7 When bad things happen: Resiliency patterns with Spring Cloud and Resilience4j

This chapter covers

- Implementing circuit breakers, fallbacks, and bulkheads
- Using the circuit breaker pattern to conserve client resources
- Using Resilience4j when a remote service fails
- Implementing Resilience4j's bulkhead pattern to segregate remote resource calls
- Tuning Resilience4j circuit breaker and bulkhead implementations
- Customizing Resilience4j's concurrency strategy

All systems, especially distributed systems, experience failure. How we build our applications to respond to that failure is a critical part of every software developer's job. However, when it comes to building resilient systems, most software engineers only take into account the complete failure of a piece of infrastructure or critical

service. They focus on building redundancy into each layer of their application using techniques such as clustering key servers, load balancing between services, and segregating infrastructure into multiple locations.

While these approaches take into account the complete (and often spectacular) loss of a system component, they address only one small part of building resilient systems. When a service crashes, it's easy to detect that it's no longer there, and the application can route around it. However, when a service is running slow, detecting that poor performance and routing around it is extremely difficult. Let's look at some reasons why:

- *Service degradation can start out as intermittent and then build momentum.* Service degradation might also occur only in small bursts. The first signs of failure might be a small group of users complaining about a problem until suddenly, the application container exhausts its thread pool and collapses completely.
- *Calls to remote services are usually synchronous and don't cut short a long-running call.* The application developer normally calls a service to perform an action and waits for the service to return. The caller has no concept of a timeout to keep the service call from hanging.
- *Applications are often designed to deal with complete failures of remote resources, not partial degradations.* Often, as long as the service has not entirely failed, an application will continue to call a poorly behaving service and won't fail fast. In this case, the calling application or service can degrade gracefully or, more likely, crash because of resource exhaustion. *Resource exhaustion* is where a limited resource, such as a thread pool or database connection, maxes out, and the calling client must wait for that resource to become available again.

What's insidious about problems caused by poorly performing remote services is that they are not only difficult to detect but can trigger a cascading effect that can ripple throughout an entire application ecosystem. Without safeguards in place, a single, poorly performing service can quickly take down multiple applications. Cloud-based, microservice-based applications are particularly vulnerable to these types of outages because these applications are composed of a large number of fine-grained, distributed services with different pieces of infrastructure involved in completing a user's transaction.

Resiliency patterns are one of the most critical aspects of the microservices architecture. This chapter will explain four resiliency patterns and how to use Spring Cloud and Resilience4j to implement them in our licensing service so that it can fail fast when needed.

7.1 What are client-side resiliency patterns?

Client-side resiliency software patterns focus on protecting a client of a remote resource (another microservice call or database lookup) from crashing when the remote resource fails because of errors or poor performance. These patterns allow the client to fail fast and not consume valuable resources, such as database connections and thread pools. They also prevent the problem of the poorly

performing remote service from spreading "upstream" to consumers of the client. In this chapter, we'll look at four client resiliency patterns. Figure 7.1 demonstrates how these patterns sit between the microservice service consumer and the microservice.

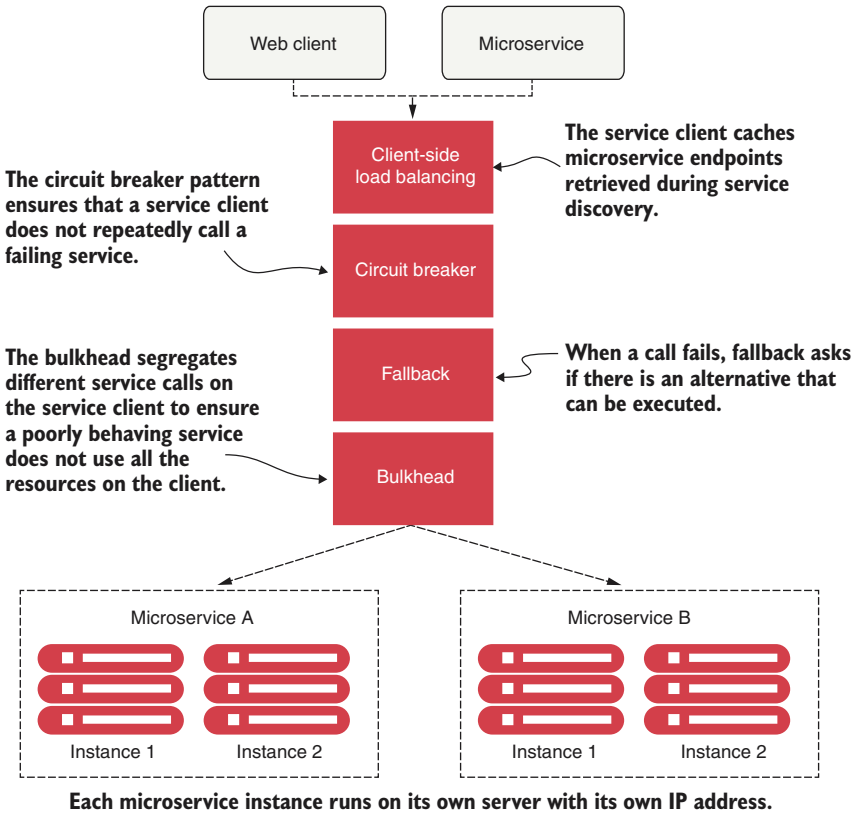

Figure 7.1 The four client resiliency patterns act as a protective buffer between a service consumer and the service.

These patterns (client-side load balancing, circuit breaker, fallback, and bulkhead) are implemented in the client (microservice) calling the remote resource. The implementation of these patterns logically sits between the client consuming the remote resources and the resource itself. Let's spend some time with each of these patterns.

7.1.1 Client-side load balancing

We introduced the client-side load balancing pattern in the last chapter when we talked about service discovery. Client-side load balancing involves having the client look up all of a service's individual instances from a service discovery agent (like Netflix Eureka) and then caching the physical location of said service instances.

When a service consumer needs to call a service instance, the client-side load balancer returns a location from the pool of service locations it maintains. Because the

client-side load balancer sits between the service client and the service consumer, the load balancer can detect if a service instance is throwing errors or behaving poorly. 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). Because we've already covered client-side load balancing with Spring Cloud Load Balancer in chapter 6, we won't go into any more detail on that in this chapter.

7.1.2 Circuit breaker

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.

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.

7.1.3 Fallback processing

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.

7.1.4 Bulkheads

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.

The same concept can be applied to a service that must interact with multiple remote resources. 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.

7.2 Why client resiliency matters

Although we've talked about these different patterns of client resiliency in the abstract, let's drill down to a more specific example of where these patterns can be applied. We'll walk through a typical scenario and see why client resiliency patterns are critical for implementing a microservice-based architecture running in the cloud.

Figure 7.2 shows a typical scenario involving the use of remote resources like a database and a remote service. This scenario doesn't contain any of the resiliency patterns that we have previously looked at, so it illustrates how an entire architecture (an ecosystem) can go down because of a single failing service. Let's take a look.

In the scenario in figure 7.2, three applications are communicating in one form or another with three different services. Applications A and B communicate directly with the licensing service. The licensing service retrieves data from a database and calls the organization service to do some work for it. The organization service retrieves data from a completely different database platform and calls out to another service, the inventory service, from a third-party cloud provider, whose service relies heavily on an internal Network Attached Storage (NAS) device to write data to a shared filesystem. Application C directly calls the inventory service.

Over the weekend, a network administrator made what they thought was a small tweak to the configuration on the NAS. This change appeared to work fine, but on Monday morning, reads to a particular disk subsystem began performing exceptionally slow.

The developers who wrote the organization service never anticipated slowdowns occurring with calls to the inventory service. They wrote their code so that the writes to their database and the reads from the service occur within the same transaction. When the inventory service starts running slowly, not only does the thread pool for requests to the inventory service start backing up, the number of database connections in the service container's connection pools becomes exhausted. These connections were held open because the calls to the inventory service never completed.

Now the licensing service starts running out of resources because it's calling the organization service, which is running slow because of the inventory service. Eventually, all three applications stop responding because they run out of resources while waiting for the requests to complete. This whole scenario could have been avoided if a

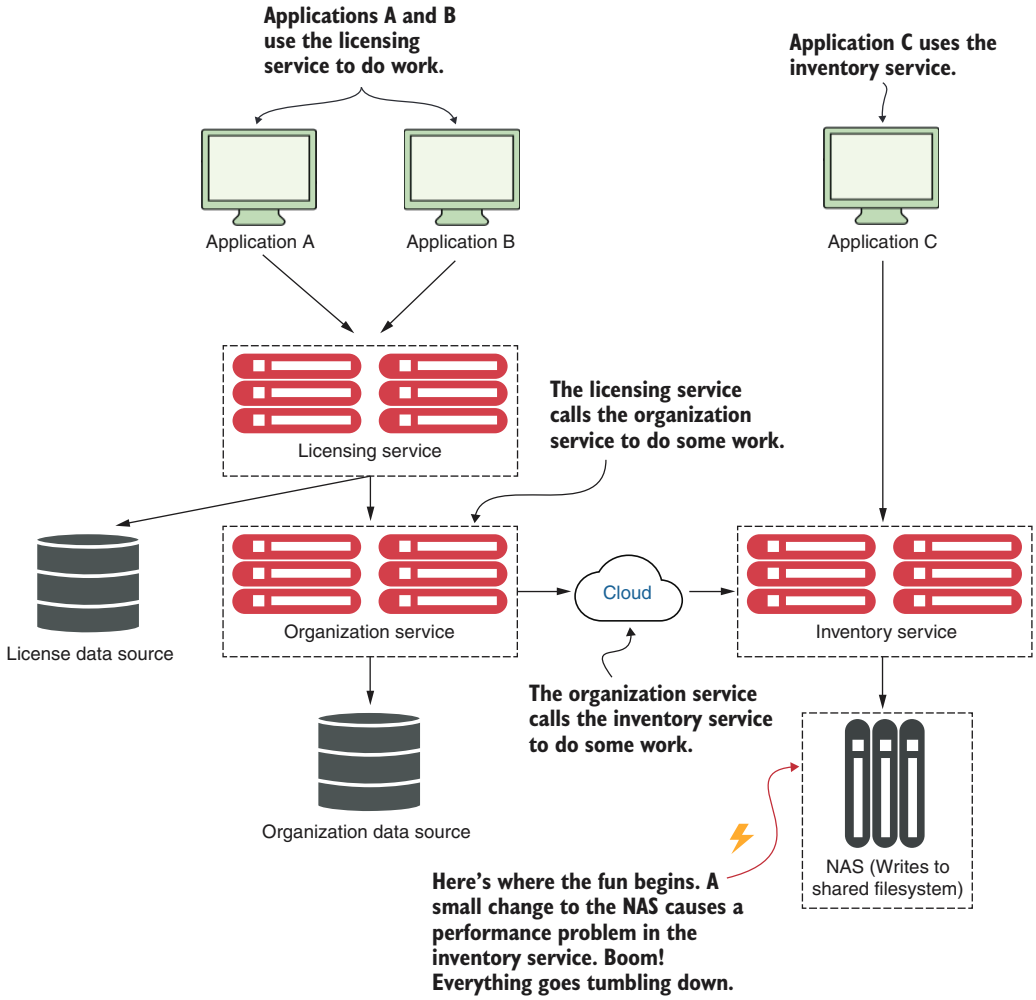

Figure 7.2 An application can be thought of as a graph of interconnected dependencies. If you don't manage the remote calls among them, one poorly behaving remote resource can bring down all the services in the graph.

circuit-breaker pattern had been implemented at each point where a distributed resource is called (either a call to the database or a call to the service).

In figure 7.2, if the call to the inventory service had been implemented with a circuit breaker, then when that service started performing poorly, the circuit breaker for that call would have tripped and failed fast without eating up a thread. If the organization service had multiple endpoints, only the endpoints that interacted with that specific call to the inventory service would be impacted. The rest of the organization service's functionality would still be intact and could fulfill user requests.

Remember, a circuit breaker acts as a middleman between the application and the remote service. In the scenario shown in figure 7.2, a circuit breaker implementation could have protected applications A, B, and C from completely crashing.

In figure 7.3, the licensing service never directly invokes the organization service. Instead, when the call is made, the licensing service delegates the actual invocation of the service to the circuit breaker, which takes the call and wraps it in a thread (usually managed in a thread pool) that's independent of the originating caller. By wrapping the call in a thread, the client is no longer directly waiting for the call to complete. Instead, the circuit breaker monitors the thread and can kill the call if the thread runs too long.

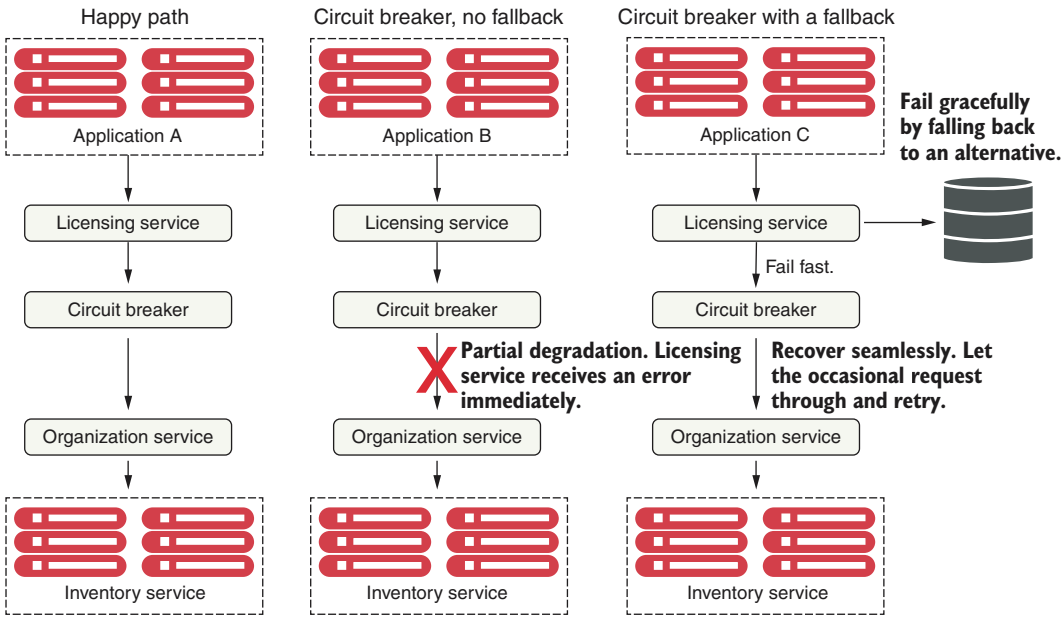

Figure 7.3 The circuit breaker trips and allows a misbehaving service call to fail quickly and gracefully.

Three scenarios are shown in figure 7.3. In the first scenario, "the happy path," the circuit breaker maintains a timer, and if the call to the remote service completes before the timer runs out, everything is good; the licensing service can continue its work.

In the second scenario, the partial degradation, the licensing service calls the organization service through the circuit breaker. This time, though, the organization service is running slow, so the circuit breaker kills the connection to the remote service if it doesn't complete before the timer on the thread maintained by the circuit breaker times out. The licensing service then returns an error from the call. The licensing service won't have its resources (its own thread or connection pool) tied up waiting for the organization service to complete.

If the call to the organization service times out, the circuit breaker starts tracking the number of failures that have occurred. If enough errors on the service occur within a specific time period, the circuit breaker now "trips" the circuit, and all calls to the organization service fail without making the call to it.

In the third scenario, the licensing service immediately knows there's a problem without having to wait for a timeout from the circuit breaker. It can then choose to either completely fail or to take action using an alternative set of code (a fallback). The organization service is given an opportunity to recover because the licensing service wasn't calling it when the circuit breaker tripped. This allows the organization service to have a bit of breathing room and helps to prevent the cascading shutdown that occurs when service degradation occurs.

The circuit breaker will occasionally let calls through to a degraded service. If those calls succeed enough times in a row, the circuit breaker resets itself. 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.

In a sizeable cloud-based application with hundreds of services, this graceful recovery is critical because it can significantly cut down the amount of time needed to restore a service. It also significantly lessens the risk of a "tired" operator or application engineer causing more problems, allowing the circuit breaker to intervene directly (restarting a failed service) in the restoration of the service.

Before Resilience4j, we worked with Hystrix, one of the most common Java libraries to implement the resiliency patterns in microservices. Because Hystrix is now in maintenance mode, which means that new features are no longer included, one of the most recommended libraries to use as a substitute is Resilience4j. That's the main reason why we chose it for demonstration purposes in this chapter. With Resilience4j, we have similar (and some additional) benefits that we'll see throughout this chapter.

7.3 Implementing 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

With Resilience4j, we can apply several patterns to the same method call by defining the annotations for that method. For example, if we want to limit the number of outgoing calls with the bulkhead and circuit breaker patterns, we can define the `@CircuitBreaker` and the `@Bulkhead` annotations for the method. It is important to note that Resilience4j's retry order is as follows:

```
Retry ( CircuitBreaker ( RateLimiter ( TimeLimiter ( Bulkhead ( Function ) )
➥ ) ) )
```

`Retry` is applied (if needed) at the end of the call. This is valuable to remember when trying to combine patterns, but we can also use the patterns as individual features.

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. In the next several sections, we'll cover how to

- Configure the licensing service's Maven build file (pom.xml) to include the Spring Boot/Resilience4j wrappers
- Use Spring Boot/Resilience4j annotations to wrapper remote calls with the circuit breaker, retry, rate limit, and bulkhead patterns
- Customize the individual circuit breakers on a remote resource to use custom timeouts for each call
- Implement a fallback strategy in the event a circuit breaker has to interrupt a call or the call fails
- Use individual thread pools in our service to isolate service calls and build bulkheads between different remote resources

7.4 Setting up the licensing service to use Spring Cloud and Resilience4j

To begin our exploration of Resilience4j, we need to set up our project pom.xml to import the dependencies. To achieve that, we will take the licensing service that we are building and modify its pom.xml by adding the Maven dependencies for Resilience4j. The following listing indicates how to do this.

Listing 7.1 Adding Resilience4j dependency in pom.xml of the licensing service

```
<properties>
     ...
     <resilience4j.version>1.5.0</resilience4j.version>
</properties>
<dependencies>
    //Part of pom.xml omitted for conciseness
    ...
    <dependency>
        <groupId>io.github.resilience4j</groupId>
        <artifactId>resilience4j-spring-boot2</artifactId>
        <version>${resilience4j.version}</version>
    </dependency>

    <dependency>
        <groupId>io.github.resilience4j</groupId>
        <artifactId>resilience4j-circuitbreaker</artifactId>
        <version>${resilience4j.version}</version>
    </dependency>
    <dependency>
        <groupId>io.github.resilience4j</groupId>
        <artifactId>resilience4j-timelimiter</artifactId>
        <version>${resilience4j.version}</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-aop</artifactId>
    </dependency>
//Rest of pom.xml omitted for conciseness
...
</dependencies>
```

The `<dependency>` tag with the `resilience4j-spring-boot2` artifact tells Maven to pull down the Resilience4j Spring Boot library, which allows us to use custom pattern annotations. The dependencies with the `resilience4j-circuitbreaker` and `resilience4j-timelimiter` artifacts contain all the logic to implement the circuit breaker and rate limiter. The final dependency is the `spring-boot-starter-aop`. We need this library in our project because it allows Spring AOP aspects to run.

Aspect-oriented programming (AOP) is a programming paradigm that aims to increase modularity by allowing us to separate parts of the program that affect other parts of the system; in other words, cross-cutting concerns. AOP adds new behaviors to

existing code without modifying the code itself. Now that we've added the Maven dependencies, we can go ahead and begin our Resilience4j implementation using the licensing and organization services we built in previous chapters.

> **NOTE** In case you didn't follow the previous chapter's code listings, you can download the code created in chapter 6 from the following link: https://github.com/ihuaylupo/manning-smia/tree/master/chapter6/Final.

7.5 Implementing a circuit breaker

To understand circuit breakers, we can think of electrical systems. What happens when there is too much current passing through a wire in an electrical system? As you'll recall, if the circuit breaker detects a problem, it breaks the connection with the rest of the system, avoiding further damage to other components. The same happens in our code architecture as well.

What we want to achieve with circuit breakers in our code is to monitor remote calls and avoid long waits on services. In these scenarios, the circuit breaker is in charge of killing those connections and monitoring if there are more failing or poorly behaving calls. This pattern then implements a fast fail and prevents future requests to a failing remote resource. In Resilience4j, the circuit breaker is implemented via a finite state machine with three normal states. Figure 7.4 shows the different states and the interaction between them.

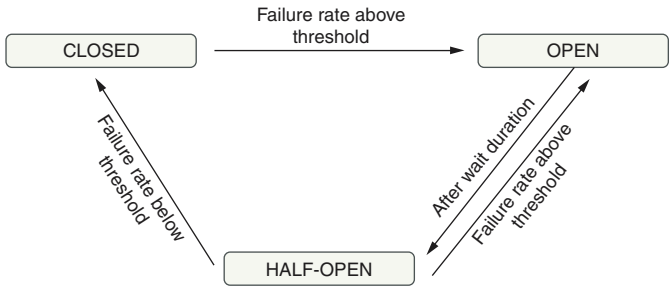

Figure 7.4 Resilience4j circuit breaker states: closed, open, and half-open

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 shows a ring buffer with 12 results.

To calculate a failure rate, the ring must be full. For example, in the previous scenario, at least 12 calls must be evaluated before the failure rate can be calculated. If only 11 requests are evaluated, the circuit breaker will not change to an open state even if all 11 calls fail. Note that the circuit breaker only opens when the failure rate is above the configurable threshold.

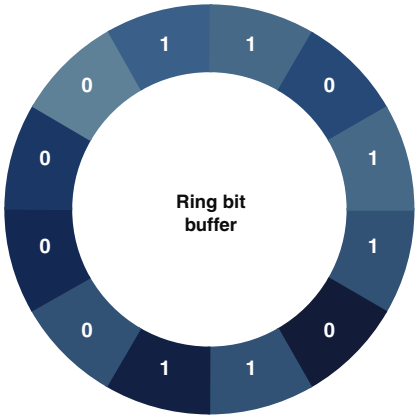

Figure 7.5 Resilience4j circuit breaker ring bit buffer with 12 results. This ring contains 0 for all the successful requests and 1 when it fails to receive a response from the invoked service.

When the circuit breaker is in the open state, all calls are rejected during a configurable time, and the circuit breaker throws a `CallNotPermittedException`. Once the configuration time expires, the circuit breaker changes to the half-open state and allows a number of requests to see if the service is still unavailable.

In the half-open state, the circuit breaker uses another configurable ring bit buffer to evaluate the failure rate. If this new failure rate is above the configured threshold, the circuit breaker changes back to open; if it is below or equal to the threshold, it changes back to closed. This might be somewhat confusing, but just remember, in the open state the circuit breaker rejects, and in the closed state, the circuit breaker accepts all the requests.

Also, in the Resilience4j circuit breaker pattern, you can define the following additional states. It is important to note that the only way to exit from the following states is to reset the circuit breaker or trigger a state transition:

- *DISABLED*—Always allow access
- *FORCED_OPEN*—Always deny access

NOTE A detailed description of these two additional states is beyond the scope of this book. If you want to know more about these states, we recommend you read the official Resilience4j documentation at https://resilience4j.readme.io/v0.17.0/docs/circuitbreaker.

In this section, we'll look at implementing Resilience4j in two broad categories. In the first category, we'll wrap all calls to our database in the licensing and organization service with a Resilience4j circuit breaker. We will then wrap the interservice calls between the licensing and organization services using Resilience4j. While these are two different categories of calls, we'll see that, with the use of Resilience4j, these calls will be exactly the same. Figure 7.6 shows the remote resources that we're going to wrap within a Resilience4j circuit breaker.

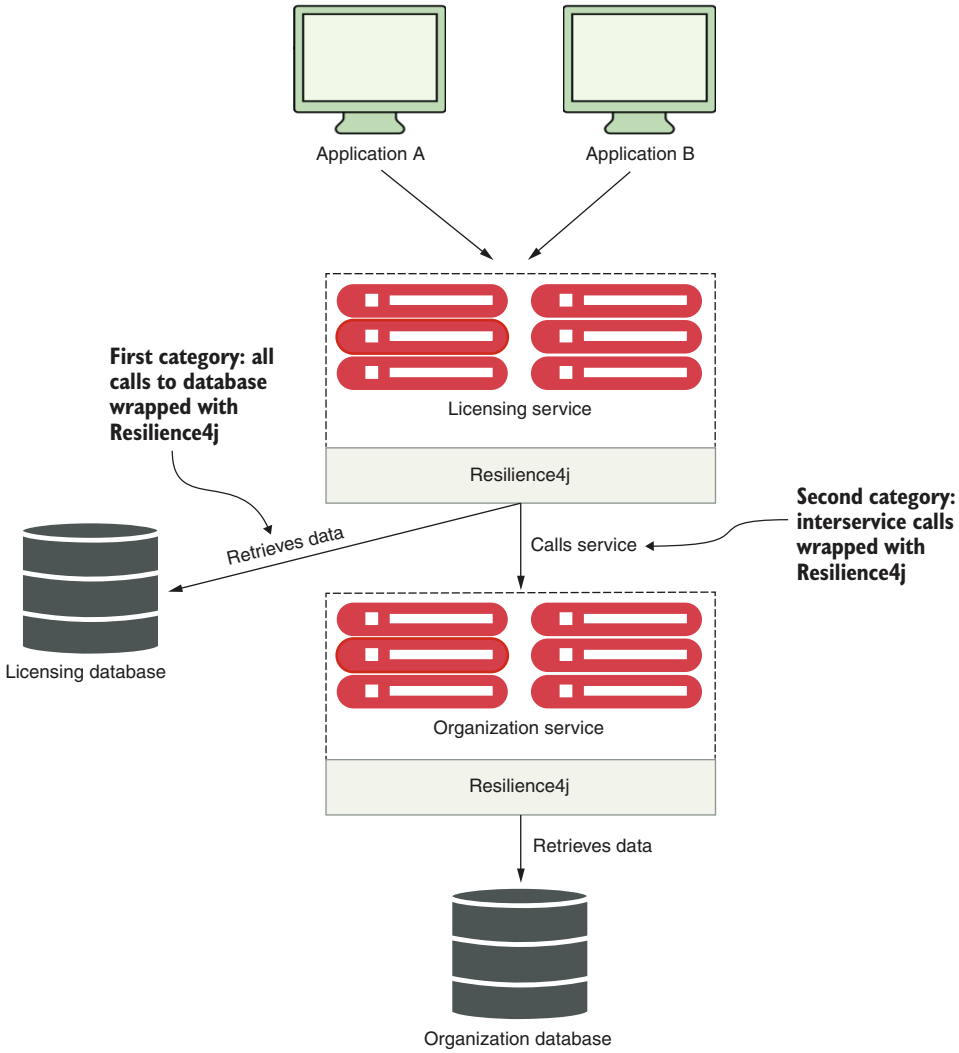

Figure 7.6 Resilience4j sits between each remote resource call and protects the client. It doesn't matter if the remote resource calls a database or a REST-based service.

Let's start our Resilience4j discussion by showing how to wrap the retrieval of licensing service data from the licensing database using a synchronous circuit breaker. With a synchronous call, the licensing service retrieves its data but waits for the SQL statement to complete or for a circuit breaker timeout before it continues processing.

Resilience4j and Spring Cloud use `@CircuitBreaker` to mark the Java class methods managed by a Resilience4j circuit breaker. When the Spring framework sees this annotation, it dynamically generates a proxy that wraps the method and manages all calls to that method through a thread pool specifically set aside to handle remote calls. Let's add `@CircuitBreaker` to the method `getLicensesByOrganization` in the

src/main/java/com/optimagrowth/license/service/LicenseService.java class file as shown in the following listing.

Listing 7.2 Wrapping a remote resource call with a circuit breaker

```
//Part of LicenseService.java omitted for conciseness

@CircuitBreaker(name = "licenseService")

public List<License> getLicensesByOrganization(String organizationId) {
    return licenseRepository.findByOrganizationId(organizationId);
}
```

@CircuitBreaker wrapper for getLicensesByOrganization() with a Resilience4j circuit breaker

NOTE If you look at the code in listing 7.2 in the source code repository, you'll see several more parameters on the `@CircuitBreaker` annotation. We'll get into those parameters later in the chapter, but the code in listing 7.2 uses `@CircuitBreaker` with all its default values.

This doesn't look like a lot of code, and it's not, but there is a lot of functionality inside this one annotation. With the use of the `@CircuitBreaker` annotation, any time the `getLicensesByOrganization()` method is called, the call is wrapped with a Resilience4j circuit breaker. The circuit breaker interrupts any failed attempt to call the `getLicensesByOrganization()` method.

This code example would be boring if the database was working correctly. Let's simulate the `getLicensesByOrganization()` method running into a slow or timed out database query. The following listing demonstrates this.

Listing 7.3 Purposely timing out a call to the licensing service database

```
//Part of LicenseService.java omitted for conciseness

private void randomlyRunLong(){
   Random rand = new Random();
   int randomNum = rand.nextInt(3) + 1;
   if (randomNum==3) sleep();
}

private void sleep(){
   try {
     Thread.sleep(5000);
     throw new java.util.concurrent.TimeoutException();
   } catch (InterruptedException e) {
     logger.error(e.getMessage());
   }
}

@CircuitBreaker(name = "licenseService")
public List<License> getLicensesByOrganization(String organizationId) {
   randomlyRunLong();
   return licenseRepository.findByOrganizationId(organizationId);
}
```

Gives us a one-in-three chance of a database call running long

Sleeps for 5000 ms (5 s) and then throws a TimeoutException

If you enter the `http://localhost:8080/v1/organization/e6a625cc-718b-48c2-ac76-1dfdff9a531e/license/` endpoint enough times in Postman, you'll see the following error message returned from the licensing service:

```
{
  "timestamp": 1595178498383,
  "status": 500,
  "error": "Internal Server Error",
  "message": "No message available",
  "path": "/v1/organization/e6a625cc-718b-48c2-ac76-1dfdff9a531e/
           license/"
}
```

If we keep executing the failing service, the ring bit buffer will eventually fill, and we should receive the error shown in figure 7.7.

Figure 7.7 A circuit breaker error indicates the circuit breaker is now in the open state.

Now that we have our circuit breaker working for the licensing service, let's continue by setting up the circuit breaker for the organization microservice.

7.5.1 Adding the circuit breaker to the organization service

The beauty of using method-level annotations for tagging calls with the circuit breaker behavior is that it's the same annotation whether we're accessing a database or calling a microservice. For instance, in our licensing service, we need to look up the name of the organization associated with the license. If we want to wrap our call to the organization service with a circuit breaker, it's as simple as breaking down a `RestTemplate` call into a method and annotating it with `@CircuitBreaker` like this:

```
@CircuitBreaker(name = "organizationService")
private Organization getOrganization(String organizationId) {
   return organizationRestClient.getOrganization(organizationId);
}
```

NOTE Although using `@CircuitBreaker` is easy to implement, we do need to be careful about using the default values of this annotation. We highly recommend that you always analyze and set the configuration that best suits your needs.

To view the default values of your circuit breaker, you can select the following URL in Postman: http://localhost:<service_port>/actuator/health. By default, the circuit breaker exposes the configuration in the Spring Boot Actuator health service.

7.5.2 Customizing the circuit breaker

In this section, we will answer one of the most common questions developers ask when using Resilience4j: how do we customize the Resilience4j circuit breaker? This is easily accomplished by adding some parameters to the application.yml, boostrap.yml, or service configuration file located in the Spring Config Server repository. The following listing demonstrates how to customize the circuit breaker pattern in the boostrap.yml for both the licensing and the organization services.

Listing 7.4 Customizing the circuit breaker

Licensing service instance configuration. (The name given to the circuit breaker in the annotation.)

```
//Part of bootstrap.yml omitted for conciseness

resilience4j.circuitbreaker:
  instances:
    licenseService:
      registerHealthIndicator: true
      ringBufferSizeInClosedState: 5
      ringBufferSizeInHalfOpenState: 3
      waitDurationInOpenState: 10s
      failureRateThreshold: 50
      recordExceptions:
        - org.springframework.web.client.HttpServerErrorException
        - java.io.IOException
        - java.util.concurrent.TimeoutException
        - org.springframework.web.client.ResourceAccessException

    organizationService:
      registerHealthIndicator: true
      ringBufferSizeInClosedState: 6
      ringBufferSizeInHalfOpenState: 4
      waitDurationInOpenState: 20s
      failureRateThreshold: 60
```

Indicates whether to expose the configuration over the health endpoint

Sets the ring buffer size in the half-open state

Sets the wait duration for the open state

Sets the failure rate threshold percentage

Sets the ring buffer size at the closed state

Sets the exceptions that should be recorded as failures

Organization service instance configuration. (The name given to the circuit breaker in the annotation.)

Resilience4j allows us to customize the behavior of the circuit breakers through the application's properties. We can configure as many instances as we want, and each instance can have a different configuration. Listing 7.4 contains the following configuration settings:

- `ringBufferSizeInClosedState`—Sets the size of the ring bit buffer when the circuit breaker is in the closed state. The default value is 100.
- `ringBufferSizeInHalfOpenState`—Sets the size of the ring bit buffer when the circuit breaker is in the half-open state. The default value is 10.
- `waitDurationInOpenState`—Sets the time the circuit breaker should wait before changing the status from open to half-open. The default value is 60,000 ms.
- `failureRateThreshold`—Configures the percentage of the failure rate threshold. Remember, when the failure rate is greater than or equal to this threshold, the circuit breaker changes to the open state and starts short-circuiting calls. The default value is 50.
- `recordExceptions`—Lists the exceptions that will be considered as failures. By default, all exceptions are recorded as failures.

In this book, we will not cover all of the Resilience4j circuit breaker parameters. If you want to know more about its possible configuration parameters, we recommend you visit the following link: https://resilience4j.readme.io/docs/circuitbreaker.

7.6 Fallback processing

Part of the beauty of the circuit breaker pattern is that because a "middleman" is between the consumer of a remote resource and the resource itself, we have the opportunity to intercept a service failure and choose an alternative course of action to take.

In Resilience4j, this is known as a fallback strategy and is easily implemented. Let's see how to build a simple fallback strategy for our licensing service that returns a licensing object that says no licensing information is currently available. The following listing demonstrates this.

Listing 7.5 Implementing a fallback in Resilience4j

```
//Part of LicenseService.java omitted for conciseness

@CircuitBreaker(name= "licenseService",
      fallbackMethod= "buildFallbackLicenseList")
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){
   List<License> fallbackList = new ArrayList<>();
   License license = new License();
```

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

Returns a hardcoded value in the fallback method

```
    license.setLicenseId("0000000-00-00000");
    license.setOrganizationId(organizationId);
    license.setProductName(
       "Sorry no licensing information currently available");
    fallbackList.add(license);
    return fallbackList;
}
```

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.

In the example in listing 7.5, the fallback method, `buildFallbackLicenseList()`, is simply constructing a single `License` object containing dummy information. We could have our fallback method read this data from an alternative data source, but for demonstration purposes, we're going to construct a list that can be returned by our original function call.

On fallbacks

Here are a few things to keep in mind as you determine whether you want to implement a fallback strategy:

- *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.

Now that we have our fallback method in place, let's go ahead and call our endpoint again. This time when we select it in Postman and encounter a timeout error (remember we have a one-in-three chance), we shouldn't get an exception back from the service call. Instead, the dummy license values will return as in figure 7.8.

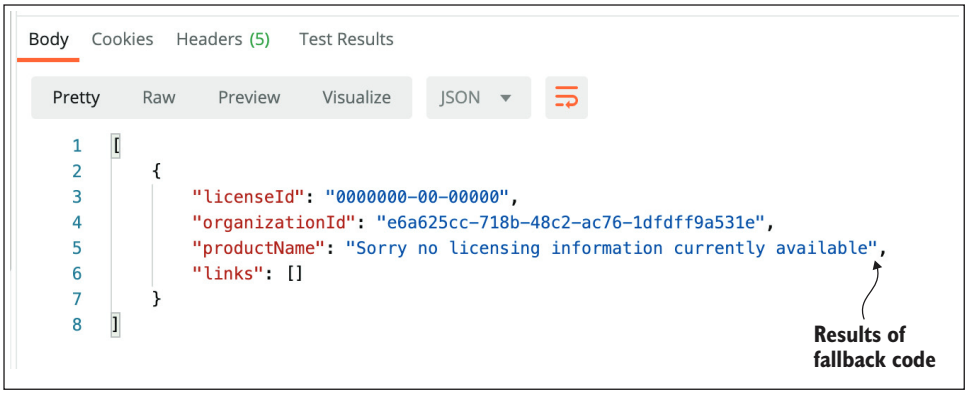

Figure 7.8 Your service invocation using a Resilience4j fallback

7.7 Implementing the 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.

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.

Resilience4j, by default, uses the semaphore bulkhead type. Figure 7.9 illustrates this type.

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.

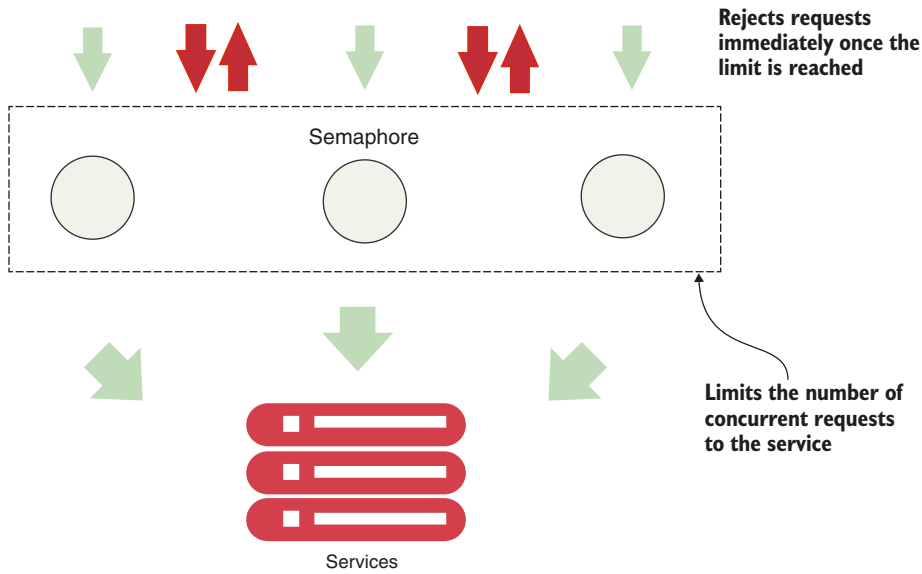

Figure 7.9 The default Resilience4j bulkhead type is the semaphore approach.

Fortunately, Resilience4j provides an easy-to-use mechanism for creating bulkheads between different remote resource calls. Figure 7.10 shows what managed resources look like when they're segregated into their own bulkheads.

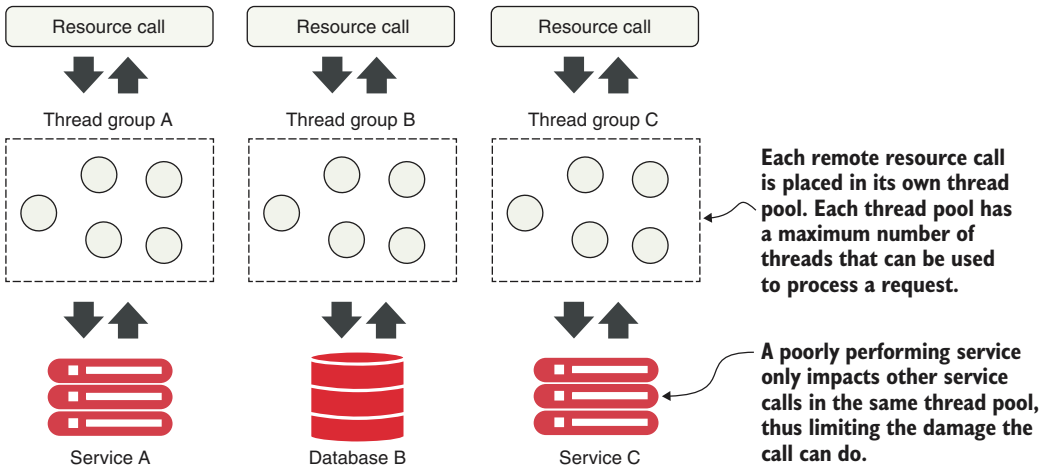

Figure 7.10 A Resilience4j command tied to segregated thread pools

To implement the bulkhead pattern in Resilience4j, we need to use an additional configuration to combine it with `@CircuitBreaker`. Let's look at some code that does this:

- Sets up a separate thread pool for the `getLicensesByOrganization()` call
- Creates the bulkhead configuration in the bootstrap.yml file
- With the semaphore approach, sets `maxConcurrentCalls` and `maxWaitDuration`
- With the thread pool approach, sets `maxThreadPoolSize`, `coreThreadPoolSize`, `queueCapacity`, and `keepAliveDuration`

The following listing shows the bootstrap.yml for the licensing service with these bulkhead configuration parameters.

Listing 7.6 Configuring the bulkhead pattern for the licensing service

```
//Part of boostrap.yml omitted for conciseness

resilience4j.bulkhead:
  instances:
    bulkheadLicenseService:
      maxWaitDuration: 10ms
      maxConcurrentCalls: 20

resilience4j.thread-pool-bulkhead:
  instances:
    bulkheadLicenseService:
      maxThreadPoolSize: 1
      coreThreadPoolSize: 1
      queueCapacity: 1
      keepAliveDuration: 20ms
```

The maximum amount of time to block a thread

The maximum number of concurrent calls

The maximum number of threads in the thread pool

The core thread pool size

The queue's capacity

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

Resilience4j also lets us customize the behavior of the bulkhead patterns through the application's properties. Like the circuit breaker, we can create as many instances as we want, and each instance can have different configurations. Listing 7.6 contains the following properties:

- `maxWaitDuration`—Sets the maximum amount of time to block a thread when entering a bulkhead. The default value is 0.
- `maxConcurrentCalls`—Sets the maximum number of concurrent calls allowed by the bulkhead. The default value is 25.
- `maxThreadPoolSize`—Sets the maximum thread pool size. The default value is `Runtime.getRuntime().availableProcessors()`.
- `coreThreadPoolSize`—Sets the core thread pool size. The default value is `Runtime.getRuntime().availableProcessors()`.
- `queueCapacity`—Sets the capacity of the queue. The default value is 100.
- `KeepAliveDuration`—Sets the maximum time that idle threads will wait for new tasks before terminating. This happens when the number of threads is higher than the number of core threads. The default value is 20 ms.

What's the proper sizing for a custom thread pool? To answer that question, you can use the following formula:

*(requests per second at peak when the service is healthy * 99th percentile latency in seconds) + small amount of extra threads for overhead*

We often don't know the performance characteristics of a service until it functions under a load. A key indicator that the thread pool properties need to be adjusted is when a service call is in the process of timing out, even if the targeted remote resource is healthy. The following listing demonstrates how to set up a bulkhead around all calls surrounding the lookup of licensing data from our licensing service.

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")

public List<License> getLicensesByOrganization(
                     String organizationId) throws TimeoutException {
   logger.debug("getLicensesByOrganization Correlation id: {}",
        UserContextHolder.getContext().getCorrelationId());
   randomlyRunLong();
   return licenseRepository.findByOrganizationId(organizationId);
}
```

Sets the instance name and fallback method for the bulkhead pattern

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")
```

7.8 Implementing the retry pattern

As its name implies, the retry pattern is responsible for retrying attempts to communicate with a service when that service initially fails. The key concept behind this pattern is to provide a way to get the expected response by trying to invoke the same service one or more times despite the failure (for example, a network disruption). For this pattern, we must specify the number of retries for a given service instance and the interval we want to pass between each retry.

Like the circuit breaker, Resilience4j lets us specify which exceptions we want to retry and not to retry. The following listing shows the bootstrap.yml for the licensing service, where it contains the retry configuration parameters.

Listing 7.8 Configuring the retry pattern in the bootstrap.yml

```
//Part of boostrap.yml omitted for conciseness

resilience4j.retry:
  instances:
    retryLicenseService:
      maxRetryAttempts: 5
      waitDuration: 10000
      retry-exceptions:
        - java.util.concurrent.TimeoutException
```

The maximum number of retry attempts

The wait duration between the retry attempts

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. The default value is empty. For this book, we only use these three parameters, but you can also set the following:

- `intervalFunction`—Sets a function to update the waiting interval after a failure.
- `retryOnResultPredicate`—Configures a predicate that evaluates if a result should be retried. This predicate should return `true` if we want to retry.
- `retryOnExceptionPredicate`—Configures a predicate that evaluates if an exception should be retried. Same as the previous predicate; we must return `true` if we want to retry.
- `ignoreExceptions`—Sets a list of error classes that are ignored and will not be retried. The default value is empty.

The following listing demonstrates how to set up the retry pattern around all calls surrounding the lookup of licensing data from our licensing service.

Listing 7.9 Creating a bulkhead around the `getLicensesByOrganization()` method

```
//Part of LicenseService.java omitted for conciseness

@CircuitBreaker(name= "licenseService",
                fallbackMethod="buildFallbackLicenseList")
@Retry(name = "retryLicenseService",
                fallbackMethod=
                   "buildFallbackLicenseList")
@Bulkhead(name= "bulkheadLicenseService",
                fallbackMethod="buildFallbackLicenseList")

public List<License> getLicensesByOrganization(String organizationId)
         throws TimeoutException {
   logger.debug("getLicensesByOrganization Correlation id: {}",
```

Sets the instance name and fallback method for the retry pattern

```
UserContextHolder.getContext().getCorrelationId());
         randomlyRunLong();
return licenseRepository.findByOrganizationId(organizationId);
}
```

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.

7.9 Implementing the rate limiter pattern

This retry pattern stops overloading the service with more calls than it can consume in a given timeframe. This is an imperative technique to prepare our API for high availability and reliability.

> **NOTE** In up-to-date cloud architectures, it is a good option to include autoscaling, but we do not cover that topic in this book.

Resilience4j provides two implementations for the rate limiter pattern: `AtomicRateLimiter` and `SemaphoreBasedRateLimiter`. The default implementation for the `RateLimiter` is the `AtomicRateLimiter`.

The `SemaphoreBasedRateLimiter` is the simplest. This implementation is based on having one `java.util.concurrent.Semaphore` store the current permissions. In this scenario, all the user threads will call the method `semaphore.tryAcquire` to trigger a call to an additional internal thread by executing `semaphore.release` when a new `limitRefreshPeriod` starts.

Unlike the `SemaphoreBasedRate`, the `AtomicRateLimiter` does not need thread management because the user threads themselves execute all the permissions logic. The `AtomicRateLimiter` splits all nanoseconds from the start into cycles, and each cycle duration is the refresh period (in nanoseconds). Then at the beginning of each cycle, we should set the active permissions to limit the period. To better understand this approach, let's look at the following settings:

- `ActiveCycle`—The cycle number used by the last call
- `ActivePermissions`—The count of available permissions after the last call
- `NanosToWait`—The count of nanoseconds to wait for permission for the last call

This implementation contains some tricky logic. To better understand it, we can consider the following Resilience4j statements for this pattern:

- Cycles are equal time pieces.
- If the available permissions are not enough, we can perform a permission reservation by decreasing the current permissions and calculating the time we need to wait for the permission to appear. Resilience4j allows this by defining the number of calls that are allowed during a time period (`limitForPeriod`); how often permissions are refreshed (`limitRefreshPeriod`); and how long a thread can wait to acquire permissions (`timeoutDuration`).

For this pattern, we must specify the timeout duration, the limit refresh, and the limit for the period. The following listing shows the bootstrap.yml for the licensing service, which contains the retry configuration parameters.

Listing 7.10 Configuring the retry pattern in the bootstrap.yml

```
//Part of boostrap.yml omitted for conciseness

resilience4j.ratelimiter:
  instances:
    licenseService:
      timeoutDuration: 1000ms
      limitRefreshPeriod: 5000
      limitForPeriod: 5
```

Defines the time a thread waits for permission

Defines the number of permissions available during a limit refresh period

Defines the period of a limit refresh

The first parameter, `timeoutDuration`, lets us define the time a thread waits for permission; the default value for this parameter is 5 s (seconds). The second parameter, `limitRefreshPeriod`, enables us to set the period that limits the refresh. After each period, the rate limiter resets the permissions count back to the `limitForPeriod` value. The default value for the `limitRefreshPeriod` is 500 ns (nanoseconds).

The final parameter, `limitForPeriod`, lets us set the number of permissions available during one refresh period. The default value for the `limitForPeriod` is 50. The following listing demonstrates how to set up the retry pattern around all calls surrounding the lookup of licensing data from our licensing service.

Listing 7.11 Creating a bulkhead around `getLicensesByOrganization()`

```
//Part of LicenseService.java omitted for conciseness

@CircuitBreaker(name= "licenseService",
     fallbackMethod= "buildFallbackLicenseList")
@RateLimiter(name = "licenseService",
     fallbackMethod = "buildFallbackLicenseList")
@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);
}
```

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.

7.10 ThreadLocal and Resilience4j

In this section, we will define some values in `ThreadLocal` to see if they are propagated throughout the methods using Resilience4J annotations. Remember, Java `ThreadLocal` allows us to create variables that can be read and written to only by the same threads. When we work with threads, all the threads of a specific object share its variables, making these threads unsafe. The most common way to make them thread-safe in Java is to use synchronization. But if we want to avoid synchronization, we can also use `ThreadLocal` variables.

Let's look at a concrete example. Often in a REST-based environment, we want to pass contextual information to a service call that will help us operationally manage the service. For example, we might pass a correlation ID or authentication token in the HTTP header of the REST call that can then be propagated to any downstream service calls. The correlation ID allows us to have a unique identifier that can be traced across multiple service calls in a single transaction.

To make this value available anywhere within our service call, we might use a Spring `Filter` class to intercept every call in our REST service. It can then retrieve this information from the incoming HTTP request and store this contextual information in a custom `UserContext` object. Then, anytime our code needs to access this value in our REST service call, our code can retrieve the `UserContext` from the `ThreadLocal` storage variable and read the value. Listing 7.12 shows an example of a Spring filter that we can use in our licensing service.

> **NOTE** You can find this code at /licensing-service/src/main/java/com/optimagrowth/license/utils/UserContextFilter.java in the source code for chapter 7. Here's the repository link: https://github.com/ihuaylupo/manning-smia/tree/master/chapter7.

Listing 7.12 The `UserContextFilter` parsing the HTTP header and retrieving data

```
package com.optimagrowth.license.utils;
...
//Imports removed for conciseness

@Component
public class UserContextFilter implements Filter {
    private static final Logger logger =
          LoggerFactory.getLogger(UserContextFilter.class);
```

```
    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse
          servletResponse, FilterChain filterChain) throws IOException,
          ServletException {

    HttpServletRequest httpServletRequest =
                        (HttpServletRequest) servletRequest;

    UserContextHolder.getContext().setCorrelationId(
              httpServletRequest.getHeader(
                     UserContext.CORRELATION_ID));
    UserContextHolder.getContext().setUserId(
              httpServletRequest.getHeader(
                     UserContext.USER_ID));
    UserContextHolder.getContext().setAuthToken(
              httpServletRequest.getHeader(
                     UserContext.AUTH_TOKEN));
    UserContextHolder.getContext().setOrganizationId(
              httpServletRequest.getHeader(
                     UserContext.ORGANIZATION_ID));

    filterChain.doFilter(httpServletRequest, servletResponse);
    }
    ...
    //Rest of UserContextFilter.java omitted for conciseness
}
```

Retrieves the values set in the HTTP header of the call to a UserContext. These are then stored in UserContextHolder.

The `UserContextHolder` class stores the `UserContext` in a `ThreadLocal` class. Once it's stored in `ThreadLocal`, any code that's executed for a request will use the `UserContext` object stored in the `UserContextHolder`.

The following listing shows the `UserContextHolder` class. You can find this class in the /licensing-service/src/main/java/com/optimagrowth/license/utils/UserContextHolder.java class file.

Listing 7.13 All `UserContext` data is managed by `UserContextHolder`

```
...
//Imports omitted for conciseness

public class UserContextHolder {
    private static final ThreadLocal<UserContext> userContext
          = new ThreadLocal<UserContext>();

    public static final UserContext getContext(){
        UserContext context = userContext.get();

        if (context == null) {
            context = createEmptyContext();
            userContext.set(context);

        }
        return userContext.get();
```

Stores UserContext in a static ThreadLocal variable

Retrieves the UserContext object for consumption

```
    }

    public static final void setContext(UserContext context) {
        userContext.set(context);
    }

    public static final UserContext createEmptyContext(){
        return new UserContext();
    }
}
```

NOTE We must be careful when we work directly with `ThreadLocal`. An incorrect development inside `ThreadLocal` can lead to memory leaks in our application.

The `UserContext` is a POJO class that contains all the specific data we want to store in the `UserContextHolder`. The following listing shows the content of this class. You can find this class in /licensing-service/src/main/java/com/optimagrowth/license/utils/UserContext.java.

Listing 7.14 Creating a `UserContext`

```
...
//Imports omitted for conciseness

@Component
public class UserContext {
    public static final String CORRELATION_ID = "tmx-correlation-id";
    public static final String AUTH_TOKEN     = "tmx-auth-token";
    public static final String USER_ID        = "tmx-user-id";
    public static final String ORGANIZATION_ID = "tmx-organization-id";

    private String correlationId= new String();
    private String authToken= new String();
    private String userId = new String();
    private String organizationId = new String();

    public String getCorrelationId() { return correlationId;}
    public void setCorrelationId(String correlationId) {
        this.correlationId = correlationId;
    }

    public String getAuthToken() {
        return authToken;
    }

    public void setAuthToken(String authToken) {
        this.authToken = authToken;
    }

    public String getUserId() {
        return userId;
    }
```

```
    public void setUserId(String userId) {
        this.userId = userId;
    }
    public String getOrganizationId() {
        return organizationId;
    }
    public void setOrganizationId(String organizationId) {
        this.organizationId = organizationId;
    }
}
```

The last step that we need to do to finish our example is to add the logging instruction to the `LicenseController.java` class, which is found in com/optimagrowth/license/controller/LicenseController.java. The following listing shows how.

Listing 7.15 Adding `logger` to the `LicenseController` `getLicenses()` method

```
//Some code omitted for conciseness
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

@RestController
@RequestMapping(value="v1/organization/{organizationId}/license")
public class LicenseController {
    private static final Logger logger =
                      LoggerFactory.getLogger(LicenseController.class);

    //Some code removed for conciseness
10

    @RequestMapping(value="/",method = RequestMethod.GET)
    public List<License> getLicenses( @PathVariable("organizationId")
                      String organizationId) {
       logger.debug("LicenseServiceController Correlation id: {}",
               UserContextHolder.getContext().getCorrelationId());
       return licenseService.getLicensesByOrganization(organizationId);
    }
}
```

At this point, we should have several log statements in our licensing service. We've already added logging to the following licensing service classes and methods:

- `doFilter()` in com/optimagrowth/license/utils/UserContextFilter.java.
- `getLicenses()` in com/optimagrowth/license/controller/LicenseController.java.
- `getLicensesByOrganization()` in com/optimagrowth/license/service/LicenseService.java. This method is annotated with `@CircuitBreaker`, `@Retry`, `@Bulkhead`, and `@RateLimiter`.

To execute our example, we'll call our service, passing in a correlation ID using an HTTP header `tmx-correlation-id` and a value of `TEST-CORRELATION-ID`. Figure 7.11 shows an HTTP GET call in Postman.

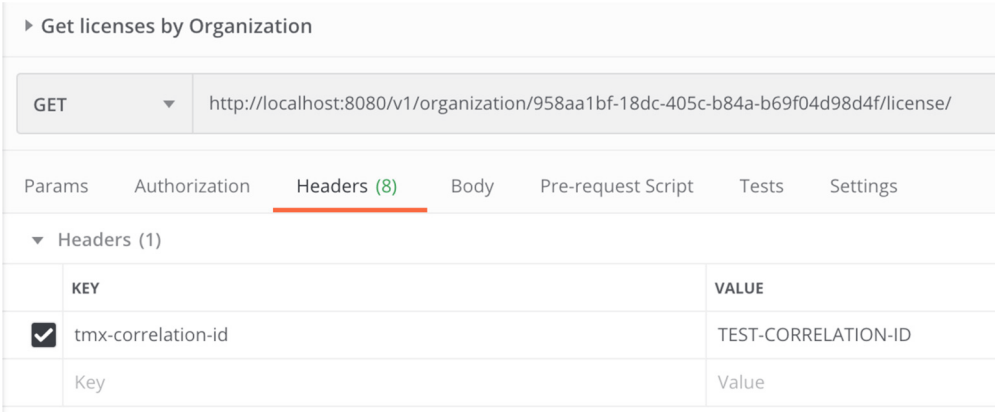

Figure 7.11 Adding a correlation ID to the licensing service HTTP header

Once this call is submitted, we should see three log messages in the console, writing out the passed-in correlation ID as it flows through the `UserContext`, `License-Controller`, and `LicenseService` classes:

```
UserContextFilter Correlation id: TEST-CORRELATION-ID
LicenseServiceController Correlation id: TEST-CORRELATION-ID
LicenseService:getLicensesByOrganization Correlation id:
```

If you don't see the log messages on your console, add the code lines shown in the following listing to the application.yml or application.properties file of the licensing service.

Listing 7.16 Logger configuration on the licensing service application.yml file

```
//Some code omitted for conciseness
logging:
  level:
    org.springframework.web: WARN
    com.optimagrowth: DEBUG
```

Then build again and execute your microservices. If you are using Docker, you can execute the following commands in the root directory where the parent pom.xml is located:

```
mvn clean package dockerfile:build
docker-compose -f docker/docker-compose.yml up
```

You'll see that once the call hits the resiliency protected method, we still get the values written out for the correlation ID. This means that the parent thread values are available on the methods using the Resilience4j annotations.

Resilience4j is an excellent choice to implement a resilience pattern in our application. With Hystrix going into maintenance mode, Resilience4j has become the number-one choice in the Java ecosystem. Now that we have seen what can be achievable with Resilience4j, we can move on with our next subject, the Spring Cloud Gateway.

Summary

- When designing highly distributed applications like a microservice, client resiliency must be taken into account.
- Outright failures of a service (for example, the server crashes) are easy to detect and deal with.
- A single, poorly performing service can trigger a cascading effect of resource exhaustion as the threads in the calling client are blocked when waiting for a service to complete.
- Three core client resiliency patterns are the circuit-breaker pattern, the fallback pattern, and the bulkhead pattern.
- The circuit breaker pattern seeks to kill slow-running and degraded system calls so that these calls fail fast and prevent resource exhaustion.
- The fallback pattern allows you to define alternative code paths in the event that a remote service call fails or the circuit breaker for the call fails.
- The bulkhead pattern segregates remote resource calls away from each other, isolating calls to a remote service into their own thread pool. If one set of service calls fails, its failure shouldn't be allowed to "eat up" all the resources in the application container.
- The rate limiter pattern limits the number of total calls in a given time period.
- Resilience4j allows us to stack and use several patterns at the same time.
- The retry pattern is responsible for making attempts when a service has temporarily failed.
- The main difference between the bulkhead and the rate limiter patterns is that the bulkhead is in charge of limiting the number of concurrent calls at one time, and the rate limiter limits the number of total calls over a given time.
- Spring Cloud and the Resilince4j libraries provide implementations for the circuit breaker, fallback, retry, rate limiter, and bulkhead patterns.
- The Resilience4j libraries are highly configurable and can be set at global, class, and thread pool levels.