

UNIVERSITY OF SOUTHERN DENMARK
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

BACHELOR THESIS

OBSERVABILITY FOR JOLIE MICROSERVICES

June 01, 2025

Author

Christian Skjolden
chskj22@student.sdu.dk

Supervisors

Marco Peressotti
peressotti@imada.sdu.dk

Matteo Trentin
trentin@imada.sdu.dk



Abstract

This thesis will work on the Jolie programming language ecosystem, which is designed for building microservice-based distributed systems. Jolie currently lacks observability support, which is critical for diagnosing, monitoring, and improving microservice architectures as they scale in complexity.

To address this gap in Jolie's ecosystem, this thesis designs and implements a standalone observability package that supports distributed tracing and logging by using existing industry and open source standards from the OpenTelemetry project, which has become a de facto standard for observability in distributed systems.

The package will allow developers to instrument their Jolie services with minimal effort and being able to configure it for deployment in diverse observability backends. This is possible due to the package's design, which is based on the OpenTelemetry Protocol (OTLP) and W3C Trace Context standards.

Through a comprehensive guide and multiple examples of how to use the package, this thesis demonstrates the practicality and effectiveness of the observability package. This lays the groundwork for making observability a focus in the Jolie ecosystem, providing a path for future enhancements and making observability feel native to Jolie.

Contents

1	Introduction	7
2	Preliminaries	8
2.1	Observability	8
2.1.1	Metrics	8
2.1.2	Tracing	8
2.1.3	Logs	8
2.1.4	Microservices and their need for evolved methods	8
2.1.5	Observability	9
2.2	OpenTelemetry	10
2.2.1	APIs	11
2.2.2	SDKs	11
2.2.3	OTLP	11
2.2.4	Collector	12
2.2.5	Supporting vendors	12
2.2.6	Important concepts	12
2.2.6.1	TraceId	12
2.2.6.2	SpanId and ParentSpanId	12
2.2.6.3	Context propagation	13
2.2.6.4	Exporters	13
2.2.6.5	Processors	13
3	Objectives	14
3.1	General goals	14
3.2	Tracing	14
3.3	Logging	15
4	Implementation	16
4.1	Dependencies	16
4.2	Initial decisions	16
4.3	Jolie interface	16
4.3.1	Types	17
4.3.2	Interfaces	17
4.3.3	Service	18
4.3.4	Generation	19
4.4	Telemetry config	20
4.4.1	Passing the config path	20
4.4.2	Jackson	20
4.4.3	Config types	22

4.4.3.1	Service Name	22
4.4.3.2	Default Tracer Name	22
4.4.3.3	Sampler	22
4.4.3.4	Cleanup Config	23
4.4.3.5	Span and Log Processor	23
4.4.3.5.1	Exporters	24
4.5	Telemetry core	25
4.5.1	Initial setup of OpenTelemetry SDK	25
4.5.1.1	Config mapping	26
4.5.1.2	Init telemetry	26
4.5.1.3	Data management & shutdown	26
4.5.1.4	Utility functions	27
4.6	Architecture overview	27
4.6.1	System relation	28
4.6.2	System example	28
4.7	Telemetry service class	29
4.7.1	General information	29
4.7.1.1	Embedding	29
4.7.1.2	Concurrent operations	29
4.7.1.3	Process data	30
4.7.1.4	Propagation data	30
4.7.2	Utility functions	31
4.7.2.1	ExtractAttributes	31
4.7.2.2	GetProcessDataOrThrow	32
4.7.2.3	GetActiveSpanOrThrow	32
4.7.3	SetTracer	33
4.7.4	StartProcess	33
4.7.5	EndProcess	33
4.7.6	StartSpan	33
4.7.7	EndSpan	33
4.7.8	GetPropagationData	33
4.7.9	Shutdown	34
4.7.10	SetStatus	34
4.7.11	AddAttributes	34
4.7.12	AddEvent	34
4.7.13	Log	34
5	Usage	35

5.1	Environment	35
5.2	Importing Telemetry	35
5.3	Running Jolie with telemetry	36
5.4	Config	37
5.5	Attributes	43
5.6	Embedding	44
5.7	SetTracer	45
5.8	StartProcess	46
5.9	StartSpan	49
5.10	GetPropagationData	51
5.11	AddAttributes	53
5.12	AddEvent	55
5.13	SetStatus	57
5.14	EndSpan	59
5.15	EndProcess	61
5.16	Shutdown	63
5.17	Log	64
6	Examples	67
6.1	Config example	67
6.2	Simple examples	68
6.2.1	Simple process	68
6.2.1.1	Simple Process	68
6.2.2	Propagate context	68
6.2.2.1	Propagate context preparation	68
6.2.2.2	Propagate context main service	69
6.2.2.3	Propagate context continue service	70
6.2.3	Attributes, Events and Logs	71
6.2.3.1	Propagate context continue service	71
6.3	Advanced examples	72
6.3.1	Multiple Tracers	72
6.3.1.1	Multiple Tracers preparation	72
6.3.1.2	Multiple Tracers math service	73
6.3.1.3	Multiple Tracers main service	74
7	Discussion and conclusion	75
7.1	Accomplishments	75
7.2	Known issues	75
7.3	Future work	76

7.3.1	New features	76
7.3.2	Jolie integration	76
7.3.3	Testing	77
7.4	Conclusion	77
	Bibliography	79

1 Introduction

Modern distributed systems increasingly rely on microservice architectures to achieve scalability, resilience, and rapid deployment. Jolie is a service-oriented programming language designed to simplify the construction of such systems. However, as these systems grow in complexity, the need for effective observability becomes essential. Observability enables developers to monitor, diagnose, and optimize distributed applications by providing an overview of their health and performance.

Jolie currently lacks integration with standardized observability frameworks such as OpenTelemetry, which limits developers' ability to monitor distributed Jolie systems effectively. To the best of our knowledge, and of the Jolie maintainers, this is the first effort to develop and integrate an OpenTelemetry-compatible observability package for Jolie.

This thesis will focus on fixing this gap by creating a standalone observability package for Jolie that fully integrates with OpenTelemetry and follows its standards. The package will provide distributed logging and tracing capabilities, giving developers the tools they need to monitor and understand their distributed applications.

The contributions of this thesis will be:

- **Research:** An in-depth exploration of existing observability technologies, frameworks and standards and selecting the most suitable ones for integration with Jolie.
- **Implementation:** Developing a standalone observability package for Jolie that supports distributed logging and tracing while making it feel native to the Jolie language.
- **Documentation:** Providing comprehensive documentation and examples to help developers understand how to use the package effectively in their Jolie applications.

This thesis will start in Section 2 by providing background into the different technologies and terms needed to understand observability in distributed systems after which the objective of the thesis will be declared in Section 3. Section 4 contains the implementation part aiming to provide a thorough understanding of how the package works, the structure and design decisions. In Section 5, the usage and example section will provide insight for developers on how to use the package in an easy and understandable way. The examples, in Section 6, aim to show how to structure the incorporation of the package into existing code. The discussion and conclusion in Section 7 will then finish up the thesis and provide insight into where work on observability in Jolie can go from here.

2 Preliminaries

2.1 Observability

Observability is a term that has gained a lot of popularity in recent times.¹ It is however nothing new in itself. Observability is just combining multiple older concepts together and evolving on the concept of monitoring. These concepts are:

- Metrics
- Tracing
- Logs

2.1.1 Metrics

Metrics is a way of getting an insight into how a service is running. This means collecting a lot of numerical data and then exporting them to another place to be consumed. Examples of this would be collecting the CPU usage, request processing time, error rates and much more. Collecting this data makes it possible to see how the health of a service evolves over time.

2.1.2 Tracing

Tracing is the act of following a request through a system. This has always been complicated to accomplish when run outside the development environment of your local machine and has lead to a chaos of log statements.

2.1.3 Logs

Logs have always been a part of programming, but it has been necessary to evolve from simple statements to the larger logging systems that is seen now. Logging can be essential for logging errors or stack traces when unexpected behavior happens in a program.

2.1.4 Microservices and their need for evolved methods

This is especially important because of how modern development has shifted. A large shift has been seen going away from monolithic services running on onpremise hardware that you could easily access and instead having microservice architectures gaining popularity. Resulting in the machine you were trying to maintain and keep an eye on being out of your hands.²

Metrics suddenly wasn't a matter of looking at one machine that you had direct access to. Instead you could have hundreds of services scaling up and down according to demand on many different machines. Making it near impossible to keep track of it.

¹<https://trends.google.com/trends/explore?date=all&q=%2Fm%2F03kb9w,%2Fg%2F112ly6qtd>, [1]

²<https://trends.google.com/trends/explore?cat=32&date=all&q=microservices>, [1]

Tracing became unmanageable. You had requests bouncing between many services and finding what caused the error or caused the request to be slow became a hassle.

Logs were now laying locally on a machine far from you and could disappear the moment the orchestration system managing your service deemed it unnecessary.

2.1.5 Observability

That meant a lot of people started looking to observability. Because by combining all these concepts with modern twists you suddenly had a much better idea of the health of a system.

This was done by collecting all the metrics data to one spot. Each service identifying themselves. Thereby being able to look into singular services causing problems.

Tracing became much larger and therefore turned into distributed tracing. A request would need to be traced across services end-to-end, which was done by giving each request an identity. This identity would be presented by a “TraceId”. This Id would be sent along with any request to other systems making it possible to understand which services it interacted with.

But in order to get a better idea of what was happening, more granular information was needed. This is what “spans” are for. A trace would contain multiple spans created throughout all services it entered.¹

Spans are smaller units of work inside the request. An example of this could be a call to a database or running an algorithm on the input. Spans would then contain a lot of information like it’s SpanId, start time, stop time, parent SpanId, events and eventual metadata. This making it possible to get a visual understanding of what happened in which order and how long it took.

¹<https://opentelemetry.io/docs/concepts/observability-primer/#distributed-traces>, [2]

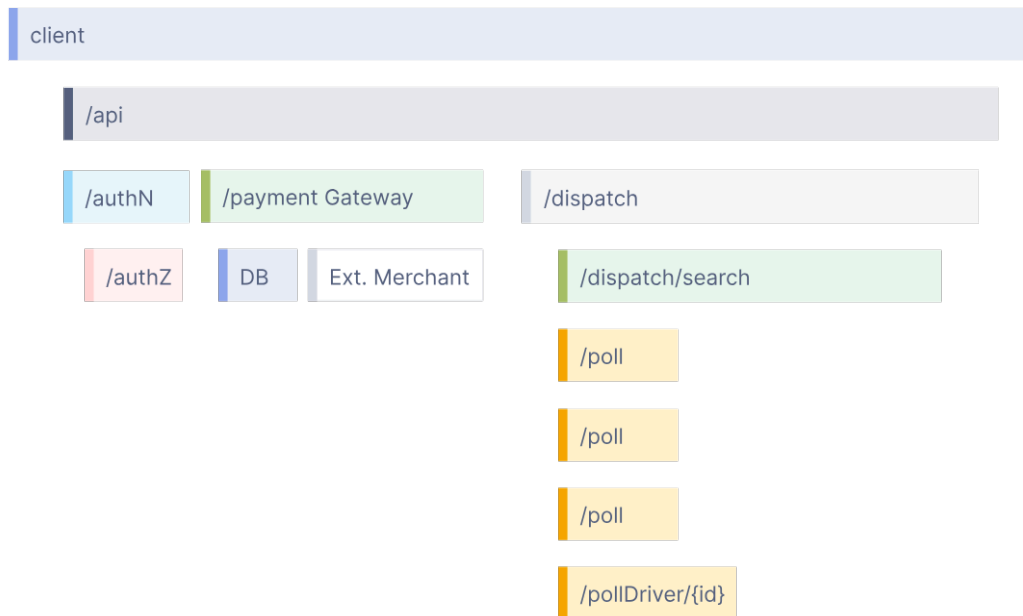


Figure 1: Waterfall diagram of a call with distributed tracing.

<https://opentelemetry.io/img/waterfall-trace.svg>

Logs could now also provide the context of the call, giving the service, TraceId and SpanId from which they came. The difference between logs and events being that events are attached to a span and comes with the span when exporting, and logs are unsorted messages that all get collected in one place and can then be sorted through with tools.¹

Due to this a lot of the blackbox behavior was eliminated. Instead you can now much more easily:

- Debug and troubleshoot problems. Now possible to trace a call and everything that led to it crashing. Inspecting only metadata relevant to that call and its data.
- Improve performance by seeing what operations take more time than expected.
- Enhance reliability by keeping a track of the metrics on different services or setting up alerts when calls began to cross certain thresholds deemed unhealthy, making developers aware of problems.

2.2 OpenTelemetry

OpenTelemetry (OTel) is an observability framework with the goal of creating a framework that is open-source and vendor agnostic. It is there to set standards for

¹<https://opentelemetry.io/docs/concepts/observability-primer/#logs>, [2]

how data is gathered and sent making it easy for all languages and systems to implement and be compatible with each other.¹

Before OpenTelemetry was a thing a lot of other people had tried to create smaller or proprietary standards. This was also how OpenTelemetry came to be, as it was the result of OpenTracing from the Cloud Native Computing Foundation and OpenCensus from Google merging.²

So by having two giants getting together and a lot of community backing, OpenTelemetry became the standard. This helps companies to avoid locking into using a specific observability backend/provider since it can easily change to another backend/provider using OpenTelemetry standards.

OpenTelemetry consists of 4 major parts:³fill: red)

- APIs
- SDKs
- OTLP
- OpenTelemetry Collector

2.2.1 APIs

The OpenTelemetry APIs define a common language and set of operations for all the telemetry generation. This makes it easy to understand across technologies and it clearly defines what needs implementing.⁴

2.2.2 SDKs

The OpenTelemetry SDKs are language-specific implementations of the API. They are libraries that have ready-made implementations for different languages whereby the developer will not have to worry about following the API, but can instead just use the tools.

These already exist for most major languages.⁵

2.2.3 OTLP

OTLP, short for OpenTelemetry Protocol, is a protocol designed and created for transportation of OpenTelemetry data. It defines how the data is encoded and transmitted between different components. It's designed to support both HTTP and gRPC.⁶

¹<https://opentelemetry.io/docs/what-is-opentelemetry/>, [2]

²<https://opentelemetry.io/docs/what-is-opentelemetry/#history>, [2]

³<https://opentelemetry.io/docs/concepts/components/#collector>, [2]

⁴<https://opentelemetry.io/docs/specs/otel/>, [2]

⁵<https://opentelemetry.io/docs/languages/>, [2]

⁶<https://github.com/open-telemetry/opentelemetry-proto/tree/main/docs>, [3]

It is made to create interoperability and make communication between OpenTelemetry components and external vendors easy and standardized.

2.2.4 Collector

The collector made by OpenTelemetry is a versatile, vendor agnostic piece of infrastructure that can easily be deployed. It's goal is to be easily deployable and uphold the standards of OpenTelemetry while being able to receive from and send to most observability tools.

It works by being able to receive from one or multiple sources in multiple formats. Then putting them through processors to do things like filtering, adding attributes or sampling. Then exporting that onto one or multiple backends through custom exporters in whatever format needed.

This collector can either be deployed side by side with the application itself and letting it handle the exporting and configuration, or it can be deployed as simply a collection point or proxy.¹

2.2.5 Supporting vendors

Many existing vendors have already added some form of OTel support with more and more adding OTLP native support.

- Jaeger (OTLP support)[4]
- Zipkin (Collector export support)²
- Dynatrace (OTLP support)[5]
- Sentry (OTLP support)[6]
- Grafana (OTLP support)[7]
- Prometheus (OTLP support)[8]

2.2.6 Important concepts

TraceId

TraceId is a an identifier for a request end-to-end which means it is globally unique and need to be propagated throughout multiple services. Therefore, it can be used to correlate all events throughout the whole transaction. This TraceID is being used to look up what happened in a transaction and the thing to be logged during errors.

SpanId and ParentSpanId

SpanId is an identifier for a certain span within a request and needs to be unique within the transaction. This id is the same used for the ParentSpanId. By doing so, a

¹<https://opentelemetry.io/docs/collector/>, [2]

²https://opentelemetry.io/docs/specs/otel/trace/sdk_exporters/zipkin, [2]

relational connection between spans is established making it easy to see what work is happening within other calls or methods.

Context propagation

Context Propagation is the act of transferring the context of a transaction to another service. This involves the TraceId to continue the work on that transaction and the active SpanId to be set as the next span's ParentSpanID. This is the core of distributed tracing. It will keep multiple services and systems aware of which operation they are currently doing work on without sending the whole information tree.

Exporters

Exporters are responsible for handling transport of all data that the OpenTelemetry packages want to export. They handle the protocols such as HTTP and GRPC, timeouts, endpoints and eventual authentication.

Processors

Processors are where Spans and data are sent from the package before exporting. They handle any attribute addition or filtering. Hence, they decide what data is sent and when.

3 Objectives

This project will create a standalone telemetry package for Jolie will be created in this project with the aim of covering tracing and logging but excluding metrics as this is out of scope.

3.1 General goals

The goal will be to create a telemetry package that is easy to use and has a low barrier of entry. The developer should be able to add telemetry to their service with minimal effort.

This should be done by providing a simple interface with a minimal API that is easy to understand and use. The functions should have sensible defaults, so the developer can get started quickly without having to worry about all the details. But still allowing them to customize the behavior of the telemetry package if they want to.

Under the hood it should have a config handling setting up OTLP-compliant exporters and processors. This config should make it easy to export into any telemetry system that supports OTLP and the telemetry backend should be reuseable across multiple Jolie services, so the developer doesn't have to worry about setting up a new telemetry backend for each service and creating that overhead.

3.2 Tracing

The tracing system should be able to handle concurrent processes. This means that the tracing system should be able to handle multiple calls at the same time without any issues.

Developers should be able to choose the key that process data is stored under, or simply use the default key. This gives the developer full control over the process data and allows them to customize the behavior of the tracing system.

When starting a process the developer should be able to choose whether to start a new request or inherit the context of the calling request. This allows the developer to control how the tracing system behaves and how it interacts with other processes.

A service should be able to have multiple tracers in the same Jolie service. These tracers should be able to have different instrumentation names.

Created spans should have custom names. This allows the developer to describe what the span is covering. The system should automatically set the correct relation to the parent spans and any propagated context.

The developer should be able to set the status of a span, add attributes to the span and add events to the span in order to save metadata about the call and also provide a better understanding of it.

3.3 Logging

The logging should provide a simple and fast way to log messages. These logs should be able to be made without the context of an active process.

The developer should be able to set the severity level and add attributes to the logs. This would allow setting the importance of the logs and providing more context to them.

It should also be possible to attach the context of an active process to allow for better correlation between logs and traces. This means that the logs should be able to include the TraceId and SpanId of the active process, allowing for better debugging and understanding of the logs.

4 Implementation

4.1 Dependencies

- Jolie 1.13[9]
- Java 21 [10]
- Jolie2Java tool¹
- OpenTelemetry Java SDK²
- Jackson³

4.2 Initial decisions

To create an easy way to add telemetry to a Jolie service, a package will be created. This package will be able to be imported into any Jolie service and will then be able to be used.

Since Jolie has support for writing the internals of it in Java and just calling it from Jolie, this will be used as an advantage allowing the use of the power of Java libraries and the ease of use and strict interfaces of Jolie.

For this, the Jolie2Java tool will be used allowing creation of the interfaces, types and faults in Jolie and then generating equivalent interfaces, types and faults in Java.

Jolie2Java, however, requires Jolie to be on 1.13 or higher and Java 21 or higher.

The OpenTelemetry Java SDK will be used. This is done in order to avoid “reinventing the wheel” and ensuring being up to standard with all OpenTelemetry APIs and not having to manage the large tool chain. Instead, the library can quickly be updated in case of updates.

4.3 Jolie interface

First a package template is created by running

```
1 npm init jolie
```

```
bash
```

Listing 1: Command to run to initialize the package template.

This will initiate a Jolie package with all needed dependencies. The most important parts are the pom file where dependencies are declared, a Java project with a pre generated interface and finally a main.ol file. Within this, all operations are declared and these operations are already connected to the equivalent Java methods.

¹<https://docs.jolie-lang.org/v1.13.x-git/language-tools-and-standard-library/technology-integration/java/index.html>, [11]

²<https://mvnrepository.com/artifact/io.opentelemetry>, [12]

³<https://mvnrepository.com/artifact/com.fasterxml.jackson.core>, [12]

4.3.1 Types

Now in the main.ol all types for the operations must be defined. This includes all types and subtypes for both the parameters and results.

The types for this package looks as so:

```

1  type PropagationData : string
2  type CallInstanceIdentifier : string
3  type StartProcessParameters {
4      id? : CallInstanceIdentifier
5      propagationData? : PropagationData
6  }
...
24 type logParameters {
25     message : string
26     severity? : string( enum([ "TRACE", "DEBUG", "INFO", "WARN",
                                "ERROR", "FATAL", "trace", "debug", "info", "warn", "error",
                                "fatal" ]) )
27     attributes? : undefined
28     id? : CallInstanceIdentifier
29 }
```

Listing 2: main.ol, Type definition.

This declares all things coming in and out of the program. Take the logParameters type as an example.

- **Message** is a required string.
- **Severity** is an optional string with an enum filter.
- **Attributes** are optional and can be of an undefined type. Also meaning that any type or complex type will be accepted.
- **Id** is an optional CallInstanceIdentifier type. So it will inherit the type of CallInstanceIdentifier.

4.3.2 Interfaces

Next is the definition of the interfaces. This defines the calls to the package from the Jolie services using it. For this, it is necessary to define which types are accepted in a request and output as response and also to define all exceptions that an operation can throw. This allows the developer to be aware of and handle these errors when they get thrown.

```

30 interface TelemetryInterface {
31     RequestResponse:
32         setTracer( string )(void),
33         startProcess(StartProcessParameters)(CallInstanceIdentifier)
           ExistingProcessTelemetryException(string),
34         startSpan(CreateSpanParameters)(void) throws
           ProcessNotFoundTelemetryException(string),
           getPropagationData(CallInstanceIdentifier)(PropagationData)
35         throws ProcessNotFoundTelemetryException(string)
           SpanNotFoundTelemetryException(string),
...
42         shutdown(void)(void),
43 }

```

Listing 3: main.ol, Interface definition.

Take in this case the startProcess:

- **Takes in** startProcessParameters
- **Ouputs** CallInstanceIdentifier
- **Can throw** an ExistingProcessTelemetryException

4.3.3 Service

Lastly is the service block. This is already generated by Jolie2Java.

```

44 service Telemetry {
45     inputPort ip {
46         location: "local"
47         interfaces: TelemetryInterface
48     }
49     foreign java {
50         class: "org.jolie_lang.joliex.telemetry.Telemetry"
51     }
52 }

```

Listing 4: main.ol, Declaration of the service itself.

This consists of:

- **Location** defining how Jolie services will connect. In this case it is locally since Jolie services will be embedding this package.
- **Interface** defining what calls there are. It references to the previously created interface.

- **Foreign Java** defining the path to the Java class created by Jolie2Java.

4.3.4 Generation

Now this command can simply be run:

```
jolie2java --translationTarget 0 --overwriteServices true --  
1 outputDirectory "./src/main/java" --sourcesPackage ".spec" bash  
main.ol
```

Listing 5: How to generate package methods from Jolie interface.

This will then generate all types as seen below:

```
spec/  
├─ faults/  
│ └─ ExistingProcessTelemetryException.java  
│ └─ ProcessNotFoundTelemetryException.java  
│ └─ SpanNotFoundTelemetryException.java  
└─ types/  
    ├─ AddAttributeParameters.java  
    ├─ AddEventParameters.java  
    ├─ CreateSpanParameters.java  
    ├─ LogParameters.java  
    ├─ SetStatusParameters.java  
    └─ StartProcessParameters.java
```

Listing 6: Tree structure of generated files.

And the Telemetry.java file will have the methods which implementation just needs to be filled. It will already have all references to the parameters, return value and exceptions.

```
public final class Telemetry extends jolie.runtime.JavaService
3 implements
  org.jolie_lang.joliex.telemetry.spec.TelemetryInterface {
4
5     public void
      addEvent( org.jolie_lang.joliex.telemetry.spec.types.AddEventParam
        request ) throws
      org.jolie_lang.joliex.telemetry.spec.faults.SpanNotFoundTelemetry
      org.jolie_lang.joliex.telemetry.spec.faults.ProcessNotFoundTelemetry
      {
6         /* developer code */
7     }
8
9     ...
10
11     public void shutdown() {
12         /* developer code */
13     }
14 }
```

Listing 7: Example of generated code in the Telemetry.java

4.4 Telemetry config

4.4.1 Passing the config path

This package will be using a config to set up the general telemetry settings. This config will be another file. The path to this file will be passed to the JVM when starting the Jolie instance. This allows the developer to set up the telemetry settings without having to change the code.

So this is how it would look:

```
1 jolie -Dtelemetry.config=PATHTOCONFIG PATHTOSERVICE/SERVICE.ol bash
```

Listing 8: Example of passing the path of the config to the telemetry service.

4.4.2 Jackson

For this config a JSON file was chosen. This is because a JSON file is easy to read and write. It is also easy to parse and can be easily extended in the future.

In order to handle the parsing of the JSON file, Jackson was chosen. Jackson was chosen instead of Gson because Jackson has a better support for polymorphic deserialization and thereby handles the different types of samplers and processors in a much better way.

An example of polymorphic deserialization is that it can handle the different types of samplers and processors in a much better way. Like in the example below:

Firstly, the `samplerType` is set to be a polymorphic type. Then the different types of samplers are referenced. This means that the deserializer will know which type to use when deserializing the JSON file based on the value of the `samplerType` field.

```

323 @JsonTypeInfo(use = JsonTypeInfo.Id.NAME, property =
    "samplerType")
324 @JsonSubTypes({
325     @JsonSubTypes.Type(value = AlwaysOnSamplerConfig.class, name =
        "AlwaysOn"),
326     @JsonSubTypes.Type(value = AlwaysOffSamplerConfig.class, name
        = "AlwaysOff"),
327     @JsonSubTypes.Type(value =
        TraceIdRatioBasedSamplerConfig.class, name = "TraceIdRatioBased")
328 })

```

Listing 9: TelemetryConfig.java, Declaration of possible classes to deserialize to.

Here the classes are showing their different needs.

```

341 class AlwaysOffSamplerConfig extends SamplerConfig {
342     @Override
343     public Sampler createSampler() {
344         return Sampler.alwaysOff();
345     }
346 }
347
348 class TraceIdRatioBasedSamplerConfig extends SamplerConfig {
349     private double ratio = 0.5;
350     ...
351     @Override
352     public Sampler createSampler() {
353         return Sampler.traceIdRatioBased(ratio);
354     }
355 }

```

Listing 10: TelemetryConfig.java, The difference between the classes.

The differences between them being that the `TraceIdRatioBased` one is now able to accept an extra value in the form of float that decides the ratio, which isn't relevant in the other classes.

So the two different ways they would look is seen below:

```
1  "sampler": {
2    "samplerType": "AlwaysOff"
3  },
4  "sampler": {
5    "samplerType": "TraceIdRatioBased",
6    "ratio": 1
7  }
```

Listing 11: Example of how samplers can look

This all allows for a much better way of handling the different types of samplers and processors. Now the classes only need to implement the methods that are relevant for them.

4.4.3 Config types

Service Name

The Service name is a mandatory identifier for your service. An example would simply be:

```
1  "serviceName": "AuthenticationBackend",
```

Listing 12: Example of serviceName.

Default Tracer Name

The default tracer name ensures a default value for the instrumentation name. This can be overwritten on a per service basis. An example of this is:

```
1  "defaultTracerName": "my-default-instrumentation",
```

Listing 13: Example of defaultTracerName.

Sampler

The sampler determines how many spans are recorded. This is done by setting a sampling strategy.

- **AlwaysOn** meaning all spans would be sampled
- **AlwaysOff** meaning no spans would be sampled
- **TraceIdRatioBased** will decide based on a float the ratio of spans that will be sampled.

An example would be:

```
1  "sampler": {
2    "samplerType": "TraceIdRatioBased",
3    "ratio": 1
4  },
```

JSON

Listing 14: Example of sampler

Also offers `createSampler()` method to create the sampler.

Cleanup Config

Cleanup config is defining all the actions done on the spans before cleaning up.

It contains these values:

- **TimeoutInSeconds** (integer, default 30): max amount of time waiting in each step.
- **AddEvent** (boolean, default true): whether an event should be added onto spans before they are ended.
- **ErrorMessage** (string, default "Jolie instance shutdown. Cleaning up spans."): the message in the event if one is added.
- **SetError** (boolean, default true): whether the status of a span should be set as "Error" before it is ended.
- **EndSpan** (boolean, default true): whether the spans should be ended at all. If this is not set to true, the spans will never be ended and therefore never sent.

An example of how it would look:

```
1  "cleanupConfig": {
2    "timeoutInSeconds": 10,
3    "addEvent": true,
4    "errorMessage": "Jolie instance shutdown. Cleaning up spans.",
5    "setError": true,
6    "endSpan": true
7  },
```

JSON

Listing 15: Example of cleanupConfig

This component also offers a `getSpanEnder()` that will return a `Consumer` that will end the span in the way defined in the config.

Span and Log Processor

The span and log processors look alike.

There are three types of processors:

- **Batch** will batch the spans and logs before sending them to the exporter.
- **Simple** will send the spans and logs immediately to the exporter.

- **Debug** will log the spans and logs to the console.

The batch processor requires the following extra values:

- **MaxExportBatchSize** (integer, default 512): max size of the batch before it is sent to the exporter.
- **MaxQueueSize** (integer, default 2048): max size of the queue before it is sent to the exporter.
- **ScheduleDelayInMs** (integer, default 1000): max time between spans being sent to the exporter.

Both the simple and batch processors have the following values:

- **Exporter** is another configuration defined in the config. It decides where the things in the processor is sent to and therefore needs to be in the processor.

An example of this is:

```
1  "processorType": "batch",
2    "maxExportBatchSize": 512,
3    "maxQueueSize": 2048,
4    "scheduleDelayInMs": 1000,
5    "exporter": {EXPORTER}
6  },
7 },
8 {
9   "processorType": "simple",
10  "exporter": {EXPORTER}
11 },
12 {
13   "processorType": "debug"
14 }
```

Listing 16: Example of processors.

This component also offers a `createProcessor()` method that will return the given processor.

Exporters

The span and log exporters also look alike.

There are two types of exporters:

- **HTTP** will send the spans and logs to the exporter using HTTP.
- **GRPC** will send the spans and logs to the exporter using GRPC.

Both the HTTP and GRPC exporters have the following values:

- **Endpoint** is the URI of where the spans and logs are sent.
- **TimeoutInSeconds** (integer, default 30): max time to wait for the exporter to send the spans and logs.
- **Headers** are headers set on the outgoing requests. Can be things such as auth.
- **ConnectTimeoutInSec** (integer, default 30): max time to wait for the exporter to connect.

Examples of this are:

```
1  "exporter": { JSON
2    "exporterType": "http",
3    "endpoint": "http://host.docker.internal:4318/v1/logs",
4    "timeoutInSeconds": 30,
5    "connectTimeoutInSeconds": 30,
6    "headers": {
7      "Authorization": "Bearer token"
8    }
9  },
10 "exporter": {
11   "exporterType": "grpc",
12   "endpoint": "http://host.docker.internal:4317",
13   "timeoutInSeconds": 30,
14   "headers": {
15     "Authorization": "Bearer token"
16   }
17 }
```

Listing 17: Example of exporters.

This component also offers a `createExporter()` method that will return the given exporter.

4.5 Telemetry core

The telemetry core has three purposes:

- Initial setup of OpenTelemetry SDK
- Manages and cleans up all TelemetryService data
- Utility functions for all TelemetryService classes

4.5.1 Initial setup of OpenTelemetry SDK

The telemetry core can only have one instance in the system. All service classes will have a reference to the same telemetry core.

The telemetry core will be the first thing to be initialized. This is done by the telemetry service class the first time one is created.

The telemetry core will then be responsible for the initial setup of OpenTelemetry SDK. This includes setting up the tracer and logger providers, as well as the processors and exporters.

Config mapping

- It will first read the system property “telemetry.config” that was passed to the JVM.
- It will then use the path in that property to locate the config file.
- The config file will subsequently be loaded into memory as a TelemetryConfig object with a Jackson object mapper.
- Next it will save the config inside the core.
- Run the init telemetry flow.

Init telemetry

- It will first build the TracerProvider and LoggerProvider with the resource attached.
- Next it will attach the processors to the providers gotten from the config.
- Both providers are then attached to the OpenTelemetry SDK.
- The default tracer is subsequently stored in the core.
- Finally the cleanup shutdown hook is created and activated.

Data management & shutdown

A big problem was encountered during the development of the telemetry package. Any spans not ended would be left in the system. This meant that if a Jolie instance was stopped, all spans that were not ended would be left in the system. This could cause a lot of lost data.

A lot of ways to fix this were explored. But there was no support in Jolie itself to tell when either a call or the instance stopped. So there was no event there that the package could use. This would, however, be the best solution in the future, but was out of scope of this project.

To try and solve this an implementation with a shutdown hook in each service class was considered. This would mean that every time a service class was initialized, a shutdown hook would be added to the JVM and thereby end all spans and flush the logger and tracer providers. The consequence was then that if a lot of service classes were created, a lot of threads would be created.

Instead, it was decided to let the core maintain a list of all service classes process data and thereby letting the core be responsible for cleaning up all spans and logs when the Jolie instance was stopped

This data would be maintained by a subscribe and unsubscribe method, whereas when a service class was created, it would subscribe to the core and subsequently add the process data to the core's list of process data.

Even if a service was stopped with spans still running, the core would have a reference to the process data and therefore resulting in the core being able to clean up all spans and logs, when the Jolie instance was stopped.

This would create a clean shutdown on normal exits(`System.exit()`, uncaught exceptions, etc.). But it would not work on power failures, kernel panics, os crashes, segmentation fault or `Runtime.halt()` since this would not trigger the shutdown hook.

Utility functions

The telemetry core also provides some utility functions for the service classes. These are:

- `GetTracer(name?)`: returns the default tracer or creates a new one with the given instrumentation name.
- `GetLogger`: returns the shared logger object.
- `GetTelemetryServiceDataMap`: generates a UUID, allocates a new `ProcessData`, stores it in the core map, and returns both
- `RemoveTelemetryServiceDataFromMap`: ends all spans for the given service id with the cleanup config and removes its `ProcessData` from the core map.

4.6 Architecture overview

Before going on with the service class here is an overview of how these classes relate to each other.

4.6.1 System relation

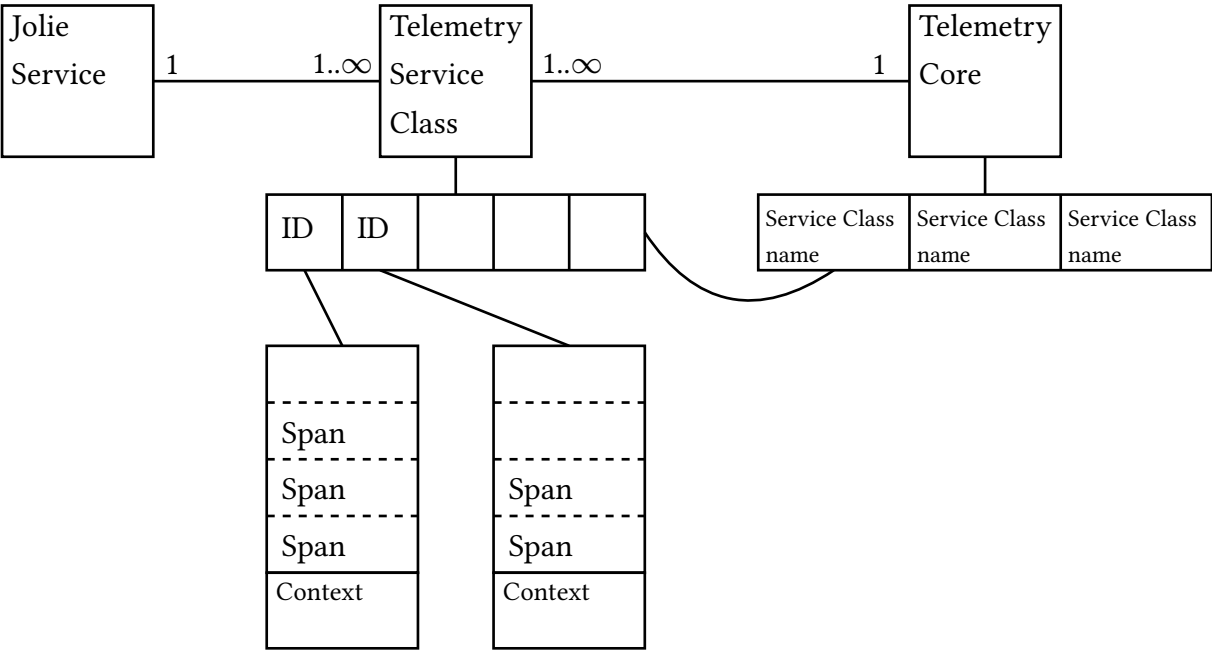


Figure 2: An overview of how services relate.

This shows how each Jolie service can have multiple Telemetry Service classes, while the Telemetry Service classes can only be attached to one Jolie Service.

The Telemetry Service class can only have one Telemetry core, while the Telemetry core is connected to all service classes because of its singleton relationship.

All Telemetry Service classes will then have a dictionary of process information including the context and spans.

The Telemetry core will have a reference to the dictionary in each Telemetry Service class.

4.6.2 System example

This means that a system could end up looking like this:

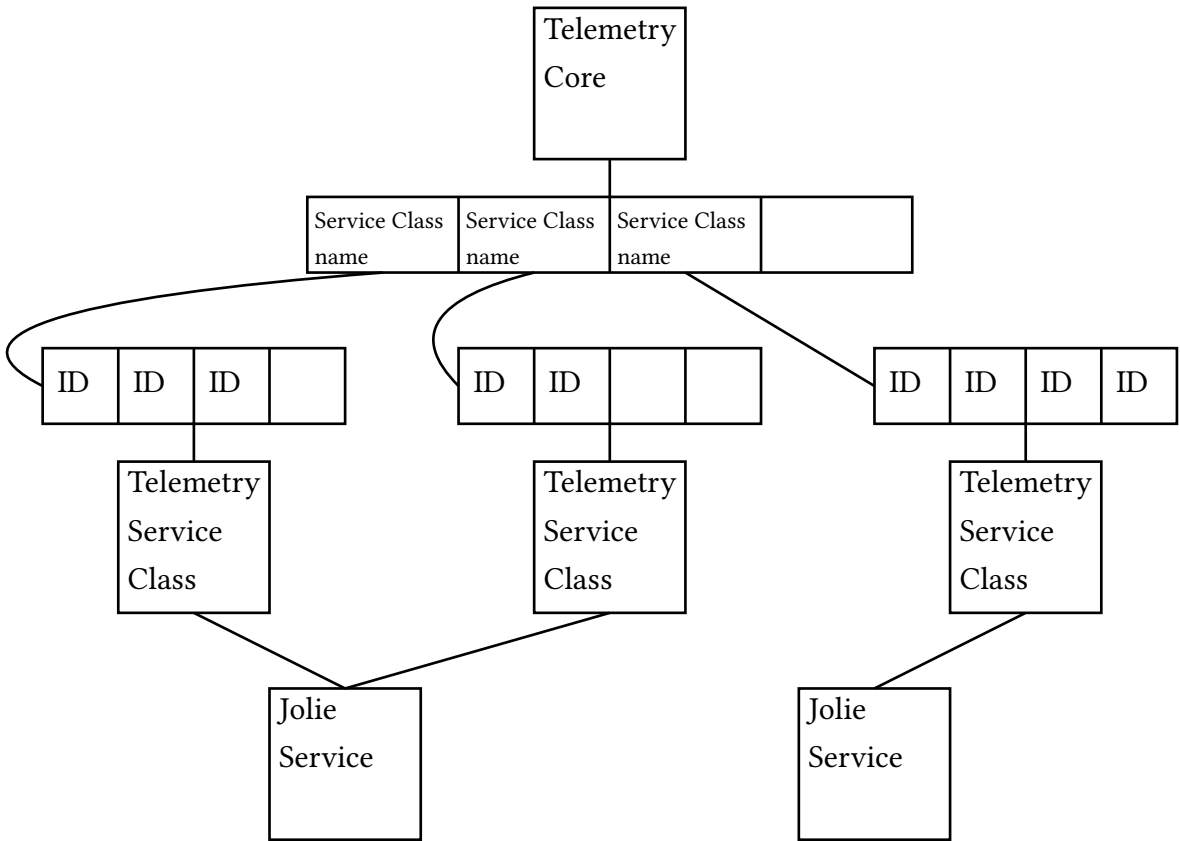


Figure 3: An example of how a system with 3 Telemetry service classes could look.

4.7 Telemetry service class

4.7.1 General information

Embedding

Each Jolie service that includes telemetry will get its own instance of the telemetry service class. To ensure a single shared telemetry core, for this a singleton pattern will be used when handling the telemetry core.

This means that the telemetry core will be created once and then shared between all telemetry service classes.

This is done by using a static final variable in the telemetry service class. This will ensure that the telemetry core is only created once and then shared between all instances of the telemetry service class.

In the telemetry service constructor it then calls the core’s utility functions to get the process map. This will then be used to store all process data for the service class.

Concurrent operations

Jolie is a language natively supporting concurrency. This means it can both support multiple concurrent calls (which will be referred to as processes) and also running

code in parallel inside a process. To handle this multiple protections will be used, such as:

- The service wide process map will use a concurrent hash map to handle concurrent operations.
- Use synchronization blocks whenever modifying, adding or removing data from the process data. This includes changing the context, adding, removing or modifying spans.

This means that the process map will be able to handle concurrent operations without any issues.

Process data

The process data consists of two different parts.

- The context will store the information like the `TraceId`.
- The span stack will store the spans that are created during the process.

The reason a stack was chosen was that spans work in a last-in-first-out(LIFO) manner. This means that the last span created will be the first one to be ended. This is because spans are created in a tree structure, where the parent span is the one that created the child span.

For this, the java `ArrayDeque` was chosen instead of the java stack, because the stack uses an old API and is based on the vector class and inbuilt synchronization. Deque instead allowed us to handle synchronization ourselves while still having the LIFO functionality.

These things together allow for a faster and more memory efficient stack implementation.[13]

Propagation data

This system will use the W3C Trace Context standard for propagation. [14]

The form of this looking like this: `Version-TraceId-SpanId-Flags`

An example is: `00-bae70494abc7d8a66aef5be8c96567ed-b45e125b65b817eb-01`

- Version: 2 hexidecimals
- TraceId: 32 hexidecimals
- SpanId: 16 hexidecimals
- Flags: 2 hexidecimals

4.7.2 Utility functions

ExtractAttributes

The job of the method is to take a Jolie value and turn it into an Attributes object that can then be used in multiple operations.

This method is interesting because of its job of translating the JolieValue object and thereby taking a dynamically typed data model and translating it into the strongly typed values that OpenTelemetry needs.

It is however designed to only capture the top-level primitive values and lists of primitive values. This is done in order to match the form of data that OpenTelemetry expects.

This means that values and code like this:

```
1  attributes = "Ignored value" // Will not be included
2  bunchOfValues[0] = 8
3  bunchOfValues[1] = "Some information"
4  bunchOfValues[2] = 3.14159
5  bunchOfNumbers[0] = 2
6  bunchOfNumbers[1] = 4
7  bunchOfNumbers[2] << bunchOfValues // Everything but the first
   element will be ignored
8  attributes << {
9    ImportantNumber = 25
10   ImportantString = "Very important information"
11   CakeIsALie = true
12   ListOfValues << bunchOfValues
13   ListOfNumbers << bunchOfNumbers
14 }
```

Jolie

Listing 18: Example of setting of attributes in Jolie.

Will be translated into this:

```
1  {
2    "ImportantNumber": 25,
3    "ImportantString": "Very important information",
4    "CakeIsALie": true,
5    "ListOfValues": ["8", "Some information", "3.14159"],
6    "ListOfNumbers": [2, 4, 8]
7  }
```

JSON

Listing 19: Example of what attributes are converted to.

The method will do its best to conserve the right primitive types. But in cases of mixed data types inside a list, they will all be turned into a string. It will also conserve the name of the keys and use them as the name of the attributes.

So how is this accomplished?

It first extracts all the top-level values. Then it will iterate through them and check the amount of data elements inside.

If this is one it will go to the `addSingleValue` method.

If it is more than one it will go to the `addArrayValue` method.

addSingleValue

Adding a single value is quite simple. Check what primitive value it is inside. Then turn that value into that primitive type. Finally add that into the `Attributes` object with the same key as it came with.

addArrayValue

Adding array values are, however, not quite as simple. Since arrays can have multiple values, all of them need to be checked because the type of array being added must be defined and the ability of having pure primitive arrays should be kept

Firstly, it must be decided whether it is only of one type or mixed by mapping them all to their primitive equivocal enum value and then checking whether there is one or more unique values. If there is more than one unique value in that map, this means that it is mixed and will be treated as such. If not, there is one value which is the type of the array.

In the case of having one type of value, an array of that type can be made and inserted that into the `Attributes` with the same key as it came with.

In case of it being mixed, the string value of each element will be added to the `Attributes` as a string array with the same key as it came with.”

GetProcessDataOrThrow

Centralizes lookup of `ProcessData` by id. If the id is missing, it will throw `ProcessNotFoundTelemetryException`.

GetActiveSpanOrThrow

Retrieves the current span from the `ProcessData`’s stack. If no span is active, it will throw `SpanNotFoundTelemetryException`.

4.7.3 SetTracer

SetTracer will get a new tracer from the core using the instrumentation name.

The default tracer will then be replaced with the new one in order to allow the developer to set a custom instrumentation name for the tracer.

4.7.4 StartProcess

StartProcess registers a new process in the map, initializes its context, and returns its identifier.

You can provide an id and/or propagation data, or omit both.

If no id is supplied, one is auto-generated (allowing custom id scenarios, such as using a Jolie CSETs).

When propagation data is present, it will be used to initialize the context. If no propagation data is present, a new context will be created.

An ExistingProcessTelemetryException is thrown if the id already exists.

4.7.5 EndProcess

EndProcess removes the process data for that id and ends all spans in it.

This is done in order to ensure that all spans are ended and sent to the processors.

4.7.6 StartSpan

StartSpan creates and stores a new span in the process data for the given id.

It retrieves the process data for the given id and uses the tracer to create a span.

It will then check if the span stack is empty. If it is then it will set the parent of it as the context in the process data. This ensures that a new span will continue any call given by propagation data during process start.

If it is not empty it will instead set the parent of the new span as the most recent one. Ensuring that the span tree is continued.

4.7.7 EndSpan

EndSpan ends the current span in the process data for the given id.

It pops it from the stack of the process and ends it. This then sends it to the processors.

4.7.8 GetPropagationData

GetPropagationData returns the propagation data for the current span.

It retrieves the process data for the given id, then extracts the trace context from the current span.

4.7.9 Shutdown

Shutdown removes the process data from the core's storage and applies the configured cleanup logic to end all spans in accordance.

4.7.10 SetStatus

SetStatus sets the status of the current span.

It retrieves the process data for the given id and sets the status of the current span.

4.7.11 AddAttributes

AddAttributes adds attributes to the current span.

It takes the Jolie attributes and converts it into an Attributes object.

It then retrieves the process data for the given id and gets the current span from the stack.

It will then add the attributes to the span.

4.7.12 AddEvent

AddEvent adds an event to the current span.

It takes in an id, event name and optional attributes. Then converts the attributes to an Attributes object, if present.

It retrieves the process data for the given id and gets the current span from the stack. Then adds the event to the span.

4.7.13 Log

Log emits a log to the logging backend.

It takes in a message and has the option to take in a process id, severity of the log and attributes.

If a process id is present it will get the process data for that id. Then get the current span and attach that context to the log.

If attributes are present it will extract them with the extractAttributes method and add the Attributes object to the log.

If a severity is present it will define the severity of the log to be that.

Finally it will add the message and emit it.

The log will then be sent to the log processors.

5 Usage

5.1 Environment

The environment needs:

- Jolie 1.13+
- NodeJS version 18+ for JPM
- JPM (Jolie package manager)
- One or more OTLP compliant span endpoints
- One or more OTLP compliant log endpoints

For both the log and span endpoint it is recommended to deploy an OpenTelemetry Collector alongside the Jolie instance. This allows the program to quickly unload spans and logs, and also unlock more advanced filtering and exporting options. See more about the collector here: <https://opentelemetry.io/docs/collector/>

5.2 Importing Telemetry

Due to current limitations in Jolie 1.13, when using java packages, the program needs to be executed from a directory containing the package. This means that any directory that the code is started from needs to be a Node project with the telemetry package downloaded through JPM.

Initialization of this can look like this:

```
1 npm init --y
2 jpm init
3 jpm install jolie-telemetry
```

This will download everything needed to run the package inside the folder you're currently in.

In order to then import it into the Jolie file simply add this:

```
1 from jolie-telemetry import Telemetry
```

Warning: The name of the package is open to change if put into the @jolie org in npm. Keep up to date here: <https://github.com/CSkjolden/Jolie-Telemetry>

5.3 Running Jolie with telemetry

Running Jolie with a reference to the config:

The telemetry package uses system properties passed when starting a program. This is how it finds the path to the config.

How it works can be seen here:

```
1 jolie -Dtelemetry.config=PATHTOCONFIG/CONFIG.json  
PATHTOFILE/FILE.ol
```

[Bash](#)

System properties are given with the -D flag. The property that needs to be set is the *telemetry.config* property.

Due to current limitations in Jolie 1.13, when using java packages, the program needs to be executed from a directory containing the package. See Section 5.2 in order to ensure that.

5.4 Config



The diagram illustrates the configuration of the Telemetry Core. On the left, a `config.json` file is shown with the following fields: `ServiceName`, `defaultTracerName`, `cleanupConfig`, `[spanProcessors]`, and `[logProcessors]`. An arrow points from this file to the **Telemetry Core** component. The **Telemetry Core** is a central box that receives configuration from several sources: `Default Tracer Name`, `Cleanup Config`, and `ServiceName` (all pointing to the Core). The Core then manages multiple `Span Processor` and `Log Processor` instances (all pointing from the Core to the processors). Ellipses between the processor boxes indicate multiple instances.

Figure 4: Illustration of the config being used

Description:

The config is a JSON file that contains all the information needed to initialize the telemetry system in Jolie.

It will set things like the service name, default tracer name and sampling rates. It will also set up where data is exported to and the way the cleanup is done when the Jolie instance is shut down.



Inputs

Name	Service name
Parameter name	serviceName
Type	String
Required	✓

Name	Default tracer name
Parameter name	defaultTracerName
Type	String
Required	✗
Default value	StandardTracer

Name	Cleanup config
Parameter name	cleanupConfig
Type	CleanupConfig
Required	✗
Default value	All default cleanup config

Name	Span processors
Parameter name	spanProcessors
Type	Processor[]
Required	✗
Default value	No span processors

Name	Log processors
Parameter name	logProcessors
Type	Processor[]
Required	✗
Default value	No log processors



Types:

CleanupConfig

Name	Timeout in seconds
Parameter name	timeoutInSeconds
Type	Integer
Required	×
Default value	30

Name	Enabling adding event
Parameter name	addEvent
Type	Boolean
Required	×
Default value	true

Name	Message when adding event
Parameter name	eventMessage
Type	String
Required	×
Default value	Jolie instance shutdown. Cleaning up spans.

Name	Enabling error status
Parameter name	setStatus
Type	Boolean
Required	×
Default value	true

Name	Enabling ending the span
Parameter name	endSpan
Type	Boolean
Required	×
Default value	true



DebugProcessor (Processor)

Name	The type of processor
Parameter name	processorType
Type	String
Required	debug

SimpleProcessor (Processor)

Name	The type of processor
Parameter name	processorType
Type	String
Required	simple

Name	The exporter of the processor
Parameter name	exporter
Type	Exporter
Required	✓



BatchProcessor (Processor)

Name	The type of processor
Parameter name	processorType
Type	String
Required	batch

Name	The exporter of the processor
Parameter name	exporter
Type	Exporter
Required	✓

Name	Max size of the export batch
Parameter name	maxExportBatchSize
Type	Integer
Required	✗
Default value	512

Name	Max size of the export queue
Parameter name	maxQueueSize
Type	Integer
Required	✗
Default value	2048

Name	Scheduled export interval
Parameter name	scheduleDelayInMs
Type	Integer
Required	✗
Default value	5000



Exporter

Name	The type of exporter
Parameter name	exporterType
Type	String (Enum): ["http", "grpc"]
Required	✓

Name	The URI of the OTLP endpoint to send to
Parameter name	endpoint
Type	String
Required	✓

Name	Headers to be added onto outgoing requests
Parameter name	headers
Type	Dict<String,String>
Required	✗
Default value	Void

Name	Time before timeout'ing when exporting a batch
Parameter name	timeoutInSeconds
Type	Integer
Required	✗
Default value	30

Name	Time before timeout'ing when connecting to endpoint
Parameter name	connectTimeoutInSeconds
Type	Integer
Required	✗
Default value	30

5.5 Attributes

```
1 attributes = "Ignored value" // Will not be included
2 bunchOfValues[0] = 8
3 bunchOfValues[1] = "Some information"
4 bunchOfValues[2] = 3.14159
5 bunchOfNumbers[0] = 2
6 bunchOfNumbers[1] = 4
7 bunchOfNumbers[2] << bunchOfValues // Everything but the first
  element will be ignored
8 attributes << {
9   ImportantNumber = 25
10  ImportantString = "Very important information"
11  CakeIsALie = true
12  ListOfValues << bunchOfValues
13  ListOfNumbers << bunchOfNumbers
14 }
```

Listing 20: Attributes set in Jolie.

```
1 {
2   "ImportantNumber": 25,
3   "ImportantString": "Very important information",
4   "CakeIsALie": true,
5   "ListOfValues": ["8", "Some information", "3.14159"],
6   "ListOfNumbers": [2, 4, 8]
7 }
```

Listing 21: Converted attributes.

Description

Attributes are key-value pairs that can be added onto multiple operations in the telemetry service. They are used to provide additional context to the spans and logs that are created.

Attributes can be any Jolie value, but they will be converted to OpenTelemetry compatible values. This means that only the top-level attributes will be sent on and any nested attributes will be ignored.

This will all be done automatically by the telemetry service. But its behavior is important to understand. An example of how attributes are set in Jolie and how they are converted can be seen in the code example above.

5.6 Embedding

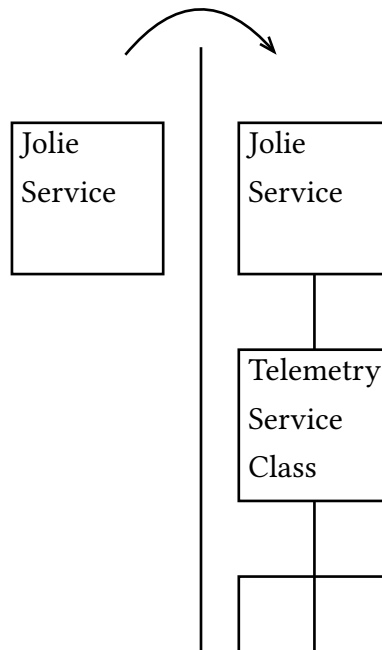


Figure 5: Embedding the Telemetry Service Class in a Jolie Service

Description

In order to use the telemetry service, it needs to be embedded into the Jolie service. Each embed will create a new instance of the telemetry service that will manage all the spans and processes for that service. But all of them will share the same configuration and telemetry system.

The first time a telemetry service is embedded, it will initialize the telemetry system from the config. But after that all telemetry services on the same Jolie instance will share the already initialized telemetry system. This means that the telemetry system will only be initialized once even if multiple telemetry services are embedded.

These multiple embeds can be useful when wanting to have different instrumentation name for different parts of the service. Example: Section 6.3.1

Code

```
1 embed Telemetry as telemetry Jolie
2 embed Telemetry as secondTelemetry
```

5.7 SetTracer

```
graph TD
    subgraph LeftState
        JS1[Jolie Service] -- newName --> TSC1[Telemetry Service Class]
        TSC1 --- DT[Default Tracer]
    end
    subgraph RightState
        JS2[Jolie Service] --- TSC2[Telemetry Service Class]
        TSC2 --- NT[newName Tracer]
    end
    LeftState --> RightState
```

Figure 6: setTracer(instrumentationName)()

Description

By default the telemetry service will use the default tracer set in the config. This operation will allow you to set a new tracer with your own instrumentation name. Example: Section 6.2.1

Code

```
1 SetTracer@telemetry("newName")()
```

Jolie

Input:

Name	New tracer name
Parameter name	None
Type	String
Required	✓

Output

Type	Void
------	------

5.8 StartProcess

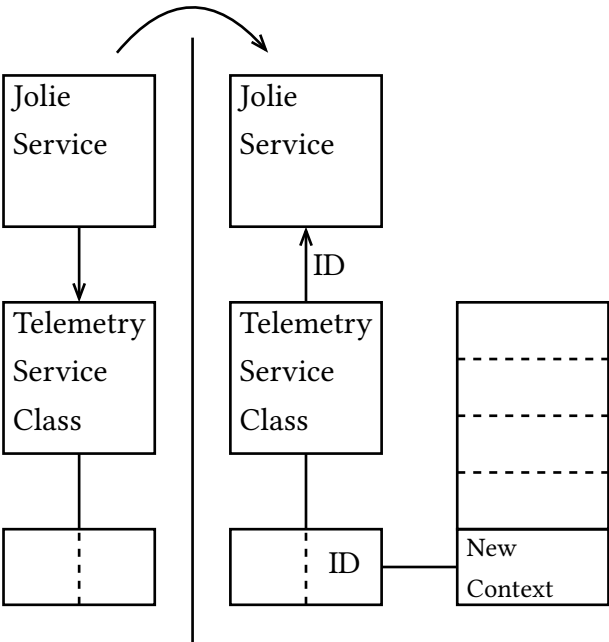


Figure 7: `startProcess()(id)` without propagation data

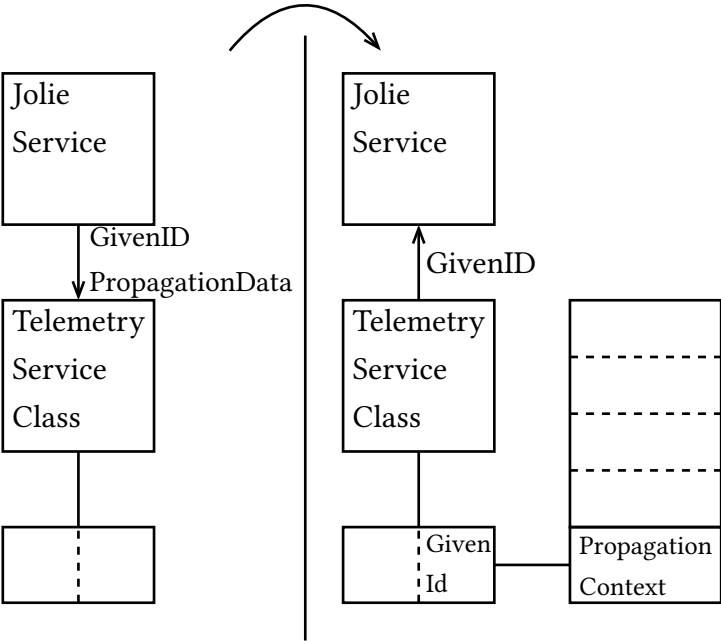


Figure 8: `startProcess(id?, propagationData?)(id)` with propagation data and Id



Description

In order to start capturing span data, a process needs to be started. This is done by the `startProcess` operation.

This process will be used to store the context and all the spans that are created during the execution of the lifetime of the process.

This process will be identified by an `Id` that is returned when the process is started. This `Id` will be used to refer to the process in other operations. Example: Section 6.2.1

This `ID` can either be system generated or a custom `ID` can be given. It is important that this custom `ID` is unique within that service during its lifetime.

When starting a process it can also continue the context of the caller. This means inheriting the `TraceId` and parent `SpanId` from the caller. This propagation data needs to be in the W3C Trace context format. If no propagation data is given the process will instead generate a new `TraceId`. Example: Section 6.2.2

Code

```
1 startProcess@telemetry() (ID) Jolie
1 startProcess@telemetry({id = "GivenId" propagationData = "PropagationData"})(GivenId) Jolie
```

Input

Name	Process ID
Parameter name	id
Type	String
Required	✗

Name	Propagation data
Parameter name	propagationData
Type	String (W3C Trace context)
Required	✗



Output

Name	Process ID
Parameter name	None
Type	String

Faults

- ExistingProcessTelemetryException

5.9 StartSpan

The diagram illustrates the `startSpan(id, spanName())` operation across two states of a system. A vertical line separates the initial state on the left from the final state on the right. In the initial state, a **Jolie Service** box has a downward arrow labeled `ID` and `newSpan` pointing to a **Telemetry Service Class** box. This class is connected to a **Tracer** box and a stack of three boxes: **FirstSpan**, **Context**, and an empty box. Below the stack is a box containing `ID`. In the final state, the **Jolie Service** box is shown without the arrow. The **Telemetry Service Class** box remains connected to the **Tracer** box and the stack. The stack now contains **NewSpan**, **FirstSpan**, and **Context**. The **Tracer** box has two arrows pointing to the **NewSpan** and **FirstSpan** boxes. Below the stack is a box containing `ID`. A curved arrow at the top indicates the transition from the initial state to the final state.

Figure 9: startSpan(id, spanName())

Description

This operation will start a new span in the process that the id refers to.

A span should be used to cover a specific workload being done within a process. The span should then be able to describe the workload and its context. Other operations can then be used to add more context to the span.

Spans will also inherit the context of the span that they are created within. This means it will build a tree of spans that can be used to understand the flow of the workload.

The spans in this package operate in a LIFO manner. Meaning you will only operate on the currently active span in the process and that will be the last span that was started.

Remember to end the span when the workload is done. Example: Section 6.2.1

Two icons are displayed at the bottom of the page: a document icon with a checkmark and a folder icon with an arrow pointing into it.

Code

```
1 startSpan@telemetry({id = "ID", spanName = "newSpan"})()
```

Jolie

Input

Name	Process ID
Parameter name	id
Type	String
Required	✓

Name	New span name
Parameter name	spanName
Type	String
Required	✓

Output

Type	Void
------	------

Faults

- ProcessNotFoundTelemetryException

5.10 GetPropagationData

The diagram illustrates the `getPropagationData(id)(propagationData)` operation. It shows two Jolie Service boxes. The left Jolie Service is connected to a Telemetry Service Class box, which is connected to a box containing ID, LastSpan, FirstSpan, and Context. The right Jolie Service is connected to a Telemetry Service Class box, which is connected to a box containing ID, LastSpan, FirstSpan, and Context. An arrow labeled 'W3C Trace context' points from the right Telemetry Service Class box to the right Jolie Service box. The arrow is labeled with 'Version-TraceId-SpanId-Flags'.

Figure 10: `getPropagationData(id)(propagationData)`

Description

This operation will get the propagation data for the currently active span for the process that the id refers to.

This propagation data will be in the W3C Trace context format. This means it will contain the TraceId and SpanId of the currently active span in the process.



Example: Section 6.2.2

This can then be used to continue the transaction in other services or systems.

Code

```
1 getPropagationData@telemetry("ID")(PropagationData)
```

Jolie



Input

Name	Process ID
Parameter name	None
Type	String
Required	✓

Output

Name	Propagation Data
Parameter name	None
Type	String (W3C Trace context)

Faults

- ProcessNotFoundTelemetryException
- SpanNotFoundTelemetryException

5.11 AddAttributes

The diagram illustrates the state of a Jolie Service and its Telemetry Service Class before and after the `addAttributes` operation. On the left, the Telemetry Service Class has a Context table with LastSpan, SecSpan, and FirstSpan, and an ID field. On the right, after the operation, the Context table has an additional Attributes column. An arrow indicates the transition from the left state to the right state.

Figure 11: `addAttributes(id, attributes)()`

Description

This operation will add attributes to the currently active span in the process that the id refers to. Example: Section 6.2.3

These attributes can be used to provide additional context to the span and its workload. See Section 5.5 for more information about how attributes are handled.

Code

```
1 addAttributes@telemetry({id = "ID" attributes =  
  Attributes})()
```

Jolie

Input

Name	Process ID
Parameter name	id
Type	String
Required	✓

Name	Attributes
Parameter name	attributes
Type	Undefined
Required	✓

Output

Type	Void
-------------	------

Faults

- ProcessNotFoundTelemetryException
- SpanNotFoundTelemetryException

5.12 AddEvent

The diagram illustrates the state of a Jolie Service and its Telemetry Service Class before and after an `addEvent` call. On the left, the Jolie Service is connected to the Telemetry Service Class via an ID. The Telemetry Service Class contains a LastSpan object, which is linked to a Context object. The Context object contains a table with 'First event' and 'Attributes'. The Telemetry Service Class also contains a table with 'Second event' and 'Attributes'. On the right, the Jolie Service is connected to the Telemetry Service Class via an ID. The Telemetry Service Class contains a LastSpan object, which is linked to a Context object. The Context object contains a table with 'First event' and 'Attributes'. The Telemetry Service Class also contains a table with 'New event' and 'Attributes'. A curved arrow indicates the transition from the left state to the right state.

Figure 12: `addEvent(id, eventName, attributes?)()`

Description

This operation will add an event to the currently active span in the process that the id refers to.



Events should be added when something of interest happens in the workload that the span covers. This can provide help when debugging or understanding the flow of the workload. Example: Section 6.2.3

Attributes can also be added to the event to provide additional context. See Section 5.5 for more information about how attributes are handled.

Code

```
1 addEvent@telemetry({id = "ID" eventName = "New event"  
  attributes = Attributes})()
```

Jolie



Input

Name	Process ID
Parameter name	id
Type	String
Required	✓

Name	Event name
Parameter name	eventName
Type	String
Required	✓

Name	Attributes
Parameter name	attributes
Type	Undefined
Required	✗

Output

Type	Void
-------------	------

Faults

- ProcessNotFoundTelemetryException
- SpanNotFoundTelemetryException

5.13 SetStatus

The diagram illustrates the `setStatus(id, status)` operation. It shows two states of a `Telemetry Service Class` associated with a `Jolie Service`.

Initial State (Left):

- `LastSpan`: (empty)
- `SecSpan`: (empty)
- `FirstSpan`: OK
- `Context`: (empty)

Final State (Right):

- `LastSpan`: ERROR
- `SecSpan`: (empty)
- `FirstSpan`: OK
- `Context`: (empty)

A curved arrow indicates the transition from the initial state to the final state, triggered by the `setStatus(id, status)` operation.

Figure 13: `setStatus(id, status)()`

Description

This operation will set the status of the currently active span in the process that the id refers to.



The status can be important to quickly understand the outcome of the workload that the span covers.

The “Error” status will as an example quickly indicate that something went wrong in the workload and that it should be investigated. Where as “OK” will show everything went well. Example: Section 6.2.3

Code

```
1 setStatus@telemetry({id = "ID" status = "ERROR"})()
```

Jolie



Input

Name	Process ID
Parameter name	id
Type	String
Required	✓

Name	Span status
Parameter name	status
Type	String (Enum): ["OK", "ok", "ERROR", "error"]
Required	✓

Output

Type	Void
-------------	------

Faults

- ProcessNotFoundTelemetryException
- SpanNotFoundTelemetryException

5.14 EndSpan

The diagram illustrates the `endSpan(ID)()` operation. It is divided into two parts by a vertical line, representing the state before and after the operation. On the left, a `Jolie Service` box has an arrow labeled `ID1` pointing to a `Telemetry Service Class` box. This class contains a `Context` box with `ID1` and `ID2` separated by a dashed line, and `FirstSpan` and `LastSpan` separated by a dashed line. The `LastSpan` is linked to a `Span Processor` box. On the right, the `Jolie Service` box is shown again, but the `Telemetry Service Class` box now only contains `ID1` and `ID2` in the `Context` box, and `FirstSpan` and `LastSpan` are no longer present. The `LastSpan` is now shown as a separate box, linked to the `Span Processor` box. The `Span Processor` box is shown with an arrow labeled `LastSpan` pointing to it. The `Span Processor` box is shown with an arrow labeled `LastSpan` pointing to it. The `Span Processor` box is shown with an arrow labeled `LastSpan` pointing to it.

Figure 14: endSpan(ID)()

Description

This operation will end the currently active span in the process that the id refers to.

Ending spans are important to do when the workload that the span covers is done. This will then prompt the telemetry service to send the span to the processors to then be exported. Example: Section 6.2.1

If spans are not ended they will not be sent to the processors until the Jolie instance is shut down and the cleanup is run.

This can lead to spans being left in the system for a long time, which can cause incorrect timing data to be sent to the processors.

In case of a catastrophic failure, the spans will not be ended at all and never sent to the processors.

Code

1 endSpan@telemetry("ID1")()

Jolie

Input

Name	Process ID
Parameter name	None
Type	String
Required	✓

Output

Type

Void

Faults

- ProcessNotFoundTelemetryException
- SpanNotFoundTelemetryException

5.15 EndProcess

The diagram illustrates the `endProcess(id)()` operation. On the left, a **Jolie Service** calls the **Telemetry Service Class** with `ID1`. The **Telemetry Service Class** then calls **Context** with `ID2`. The **Context** contains **LastSpan**, **FirstSpan**, and **Context**. The diagram shows the process being ended, with spans being sent to **Span Processor** units. The **Span Processor** units receive **LastSpan** and **FirstSpan** data. The diagram shows the process being ended, with spans being sent to **Span Processor** units. The **Span Processor** units receive **LastSpan** and **FirstSpan** data.

Figure 15: endProcess(id)() of active process.

Description

When finished with a process, it is important to end it. This will ensure that all spans in the process are ended and sent to the processors to be exported and process is removed from the telemetry service’s data map. Example: Section 6.2.1

This can be smart to run after a call is returned. More about how that can be done can be found in the Jolie docs: [Jolie docs](#). Examples of it done can be found here: Section 6.2.2

Code

```
1 endProcess@telemetry("ID1")()
```

Jolie

Input

Name

Process ID

Parameter name

None

Type

String

Required

✓

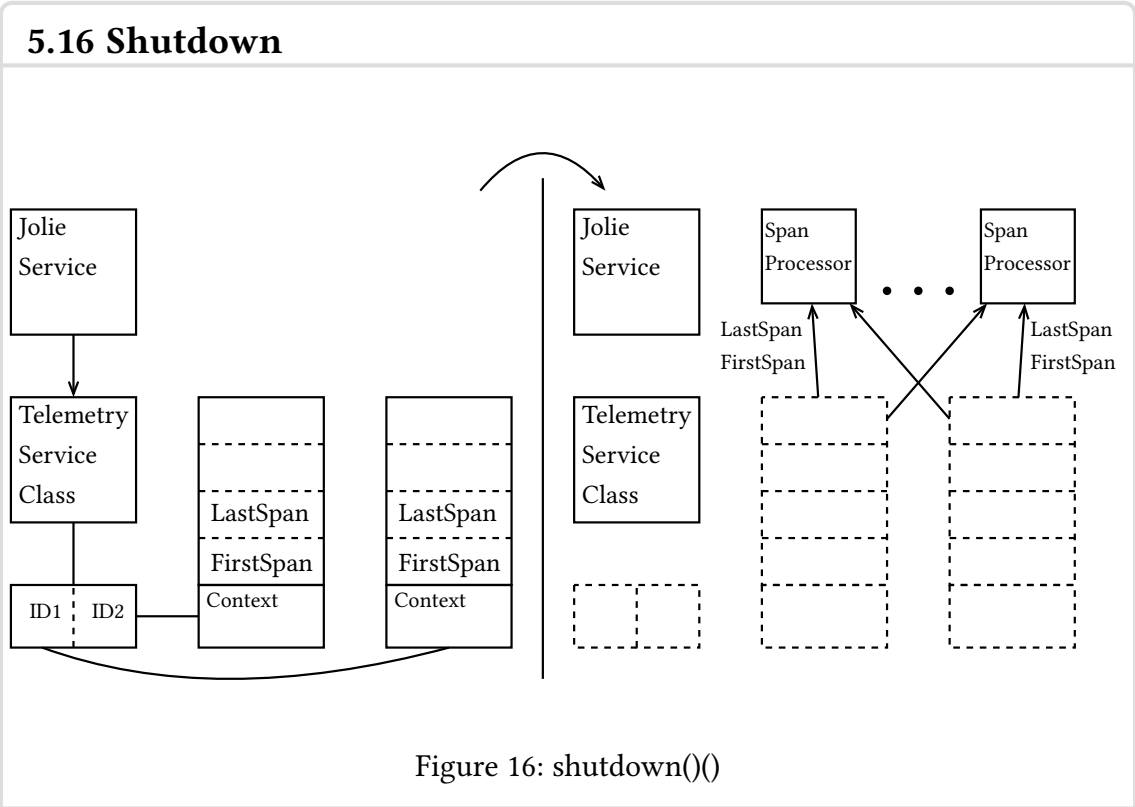
Output

Type

Void

Faults

- ProcessNotFoundTelemetryException



Description

This is to shut down the telemetry service once it is no longer needed. This will ensure that all spans are ended and sent to the processors to be exported and not waiting until the final shutdown of the Jolie instance. Example: Section 6.2.1

This will also clear the telemetry service from being tracked by the cleanup job.

It is smart to run this in single execution services where the telemetry service is not needed after the execution is done. This way the telemetry service will not hold onto any data that is not needed anymore.

Code

```
1 shutdown@telemetry()()
```

Jolie

Input

Type Void

Output

Type Void

5.17 Log

```
graph TD
    JS1[Jolie Service] -- "Message<br/>Raw Attributes?<br/>Severity?<br/>ID?" --> TSC1[Telemetry Service Class]
    TSC1 -- "Message<br/>Attributes?<br/>Severity?<br/>Context?" --> LP1[Log Processor]
    LP1 -- "Message<br/>Attributes?<br/>Severity?<br/>Context?" --> LP2[Log Processor]
    LP1 -- "LastSpan<br/>Context" --> TSC1
    TSC1 -- "ID" --> ID1[ID]
    ID1 -- "Context" --> C1[Context]
```

Figure 17: log(message, attributes?, severity?, id?)()

Description

This operation will log a message to the telemetry service. This is useful for logging messages that are not related to a specific process or span. Example: Section 6.2.3

But it is however possible to quickly attach the context of a currently active process to the log by providing the id of the process. This will then automatically add the TraceId and SpanId of the currently active span in that process to the log.

This logging can also have attributes added to it. This can be useful for providing additional context to the log message. See Section 5.5 for more information about how attributes are handled.

In order to quickly categorize the log message, a severity can also be set. This will allow for filtering of the logs based on their severity.

Code

1 log@telemetry({message = "Message"})()

1 log@telemetry({message = "Message" id = "ID" severity = "info" attributes = Attributes})()

Input

Name

Message content

Parameter name

message

Type

String

Required

✓

Name

Process ID

Parameter name

id

Type

String

Required

✗

Name

Severity of the log

Parameter name

severity

Type

String (Enum):
["TRACE", "DEBUG", "INFO", "WARN", "ERROR", "FATAL", "trace", "debug", "info", "warn", "error", "fatal"]

Required

✗

Name

Attributes

Parameter name

attributes

Type

Undefined

Required

✗

65 of 80

Output
<div>Type Void</div>
Faults <ul style="list-style-type: none">• ProcessNotFoundTelemetryException• SpanNotFoundTelemetryException

6 Examples

6.1 Config example

```
1  { JSON
2    "serviceName": "MyService",
3    "defaultTracerName": "my-tracer",
4    "spanProcessors": [
5      {
6        "processorType": "batch",
7        "maxExportBatchSize": 512,
8        "maxQueueSize": 2048,
9        "scheduleDelayInMs": 1000,
10       "exporter": {
11         "exporterType": "grpc",
12         "endpoint": "http://localhost:4317",
13         "timeoutInSeconds": 30,
14         "connectTimeoutInSeconds": 30,
15         "headers": {
16           "Authorization": "Bearer token"
17         }
18       }
19     },
20     {
21       "processorType": "debug"
22     }
23   ],
24   "sampler": {
25     "samplerType": "TraceIdRatioBased",
26     "ratio": 1
27   },
28   "cleanupConfig": {
29     "timeoutInSeconds": 10,
30     "addEvent": true,
31     "eventMessage": "Jolie instance shutdown. Cleaning up spans.",
32     "setError": true,
33     "endSpan": true
34   },
35   "logProcessors": [
36     {
37       "processorType": "simple",
38       "exporter": {
39         "exporterType": "http",
40         "endpoint": "http://localhost:4318/v1/logs"
41       }
42     },
43     {
44       "processorType": "debug"
45     }
46   ]
47 }
```

6.2 Simple examples

6.2.1 Simple process

Simple Process

```
1  service main( ) {
2      execution: single
3      embed Console as console
4      embed StringUtils as stringUtils
5      embed Telemetry as telemetry
6
7      init
8      {
9          setTracer@telemetry("simpleSingleExecutionExample")() // Sets the tracer name for
the telemetry service
10     }
11
12     main{
13         startProcess@telemetry()( identifier ) // Starts a new process and returns an
identifier for it
14         startSpan@telemetry({id = identifier, spanName = "Process start"})() // Starts the
first span of the process
15         // Workload
16         endSpan@telemetry(identifier)() // Ends the first span of the process
17         endProcess@telemetry(identifier)() // Ends the process and ends the span
18         shutdown@telemetry()() // Useful to shutdown and cleanup the memory for a single
execution service
19     }
20 }
```

6.2.2 Propagate context

Propagate context preparation

```
1  interface continueServiceAPI {
2      requestResponse:
3          operation( PropagationData )( void )
4  }
```

Propagate context main service

```
6  service main( ) { Jolie
7    execution: single
8    embed Telemetry as telemetry
9    embed continueService as continueService
10
11   init
12   {
13     setTracer@telemetry("PropagationExample")() // Sets the tracer name for the
        telemetry service
14   }
15
16   main{
17     startProcess@telemetry()( identifier ) // Starts a new process and returns an
        identifier for it
18     startSpan@telemetry({id = identifier, spanName = "Process start"})() // Starts the
        first span of the process
19     operation@continueService(getPropagationData@telemetry(identifier))() // Calls the
        continueService with the propagation data
20     endSpan@telemetry(identifier)( ) // Ends the first span of the process
21     endProcess@telemetry(identifier)( ) // Ends the process and ends the span
22     shutdown@telemetry()() // Useful to shutdown and cleanup the memory for a single
        execution service
23   }
24 }
```

Propagate context continue service

```
24 } Jolie
25
26 service continueService {
27   execution: sequential
28   embed Telemetry as telemetry
29
30   inputPort ip {
31     location: "local"
32     interfaces: continueServiceAPI
33   }
34
35   init
36   {
37     setTracer@telemetry("ContinueService")() // Sets the tracer name for the telemetry
        service
38   }
39
40   main{
41     [ operation( propData )() {
42       scope (mainScope) {
43         startProcess@telemetry({propagationData << propData})( identifier ) // Starts a
            new process with the propagation data
44         startSpan@telemetry({id = identifier, spanName = "ContinueService operation"})
            ( ) // Starts the first span of the process while continuing the propagation
            data
45         endSpan@telemetry(identifier)( )
46       }
47     } ]
48     {
49       endProcess@telemetry(identifier)( )
50     }
51   }
52 }
```

6.2.3 Attributes, Events and Logs

Propagate context continue service

```

1  service main( ) {
2      execution: single
3      embed Telemetry as telemetry
4      init
5      {
6          log@telemetry({message = "Telemetry service initialized" severity = "info"})() //
            Logs a message to the telemetry service
7          setTracer@telemetry("AttributesEventAndLogExamples")() // Sets the tracer name for
            the telemetry service
8      }
9
10     main{
11         // Declaring attributes
12         attributes = "Ignored value" // Will not be included
13         bunchOfValues[0] = 8
14         bunchOfValues[1] = "Some information"
15         bunchOfValues[2] = 3.14159
16         bunchOfNumbers[0] = 2
17         bunchOfNumbers[1] = 4
18         bunchOfNumbers[2] << bunchOfValues // Everything but the first element will be
            ignored
19         attributes << {
20             ImportantNumber = 25
21             ImportantString = "Very important information"
22             CakeIsALie = true
23             ListOfValues << bunchOfValues
24             ListOfNumbers << bunchOfNumbers
25         }
26
27         startProcess@telemetry()( identifier ) // Starts a new process and returns an
            identifier for it
28         startSpan@telemetry({id = identifier, spanName = "Process start"})() // Starts the
            first span of the process
29         addAttributes@telemetry({id = identifier, attributes << attributes})() // Adds
            attributes to the current span
30         addEvent@telemetry({id = identifier, eventName = "Event with attributes", attributes
            << attributes})() // Adds an event to the current span
31         // Workload
32         log@telemetry({message = "Log with both attributes and trace context", id =
            identifier, severity = "info", attributes << attributes})() // Logs a message to
            the telemetry service with attributes
33         setStatus@telemetry({id = identifier, status = "OK"})() // Sets the status of the
            current span
34         endSpan@telemetry(identifier)() // Ends the first span of the process
35         endProcess@telemetry(identifier)() // Ends the process and ends the span
36         shutdown@telemetry()() // Useful to shutdown and cleanup the memory for a single
            execution service
37     }
38 }

```

6.3 Advanced examples

6.3.1 Multiple Tracers

Multiple Tracers preparation

```
1  type multiplicationRequest {
2    a : int
3    b : int
4    propagationData : PropagationData
5  }
6
7  type powerRequest {
8    a : int
9    b : int
10   propagationData : PropagationData
11 }
12
13 interface MathServiceAPI {
14   requestResponse:
15     multiplication( multiplicationRequest )( int ),
16     power( powerRequest )( int )
17 }
18
```

Jolie

Multiple Tracers math service

```

19 service MathService {
20   execution: concurrent
21
22   embed Telemetry as simpleTelemetry
23   embed Telemetry as advancedTelemetry
24
25   inputPort ip {
26     location: "local"
27     interfaces: MathServiceAPI
28   }
29
30   init
31   {
32     setTracer@simpleTelemetry("SimpleMathService")() // Sets the tracer name for the
33     simple telemetry service
34     setTracer@advancedTelemetry("AdvancedMathService")() // Sets the tracer name for the
35     advanced telemetry service
36   }
37
38   main {
39     [ multiplication( request )( result ) {
40       scope (mainScope) {
41         startProcess@simpleTelemetry({propagationData << request.propagationData})
42         ( identifier )
43         startSpan@simpleTelemetry({id = identifier, spanName = "Multiplication"})()
44         log@simpleTelemetry({message = "Multiplication", id = identifier, severity =
45         "info", attributes << {a = request.a b = request.b}})()
46         result = request.a * request.b
47         endSpan@simpleTelemetry(identifier)()
48       }
49     } ] { endProcess@simpleTelemetry(identifier)()
50     }
51
52     [ power( request )( result ) {
53       scope (mainScope) {
54         startProcess@advancedTelemetry({propagationData << request.propagationData})
55         ( identifier )
56         startSpan@advancedTelemetry({id = identifier, spanName = "Power"})()
57         log@advancedTelemetry({message = "Power", id = identifier, severity = "info",
58         attributes << {a = request.a b = request.b}})()
59         multiplier = request.a
60         while (request.b > 1) {
61           request.a = request.a * multiplier
62           request.b = request.b - 1
63         }
64         result = request.a
65         endSpan@advancedTelemetry(identifier)()
66       }
67     } ] { endProcess@advancedTelemetry(identifier)()
68     }
69   }
70 }

```

Multiple Tracers main service

```

63 service main( ) {
64   execution: single
65   embed Telemetry as telemetry
66   embed MathService as mathService
67
68   init
69   {
70     setTracer@telemetry("MultipleTracersExample")() // Sets the tracer name for the
       telemetry service
71   }
72
73   main{
74     startProcess@telemetry()( identifier ) // Starts a new process and returns an
       identifier for it
75     startSpan@telemetry({id = identifier, spanName = "Main Process"})() // Starts the
       first span of the process
76     getPropagationData@telemetry(identifier)(propData) // Retrieves the propagation
       data for the current process
77
78     //Example of using MathService with propagation data
79     {(multiplication@mathService({
80       a = 5
81       b = 10
82       propagationData << propData
83     })(resultMultiplication))
84     |
85     {power@mathService({
86       a = 2
87       b = 3
88       propagationData << propData
89     })(resultPower)}}
90
91     log@telemetry({message = "Results", id = identifier, severity = "info", attributes
       << {multiplicationResult = resultMultiplication, powerResult = resultPower}})( )
92
93     endSpan@telemetry(identifier)( ) // Ends the first span of the process
94     endProcess@telemetry(identifier)( ) // Ends the process and ends the span
95     shutdown@telemetry()() // Useful to shutdown and cleanup the memory for a single
       execution service
96   }
97 }

```

7 Discussion and conclusion

7.1 Accomplishments

Throughout this project the focus has very deliberately been on OpenSource solutions and standards. OpenSource sets the standards across products and technologies and it allows multiple companies to use the same standards. Hence, this project has been centered around OpenTelemetry.

OpenTelemetry has only been growing with more and more companies joining in using and supporting it and therefore it makes sense for our solution to adopt this having the Jolie language join into a much larger enterprise adopted ecosystem. Any attempt to create a new standard would instead just isolate Jolie from larger adoption, ruining the goal of making Jolie compatible with currently running solutions and architecture.

This choice is also due to support and development efforts as Jolie is supported by volunteer efforts and not a large company and thereby using the community support for OpenTelemetry and their efforts in bug fixes and vulnerability patching.

This also explains the choice of using W3C trace context instead of other solutions such as B3 from Zipkin, because OpenTelemetry has chosen to use the W3C standard as the default in their propagation solution.

Using OpenTelemetry and their OTLP protocol has also allowed for coupling into more advanced components like the OpenTelemetry collector as an example. The collector allows for much more advanced and open source functions that would demand a lot more effort to include in the package. Take the internal processor features as an example. These features are but not limited to:

- Attribute filtering
- Span enrichment
- Tail based sampling
- Sensitive data cleanup

So by deploying this collector alongside the Jolie instance will allow for a much more advanced observability setup than a Jolie package in itself could offer. All while inheriting OpenTelemetry's vendor neutrality by using this middle component that is designed to be lightweight and secure.

7.2 Known issues

Throughout the last stages of this project a few problems came up.

One of these is the `PropagationData` type used in the package. It was originally intended to be a string with a REGEX filter on it to only accept the W3C trace context. However, this wasn't achieved. While Jolie supports it, it simply wouldn't work. A correct REGEX for the W3C trace context would reject both correct and incorrect inputs. So instead a primitive string was used in its place.

The other issue consists of two parts but regards error handling in general. The main thing being that if traces fail to be exported, the system has no clear way of communicating that.

Since the exporters are running in a separate thread and are not sent to the console they will instead fail silently. And this ties into the second part being a lack of testing of edge cases. Due to a limited timeframe on this project, a proper test phase with other participants was unachievable leading to the last part of the discussion

7.3 Future work

This project may be finished, but the effort is far from. This project has managed to achieve 2 out of the 3 corner stones of observability and more further work on refining it in the future will be beneficial.

7.3.1 New features

For new features metrics would be the next obvious choice to add to this, allowing a full observability implementation. While it is under the observability umbrella it does, however, require a whole new part to be added. Metrics would have to be able to pull a lot of internal data from Jolie internals to get the health and usage information.

Another interesting addition to this project would be a more automatic solution. While the current solutions rely solely on the developer to add all instrumentation to their code, a solution doing this automatically would add to its plug and play potential. This could for example be logging everything happening in the console or automatic tracing of all outgoing calls.

7.3.2 Jolie integration

While this project has already used a deep integration into how Jolie's internals work, further work would be very interesting.

At the current time of this project no hooks or methods to observe when a call to Jolie finishes or crashes exists. Instead the project relies on the shutdown hook of the whole JVM.

An interesting addition would then be the creation of a “finish” block doing the opposite of the init block or a finish hook that a package could add code to that would then be executed at shutdown or crashes of individual services.

This would allow for a more immediate evacuation of potentially important data as to what caused the crashes.

7.3.3 Testing

While this project aims to provide an easy and understandable interface that’s easy to use, it has only been tested while being developed and had few outside eyes on it.

Having a proper user acceptance testing phase would provide a better insight into how a more general user base would use this package and how intuitive both the package itself and the documentation is.

While unit testing hasn’t really been relevant for this project due to its reliance on external systems, a deep dive into its reliance and performance under load would be interesting. This could provide a better insight into the overhead of the package and its limitations.

However, more needs and work could be discovered during this.

7.4 Conclusion

This thesis addresses the previous lack of observability in Jolie programming language, by creating a standalone observability package that integrates with OpenTelemetry standards. The contributions being:

- **Research:** A deep exploration of existing observability technologies, frameworks, and standards, leading to the selection of OpenTelemetry and W3C trace context for integration with Jolie.
- **Implementation:** Creation of an importable and published package for Jolie that now allows for distributed logging and tracing, making it possible to monitor and understand distributed applications written fully in Jolie or in a mixed environment with other languages through the W3C trace context standard.
- **Usability:** Through joining the OpenTelemetry ecosystem, the package can be used in addition with existing tools and frameworks, allowing utilization of the OpenTelemetry collector and other OTLP-compatible systems. This allows access to a more advanced observability setup.
- **Documentation:** Comprehensive documentation and illustrations to help developers understand how to use the package and what happens when they use different operations. Further backed by examples showing how the addition of the package can be structured into existing code.

This project has successfully achieved its goals and now has a published package free for anyone to use. It provides a solid foundation for observability in Jolie already allowing for distributed logging and tracing, while also being ready to be extended in the future with more features and improvements.

Bibliography

- [1] Google, “Google trends.” Accessed: May 23, 2025. [Online]. Available: <https://trends.google.com/trends>
- [2] OpenTelemetry, “OpenTelemetry Documentation.” Accessed: May 23, 2025. [Online]. Available: <https://opentelemetry.io/docs>
- [3] OpenTelemetry, “OpenTelemetry github.” Accessed: May 23, 2025. [Online]. Available: <https://github.com/open-telemetry>
- [4] Yuri Shkuro, “Introducing native support for OpenTelemetry in Jaeger.” Accessed: May 23, 2025. [Online]. Available: <https://medium.com/jaegertracing/introducing-native-support-for-opentelemetry-in-jaeger-eb661be8183c>
- [5] Dynatrace, “OpenTelemetry and Dynatrace.” Accessed: May 23, 2025. [Online]. Available: <https://docs.dynatrace.com/docs/ingest-from/opentelemetry>
- [6] Sentry, “Code-level application monitoring for OpenTelemetry.” Accessed: May 23, 2025. [Online]. Available: <https://sentry.io/for/opentelemetry/>
- [7] Grafana, “Send data to the Grafana Cloud OTLP endpoint.” Accessed: May 23, 2025. [Online]. Available: <https://grafana.com/docs/grafana-cloud/send-data/otlp/send-data-otlp>
- [8] Prometheus, “Using Prometheus as your OpenTelemetry backend.” Accessed: May 23, 2025. [Online]. Available: <https://prometheus.io/docs/guides/opentelemetry>
- [9] Jolie, “Jolie, The service-oriented programming language.” Accessed: May 23, 2025. [Online]. Available: <https://www.jolie-lang.org/index.html>
- [10] Oracle, “Java SE 21 Archive Downloads.” Accessed: May 23, 2025. [Online]. Available: <https://www.oracle.com/java/technologies/javase/jdk21-archive-downloads.html>
- [11] Jolie, “Jolie documentation.” Accessed: May 23, 2025. [Online]. Available: <https://docs.jolie-lang.org/>
- [12] Maven, “Maven repository.” Accessed: May 23, 2025. [Online]. Available: <https://mvnrepository.com/>
- [13] yevgenp, “Why Use ArrayDeque Instead of Stack in Java.” Accessed: May 23, 2025. [Online]. Available: <https://yevgenp.medium.com/why-use-arraydeque-instead-of-stack-in-java-02e92eb86e4c>

- [14] W3C, “Trace Context Level 2.” Accessed: May 23, 2025. [Online]. Available: <https://www.w3.org/TR/trace-context-2/>