

OPENTELEMETRY

Let's start!

- Let's start by an example when would OpenTelemetry be beneficial
- You got an alert (based on logs) that service-a isn't able to write to db-1

Let's understand this with a real-life example.

Suppose we get an alert, based on our logs, that "SERVICE_A " isn't available to write to "db-1".

We're even at a data loss situation. This is something that we need to put our mind into we need to fix it we need to act fast.

- ➔ Downtimes, bugs and critical issues are bound to happen, the question is how fast we can fix it and how accurate are we in our fix.

Let's brainstorm what we can understand from an alert from your log solution saying **service_a** can't write to **db1**

1st assumption

I know service a is sending some exceptions, I can locate this exception in my code and then I can track the flow in my flow in my code that led to this exception maybe it will give me some hints,

2nd assumption

Maybe this service was recently deployed and that's the issue,

3rd assumption

Maybe there are some database metrics that we want to check out or maybe that's the reason db1 is not available at all

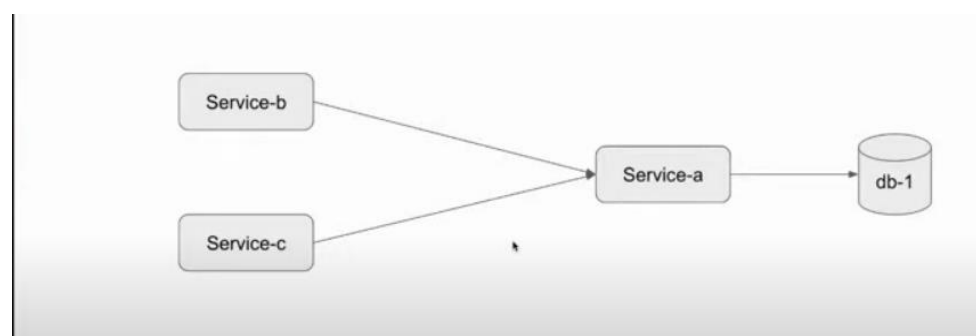
We can sum out plethora of assumptions on what's the error.

Service-a can't write to db-1

- Error logs show exception
- Metric show high CPU in db1
- Maybe an increase in traffic?
 - Which HTTP Route is causing query to db-1?
 - Other types of communication?

- ➔ The logs show an exception you follow the law the code and we don't find anything too interesting,
- ➔ When we look at the metrics of db1 we do see high CPU which is interesting as it's something new that that has changed and here a fair question arises that maybe we are just having more traffic, which is a good thing,
- ➔ More traffic is creating a problem though but it's a good thing. So maybe we need to look at traffic and then it's to ask ourselves which http router causing to service a to write in db1.

LET'S UNDERSTAND OVERALL SERVICE ARCHITECHURE.



We have here service a and service a is writing to db1 and if we need to know who is calling service a whether it's the front end or some mobile device or other services it's hard for us to know.

Thinking about metrics and logs it's even more complicated to know who called service-a and there are tools that may help us, some tools that can't help us but let's just say that we know from the top of our head again that both **service b** and **service c** are calling **service a**.

Now the question arises which one of them is calling service-a that is producing the error that we're experiencing, is it service-c or service-b or maybe it's just both.

Thinking about logs and metrics is kind of hard to answer this question.

Let's unveil the third and important part of open telemetry which is a **trace**.

LOGS: A recording of an event.

Eg: There's an error can't write to db-1

METRICS: A measurement recorded at runtime.

Eg: High CPU utilisation.

TRACES: The Path of a request through your application.

Eg: the context, the "path" within the system.

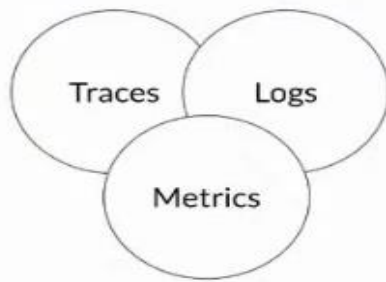
It's going to tell us the path that a specific API called specific backend interruption, what path it took, which services it visited before it hit that that error.

In the same way we would try to look at the call stack and see that a specific invocation jumps from function to function, we want to know from each service to service.

once we understand and we can visualize, we can see that maybe all the issues that we had are related to service b and this is a very important piece of information for us to resolve the issue. Now we can resolve it fast because now we know that we need to look whether the increase in traffic is related to service b or if it's specifically service b then maybe deployment in service b is the one that caused issues in service a.

So, to resolve this issue fast we need to have all three of logs, metrics and traces.

OpenTelemetry three pillars



Trace can have logs inside it

Logs can point to trace

Metrics be correlated via time to both.

LOGS: A recording of an event.

Eg: There's an error can't write to db-1

METRICS: A measurement recorded at runtime.

Eg: High CPU utilisation.

TRACES: The Path of a request through your application.

Eg: The context of why things are happening.

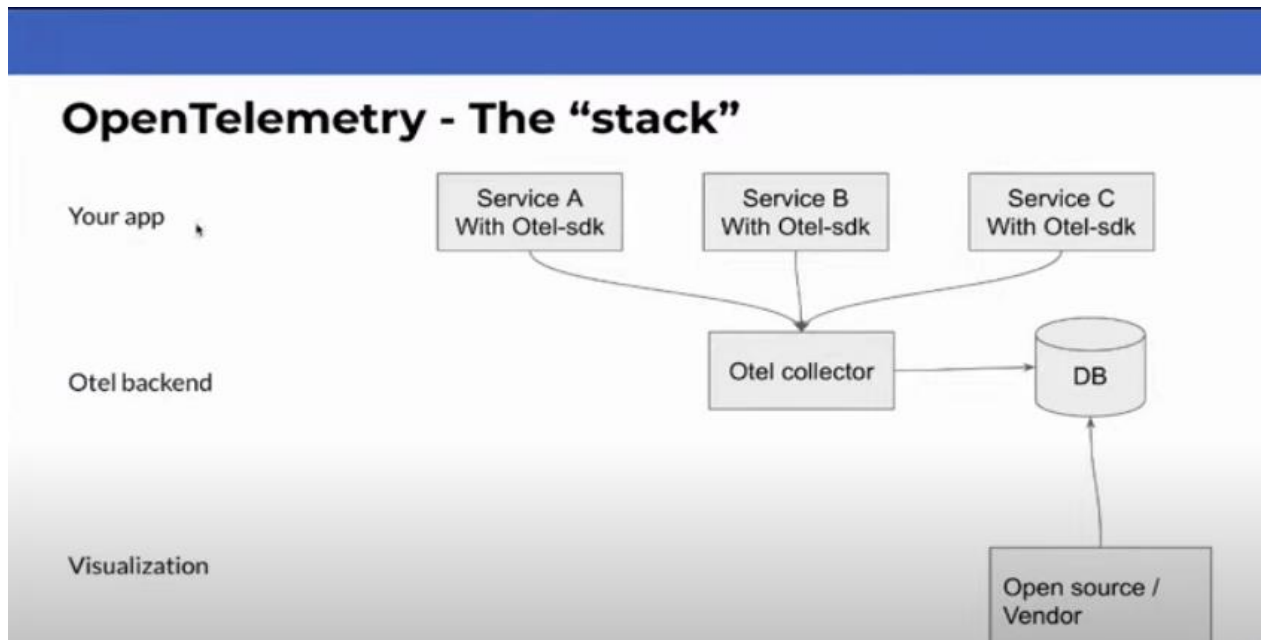
WHAT IS OPENTELEMETRY?

Open Telemetry is an Observability framework and toolkit designed to create and manage telemetry data such as traces, metrics, and logs. Crucially, Open Telemetry is vendor- and tool-agnostic, meaning that it can be used with a broad variety of Observability backends, including open-source tools like Jaeger and Prometheus, as well as commercial offerings.

Open Telemetry is not an observability backend like Jaeger, Prometheus, or other commercial vendors. Open Telemetry is focused on the generation, collection, management, and export of telemetry. A major goal of Open Telemetry is that you can easily instrument your applications or systems, no matter their language, infrastructure, or runtime environment. Crucially, the storage and visualization of telemetry is intentionally left to other tools

Open Telemetry is the ability to collect all three pillars under one unified SDK under CNCF with one specification, implementation for every programming language.

An SDK that collects the three pillars, it will ship to a backend, then to a DB. We will need visualisation layer to display traces, logs and metrics.



WHAT ARE MICROSERVICES?

Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are:

- Independently deployable
- Loosely coupled

Services are typically organized around business capabilities. Each service is often owned by a single, small team.

Here Services A, B, and C are microservices.

Open Telemetry also known as Otel.

- ➔ SDK : collects traces, logs and metrics and export them,
- ➔ Collector: receives telemetry and process it and then export it,
- ➔ A Database to store data,
- ➔ Visualization Layer: where the data will be sent.

We can reduce and increase the number of layers.

OpenTelemetry - How does the SDK works??



To have a preview of a trace we need to have a parent and child relation.

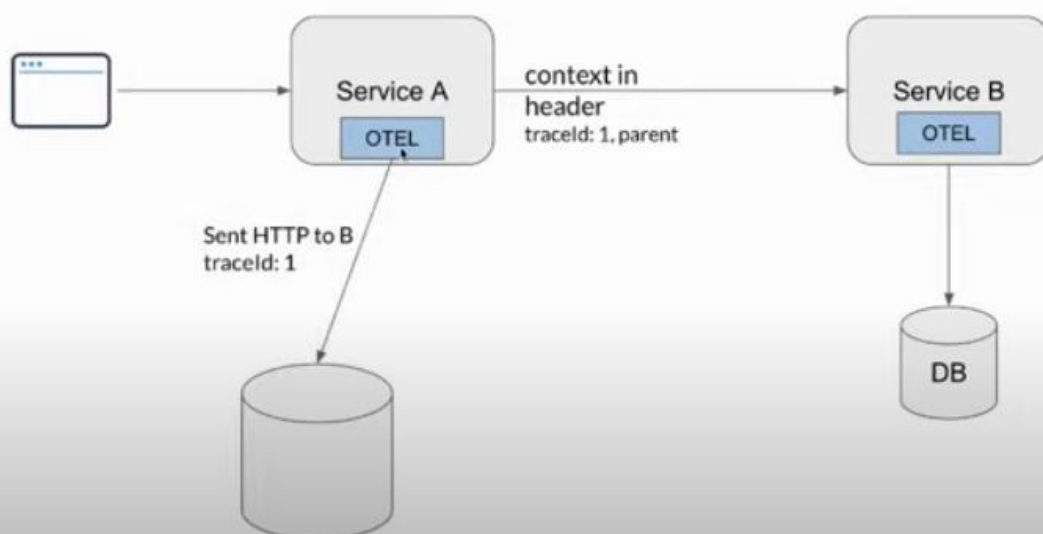
That means that any data that would be sent from service B it needs to be under the context as child of what happened in service A.

For that reason, when service a is sending an API call to service b, it's going to inject some headers in the http headers and that would help us to understand

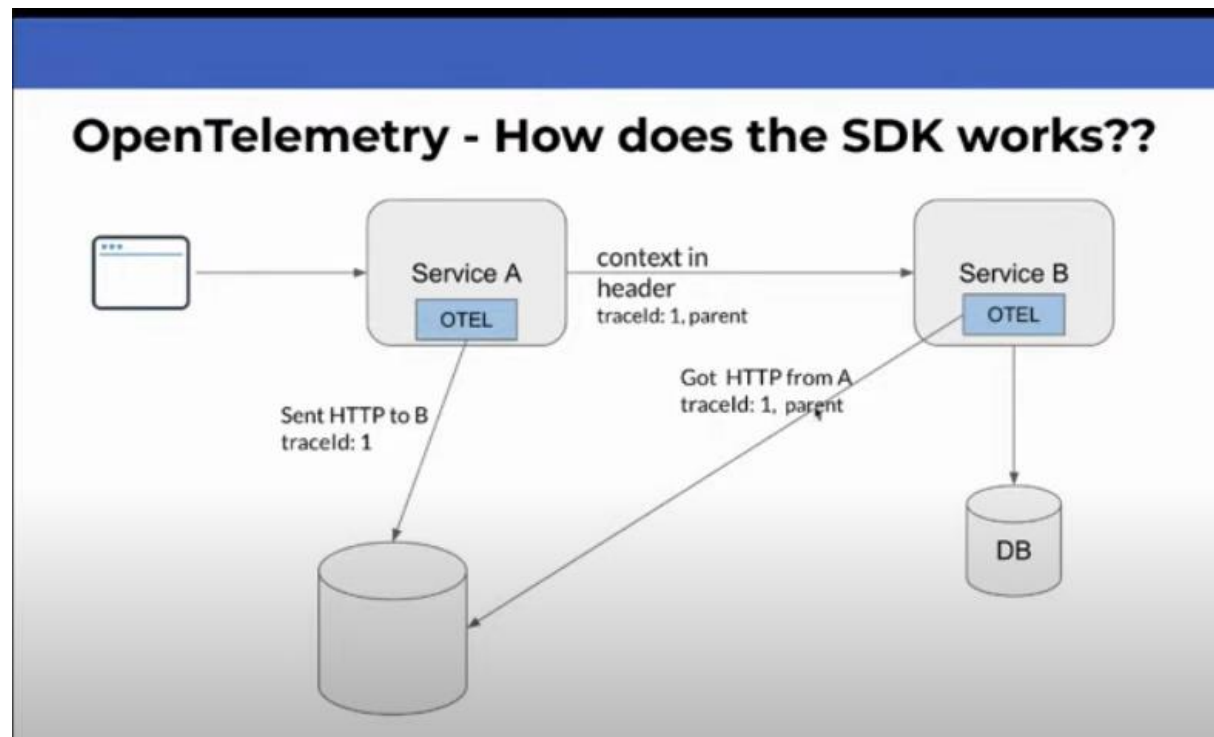
- ➔ Who is the parent,
- ➔ What is the cause,
- ➔ What led up to this API call

So, we would know that Service B is the child of Service A in this case.

OpenTelemetry - How does the SDK works??

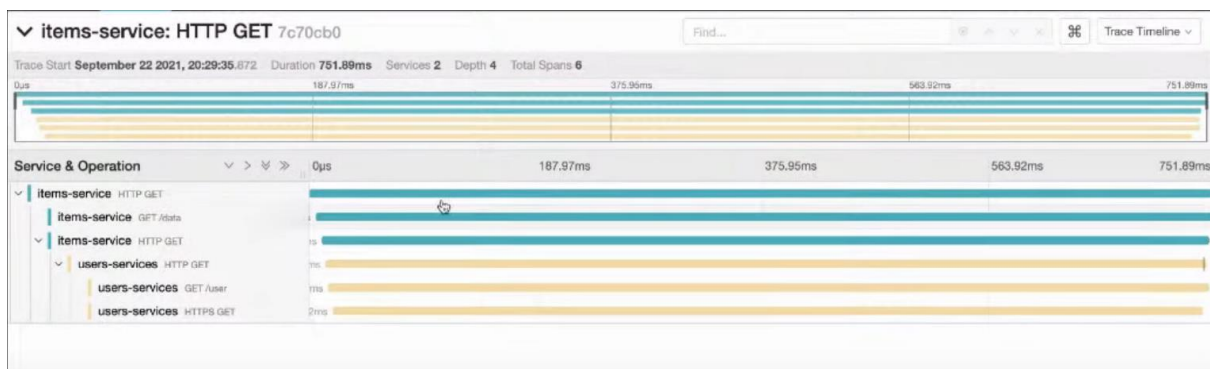
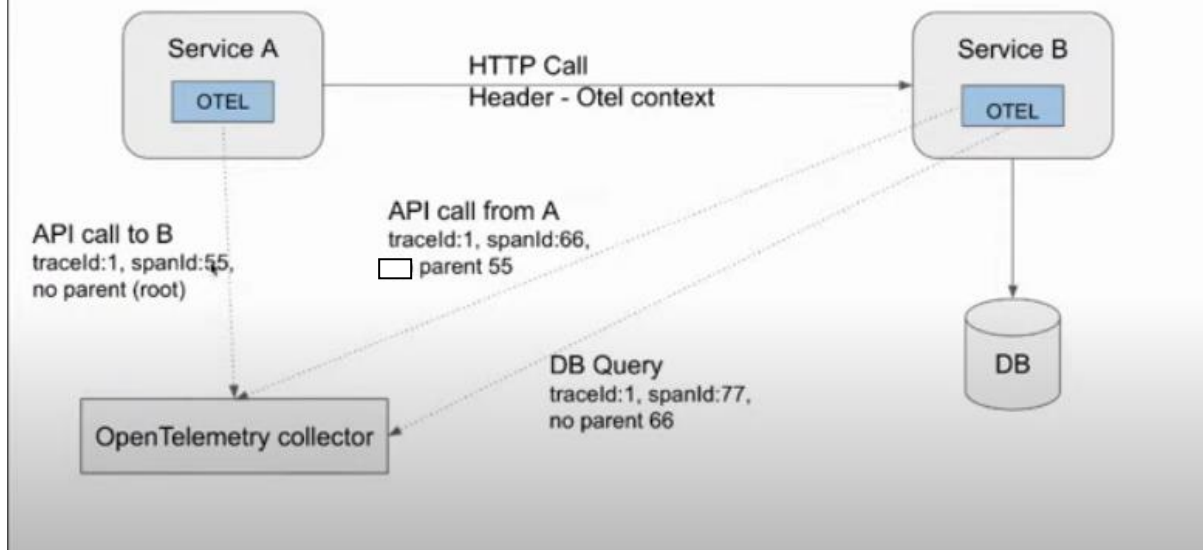


Now, the OTel is going to send data to our collector and going to say hey collector, “please you should know that we sent an API call to service b and this is all done under **trace ID number 1** because we want to know how to find this trace later on.



Then service B is going to say that, “I got a an API call from service A and now I do have a parent, so now any data sent from Service B”

OpenTelemetry - How does the SDK works??



Visualising the trace using JAEGER-UI

```
src > TS items-service.ts > app.use() callback
1  import init from './tracer';
2  const { meter } = init('items-service', 8081);
3
```

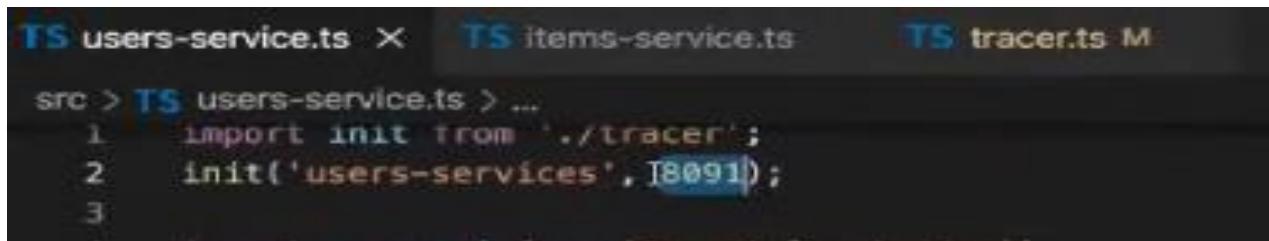
Importing Otel file (tracer.ts)

LET'S DEEP DIVE IN TRACER.TS FILE

```
// Define metrics
const metricExporter = new PrometheusExporter({ port: metricPort }, () => {
  console.log(`scrape: http://localhost:${metricPort}${PrometheusExporter.DEFAULT_OPTIONS.endpoint}`);
});
const meter = new MeterProvider({ exporter: metricExporter, interval: 1000 }).getMeter(serviceName);
```

Metrics Configuration for our MOCK.IO

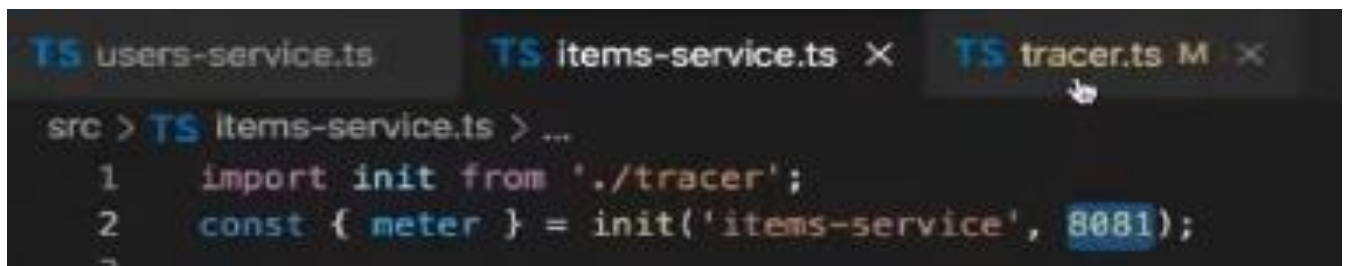
When we configure our metrics we need to send our metrics somewhere and here Prometheus exporter is used.



```
src > TS users-service.ts > ...
1  import init from './tracer';
2  init('users-services', 8091);
3
```

In my user-services file, setting metric port to 8091.

Similarly, item-service will have a different port.



```
src > TS items-service.ts > ...
1  import init from './tracer';
2  const { meter } = init('items-service', 8081);
3
```

Here, we are using port 8081.

Prometheus scrapes the data from service, so we need to expose some API call, some API endpoint that Prometheus can fetch the data from to show us how it looks

```
// Define traces
const traceExporter = new JaegerExporter({ endpoint: 'http://localhost:14268/api/traces' })
const provider = new NodeTracerProvider({
  resource: new Resource({
    [SemanticResourceAttributes.SERVICE_NAME]: 'serviceName',
  })
});
```

Defining TRACES in the Jaeger Exporter.

```
provider.addSpanProcessor(new SimpleSpanProcessor(traceExporter))
provider.register();
registerInstrumentations({
  instrumentations: [
    new ExpressInstrumentation(),
    new HttpInstrumentation()
  ]
});
```

we are asking to instrument http and express for us to close see the data that is incoming to express via the rest API and the outgoing API calls that are being sent from item service to user service and from user service to the MOCK.

We're looking at the code that is being executed so we are in the /get it's not failing

We are sending a response, and that's it

We barely don't see any open telemetry data here and this is because everything works fine and we don't need to do anything special, so the trace would look as follows:



But when things are failing and we're having an exception this is where things are starting to get a bit more complicated.

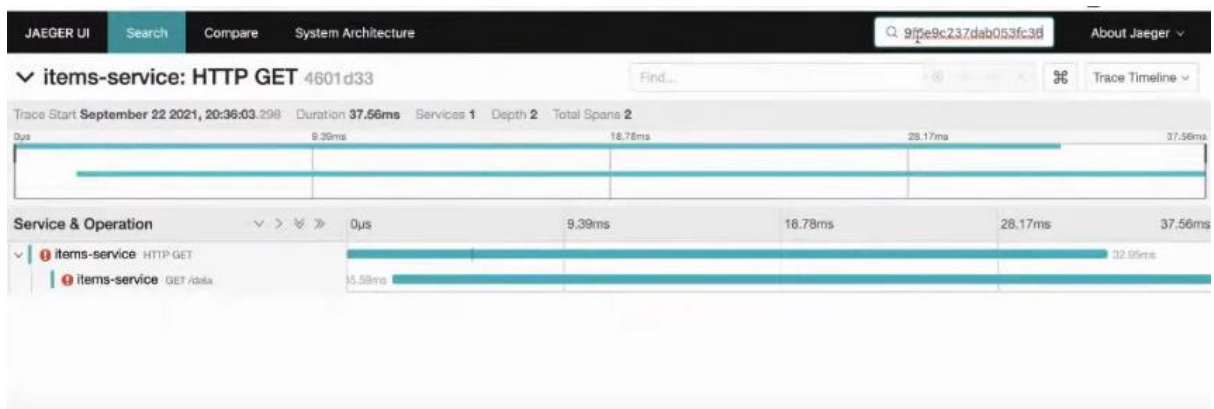
```

Loading data... | Docs
app.get('/data', async (request, response) => {
  try {
    if(request.query['fail']){
      throw new Error('A really bad error :/')
    }
    Loading data... | Docs
    const user = await axios.get('http://localhost:8090/user');
    response.json(user.data);
  } catch (e) {
    const activeSpan = api.trace.getSpan(api.context.active());
    console.error('Critical error', { traceId: activeSpan.spanContext().traceId});
    activeSpan.recordException(e);
    response.sendStatus(500);
  }
})

```

So, we wanted to be able to correlate between our logs and our traces.

For every log, a trace ID is attached.



By creating that exception, lets suppose if we have any failure or error, we can check just by the trace ID.

We can write a log within our trace.

