Logging and Tracing with Context Information

Take a deep dive into log4j version 2 and Zipkin to learn how to add context information in your logs, use trace IDs to correlate log entries across services, and how to switch on debug logs for single requests.

Figuring out what your application is doing

It is a team effort to deliver high quality software. But when a misfortunate event happens to one of your users, it is on you as a developer to find out what happened.

The user's ticket is your starting point: The user reports that he wasn't able to print the invoices. Let's turn to the logs on the server to find out what happened. How well are you prepared? Let's have a look at the first log entry (**Listing 1**).

There is a first clue: the ID of an item belonging to the invoice. How does this solve the case when there are many items on one invoice, and item ID 4711 is very common? Given the following code, how could we put in some breadcrumbs into the log to analyze this in a better way? (**Listing 2**).

At least the invoice ID and some user related information would have been helpful in the log. Let's see how this can be done. You'll find all code examples in the Git repository referenced at the end of this article.

```
07:26:00.595 d.a.t.d.Invoice ERROR – can't load item ID 4711
```

Listing 1

```
for (Invoice i : invoiceRepository.findAll()) {
    i.calculateTotal();
}
```

Listing 2

```
for (Invoice i : repository.findAll()) {
    MDC.put("invoiceId", Long.toString(i.getId()));
    try {
        i.calculateTotal();
    } finally {
        MDC.remove("invoiceId");
    }
}
```

Listing 3

Logging Framework log4j

The log4j logging framework has come a long way: Version 1.2 was around from May 2002 to 2015. Version 2.0 was released in July 2014, with 2.8 being available in January 2017.

Chances are good that you have come across log4j before. Most of the features described in this article have been part of the 1.2 version already, except when we point out that it's a 2.x feature.

Feeding information to the context

There are two ways to feed information to the logging context of log4 1.x: The Mapped Diagnostic Context (MDC) and the Nested Diagnostic Context (NDC). They behave like a map and a stack where you can store information that will be put into the log automatically on every log statement of the current thread. Depending on the framework you're using the context information might be passed to asynchronous tasks automatically.

In the example given above it would have been helpful to have the invoice number as part of the log statement. The following piece of code puts this information into the *MDC* (**Listing 3**).

Alexander Schwartz is a Principal IT Consultant at msg systems. He's been in Web development for more than 15 years and enjoys productive working environments, agile projects and automated tests. At conferences and user group meetings he talks about the things he is passionate about.

You can put any value into the map as long as it is a string - and as long as you remove it after your processing. If you wouldn't clean it up, the value would stay in the context of the current thread no matter what it would execute next.

When using log4j 2.x *MDC* and *NDC* have been merged to *ThreadContext*. If you prefer the old syntax with MDC, use slf4j as a logging facade.

If you're using a PatternLayout in your log4j configuration you'll just need to add %X to print the MDC contents on every log entry. If you are using a JSONLayout, this information will be included automatically in the log (**Listing 4**).

The new log entry produced by the updated code and the updated log configuration contains the invoice ID (**Listing 5**).

Now we can narrow it down to a single invoice that is problematic.

Automatically collecting context information

We've updated the code to add the context related to the invoice. But logging (as tracing) is also a crosscutting concern: In most applications there's a user, a URL, and a remote IP for every processed request.

When you're running in a Java EE environment, you could instantiate a Servlet Filter. If you're running in JAX-RS environment, you can implement a context handler for all incoming and outgoing requests (please see *RequestFilter* and *ReponseFilter* in the examples project).

Once you have these filters in place, your log statement will be enhanced without changing a single line of business code (**Listing 6**).

Tracing calls within and across services

With the information we've collected so far we can relate log entries to a user. Looking at the timestamp of the logs we can see the user's retry attempts. But in the logs we don't have an identifier to aggregate the log entries of an attempt. Furthermore, we don't have information to correlate logs across services.

Google's Dapper concept describes how to trace calls within a service and across services: When the request starts on the first service, it's assigned a random trace ID. When there is a request from the first service to a remote service, this call is assigned both the trace ID and a new span ID. When there is a second call to a remote service, it is assigned the same trace ID and different span ID. It also records start and end times on client and server side - but more on that later.

Dapper is an internal tool at Google that was described in one of Google's research papers. The paper outlined the concepts and named the additional HTTP headers that are used to forward trace information between services. Zipkin implements a central store that can search and visualize the traces. It also has client libraries to transmit traces from a service to the central store. Zipkin has its roots at Twitter, but all code is now available together with several additional libraries as part of the OpenZipkin project.

To collect trace information in your applications there are several libraries to help you: As part of the OpenZipkin project there are the brave libraries that wrap existing server and client side Java components to collect the trace information. There are other libraries to handle Go, JavaScript and other languages. When you're using Spring Boot there is Spring Sleuth that will instrument your classes to collect the trace information once you add it as a dependency to your project.

```
<PatternLayout pattern="%d{HH:mm:ss.SSS} %X %-5level ..."/>
```

Listing 4

```
08:39:42.969 {invoiceId=1} ... – can't load item ID 4711
```

Listing 5

```
08:52:54.276 {http.method=GET,
http.url=http://localhost:8080/api/startBillingRun, ...,
user=Theo Tester} ERROR d.a.t.d.Invoice – can't load item ID
4711
```

Listing 6

The following examples assume that you are using Spring Boot with Sleuth to collect the traces in your application, and Zipkin to store, search and visualize the traces.

Technicalities to trace calls across services

In order to send the trace information to a remote service there are standardized HTTP headers (**Listing 7**).

```
GET /api/callback HTTP/1.1
Host: localhost:8080
...
X-B3-SpanId: 34e628fc44c0cff1
X-B3-TraceId: a72f03509a36daae
...
```

Listing 7

Sleuth adds these headers to outgoing requests from the client and parses them when they arrives at the server. It also places it into the Mapped Diagnostic Context. This leads to a log statement like this (**Listing 8**).

```
09:20:05.840 {X-B3-SpanId=34e628fc44c0cff1, X-B3-
TraceId=a72f03509a36daae, ...} ERROR d.a.t.d.Invoice - can't
load item ID 4711
```

Listing 8

Now we can search our logs: all entries with the same trace ID across all our services will originate from a single user's request.

You can also extract the trace ID and append it to any error message you show to your user. This way you'll have it at hand once the user opens a ticket and you can start filtering your logs immediately.

Running Zipkin full scale

But Zipkin can do more: it can collect your trace and span information including timing information and provide you with an overview of your service landscape. This allows you to search for a specific entry, or entries that produced errors or took particularly long. These information are collected separately from your logs. In production environments you usually collect only a small percentage of the traces to avoid storing too much data.

Zipkin comes with a web frontend, a server application and a backend to store the information (either Cassandra, MySQL or Elasticsearch). The web frontend allows you to generate a map of the interactions of your services and to drill down single requests.

Not Only Developer Tools for Zipkin

A server will only create a new trace ID when the incoming request lacks this information. This fact is used by the Chrome Zipkin plugin: it will create the trace ID in the browser and forward it to the server. This way the trace ID is known in advance: from the developer tools you can use it to directly link to the web frontend of the Zipkin server. It also sets the additional 'X-B3-Sampled' header: This ensures that the trace is forwarded to the Zipkin server, independent of the trace configuration of the application.

Sleuth takes care that this header received by the first server is forwarded to all other servers handling the request, so all parts of the trace are collected and forwarded to the Zipkin server.

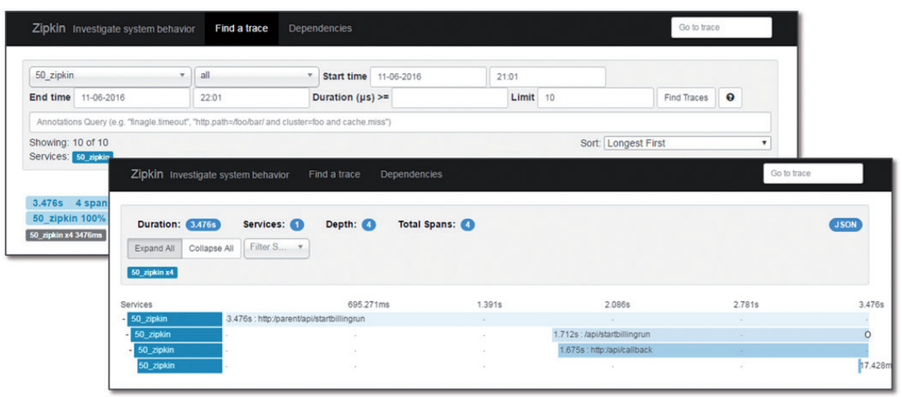

Figure 1. Zipkin Web UI to search and explore traces

```
<DynamicThresholdFilter key="X-B3-Flags-debug" onMatch="ACCEPT"
defaultThreshold="warn" onMismatch="NEUTRAL">
    <KeyValuePair key="true" value="trace"/>
</DynamicThresholdFilter>
```

Listing 9

This fact has another benefit: you can't only use this in a development environment, but also in the production environment to collect traces for a specific request where you normally sample only a small percentage of your requests. A caveat of the plugin: it sends the HTTP headers for all browser requests. Be sure to deactivate the plugin when you don't need it.

Thinking this through, you'll probably want to block any 'X-B3-*' HTTP header from reaching your servers from the outside or someone might trigger your debug logs when you don't want to.

Per-Request Debugging with log4j2 and Zipkin

The additional HTTP headers set by the developer tools allow you to collect the traces in production. But what about the log4j logs? They are usually set to warn-only in production.

Log4j 2.x ships with a solution for this: Zipkin also forwards a 'X-B3-Flags' header between servers with the value '1' representing 'debug on'. Read this information and place into the MDC as a key 'X-B3-Flags-debug'. With the following configuration *DynamicThresholdFilter* will read it and create log statements at trace level for every request with a header 'X-B3-Flags' of value '1'. (**Listing 9**).

Just make sure you are using log4j 2.7 or later for this: In earlier versions, there was a bug that prevented this to work properly for log statements with parameters (see ticket LOG4J2-1511 for details).

Choosing the right tools for your environment

This article showed many possibilities to enrich your log statements. But where should you start?

Usually the filter to store request information like user name and HTTP URL in the MDC is a place to begin: as a cross cutting implementation it will enrich all logs and it will not need any additional libraries. It will even work with older log4j version and other logging implementations like Logback.

If you are in a microservice environment where calls are forwarded to other services, implementing the Dapper concept is also a low hanging fruit. It comes with the cost of an additional library in your application. It will enhance the logs of your application even if you are not using a central Zipkin server to store the traces.

Once Dapper like tracing is in place you can start adding trace IDs to error messages to connect an error reported by the user to the entries in the log.

After that (and if you are able to update to log4j 2.7 or later) you might want to look into per-request debugging.

Taking these steps will prepare you for the next mystery case. It will be only a matter of time until it knocks on your door. ■

REFERENTIES

Log4j 2, especially Contexts **https://logging.apache.org/log4j/2.x/manual/thread-context.html**
Zipkin **https://github.com/openzipkin/zipkin**
Brave **https://github.com/openzipkin/brave**
Spring Sleuth **https://github.com/spring-cloud/spring-cloud-sleuth**
Dapper **http://research.google.com/pubs/pub36356.html**
Example Project **https://github.com/ahus1/logging-and-tracing**