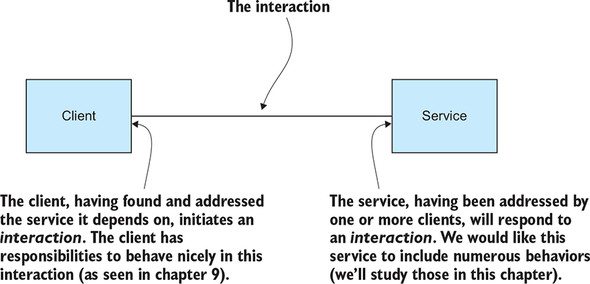
**Chapter 10. Fronting services: Circuit breakers and API gateways**

*This chapter covers*

* The service side of an interaction between two microservices
* Circuit breakers
* API gateways
* Sidecars and service mesh

I began talking about interactions between services in [chapter 8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08), with a focus on dynamic routing and service discovery; I was talking about how clients can find and access a service they depend on. After the client finds and addresses the needed service, it initiates an interaction. The previous chapter and this one come together to consider both sides of that interaction—as shown in [figure 10.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig01). In [chapter 9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09), I talked about the resilience of this interaction; I was largely talking about request redundancy, something the client is responsible for and applies controls to. Now I want to turn to the service side of that client/service interaction and the essential design patterns that play a role here.

**Figure 10.1. Just as the client can and should implement certain patterns to act as a good participant in the interaction, so too should the service. These are the patterns covered in this chapter.**



As the developer of the service, you have to account for many interaction-related concerns:

* In the previous chapter, I presented a solution to the retry storm problem that was implemented on the client side of the interaction (*kind retries*, I called them). But the service developer can’t depend on clients always being kind and must therefore guard against retry storms. From the service perspective, a *retry storm* is simply a case of having more incoming requests than it can handle. The service is ultimately responsible for protecting itself from intentional or inadvertent denial-of-service attacks.
* I previously talked about techniques for deploying a new version of a service—blue/green and rolling upgrades in particular. You’ll recall that I also talked about parallel deployments, whereby multiple versions of a service are running at the same time, with some requests being served by one of those services and some requests being served by another. In most cases, decisions on which service version should respond to a given request are handled on the service side of the client/service interaction.
* The service should respond only to requests from authorized parties.
* A service is also responsible for making monitoring and logging information available (a foreshadow of the next chapter).

This chapter covers two patterns that address these concerns: circuit breakers and API gateways. Circuit breakers explicitly target the first of these concerns and are used to protect the service from being overwhelmed with too much traffic. API gateways are used to address all of these concerns, and then some. Although API gateways have been in use for some time, I specifically cover the needs brought by the cloud-native architectures that have been in use only recently.

I close the chapter by covering a recently popularized implementation approach for the patterns of both the server side and the client side of the interactions: sidecars. Yes, I’ll be talking about Istio and friends here.

**10.1. Circuit breakers**

The concept of a *circuit breaker* in your software is exactly the same as that of the electrical system in your home. You have any number of potentially power-drawing sources in your home—lights, outlets, appliances, and so forth. The more power that’s simultaneously drawn on your wires, the hotter the wires will get, and if there’s enough load, the wires could get hot enough to light the walls through which they run on fire. To keep this from happening, the wires are run through a circuit breaker that will detect when the power draw gets dangerously high and will open the circuit so that all power will be cut off. Better to have no power than to light the house on fire.

**10.1.1. The software circuit breaker**

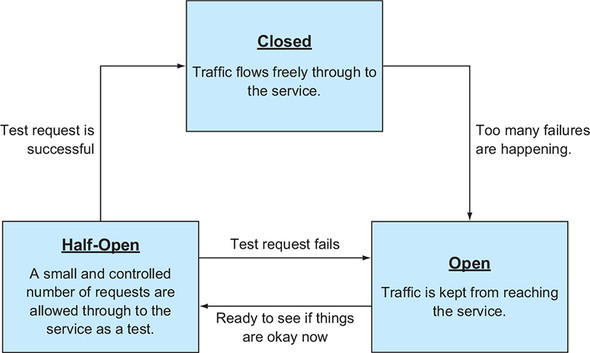
In your software, circuit breakers operate in essentially the same way. When the load is too high, a circuit opens and keeps traffic from flowing through. But two differences exist. First, the mechanism for detecting when the circuit should open is based on actual failures, not a prediction of possible failures (you wouldn’t want an electrical circuit to trip only after a small fire was detected). And second, the software circuit breaker usually has a self-healing mechanism built in (as opposed to having a human stumble through a dark house to find the electrical panel to manually flip the breaker).

The basic idea is this: If a service starts to fail more than a little bit, you stop all traffic to that service for a while, hopefully giving it time to recover from whatever is causing it to fail. Then, after some time, you check to see how it’s doing by allowing a single request through. If that request fails, you keep the protection in place, not allowing further traffic. If that request succeeds, you treat the service as healthy and allow traffic to flow freely again.

We can model this behavior by defining three states that a circuit breaker can be in (Closed, Open, or Half-Open), as shown in [figure 10.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig02). We can then describe the events that drive changes in state, as follows:

* The ideal state of your circuit breaker is *Closed*: traffic is flowing through the circuit, to the service that the circuit is protecting.
* The circuit breaker sits in that flow of traffic and looks out for failures. A small number of failures is not a problem; indeed, resilience to such “blips” is part of a good cloud-native design. When the failure rate gets too high, the state of the circuit breaker will become *Open*.
* While the circuit breaker is Open, no traffic will be allowed through to the service that the circuit breaker is protecting. If the service had started failing because it was overwhelmed with request load, or an intermittent network outage was causing trouble, the break from handling load may allow the service to return to a healthy state.
* After some time has passed, you want to try the service again to see whether it has recovered. You do this by putting the circuit breaker into a *Half-Open* state.
* When in the Half-Open state, the circuit breaker implementation will test the service by allowing a single request or a small number of requests through to the service.
* If the test requests are successful, the circuit will transition back to the Closed state. If the test fails, the circuit will transition back to the Open state and wait it out just a bit longer.

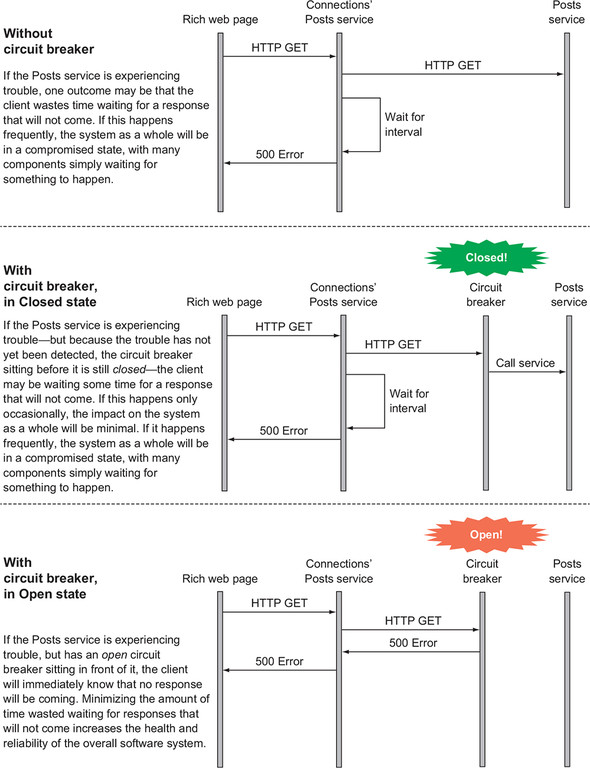
**Figure 10.2. You model the operation of the circuit breaker via three states and define the conditions or events that cause transitions between them. When the circuit is Closed, traffic flows freely. When the circuit is Open, requests won’t reach the service. The Half-Open state is transitory, the means by which the circuit can be reset to Closed.**



I’ve described the circuit breaker intuitively, but you and/or the circuit breaker implementation need to concretely define the specifics for the state changes. What constitutes “too many failures,” for example? In a moment, you’ll look at a concrete implementation and study these details. First, I want to draw your attention to one concept that’s not depicted in this diagram: how the use of a circuit breaker affects the client/service interaction that’s so central in this and the previous chapter.

The sequence diagram in [figure 10.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig03) shows three scenarios of a single interaction between a client and a service at a time when the service is experiencing trouble. In the first case, no circuit breaker is in use. In the second, you have a circuit breaker, and the state is Closed. Finally, you have a circuit breaker with an Open state.

**Figure 10.3. At times when a service is unavailable—because of network outages, trouble with the service itself, or another issue—one of the main benefits of circuit breakers is that they significantly reduce the amount of time wasted while waiting for responses that are at that moment unlikely to come.**



In the first two cases, you can see that the behavior is effectively identical: the client makes a request, and because of the trouble the service is experiencing, it times out waiting for a response. But in the last case, when the circuit is open because the circuit breaker has detected trouble, the client will quickly receive a response. The key here is that delays are disastrous in a complex distributed system, and the circuit breaker significantly reduces the length and frequency thereof. I like to think of circuit breakers as a “kindness” pattern implemented on the service side.

Let’s now look at an implementation of the circuit breaker in our running example. This will demonstrate basic use and configurability, and allow you to think a bit more deeply about the structure of your service implementations.

**10.1.2. Implementing a circuit breaker**

As is usual, you can run the code examples by checking out the Git repository, two specific tags for the examples herein, and deploying to a Kubernetes cluster. As I’ve done all along, I’ve built the code samples and bundled them in Docker images that are available to you in Docker Hub, so you needn’t build the code from source. If you do want to build from source, I’ve included the Maven and Docker build files for your convenience. Before running the examples, let’s have a look at the code.

Assuming you’ve already cloned the repository, please check out your first [chapter 10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10) tag with the following command:

git checkout circuitbreaker/0.0.1

The code is all located in the cloudnative-circuitbreaker directory, so change into that now. You’ll notice that there are implementations only for the Posts and the Connections services, because the Connections’ Posts service is the client side of the interaction and is unchanged from that of the previous chapter.

The service you’ll protect with the circuit breaker is the Posts service, so let’s start by looking at the code in the source directory for that service. The first thing you’ll notice is a new Java class: PostsService. The circuit breaker implementation fronts an actual service, and the way that it’s implemented here, you have the circuit breaker running within the same process as the main service implementation; the circuit breaker is close to the actual service (spoiler alert—I’ll explain this more when I talk about Istio at the end of this chapter).

You originally had a lot of logic in the Posts controller itself. But what you’ve done now is put the core of the service implementation into that new PostsService class, and you now have the controller handling only the front edge of the service interaction. The controller still handles tasks like request parsing and response generation, as well as some of the basic authentication and authorization logic. The new PostsService doesn’t deal with the HTTP protocol and instead focuses only on the core logic of the service, which for our simple example is only a database query and response object generation.

Most pertinent to our discussion is the addition of an annotation around the PostsService’s get method, as shown in the [listing 10.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10ex01).

**Listing 10.1. Method from PostsService.java**

@HystrixCommand()

public Iterable<Post> getPostsByUserId(String userIds,

String secret) throws Exception {

logger.info(utils.ipTag() + "Attempting getPostsByUserId");

Iterable<Post> posts;

if (userIds == null) {

logger.info(utils.ipTag() + "getting all posts");

posts = postRepository.findAll();

return posts;

} else {

ArrayList<Post> postsForUsers = new ArrayList<Post>();

String userId[] = userIds.split(",");

for (int i = 0; i < userId.length; i++) {

logger.info(utils.ipTag() +

"getting posts for userId " + userId[i]);

posts = postRepository.findByUserId(Long.parseLong(userId[i]));

posts.forEach(post -> postsForUsers.add(post));

}

return postsForUsers;

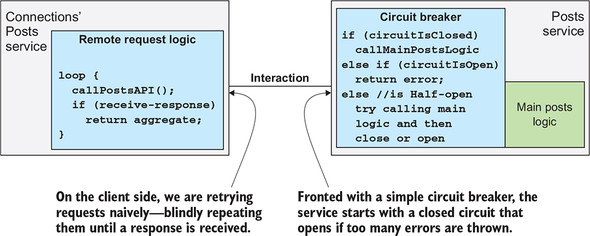
}

}

The @HystrixCommand() indicates that this method is to be fronted with a circuit breaker, and the Spring Framework will insert the implementation. It does so with an aspect that intercepts all incoming requests and implements the protocol I described previously.

Okay, so let’s see this in action—in particular, through the lens of the retry-storm scenario you learned about in the previous chapter. I want to take the naïve retry implementation of Connections’ Posts, the one that caused the system to remain unhealthy for an extended period of time after the network was reestablished, and couple that with a circuit breaker protecting the Posts service. See [figure 10.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig04). You’ll run the same load tests as you did before.

**Figure 10.4. For your first test run, you’ll be running with the naïve retries on the client side of the interaction, and a simple circuit breaker at the front of the service.**



**Setting up**

Once again, I refer you to the setup instructions for running the samples in earlier chapters in this text. There are no new requirements for running the sample in this chapter.

You’ll be accessing files in the cloudnative-circuitbreaker directory, so in your terminal window, change into that directory.

And as I’ve described in previous chapters, I’ve already prebuilt Docker images and made them available in Docker Hub. If you want to build the Java source and Docker images and push them to your own image repository, I refer you to earlier chapters (the most detailed instructions are in [chapter 5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_017.html#ch05)).

**Running the apps**

As you progress through the chapter, you’ll have different versions of the circuit breaker, so to start, you’ll need to check out the right tag on the GitHub repo:

git checkout circuitbreaker/0.0.1

You’ll need a Kubernetes cluster with sufficient capacity, as I described in [chapter 9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09). If you still have the examples running from the previous chapter, no need to clean up and start again; the commands you’ll run here will update the versions of all microservices appropriately. If you do want to start from scratch, you may use deleteDeploymentComplete.sh as I’ve previously described. This simple bash script allows you to keep MySQL, Redis, and SCCS running. Calling it with no options deletes only the three microservice deployments; calling the script with all as an argument deletes MySQL, Redis, and SCCS as well.

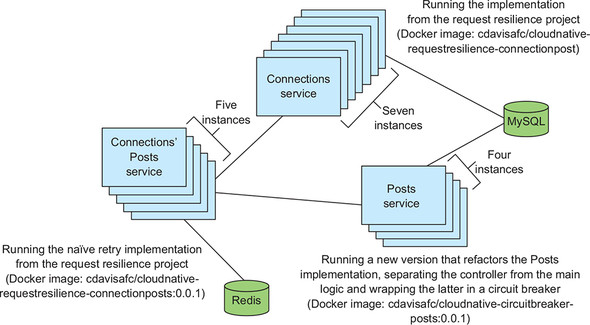
Given that you’ve checked out the Git tag as described previously, you can deploy or update the running services by running the following script (or issuing the kubectl apply commands contained therein):

./deployApps.sh

If you do this while running watch kubectl get all in another window, you’ll either see the Posts service upgraded—for this first example, only this service has changed—or you’ll see all three microservices deployed. The application topology is shown in [figure 10.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig05) and deploys the following app versions:

* ***Connections’ Posts—***This is the version from the request resilience project (of [chapter 9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09)) that has the naïve retry implementation, the one that blindly retries timed-out requests forever.
* ***Connections—***This is the version from the request resilience project and is the standard connections implementation.
* ***Posts—***This is a new version of the app that has been refactored to separate the controller from the main logic of the service. The main method in the latter has now been wrapped in a Hystrix circuit breaker.

**Figure 10.5. The deployment topology has versions of the Connections’ Posts and Connections services from the previous chapter and provides a new version of the Posts service. This implementation wraps the main Posts logic in a circuit breaker.**



Let’s now send some load to this implementation. You do so by issuing the following two commands:

kubectl create configmap jmeter-config \

--from-file=jmeter\_run.jmx=loadTesting/ConnectionsPostsLoad.jmx

kubectl create -f loadTesting/jmeter-deployment.yaml

If you ran the first command during the experiments of [chapter 9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09), rerunning it here is unnecessary, as the config map for the Apache JMeter deployment will already exist. Now let’s look at the output of the load test:

$ kubectl logs -f <name of your jmeter pod>

START Running Jmeter on Sun Feb 24 05:21:46 UTC 2019

JVM\_ARGS=-Xmn442m -Xms1768m -Xmx1768m

jmeter args=-n -t /etc/jmeter/jmeter\_run.jmx -l resultsconnectionsposts

Feb 24, 2019 5:21:48 AM java.util.prefs.FileSystemPreferences$1 run

INFO: Created user preferences directory.

Creating summariser <summary>

Created the tree successfully using /etc/jmeter/jmeter\_run.jmx

Starting the test @ Sun Feb 24 05:21:48 UTC 2019 (1550985708891)

Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445

summary + 85 in 00:00:10 = 8.1/s Err: 0 (0.00%) Active: 85

summary + 538 in 00:00:30 = 18.0/s Err: 0 (0.00%) Active: 332

summary = 623 in 00:00:40 = 15.4/s Err: 0 (0.00%)

summary + 1033 in 00:00:30 = 34.5/s Err: 0 (0.00%) Active: 579

summary = 1656 in 00:01:10 = 23.5/s Err: 0 (0.00%)

summary + 1529 in 00:00:30 = 51.0/s Err: 0 (0.00%) Active: 829

summary = 3185 in 00:01:40 = 31.7/s Err: 0 (0.00%)

summary + 2029 in 00:00:30 = 67.6/s Err: 0 (0.00%) Active: 1077

summary = 5214 in 00:02:10 = 40.0/s Err: 0 (0.00%)

summary + 2520 in 00:00:30 = 84.1/s Err: 0 (0.00%) Active: 1325

summary = 7734 in 00:02:40 = 48.2/s Err: 0 (0.00%)

summary + 2893 in 00:00:30 = 96.4/s Err: 0 (0.00%) Active: 1500

summary = 10627 in 00:03:10 = 55.8/s Err: 0 (0.00%)

summary + 3055 in 00:00:30 = 101.8/s Err: 0 (0.00%) Active: 1500

summary = 13682 in 00:03:40 = 62.1/s Err: 0 (0.00%)

summary + 3007 in 00:00:30 = 100.2/s Err: 0 (0.00%) Active: 1500

summary = 16689 in 00:04:10 = 66.7/s Err: 0 (0.00%)

<time marker 1 – I have broken the network between Posts and MySQL>

summary + 2510 in 00:00:30 = 83.6/s Err: 2084 (83.03%) Active: 1500

summary = 19199 in 00:04:40 = 68.5/s Err: 2084 (10.85%)

summary + 3000 in 00:00:30 = 100.0/s Err: 3000 (100.00%) Active: 1500

summary = 22199 in 00:05:10 = 71.5/s Err: 5084 (22.90%)

summary + 3000 in 00:00:30 = 100.0/s Err: 3000 (100.00%) Active: 1500

summary = 25199 in 00:05:40 = 74.0/s Err: 8084 (32.08%)

summary + 2953 in 00:00:30 = 98.4/s Err: 2953 (100.00%) Active: 1500

summary = 28152 in 00:06:10 = 76.0/s Err: 11037 (39.21%)

summary + 2916 in 00:00:30 = 96.9/s Err: 2916 (100.00%) Active: 1500

summary = 31068 in 00:06:40 = 77.6/s Err: 13953 (44.91%)

summary + 3046 in 00:00:30 = 101.7/s Err: 3046 (100.00%) Active: 1500

summary = 34114 in 00:07:10 = 79.3/s Err: 16999 (49.83%)

summary + 3019 in 00:00:30 = 100.7/s Err: 3019 (100.00%) Active: 1500

summary = 37133 in 00:07:40 = 80.7/s Err: 20018 (53.91%)

<time marker 2 – I have repaired the network between Posts and MySQL>

summary + 2980 in 00:00:30 = 99.3/s Err: 2980 (100.00%) Active: 1500

summary = 40113 in 00:08:10 = 81.8/s Err: 22998 (57.33%)

summary + 3015 in 00:00:30 = 100.5/s Err: 3015 (100.00%) Active: 1500

summary = 43128 in 00:08:40 = 82.9/s Err: 26013 (60.32%)

summary + 3020 in 00:00:30 = 100.7/s Err: 3020 (100.00%) Active: 1500

summary = 46148 in 00:09:10 = 83.8/s Err: 29033 (62.91%)

summary + 3075 in 00:00:30 = 102.5/s Err: 3072 (99.90%) Active: 1500

summary = 49223 in 00:09:40 = 84.8/s Err: 32105 (65.22%)

summary + 3049 in 00:00:30 = 101.6/s Err: 2395 (78.55%) Active: 1500

summary = 52272 in 00:10:10 = 85.6/s Err: 34500 (66.00%)

summary + 3191 in 00:00:30 = 106.4/s Err: 2263 (70.92%) Active: 1500

summary = 55463 in 00:10:40 = 86.6/s Err: 36763 (66.28%)

summary + 2995 in 00:00:30 = 99.7/s Err: 1203 (40.17%) Active: 1500

summary = 58458 in 00:11:10 = 87.2/s Err: 37966 (64.95%)

summary + 3031 in 00:00:30 = 101.1/s Err: 1193 (39.36%) Active: 1500

summary = 61489 in 00:11:40 = 87.8/s Err: 39159 (63.68%)

summary + 3009 in 00:00:30 = 100.3/s Err: 1182 (39.28%) Active: 1500

summary = 64498 in 00:12:10 = 88.3/s Err: 40341 (62.55%)

summary + 3083 in 00:00:30 = 102.8/s Err: 859 (27.86%) Active: 1500

summary = 67581 in 00:12:40 = 88.9/s Err: 41200 (60.96%)

summary + 3110 in 00:00:30 = 103.7/s Err: 597 (19.20%) Active: 1500

summary = 70691 in 00:13:10 = 89.4/s Err: 41797 (59.13%)

summary + 2999 in 00:00:30 = 99.9/s Err: 0 (0.00%) Active: 1500

summary = 73690 in 00:13:40 = 89.8/s Err: 41797 (56.72%)

summary + 3001 in 00:00:30 = 100.1/s Err: 0 (0.00%) Active: 1500

summary = 76691 in 00:14:10 = 90.2/s Err: 41797 (54.50%)

Just as you did for the tests in the previous chapter, after all the load has been established (at time marker 1 in the preceding log), you break the network between all instances of the Posts service and the MySQL database. As you can see, this results in all requests to Connections’ Posts failing (this is what you’re calling from the JMeter tests). After a roughly 3-minute outage, you reestablish the network, at time marker 2. Studying the log output, you can see that it took only about 1 minute for the first signs of recovery, and another 3.5 to 4 minutes for full recovery. [Table 10.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10table01) shows these results side by side with the results of having the naïve retry implementation (an unkind client indeed) with no protection around the service.

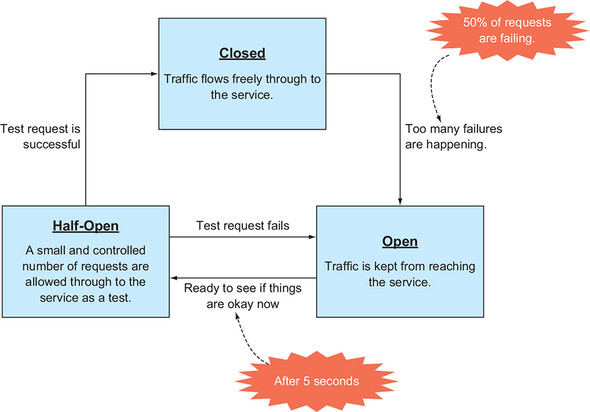
**Table 10.1. A circuit breaker provides significant protections against retry storms.**

| Version of Connections’ Posts service | Version of Posts service | Time to initial signs of recovery | Additional time to full recovery |
| --- | --- | --- | --- |
| Naïve retries | No circuit breaker | 9 minutes | 12–13 minutes |
| Naïve retries | Circuit breaker protecting the service | 1–2 minutes | 4–5 minutes |

That’s already a stark difference! I’ll point out that the circuit breaker provides protection not only from retry storms, but from excessive load or other error conditions, regardless of the cause. But in this case, how did the circuit breaker change the interaction between the Connections’ Posts and the Posts service that allowed the system to recover so much faster? It goes back to [figure 10.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig03); you’ve implemented the third scenario, so that instead of Connections’ Posts timing out for every one of the retries, after the circuit is open, it quickly receives a response from the Posts service, a response that clearly indicates a problem (with a 500 status code), so the retry backlog, if you will, is then much smaller.

Let’s explore this first implementation. As you can imagine, the @HistrixCommand() annotation allows numerous configuration options to control how it behaves. In this first example, you’ve simply accepted the defaults. Looking at the simple state diagram, in [figure 10.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig06), I’ve annotated it with those defaults. The circuit breaker will trip when 50% of the requests to the service fail and will remain open for 5 seconds before entering the Half-Open state.

**Figure 10.6. The default Hystrix implementation trips the circuit when 50% of requests fail and moves from the Open state to a Half-Open state seconds after entering the Open state. Successful or unsuccessful requests made while in the Half-Open state will transition the circuit breaker to Closed and Open states, respectively.**



Numerous other configuration options can be used for the Hystrix circuit breaker.**[**[**1**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fn1)**]** You can, for example, set a minimum number of failures before the circuit will trip. But the one that I want to focus on here is setting a fallback method. Remember how, when you implemented the kinder request redundancy with Spring Retry in the preceding chapter, you added a fallback method whereby you cached previous results and used those when the Posts service wasn’t responding? Same idea here: when the circuit is open, instead of returning an error as your implementation currently does, you’ll return something to stand in for the real result.

***1***

*These configuration options are available on GitHub at*[*http://mng.bz/O2rK*](http://mng.bz/O2rK)*.*

|  |
| --- |
|  |

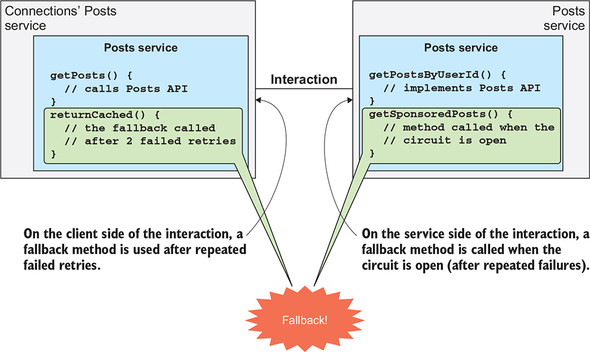
**Note**

The subtitle of this book is *Designing Change-tolerant Software.* One of the single most important things you must do when designing your software is always be thinking, “What should the software do in the event the operation I’m invoking isn’t successful?”

|  |
| --- |
|  |

It’s quite telling that frameworks that help you implement resiliency patterns have built-in primitives for fallback, on both sides of an interaction; [figure 10.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig07) shows this clearly.

**Figure 10.7. Failures can occur on either side of an interaction, and fallbacks provide safeguards.**



The context for each of these fallback methods differs. On the left side of [figure 10.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig07), the Connections’ Posts service is the *consumer* of information sought via the interaction, and can decide what it should do when live information isn’t available. Is stale content better than no content? In our final implementation in [chapter 9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09), you made the call that it was and returned cached content. On the right-hand side of [figure 10.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig07), the Posts service is the *provider* of information sought via the interaction and must decide whether any type of “success” response is better than an outright failure. Whatever the alternate success return is, this behavior must be well documented so that clients aren’t lured into believing they have one set of data when, in fact, they’re receiving an alternate set.

Let’s check this out with a running implementation. You’ll make only minor changes to our previous example. To see them, please check out the next Git tag with the following command:

git checkout circuitbreaker/0.0.2

Looking at the code for the Posts API, and the PostsService class specifically, you see that now a fallback method is provided, and the @HystrixCommand annotation points to it. In [listing 10.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10ex02) you see that the fallback implementation returns precanned results, sponsored content in the event that live data isn’t available:

**Listing 10.2. Methods from PostsService.java**

@HystrixCommand(fallbackMethod = "getSponsoredPosts")

public Iterable<Post> getPostsByUserId(String userIds,

String secret) throws Exception {

logger.info(utils.ipTag() + "Attempting getPostsByUserId");

Iterable<Post> posts;

if (userIds == null) {

logger.info(utils.ipTag() + "getting all posts");

posts = postRepository.findAll();

return posts;

} else {

ArrayList<Post> postsForUsers = new ArrayList<Post>();

String userId[] = userIds.split(",");

for (int i = 0; i < userId.length; i++) {

logger.info(utils.ipTag() +

"getting posts for userId " + userId[i]);

posts = postRepository.findByUserId(Long.parseLong(userId[i]));

posts.forEach(post -> postsForUsers.add(post));

}

return postsForUsers;

}

}

public Iterable<Post> getSponsoredPosts(String userIds,

String secret) {

logger.info(utils.ipTag() +

"Accessing Hystrix fallback getSponsoredPosts");

ArrayList<Post> posts = new ArrayList<Post>();

posts.add(new Post(999L, "Some catchy title",

"Some great sponsored content"));

posts.add(new Post(999L, "Another catchy title",

"Some more great sponsored content"));

return posts;

}

I want to draw your attention to two points here:

* The fallback method is called anytime an error is returned from the Hystrix-protected command (the getPostsByUserId method, in this case), even when the circuit is closed. The Hystrix library favors attempting fallbacks in all cases of failure, even if they’re never part of a major failure.
* Hystrix fallback methods can be chained; if the primary method fails, fallbackMethod1 can be called. This could, for example, attempt to calculate its results by using cached data or load data via an alternate channel. If fallbackMethod1 were to fail, control could be passed to fallbackMethod2, and so on. This is a powerful abstraction at your disposal.

You’ll notice that our fallback implementation is very, very naïve. It even hardcodes content (in the code!) rather than drawing it from a data store; this is purely to keep your implementation simple. Please don’t hardcode content into your source code!

**Running the apps**

I assume that you already have running the example from earlier in this section, and that you have checked out the Git branch as I described previously. If your previous load test is still running, stop it with the following command:

kubectl delete deploy jmeter-deployment

You can update your deployment to the version that implements the fallback behavior by running the following bash script or by executing the commands contained therein:

./deployApps.sh

Again, if you’re watching a kubectl get all command, you’ll see both the Connections and the Posts services being updated. An update was needed for Connections only to preload the sponsor’s user ID. The Connections’ Posts service wasn’t updated. You’ll still be running the naïve retry implementation from [chapter 9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09). And finally, let’s place some load against this deployment:

kubectl create -f loadTesting/jmeter-deployment.yaml

And now you’ll have a look at the logs for this deployment:

START Running Jmeter on Sun Feb 24 04:39:23 UTC 2019

JVM\_ARGS=-Xmn542m -Xms2168m -Xmx2168m

jmeter args=-n -t /etc/jmeter/jmeter\_run.jmx -l resultsconnectionsposts

Feb 24, 2019 4:39:25 AM java.util.prefs.FileSystemPreferences$1 run

INFO: Created user preferences directory.

Creating summariser <summary>

Created the tree successfully using /etc/jmeter/jmeter\_run.jmx

Starting the test @ Sun Feb 24 04:39:25 UTC 2019 (1550983165958)

Waiting for possible Shutdown/StopTestNow/Heapdump message on port 4445

summary + 217 in 00:00:21 = 10.4/s Err: 0 (0.00%) Active: 171

summary + 712 in 00:00:30 = 23.7/s Err: 0 (0.00%) Active: 419

summary = 929 in 00:00:51 = 18.3/s Err: 0 (0.00%)

summary + 1209 in 00:00:30 = 40.3/s Err: 0 (0.00%) Active: 667

summary = 2138 in 00:01:21 = 26.4/s Err: 0 (0.00%)

summary + 1706 in 00:00:30 = 57.0/s Err: 0 (0.00%) Active: 916

summary = 3844 in 00:01:51 = 34.7/s Err: 0 (0.00%)

summary + 2205 in 00:00:30 = 73.5/s Err: 0 (0.00%) Active: 1166

summary = 6049 in 00:02:21 = 43.0/s Err: 0 (0.00%)

summary + 2705 in 00:00:30 = 90.2/s Err: 0 (0.00%) Active: 1415

summary = 8754 in 00:02:51 = 51.2/s Err: 0 (0.00%)

summary + 2998 in 00:00:30 = 99.9/s Err: 0 (0.00%) Active: 1500

summary = 11752 in 00:03:21 = 58.5/s Err: 0 (0.00%)

<time marker 1 – I have broken the network between Posts and MySQL>

summary + 3004 in 00:00:30 = 100.0/s Err: 0 (0.00%) Active: 1500

summary = 14756 in 00:03:51 = 63.9/s Err: 0 (0.00%)

summary + 2997 in 00:00:30 = 99.9/s Err: 0 (0.00%) Active: 1500

summary = 17753 in 00:04:21 = 68.1/s Err: 0 (0.00%)

summary + 3001 in 00:00:30 = 100.1/s Err: 0 (0.00%) Active: 1500

summary = 20754 in 00:04:51 = 71.4/s Err: 0 (0.00%)

summary + 3000 in 00:00:30 = 100.0/s Err: 0 (0.00%) Active: 1500

summary = 23754 in 00:05:21 = 74.0/s Err: 0 (0.00%)

summary + 3000 in 00:00:30 = 100.0/s Err: 0 (0.00%) Active: 1500

summary = 26754 in 00:05:51 = 76.3/s Err: 0 (0.00%)

summary + 3000 in 00:00:30 = 100.0/s Err: 0 (0.00%) Active: 1500

summary = 29754 in 00:06:21 = 78.1/s Err: 0 (0.00%)

summary + 2995 in 00:00:30 = 99.9/s Err: 0 (0.00%) Active: 1500

summary = 32749 in 00:06:51 = 79.7/s Err: 0 (0.00%)

<time marker 2 – I have repaired the network between Posts and MySQL>

summary + 3005 in 00:00:30 = 100.2/s Err: 0 (0.00%) Active: 1500

summary = 35754 in 00:07:21 = 81.1/s Err: 0 (0.00%)

summary + 2997 in 00:00:30 = 99.9/s Err: 0 (0.00%) Active: 1500

summary = 38751 in 00:07:51 = 82.3/s Err: 0 (0.00%)

As usual, time marker 1 indicates the time that you broke the network connection between the Posts service and the MySQL database, and time marker 2 indicates when you reestablished the connections. And as you can see, calls to Connections’ Posts never failed, even during the network outage. This is exactly as you’d expect given that the fallback method for the circuit breaker returns sponsored content on any failures of the Posts service.

Perhaps a more interesting metric is how quickly live content is returned after the network is reestablished. Any guesses? Yes, you’re right: less than 5 seconds. Recall that the default setting for sleepWindowMilliseconds is 5000, meaning the circuit state will be set to Half-Open 5 seconds after being set to Open. As soon as that happens, the trial request that you allow through to the Post service logic will succeed, and the circuit will close, returning the application back to a stable state. You can see this transition in the log output for one of the Posts service instances:

2019-02-23 02:59:03.084 getting posts for userId 2

2019-02-23 02:59:03.148 Attempting getPostsByUserId

2019-02-23 02:59:03.148 getting posts for userId 2

2019-02-23 02:59:03.167 Attempting getPostsByUserId

2019-02-23 02:59:03.167 getting posts for userId 2

<time marker 1 – I have broken the network between Posts and MySQL>

2019-02-23 02:59:03.213 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 02:59:03.237 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 02:59:03.243 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 02:59:03.313 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 02:59:03.351 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 02:59:03.357 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 02:59:03.394 Accessing Hystrix fallback getSponsoredPosts

... (there are many more of these log lines)

<time marker 2 – I have repaired the network between Posts and MySQL>

... (another 5 seconds or so of Hystrix mentioning messages)

(then, ...)

2019-02-23 03:02:33.705 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 03:02:33.717 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 03:02:33.717 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 03:02:33.898 getting posts for userId 3

2019-02-23 03:02:33.898 getting posts for userId 3

2019-02-23 03:02:33.899 getting posts for userId 3

2019-02-23 03:02:33.899 getting posts for userId 3

2019-02-23 03:02:33.900 getting posts for userId 3

2019-02-23 03:02:33.905 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 03:02:33.911 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 03:02:33.943 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 03:02:34.080 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 03:02:34.100 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 03:02:34.113 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 03:02:34.216 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 03:02:34.225 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 03:02:34.300 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 03:02:34.368 Accessing Hystrix fallback getSponsoredPosts

2019-02-23 03:02:34.398 Attempting getPostsByUserId

2019-02-23 03:02:34.398 getting posts for userId 2

2019-02-23 03:02:34.400 getting posts for userId 3

2019-02-23 03:02:34.433 Attempting getPostsByUserId

2019-02-23 03:02:34.433 getting posts for userId 2

2019-02-23 03:02:34.434 Attempting getPostsByUserId

2019-02-23 03:02:34.434 getting posts for userId 2

2019-02-23 03:02:34.435 getting posts for userId 3

2019-02-23 03:02:34.437 getting posts for userId 3

2019-02-23 03:02:34.472 Attempting getPostsByUserId

2019-02-23 03:02:34.472 getting posts for userId 2

2019-02-23 03:02:34.475 getting posts for userId 3

2019-02-23 03:02:34.556 Attempting getPostsByUserId

2019-02-23 03:02:34.556 getting posts for userId 2

2019-02-23 03:02:34.559 getting posts for userId 3

2019-02-23 03:02:34.622 Attempting getPostsByUserId

(and operation has returned to normal)

[Table 10.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10table02) shows the results from each of the tests you’ve run through the previous chapter and this one. Each case simulated the same 3-minute network outage between Connections’ Posts and Posts, but with different patterns applied to the client side (in [chapter 9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09)) or the server side (in [chapter 10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10)) of the interaction.

**Table 10.2. Results from the network outage simulations show the benefit of applying certain cloud-native patterns to service interactions.**

| Version of Connections’ Posts service | Version of Posts service | During network outage | Time to initial signs of recovery | Time to full recovery | Test run in chapter |
| --- | --- | --- | --- | --- | --- |
| Naïve retries | No circuit breaker | 100% error | 9 minutes | 12–13 minutes | 9 |
| Kind retries using Spring Retry and no fallback | No circuit breaker | 100% error | 1 minute | 3 minutes | 9 |
| Kind retries using Spring Retry with fallback method | No circuit breaker | 0% error | N/A—no failure during network outage | N/A | 9 |
| Naïve retries | Circuit breaker protecting the service—no fallback method | 100% error | 1–2 minutes | 4–5 minutes | 10 |
| Naïve retries | Circuit breaker protecting the service—with fallback method | 0% error | N/A—no failure during network outage | < 5 seconds Considering “full recovery” the time when actual results, not sponsor results, are again returned | 10 |

This summary is interesting indeed. Implementing patterns focused on resilience of interactions makes a big difference in the overall health of your software. Although, clearly, patterns are applicable both on the client and on the server side of an interaction, you often won’t be responsible for implementations on both sides. Therefore, particularly if you’re implementing a consumer, it’s crucial that you fully understand the contract for the API—whether the service you’re consuming will alter results when execution deviates from the “happy path.” And when you’re providing a service, make sure to fully specify that contract.

Reflecting on what a circuit breaker does, and particularly in light of the use of aspects in the Hystrix implementation, you can see that the circuit breaker is essentially acting as a gateway to the Posts service. Ah, but the circuit breaker is but one example of functionality you may want to place in front of a service. Let’s now have a look at the API gateway as a more generic pattern.

**10.2. API gateways**

The availability of both open source and commercial API gateways predates the rise of microservice and cloud-based architectures. For example, Apigee (since acquired by Google) and Mashery (since acquired by Intel and then sold to TIBCO) were companies both founded in the early 2000s, and both focused on API gateways.

The role of the API gateway in a software architecture has always been exactly what the title of this chapter says, to sit in front of bits of implementation and provide a whole host of services. These services might include the following:

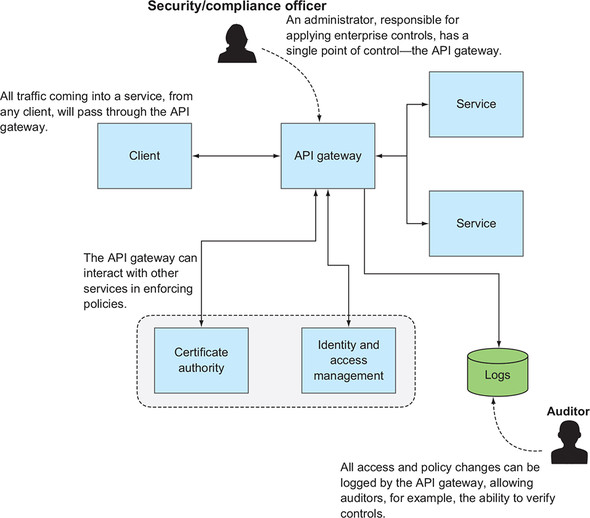
* ***Authentication and authorization—***Controlling access to the service the API gateway fronts. The mechanisms for this access control vary and can include secrets-based approaches such as the use of passwords or tokens, or could be network-based, either integrating with or implementing firewall-type services.
* ***Encryption of data in flight—***The API gateway can handle decryption and is therefore the place where certificates must be managed.
* ***Protecting the service from load spikes—***Configured properly, the API gateway becomes the only way that clients can access a service. Therefore, load-throttling mechanisms implemented here can provide significant protections. You might be thinking that what we just covered with circuit breakers sounds a bit like this, and you’d be right.
* ***Access logging—***Because all traffic into the service comes through the API gateway, you have the ability to log all access. These logs can support a myriad of use cases, including auditing and operations observability.

Any of these concerns could be addressed within the service itself, yet it’s obvious that these are cross-cutting concerns that needn’t be implemented over and over again. Use of API gateways relieves the developer of functionality that could just be viewed as plumbing, allowing them to focus on business needs. But perhaps even more important, it provides a point where enterprise controls can be uniformly applied. Certainly, one of the most challenging things for IT operations is to demonstrate that security and compliance requirements are met on everything—centralized control is key.

The API gateway can perform its duties by interfacing with many other services. For example, the gateway doesn’t itself store the users that need to be authenticated and authorized. Instead, it depends on an identity and access management solution and identity store (for example, LDAP) for those services; it simply enforces the policies expressed therein.

[Figure 10.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig08) depicts a simple scenario: an API gateway is fronting a couple of services, all access to those services comes through the gateway, and it interfaces with other components to support the functionality it offers. An IT systems administrator, using an interface for the API gateway, will configure the needed policies. The figure also depicts an auditor reviewing access logs for each of the services.

**Figure 10.8. The API gateway fronts all services and becomes the policy configuration and enforcement point for them.**



**10.2.1. The case for API gateways in cloud-native software**

If API gateways have been around and in use for more than 15 years, why am I covering them in this book? Well, as you can imagine, just as with many other topics we’ve discussed so far, the evolution to cloud-native software architectures introduces new requirements on the API gateway:

* It’s pretty clear that the componentization of software that yields many more independent (micro) services increases the number of services to control by several orders of magnitude. Although it certainly wasn’t ideal even then, it was at least theoretically possible for IT staff to manage service access without a centralized control plane. When you have thousands or even tens of thousands of service instances, this is no longer possible.
* The constant change being exerted on the service instances as they’re re-created during outages and scheduled upgrades similarly means that any manual configurations that may have been done when deployments changed only annually or biannually (firewall rules, for example) are now completely intractable without a software solution to assist.
* Highly distributed systems have led to the implementation of other resilience patterns, such as the retries you’ve just been studying, which brings different load profiles to a service. The load that will be coming at a service is less predictable than it had been before. You need to protect services from unexpected and extreme request volumes. The circuit breaker you studied in the early part of this chapter is one type of this protection that has made its way into the API gateway.
* Cloud-native architectures have played a role in truly enabling new business models that allow for fee-based consumption of services. An API gateway enables the necessary metering, possibly with load throttling.
* In earlier chapters, I talked about parallel deploys. The API gateway is an excellent place to implement the routing logic that’s so critical to things like safe upgrade processes.

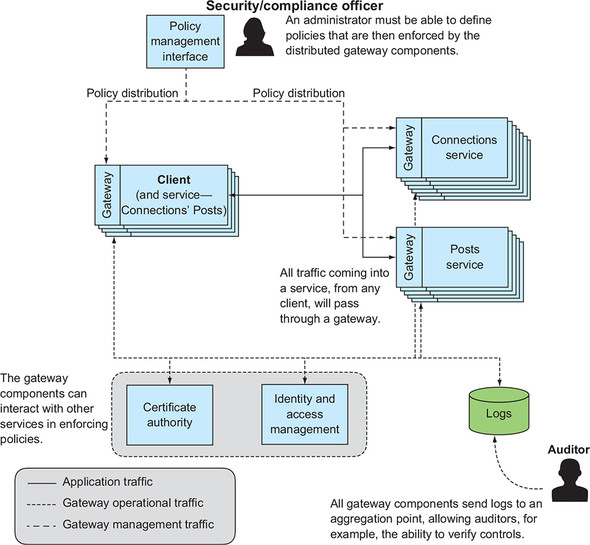
Whereas API gateways may have been strongly advised yet not mandatory a decade ago, the characteristics of cloud-native software render them absolutely critical now.

**10.2.2. API gateway topology**

I hope at this point you’re thinking something like, “Okay, I get why they’re needed, but I don’t like your [figure 10.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig08). A centralized gateway sure seems like a cloud-native antipattern.” You’re right! You need consistent application of policies across all of your services—that’s one of the value props of API gateway patterns. But that doesn’t mean the implementation must be centralized. Yes, 15 years ago the API gateway was often deployed as a centralized, even if clustered, component, but that has changed in cloud-native architectures.

As I’ve already alluded to several times, the circuit breaker you studied and implemented in the first section of this chapter is an example of a gateway pattern, and the implementation certainly was distributed. In fact, it was compiled into the binary for the service itself (remember the inclusion of the @HystrixCommand annotation?). To get right to the punch line, what you need is distributed implementations of API gateway patterns. To get a visual of this, look at [figure 10.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig09). There you can see that each of the services has had a gateway tacked to the front of it, and this gateway, just as in the previous diagram, interfaces with the set of components that support or are needed for its operations.

**Figure 10.9. You can think of the gateway as a single logical entity, something that’s needed for administration. But for cloud-native architectures, the implementation is best distributed.**



You’ll also notice that I’ve depicted many more service instances than in the previous picture. You can imagine that if you had that many services with a centralized API gateway, and *all* interaction flowed through that gateway, you’d have to concern yourself with ensuring that the gateway was sized properly to handle the traffic to a heterogeneous and therefore difficult-to-predict set of app instances. By distributing the processing, each gateway instance handles the load for its service alone, and the proper sizing of the gateway is far more tractable.

I’d like to take a moment for you to explore an open source API gateway that has become popular in the last few years. It comes from our microservice heroes, Netflix. Zuul (named after a gatekeeper figure in the movie *Ghostbusters*), is described as “an edge service that provides dynamic routing, monitoring, resiliency, security, and more.” These are the very things I’ve been attributing to the API gateway pattern. Zuul uses or embeds several other components from the Netflix microservices framework, including Hystrix (circuit breakers), Ribbon (load balancing), Turbine (metrics), and more.

Zuul, written in Java and therefore running in the JVM, is configured to front services via a URL. For example, to configure it to act as the gateway for our Posts service, you’d provide configuration data such as the following:

zuul.routes.connectionPosts.url=http://localhost:8090/connectionPosts

server.port=8080

The question then is how to include Zuul in your software topology. It’s absolutely possible to create a deployment that looks like that of [figure 10.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig08), but in a highly distributed software architecture, one closer to that of [figure 10.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig09) is advised. In fact, Spring Cloud provides a way for Zuul to be embedded in your service in much the same way that the circuit breaker was embedded in our earlier example.**[**[**2**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fn2)**]** In doing so, you achieve the deployment topology shown in [figure 10.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig09).

***2***

*As is typical with Spring Boot, including Zuul is as straightforward as including spring-cloud-starter-netflix-zuul in your Maven or Gradle dependencies:*[*http://mng.bz/YP4o*](http://mng.bz/YP4o)*.*

Having the gateway embedded into the service carries some obvious advantages: there’s no network hop between the gateway and the service itself, the host name is no longer needed for configuration, only the path is needed, cross-origin resource sharing (CORS) concerns go away, and so on. But there are several disadvantages as well.

First, recall from the earlier discussions on application lifecycle that binding configuration later in the cycle offers more flexibility. If you’ve included the preceding configuration in an application.properties file, a change in configuration requires a recompilation. As we discussed, property values can be injected later via environment variables, but that still requires a restart of the JVM (or at least a refresh of the application context).

Second, if you’re embedding a Java component, that pretty much means your service implementation must also be in Java, or at least be running in the JVM. Although all of the code examples here are in Java, the patterns I espouse are applicable to and should be implemented in whatever language is most appropriate in your scenario. I’m not a huge fan of Java-only solutions.

And finally, one of the goals of the API gateway pattern is to separate the concerns of the service developer from those of the operator. You want to give the latter the ability to apply consistent controls across all of the running services and offer them a control plane that makes this manageable.

How then do you achieve something like this gateway pattern in a way that’s programming-language agnostic, more loosely coupled, and manageable? Enter the service mesh.

**10.3. The service mesh**

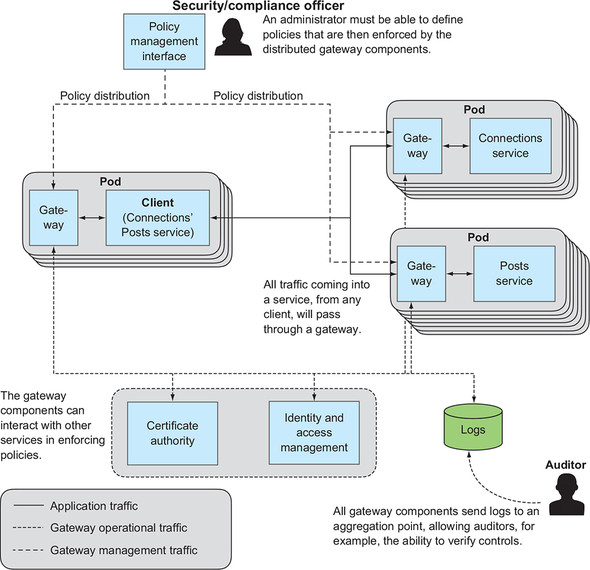
We don’t have to go all the way to the service mesh in one step, so let me back up a little bit and start with a primitive that plays the central role in the service mesh. Then I will go on to introduce the service mesh and the role it’s increasingly playing in the cloud-native software architecture.

**10.3.1. The sidecar**

Going back to the question that I just posed—how to provide distributed API gateway functionality that avoids the disadvantages of an embedded Java component—the answer is the sidecar. At the simplest level, a *sidecar* is a process that runs alongside your main service. If you look back at [figure 10.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig09), you can imagine that the gateway services could be thought of as running alongside the services, not necessarily embedded. To meet the requirement that it not be compiled into the service binary, this, of course, means that the gateway sidecar is running as a separate process alongside the main service process.

Brilliantly, Kubernetes offers an abstraction that makes this work beautifully: the Kubernetes pod. A *pod* is the smallest unit of deployment in Kubernetes, and it contains one or more containers. You can host your main service in one container and the gateway services in another, both running in the same pod. We can now redraw our earlier diagram to use these constructs; see [figure 10.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig10).

**Figure 10.10. The distributed gateway runs as a sidecar to each service. In Kubernetes this is achieved by running two containers in a single pod—one that’s the primary service and the other a gateway sidecar.**



Each container has its own runtime environment, so the main service could, say, be running in a JVM, and the gateway sidecar could be implemented in, say, C++. Check—one disadvantage addressed. But now the communication between the gateway and the main service is interprocess, and even intercontainer; that means a network hop. Again, the architecture of Kubernetes comes to our assistance. All services running in a Kubernetes pod are hosted at the same IP address, meaning they can address each other over localhost and the network hop is therefore minimal.

One of the most popular sidecar implementations in use today is Envoy ([www.envoyproxy.io](http://www.envoyproxy.io/)). Originally built by ride-share company Lyft, Envoy is a distributed proxy written in C++, making it extremely efficient. It can be used within various deployment topologies, though the most common is having each instance front a single instance of a service (in a topology such as the one reflected in [figure 10.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig10)).

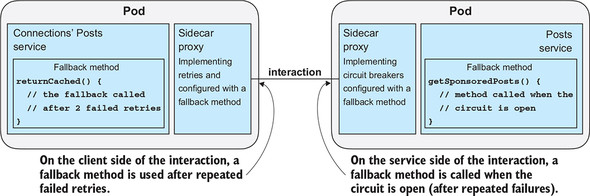
But this description is a bit disingenuous. Notice that I described Envoy as a proxy, not a gateway. Envoy does more than act as a gateway; it also proxies clients. I want to draw your attention all the way back to [figure 10.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig07), which depicts both a client and a service that are participating in an interaction. That diagram specifically shows that you added retry behaviors to the outbound interaction on the client side and added a circuit breaker at the front of the inbound to the service-side code. The latter is implementing a gateway pattern; the former is a proxy. The punch line? Envoy implements a proxy on the client side and a reverse-proxy/gateway on the service side of the interaction.

Okay, so that’s pretty cool.

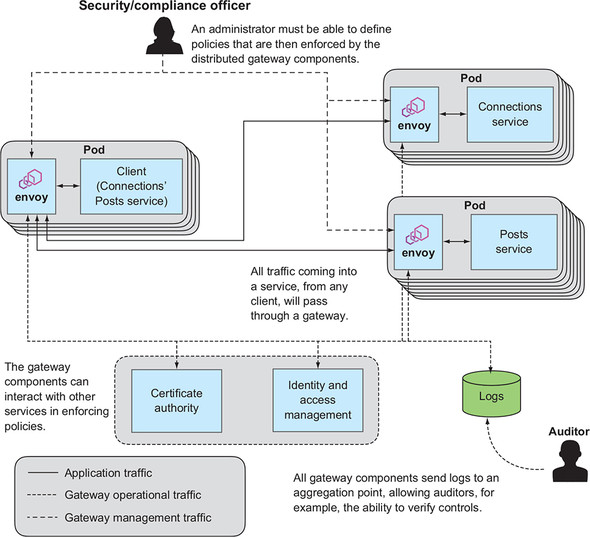
Redrawing [figures 10.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig07) and [10.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig10) in [figures 10.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig11) and [10.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig12), respectively, you can now see that a key element in cloud-native architectures, the interaction, is programmed through sidecars. Envoy implements a host of patterns at the edges of these interactions, including retries, circuit breakers, rate limiting, load balancing, service discovery, observability, and more. As I’ve said many times before, although you, the application developer or architect, must understand the patterns covered throughout this book, you aren’t always responsible for implementing them. I’ll say it again: this is pretty cool.

Notice that the interaction is now between proxies; the same is true in [figure 10.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig12).

**Figure 10.11.**[**Figure 10.7**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig07)**redrawn with retry behavior on the client side of the interaction implemented in a sidecar, and the circuit breaker on the service side also implemented in a sidecar.**



**Figure 10.12. The abstract “gateway” of**[**figure 10.10**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig10)**is now made concrete with an Envoy sidecar, one of several sidecar implementations.**



I haven’t yet addressed the other two disadvantages of an embedded gateway, both of which come down to manageability of the proxies and gateways. That’s where the service mesh comes in.

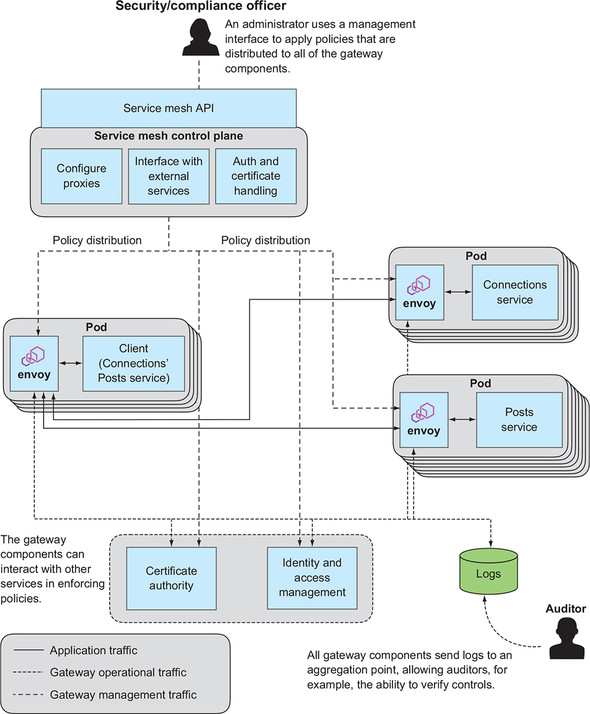
**10.3.2. The control plane**

Looking at [figure 10.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig12), you see a whole bunch of Envoy proxies connected via the channels that will carry interactions between them. It looks like a mesh; hence, the name. The service mesh encompasses the set of interconnected sidecars and adds a control plane for management of those proxies.

One of the most widely used service meshes available today comes from Istio (<https://istio.io/>), an open source project that was incubated by Google, IBM, and Lyft. It extends Kubernetes, using the pod primitive as the deployment mechanism for Envoy sidecars. Istio’s tagline is “Connect, secure, control, and observe services,” and it does so by supporting automatic sidecar injection and providing components that support configuration of the Envoy proxies, certificate handling, and policy enforcement. A control plane API offers the interface to this management control plane.

[Figure 10.13](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10fig13) completes the picture that we’ve been deriving through the chapter.

**Figure 10.13. A service mesh brings together sidecars and a control plane for their management.**



This chapter and the preceding one have focused on the two sides of an interaction between services. Because the interaction crosses processes and sometimes even network boundaries, a variety of patterns are necessary to provide a robust software implementation that’s tolerant to the inevitable changes in these distributed, cloud-based deployments. I’ve covered two of the key ones, retries on the client side and circuit breakers on the service side, the latter generalizing to a gateway pattern. Most notably, the service mesh has emerged as an essential part of the platform for running cloud-native applications. I strongly encourage you to use this technology.

I want to cover one final topic that centers on interaction: troubleshooting. Whether the main flow is request/response or event driven, the experience that a user has with the software is a reflection of the operation of dozens or even hundreds of services, all interacting with one another. When something doesn’t go quite right, how on earth do we find the root cause of the trouble? This is covered in the next chapter.

**Summary**

* A host of patterns are designed to sit at the front edge of a service, which control the way an interaction with that service is handled.
* Circuit breakers are an essential pattern for protecting a service from being overwhelmed by load, including for traffic produced through retry storms.
* API gateways, which predate cloud-native software architectures, have evolved to operate well in the new context of highly distributed, constantly changing software deployments.
* Patterns applied on both the client side and the service side of an interaction can be encapsulated in and deployed as a sidecar proxy.
* The service mesh adds to the sidecar proxy a management plane that allows an operator to control security, offer observability, and allow configuration of the collection of services/apps that make up the cloud-native software.