Spring Cloud

**Spring Cloud Config**

The Spring Cloud Config server is an externalized configuration server in which applications and services can deposit, access, and manage all runtime configuration properties. The Spring Config server also supports version control of the configuration properties.

In the earlier examples with 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.

A better approach is to use the concept of profiles. Profiles, as discussed in [Chapter 2](https://www.safaribooksonline.com/library/view/spring-developing-java/9781787127555/ch21.html), *Building Microservices with Spring Boot*, is used for partitioning different properties for different environments. The profile-specific configuration will be named application-{profile}.properties. For example, application-development.properties represents a property file targeted for the development environment.

However, the disadvantage of this approach is that the configurations are statically packaged along with the application. Any changes in the configuration properties require the application to be rebuilt.

There are alternate ways to externalize the configuration properties from the application deployment package. Configurable properties can also be read from an external source in a number of ways:

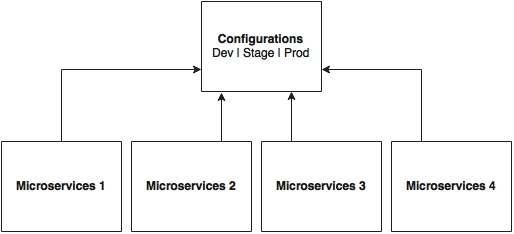
* From an external JNDI server using JNDI namespace (java:comp/env)
* Using the Java system properties (System.getProperties()) or using the –D command line option
* Using the PropertySource configuration:
* @PropertySource("file:${CONF\_DIR}/application.properties")
* public class ApplicationConfig {

}

* Using a command-line parameter pointing a file to an external location:
* **java -jar myproject.jar --spring.config.location=**

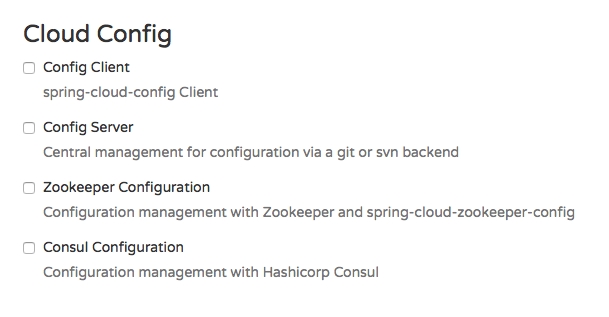
JNDI operations are expensive, lack flexibility, have difficulties in replication, and are not version controlled. System.properties is not flexible enough for large-scale deployments. The last two options rely on a local or a shared filesystem mounted on the server.

For large scale deployments, a simple yet powerful centralized configuration management solution is required:



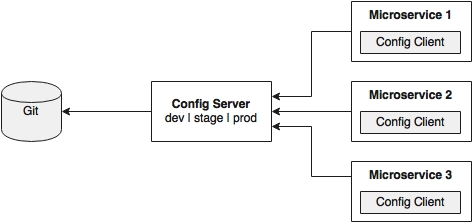
As shown in the preceding diagram, all microservices point to a central server to get the required configuration parameters. The microservices then locally cache these parameters to improve performance. The Config server propagates the configuration state changes to all subscribed microservices so that the local cache's state can be updated with the latest changes. The Config server also uses profiles to resolve values specific to an environment.

As shown in the following screenshot, there are multiple options available under the Spring Cloud project for building the configuration server. **Config Server**, **Zookeeper Configuration**, and **Consul Configuration** are available as options. However, this chapter will only focus on the Spring Config server implementation:



The Spring Config server stores properties in a version-controlled repository such as Git or SVN. The Git repository can be local or remote. A highly available remote Git server is preferred for large scale distributed microservice deployments.

The Spring Cloud Config server architecture is shown in the following diagram:

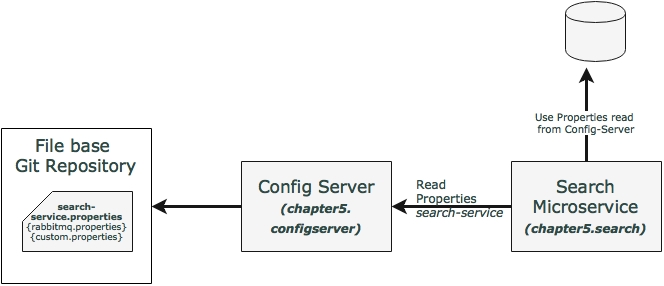


As shown in the preceding diagram, the Config client embedded in the Spring Boot microservices does a configuration lookup from a central configuration server using a simple declarative mechanism, and stores properties into the Spring environment. The configuration properties can be application-level 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. To make this work in a Spring Boot application, rename the application.\* file to bootstrap.\*.

**What's next?**

The next few sections demonstrate how to use the Config server in a real-world scenario. In order to do this, we will modify our search microservice (chapter5.search) to use the Config server. The following diagram depicts the scenario:



In this example, the Search service will read the Config server at startup by passing the service name. In this case, the service name of the search service will be search-service. The properties configured for the search-service include the RabbitMQ properties as well as a custom property.

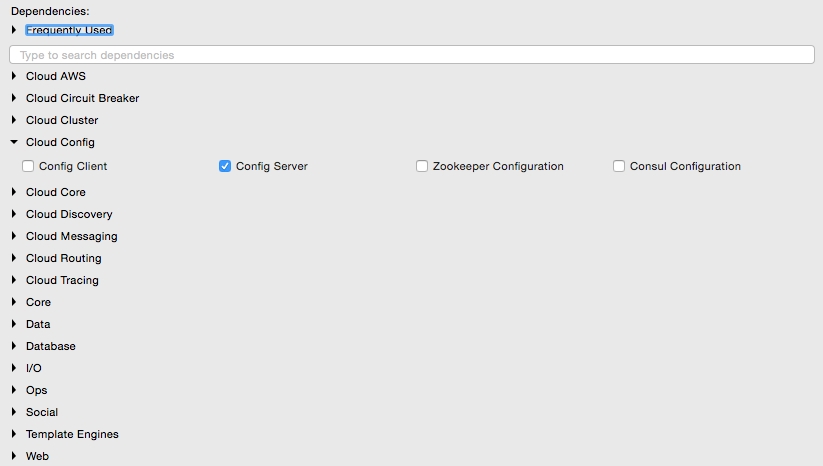
**Note**

The full source code of this section is available under the chapter5.configserver project in the code files.

**Setting up the Config server**

The following steps need to be followed to create a new Config server using STS:

1. Create a new **Spring Starter Project**, and select **Config Server** and **Actuator** as shown in the following diagram:

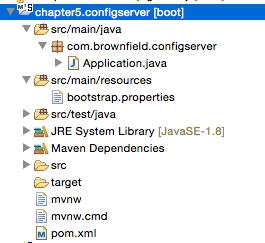


1. Set up a Git repository. This can be done by pointing to a remote Git configuration repository like the one at <https://github.com/spring-cloud-samples/config-repo>. This URL is an indicative one, a Git repository used by the Spring Cloud examples. We will have to use our own Git repository instead.
2. Alternately, a local filesystem-based Git repository can be used. In a real production scenario, an external Git is recommended. The Config server in this chapter will use a local filesystem-based Git repository for demonstration purposes.
3. Enter the commands listed next to set up a local Git repository:
4. **$ cd $HOME**
5. **$ mkdir config-repo**
6. **$ cd config-repo**
7. **$ git init .**
8. **$ echo message : helloworld > application.properties**
9. **$ git add -A .**
10. **$ git commit -m "Added sample application.properties"**

This code snippet creates a new Git repository on the local filesystem. A property file named application.properties with a message property and value helloworld is also created.

The file application.properties is created for demonstration purposes. We will change this in the subsequent sections.

1. The next step is to change the configuration in the Config server to use the Git repository created in the previous step. In order to do this, rename the file application.properties to bootstrap.properties:



1. Edit the contents of the new bootstrap.properties file to match the following:
2. server.port=8888

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

Port 8888 is the default port for the Config server. Even without configuring server.port, the Config server should bind to 8888. In the Windows environment, an extra / is required in the file URL.

1. Optionally, rename the default package of the auto-generated Application.java from com.example to com.brownfield.configserver. Add @EnableConfigServer in Application.java:
2. @EnableConfigServer
3. @SpringBootApplication

public class ConfigserverApplication {

1. Run the Config server by right-clicking on the project, and running it as a Spring Boot app.
2. 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.
3. Check http://localhost:8888/application/default/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","version":"6046fd2ff4fa09d3843767660d963866ffcc7d28","propertySources":[{"name":"file:///Users/rvlabs /config-repo /application.properties","source":{"message":"helloworld"}}]}

**Understanding the Config server URL**

In the previous section, we used http://localhost:8888/application/default/master to explore the properties. How do we interpret this URL?

The first element in the URL is the application name. In the given example, the application name should 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. There can be more than one profile configured within the repository for an application. The profiles can be used in various scenarios. The two common scenarios are segregating different environments such as Dev, Test, Stage, Prod, and the like, or segregating server configurations such as Primary, Secondary, and so on. The first one represents different environments of an application, whereas the second one represents different servers where an application is deployed.

The profile names are logical names that will be used for matching the file name in the repository. The default profile is named default. To configure properties for different environments, we have to configure different files as given in the following example. In this example, the first file is for the development environment whereas the second is for the production environment:

application-development.properties

application-production.properties

These are accessible using the following URLs respectively:

* http://localhost:8888/application/development
* http://localhost:8888/application/production

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.

In short, the URL is based on the following pattern: http://localhost:8888/{name}/{profile}/{label}.

The configuration can also be accessed by ignoring the profile. In the preceding example, all the following three URLs point to the same configuration:

* http://localhost:8888/application/default
* http://localhost:8888/application/master
* http://localhost:8888/application/default/master

There is an option to have different Git repositories for different profiles. This makes sense for production systems, since the access to different repositories could be different.

**Accessing the Config Server from clients**

In the previous section, a Config server is set up and accessed using a web browser. In this section, the Search microservice will be modified to use the Config server. The Search microservice will act as a Config client.

Follow these steps to use the Config server instead of reading properties from the application.properties file:

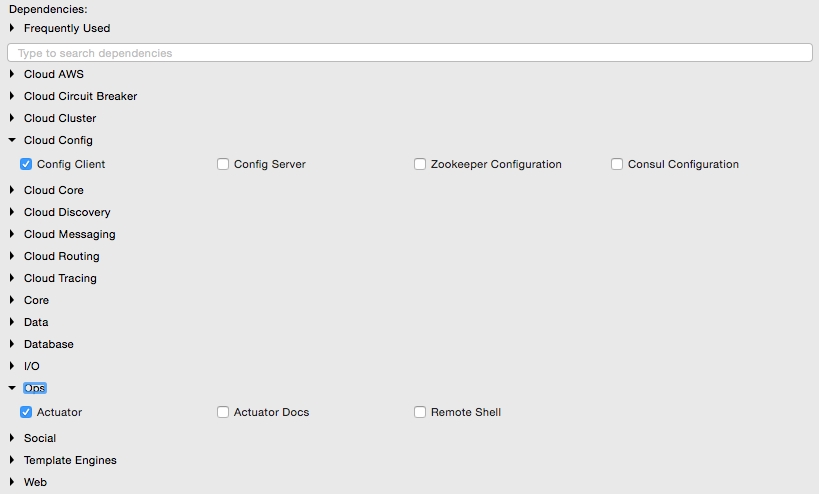
1. Add the Spring Cloud Config dependency and the actuator (if the actuator is not already in place) to the pom.xml file. The actuator is mandatory for refreshing the configuration properties:
2. <dependency>
3. <groupId>org.springframework.cloud</groupId>
4. <artifactId>spring-cloud-starter-config</artifactId>

</dependency>

1. 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. This is not required if the project is created from scratch:
2. <dependencyManagement>
3. <dependencies>
4. <dependency>
5. <groupId>org.springframework.cloud</groupId>
6. <artifactId>spring-cloud-dependencies</artifactId>
7. <version>Brixton.RELEASE</version>
8. <type>pom</type>
9. <scope>import</scope>
10. </dependency>
11. </dependencies>

</dependencyManagement>

1. The next screenshot shows the Cloud starter library selection screen. If the application is built from the ground up, select the libraries as shown in the following screenshot:



1. Rename application.properties to bootstrap.properties, and add an application name and a configuration server URL. The configuration server URL is not mandatory if the Config server is running on the default port (8888) on the local host:

The new bootstrap.properties file will look as follows:

spring.application.name=search-service

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

server.port=8090

spring.rabbitmq.host=localhost

spring.rabbitmq.port=5672

spring.rabbitmq.username=guest

spring.rabbitmq.password=guest

search-service is a logical name given to the Search microservice. This will be treated as service ID. The Config server will look for search-service.properties in the repository to resolve the properties.

1. Create a new configuration file for search-service. Create a new search-service.properties under the config-repo folder where the Git repository is created. Note that search-service is the service ID given to the Search microservice in the bootstrap.properties file. Move service-specific properties from bootstrap.properties to the new search-service.properties file. The following properties will be removed from bootstrap.properties, and added to search-service.properties:
2. spring.rabbitmq.host=localhost
3. spring.rabbitmq.port=5672
4. spring.rabbitmq.username=guest

spring.rabbitmq.password=guest

1. In order to demonstrate the centralized configuration of properties and propagation of changes, add a new application-specific property to the property file. We will add originairports.shutdown to temporarily take out an airport from the search. Users will not get any flights when searching for an airport mentioned in the shutdown list:

originairports.shutdown=SEA

In this example, we will not return any flights when searching with SEA as origin.

1. Commit this new file into the Git repository by executing the following commands:
2. **git add –A .**
3. **git commit –m "adding new configuration"**
4. The final search-service.properties file should look as follows:
5. spring.rabbitmq.host=localhost
6. spring.rabbitmq.port=5672
7. spring.rabbitmq.username=guest
8. spring.rabbitmq.password=guest

originairports.shutdown:SEA

1. The chapter5.search project's bootstrap.properties should look like the following:
2. spring.application.name=search-service
3. server.port=8090

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

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:

@RefreshScope

1. Add the following instance variable as a place holder for the new property that is just added in the Config server. The property names in the search-service.properties file must match:
2. @Value("${originairports.shutdown}")

private String originAirportShutdownList;

1. Change the application code to use this property. This is done by modifying the search method as follows:
2. @RequestMapping(value="/get", method = RequestMethod.POST)
3. List<Flight> search(@RequestBody SearchQuery query){
4. logger.info("Input : "+ query);
5. if(Arrays.asList(originAirportShutdownList.split(",")).contains(query.getOrigin())){
6. logger.info("The origin airport is in shutdown state");
7. return new ArrayList<Flight>();
8. }
9. 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, the search method will return an empty flight list.

1. Start the Config server. Then start the Search microservice. Make sure that the RabbitMQ server is running.
2. Modify the chapter5.website project to match the bootstrap.properties content as follows to utilize the Config server:
3. spring.application.name=test-client
4. server.port=8001

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

1. Change the run method of CommandLineRunner in Application.java to query SEA as the origin airport:

SearchQuery = new SearchQuery("SEA","SFO","22-JAN-16");

1. Run the chapter5.website project. The CommandLineRunner will now return an empty flight list. The following message will be printed in the server:
2. **The origin airport is in shutdown state**

**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

Commit the change in the Git repository. Refresh the Config server URL (http://localhost:8888/search-service/default) for this service and see whether the property change is reflected. If everything is fine, we will see the property change. The preceding request will force the Config server to read the property file again from the repository.

1. Rerun the website project again, and observe the CommandLineRunner execution. Note that in this case, we are not restarting the Search microservice nor the Config server. The service returns an empty flight list as earlier, and still complains as follows:
2. **The origin airport is in shutdown state**

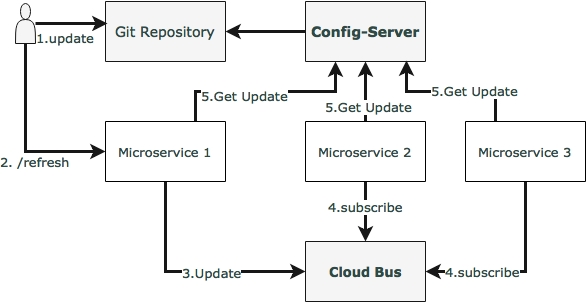
This means the change is not reflected in the Search service, and the service is still working with an old copy of the configuration properties.

1. 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:
2. **curl –d {} localhost:8090/refresh**
3. Rerun the website project, and observe the CommandLineRunner execution. This should return the list of flights that we have requested from SEA. Note that the website project may fail if the Booking service is not up and running.

The /refresh endpoint will refresh the locally cached configuration properties, and reload fresh values from the Config server.

**Spring Cloud Bus for propagating configuration changes**

With the preceding approach, configuration parameters can be changed without restarting the microservices. This is good when there are only one or two instances of the services running. What happens if there are many instances? For example, if there are five instances, then we have to hit /refresh against each service instance. This is definitely a cumbersome activity:



The Spring Cloud Bus provides a mechanism to refresh configurations across multiple instances without knowing how many instances there are, or their locations. This is particularly handy when there are many service instances of a microservice running or when there are many microservices of different types running. This is done by connecting all service instances through a single message broker. Each instance subscribes for change events, and refreshes its local configuration when required. This refresh is triggered by making a call to any one instance by hitting the /bus/refresh endpoint, which then propagates the changes through the cloud bus and the common message broker.

In this example, RabbitMQ is used as the AMQP message broker. Implement this by following the steps documented as follows:

1. Add a new dependency in the chapter5.search project's pom.xml file to introduce the Cloud Bus dependency:
2. <dependency>
3. <groupId>org.springframework.cloud</groupId>
4. <artifactId>spring-cloud-starter-bus-amqp</artifactId>

</dependency>

1. The Search microservice also needs connectivity to the RabbitMQ, but this is already provided in search-service.properties.
2. Rebuild and restart the Search microservice. In this case, we will run two instances of the Search microservice from a command line, as follows:
3. **java -jar -Dserver.port=8090 search-1.0.jar**
4. **java -jar -Dserver.port=8091 search-1.0.jar**

The two instances of the Search service will be now running, one on port 8090 and another one on 8091.

1. Rerun the website project. This is just to make sure that everything is working. The Search service should return one flight at this point.
2. Now, update search-service.properties with the following value, and commit to Git:

originairports.shutdown:SEA

1. Run the following command to /bus/refresh. Note that we are running a new bus endpoint against one of the instances, 8090 in this case:
2. **curl –d {} localhost:8090/bus/refresh**
3. Immediately, we will see the following message for both instances:
4. **Received remote refresh request. Keys refreshed [originairports.shutdown]**

The bus endpoint sends a message to the message broker internally, which is eventually consumed by all instances, reloading their property files. Changes can also be applied to a specific application by specifying the application name like so:

/bus/refresh?destination=search-service:\*\*

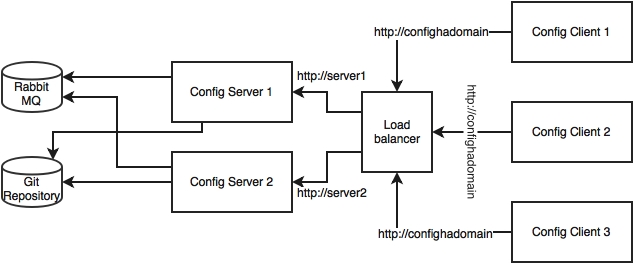
We can also refresh specific properties by setting the property name as a parameter.

**Setting up high availability for the Config server**

The previous sections explored how to set up the Config server, allowing real-time refresh of configuration properties. However, the Config server is a single point of failure in this architecture.

There are three single points of failure in the default architecture that was established in the previous section. One of them is the availability of the Config server itself, the second one is the Git repository, and the third one is the RabbitMQ server.

The following diagram shows a high availability architecture for the Config server:



The architecture mechanisms and rationale are explained as follows:

The Config server requires high availability, since the services won't be able to bootstrap if the Config server is not available. Hence, redundant Config servers are required for high availability. However, the applications can continue to run if the Config server is unavailable after the services are bootstrapped. In this case, services will run with the last known configuration state. Hence, the Config server availability is not at the same critical level as the microservices availability.

In order to make the Config server highly available, we need multiple instances of the Config servers. Since the Config server is a stateless HTTP service, multiple instances of configuration servers can be run in parallel. Based on the load on the configuration server, a number of instances have to be adjusted. The bootstrap.properties file is not capable of handling more than one server address. Hence, multiple configuration servers should be configured to run behind a load balancer or behind a local DNS with failover and fallback capabilities. The load balancer or DNS server URL will be configured in the microservices' bootstrap.properties file. This is with the assumption that the DNS or the load balancer is highly available and capable of handling failovers.

In a production scenario, it is not recommended to use a local file-based Git repository. The configuration server should be typically backed with a highly available Git service. This is possible by either using an external highly available Git service or a highly available internal Git service. SVN can also be considered.

Having said that, an already bootstrapped Config server is always capable of working with a local copy of the configuration. Hence, we need a highly available Git only when the Config server needs to be scaled. Therefore, this too is not as critical as the microservices availability or the Config server availability.

**Note**

The GitLab example for setting up high availability is available at <https://about.gitlab.com/high-availability/>.

RabbitMQ also has to be configured for high availability. The high availability for RabbitMQ is needed only to push configuration changes dynamically to all instances. Since this is more of an offline controlled activity, it does not really require the same high availability as required by the components.

RabbitMQ high availability can be achieved by either using a cloud service or a locally configured highly available RabbitMQ service.

**Note**

Setting up high availability for Rabbit MQ is documented at <https://www.rabbitmq.com/ha.html>.

**Monitoring the Config server health**

The Config server is nothing but a Spring Boot application, and is, by default, configured with an actuator. Hence, all actuator endpoints are applicable for the Config server. The health of the server can be monitored using the following actuator URL: http://localhost:8888/health.

**Config server for configuration files**

We may run into scenarios where we need a complete configuration file such as logback.xml to be externalized. The Config server provides a mechanism to configure and store such files. This is achievable by using the URL format as follows: /{name}/{profile}/{label}/{path}.

The name, profile, and label have the same meanings as explained earlier. The path indicates the file name such as logback.xml.

**Completing changes to use the Config server**

In order to build this capability to complete BrownField Airline's PSS, we have to make use of the configuration server for all services. All microservices in the examples given in chapter5.\* need to make similar changes to look to the Config server for getting the configuration parameters.

The following are a few key change considerations:

* The Fare service URL in the booking component will also be externalized:
* private static final String FareURL = "/fares";
* @Value("${fares-service.url}")
* private String fareServiceUrl;

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

As shown in the preceding code snippet, the Fare service URL is fetched through a new property: fares-service.url.

* We are not externalizing the queue names used in the Search, Booking, and Check-in services at the moment. Later in this chapter, these will be changed to use Spring Cloud Streams.

# 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, and data is sent across to the other service. 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.

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 code snippet is the existing code in the Booking microservice for calling the Fare service:

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

In order to use Feign, first we need to change the pom.xml file to include the Feign dependency as follows:

<dependency>

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

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

</dependency>

For a new Spring Starter project, **Feign** can be selected from the starter library selection screen, or from <http://start.spring.io/>. This is available under **Cloud Routing** as shown in the following screenshot:



The next step is to create a new FareServiceProxy interface. This will act as a proxy interface of the actual Fare service:

@FeignClient(name="fares-proxy", url="localhost:8080/fares")

public interface FareServiceProxy {

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

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

}

The FareServiceProxy interface has a @FeignClient annotation. This annotation tells Spring to create a REST client based on the interface provided. The value could be a service ID or a logical name. The url indicates the actual URL where the target service is running. Either name or value is mandatory. In this case, since we have url, the name attribute is irrelevant.

Use this service proxy to call the Fare service. 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.

Change BookingComponent, and make changes to the calling part. This is as simple as calling another Java interface:

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

Rerun the Booking microservice to see the effect.

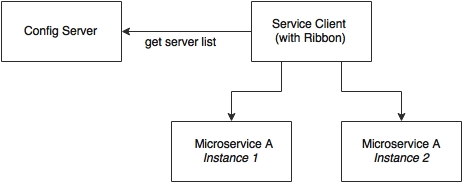
The URL of the Fare service in the FareServiceProxy interface is hardcoded: url="localhost:8080/fares".

For the time being, we will keep it like this, but we are going to change this later in this chapter.

# Ribbon for load balancing

In the previous setup, 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. It also helps us to handle server failures transparent to the client.

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. There could be other load balancing algorithms possible with the Ribbon library. Spring Cloud offers a declarative way to configure and use the Ribbon client.



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.

In order to use the Ribbon client, we will have to add the following dependency to the pom.xml file:

<dependency>

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

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

</dependency>

In case of development from ground up, this can be selected from the Spring Starter libraries, or from <http://start.spring.io/>. Ribbon is available under **Cloud Routing**:



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

fares-proxy.ribbon.listOfServers=localhost:8080,localhost:8081

Going back and editing the FareServiceProxy class created in the previous section to use the Ribbon client, we note that the value of the @RequestMapping annotations is changed from /get to /fares/get so that we can move the host name and port to the configuration easily:

@FeignClient(name="fares-proxy")

@RibbonClient(name="fares")

public interface FareServiceProxy {

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

We can now run two instances of the Fares microservices. Start one of them on 8080, and the other one on 8081:

**java -jar -Dserver.port=8080 fares-1.0.jar**

**java -jar -Dserver.port=8081 fares-1.0.jar**

Run the Booking microservice. When the Booking microservice is bootstrapped, the CommandLineRunner automatically inserts one booking record. This will go to the first server.

When running the website project, it calls the Booking service. This request will go to the second server.

On the Booking service, we see the following trace, which says there are two servers enlisted:

**DynamicServerListLoadBalancer:{NFLoadBalancer:name=fares-proxy,current**

**list of Servers=[localhost:8080, localhost:8081],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;]},**

**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.

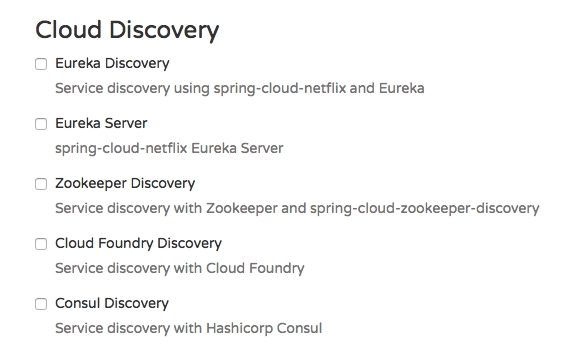
**Understanding dynamic service registration and discovery**

Dynamic registration is primarily from the service provider's point of view. 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. The registry always keeps up-to-date information of the services available, as well as their metadata.

Dynamic discovery is applicable from the service consumer's point of view. 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.

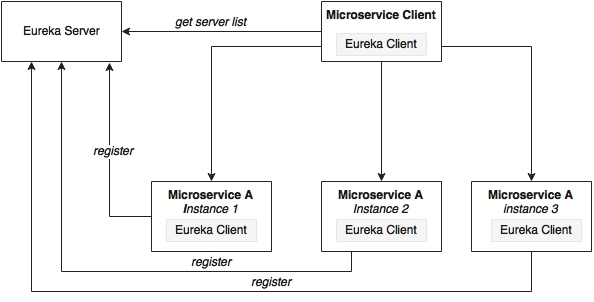
The clients may keep a local cache of the registry data for faster access. Some registry implementations allow clients to keep a watch on the items they are interested in. In this approach, the state changes in the registry server will be propagated to the interested parties to avoid using stale data.

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, as shown in the <http://start.spring.io/> screenshot given next. Etcd is another service registry available outside of Spring Cloud to achieve dynamic service registration and discovery. In this chapter, we will focus on the Eureka implementation:



**Understanding Eureka**

Spring Cloud Eureka also comes from Netflix OSS. The Spring Cloud project provides a Spring-friendly declarative approach for integrating Eureka with Spring-based applications. 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.

When a microservice is bootstrapped, it reaches out to the Eureka server, and advertises its existence with the binding information. Once registered, the service endpoint sends ping requests to the registry every 30 seconds to renew its lease. If a service endpoint cannot renew its lease in a few attempts, that service endpoint will be taken out of the service registry. The registry information will be replicated to all Eureka clients so that the clients have to go to the remote Eureka server for each and every request. Eureka clients fetch the registry information from the server, and cache it locally. After that, the clients use that information to find other services. This information is updated periodically (every 30 seconds) by getting the delta updates between the last fetch cycle and the current one.

When a client wants to contact a microservice endpoint, the Eureka client provides a list of currently available services based on the requested service ID. The Eureka server is zone aware. Zone information can also be supplied when registering a service. When a client requests for a services instance, the Eureka service tries to find the service running in the same zone. The Ribbon client then load balances across these available service instances supplied by the Eureka client. The communication between the Eureka client and the server is done using REST and JSON.

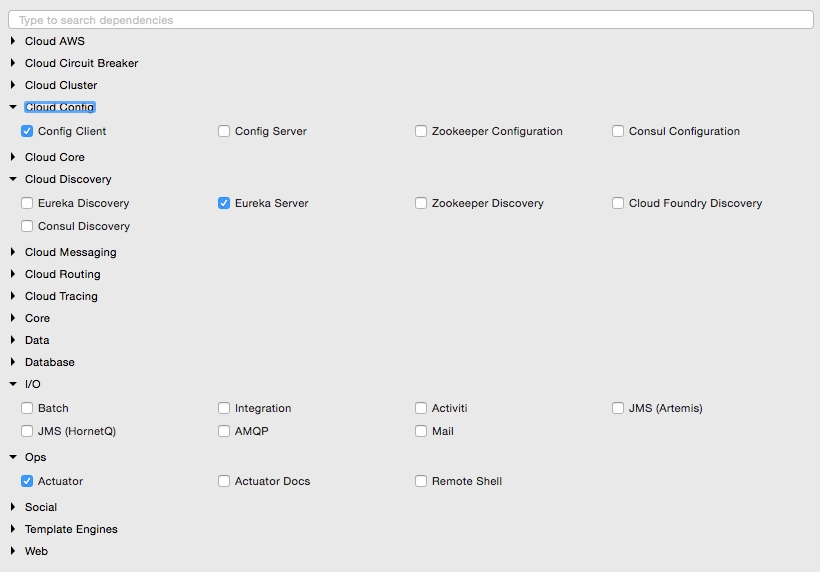
**Setting up the Eureka server**

In this section, we will run through the steps required for setting up the Eureka server.

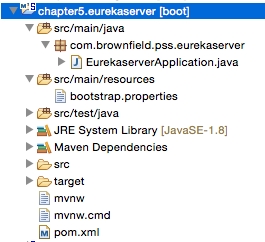
**Note**

The full source code of this section is available under the chapter5.eurekaserver project in the code files. Note that the Eureka server registration and refresh cycles take up to 30 seconds. Hence, when running services and clients, wait for 40-50 seconds.

1. Start a new Spring Starter project, and select **Config Client**, **Eureka Server**, and **Actuator**:



The project structure of the Eureka server is shown in the following image:



Note that the main application is named EurekaserverApplication.java.

1. Rename application.properties to bootstrap.properties since this is using the Config server. As we did earlier, configure the details of the Config server in the bootsratp.properties file so that it can locate the Config server instance. The bootstrap.properties file will look as follows:
2. spring.application.name=eureka-server1
3. server.port:8761

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

The Eureka server can be set up in a standalone mode or in a clustered mode. We will start with the standalone mode. By default, the Eureka server itself is another Eureka client. This is particularly useful when there are multiple Eureka servers running for high availability. The client component is responsible for synchronizing state from the other Eureka servers. The Eureka client is taken to its peers by configuring the eureka.client.serviceUrl.defaultZone property.

In the standalone mode, we point eureka.client.serviceUrl.defaultZone back to the same standalone instance. Later we will see how we can run Eureka servers in a clustered mode.

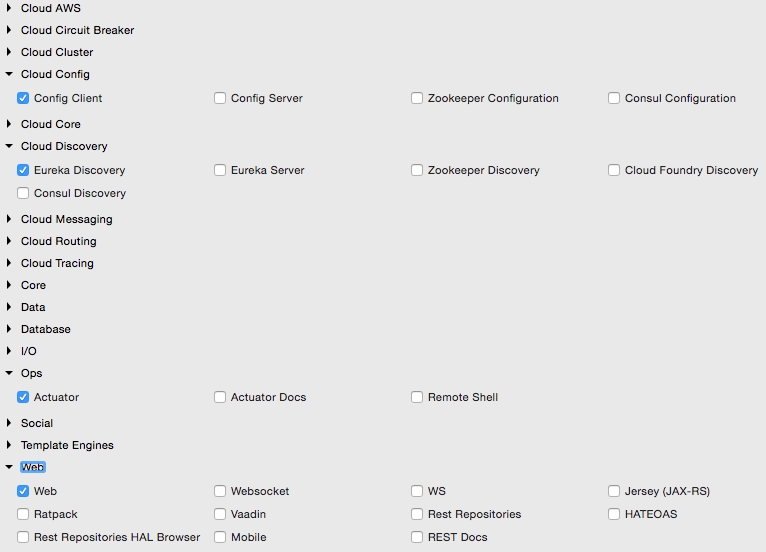
1. Create a eureka-server1.properties file, and update it in the Git repository. eureka-server1 is the name of the application given in the application's bootstrap.properties file in the previous step. As shown in the following code, serviceUrl points back to the same server. Once the following properties are added, commit the file to the Git repository:
2. spring.application.name=eureka-server1
3. eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/
4. eureka.client.registerWithEureka:false

eureka.client.fetchRegistry:false

1. Change the default Application.java. In this example, the package is also renamed as com.brownfield.pss.eurekaserver, and the class name changed to EurekaserverApplication. In EurekaserverApplication, add @EnableEurekaServer:
2. @EnableEurekaServer
3. @SpringBootApplication

public class EurekaserverApplication {

1. We are now ready to start the Eureka server. Ensure that the Config server is also started. Right-click on the application and then choose **Run As** | **Spring Boot App**. Once the application is started, open http://localhost:8761 in a browser to see the Eureka console.
2. In the console, note that there is no instance registered under **Instances currently registered with Eureka**. Since no services have been started with the Eureka client enabled, the list is empty at this point.
3. Making a few changes to our microservice will enable dynamic registration and discovery using the Eureka service. To do this, first we have to add the Eureka dependencies to the pom.xml file. If the services are being built up fresh using the Spring Starter project, then select **Config Client**, **Actuator**, **Web** as well as **Eureka discovery** client as follows:



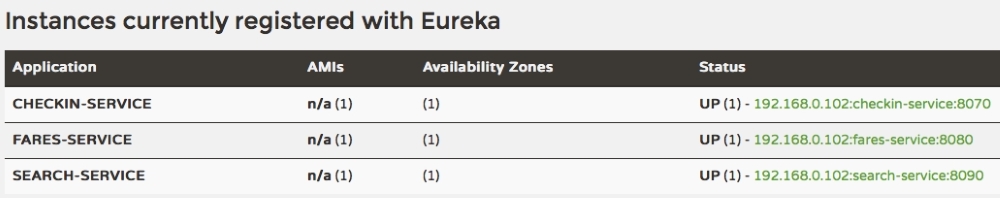
1. Since we are modifying our microservices, add the following additional dependency to all microservices in their pom.xml files:
2. <dependency>
3. <groupId>org.springframework.cloud</groupId>
4. <artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

1. The following property has to be added to all microservices in their respective configuration files under config-repo. This will help the microservices to connect to the Eureka server. Commit to Git once updates are completed:

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

1. Add @EnableDiscoveryClient to 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. Start all servers except Booking. Since we are using the Ribbon client on the Booking service, the behavior could be different when we add the Eureka client in the class path. We will fix this soon.
3. Going to the Eureka URL (http://localhost:8761), you can see that all three instances are up and running:



Time to fix the issue with Booking. We will remove our earlier Ribbon client, and use Eureka instead. Eureka internally uses Ribbon for load balancing. Hence, the load balancing behavior will not change.

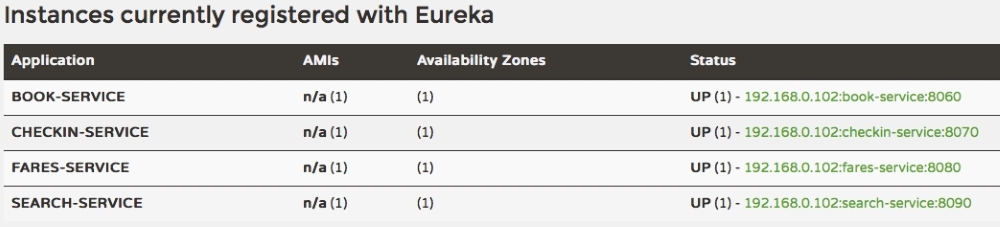
1. Remove the following dependency:
2. <dependency>
3. <groupId>org.springframework.cloud</groupId>
4. <artifactId>spring-cloud-starter-ribbon</artifactId>

</dependency>

1. Also remove the @RibbonClient(name="fares") annotation from the FareServiceProxy class.
2. Update @FeignClient(name="fares-service") to match the actual Fare microservices' service ID. In this case, fare-service is the service ID configured in the Fare microservices' bootstrap.properties. This is the name that the Eureka discovery client sends to the Eureka server. The service ID will be used as a key for the services registered in the Eureka server.
3. Also remove the list of servers from the booking-service.properties file. With Eureka, we are going to dynamically discover this list from the Eureka server:

fares-proxy.ribbon.listOfServers=localhost:8080, localhost:8081

1. Start the Booking service. You will see that CommandLineRunner successfully created a booking, which involves calling the Fare services using the Eureka discovery mechanism. Go back to the URL to see all the registered services:



1. Change the website project's bootstrap.properties file to make use of Eureka rather than connecting directly to the service instances. We will not use the Feign client in this case. Instead, for demonstration purposes, we will use the load balanced RestTemplate. Commit these changes to the Git repository:
2. spring.application.name=test-client

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

1. Add @EnableDiscoveryClient to the Application class to make the client Eureka-aware.
2. Edit both Application.java as well as BrownFieldSiteController.java. Add three RestTemplate instances. This time, we annotate them with @Loadbalanced to ensure that we use the load balancing features using Eureka and Ribbon. RestTemplate cannot be automatically injected. Hence, we have to provide a configuration entry as follows:
3. @Configuration
4. class AppConfiguration {
5. @LoadBalanced
6. @Bean
7. RestTemplate restTemplate() {
8. return new RestTemplate();
9. }
10. }
11. @Autowired
12. RestTemplate searchClient;
14. @Autowired
15. RestTemplate bookingClient;
17. @Autowired

RestTemplate checkInClient;

1. We use these RestTemplate instances to call the microservices. Replace the hardcoded URLs with service IDs that are registered in the Eureka server. In the following code, we use the service names search-service, book-service, and checkin-service instead of explicit host names and ports:
2. Flight[] flights = searchClient.postForObject("http://search-service/search/get", searchQuery, Flight[].class);
3. long bookingId = bookingClient.postForObject("http://book-service/booking/create", booking, long.class);

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

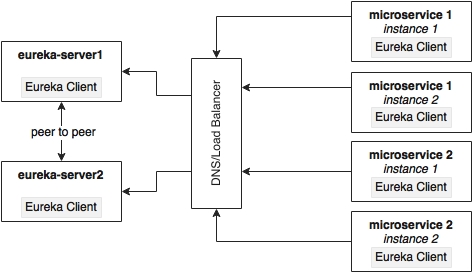
1. We are now ready to run the client. Run the website project. If everything is fine, the website project's CommandLineRunner will successfully perform search, booking, and check-in. The same can also be tested using the browser by pointing the browser to http://localhost:8001.

**High availability for Eureka**

In the previous example, there was only one Eureka server in standalone mode. This is not good enough for a real production system.

The Eureka client connects to the server, fetches registry information, and stores it locally in a cache. The client always works with this local cache. The Eureka client checks the server periodically for any state changes. In the case of a state change, it downloads the changes from the server, and updates the cache. If the Eureka server is not reachable, then the Eureka clients can still work with the last-known state of the servers based on the data available in the client cache. However, this could lead to stale state issues quickly.

This section will explore the high availability for the Eureka server. The high availability architecture is shown in the following diagram:



The Eureka server is built with a peer-to-peer data synchronization mechanism. The runtime state information is not stored in a database, but managed using an in-memory cache. The high availability implementation favors availability and partition tolerance in the CAP theorem, leaving out consistency. Since the Eureka server instances are synchronized with each other using an asynchronous mechanism, the states may not always match between server instances. The peer-to-peer synchronization is done by pointing serviceUrls to each other. If there is more than one Eureka server, each one has to be connected to at least one of the peer servers. Since the state is replicated across all peers, Eureka clients can connect to any one of the available Eureka servers.

The best way to achieve high availability for Eureka is to cluster multiple Eureka servers, and run them behind a load balancer or a local DNS. The clients always connect to the server using the DNS/load balancer. At runtime, the load balancer takes care of selecting the appropriate servers. This load balancer address will be provided to the Eureka clients.

This section will showcase how to run two Eureka servers in a cluster for high availability. For this, define two property files: eureka-server1 and eureka-server2. These are peer servers; if one fails, the other one will take over. Each of these servers will also act as a client for the other so that they can sync their states. Two property files are defined in the following snippet. Upload and commit these properties to the Git repository.

The client URLs point to each other, forming a peer network as shown in the following configuration:

**eureka-server1.properties**

eureka.client.serviceUrl.defaultZone:http://localhost:8762/eureka/

eureka.client.registerWithEureka:false

eureka.client.fetchRegistry:false

**eureka-server2.properties**

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

eureka.client.registerWithEureka:false

eureka.client.fetchRegistry:false

Update the bootstrap.properties file of Eureka, and change the application name to eureka. Since we are using two profiles, based on the active profile supplied at startup, the Config server will look for either eureka-server1 or eureka-server2:

spring.application.name=eureka

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

Start two instances of the Eureka servers, server1 on 8761 and server2 on 8762:

**java -jar –Dserver.port=8761 -Dspring.profiles.active=server1 demo-0.0.1-SNAPSHOT.jar**

**java -jar –Dserver.port=8762 -Dspring.profiles.active=server2 demo-0.0.1-SNAPSHOT.jar**

All our services still point to the first server, server1. Open both the browser windows: http://localhost:8761 and http://localhost:8762.

Start all microservices. The one which opened 8761 will immediately reflect the changes, whereas the other one will take 30 seconds for reflecting the states. Since both the servers are in a cluster, the state is synchronized between these two servers. If we keep these servers behind a load balancer/DNS, then the client will always connect to one of the available servers.

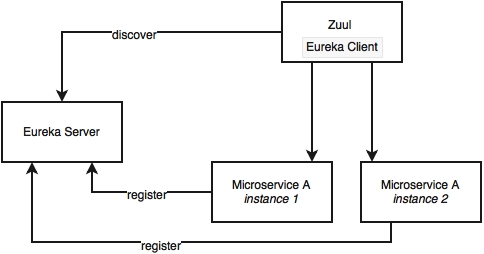
After completing this exercise, switch back to the standalone mode for the remaining exercises.

**Zuul proxy as the API gateway**

In most microservice implementations, internal microservice endpoints are not exposed outside. They are kept as private services. A set of public services will be exposed to the clients using an API gateway. There are many reasons to do this:

* Only a selected set of microservices are required by the clients.
* If there are client-specific policies to be applied, it is easy to apply them in a single place rather than in multiple places. An example of such a scenario is the cross-origin access policy.
* It is hard to implement client-specific transformations at the service endpoint.
* If there is data aggregation required, especially to avoid multiple client calls in a bandwidth-restricted environment, then a gateway is required in the middle.

Zuul is a simple gateway service or edge service that suits these situations well. Zuul also comes from the Netflix family of microservice products. Unlike many enterprise API gateway products, Zuul provides complete control for the developers to configure or program based on specific requirements:



The Zuul proxy internally uses the Eureka server for service discovery, and Ribbon for load balancing between service instances.

The Zuul proxy is also capable of routing, monitoring, managing resiliency, security, and so on. In simple terms, we can consider Zuul a reverse proxy service. With Zuul, we can even change the behaviors of the underlying services by overriding them at the API layer.

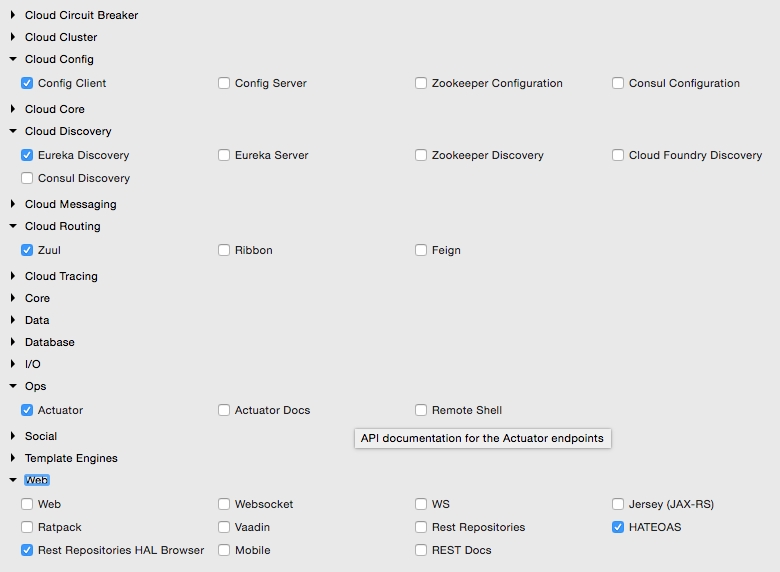
**Setting up Zuul**

Unlike the Eureka server and the Config server, in typical deployments, Zuul is specific to a microservice. However, there are deployments in which one API gateway covers many microservices. In this case, we are going to add Zuul for each of our microservices: Search, Booking, Fare, and Check-in:

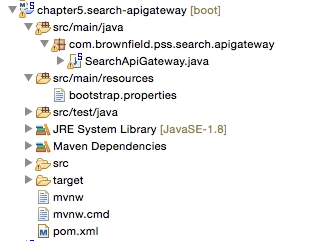
**Note**

The full source code of this section is available under the chapter5.\*-apigateway project in the code files.

1. Convert the microservices one by one. Start with Search API Gateway. Create a new Spring Starter project, and select **Zuul**, **Config Client**, **Actuator**, and **Eureka Discovery**:



The project structure for search-apigateway is shown in the following diagram:



1. The next step is to integrate the API gateway with Eureka and the Config server. Create a search-apigateway.property file with the contents given next, and commit to the Git repository.

This configuration also sets a rule on how to forward traffic. In this case, any request coming on the /api endpoint of the API gateway should be sent to search-service:

spring.application.name=search-apigateway

zuul.routes.search-apigateway.serviceId=search-service

zuul.routes.search-apigateway.path=/api/\*\*

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

search-service is the service ID of the Search service, and it will be resolved using the Eureka server.

1. Update the bootstrap.properties file of search-apigateway as follows. There is nothing new in this configuration—a name to the service, the port, and the Config server URL:
2. spring.application.name=search-apigateway
3. server.port=8095

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

1. Edit Application.java. In this case, the package name and the class name are also changed to com.brownfield.pss.search.apigateway and SearchApiGateway respectively. Also add @EnableZuulProxy to tell Spring Boot that this is a Zuul proxy:
2. @EnableZuulProxy
3. @EnableDiscoveryClient
4. @SpringBootApplication

public class SearchApiGateway {

1. Run this as a Spring Boot app. Before that, ensure that the Config server, the Eureka server, and the Search microservice are running.
2. Change the website project's CommandLineRunner as well as BrownFieldSiteController to make use of the API gateway:

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

In this case, the Zuul proxy acts as a reverse proxy which proxies all microservice endpoints to consumers. In the preceding example, the Zuul proxy does not add much value, as we just pass through the incoming requests to the corresponding backend service.

Zuul is particularly useful when we have one or more requirements like the following:

* Enforcing authentication and other security policies at the gateway instead of doing that on every microservice endpoint. The gateway can handle security policies, token handling, and so on before passing the request to the relevant services behind. It can also do basic rejections based on some business policies such as blocking requests coming from certain black-listed users.
* Business insights and monitoring can be implemented at the gateway level. Collect real-time statistical data, and push it to an external system for analysis. This will be handy as we can do this at one place rather than applying it across many microservices.
* API gateways are useful in scenarios where dynamic routing is required based on fine-grained controls. For example, send requests to different service instances based on business specific values such as "origin country". Another example is all requests coming from a region to be sent to one group of service instances. Yet another example is all requests requesting for a particular product have to be routed to a group of service instances.
* Handling the load shredding and throttling requirements is another scenario where API gateways are useful. This is when we have to control load based on set thresholds such as number of requests in a day. For example, control requests coming from a low-value third party online channel.
* The Zuul gateway is useful for fine-grained load balancing scenarios. The Zuul, Eureka client, and Ribbon together provide fine-grained controls over the load balancing requirements. Since the Zuul implementation is nothing but another Spring Boot application, the developer has full control over the load balancing.
* The Zuul gateway is also useful in scenarios where data aggregation requirements are in place. If the consumer wants higher level coarse-grained services, then the gateway can internally aggregate data by calling more than one service on behalf of the client. This is particularly applicable when the clients are working in low bandwidth environments.

Zuul also provides a number of filters. These filters are classified as pre filters, routing filters, post filters, and error filters. As the names indicate, these are applied at different stages of the life cycle of a service call. Zuul also provides an option for developers to write custom filters. In order to write a custom filter, extend from the abstract ZuulFilter, and implement the following methods:

public class CustomZuulFilter extends ZuulFilter{

public Object run(){}

public boolean shouldFilter(){}

public int filterOrder(){}

public String filterType(){}

Once a custom filter is implemented, add that class to the main context. In our example case, add this to the SearchApiGateway class as follows:

@Bean

public CustomZuulFilter customFilter() {

return new CustomZuulFilter();

}

As mentioned earlier, the Zuul proxy is a Spring Boot service. We can customize the gateway programmatically in the way we want. As shown in the following code, we can add custom endpoints to the gateway, which, in turn, can call the backend services:

@RestController

class SearchAPIGatewayController {

@RequestMapping("/")

String greet(HttpServletRequest req){

return "<H1>Search Gateway Powered By Zuul</H1>";

}

}

In the preceding case, it just adds a new endpoint, and returns a value from the gateway. We can further use @Loadbalanced RestTemplate to call a backend service. Since we have full control, we can do transformations, data aggregation, and so on. We can also use the Eureka APIs to get the server list, and implement completely independent load-balancing or traffic-shaping mechanisms instead of the out-of-the-box load balancing features provided by Ribbon.

**High availability of Zuul**

Zuul is just a stateless service with an HTTP endpoint, hence, we can have as many Zuul instances as we need. There is no affinity or stickiness required. However, the availability of Zuul is extremely critical as all traffic from the consumer to the provider flows through the Zuul proxy. However, the elastic scaling requirements are not as critical as the backend microservices where all the heavy lifting happens.

The high availability architecture of Zuul is determined by the scenario in which we are using Zuul. The typical usage scenarios are:

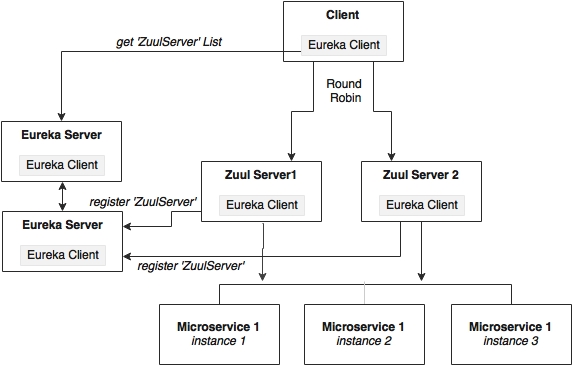
* When a client-side JavaScript MVC such as AngularJS accesses Zuul services from a remote browser.
* Another microservice or non-microservice accesses services via Zuul

In some cases, the client may not have the capabilities to use the Eureka client libraries, for example, a legacy application written on PL/SQL. In some cases, organization policies do not allow Internet clients to handle client-side load balancing. In the case of browser-based clients, there are third-party Eureka JavaScript libraries available.

It all boils down to whether the client is using Eureka client libraries or not. Based on this, there are two ways we can set up Zuul for high availability.

**High availability of Zuul when the client is also a Eureka client**

In this case, since the client is also another Eureka client, Zuul can be configured just like other microservices. Zuul registers itself to Eureka with a service ID. The clients then use Eureka and the service ID to resolve Zuul instances:



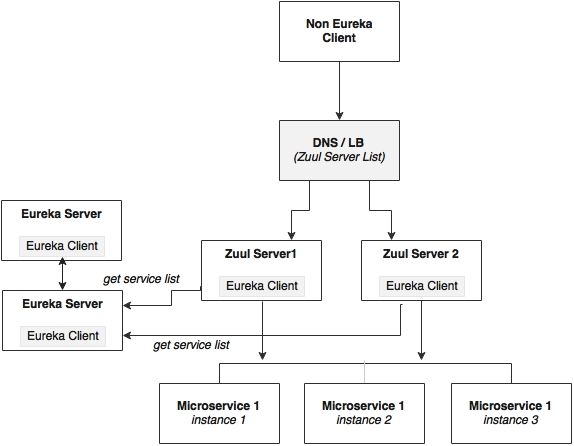
As shown in the preceding diagram, Zuul services register themselves with Eureka with a service ID, search-apigateway in our case. The Eureka client asks for the server list with the ID search-apigateway. The Eureka server returns the list of servers based on the current Zuul topology. The Eureka client, based on this list picks up one of the servers, and initiates the call.

As we saw earlier, the client uses the service ID to resolve the Zuul instance. In the following case, search-apigateway is the Zuul instance ID registered with Eureka:

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

**High availability when the client is not a Eureka client**

In this case, the client is not capable of handling load balancing by using the Eureka server. As shown in the following diagram, the client sends the request to a load balancer, which in turn identifies the right Zuul service instance. The Zuul instance, in this case, will be running behind a load balancer such as HAProxy or a hardware load balancer like NetScaler:



The microservices will still be load balanced by Zuul using the Eureka server.

**Completing Zuul for all other services**

In order to complete this exercise, add API gateway projects (name them as \*-apigateway) for all our microservices. The following steps are required to achieve this task:

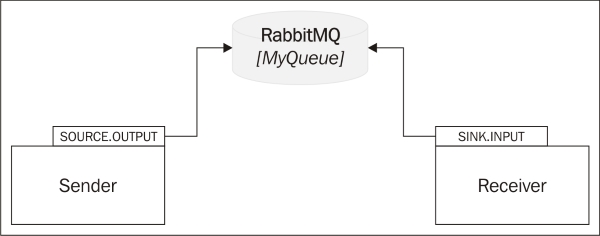
1. Create new property files per service, and check in to the Git repositories.
2. Change application.properties to bootstrap.properties, and add the required configurations.
3. Add @EnableZuulProxy to Application.java in each of the \*-apigateway projects.
4. Add @EnableDiscoveryClient in all the Application.java files under each of the \*-apigateway projects.
5. Optionally, change the package names and file names generated by default.

In the end, we will have the following API gateway projects:

* chapter5.fares-apigateway
* chapter5.search-apigateway
* chapter5.checkin-apigateway
* chapter5.book-apigateway

**Streams for reactive microservices**

Spring Cloud Stream provides an abstraction over the messaging infrastructure. The underlying messaging implementation can be RabbitMQ, Redis, or Kafka. Spring Cloud Stream provides a declarative approach for sending and receiving messages:

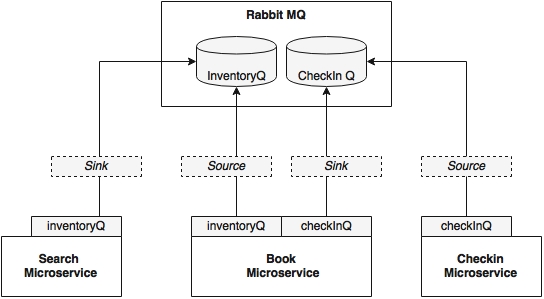


As shown in the preceding diagram, Cloud Stream works on the concept of a **source** and a sink. The source represents the sender perspective of the messaging, and sink represents the receiver perspective of the messaging.

In the example shown in the diagram, the sender defines a logical queue called Source.OUTPUT to which the sender sends messages. The receiver defines a logical queue called Sink.INPUT from which the receiver retrieves messages. The physical binding of OUTPUT to INPUT is managed through the configuration. In this case, both link to the same physical queue—MyQueue on RabbitMQ. So, while at one end, Source.OUTPUT points to MyQueue, on the other end, Sink.INPUT points to the same MyQueue.

Spring Cloud offers the flexibility to use multiple messaging providers in one application such as connecting an input stream from Kafka to a Redis output stream, without managing the complexities. Spring Cloud Stream is the basis for message-based integration. The Cloud Stream Modules subproject is another Spring Cloud library that provides many endpoint implementations.

As the next step, rebuild the inter-microservice messaging communication with the Cloud Streams. As shown in the next diagram, we will define a SearchSink connected to InventoryQ under the Search microservice. Booking will define a BookingSource for sending inventory change messages connected to InventoryQ. Similarly, Check-in defines a CheckinSource for sending the check-in messages. Booking defines a sink, BookingSink, for receiving messages, both bound to the CheckinQ queue on the RabbitMQ:



In this example, we will use RabbitMQ as the message broker:

1. Add the following Maven dependency to Booking, Search, and Check-in, as these are the three modules using messaging:
2. <dependency>
3. <groupId>org.springframework.cloud</groupId>
4. <artifactId>spring-cloud-starter-stream-rabbit</artifactId>

</dependency>

1. Add the following two properties to booking-service.properties. These properties bind the logical queue inventoryQ to physical inventoryQ, and the logical checkinQ to the physical checkinQ:
2. spring.cloud.stream.bindings.inventoryQ.destination=inventoryQ

spring.cloud.stream.bindings.checkInQ.destination=checkInQ

1. Add the following property to search-service.properties. This property binds the logical queue inventoryQ to the physical inventoryQ:

spring.cloud.stream.bindings.inventoryQ.destination=inventoryQ

1. Add the following property to checkin-service.properties. This property binds the logical queue checkinQ to the physical checkinQ:

spring.cloud.stream.bindings.checkInQ.destination=checkInQ

1. Commit all files to the Git repository.
2. The next step is to edit the code. The Search microservice consumes a message from the Booking microservice. In this case, Booking is the source and Search is the sink.

Add @EnableBinding to the Sender class of the Booking service. This enables the Cloud Stream to work on autoconfigurations based on the message broker library available in the class path. In our case, it is RabbitMQ. The parameter BookingSource defines the logical channels to be used for this configuration:

@EnableBinding(BookingSource.class)

public class Sender {

1. In this case, BookingSource defines a message channel called inventoryQ, which is physically bound to RabbitMQ's inventoryQ, as configured in the configuration. BookingSource uses an annotation, @Output, to indicate that this is of the output type—a message that is outgoing from a module. This information will be used for autoconfiguration of the message channel:
2. interface BookingSource {
3. public static String InventoryQ="inventoryQ";
4. @Output("inventoryQ")
5. public MessageChannel inventoryQ();

}

1. Instead of defining a custom class, we can also use the default Source class that comes with Spring Cloud Stream if the service has only one source and sink:
2. public interface Source {
3. @Output("output")
4. MessageChannel output();

}

1. Define a message channel in the sender, based on BookingSource. The following code will inject an output message channel with the name inventory, which is already configured in BookingSource:
2. @Output (BookingSource.InventoryQ)
3. @Autowired

private MessageChannel;

1. Reimplement the send message method in BookingSender:
2. public void send(Object message){
3. messageChannel.
4. send(MessageBuilder.withPayload(message).
5. build());

}

1. Now add the following to the SearchReceiver class the same way we did for the Booking service:
2. @EnableBinding(SearchSink.class)

public class Receiver {

1. In this case, the SearchSink interface will look like the following. This will define the logical sink queue it is connected with. The message channel in this case is defined as @Input to indicate that this message channel is to accept messages:
2. interface SearchSink {
3. public static String INVENTORYQ="inventoryQ";
4. @Input("inventoryQ")
5. public MessageChannel inventoryQ();

}

1. Amend the Search service to accept this message:
2. @ServiceActivator(inputChannel = SearchSink.INVENTORYQ)
3. public void accept(Map<String,Object> fare){
4. searchComponent.updateInventory((String)fare.
5. get("FLIGHT\_NUMBER"),(String)fare.
6. get("FLIGHT\_DATE"),(int)fare.
7. get("NEW\_INVENTORY"));

}

1. We will still need the RabbitMQ configurations that we have in our configuration files to connect to the message broker:
2. spring.rabbitmq.host=localhost
3. spring.rabbitmq.port=5672
4. spring.rabbitmq.username=guest
5. spring.rabbitmq.password=guest

server.port=8090

1. Run all services, and run the website project. If everything is fine, the website project successfully executes the Search, Booking, and Check-in functions. The same can also be tested using the browser by pointing to http://localhost:8001.