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.

```java
@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.

```java
@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.

```java
    @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
```

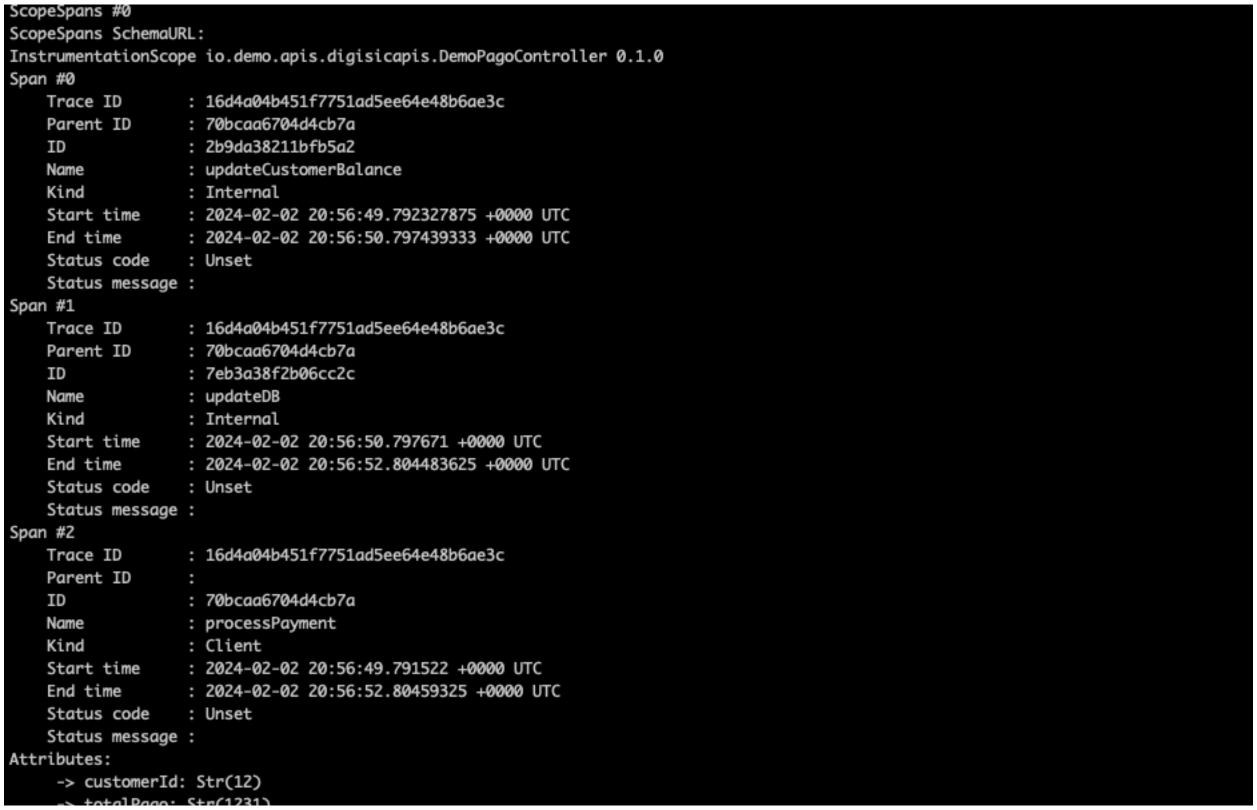
Start the application.

```
./run.sh
```

Put some load.

```
./load.sh
```

Check collector logs to find traces being logged.



```
ScopeSpans #0
ScopeSpans SchemaURL:
InstrumentationScope io.demo.apis.digisicapis.DemoPagoController 0.1.0
Span #0
    Trace ID       : 16d4a04b451f7751ad5ee64e48b6ae3c
    Parent ID      : 70bcaa6704d4cb7a
    ID             : 2b9da38211bfb5a2
    Name           : updateCustomerBalance
    Kind           : Internal
    Start time     : 2024-02-02 20:56:49.792327875 +0000 UTC
    End time       : 2024-02-02 20:56:50.797439333 +0000 UTC
    Status code    : Unset
    Status message :
Span #1
    Trace ID       : 16d4a04b451f7751ad5ee64e48b6ae3c
    Parent ID      : 70bcaa6704d4cb7a
    ID             : 7eb3a38f2b06cc2c
    Name           : updateDB
    Kind           : Internal
    Start time     : 2024-02-02 20:56:50.797671 +0000 UTC
    End time       : 2024-02-02 20:56:52.804483625 +0000 UTC
    Status code    : Unset
    Status message :
Span #2
    Trace ID       : 16d4a04b451f7751ad5ee64e48b6ae3c
    Parent ID      :
    ID             : 70bcaa6704d4cb7a
    Name           : processPayment
    Kind           : Client
    Start time     : 2024-02-02 20:56:49.791522 +0000 UTC
    End time       : 2024-02-02 20:56:52.80459325 +0000 UTC
    Status code    : Unset
    Status message :
Attributes:
     -> customerId: Str(12)
     -> totalPago: Str(1231)
```

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.

```java
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
```

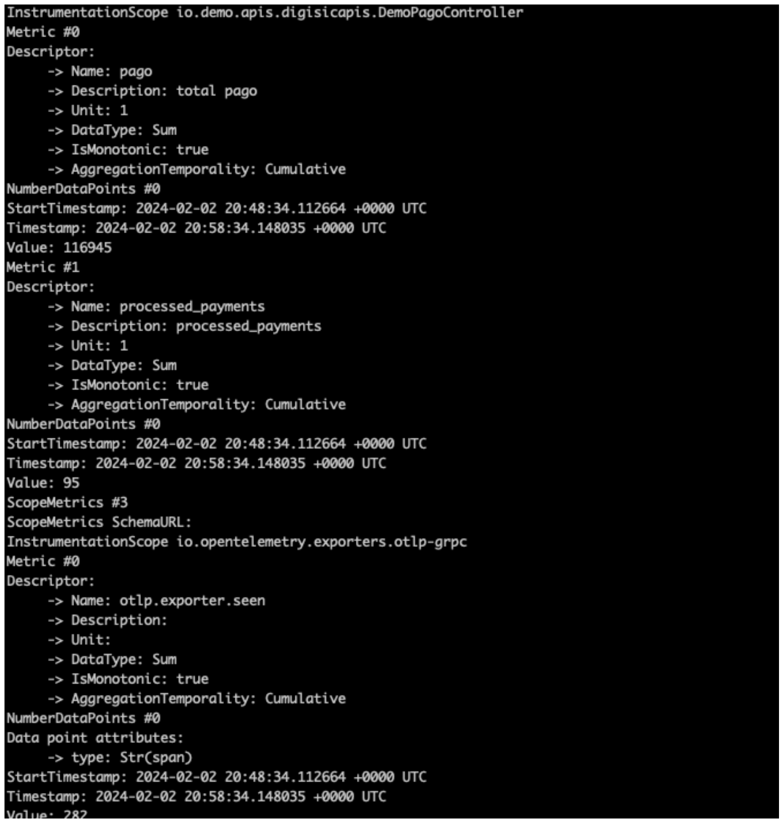
Start the application.

```
./run.sh
```

Put some load.

```
./load.sh
```

Check collector logs to find metrics being logged.



```
InstrumentationScope io.demo.apis.digisicapis.DemoPagoController
Metric #0
Descriptor:
     -> Name: pago
     -> Description: total pago
     -> Unit: 1
     -> DataType: Sum
     -> IsMonotonic: true
     -> AggregationTemporality: Cumulative
NumberDataPoints #0
StartTimestamp: 2024-02-02 20:48:34.112664 +0000 UTC
Timestamp: 2024-02-02 20:58:34.148035 +0000 UTC
Value: 116945
Metric #1
Descriptor:
     -> Name: processed_payments
     -> Description: processed_payments
     -> Unit: 1
     -> DataType: Sum
     -> IsMonotonic: true
     -> AggregationTemporality: Cumulative
NumberDataPoints #0
StartTimestamp: 2024-02-02 20:48:34.112664 +0000 UTC
Timestamp: 2024-02-02 20:58:34.148035 +0000 UTC
Value: 95
ScopeMetrics #3
ScopeMetrics SchemaURL:
InstrumentationScope io.opentelemetry.exporters.otlp-grpc
Metric #0
Descriptor:
     -> Name: otlp.exporter.seen
     -> Description:
     -> Unit:
     -> DataType: Sum
     -> IsMonotonic: true
     -> AggregationTemporality: Cumulative
NumberDataPoints #0
Data point attributes:
     -> type: Str(span)
StartTimestamp: 2024-02-02 20:48:34.112664 +0000 UTC
Timestamp: 2024-02-02 20:58:34.148035 +0000 UTC
Value: 282
```

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.

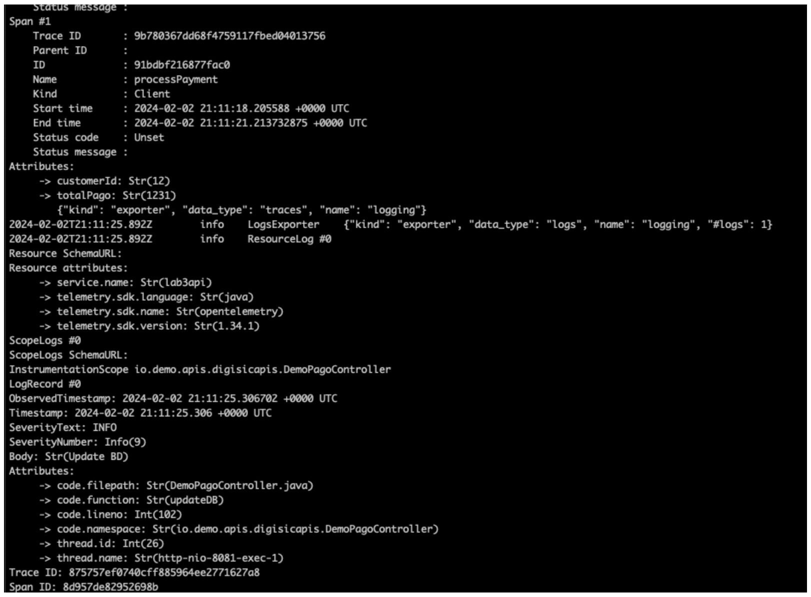
Now we build and run again.

```
mvn clean package
```

Put some load

```
./load.sh
```

Check collector logs to find logs.



```
    Status message :
Span #1
    Trace ID       : 9b780367dd68f4759117fbed04013756
    Parent ID      :
    ID             : 91bdbf216877fac0
    Name           : processPayment
    Kind           : Client
    Start time     : 2024-02-02 21:11:18.205588 +0000 UTC
    End time       : 2024-02-02 21:11:21.213732875 +0000 UTC
    Status code    : Unset
    Status message :
Attributes:
     -> customerId: Str(12)
     -> totalPago: Str(1231)
        {"kind": "exporter", "data_type": "traces", "name": "logging"}
2024-02-02T21:11:25.892Z        info    LogsExporter    {"kind": "exporter", "data_type": "logs", "name": "logging", "#logs": 1}
2024-02-02T21:11:25.892Z        info    ResourceLog #0
Resource SchemaURL:
Resource attributes:
     -> service.name: Str(lab3api)
     -> telemetry.sdk.language: Str(java)
     -> telemetry.sdk.name: Str(opentelemetry)
     -> telemetry.sdk.version: Str(1.34.1)
ScopeLogs #0
ScopeLogs SchemaURL:
InstrumentationScope io.demo.apis.digisicapis.DemoPagoController
LogRecord #0
ObservedTimestamp: 2024-02-02 21:11:25.306702 +0000 UTC
Timestamp: 2024-02-02 21:11:25.306 +0000 UTC
SeverityText: INFO
SeverityNumber: Info(9)
Body: Str(Update BD)
Attributes:
     -> code.filepath: Str(DemoPagoController.java)
     -> code.function: Str(updateDB)
     -> code.lineno: Int(102)
     -> code.namespace: Str(io.demo.apis.digisicapis.DemoPagoController)
     -> thread.id: Int(26)
     -> thread.name: Str(http-nio-8081-exec-1)
Trace ID: 875757ef0740cff885964ee2771627a8
Span ID: 8d957de82952698b
```

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.

```yaml
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]
```

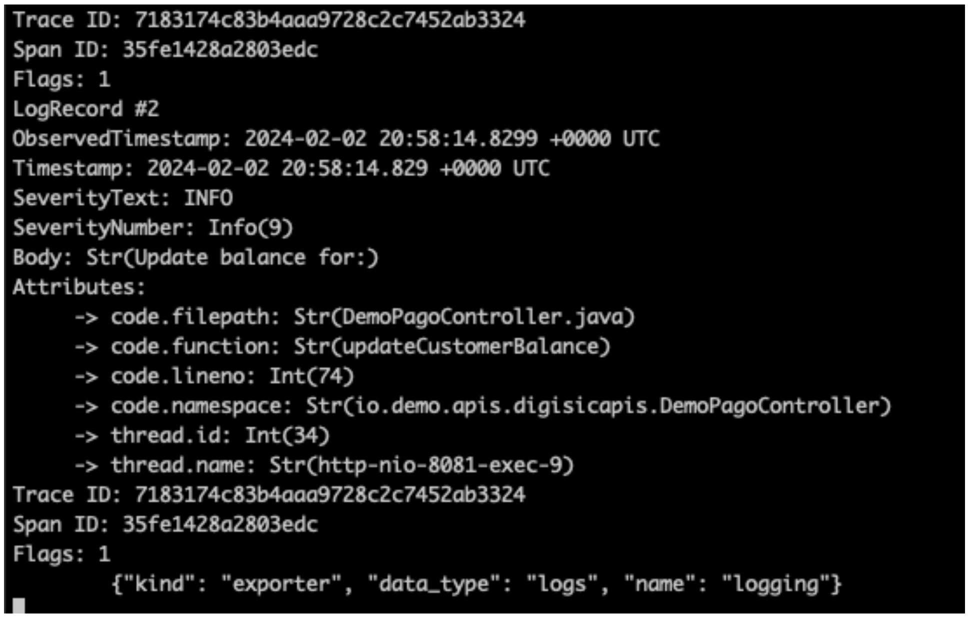
Stop and Start the collector.

```
docker-compose down.
docker-compose up -d
```

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

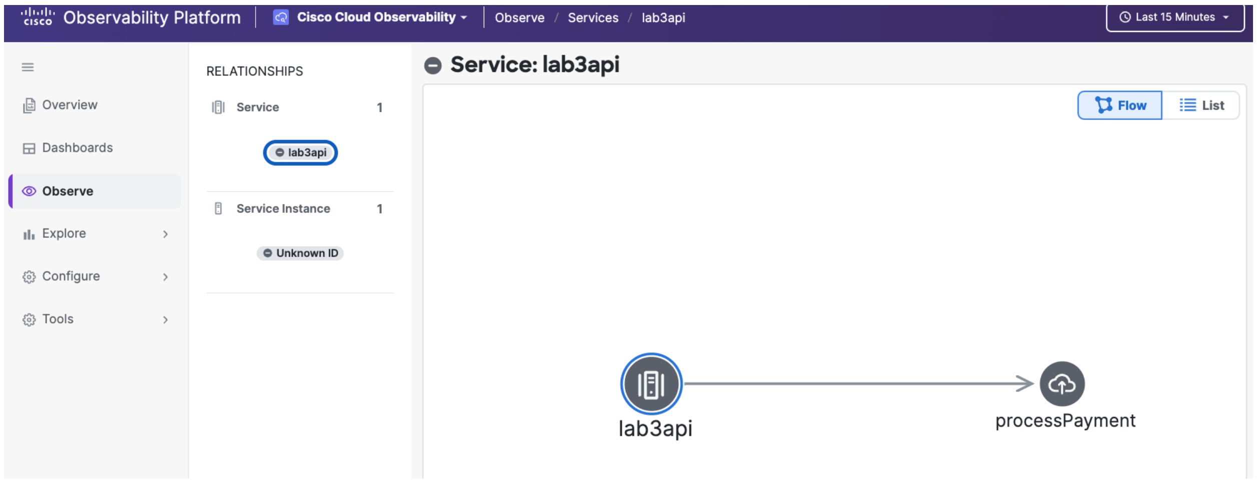
```
./load.sh
```

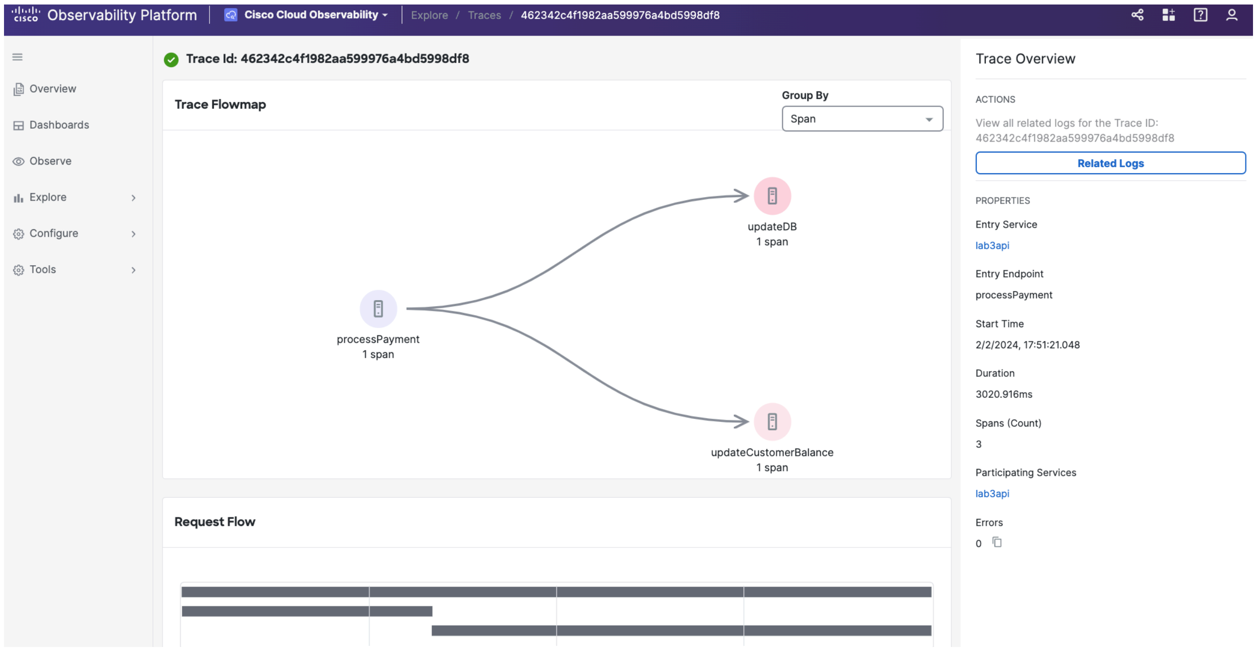
Check collector logs.



```
Trace ID: 7183174c83b4aaa9728c2c7452ab3324
Span ID: 35fe1428a2803edc
Flags: 1
LogRecord #2
ObservedTimestamp: 2024-02-02 20:58:14.8299 +0000 UTC
Timestamp: 2024-02-02 20:58:14.829 +0000 UTC
SeverityText: INFO
SeverityNumber: Info(9)
Body: Str(Update balance for:)
Attributes:
     -> code.filepath: Str(DemoPagoController.java)
     -> code.function: Str(updateCustomerBalance)
     -> code.lineno: Int(74)
     -> code.namespace: Str(io.demo.apis.digisicapis.DemoPagoController)
     -> thread.id: Int(34)
     -> thread.name: Str(http-nio-8081-exec-9)
Trace ID: 7183174c83b4aaa9728c2c7452ab3324
Span ID: 35fe1428a2803edc
Flags: 1
        {"kind": "exporter", "data_type": "logs", "name": "logging"}
```

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





Trace Id: 462342c4f1982aa599976a4bd5998df8

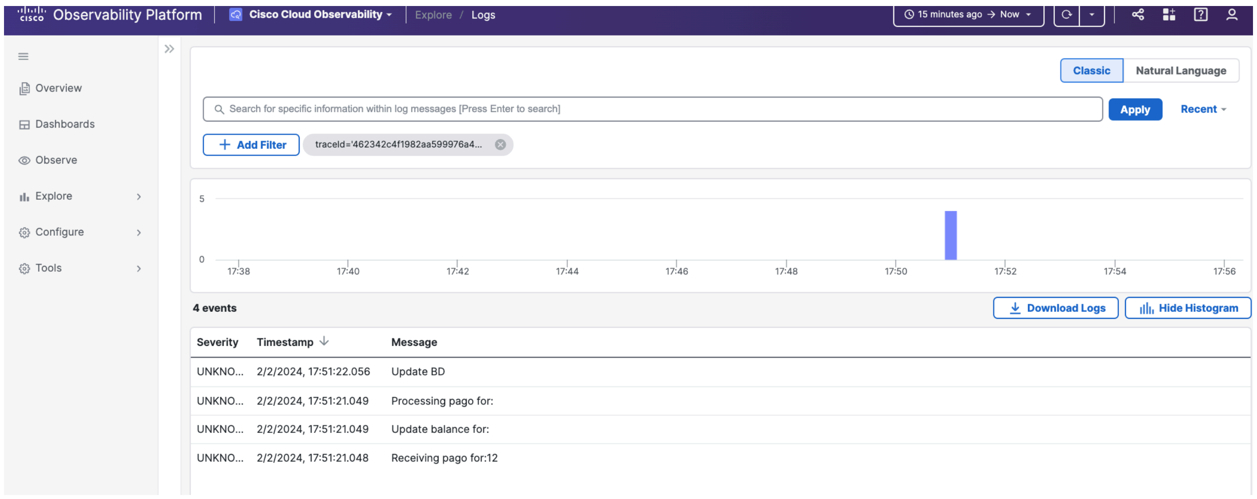
Trace Overview

ACTIONS

View all related logs for the Trace ID: 462342c4f1982aa599976a4bd5998df8

PROPERTIES

Entry Service	lab3api
Entry Endpoint	processPayment
Start Time	2/2/2024, 17:51:21.048
Duration	3020.916ms
Spans (Count)	3
Participating Services	lab3api
Errors	0



4 events

Severity	Timestamp	Message
UNKNO...	2/2/2024, 17:51:22.056	Update BD
UNKNO...	2/2/2024, 17:51:21.049	Processing pago for:
UNKNO...	2/2/2024, 17:51:21.049	Update balance for:
UNKNO...	2/2/2024, 17:51:21.048	Receiving pago for:12