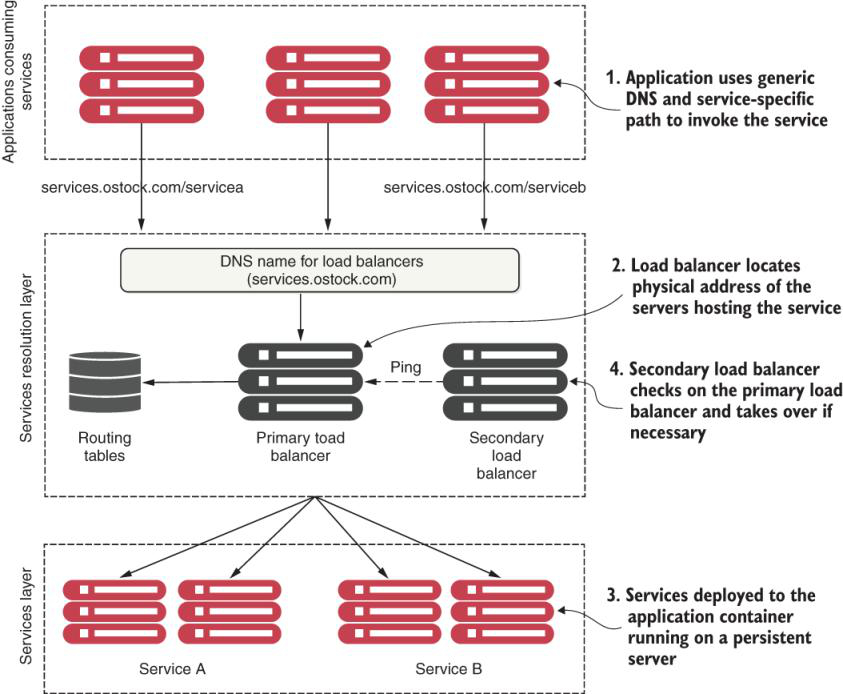
**How does traditional service resolution model work and what are challenges it puts forward?**

A traditional service location resolution model uses a DNS and a load balancer.



While this type of model works well with applications running inside the four walls of a corporate data center, and with a relatively small number of services running on a group of static servers, it doesn’t work well for cloud-based microservice applications. The reasons for this include the following:

* While the load balancer can be made highly available, it’s a single point of failure for your entire infrastructure. If the load balancer goes down, every application relying on it goes down too. While you can make a load balancer highly available, load balancers tend to be centralized chokepoints within your application infrastructure.
* Centralizing your services into a single cluster of load balancers limits your ability to scale horizontally your load-balancing infrastructure across multiple servers. Many commercial load balancers are constrained by two things: their redundancy model and their licensing costs.

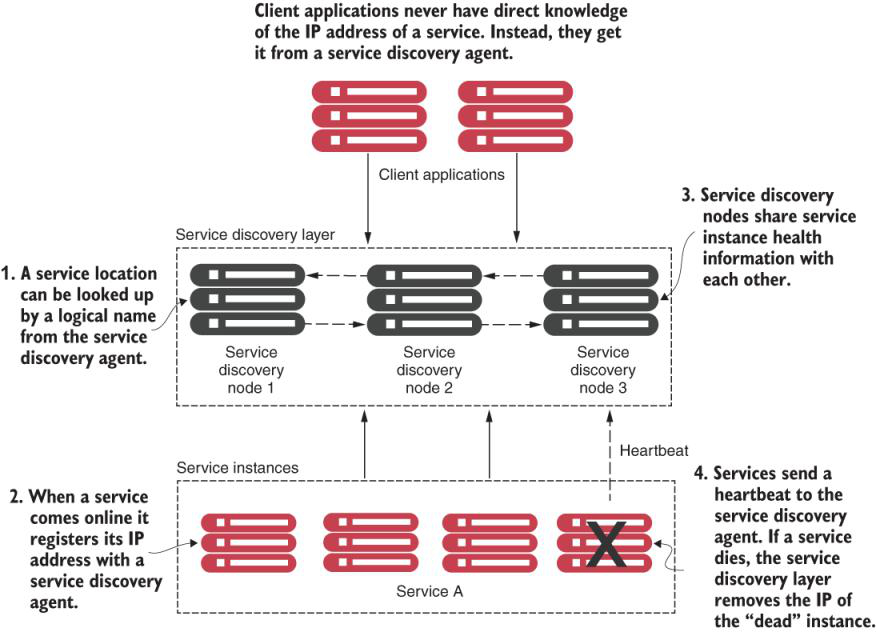
Most commercial load balancers use a hot-swap model for redundancy, so you only have a single server to handle the load, while the secondary load balancer is there only for failover in case the primary load balancer goes down. You are, in essence, constrained by your hardware. Commercial load balancers also have restrictive licensing models geared toward a fixed capacity rather than a more variable model.

* Most traditional load balancers are statically managed. They aren’t designed for fast registration and deregistration of services. Traditional load balancers use a centralized database to store the routes for rules, and the only way to add new routes is often through the vendor’s proprietary API.
* Because a load balancer acts as a proxy to the services, service consumer requests need to have them mapped to the physical services. This translation layer often adds another layer of complexity to your service infrastructure because the mapping rules for the service have to be defined and deployed by hand. Also, in a traditional load balancer scenario, the registration of new service instances is not done when a new service instance starts.

**What is main principle of service discovery?**

The principal objective of service discovery is to have an architecture where our services indicate where they are physically located instead of having to manually configure their location.

As service instances are added or removed, the service discovery nodes are updated and made available to process user requests.



**What are typical concepts of service discovery?**

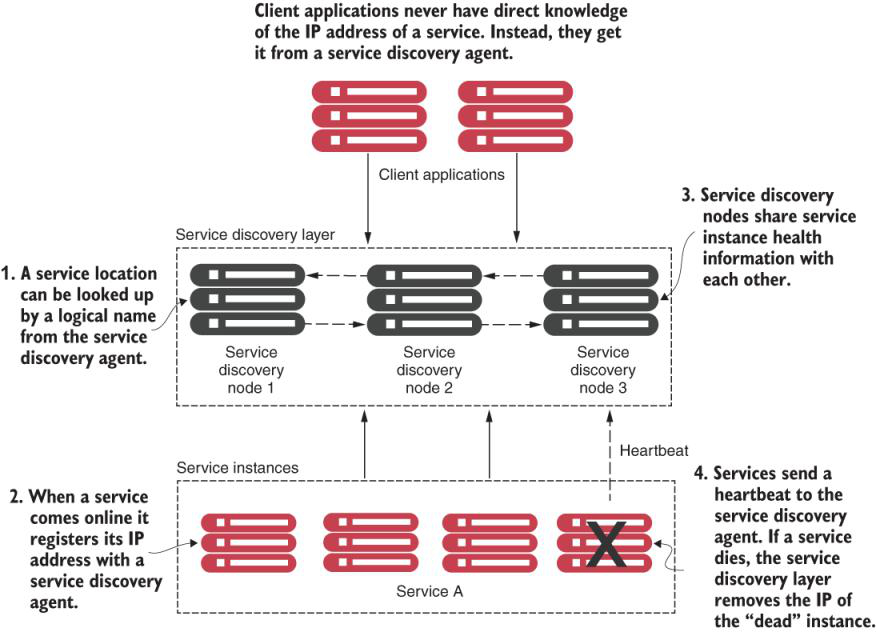
These general concepts are often shared across all service discovery implementations:

* Service registration—How a service registers with the service discovery agent
* Client lookup of service address—How a service client looks up service information
* Information sharing—How nodes share service information
* Health monitoring—How services communicate their health back to the service discovery agent.

Below figure shows the flow of the previous four bulleted points (service registration, service discovery lookup, information sharing, and health monitoring) and what typically occurs when we implement a service discovery pattern.

In the figure, one or more service discovery nodes have started. These service discovery instances usually don’t have a load balancer that sits in front of them.

As service instances start, they’ll register their physical location, path, and port that one or more service discovery instances can use to access the instances. While each instance of a service has a unique IP address and port, each service instance that comes up registers under the same service ID. A service ID is nothing more than a key that uniquely identifies a group of the same service instances.



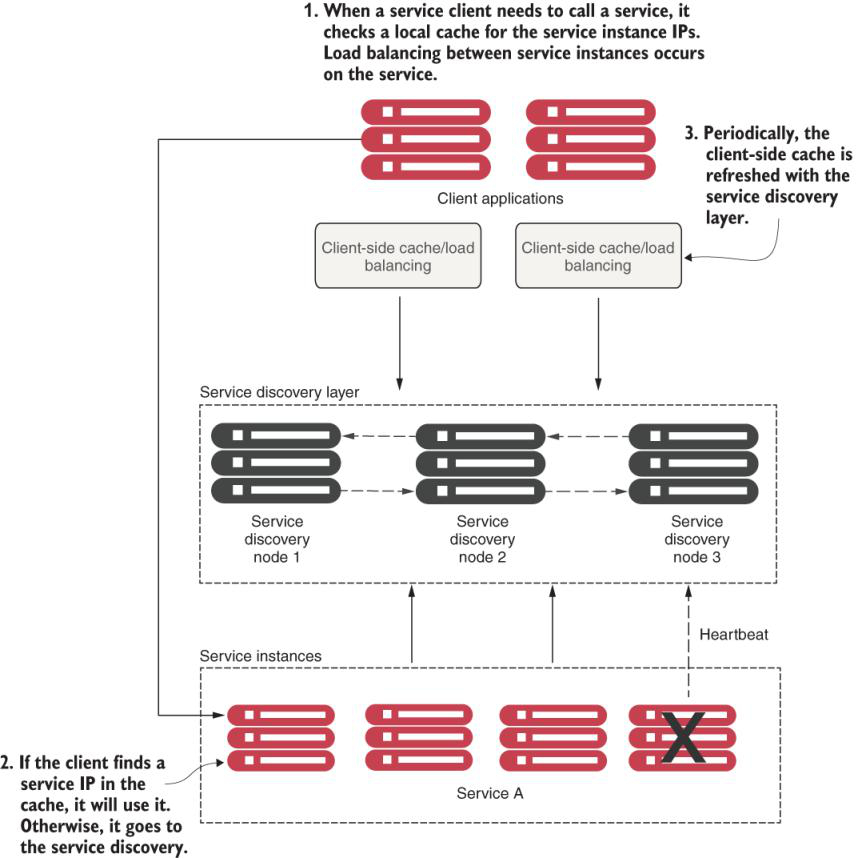
**What are different models for service clients to access services once services are registered with discovery service?**

Once a service is registered with a service discovery service, it’s ready to be used by an application or service that needs to make use of its capabilities. Different models exist for a client to discover a service.

1. As a first approach, the client relies solely on the service discovery engine to resolve service locations each time a service is called. With this approach, the service discovery engine is invoked each time a call to a registered microservice instance is made. Unfortunately, this approach is brittle because the service client is completely dependent on the service discovery engine to find and invoke a service.
2. A more robust approach uses what’s called client-side load balancing. This mechanism uses an algorithm like zone-specific or round-robin to invoke the instances of the calling services. When we say “round-robin algorithm load balancing,” we are referring to a way of distributing client requests across several servers. This consists of forwarding a client request to each of the servers in turn. An advantage of using the client-side load balancer with Eureka is that when a service instance goes down, it is removed from the registry. Once that is done, the client-side load balancer updates itself without manual intervention by establishing constant communication with the registry service.

**How does client-side load balancing work in context of Eureka discovery?**

Client-side load balancing caches the location of the services so that the service client doesn’t need to contact service discovery on every call.



In this model, when a consuming client needs to invoke a service

1. It contacts the discovery service for all the instances a service consumer (client) is asking for and then caches data locally on the service consumer’s machine.
2. Each time a client wants to call the service, the service consumer looks up the location information for the service from the cache. Usually, client-side caching will use a simple load-balancing algorithm like the round-robin load-balancing algorithm to ensure that service calls are spread across multiple service instances.
3. The client then periodically contacts the discovery service and refreshes its cache of service instances. The client cache is eventually consistent, but there’s always a risk that when the client contacts the service discovery instance for a refresh and calls are made, calls might be directed to a service instance that isn’t healthy.

If during the course of calling a service, the service call fails, the local service discovery cache is invalidated and the service discovery client will attempt to refresh its entries from the service discovery agent. Let’s now take the generic service discovery pattern and apply it to our O-stock problem domain.

**What is difference between service discovery and service discovery agent?**

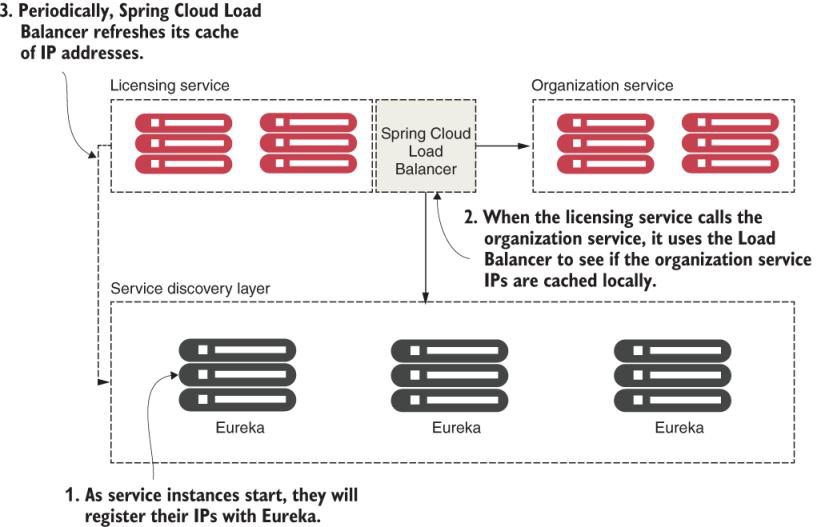
Service discovery is concept whereas service discovery agent is tool to implement service discovery concept.

**How does Service Discovery work Using Spring and Netflix Eureka?**

In this chapter, we are not going to use Ribbon. Ribbon was the de facto client-side load balancer for REST-based communications among applications using Spring Cloud. Although Netflix Ribbon client-side load balancing was a stable solution, it has now entered a maintenance mode, so unfortunately, it will not be developed anymore.

In this section, we will explain how to use the Spring Cloud Load Balancer, which is a replacement for Ribbon. Currently, the Spring Cloud Load Balancer is still under active development, so expect new functionalities soon.

By implementing client-side caching and Eureka with O-stock’s licensing and organization services, you can lessen the load on the Eureka Servers and improve client stability if Eureka becomes unavailable.



**How much time does it take to services to register themselves with Eureka service and show up in Eureka service list?**

Individual services registering with Eureka take up to 30 seconds to show up in the Eureka service. That’s because Eureka requires three consecutive heartbeat pings from the service, which are 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.

**How do you enable your service as Eureka service?**

We only use a new annotation, @EnableEurekaServer, to enable our service as an Eureka service.

**What are different ways to lookup a service which is registered on Eureka service?**

* Spring Discovery Client
* Spring Discovery Client–enabled REST template
* Netflix Feign client

**Why we have directly instantiated RestTemplate in following example while using DiscoveryClient for calling another microservices**

@Component

public class OrganizationDiscoveryClient {

@Autowired

private DiscoveryClient discoveryClient; ❶

public Organization getOrganization(String organizationId) {

RestTemplate restTemplate = new RestTemplate();

List<ServiceInstance> instances = ❷

discoveryClient.getInstances("organization-service");

if (instances.size()==0) return null;

String serviceUri = String.format

("%s/v1/organization/%s",instances.get(0)

.getUri().toString(),

organizationId); ❸

ResponseEntity<Organization> restExchange = ❹

restTemplate.exchange(

serviceUri, HttpMethod.GET,

null, Organization.class, organizationId);

return restExchange.getBody();

}

}

You should only use the Discovery Client when your service needs to query the Load Balancer to understand what services and service instances are registered with it. There are several problems with the code in listing 6.12, including these:

* You aren’t taking advantage of the Spring Cloud client-side Load Balancer. By calling the Discovery Client directly, you get a list of services, but it becomes your responsibility to choose which returned service instance you’re going to invoke.
* You’re doing too much work. In the code, you have to build the URL that you’ll use 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 also noticed that we directly instantiated the RestTemplate class in the code. This is antithetical to usual Spring REST invocations because you’ll usually have the Spring framework inject the RestTemplate class via the @Autowired annotation.

We instantiated the RestTemplate class in listing 6.12. Once we’ve enabled the Spring Discovery Client in the application class via @EnableDiscoveryClient, all REST templates managed by the Spring framework will have a Load Balancer–enabled interceptor injected into those instances. This will change how URLs are created with the RestTemplate class. Directly instantiating RestTemplate allows you to avoid this behavior.

**Using a Load Balancer–backed RestTemplate to call a service?**

//Package and import definitions left out for conciseness

@Component

public class OrganizationRestTemplateClient {

@Autowired

RestTemplate restTemplate;

public Organization getOrganization(String organizationId){

ResponseEntity<Organization> restExchange =

restTemplate.exchange(

"http://organization-service/v1/

organization/{organizationId}", ❶

HttpMethod.GET, null,

Organization.class, organizationId);

return restExchange.getBody();

}

}

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

The Load Balancer–enabled RestTemplate class parses the URL passed into it and uses whatever is passed in as the server name as the key to query the Load Balancer for an instance of a service. The actual service location and port are entirely abstracted from the developer. Also, by using the RestTemplate class, the Spring Cloud Load Balancer will round-robin load balance all requests among all the service instances.

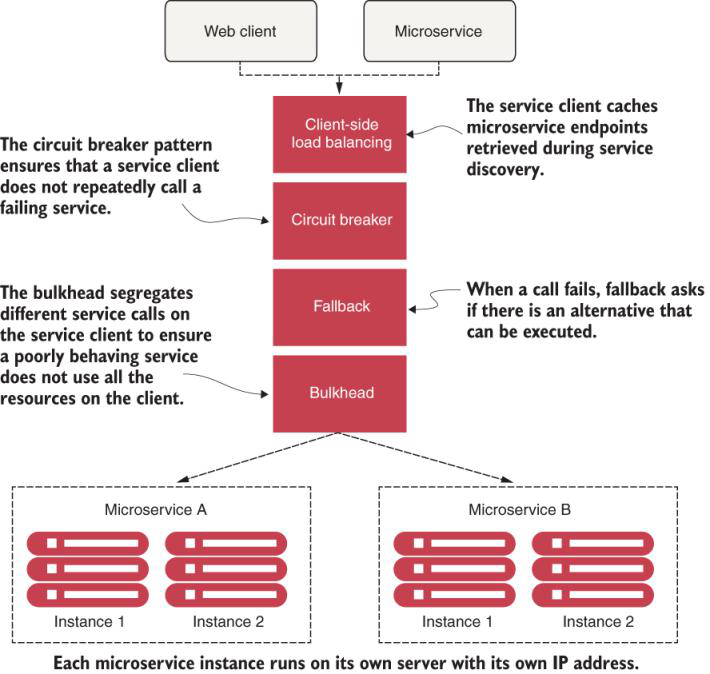
**How cloud side load balancer such as Spring cloud load balancer handles the failing instances?**

If the client-side load balancer detects a problem, it can remove that service instance from the pool of available service locations and prevent any future calls from hitting that service instance.

This is precisely the behavior that the Spring Cloud Load Balancer libraries provide out of the box (with no extra configuration).

**What are different types of client resiliency patterns?**

The four client resiliency patterns act as a protective buffer between a service consumer and the service



**What is circuit breaker design pattern?**

With a software circuit breaker, when a remote service is called, the circuit breaker monitors the call. If the calls take too long, the circuit breaker intercedes and kills the call. The circuit breaker pattern also monitors all calls to a remote resource, and if enough calls fail, the circuit breaker implementation will “pop,” failing fast and preventing future calls to the failing remote resource.

The circuit breaker pattern is modeled after an electrical circuit breaker. In an electrical system, a circuit breaker detects if there’s too much current flowing through the wire. If the circuit breaker detects a problem, it breaks the connection with the rest of the electrical system and keeps the system from frying the downstream components.

**What is Fallback pattern in Spring cloud microservices?**

With the fallback pattern, when a remote service call fails, rather than generating an exception, the service consumer executes an alternative code path and tries to carry out the action through another means. This usually involves looking for data from another data source or queueing the user’s request for future processing. The user’s call is not shown an exception indicating a problem, but they can be notified that their request will have to be tried later.

For instance, let’s suppose you have an e-commerce site that monitors your user’s behavior and gives them recommendations for other items they might want to buy. Typically, you’d call a microservice to run an analysis of the user’s past behavior and return a list of recommendations tailored to that specific user. However, if the preference service fails, your fallback might be to retrieve a more general list of preferences that are based on all user purchases, which is much more generalized. And, this data might come from a completely different service and data source.

**What is Fallback pattern in Spring cloud microservices?**

When using the bulkhead pattern, you break the calls to remote resources into their own thread pools and reduce the risk that a problem with one slow remote resource call will take down the entire application.

The thread pools act as the bulkheads for your service. Each remote resource is segregated and assigned to a thread pool. If one service is responding slowly, the thread pool for that type of service call can become saturated and stop processing requests. Assigning services to thread pools helps to bypass this type of bottleneck so that other services won’t become saturated.

The bulkhead pattern is based on a concept from building ships. A ship is divided into compartments called bulkheads, which are entirely segregated and watertight. Even if the ship’s hull is punctured, one bulkhead keeps the water confined to the area of the ship where the puncture occurred and prevents the entire ship from filling with water and sinking.

**Explain me what are problems can occur when you are working with microservice-based architecture running in the cloud?**

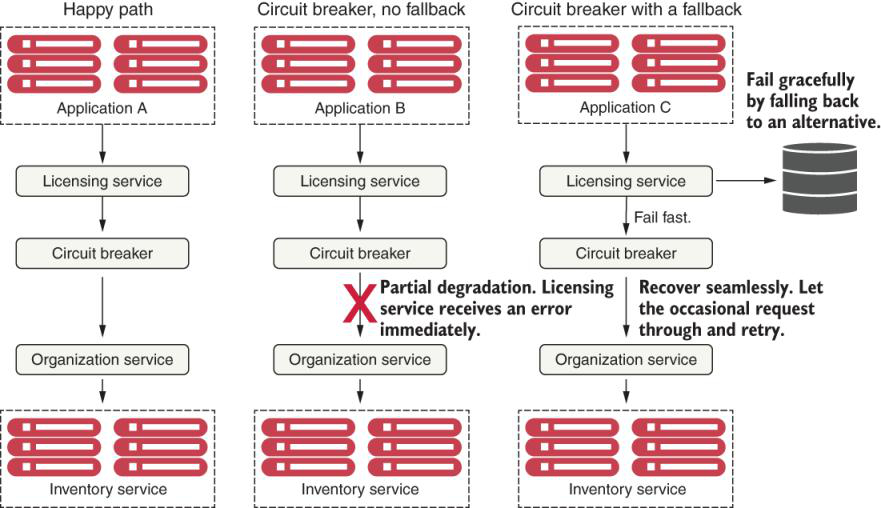
An application can be thought of as a graph of interconnected dependencies. If you don’t manage the remote calls among them, one poorly behaving remote resource can bring down all the services in the graph.

Diagram of a cloud service

Description automatically generated

**Explain me with typical example how circuit breaker works?**

The circuit breaker trips and allows a misbehaving service call to fail quickly and gracefully.



**What are key benefits of using a circuit breaker pattern?**

The key benefits a circuit break pattern offers is the ability for remote calls to

* Fail fast—When a remote service is experiencing a degradation, the application will fail fast and prevent resource-exhaustion issues that generally shut down the entire application. ***In most outage situations, it’s better to be partially down rather than being entirely down.***
* Fail gracefully—By timing out and failing fast, the circuit breaker pattern gives us the ability to fail gracefully or seek alternative mechanisms to carry out the user’s intent. For instance***, if a user is trying to retrieve data from one data source and that data source is experiencing service degradation, then our services can retrieve that data from another location.***
* Recover seamlessly—With the circuit breaker pattern acting as an intermediary, the **circuit breaker can periodically check to see if the resource being requested is back online** **and reenable access to it without human intervention.**

**What are some Netflix Spring cloud module which are in maintenance phase now and what are their recommended replacements?**

Resilience4J for Netflix Hystrix for resiliency

Spring Cloud Load Balancer for Netflix Ribbon for client-side load balancing

**What is Resilience4J?**

Resilience4j is a fault tolerance library inspired by Hystrix. It offers the following patterns for increasing fault tolerance due to network problems or failure of any of our multiple services:

* **Circuit breaker**—Stops making requests when an invoked service is failing.
* **Retry**—Retries a service when it temporarily fails.
* **Bulkhead**—Limits the number of outgoing concurrent service requests to avoid overload.
* **Rate limit**—Limits the number of calls that a service receives at a time.
* **Fallback**—Sets alternative paths for failing requests.

**What are prerequisites knowledge for implementing circuit breaker, retry, rate limit, fallback, and bulkhead patterns?**

Building implementations of the circuit breaker, retry, rate limit, fallback, and bulkhead patterns requires intimate knowledge of threads and thread management. To apply a high-quality set of implementations for these patterns requires a tremendous amount of work. Fortunately, we can use Spring Boot and the Resilience4j library to provide us with a battle-tested tool that’s used daily in several microservice architectures.

**What are different states in Resilience4J?**

Resilience4j circuit breaker states: closed, open, and half-open.

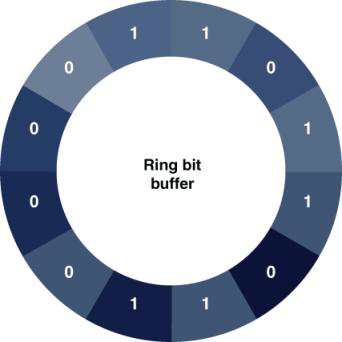
A diagram of a diagram

Description automatically generated

**How does Resilience4J changes different Closed, Open, Half-Open states?**

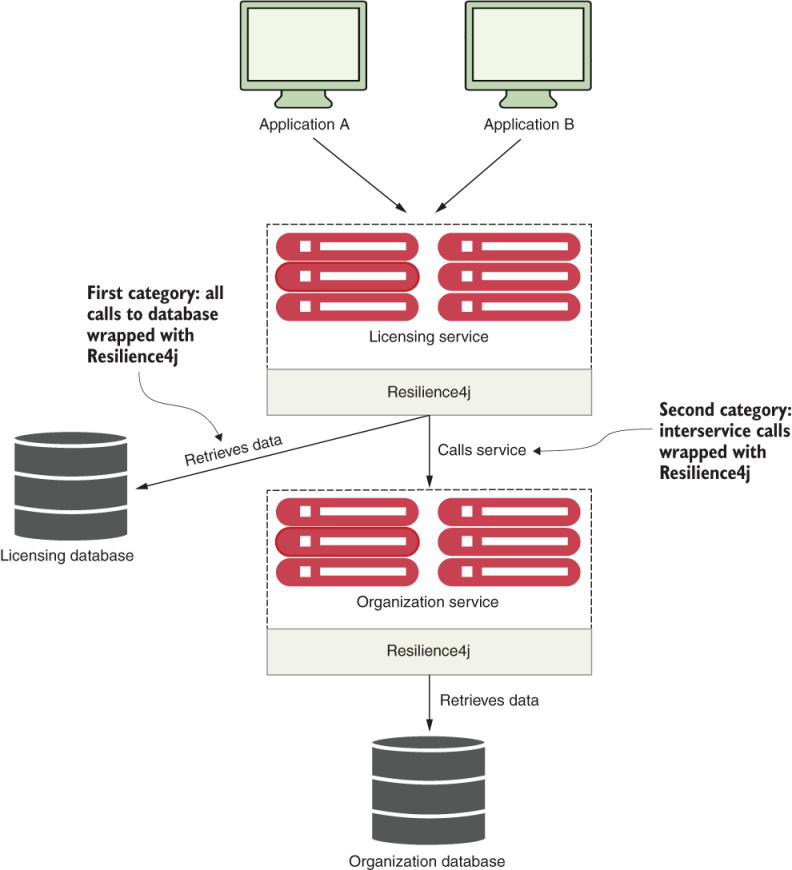
Initially, the Resilience4j circuit breaker starts in a closed state and waits for client requests. The closed state uses a ring bit buffer to store the success or failure status of the requests. When a successful request is made, the circuit breaker saves a 0 bit in the ring bit buffer. But if it fails to receive a response from the invoked service, it saves a 1 bit. [Figure 7.5](https://cdn2.percipio.com/1695194192.26062f4989754d90660f7a0a3e0a7651bc1375e2/eod/books/157553/OEBPS/section-75-62.xhtml#ch7fig5) shows a ring buffer with 12 results.

Figure 7.5: Resilience4j circuit breaker ring bit buffer with 12 results. This ring contains 0 for all the successful requests and 1 when it fails to receive a response from the invoked service



**How does Resilience4J works?**

Resilience4j sits between each remote resource call and protects the client. It doesn’t matter if the remote resource calls a database or a REST-based service.



**How do you implement fallback using Resilience4J?**

//Part of LicenseService.java omitted for conciseness

@CircuitBreaker(name= "licenseService",

fallbackMethod= "buildFallbackLicenseList") ❶

public List<License> getLicensesByOrganization(

String organizationId) throws TimeoutException {

logger.debug("getLicensesByOrganization Correlation id: {}",

UserContextHolder.getContext().getCorrelationId());

randomlyRunLong();

return licenseRepository.findByOrganizationId(organizationId);

}

private List<License> buildFallbackLicenseList(String organizationId,

➥ Throwable t){ ❷

List<License> fallbackList = new ArrayList<>();

License license = new License();

license.setLicenseId("0000000-00-00000");

license.setOrganizationId(organizationId);

license.setProductName(

"Sorry no licensing information currently available");

fallbackList.add(license);

return fallbackList;

}

❶ Defines a single function that’s called if the calling service fails

❷ Returns a hardcoded value in the fallback method

**What are key points to keep in mind while implementing a fallback using Resilience4J?**

To implement a fallback strategy with Resilience4j, we need to do two things. First, we need to add a fallbackMethod attribute to @CircuitBreaker or any other annotation (we will explain this later on). This attribute must contain the name of the method that will be called when Resilience4j interrupts a call because of a failure.

The second thing we need to do is to define a fallback method***. This method must reside in the same class as the original method that was protected by @CircuitBreaker. To create the fallback method in Resilience4j, we need to create a method that contains the same signature as the originating function plus one extra parameter, which is the target exception parameter. With the same signature, we can pass all the parameters from the original method to the fallback method***.

* Fallbacks provide a course of action when a resource has timed out or failed. If you find yourself using fallbacks to catch a timeout exception and then doing nothing more than logging the error, you should use a standard try…catch block around your service invocation instead: catch the exception and put the logging logic in the try…catch block.
* Be aware of the actions you take with your fallback functions. If you call out to another distributed service in your fallback service, you may need to wrap the fallback with a @CircuitBreaker. Remember, the same failure that you’re experiencing with your primary course of action might also impact your secondary fallback option. Code defensively.

**What will happen if you don’t use bulkhead pattern?**

In a microservice-based application, we’ll often need to call multiple microservices to complete a particular task. Without using a bulkhead pattern, the default behavior for these calls is that these are executed using the same threads that are reserved for handling requests for the entire Java container. In high volumes, performance problems with one service out of many can result in all of the threads for the Java container being maxed out and waiting to process work, while new requests for work back up. The Java container will eventually crash.

**What problem does bulkhead pattern solve?**

The bulkhead pattern segregates remote resource calls in their own thread pools so that a single misbehaving service can be contained and not crash the container. Resilience4j provides two different implementations of the bulkhead pattern. You can use these implementations to limit the number of concurrent executions:

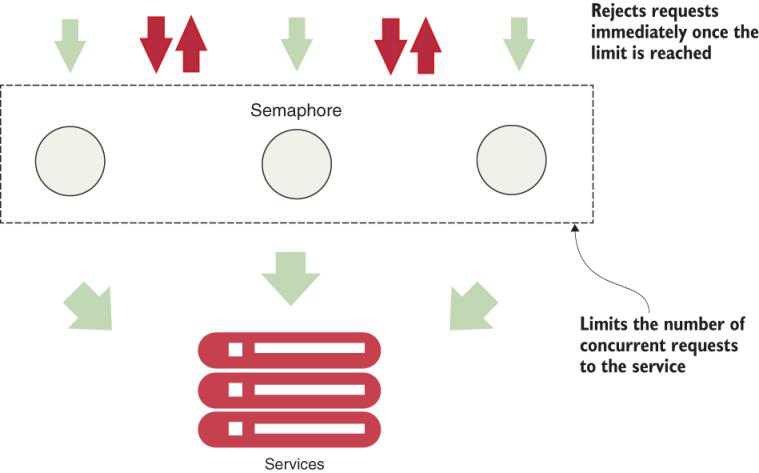
* Semaphore bulkhead—Uses a semaphore isolation approach, limiting the number of concurrent requests to the service. Once the limit is reached, it starts rejecting requests.
* Thread pool bulkhead—Uses a bounded queue and a fixed thread pool. This approach only rejects a request when the pool and the queue are full.

**By default, what technique Resilience4J uses and what what?**

Semaphore bulkhead pattern

Resilience4j, by default, uses the semaphore bulkhead type. [Figure 7.9](https://cdn2.percipio.com/1695198534.3eb6b435428dfc4b5505eda29a15571e7cec096a/eod/books/157553/OEBPS/section-77-64.xhtml#ch7fig9) illustrates this type.

Figure 7.9: The default Resilience4j bulkhead type is the semaphore approach.

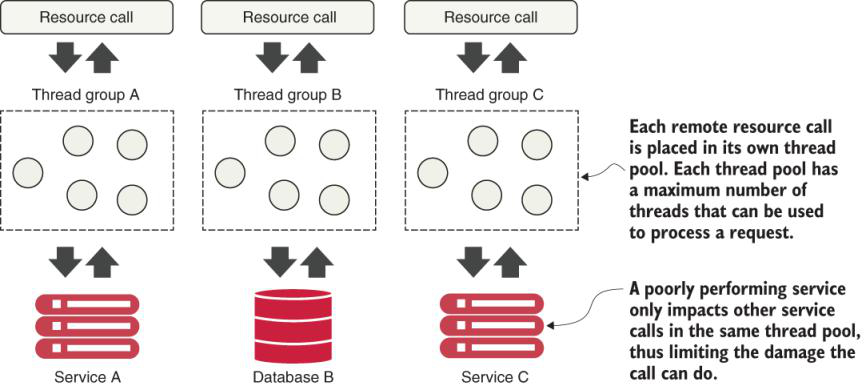


This model works fine if we have a small number of remote resources being accessed within an application, and the call volumes for the individual services are (relatively) evenly distributed. The problem is that if we have services with far higher volumes or longer completion times than other services, we can end up introducing thread exhaustion into our thread pools because one service ends up dominating all of the threads in the default thread pool.

**What is thread pool bulkhead pattern?**

Resilience4j provides an easy-to-use mechanism for creating bulkheads between different remote resource calls. [Figure 7.10](https://cdn2.percipio.com/1695198534.3eb6b435428dfc4b5505eda29a15571e7cec096a/eod/books/157553/OEBPS/section-77-64.xhtml#ch7fig10) shows what managed resources look like when they’re segregated into their own bulkheads.

Figure 7.10: A Resilience4j command tied to segregated thread pools



**How do configure the bulkhead pattern for a remote microservice call?**

**Listing 7.6: Configuring the bulkhead pattern for the licensing service**

//Part of boostrap.yml omitted for conciseness

resilience4j.bulkhead:

instances:

bulkheadLicenseService:

maxWaitDuration: 10ms ❶

maxConcurrentCalls: 20 ❷

resilience4j.thread-pool-bulkhead:

instances:

bulkheadLicenseService:

maxThreadPoolSize: 1 ❸

coreThreadPoolSize: 1 ❹

queueCapacity: 1 ❺

keepAliveDuration: 20ms ❻

❶ The maximum amount of time to block a thread

❷ The maximum number of concurrent calls

❸ The maximum number of threads in the thread pool

❹ The core thread pool size

❺ The queue’s capacity

❻ The maximum time that idle threads wait for new tasks before terminating

**Listing 7.7: Creating a bulkhead around the getLicensesByOrganization() method**

//Part of LicenseService.java omitted for conciseness

@CircuitBreaker(name= "licenseService",

fallbackMethod= "buildFallbackLicenseList")

@Bulkhead(name= "bulkheadLicenseService",

fallbackMethod= "buildFallbackLicenseList") ❶

public List<License> getLicensesByOrganization(

String organizationId) throws TimeoutException {

logger.debug("getLicensesByOrganization Correlation id: {}",

UserContextHolder.getContext().getCorrelationId());

randomlyRunLong();

return licenseRepository.findByOrganizationId(organizationId);

}

❶ Sets the instance name and fallback method for the bulkhead pattern

**How do you change the default bulkhead type?**

The first thing we should notice is that we’ve introduced a new annotation: @Bulkhead. This annotation indicates that we are setting up a bulkhead pattern. If we set no further values in the application properties, Resilience4j uses the default values previously mentioned for the semaphore bulkhead type.

The second thing to note in listing 7.7 is that we are not setting up the bulkhead type. In this case, the bulkhead pattern uses the semaphore approach. In order to change this to the thread pool approach, we need to add that type to the @Bulkhead annotation like so:

@Bulkhead(name = "bulkheadLicenseService", type = Bulkhead.Type.THREADPOOL,

➥ fallbackMethod = "buildFallbackLicenseList")

**How do you implement retried with Resilience4J?**

//Part of boostrap.yml omitted for conciseness

resilience4j.retry:

instances:

retryLicenseService:

maxRetryAttempts: 5 ❶

waitDuration: 10000 ❷

retry-exceptions: ❸

- java.util.concurrent.TimeoutException

❶ The maximum number of retry attempts

❷ The wait duration between the retry attempts

❸ The list of exceptions you want to retry

The first parameter, maxRetryAttempts, allows us to define the maximum number of retry attempts for our service. The default value for this parameter is 3. The second parameter, waitDuration, allows us to define the wait duration between the retry attempts. The default value for this parameter is 500 ms. The third parameter, retry-exceptions, sets a list of error classes that will be retried.

//Part of LicenseService.java omitted for conciseness

@CircuitBreaker(name= "licenseService",

fallbackMethod="buildFallbackLicenseList")

@Retry(name = "retryLicenseService",

fallbackMethod=

"buildFallbackLicenseList") ❶

@Bulkhead(name= "bulkheadLicenseService",

fallbackMethod="buildFallbackLicenseList")

public List<License> getLicensesByOrganization(String organizationId)

throws TimeoutException {

logger.debug("getLicensesByOrganization Correlation id: {}",

UserContextHolder.getContext().getCorrelationId());

randomlyRunLong();

return licenseRepository.findByOrganizationId(organizationId);

}

❶ Sets the instance name and fallback method for the retry pattern

Now that we know how to implement the circuit breaker and the retry pattern, let’s continue with the rate limiter. Remember, Resilience4j allows us to combine different patterns in the same method calls.

**When to user bulkhead pattern and when to use rate limiter?**

//Part of LicenseService.java omitted for conciseness

@CircuitBreaker(name= "licenseService",

fallbackMethod= "buildFallbackLicenseList")

@RateLimiter(name = "licenseService",

fallbackMethod = "buildFallbackLicenseList") ❶

@Retry(name = "retryLicenseService",

fallbackMethod = "buildFallbackLicenseList")

@Bulkhead(name= "bulkheadLicenseService",

fallbackMethod= "buildFallbackLicenseList")

public List<License> getLicensesByOrganization(String organizationId)

throws TimeoutException {

logger.debug("getLicensesByOrganization Correlation id: {}",

UserContextHolder.getContext().getCorrelationId());

randomlyRunLong();

return licenseRepository.findByOrganizationId(organizationId);

}

❶ Sets the instance name and fallback method for the rate limiter pattern

The main difference between the bulkhead and the rate limiter pattern is that the bulkhead pattern is in charge of limiting the number of concurrent calls (for example, it only allows *X* concurrent calls at a time). With the rate limiter, we can limit the number of total calls in a given timeframe (for example, allow *X* number of calls every *Y* seconds).

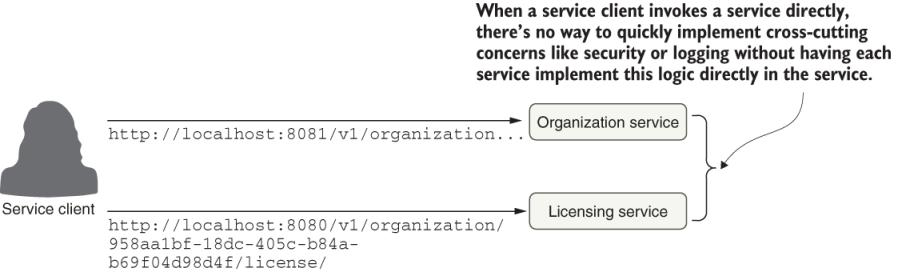
In order to choose which pattern is right for you, double-check what your needs are. If you want to block concurrent times, your best choice is a bulkhead, but if you want to limit the total number of calls in a specific time period, your best option is the rate limiter. If you are looking at both scenarios, you can also combine them.

**What is importance of service gateway?**

The service gateway sits as the gatekeeper for all inbound traffic to microservice calls within our application. With a service gateway in place, our service clients never directly call the URL of an individual service, but instead place all calls to the service gateway.

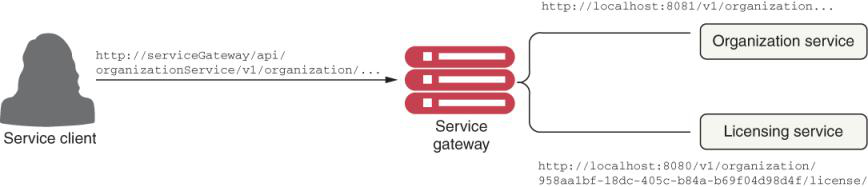
**What are the challenges without service gateway?**

Without a service gateway, the service client calls distinct endpoints for each service



**What problem does service gateway solves?**

The service gateway sits between the service client and the corresponding service instances. All service calls (both internal-facing and external) should flow through the service gateway



**What another name you can you use for service gateway?**

Gatekeeper

The service gateway sits as the gatekeeper for all inbound traffic to microservice calls within our application. With a service gateway in place, our service clients never directly call the URL of an individual service, but instead place all calls to the service gateway.

**What are cross-cutting concerns that can be implemented in a service gateway?**

* Static routing—A service gateway places all service calls behind a single URL and API route. This simplifies development as we only have to know about one service endpoint for all of our services.
* Dynamic routing—A service gateway can inspect incoming service requests and, based on the data from the incoming request, perform intelligent routing for the service caller. For instance, customers participating in a beta program might have all calls to a service routed to a specific cluster of services that are running a different version of code from what everyone else is using.
* Authentication and authorization—Because all service calls route through a service gateway, the service gateway is a natural place to check whether the callers of a service have authenticated themselves.
* Metric collection and logging—A service gateway can be used to collect metrics and log information as a service call passes through it. You can also use the service gateway to confirm that critical pieces of information are in place for user requests, thereby ensuring that logging is uniform. This doesn’t mean that you shouldn’t collect metrics from within your individual services. Rather, a service gateway allows you to centralize the collection of many of your basic metrics, like the number of times the service is invoked and the service response times.

**Wait—isn’t a service gateway a single point of failure and a potential bottleneck?**

A service gateway, if not implemented correctly, can carry the same risk. Keep the following in mind as you build your service gateway implementation:

* Load balancers are useful when placed in front of individual groups of services. In this case, a load balancer sitting in front of multiple service gateway instances is an appropriate design and ensures that your service gateway implementation can scale as needed. But having a load balancer sitting in front of all your service instances isn’t a good idea because it becomes a bottleneck.
* Keep any code you write for your service gateway stateless. Don’t store any information in memory for the service gateway. If you aren’t careful, you can limit the scalability of the gateway. Then, you will need to ensure that the data gets replicated across all service gateway instances.
* Keep the code you write for your service gateway light. The service gateway is the “chokepoint” for your service invocation. Complex code with multiple database calls can be the source of difficult-to-track performance problems in the service gateway.

**What is non-blocking application?**

Nonblocking applications are written in such a way that the main threads are never blocked. Instead, these threads are always available to serve requests and to process them asynchronously in the background to return a response once processing is done.

**What are capabilities of Spring cloud gateway?**

Spring Cloud Gateway offers several capabilities, including

* Mapping the routes for all the services in your application to a single URL. The Spring Cloud Gateway isn’t limited to a single URL, however. Actually, with it, we can define multiple route entry points, making route mapping extremely fine-grained (each service endpoint gets its own route mapping). But the first and most common use case is to build a single entry point through which all service client calls will flow.
* Building filters that can inspect and act on the requests and responses coming through the gateway. These filters allow us to inject policy enforcement points in our code and to perform a wide number of actions on all of our service calls in a consistent fashion. In other words, these filters allow us to modify the incoming and outgoing HTTP requests and responses.
* Building predicates, which are objects that allow us to check if the requests fulfill a set of given conditions before executing or processing a request. The Spring Cloud Gateway includes a set of built-in Route Predicate Factories.

**What is use of Spring Cloud Gateway at high level?**

At its heart, the Spring Cloud Gateway is a reverse proxy.

A reverse proxy is an intermediate server that sits between the client trying to reach a resource and the resource itself. The client has no idea it’s even communicating with a server. The reverse proxy takes care of capturing the client’s request and then calls the remote resource on the client’s behalf.

In the case of a microservice architecture, Spring Cloud Gateway (our reverse proxy) takes a microservice call from a client and forwards it to the upstream service. The service client thinks it’s only communicating with the gateway. But it is not actually as simple as that. To communicate with the upstream services, the gateway has to know how to map the incoming call to the upstream route. The Spring Cloud Gateway has several mechanisms to do this, including

* Automated mapping of routes using service discovery
* Manual mapping of routes using service discovery

**How does of Spring Cloud Gateway automatically maps of Routes via Service Discovery. Can you give typical example of Spring Cloud Gateway working?**

if we want to call our organization service and use automated routing via the Spring Cloud Gateway, we would have our client call the Gateway service instance using the following URL as the endpoint:

http://localhost:8072/organization-service/v1/organization/958aa1bf-18dc-

➥ 405c-b84a-b69f04d98d4f

The Gateway server is accessed via the http://localhost:8072 endpoint. The service we want to invoke (the organization service) is represented by the first part of the endpoint path in the service. [Figure 8.5](https://cdn2.percipio.com/1695318650.c45133d176f0a88320c7bc6dfb43cbb94ed471c2/eod/books/157553/OEBPS/section-83-72.xhtml#ch8fig5) illustrates this mapping in action.

Figure 8.5: The Spring Cloud Gateway uses the organization-service application name to map requests to instances of the organization service.

**How do you manually map the routes?**

#### Listing 8.6: Mapping routes manually in the gateway-server.yml file

spring:

cloud:

gateway:

discovery.locator:

enabled: true

lowerCaseServiceId: true

routes:

- id: organization-service ❶

uri: lb://organization-service ❷

predicates: ❸

- Path=/organization/\*\*

filters: ❹

- RewritePath=/organization/

(?<path>.\*), /$\{path} ❺

❶ This optional ID is an arbitrary route ID.

❷ Sets the route’s destination URI

❸ The path, although set by the load() method, is just another option.

❹ Filters a collection of Spring web.filters to modify the request or response before or after sending the response

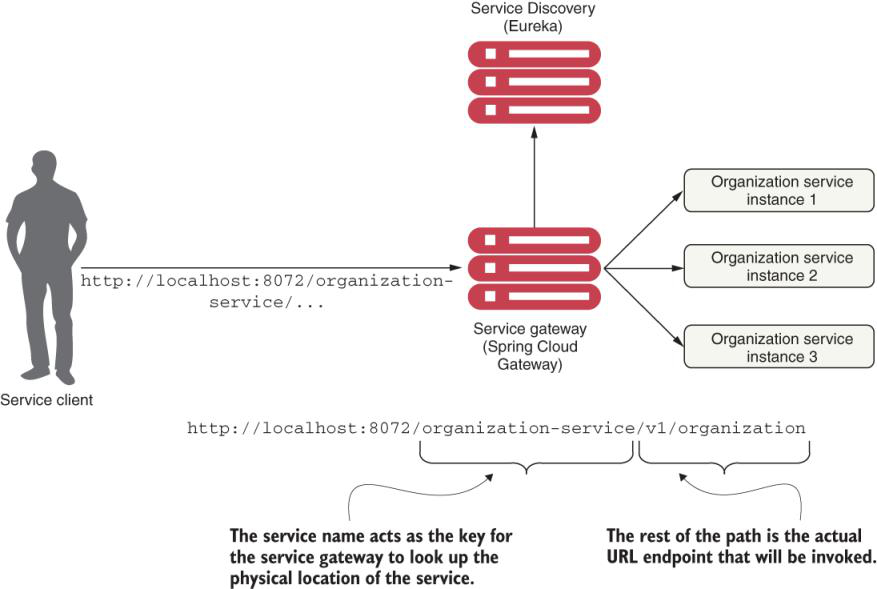
❺ Rewrites the request path, from /organization/\*\* to /\*\*, by taking the path regexp as a parameter and a replacement order

**How do you dynamically refresh the routes?**

The ability to dynamically reload routes is useful because it allows us to change the mapping of routes without having to restart the Gateway server(s). Existing routes can be modified quickly, and new routes will have to go through the act of recycling each Gateway server in our environment.

If we enter the actuator/gateway/routes endpoint, we should see our organization service currently shown in the gateway. Now, if we want to add new route mappings on the fly, all we have to do is make the changes to the configuration file and commit those changes back to the Git repository where Spring Cloud Config pulls its configuration data. Then we can commit the changes to GitHub.

Spring Actuator exposes a POST-based endpoint route, actuator/gateway/ refresh, that will cause it to reload its route configuration. Once this actuator/ gateway/refresh is reached, if you then enter the /routes endpoint, you’ll see that two new routes are exposed. The response of the actuator/gateway/refresh returns an HTTP 200 status code without a response body.



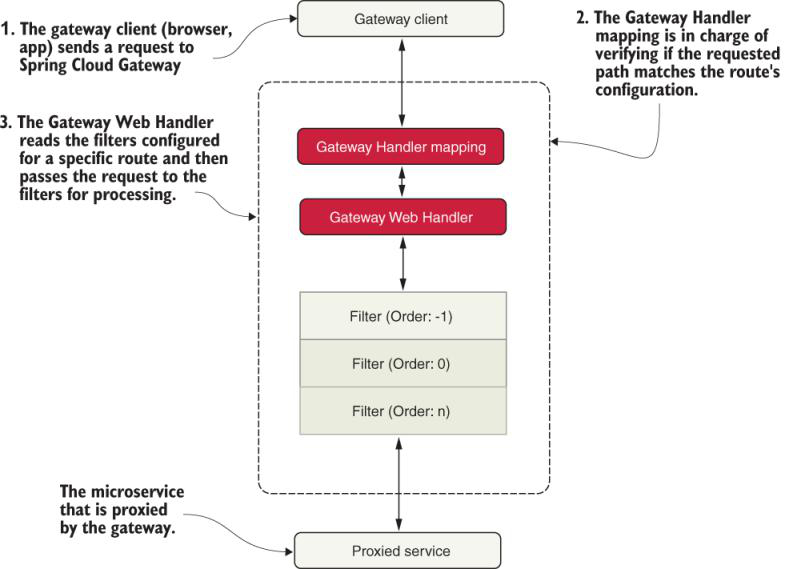
The beauty of using Spring Cloud Gateway with Eureka is that not only do we now have a single endpoint through which we can make calls, but we can also add and remove instances of a service without ever having to modify the gateway. For instance, we can add a new service to Eureka, and the gateway automatically routes calls to it because it’s communicating with Eureka about where the actual physical service endpoints are located.

If we want to see the routes managed by the Gateway server, we can list the routes via the actuator/gateway/routes endpoint on the Gateway server. This will return a listing of all the mappings on our service. [Figure 8.6](https://cdn2.percipio.com/1695318650.c45133d176f0a88320c7bc6dfb43cbb94ed471c2/eod/books/157553/OEBPS/section-83-72.xhtml#ch8fig6) shows the output from selecting http://localhost:8072/actuator/gateway/routes.

**What is difference in usage of Servlet filter and Spring aspect and Spring Cloud Gateway and its Predicate and Filter Factories?**

While a servlet filter or Spring aspect is localized to a specific service, using the Gateway and its Predicate and Filter Factories allows us to implement cross-cutting concerns across all the services being routed through the gateway.

How the Spring Cloud Gateway architecture applies the predicates and filters when a request is made

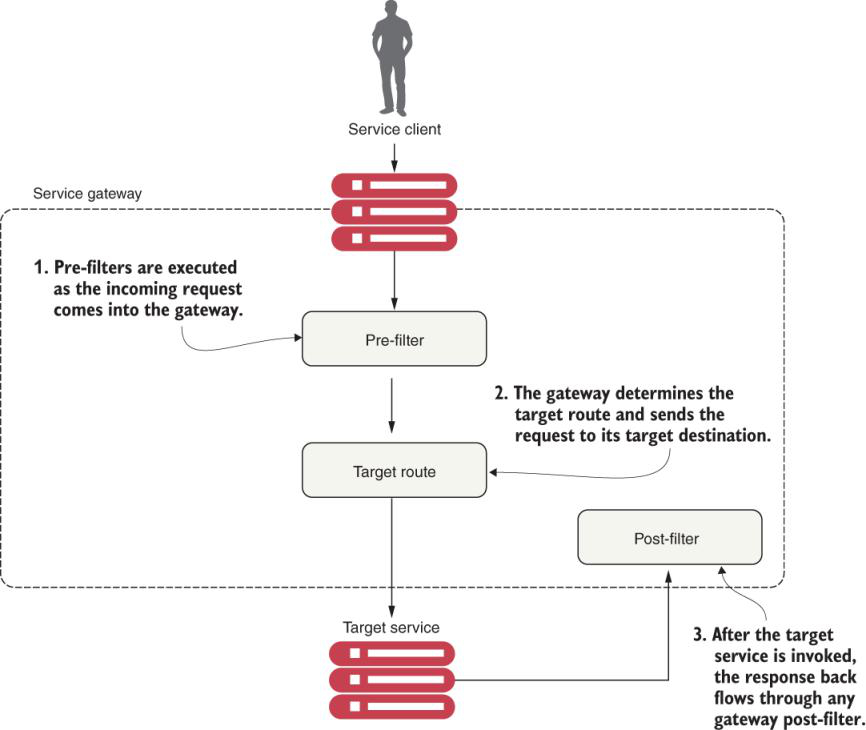


First, the gateway client (browsers, apps, and so forth) sends a request to Spring Cloud Gateway. Once that request is received, it goes directly to the Gateway Handler that is in charge of verifying that the requested path matches the configuration of the specific route it is trying to access. If everything matches, it enters the Gateway Web Handler that is in charge of reading the filters and sending the request to those filters for further processing. Once the request passes all the filters, it is forwarded to the routing configuration: a microservice.

**What is use of Custom filter in Spring cloud Gateway?**

The ability to proxy all requests through the gateway lets us simplify our service invocations. But the real power of Spring Cloud Gateway comes into play when we want to write custom logic that can be applied against all the service calls flowing through the gateway. Most often, this custom logic is used **to enforce a consistent set of application policies like security, logging, and tracking** among all the services.

he pre-filters, target route, and post-filters form a pipeline in which a client request flows. As a request comes into the gateway, custom filters can manipulate the incoming request



**How do you track service calls using Custom filters in Spring Cloud Gateway?**

Gateway filters provide centralized tracking of service calls and logging. These filters allow us to enforce custom rules and policies against microservice calls

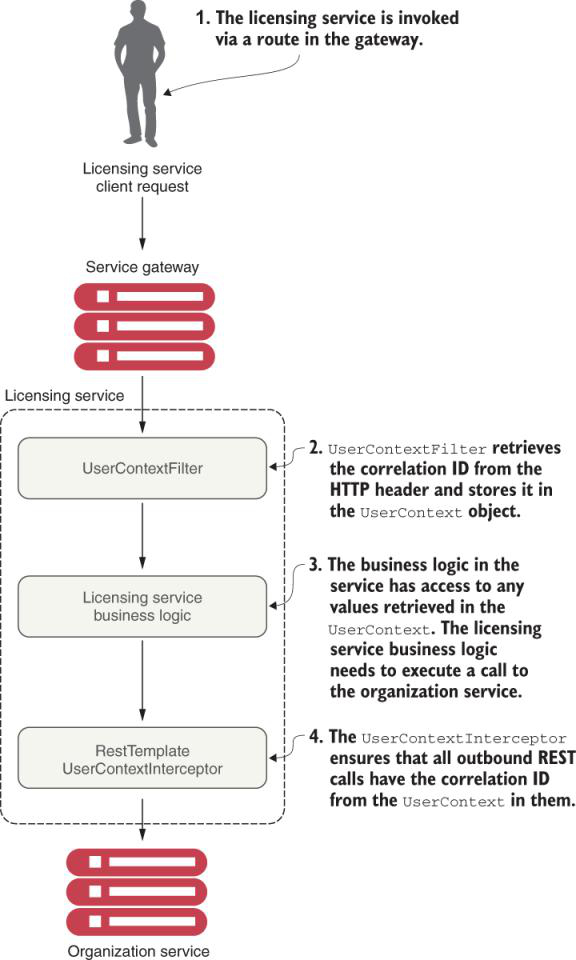
A diagram of a service

Description automatically generated

* Tracking filter—The tracking filter is a pre-filter that ensures that every request flowing from the gateway has a correlation ID associated with it. A correlation ID is a unique ID that gets carried across all the microservices that are executed when carrying out a customer request. A correlation ID allows us to trace the chain of events that occur as a call goes through a series of microservice calls.
* Target service—The target service can either be an organization or the licensing service. Both services receive the correlation ID in the HTTP request header.
* Response filter—The response filter is a post-filter that injects the correlation ID associated with the service call into the HTTP response header sent to the client. This way, the client will have access to the correlation ID associated with the request.

**How do you ensure that correlation\_id is propagated on to all downstream call?**

Using a set of common classes so that the correlation ID can be propagated to downstream service calls.



**When you are to choose between repeated code vs. shared libraries which one would you choose?**

Common libraries are fine when dealing with infrastructure-style tasks. If you start sharing business-oriented classes, you’re asking for trouble because you’ll end up breaking down the boundaries between the services.

We seem to be breaking our own advice with the code examples in this chapter, however. If you look at all the services in the chapter, these have their own copies of the UserContextFilter, UserContext, and UserContextInterceptor classes.

**What is OAuth2?**

**OAuth2 is a token-based security framework** **that describes patterns for granting authorization but does not define how to actually perform authentication**. Instead, it allows users to authenticate themselves with a third-party authentication service, called an identity provider (IdP). If the user successfully authenticates, they are presented with a token that must be sent with every request. The token can then be validated back to the authentication service.

The real power behind OAuth2 is that it allows application developers to easily integrate with third-party cloud providers and authenticate and authorize users with those services without having to pass the user’s credentials continually to the third-party service.

OpenID Connect (OIDC) is a layer on top of the OAuth2 framework that provides authentication and profile information about who is logged in to the application (the identity). When an authorization server supports OIDC, it is sometimes called an identity provider.

**What is difference between Authentication and Authorization?**

Authentication is the act of a user proving who they are by providing credentials. Authorization determines whether a user is allowed to do what they want to do. For instance, user Illary could prove her identity by providing a user ID and password, but she may not be authorized to look at sensitive data (payroll data, for example). For our discussion, a user must be authenticated before authorization takes place.

**What is Keycloak?**

Keycloak is an open source identity and access management solution for our services and applications. The main objective of Keycloak is to facilitate the protection of the services and applications with little or no code.

**What is realm concept in Keycloak?**

A realm is a concept that Keycloak uses to refer to an object that manages a set of users, credentials, roles, and groups.

**Once protected resources is called with HTTP header containing bearer token, how token is checked if it is valid or not?**

Once we set up the organization service as a protected resource, every time a call is made to the service, the caller must include the authentication HTTP header containing a Bearer access token to the service. Our protected resource then has to call back to the Keycloak server to see if the token is valid.

**How do you protect a microservice using Keycloak authentication provider?**

While the creation and management of access tokens is the Keycloak server’s responsibility, in Spring, the definition of which user roles have permissions to do what actions occurs at the individual service level.

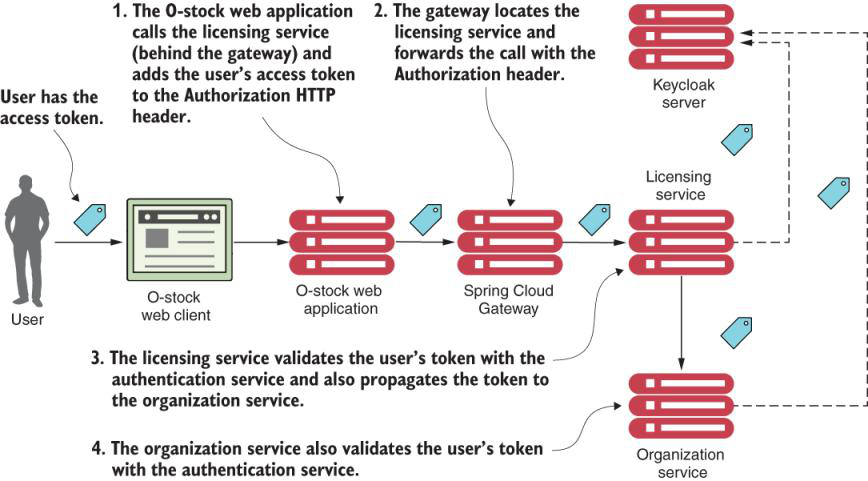
**What is difference between Unauthorized and Forbidden?**

**Unauthorized:** May be, you are not provided credentials or token in the request.

**Forbidden:** May be, you don’t have proper permissions.

**How access token received from authentication service propagation throughout the entire call chain?**

The access token must be carried throughout the entire call chain



* The user has already authenticated with the Keycloak server and places a call to the O-stock web application. The user’s access token is stored in the user’s session. The O-stock web application needs to retrieve some licensing data and call the licensing service REST endpoint (1). As part of the call to the licensing REST endpoint, the O-stock web application adds the access token via the HTTP Authorization header. The licensing service is only accessible behind a Spring Cloud Gateway.
* The gateway looks up the licensing service endpoint and then forwards the call to one of the licensing service’s servers (2). The services gateway copies the authorization HTTP header from the incoming call and ensures that the HTTP header is forwarded on to the new endpoint.
* The licensing service receives the incoming call. Because the licensing service is a protected resource, the licensing service will validate the token with the Keycloak server (3) and then check the user’s roles for the appropriate permissions. As part of its work, the licensing service invokes the organization service. When doing this, the licensing service needs to propagate the user’s access token to the organization service.
* When the organization service receives the call, it takes the HTTP Authorization header and validates the token with the Keycloak server (4).

**What are different steps involved when enabling propagation of token in call chain in a microservices architecture?**

1. The first step is to modify the gateway so that it propagates the access token to the licensing service. By default, the gateway doesn’t forward sensitive HTTP headers like cookie, set-cookie, and authorization to downstream services. To allow the propagation of the authorization HTTP header, we need to add to each route the following filter in the gateway-server.yml configuration file located in the Spring Cloud Config repository:

- RemoveRequestHeader= Cookie,Set-Cookie

This configuration is a blacklist of the sensitive headers that the gateway will keep from being propagated to a downstream service. The absence of the Authorization value in the RemoveRequestHeader list means that the gateway will allow that header through. If we don’t set this configuration property, the gateway automatically blocks the propagation of all three values (Cookie, Set-Cookie, and Authorization).

2. We need to configure our licensing service to include the Keycloak and Spring Security dependencies and set up any authorization rules we want for the service.

3. We need to add the Keycloak properties to the application properties file in the configuration server.

4. Without Spring Security, we’d have to write a servlet filter to grab the HTTP header of the incoming licensing service call and then manually add it to every outbound service call in the licensing service. Keycloak provides a new REST template class that supports these calls. The class is called KeycloakRestTemplate.

**How can you parse custom field from JWT token?**

<dependency>

<groupId>commons-codec</groupId>

<artifactId>commons-codec</artifactId>

</dependency>

<dependency>

<groupId>org.json</groupId>

<artifactId>json</artifactId>

<version>20190722</version>

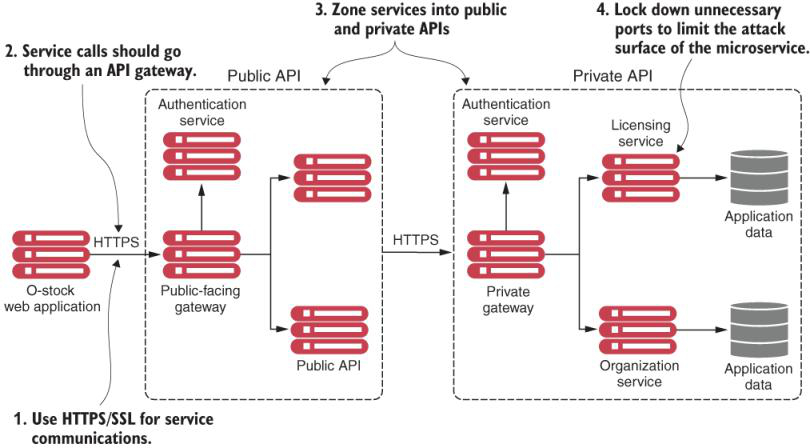
</dependency>

**What is difference between OpenId and OAuth2?**

OAuth2 is only concerned with authorization, while OpenID Connect is also concerned with authentication.

**What are some best practices to use while building microservice security?**

* Use HTTPS/Secure Sockets Layer (SSL) for all service communications.
* Use an API gateway for all service calls.
* Provide zones for your services (for example, a public API and private API).
* Limit the attack surface of your microservices by locking down unneeded network ports.



**What is Spring cloud stream?**

Spring Cloud Stream allows us to easily implement message publication and consumption while shielding our services from the implementation details associated with the underlying messaging platform.

**What do we need to consider when implementing a caching solution?**

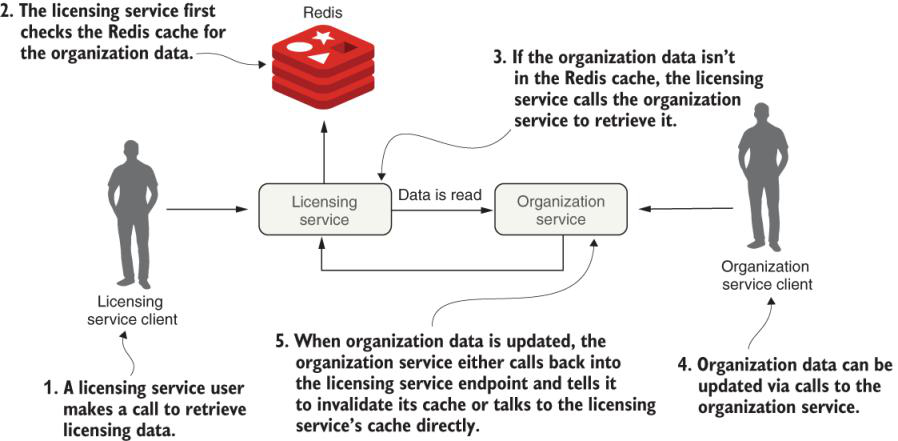
1. Cached data needs to be consistent across all instances of the licensing service. This means that we can’t cache the data locally within the licensing service because we want to guarantee that the same organization data is read regardless of the service instance hitting it.
2. We cannot cache the organization data within the memory of the container hosting the licensing service. The run-time container hosting our service is often restricted in size and can obtain data using different access patterns. A local cache can introduce complexity because we have to guarantee our local cache is in sync with all of the other services in the cluster.
3. When an organization record changes via an update or delete, we want the licensing service to recognize that there has been a state change in the organization service. The licensing service should then invalidate any cached data it has for that specific organization and evict it from the cache.

**What are different uses cases of Redis?**

Redis is distributed key-value store used as:

1. Database
2. Cache
3. Message Broker

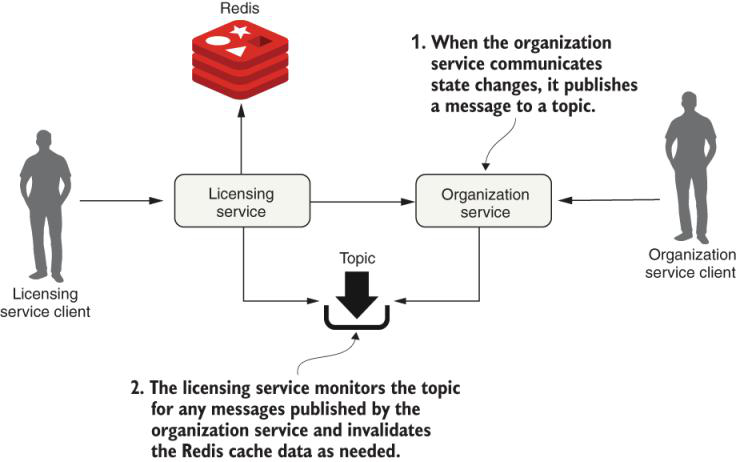
**How do you build a caching solution using a traditional synchronous request-response programming model such as Redis?**



**What are different ways of communicating state changes between microservices?**

1. The first approach will implement the previously stated requirements using a synchronous request-response model. When the organization state changes, the licensing and organization services will communicate back and forth via their REST endpoints.
2. The organization service will emit an asynchronous event (message) to communicate that its organization data has changed. The organization service will then publish a message to a queue, which will indicate that an organization record was updated or deleted—a change in state. The licensing service will listen with an intermediary (message broker or queue) to determine if an organization event occurred, and if so, clear the organization data from its cache.

**How do you use Messaging to Communicate State Changes between Services?**



**What are challenges working with a messaging in microservices architecture?**

1. It requires that we understand how our application will behave based on the order in which messages are consumed and what happens if a message is processed out of order.

2. Using messages in our microservices often means a mix of synchronous service calls and asynchronous service processing. The asynchronous nature of messages means they might not be received or processed in close proximity to when the message is published or consumed.

**How do you integrate messaging in Spring based microservices?**

Spring Cloud makes it easy to integrate messaging into our Spring-based microservices. It does this through the Spring Cloud Stream project (<https://spring.io/projects/spring-cloud-stream>), which is an annotation-driven framework that allows us to easily build message publishers and consumers in our Spring applications.

**What are different messaging platforms we can use with Spring cloud stream?**

We can use multiple message platforms with Spring Cloud Stream, including the **Apache Kafka** project and **RabbitMQ**, and the platform’s implementation-specific details are kept out of the application code. The implementation of message publication and consumption in your application is done through platform-neutral Spring interfaces.

**How do you do distributed debugging in a microservices architecture?**

* Using correlation IDs to link together transactions across multiple services
* Aggregating log data from various services into a single searchable source
* Visualizing the flow of a user transaction across multiple services to understand each part of the transaction’s performance characteristics
* Analyzing, searching, and visualizing log data in real time using the ELK stack

**What are the different tools that can used for distributed debugging?**

* Spring Cloud Sleuth (<https://cloud.spring.io/spring-cloud-sleuth/reference/html/>)—The Spring Cloud Sleuth project instruments our incoming HTTP requests with trace IDs (aka correlation IDs). It does this by adding filters and interacting with other Spring components to let the generated correlation IDs pass through to all the system calls.
* Zipkin (<https://zipkin.io/>)—Zipkin is an open source data-visualization tool that shows the flow of a transaction across multiple services. Zipkin allows us to break a transaction down into its component pieces and visually identify where there might be performance hotspots.
* ELK stack (<https://www.elastic.co/what-is/elk-stack>)—The ELK stack combines three open source tools—Elasticsearch, Logstash, and Kibana—that allow us to analyze, search, and visualize logs in real time.
  + Elasticsearch is a distributed analytics engine for all types of data (structured and non-structured, numeric, text-based, and so on).
  + Logstash is a server-side data processing pipeline that allows us to add and ingest data from multiple sources simultaneously and transform it before it is indexed in Elasticsearch.
  + Kibana is the visualization and data management tool for Elasticsearch. It provides charts, maps, and real-time histograms.

**How do you implement distributed debugging if you are not allowed to use Spring cloud sleuth?**

We first introduced the concept of correlation IDs in [chapters 7](https://cdn2.percipio.com/1695969597.edeac2559eb2ed8820aecbc12a07b8573ab8667b/eod/books/157553/OEBPS/chapter-7-57.xhtml#ch07) and [8](https://cdn2.percipio.com/1695969597.edeac2559eb2ed8820aecbc12a07b8573ab8667b/eod/books/157553/OEBPS/chapter-8-69.xhtml#ch08). A correlation ID is a randomly generated unique number or string assigned to a transaction as the transaction is initiated. As the transaction flows across multiple services, the correlation ID is propagated from one service call to another.

In the context of [chapter 8](https://cdn2.percipio.com/1695969597.edeac2559eb2ed8820aecbc12a07b8573ab8667b/eod/books/157553/OEBPS/chapter-8-69.xhtml#ch08), we used a Spring Cloud Gateway filter to inspect all incoming HTTP requests and inject a correlation ID into the request if one wasn’t present. Once the correlation ID was present, we used a custom Spring HTTP filter on each one of our services to map the incoming variable to a custom UserContext object. With this object in place, we manually added the correlation ID to our log statements by appending the correlation ID to a log statement or, with a little work, adding the correlation ID directly to Spring’s Mapped Diagnostic Context (MDC). MDC is a map that stores a set of key-value pairs provided by the application that’s inserted in the log messages.

In that chapter, we also wrote a Spring interceptor to ensure that all HTTP calls from a service would propagate the correlation ID by adding the correlation ID to the HTTP headers of outbound calls.

**How does correlation id gets added to calls?**

#### With Spring Cloud Sleuth, if we use Spring Boot’s logging implementation, we’ll automatically get correlation IDs added to the log statements we put in our microservices.

**What happens once you add Spring cloud sleuth dependency in your microservice?**

<dependency>

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

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

</dependency>

These dependencies pull in all the core libraries needed for Spring Cloud Sleuth. That’s it. Once they are pulled in, our service will now.

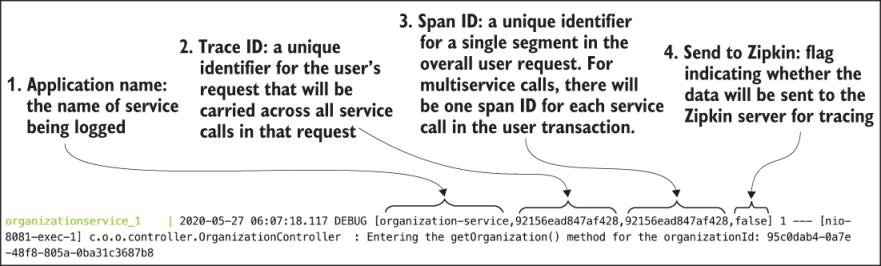
* Inspect every incoming HTTP service and determine whether Spring Cloud Sleuth tracing information exists in the incoming call. If the Spring Cloud Sleuth tracing data is present, the tracing information passed into our microservice will be captured and made available to our service for logging and processing.
* Add Spring Cloud Sleuth tracing information to the Spring MDC so that every log statement created by our microservice will be added to the log.
* Inject Spring Cloud tracing information into every outbound HTTP call and Spring messaging channel message our service makes.

**How does Spring Cloud Sleuth works?**

http://localhost:8072/organization/v1/organization/95c0dab4-0a7e-48f8-805a-

➥ 0ba31c3687b8

Figure 11.1: Spring Cloud Sleuth adds tracing information to each log entry written by our organization service. This data helps tie together service calls for a user’s request



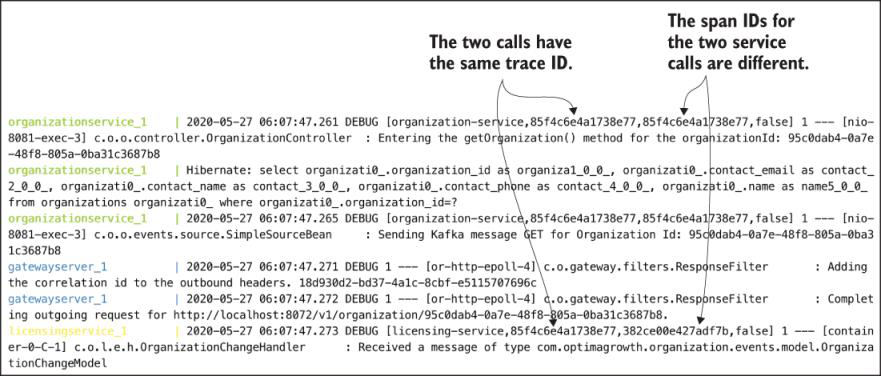
Spring Cloud Sleuth adds four pieces of information to each log entry. These four pieces (numbered to correspond with the numbers in [figure 11.1](https://cdn2.percipio.com/1695971456.beb3dcb5b849c28b77eda5a3913e9f39bce7a5d9/eod/books/157553/OEBPS/section-111-92.xhtml#ch11fig1)) are as follows:

1. The application name of the service where the log entry is entered. By default, Spring Cloud Sleuth uses the application name (spring.application.name) as the name that gets written in the trace.
2. The trace ID, which is the equivalent term for correlation ID. This is a unique number that represents an entire transaction.
3. The span ID, which is a unique ID that represents part of the overall transaction. Each service participating within the transaction will have its own span ID. Span IDs are particularly relevant if you integrate with Zipkin to visualize your transactions.
4. Export, a true/false indicator that determines whether trace data is sent to Zipkin. In high-volume services, the amount of trace data generated can be overwhelming and not add a significant amount of value. Spring Cloud Sleuth lets us determine when and how to send a transaction to Zipkin.

**Notes:** By default, any application flow starts with the same trace and span IDs.

Up to now, we’ve only looked at the logging data produced by a single service call. Let’s look at what happens when you also make a call to the licensing service. [Figure 11.2](https://cdn2.percipio.com/1695971456.beb3dcb5b849c28b77eda5a3913e9f39bce7a5d9/eod/books/157553/OEBPS/section-111-92.xhtml#ch11fig2) shows the logging output from the two service calls.

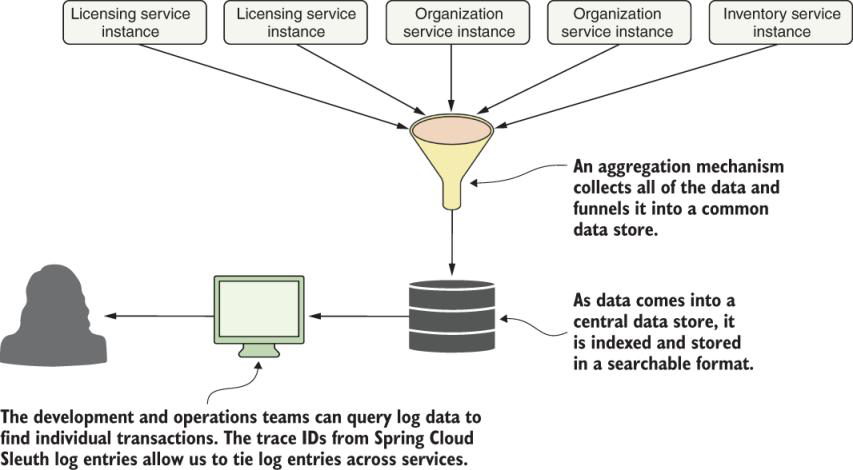
Figure 11.2: With multiple services involved in a transaction, we see that they share the same trace ID



In [figure 11.2](https://cdn2.percipio.com/1695971456.beb3dcb5b849c28b77eda5a3913e9f39bce7a5d9/eod/books/157553/OEBPS/section-111-92.xhtml#ch11fig2), we can see that both the licensing and organization services have the same trace ID, 85f4c6e4a1738e77. However, the organization service has a span ID of 85f4c6e4a1738e77 (the same value as the transaction ID). The licensing service has a span ID of 382ce00e427adf7b. By adding nothing more than a few .pom dependencies, we’ve replaced all the correlation ID infrastructure that you built in [chapters 7](https://cdn2.percipio.com/1695971456.beb3dcb5b849c28b77eda5a3913e9f39bce7a5d9/eod/books/157553/OEBPS/chapter-7-57.xhtml#ch07) and [8](https://cdn2.percipio.com/1695971456.beb3dcb5b849c28b77eda5a3913e9f39bce7a5d9/eod/books/157553/OEBPS/chapter-8-69.xhtml#ch08).

**How does logging architecture work in a distributed microservices?**

The combination of aggregated logs and a unique transaction ID across service log entries makes debugging distributed transactions more manageable.



**How Spring cloud sleuth fit together with ELK stack?**

we can see how Spring Cloud Sleuth and ELK fit together for our solution.

