Spring Cloud

Cloud computing promises many benefits, such as cost advantage, speed, agility, flexibility, and elasticity. There are many cloud providers such as Amazon (AWS), Microsoft, (Azure), Rackspace (Rackspace), IBM (Blue Mix), Google (Google cloud), Pivotal (Cloud Foundry), Redhat (Open Shift), etc. manual deployment could severely challenge the microservices rollouts. With many server instances running, this could lead to significant operational overheads. Moreover, the chances of errors are high in this manual approach.

Microservices require a supporting elastic cloud-like infrastructure which can automatically provision VMs or containers, automatically deploy applications, adjust traffic flows, replicate new version to all instances, and gracefully phase out older versions. The automation also takes care of scaling up elastically by adding containers or VMs on demand, and scaling down when the load falls below threshold.

**The application (in our case Microservice) has to follow and check to Twelve-factor rules before moving to cloud** i.e., in order to run Microservices seamlessly (means portably) across multiple cloud providers, it is important to follow Twelve-factor rules while developing cloud native microservices.

Spring Cloud by itself is not a cloud , Rather, it provides a number of capabilities that are essential when developing applications targeting cloud deployments that adhere to the Twelve-Factor application principles.

The cloud-ready solutions that are developed using Spring Cloud are also agnostic and portable across many cloud providers such as Cloud Foundry, AWS, Heroku, and so on.

The spring cloud related capabilities such as Load balancer, Service Registry, Configuration Service, Cloud Messaging, API Gateways, etc are highlighted in green colour.



**Twelve-Factor Apps**

Twelve-Factor Apps defines a set of principles of developing applications targeting the cloud.

Many Organizations prefer to lift and shift their applications to the cloud. **The application (in our case Microservice) has to follow and check to Twelve-factor rules before moving to cloud** i.e., in order to run Microservices seamlessly (means portably) across multiple cloud providers, it is important to follow Twelve-factor rules while developing cloud native microservices.

1. **Single code base**

Each microservice has its own code base. Code is typically managed in a source control system such as Git, Subversion, and so on

1. **Building Dependencies**

Each microservice should bundle all the required dependencies and execution libraries such as the HTTP listener and so on in the final executable bundle.

1. **Externalizing configurations**

This principle advises the externalization of all configuration parameters from the code. The microservices configuration parameters should be loaded from an external server.

1. **Backing services are addressable**

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



microservices either talk to a messaging system to send or receive messages, or they could accept or send messages to other service APIs. In a regular case, these are either HTTP endpoints using REST and JSON or TCP- or HTTP-based messaging endpoints.

1. **Isolation between build, release, and run**

This principle advocates a strong isolation between the build, release, and run stages. In microservices, the build will create executable JAR files, including the service runtime such as an HTTP listener. During the release phase, these executables will be combined with release configurations such as production URLs and so on and create a release version, most probably as a container similar to Docker. In the run stage, these containers will be deployed on production via a container scheduler.

1. **Stateless, shared nothing processes**

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

1. **Exposing services through port bindings**

A Twelve-Factor application is expected to be self-contained. Traditionally, applications are deployed to a server: a web server or an application server such as Apache Tomcat or JBoss. A Twelve-Factor application does not rely on an external web server. HTTP listeners such as Tomcat or Jetty have to be embedded in the service itself. Port binding is one of the fundamental requirements for microservices to be autonomous and self-contained. Microservices embed service listeners as a part of the service itself.

1. **Concurrency to scale out**

In the microservices world, services are designed to scale out rather than scale up. The x axis scaling technique is primarily used for a scaling service by spinning up another identical service instance. The services can be elastically scaled or shrunk based on the traffic flow. Further to this, microservices may make use of parallel processing and concurrency frameworks to further speed up or scale up the transaction processing.

1. **Disposability with minimal overhead**

This principle advocates building applications with minimal startup and shutdown times with graceful shutdown support. In the microservices context, in order to achieve full automation, it is extremely important to keep the size of the application as thin as possible, with minimal startup and shutdown time.

1. **Development and production parity**

This principle states the importance of keeping development and production environments as identical as possible.

In a development environment, we tend to run all of them on a single machine, whereas in production, we will facilitate independent machines to run each of these processes. This is primarily to manage the cost of infrastructure. The downside is that if production fails, there is no identical environment to re-produce and fix the issues. Not only is this principle valid for microservices, but it is also applicable to any application development.

1. Externalizing logs

A Twelve-Factor application never attempts to store or ship log files. In a cloud, it is better to avoid local I/Os. If the I/Os are not fast enough in a given infrastructure, it could create a bottleneck. The solution to this is to use a centralized logging framework. Splunk, Greylog, Logstash, Logplex, and Loggly are some examples of log shipping and analysis tools. In a microservices ecosystem, this is very important as we are breaking a system into a number of smaller services, which could result in decentralized logging. If they store logs in a local storage, it would be extremely difficult to correlate logs between services.

1. **Package admin processes**

Apart from application services, most applications provide admin tasks as well. This principle advises to use the same release bundle as well as an identical environment for both application services and admin tasks. Admin code should also be packaged along with the application code.

**Components of Spring Cloud**

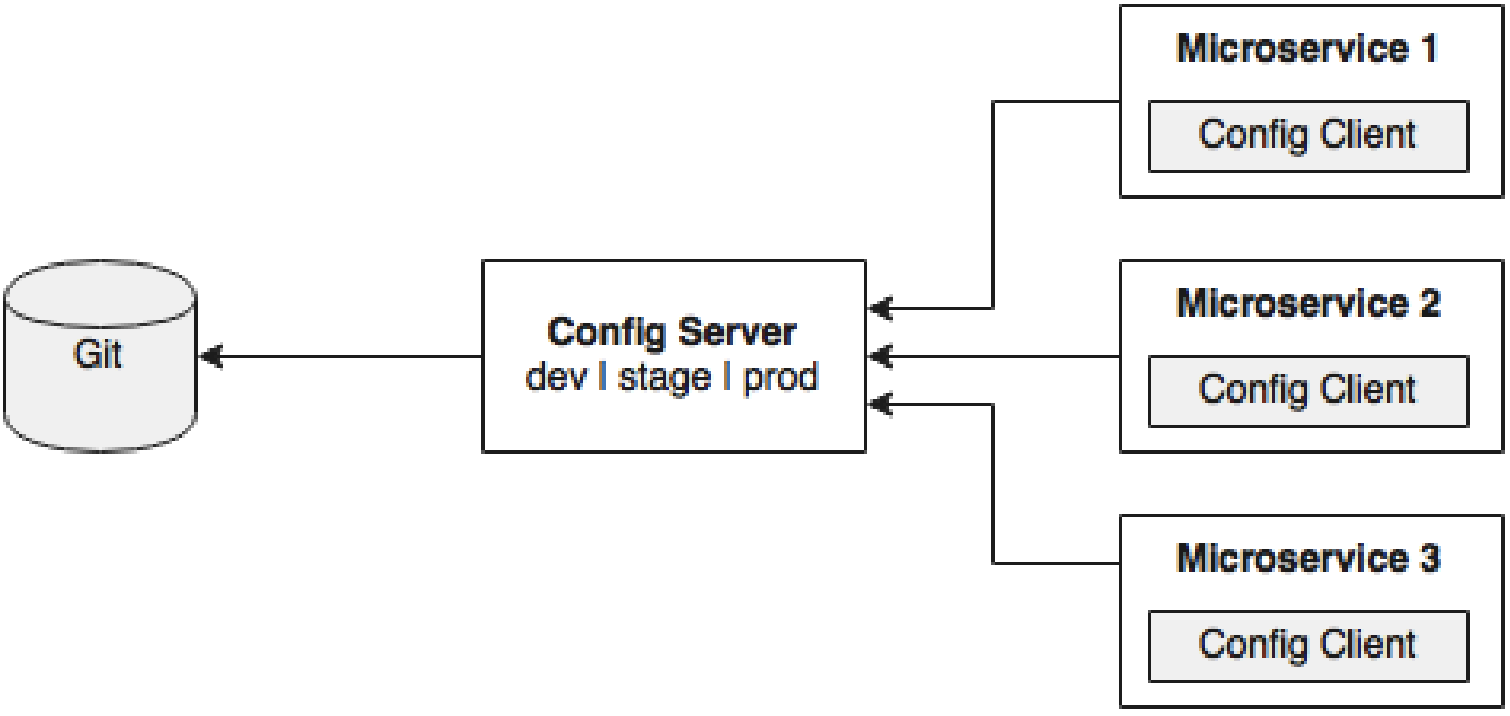
Each Spring Cloud component specifically addresses certain distributed system capabilities. The greyed-out boxes at the bottom of the following diagram show the capabilities, and the boxes placed on top of these capabilities showcase the Spring Cloud subprojects addressing these capabilities:



**Spring Cloud Config**

In Spring Boot, all configuration parameters were read from a property file packaged inside the project, either application.properties or application.yaml. This approach is good, since all properties are moved out of code to a property file. However, when microservices are moved from one environment to another, these properties need to undergo changes, which require an application re-build. This is violation of one of the Twelve-Factor application principles, which advocate one-time build and moving of the binaries across environments.

Hence it is always recommended to **externalize** and **centralize** microservice configuration parameters using **Spring Cloud Config.** The spring config server stores properties in a version-controller repository such as Git, or SVN. The spring cloud configure server architecture is shown in the following diagram.



As shown in the preceding diagram, the Config client embedded in the Spring Boot microservices does a configuration lookup from a central configuration server using a simple declarative mechanism, and stores properties into the Spring environment. The configuration properties can be application-level configurations such as trade limit per day, or infrastructure-related configurations such as server URLs, credentials, and so on.

Unlike Spring Boot, Spring Cloud uses a bootstrap context, which is a parent context of the main application. Bootstrap context is responsible for loading configuration properties from the Config server. The bootstrap context looks for bootstrap.yaml or bootstrap.properties for loading initial configuration properties. Hence rename the application.properties as bootstrap.properties

**Setting up the Config server**

The following steps need to be followed to setting up config server:

1. Download and install Git from <https://git-scm.com/downloads>

Create ‘config-repo’ folder in windows home[${user.home}]. Navigate to ${user.home}/config-repo

Note: Run below git commands only once.

C:\Users\Praveen\config-repo> git config –global user.email [abc@gmail.com](mailto:abc@gmail.com)

C:\Users\Praveen\config-repo> git config –global user.name ”Praveen”

C:\Users\Praveen\config-repo> git init

C:\Users\Praveen\config-repo> echo message = helloworld > sample.properties

Add and commit changes to git repository

git add -A .

git commit -m “Adding Sample.properties”

Note: This code snippet creates a new git repository on the local file system. A property file named sample.properties with a ‘message’ property and value ‘Hello World’ is also created.

1. Go to STS, create a new **Spring starter project** named ‘ConfigServer’, And select ConfigServer and Actuator starters
2. Expand config server project, go to ‘src/main/resource’ folder , rename application.properties file as bootstrap.properties . change the configuration in the config server to use the git repository created in the previous step. For this add the following properties on the bootstrap.properties file:

server.port=8888

spring.clod.config.server.git.uri:file://${user.home}/config-repo

management.security.enabled=false

note: port 8888 is the default port for the config server. Even with out configuring server.port, by default config server will bind to 8888.

1. Add @EnableConfigServer in Application.java file in ConfigServer project
2. Run the Config server by right-clicking on the project, and running it as a Spring Boot app.
3. Visit http://localhost:8888/actuator/env to see whether the server is running. If everything is fine, this will list all environment configurations. Note that actuator/env is an actuator endpoint.

Note: Ensure that management.security.enable=false property should be added in bootstrap.properties file to avoid (type=Unauthorized, status=401)

Note: use ‘https://jsoneditoronline.org/ for formatting json message.

1. Check http://localhost:8888/application/sample/master to see the properties specific to application.properties, which were added in the earlier step. The browser will display the properties configured in application.properties. The browser should display contents similar to the following: {"name":"application","profiles":["default"],"label":"master","ver sion":"6046fd2ff4fa09d3843767660d963866ffcc7d28","propertySources" :[{"name":"file:///Users/Praveen /config-repo /application.properti es","source":{"message":"helloworld"}}]}
2. The first element in the URL is the application name. In the given example, the application name shokleuld be application. The application name is a logical name given to the application, using the spring.application.name property in bootstrap.properties of the Spring Boot application. Each application must have a unique name. The Config server will use the name to resolve and pick up appropriate properties from the Config server repository. The application name is also sometimes referred to as service ID. If there is an application with the name myapp, then there should be a myapp.properties in the configuration repository to store all the properties related to that application. The second part of the URL represents the profile. The defaukt profile is named default. The last part of the URL is the lable and is named master by default. The label is an optional git label that can be used, if required

**Config Client**

In this section, the Search microservice will be modified to use the Config server. The Search microservice will act as a Config client.

1. Stop all the microservices and Add the Spring Cloud Config dependency and the actuator (if the actuator is not already in place) to the pom.xml file. The actuator is mandatory for refreshing the configuration properties

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

Since we are modifying the Spring Boot Search microservice from the earlier chapter, we will have to add the following to include the Spring Cloud dependencies.

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Brixton.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

1. Rename application.properties to bootstrap.properties in src/main/resources folder and add config server URL. Also comment out configuration properties.

The new bootstrap.properties file will look as follows:

spring.application.name=fares-service

spring.cloud.config.uri=http://localhost:8888

server.port=8081

#Move all below properties to fares-service.properties file in Git repo

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

spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl

spring.datasource.username=fareuser

spring.datasource.password=fareuser

spring.jpa.properties.hibernate.default\_schema=FAREUSER

#tomcat-connection settings

spring.datasource.tomcat.initialSize=20

spring.datasource.tomcat.max-active=25

spring.jpa.hibernate.ddl-auto=create

spring.jpa.show-sql=true

#Turn ON Spring Security to protect actuators endpoints

management.security.enabled=false

Note: fare-service is a logical name given to the Fare microservice. This will be treated as service ID. The Config server will look for fare-service. properties in the Git repository to resolve the properties. Hence fares-service.properties file should be created in ${user.home}/config-repo which is explained in next step.

1. Create a new fares-service.properties under the config-repo folder where the Git repository is created. Move service-specific properties from bootstrap.properties to the new fares-service. properties file.

**# C:/users/Praveen/config-repo/fares-service.properties**

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

spring.datasource.url=jdbc:oracle:thin:@localhost:1521:orcl

spring.datasource.username=fareuser

spring.datasource.password=fareuser

spring.jpa.properties.hibernate.default\_schema=FAREUSER

#tomcat-connection settings

spring.datasource.tomcat.initialSize=20

spring.datasource.tomcat.max-active=25

spring.jpa.hibernate.ddl-auto=create

spring.jpa.show-sql=true

#Turn ON Spring Security to protect actuators endpoints

management.security.enabled=false

Add and commit all changes in the Git repository.

git add -A .

git commit -m ‘Adding fares-service.proprties’

1. Type config server URL “ <http://localhost:8888/fares-service/default/master>” to see the properties specific to fares-service.properties, which are added in the earlier steps.
2. Repeat all above steps in search, booking, check-in microservices and web site.
3. **Start**  all the services and web site. Also perform booking and check-in through web site using <http://localhost:8001>

**User name: guest**

**Password**: guest123

1. In order to demonstrate the centralised configuration of properties and propagation of changes, add a new application-specific property in the search-service.properties file. We will add **originairports.shutdown** property to temporarily take out an airport from the search. Users will not get any flight info when searching for an airport mentioned in the shut down list. Add property in search-service.properties file in git. #C:/users/Praveen/config-repo/search-service.propeties

originairports.shutdown=SEA

Add and commit changes to Git repository

git add -A .

git commit -m ‘Adding origin airports shutdown properties’

1. Modify the Search microservice code to use the configured parameter, originairports.shutdown. A **@RefreshScope** annotation has to be added at the class level to allow properties to be refreshed when there is a change. In this case, we are adding a refresh scope to the SearchRestController class:

// A @RefreshScope annotation has to be added at the class level to allow properties to be refreshed when there is a change in search-service.properties file if and only if

/refresh actuator endpoint is executed.

@RefreshScope

public class SearchRestController{ }

Also, add the flowing instance variable as a place holder for the new property that was just added in the config server. The property name in the search-service.properties file must match.

@RefeshScope

public class SearchRestController{

@Value(“${originairports.shutdown}”)

private String originAirportShutdownList;

…

}

Change the application code to use this property. This is done by modifying the search method as follows:

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.cloud.context.config.annotation.RefreshScope;

@RefreshScope

class SearchRestController {

private static final Logger logger = LoggerFactory.getLogger(SearchRestController.class);

    @Value("${originairports.shutdown}")

    private String originAirportShutdownList;

    private SearchComponent searchComponent;

...

@RequestMapping(value = "/get", method = RequestMethod.POST)

    public List<Flight> search(@RequestBody SearchQuery query){

         logger.info("Input : " + query);

if(Arrays.asList(originAirportShutdownList.split(",")) .contains(query.getOrigin())) {

            logger.info("The origin airport is in shutdown state");

            return new ArrayList<Flight>();

        }

        return searchComponent.search(query);

    }

}

The search method is modified to read the parameter originAirportShutdownList and see whether the requested origin is in the shutdown list. If there is a match, then instead of proceeding with the actual search, rather the search method will return an empty flight list.

1. Start search microservice.
2. Go to web site (<http://localhost:8001>) and search travelling from SEA. And going to SFO.

**Observation**: 1. An empty flights list should be displayed in web page.

2. ‘ The origin airport is in shutdown state’ should be printed on search console.

**Handling Configuration Changes**

This section will demonstrate how to propagate configuration properties when there is a change:

1. Change the property in the search-service.properties file to the following: originairports.shutdown:NYC

Add and commit changes to Git repository.

git add -A .

git -m commit ‘Modifying origin airports shutdown value’

Optionally refresh the Config server URL (http://localhost:8888/search-service/default/master) for this service and see whether the property change is reflected.

1. Check in website now without restarting search microservice. We can observe that the change is not reflected in the search service, and the service is working with an old copy of the configuration properties.
2. In order to force reloading of the configuration properties, call the /refresh endpoint of the Search microservice. This is actually the actuator's endpoint. The following command will send an empty POST JSON to the /refresh endpoint:

curl –d {} localhost:8090/refresh

Note: if we install Git for windows, we get automatically too. The installation comes with GNU bash( git-bash.exe), a really powerful shell.

Double click on ‘C:/Program Files/git/git-bash.exe’

**$curl -d{} localhost:8090/refresh**

1. Now check in website now without restarting search microservice. We can observe that the change is not reflected in the search service, and the service is working with an new copy of the configuration properties.

Observation: with this approach, configuration parameters can be propagated without restarting the microservices.

**Spring Cloud Bus**

The above approach was good in case of few Instances. In case of many instances, hitting refresh for every instance is not good.

The **Spring Cloud Bus** provides a mechanism to refresh configuration across multiple micro services without knowing how many microservices are there.

The following steps are used to configure cloud bus:

1. Add fares. Discount property in fares-service.properties file in Git repo

**fares.discount=5**

Add and commit changes to Git repository.

git add -A .

git commit -m ‘Adding fares.discount property’

Refresh the config server URL “ <http://localhost:8888/fares-service/default/master>” for this service and see whether the property change is reflected.

1. Stop both fare and search microservices and add below starter dependency:

<dependency>

<groupId>org.springframework.cloud</groupId>

**<artifactId>spring-cloud-starter-bus-amqp</artifactId>**

</dependency>

1. Modify the Fares microservice code to use the configured parameter, fares.discount. A **@RefreshScope** annotation has to be added at the class level to allow properties to be refreshed when there is a change. In this case, we are adding a refresh scope to the FaresRestController class:

@RefreshScope

public class FaresController{ }

1. Change the application code to use this property.

@RefreshScope

public class FaresController {

     private static final Logger logger = LoggerFactory.getLogger(FaresController.class);

 @Value("${fares.discount}")

    private String faresDiscount;

    FaresComponent faresComponent;

    @RequestMapping("/get")

    Fare getFare(@RequestParam(value="flightNumber") String flightNumber, @RequestParam(value="flightDate") String flightDate){

        logger.info("Fares Discount = " + faresDiscount);

        return faresComponent.getFare(flightNumber,flightDate);

    }

}

1. Start both fare and search microservices.

Note: Boot version and Spring cloud version should be compatible.

Ex: Boot version 1.5.9 and Edgware.SR3 are compatible versions.

Ex: Boot version 1.5.3 and Dalston.SR2 are compatible versions.

1. Now, change origin airports shutdown value in search-service.properties and change discount value in fares-services.properties.

#fares-services.properties

fares.discount=15

#search-service.properties

Orginairports.shutdown=SEA

Add and commit the changes in Git repository.

git add -A .

git commit -m ‘Changing origin airport shutdown value and discount values’

1. Run the following command with **actuator/bus-refresh**. Note that we are running a new bus endpoint against one of the instances, 8090 in this case:

$ curl -d {} <http://localhost:8090/bus/refresh>

1. Make one more booking and observe search screen and fares console.

**Feign as a declarative REST client**

In the booking micro service, there is a REST call to Fair. RestTemplate is used for making the REST call. When using RestTemplate, the URL parameter should be constructed programmatically. In more complex scenarios, we will have to get to the details of the HTTP APIs provided by RestTemplate or even to APIs at a much lower level.

Ex:

@Service

public class BookingComponent {

    private static final Logger logger = LoggerFactory.getLogger(BookingComponent.class);

    private static final String FareURL = "http://localhost:8081/fares";

    @Autowired

    private RestTemplate restTemplate;

    //The below method is invoked while booking flight ticket

    public long book(BookingRecord record) {

        logger.info("calling fares to get fare");

        Fare fare = null;

        try{

        // Make a rest call with Fare micro service to get flight price details

        fare = restTemplate.getForObject(FareURL + "/get?flightNumber=" + record.getFlightNumber() + "&flightDate=" + record.getFlightDate(), Fare.class);

        logger.info("calling fares to get fare " + fare);

        }catch(Exception e){

            logger.error("FARE SERVICE IS NOT AVAILABLE");

        }

}

}

**Feign** is a Spring Cloud Netflix library for providing a higher level of abstraction over REST-based service calls. Spring Cloud Feign works on a declarative principle. When using Feign, we write declarative REST service interfaces at the client, and use those interfaces to program the client. The developer need not worry about the implementation of this interface. This will be dynamically provisioned by Spring at runtime. With this declarative approach, developers need not get into the details of the HTTP level APIs provided by RestTemplate.

**The following steps are required to use Feign**

1. Stop both Booking and CheckIn microservices. In order to use feign, add below dependency to the pom.xml file in Booking Microservice:

<dependency>

<groupId>org.springframework.cloud</groupId>

**<artifactId>spring-cloud-starter-openfeign</artifactId>**

</dependency>

1. Create a new FareServiceProxy interface. This will act as a proxy interface of the actual fare service.

package com.brownfield.pss.book.component;

//Feign makes writing web service (REST)clients easier

//This annotation tells Spring to create a REST client based on the interface provided.

//The "fares-proxy" is an arbitrary client name, which is used by Ribbon load balancer

@FeignClient(name = "fares-proxy", url = "http://localhost:8081/fares")

public interface FareServiceProxy {

    @RequestMapping(value = "/get", method = RequestMethod.GET)

    Fare getFare(@RequestParam(value = "flightNumber") String flightNumber,

    @RequestParam(value = "flightDate") String flightDate);

}

1. Use this FairProxy to call the Fare microservice. In the Booking microservice, we have to tell Spring that Feign clients exist in the Spring Boot application, which are to be scanned and discovered. This will be done by adding @EnableFeignClients at the class level of BookingComponent. Optionally, we can also give the package names to scan.

package com.brownfield.pss.book.component;

@EnableFeignClients

public class BookingComponent{ }

1. In BookingComponent, make changes to the calling part. This is as simple as calling other java interface.

@EnableFeignClients

public class BookingComponent {

    private static final Logger logger = LoggerFactory.getLogger(BookingComponent.class);

    //private static final String FareURL = "http://localhost:8081/fares";

    // @Autowired

    //private RestTemplate restTemplate;

    @Autowired

    FareServiceProxy fareServiceProxy;

    //The below method is invoked while booking flight ticket

    public long book(BookingRecord record) {

        logger.info("calling fares to get fare");

        Fare fare = null;

        try{

        // Make a rest call with Fare micro service to get flight price details

        //fare = restTemplate.getForObject(FareURL + "/get?flightNumber=" + record.getFlightNumber() + "&flightDate=" + record.getFlightDate(), Fare.class);

        fare = fareServiceProxy.getFare(record.getFlightNumber(), record.getFlightDate());

        logger.info("calling fares to get fare " + fare);

        }catch(Exception e){

            logger.error("FARE SERVICE IS NOT AVAILABLE");

        }

}

}

1. Start both Booking and CheckIn services.

Observation: ‘Looking for fares flightNumber BF101 flight date 22-JAN-16’ in fare microservice console is printed.

**Load Balancing Using Ribbon**

So far we were always running with a single instance of the microservice. The URL is hardcoded both in client as well as in the service-to-service calls. In the real world, this is not a recommended approach, since there could be more than one service instance. If there are multiple instances, then ideally, we should use a load balancer or a local DNS server to abstract the actual instance locations, and configure an alias name or the load balancer address in the clients. The load balancer then receives the alias name, and resolves it with one of the available instances. With this approach, we can configure as many instances behind a load balancer.

This is achievable with Spring Cloud Netflix Ribbon. Ribbon is a client-side load balancer which can do round-robin load balancing across a set of servers.

As shown in the preceding diagram, the Ribbon client looks for the Config server to get the list of available microservice instances, and, by default, applies a round-robin load balancing algorithm.

**The following steps are used to configure load balancing:**

1. Copy fare microservice project and set name as FaresFlightTickets2. Set port number as 8082 and start this project. The two instances of the fare services are running now, one on port 8081 and another one on 8082.
2. In order to use the Ribbon, stop both Booking and CheckIn Microservices, and add the following dependency

to the pom.xml file in BookingMicroservice:

<dependency>

<groupId>org.springframework.cloud</groupId>

**<artifactId>spring-cloud-starter-ribbon</artifactId>**

</dependency>

1. Modify FareServiceProxy interface in Booking Microservice to use Ribbon client.

//Feign makes writing web service (REST)clients easier

//This annotation tells Spring to create a REST client based on the interface provided.

//The "fares-proxy" is an arbitrary client name, which is used by Ribbon load balancer

//@FeignClient(name = "fares-proxy", url = "http://localhost:8081/fares")

@FeignClient(name = "fares-proxy")

@RibbonClient

public interface FareServiceProxy {

    @RequestMapping(value = "/fares/get", method = RequestMethod.GET)

    Fare getFare(@RequestParam(value = "flightNumber") String flightNumber,

     @RequestParam(value = "flightDate") String flightDate);

}

1. Update the Booking microservice configuration file, **booking-service.properties**, to include a new property to keep the list of the Fare micro-services:

#booking-service.properties  
**fares-proxy.ribbon.listOfServers=localhost:8081, localhost:8082**

Add and commit the chages in the Git reposirory.

git add -A .

git commit -m ‘adding new configuration’

1. Start Booking and Checkin microservices

Observation: The following text will be printed on Booking microservice console:

**DynamicServerListLoadBalancer**:{NFLoadBalancer:name=**fares-proxy**, current list of Servers=[**localhost:8081,localhost:8082**], Load balancer stats=Zone stats:

{unknown=[Zone:unknown; Instance count:2; Active connections count:0;Circuit breaker tripped count:0;Active connection per server:0.0;]

1. When we book minimum two tickets through website, the request will go to two fare instances by using round robin algorithm.

Observation:

The following text will be printed on fare service1 console:

com.brownfields.pss.fares.component.FaresComponent: Looking for fares flightNumber BF101 flightDate 22-jan-16

The following text will be printed on fare service2 console:

com.brownfield.pss.fares.component.FaresComponent: Looking for fares flightNumber BF101 flightDate 22-jan-16

**Eureka for registration and discovery**

So far, we have achieved externalizing configuration parameters as well as load balancing across many service instances.

Ribbon-based load balancing is sufficient for most of the microservices requirements. However, this approach falls short in a couple of scenarios:

• If there is a large number of microservices, and if we want to optimize infrastructure utilization, we will have to dynamically change the number of service instances and the associated servers. It is not easy to predict and preconfigure the server URLs in a configuration file.

• When targeting cloud deployments for highly scalable microservices, static registration and discovery is not a good solution considering the elastic nature of the cloud environment.

• In the cloud deployment scenarios, IP addresses are not predictable, and will be difficult to statically configure in a file. We will have to update the configuration file every time there is a change in address.

The Ribbon approach partially addresses this issue. With Ribbon, we can dynamically change the service instances, but whenever we add new service instances or shut down instances, we will have to manually update the Config server. Though the configuration changes will be automatically propagated to all required instances, the manual configuration changes will not work with large scale deployments. When managing large deployments, automation, wherever possible, is paramount.

To fix this gap, the microservices should self-manage their life cycle by dynamically registering service availability, and provision automated discovery for consumers.

With Dynamic registration, when a new service is started, it automatically enlists its availability in a central service registry.

Dynamic discovery is where clients look for the service registry to get the current state of the services topology, and then invoke the services accordingly. In this approach, instead of statically configuring the service URLs, the URLs are picked up from the service registry.

There are a number of options available for dynamic service registration and discovery. Netflix Eureka, ZooKeeper, and Consul are available as part of Spring Cloud, In this chapter, we will focus on the Eureka implementation.

Eureka is primarily used for self-registration, dynamic discovery, and load balancing. Eureka uses Ribbon for load balancing internally.

As shown in the preceding diagram, Eureka consists of a server component and a client-side component. The server component is the registry in which all microservices register their availability. The registration typically includes service identity and its URLs. The microservices use the Eureka client for registering their availability. The consuming components will also use the Eureka client for discovering the service instances.

By default ,the Eureka server itself is another Eureka client. This is particularly useful when there are multiple Eureka servers running for high availability

**Eureka Server**

The following steps required for setting up Eureka server:

1. Create a new Spring starter project named ‘EurekaServer’, and select **Eureka Servers** , Config client and Actuator.
2. Rename application.properties to bootstrap.properties since this is using the Config server.

spring.application.name=eureka-server

server.port:8761

spring.cloud.config.uri=http://localhost:8888

1. Create Eureka-server.properties in Git repo.

spring.application.name=eureka-server

#back to the same standalone instance

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

eureka.client.registerWithEureka:false

eureka.client.fetchRegistry:false

Add and commit changes to the Git repository

git add -A .

git commit -m ‘Adding new Configuration’

1. Add **@EnableEurekaServer** in the spring bootstrap file (EurekaServerApplication.java)
2. Start Eureka Server.

Ensure that config server was already started before starting Eureka server. Once the server is started, type http://localhost:8761 in a browser to see the Eureka console.

**Observation**: No instances available

1. Stop all micro services and add the following additional dependency in all micro services in their pom.xml file to enable dynamic registration and discovery using the Eureka client.

<dependency>

<groupId>org.springframework.cloud</groupId>

**<artifactId>spring-cloud-starter-eureka</artifactId>**

</dependency>

Note: The Config Client, Actuator, Web dependencies should be in pom.xml file.

1. Add @EnableDiscoveryClient in all microservices in their respective Spring Boot main classes. This asks Spring Boot to register these services at start-up to advertise their availability.
2. The below following property has to be added to all microservices in their

Respective configuration files under config-repo. This will help themicroservices to connect to the Eureka server.

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

Add and commit all changes to Git repo

git add -A .

git commit -m ‘Adding new Configuration’

1. Start Fare and Search Microservices.

Observation:

On fare microservice console:

DiscoveryClient\_FARES-SERVICE/Praveen-PC:fares-service:8081 -registration status:204.

Eureka server console:

Registered instance FARES-SERVICE/Praveen:fares-swrvice:8081 with status up

Web: refresh <http://localhost:8761>

Instances currently registered with Eureka

FARES-SERVICE n/a (1) (1) UP (1) -Praveen-PC:fares-service:8081

1. Eureka internally uses Ribbon for load balancing hence remove ribbon dependency from booking microservices pom.xml file. Ensure that Eureka, config, actuator and web starters should be in pom.xml file.

<dependency>

<groupId>org.springframework.cloud</groupId>

**<artifactId>spring-cloud-starter-ribbon</artifactId>**

</dependency>

1. Also remove the @RibbonClient annotation from the FareServiceProxy interface.
2. Update @FeignClient(name="fares-service") to match the actual Fare microservices' service ID.
3. Also remove the list of servers from the booking-service.properties file. Ensure that eureka.client.service.URL.defaultZone property should be added in booking-service.properties file.

# fares-proxy.ribbon.listOfServers=localhost:8081, localhost:8082

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

Add and commit all changes to Git repo

git add -A .

git commit -m ‘Adding new Configuration’

1. Set Service-ID of Booking microservice in CheckinComponent.java file:

@Configuration class AppConfig {

@LoadBalanced

@Bean

RestTemplate restTemplate() {

return new RestTemplate();

}

}

private static final String bookingURL= <http://booking-service/booking>;

1. Start Booking and Check-in microservices.

Observation:

On Booking Console:

**DynamicServerListLoadBalancer**:{NFLoadBalancer:name=**fares-proxy**, current list of Servers=[**Praveen:8081,Praveen:8082**], Load balancer stats=Zone stats: …]

1. Add Eureka client to web site project. Also make below changes in BrownFieldSiteController.java and Application.java file.

@Configuration class AppConfig {

@LoadBalanced

@Bean

RestTemplate restTemplate() {

return new RestTemplate();

}

}

     Flight[] flights = searchClient.postForObject("http://localhost:8090/search/get", searchQuery, Flight[].class);

        bookingId = bookingClient.postForObject("http://localhost:8060/booking/create", booking, long.class);

       BookingRecord booking = bookingClient.getForObject("http://localhost:8060/booking/get/"+id, BookingRecord.class);

long checkinId = checkInClient.postForObject("http://localhost:8070/checkin/create", checkIn, long.class);