**Chapter 9. Distributed tracing with Spring Cloud Sleuth and Zipkin**

*This chapter covers*

* Using Spring Cloud Sleuth to inject tracing information into service calls
* Using log aggregation to see logs for distributed transaction
* Querying via a log aggregation tool
* Using OpenZipkin to visually understand a user’s transaction as it flows across multiple microservice calls
* Customizing tracing information with Spring Cloud Sleuth and Zipkin

The microservices architecture is a powerful design paradigm for breaking down complex monolithic software systems into smaller, more manageable pieces. These manageable pieces can be built and deployed independently of each other; however, this flexibility comes at a price: complexity. Because microservices are distributed by nature, trying to debug where a problem is occurring can be maddening. The distributed nature of the services means that you have to trace one or more transactions across multiple services, physical machines, and different data stores, and try to piece together what exactly is going on.

This chapter lays out several techniques and technologies for making distributed debugging possible. In this chapter, we look at the following:

* Using correlation IDs to link together transactions across multiple services
* Aggregating log data from multiple services into a single searchable source
* Visualizing the flow of a user transaction across multiple services and understanding the performance characteristics of each part of the transaction

To accomplish the three things you’re going to use three different technologies:

* *Spring Cloud Sleuth* (<https://cloud.spring.io/spring-cloud-sleuth/>)—Spring Cloud Sleuth is a Spring Cloud project that instruments your HTTP calls with correlation IDs and provides hooks that feed the trace data it’s producing into OpenZipkin. It does this by adding the filters and interacting with other Spring components to let the correlation IDs being generated pass through to all the system calls.
* *Papertrail* ([https://papertrailapp.com](https://papertrailapp.com/))—Papertrail is a cloud-based service (freemium-based) that allows you to aggregate logging data from multiple sources into single searchable database. You have options for log aggregation, including on-premise, cloud-based, open source, and commercial solutions. We’ll explore several of these alternatives later in the chapter
* *Zipkin* ([http://zipkin.io](http://zipkin.io/))—Zipkin is an open source data-visualization tool that can show the flow of a transaction across multiple services. Zipkin allows you to break a transaction down into its component pieces and visually identify where there might be performance hotspots.

To begin this chapter, we start with the simplest of tracing tools, the correlation ID.

|  |
| --- |
|  |

**Note**

Parts of this chapter rely on material covered in [chapter 6](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_015.html#ch06) (specifically the material on Zuul pre-, response, and post filters). If you haven’t read [chapter 6](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_015.html#ch06) yet, I recommend that you do so before you read this chapter.

|  |
| --- |
|  |

**9.1. SPRING CLOUD SLEUTH AND THE CORRELATION ID**

We first introduced the concept of correlation IDs in [chapter 5](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_014.html#ch05) and 6. A correlation ID is a randomly generated, unique number or string that’s assigned to a transaction when a transaction is initiated. As the transaction flows across multiple services, the correlation ID is propagated from one service call to another. In the context of [chapter 6](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_015.html#ch06), you used a Zuul filter to inspect all incoming HTTP requests and inject a correlation ID if one wasn’t present.

Once the correlation ID was present, you used a custom Spring HTTP filter on every one of your services to map the incoming variable to a custom UserContext object. With the UserContext object in place, you could now manually add the correlation ID to any of your log statements by making sure you appended the correlation ID to the log statement, or, with a little work, add the correlation ID directly to Spring’s Mapped Diagnostic Context (MDC). You also wrote a Spring Interceptor that would ensure that all HTTP calls from a service would propagate the correlation ID by adding the correlation ID to the HTTP headers on any outbound calls.

Oh, and you had to perform Spring and Hystrix magic to make sure the thread context of the parent thread holding the correlation ID was properly propagated to Hystrix. Wow—in the end this was a lot of infrastructure that was put in place for something that you hope will only be looked at when a problem occurs (using a correlation ID to trace what’s going on with a transaction).

Fortunately, Spring Cloud Sleuth manages all this code infrastructure and complexity for you. By adding Spring Cloud Sleuth to your Spring Microservices, you can

* Transparently create and inject a correlation ID into your service calls if one doesn’t exist.
* Manage the propagation of the correlation ID to outbound service calls so that the correlation ID for a transaction is automatically added to outbound calls.
* Add the correlation information to Spring’s MDC logging so that the generated correlation ID is automatically logged by Spring Boots default SL4J and Logback implementation.
* Optionally, publish the tracing information in the service call to the Zipkin--distributed tracing platform.

|  |
| --- |
|  |

**Note**

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

|  |
| --- |
|  |

Let’s go ahead and add Spring Cloud Sleuth to your licensing and organization services.

**9.1.1. Adding Spring Cloud sleuth to licensing and organization**

To start using Spring Cloud Sleuth in your two services (licensing and organization), you need to add a single Maven dependency to the pom.xml files in both services:

<dependency>

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

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

</dependency>

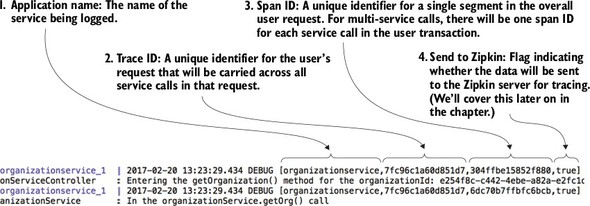
This dependency will pull in all the core libraries needed for Spring Cloud Sleuth. That’s it. Once this dependency is pulled in, your service will now

1. Inspect every incoming HTTP service and determine whether or a not Spring Cloud Sleuth tracing information exists in the incoming call. If the Spring Cloud Sleuth tracing data does exist, the tracing information passed into your microservice will be captured and made available to your service for logging and processing.
2. Add Spring Cloud Sleuth tracing information to the Spring MDC so that every log statement created by your microservice will be added to the logs.
3. Inject Spring Cloud tracing information into to every outbound HTTP call and Spring messaging channel message your service makes.

**9.1.2. Anatomy of a Spring Cloud Sleuth trace**

If everything is set up correctly, any log statements written within your service application code will now include Spring Cloud Sleuth trace information. For example, [figure 9.1](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig01) shows what the service’s output would look like if you were to do an HTTP GET http://localhost:5555/api/organization/v1/organizations/e254f8c c442-4ebe-a82a-e2fc1d1ff78a on the organization service.

**Figure 9.1. Spring Cloud Sleuth adds four pieces of tracing information to each log entry written by your service. This data helps tie together service calls for a user’s request.**



Spring Cloud Sleuth will add four pieces of information to each log entry. These four pieces (numbered to correspond with the numbers in [figure 9.1](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig01)) are

1. ***Application name of the service—***This is going to be the name of the application the log entry is being made in. By default, Spring Cloud Sleuth uses the name of the application (spring.application.name) as the name that gets written in the trace.
2. ***Trace ID—***Trace ID is the equivalent term for correlation ID. It’s a unique number that represents an entire transaction.
3. ***Span ID—***A span ID is a unique ID that represents part of the overall transaction. Each service participating within the transaction will have its own span ID. Span IDs are particularly relevant when you integrate with Zipkin to visualize your transactions.
4. ***Whether trace data was sent to Zipkin—***In high-volume services, the amount of trace data generated can be overwhelming and not add a significant amount of value. Spring Cloud Sleuth lets you determine when and how to send a transaction to Zipkin. The true/false indicator at the end of the Spring Cloud Sleuth tracing block tells you whether the tracing information was sent to Zipkin.

Up to now, we’ve only looked at the logging data produced by a single service call. Let’s look at what happens when you make a call to the licensing service at -GET http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc 1d1ff78a. Remember, the licensing service also has to call out to the organization service. [Figure 9.2](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig02) shows the logging output from the two service calls.

**Figure 9.2. With multiple services involved in a transaction, you can see that they share the same trace ID.**



By looking at [figure 9.2](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig02), you can see that both the licensing and organization services have the same trace ID a9e3e1786b74d302. However, the licensing service has a span ID of a9e3e1786b74d302 (the same value as the transaction ID). The organization service has a span ID of 3867263ed85ffbf4.

By adding nothing more than a few POM dependencies, you’ve replaced all the correlation ID infrastructure that you built out in [chapters 5](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_014.html#ch05) and [6](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_015.html#ch06). Personally, nothing makes me happier in this world then replacing complex, infrastructure-style code with someone else’s code.

**9.2. LOG AGGREGATION AND SPRING CLOUD SLEUTH**

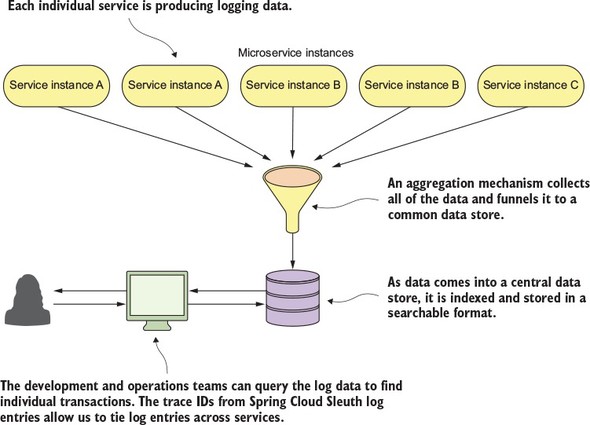
In a large-scale microservice environment (especially in the cloud), logging data is a critical tool for debugging problems. However, because the functionality for a microservice-based application is decomposed into small, granular services and you can have multiple service instances for a single service type, trying to tie to log data from multiple services to resolve a user’s problem can be extremely difficult. Developers trying to debug a problem across multiple servers often have to try the following:

* Log into multiple servers to inspect the logs present on each server. This is an extremely laborious task, especially if the services in question have different transaction volumes that cause logs to rollover at different rates.
* Write home-grown query scripts that will attempt to parse the logs and identify the relevant log entries. Because every query might be different, you often end up with a large proliferation of custom scripts for querying data from your logs.
* Prolong the recovery of a down service process because the developer needs to back up the logs residing on a server. If a server hosting a service crashes completely, the logs are usually lost.

Each of the problems listed are real problems that I’ve run into. Debugging a problem across distributed servers is ugly work and often significantly increases the amount of time it takes to identify and resolve an issue.

A much better approach is to stream, real-time, all the logs from all of your service instances to a centralized aggregation point where the log data can be indexed and made searchable. [Figure 9.3](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig03) shows at a conceptual level how this “unified” logging architecture would work.

**Figure 9.3. The combination of aggregated logs and a unique transaction ID across service log entries makes debugging distributed transactions more manageable.**



Fortunately, there are multiple open source and commercial products that can help you implement the previously described logging architecture. Also, multiple implementation models exist that will allow you to choose between an on-premise, locally managed solution or a cloud-based solution. [Table 9.1](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09table01)summarizes several of the choices available for logging infrastructure.

**Table 9.1. Options for Log Aggregation Solutions for Use with Spring Boot**

| **Product Name** | **Implementation Models** | **Notes** |
| --- | --- | --- |
| Elasticsearch, Logstash, Kibana (ELK) | Open source Commercial Typically implemented on premise | [http://elastic.co](http://elastic.co/) General purpose search engine Can do log-aggregation through the (ELK-stack) Requires the most hands-on support |
| Graylog | Open source Commercial On-premise | [http://graylog.org](http://graylog.org/) Open-source platform that’s designed to be installed on premise |
| Splunk | Commercial only On-premise and cloud-based | [http://splunk.com](http://splunk.com/) Oldest and most comprehensive of the log management and aggregation tools Originally an on-premise solution, but have since offered a cloud offering |
| Sumo Logic | Freemium Commercial Cloud-based | [http://sumologic.com](http://sumologic.com/) Freemium/tiered pricing model Runs only as a cloud service Requires a corporate work account to signup (no Gmail or Yahoo accounts) |
| Papertrail | Freemium Commercial Cloud-based | [http://papertrailapp.com](http://papertrailapp.com/) Freemium/tiered pricing model Runs only as a cloud service |

With all these choices, it might be difficult to choose which one is the best. Every organization is going to be different and have different needs.

For the purposes of this chapter, we’re going to look at Papertrail as an example of how to integrate Spring Cloud Sleuth-backed logs into a unified logging platform. I chose Papertrail because

1. It has a freemium model that lets you sign up for a free-tiered account.
2. It’s incredibly easy to set up, especially with container runtimes like Docker.
3. It’s cloud-based. While I believe a good logging infrastructure is critical for a microservices application, I don’t believe most organizations have the time or technical talent to properly set up and manage a logging platform.

**9.2.1. A Spring Cloud Sleuth/Papertrail implementation in action**

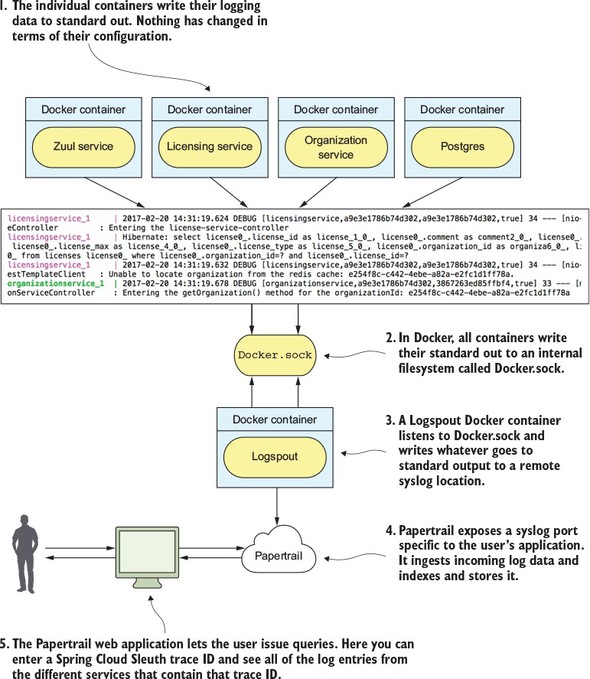
In [figure 9.3](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig03) we saw a general unified logging architecture. Let’s now see how the same architecture can be implemented with Spring Cloud Sleuth and Papertrail.

To set up Papertrail to work with your environment, we have to take the following actions:

1. Create a Papertrail account and configure a Papertrail syslog connector.
2. Define a Logspout Docker container (<https://github.com/gliderlabs/logspout>) to capture standard out from all the Docker containers.
3. Test the implementation by issuing queries based on the correlation ID from Spring Cloud Sleuth.

[Figure 9.4](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig04) shows the end state for your implementation and how Spring Cloud Sleuth and Papertrail fit together for your solution.

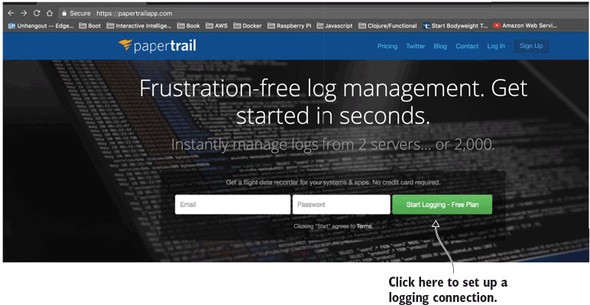
**Figure 9.4. Using native Docker capabilities, logspot, and Papertrail allows you to quickly implement a unified logging architecture.**



**9.2.2. Create a Papertrail account and configure a syslog connector**

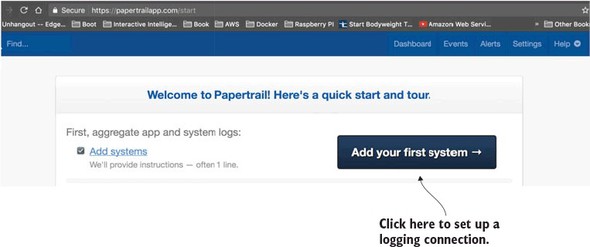
You’ll begin by setting up a Papertrail. To get started, go to [https://papertrailapp.com](https://papertrailapp.com/) and click on the green “Start Logging – Free Plan” button. [Figure 9.5](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig05) shows this.

**Figure 9.5. To begin, create an account on Papertrail.**



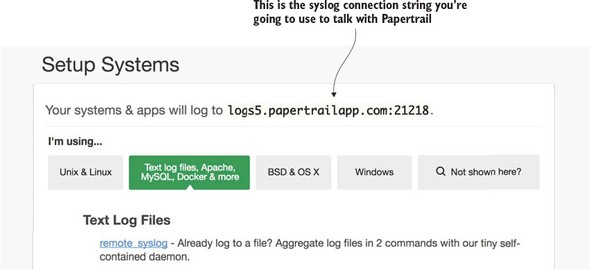
Papertrail doesn’t require a significant amount of information to get started; only a valid email address. Once you’ve filled out the account information, you’ll be presented with a screen to set up your first system to log data from. [Figure 9.6](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig06) shows this screen.

**Figure 9.6. Next choose how you’re going to send log data to Papertrail.**



By default, Papertrail allows you to send log data to it via a Syslog call (<https://en.wikipedia.org/wiki/Syslog>). Syslog is a log messaging format that originated in UNIX. Syslog allows for the sending of log messages over both TCP and UDP. Papertrail will automatically define a Syslog port that you can use to write log messages to. For the purposes of this discussion, you’ll use this default port. [Figure 9.7](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig07) shows you the Syslog connect string that’s automatically generated when you click on the “Add your first system” button shown in [figure 9.6](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig06).

**Figure 9.7. Papertrail uses Syslog as one of the mechanisms for sending data to it.**



At this point you’re all set up with Papertrail. You now have to configure your Docker environment to capture output from each of the containers running your services to the remote syslog endpoint defined in [figure 9.7](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig07).

|  |
| --- |
|  |

**Note**

The connection string from [figure 9.7](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig07) is unique to my account. You’ll need to make sure you use the connection string generated for you by Papertrail or define one via the Papertrail Settings > Log destinations menu option.

|  |
| --- |
|  |

**9.2.3. Redirecting Docker output to Papertrail**

Normally, if you’re running each of your services in their own virtual machine, you’ll have to configure each individual service’s logging configuration to send its logging information to a to a remote syslog endpoint (like the one exposed through Papertrail).

Fortunately, Docker makes it incredibly easy to capture all the output from any Docker container running on a physical or virtual machine. The Docker daemon communicates with all of the Docker containers it’s managing through a Unix socket called docker.sock. Any container running on the server where Docker is running can connect to the docker.sock and receive all the messages generated by all of the other containers running on that server. In the simplest terms, docker.sock is like a pipe that your containers can plug into and capture the overall activities going on within the Docker runtime environment on the virtual server the Docker daemon is running on.

You’re going to use a “Dockerized” piece of software called Logspout (<https://github.com/gliderlabs/logspout>) that will listen to the docker.sock socket and then capture any standard out messages generated in Docker runtime and redirect the output to a remote syslog (Papertrail). To set up your Logspout container, you have to add a single entry to the docker-compose.yml file you use to fire up all of the Docker containers used for the code examples in this chapter. The docker/common/docker-compose.yml file you need to modify should have the following entry added to it:

logspout:

image: gliderlabs/logspout

command: syslog://logs5.papertrailapp.com:21218

volumes:

- /var/run/docker.sock:/var/run/docker.sock

|  |
| --- |
|  |

**Note**

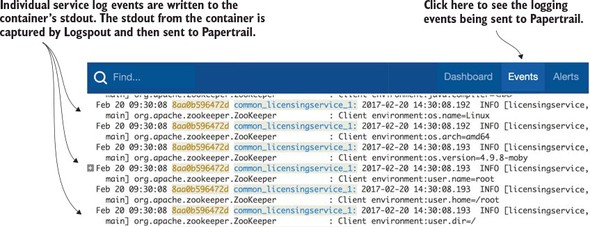
In the previous code snippet, you’ll need to replace the value in the “command” attribute with the value supplied to you from Papertrail. If you use the previous Logspout snippet, your Logspout container will happily write your log entries to my Papertrail account.

|  |
| --- |
|  |

Now when you fire up your Docker environment in this chapter, all data sent to a container’s standard output will be sent to Papertrail. You can see this for yourself by logging into your Papertrail account after you’ve started [chapter 9](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09)’s Docker examples and clicking on the Events button in the top right part of your screen.

[Figure 9.8](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig08) shows an example of what the data sent to Papertrail looks like.

**Figure 9.8. With the Logspout Docker container defined, data written to each container’s standard out will be sent to Papertrail.**



|  |
| --- |
|  |

**Why not use the Docker logging driver?**

Docker 1.6 and above do allow you to define alternative logging drivers to write the stdout/stderr messages written from each container. One of the logging drivers is a syslog driver that can be used to write the messages to a remote syslog listener.

Why did I choose Logspout instead of using the standard Docker log driver? The main reason is flexibility. Logspout offers features for customizing what logging data gets sent to your log aggregation platform. The features Logspout offers include

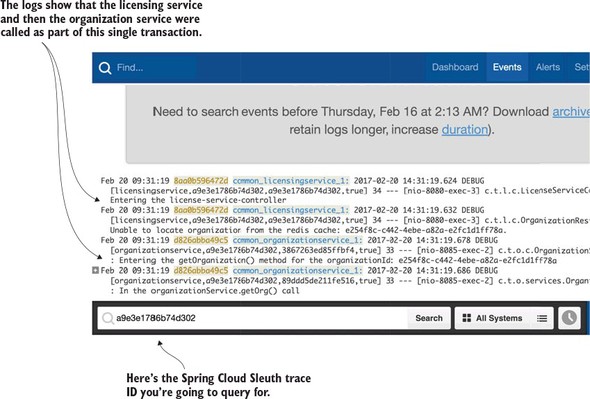
* *The ability to send log data to multiple endpoints at once.* Many companies will want to send their log data to a log aggregation platform, and will also want security monitoring tools that will monitor the produced logs for sensitive data.
* *A centralized location for filtering which containers are going to send their log data.* With the Docker driver, you need to manually set the log driver for each container in your docker-compose.yml file. Logspout lets you define filters to specific containers and even specific string patterns in a centralized configuration.
* *Custom HTTP routes that let applications write log information via specific HTTP endpoints.* This feature allows you to do things like write specific log messages to a specific downstream log aggregation platform. For example, you might have general log messages from stdout/stderr go to Papertrail, where you might want to send specific application audit information to an in-house Elasticsearch server.
* *Integration with protocols beyond syslog.* Logspout allows you to send messages via UDP and TCP protocols. Logspout also has third-party modules that can integrate the stdout/stderr from Docker into Elasticsearch.

|  |
| --- |
|  |

**9.2.4. Searching for Spring Cloud Sleuth trace IDs in Papertrail**

Now that your logs are flowing to Papertrail, you can really start appreciating Spring Cloud Sleuth adding trace IDs to all your log entries. To query for all the log entries related to a single transaction, all you need to do is take a trace ID and query for it in the query box of Papertrail’s event screen. [Figure 9.9](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig09)shows how to execute a query by the Spring Cloud sleuth trace ID we used earlier in [section 9.1.2](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09lev2sec2): a9e3e1786b74d302

**Figure 9.9. The trace ID allows you to filter all log entries related to that single transaction.**



|  |
| --- |
|  |

**Consolidate logging and praise for the mundane**

Don’t underestimate how important it is to have a consolidated logging architecture and a service correlation strategy thought out. It seems like a mundane task, but while I was writing this chapter, I used log aggregation tools similar to Papertrail to track down a race condition between three different services for a project I was working on. It turned out that the race condition has been there for over a year, but the service with the race condition had been functioning fine until we added a bit more load and one other actor in the mix to cause the problem.

We found the issue only after spending 1.5 weeks doing log queries and walking through the trace output of dozens of unique scenarios. We wouldn’t have found the problem without the aggregated logging platform that had been put in place. This experience reaffirmed several things:

1. ***Make sure you define and implement your logging strategies early on in your service development—***Implementing logging infrastructure is tedious, sometimes difficult, and time-consuming once a project is well underway.
2. ***Logging is a critical piece of microservice infrastructure—***Think long and hard before you implement your own logging solution or even try to implement an on-premise logging solution. Cloud-based logging platforms are worth the money that’s spent on them.
3. ***Learn your logging tools—***Almost every logging platform will have a query language for querying the consolidated logs. Logs are an incredible source of information and metrics. They’re essentially another type of database, and the time you spend learning to query will pay huge dividends.

|  |
| --- |
|  |

**9.2.5. Adding the correlation ID to the HTTP response with Zuul**

If you inspect the HTTP response back from any service call made with Spring Cloud Sleuth, you’ll see that the trace ID used in the call is never returned in the HTTP response headers. If you inspect the documentation for Spring Cloud Sleuth, you’ll see that the Spring Cloud Sleuth team believes that returning any of the tracing data can be a potential security issue (though they don’t explicitly list their reasons why they believe this.)

However, I’ve found that the returning of a correlation or tracing ID in the HTTP response is invaluable when debugging a problem. Spring Cloud Sleuth does allow you to “decorate” the HTTP response information with its tracing and span IDs. However, the process to do this involves writing three classes and injecting two custom Spring beans. If you’d like to take this approach, you can see it in the Spring Cloud Sleuth documentation (<http://cloud.spring.io/spring-cloud-static/spring-cloud-sleuth/1.0.12.RELEASE/>). A much simpler solution is to write a Zuul “POST” filter that will inject the trace ID in the HTTP response.

In [chapter 6](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_015.html#ch06) when we introduced the Zuul API gateway, we saw how to build a Zuul “POST” response filter to add the correlation ID you generated for use in your services to the HTTP response returned by the caller. You’re now going to modify that filter to add the Spring Cloud Sleuth header.

To set up your Zuul response filter, you need to add a single JAR dependencies to your Zuul server’s pom.xml file: spring-cloud-starter-sleuth. The spring-cloud-starter-sleuth dependency will be used to tell Spring Cloud Sleuth that you want Zuul to participate in a Spring Cloud trace. Later in the chapter, when we introduce Zipkin, you’ll see that the Zuul service will be the first call in any service invocation.

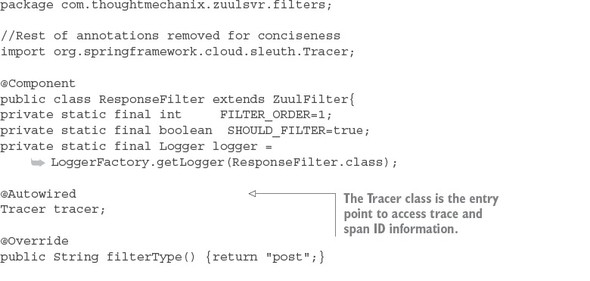
For [chapter 9](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09), this file can be found in zuulsvr/pom.xml. The following listing shows these dependencies.

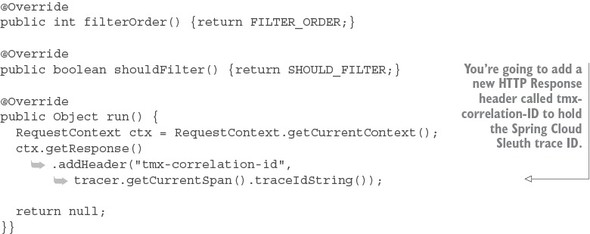
**Listing 9.1. Adding Spring Cloud Sleuth to Zuul**



Once this new dependency is in place, the actual Zuul “post” filter is trivial to implement. The following listing shows the source code used to build the Zuul filter. The file is located in zuulsvr/src/main/java/com/thoughtmechanix/zuulsvr/filters/ResponseFilter.java.

**Listing 9.2. Adding the Spring Cloud Sleuth trace ID via a Zuul POST filter**





Because Zuul is now Spring Cloud Sleuth-enabled, you can access tracing information from within your ResponseFilter by autowiring in the Tracer class into the ResponseFilter. The Tracer class allows you to access information about the current Spring Cloud Sleuth trace being executed. The tracer.getCurrentSpan().traceIdString() method allows you to retrieve as a String the current trace ID for the transaction underway.

It’s trivial to add the trace ID to the outgoing HTTP response passing back through Zuul. This is done by calling

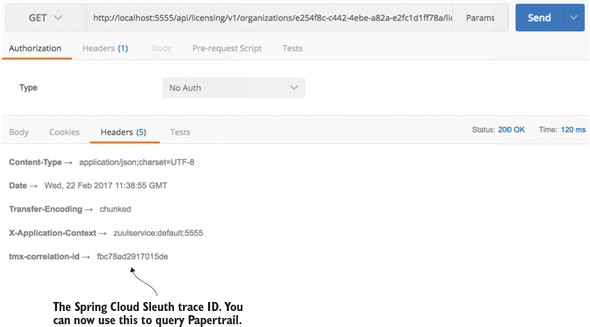
RequestContext ctx = RequestContext.getCurrentContext();

ctx.getResponse().addHeader("tmx-correlation-id",

https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/enter.jpg tracer.getCurrentSpan().traceIdString());

With this code now in place, if you invoke an EagleEye microservice through your Zuul gateway, you should get a HTTP response back called tmx-correlation-id. [Figure 9.10](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig10) shows the results of a call to GET http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a.

**Figure 9.10. With the Spring Cloud Sleuth trace ID returned, you can easily query Papertrail for the logs.**



[Figure 9.10](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig10) shows the results of a call to GET http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a.

**9.3. DISTRIBUTED TRACING WITH OPEN ZIPKIN**

Having a unified logging platform with correlation IDs is a powerful debugging tool. However, for the rest of the chapter we’re going to move away from tracing log entries and instead look at how to visualize the flow of transactions as they move across different microservices. A clean, concise picture can be work more than a million log entries.

Distributed tracing involves providing a visual picture of how a transaction flows across your different microservices. Distributed tracing tools will also give a rough approximation of individual microservice response times. However, distributed tracing tools shouldn’t be confused with full-blown Application Performance Management (APM) packages. These packages can provide out-of-the-box, low-level performance data on the actual code within your service and can also provider performance data beyond response time, such as memory, CPU utilization, and I/O utilization.

This is where Spring Cloud Sleuth and the OpenZipkin (also referred to as Zipkin) project shine. Zipkin (<http://zipkin.io/>) is a distributed tracing platform that allows you to trace transactions across multiple service invocations. Zipkin allows you to graphically see the amount of time a transaction takes and breaks down the time spent in each microservice involved in the call. Zipkin is an invaluable tool for identifying performance issues in a microservices architecture.

Setting up Spring Cloud Sleuth and Zipkin involves four activities:

* Adding Spring Cloud Sleuth and Zipkin JAR files to the services that capture trace data
* Configuring a Spring property in each service to point to the Zipkin server that will collect the trace data
* Installing and configuring a Zipkin server to collect the data
* Defining the sampling strategy each client will use to send tracing information to Zipkin

**9.3.1. Setting up the Spring Cloud Sleuth and Zipkin dependencies**

Up to now you’ve included two sets of Maven dependencies to your Zuul, licensing, and organization services. These JAR files were the spring-cloud-starter-sleuth and the spring-cloud-sleuth-core dependencies. The spring-cloud-starter-sleuth dependencies are used to include the basic Spring Cloud Sleuth libraries needed to enable Spring Cloud Sleuth within a service. The spring-cloud-sleuth-core dependencies are used whenever you have to programmatically interact with Spring Cloud Sleuth (which you’ll do again later in the chapter).

To integrate with Zipkin, you need to add a second Maven dependency called spring-cloud-sleuth-zipkin. The following listing shows the Maven entries that should be present in the Zuul, licensing, and organization services once the spring-cloud-sleuth-zipkin dependency is added.

**Listing 9.3. Client-side Spring Cloud Sleuth and Zipkin dependences**

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

**9.3.2. Configuring the services to point to Zipkin**

With the JAR files in place, you need to configure each service that wants to communicate with Zipkin. You do this by setting a Spring property that defines the URL used to communicate with Zipkin. The property that needs to be set is the spring.zipkin.baseUrl property. This property is set in each service’s application.yml properties file.

|  |
| --- |
|  |

**Note**

The spring.zipkin.baseUrl can also be externalized as a property in Spring Cloud Config.

|  |
| --- |
|  |

In the application.yml file for each service, the value is set to http://localhost:9411. However, at runtime I override this value using the ZIPKIN\_URI (http://zipkin:9411) variable passed on each services Docker configuration (docker/common/docker-compose.yml) file.

|  |
| --- |
|  |

**Zipkin, RabbitMQ, and Kafka**

Zipkin does have the ability to send its tracing data to a Zipkin server via RabbitMQ or Kafka. From a functionality perspective, there’s no difference in Zipkin behavior if you use HTTP, RabbitMQ, or Kafka. With the HTTP tracing, Zipkin uses an asynchronous thread to send performance data. The main advantage to using RabbitMQ or Kafka to collect your tracing data is that if your Zipkin server is down, any tracing messages sent to Zipkin will be “enqueued” until Zipkin can pick up the data.

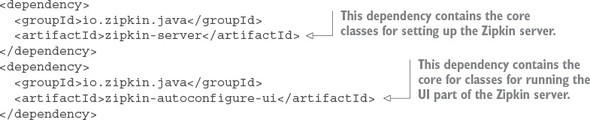
The configuration of Spring Cloud Sleuth to send data to Zipkin via RabbitMQ and Kafka is covered in the Spring Cloud Sleuth documentation, so we won’t cover it here in any further detail.

|  |
| --- |
|  |

**9.3.3. Installing and configuring a Zipkin server**

To use Zipkin, you first need to set up a Spring Boot project the way you’ve done multiple times throughout the book. (In the code for the chapter, this is call zipkinsvr.) You then need to add two JAR dependencies to the zipkinsvr/pom.xml file. These two jar dependences are shown in the following listing.

**Listing 9.4. JAR dependencies needed for Zipkin service**



|  |
| --- |
|  |

**@EnableZipkinServer vs. @EnableZipkinStreamServer: which annotation?**

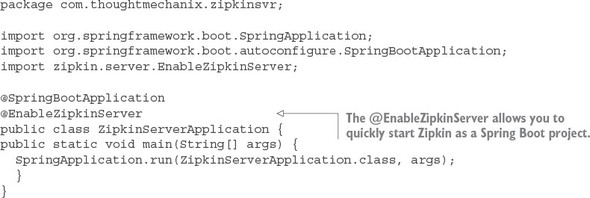
One thing to notice about the JAR dependencies above is that they’re not Spring-Cloud-based dependencies. While Zipkin is a Spring-Boot-based project, the @EnableZipkinServer is not a Spring Cloud annotation. It’s an annotation that’s part of the Zipkin project. This often confuses people who are new to the Spring Cloud Sleuth and Zipkin, because the Spring Cloud team did write the @EnableZipkinStreamServer annotation as part of Spring Cloud Sleuth. The @EnableZipkinStreamServer annotation simplifies the use of Zipkin with RabbitMQ and Kafka.

I chose to use the @EnableZipkinServer because of its simplicity in setup for this chapter. With the @EnableZipkinStream server you need to set up and configure the services being traced and the Zipkin server to publish/listen to RabbitMQ or Kafka for tracing data. The advantage of the @EnableZipkinStreamServer annotation is that you can continue to collect trace data even if the Zipkin server is unavailable. This is because the trace messages will accumulate the trace data on a message queue until the Zipkin server is available for processing the records. If you use the @EnableZipkinServer annotation and the Zipkin server is unavailable, the trace data that would have been sent by the service(s) to Zipkin will be lost.

|  |
| --- |
|  |

After the Jar dependencies are defined, you now need to add the @EnableZipkin Server annotation to your Zipkin services bootstrap class. This class is located in -zipkinsvr/src/main/java/com/thoughtmechanix/zipkinsvr/ZipkinServer Application.java. The following listing shows the code for the bootstrap class.

**Listing 9.5. Building your Zipkin servers bootstrap class**



The key thing to note in this listing is the use of the @EnableZipkinServer annotation. This annotation enables you to start this Spring Boot service as a Zipkin server. At this point, you can build, compile, and start the Zipkin server as one of the Docker containers for the chapter.

Little configuration is needed to run a Zipkin server. One of the only things you’re going to have to configure when you run Zipkin is the back end data store that Zipkin will use to store the tracing data from your services. Zipkin supports four different back end data stores. These data stores are

1. In-memory data
2. MySQL: [http://mysql.com](http://mysql.com/)
3. Cassandra: [http://cassandra.apache.org](http://cassandra.apache.org/)
4. Elasticsearch: [http://elastic.co](http://elastic.co/)

By default, Zipkin uses an in-memory data store for storing tracing data. The Zipkin team recommends against using the in-memory database in a production system. The in-memory database can hold a limited amount of data and the data is lost when the Zipkin server is shut down or lost.

|  |
| --- |
|  |

**Note**

For the purposes of this book, you’ll use Zipkin with an in-memory data store. Configuring the individual data stores used in Zipkin is outside of the scope of this book, but if you’re interested in the topic, you can find more information at the Zipkin GitHub repository (<https://github.com/openzipkin/zipkin/tree/master/zipkin-server>).

|  |
| --- |
|  |

**9.3.4. Setting tracing levels**

At this point you have the clients configured to talk to a Zipkin server and you have the server configured and ready to be run. You need to do one more thing before you start using Zipkin. You need to define how often each service should write data to Zipkin.

By default, Zipkin will only write 10% of all transactions to the Zipkin server. The transaction sampling can be controlled by setting a Spring property on each of the services sending data to Zipkin. This property is called spring.sleuth.sampler.percentage. The property takes a value between 0 and 1:

* A value of 0 means Spring Cloud Sleuth won’t send Zipkin any transactions.
* A value of .5 means Spring Cloud Sleuth will send 50% of all transactions.

For our purposes, you’re going to send trace information for all services. To do this, you can set the value of spring.sleuth.sampler.percentage or you can replace the default Sampler class used in Spring Cloud Sleuth with the AlwaysSampler. The AlwaysSampler can be injected as a Spring Bean into an application. For example, the licensing service has the AlwaysSampler defined as a Spring Bean in its licensing-service/src/main/java/com/thoughtmechanix/licenses/Application.java class as

@Bean

public Sampler defaultSampler() { return new AlwaysSampler();}

The Zuul, licensing, and organization services all have the AlwaysSampler defined in them so that in this chapter all transactions will be traced with Zipkin.

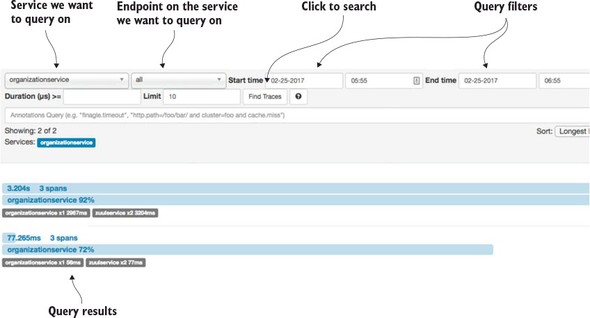
**9.3.5. Using Zipkin to trace transactions**

Let’s start this section with a scenario. Imagine you’re one of the developers on the EagleEye application and you’re on-call this week. You get a support ticket from a customer who’s complaining that one of the screens in the EagleEye application is running slow. You have a suspicion that the licensing service being used by the screen is running slow. But why and where? The licensing service relies on the organization service and both services make calls to different databases. Which service is the poor performer? Also, you know that these services are constantly being modified, so someone might have added a new service call into the mix. Understanding all the services that participate in the user’s transaction and their individual performance times is critical to supporting a distributed architecture such as a microservice architecture.

You’ll begin by using Zipkin to watch two transactions from your organization service as they’re traced by the Zipkin service. The organization service is a simple service that only makes a call to a single database. What you’re going to do is use POSTMAN to send two calls to the organization service (GET http://localhost:5555/api/organization/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a). The organization service calls will flow through a Zuul API gateway before the calls get directed downstream to an organization service instance.

After you’ve made two calls to the organization service, go to http://localhost:9411 and see what Zipkin has captured for trace results. Select the “organization service” from the dropdown box on the far upper left of the screen and then press the Find traces button. [Figure 9.11](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig11) shows the Zipkin query screen after you’ve taken these actions.

**Figure 9.11. The Zipkin query screen lets you select the service you want to trace on, along with some basic query filters.**

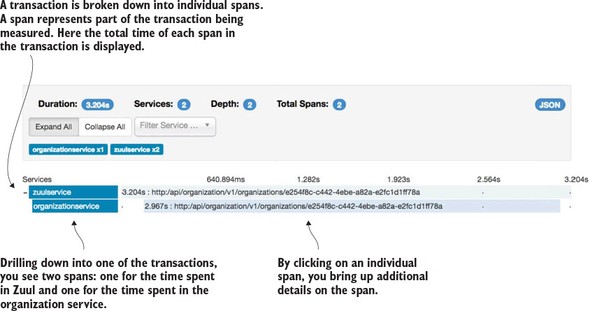


Now if you look at the screenshot in [figure 9.11](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig11), you’ll see that Zipkin captured two transactions. Each of the transactions is broken down into one or more spans. In Zipkin, a span represents a specific service or call in which timing information is being captured. Each of the transactions in [figure 9.11](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig11) has three spans captured in it: two spans in the Zuul gateway, and then a span for the organization service. Remember, the Zuul gateway doesn’t blindly forward an HTTP call. It receives the incoming HTTP call, terminates the incoming call, and then builds a new call out to the targeted service (in this case, the organization service). This termination of the original call is how Zuul can add pre-, response, and post filters to each call entering the gateway. It’s also why we see two spans in the Zuul service.

The two calls to the organization service through Zuul took 3.204 seconds and 77.2365 milliseconds respectively. Because you queried on the organization service calls (and not the Zuul gateway calls), you can see that the organization service took 92% and 72% of the total amount of time of the transaction time.

Let’s dig into the details of the longest running call (3.204 seconds). You can see more detail by clicking on the transaction and drilling into the details. [Figure 9.12](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig12) shows the details after you’ve clicked to drill down into further details.

**Figure 9.12. Zipkin allows you to drill down and see the amount of time each span in a transaction takes.**



In [figure 9.12](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig12) you can see that the entire transaction from a Zuul perspective took approximately 3.204 seconds. However, the organization service call made by Zuul took 2.967 seconds of the 3.204 seconds involved in the overall call. Each span presented can be drilled down into for even more detail. Click on the organizationservice span and see what additional details can be seen from the call. [Figure 9.13](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig13)shows the detail of this call.

**Figure 9.13. Clicking on an individual span gives further details on call timing and the details of the HTTP call.**



One of the most valuable pieces of information in [figure 9.13](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig13) is the breakdown of when the client (Zuul) called the organization service, when the organization service received the call, and when the organization service responded back. This type of timing information is invaluable in detecting and identifying network latency issues.

**9.3.6. Visualizing a more complex transaction**

What if you want to understand exactly what service dependencies exist between service calls? You can call the licensing service through Zuul and then query Zipkin for licensing service traces. You can do this with a GET call to the licensing serviceshttp://localhost:5555/api/licensing/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a endpoint.

[Figure 9.14](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig14) shows the detailed trace of the call to the licensing service.

**Figure 9.14. Viewing the details of a trace of how the licensing service call flows from Zuul to the licensing service and then through to the organization service**



In [figure 9.14](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig14), you can see that the call to the licensing service involves 4 discrete HTTP calls. You see the call to the Zuul gateway and then from the Zuul gateway to the licensing service. The licensing service then calls back through Zuul to call the organization service.

**9.3.7. Capturing messaging traces**

Spring Cloud Sleuth and Zipkin don’t trace HTTP calls. Spring Cloud Sleuth also sends Zipkin trace data on any inbound or outbound message channel registered in the service.

Messaging can introduce its own performance and latency issues inside of an application. A service might not be processing a message from a queue quickly enough. Or there could be a network latency problem. I’ve encountered all these scenarios while building microservice-based applications.

By using Spring Cloud Sleuth and Zipkin, you can identify when a message is published from a queue and when it’s received. You can also see what behavior takes place when the message is received on a queue and processed.

As you’ll remember from [chapter 8](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_017.html#ch08), whenever an organization record is added, updated, or deleted, a Kafka message is produced and published via Spring Cloud Stream. The licensing service receives the message and updates a Redis key-value store it’s using to cache data.

Now you’ll go ahead and delete an organization record and watch the transaction be traced by Spring Cloud Sleuth and Zipkin. You can issue a DELETE http://local-host:5555/api/organization/v1/organizations/e254f8c-c442-4ebe-a82a-e2fc1d1ff78a via POSTMAN to the organization service.

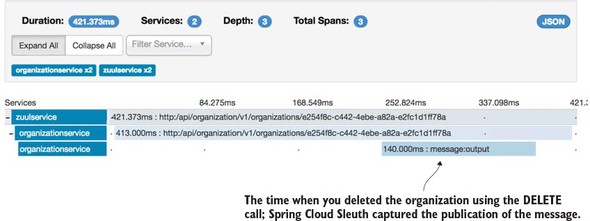
Remember, earlier in the chapter we saw how to add the trace ID as an HTTP response header. You added a new HTTP response header called tmx-correlation-id. In my call, I had the tmx-correlation-id returned on my call with a value of 5e14cae0d90dc8d4. You can search Zipkin for this specific trace ID by entering the trace ID returned by your call via the search box in the upper-right hand corner of the Zipkin query screen. [Figure 9.15](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig15) shows where you can enter the trace ID.

**Figure 9.15. With the trace ID returned in the HTTP Response tmx-correlation-id field you can easily find the transaction you’re looking for.**



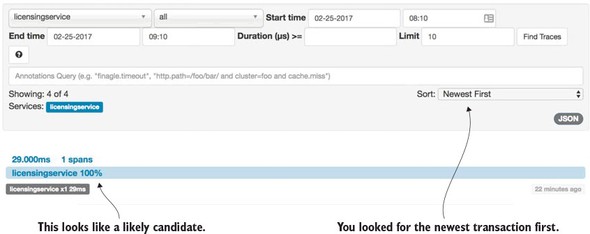
With the trace ID in hand you can query Zipkin for the specific transaction and can see the publication of a delete message to your output message change. This message channel, output, is used to publish to a Kafka topic call orgChangeTopic. [Figure 9.16](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig16) shows the output message channel and how it appears in the Zipkin trace.

**Figure 9.16. Spring Cloud Sleuth will automatically trace the publication and receipt of messages on Spring message channels.**



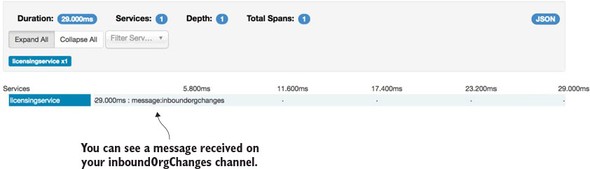
You can see the licensing service receive the message by querying Zipkin and looking for the received message. Unfortunately, Spring Cloud Sleuth doesn’t propagate the trace ID of a published message to the consumer(s) of that message. Instead, it generates a new trace ID. However, you can query Zipkin server for any license service transactions and order them by newest message. [Figure 9.17](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig17) shows the results of this query.

**Figure 9.17. You’re looking for the licensing service invocation where a Kafka message is received.**



Now that you’ve found your target licensing service transaction, you can drill down into the transaction. [Figure 9.18](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig18) shows the results of this drilldown.

**Figure 9.18. Using Zipkin you can see the Kafka message being published by the organization service.**



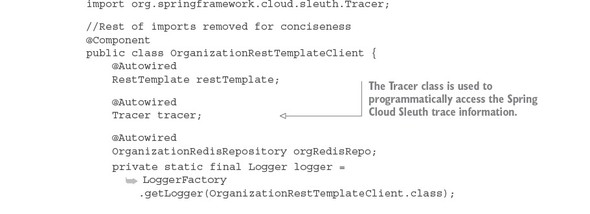
Until now you’ve used Zipkin to trace your HTTP and messaging calls from within your services. However, what if you want to perform traces out to third-party services that aren’t instrumented by Zipkin? For example, what if you want to get tracing and timing information for a specific Redis or Postgres SQL call? Fortunately, Spring Cloud Sleuth and Zipkin allow you to add custom spans to your transaction so that you can trace the execution time associated with these third-party calls.

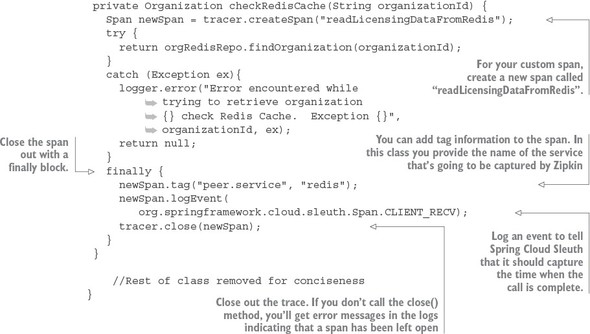
**9.3.8. Adding custom spans**

Adding a custom span is incredibly easy to do in Zipkin. You can start by adding a custom span to your licensing service so that you can trace how long it takes to pull data out of Redis. Then you’re going to add a custom span to the organization service to see how long it takes to retrieve data from your organization database.

To add a custom span to the licensing service’s call to Redis, you’re going to instrument the licensing-service/src/main/java/com/thoughtmechanix/licenses/clients/OrganizationRestTemplateClient.javaclass. In this class you’re going to instrument the checkRedisCache() method. The following listing shows this code.

**Listing 9.6. Instrumenting the call to read licensing data from Redis**





The code in [listing 9.6](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09ex06) creates a custom span called readLicensingDataFrom-Redis. Now you’ll also add a custom span, called getOrgDbCall, to the organization service to monitor how long it takes to retrieve organization data from the Postgres database. The trace for organization service database calls can be seen in the organization-service/src/main/java/com/thoughtmechanix/organization/services/OrganizationService.javaclass. The method containing the custom trace is the getOrg() method call.

The following listing shows the source code from the organization service’s getOrg() method.

**Listing 9.7. The instrumented getOrg() method**

package com.thoughtmechanix.organization.services;

//Removed the imports for conciseness

@Service

public class OrganizationService {

@Autowired

private OrganizationRepository orgRepository;

@Autowired

private Tracer tracer;

@Autowired

SimpleSourceBean simpleSourceBean;

private static final Logger logger =

https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/enter.jpg LoggerFactory.getLogger(OrganizationService.class);

public Organization getOrg (String organizationId) {

Span newSpan = tracer.createSpan("getOrgDBCall");

logger.debug("In the organizationService.getOrg() call");

try {

return orgRepository.findById(organizationId);

}finally{

newSpan.tag("peer.service", "postgres");

newSpan

.logEvent(

org.springframework.cloud.sleuth.Span.CLIENT\_RECV);

tracer.close(newSpan);

}

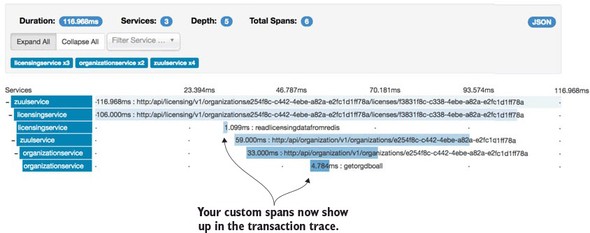
}

//Removed the code for conciseness

}

With these two custom spans in place, restart the services and then hit the GET http://localhost:5555/api/licensing/v1/organizations/e254f8c-c442 4ebe-a82a-e2fc1d1ff78a/licenses/f3831f8c-c338-4ebe-a82a-e2fc1d1ff78a endpoint. If we you look at the transaction in Zipkin, you should see the addition of the two additional spans. [Figure 9.19](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig19) shows the additional custom spans added when you call the licensing service endpoint to retrieve licensing information.

**Figure 9.19. With the custom spans defined, they’ll now show up in the transaction trace.**



From [figure 9.19](https://www.safaribooksonline.com/library/view/spring-microservices-in/9781617293986/kindle_split_018.html#ch09fig19) you can now see additional tracing and timing information related to your Redis and database lookups. You can break out that the read call to Redis took 1.099 milliseconds. Since the call didn’t find an item in the Redis cache, the SQL call to the Postgres database took 4.784 milliseconds.

**9.4. SUMMARY**

* Spring Cloud Sleuth allows you to seamlessly add tracing information (correlation ID) to your microservice calls.
* Correlation IDs can be used to link log entries across multiple services. They allow you to see the behavior of a transaction across all the services involved in a single transaction.
* While correlation IDs are powerful, you need to partner this concept with a log aggregation platform that will allow you to ingest logs from multiple sources and then search and query their contents.
* While multiple on-premise log aggregation platforms exist, cloud-based services allow you to manage your logs without having to have extensive infrastructure in place. They also allow you to easily scale as your application logging volume grows.
* You can integrate Docker containers with a log aggregation platform to capture all the logging data being written to the containers stdout/stderr. In this chapter, you integrated your Docker containers with Logspout and an online cloud logging provider, Papertrail, to capture and query your logs.
* While a unified logging platform is important, the ability to visually trace a transaction through its microservices is also a valuable tool.
* Zipkin allows you to see the dependencies that exist between services when a call to a service is made.
* Spring Cloud Sleuth integrates with Zipkin. Zipkin allows you to graphically see the flow of your transactions and understand the performance characteristics of each microservice involved in a user’s transaction.
* Spring Cloud Sleuth will automatically capture trace data for an HTTP call and inbound/outbound message channel used within a Spring Cloud Sleuth enabled service.
* Spring Cloud Sleuth maps each of the service call to the concept of a span. Zipkin allows you to see the performance of a span.
* Spring Cloud Sleuth and Zipkin also allow you to define your own custom spans so that you can understand the performance of non-Spring-based resources (a database server such as Postgres or Redis).