**Chapter 11. Troubleshooting: Finding the needle in the haystack**

*This chapter covers*

* Application logging for services in ephemeral environments
* Application monitoring for services in ephemeral environments
* Distributed tracing

Back in 2013, I was working with one of the first enterprise customers of the Cloud Foundry open source platform. I visited a particular client every couple of weeks to check on their progress and to help explain capabilities that at that time were really quite new to the industry. And no matter how cool the feature set I was describing, one of the engineers from that organization invariably responded with something along the lines of “Cornelia, you are giving me a Ferrari without a dashboard.” You see, we hadn’t yet done a good job of adding capabilities in the area of observability, and there was simply no way that this customer (or any customer, really) could put a system in production without the ability to adequately monitor its health and the health of the applications running on the platform. That customer engineer, Srini, was absolutely right!

Solutions for system and application observability are nothing new. A big part of the operational practices for a piece of software, often captured in a runbook, is centered on how to assess if that software is running well, and how to, as early as possible, recognize when something goes awry. Over the past decades, tools along with best practices have been established that have helped turn the task of observability into a robust and dependable practice. But just as with many other well-established aspects of software, a cloud-native architecture brings with it a new set of challenges for which we must establish a new set of tools and practices. What are some of the new concerns for our highly distributed, constantly changing software?

As you’ve now seen many times, the constant change I speak of manifests itself as an ephemerality of the running apps and the environment in which they reside. The container a service is running in is constantly in flux—deleted and replaced by new instances during lifecycle operations such as an upgrade, or to recover instances that have had some type of catastrophic event (such as out of memory). This poses problems for many of the familiar troubleshooting practices of the past, which often involved poking around the runtime environment for clues. Now that you can’t count on that runtime environment being around, how do you ensure access to the information you need to diagnose potential problems?

And the highly distributed nature of our software also brings new challenges. When a single user request fans out into dozens or hundreds of downstream requests, how can you pinpoint the cause of trouble in that complex hierarchy? Where you once had many components running in a single process and could therefore navigate through a call stack with relative ease, you now have calls that span a large number of distributed services, yet you’d still like to understand what that “call stack” looks like.

This chapter focuses on both of these elements. You’ll see how to generate and handle log and metric data in a way that accounts for ephemeral runtime environments. And you’ll learn about distributed tracing—a set of techniques and tools that mimic the intraprocess tracing techniques of the past, allowing an operator to follow the flow of related requests throughout a distributed network of microservices.

**11.1. Application logging**

I won’t spend any time convincing you to write entries to logs; that’s table stakes. But some of you may have performed log management from within an application; for example, you might have opened files and written to them. What I will argue for is that management of logs should be completely outside the application code.

Truth be told, this isn’t an argument specific to cloud-native apps. It’s a good idea for all software. The app code should express what should be logged, and the location where that log entry appears should be entirely controlled by the application deployment, not the app itself. Plenty of frameworks support this approach: Apache Log4j and its successor, Logback (<https://logback.qos.ch/>), for example, do just that; we’ve been using the latter in our code samples throughout this book. These allow the application code to simply have statements such as the following:

logger.info(utils.ipTag() + "New post with title " + newPost.getTitle());

Whether that log message then appears in a particular file, or in a console, or in something else is determined as part of the deployment.

For cloud-native applications, that deployment configuration should send the log lines to stdout and stderr. I know that’s a pretty opinionated statement, so let me lend it a bit of support:

* Files are off-limits. The local filesystem lives only as long as the container does. You’re going to need to access logs even after (dare I say, particularly after) an app instance and its container are gone. True, some container orchestration systems do support allowing a container to connect to an external storage volume whose lifecycle is independent from that of the container, but the semantics for doing so are complex, and the risk for contention issues with many other applications and application instances is significant.
* Driven in large part by the surge in popularity of open source and its accompanying resistance to proprietary solutions, we aim for some level of standardization wherever we can get it. We don’t want to do logging one way if we deploy on JBoss, another way if we deploy on WebSphere, and yet another way on WebLogic. Stdout and stderr are ubiquitous; there’s no vendor lock-in there.
* Stdout and stderr are not only vendor agnostic, but operating system agnostic. The concepts are the same whether on Linux, Windows, or another OS, and implementations deliver the same capabilities.
* Stdout and stderr are streaming APIs, and logs are most definitely streams. They don’t have a start or end; instead, entries just keep flowing. As those entries appear in the stream, a stream-processing system can appropriately handle them.

Okay, then, application developers don’t need to concern themselves with more than making a method call on a logger object (for example), but let’s talk for just a moment about how the logs are then handled. As with many other topics we’ve already discussed, platforms are your friends. Case in point: you’ve already had ample practice as a user of the log-handling features of Kubernetes. Our app instances use SLF4J (a façade for logging frameworks such as Logback) objects to create log entries that are sent to the stdout and stderr streams. When you execute a command such as kubectl logs -f pod/posts-fc74d75bc-92txh, the Kubernetes CLI will connect into and present the stream entries to your terminal.

What I haven’t spent any time talking about is how logging works when you have many instances of a service. For the most part, I’ve had you stream the logs for a single application instance. But in some situations, you may be interested in looking at the logs for an app, regardless of the number of instances running. For example, you may want to check whether a specific request was handled by any of your application instances without having to check the logs for each instance individually. Kubernetes will allow you to do this with a command such as the following:

$ kubectl logs -l app=posts

2018-12-02 22:41:42.644 ... s.c.a.AnnotationConfigApplicationContext ...

2018-12-02 22:41:43.582 ... trationDelegate$BeanPostProcessorChecker ...

. \_\_\_\_ \_ \_\_ \_ \_

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

=========|\_|==============|\_\_\_/=/\_/\_/\_/

:: Spring Boot :: (v1.5.6.RELEASE)

2018-12-02 22:41:44.309 ... c.c.c.ConfigServicePropertySourceLocator ...

...

2018-12-02 22:42:38.098 : [10.44.4.61:8080] Accessing posts using secret

2018-12-02 22:42:38.102 : [10.44.4.61:8080] getting posts for userId 2

2018-12-02 22:42:38.119 : [10.44.4.61:8080] getting posts for userId 3

2018-12-02 22:42:40.806 : [10.44.4.61:8080] Accessing posts using secret

2018-12-02 22:42:40.809 : [10.44.4.61:8080] getting posts for userId 2

2018-12-02 22:42:40.819 : [10.44.4.61:8080] getting posts for userId 3

2018-12-02 22:42:43.399 : [10.44.4.61:8080] Accessing posts using secret

2018-12-02 22:42:43.399 : [10.44.4.61:8080] getting posts for userId 2

2018-12-02 22:42:43.408 : [10.44.4.61:8080] getting posts for userId 3

2018-12-02 22:53:27.039 : [10.44.4.61:8080] Accessing posts using secret

2018-12-02 22:53:27.039 : [10.44.4.61:8080] getting posts for userId 2

2018-12-02 22:53:27.047 : [10.44.4.61:8080] getting posts for userId 3

2018-12-02 22:41:21.155 ... s.c.a.AnnotationConfigApplicationContext ...

2018-12-02 22:41:22.130 ... trationDelegate$BeanPostProcessorChecker ...

. \_\_\_\_ \_ \_\_ \_ \_

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

=========|\_|==============|\_\_\_/=/\_/\_/\_/

:: Spring Boot :: (v1.5.6.RELEASE)

2018-12-02 22:41:23.085 ... c.c.c.ConfigServicePropertySourceLocator ...

...

2018-12-02 22:42:46.297 : [10.44.2.57:8080] Accessing posts using secret

2018-12-02 22:42:46.298 : [10.44.2.57:8080] getting posts for userId 2

2018-12-02 22:42:46.305 : [10.44.2.57:8080] getting posts for userId 3

2018-12-02 22:53:30.260 : [10.44.2.57:8080] Accessing posts using secret

2018-12-02 22:53:30.260 : [10.44.2.57:8080] getting posts for userId 2

2018-12-02 22:53:30.266 : [10.44.2.57:8080] getting posts for userId 3

Taking a closer look at this output, you’ll notice that the first part shows log entries from one pod instance, followed by entries from the second instance; the logs from multiple instances aren’t interleaved. In many cases, it may be helpful to see the messages in time order, across all instances. For example, here’s an ordering of the previous logs:

2018-12-02 22:41:21.155 ... s.c.a.AnnotationConfigApplicationContext ...

2018-12-02 22:41:22.130 ... trationDelegate$BeanPostProcessorChecker ...

. \_\_\_\_ \_ \_\_ \_ \_

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

=========|\_|==============|\_\_\_/=/\_/\_/\_/

:: Spring Boot :: (v1.5.6.RELEASE)

2018-12-02 22:41:23.085 ... c.c.c.ConfigServicePropertySourceLocator ...

2018-12-02 22:41:42.644 ... s.c.a.AnnotationConfigApplicationContext ...

2018-12-02 22:41:43.582 ... trationDelegate$BeanPostProcessorChecker ...

. \_\_\_\_ \_ \_\_ \_ \_

/\\ / \_\_\_'\_ \_\_ \_ \_(\_)\_ \_\_ \_\_ \_ \ \ \ \

( ( )\\_\_\_ | '\_ | '\_| | '\_ \/ \_` | \ \ \ \

\\/ \_\_\_)| |\_)| | | | | || (\_| | ) ) ) )

' |\_\_\_\_| .\_\_|\_| |\_|\_| |\_\\_\_, | / / / /

=========|\_|==============|\_\_\_/=/\_/\_/\_/

:: Spring Boot :: (v1.5.6.RELEASE)

2018-12-02 22:41:44.309 ... c.c.c.ConfigServicePropertySourceLocator ...

...

2018-12-02 22:42:38.098 : [10.44.4.61:8080] Accessing posts using secret

2018-12-02 22:42:38.102 : [10.44.4.61:8080] getting posts for userId 2

2018-12-02 22:42:38.119 : [10.44.4.61:8080] getting posts for userId 3

2018-12-02 22:42:40.806 : [10.44.4.61:8080] Accessing posts using secret

2018-12-02 22:42:40.809 : [10.44.4.61:8080] getting posts for userId 2

2018-12-02 22:42:40.819 : [10.44.4.61:8080] getting posts for userId 3

2018-12-02 22:42:43.399 : [10.44.4.61:8080] Accessing posts using secret

2018-12-02 22:42:43.399 : [10.44.4.61:8080] getting posts for userId 2

2018-12-02 22:42:43.408 : [10.44.4.61:8080] getting posts for userId 3

2018-12-02 22:42:46.297 : [10.44.2.57:8080] Accessing posts using secret

2018-12-02 22:42:46.298 : [10.44.2.57:8080] getting posts for userId 2

2018-12-02 22:42:46.305 : [10.44.2.57:8080] getting posts for userId 3

2018-12-02 22:53:27.039 : [10.44.4.61:8080] Accessing posts using secret

2018-12-02 22:53:27.039 : [10.44.4.61:8080] getting posts for userId 2

2018-12-02 22:53:27.047 : [10.44.4.61:8080] getting posts for userId 3

2018-12-02 22:53:30.260 : [10.44.2.57:8080] Accessing posts using secret

2018-12-02 22:53:30.260 : [10.44.2.57:8080] getting posts for userId 2

2018-12-02 22:53:30.266 : [10.44.2.57:8080] getting posts for userId 3

Although it can be helpful to see the logs aggregated in this way, with entries interleaved in time order, it’s almost always important that log entries can be attributed to specific instances of the app. In these log entries, you can see this through the IP address: one instance has IP address 10.44.4.61, and the other has 10.44.2.57. Ideally, the instance designation would be added by the framework or platform so that the application code can be agnostic of the runtime environment.

Kubernetes doesn’t do this. The IP address and port you see here are added through the Utils package in our implementation. I have, using the guidance from [chapter 6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06) on application configuration, taken care to abstract the details of the platform, injecting the IP address through an environment variable, but I’d much rather have the platform include designation such as this with absolutely no effort from the developer. The takeaway is that as an application developer, you may need to place attention on ensuring that your log output includes information that identifies which app instance the entry is attributed to.

When I talk about a platform for handling logs, aggregation is only one of the necessary elements. Logs need to be ingested at scale and must be stored, and interfaces must support search and analysis of these potentially large volumes of data. The ELK stack ([www.elastic.co/elk-stack](http://www.elastic.co/elk-stack)) draws together three open source projects—Elasticsearch, Logstash, and Kibana—to meet these requirements. Commercial offerings such as Splunk provide comparable capabilities. When you send your logs to stdout and stderr and ensure that the entries are attributable to specific app instances, you’re doing all that’s needed to allow systems such as these to provide powerful observability features. And you’ll allow for the preservation of log entries even when application containers disappear.

**11.2. Application metrics**

In addition to log data, application metrics are needed for holistic application monitoring. Metrics generally provide finer-grained insight into running applications than log files do; metrics are structured, whereas log files are usually unstructured or semistructured at best. Frameworks are virtually always used to both automatically generate a default set of metrics and allow for custom metrics to be emitted. Default metrics generally include values around memory and CPU consumption, as well as HTTP interactions (when appropriate). For a language such as Java, metrics around garbage collection and the class loader are often included as well.

You’ve already been using the Spring Framework for metrics, simply by including the actuator dependency. In addition to the /actuator/env endpoint that you previously used, the actuator provides an /actuator/metrics endpoint that emits the standard and custom metrics for your Spring Boot apps. The following shows output served by this endpoint on the Connections’ Posts service:

$ curl 35.232.22.58/actuator/metrics | jq

{

"mem": 853279,

"mem.free": 486663,

"processors": 2,

"instance.uptime": 2960448,

"uptime": 2975881,

"systemload.average": 1.33203125,

"heap.committed": 765440,

"heap.init": 120832,

"heap.used": 278776,

"heap": 1702400,

"nonheap.committed": 90584,

"nonheap.init": 2496,

"nonheap.used": 87839,

"nonheap": 0,

"threads.peak": 43,

"threads.daemon": 41,

"threads.totalStarted": 63,

"threads": 43,

"classes": 8581,

"classes.loaded": 8583,

"classes.unloaded": 2,

"gc.ps\_scavenge.count": 1019,

"gc.ps\_scavenge.time": 8156,

"gc.ps\_marksweep.count": 3,

"gc.ps\_marksweep.time": 643,

"httpsessions.max": -1,

"httpsessions.active": 0,

"gauge.response.metrics": 1,

"gauge.response.connectionPosts": 56,

"gauge.response.star-star": 20,

"gauge.response.login": 2,

"counter.span.accepted": 973,

"counter.status.200.metrics": 3,

"counter.status.404.star-star": 1,

"counter.status.200.connectionPosts": 32396,

"counter.status.200.login": 53

}

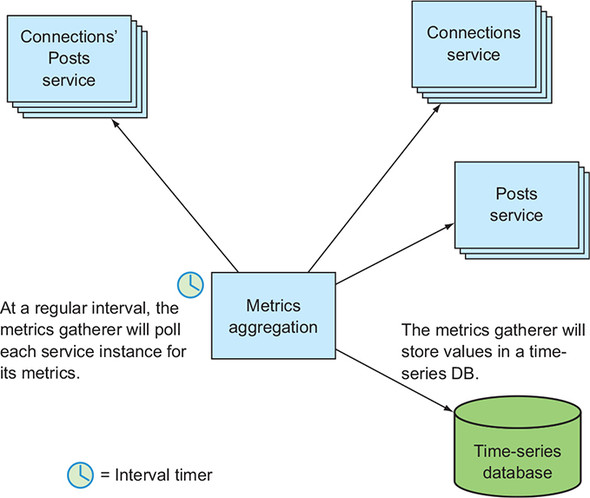
In addition to the values reported for memory, threads, and the class loader, notice that this instance has successfully served a great many (32,396) results on the /connectionsposts endpoint, several (53) on the /login endpoint, and three on the very /actuator/metrics endpoint you’re using to get this data. It also responded once with a 404 status code.

Application metrics have been in widespread use for far longer than cloud-native applications, and once again, I want to focus on what changes in this new cloud context. Just as with logs, a key concern is ensuring that metrics data is available even after a runtime environment is no longer available. You need to get the metrics out of the app and runtime context, and two basic approaches are available: a pull-based model and a push-based one.

**11.2.1. Pulling metrics from cloud-native applications**

In a *pull-based approach*, a metrics aggregator is implemented as a collector that requests metrics data from each of the application instances and stores those metrics in a time-series database ([figure 11.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig01)). It’s a bit like the curl to the /actuator/metrics endpoint that you saw just a moment ago; the collector as a client makes a request, and the app instance responds with the needed data.

**Figure 11.1. With a pull-based metrics-gathering approach, each service implements a metrics endpoint that’s accessed at a regular interval to capture and store the values for subsequent search and analysis.**

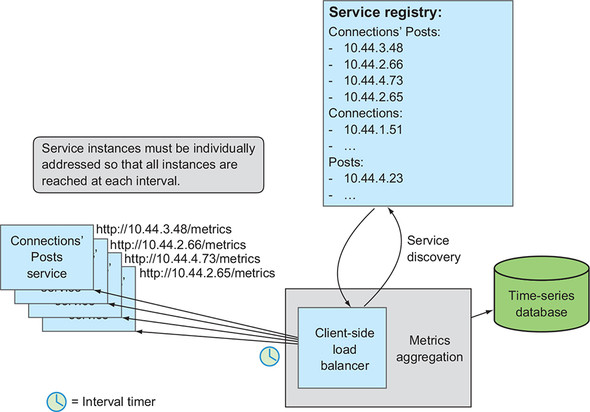


But what you’ve done with that curl isn’t quite right. In the days where you had only a single application instance, or you treated each of a handful of instances as a separate entity, making a request via HTTP was fine; you could target each app instance directly. But now, when you have multiple app instances that are load balanced, you’ll be getting metrics from only one of the app instances, and further, you don’t know which one. This sounds familiar, right? It’s the same issue you had with log entries not being associated with a specific instance, a problem you at least partially solved with the use of the Utils package that included an instance identity in the log entries. But it’s more than that; in collecting metrics, you want to consistently gather values for each of the instances at a regular interval, and load balancers typically won’t distribute requests as uniformly as you need.

The solution is to have the collector fully control which app instances it will pull from and at what interval. That collector wants to control where requests are made rather than allowing a load balancer to choose; again, I hope this sounds familiar. This is akin to client-side load balancing that you learned about in [chapter 8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08), and part of client-side load balancing is service discovery. [Figure 11.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig02) depicts the flow:

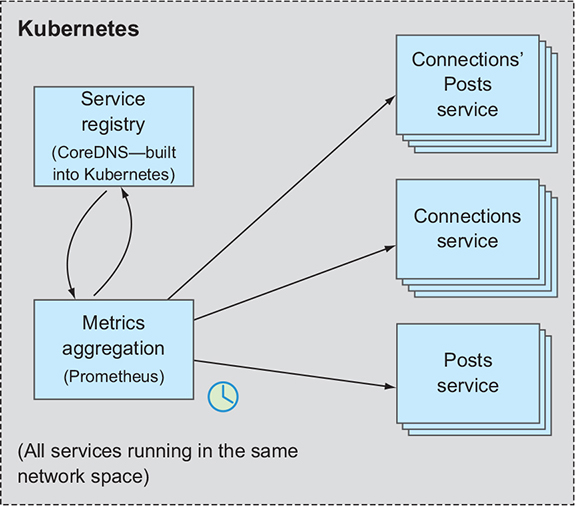
* At every interval, the collector requests metrics data from each instance.
* The set of instances is found through a service-discovery protocol, and how often the collector invokes this protocol to get the latest instance identities can vary. Doing so at every interval can be expensive, yet ensures that any changes in application topology are reflected as soon as possible. If it’s acceptable that metrics for a new instance to be left out of the collection for a short time, the service discovery protocol can be executed less frequently.
* Performing service discovery at an interval separate from the metrics collection one yields a more loosely coupled solution.

**Figure 11.2. The collector that implements the metrics aggregator must reach each service instance at every interval and must therefore control the load balancing. It will interact through a service discovery protocol to keep up to date with IP address changes.**



One complication with a pull-based approach is that the collector must, of course, have access to each of the instances from which it will request data. The IP address for each of the instances must be addressable from the metrics aggregator. Often, service instances are only individually addressable from well within an execution environment. You’ve seen this in our sample deployments, where only the Connections’ Posts service was available from outside the cluster—and therefore the metrics gatherer must also be deployed within that network space. A common deployment topology for Kubernetes-based environments is to deploy Prometheus (<https://prometheus.io/>) *inside* the Kubernetes cluster itself. With this deployment topology, Prometheus uses the embedded DNS service and has direct access to the application instances ([figure 11.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig03)).

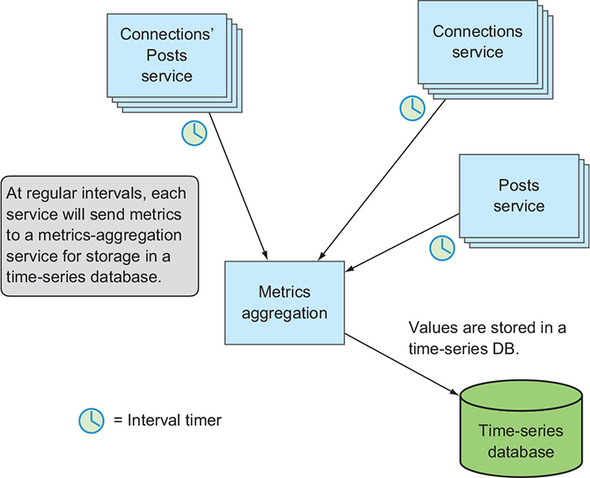
**Figure 11.3. The metrics gatherer must address each service instance individually, and because those IP addresses are accessible only from *within* a runtime environment such as Kubernetes (external access is via load balancer), the metrics gather is also generally deployed within that network space. In the case of Kubernetes, this also allows you to use the built-in DNS service for service discovery.**



**11.2.2. Pushing metrics from cloud-native applications**

An alternative to the pull-based model for gathering app metrics is a push-based one, whereby each app instance is responsible for delivering metrics to a metrics aggregator at a regular interval ([figure 11.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig04)). As an application developer, you might have an aversion to taking on the burden of delivering metrics data—working on code for that takes you away from the core business logic that’s delivering value to your customers and organization. The good news is that like many of the cross-cutting concerns I’ve spoken about throughout this text, much of the work of metrics generation and delivery is taken care of by our trusty frameworks and platforms.

**Figure 11.4. In a push-based metrics solution, each service sends metrics to an aggregation and storage service at a certain interval.**

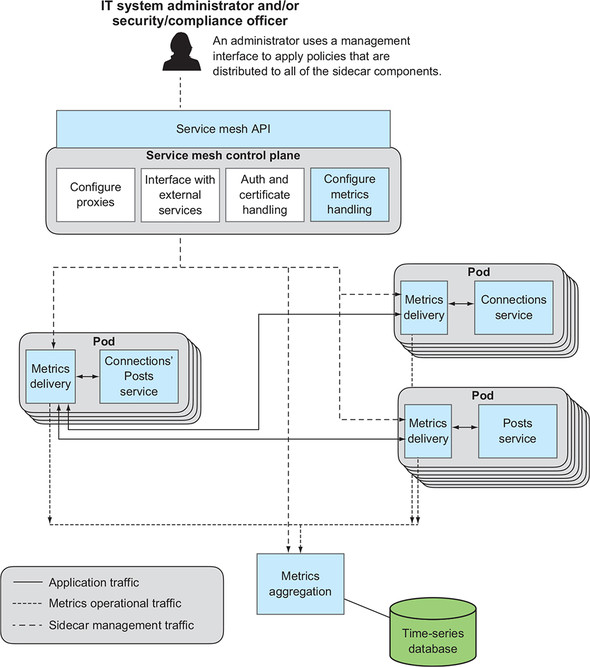


Frameworks delivering push-based metrics implementations generally do so with the use of an agent that takes care of gathering and delivering metrics to the metrics aggregator. The agent is usually compiled into the application binary with the inclusion of a dependency in something like a POM or Gradle build file. What is trickier, because of the constantly changing environment our apps and the agents live in, is proper configuration of that agent on deployment and during ongoing systems management.

For example, the IP address of the metrics aggregator must be configured into the running app so the agent knows where to send metrics. Using the best practices described in [chapter 6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_018.html#ch06), this is straightforward for initial deployment, but making configuration changes to an already running application must be done with care, as covered in [chapter 7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_019.html#ch07). You might be thinking this sounds like standard service discovery (also discussed earlier, in [chapter 8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_020.html#ch08)), but because metrics delivery often is fairly resource intensive, adding a service discovery protocol into the flow of delivering metrics may generate unacceptable latency.

One more backward reference: in [chapter 10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10), I talked about the sidecar providing API gateway functionality and implementing protocols around retries and circuit breakers. But the sidecar is also ideally suited for metrics collection. Recall that the sidecar is addressable from other containers in the pod via localhost, effectively shielding the app from changes in the metrics-gathering service. The in-app agent simply delivers metrics to the sidecar, and the sidecar then assumes responsibility for forwarding the data on to an external collector ([figure 11.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig05)). If the coordinates of the collector change, the app configuration doesn’t change, and therefore no app lifecycle operations are needed. The sidecar/service mesh is specially designed to handle the constant change present in cloud-native applications and now takes on that responsibility. For example, the control plane for the service mesh may push any new IP address out into the mesh, updating all the sidecars. And sidecars such as Envoy are designed to adapt to application configuration changes more easily with capabilities such as hot restarts.

**Figure 11.5. When a service mesh is used, the application services are simply configured to connect to the local sidecar proxy, and the service mesh control plane is used to keep the configuration of the metrics delivery components up to date.**



Did you notice? In the preceding discussion, I made a whole host of references to earlier chapters. Solving the metrics management problem for cloud-native apps is best done by applying cloud-native patterns. The example in this section is a great example.

Finally, having a sidecar proxy can provide value over and above simply proxying outbound metric pushes from the app, as it can also provide some level of observability even when no agent is installed into the application. Because it proxies traffic coming into and going out of the application, it can generate many metrics on behalf of the application. For example, counts of HTTP status codes, latencies, and more can be gathered or calculated, and delivered without anything being done within the application code.

This is an outstanding example of clever architectures and innovative frameworks allowing the separation of business concerns in the application from operational ones. Using the right platform can relieve the application developer of a great many concerns, allowing them to focus on the business outcomes of their code.

**11.3. Distributed tracing**

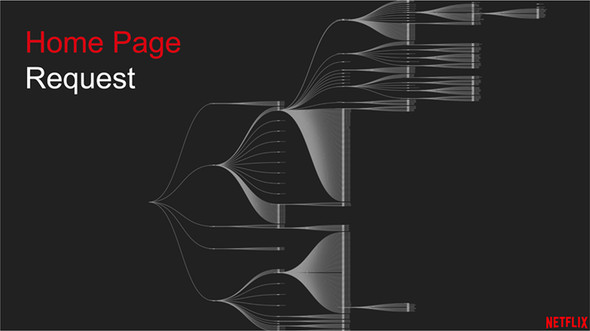
Let’s look at another capability that a combination of application frameworks and cloud-native platforms can bring. This capability, *distributed tracing*, is critically important for the highly distributed cloud-native application.

In an environment where our code is all running within the same process, we can use well-established tools to follow and troubleshoot the execution flow of an application. Source-level debugging will jump from method to method, and when configured correctly will even step into code that’s brought into the application via inclusion of a library (code that you haven’t written). When exceptions occur, the call stack that’s printed to the console and/or output to logs shows the sequence of calls that were made, something that’s often helpful in diagnosing problems.

But now an invocation of your application can result in a cascade of downstream requests that are usually running out of process and, in fact, are most often running in totally different runtime contexts (in different containers or on different hosts). How can you see the equivalent of a call stack, or simply get a view into what is happening as a result of an application invocation in this distributed scenario? The technique that has become prevalent in the industry and has solid tooling behind it is distributed tracing.

Distributed tracing, exactly as it sounds, is about tracing program flow across a distributed set of components. It’s what allows us to gain visibility into the fan-out of all of the downstream requests that result from, for example, a Netflix homepage access. In [figure 11.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig06), the point on the left side represents the homepage request, and the lines to other points represent calls to additional services made to gather the contents of a user’s homepage display.

**Figure 11.6. Diagram appearing in a presentation from Scott Mansfield of Netflix shows a request to the Netflix homepage results in a series of downstream service invocations. Distributed tracing allows you to gain visibility into this complex call tree.**



A rather popular technology being used today is Zipkin (<https://zipkin.io/>), a project modeled after the research on distributed tracing that was first published in the Google Dapper paper in 2010.**[**[**1**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fn1)**]** At the core of the technique are the following:

***1***

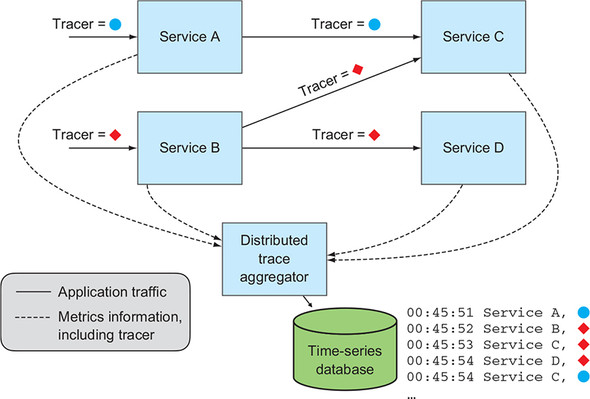
*The paper, “Dapper, a Large-Scale Distributed Systems Tracing Infrastructure,” is available from Google at*[*http://mng.bz/178V*](http://mng.bz/178V)*.*

* The use of *tracers*, unique identifiers that are inserted into requests and responses so that related app invocations can be found
* A control plane that uses these tracers to assemble the call graph for a set of what otherwise are independent (by design!) invocations

When a service is invoked and that service makes a downstream request to another service, any tracer included in the request to the former will be passed on to the latter. That tracer is then available in the runtime context of each of the services and—*here’s the key*—can be included in any metrics or log output. Using that tracer along with other data from the services context (timestamps, for example) allows you to piece together the flow through a set of services that in combination build the response to a service request.

In [figure 11.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig07), you can see a set of services and invocations that carry these tracers. Also depicted in this diagram is a database that collects output from each of the services—data that includes the tracer values. From the data stored in there, you can rebuild the “call stack” for the set of related component invocations. For example, you can see that a request coming into service A created a subsequent call to service C; and an unrelated request to service B also led to a downstream request to service C, followed by a request to service D.

**Figure 11.7. Requests carry tracers that are propagated in downstream requests. The tracers are then available in the runtime context of a service invocation and annotates data that’s aggregated into a distributed tracing service.**



To make this more concrete, let’s get our sample code running and look at new output.

**Setting up**

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

You’ll be accessing files in the cloudnative-troubleshooting 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 application**

You’ll need a Kubernetes cluster with sufficient capacity, as described in the first example of [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, let’s get those cleaned up. Run the script I’ve provided as follows:

./deleteDeploymentComplete.sh all

Running this deletes all instances of the Posts, Connections, and Connections’ Posts services, as well as MySQL, Redis, and SCCS that are running. If you have other things running in your Kubernetes cluster, you may want to clear some of that out. Just make sure you have enough capacity.

A slight start order dependency exists. After creating the MySQL server, you need to create the actual databases therein, so let’s get it and the other backing services running first:

./deployServices.sh

After the MySQL database is up and running, which you can see by running kubectl get all, you’ll create the database by using the MySQL CLI as follows:

mysql -h <public IP address of your MySQL service> \

-P <port for your MySQL service> -u root -p

The password is password. After you’re in, you’ll run the command to create the database:

create database cookbook;

Now you can start the microservices, which you do by running the following script:

./deployApps.sh

I’ll go into the details of the implementation in just a moment, but let’s start by invoking our Connections’ Posts service and viewing log output. You’ll first log in with the following command:

curl -i -X POST -c cookie \

<connectionsposts-svc IP>/login?username=cdavisafc

And then get your Connections’ Posts with the following:

curl -b cookie <connectionsposts-svc IP>/connectionsposts | jq

Now let’s look at the logs for each of our microservices. As I talked about in earlier sections, because you have multiple instances of each microservice, some type of log aggregation would be helpful, and although that offered in Kubernetes could be better, it’s sufficient for our purposes here. Run each of the following commands, and then you’ll study the results:

kubectl logs -l app=connectionsposts

kubectl logs -l app=connections

kubectl logs -l app=posts

**11.3.1. Tracer output**

The following three log output listings are excerpts from the output of each of these commands.

**Listing 11.1. Log output from Connections’ Posts**

2019-02-25 **02:20:11.969** [mycookbook-connectionsposts,**2e30**...,**2e30**...]

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg getting posts for user network cdavisafc

2019-02-25 **02:20:11.977** [mycookbook-connectionsposts,**2e30**...,**2e30**...]

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg connections = 2,3

**Listing 11.2. Log output from Connections**

2019-02-25 **02:20:11.974** [mycookbook-connections,**2e30**...,**9b5f**...] getting

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg connections for username cdavisafc

2019-02-25 **02:20:11.974** [mycookbook-connections,**2e30**...,**9b5f**...] getting

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg user cdavisafc

...

2019-02-25 **02:20:11.987** [mycookbook-connections,**2e30**...,**b915**...] getting

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg user 2

...

2019-02-25 **02:20:11.994** [mycookbook-connections,**2e30**...,**990f**...] getting

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg user 3

**Listing 11.3. Log output from Posts**

2019-02-25 **02:20:11.980** [mycookbook-posts,**2e30**...,**33ac**...] Accessing posts

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg using secret ...

2019-02-25 **02:20:11.980** [mycookbook-posts,**2e30**...,**33ac**...] getting posts

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg for userId 2

2019-02-25 **02:20:11.981**... [mycookbook-posts,**2e30**...,**33ac**...] getting posts

https://learning.oreilly.com/api/v2/epubs/urn:orm:book:9781617294297/files/enter.jpg for userId 3

The log output now includes new values enclosed in square brackets. The first is the application name. The second is the trace ID; this is exactly the tracer that I’ve been talking about. The third value is the span ID, used to identify each unique invocation of an app; for example, in the preceding log output, the Connections app was invoked three times, as indicated by the three span IDs (9b5f..., b915... and 990f...). The span ID can be used to correlate metrics or log outputs that are part of a single service execution.

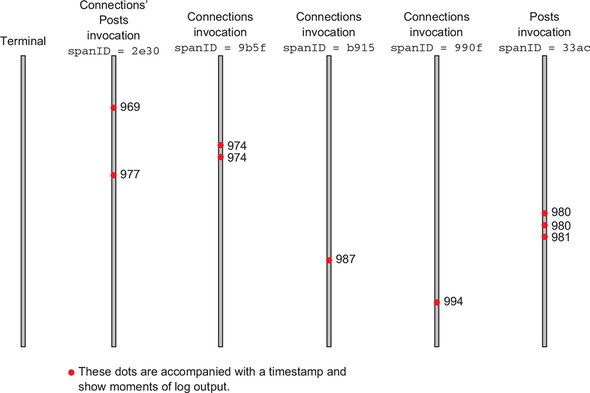
Studying the preceding output, which truncates the trace IDs and span IDs to the first four digits of hex numbers generated by the Spring Framework, you can see the following:

* When you curl the Connections’ Posts service, a trace ID starting with 2e30 is generated.
* Because the call is the outermost invocation, that number is also the span ID (2e30...) and represents the work being done to generate the list of posts for the people that cdavisafc follows.
* Any log output from the Connections’ Posts service has these values for trace ID and span ID.
* The Connections service was invoked three times:
  + Because all the outputs include a trace ID of 2e30, you know that these invocations were all downstream requests from your curl to Connections’ Posts.
  + Because this output has three span IDs, you know that the Connections service was invoked three times.
* The Posts service was invoked once. Because the trace ID is 2e30, you know that the invocation is a downstream request from your original curl command.
* Finally, at the beginning of each line of log output is a timestamp.

This data allows you to piece together part of the flow, as depicted in [figure 11.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig08). The dots are accompanied with a timestamp showing moments when log output was generated. This diagram shows the following:

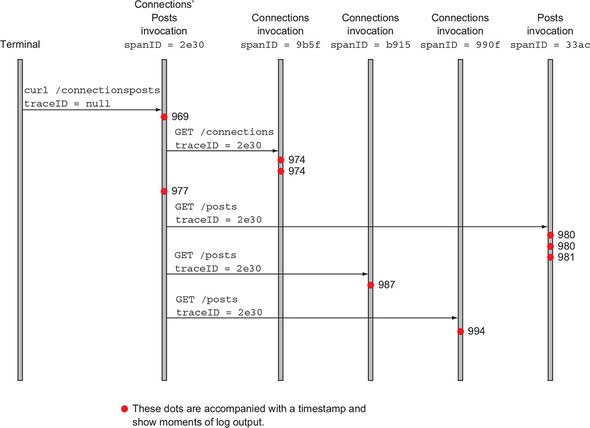
* (With your original curl) the Connections’ Posts service is called.
* The Connections service is called (to get a list of the users I follow).
* The Posts service is called (with the list of those connections to get a list of posts for those connections).
* (For each post returned from there, and there are two), the Connections service is called (to obtain the name of the user that made the post).

**Figure 11.8. From the annotated log output, you can piece together some of the flow of a single request—that with trace ID 2e30. Notice that you can’t see where the calls to the services came from.**



I’ve parenthesized parts of the preceding flow because those are the semantics that you and I know about our sample application, but the nonparenthesized parts of the phrases are what can be seen in the trace output. This is interesting; in [figure 11.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig08), you’re blind to the details that are within the parentheses in the preceding description. [Figure 11.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig09) fills in the details that come from these additional phrases.

**Figure 11.9. Here, the service invocations are overlaid on the time, trace ID, and span ID stamped log output shown in the previous figure. This information isn’t currently appearing in your logs.**



Note that the data represented in the arrows of [figure 11.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig09) could also be inserted into log output, but that’s not done with the Spring Cloud libraries.

**11.3.2. Assembling traces via Zipkin**

I’ve had you explore what is happening with the trace and span IDs by inspecting the logs, but tooling such as Zipkin allows you to analyze these types of values more effectively. Zipkin provides a data store for trace- and span-annotated metrics, and a user interface that displays data and supports navigation through that data.

Services are responsible for delivering the data to the Zipkin store, and this brings us to an important consideration. The act of sending data out of a service takes resources; it consumes memory, CPU cycles, and I/O bandwidth. The metrics that I spoke of earlier were scoped to a service. You were gathering data about how the running service was operating. The metrics I’m now covering are scoped to a service invocation. With the former, you could decide to gather metrics once a second, but if a service is responding to 100 requests per second and you’re gathering metrics for every invocation, that’s two orders of magnitude more resource intensive. As a result, the best practice for distributed tracing is to gather metrics for only a subset of all service requests.

I’ve covered this nuance here because I want you to apply the use of distributed tracing to experiments we ran in [chapters 9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09) and [10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10). You’re going to put our application under load, but you want to limit the impact that the act of tracing has on our system, so you need to configure it so that it emits tracing info for only a subset of the calls. In the deployment for each of our services, you’ll see a configuration such as the following:

- name: SPRING\_SLEUTH\_SAMPLER\_PERCENTAGE

value: "0.01"

This will cause 1% of requests to generate tracing metrics and send them to Zipkin (I will explain Spring Cloud Sleuth in a moment). Let’s now place load on the system. I’ve changed the volume of requests in our simulation, so I need you to upload a new JMeter configuration into Kubernetes with the following:

kubectl create configmap zipkin-jmeter-config \

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

You can then start the simulation running:

kubectl create -f loadTesting/jmeter-deployment.yaml

This is now repeating the accesses of the Connections’ Posts service, and for a subset of these requests, trace data is being stored in the Zipkin database. To access the URL for the Zipkin user interface, look up the IP address and port for the Zipkin service with the following command:

echo http://\

$(kubectl get service zipkin-svc \

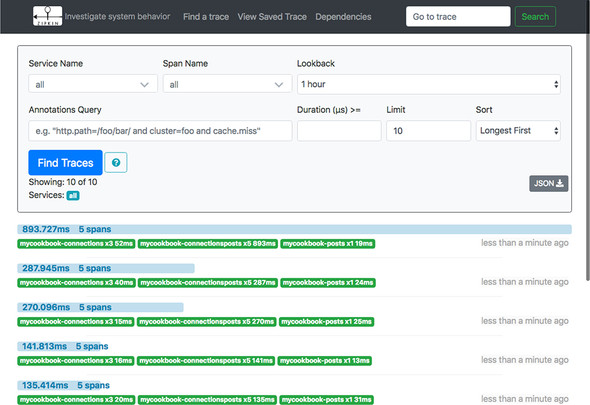
-o=jsonpath={.status.loadBalancer.ingress[0].ip})"/"\

$(kubectl get service zipkin-svc \

-o=jsonpath={.spec.ports[0].port})

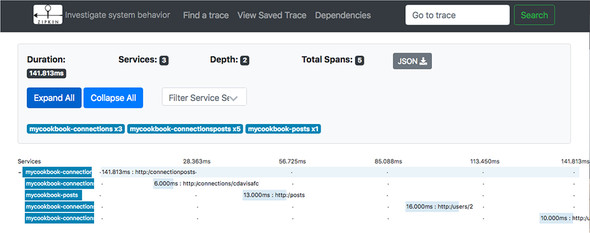
Access that URL in your browser and click the Find Traces button. This will show results such as in [figure 11.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig10).

**Figure 11.10. Zipkin provides a user interface that allows for search through the data stored in the distributed metrics database and pulls together entries that are related through a common trace ID. This display shows five such traces, each of which draws together five individual service requests, represented as *spans*.**



Displayed here are five traces through our distributed application. Each of these results corresponds to a separate curl to the Connections’ Posts service. The first took almost 900 ms to complete; the second and third, less than 300 ms; and the last two, less than 150 ms. Clicking on the light-grey bar for the fourth displayed invocation, the one that reads “141.813ms 5 spans,” renders the display shown in [figure 11.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig11).

**Figure 11.11. Zipkin display showing the details of a single request to the Connections’ Posts service. This service request resulted in four downstream requests: one to the Connections service, one to the Posts service, and then two more requests to the Connections service. The latency for each request, represented as a span, is also shown.**



This display lays out the spans that make up a single invocation, those spans having been gathered together because they share the same trace ID. Recall the log output you looked at just a moment ago and pieced together in [figure 11.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig08); in fact, [figure 11.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig11) is similar to [figure 11.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig08) laid on its side. You can see that the span for the Connections’ Posts service stretches for the full 141 ms, and you also see the spans for the downstream requests—the request to Connections to obtain the list of followed users, the request to the Posts service to get the list of posts, and the two requests back to the Connections service to get the names of the post authors. This is exactly the flow derived from the earlier log output.

Now let’s disrupt our system with a network outage, just as you did in [chapters 9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09) and [10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10). I’ve provided you with a script that will break the network connections between the instances of our Posts service and the MySQL database. You’ll have to update that script to point to your MySQL pod and to include the IP addresses of your Posts service instances. You can then invoke this script with the following command:

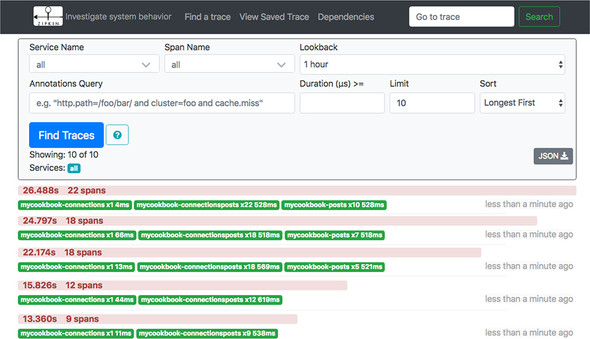
./loadTesting/alternetwork-db.sh add

Leave the network broken for 10–15 seconds and then restore the network by executing the following command:

./loadTesting/alternetwork-db.sh delete

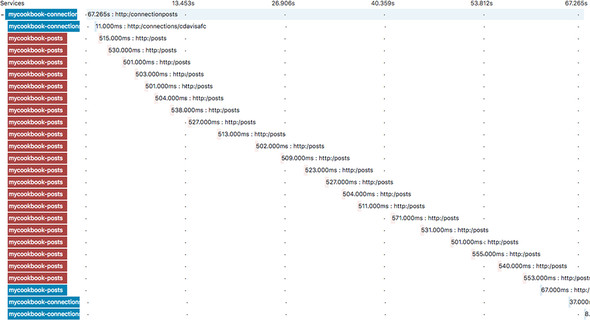
Let’s now return to the main Zipkin dashboard and click the Find Traces button. You’ll see something that looks like [figure 11.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig12).

**Figure 11.12. While the network was disrupted, requests to the Connections’ Posts service resulted in failed downstream requests. You see, for example, that what normally was satisfied with four downstream requests resulted in many more requests/spans. The bars reporting the time and number of spans (for example, “26.488s 22 spans”) are also now colored red, indicating that some of those downstream requests returned errors.**



The bars that represent the full length of a specific invocation (those reading “26.488s 22 spans”) have turned light red, giving you the first indication of trouble. Looking at the details, you see that each invocation has more than the five spans that you saw when the system was healthy. Clicking on one of these red bars shows the details shown in [figure 11.13](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig13).

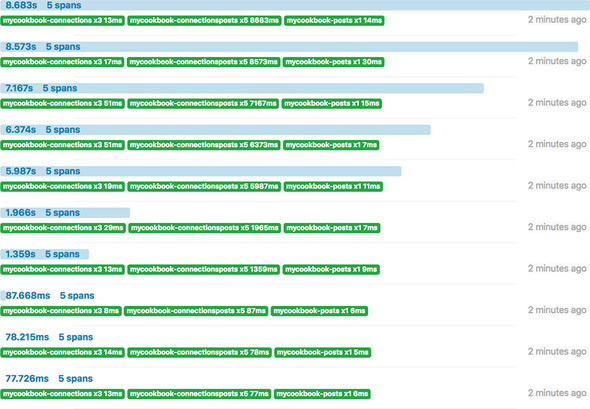
**Figure 11.13. The details of one of the request traces show repeated failed calls to the Posts service, each taking roughly 500 ms. This is the value of the request time-out on the HTTP invocations made from the Connections’ Posts service.**



Here you see the retries! Because of the network disruption between the Posts service and its database, the Posts service is unable to generate a response, and the request coming from the code of the Connections’ Posts service times out. You can now see that the version of the code that I pulled from [chapter 9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_021.html#ch09) into this project is that which does brute-force, or very unkind, retries. After the network connection is reestablished, the call to the Posts service succeeds, and the execution of the Connections Posts concludes.

Finally, returning to the Zipkin homepage and viewing the list of traces shortly after the restoration of the network, you see displayed the data shown in [figure 11.14](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_023.html#ch11fig14).

**Figure 11.14. When the network is restored, downstream requests again succeed, but you can see from the time taken to fulfill Connections’ Posts requests that a backlog of traffic had been generated and required time to dissipate.**



This shows that the time needed for Connections’ Posts processing returns to normal after the traffic from the retry storm dissipates.

By using distributed tracing techniques, you gained valuable insight into how our cloud-native application was performing. You were able to quickly see when and where errors were thrown, and you were able to track the route back to stability following the repair of an earlier outage. And the good news is that when using something such as the Spring Framework, these capabilities are easily added to an implementation.

**11.3.3. Implementation details**

Recall that at the core of distributed tracing are two specific techniques:

* Insertion of trace IDs
* A control plane that collects metrics that include these trace IDs, and uses them to link together related service invocations

These two concerns are addressed with the inclusion of two dependencies in your project POM files:

**Listing 11.4. Added to pom.xml of each of the three microservices**

<dependency>

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

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

<version>2.0.3.RELEASE</version>

</dependency>

<dependency>

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

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

<version>2.0.3.RELEASE</version>

</dependency>

Spring Cloud Sleuth instruments the generation and propagation of trace and span IDs. Including the first of the preceding dependencies causes these values to be included in the log files that you studied earlier. The second of these dependencies adds the delivery of the metrics to a Zipkin server, the address of which is configured into each of your services by setting the spring.zipkin.baseUrl property. You can see this setting, along with the sampler rate, in the Kubernetes deployment files for each of the services. (Notice that you’re addressing the Zipkin service via name; the service discovery protocol built into Kubernetes assists with the actual binding.)

**Listing 11.5. Added to deployment yaml file for each of the three microservices**

- name: SPRING\_APPLICATION\_JSON

value: '{"spring":{"zipkin":{"baseUrl":"http://zipkin-svc:9411/"}}}'

- name: SPRING\_SLEUTH\_SAMPLER\_PERCENTAGE

value: "0.01"

I’ve included a Zipkin service in the sample application deployment with the zipkin-deployment.yaml file.

And that’s it. That’s right—you don’t need to change anything else in the code to enable distributed tracing. It’s entirely handled by the Spring Framework. The value that distributed tracing brings is worth this level of effort, and even a bit more, should you be programming in a language that doesn’t provide the same level of support as this. At the time of writing, Zipkin libraries exist for Java, JavaScript, C#, Golang, Ruby, Scala, PHP, Python, and more. This broadly adopted technology is also included in other fabrics such as Istio (see [section 10.3.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10lev2sec6) in [chapter 10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_022.html#ch10).)

**Summary**

* Both metrics and log entries must be proactively pulled out of the runtime context in which our services are executing, because those execution environments are often unavailable after a service has either experienced trouble or has been upgraded. Execution environments for our services should be thought of as ephemeral.
* Aggregation of log entries from multiple instances of a service is important for observability. A solution that interleaves, in time order, entries from the different services is usually preferred.
* Collection of observability information, logs, metrics, and tracing data is effectively implemented in sidecar proxies, which allows the application to focus on business logic and concentrates the operational needs into the service mesh.
* Well-established distributed-tracing techniques, and implementations thereof, provide valuable insight into the health and performance of your distributed applications.
* Many of the patterns covered earlier in this book are used in the solutions that offer the necessary observability. Application configuration, application lifecycle, service discovery, gateways, and service mesh all come into play.