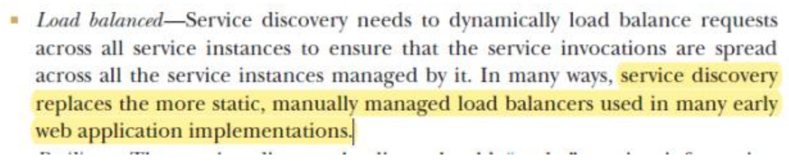
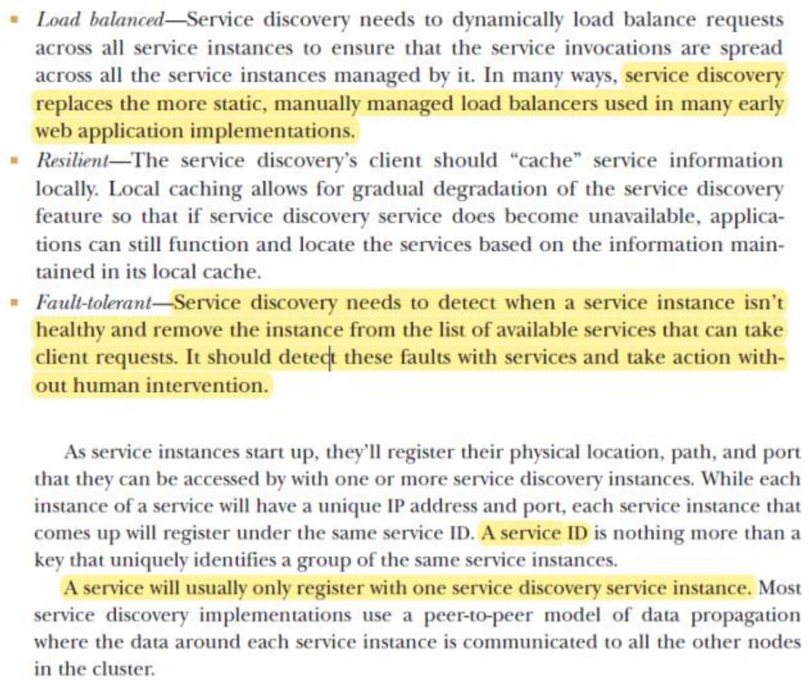
Friday, September 6, 2019 4:11 PM

- service discovery agent
- *spring-cloud-starter-eureka* will include *netflix ribbon*
- Netflix Ribbon will cache the service instance details.
- Individual services registering will take up to 30 seconds to show up in the Eureka service because Eureka requires three con secutive heartbeat pings from the service spaced 10 seconds apart before it will say the service is ready for use.

spring-cloud-starter-eureka-server
spring-cloud-starter-eureka



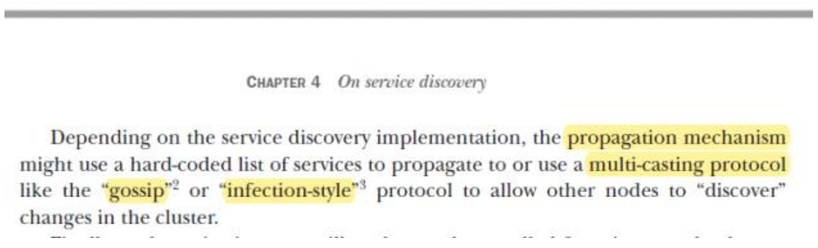
- *Load balanced*—Service discovery needs to dynamically load balance requests across all service instances to ensure that the service invocations are spread across all the service instances managed by it. In many ways, service discovery replaces the more static, manually managed load balancers used in many early web application implementations.



- *Load balanced*—Service discovery needs to dynamically load balance requests across all service instances to ensure that the service invocations are spread across all the service instances managed by it. In many ways, service discovery replaces the more static, manually managed load balancers used in many early web application implementations.
- *Resilient*—The service discovery's client should "cache" service information locally. Local caching allows for gradual degradation of the service discovery feature so that if service discovery service does become unavailable, applications can still function and locate the services based on the information maintained in its local cache.
- *Fault-tolerant*—Service discovery needs to detect when a service instance isn't healthy and remove the instance from the list of available services that can take client requests. It should detect these faults with services and take action without human intervention.

As service instances start up, they'll register their physical location, path, and port that they can be accessed by with one or more service discovery instances. While each instance of a service will have a unique IP address and port, each service instance that comes up will register under the same service ID. A service ID is nothing more than a key that uniquely identifies a group of the same service instances.

A service will usually only register with one service discovery service instance. Most service discovery implementations use a peer-to-peer model of data propagation where the data around each service instance is communicated to all the other nodes in the cluster.



Depending on the service discovery implementation, the propagation mechanism might use a hard-coded list of services to propagate to or use a multi-casting protocol like the "gossip"[2] or "infection-style"[3] protocol to allow other nodes to "discover" changes in the cluster.

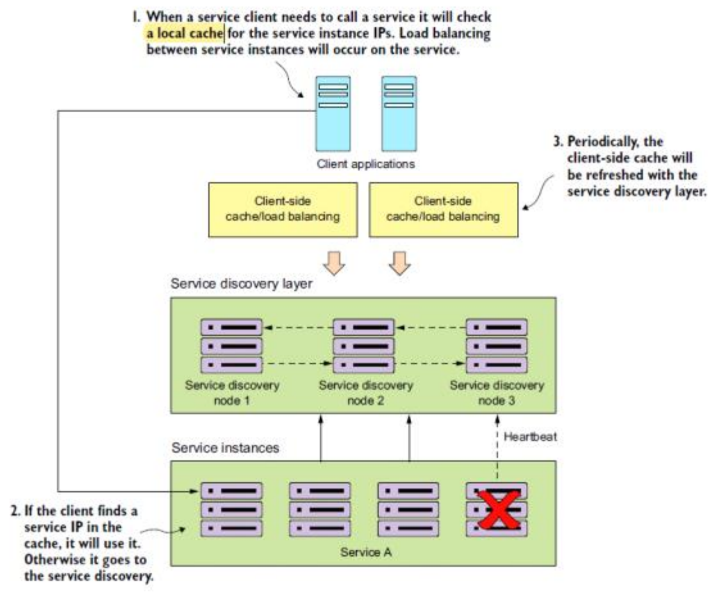

1. When a service client needs to call a service it will check a local cache for the service instance IPs. Load balancing between service instances will occur on the service.
Client applications
3. Periodically, the client-side cache will be refreshed with the service discovery layer.
Client-side cache/load balancing
Client-side cache/load balancing
Service discovery layer
Service discovery node 1
Service discovery node 2
Service discovery node 3
Heartbeat
Service instances
Service A
2. If the client finds a service IP in the cache, it will use it. Otherwise it goes to the service discovery.

Figure 4.3 Client-side load balancing caches the location of the services so that the service client doesn't have to contact service discovery on every call.

Once again, the Spring Cloud project makes this type of setup trivial to undertake. You'll use Spring Cloud and Netflix's Eureka service discovery engine to implement your service discovery pattern. For the client-side load balancing you'll use Spring Cloud and Netflix's Ribbon libraries.

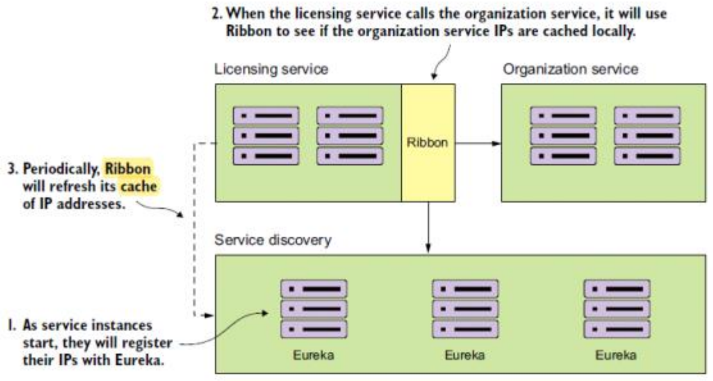

2. When the licensing service calls the organization service, it will use Ribbon to see if the organization service IPs are cached locally.
Licensing service
Organization service
Ribbon
3. Periodically, Ribbon will refresh its cache of IP addresses.
Service discovery
1. As service instances start, they will register their IPs with Eureka.
Eureka
Eureka
Eureka

Figure 4.4 By implementing client-side caching and Eureka with the licensing and organization services, you can lessen the load on the Eureka servers and improve client stability if Eureka becomes unavailable.

1. As the services are bootstrapping, the licensing and organization services will also register themselves with the Eureka Service. This registration process will tell Eureka the physical location and port number of each service instance along with a service ID for the service being started.
2. When the licensing service calls to the organization service, it will use the Netflix Ribbon library to provide client-slide load balancing. Ribbon will contact the Eureka service to retrieve service location information and then cache it locally.
3. Periodically, the Netflix Ribbon library will ping the Eureka service and refresh its local cache of service locations.

```
<!--Not showing the maven definitions for using Spring Cloud Parent-->
  <dependencies>
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-eureka-server</artifactId>
    </dependency>
  </dependencies>

Rest of pom.xml removed for conciseness
....
</project>
```

Tells your maven build to include the Eureka libraries (which will include Ribbon)

Listing 4.2 Setting up your Eureka configuration in the application.yml file

```
server:
 port: 8761

eureka:
 client:
  registerWithEureka: false


fetchRegistry: false
server:
  waitTimeInMsWhenSyncEmpty: 5
```

Port Eureka Server is going to listen on

Don't register with Eureka service.

Don't cache registry information locally.

Initial time to wait before server takes requests

You'll notice that the last attribute, `eureka.server.waitTimeInMsWhenSyncEmpty`, is commented out. When you're testing your service locally you should uncomment this line because Eureka won't immediately advertise any services that register with it. It will wait five minutes by default to give all of the services a chance to register with it before advertising them. Uncommenting this line for local testing will help speed up the amount of time it will take for the Eureka service to start and show services registered with it.

Individual services registering will take up to 30 seconds to show up in the Eureka service because Eureka requires three consecutive heartbeat pings from the service spaced 10 seconds apart before it will say the service is ready for use. Keep this in mind as you're deploying and testing your own services.

Client

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

Includes the Eureka libraries so that the service can register with Eureka

Listing 4.4 Modifying your organization service's application.yml to talk to Eureka

```
spring:
 application:
  name: organizationservice
 profiles:
  active:
   default
 cloud:
  config:
   enabled: true
```

Logical name of the service that will be registered with Eureka

```
eureka:
 instance:
  preferIpAddress: true
 client:
  registerWithEureka: true
  fetchRegistry: true
  serviceUrl:
    defaultZone: http://localhost:8761/eureka/
```

Register the IP of the service rather than the server name.

Register the service with Eureka.

Pull down a local copy of the registry.

Location of the Eureka Service

Every service registered with Eureka will have two components associated with it: the application ID and the instance ID. The application ID is used to represent a group service instance. In a Spring-Boot-based microservice, the application ID will always be the value set by the `spring.application.name` property. For your organization ser-

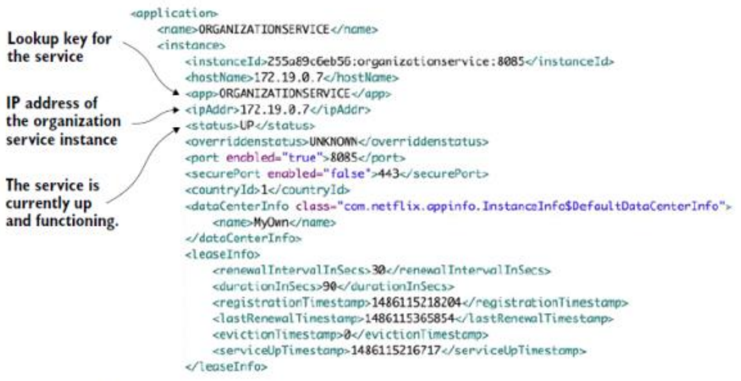
The `eureka.client.registerWithEureka` attribute is the trigger to tell the organization service to register itself with Eureka. The `eureka.client.fetchRegistry` attribute is used to tell the Spring Eureka Client to fetch a local copy of the registry. Setting this attribute to true will cache the registry locally instead of calling the Eureka service with every lookup. Every 30 seconds, the client software will re-contact the Eureka service for any changes to the registry.

Eureka high availability

Setting up multiple URL services isn't enough for high availability. The `eureka.serviceUrl.defaultZone` attribute only provides a list of Eureka services for the client to communicate with. You also need to set up the Eureka services to replicate the contents of their registry with each other.

A group of Eureka registries communicate with each other using a peer-to-peer communication model where each Eureka service has to be configured to know about the other nodes in the cluster. Setting up a Eureka cluster is outside of the scope of this book. If you're interested in setting up a Eureka cluster, please visit the Spring Cloud project's website for further information.[a]

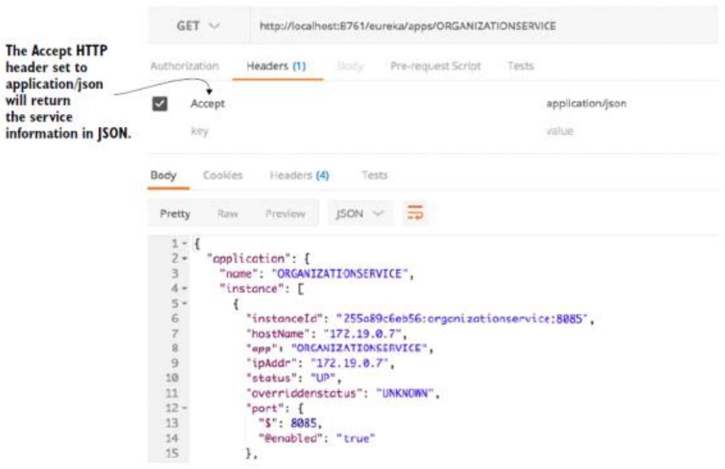
For instance, to see the organization service in the registry you can call `http://localhost:8761/eureka/apps/organizationservice`.


Lookup key for the service
IP address of the organization service instance
The service is currently up and functioning.

```
<application>
    <name>ORGANIZATIONSERVICE</name>
    <instance>
        <instanceId>255a89c6eb56:organizationservice:8085</instanceId>
        <hostName>172.19.0.7</hostName>
        <app>ORGANIZATIONSERVICE</app>
        <ipAddr>172.19.0.7</ipAddr>
        <status>UP</status>
        <overriddenstatus>UNKNOWN</overriddenstatus>
        <port enabled="true">8085</port>
        <securePort enabled="false">443</securePort>
        <countryId>1</countryId>
        <dataCenterInfo class="com.netflix.appinfo.InstanceInfo$DefaultDataCenterInfo">
            <name>MyOwn</name>
        </dataCenterInfo>
        <leaseInfo>
            <renewalIntervalInSecs>30</renewalIntervalInSecs>
            <durationInSecs>90</durationInSecs>
            <registrationTimestamp>1486115218204</registrationTimestamp>
            <lastRenewalTimestamp>1486115365854</lastRenewalTimestamp>
            <evictionTimestamp>0</evictionTimestamp>
            <serviceUpTimestamp>1486115216717</serviceUpTimestamp>
        </leaseInfo>
```

Figure 4.5 Calling the Eureka REST API to see the organization will show the IP address of the service instances registered in Eureka, along with the service status.

The default format returned by the Eureka service is XML. Eureka can also return the data in figure 4.5 as a JSON payload, but you have to set the `Accept` HTTP header to be `application/json`. An example of the JSON payload is shown in figure 4.6.



GET http://localhost:8761/eureka/apps/ORGANIZATIONSERVICE

✓ Accept	application/json

```
{
  "application": {
    "name": "ORGANIZATIONSERVICE",
    "instance": [
      {
        "instanceId": "255a89c6eb56:organizationservice:8085",
        "hostName": "172.19.0.7",
        "app": "ORGANIZATIONSERVICE",
        "ipAddr": "172.19.0.7",
        "status": "UP",
        "overriddenstatus": "UNKNOWN",
        "port": {
          "$": 8085,
          "@enabled": "true"
        },
```

Figure 4.6 Calling the Eureka REST API with the results being JSON

On Eureka and service startups: don't be impatient

When a service registers with Eureka, Eureka will wait for three successive health checks over the course of 30 seconds before the service becomes available via a Eureka. This warm-up period throws developers off because they think that Eureka hasn't registered their services if they try to call their service immediately after the service has been launched. This is evident in our code examples running in the Docker environment, because the Eureka service and the application services (licensing and organization services) all start up at the same time. Be aware that after starting the application, you may receive 404 errors about services not being found, even though the service itself has started. Wait 30 seconds before trying to call your services.

In a production environment, your Eureka services will already be running and if you're deploying an existing service, the old services will still be in place to take requests.

For our purposes, we're going to look at three different Spring/Netflix client libraries in which a service consumer can interact with Ribbon. These libraries will move from the lowest level of abstraction for interacting with Ribbon to the highest. The libraries we'll explore include

- Spring Discovery client
- Spring Discovery client enabled RestTemplate
- Netflix Feign client

1. Using Spring Discovery Client

Listing 4.8 Using the DiscoveryClient to look up information

```
/*Packages and imports removed for conciseness*/

@Component
public class OrganizationDiscoveryClient {

  @Autowired
  private DiscoveryClient discoveryClient;

  public Organization getOrganization(String organizationId) {
    RestTemplate restTemplate = new RestTemplate();
    List<ServiceInstance> instances =

      discoveryClient.getInstances("organizationservice");

    if (instances.size()==0) return null;
    String serviceUri = String.format("%s/v1/organizations/%s",

     instances.get(0).getUri().toString(),

     organizationId);

    ResponseEntity< Organization > restExchange =
        restTemplate.exchange(
            serviceUri,
            HttpMethod.GET,
            null, Organization.class, organizationId);

    return restExchange.getBody();
  }
}
```

DiscoveryClient is auto-injected into the class.

Gets a list of all the instances of organization services

Retrieves the service endpoint we are going to call

Uses a standard Spring REST Template class to call the service

a standard Spring RestTemplate to call your organization service and retrieve data.

The DiscoveryClient and real life

I'm walking through the DiscoveryClient to be completed in our discussion of building service consumers with Ribbon. The reality is that you should only use the Discovery-Client directly when your service needs to query Ribbon to understand what services and service instances are registered with it. There are several problems with this code including the following:

You aren't taking advantage of Ribbon's client side load-balancing—By calling the DiscoveryClient directly, you get back a list of services, but it becomes your responsibility to choose which service instances returned you're going to invoke.

You're doing too much work—Right now, you have to build the URL that's going to be used to call your service. It's a small thing, but every piece of code that you can avoid writing is one less piece of code that you have to debug.

Observant Spring developers might have noticed that you're directly instantiating the RestTemplate class in the code. This is antithetical to normal Spring REST invocations, as normally you'd have the Spring Framework inject the RestTemplate the class using it via the `@Autowired` annotation.

You instantiated the RestTemplate class in listing 4.8 because once you've enabled the Spring DiscoveryClient in the application class via the `@EnableDiscoveryClient` annotation, all RestTemplates managed by the Spring framework will have a Ribbon-enabled interceptor injected into them that will change how URLs are created with the RestTemplate class. Directly instantiating the RestTemplate class allows you to avoid this behavior.

In summary, there are better mechanisms for calling a Ribbon-backed service.

2. Using Spring Cloud and Ribbon-backed RestTemplate

4.5.2 Invoking services with Ribbon-aware Spring RestTemplate

Next, we're going to see an example of how to use a RestTemplate that's Ribbon-aware. This is one of the more common mechanisms for interacting with Ribbon via Spring. To use a Ribbon-aware RestTemplate class, you need to define a Rest-Template bean construction method with a Spring Cloud annotation called @Load-Balanced. For the licensing service, the method that will be used to create the RestTemplate bean can be found in src/main/java/com/thoughtmechanix/licenses/Application.java.

Listing 4.9 Annotating and defining a RestTemplate construction method

```
package com.thoughtmechanix.licenses;

//…Most of import statements have been removed for consiceness
import org.springframework.cloud.client.loadbalancer.LoadBalanced;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class Application {

  @LoadBalanced
  @Bean
  public RestTemplate getRestTemplate(){
      return new RestTemplate();
  }

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

Because we're using multiple client types in the examples, I'm including them in the code. However, the @EnableDiscoveryClient and @EnableFeignClients application aren't needed when using the Ribbon backed RestTemplate and can be removed.

The @LoadBalanced annotation tells Spring Cloud to create a Ribbon backed RestTemplate class.

NOTE In early releases of Spring Cloud, the RestTemplate class was automatically backed by Ribbon. It was the default behavior. However, since Spring Cloud Release Angel, the RestTemplate in Spring Cloud is no longer backed by Ribbon. If you want to use Ribbon with the RestTemplate, you must explicitly annotate it using the @LoadBalanced annotation.

Listing 4.10 Using a Ribbon-backed RestTemplate to call a service

```
/*Package and import definitions left off for conciseness*/
@Component
```

```
public class OrganizationRestTemplateClient {
  @Autowired
  RestTemplate restTemplate;

  public Organization getOrganization(String organizationId){
    ResponseEntity<Organization> restExchange =
        restTemplate.exchange(
            "http://organizationservice/v1/organizations/{organizationId}",
            HttpMethod.GET,
            null, Organization.class, organizationId);

    return restExchange.getBody();
  }
}
```

When using a Ribbon-back RestTemplate, you build the target URL with the Eureka service ID.

This code should look somewhat similar to the previous example, except for two key differences. First, the Spring Cloud DiscoveryClient is nowhere in sight. Second, the URL being used in the restTemplate.exchange() call should look odd to you:

```
restTemplate.exchange(

  "http://organizationservice/v1/organizations/{organizationId}",
   HttpMethod.GET,
 null, Organization.class, organizationId);
```

The server name in the URL matches the application ID of the organizationservice key

3. Using Spring Cloud and Netflix's Feign client

Listing 4.11 Enabling the Spring Cloud/Netflix Feign client in the licensing service

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class Application {
  public static void main(String[] args) {
    SpringApplication.run(Application.class, args);
  }
}
```

Because we're only using the FeignClient, in your own code you can remove the @EnableDiscoveryClient annotation.

The @EnableFeignClients annotation is needed to use the FeignClient in your code.

Now that you've enabled the Feign client for use in your licensing service, let's look at a Feign client interface definition that can be used to call an endpoint on the organization service. The following listing shows an example. The code in this listing can be found in the `src/main/java/com/thoughtmechanix/licenses/clients/OrganizationFeignClient.java` class.

Listing 4.12 Defining a Feign interface for calling the organization service

```
/*Package and import left off for conciseness*/
@FeignClient("organizationservice")
public interface OrganizationFeignClient {
@RequestMapping(
      method= RequestMethod.GET,
      value="/v1/organizations/{organizationId}",
      consumes="application/json")
 Organization getOrganization(
  @PathVariable("organizationId") String organizationId);
}
```

Identify your service to Feign using the FeignClient Annotation.

The path and action to your endpoint is defined using the @RequestMapping annotation.

The parameters passed into the endpoint are defined using the @PathVariable endpoint.

You start the Feign example by using the `@FeignClient` annotation and passing it the name of the application id of the service you want the interface to represent. Next

On error handling

When you use the standard Spring `RestTemplate` class, all service calls' HTTP status codes will be returned via the `ResponseEntity` class's `getStatusCode()` method. With the Feign Client, any HTTP 4xx – 5xx status codes returned by the service being called will be mapped to a `FeignException`. The `FeignException` will contain a JSON body that can be parsed for the specific error message.

Feign does provide you the ability to write an error decoder class that will map the error back to a custom Exception class. Writing this decoder is outside the scope of this book, but you can find examples of this in the Feign GitHub repository at (https://github.com/Netflix/feign/wiki/Custom-error-handling).