OTel Without Reservations

Ethan Kent

June 21, 2023

Overview

Introduction

Signals

Context and Context Propagation

API, SDK, and Semantic Conventions

Brief aside, wherein Ethan gets back up on his Dependency-Inversion soapbox

API, SDK, and Semantic Conventions, resumed

Practical Use of OpenTelemetry

Conclusion

Introduction

What is OpenTelemetry?

According to the OpenTelemetry people,[1]

> *OpenTelemetry is a collection of tools, APIs, and SDKs. Use it to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) to help you analyze your software's performance and behavior.*[2]

[1] OpenTelemetry is a project of The Cloud Native Computing Foundation and is a merger of the OpenTracing and OpenCensus projects.

[2] The OpenTelemetry Authors. **OpenTelemetry.** 2023. URL: https://opentelemetry.io/ (visited on 06/20/2023).

Why do I care?

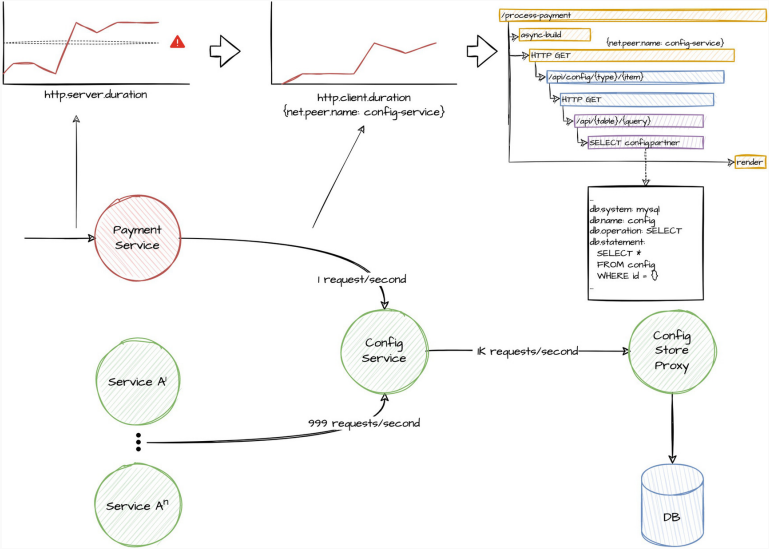
A distributed trace, more commonly known as a trace, records the paths taken by requests (made by an application or end-user) as they propagate through multi-service architectures, like microservice and serverless applications.

Without tracing, it is challenging to pinpoint the cause of performance problems in a distributed system.

It improves the visibility of our application or system's health and lets us debug behavior that is difficult to reproduce locally. Tracing is essential for distributed systems, which commonly have nondeterministic problems or are too complicated to reproduce locally.[3]

[3] The OpenTelemetry Authors. **Obervability Primer, OpenTelemetry.** 2023. URL: https://opentelemetry.io/docs/concepts/observability-primer/ (visited on 06/20/2023).

Why do I care? (cont'd)



Why do I care? (cont'd)

Observability OpenTelemetry (OTel) offers comprehensive observability across your systems. OTel provides a clear picture of what's happening within your services by bringing together traces, metrics, and logs.

Reduced Complexity OTel unifies multiple observability signals into a coherent framework. A single framework simplifies instrumentation and eliminates the need to learn and manage various systems.

Standards-Based OTel is open-source and vendor-neutral, providing standardized APIs and instrumentation that integrates with any backend.

Extensibility OTel's modular design allows you to use what you need, and its SDKs let you build custom integrations as required.

Cost and Resource Efficient OTel can reduce the overhead and costs of running multiple observability tools.

Why do I care? (cont'd)

Support for Modern Architectures OTel provides first-class support for Kubernetes, serverless functions, and service meshes.

Enhanced Debugging With better visibility, engineers can identify, understand, and resolve issues faster.

Performance Monitoring OTel helps you monitor system performance and user behavior in real time, helping you make data-driven decisions to improve your services' performance and user experience.

Future-Proof OTel's wide adoption and large community likely mean that as new standards and best practices emerge, OTel will evolve with them.

Signals

Signals

"In OpenTelemetry, a signal refers to a category of telemetry."[4]

The four kinds of signals currently supported are

- Traces,
- Metrics,
- Logs, and
- Baggage.[5]

[4] The OpenTelemetry Authors. **Signals, OpenTelemetry.** 2023. URL: https://opentelemetry.io/docs/concepts/signals/ (visited on 06/20/2023).

[5] The OpenTelemetry Authors, *Signals, OpenTelemetry*, see n. 4.

Traces

Spans are the bread and butter of distributed tracing. A *Trace* has many *Spans*.

> *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 application.*[6]

[6] The OpenTelemetry Authors. **Traces, OpenTelemetry.** 2023. URL: https://opentelemetry.io/docs/concepts/signals/traces/ (visited on 06/20/2023).

Metrics

> *A metric is a measurement about a service, captured at runtime. Logically, the moment of capturing one of these measurements is known as a metric event which consists not only of the measurement itself, but the time that it was captured and associated metadata.*[7]

[7] The OpenTelemetry Authors. **Metrics, OpenTelemetry.** 2023. URL: https://opentelemetry.io/docs/concepts/signals/metrics/ (visited on 06/20/2023).

Metrics (cont'd)

Use metrics when you want metrics.

> *Some telemetry backends allow to execute ad hoc queries on spans, [and it] is also possible to derive metrics from spans in OpenTelemetry Collectors.* *This often results in an overuse of spans as a substitute for metrics when teams start to adopt tracing.* *Trace sampling can affect any interpretations extracted directly from spans, . . . [and] the high volumes of data and cardinality processed by trace pipelines in high-throughput systems normally result in signals that are less stable than metrics originating directly from a service . . . .*[8]

[8] Daniel Gomez Blanco. **Practical OpenTelemetry: Adopting Open Observability Standards Across Your Organization.** Berkeley, CA: Apress, 2023, ch. 6, emphasis added.

Logs

> *In OpenTelemetry, any data that is not part of a distributed trace or a metric is a log. For example, events are a specific type of log. Logs often contain detailed debugging/diagnostic info, such as inputs to an operation, the result of the operation, and any supporting metadata for that operation.*[9]

[9] The OpenTelemetry Authors. **Logs, OpenTelemetry.** 2023. URL: https://opentelemetry.io/docs/concepts/signals/logs/ (visited on 06/20/2023).

Logs: not well-supported yet

Language Support

Metrics are a stable signal in the Oper

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

Baggage

"In OpenTelemetry, 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."[10]

It's in the HTTP headers, so don't store sensitive data.

"Common use cases include . . . Account Identification, User Ids, Product Ids, and origin IPs. . . . 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 back-end."[11]

[10] The OpenTelemetry Authors. **Baggage, OpenTelemetry.** 2023. URL: https://opentelemetry.io/docs/concepts/signals/baggage/ (visited on 06/20/2023).
[11] The OpenTelemetry Authors, *Baggage, OpenTelemetry*, see n. 10.

Context and Context Propagation

How does this distributed stuff work?

Distributed tracing is hard because it's tricky to reconstruct cause-and-effect relationships. In distributed systems, many separate services—some internal and some external—may collaborate to produce a final result. The secret sauce for OpenTelemetry can be split into three pieces:

1. A trace ID, additional correlation IDs, and other metadata, collectively called "Context."
2. Standards governing how Context is transmitted within and among services, called "Context Propagation."
3. An ecosystem comprising open standards, libraries, SDKs, vendors, etc.

In short, OpenTelemetry is a well-thought-out system for assigning IDs in a tree-like structure, passing them across service boundaries, and then allowing Humpty-Dumpty to be put back together again.

Context

In order for OpenTelemetry to work, it must store and propagate important telemetry data. For example, when a request is received and a span is started it must be available to a component which creates its child span. To solve this problem, OpenTelemetry stores the span in the Context.[12]

[12] The OpenTelemetry Authors. **Context, OpenTelemetry.** 2023. URL: https://opentelemetry.io/docs/instrumentation/js/context/ (visited on 06/21/2023).

Context (cont'd)

A Context is a propagation mechanism which [sic] carries execution-scoped values across API boundaries and between logically associated execution units. Cross-cutting concerns access their data in-process using the same shared Context object.[13]

Execution Unit *An umbrella term for the smallest unit of sequential code execution, used in different concepts of multitasking. Examples are threads, coroutines or fibers.*[14]

[13] The OpenTelemetry Authors. **Context, OpenTelemetry.** 2023. URL: https://opentelemetry.io/docs/specs/otel/context/ (visited on 06/21/2023).
[14] The OpenTelemetry Authors. **Glossary, OpenTelemetry.** 2023. URL: https://opentelemetry.io/docs/specs/otel/glossary/ (visited on 06/21/2023).

Context Propagation

> *Propagation is the mechanism that moves data between [sic] services and processes. Although not limited to tracing, it is what allows traces to build causal information about a system across services that are arbitrarily distributed across process and network boundaries.*[15]

[15] The OpenTelemetry Authors. **Propagation, OpenTelemetry.** 2023. URL: https://opentelemetry.io/docs/instrumentation/js/propagation/ (visited on 06/21/2023).

Context Propagation (cont'd)

You can think of a Propagators as serializers/deserializers or marshallers/unmarshallers for Context.

> *Cross-cutting concerns send their state to the next process using Propagators, which are defined as objects used to read and write context data to and from messages exchanged by the applications. Each concern creates a set of Propagators for every supported Propagator type.*[16]

[16] The OpenTelemetry Authors. **Propagators API, OpenTelemetry.** 2023. URL: https://opentelemetry.io/docs/specs/otel/context/api-propagators/ (visited on 06/21/2023).

API, SDK, and Semantic Conventions

API and SDK

Remember my talks on Onion Architecture and SOLID, specifically the Dependency-Inversion Principle? What, you think you can pick up in Season 3 and not miss anything?

Okay, VERY BRIEFLY THEN.

Brief aside, wherein Ethan gets back up on his Dependency-Inversion soapbox

Not dependency inverted

```
class MyBusinessLogic {
  private logger: ConsoleLogger;
  private businessDataWriter: MySqlBusinessDataWriter;

  constructor() {
    this.logger = new ConsoleLogger();
    this.businessDataWriter = new MySqlBusinessDataWriter();
  }

  public processBusinessData(businessData: BusinessData) {
    this.businessDataWriter
      .writeBusinessData(businessData)
      .then(() => this.logger.logDebug("It worked fine"))
      .catch((e) => this.logger.logError(`We had a problem: ${e}`));
  }
}
```

Not dependency inverted (cont'd)

Requirements change, hotshot: let's use Pino instead of the console to do logging, and let's use PostgreSQL instead of MySql for storing the data.

Should the implementation have to change for our business-logic class if only the dependencies, not the business logic, changed?

Can you do that without changing the innards of MyBusinessLogic as presently written? Does this violate the Dependency-Inversion Principle? Does it violate the Open–Closed Principle? Does a business stakeholder now potentially have to care about your database or logging choice?

Answer Key: No (uh-oh)
No (whoops), Yes (whoops), Yes (whoops), Yes (whoops).

Dependency inverted

```
type LoggingService = {
  logDebug: (message: string) => void;
  logError: (message: string) => void;
};

type BusinessDataWriterService = {
  writeBusinessData: (businessData: BusinessData) => Promise<void>;
};
```

Dependency inverted (cont'd)

```
class MyBusinessLogic {
  constructor(
    private logger: LoggingService,
    private businessDataWriter: BusinessDataWriterService
  ) {}

  public processBusinessData(businessData: BusinessData) {
    this.businessDataWriter
      .writeBusinessData(businessData)
      .then(() => this.logger.logDebug("It worked fine"))
      .catch((e) => this.logger.logError(`We had a problem: ${e}`));
  }
}
```

Dependency inverted (cont'd)

```
// New file pino-logger.ts
class PinoLogger implements LoggingService { /* ...impl */ }

// New file postgres-sql-writer.ts
class PostgreSqlBusinessDataWriter implements BusinessDataWriterService {
  /* ...impl */
}

// In main.ts, update this
new MyBusinessLogic(new ConsoleLogger(), new MySqlBusinessDataWriter());

// to this
new MyBusinessLogic(new PinoLogger(), new PostgreSqlBusinessDataWriter());
```

Dependency inverted (cont'd)

Requirements change, hotshot: let's use Pino instead of the console to do logging, and let's use PostgreSQL instead of MySql for storing the data.

Should the implementation have to change for our business-logic class if only the dependencies, not the business logic, changed?

Can you do that without changing the innards of MyBusinessLogic as presently written? Does this violate the Dependency-Inversion Principle? Does it violate the Open–Closed Principle? Does a business stakeholder now potentially have to care about your database or logging choice?

Answer Key: Heck no!
heck yes, heck no, heck no, heck no.

Dependency inverted (cont'd)

How did we do this?

> *Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.*[17]

[17] Wikipedia contributors. **Dependency inversion principle — Wikipedia, The Free Encyclopedia.** [Online; accessed 21-June-2023]. 2023. URL: https://en.wikipedia.org/w/index.php?title=Dependency_inversion_principle&oldid=1148320529.

API, SDK, and Semantic Conventions, resumed

API and SDK (cont'd)

The API is the dependency-inverted abstraction. Remember, depend on abstractions.

> *API packages consist of the cross-cutting public interfaces used for instrumentation. Any portion of an OpenTelemetry client which is imported into third-party libraries and application code is considered part of the API.*[18]

[18] The OpenTelemetry Authors. **Overview, OpenTelemetry.** 2023. URL: https://opentelemetry.io/docs/specs/otel/overview/ (visited on 06/21/2023).

API and SDK (cont'd)

The SDK is the dependency-inverted set of concretions. The actual implementations.

> *The SDK is the implementation of the API provided by the OpenTelemetry project. Within an application, the SDK is installed and managed by the application owner. Note that the SDK includes additional public interfaces which are not considered part of the API package, as they are not cross-cutting concerns. These public interfaces are defined as constructors and plugin interfaces. Application owners use the SDK constructors; plugin authors use the SDK plugin interfaces. Instrumentation authors MUST NOT directly reference any SDK package of any kind, only the API.*[19]

[19] The OpenTelemetry Authors, *Overview, OpenTelemetry*, see n. 18.

Semantic Conventions

"Semantic Conventions" is the name for what amounts to a set of constants specified to allow naming things consistently. They live in their own repo, https://github.com/open-telemetry/semantic-conventions.

Practical Use of OpenTelemetry

Auto Instrumentation

Zero-touch model: *The instrumented application code is unchanged, and the OpenTelemetry SDK and instrumentation libraries are initialized by a separate component and configured independently from the instrumented code. This model is the easiest one to implement for service owners, but it's not available in every language and normally requires being able to modify the command used to start the application, for example, attaching the Java agent or using Python's telemetry—instrument command, or having access to configure the runtime environment as in .NET's CLR profiler. In some cases, this model can limit the options to configure or customize certain aspects of instrumentation libraries.*[20]

[20] Gomez Blanco, see n. 8, ch. 4.

Manual Instrumentation

Implementation model: The OpenTelemetry SDK and instrumentation libraries are configured and initialized by the application owner as part of the instrumented application code, normally happening during application startup. This may provide more flexibility to configure instrumentations, but it requires extra effort to install and maintain. In some languages, like Java, this model can have additional drawbacks, as instrumentation libraries that dynamically inject bytecode in the agent model cover a wider set of instrumented libraries and frameworks.[21]

[21] Gomez Blanco, see n. 8, ch. 4.

Automatic v. Manual Instrumentation

In practice, a combination of both may be used, with auto instrumentation covering standard libraries and frameworks, and manual instrumentation filling in the gaps for custom or unsupported libraries, or for more granular control over spans and attributes.

Getting Started with OpenTelemetry

Explain the first steps in installing and using OpenTelemetry in a chosen language, including adding necessary dependencies and initializing OpenTelemetry in the application's main function.

Instrumenting Code

Describe how to instrument code, explaining how to create spans, set attributes, add events, and handle errors. Include example code to make this practical.

Collecting and Exporting Data

Discuss the OpenTelemetry Collector and how it can be used to receive, process, and export telemetry data. Describe the different exporters available, such as Jaeger, Zipkin, and Prometheus, and how to configure them.

Visualizing Traces

Show how to use a tool like Jaeger or Zipkin to visualize traces. Include screenshots of a trace and explain how to interpret it.

Conclusion

Conclusion

Wrap up by summarizing the importance of OpenTelemetry for observability in modern, distributed systems and the benefits of its unified approach to traces, metrics, and logs. Highlight again the fact that OpenTelemetry is vendor-neutral, open-source, and widely adopted, which makes it a safe choice for most organizations.

Further Reading

Recommend resources for further reading or learning, such as the official OpenTelemetry documentation, relevant blog posts, tutorials, and books.

Q&A

Leave time for questions and answers from the audience.