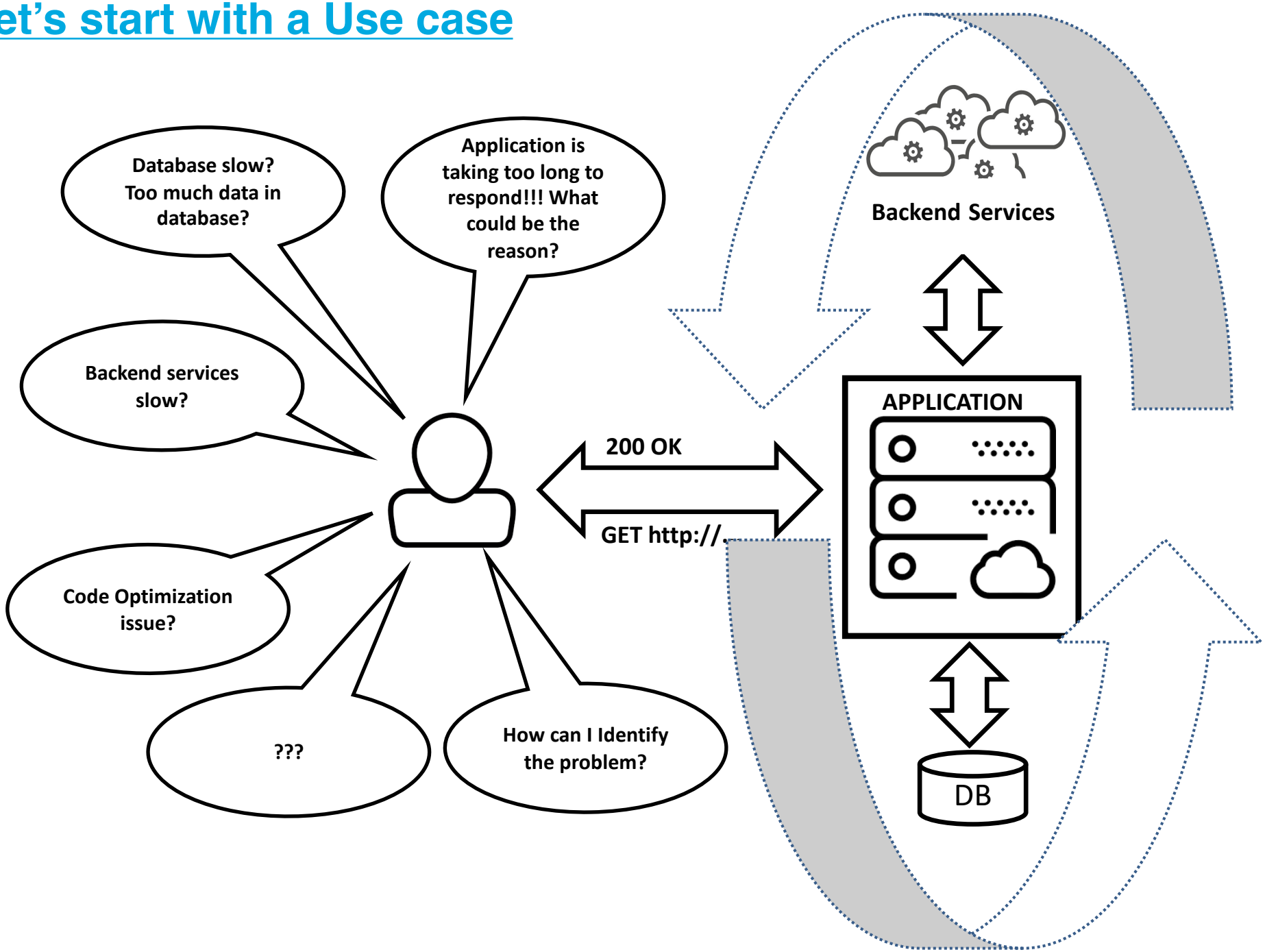
# Distributed Tracing Using OpenTelemetry

# Agenda

- What is OpenTelemetry
- OpenTelemetry Concepts
- Automatic Instrumentation & Demo
- Manual Instrumentation & Demo
- Recommendations
- Q&A

# What is OpenTelemetry?

# Let's start with a Use case



# Introduction

- OpenTelemetry, also known as OTel for short, is an opensource observability framework for instrumenting, generating, collecting, and exporting telemetry data such as traces, metrics, logs. As an industry-standard, it is natively supported by a number of vendors.

- OpenTelemetry standardizes the way telemetry data is collected and transmitted to backend platforms (e.g. Jaeger , Zipkin etc..).

- OpenTelemetry code instrumentation is supported for many popular programming languages like Java, Python, Go, C++, .NET etc..

- Depending on the programming language, below options are supported by telemetry:

  - Automatic instrumentation – No or Minimal code changes are required (For example, In case of java, agent to be attached)

  - Manual instrumentation – Involved code changes (OpenTelemetry SDK to be installed)

Note: OpenTelemetry is NOT an observability back-end like Jaeger or Prometheus. Instead, it supports exporting data to a variety of open-source and commercial back-ends.

# History

- OpenTelemetry is the combination of two open source distributed tracing projects, OpenTracing and OpenCensus, merged into a single project.
- OpenTracing, hosted by the Cloud Native Computing Foundation (CNCF), was an effort to provide a standardized API for tracing.
- OpenCensus, on the other hand, was developed by Google based on its internal tracing platform. It was eventually open-sourced, and Microsoft, along with other vendors and contributors, got involved and began evolving the standard.
- OpenTelemetry came about when the two projects agreed to merge the codebases of the OpenTracing and OpenCensus projects, incorporating the strengths of each under CNCF control.

# Benefits

OTel has broad industry support and adoption from cloud providers, vendors and end users. It provides you with:

- **Consistency and vendor neutrality:** OpenTelemetry provides a consistent, vendor-neutral way to instrument applications and systems, making it easier for developers to work with multiple tools and technologies.

- **Ease of use:** OpenTelemetry provides a set of APIs and libraries that are easy for developers to use, which can help them quickly and easily instrument their applications.

- **Rich data:** OpenTelemetry provides a comprehensive set of APIs and libraries for collecting a wide range of telemetry data, including request latencies, error rates, resource utilization, and more. This can help developers and organizations get a more complete picture of the performance and behavior of their applications and systems.

- **Scalability:** OpenTelemetry is designed to be scalable and can handle high volumes of telemetry data, making it suitable for use in large, distributed systems.

- **Integration with other tools:** OpenTelemetry provides integration with a wide range of monitoring, logging, and analytics tools, making it easier for developers and organizations to work with the data collected by OpenTelemetry.

- **Cost-effectiveness:** OpenTelemetry is free and open source.

# Potential Use Cases of OpenTelemetry

- **Monitoring the health and performance of applications and systems:** Telemetry data collected with OpenTelemetry can be used to monitor the health and performance of applications and systems in real-time, helping organizations identify and fix issues before they become critical.

- **Troubleshooting Issues:** Telemetry data can be used to troubleshoot issues that arise in applications and systems, such as identifying the root cause of an error or identifying performance bottlenecks

- **Capacity Planning:** Telemetry data can be used to understand the resource utilization of applications and systems, which can help organizations plan for capacity and scale their systems as needed

- **Optimization:** Telemetry data can be used to identify opportunities for optimization, such as identifying areas where applications or systems are underperforming or where resources are being wasted

# Opentelemetry and AI

- OpenTelemetry can feed collected data into an AI engine to automatically produce actionable insights. As an example, the AI engine can continuously analyze data that OpenTelemetry captures along with other useful and desirable data and look for anomalies throughout the full stack without any human intervention.

- It can also automatically identify the source of issues and, when possible or desirable, fix them before they impact end users.

- Through this continuous process, the AI will learn what the “normal” state of the system is and adapt its responses to improve performance over time, including the potential for predicting problems before they arise.

- Essentially, AI integration increases the value of OpenTelemetry by reducing the manual effort behind using observability to analyze conditions and making it easier to get actionable insights.

# Language Support

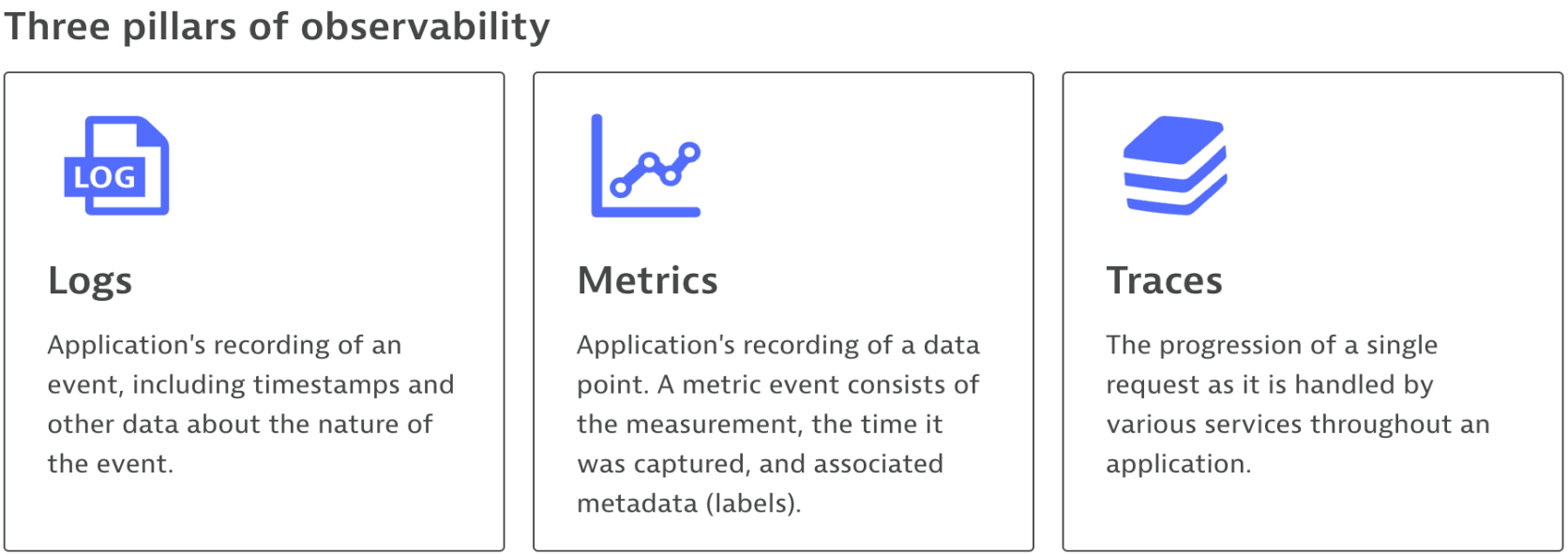
| Language | Traces | Metrics | Logs |
|---|---|---|---|
| C++ | Stable | Stable | Experimental |
| C#/.NET | Stable | Stable | Mixed* |
| Erlang/Elixir | Stable | Experimental | Experimental |
| Go | Stable | Beta | Not yet implemented |
| Java | Stable | Stable | Stable |
| JavaScript | Stable | Stable | Development |
| PHP | Beta | Beta | Alpha |
| Python | Stable | Stable | Experimental |
| Ruby | Stable | Not yet implemented | Not yet implemented |
| Rust | Beta | Alpha | Not yet implemented |
| Swift | Stable | Experimental | In development |

# OpenTelemetry Concepts

# 1. Three pillars of open telemetry

- OpenTelemetry specification is organized into distinct types of telemetry, which we call signals. The primary signal is tracing. Logs and metrics are other examples.

  **Traces** – "Where is the problem?"
  **Metrics** – "Is there a problem?"
  **Logs** – "What is the problem?"



# 2. OTLP Protocol

- OpenTelemetry defines a vendor and tool agnostic protocol specification called OTLP (OpenTelemetry Protocol) for transmitting traces, metrics and logs telemetry data. With that in place, replacing a backend analysis tool would be as easy as a configuration change on the collector.

- OTLP can be used for transmitting telemetry data from the SDK to the Collector, as well as from the Collector to the backend tool of choice. The OTLP specification defines the encoding, transport, and delivery mechanism for the data, and is the future-proof choice.

- Some systems may, however, be using third party tools and frameworks, which may come with built-in instrumentation other than OTLP, such as Zipkin or Jaeger formats. OpenTelemetry Collector can ingest these other protocols using appropriate Receivers as mentioned above.

# 3. Collector

The Collector's job is to process, filter, aggregate the telemetry data, giving developers greater flexibility for receiving, shaping, and sending data to multiple back ends.

It have 3 components:

- Receivers are in charge of  the applications' signals by listening for calls on particular ports on the Collector. They work with both gRPC and HTTP protocols. A complete list of receivers for specific scenarios or frameworks can be found on GitHub.

- Processors sit between receivers and exporters; they enable us to shape the data by filtering, formatting, and enriching it before it goes through the exporter to a back end. Common use cases include data sanitization to remove sensitive or private information, exporting metrics from spans, or deciding which signals are saved to the back end.

- Exporters can push data into one or multiple configured back ends or destinations. They work by transforming the data into a different format if needed and sending it to the endpoint defined. Popular exporters include Jaeger, Prometheus, and Zipkin

# 4. Traces

- A trace represents the flow of a single transaction or request as it goes through the system.

- They provide a holistic view of the chain of events triggered by requests and are defined by a tree of nested spans — one for each unit of work they represent and a parent span

Sample trace:

```
{
  "traceID": "81289be65e00618d84366dfe2f7fc1a2",
  "spans": [
    {
      "traceID": "81289be65e00618d84366dfe2f7fc1a2",
      "spanID": "e03e8cca690f81c1",
      "operationName": "read_json_from_file",
    }
    // ...
    {
      "traceID": "81289be65e00618d84366dfe2f7fc1a2",
      "spanID": "75473187e1bc7579",
      "operationName": "word-by-language"
      // ...
    },
    {
      "traceID": "81289be65e00618d84366dfe2f7fc1a2",
      "spanID": "45c0d587ebdddf60",
      "operationName": "/words"
      // ...
    }
}
```

# 5. Spans

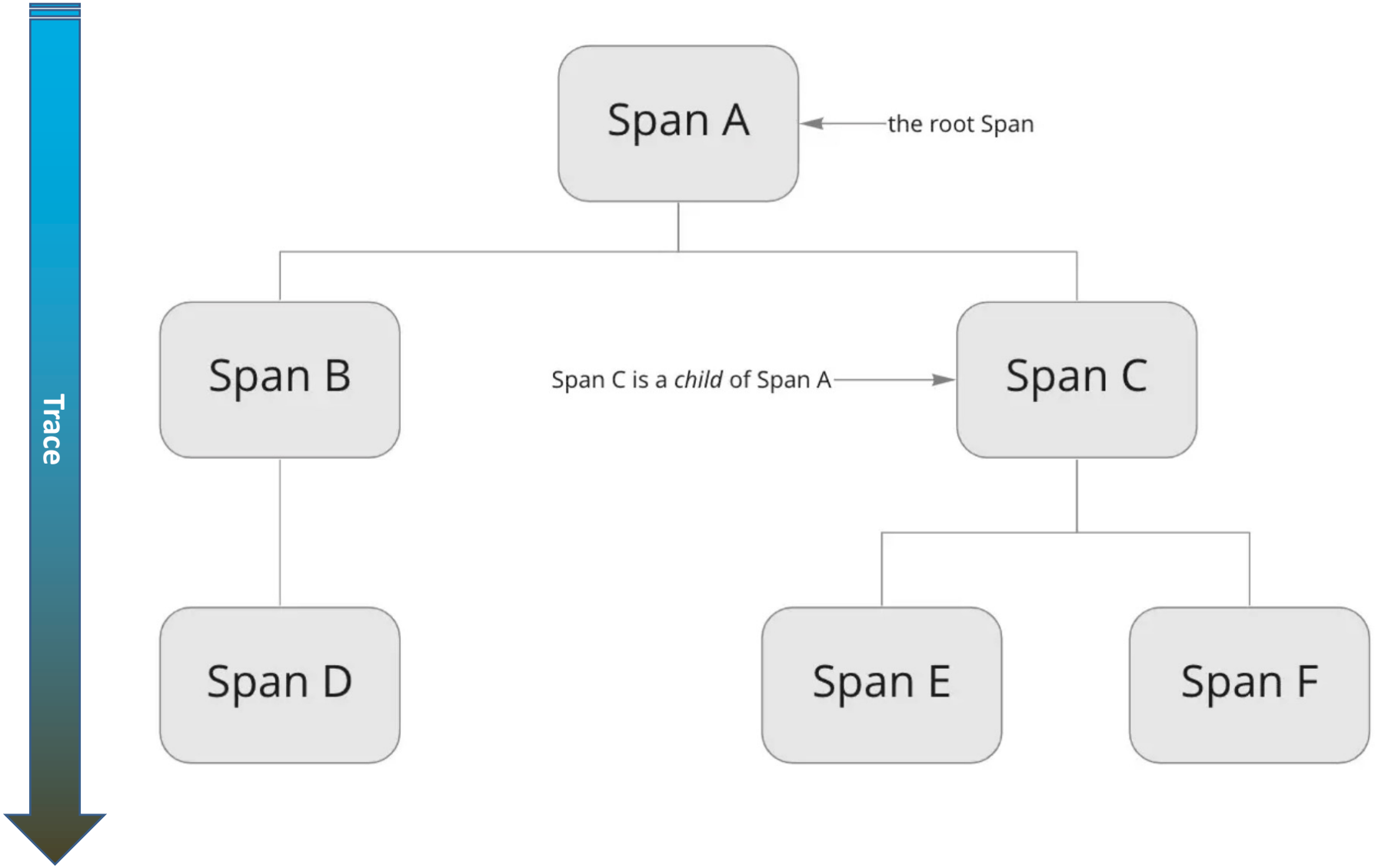
- A span represents a unit of work or operation. Spans are the building blocks of Traces. In OpenTelemetry, they include the following information:

  - Name
  - Start and End Timestamps
  - Span Context
  - Span Attributes are key-value pairs added to a span to help analyze the trace data] etc…

Sample Span:


```
{
  "trace_id": "7bba9f33312b3dbb8b2c2c62bb7abe2d",
  "span_id": "086e83747d0e381e",
  "name": "/v1/sys/health",
  "start_time": "2021-10-22 16:04:01.209458162 +0000 UTC",
  "end_time": "2021-10-22 16:04:01.209514132 +0000 UTC",
  "status_code": "STATUS_CODE_OK",
  "status_message": "",
  "attributes": {
    "net.transport": "IP.TCP",
    "net.peer.ip": "172.17.0.1",
    "net.peer.port": "51820",
    "net.host.ip": "10.177.2.152",
    ......
  }
}
```

# Trace-Span Relation



# 6. Metrics

- A metric is a series of data points with a value associated with timestamps, which has led to the word “timeseries” to be largely considered a synonym for “metrics”. The value of data points are often numeric, e.g., the count of how many requests served within a certain timeframe, but in some monitoring systems, it can also be strings (e.g., the “INFO” metrics of Prometheus) or Booleans.

- In order to reduce the amount of computing resources to store and process metrics over long timeframes, it is common practice to “aggregate” their values, for example reducing the granularity of a metrics from having one data point every second, to storing the average, mean and (in some cases, percentiles) of data points over a minute or ten.

# 7. Logs

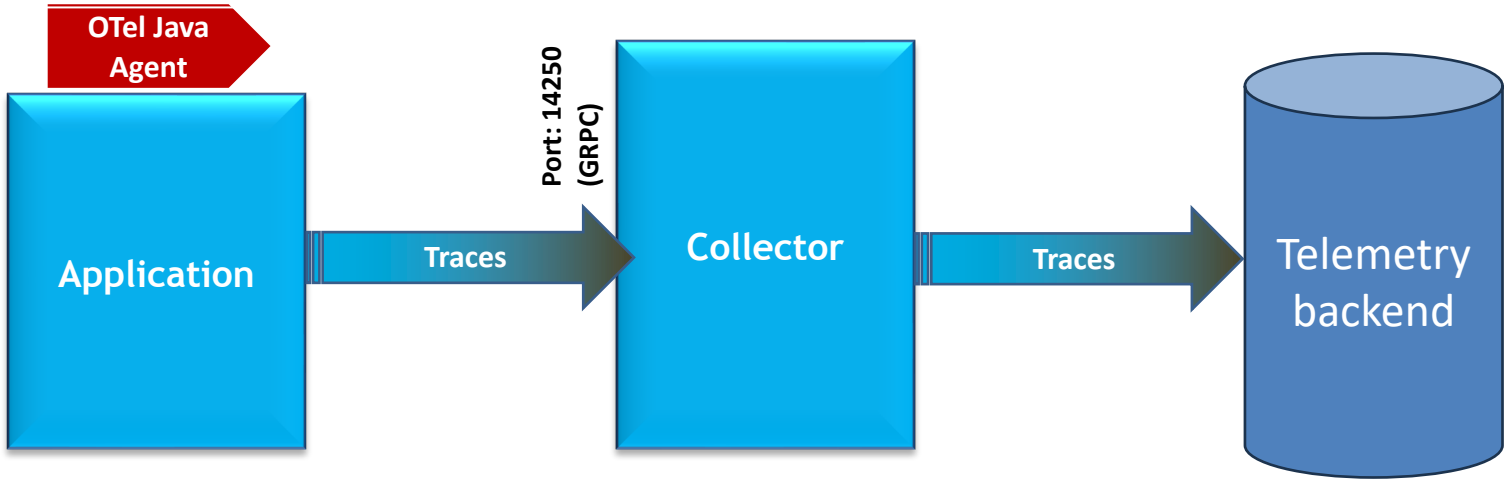
- A log is a textual record of a specific event that occurred at a specific point in time. The trigger to generate the log entry is part of the code of the application, so systems produce log entries repeatedly when the relative code is executed. The entry records the time of the event and provides a payload including a message to describe the nature of the event, context about that event, and additionally other metadata that can be useful later for analysis.

- Depending to how logs are created, which formatting rules are used, and how easy it is for automated logic to process them, logs can be broadly categorized as follows:

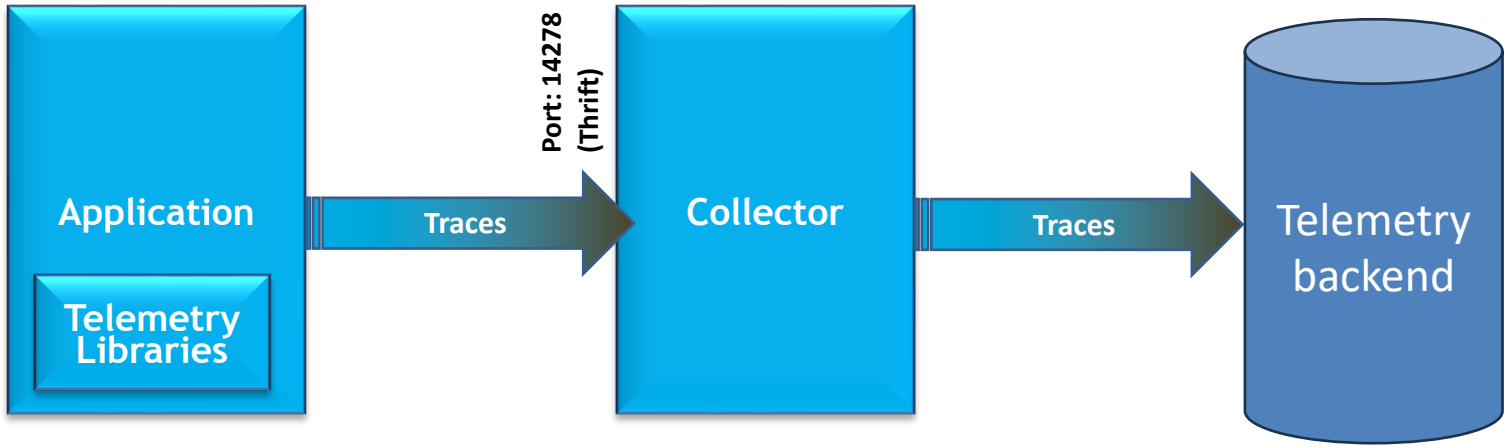
  **Unstructured logs**—includes text that is written for humans to process, and it may not include metadata that is easy to process for machines. It is generally considered the most common approach to logging, and unfortunately it is usually hard to parse for analysis.

  **Structured logs**—includes data organized into a standard format with a structure that is easy to parse for other code (e.g., JSON). It includes additional metadata that makes querying (especially filtering and grouping) logs easier.

## Automatic Instrumentation




## Manual Instrumentation



# Automatic Instrumentation

# Overview

- OpenTelemetry auto instrumentation libraries are the best option for someone who doesn't want to modify their application code for generating telemetry data(logs, metrics, and traces).
- Automatic instrumentation uses telemetry agents for generating telemetry data. For example, Java uses a Java agent JAR that can be attached to any Java 8+ application.
- It dynamically injects bytecode to capture telemetry from many popular libraries and frameworks.
- It can be used to capture telemetry data at the “edges” of an app or service, such as inbound requests, outbound HTTP calls, database calls, and so on

# Steps Involved for Spring Boot Application

1. Download opentelemetry-javaagent.jar from Releases of the opentelemetry-java-instrumentation repo. The JAR file contains the agent and all automatic instrumentation packages. We can also download the same from Maven repository

```
<dependency>
<groupId>io.opentelemetry.javaagent</groupId>
<artifactId>opentelemetry-javaagent</artifactId>
</dependency>
```

2. Configure the agent and exporters. Configuration can be done in different ways and next slide will cover the same.

3. Run your **application with agent**: java -javaagent:path/to/opentelemetry-javaagent.jar -jar ./build/libs/java-simple.jar and note the output

# Ways To Configure Java Agent

Agent is highly configurable and support below options:

1. Pass the agent jar file as a command line arguments

```
java -javaagent:path/to/opentelemetry-javaagent.jar
-Dotel.service.name=your-service-name
-Dotel.traces.exporter= jaeger
-jar myapp.jar
```

2. Use environment variables to configure the agent:

```
OTEL_SERVICE_NAME=your-service-name
OTEL_TRACES_EXPORTER=jaeger
java -javaagent:path/to/opentelemetry-javaagent.jar  -jar myapp.jar
```

3. Supply Java properties file and load configuration values from there:

```
java -javaagent:path/to/opentelemetry-javaagent.jar
Dotel.javaagent.configuration-file=path/to/properties/file.properties
-jar myapp.jar
```

# Exporter Configuration

- In order to visualize and analyse your traces and metrics, you will need to export them to a backend (Jaeger, Zipkin etc… )
- Exporters can be configured in different ways and most common ways are using environment variables and system properties
- Sample Jaeger exporter configuration is shown below:

| System property | Environment variable | Description |
|---|---|---|
| otel.traces.exporter=jaeger | OTEL_TRACES_EXPORTER=jaeger | Select the Jaeger exporter |
| otel.exporter.jaeger.endpoint | OTEL_EXPORTER_JAEGER_ENDPOINT | The Jaeger gRPC endpoint to connect to. Default is http://localhost:14250. |
| otel.exporter.jaeger.timeout | OTEL_EXPORTER_JAEGER_TIMEOUT | The maximum waiting time, in milliseconds, allowed to send each batch. Default is 10000. |

To see the full range of exporter configuration options, refer Exporter Configuration.

# Annotations

1. For most users, the out-of-the-box instrumentation is completely sufficient, and nothing more has to be done. Sometimes, however, users wish to create spans for their own custom code without doing too much code change.

2. We need to add a dependency on the opentelemetry-instrumentation-annotations library to use the @WithSpan annotation.

```
<dependency>
<groupId>io.opentelemetry.instrumentation</groupId>
<artifactId>opentelemetry-instrumentation-annotations</artifactId>
</dependency>
```

3. To create a span corresponding to one of your method, annotate the method with @WithSpan.

```
import io.opentelemetry.instrumentation.annotations.WithSpan;

public class MyClass
{ @WithSpan
public void myMethod() { <...> } }
```

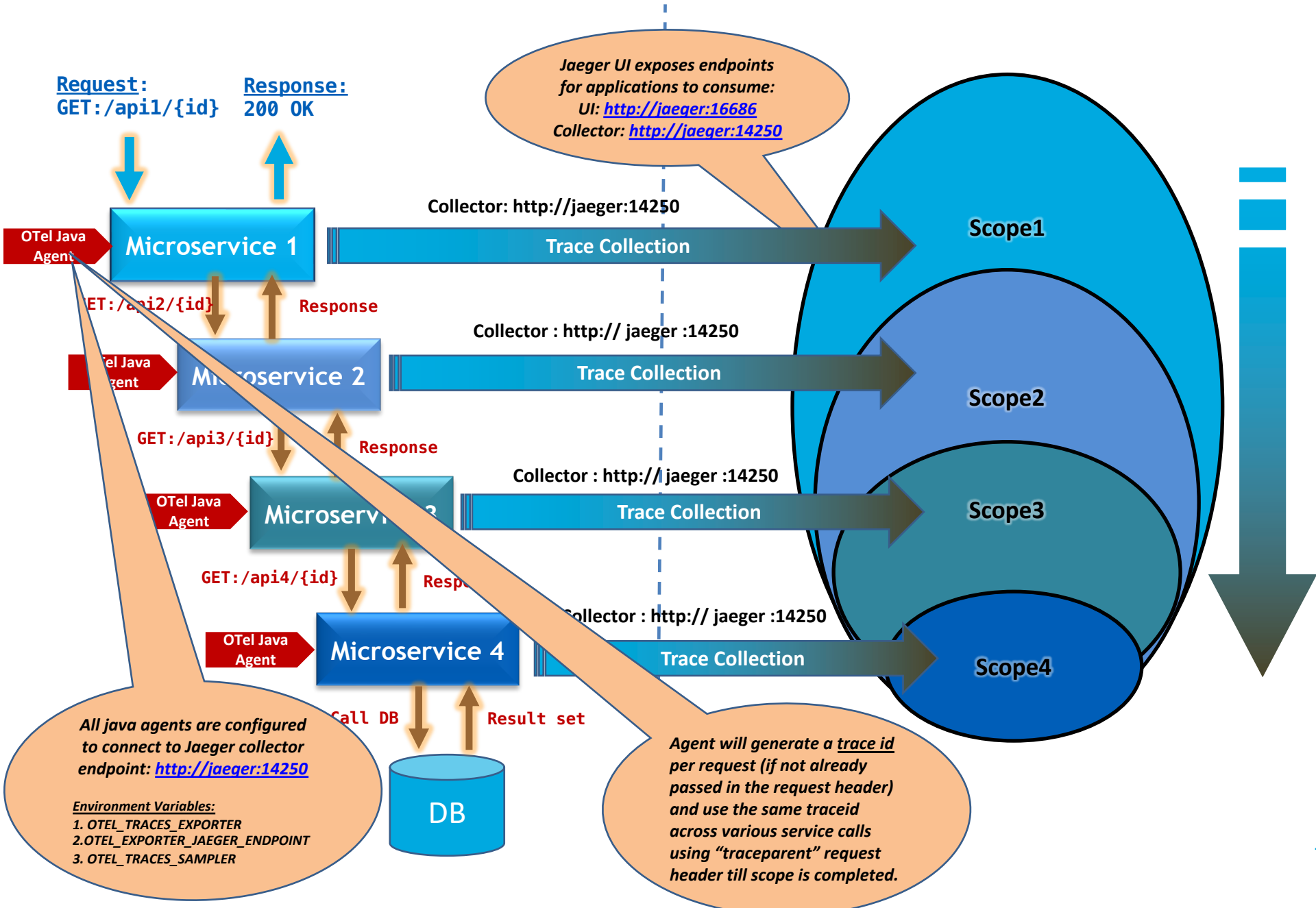

**Scenario 1: Happy Path / No Errors**

**Scenario 2: Adding New Span to the trace**

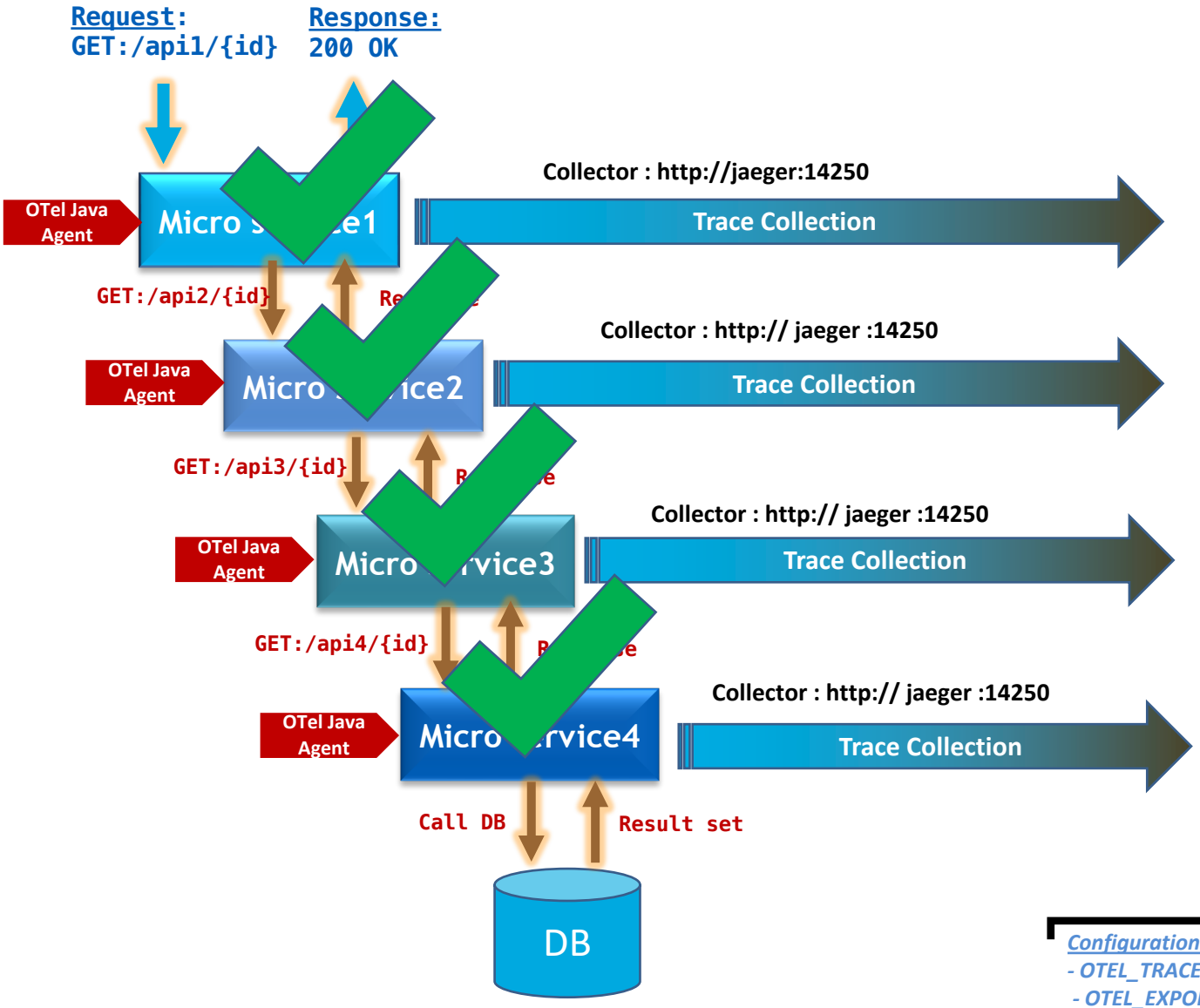
**Scenario 3: Error/Exception in Microservice**

**Scenario 4: Disabling traces in Microservice**

# Open Telemetry Automatic Instrumentation



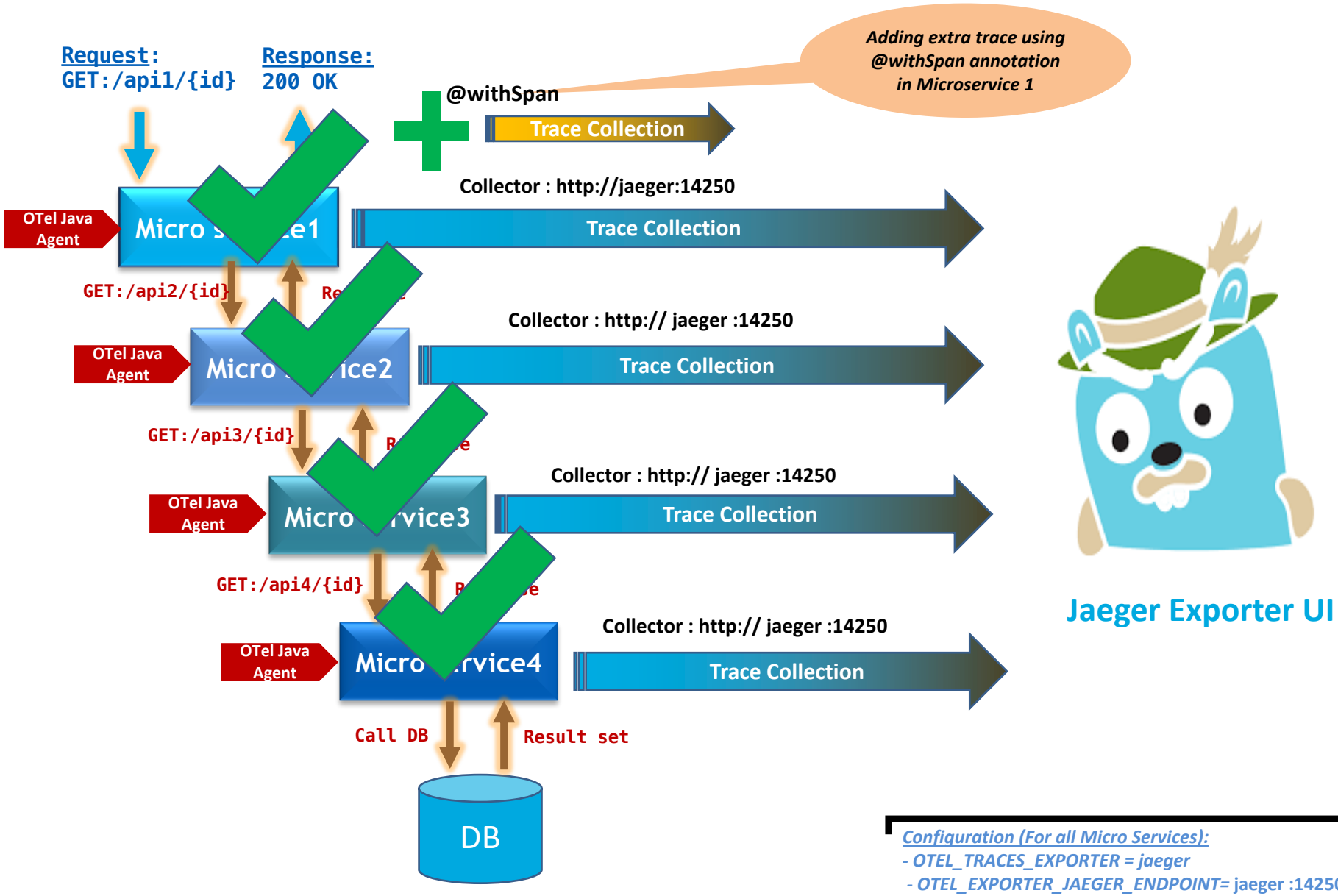
# Scenario 1: Happy Path / No Errors




Jaeger Exporter UI

*Configuration (For all Micro Services):*
- *OTEL_TRACES_EXPORTER = jaeger*
- *OTEL_EXPORTER_JAEGER_ENDPOINT= jaeger :14250*
- *OTEL_TRACES_SAMPLER=always_on*

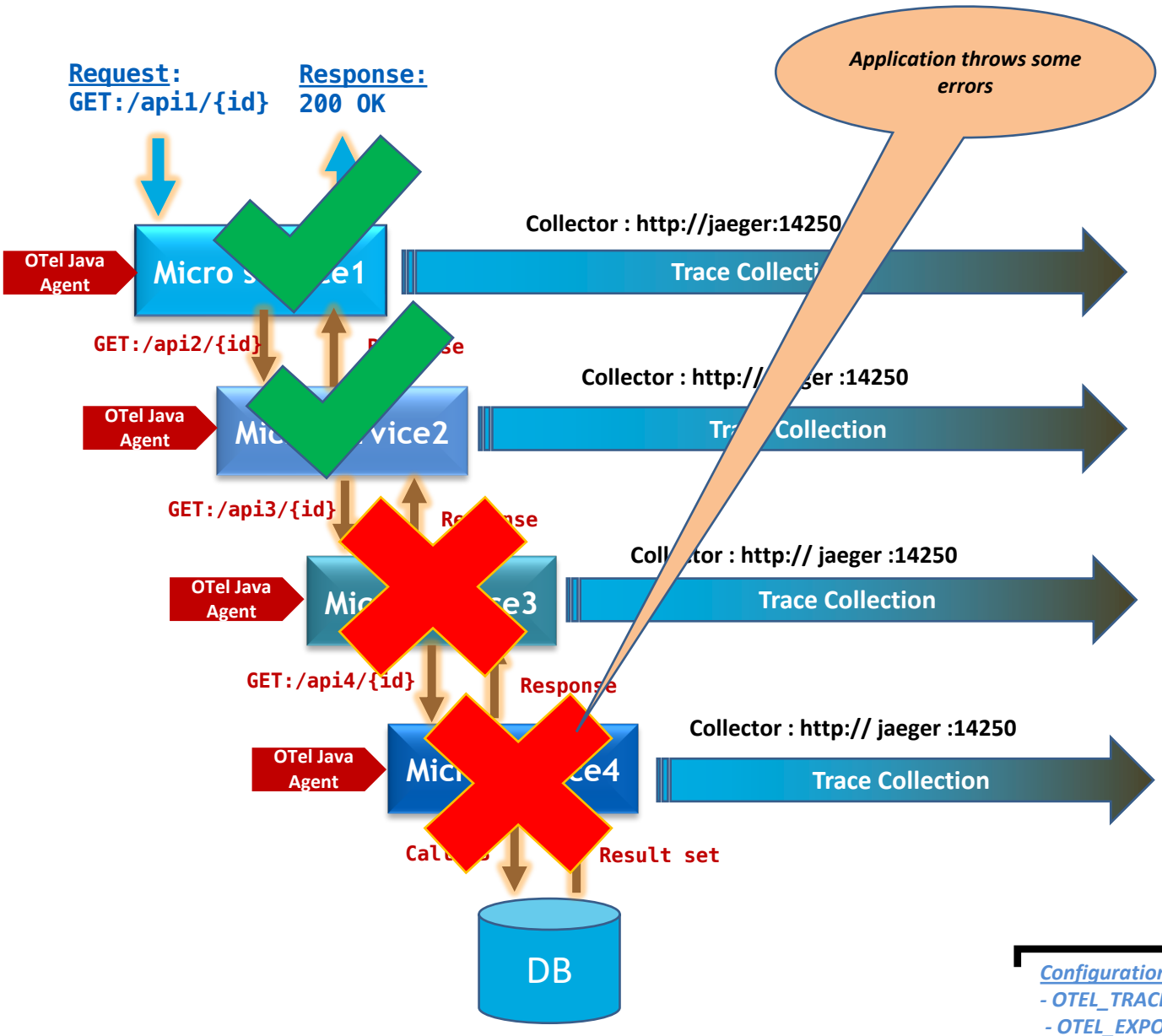
# Scenario 2: Adding New Span To The Trace




*Configuration (For all Micro Services):*

- *OTEL_TRACES_EXPORTER = jaeger*
- *OTEL_EXPORTER_JAEGER_ENDPOINT= jaeger :14250*
- *OTEL_TRACES_SAMPLER=always_on*

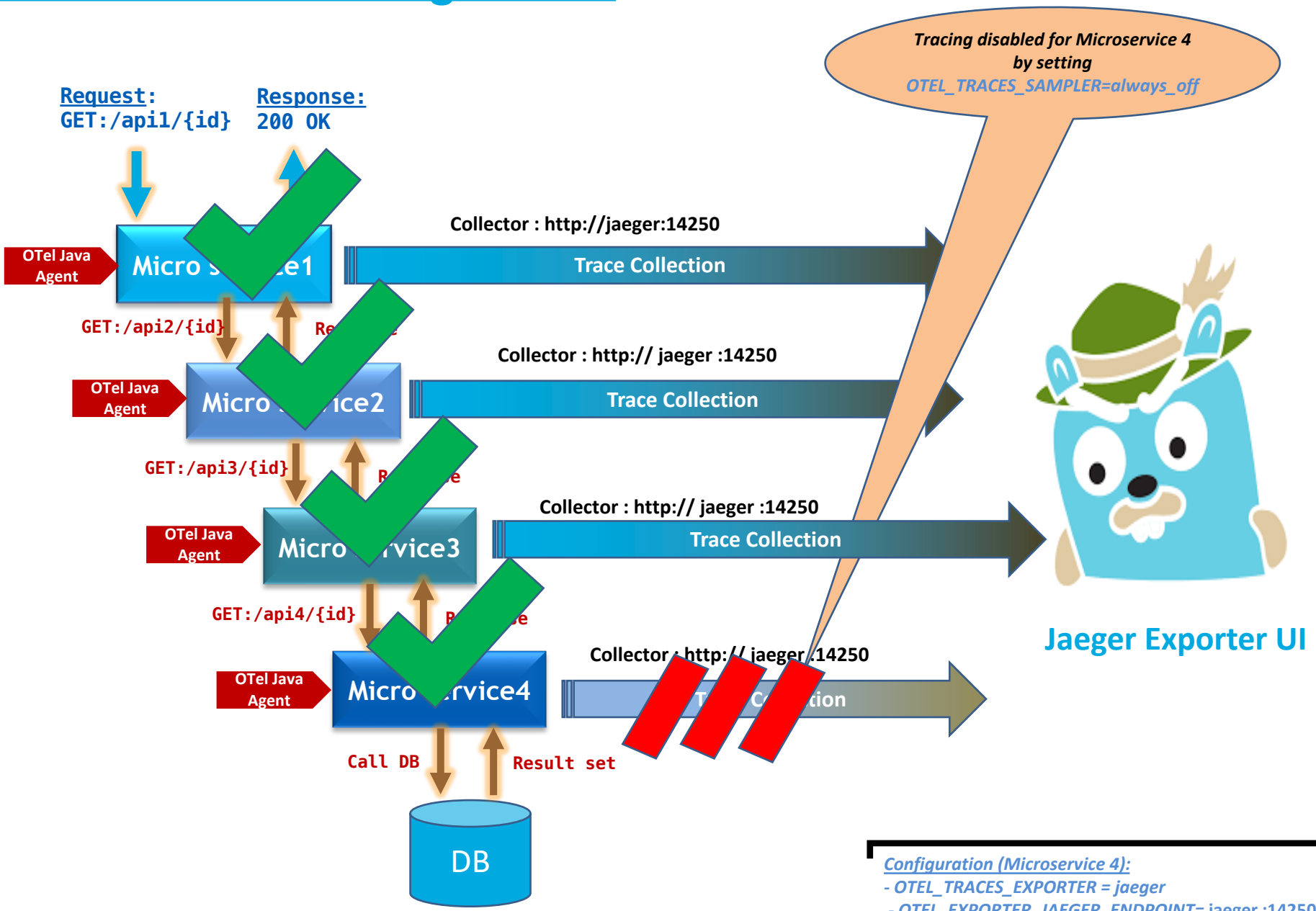
# Scenario 3: Error/Exception in Microservice




Jaeger Exporter UI

*Configuration (For all Micro Services):*
- *OTEL_TRACES_EXPORTER = jaeger*
- *OTEL_EXPORTER_JAEGER_ENDPOINT= jaeger :14250*
- *OTEL_TRACES_SAMPLER=always_on*

# Scenario 4: Disabling Traces




*Configuration (Microservice 4):*
- *OTEL_TRACES_EXPORTER = jaeger*
- *OTEL_EXPORTER_JAEGER_ENDPOINT= jaeger :14250*
- *OTEL_TRACES_SAMPLER=always_off*

# Pros and Cons

**Pros:**

- Automatic instrumentation requires no code changes and no need for recompilation of the application. This method uses an intelligent agent that attaches to the running application and extracts tracing data.
- Provides good coverage of application endpoints and operations.
- Saves time instrumenting your code and lets you focus on the business.
- Reduces the need for code changes due to updates to the instrumentation (such as new metadata captured in the payload)

**Cons:**

- Not all the languages and frameworks provide auto-instrumentation. It support limited languages like Python, Java, .NET , PHP etc..
- Offers less flexibility than manual instrumentation, typically in the scope of a function or a method call
- Only instruments basic metrics around usage and performance. Business metrics or other custom metrics need to be manually instrumented

# Manual Instrumentation

# Overview

- Manual instrumentation is the act of adding observability code to an app yourself.
- For manual instrumentation, you need to use the OpenTelemetry SDK for your language. You'll then use the SDK to initialize OpenTelemetry and the API to instrument your code. This will emit telemetry from your app, and any library you installed that also comes with instrumentation.
- Manual instrumentation, while requiring more work on the user/developer side, enables far more options for customization, from naming various components within OpenTelemetry (for example, spans and the tracer) to adding your own attributes, specific exception handling, and more.
- It provides complete control of instrumentation data.

# Configuration in Spring Boot

1. Include Opentelemetry libraries for instrumentation

```
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-api</artifactId>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-sdk</artifactId>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId>opentelemetry-exporter-otlp</artifactId>
</dependency>
<dependency>
    <groupId>io.opentelemetry</groupId>
    <artifactId> opentelemetry-semconv</artifactId>
</dependency>
```

# Configuration in Spring Boot (cont.)

2. Configure the instance of OpenTelemetry interface

```
//Resource instance represents the entity producing telemetry data
Resource resource = Resource.create (
        Attributes.of(ResourceAttributes.SERVICE_NAME, "service-name"));

//Trace Exporter represents how to send telemetry data to consumer
JaegerThriftSpanExporter exporter = JaegerThriftSpanExporter.builder()
                .setEndpoint("http://localhost:14268/api/traces")
        .build();

//Trace Provider configuration
SdkTracerProvider tracerProvider = SdkTracerProvider.builder()
        .setResource(resource)
        .addSpanProcessor(exporter).build();

//Configure the instance of OpenTelemetrySdk
OpenTelemetrySdk openTelemetry = OpenTelemetrySdk.builder()
        .setTracerProvider(tracerProvider).build();
```

# Tracer

- A Tracer creates spans containing more information about what is happening for a given operation, such as a request in a service. Tracers are created from Tracer Providers.

- Below code shows how to create a span in java

```
import io.opentelemetry.api;

//...
Tracer tracer = openTelemetry.getTracer("instrumentation-service-name",
"1.0.0");
```

# Span

- Span represents a unit of work or operation.
- Spans are the building blocks of Traces.
- To create spans, you only need to specify the name of the span. The start and end time of the span is automatically set by the OpenTelemetry SDK.
- Below code shows how to create a span in java

```
Span span = tracer.spanBuilder("my span").startSpan();

// Make the span the current span
try (Scope ss = span.makeCurrent()) {
// In this scope, the span is the current/active span
}
finally {
        span.end();
}
```

# Nested Span

- Spans can be nested, and have a parent-child relationship with other spans. When a given span is active, the newly created span inherits the active span's trace ID, and other context attributes.

**Code snippet for parent span:**

```
void parentOne() {
    Span parentSpan = tracer.spanBuilder("parent").startSpan();

    try {
        childOne(parentSpan);
    }
    finally {
        parentSpan.end();
    }
}
```

**Code snippet for child/nested span:**

```
void childOne(Span parentSpan) {
    Span childSpan = tracer.spanBuilder("child").setParent(
                        Context.current().with(parentSpan)) .startSpan();
    try {
        // do stuff
    } finally {
        childSpan.end();
    }
}
```

# Context Propagation

- Context propagation enables all data sources to share an underlying context mechanism for storing state and accessing data across the lifespan of a transaction.

- The context refers to metadata that is collected, stored, and carried across API boundaries by components within a distributed system.

    1. Information is injected into the request on the client's side (for example, into HTTP headers) and then extracted from the headers on the server side.

    2. On the server side, they are de-serialized, and a new context object is created containing the original values as well as the new and updated ones.

- Context propagation allows your distributed traces to be connected to their parent/child spans and follow the flow of execution. Without context propagation, you would find several orphaned spans in your traces.

- The headers are standardized under the W3C standards for trace context to provide vendor-agnostic data.

# Trace Context

- The traceparent HTTP header field identifies the incoming request in a tracing system.

- The format is {version}-{trace-id}-{span-id}-{trace-flags}
  - Version : Default to 00
  - Trace Id : Uniquely identifies a distributed trace
  - Span Id : Span's span Id
  - Trace Flags : Determine whether span to be traced.

***Sample trace parent value***

*Value = 00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01*
*base16(version) = 00*
*base16(trace-id) = 4bf92f3577b34da6a3ce929d0e0e4736*
*base16(span-id) = 00f067aa0ba902b7*
*base16(trace-flags) = 01*


**Scenario 1: Happy Path / No Errors**

**Scenario 2: Error/Exception in Microservice**

**Scenario 3: Disabling traces in Microservice**

# Pros and Cons

## Pros

- It support many programming languages like Java, .Net, C++, Golang, Python, Node.js, PHP, Ruby, Rust, Swift, Erlang, NestJS etc..
- Manual instrumentation gives you maximum control over the data that is being generated.
- You can instrument custom code blocks.

## Cons

- It is time consuming.
- There is a learning curve to perfect it.
- Can cause performance overhead.
- More room for human errors resulting in broken span context.
- Change the instrumentation may require recompiling the application.

# Recommendations

# Instrumentation Recommendations

- Identify the tools in your application stack that provide built-in instrumentation and enable their instrumentation for infrastructure baseline traces. For each one, verify in which format and protocol it exports trace data, and make sure you can ingest this format (using an appropriate receiver)

- Leverage auto-instrumentation as much as possible. Use agents for your programming language that can generate trace data themselves or via software frameworks, libraries and middleware you use.

- Verify the release and maturity level of each component you use, whether the collector, client library, protocol or others, as each component has its own release lifecycle.

# Questions?

# References:

- https://opentelemetry.io/docs/
- https://opentelemetry.io/docs/instrumentation/java/automatic/
- https://opentelemetry.io/docs/instrumentation/java/manual/
- https://github.com/open-telemetry/opentelemetry-java/blob/main/sdk-extensions/autoconfigure/README.md
- https://www.jaegertracing.io/docs/1.39/getting-started/

# Thank you!!!