In this lab we will work with OTEL manual instrumentation. As opposed to the other labs where the agents did everything for us, we will see how to export traces, metrics and logs using OpenTelemetry manual instrumentation. This is an interesting case when we want to enhance visibility or create specific kinds of metrics. It is also good to learn this because not every language has an auto instrumentation agent.

For this this lab we will instrument a spring boot application. First, we will create traces, then metrics and finally logs.

The first step is to add the OTEL dependencies to the Project. Add this to your pom.xml file inside the *lab3\_manual\_instrumentation* folder.

<!-- OTEL DEP MGMT -->

<dependencyManagement>

<dependencies>

<dependency>

<groupId>io.opentelemetry</groupId>

<artifactId>opentelemetry-bom</artifactId>

<version>1.34.1</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<!-- OTEL BEGIN-->

<dependency>

<groupId>io.opentelemetry</groupId>

<artifactId>opentelemetry-api</artifactId>

</dependency>

<dependency>

<groupId>io.opentelemetry</groupId>

<artifactId>opentelemetry-api</artifactId>

</dependency>

<dependency>

<groupId>io.opentelemetry</groupId>

<artifactId>opentelemetry-sdk</artifactId>

</dependency>

<dependency>

<groupId>io.opentelemetry</groupId>

<artifactId>opentelemetry-sdk-metrics</artifactId>

</dependency>

<dependency>

<groupId>io.opentelemetry</groupId>

<artifactId>opentelemetry-exporter-logging</artifactId>

</dependency>

<dependency>

<!-- Not managed by opentelemetry-bom -->

<groupId>io.opentelemetry.semconv</groupId>

<artifactId>opentelemetry-semconv</artifactId>

<version>1.23.1-alpha</version>

</dependency>

<!-- OTEL auto config-->

<dependency>

<groupId>io.opentelemetry</groupId>

<artifactId>opentelemetry-sdk-extension-autoconfigure</artifactId>

</dependency>

<dependency>

<groupId>io.opentelemetry</groupId>

<artifactId>opentelemetry-sdk-extension-autoconfigure-spi</artifactId>

</dependency>

<!-- OTLP exporter-->

<dependency>

<groupId>io.opentelemetry</groupId>

<artifactId>opentelemetry-exporter-otlp</artifactId>

</dependency>

<!-- OTEL END-->

This will add all dependencies for now. To make things easier OTEL SDK provides a way to bootstrap and configure your application. Edit DigisicapisApplication.java and add this new bean.

@SpringBootApplication

public class DigisicapisApplication {

public static void main(String[] args) {

SpringApplication.run(DigisicapisApplication.class, args);

}

@Bean

public OpenTelemetry openTelemetry() {

return AutoConfiguredOpenTelemetrySdk.initialize().getOpenTelemetrySdk();

}

}

## Traces

Now we will change the code to create a trace and some spans. In the previous step we have initialized the SDK, hence we already have a TracerProvider. Now we will edit DemoPagoController.java and create a tracer object.

@RestController

@RequestMapping("/banking")

public class DemoPagoController {

private static final org.apache.logging.log4j.Logger log4jLogger = LogManager.getLogger("log4j-logger");

//OTEL BEGIN - Acquiring a Trace

private final Tracer tracer;

@Autowired

DemoPagoController(OpenTelemetry openTelemetry) {

tracer = openTelemetry.getTracer(DemoPagoController.class.getName(), "0.1.0");

}

//OTEL END

With the trace object in hand, we can start creating spans. Remember that a trace is a parent span with its child. Still in the DemoPagoController, edit the pago method to create your first span.

@RequestMapping(value = "/pago", method = RequestMethod.GET)

@ResponseBody

public String teste(@RequestParam(name="total",required = true) String totalPago,@RequestParam(name="customerId",required = true) String customerId){

//OTEL

Span span = tracer.spanBuilder("pago").startSpan();

try (Scope scope = span.makeCurrent()) {

System.out.println(totalPago);

log4jLogger.info("Receiving pago for:",customerId);

processPayment(totalPago,customerId);

} catch (Exception e){

span.recordException(e);

}finally{

span.end();

}

return "{'transactionId':'kadsbflajkhdfas','status':'OK'}";

}

We have create a parent span in the pago method. Now go down to the other methods that are called by processPayment and create new child spans.

private void updateCustomerBalance(String pagoTotal,String pagoCustomerId) throws InterruptedException{

Span childSpan = tracer.spanBuilder("updateCustomerBalance").startSpan();

log4jLogger.info("Update balance for:",pagoCustomerId);

System.out.println("updateCustomerBalance");

childSpan.end();

}

private void updateDB() throws InterruptedException{

Span childSpan = tracer.spanBuilder("updateDB").startSpan();

log4jLogger.info("Update BD");

System.out.println("updateDB");

childSpan.end();

}

We are ready for the first test. Let’s build the application and check it is working as expected.

|  |
| --- |
| mvn clean package |

Start the OTEL collector.

|  |
| --- |
| docker-compose up -d |

Start the application.

|  |
| --- |
| ./run.sh |

Put some load.

|  |
| --- |
| ./load.sh |

Check collector logs to find traces being logged.

A screenshot of a computer

Description automatically generated

## Metrics

Now we will create some custom metrics. The first step is to edit DemoPagoController and add the lines below to get a meter instance.

//OTEL BEGIN - Acquiring a Trace

private final Tracer tracer;

private final Meter meter;

@Autowired

DemoPagoController(OpenTelemetry openTelemetry) {

tracer = openTelemetry.getTracer(DemoPagoController.class.getName(), "0.1.0");

meter = openTelemetry.getMeter(DemoPagoController.class.getName());

}

//OTEL END

Now we will create the metrics. In the updateCustomerBalance method let’s a create a metric to count the payments and another to sum the payments.

private void updateCustomerBalance(String pagoTotal,String pagoCustomerId) throws InterruptedException{

Span childSpan = tracer.spanBuilder("updateCustomerBalance").startSpan();

log4jLogger.info("Update balance for:",pagoCustomerId);

System.out.println("updateCustomerBalance");

// Build counter e.g. LongCounter

LongCounter counter = meter

.counterBuilder("processed\_payments")

.setDescription("processed\_payments")

.setUnit("1")

.build();

// Record data

counter.add(1);

// Build counter e.g. LongCounter

LongCounter counter2 = meter

.counterBuilder("pago")

.setDescription("total pago")

.setUnit("1")

.build();

// Record data

counter2.add(Integer.parseInt(pagoTotal));

childSpan.end();

}

Now we build and run again.

|  |
| --- |
| mvn clean package |

Start the application.

|  |
| --- |
| ./run.sh |

Put some load.

|  |
| --- |
| ./load.sh |

Check collector logs to find metrics being logged.

A screenshot of a computer program

Description automatically generated

## Logs

Logs work in a different way. We will not create and publish logs. We will create a bridge between supported log frameworks and OTEL Collector. In our case we will bridge springboot logback.

First, we will need to update the pom.xml file with new dependencies.

<dependency>

<groupId>io.opentelemetry.instrumentation</groupId>

<artifactId>opentelemetry-logback-appender-1.0</artifactId>

<version>1.31.0-alpha</version>

</dependency>

<dependency>

<groupId>io.opentelemetry</groupId>

<artifactId>opentelemetry-exporter-otlp</artifactId>

</dependency>

Creat a logback-spring.xml file inside the resources folder

<?xml version="1.0" encoding="UTF-8"?>

<configuration>

<appender name="console" class="ch.qos.logback.core.ConsoleAppender">

<encoder>

<pattern>

%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n

</pattern>

</encoder>

</appender>

<appender name="OpenTelemetry"

class="io.opentelemetry.instrumentation.logback.appender.v1\_0.OpenTelemetryAppender">

<captureExperimentalAttributes>true</captureExperimentalAttributes>

<captureKeyValuePairAttributes>true</captureKeyValuePairAttributes>

<captureCodeAttributes>true</captureCodeAttributes>

<captureMarkerAttribute>true</captureMarkerAttribute>

<captureMdcAttributes>\*</captureMdcAttributes>

</appender>

<root level="INFO">

<appender-ref ref="console"/>

<appender-ref ref="OpenTelemetry"/>

</root>

</configuration>

With that we have configure the log appender to forward the logs to the OTEL collector. Now we need to do more changes in the code to finish this configuration.

Edit the DigisicapisApplication.java and add this code to instantiate the log appender.

@Bean

public OpenTelemetry openTelemetry() {

Object o = AutoConfiguredOpenTelemetrySdk.initialize().getOpenTelemetrySdk();

//install OTEL log appender

OpenTelemetryAppender.install((OpenTelemetrySdk) o);

return (OpenTelemetry) o;

}

@Bean

SdkLoggerProvider otelSdkLoggerProvider(final Environment environment, final ObjectProvider<LogRecordProcessor> logRecordProcessors) {

final String applicationName = environment.getProperty("spring.application.name", "application");

final Resource resource = Resource.create(Attributes.of(ResourceAttributes.SERVICE\_NAME, applicationName));

final SdkLoggerProviderBuilder builder = SdkLoggerProvider.builder()

.setResource(Resource.getDefault().merge(resource));

logRecordProcessors.orderedStream().forEach(builder::addLogRecordProcessor);

return builder.build();

}

@Bean

LogRecordProcessor otelLogRecordProcessor() {

return BatchLogRecordProcessor

.builder(

OtlpGrpcLogRecordExporter.builder()

.setEndpoint("http://localhost:4317")

.build())

.build();

}

With that last change we are good to go.

Now we build and run again.

|  |
| --- |
| mvn clean package |

Put some load

|  |
| --- |
| ./load.sh |

Check collector logs to find logs.

A screenshot of a computer

Description automatically generated

To close this lab, we will change the collector configuration to send the information to Cisco Observability Platform

Edit otel-collector-config.yaml and make it like the example below. You can also grab the config file from the previous lab.

receivers:

otlp:

protocols:

grpc:

http:

exporters:

logging:

verbosity: detailed

jaeger:

endpoint: jaeger:14250

tls:

insecure: true

otlphttp:

auth:

authenticator: oauth2client

traces\_endpoint: https://<tenant\_host>/data/v1/trace

logs\_endpoint: https://<tenant\_host>/data/v1/logs

processors:

batch: #### Optional for trace batching for AppDynamics Cloud

send\_batch\_max\_size: 1000

send\_batch\_size: 1000

timeout: 10s

extensions: #### Mandatory for AppDynamics Cloud

oauth2client:

client\_id: xxxx

client\_secret: xxxx

token\_url: https://tenant\_host>auth/xxxxx/default/oauth2/token

service:

extensions: #### Mandatory for AppD Cloud

- oauth2client

pipelines:

traces:

receivers: [otlp]

processors: [batch]

exporters: [logging,jaeger,otlphttp]

metrics:

receivers: [otlp]

exporters: [logging]

logs:

receivers: [otlp]

exporters: [logging,otlphttp]

Stop and Start the collector.

|  |
| --- |
| docker-compose down.  docker-compose up -d |

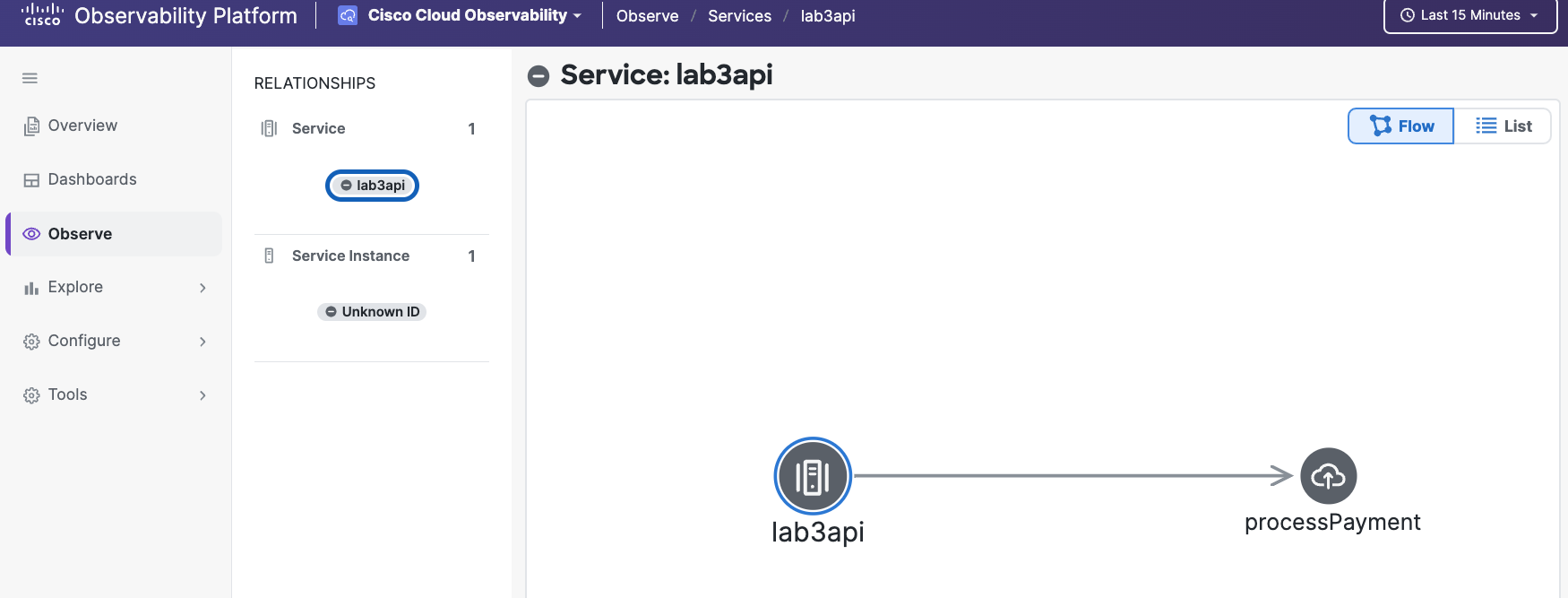
If not running, start your app and put some load.

|  |
| --- |
| ./load.sh |

Check collector logs.



Now check Cisco Observability Platform and AppDynamics to see the results.



A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated