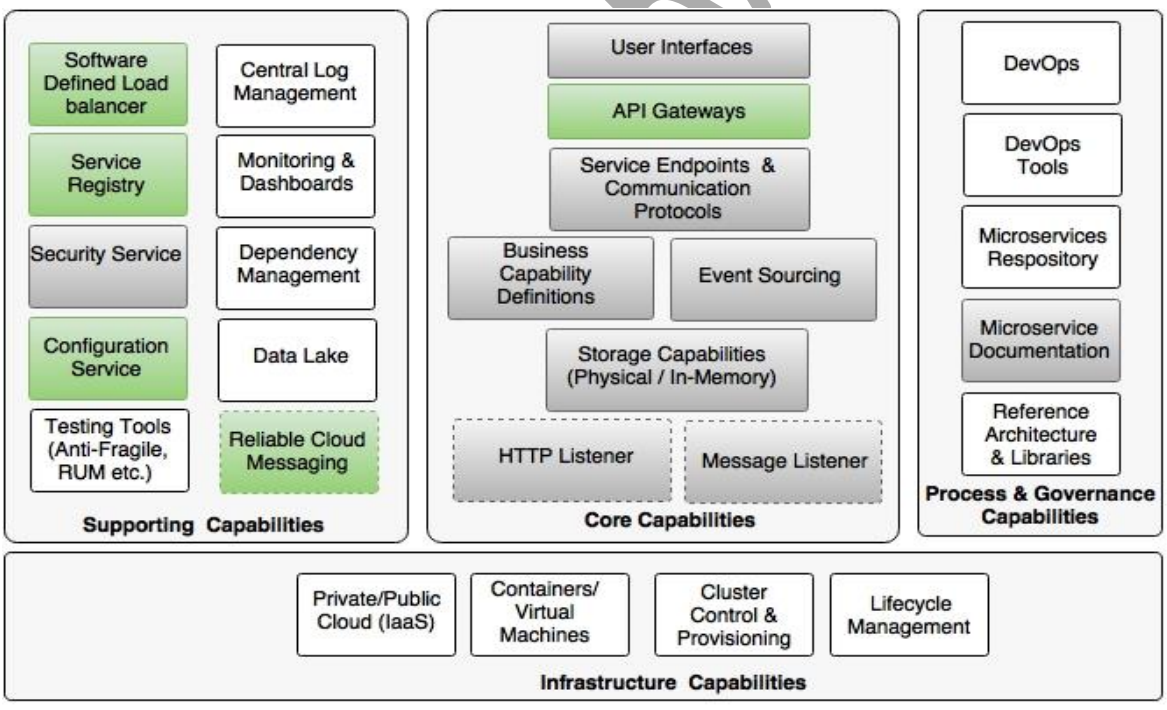
SPRING CLOUD

Spring Cloud provides capabilities such as **centralized configuration**, **load balancing**, **service registry**, monitoring, service gateway, and so on.

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), Google (Google cloud), Pivotal (Cloud Foundry), IBM (Blue Mix), Redhat (Open Shift), Rackspace (Rackspace), 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 spring cloud related capabilities such as Load balancer, Service Registry, Configuration service, Cloud messaging, API Gateways, etc are highlighted in green color.



Spring Cloud by itself is not a cloud solution. 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 portable across many cloud providers such as **Cloud Foundry**, **AWS**, and so on.

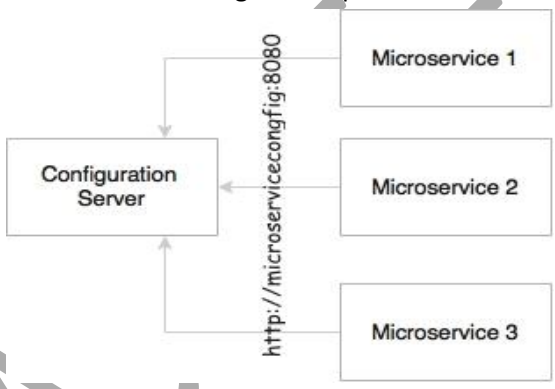
Twelve-Factor apps

Twelve-Factor Apps define 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 Twelve-factor rules before moving to cloud** i.e., in order to run microservices seamlessly across multiple cloud providers, it is important to follow Twelve-factor rules while developing cloud native microservices.

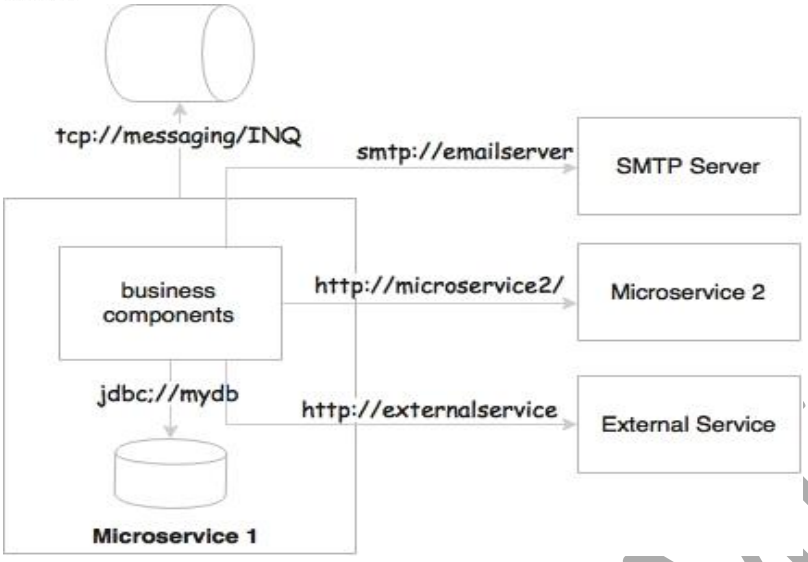
1) **Single code base**
   Each microservice has its own code base. Code is typically managed in a source control system such as Git, Subversion, etc.

2) **Bundling 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.

3) **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.



4) **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.



Microservices either talk to a messaging system to send or receive messages OR they could accept or send messages to other service APIs using REST/JSON over HTTP.

5) **Isolation between build, release, and run**

This principle advocates a strong isolation among the build, release, and run stages.

In microservices, the **build** will create executable fat 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.

6) **Stateless, shared nothing processes**

This principle suggests that processes should be stateless and share nothing. If the application is stateless, then it 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.

7) **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.

8) **Concurrency to scale out**

In the microservices, services are designed to scale out. 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.

9) **Disposability with minimal overhead**

This principle advocates building applications with minimal startup and shutdown times with graceful shutdown support.

In the microservices, 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.

10) **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.

11) **Externalizing logs**

A Twelve-Factor applicaion 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.

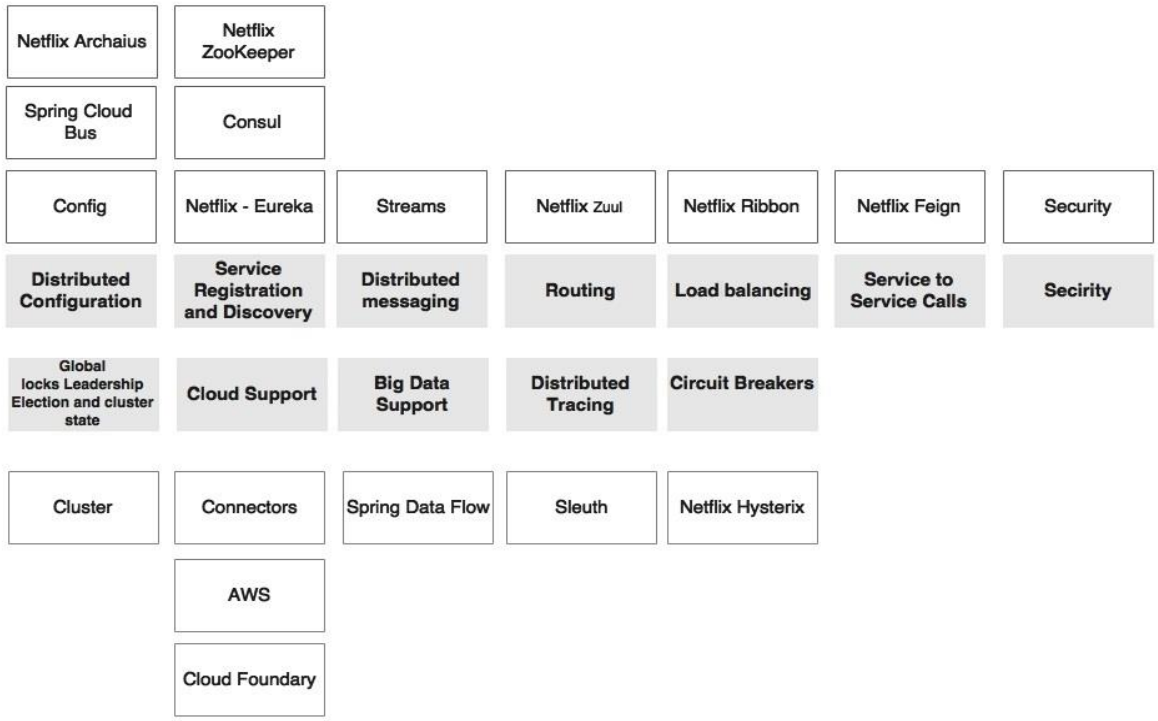
In a microservices ecosystem, this is very important as we are breaking a system into a number of smaller services, which could result in decentralized logging. If they store logs in a local storage, it would be extremely difficult to correlate logs between services.

12) **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 grayed-out boxes show the capabilities, and the boxes placed on top of these capabilities showcase the Spring Cloud subprojects addressing these capabilities:



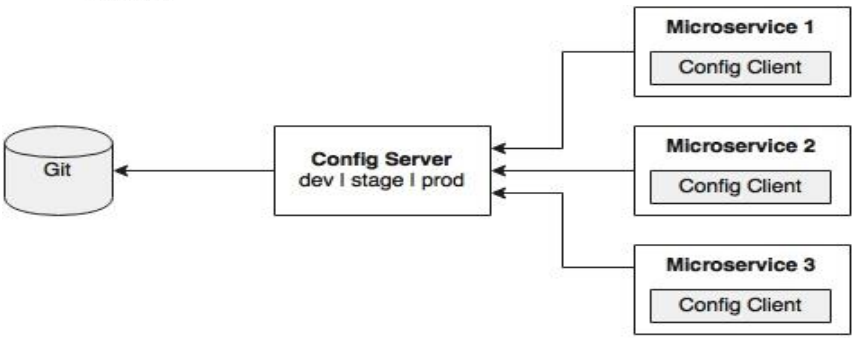
Spring Cloud Config

In Spring Boot applications, all configuration parameters were read from a property file packaged inside the project, either in application.properties or application.yml. 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 server**.

The Spring Config server stores properties in a version-controlled repository such as Git or SVN.

The Spring Cloud Config 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. The configuration properties may be application related (such as trade limit per day) or infrastructure related properties (urls, credentials, etc).

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.properties** or bootstrap.yml for loading initial configuration properties. Hence rename application.properties as bootstrap.properties.

Setting up the Config server

The following steps are used to setting up config server:

1) Create a new **Spring Starter Project** named **'ConfigServer'**, and select **Config Server** and **Actuator**.

2) Download and install **Git**. The 'D:\Program Files\Git\cmd' automatically added to path.
Create 'config-repo' folder in windows home [${user.home}]. Navigate to ${user.home}/config-repo
Note: Run below two commands once
C:\Users\Aspire-Ramesh\config-repo>git config --global user.email ramesh@java2aspire.com
C:\Users\Aspire-Ramesh\config-repo>git config --global user.name "ramesh"

C:\Users\Aspire-Ramesh\config-repo>git init .
C:\Users\Aspire-Ramesh\config-repo>echo message : helloworld > sample.properties

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 filesystem. A property file named sample.properties with a message property and value helloworld is also created.

3) Goto STS, expand ConfigServer project, goto src/main/resources folder, rename application.properties file to **bootstrap.properties**. Change the configuration in the config server to use the Git repository created in the previous step. For this, add following properties in bootstrap.properties file:
server.port=8888
spring.cloud.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 without configuring server.port, the Config server will bind to 8888.

4) Add @**EnableConfigServer** in Application.java file in ConfigServer project.

5) Run Config Server by right-click on the project and Run as **Spring Boot App**.

6) Visit **`http://localhost:8888/env`** to see whether the server is running. If everything is fine, this will list all environment configurations. **Note that `/env` is an actuator endpoint**.
Note: Ensure that **management.security.enabled=false** property should be added in bootstrap.properties file to avoid (type=Unauthorized, status=401)
Note: Use 'http://www.jsoneditoronline.org/ for formatting JSON message.

7) Also check Config server URL "http://localhost:8888/sample/default/master" to see the properties specific to `sample.properties`, which were added in the earlier step.
The first element in the URL is the application name. It is a logical name given to the application, using the **`spring.application.name`** property in `bootstrap.properties` of the Spring Boot application. 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 **`search-service`**, then there should be a **`search-service.properties`** in the configuration repository to store all the properties related to that application.
The second part of the URL represents the profile. The default profile is named `default`.
The last part of the URL is the label, and is named master by default. The label is an optional Git label that can be used, if required.

Accessing the Config Server from clients

In this section, all microservices will be modified to use the Config server. All our microservices will act as Config clients.

1) Stop all microservices and add the **Spring Cloud Config dependency** and the actuator (if the actuator is not already in place) to the `pom.xml` file.

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

Since we are modifying the Spring Boot fares microservice from the earlier project, 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>Dalston.SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

2) Rename application.properties to **bootstrap.properties** in src/main/resources folder and add an application name and a configuration server URL. Also comment out configuration properties.

```
#src/main/resources/bootstrap.properties
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:xe
#spring.datasource.username=fareuser
#spring.datasource.password=aspire123
#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

#management.security.enabled=false

Note: The 'fares-service' is a logical name given to the fares microservice. This will be treated as service ID. The Config server will look for **`fares-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.

3) 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/Aspire-Ramesh/config-repo/fares-services.properties
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=fareuser
spring.datasource.password=aspire123
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

management.security.enabled=false

Commit all changes in the Git repository.
git add -A .
git commit -m "Adding fares-service.properties"

4) Type Config server URL "http://localhost:8888/fares-service/default/master" to see the properties specific to `fares-service.properties`, which were added in the earlier step.

5) Repeat all above steps in search, booking and check-in microservices.

6) **Start** all services and website. Also perform booking and check-in through website using http://localhost:8001
Username: guest
Password: guest123

7) In order to demonstrate the centralized configuration of properties and propagation of changes, add a new application-specific property to the search-service.properties file. We will add **originairports.shutdown** to temporarily take out an airport from the search. Users will not get any flights info when searching for an airport mentioned in the shutdown list.
Add below property in search-service.properties file in Git.
 # C:/Users/Ramesh-PC/config-repo/search-services.properties
originairports.shutdown=SEA

Commit the changes in the Git repository.
git add –A .
git commit –m "adding origin airports shutdown property"

8) Stop search microservice. 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 when /refresh actuator is used.
@RefreshScope
Public class SearchRestController {

}
```

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

```
@RefreshScope
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
public class SearchRestController {
        private static final Logger logger = LoggerFactory.getLogger(SearchRestController.class);

        @Value("${originairports.shutdown}")
        private String originAirportShutdownList;
        …

        @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.

9) Start search microservice.
10) Goto website (http://localhost:8001) and search travelling from SEA and going to SFO.
Observation: 1) 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 changes automatically when there is a change:

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

Commit the changes in the Git repository.
git add –A .
git commit –m "modifying origin airports shutdown value"

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

2) Check in website project now without restarting search microservice. We can observe that the change is not reflected in the search service, and the service is still working with an old copy of the configuration properties.

3) In order to **force reloading of the configuration properties**, call the **/refresh** endpoint of the Search microservice. This is actually the actuator's refresh endpoint. The following command will send an empty POST to the /refresh endpoint:
curl –d {} localhost:8090/refresh

Note: If we install Git for Windows we get Curl automatically too. The installation comes with GNU bash (**git-bash.exe**), a really powerful shell.
Double click on "D:\Program Files\Git\git-bash.exe"
$curl –d {} localhost:8090/refresh

4) Now check in website without restarting search microservice. We can observe that the change is reflected in the search service, and the service is refreshing with new copy of the configuration properties.
Observation: With this approach, configuration parameters can be changed without restarting the microservices.

Spring Cloud Bus

The above approach is 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 configurations across multiple instances without knowing how many instances there are, or their locations.
The following steps are used to configure cloud bus:

1) Stop search microservice and add below dependency:

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

2) Copy search microservice project. Set port number as 8091 and **start both instances**. The two instances of the Search service are running now, one on port 8090 and another one on 8091.

3) Now, change origin airports shutdown value in search-service.properties.
originairports.shutdown:SEA

Commit the changes in the Git repository.
git add –A .
git commit –m "changing origin airports shutdown value"

4) Run the following command with **/bus/refresh**. Note that we are running a new bus endpoint against one of the instances, 8090 in this case:
$ curl –d {} localhost:8090/bus/refresh

5) Immediately, we will see the following message for both instances on search console:
Observation: Received remote refresh request. Keys refreshed [originairports.shutdown]

Feign as a declarative REST client

In the booking microservice, there is a synchronous call to Fare. RestTemplate is used for making the synchronous call. When using RestTemplate, **the URL parameter is 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.

Example:

```
@Component
public class BookingComponent {
        private static final String FareURL = "http://localhost:8081/fares";
        public long book(BookingRecord record) {
                //call fares to get fare
                Fare fare = restTemplate.getForObject(FareURL +"/get?flightNumber=" +
                                record.getFlightNumber()+"&flightDate="+record.getFlightDate(),Fare.class);
}
```

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 to 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-feign</artifactId>
</dependency>
```

2) 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 = "localhost:8081/fares")
public interface FareServiceProxy {
        @RequestMapping(value = "/get", method = RequestMethod.GET)
        Fare getFare(@RequestParam(value = "flightNumber") String flightNumber,
                                @RequestParam(value = "flightDate") String flightDate);
}
```

3) Use this fare proxy to call 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 {

}
```

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

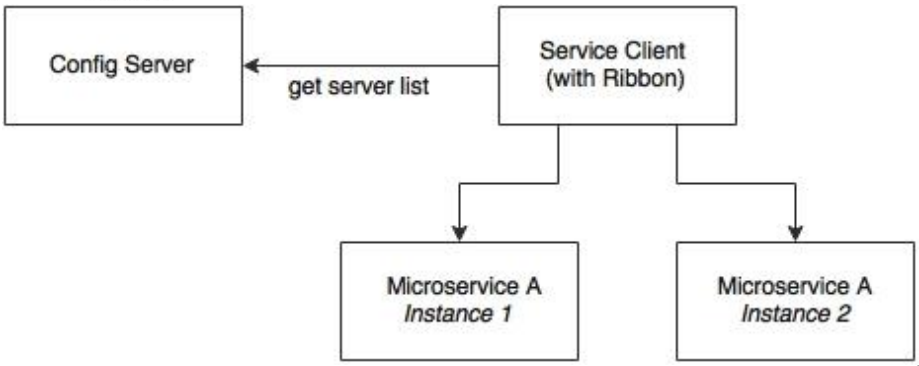
```
@EnableFeignClients
public class BookingComponent {
        //private static final String FareURL = "http://localhost:8081/fares"; //Not required if we use Feign
```

```
//@Autowired
//private RestTemplate restTemplate; //Not required if we use Feign

@Autowired
FareServiceProxy fareServiceProxy;

public long book(BookingRecord record) {
Fare fare = null;
try{
 // call fares to get fare
 //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");
}
}
}
```

5) Start **both** booking and checkin services.
   Observation: 'Looking for fares flightNumber BF101 flightDate 22-JAN-16' in fare microservice console is printed.

Load Balancing Using Ribbon

So far we were 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 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) In order to use Ribbon, stop both booking and checkin microservices and add the following dependency to the `pom.xml` file in booking microservice:

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

2) Copy fare microservice project named as FareFlightTickets2. Set port number as 8082 and start this project. The two instances of the fare service are running now, one on port 8081 and another one on 8082.

3) Modify FareServiceProxy class in booking microservice to use Ribbon client.

```
//The "fares-proxy" is an arbitrary client name, which is used by Ribbon load balancer
//@FeignClient(name = "fares-proxy", url = "localhost:8081/fares") //without ribbon
@FeignClient(name = "fares-proxy")
@RibbonClient
public interface FareServiceProxy {
        //@RequestMapping(value = "/get", method = RequestMethod.GET) //without ribbon
        @RequestMapping(value = "/fares/get", method = RequestMethod.GET)
        Fare getFare(@RequestParam(value = "flightNumber") String flightNumber,
@RequestParam(value = "flightDate") String flightDate);
}
```

4) Update the Booking microservice configuration file, **booking-service.properties**, to include a new property to keep the list of the Fare microservices:

```
#booking-service.properties
```

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

Commit the changes in the Git repository.
git add –A .
git commit –m "adding new configuration"

5) 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 connections per server: 0.0;]

When booking microservice is bootstrapped, the CommandLineRunner automatically inserts one booking record. This will go to the first server.
Observation: The following text will be printed on fare service1 console:
c.b.pss.fares.component.FaresComponent: Looking for fares flightNumber BF101 flightDate 22-JAN-16

6) When we book ticket through website project, it calls the booking service. This request will go to the second server.
Observation: The following text will be printed on fare service2 console:
c.b.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:

a) 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.
b) 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.
c) 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.

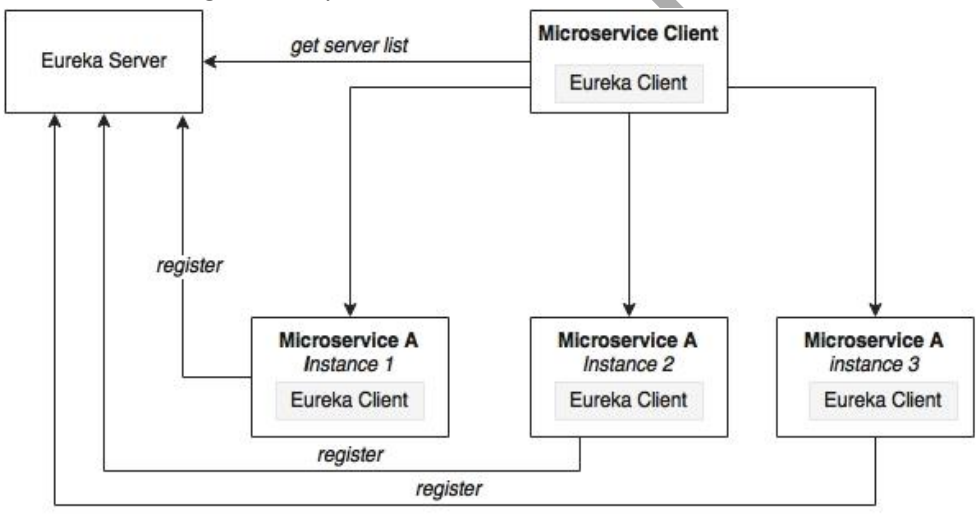
To fix this gap, the microservices should self-manage their life cycle by **dynamically registering service availability**, and **provision automated discovery for consumers**.

With dynamic registration, when a new service is started, it automatically enlists its availability in a central service registry. Similarly, when a service goes out of service, it is automatically delisted from the 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 in config server rather 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.

Setting up the Eureka server

The following steps required for setting up the Eureka server:

1) Create a new Spring Starter project named 'EurekaServer', and select **Eureka Server**, **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
```

3) Create a **eureka-server.properties** file 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
```

Commit changes to the Git repository.

```
git add –A .
git commit –m "adding new configuration"
```

4) Rename package name to 'edu.aspire.eurekaserver' and add **@EnableEurekaServer** in EurekaServerApplication.java file.

5) Start Eureka server. Ensure that config server should be 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

6) Stop all Microservices and add the following additional dependency in all microservices in their pom.xml files to enable dynamic registration and discovery using the Eureka service.

```
<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.

7) The below property should be added to all microservices in their respective configuration files under `config-repo`. This will help the microservices to connect to the Eureka server.
eureka.client.serviceUrl.defaultZone: http://localhost:8761/eureka/

Commit all changes to Git repo
git add –A .
git commit –m "adding new configuration"

8) 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.

9) Start fare and search microservices
Observation:
On fare microservice console:
DiscoveryClient_FARES-SERVICE/Ramesh-PC:fares-service:8081 - registration status: 204

Eureka server console:
Registered instance FARES-SERVICE/Ramesh:fares-service:8081 with status UP

Web: Refresh http://localhost:8761
Instances currently registered with Eureka
FARES-SERVICE n/a (1) (1) UP (1) – Ramesh-PC:fares-service:8081

10) Eureka internally uses Ribbon for load balancing hence remove ribbon dependency from booking microservice pom 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> -->
```

11) Also remove the `@RibbonClient` annotation from the `FareServiceProxy` interface.

12) Update `@FeignClient(name="fares-service")` to match the actual Fare microservices' service ID.

13) Also remove the list of servers from the `booking-service.properties` file. Ensure that eureka.client.serviceUrl.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/
```

Commit changes to Git repository

```
git add –A .
git commit –m "adding new configuration"
```

14) Start booking and check-in microservices. Also start website.

Observation:

On booking console:

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