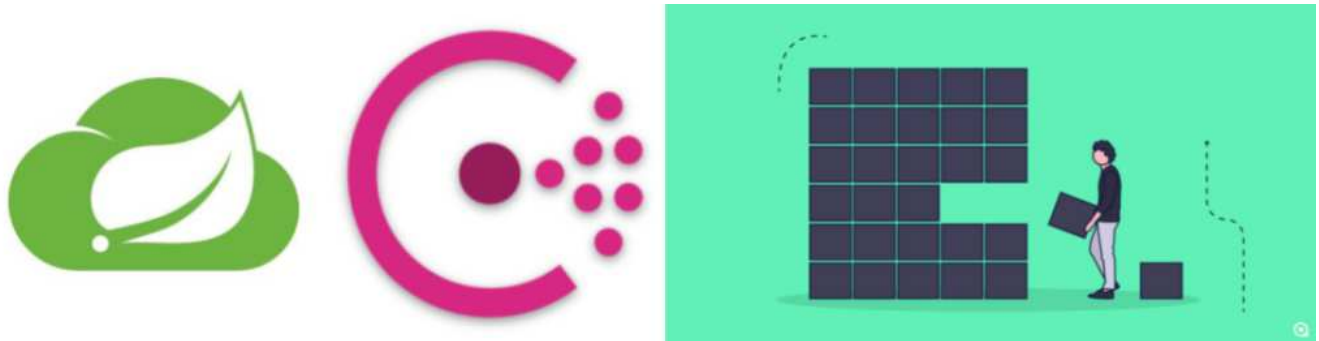# Piotr's TechBlog

Microservices with Spring Boot, Spring Cloud Gateway and Consul Cluster

The **Spring Cloud Consul** project provides integration for Consul and Spring Boot applications through auto-configuration. By using the well-known Spring Framework annotation style, we may enable and configure common patterns within microservice-based environments. These patterns include service discovery using Consul agent, distributed configuration using Consul key/value store, distributed events with Spring Cloud Bus, and Consul Events. The project also supports a client-side load balancer based on Netflix's Ribbon and an API gateway based on Spring Cloud Gateway.

In this article I will cover the following topics:

- Integrating Spring Boot application with Consul discovery
- Integrating Spring Cloud Gateway with Consul discovery
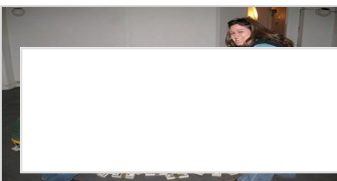
Microservices Architecture

Let's proceed to the example system. It consists of four independent microservices. Some of them may call endpoints exposed by the others. The application source code is available on GitHub here: https://github.com/piomin/sample-spring-cloud-consul.git.

In the current example, we will try to develop a simple order system where customers may buy products. If a customer decides to confirm a selected list of products to buy, the POST request is sent to the *order-service*. It is processed by the `Order prepare(@RequestBody Order order)` method inside REST controller. This method is responsible for order preparation. First, it calculates the final price, considering the price of each product from the list, customer order history, and their category in the system by calling the proper API method from the *customer-service*. Then, it verifies if the customer's account balance is enough to execute the order by calling the `account-service`, and finally, it returns the calculated price. If the customer confirms the action, the `PUT /{id}` method is called. The request is processed by the method `Order accept(@PathVariable Long id)` inside REST controller. It changes the order status and withdraws money from the customer's account. The system architecture is broken down into the individual microservices hidden behind API gateway as shown here:
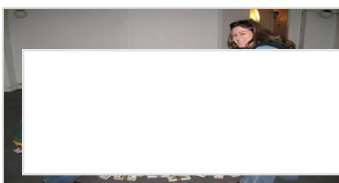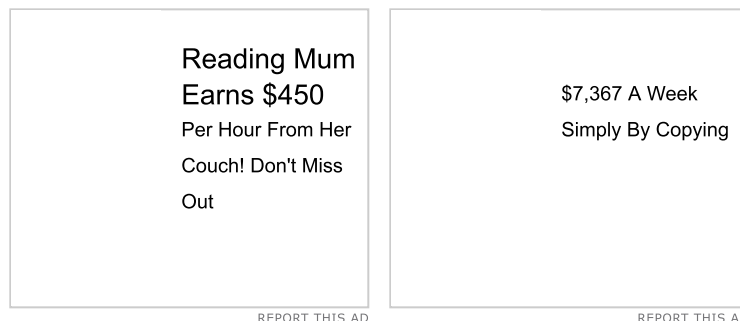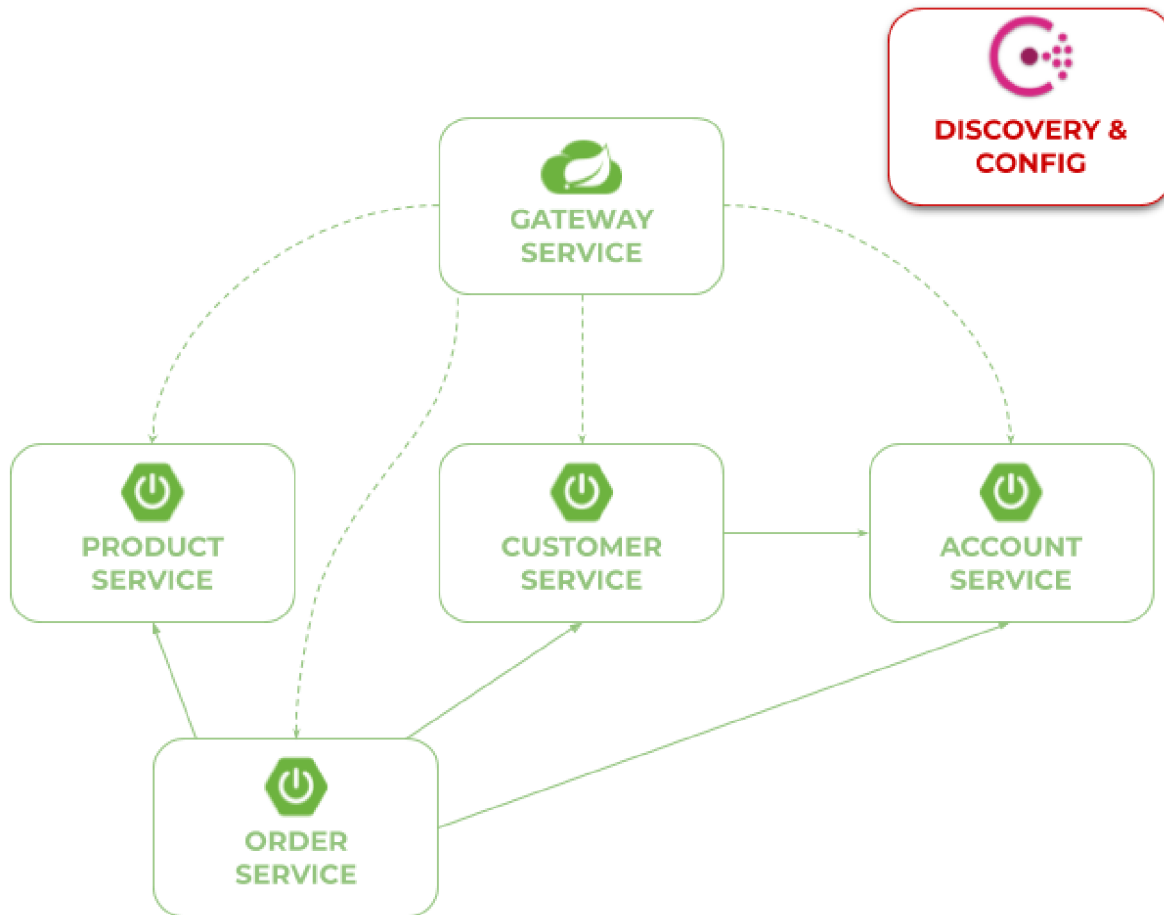
The description created above should give you a big picture of our example system. However, the business logic plays a supporting role, technically we have four Spring Boot applications using Consul discovery and KV store communicating with each other through REST APIs. The whole system is hidden for the external client behind API gateway built on top of Spring Cloud Gateway. Let's proceed to the implementation.

1. Building Microservices

Let's begin from dependencies. We use the current newest stable version of Spring Boot – `2.2.0.RELEASE` together with Spring Cloud Release Train `Hoxton.RC1`. The minimal set of required dependencies is to have Spring Web, Actuator (optionally)
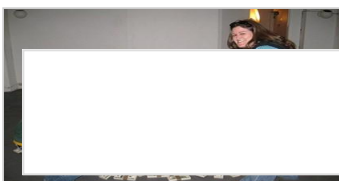
```xml
 4          <version>2.2.0.RELEASE</version>
 5      </parent>
 6
 7      <dependencyManagement>
 8          <dependencies>
 9              <dependency>
10                  <groupId>org.springframework.cloud</groupId>
11                  <artifactId>spring-cloud-dependencies</artifactId>
12                  <version>Hoxton.RC1</version>
13                  <type>pom</type>
14                  <scope>import</scope>
15              </dependency>
16          </dependencies>
17      </dependencyManagement>
18
19      <dependencies>
20          <dependency>
21              <groupId>org.springframework.cloud</groupId>
22              <artifactId>spring-cloud-starter-consul-all</artifactId>
23          </dependency>
24          <dependency>
25              <groupId>org.springframework.boot</groupId>
26              <artifactId>spring-boot-starter-web</artifactId>
27          </dependency>
28          <dependency>
29              <groupId>org.springframework.boot</groupId>
30              <artifactId>spring-boot-starter-actuator</artifactId>
31          </dependency>
32      </dependencies>
```

When running the application we will use dynamic listen port number generation feature by setting property `server.port` to `0`. Because we will run more than instance of every service we also need to override default value of `spring.cloud.consul.discovery.instance-id` which is based on port number that is not applicable when it is set to `0`. Here's our `application.yml` file for *account-service.*
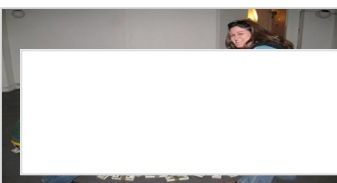
```
7    server:
8      port: 0
```

The configuration is deployed on Consul, which means we are only having `bootstrap.yml` file on classpath. If you have both Spring Cloud Consul Discovery and Config dependencies distributed configuration is enabled by default. You only have to override address of Consul server if required.

```
1    spring:
2      application:
3        name: account-service
4      cloud:
5        consul:
6          host: 192.168.99.100
7          port: 8500
```

In the current version of Spring Cloud we don't have to enable anything, so just need to declare the main class:

```
1    @SpringBootApplication
2    public class AccountApplication {
3
4        public static void main(String[] args) {
5            SpringApplication.run(AccountApplication.class);
6        }
7    }
```

2. Running Consul Cluster using Docker

In this section how to setup the local environment similar to the production mode. Therefore, we would like to have a scalable, production-grade service discovery infrastructure, consisting of some nodes working together inside the cluster. Consul provides support for clustering based on a gossip protocol used for communication between members and a Raft consensus protocol for a leadership election. I wouldn't like to go into the details of that process, but some basics about Consul architecture should be clarified.

The first important element is Consul agent. An agent is the long-running daemon
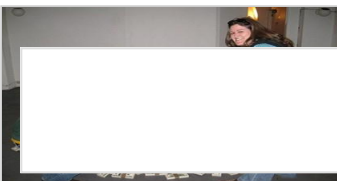
configure the Consul cluster using its Docker image. First, we will start the
container, which acts as a leader of the cluster. There is only one difference in the
currently used Docker command than for the standalone Consul server. We have set
the environment variable `CONSUL_BIND_INTERFACE=eth0` in order to change the
network address of the cluster agent from `127.0.0.1` to the one available for the
other member containers. My Consul server is now running at the internal address
`172.17.0.2`. To check out what your address is (it should be the same) you may run
the command `docker logs consul`. The appropriate information is logged just after
container startup. Here's the command that starts the first Consul node:

```
1  $ docker run -d --name consul-1 -p 8500:8500 -e CONSUL_BIND_INTERFACE=
```

Knowledge of that address is very important, since now we have to pass it to every
member container startup command as a cluster join parameter. We also bind it to
all
interfaces by setting `0.0.0.0` as a client address. Now, we may easily expose the
client agent API outside the container using the -p parameter:

```
1  $ docker run -d --name consul-2 -e CONSUL_BIND_INTERFACE=eth0 -p 8501:
2  $ docker run -d --name consul-3 -e CONSUL_BIND_INTERFACE=eth0 -p 8502:
```

After running two containers with Consul agent, you may check out the full list of
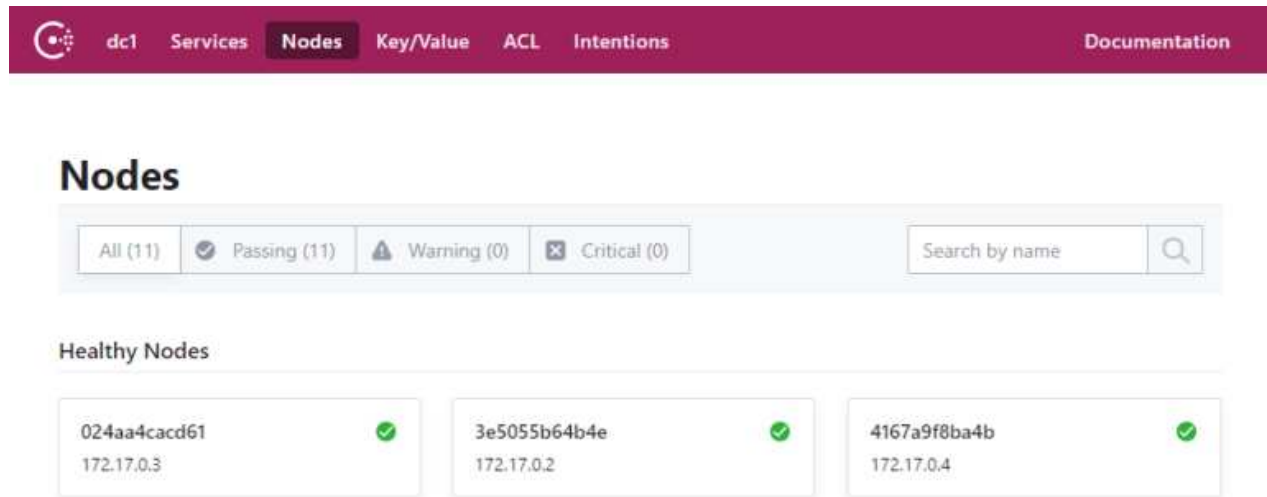cluster members by executing the following command on the leader's container:

```
PS C:\Users\minkowp> docker exec -t consul-1 consul members
```
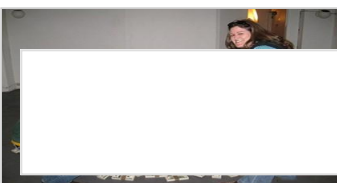
We can always get the same information using Consul Web Console.

We may easily change the default Consul node address for the Spring Boot application by changing configuration properties. Spring Cloud allows you to define only a single host address and port number of Consul agent. It is worth to note that in normal production mode with multiple machines you would install only Consul agent on every machine, which is connected with cluster of Consul servers.

```
1   spring:
2     application:
3       name: customer-service
4     cloud:
5       consul:
6         host: 192.168.99.100
7         port: 8501
```

3. Inter-service Communication

An inter-service communication is performed using OpenFeign declarative REST client. We can also include Spring Cloud Sleuth dependency for propagating `correlationId` between subsequent calls.
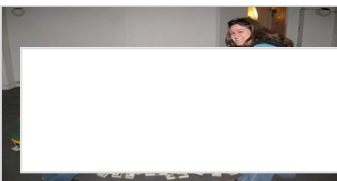
```
1   <dependency>
2     <groupId>org.springframework.cloud</groupId>
```

The OpenFeign client is auto-integrated with service discovery. To use it we need to declare interface with required methods for communication. The interface has to be annotated with `@FeignClient` that points to service using its discovery name.

```java
@FeignClient(name = "account-service")
public interface AccountClient {

    @GetMapping("/customer/{customerId}")
    List<Account> findByCustomer(@PathVariable("customerId") Long cust

}
```

Finally, OpenFeign client need to be enabled for the whole application.

```java
@SpringBootApplication
@EnableFeignClients
public class CustomerApplication {

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

}
```

4. Enable Zone Affinity Mechanism

When using Spring Cloud Discovery we may take advantage of zones affinity mechanism. If your microservices has been deployed to multiple zones, you may prefer that those services communicate with other services within the same zone before trying to access them in another zone. The same rule applies to API gateway

architecture of our system looks as shown below.
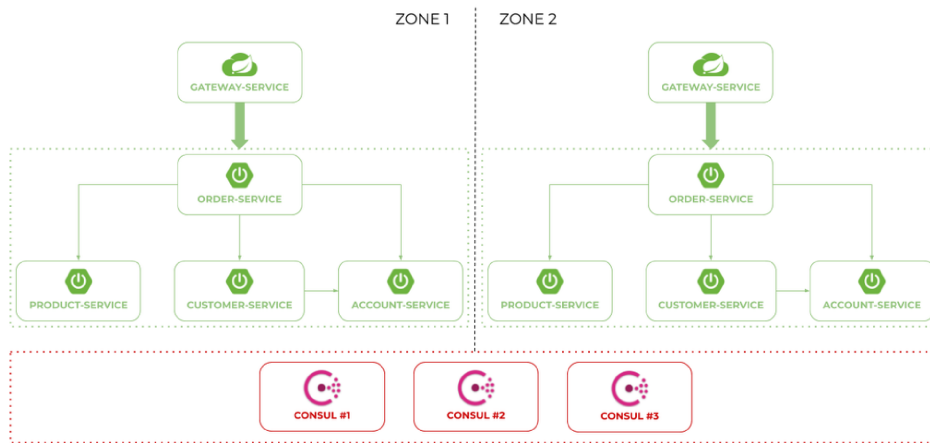
The whole mechanism is enabled through the configuration. We need to set default zone name for our microservice using property

`spring.cloud.consul.discovery.instanceZone`. I defined two profiles for each application, that may be set during startup with `--spring.profiles.active` command-line argument.
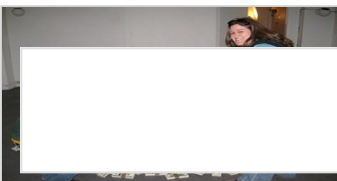
```
1    ---
2    spring:
3      profiles: zone1
4      cloud:
5        consul:
6          discovery:
7            instanceZone: zone1
8
9    ---
10   spring:
11     profiles: zone2
12     cloud:
13       consul:
14         discovery:
15           instanceZone: zone2
```
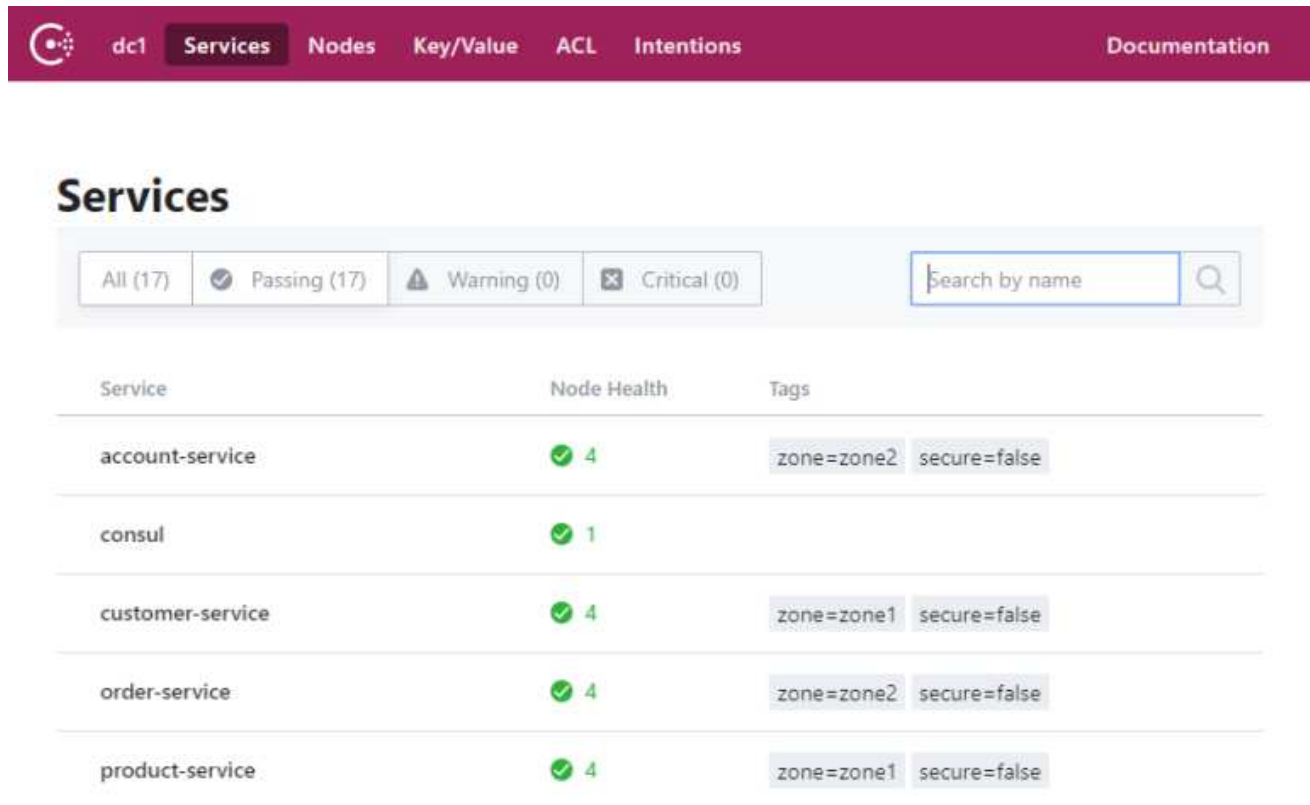
overridden with `spring.cloud.consul.discovery.defaultZoneMetadataName` property. Assuming you have run two instances of each microservice divided into two zones using the command, for example `java -jar --spring.profiles.active=zone1 target/order-service-1.1.jar` , you should see the following list of registered services on your Consul instance.

Here's more detailed view is *Nodes* section that prints all tags and listen port number for every instance of microservice.

We can also display all running instances of a single service. In the following picture you can see instances of *account-service*.

5. Building API Gateway with Spring Cloud

Since now, we have succesfully run all the microservices in two instances distributed across two different zones. Because they are all listening on dynamically generated ports we need an API gateway which is exposed on static port to an external client. Here's the list of dependencies used for building *gateway-service*:

```
1   <dependency>
2       <groupId>org.springframework.cloud</groupId>
3       <artifactId>spring-cloud-starter-gateway</artifactId>
4   </dependency>
5   <dependency>
6       <groupId>org.springframework.cloud</groupId>
7       <artifactId>spring-cloud-starter-consul-all</artifactId>
8   </dependency>
9   <dependency>
10      <groupId>org.springframework.boot</groupId>
11      <artifactId>spring-boot-starter-actuator</artifactId>
12  </dependency>
```
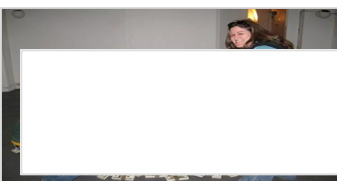
Because we would like run gateway on static port the configuration of Maven profiles is slightly larger than for microservices. We also don't need to register

```
 1   ---
 2   spring:
 3     profiles: zone1
 4     cloud:
 5       consul:
 6         discovery:
 7           instanceZone: zone1
 8           register: false
 9           registerHealthCheck: false
10           tags: zone=zone1
11   server:
12     port: ${PORT:8080}
13
14   ---
15   spring:
16     profiles: zone2
17     cloud:
18       consul:
19         discovery:
20           instanceZone: zone2
21           register: false
22           registerHealthCheck: false
23           tags: zone=zone2
24   server:
25     port: ${PORT:9080}
```

To enable integration with Consul discovery we need to set property
`spring.cloud.gateway.discovery.locator.enabled` `to` `true` . In order to expose service
under custom path we should define `Path` predicate and `RewritePath` filter for each
service. In that case *account-service* is available under address
http://localhost:8080/account/ , *customer-service* under
http://localhost:8080/customer/ etc.

```
 1   spring:
 2     cloud:
```
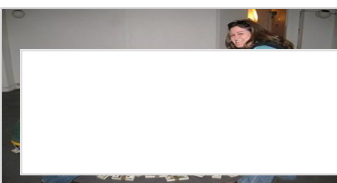
```
 9              uri: lb://account-service
10              predicates:
11                - Path=/account/**
12              filters:
13                - RewritePath=/account/(?<path>.*), /$\{path}
14          - id: customer-service
15              uri: lb://customer-service
16              predicates:
17                - Path=/customer/**
18              filters:
19                - RewritePath=/customer/(?<path>.*), /$\{path}
20          - id: order-service
21              uri: lb://order-service
22              predicates:
23                - Path=/order/**
24              filters:
25                - RewritePath=/order/(?<path>.*), /$\{path}
26          - id: product-service
27              uri: lb://product-service
28              predicates:
29                - Path=/product/**
30              filters:
31                - RewritePath=/product/(?<path>.*), /$\{path}
```

Now you can be sure that each request incoming to *gateway-service* started in `zone1` would be forwarded to in the first place to microservice also started in `zone1`. And the same for `zone2`.

6. Distributed Configuration

Consul Config is automatically enabled for the application just after including dependency `spring-cloud-starter-consul-config`. Of course it is included together with `spring-cloud-starter-consul-all` also. Configuration is stored in the `/config` folder by default. We can create the configuration per all applications or just for single application in a dedicated folder. Assuming we have four microservices and API gateway deployed in two zones we would have to define ten configuration folders. We have different options for storing application properties, but I chose YAML format. YAML must be set in the appropriate data key in consul. So the Consul folders structure for all our sample applications looks as shown below.
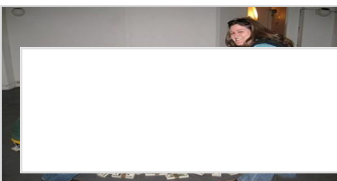
```
 6    config/order-service,zone2/data
 7    config/product-service,zone1/data
 8    config/product-service,zone2/data
 9    config/gateway-service,zone1/data
10    config/gateway-service,zone2/data
```
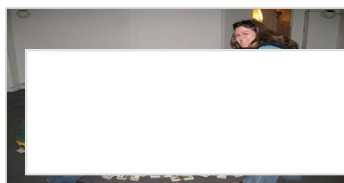
Here's the typical configuration for one our sample microservice running in `zone1` zone.

```
1    spring:
2      cloud:
3        consul:
4          discovery:
5            instanceId: "${spring.cloud.client.hostname}:${spring.applicat
6            instanceZone: zone1
7    server.port: 0
```

And the same configuration created on Consul for `account-service` with active `zone1` profile.

‹ Key / Values ‹ config

config/account-service,zone1/data

Key or folder

config/account-service.zone1/data

To create a folder, end a key with `/`

Value   🔵 Code

```yaml
1 spring:
2   cloud:
3     consul:
4       discovery:
5         instanceId: "${spring.cloud.client.hostname}:${spring.application.name}:${random.int[1,999999]}"
6         instanceZone: zone1
7
8 server.port: 0
```
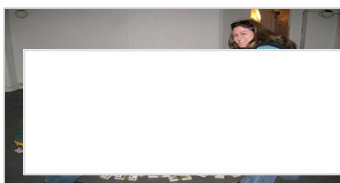
YAML ▼

Save

In case we use Consul Config for our application the only file that should be available on classpath is `bootstrap.yml`. Except overriding Consul IP address or port if required we have to set the format of configuration properties to `YAML`. Here's `bootstrap.yml` file for *account-service*.

```yaml
1  spring:
2    application:
3      name: account-service
4    cloud:
5      consul:
6        host: 192.168.99.100
7        port: 8500
8        config:
9          format: YAML
```

Summary