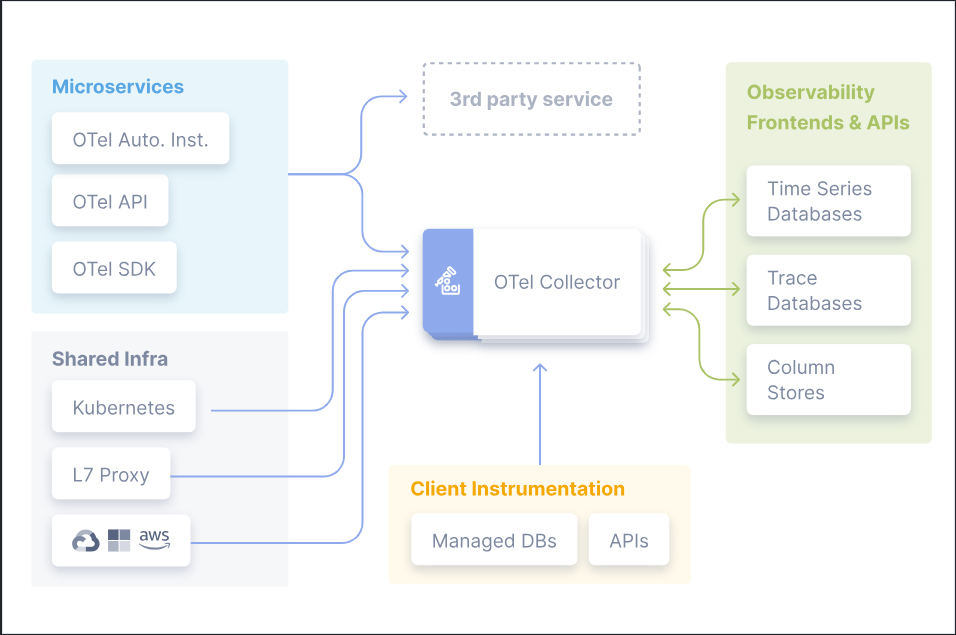
**OpenTelemetry**

OpenTelemetry, also known as OTel, is an open-source vendor-neutral observability framework designed to provide standardized methods for collecting, processing, and exporting telemetry data such as traces, metrics, and logs. It is a collaborative project under the Cloud Native Computing Foundation (CNCF), created through the merger of OpenTracing and OpenCensus.



* ***Observability***

Observability is a term used in the context of software systems to describe the ability to measure the internal state of a system by examining its outputs. It is crucial for understanding and maintaining the health, performance, and behavior of complex, distributed systems. Observability allows engineers to:

-**Diagnose Issues**: Quickly identify and understand the root causes of problems.

**-Monitor Performance**: Track the performance and health of systems over time.

-**Optimize Systems**: Improve system performance and efficiency through data-driven insights.

Observability is typically achieved through the collection and analysis of three main types of data:

-**Logs**: Detailed records of events that occur within an application, which can help trace the flow of execution and identify errors.

-**Metrics**: Quantitative measurements that provide insights into the performance and resource utilization of a system, such as CPU usage, memory consumption, and request rates.

-**Traces**: Distributed traces that track the flow of requests through various services in a distributed system, helping to pinpoint where latency or errors occur.

To make a system observable, it must be instrumented. That is, the code must emit traces, metrics, or logs. The instrumented data must then be sent to an observability backend

* ***Traces in Observability***

Traces are a key component of observability, especially in the context of distributed systems where applications are composed of multiple microservices or components. Traces provide a way to understand the lifecycle of a request as it propagates through different services and components. Here’s a more detailed look at traces:

-**Distributed Tracing**: This is the practice of tracing requests as they move through different services in a distributed system. Each trace captures the complete journey of a request, including each service it touches and the time spent in each service.

-**Spans**: A trace is composed of multiple spans. A span represents a single operation within a trace. For example, a span might represent an HTTP request handled by a service, a database query, or any other discrete operation. Spans contain metadata such as:

-Operation Name: The name of the operation.

-Start and End Timestamps: When the operation started and ended.

-Tags/Attributes: Key-value pairs that provide additional information about the operation

(e.g., HTTP method, status code).

-Logs/Events: Time-stamped records of significant events within the span (e.g., errors).

-**Context Propagation**: To trace requests across services, context propagation mechanisms are used. This involves passing trace context information (like trace IDs and span IDs) along with the requests so that all parts of the trace can be correlated.

For example, if Service A calls Service B, then a span from Service A whose ID is in context will be used as the parent span for the next span created in Service B. The trace ID that is in context will be used for the next span created in Service B as well, which signifies that the span is part of the same trace as the span from Service A.

**Example Scenario**

Consider an e-commerce application where a user request to purchase an item involves several services:

Frontend Service: Receives the user request.

Order Service: Processes the order.

Inventory Service: Checks and updates inventory.

Payment Service: Processes the payment.

Notification Service: Sends a confirmation email.

A trace for this purchase request would span all these services, capturing the start and end times for each service call, along with any additional metadata. If there’s an issue (e.g., the payment service is slow), the trace will help identify where the delay occurs.

**Benefits of Tracing**

-Root Cause Analysis: Helps quickly pinpoint where errors or performance issues are occurring within a distributed system.

-Performance Optimization: Identifies bottlenecks and inefficient operations that can be optimized.

-Improved Visibility: Provides a clear view of how requests are processed across multiple services, improving understanding of system behavior.

* ***Signals in OTel*** :-

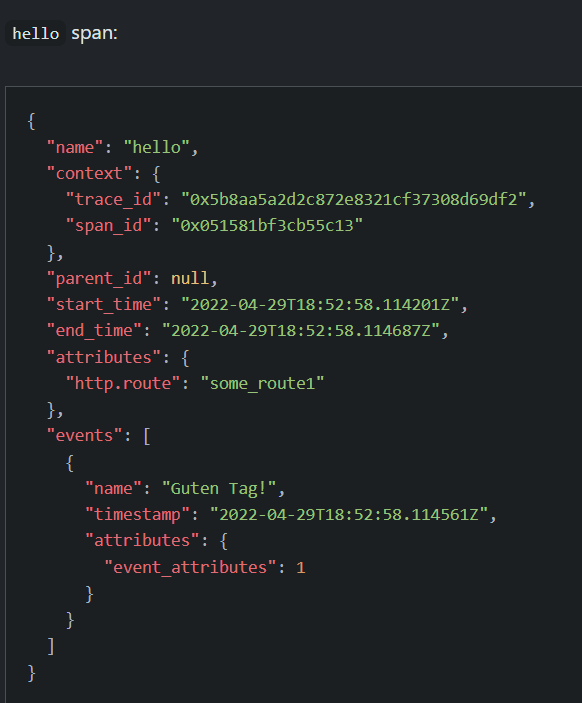
The purpose of OpenTelemetry is to collect, process, and export signals. Signals are system outputs that describe the underlying activity of the operating system and applications running on a platform. A signal can be something you want to measure at a specific point in time, like temperature or memory usage, or an event that goes through the components of your distributed system that you’d like to trace. You can group different signals together to observe the inner workings of the same piece of technology under different angles.

OpenTelemetry currently supports traces, metrics, logs and baggage.

* **Traces** :-

Traces give us the big picture of what happens when a request is made to an application. Whether your application is a monolith with a single database or a sophisticated mesh of services, traces are essential to understanding the full “path” a request takes in your applicationl. Let’s understand the Traces with an example :

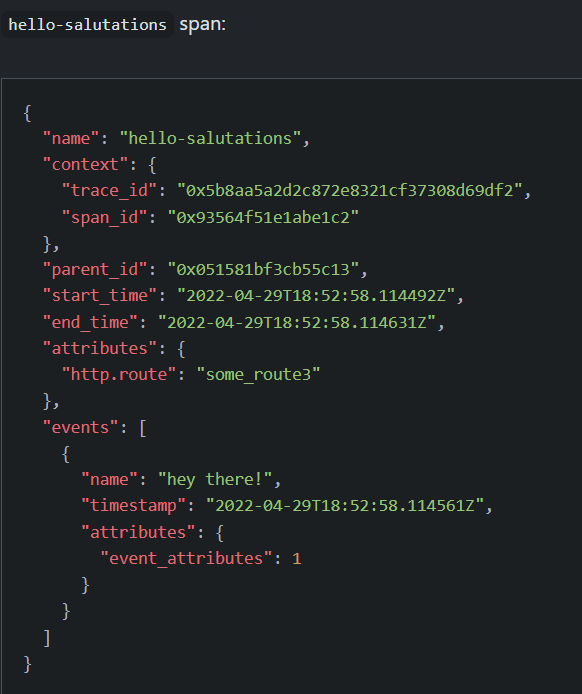
‘hello’ span : This is the root span, denoting the beginning and end of the entire operation. Note that it has a trace\_id field indicating the trace, but has no parent\_id. That’s how you know it’s the root span.



‘hello-greetings’ span: This span encapsulates specific tasks, like saying greetings, and its parent is the hello span. Note that it shares the same trace\_id as the root span, indicating it’s a part of the same trace. Additionally, it has a parent\_id that matches the span\_id of the hello span.



‘hello-salutations’ span: This span represents the third operation in this trace and, like the previous one, it’s a child of the hello span. That also makes it a sibling of the hello-greetings span.



These three blocks of JSON all share the same trace\_id, and the parent\_id field represents a hierarchy. That makes it a Trace!

Use Case: Traces help track the flow and performance of requests through distributed systems.

* **Metrics** :-

Metrics provide quantitative data about the performance and health of a system. They are typically numerical measurements collected at regular intervals. Metrics are used for monitoring system health, identifying trends, and triggering alerts.

Types of Metrics:

-Counters: Measure the count of events or operations, e.g., the number of HTTP requests received.

-Gauges: Measure a value that can go up and down, e.g., current memory usage.

-Histograms: Measure the distribution of values over a range, e.g., response times.

-Summaries: Provide aggregated data such as min, max, and percentiles, useful for summarizing large amounts of data.

Use Case: Monitoring CPU usage, memory consumption, request rates, and error rates in a service.

* **Logs** :-

Logs are detailed, timestamped records of discrete events that occur within a system. They provide context-rich information about the system's operations and are essential for debugging and forensic analysis.

Log Entries: Each log entry typically includes a timestamp, severity level (e.g., INFO, ERROR), and a message describing the event. They may also include additional context such as stack traces for errors.

Structured Logs: Modern logging practices often involve structured logging, where logs are output in a structured format (e.g., JSON), making them easier to query and analyze.

Use Case: Capturing error messages and stack traces when a service encounters an exception, aiding in debugging.

* **Baggage** :-

Baggage is contextual information that’s passed between spans. It’s a key-value store that resides alongside span context in a trace, making values available to any span created within that trace.

For example, imagine you want to have a ‘CustomerId’ attribute on every span in your trace, which involves multiple services; however, ‘CustomerId’ is only available in one specific service. To accomplish your goal, you can use OpenTelemetry Baggage to propagate this value across your system.

OpenTelemetry uses Context Propagation to pass Baggage around, and each of the different library implementations has propagators that parse and make that Baggage available without you needing to explicitly implement it.

Use Case: Common use cases include information that’s only accessible further up a stack. This can include things like Account Identification, User IDs, Product IDs, and origin IPs, for example. Passing these down your stack allows you to then add them to your Spans in downstream services to make it easier to filter when you’re searching in your Observability backend.

* ***Instrumentation*** :-

In order to make a system observable, it must be instrumented: That is, code from the system’s components must emit traces, metrics, and logs.

Using OpenTelemetry, you can instrument your code in two primary ways, You can use both solutions simultaneously :

-Code-based solutions via official APIs and SDKs for most languages

-Zero-code solutions

Code-based solutions allow you to get deeper insight and rich telemetry from your application itself. They let you use the OpenTelemetry API to generate telemetry from your application, which acts as an essential complement to the telemetry generated by zero-code solutions.

Zero-code solutions are great for getting started, or when you can’t modify the application you need to get telemetry out of. They provide rich telemetry from libraries you use and/or the environment your application runs in. Another way to think of it is that they provide information about what’s happening at the edges of your application.