Introduction to Distributed Tracing

In this lesson you will learn what is distributed tracing and why you should use it to diagnose failures and performance issues in your microservices based systems.

Why Distributed Tracing?

Let's go over a quick example that will help demonstrate what is distributed tracing and why you need it in your microservices architecture.

Let's say you have a player that, via one of the available clients, wants to perform a purchase of 5 potions. As you know, and setting the API Gateway aside for a moment, that will result in a call to the REST API of your Trading microservice.

We know from experience that normally Trading completes the purchase and notifies the result back to the client in about 2 seconds. However, starting early today, we have been getting complaints from many players claiming that it's taking up to 15 seconds for their purchases to complete, and, sometimes, the purchase does not even complete in the client, and instead just presents an error stating that the server took too much time to respond.

Now, we know that our purchase process involves more than just the Trading microservice. There are at least two other services that need to play their part in the corresponding Saga state machine. So we know that there must be some issue in one of these microservices, but where?

Luckily, we have already added logging all over the place, so we can query our logs and get a list of the most recent logged events. To figure out what is causing the overall slowliness, we'll need to filter the logs so we can isolate those that correspond to one specific purchase. Our correlation ids should help us here.

Hopefully, by looking closely at the timestamps of the filtered events, we should be able to tell at which point the next operation took too much time. And, when we know that, we can start investigating more about possible issues in the corresponding microservice.

Now, the main problem with this approach is that it relies on all microservices to implement a custom correlation id in all their relevant operations. If any microservice misses this, then you won't get a complete picture.

On the other hand, you need to manually calculate the duration of operations, or spot issues in them, based on what you can find in logs, and this can be very time consuming considering that you will have a massive number of logs as your system scales.

This is why you need distributed tracing in your system.

What is Distributed Tracing?

So, what is distributed tracing? Well, let's go through our purchase example once again, but this time we will include a new very important detail, which is called the Trace Id.

The trace id is part of what we call the trace context and uniquely identifies a distributed trace across the system. The trace context includes additional details of each operation performed by each microservice, like a span id to uniquely identify the operation, an operation name, the start time and duration of the operation.

And the great thing is that the trace context is automatically created and propagated both by ASP.NET Core and messaging libraries like MassTransit.

Thanks to this, when Trading sends a message to Inventory, the trace id is propagated to that operation and a trace context is generated for it. Same thing when Inventory sends a message back to Trading and also when messages are exchanged between Trading and Identity.

And of course, the trace context would also be propagated by direct HTTP calls between microservices.

The trace id will get automatically included in the events you log during your application lifetime, regardless of if you remember to capture it or not, which is already fantastic for keeping track of operations that belong together.

But, even better, you can use tools like OpenTelemetry to collect and export this tracing information to a growing ecosystem of tools that can help you visualize the sequence of operations across microservices. For instance, you can see here how things would look like for this purchase flow in Jaeger, a popular distributed tracing visualization tool.

All you need to do is enter the trace id, and Jaeger will present this detailed graph of everything that happened across the purchase flow, including how much time each step took and any errors that might have appeared along the way.

In summary, distributed tracing is a diagnostic technique that helps engineers localize failures and performance issues within applications, especially those distributed across multiple microservices.

In the next lesson you will update your code to start collecting distributed tracing information.