OPENTELEMETRY – SDK FULL BREAKDOWN

What is the purpose of Open Telemetry SDK?

- ➔ Collect data about the application,
- ➔ Propagate the context between services,
- ➔ Ship it somewhere.

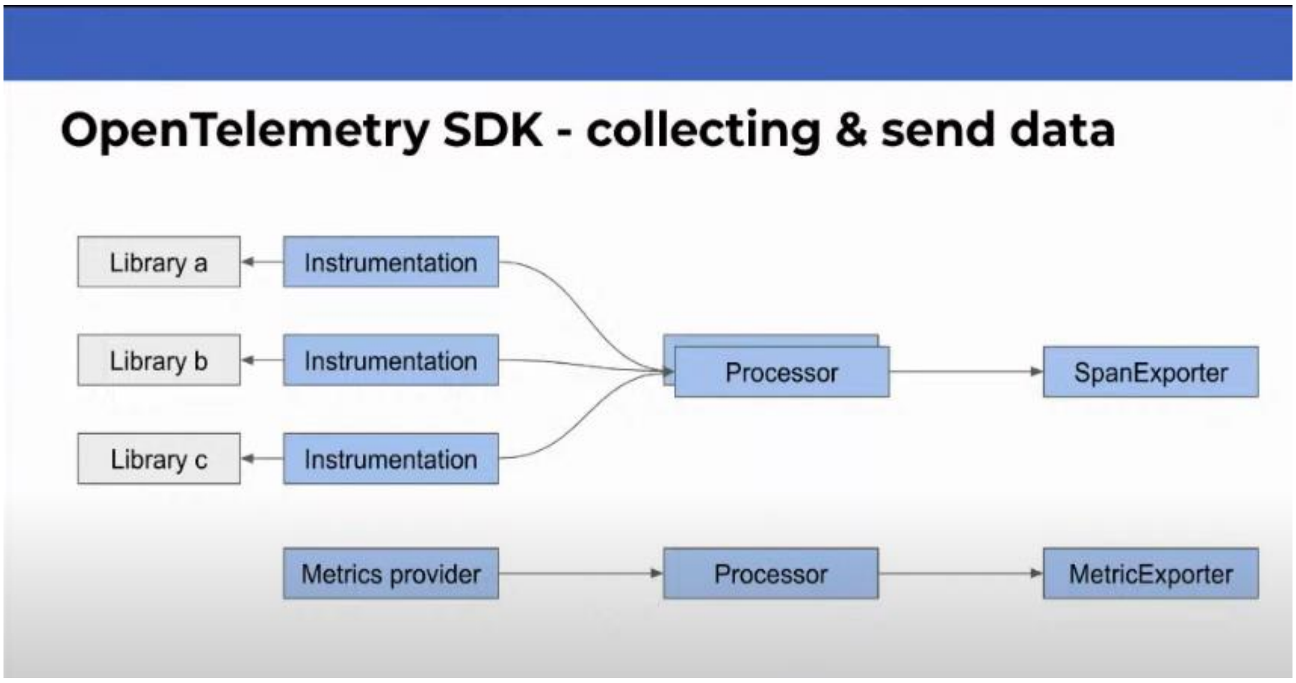
By shipping, it means ship it to a backend part to store it. So, we can process it, visualize it, search on it, etc.

Whenever we talk about Open Telemetry, most likely we are talking about distributed application.

What is a DISTRIBUTED APPLICATION?

Distributed computing is a field of computer science that studies distributed systems, defined as computer systems whose inter-communicating components are located on different networked computers.

Eg: Web Browsers



OpenTelemetry SDK - collecting & send data

In this diagram we can see that we have the libraries.

Those are the libraries that we're working with in your code base.

This could be our http library, our rest library our database library or any library and each one of these libraries in runtime are doing some actions so, the libraries are invoking some method calls and with that sending an API call without sending a database query.

INSTRUMENTATION:

Instrumentation is something that attaches to a library and in some way is going to collect data in runtime about what those libraries are doing.

This can be an AUTO INSTRUMENTATION, where it's automatically working with this library, and we see there is something else called MANUAL INSTRUMENTATION that in case you want to collect data outside of the library.

This is the step 1 and that's collecting the data.

The other end of that is to export data, to send it to a vendor or to send it to a database or to send it to some visualization tool

So that would take the data generated by the instrumentation and send it.

PROCESSOR

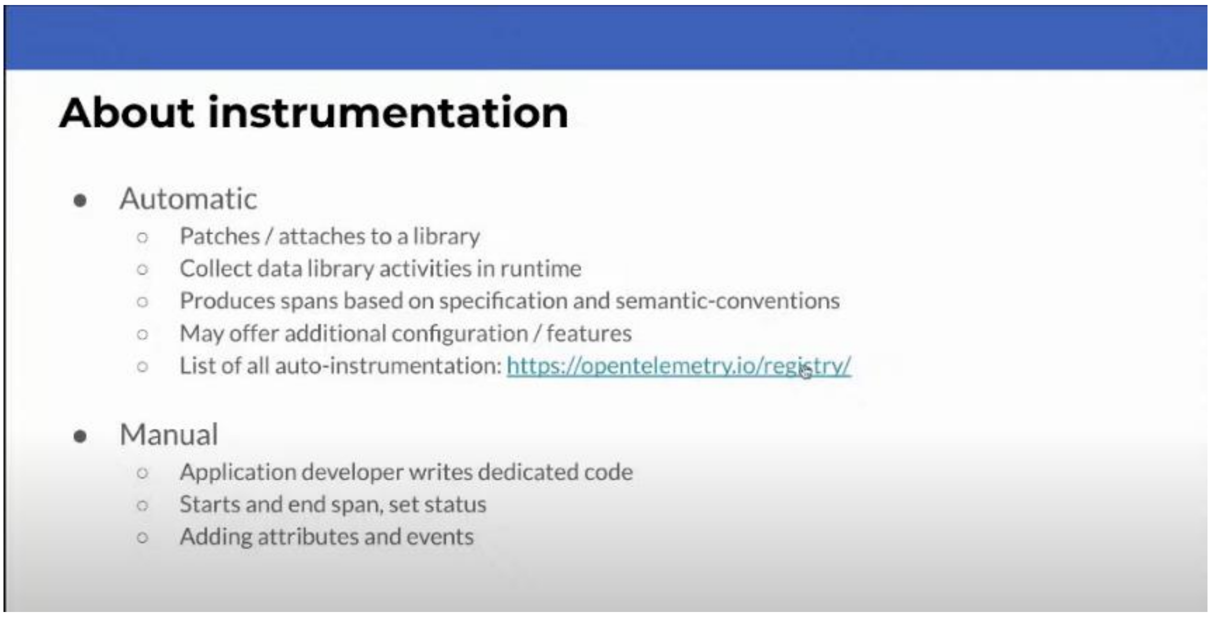
In between we have a processor.

Processor is kind of a data pipeline; it's going to get the data from instrumentation process it and then export it.

ABOUT INSTRUMENTATION:

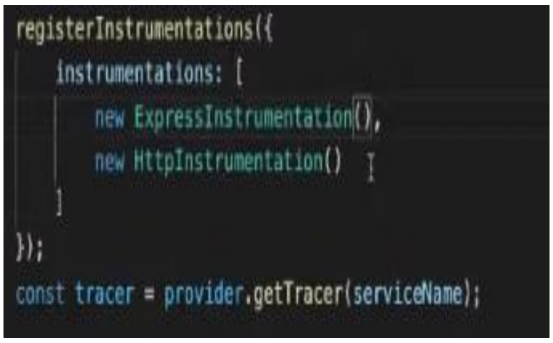
There are 2 types of instrumentation

- ➔ AUTOMATIC INSTRUMENTATION,
- ➔ MANUAL INSTRUMENTATION



About instrumentation

- Automatic
  - Patches / attaches to a library
  - Collect data library activities in runtime
  - Produces spans based on specification and semantic-conventions
  - May offer additional configuration / features
  - List of all auto-instrumentation: https://opentelemetry.io/registry/
- Manual
  - Application developer writes dedicated code
  - Starts and end span, set status
  - Adding attributes and events



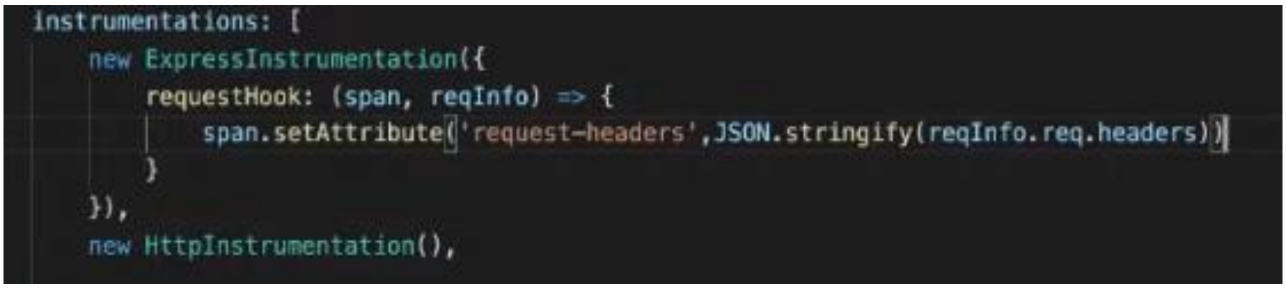
```
registerInstrumentations({
    instrumentations: [
        new ExpressInstrumentation(),
        new HttpInstrumentation()
    ]
});
const tracer = provider.getTracer(serviceName);
```

In this block of code, we created 2 instrumentations.

ExpressInstrumentaion :a restful service,

HttpInstrumentation: the http model responsible on incoming messages and outgoing,

When I want to add a instrument, I simply have to go to registry, select the language and we have ourselves an instrumentation



```
instrumentations: [
    new ExpressInstrumentation({
        requestHook: (span, reqInfo) => {
            span.setAttribute('request-headers',JSON.stringify(reqInfo.req.headers))
        }
    }),
    new HttpInstrumentation(),
```

We created a request hook in our instrumentation, anytime if we miss something we might want to check for instrumentation responsible for that and see if there is already a solution.

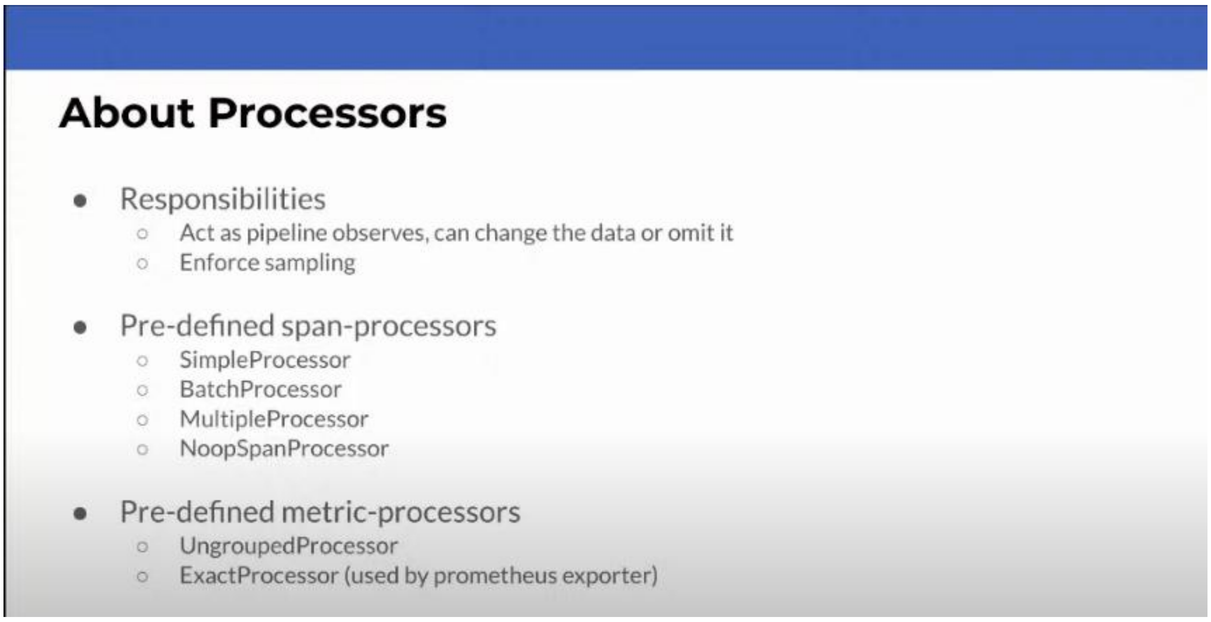
THESE ARE AUTO INSTRUMENTATIONS

We can create Manual Instrument as well.

Use cases of Manual Instrument:

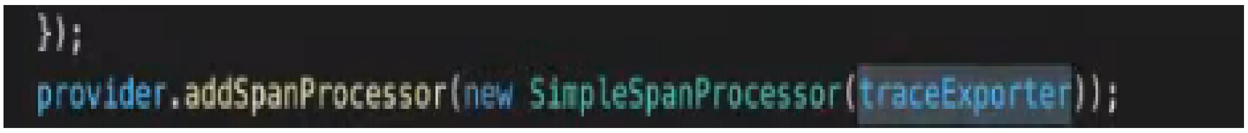
- ➔ INTERNAL ACTIVITES, such as creating timers,
- ➔ Adding Additional data, which is not present,
- ➔ If Auto Instrumentation is not supported, like Cassandra databse in NodeJS.

PROCESSORS



About Processors

- Responsibilities
  - Act as pipeline observes, can change the data or omit it
  - Enforce sampling
- Pre-defined span-processors
  - SimpleProcessor
  - BatchProcessor
  - MultipleProcessor
  - NoopSpanProcessor
- Pre-defined metric-processors
  - UngroupedProcessor
  - ExactProcessor (used by prometheus exporter)

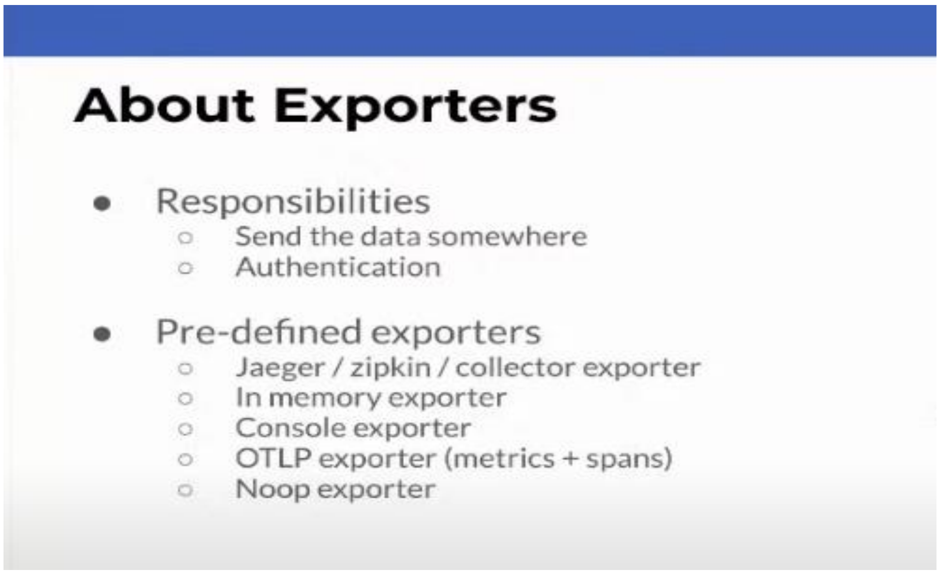


```
});
provider.addSpanProcessor(new SimpleSpanProcessor(traceExporter));
```

This is a simple Processor and its needs an exporter.

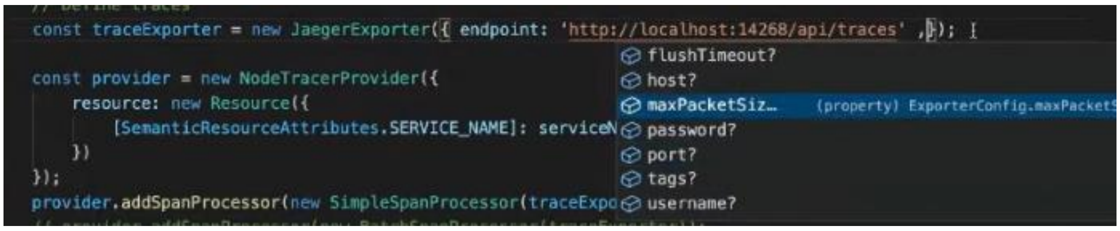
The Processor allows us to detach instrumentation and exporters because they are 2 unrelated things. Basically, takes the data from the instrumentation and pass it down to the exporter.

EXPORTERS



About Exporters

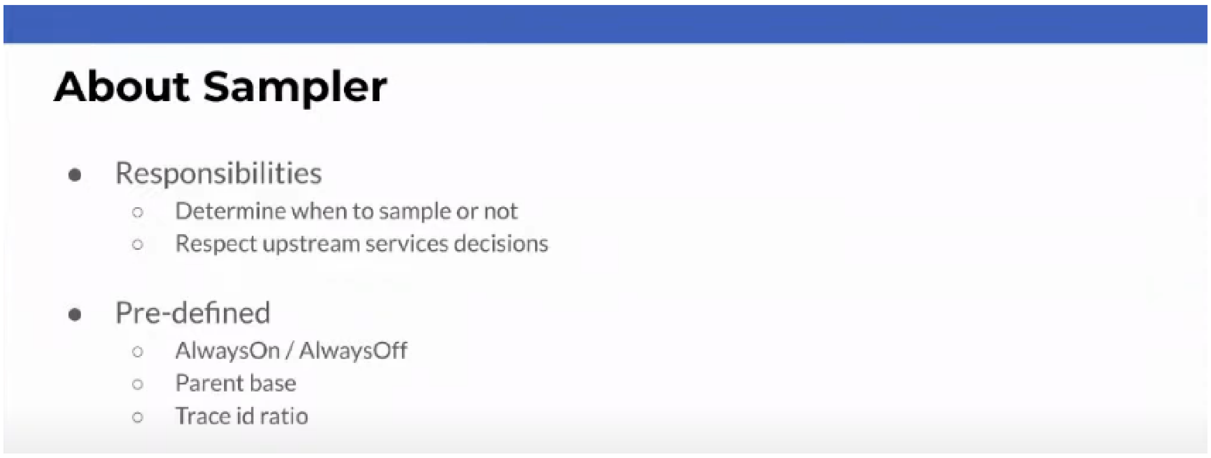
- Responsibilities
  - Send the data somewhere
  - Authentication
- Pre-defined exporters
  - Jaeger / zipkin / collector exporter
  - In memory exporter
  - Console exporter
  - OTLP exporter (metrics + spans)
  - Noop exporter



```
const traceExporter = new JaegerExporter({ endpoint: 'http://localhost:14268/api/traces' ,}); I

const provider = new NodeTracerProvider({
    resource: new Resource({
        [SemanticResourceAttributes.SERVICE_NAME]: serviceN
    })
});
provider.addSpanProcessor(new SimpleSpanProcessor(traceExpo
```

Jaegar Exporter and what we can perform with it.

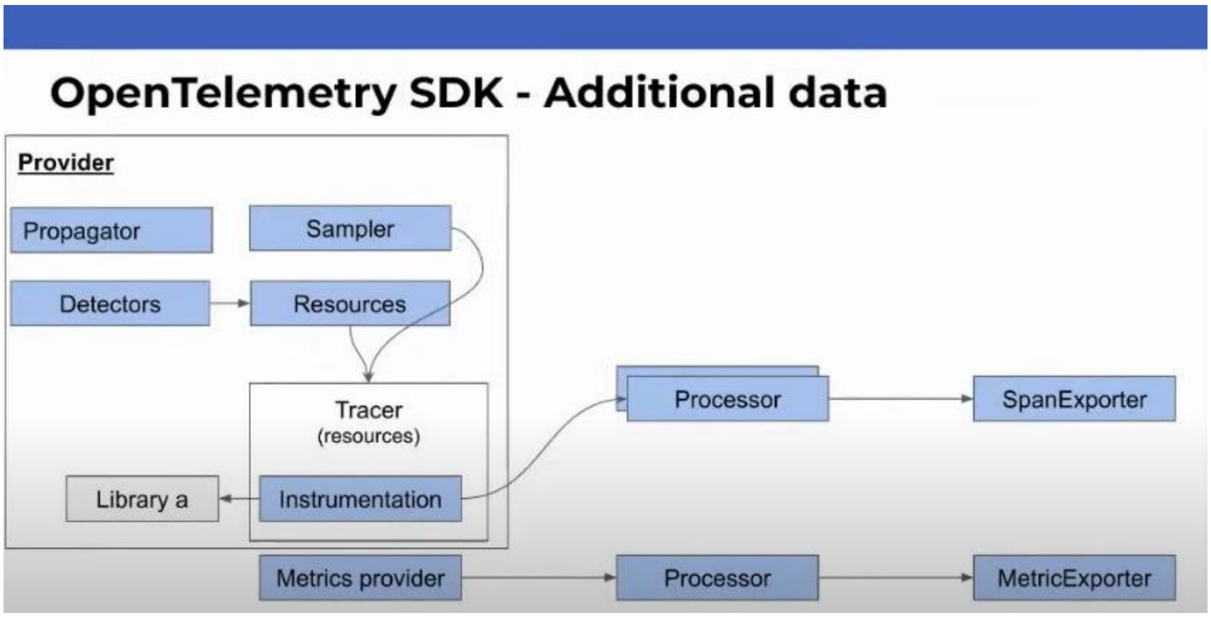


About Sampler

- Responsibilities
  - Determine when to sample or not
  - Respect upstream services decisions
- Pre-defined
  - AlwaysOn / AlwaysOff
  - Parent base
  - Trace id ratio

Responsibilities of Sampler

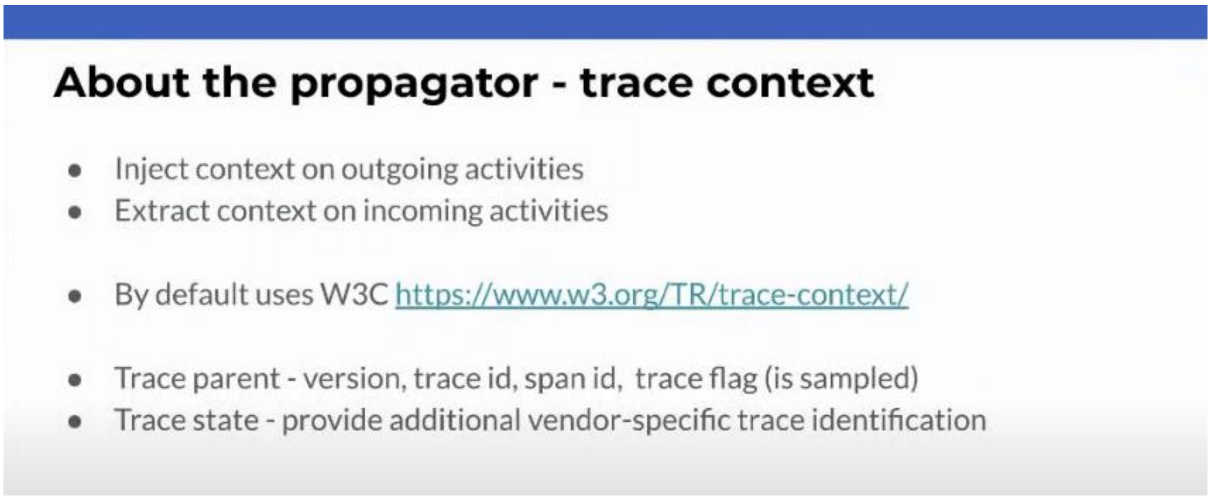
- ➔ Determine when to sample or not means whether to record data or not,
- ➔ Whether to respect upstream decisions or not.



OpenTelemetry SDK - Additional data

Till now we talked about how data is collected and how to ship data.

Let’s see how data gets propagated.



About the propagator - trace context

- Inject context on outgoing activities
- Extract context on incoming activities

- By default uses W3C https://www.w3.org/TR/trace-context/

- Trace parent - version, trace id, span id, trace flag (is sampled)
- Trace state - provide additional vendor-specific trace identification

Propagator injects the context on outgoing activities AND EXTRACT context on incoming activities.

2 headers that are always going to be injected are trace parent and trace state.

MANUAL INSTRUMENTATION

When we work with open telemetry we would work in a distributed system and when we work with distributed system, we may want to use with a messaging system and most messaging systems allow you to inject headers or metadata or attribute into messages like we have attributes in kafka, we have headers and then we can do an auto instrumentation because the instrumentation itself once it sends a message we can inject the headers and when we consume the message we can extract the messages, the headers.

However, some of those does not support it some of those do not allow you to have headers and this is a problem that you are going to have to solve us as a developer as a consumer of open telemetry and we would have to solve it ourselves because an auto instrumentation won't be able to solve it for us.