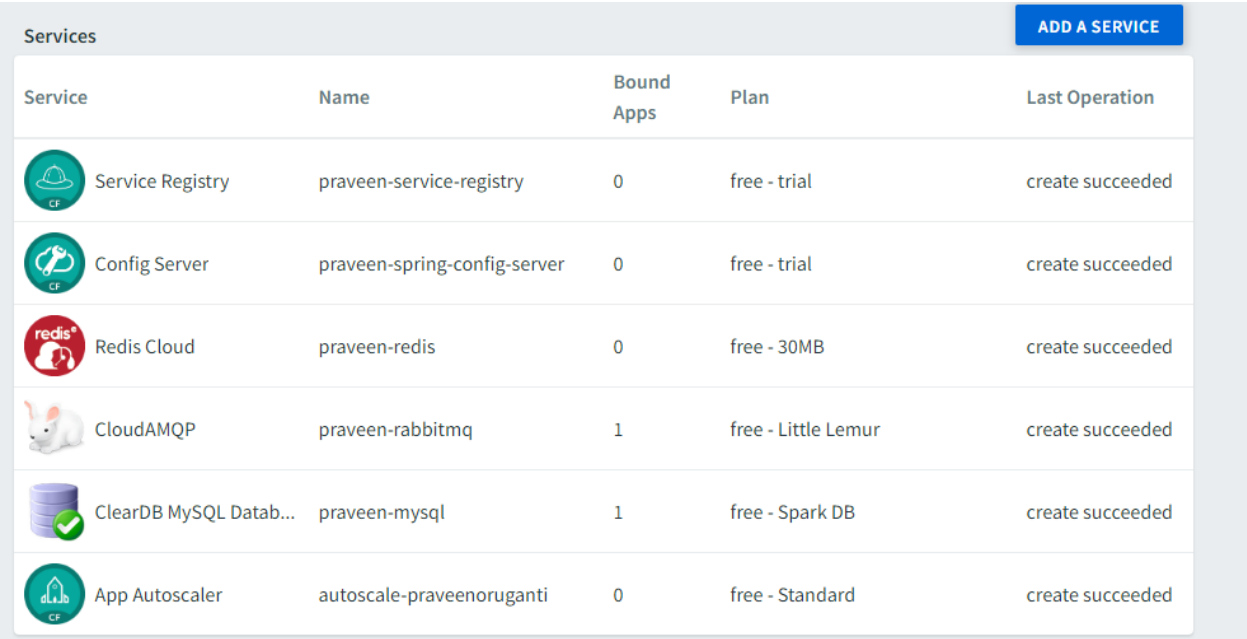
cf scale praveen-user-management-service -i 2

What are various Roles and associated permissions in PCF?

Role	Permissions
Admin	An admin user has permissions on all orgs and spaces
Admin Read-Only	This role has read-only access to all Cloud Controller API resources.
Global Auditor	This role has read-only access to all Cloud Controller API resources except for secrets such as environment variables.
Org Managers	managers or other users who need to administer the org
Org Auditors	view but cannot edit user information and org quota usage information
Org Billing Managers	create and manage billing account and payment information
Org Users	Can view the list of other org users and their roles. When an Org Manager gives a person an Org or Space role, that person automatically receives Org User status in that Org
Space Managers	Managers or other users who administer a space within an org
Space Developers	Application developers or other users who manage applications and services in a space
Space Auditors	View but cannot edit the space

PCF Inbuilt Services

There is an advantage of PCF i.e... provision of inbuilt services. For example we have service-registry, config-server, redis, rabbitmq, circuitbreaker, mysql and many inbuilt services are available….



Service	Name	Bound Apps	Plan	Last Operation
Service Registry	praveen-service-registry	0	free - trial	create succeeded
Config Server	praveen-spring-config-server	0	free - trial	create succeeded
Redis Cloud	praveen-redis	0	free - 30MB	create succeeded
CloudAMQP	praveen-rabbitmq	1	free - Little Lemur	create succeeded
ClearDB MySQL Datab...	praveen-mysql	1	free - Spark DB	create succeeded
App Autoscaler	autoscale-praveenoruganti	0	free - Standard	create succeeded

Spring Cloud

Why Spring cloud is required?

Spring Cloud solves the challenges of microservices by using **Netflix Eureka** for service registry and discovery which is known as Naming Server ,**Spring Cloud config** which provides central GIT location for all the configuration files, **Spring cloud sleuth** for distributed tracing, **Netflix Hystrix** for fault tolerance which acts as circuit breaker, **Netflix Ribbon** for Client side load balancing and **Netflix Zuul** is a gateway service that provides dynamic routing, monitoring, resiliency, security.

Let's see component by component by developing a restful webservice

1. **Microservice Registry and Discovery**

Local Environment Using Netflix Eureka

Eureka Server 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.

Normally in Micro Service Architecture Design we are developing separate Services and exposing each API as service Endpoint and whenever we required to access other services in simple we will be using **RestTemplate**.

It is highly impossible to remember the microservice restful service endpoint URL's with hostname and port so Netflix team came up with solution with Eureka server where all services endpoints will be registered.

Registering with Eureka

When a client registers with Eureka, it provides meta-data about itself such as host and port, health indicator URL, homepage etc. Eureka receives heartbeat messages from each instance belonging to a service. If the heartbeat fails over a configurable timetable, the instance is normally removed from the registry.

In local, we need to develop eureka server SpringBoot application

We need to add below eureka server dependency in pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

We need to include @EnableEurekaServer on top of main class

```
package com.praveen.eureka;

import org.springframework.boot.SpringApplication;

@SpringBootApplication
@EnableEurekaServer
public class SpringCloudEurekaServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringCloudEurekaServerApplication.class, args);
    }

}
```

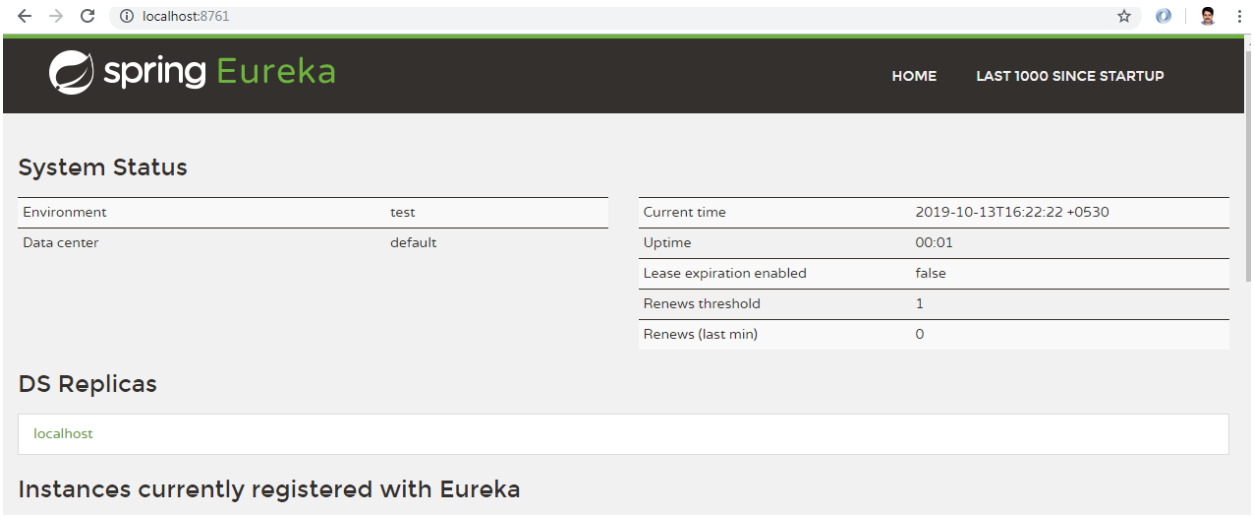
We need to configure below eureka server properties in application.yml

application.yml

```
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
server:
  port: 8761
spring:
  application:
    name: praveen-spring-cloud-eureka-server
```

You can refer my repository for the eureka server code (https://github.com/praveenoruganti/praveenoruganti-springcloud-microservices-master/tree/master/praveen-sping-cloud-eureka-server)

Let's Open URL for eureka server application



Application	AMIs	Availability Zones	Status
No instances available			

General Info

Name	Value
total-avail-memory	159mb
environment	test
num-of-cpus	4
current-memory-usage	48mb (30%)
server-uptime	00:01
registered-replicas	http://localhost:8761/eureka/
unavailable-replicas	http://localhost:8761/eureka/,
available-replicas	

If you see above there are no application registered in eureka server.

Let's consider an example, we have 2 microservices i.e... Flipkart Order Management Service and Flipkart Billing service.

Here the flow will be Flipkart Order Management Service will call Flipkart Billing Service for successful creation of order based on orderid. Here we will call the service using RestTemplate.

praveen-flipkart-ordermanagement-service

We need to add the below dependency in pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

We need to add @EnableEurekaClient on top of main class

```
package com.praveen.billing;

import static springfox.documentation.builders.PathSelectors.regex;

@SpringBootApplication
@EnableSwagger2
@EnableEurekaClient
public class FlipkartBillingServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(FlipkartBillingServiceApplication.class, args);
    }
```

We need to configure eureka properties in application.yml file

application.yml

```
eureka:
  client:
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  instance:
    hostname: localhost
server:
  port: 8091
spring:
  application:
    name: praveen-flipkart-billing-service
```

You can also fetch the code from my repository (https://github.com/praveenoruganti/praveenoruganti-springcloud-microservices-master/tree/master/praveen-flipkart-billing-service)

praveen-flipkart-ordermanagement-service

We need to add the below dependency in pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

We need to add @EnableEurekaClient on top of main class

```
@SpringBootApplication
@EnableEurekaClient
@EnableSwagger2
@EnableHystrixDashboard
@EnableCircuitBreaker
public class FlipkartOrderManagementServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(FlipkartOrderManagementServiceApplication.class, args);
    }

    @Bean
    @LoadBalanced
       public RestTemplate getRestTemplate() {
//      HttpComponentsClientHttpRequestFactory factory =new  HttpComponentsClientHttpRequestFactory();
//      factory.setConnectTimeout(3000);
//        return new RestTemplate(factory);
        return new RestTemplate();
       }
```

We need to configure eureka properties in application.yml file

```
application.yml
management:
 endpoints:
    web:
      exposure:
        include: hystrix.stream

eureka:
  client:
    registerWithEureka: true
    fetchRegistry: true
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
  instance:
    hostname: localhost
server:
  port: 8090
spring:
  application:
    name: praveen-flipkart-ordermanagement-service

praveen-flipkart-ordermanagement-service:
                    billingURL: http://praveen-flipkart-billing-service/rest/billingservice/billingorder
```

You can also fetch the code from my repository (https://github.com/praveenoruganti/praveenoruganti-springcloud-microservices-master/tree/master/praveen-flipkart-ordermanagement-service)

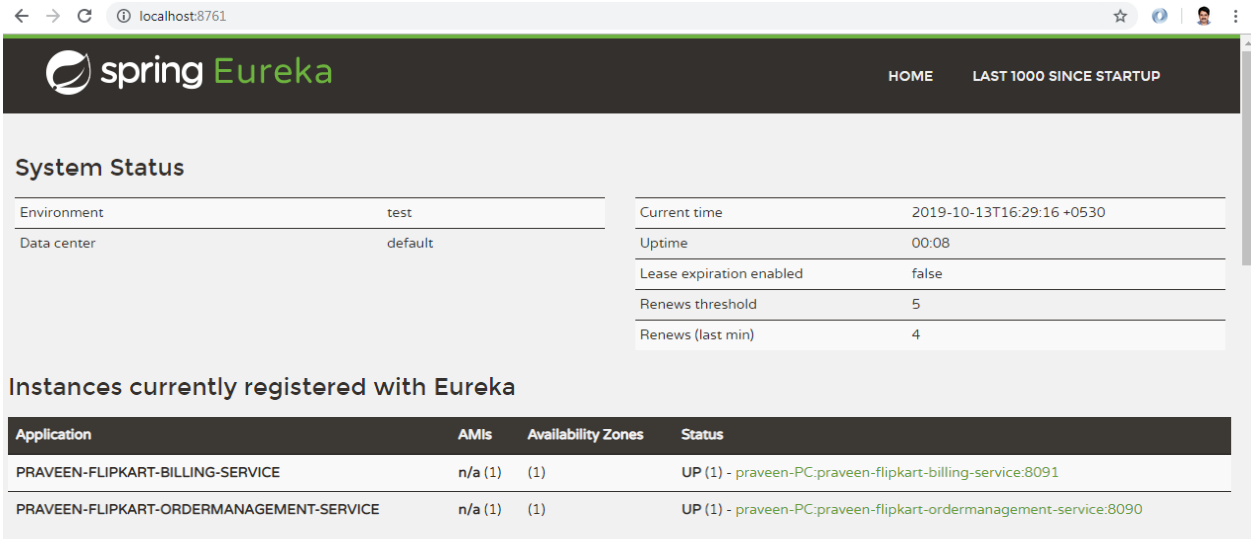
As we have discussed earlier, praveen-flipkart-ordermanagement-service is interacting with praveen-flipkart-billing-service with the help of rest template and let's see the code of it.

```java
public String createOrder(String orderid) throws Exception {
    try {
        HttpHeaders headers = new HttpHeaders();
        headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
        HttpEntity<String> entity = new HttpEntity<String>(orderid, headers);
        // call billing service
        String msg = restTemplate.exchange(billingURL, HttpMethod.POST, entity, String.class).getBody();
        return msg;
    } catch (Exception e) {
        throw new Exception(e);
    }

}
```

Here if you see billingURL is mentioned in application.yml as http://praveen-flipkart-billing-service/rest/billingservice/billingorder

As we have used Eureka Server, there is no need for providing the hostname and port for discovery as praveen-flipkart-ordermanagement-service and praveen-flipkart-billing-service are already registered in Eureka server rather we have used the applicationname in the URL for service interaction via rest template.

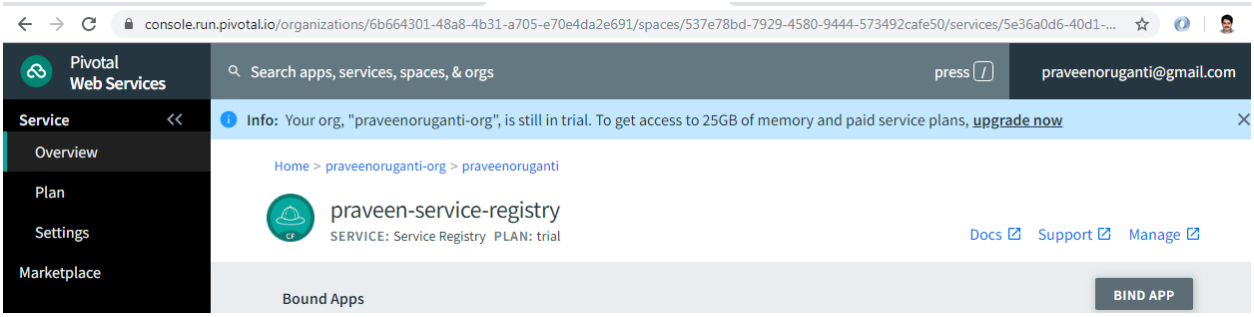
Let's open URL for eureka server application



PCF configuration for service registry and discovery

While using PCF there is no need to develop separate Netflix eureka server springboot application rather we will use the PCF inbuilt **Service Registry** service.

Login into PCF and go to market place and select Service Registry service.



There is no need to use spring-cloud-starter-netflix-eureka-client dependency rather we will be including spring-cloud-services-starter-service-registry dependency in pom.xml

```
<dependency>
    <groupId>io.pivotal.spring.cloud</groupId>
    <artifactId>spring-cloud-services-starter-service-registry</artifactId>
</dependency>
```

All other configurations which we discussed in earlier section related to local environment using eureka server holds good.

2. Client Load Balancing using Netflix Ribbon with Eureka Server

To handle multiple requests made by HTTP client (or consumer) in less time, the provider should run in multiple instances and handle requests in parallel such concept is called as **Load Balancing.**

Spring RestTemplate can be used for client side load balancing.

Client side vs server side load balancing

The multiple instances of the same microservice is run on different computers for high reliability and availability.

Server side load balancing is distributing the incoming requests towards multiple instances of the service.

Client side load balancing is distributing the outgoing request from the client itself.

Netflix provided Ribbon for Client Load Balancing and Ribbon clients are typically created and configured for each of the target services. Ribbon's Client component offers a good set of configuration options such as connection timeouts, retries, retry algorithm (exponential, bounded backoff) etc.

Netflix Ribbon can be easily integrated with a Service Discovery component such as Netflix's Eureka

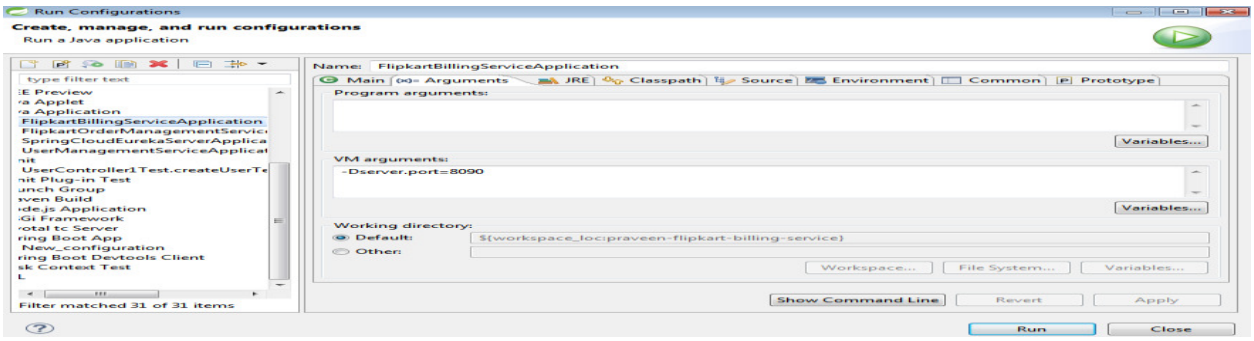
Ribbon chooses one Provider URI,based on InstanceId with the help of LBSRegister which maintains request count.

Let's start coding ribbon using Eureka server.

Ribbon can automatically be configured by registering RestTemplate as a bean and annotating it with @LoadBalanced.

```
@SpringBootApplication
@EnableEurekaClient
@EnableSwagger2
@EnableHystrixDashboard
@EnableCircuitBreaker
public class FlipkartOrderManagementServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(FlipkartOrderManagementServiceApplication.class, args);
    }

    @Bean
    @LoadBalanced
       public RestTemplate getRestTemplate() {
//      HttpComponentsClientHttpRequestFactory factory =new  HttpComponentsClientHttpRequestFactory();
//      factory.setConnectTimeout(3000);
//        return new RestTemplate(factory);
        return new RestTemplate();
       }
```

Run the eureka server and run the praveen-flipkart-billing-service with multiple ports i.e.. 8091 and 8092 by setting the vm arguments -Dserver.port=8091 and -Dserver.port=8092.



<u>Now let's see how we can use Ribbon client without Eureka server.</u>

First configure the below dependency in application.yml of praveen-flipkart-ordermanagement-service

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-ribbon</artifactId>
</dependency>
```

You need to have the below ribbon configuration for list of servers and disable the eureka configuration as well

```
billingservice:
 ribbon:
   eureka: disable
   listOfServers: localhost:8091,localhost:8092
   ServerListRefreshInterval: 2000
```

Then you need to create RibbonConfiguration

```
public class RibbonConfiguration {

    @Autowired
    IClientConfig ribbonclinet;

    @Bean
    public IPing ribbonPing(IClientConfig config) {
        return new PingUrl();
    }

    @Bean
    public IRule ribbonRule(IClientConfig config) {
        return new AvailabilityFilteringRule();
    }

}
```

Include @RibbonClient annotation on top of main class

```
@SpringBootApplication
@EnableEurekaClient
@EnableSwagger2
@EnableHystrixDashboard
@EnableCircuitBreaker
@RibbonClient(name = "billingservice", configuration = RibbonConfiguration.class)
public class FlipkartOrderManagementServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(FlipkartOrderManagementServiceApplication.class, args);
    }

    @Bean
    @LoadBalanced
       public RestTemplate getRestTemplate() {
//      HttpComponentsClientHttpRequestFactory factory =new  HttpComponentsClientHttpRequestFactory();
//      factory.setConnectTimeout(3000);
//        return new RestTemplate(factory);
        return new RestTemplate();
       }
```

Please note there is no need to include ribbon configuration if you are using FeignClient as it has inbuilt ribbon configuration.

Let's start coding by replacing rest template with feign client

First include the below dependency in pom.xml of praveen-flipkart-ordermanagement-service

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

Include @EnableFeignClients on top of main class

```
@SpringBootApplication
@EnableFeignClients
public class FlipkartOrderManagementServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(FlipkartOrderManagementServiceApplication.class, args);
    }
```

Now create FeignBillingServiceProxy class

```
@FeignClient(name="praveen-flipkart-billing-service" ) //Service Id of Billing service
public interface FeignBillingServiceProxy {
    @PostMapping("/billingorder")
    public String createOrder(String orderid);

}
```

Now call this proxy in controller class

```
@RestController
@Api(tags = "Flipkart Order Management Restful Service", value = "OrderManagementController", description = "Controller for Flipkart Order Management Service")
@RequestMapping("/rest/flipkartordermanagement")
public class OrderManagementController {
    @Autowired
    OrderManagementService orderManagementService;
    @Autowired
    FeignBillingServiceProxy billingserviceproxy;


    @PostMapping("/feign/submitorder")
    public ResponseEntity<String> createOrderFeign(@Valid @RequestBody String orderid, UriComponentsBuilder builder) {
        try {
            String msg=billingserviceproxy.createOrder(orderid);
            HttpHeaders headers = new HttpHeaders();
            headers.setLocation(builder.path("/rest/flipkartordermanagement/submitorder/").buildAndExpand(orderid).toUri());
            return  ResponseEntity.status(HttpStatus.CREATED).body(msg);
        }catch(Exception e) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, e.getMessage(),e.getCause());
        }

    }
```

You can see the complete code of this in my github repository (https://github.com/praveenoruganti/praveenoruganti-springcloud-microservices-master/tree/master/praveen-flipkart-ordermanagement-service)

3. Spring Cloud Config Server

Instead of having properties or yml in each springboot application jar, it will be good if we have remote configurations for our applications and then spring boot application access those.

Local Environment Configuration

Create a springboot application with application name praveen-spring-cloud-config-server

Add below dependency in pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

Include @EnableConfigServer on top of main class

```
@EnableConfigServer
@SpringBootApplication
public class PraveenSpringCloudConfigServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(PraveenSpringCloudConfigServerApplication.class, args);
    }

}
```

Include the below GIT path where the application properties are placed in application.yml

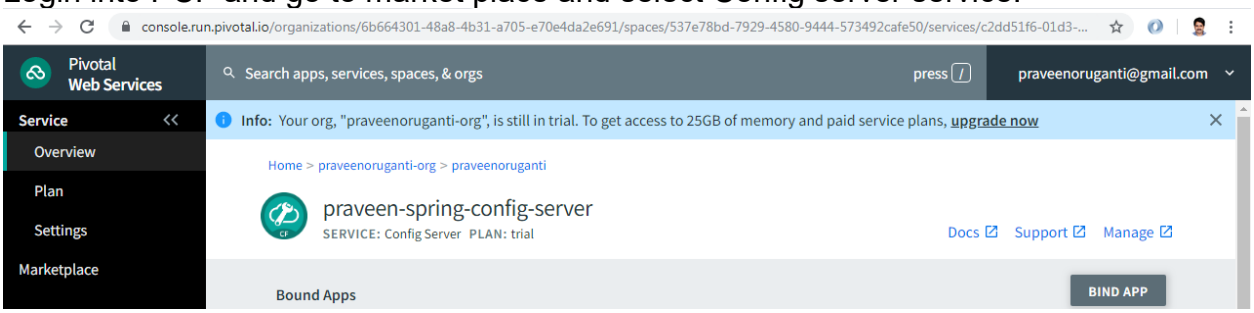
```
application.yml
---
spring:
 application:
  name: praveen-spring-cloud-config-server

 cloud:
   config:
     server:
       git:
         uri: https://github.com/praveenoruganti/praveen-spring-config-server
         searchPaths: config
server:
 port: 8888
```

You can refer the complete code present in my git repository (https://github.com/praveenoruganti/praveenoruganti-springcloud-microservices-master/tree/master/praveen-spring-cloud-config-server)

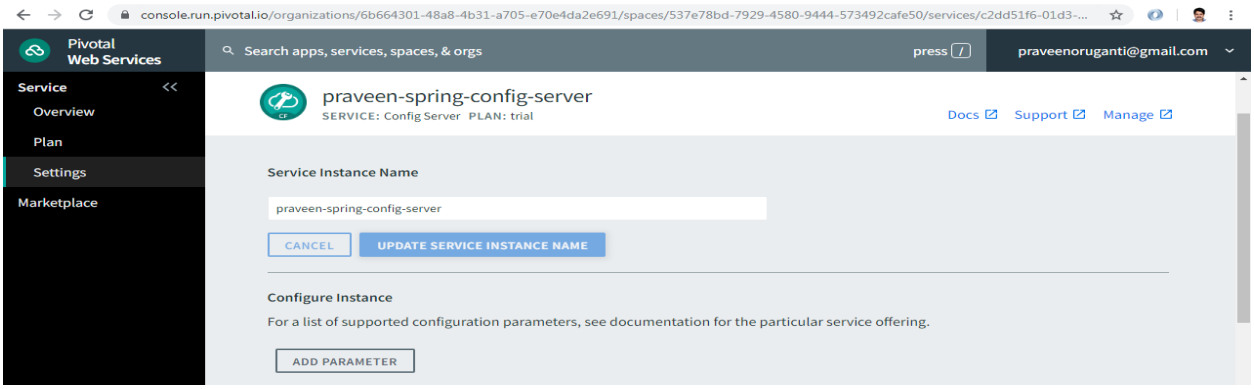
PCF Configuration of Config Server service

While using PCF there is no need to develop separate spring cloude config server springboot application rather we will use the PCF inbuilt **Config Server** service.

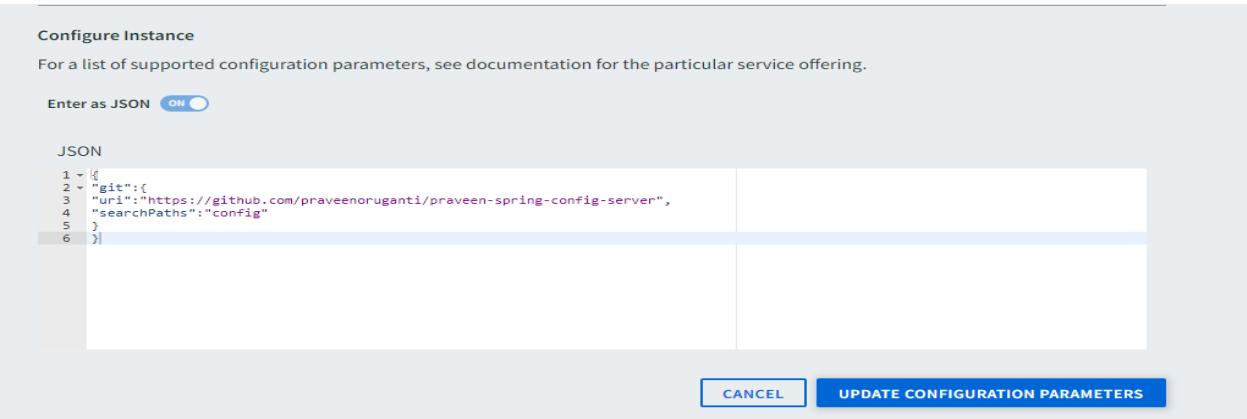
Login into PCF and go to market place and select Config server service.



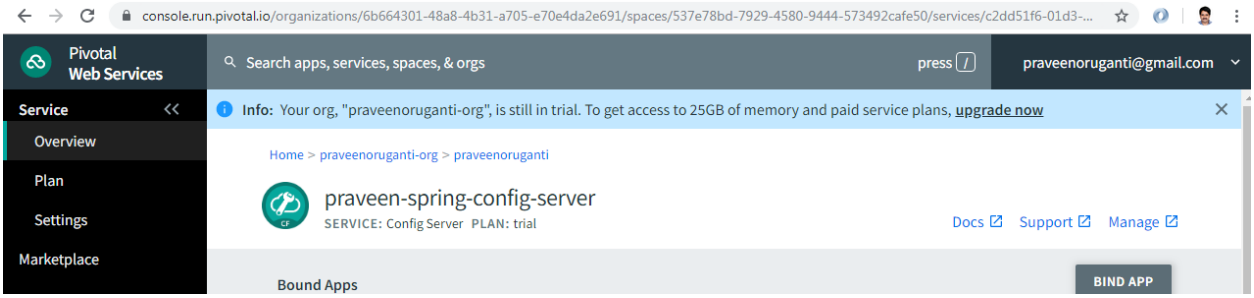
Then click on Settings present in the left tab

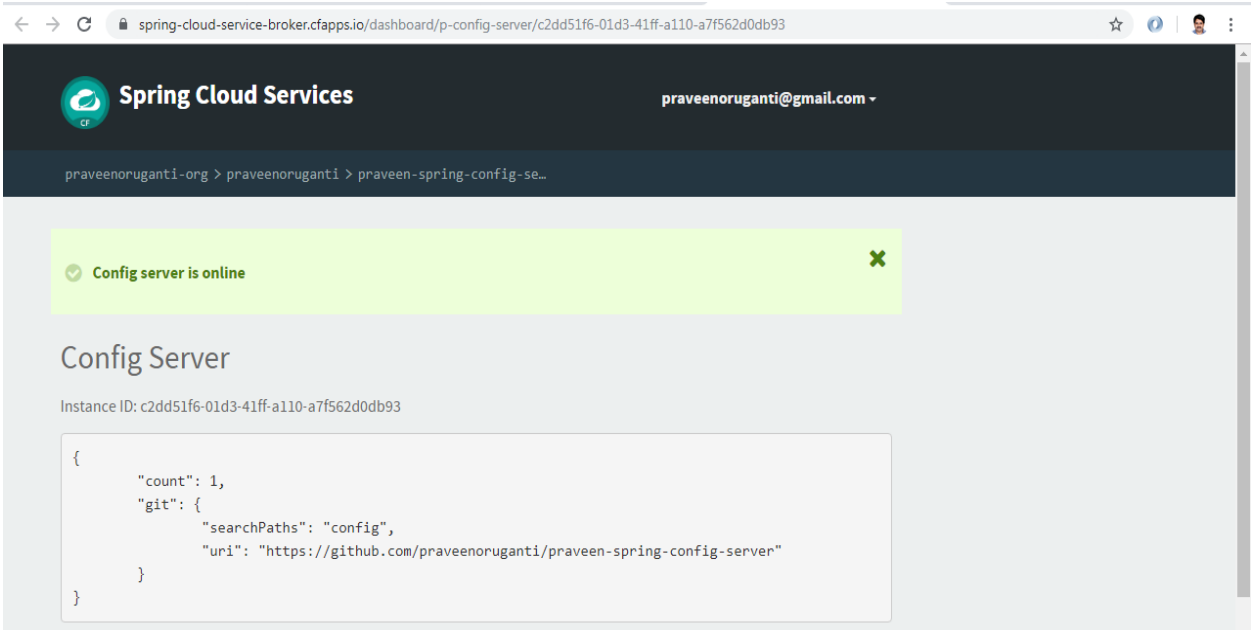


Then click on Add Parameter and select Enter as JSON **ON** and then provide the git configuration properties and click on Update Configuration Parameters button.



Then check whether the git information is properly updated by clicking Manage button.





4. Spring Cloud Sleuth for distributed tracing

If you have, let's say 3 services, A, B and C. We made three different requests.

One request went from A → B, another from A →B →C, and last one went from B →C.

A -> B

A -> B -> C

B -> C

As the number of microservices grow, tracing requests that propagate from one microservice to another and figure out how a requests travels through the application can be quite daunting.

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

Application name(1st) representis name of the application. A trace id (2nd) is used for tracking across the microservices; represents the whole journey of a request across all the microservices, while span id (3rd) is used for tracking within the individual microservice.

Export(4th) exports to log handling server like splunk or zipkin.

To use Sleuth, add below dependency in pom.xml

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

And In service class of praveen-flipkart-billing-service, use logger to log some info.

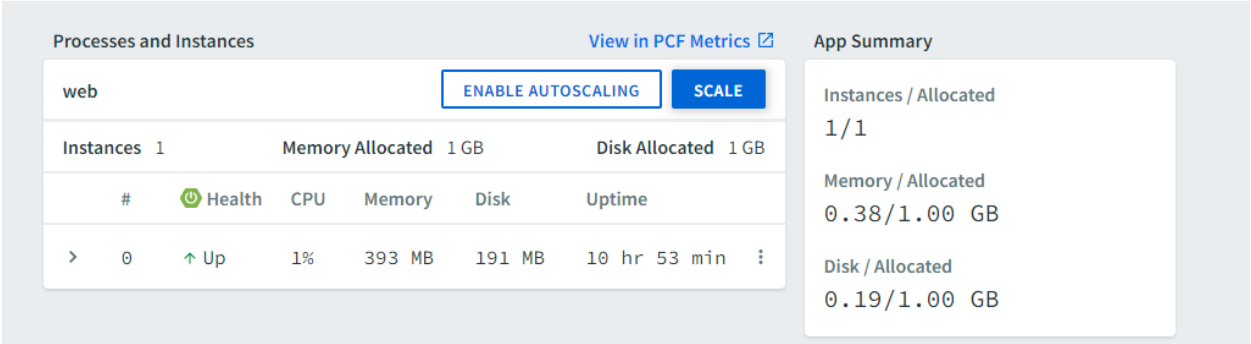
```
package com.praveen.billing.service;

import javax.validation.Valid;
@Service
public class BillingService {
    Logger log = LoggerFactory.getLogger(BillingService.class);

    public String billingOrder(@Valid String orderid) throws Exception {
        int orderidNum=Integer.parseInt(orderid);
        // This logic is just used for testing the service and will not be used in real time.
        if(orderidNum<=0) {
            log.info(orderid);
            throw new Exception("Invalid Order ID " + orderid);
        }else if(orderidNum>0 && orderidNum<500) {
            log.info(orderid);
            throw new Exception("Payment failed for Order ID "+orderid);
        }else {
            log.info(orderid);
            return "Successfully placed the order "+orderid;
        }

    }

}
```

Now check the logs of praveen-flipkart-billing-service by posting some transactions and you can see applicationname, traceid, spanid and log handling server.

Let's take one log transaction from below screenshot and we can see application name is **praveen-flipkart-billing-service**, trace id is **45d0692c1a9bf3be**, span id is **a74949caa31d587a** and log handling server is **false**

```
01:00:14.751  INFO [praveen-flipkart-billing-service,45d0692c1a9bf3be,a74949caa31d587a,false] 7104 --- [nio-8091-exec-1] c.p.billing.service.BillingService      : 30000
01:00:29.306  INFO [praveen-flipkart-billing-service,d0fe1c5e159ad5ea,b610cbb528768fd5,false] 7104 --- [nio-8091-exec-3] c.p.billing.service.BillingService      : 30000
01:00:30.213  INFO [praveen-flipkart-billing-service,b1a9f327254fca8e,ddc7a2556bf33d53,false] 7104 --- [nio-8091-exec-9] c.p.billing.service.BillingService      : 30000
01:00:30.730  INFO [praveen-flipkart-billing-service,d6277c809869b093,56d10897586cb9c5,false] 7104 --- [nio-8091-exec-7] c.p.billing.service.BillingService      : 30000
01:00:31.092  INFO [praveen-flipkart-billing-service,f8a123e61032b389,6f428ee37f0f9842,false] 7104 --- [nio-8091-exec-4] c.p.billing.service.BillingService      : 30000
01:00:31.463  INFO [praveen-flipkart-billing-service,2b903faefaac6dc1,29dc8f3c19286897,false] 7104 --- [nio-8091-exec-8] c.p.billing.service.BillingService      : 30000
01:00:31.694  INFO [praveen-flipkart-billing-service,90d72e11771841ac,bfa2f0f26eb12c55,false] 7104 --- [nio-8091-exec-2] c.p.billing.service.BillingService      : 30000
01:00:33.429  INFO [praveen-flipkart-billing-service,2eee29abbd65d471,9087689b90c2f11b,false] 7104 --- [io-8091-exec-10] c.p.billing.service.BillingService      : 30000
01:00:34.671  INFO [praveen-flipkart-billing-service,03b70041cb6fbba1,33d59972519dca9c,false] 7104 --- [nio-8091-exec-6] c.p.billing.service.BillingService      : 30000
01:00:35.655  INFO [praveen-flipkart-billing-service,1b965bd0bc35f7c7,973c9118c154f531,true] 7104 --- [nio-8091-exec-1] c.p.billing.service.BillingService      : 30000
```

If you want to check logs in PCF then click on View in PCF Metrics Link present under your SpringBoot app.



5. Application Resiliency using Netflix Hystrix Circuit Breaker

What is Circuit Breaker Pattern?

If we design our systems on microservice based architecture, we will generally develop many Microservices and those will interact with each other heavily in achieving certain business goals. Now, all of us can assume that this will give expected result if all the services are up and running and response time of each service is satisfactory.

Now what will happen if any service, of the current Eco system, has some issue and stopped servicing the requests. It will result in timeouts/exception and the whole Eco system will get unstable due to this single point of failure.

Here circuit breaker pattern comes handy and it redirects traffic to a fall back path once it sees any such scenario. Also it monitors the defective service closely and restore the traffic once the service came back to normalcy.

So circuit breaker is a kind of a wrapper of the method which is doing the service call and it monitors the service health and once it gets some issue, the circuit breaker trips and all further calls go to the circuit breaker fall back and finally restores automatically once the service came back.

What circuit breaker does?

- ✓ Failing fast
- ✓ Fallback functionality
- ✓ Automatic recovery

Netflix Hystrix ---> The capacity to recover quickly from failures.

- ✓ Hystrix came into picture on 2011.
- ✓ Hystrix is a latency and fault tolerance library designed to isolate the point of access to
  - remote systems,
  - services and
  - 3rd party libraries
- ✓ Stop cascading failures.
- ✓ Enable resiliency in complex distributed systems where failure is inevitable.

Add below dependencies for Hystrix, Hystric dashboard and Actuator in pom.xml

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

Include @EnableCircuitBreaker and @EnableHystrixDashboard in main class

```
@EnableEurekaClient
@EnableSwagger2
@EnableHystrixDashboard
@EnableCircuitBreaker
//@RibbonClient(name = "billingservice", configuration = RibbonConfiguration.class)
@SpringBootApplication
@EnableFeignClients
public class FlipkartOrderManagementServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(FlipkartOrderManagementServiceApplication.class, args);
    }
```

Add @HystrixCommand to methods that need circuit breakers and also Configure Hystrix behavior (i.e... configuring parameters)

```java
@Service
public class OrderManagementService {

    @Autowired
    RestTemplate restTemplate;

    @Autowired
    @Value("${praveen-flipkart-ordermanagement-service.billingURL}")
    private String billingURL;

    @HystrixCommand(fallbackMethod = "callcreateOrder_Fallback",
            commandProperties = {
                    //https://github.com/Netflix/Hystrix/wiki/Configuration
                    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "5000"),
                    @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold",value="5"),
                    @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage",value="50"),
                    @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds",value="5000")
    })
    public String createOrder(String orderid) throws Exception {
        try {
            HttpHeaders headers = new HttpHeaders();
            headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
            HttpEntity<String> entity = new HttpEntity<String>(orderid, headers);
            // call billing service
            String msg = restTemplate.exchange(billingURL, HttpMethod.POST, entity, String.class).getBody();
            return msg;
        } catch (Exception e) {
            throw new Exception(e);
        }

    }

    @SuppressWarnings("unused")
    private String callcreateOrder_Fallback(String orderid) {
        System.out.println("Billing Service is down!!! fallback route enabled...");
        return "No Response From Billing Service at this moment. " + " Service will be back shortly - " + new Date();
    }

}
```

Now let's consider billing service is up and hit requests POST http://localhost:8090/rest/flipkartordermanagement/submitorder

Curl

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: text/plain' -d '1000' 'http://localhost:8090/rest/flipkar
```

Request URL

```
http://localhost:8090/rest/flipkartordermanagement/submitorder
```

Request Headers

```
{
  "Accept": "*/*"
}
```

Response Body

```
Successfully placed the order 1000
```

Response Code

```
201
```

Response Headers

```
{
  "date": "Wed, 11 Sep 2019 21:01:42 GMT",
  "content-length": "34",
  "content-type": "text/plain;charset=UTF-8"
}
```

Now let's bring down billing service and hit requests POST http://localhost:8090/rest/flipkartordermanagement/submitorder

Curl

```
curl -X POST --header 'Content-Type: application/json' --header 'Accept: text/plain' -d '1000' 'http://localhost:8090/rest/flipkar
```

Request URL

```
http://localhost:8090/rest/flipkartordermanagement/submitorder
```

Request Headers

```
{
  "Accept": "*/*"
}
```

Response Body

```
No Response From Billing Service at this moment.  Service will be back shortly - Thu Sep 12 02:34:00 IST 2019
```

Response Code

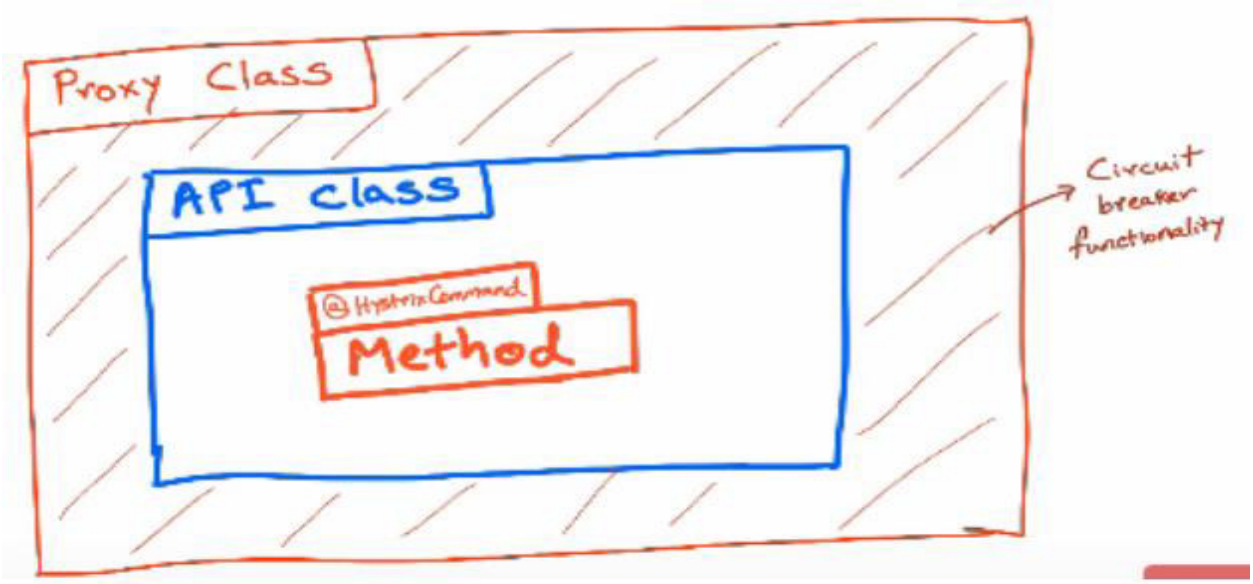
```
201
```

Response Headers

```
{
  "date": "Wed, 11 Sep 2019 21:04:00 GMT",
  "content-length": "109",
  "content-type": "text/plain;charset=UTF-8"
}
```

You can also refer the code from my repository (https://github.com/praveenoruganti/praveenoruganti-springcloud-microservices-master/tree/master/praveen-flipkart-ordermanagement-service)

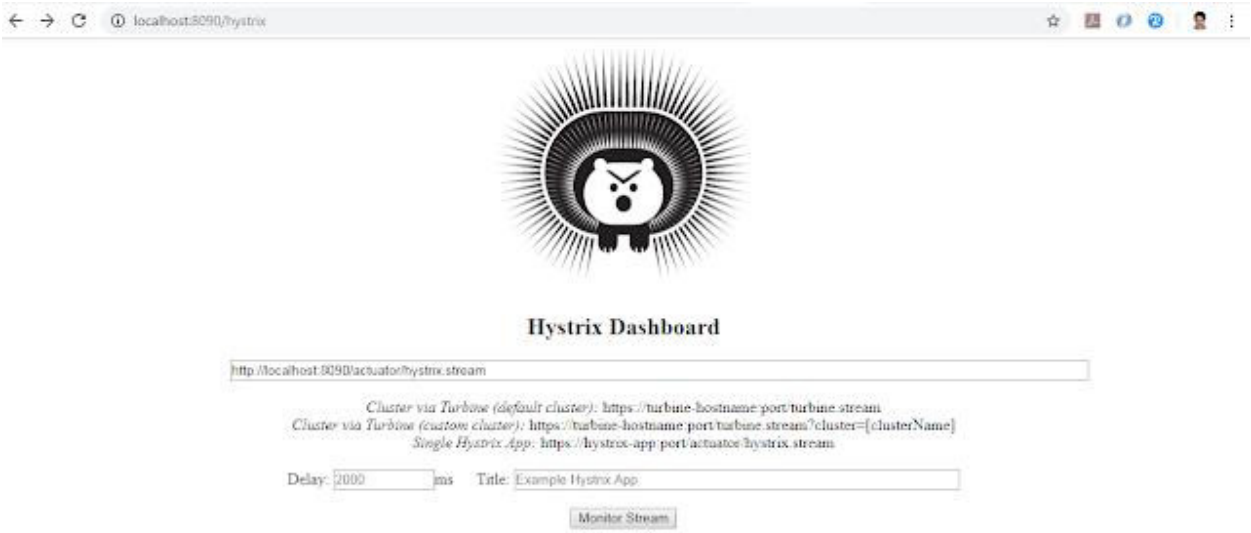
How does circuit breaker works?



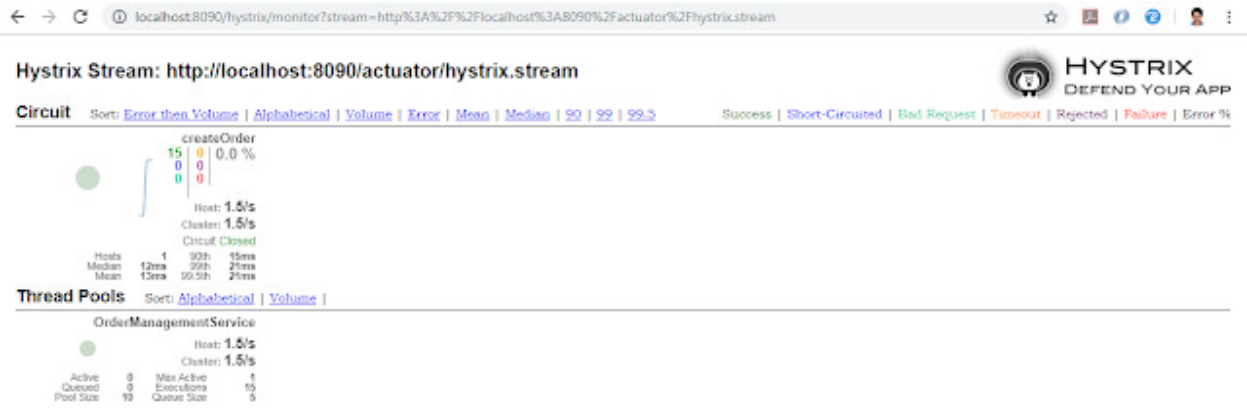
We need to add the below property in application.yml for enabling the hystrix dashboard

```
management:
 endpoints:
    web:
      exposure:
        include: hystrix.stream
```

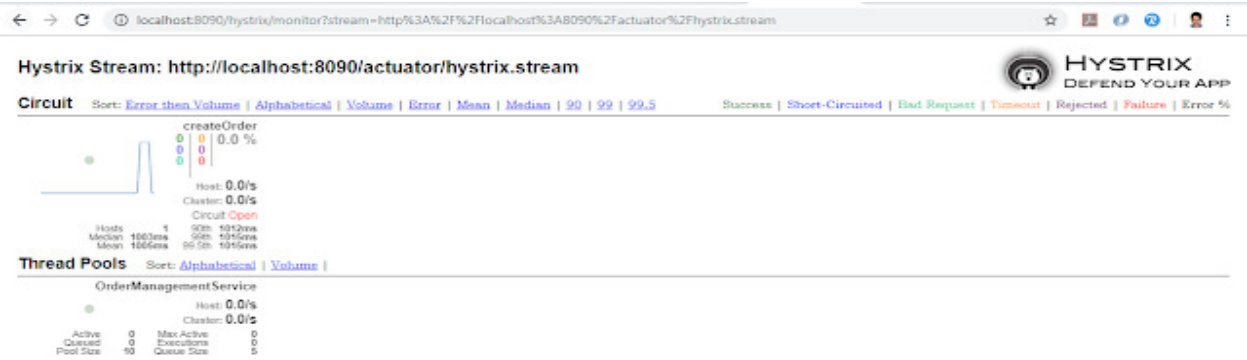
Open URL: http://localhost:8090/actuator/hystrix.stream



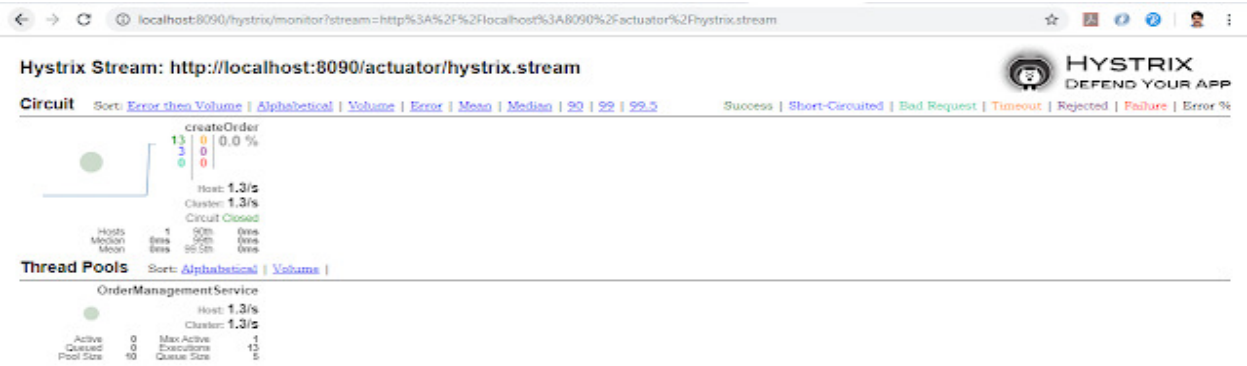
Hit requests POST
http://localhost:8090/rest/flipkartordermanagement/submitorder



Now bring down the praveen-flipkart-billing-service and hit requests POST
http://localhost:8090/rest/flipkartordermanagement/submitorder



Now bring up the praveen-flipkart-billing-service and hit requests POST
http://localhost:8090/rest/flipkartordermanagement/submitorder



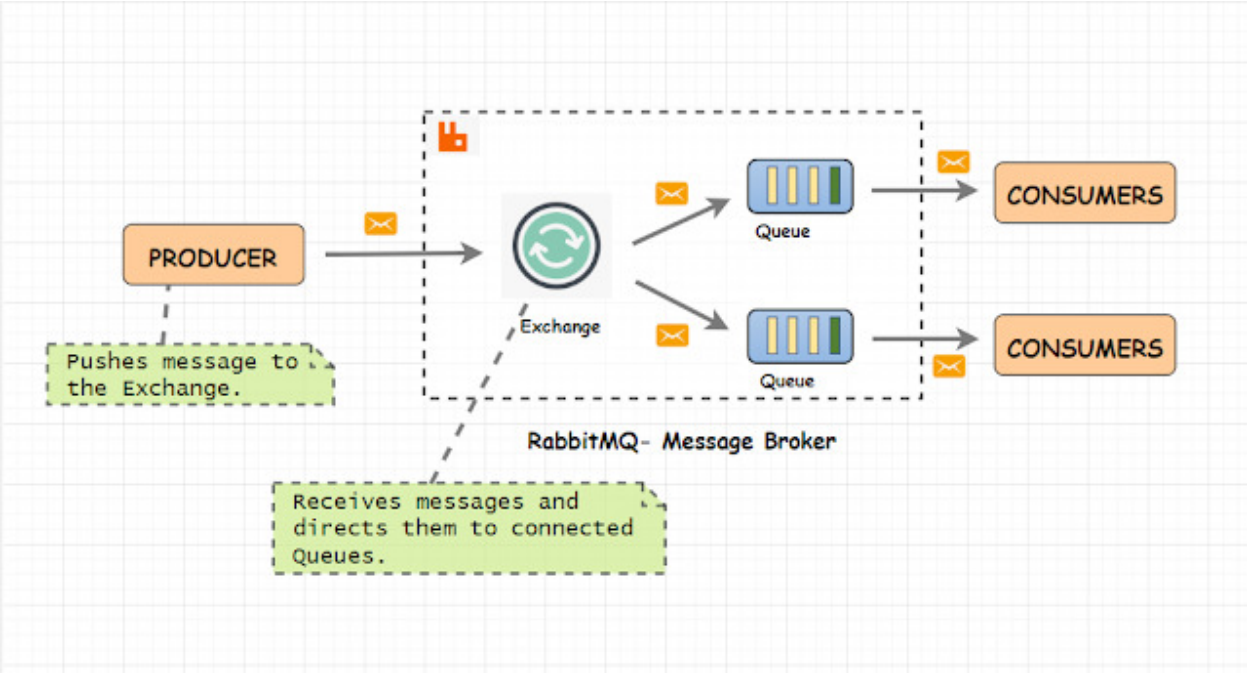
6. Rabbit MQ for Asynchronous calls

RabbitMQ is a message broker. In essence, it accepts messages from **producers**, and delivers them to **consumers**. In-between, it can route, buffer, and persist the messages according to rules you give it.

The way RabbitMQ routes messages depends upon the messaging protocol it implements. RabbitMQ supports multiple messaging protocols. However, the one we are interested in is AMQP. It is an acronym for Advanced Message Queuing Protocol.

Advanced Message Queuing Protocol
The conceptual model of AMQP is quite simple and straightforward. It has three entities:
1.Queue
2.Binding
3.Exchange

When a publisher pushes a message to RabbitMQ, it first arrives at an exchange. The exchange then distributes copies of these messages to variously connected queues. Finally, consumers receive these messages.



Producer:
Producer will send messages to RabbitMQ Exchanges with a routingKey(queuename). RabbitMQ uses routingKey(queuename) to determine which queues for routing messages.

Queue:
Here, our message will be stored. Once consumed by consumers, message will be removed from queue.
RabbitMQ queues also follow FIFO — First-In-First-Out methodology.

Consumer:
Consumer listens on a RabbitMQ Queue to receive messages.

Bindings

Bindings are the rules that a queue defines while establishing a connection with an exchange. You can have a queue connected to multiple exchanges. Every queue is also connected to a default exchange. An exchange will use these bindings to route messages to queues.

Exchange
An Exchange is a gateway to RabbitMQ for your messages. The distance the message has to travel inside RabbitMQ depends on the type of exchange.

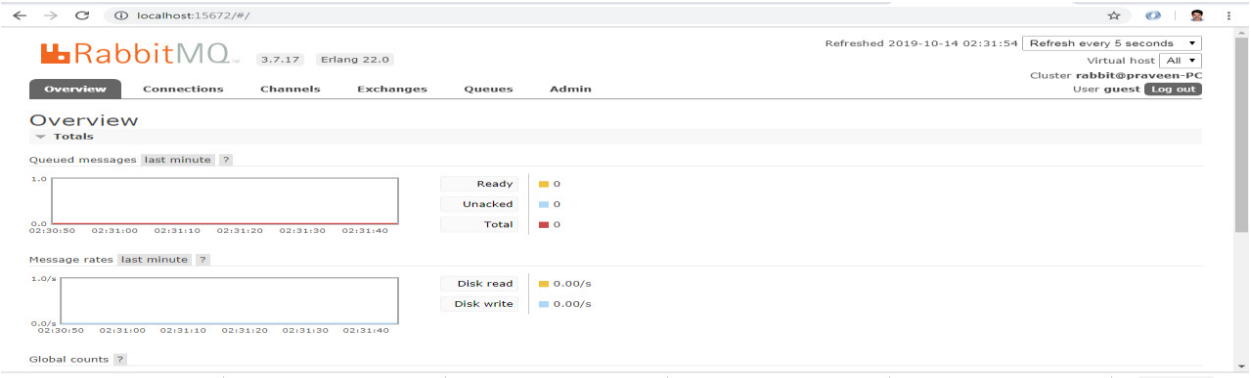
Primarily there are four types of exchanges

- ✓ **Direct Exchange** - It routes messages to a queue by matching routing key equal to binding key. **Routing key == Binding key**
- ✓ **Fanout Exchange** - It ignores the routing key and sends message to all the available queues.
- ✓ **Topic Exchange – It routes messages to multiple queues by a partial matching of a routing key. It** uses patterns to match the routing and binding key. **Routing key == Pattern in binding key.**
- ✓ **Headers Exchange** – It uses message header instead of routing key.
- ✓ **Default(Nameless) Exchange** - It routes the message to queue name that exactly matches with the routing key.

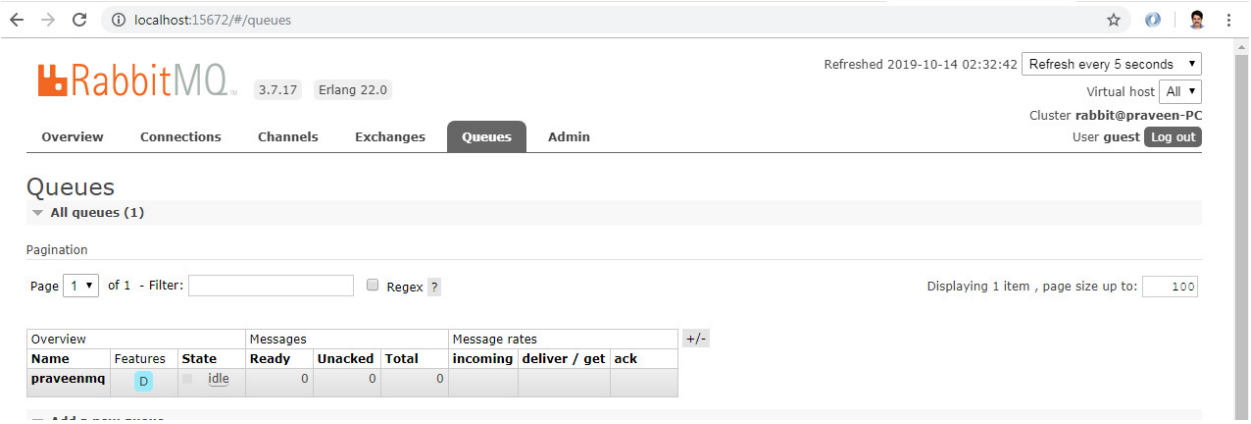
Local Environment Configuration

How to install RabbitMQ in your local?

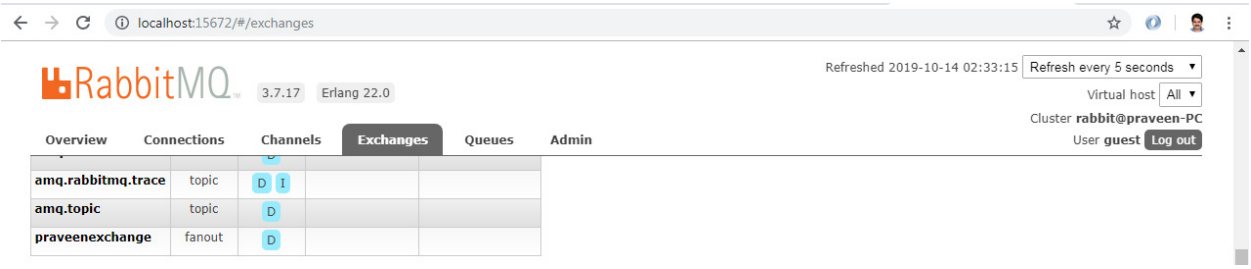
- ✓ Download supporting ERLANG component from here and install.
- ✓ Download Rabbit MQ from here and install
- ✓ Enable management plugin using below command C:\Program Files\RabbitMQ Server\rabbitmq_server-3.7.17\sbin > rabbitmq-plugins enable rabbitmq_management
- ✓ Login into Rabbit MQ browser using URL http://localhost:15672 using userid: guest and password: guest.



✓ Now click on Queues Tab and create queue named praveenmq.



✓ Now click on Exchanges Tab and create exchange named praveenexchange and bind it to praveenmq.



Add below dependency in pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Lets configure the above RabbitMQ properties in application of praveen-user-management-service

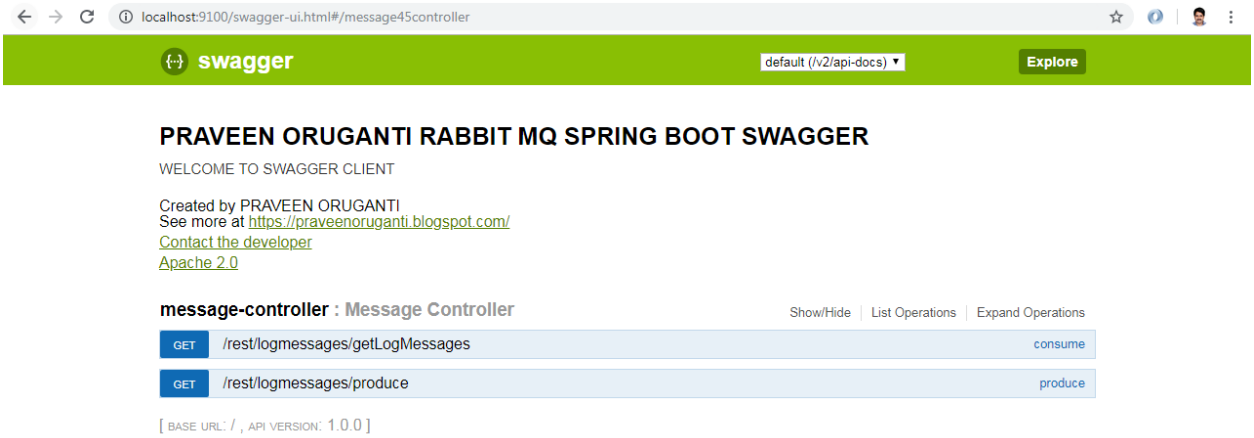
```
  rabbitmq:
   host: localhost
   port: 5672
   username: guest
   password: guest

praveen-rabbitmq-integration-service:
         rabbitmq:
           queueName: praveenmq
           topicExchange: praveenexchange
```

Let's create the producer and consumer components for RabbitMQ in praveen-rabbitmq-integration-service

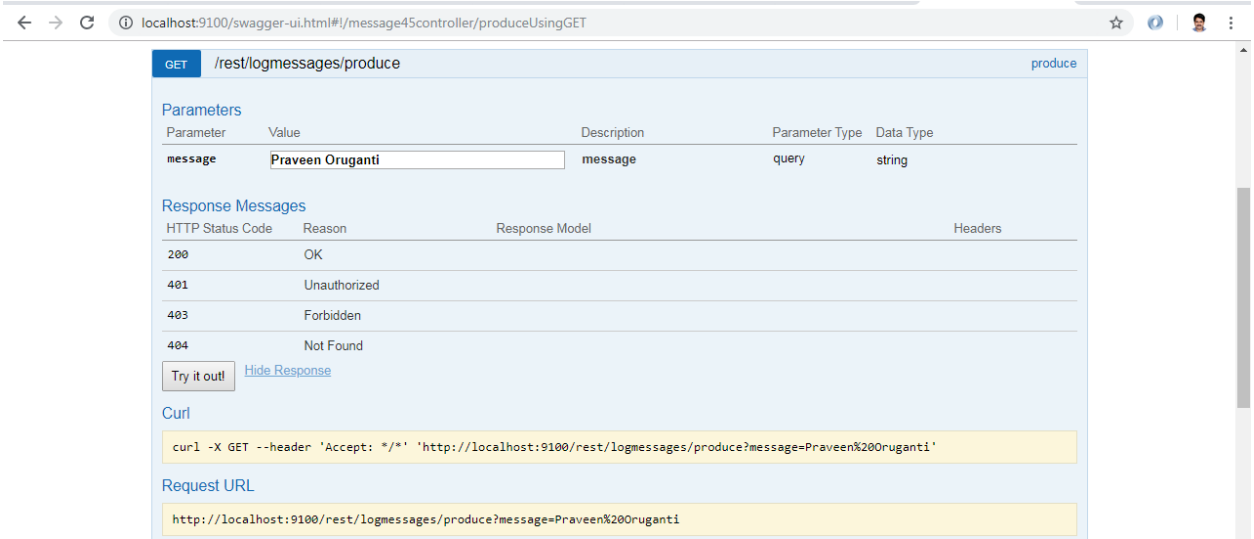
```
@Component
@Slf4j
public class MessageProducer {

    @Autowired
      private AmqpTemplate amqpTemplate;

    @Value("${praveen-rabbitmq-integration-service.rabbitmq.queueName}")
    private String queueName;

    @Value("${praveen-rabbitmq-integration-service.rabbitmq.topicExchange}")
    private String topicExchange;

    public void produceMsg(String msg){
        log.info(LocalDate.now().toString());
        amqpTemplate.convertAndSend(topicExchange, queueName, msg);
        log.info("Send msg = " + msg);
        log.info(LocalDate.now().toString());
      }
}
```

```java
@Component
@Slf4j
public class MessageListener {
    @Autowired
    private PraveenLogDAO praveenLogDAO;

    @RabbitListener(queues = "${praveen-rabbitmq-integration-service.rabbitmq.queueName}")
    public void receivedMessage(String message) {
        log.info(LocalDate.now().toString());
        try {
            Thread.sleep(5000);
            PraveenLog praveenLog = new PraveenLog();
            praveenLog.setLogMessage(message);
            praveenLog.setLogDate(Date.valueOf(LocalDate.now()));
            praveenLogDAO.createLog(praveenLog);
            log.info("Message Received:" + message);
            log.info(LocalDate.now().toString());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

Now open URL http://localhost:9100/swagger-ui.html



Now I pushed the message Praveen Oruganti to Queue



Log information of Message Producer component and it was successfully pushed to praveenmq Queue

```
2019-10-14 03:00:08.015  INFO 6860 --- [nio-9100-exec-1] c.p.rabbitmq.producer.MessageProducer    : 2019-10-14
2019-10-14 03:00:08.059  INFO 6860 --- [nio-9100-exec-1] c.p.rabbitmq.producer.MessageProducer    : Send msg = Praveen Oruganti
2019-10-14 03:00:08.060  INFO 6860 --- [nio-9100-exec-1] c.p.rabbitmq.producer.MessageProducer    : 2019-10-14
```

I see Message Listener fetched the message from praveenmq Queue and pushed to mysql db for tracking purpose.

```
2019-10-14 03:00:13.833  INFO 6860 --- [ntContainer#0-1] c.p.rabbitmq.listener.MessageListener    : Message Received:Praveen Oruganti
2019-10-14 03:00:13.833  INFO 6860 --- [ntContainer#0-1] c.p.rabbitmq.listener.MessageListener    : 2019-10-14
```

I am able to see the message pushed by Message Listener to mysqldb

```
2019-10-14 03:00:08.166  INFO 6860 --- [ntContainer#0-1] c.p.rabbitmq.listener.MessageListener    : 2019-10-14
2019-10-14 03:00:13.183  INFO 6860 --- [ntContainer#0-1] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2019-10-14 03:00:13.591  INFO 6860 --- [ntContainer#0-1] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed
Message Listener Pushed the Message Praveen Oruganti to mysql db
```

Let's see the message in mysql db

SELECT * from PRAVEENLOG

PRAVEENLOG_ID	PRAVEENLOG_MESSAGE	PRAVEENLOG_DATE
1	Praveen Oruganti	2019-10-14

You can refer the code from my got repository (https://github.com/praveenoruganti/praveenoruganti-springcloud-microservices-master/tree/master/praveen-rabbitmq-integration-service)

PCF Configuration For RabbitMQ

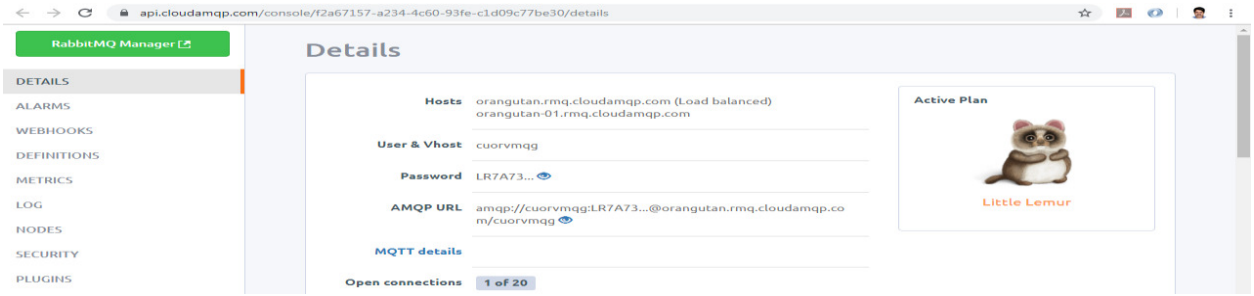
All other code and configurations will be similar to Local Environment changes except we will use CloudAMQP instead of local rabbitmq.

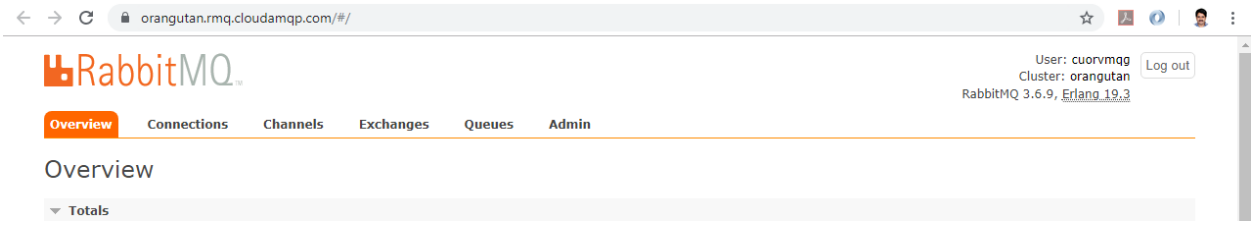
Let's start creating the PCF Service Rabbit MQ

- ✓ **Click on ADD A SERVICE button and search for Rabbit MQ in Market Place**

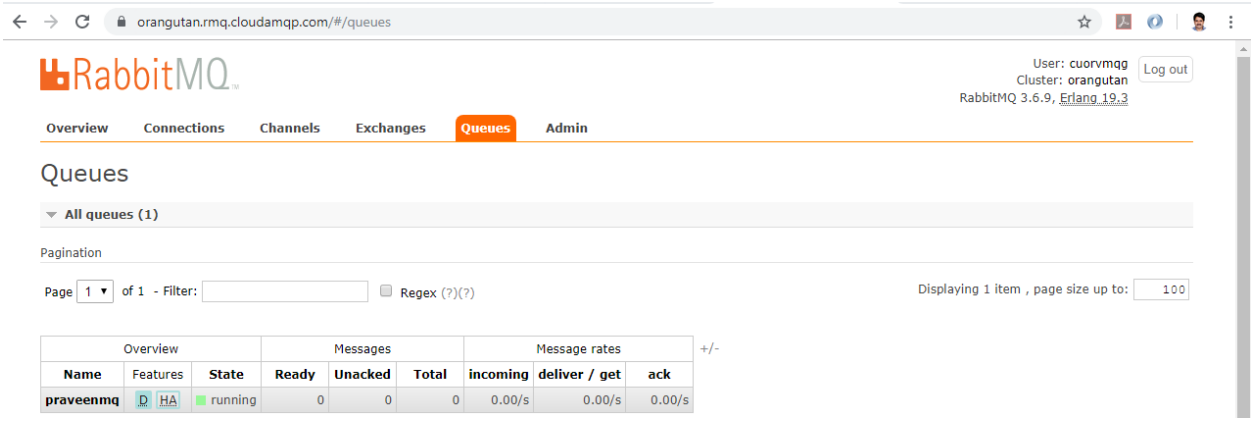
Provide the name of the service as praveen-rabbitmq and bind to the praveen-user-management-service.
Please find the below screenshot for you reference.

- ✓ **Click on Manage button and then click on RabbitMQ Manager button**

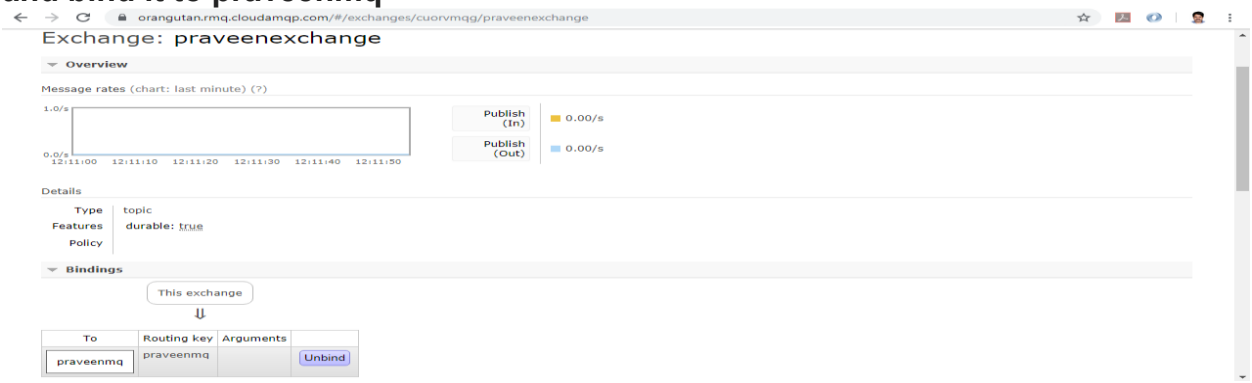




✓ **Now click on Queues Tab and create queue named praveenmq**



✓ **Now click on Exchanges Tab and create exchange named praveenexchange and bind it to praveenmq**



7. Apache Kafka for Asynchronous Calls

Apache Kafka

Apache Kafka is a distributed streaming platform with capabilities such as publishing and subscribing to a stream of records, storing the records in a fault tolerant way, and processing that stream of records.

It is used to build real-time streaming data pipelines, that can perform functionalities such as reliably passing a stream of records from one application to another and processing and transferring the records to the target applications.

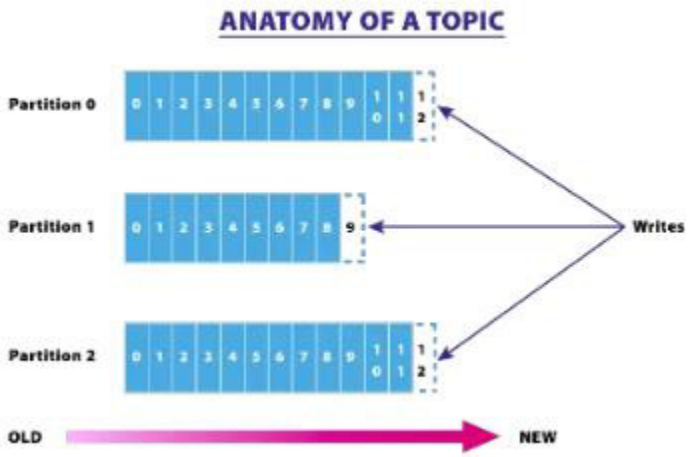
Topics

Kafka is run as a cluster in one or more servers and the cluster stores/retrieves the records in a feed/category called Topics. Each record in the topic is stored with a key, value, and timestamp.

The topics can have zero, one, or multiple consumers, who will subscribe to the data written to that topic. In Kafka terms, topics are always part of a multi-subscriber feed.

Partitions

The Kafka cluster uses a partitioned log for each topic.



The partition maintains the order in which data was inserted and once the record is published to the topic, it remains there depending on the retention period (which is configurable). The records are always appended at the end of the partitions. It maintains a flag called 'offsets,' which uniquely identifies each record within the partition.

The offset is controlled by the consuming applications. Using offset, consumers might backtrace to older offsets and reprocess the records if needed.

Producers

The stream of records, i.e. data, is published to the topics by the producers. They can also assign the partition when it is publishing data to the topic. The producer can send data in a round robin way or it can implement a priority system based on sending records to certain partitions based on the priority of the record.

Consumers

Consumers consume the records from the topic. They are based on the concept of a consumer-group, where some of the consumers are assigned in the group. The record which is published to the topic is only delivered to one instance of the consumer from one consumer-group. Kafka internally uses a mechanism of consuming records inside the consumer-group. Each instance of the consumer will get hold of the particular

partition log, such that within a consumer-group, the records can be processed parallelly by each consumer.

Spring Boot Kafka Application

Spring provides good support for Kafka and provides the abstraction layers to work with over the native Kafka Java clients.

We can add the below dependencies to get started with Spring Boot and Kafka.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

Steps to download and run Apache Kafka

Step 1: Download the apache kafka and unzip using winrar

http://apachemirror.wuchna.com/kafka/2.3.0/kafka_2.12-2.3.0.tgz

Step 2: Start the server

Once you download Kafka, you can issue a command to start ZooKeeper which is used by Kafka to store metadata.

D:\Praveen\Softwares\kafka_2.12-2.3.0>bin\windows\zookeeper-server-start.bat config\zookeeper.properties

Next, we need to start the Kafka cluster locally by issuing the below command.

D:\Praveen\Softwares\kafka_2.12-2.3.0>bin\windows\kafka-server-start.bat .\config\server.properties

Now, by default, the Kafka server starts on localhost:9092.

Let's develop a simple REST controller with Swagger integration and expose with one endpoint, /publish, as shown below. It is used to publish the message to the topic.

KafkaController.java

```java
package com.praveen.kafka.controller;

import org.springframework.beans.factory.annotation.Autowired;

@RestController
@Api(tags = "Apache Kafka Restful Service", value = "KafkaController", description = "Controller for Kafka Publishing")
@RequestMapping(value = "/rest/kafka")
public class KafkaController {

    @Autowired
    ProducerService producerService;

    @PostMapping(value = "/publish")
    @ApiOperation(value = "Publishing Message")
    public void sendMessageToKafkaTopic(@RequestParam("message") String message) {
        this.producerService.sendMessage(message);
    }

}
```

We can then write the producer which uses Spring's KafkaTemplate to send the message to a topic named users, as shown below.

ProducerService.java

```java
package com.praveen.kafka.service;

import org.slf4j.Logger;

@Service
public class ProducerService {
    private static final Logger logger = LoggerFactory.getLogger(ProducerService.class);
    private static final String TOPIC = "users";
    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate;

    public void sendMessage(String message) {
        logger.info(String.format("$$ -> Producing message --> %s", message));
        this.kafkaTemplate.send(TOPIC, message);
    }
}
```

We can also write the consumer as shown below, which consumes the message from the topic users and output the logs to the console.

ConsumerService.java

```java
package com.praveen.kafka.service;

import org.slf4j.Logger;

@Service
public class ConsumerService {
    private final Logger logger = LoggerFactory.getLogger(ConsumerService.class);

    @KafkaListener(topics = "users", groupId = "group_id")
    public void consume(String message) {
        logger.info(String.format("$$ -> Consumed Message -> %s", message));
    }
}
```

Now, we need a way to tell our application where to find the Kafka servers and create a topic and publish to it. We can do it using application.yaml as shown below.

application.yml

```yaml
server:
  port: 9091
spring:
  application:
    name: praveen-springboot-kafka
  kafka:
    consumer:
      bootstrap-servers: localhost:9092
      group-id: group-id
      auto-offset-reset: earliest
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
    producer:
      bootstrap-servers: localhost:9092
      key-deserializer: org.apache.kafka.common.serialization.StringDeserializer
      value-deserializer: org.apache.kafka.common.serialization.StringDeserializer
```

Let's see the main application class with Swagger integration

SpringbootKafkaApplication.java

```java
package com.praveen.kafka;

import static springfox.documentation.builders.PathSelectors.regex;

@SpringBootApplication
@EnableSwagger2
public class SpringbootKafkaApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootKafkaApplication.class, args);
    }

    @Bean
    public Docket configDock() {
        return new Docket(DocumentationType.SWAGGER_2).select().apis(basePackage("com.praveen.kafka.controller"))
                .paths(regex("/rest.*")).build().apiInfo(apiInfo());
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder().title("SpringBootKafka").description("WELCOME TO SWAGGER CLIENT")
                .contact(new Contact("PRAVEEN ORUGANTI", "https://praveenoruganti.blogspot.com/",
                        "praveenoruganti@gmail.com"))
                .license("Apache 2.0").licenseUrl("http://www.apache.org/licenses/LICENSE-2.0.html").version("1.0.0")
                .build();
    }

}
```

Now, if we run the application and hit the endpoint as shown below, we have published a message to the topic.

Now, if we check the logs from the console, it should print the message which was sent to the publish endpoint as seen below.

```
2019-10-25 11:21:41.192  INFO 2296 --- [nio-9091-exec-5] o.a.kafka.common.utils.AppInfoParser     : Kafka version : 2.0.1
2019-10-25 11:21:41.193  INFO 2296 --- [nio-9091-exec-5] o.a.kafka.common.utils.AppInfoParser     : Kafka commitId : fa14705e51bd2ce5
2019-10-25 11:21:41.217  INFO 2296 --- [ad | producer-1] org.apache.kafka.clients.Metadata        : Cluster ID: jEcl-3zVQiupjjG4kyV-Eg
2019-10-25 11:21:41.354  INFO 2296 --- [ntainer#0-0-C-1] c.praveen.kafka.service.ConsumerService  : $$ -> Consumed Message -> praveen oruganti
```

You can find the complete code of this in my repository(https://github.com/praveenoruganti/praveenoruganti-springcloud-microservices-master/tree/master/praveenoruganti-springboot-kafka)

8. MQSQL DB and Redis integration

What Is Redis?

Redis is an open-source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. It supports data structures such as string, hashes, lists, sets, sorted sets with range queries, bitmaps, hyper logs, and geospatial indexes with radius queries.

Redis is written in ANSI C and works in most POSIX systems like Linux, BSD, and OS X without external dependencies. Linux and OS X are the two operating systems where Redis is developed and more tested. Redis may work in Solaris-derived systems like SmartOS. There is no official support for Windows builds, but Microsoft develops and maintains a Win-64 port of Redis.

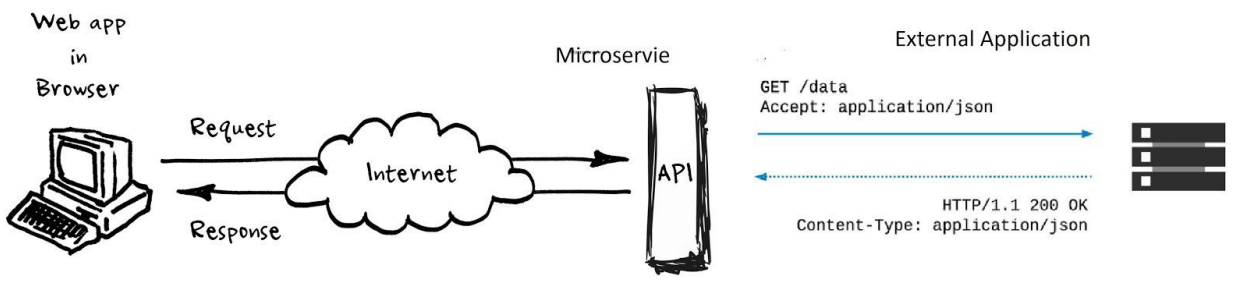
Why Redis?

Redis is basically used for cache management. It reduces the client workload and speeds up the application.

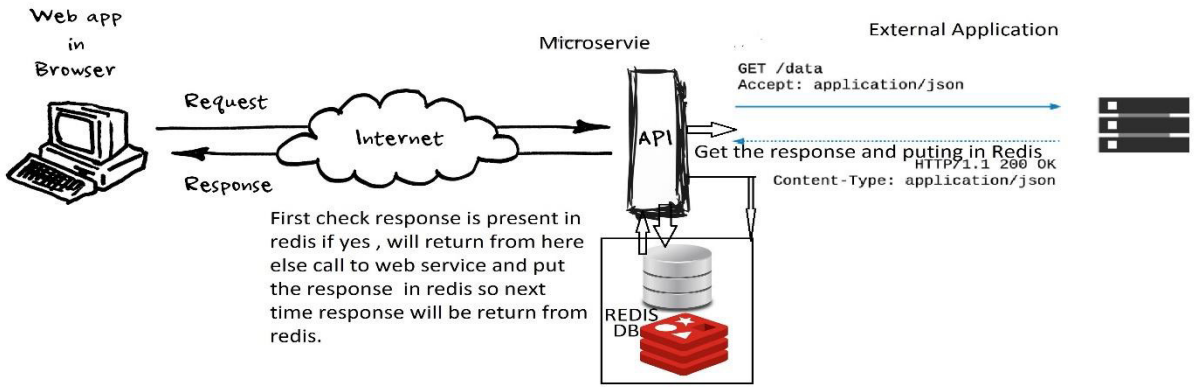
Let's discuss a scenario where Redis could be helpful.

a) Calling an External Application API

Your microservice is calling other application's API in order to get some data. The call is made often, the data in response is huge, and you know the response data is not going to change often in the other application. In our case, we are calling one API and we get the response in 10000 to 15000 ms. It is not good for any application to wait for such a long time to get the response. In that case, Redis is helpful.



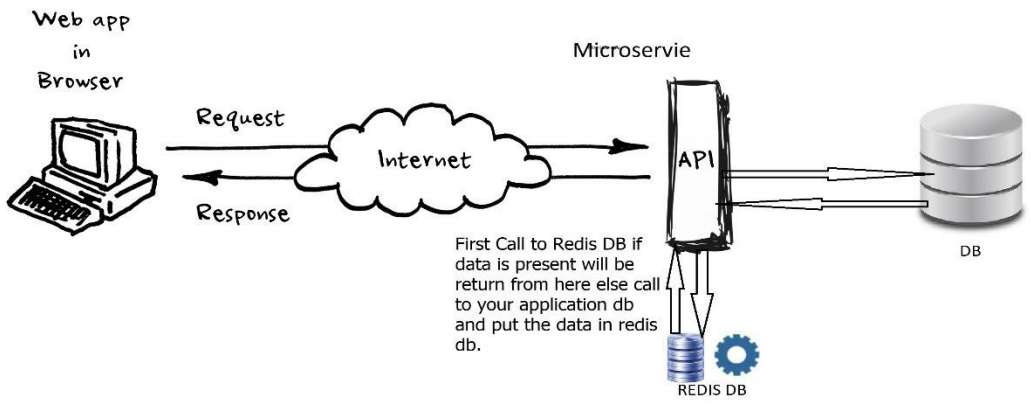
We can cache the response in Redis with an expiration time so instead of the actual call, you will get the response from the Redis cache. In our case, we had set the expiration time of the data to 1 day. When the data expires, the call goes to the actual web service and refreshes the data in the cache. The next time, the data will be returned from the cache.



Here, we reduce the load time of the data by 90% and we were getting the same data in between 7000 to 1000 milliseconds.

b) Frequently Querying the Reference Table or Master Table in the Database

Another scenario is when you have a table in your database, and that table contains reference data that is not going to change frequently. You query that table often to get the data and populate the UI. In this scenario, you can also use Redis. Instead of querying the database each time from the disk, you can cache the table in Redis with an expiration time so the load time of your UI will be faster.



Let's create user-management-service springboot application with spring-jdbc component for mysqldb with Hikari connection pool and spring-data-redis for redis integration.

Local Environment Changes

Install Redis in Local

In Windows, you just have to download the zip and extract it into any location. Open redis-server.exe from the extracted folder and the Redis server will be started.

After that, you can open redis-cli.exe from the same folder. redis-cli is the Redis command line interface, a simple program that allows you to send commands to Redis and read the replies sent by the server directly from the terminal.

Add below dependencies in pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.apache.tomcat</groupId>
            <artifactId>tomcat-jdbc</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <scope>runtime</scope>
</dependency>

 <dependency>
     <groupId>org.springframework.boot</groupId>
     <artifactId>spring-boot-starter-data-redis</artifactId>
 </dependency>
```

Add below properties in application.yml

```
redis:
   host: localhost
   port: 6379


  datasource:
         type: com.zaxxer.hikari.HikariDataSource
         driver-class-name: com.mysql.cj.jdbc.Driver
         url: jdbc:mysql://root:password@localhost:3306/praveendb?reconnect=true
         username: root
         password: password
         hikari:
           connectionTimeout : 30000
           idleTimeout : 600000
           maxLifetime : 1800000
           maximumPoolSize : 5
```

Redis Configuration

Implement the Redis Configuration class in your microservice that is going to establish the connection with Redis.

- ✓ **RedisStandaloneConfiguration**: This class used for setting up the connection for single node Redis installation.
- ✓ **JedisClientConfiguration**: This class provides an optional feature for SSLSocketFactory and JedisPoolConfig specific to Jedis client features.

- ✓ **JedisConnectionFactory**: This is the connection factory for the Jedis base connection. It uses the JedisClientConfiguration and RedisStandaloneConfiguration.
- ✓ **RedisTemplate<K,V>:** This helper simplifies data access for Redis. It uses automatic serialization and deserialization for a given object and underlying binary data in the Redis store.

Create RedisConfigurationConfiguration class

```
package com.praveen.restservices.config;

import org.springframework.beans.factory.annotation.Value;

@Getter
@Setter
@RefreshScope
@Configuration
public class RedisConfiguration {
    @Value("${spring.redis.host}")
    private String REDIS_HOSTNAME;
    @Value("${spring.redis.port}")
    private int REDIS_PORT;

    @Bean
    protected JedisConnectionFactory jedisConnectionFactory() {
        RedisStandaloneConfiguration configuration = new RedisStandaloneConfiguration(REDIS_HOSTNAME, REDIS_PORT);
        JedisClientConfiguration jedisClientConfiguration = JedisClientConfiguration.builder().usePooling().build();
        JedisConnectionFactory factory = new JedisConnectionFactory(configuration, jedisClientConfiguration);
        factory.afterPropertiesSet();
        return factory;
    }

    @Bean
    public RedisTemplate<String, Object> redisTemplate() {
        final RedisTemplate<String, Object> redisTemplate = new RedisTemplate<String, Object>();
        redisTemplate.setKeySerializer(new StringRedisSerializer());
        redisTemplate.setHashKeySerializer(new GenericToStringSerializer<Object>(Object.class));
        redisTemplate.setHashValueSerializer(new JdkSerializationRedisSerializer());
        redisTemplate.setValueSerializer(new JdkSerializationRedisSerializer());
        redisTemplate.setConnectionFactory(jedisConnectionFactory());
        return redisTemplate;
    }
}
```

Now that the connection is established, we have to perform an operation with the Redis database, so we have created one generic util class that is going to perform the operation on the Redis database for the different data structure.

RedisTemplate provides many methods in order to perform an operation for multiple data structures.

We have used some in the below classes:

- ✓ **opsForHash():** Return the HashOperations<K,HK,HV> class. This class use for hash operations on Redis.
- ✓ **opsForList():** Return the ListOperations<K,V> used for list operations.
- ✓ **opsForValue():** Return the ValueOperations<K,V> Perform simple key value operations; each key will have an entry in Redis for its associated value

Let's create DAO and DAOImpl classes

```java
package com.praveen.restservices.dao;

import java.util.List;

public interface UserDAO1 {
    abstract List<User1> findAll1() throws Exception;
    abstract User1 findUserById1(String userid1) throws Exception;
    abstract int create1(final User1 user1) throws Exception;
    abstract void deleteByUserId1(final String userId1) throws Exception
    abstract int updateUserbyId1(User1 user1) throws Exception;

}
```

```java
package com.praveen.restservices.dao.impl;

import java.sql.ResultSet;

@Repository
public class UserDAOImpl1 implements UserDAO1 {
    private JdbcTemplate jdbcTemplate;

    @Autowired
    UserDAOImpl1(DataSource dataSource1) {
        this.jdbcTemplate = new JdbcTemplate(dataSource1);
    }

    class UserRowMapper1 implements RowMapper<User1> {
        @Override
        public User1 mapRow(ResultSet rs, int rowNum) throws SQLException {
            User1 user1 = new User1();
            user1.setUserId(rs.getInt("userId"));
            user1.setUserName(rs.getString("userName"));
            user1.setUserEmail(rs.getString("userEmail"));
            user1.setAddress(rs.getString("address"));
            return user1;
        }
    }

    @Override
    @Transactional
    public List<User1> findAll1() throws Exception {
        List<User1> userList = jdbcTemplate.query("select * from users1", new UserRowMapper1());
        if (userList.size() > 0) {
            return userList;
        } else {
            throw new Exception("No User records found in DB");
        }
```

```java
    @Override
    @Transactional
    public User1 findUserById1(String userid1) throws Exception {
        if (isUserExistsByID(userid1)) {
            User1 user1 = jdbcTemplate.queryForObject("select * from users1 where userId=?",
                    new Object[] { Integer.parseInt(userid1) }, new UserRowMapper1());
            return user1;
        } else {
            throw new Exception("User Doesn't Exist");
        }
    }

    @Override
    @Transactional
    public int create1(User1 user1) throws Exception {
        if (!isUserExistsByID(String.valueOf(user1.getUserId())) && !isUserExists(user1)) {
            logger.info("create1 user1id "+ user1.getUserId() +" inserted in DB");
            final String insertSql = "insert into users1(userId,userName,userEmail,address) values(?,?,?,?)";
            Object[] params = { user1.getUserId(), user1.getUserName(), user1.getUserEmail(), user1.getAddress() };
            return jdbcTemplate.update(insertSql, params);
        } else {
            throw new Exception("User Already Exists");
        }
    }

    @Override
    @Transactional
    public void deleteByUserId1(String userId1) throws Exception {
        if (isUserExistsByID(userId1)) {
            logger.info("Deleted userId "+ userId1 +" from DB");
            final String deleteSql = "delete from users1 where userId=?";
            jdbcTemplate.update(deleteSql, userId1);
        } else {
            throw new Exception("User doesn't Exists");
        }
```

```java
    }

    public boolean isUserExistsByID(String userid1) {
        final String sql = "select COUNT(1) from users1 where userId=?";
        Object[] params = { Integer.parseInt(userid1) };
        int row = jdbcTemplate.queryForObject(sql, params, Integer.class);
        if (row > 0) {
            return true;
        } else {
            return false;
        }
    }

    public boolean isUserExists(User1 user1) {
        final String sql = "select COUNT(1) from users1 where userName=? and userEmail=? and address=?";
        Object[] params = { user1.getUserName(), user1.getUserEmail(), user1.getAddress() };
        int row = jdbcTemplate.queryForObject(sql, params, Integer.class);
        if (row > 0) {
            return true;
        } else {
            return false;
        }
    }

    @Override
    public int updateUserbyId1(User1 user1) throws Exception {
        if (isUserExistsByID(String.valueOf(user1.getUserId()))) {
            final String updateSql = "update users1 set userName=? , userEmail=? , address=? where userId=?";
            int rows = jdbcTemplate.update(updateSql, user1.getUserName(), user1.getUserEmail(), user1.getAddress(),
                    user1.getUserId());
            return rows;
        } else {
            throw new Exception("User Doesn't Exist");
        }
    }
```

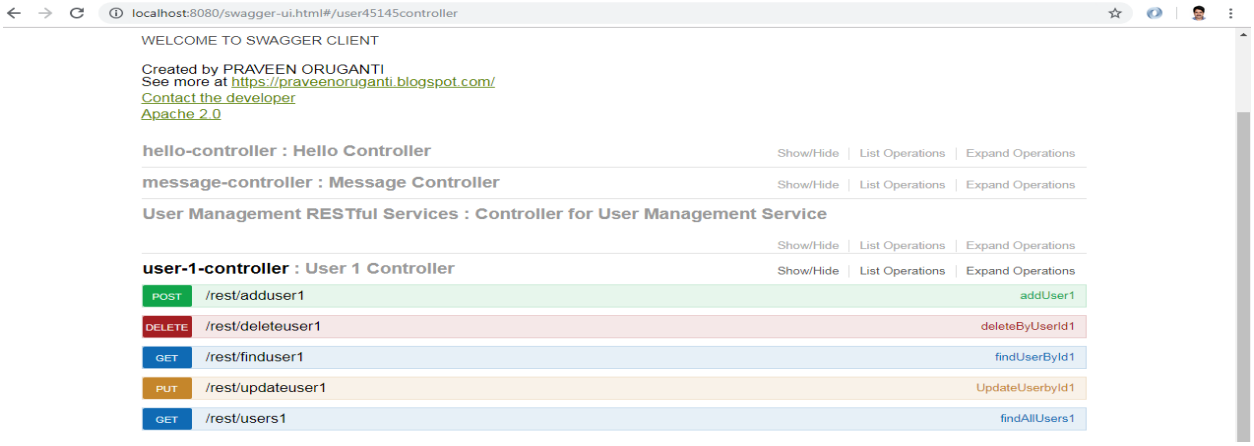
Let's see User1 model class

```java
package com.praveen.restservices.model;

import java.io.Serializable;

//DROP TABLE IF EXISTS users1;
//CREATE TABLE users1(userId int NOT NULL DEFAULT '0',userName VARCHAR(100) NOT NULL,userEmail VARCHAR(100) DEFAULT NULL,address VARCHAR(100) DEFAULT NULL, PRI
@NoArgsConstructor
@AllArgsConstructor
@Data
public class User1  implements Serializable{
    private static final long serialVersionUID = 7096186377859686600L;
    private Integer userId;
    private String userName;
    private String userEmail;
    private String address;
}
```

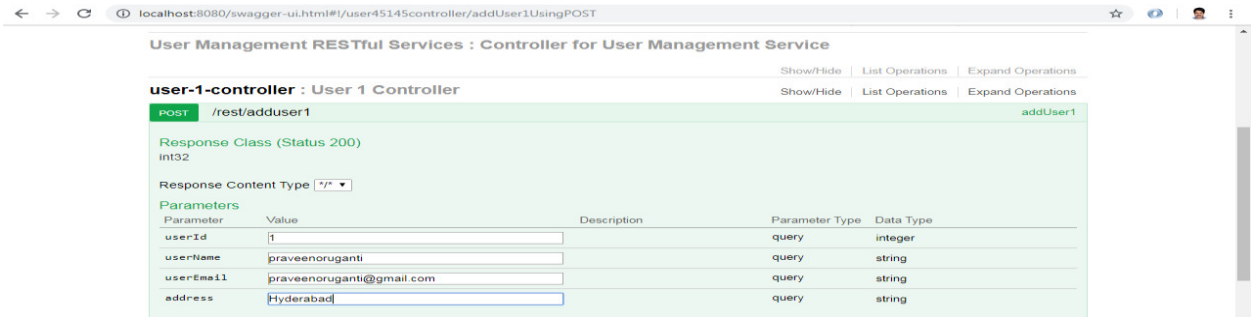
Now let's move on to Service class where we will use redistemplate for caching the user data.

```java
package com.praveen.restservices.service;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.redis.core.HashOperations;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

import com.praveen.restservices.dao.UserDAO1;
import com.praveen.restservices.model.User1;

@Service
public class User1Service {
    @Autowired
    private UserDAO1 userDAO1;
    private HashOperations hashOperations;
    private RedisTemplate redisTemplate;
    @Autowired
    public User1Service(RedisTemplate redisTemplate) {
        this.redisTemplate = redisTemplate;
        this.hashOperations = redisTemplate.opsForHash();
    }

    @Transactional(readOnly = true)
    public List<User1> findAll1() throws Exception {
        return userDAO1.findAll1();
    }

    -

    @Transactional(readOnly = true)
    public User1 findUserById1(String userid1) throws Exception {
        if (hashOperations.entries("USER").containsKey(userid1)) {
            logger.info("findUserById1 Cache Data "+ userid1);
            return (User1)hashOperations.get("USER", userid1);
        } else {
            logger.info("findUserById1 DB Data "+ userid1);
            return userDAO1.findUserById1(userid1);
        }
    }
    @Transactional(isolation=Isolation.READ_COMMITTED,propagation=Propagation.REQUIRED,readOnly=false,timeout=100,rollbackFor=Exception.class)
    public int create1(User1 user1) throws Exception {
        logger.info("create1 user1id "+ user1.getUserId() +" inserted in Cache");
        hashOperations.put("USER", String.valueOf(user1.getUserId()), user1);
        return userDAO1.create1(user1);
    }

    public void deleteByUserId1(String userId) throws Exception {
        if (hashOperations.entries("USER").containsKey(userId)) {
            logger.info("Deleted userId "+ userId +" from Cache");
            hashOperations.delete("USER", userId);
        }
         userDAO1.deleteByUserId1(userId);
    }

    public int updateUserbyId1(User1 user1) throws Exception {
        if (hashOperations.entries("USER").containsKey(String.valueOf(user1.getUserId()))) {
            logger.info("updateUserbyId1  "+ user1.getUserId() +" in Cache");
            hashOperations.put("USER", String.valueOf(user1.getUserId()), user1);
        }
        return userDAO1.updateUserbyId1(user1);
    }
}
```

Now open URL: http://localhost:8080/swagger-ui.html and go to User1Controller operations



Create data for user

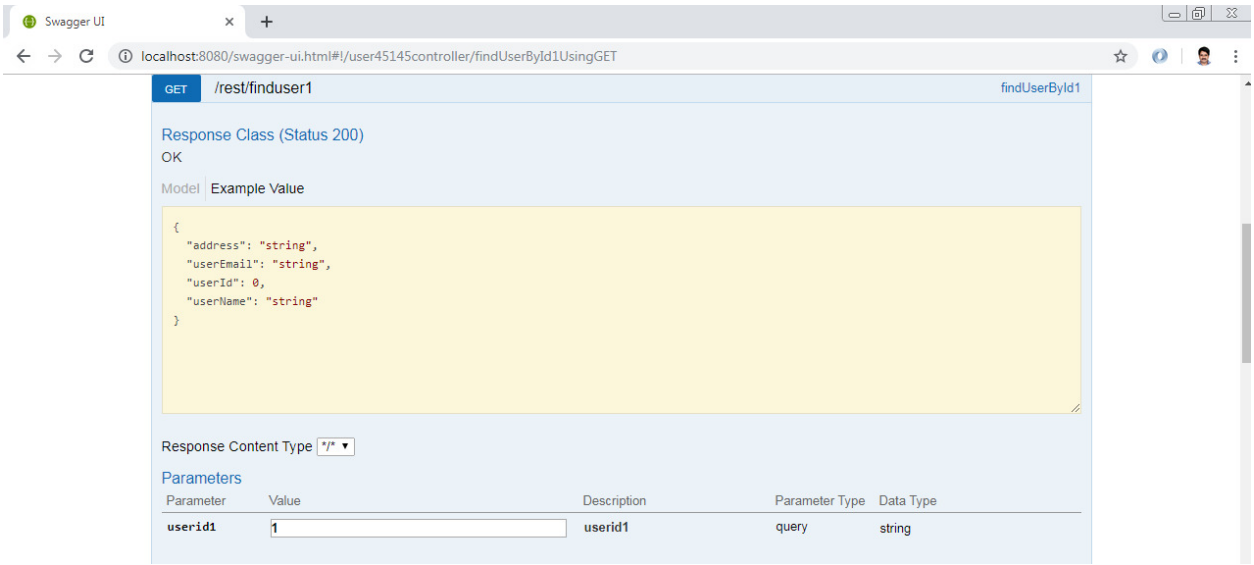


Logs clearly shows that the data has been inserted in Cache and DB as well.

```
04:37:07.010 [http-nio-8080-exec-1] INFO  c.p.r.service.User1Service - create1 user1id 1 inserted in Cache
04:37:07.714 [http-nio-8080-exec-1] INFO  c.p.r.dao.impl.UserDAOImpl1 - create1 user1id 1 inserted in DB
```

userId	userName	userEmail	address
1	praveenoruganti	praveenoruganti@gmail.com	Hyderabad

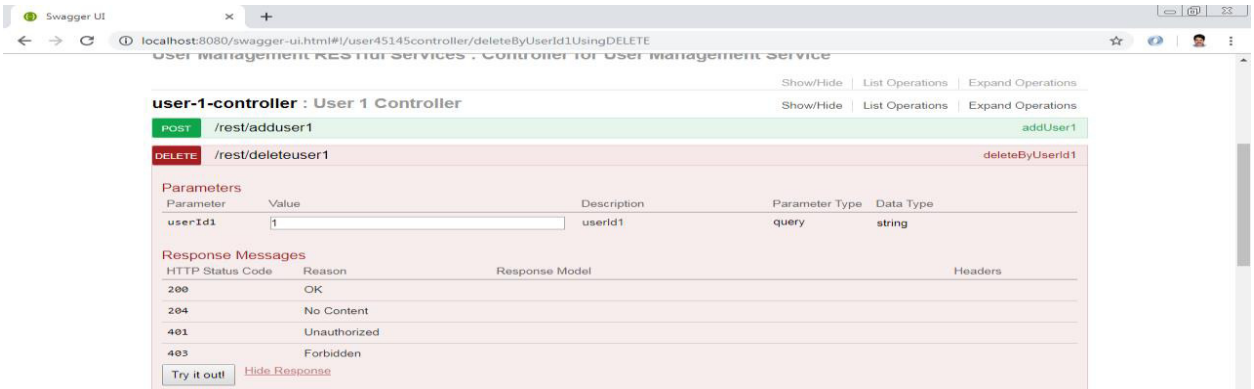
Let's perform findUserById1 operation



Log clearly shows data is fetched from Cache

```
04:41:25.363 [http-nio-8080-exec-5] INFO  c.p.r.service.User1Service - findUserById1 Cache Data 1
```

Let's perform deleteByUserId1 operation



Let's see the log and it clearly shows the data has been deleted from Cache and DB.

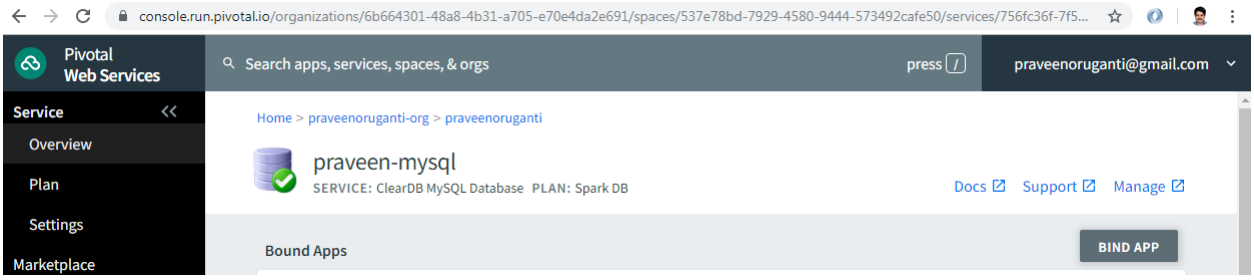
```
04:43:58.611 [http-nio-8080-exec-7] INFO  c.p.r.service.User1Service - Deleted userId 1 from Cache
04:43:58.617 [http-nio-8080-exec-7] INFO  c.p.r.dao.impl.UserDAOImpl1 - Deleted userId 1 from DB
```

userId	userName	userEmail	address
NULL	NULL	NULL	NULL

You can refer the code from my git repository (https://github.com/praveenoruganti/praveenoruganti-springcloud-microservices-master/tree/master/praveen-user-management-service)

PCF Configuration for mysql and redis service

Login into PCF and go to market place and select ClearDB MySQL Database service.



Login into PCF and go to market place and select Redis Cloud service

Please note once you create any PCF service you need to click on BIND APP button and select appropriate SpringBoot application to bind to the service.