

Cloud-Native Observability with OpenTelemetry

Learn to gain visibility into systems by combining tracing, metrics, and logging with OpenTelemetry

Alex Boten

Foreword by Charity Majors, CTO, Honeycomb



Cloud-Native Observability with OpenTelemetry

Learn to gain visibility into systems by combining tracing, metrics, and logging with OpenTelemetry

Alex Boten



BIRMINGHAM—MUMBAI

Cloud-Native Observability with OpenTelemetry

Copyright © 2022 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Rahul Nair

Publishing Product Manager: Shrilekha Malpani

Senior Editor: Arun Nadar

Content Development Editor: Sujata Tripathi

Technical Editor: Rajat Sharma

Copy Editor: Safis Editing

Project Coordinator: Shagun Saini

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Production Designer: Ponraj Dhandapani

Marketing Coordinator: Nimisha Dua

First published: April 2022

Production reference: 1140422

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80107-770-5

www.packtpub.com

*To my mother, sister, and father. Thank you for teaching me to persevere in
the face of adversity, always be curious, and work hard.*

Foreword

It has never been a better time to be a software engineer.

As engineers, we are motivated by impact and efficiency—and who can argue that both are not skyrocketing, particularly in comparison with time spent and energy invested?

These days, you can build out a scalable, elastic, distributed system to serve your code to millions of users per day with a few clicks—without ever having to personally understand much about operations or architecture. You can write lambda functions or serverless code, hit save, and begin serving them to users immediately.

It feels like having superpowers, especially for those of us who remember the laborious times before. Every year brings more powerful APIs and higher-level abstractions – many, many infinitely complex systems that "just work" at the click of a button or the press of a key.

But when it *doesn't* "just work," it has gotten harder than ever to untangle the reasons and understand why.

Superpowers don't come for free it turns out. The winds of change may be sweeping us all briskly out toward a sea of ever-expanding options, infinite flexibility, automated resiliency, and even cost-effectiveness, but these glories have come at the price of complexity—skyrocketing, relentlessly compounding complexity and the cognitive overload that comes with it.

Systems no longer fail in predictable ways. Static dashboards are no longer a viable tool for understanding your systems. And though better tools will help, digging ourselves out of this hole is not merely an issue of switching from one tool to another. We need to rethink the way software gets built, shipped, and maintained, to be *production-focused* from day 1.

For far too long now, we have been building and shipping software in the dark. Software engineers act like all they need to do is write tests and make sure their code passes. While tests are important, all they can really do is validate the logic of your code and increase your confidence that you have not introduced any serious regressions. Operations engineers, meanwhile, rely on monitoring checks, but those are a blunt tool at best. Most bugs will never rise to the criticality of a paging alert, which means that as a system gets more mature and sophisticated, most issues will have to be found and reported by your users.

And this isn't just a problem of bugs, firefighting, or outages. This is about understanding your software in the wild—as *your users run your code on your infrastructure*, at a given time. Production remains far too much of a black box for too many people, who are then forced to try and reason about it by reading lines of code and using elaborate mental models.

Because we've all been shipping code blindly, all this time, we ship changes we don't fully understand to a production system that is a hairball of changes we've never truly understood. We've been shipping blindly for years and years now, leaving SRE teams and ops teams to poke at the black boxes and try to clean up the mess—all the while still blindfolded. The fact that anything has *ever* worked is a testament to the creativity and dedication of these teams.

A funny thing starts happening when people begin instrumenting their code for observability and inspecting it in production—regularly, after every deployment, as a habit. You find bugs *everywhere*, bugs you never knew existed. It's like picking up a rock and watching all the little nasties lurking underneath scuttle away from the light.

With monitoring tools and aggregates, we were always able to see that errors existed, but we had no way of correlating them to an event or figuring out what was different about the erroring requests. Now, all of a sudden, we are able to look at an error spike and say, "Ah! All of these errors are for requests coming from clients running app version 1.63, calling the /export endpoint, querying the primaries for mysql-shard3, shard5, and shard7, with a payload of over 10 KB, and timing out after 15 seconds." Or we can pull up a trace and see that one of the erroring requests was issuing thousands of serial database queries in a row. So many gnarly bugs and opaque behaviors become shallow once you can visualize them. It's the most satisfying experience in the world.

But yes, you do have to instrument your code. (Auto-instrumentation is about as effective as automated code commenting.) So let's talk about that.

I can hear you now—"Ugh, instrumentation!" Most people would rather get bitten by a rattlesnake than refactor their logging and instrumentation code. I know this, and so does every vendor under the sun. This is why even legacy logging companies are practically printing money. Once they get your data flowing in, it takes an act of God to move it or turn it off.

This is a big part of the reason we, as an industry, are so behind when it comes to public, reusable standards and tooling for instrumentation and observability, which is why I am so delighted to participate in the push for OpenTelemetry. Yes, it's in the clumsy toddler years of technological advancement. But it will get better. It *has gotten* better. I was cynical about OTEL in the early days, but the community excitement and uptake have exceeded my expectations at every step. As well it should. Because the promise of OpenTelemetry is that you may need to instrument your code once, but only once. And then you can move from vendor to vendor *without re-instrumenting*.

This means vendors will have to compete for your business on features, usability, and cost-effectiveness, instead of vendor lock-in. OTel has the potential to finally break this stranglehold—to make it so you only instrument once, and you can move from vendor to vendor with just a few lines of configuration changes. This is brilliant—this changes everything. This is one battle you should absolutely join and fight.

Software systems aren't going to get simpler anytime soon. Yet the job of developing and maintaining software may paradoxically be poised to get faster and easier, by forcing us to finally adopt better real-time instrumentation and telemetry. Going from monitoring to observability is like the difference between **visual flight rating (VFR)** and **instrument flight rating (IFR)** for pilots. Yeah, learning to fly (or code) by instrumentation feels a little strange at first, but once you master it, you can fly so much faster, farther, and more safely than ever before.

It's not *just* about observability. There are lots of dovetailing trends in tech right now—feature flags, chaos engineering, progressive deployment, and so on—all of which center production, and focus on shrinking the distance and tightening the feedback loops between dev and prod. Together they deliver compounding benefits that help teams move swiftly and safely, devoting more of their time to solving new and interesting puzzles that move the business forward, and less time to toil and yak shaving.

It's not just about observability... but it starts with observability. The ability to see what is happening is the most important feedback loop of all.

And observability starts with instrumentation.

So, here we go.

Charity Majors

CTO, Honeycomb

Contributors

About the author

Alex Boten is a senior staff software engineer at Lightstep and has spent the last 10 years helping organizations adapt to a cloud-native landscape. From building core network infrastructure to mobile client applications and everything in between, Alex has first-hand knowledge of how complex troubleshooting distributed applications is.

This led him to the domain of observability and contributing to open source projects in the space. A contributor, approver, and maintainer in several aspects of OpenTelemetry, Alex has helped evolve the project from its early days in 2019 into the massive community effort that it is today.

More than anything, Alex loves making sense of the technology around us and sharing his learnings with others.

About the reviewer

Yuri Grinshteyn strongly believes that reliability is a key feature of any service and works to advocate for site reliability engineering principles and practices. He graduated from Tufts University with a degree in computer engineering and has worked in monitoring, diagnostics, observability, and reliability throughout his career. Currently, he is a site reliability engineer at Google Cloud, where he works with customers to help them achieve appropriate reliability for their services; previously, he worked at Oracle, Compuware, Hitachi Consulting, and Empirix. You can find his work on YouTube, Medium, and GitHub. He and his family live just outside of San Francisco and love taking advantage of everything California has to offer.

Table of Contents

Preface

Section 1: The Basics

1

The History and Concepts of Observability

Understanding cloud-native applications	4	OpenTracing	11
Looking at the shift to DevOps	6	OpenCensus	13
Reviewing the history of observability	7	Observability for cloud-native software	15
Centralized logging	8	Understanding the concepts of OpenTelemetry	16
Using metrics and dashboards	8	Signals	16
Applying tracing and analysis	9	Pipelines	20
Understanding the history of OpenTelemetry	10	Resources	22
		Context propagation	23
		Summary	25

2

OpenTelemetry Signals – Traces, Metrics, and Logs

Technical requirements	28	Metrics	39
Traces	33	Anatomy of a metric	40
Anatomy of a trace	34	Data point types	42
Details of a span	37	Exemplars	47
Additional considerations	38	Additional considerations	47

Logs	48	Additional considerations	52
Anatomy of a log	48	Semantic conventions	52
Correlating logs	50	Summary	55

3

Auto-Instrumentation

Technical requirements	58	Runtime hooks and monkey patching	66
What is auto-instrumentation?	60	Instrumenting libraries	66
Challenges of manual instrumentation	60	The Instrumentor interface	67
Components of auto-instrumentation	61	Wrapper script	68
Limits of auto-instrumentation	62		
Bytecode manipulation	63	Summary	71
OpenTelemetry Java agent	64		

Section 2: Instrumenting an Application

4

Distributed Tracing – Tracing Code Execution

Technical requirements	76	Propagating context	106
Configuring the tracing pipeline	77	Additional propagator formats	109
Getting a tracer	79	Composite propagator	110
Generating tracing data	80	Recording events, exceptions, and status	116
The Context API	82	Events	117
Span processors	90	Exceptions	118
Enriching the data	92	Status	122
ResourceDetector	94		
Span attributes	96	Summary	125
SpanKind	100		

5

Metrics – Recording Measurements

Technical requirements	128	Customizing metric outputs with views	149
Configuring the metrics pipeline	129		
Obtaining a meter	132	Filtering	149
Push-based and pull-based exporting	134	Dimensions	152
		Aggregation	155
Choosing the right OpenTelemetry instrument	137	The grocery store	157
Counter	138	Number of requests	158
Asynchronous counter	140	Request duration	162
An up/down counter	142	Concurrent requests	167
Asynchronous up/down counter	143	Resource consumption	169
Histogram	145	Summary	171
Asynchronous gauge	147		
Duplicate instruments	148		

6

Logging – Capturing Events

Technical requirements	174	A logging signal in practice	185
Configuring OpenTelemetry logging	175	Distributed tracing and logs	187
Producing logs	177	OpenTelemetry logging with Flask	189
Using LogEmitter	177	Logging with WSGI middleware	191
The standard logging library	180	Resource correlation	192
		Summary	193

7

Instrumentation Libraries

Technical requirements	196	Command-line options	204
Auto-instrumentation configuration	198	Requests library instrumentor	205
OpenTelemetry distribution	201	Additional configuration options	206
OpenTelemetry configurator	202	Manual invocation	206
Environment variables	203	Double instrumentation	210

Automatic configuration	211	Shopper	221
Configuring resource attributes	211	Flask library instrumentor	225
Configuring traces	213	Additional configuration options	225
Configuring metrics	215		
Configuring logs	216	Finding instrumentation libraries	226
Configuring propagation	217		
Revisiting the grocery store	218	OpenTelemetry registry	226
Legacy inventory	218	opentelemetry-bootstrap	227
Grocery store	219	Summary	227

Section 3: Using Telemetry Data

8

OpenTelemetry Collector

Technical requirements	232	Transporting telemetry via OTLP	249
The purpose of OpenTelemetry Collector	234	Encodings and protocols	251
Understanding the components of OpenTelemetry Collector	235	Additional design considerations	251
Receivers	236	Using OpenTelemetry Collector	252
Processors	239	Configuring the exporter	253
Exporters	247	Configuring the collector	254
Extensions	248	Modifying spans	258
Additional components	249	Filtering metrics	259
		Summary	262

9

Deploying the Collector

Technical requirements	264	System-level telemetry	272
Collecting application telemetry	267	Deploying the agent	272
Deploying the sidecar	269	Connecting the sidecar and the agent	274
		Adding resource attributes	277

Collector as a gateway	279	OpenTelemetry Operator	282
Autoscaling	282	Summary	282

10

Configuring Backends

Technical requirements	286	Running in production	306
Backend options for analyzing telemetry data	288	High availability	306
Tracing	289	Scalability	306
Metrics	299	Data retention	307
Logging	302	Privacy regulations	308
		Summary	308

11

Diagnosing Problems

Technical requirements	310	Experiment #3 – unexpected shutdown	323
Introducing a little chaos	311	Using telemetry first to answer questions	326
Experiment #1 – increased latency	313		
Experiment #2 – resource pressure	318	Summary	328

12

Sampling

Technical requirements	330	Sampling at the application level via the SDK	338
Concepts of sampling across signals	331	Using the OpenTelemetry Collector to sample data	340
Traces	332	Tail sampling processor	340
Metrics	333		
Logs	333	Summary	345
Sampling strategies	334		
Samplers available	337		

Index

Other Books You May Enjoy

Preface

Cloud-Native Observability with OpenTelemetry is a guide to helping you look for answers to questions about your applications. This book teaches you how to produce telemetry from your applications using an open standard to retain control of data. OpenTelemetry provides the tools necessary for you to gain visibility into the performance of your services. It allows you to instrument your application code through vendor-neutral APIs, libraries and tools.

By reading *Cloud-Native Observability with OpenTelemetry*, you'll learn about the concepts and signals of OpenTelemetry - traces, metrics, and logs. You'll practice producing telemetry for these signals by configuring and instrumenting a distributed cloud-native application using the OpenTelemetry API. The book also guides you through deploying the collector, as well as telemetry backends necessary to help you understand what to do with the data once it's emitted. You'll look at various examples of how to identify application performance issues through telemetry. By analyzing telemetry, you'll also be able to better understand how an observable application can improve the software development life cycle.

By the end of this book, you'll be well-versed with OpenTelemetry, be able to instrument services using the OpenTelemetry API to produce distributed traces, metrics and logs, and more.

Who this book is for

This book is for software engineers and systems operators looking to better understand their infrastructure, services, and applications by using telemetry data like never before. Working knowledge of Python programming is assumed for the example applications you'll build and instrument using the OpenTelemetry API and SDK. Some familiarity with Go programming, Linux, Docker, and Kubernetes is preferable to help you set up additional components in various examples throughout the book.

What this book covers

Chapter 1, The History and Concepts of Observability, provides an overview of the evolution of observability. It describes the challenges and fragmentation that ultimately created the need for an open standard. It provides an overview of both OpenTracing and OpenCensus. The last section of this chapter dives into OpenTelemetry, how it started, and how it got to where it is today. This concepts chapter will provide you with an overview of different concepts vital in understanding OpenTelemetry. This chapter introduces signals before going over the pipeline for generating, processing, and exporting telemetry. The resources section will describe the purpose of resources.

Chapter 2, OpenTelemetry Signals – Traces, Metrics, and Logs, describes the different signals that comprise OpenTelemetry: traces, metrics, and logs. It begins by giving you an understanding of the concepts of distributed tracing by defining spans, traces, and context propagation. The following section explores metrics by looking at the different measurements and the instruments OpenTelemetry offers to capture this information. The section on logging describes how logging fits in with the other signals in OpenTelemetry. Semantic conventions are also covered in this chapter to understand their significance and role in each signal.

Chapter 3, Auto-Instrumentation, explains the challenges of manual instrumentation and how the OpenTelemetry project sets out to solve those challenges. After that, you will dive into the mechanics of auto instrumentation in different languages.

Chapter 4, Distributed Tracing – Tracing Code Execution, begins by introducing the grocery store application we will instrument throughout the book. You will then start using the OpenTelemetry APIs to configure a tracing pipeline and its various components: the tracer provider, span processor, and exporter. After obtaining a tracer, you will instrument code to generate traces. The remainder of the chapter will discuss augmenting that tracing data with attributes, events, links, statuses, and exceptions.

Chapter 5, Metrics – Recording Measurements, teaches you how to capture application metrics. You will begin by configuring the components of a metrics pipeline: the meter provider, meter, and exporter. The chapter then describes the instruments available in OpenTelemetry to collect metrics before using each one in the context of the grocery store application.

Chapter 6, Logging – Capturing Events, covers logging, the last of the core signals of OpenTelemetry discussed in this book. The chapter walks you through configuring the components of the logging pipeline to emit telemetry. You will then use existing logging libraries to enhance logged events through correlation with OpenTelemetry.

Chapter 7, Instrumentation Libraries, teaches you how to use instrumentation libraries to instrument the grocery store application automatically after learning how to do so manually. Using auto instrumentation and environment variables supported by OpenTelemetry, this chapter shows you how to obtain telemetry from your code quickly.

Chapter 8, OpenTelemetry Collector, explores another core component that OpenTelemetry provides: the OpenTelemetry Collector. The Collector allows users to collect and aggregate data before transmitting it to various backends. This chapter describes the concepts present in the Collector, presents its use cases, and explains the challenges it solves. After learning about the **OpenTelemetry Protocol (OTLP)**, you will modify the grocery store application to emit telemetry to the collector via OTLP.

Chapter 9, Deploying the Collector, puts the OpenTelemetry Collector to work in a Kubernetes environment in various deployment scenarios. You will use Kubernetes to deploy the Collector as a sidecar, agent, and gateway to collect application-level and system-level telemetry.

Chapter 10, Configuring Backends, teaches you about various open source telemetry backend options to store and visualize data. This chapter explores using OpenTelemetry with Zipkin, Jaeger, Prometheus, and Loki utilizing a local environment. You will configure exporters in application code and the OpenTelemetry Collector to emit data to these backends. After instrumenting and collecting all the telemetry from applications, it's finally time to start using this information to identify issues in a system.

Chapter 11, Diagnosing Problems, dives into techniques used to correlate data across the different OpenTelemetry signals to identify the root cause of common problems in production effectively. This chapter introduces you to chaos engineering and tools to generate synthetic loads and service interruptions to produce different scenarios.

Chapter 12, Sampling, explains the concept of sampling and how it applies to distributed tracing. Head, tail, and probability sampling strategies are introduced in this chapter. You will configure sampling using the OpenTelemetry APIs and the OpenTelemetry Collector, comparing the results of different sampling configurations.

To get the most out of this book

The examples in this book were developed on macOS x86-64 using versions of Python ranging from 3.6 to 3.9. The latest version of OpenTelemetry for Python tested is version 1.10.0, which includes experimental support for both metrics and logging. It's likely that the API will change in subsequent releases, so be aware of the version installed as you go through the examples. Consult the changelog of the OpenTelemetry Python repository (<https://github.com/open-telemetry/opentelemetry-python/blob/main/CHANGELOG.md>) for the latest updates.

Software/hardware covered in the book	Operating system requirements
OpenTelemetry for Python 1.9.0+	Any modern operating system capable of running Python 3.6+ and Docker. Examples developed on macOS x86-64. Testing of many examples has also been performed on Ubuntu Linux and macOS arm64.
OpenTelemetry Collector v0.42.0+	Any modern operating system capable of running Python 3.6+ and Docker. Examples developed on macOS x86-64. Testing of many examples has also been performed on Ubuntu Linux and macOS arm64.

Many examples in the book rely on Docker and Docker Compose to deploy environments locally. As of January 2022, the license for Docker Desktop still allows users to install it for free for personal use, education, and non-commercial open source projects. If the licensing prevents you from using Docker Desktop, there are alternatives available.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at <https://github.com/PacktPublishing/Cloud-Native-Observability>. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots and diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781801077705_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

Code in text: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The code then calls the global `set_meter_provider` method to set the meter provider for the entire application."

A block of code is set as follows:

```
from opentelemetry._metrics import set_meter_provider
from opentelemetry.sdk._metrics import MeterProvider
from opentelemetry.sdk.resources import Resource

def configure_meter_provider():
    provider = MeterProvider(resource=Resource.create())
    set_meter_provider(provider)

if __name__ == "__main__":
    configure_meter_provider()
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
from opentelemetry._metrics import get_meter_provider, set_meter_provider
...
if __name__ == "__main__":
    configure_meter_provider()
    meter = get_meter_provider().get_meter(
        name="metric-example",
        version="0.1.2",
        schema_url=" https://opentelemetry.io/schemas/1.9.0",
    )
```

Any command-line input or output is written as follows:

```
$ git clone https://github.com/PacktPublishing/Cloud-Native-  
Observability  
$ cd Cloud-Native-Observability/chapter7
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "Search for traces by clicking the **Run Query** button."

Tips or Important Notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at customercare@packtpub.com and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Share Your Thoughts

Once you've read *Cloud-Native Observability with OpenTelemetry*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Section 1: The Basics

In this part, you will learn about the origin of OpenTelemetry and why it was needed. We will then dive into the various components and concepts of OpenTelemetry.

This part of the book comprises the following chapters:

- *Chapter 1, The History and Concepts of Observability*
- *Chapter 2, OpenTelemetry Signals: Traces, Metrics, and Logs*
- *Chapter 3, Auto-Instrumentation*

1

The History and Concepts of Observability

The term **observability** has only been around in the software industry for a short time, but the concepts and goals it represents have been around for much longer. Indeed, ever since the earliest days of computing, programmers have been trying to answer the question: **is the system doing what I think it should be?**

For some, observability consists of buying a one-size-fits-all solution that includes logs, metrics, and traces, then configuring some off-the-shelf integrations and calling it a day. These tools *can* be used to increase visibility into a piece of software's behavior by providing mechanisms to produce and collect telemetry. The following are **some examples of telemetry** that can be added to a system:

- Keeping a count of the number of requests received
- Adding a log entry when an event occurs
- Recording a value for current memory consumption on a machine
- Tracing a request from a client all the way to a backend service

However, producing high-quality telemetry is only one part of the observability challenge. The other part is ensuring that events occurring across the different types of telemetry can be correlated in meaningful ways during analysis. The goal of observability is to answer questions that you may have about the system:

- If a problem occurred in production, what evidence would you have to be able to identify it?
- Why is this service suddenly overwhelmed when it was fine just a minute ago?
- If a specific condition from a client triggers an anomaly in some underlying service, would you know it without customers or support calling you?

These are some of the questions that the domain of observability can help answer. Observability is about empowering the people who build and operate distributed applications to understand their code's behavior while running in production. In this chapter, we will explore the following:

- Understanding cloud-native applications
- Looking at the shift to DevOps
- Reviewing the history of observability
- Understanding the history of OpenTelemetry
- Understanding the concepts of OpenTelemetry

Before we begin looking at the history of observability, it's important to understand the changes in the software industry that have led to the need for observability in the first place. Let's start with the shift to the cloud.

Understanding cloud-native applications

The way applications are built and deployed has drastically changed in the past few years with the increased adoption of the internet. An unprecedented increase in demand for services (for example, streaming media, social networks, and online shopping) powered by software has raised expectations for those services to be readily available. In addition, this increase in demand has fueled the need for developers to be able to scale their applications quickly. Cloud providers, such as Microsoft, Google, and Amazon, offer infrastructure to run applications at the click of a button and at a fraction of the cost, and reduce the risk of deploying servers in traditional data centers. This enables developers to experiment more freely and reach a wider audience. Alongside this infrastructure, these cloud providers also offer managed services for databases, networking infrastructure, message queues, and many other services that, in the past, organizations would control internally.

One of the advantages these cloud-based providers offer is freeing up organizations to focus on the code that matters to their businesses. This replaces costly and time-consuming hardware implementations, or operating services they lack expertise in. To take full advantage of cloud platforms, developers started looking at how applications that were originally developed as monoliths could be re-architected to take advantage of cloud platforms. The following are challenges that could be encountered when deploying monoliths to a cloud provider:

- Scaling a monolith is traditionally done by increasing the number of resources available to the monolith, also known as vertical scaling. Vertically scaling applications can only go as far as the largest available resource offered by a cloud provider.
- Improving the reliability of a monolith means deploying multiple instances to handle multiple failures, thus avoiding downtime. This is also known as horizontal scaling. Depending on the size of the monolith, this could quickly ramp up costs. This can also be wasteful if not all components of the monolith need to be replicated.

The specific challenges of building applications on cloud platforms have led developers to increasingly adopt a service-oriented architecture, or microservice architecture, that organizes applications as loosely coupled services, each with limited scope. The following figure shows a monolith architecture on the left, where all the services in the application are tightly coupled and operate within the same boundary. In contrast, the microservices architecture on the right shows us that the services are loosely coupled, and each service operates independently:

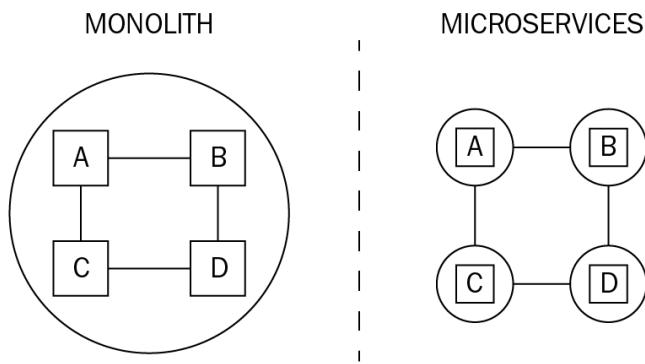


Figure 1.1 – Monolith versus microservices architecture

Applications built using microservices architecture provide developers with the ability to scale only the components needed to handle the additional load, meaning horizontal scaling becomes a much more attractive option. As it often does, a new architecture comes with its own set of trade-offs and challenges. The following are some of the new challenges cloud-native architecture presents that did not exist in traditional monolithic systems:

- Latency introduced where none existed before, causing applications to fail in unexpected ways.
- Dependencies can and will fail, so applications must be built defensively to minimize cascading failures.
- Managing configuration and secrets across services is difficult.
- Service orchestration becomes complex.

With this change in architecture, the scope of each application is reduced significantly, making it easier to understand the needs of scaling each component. However, the increased number of independent services and added complexity also creates challenges for traditional operations (ops) teams, meaning organizations would also need to adapt.

Looking at the shift to DevOps

The shift to microservices has, in turn, led to a shift in how development teams are organized. Instead of a single large team managing a monolithic application, many teams each manage their own microservices. In traditional software development, a software development team would normally hand off the software once it was deemed complete. The handoff would be to an operations team, who would deploy the software and operate it in a production environment. As the number of services and teams grew, organizations found themselves growing their operations teams to unmanageable sizes, and quite often, those teams were still unable to keep up with the demands of the changing software.

This, in turn, led to an explosion of development teams that began the transition from the traditional development and operations organization toward the use of new hybrid DevOps teams. Using the DevOps approach, development teams write, test, build, package, deploy, and operate the code they develop. This ownership of the code through all stages of its life cycle empowers many developers and organizations to accelerate their feature development. This approach, of course, comes with different challenges:

- Increased dependencies across development teams mean it's possible that no one has a full picture of the entire application.
- Keeping track of changes across an organization can be difficult. This makes the answer to the "what caused this outage?" question more challenging to find.

Individual teams must become familiar with many more tools. This can lead to too much focus on the tools themselves, rather than on their purpose. The quick adoption of DevOps creates a new problem. Without the right amount of visibility across the systems managed by an organization, teams are struggling to identify the root causes of issues encountered. This can lead to longer and more frequent outages, severely impacting the health and happiness of people across organizations. Let's look at how the methods of observing systems have evolved to adapt to this changing landscape.

Reviewing the history of observability

In many ways, being able to understand what a computer is doing is both fun and challenging when working with software. The ability to understand how systems are behaving has gone through quite a few iterations since the early 2000s. Many different markets have been created to solve this need, such as systems monitoring, log management, and application performance monitoring. As is often the case, when new challenges come knocking, the doors of opportunity open to those willing to tackle those challenges. Over the same period, countless vendors and open source projects have sprung up to help people who are building and operating services in managing their systems. The term observability, however, is a recent addition to the software industry and comes from control theory.

Wikipedia (<https://en.wikipedia.org/wiki/Observability>) defines observability as:

"In control theory, observability is a measure of how well internal states of a system can be inferred from knowledge of its external outputs."

Observability is an evolution of its predecessors, built on lessons learned through years of experience and trial and error. To better understand where observability is today, it's important to understand where some of the methods used today by cloud-native application developers come from, and how they have changed over time. We'll start by looking at the following:

- Centralized logging
- Metrics and dashboards
- Tracing and analysis

Centralized logging

One of the first pieces of software a programmer writes when learning a new language is a form of observability: "Hello, World!". Printing some text to the terminal is usually one of the quickest ways to provide users with feedback that things are working, and that's why "Hello, World" has been a tradition in computing since the late 1960s.

One of my favorite methods for debugging is still to add print statements across the code when things aren't working. I've even used this method to troubleshoot an application distributed across multiple servers before, although I can't say it was my proudest moment, as it caused one of our services to go down temporarily because of a typo in an unfamiliar editor. Print statements are great for simple debugging, but unfortunately, this only scales so far.

Once an application is large enough or distributed across enough systems, searching through the logs on individual machines is not practical. Applications can also run on ephemeral machines that may no longer be present when we need those logs. Combined, all of this created a need to make the logs available in a central location for persistent storage and searchability, and thus centralized logging was born.

There are many available vendors that provide a destination for logs, as well as features around searching, and alerting based on those logs. There are also many open source projects that have tried to tackle the challenges of standardizing log formats, providing mechanisms for transport, and storing the logs. The following are some of these projects:

- **Fluentd** - <https://www.fluentd.org>
- **Logstash** - <https://github.com/elastic/logstash>
- **Apache Flume** - <https://flume.apache.org>

Centralized logging additionally provides the opportunity to produce metrics about the data across the entire system.

Using metrics and dashboards

Metrics are possibly the most well-known of the tools available in the observability space. Think of the temperature in a thermometer, the speed on the odometer of a car, or the time on a watch. We humans love measuring and quantifying things. From the early days of computing, being able to keep track of how resources were utilized was critical in ensuring that multi-user environments provided a good user experience for all users of the system.

Nowadays, measuring application and system performance via the collection of metrics is common practice in software development. This data is converted into graphs to generate meaningful visualizations for those in charge of monitoring the health of a system.

These metrics can also be used to configure alerting when certain thresholds have been reached, such as when an error rate becomes greater than an acceptable percentage. In certain environments, metrics are used to automate workflows as a reaction to changes in the system, such as increasing the number of application instances or rolling back a bad deployment. As with logging, over time, many vendors and projects provided their own solutions to metrics, dashboards, monitoring, and alerting. Some of the open source projects that focus on metrics are as follows:

- **Prometheus** - <https://prometheus.io>
- **StatsD** - <https://github.com/statsd/statsd>
- **Graphite** - <https://graphiteapp.org>
- **Grafana** - <https://github.com/grafana/grafana>

Let's now look at tracing and analysis.

Applying tracing and analysis

Tracing applications means having the ability to run through the application code and ensure it's doing what is expected. This can often, but not always, be achieved in development using a debugger such as GDB (<https://www.gnu.org/software/gdb/>) or PDB (<https://docs.python.org/3/library/pdb.html>) in Python. This becomes impossible when debugging an application that is spread across multiple services on different hosts across a network. Researchers at Google published a white paper on a large-scale distributed tracing system built internally: Dapper (<https://research.google/pubs/pub36356/>). In this paper, they describe the challenges of distributed systems, as well as the approach that was taken to address the problem. This research is the basis of distributed tracing as it exists today. After the paper was published, several open source projects sprung up to provide users with the tools to trace and visualize applications using distributed tracing:

- **OpenTracing** - <https://opentracing.io>
- **OpenCensus** - <https://opencensus.io>
- **Zipkin** - <https://zipkin.io>
- **Jaeger** - <https://www.jaegertracing.io>

As you can imagine, with so many tools, it can be daunting to even know where to begin on the journey to making a system observable. Users and organizations must spend time and effort upfront to even get started. This can be challenging when other deadlines are looming. Not only that, but the time investment needed to instrument an application can be significant depending on the complexity of the application, and the return on that investment sometimes isn't made clear until much later. The time and money invested, as well as the expertise required, can make it difficult to change from one tool to another if the initial implementation no longer fits your needs as the system evolves.

Such a wide array of methods, tools, libraries, and standards has also caused fragmentation in the industry and the open source community. This has led to libraries supporting one format or another. This leaves it up to the user to fix any gaps within the environments themselves. This also means there is effort required to maintain feature parity across different projects. All of this could be addressed by bringing the people working in these communities together.

With a better understanding of different tools at the disposal of application developers, their evolution, and their role, we can start to better appreciate the scope of what OpenTelemetry is trying to solve.

Understanding the history of OpenTelemetry

In early 2019, the **OpenTelemetry** project was announced as a merger of two existing open source projects: **OpenTracing** and **OpenCensus**. Although initially, the goal of this endeavor was to bring these two projects together, its ambition to provide an observability framework for cloud-native software goes much further than that. Since OpenTelemetry combines concepts of both OpenTracing and OpenCensus, let's first look at each of these projects individually. Please refer to the following Twitter link, which announced OpenTelemetry by combining both concepts:

[https://twitter.com/opencensusio/status/1111388599994318848.](https://twitter.com/opencensusio/status/1111388599994318848)

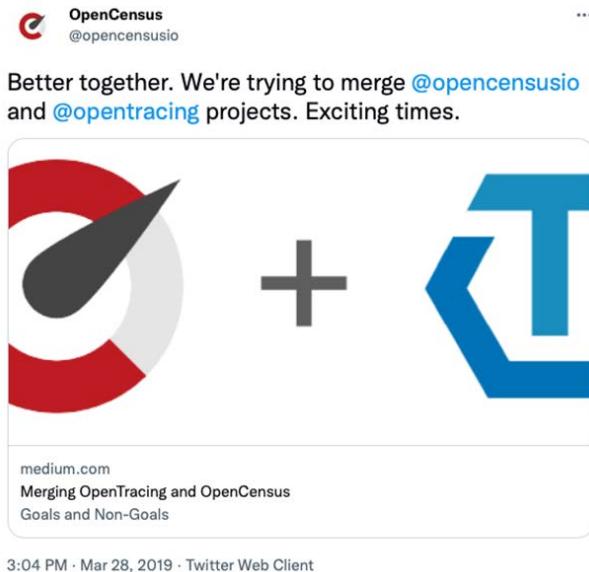


Figure 1.2 - Screenshot of the aforementioned tweet

OpenTracing

The OpenTracing (<https://opentracing.io>) project, started in 2016, was focused on solving the problem of increasing the adoption of distributed tracing as a means for users to better understand their systems. One of the challenges identified by the project was that adoption was difficult because of cost instrumentation and the lack of consistent quality instrumentation in third-party libraries. OpenTracing provided a specification for **Application Programming Interface (APIs)** to address this problem. This API could be leveraged independently of the implementation that generated distributed traces, therefore allowing application developers and library authors to embed calls to this API in their code. By default, the API would act as a no-op operation, meaning those calls wouldn't do anything unless an implementation was configured.

Let's see what this looks like in code. The call to an API to trace a specific piece of code resembles the following example. You'll notice the code is accessing a global variable to obtain a *Tracer* via the `global_tracer` method. A **Tracer** in OpenTracing, and in OpenTelemetry (as we'll discuss later in *Chapter 2, OpenTelemetry Signals – Tracing, Metrics, and Logging*, and *Chapter 4, Distributed Tracing – Tracing Code Execution*), is a mechanism used to generate trace data. Using a globally configured tracer means that there's no configuration required in this instrumentation code – it can be done completely separately. The next line starts a primary building block, `span`. We'll discuss this further in *Chapter 2, OpenTelemetry Signals – Tracing, Metrics, and Logging*, but it is shown here to give you an idea of how a *Tracer* is used in practice:

```
import opentracing
tracer = opentracing.global_tracer()
with tracer.start_active_span('doWork'):
    # do work
```

The default no-op implementation meant that code could be instrumented without the authors having to make decisions about how the data would be generated or collected at instrumentation time. It also meant that users of instrumented libraries, who didn't want to use distributed tracing in their applications, could still use the library without incurring a performance penalty by not configuring it. On the other hand, users who wanted to configure distributed tracing could choose how this information would be generated. The users of these libraries and applications would choose a *Tracer* implementation and configure it. To comply with the specification, a *Tracer* implementation only needed to adhere to the API defined (<https://github.com/opentracing/opentracing-python/blob/master/opentracing/tracer.py>), which includes the following methods:

- Start a new span.
- Inject an existing span's context into a carrier.
- Extract an existing span from a carrier.

Along with the specification for this API, OpenTracing also provides semantic conventions. These conventions describe guidelines to improve the quality of the telemetry emitted by instrumenting. We'll discuss semantic conventions further when exploring the concepts of OpenTelemetry.

OpenCensus

OpenCensus (<https://opencensus.io>) started as an internal project at Google, called Census, but was open sourced and gained popularity with a wider community in 2017. The project provided libraries to make the generation and collection of both traces and metrics simpler for application developers. It also provided the OpenCensus Collector, an agent run independently that acted as a destination for telemetry from applications and could be configured to process the data before sending it along to backends for storage and analysis. Telemetry being sent to the collector was transmitted using a wire format specified by OpenCensus. The collector was an especially powerful component of OpenCensus. As shown in *Figure 1.3*, many applications could be configured to send data to a single destination. That destination could then control the flow of the data without having to modify the application code any further:

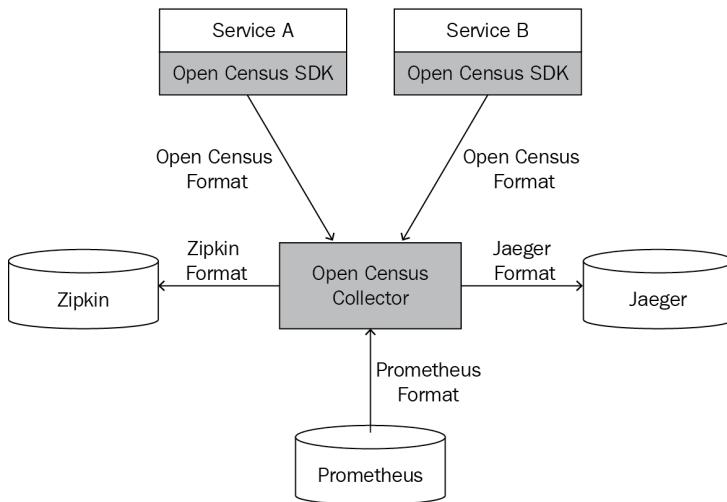


Figure 1.3 – OpenCensus Collector data flow

The concepts of the API to support distributed tracing in OpenCensus were like those of OpenTracing's API. In contrast to OpenTracing, however, the project provided a tightly coupled API and **Software Development Kit (SDK)**, meaning users could use OpenCensus without having to install and configure a separate implementation. Although this simplified the user experience for application developers, it also meant that in certain languages, the authors of third-party libraries wanting to instrument their code would depend on the SDK and all its dependencies. As mentioned before, OpenCensus also provided an API to generate application metrics. It introduced several concepts that would become influential in OpenTelemetry:

- **Measurement:** This is the recorded output of a measure, or a generated metric point.
- **Measure:** This is a defined metric to be recorded.

- **Aggregation:** This describes how the measurements are aggregated.
- **Views:** These combine measures and aggregations to determine how the data should be exported.

To collect metrics from their applications, developers defined a *measure* instrument to record *measurements*, and then configured a *view* with an *aggregation* to emit the data to a backend. The supported aggregations were *count*, *distribution*, *sum*, and *last value*.

As the two projects gained popularity, the pain for users only grew. The existence of both projects meant that it was unclear for users what project they should rely on. Using both together was not easy. One of the core components of distributed tracing is the ability to propagate context between the different applications in a distributed system, and this didn't work out of the box between the two projects. If a user wanted to collect traces and metrics, they would have to use OpenCensus, but if they wanted to use libraries that only supported OpenTracing, then they would have to use both – OpenTracing for distributed traces, and OpenCensus for metrics. It was a mess, and when there are too many standards, the way to solve all the problems is to invent a new standard!

It was a mess, and when there are too many standards, the way to solve all the problems is to invent a new standard! The following XKCD comic captures the sentiment very aptly:



Figure 1.4 – How standards proliferate comic (credit: XKCD, <https://xkcd.com/927/>)

Sometimes a new standard is a correct solution, especially when that solution:

- Is built using the lessons learned from its predecessors
- Brings together the communities behind other standards
- Supersedes two existing competing standards

The OpenCensus and OpenTracing organizers worked together to ensure the new standard would support a migration path for existing users of both communities, allowing the projects to eventually become deprecated. This would also make the lives of users easier by offering a single standard to use when instrumenting applications. There was no longer any need to guess what project to use!

Observability for cloud-native software

OpenTelemetry aims to standardize how applications are instrumented and how telemetry data is generated, collected, and transmitted. It also aims to give users the tools necessary to correlate that telemetry across systems, languages, and applications, to allow them to better understand their software. One of the initial goals of the project involved ensuring all the functionality that was key to both OpenCensus and OpenTracing users would become part of the new project. The focus on pre-existing users also leads to the project organizers establishing a migration path to ease the transition from OpenTracing and OpenCensus to OpenTelemetry. To accomplish its lofty goals, OpenTelemetry provides the following:

- An open specification
- Language-specific APIs and SDKs
- Instrumentation libraries
- Semantic conventions
- An agent to collect telemetry
- A protocol to organize, transmit, and receive the data

The project kicked off with the initial commit on May 1, 2019, and brought together the leaders from OpenCensus and OpenTracing. The project is governed by a governance committee that holds elections annually, with elected representatives serving on the committee for two-year terms. The project also has a technical committee that oversees the specification, drives project-wide discussion, and reviews language-specific implementations. In addition, there are various **special interest groups (SIGs)** in the project, focused on features or technologies supported by the project. Each language implementation has its own SIG with independent maintainers and approvers managing separate repositories with tools and processes tailored to the language. The initial work for the project was heavily focused on the open specification. This provides guidance for the language-specific implementations. Since its first commit, the project has received contributions from over 200 organizations, including observability leaders and cloud providers, as well as end users of OpenTelemetry. At the time of writing, OpenTelemetry has implementations in 11 languages and 18 special interest or working groups.

Since the initial merger of OpenCensus and OpenTracing, communities from additional open source projects have participated in OpenTelemetry efforts, including members of the Prometheus and OpenMetrics projects. Now that we have a better understanding of how OpenTelemetry was brought to life, let's take a deeper look at the concepts of the project.

Understanding the concepts of OpenTelemetry

OpenTelemetry is a large ecosystem. Before diving into the code, having a general understanding of the concepts and terminology used in the project will help us. The project is composed of the following:

- Signals
- Pipelines
- Resources
- Context propagation

Let's look at each of these aspects.

Signals

With its goal of providing an open specification for encompassing such a wide variety of telemetry data, the OpenTelemetry project needed to agree on a term to organize the categories of concern. Eventually, it was decided to call these **signals**. A signal can be thought of as a standalone component that can be configured, providing value on its own. The community decided to align its work into deliverables around these signals to deliver value to its users as soon as possible. The alignment of the work and separation of concerns in terms of signals has allowed the community to focus its efforts. The tracing and baggage signals were released in early 2021, soon followed by the metrics signal. Each signal in OpenTelemetry comes with the following:

- A set of specification documents providing guidance to implementors of the signal
- A data model expressing how the signal is to be represented in implementations
- An API that can be used by application and library developers to instrument their code
- The SDK needed to allow users to produce telemetry using the APIs
- Semantic conventions that can be used to get consistent, high-quality data
- Instrumentation libraries to simplify usage and adoption

The initial signals defined by OpenTelemetry were tracing, metrics, logging, and baggage. Signals are a core concept of OpenTelemetry and, as such, we will become quite familiar with them.

Specification

One of the most important aspects of OpenTelemetry is ensuring that users can expect a similar experience regardless of the language they're using. This is accomplished by defining the standards for what is expected of OpenTelemetry-compliant implementations in an open specification. The process used for writing the specification is flexible, but large new features or sections of functionality are often proposed by writing an **OpenTelemetry Enhancement Proposal (Otep)**. The OTEP is submitted for review and is usually provided along with prototype code in multiple languages, to ensure the proposal isn't too language-specific. Once an OTEP is approved and merged, the writing of the specification begins. The entire specification lives in a repository on GitHub (<https://github.com/open-telemetry/opentelemetry-specification>) and is open for anyone to contribute or review.

Data model

The data model defines the representation of the components that form a specific signal. It provides the specifics of what fields each component must have and describes how all the components interact with one another. This piece of the signal definition is particularly important to give clarity as to what use cases the APIs and SDKs will support. The data model also explains to developers implementing the standard how the data should behave.

API

Instrumenting applications can be quite expensive, depending on the size of your code base. Providing users with an API allows them to go through the process of instrumenting their code in a way that is vendor-agnostic. The API is decoupled from the code that generates the telemetry, allowing users the flexibility to swap out the underlying implementations as they see fit. This interface can also be relied upon by library and frameworks authors, and only configured to emit telemetry data by end users who wish to do so. A user who instruments their code by using the API and does not configure the SDK will not see any telemetry produced by design.

SDK

The SDK does the bulk of the heavy lifting in OpenTelemetry. It implements the underlying system that generates, aggregates, and transmits telemetry data. The SDK provides the controls to configure how telemetry should be collected, where it should be transmitted, and how. Configuration of the SDK is supported via in-code configuration, as well as via environment variables defined in the specification. As it is decoupled from the API, using the SDK provided by OpenTelemetry is an option for users, but it is not required. Users and vendors are free to implement their own SDKs if doing so will better fit their needs.

Semantic conventions

Producing telemetry can be a daunting task, since you can call anything whatever you wish, but doing so would make analyzing this data difficult. For example, if server A labels the duration of an `http.server.duration` request and server B labels it `http.server.request_length`, calculating the total duration of a request across both servers requires additional knowledge of this difference, and likely additional operations. One way in which OpenTelemetry tries to make this a bit easier is by offering semantic conventions, or definitions for different types of applications and workloads to improve the consistency of telemetry. Some of the types of applications or protocols that are covered by semantic conventions include the following:

- HTTP
- Database
- Message queues
- **Function-as-a-Service (FaaS)**
- **Remote procedure calls (RPC)**
- Process metrics

The full list of semantic conventions is quite extensive and can be found in the specification repository. The following figure shows a sample of the semantic convention for tracing database queries:

Attribute	Type	Description	Examples	Required
<code>db.system</code>	string	An identifier for the database management system (DBMS) product being used. See below for a list of well-known identifiers.	<code>other_sql</code>	Yes
<code>db.connection_string</code>	string	The connection string used to connect to the database. It is recommended to remove embedded credentials.	<code>Server=(localdb)\v11.0;Integrated Security=true;</code>	No
<code>db.user</code>	string	Username for accessing the database.	<code>readonly_user ; reporting_user</code>	No
<code>net.peer.ip</code>	string	Remote address of the peer (dotted decimal for IPv4 or RFC5952 for IPv6).	<code>127.0.0.1</code>	See below.
<code>net.peer.name</code>	string	Remote hostname or similar, see note below.	<code>example.com</code>	See below.
<code>net.peer.port</code>	int	Remote port number.	<code>80 ; 8080 ; 443</code>	Conditional [1]
<code>net.transport</code>	string	Transport protocol used. See note below.	<code>IP.TCP</code>	Conditional [2]

Table 1.1 – Database semantic conventions as defined in the OpenTelemetry specification (https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/trace/semantic_conventions/database.md#connection-level-attributes)

The consistency of telemetry data reported will ultimately impact the user of that data's ability to use this information. Semantic conventions provide both the guidelines of what telemetry should be reported, as well as how to identify this data. They provide a powerful tool for developers to learn their way around observability.

Instrumentation libraries

To ensure users can get up and running quickly, instrumentation libraries are made available by OpenTelemetry SIGs in various languages. These libraries provide instrumentation for popular open source projects and frameworks. For example, in Python, the instrumentation libraries include Flask, Requests, Django, and others. The mechanisms used to implement these libraries are language-specific and may be used in combination with auto-instrumentation to provide users with telemetry with close to zero code changes required. The instrumentation libraries are supported by the OpenTelemetry organization and adhere to semantic conventions.

Signals represent the core of the telemetry data that is generated by instrumenting cloud-native applications. They can be used independently, but the real power of OpenTelemetry is to allow its users to correlate data across signals to get a better understanding of their systems. Now that we have a general understanding of what they are, let's look at the other concepts of OpenTelemetry.

Pipelines

To be useful, the telemetry data captured by each signal must eventually be exported to a data store, where storage and analysis can occur. To accomplish this, each signal implementation offers a series of mechanisms to generate, process, and transmit telemetry. We can think of this as a pipeline, as represented in the following figure:

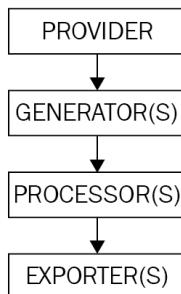


Figure 1.5 – Telemetry pipeline

The components in the telemetry pipeline are typically initialized early in the application code to ensure no meaningful telemetry is missed.

Important note

In many languages, the pipeline is configurable via environment variables. This will be explored further in *Chapter 7, Instrumentation Libraries*.

Once configured, the application generally only needs to interact with the generator to record telemetry, and the pipeline will take care of collecting and sending the data. Let's look at each component of the pipeline now.

Providers

The starting point of the telemetry pipeline is the provider. A provider is a configurable factory that is used to give application code access to an entity used to generate telemetry data. Although multiple providers may be configured within an application, a default global provider may also be made available via the SDK. Providers should be configured early in the application code, prior to any telemetry data being generated.

Telemetry generators

To generate telemetry at different points in the code, the telemetry generator instantiated by a provider is made available in the SDK. This generator is what most users will interact with through the instrumentation of their application and the use of the API. Generators are named differently depending on the signal: the tracing signal calls this a tracer, the metrics signal a meter. Their purpose is generally the same – to generate telemetry data. When instantiating a generator, applications and instrumenting libraries must pass a name to the provider. Optionally, users can specify a version identifier to the provider as well. This information will be used to provide additional information in the telemetry data generated.

Processors

Once the telemetry data has been generated, processors provide the ability to further modify the contents of the data. Processors may determine the frequency at which data should be processed or how the data should be exported. When instantiating a generator, applications and instrumenting libraries must pass a name to the provider. Optionally, users can specify a version identifier to the provider as well.

Exporters

The last step before telemetry leaves the context of an application is to go through the exporter. The job of the exporter is to translate the internal data model of OpenTelemetry into the format that best matches the configured exporter's understanding. Multiple export formats and protocols are supported by the OpenTelemetry project:

- OpenTelemetry protocol
- Console
- Jaeger
- Zipkin
- Prometheus
- OpenCensus

The pipeline allows telemetry data to be produced and emitted. We'll configure pipelines many times over the following chapters, and we'll see how the flexibility provided by the pipeline accommodates many use cases.

Resources

At their most basic, resources can be thought of as a set of attributes that are applied to different signals. Conceptually, a resource is used to identify the source of the telemetry data, whether a machine, container, or function. This information can be used at the time of analysis to correlate different events occurring in the same resource. Resource attributes are added to the telemetry data from signals at the export time before the data is emitted to a backend. Resources are typically configured at the start of an application and are associated with the providers. They tend to not change throughout the lifetime of the application. Some typical resource attributes would include the following:

- A unique name for the service: `service.name`
- The version identifier for a service: `service.version`
- The name of the host where the service is running: `host.name`

Additionally, the specification defines resource detectors to further enrich the data. Although resources can be set manually, resource detectors provide convenient mechanisms to automatically populate environment-specific data. For example, the **Google Cloud Platform (GCP)** resource detector (<https://www.npmjs.com/package/@opentelemetry/resource-detector-gcp>) interacts with the Google API to fill in the following data:

Google Cloud Platform resource detector attributes	
<code>cloud.provider</code>	<code>k8s.cluster.name</code>
<code>cloud.account.id</code>	<code>k8s.namespace.name</code>
<code>cloud.availability_zone</code>	<code>k8s.pod.name</code>
<code>host.id</code>	<code>container.name</code>

Table 1.2 – GCP resource detector attributes

Resources and resource detectors adhere to semantic conventions. Resources are a key component in making telemetry data-rich, meaningful, and consistent across an application. Another important aspect of ensuring the data is meaningful is context propagation.

Context propagation

One area of observability that is particularly powerful and challenging is context propagation. A core concept of distributed tracing, context propagation provides the ability to pass valuable contextual information between services that are separated by a logical boundary. Context propagation is what allows distributed tracing to tie requests together across multiple systems. OpenTelemetry, as OpenTracing did before it, has made this a core component of the project. In addition to tracing, context propagation allows for user-defined values (known as baggage) to be propagated. Baggage can be used to annotate telemetry across signals.

Context propagation defines a context API as part of the OpenTelemetry specification. This is independent of the signals that may use it. Some languages already have built-in context mechanisms, such as the `ContextVar` module in **Python 3.7+** and the `context` package in Go. The specification recommends that the context API implementations leverage these existing mechanisms. OpenTelemetry also provides for the interface and implementation of mechanisms required to propagate context across boundaries. The following abbreviated code shows how two services, A and B, would use the context API to share context:

```
from opentelemetry.propagate import extract, inject

class ServiceA:
    def client_request():
        inject(headers, context=current_context)
        # make a request to ServiceB and pass in headers

class ServiceB:
    def handle_request():
        # receive a request from ServiceA
        context = extract(headers)
```

In *Figure 1.6*, we can see a comparison between two requests from service A to service B. The top request is made without propagating the context, with the result that service B has neither the trace information nor the baggage that service A does. In the bottom request, this contextual data is injected when service A makes a request to service B, and extracted by service B from the incoming request, ensuring service B now has access to the propagated data:

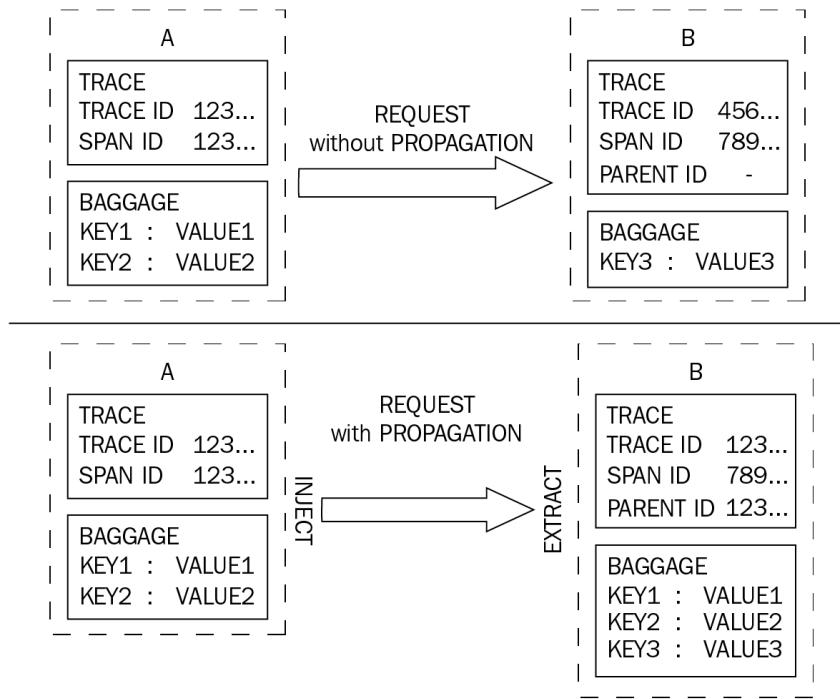


Figure 1.6 – Request between service A and B with and without context propagation

The propagation of context we have demonstrated allows backends to tie the two sides of the request together, but it also allows service B to make use of the dataset in service A. The challenge with context propagation is that when it isn't working, it's hard to know why. The issue could be that the context isn't being propagated correctly due to configuration issues or possibly a networking problem. This is a concept we'll revisit many times throughout the book.

Summary

In this chapter, we've looked at what observability is, and the challenges it can solve as regards the use of cloud-native applications. By exploring the different mechanisms available to generate telemetry and improve the observability of applications, we were also able to gain an understanding of how the observability landscape has evolved, as well as where some challenges remain.

Exploring the history behind the OpenTelemetry project gave us an understanding of the origin of the project and its goals. We then familiarized ourselves with the components forming tracing, metrics, logging signals, and pipelines to give us the terminology and building blocks needed to start producing telemetry using OpenTelemetry. This learning will allow us to tackle the first challenge of observability – producing high-quality telemetry. Understanding resources and context propagation will help us correlate events across services and signals to allow us to tackle the second challenge – connecting the data to better understand systems.

Let's now take a closer look at how this all works together in practice. In the next chapter, we will dive deeper into the concepts of distributed tracing, metrics, logs, and semantic conventions by launching a grocery store application instrumented with OpenTelemetry. We will then explore the telemetry generated by this distributed system.

2

OpenTelemetry Signals – Traces, Metrics, and Logs

Learning how first to instrument an application can be a daunting task. There's a fair amount of terminology to understand before jumping into the code. I always find that seeing the finish line helps me get motivated and stay on track. This chapter's goal is to see what telemetry generated by OpenTelemetry looks like in practice while learning about the theory. In this chapter, we will dive into the specifics of the following:

- Distributed tracing
- Metrics
- Logs
- Producing consistent quality data with semantic conventions

To help us get a more practical sense of the terminology and get comfortable with telemetry, we will look at the data using various open source tools that can help us to query and visualize telemetry.

Technical requirements

This chapter will use an application that is already instrumented with OpenTelemetry, a grocery store, and several backends to walk through the different concepts of the signals. The environment we will be launching relies on Docker Compose. The first step is to install Docker by following the installation instructions at <https://docs.docker.com/get-docker/>. Ensure Docker is running on your local system by using the following command:

```
$ docker version  
Client:  
Cloud integration: 1.0.14  
Version: 20.10.6  
API version: 1.41  
Go version: go1.16.3 ...
```

Next, let's ensure Compose is also installed by running the following command:

```
$ docker compose version  
Docker Compose version 2.0.0-beta.1
```

Important Note

Compose was added to the Docker client in more recent client versions. If the previous command returns an error, follow the instructions on the Docker website (<https://docs.docker.com/compose/install/>) to install Compose. Alternatively, you may want to try the `docker-compose` command to see if you already have an older version installed.

The following diagram shows an overview of the containers we are launching in the Docker environment to give you an idea of the components involved. The applications on the left are emitting telemetry processed by the Collector and forwarded to the telemetry backends. The diagram also shows the port number exposed by each container for future reference.

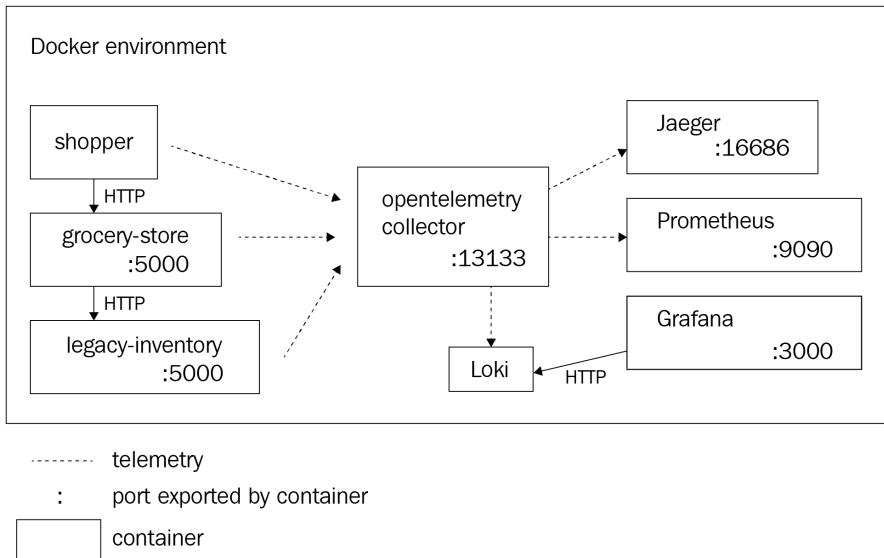


Figure 2.1 – Containers within Docker environment

This chapter briefly introduces the following open source projects that support the storage and visualization of OpenTelemetry data:

- Jaeger (<https://www.jaegertracing.io>)
- Prometheus (<https://prometheus.io>)
- Loki (<https://github.com/grafana/loki>)
- Grafana (<https://grafana.com/oss/grafana/>)

I strongly recommend visiting the website for each project to gain familiarity with the tools as we will use them throughout the chapter. Each of these tools will be revisited in *Chapter 10, Configuring Backends*. No prior knowledge of them is required to go through the examples, but they are pretty helpful to have in your toolbelt. The configuration files necessary to launch the applications in this chapter are available in the companion repository (<https://github.com/PacktPublishing/Cloud-Native-Observability>) in the chapter2 directory. The following downloads the repository using the git command:

```
$ git clone https://github.com/PacktPublishing/Cloud-Native-Observability
$ cd Cloud-Native-Observability/chapter02
```

To bring up the applications and telemetry backends, run the following command:

```
$ docker compose up
```

We will test the various tools to ensure each one is working as expected and is accessible from your browser. Let's start with **Jaeger** by accessing the following URL: `http://localhost:16686`. The following screenshot shows the interface you should see:

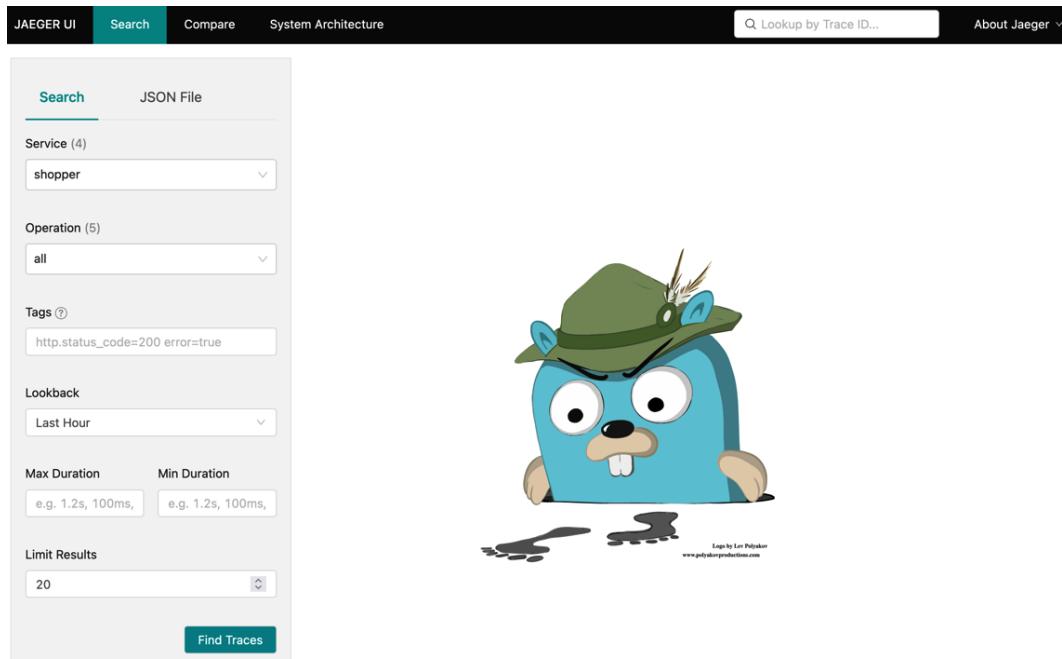


Figure 2.2 – The Jaeger web interface

The next backend this chapter will use for metrics is **Prometheus**; let's test the application by visiting `http://localhost:9090`. The following screenshot is a preview of the Prometheus web interface:

A screenshot of the Prometheus web interface. At the top, there is a dark header bar with the word "Prometheus" and several navigation links: "Alerts", "Graph", "Status", "Help", and "Classic UI". Below the header are several configuration checkboxes: "Use local time", "Enable query history", "Enable autocomplete" (which is checked), "Use experimental editor", "Enable highlighting" (which is checked), and "Enable linter" (which is checked). A search bar labeled "Expression (press Shift+Enter for newlines)" is followed by a blue "Execute" button. Below the search bar, there are two tabs: "Table" and "Graph", with "Graph" being the active tab. Underneath the tabs is a section for "Evaluation time" with arrows for navigating between time points. A message "No data queried yet" is displayed. In the bottom right corner of the main panel, there is a "Remove Panel" link. At the very bottom left, there is a blue "Add Panel" button.

Figure 2.3 – The Prometheus web interface

The last tool we need to ensure is working in our backend for logs is **Loki**. We will use **Grafana** as a dashboard to visualize the logs being emitted. Begin by visiting `http://localhost:3000/explore` to ensure Grafana is up; you should be greeted by an interface like the one in *Figure 2.4*:

A screenshot of the Grafana web interface. The top navigation bar includes "Explore" (with a dropdown arrow), a "Loki" dropdown, "Split" (button), "Last 1 hour" (dropdown), a search icon, "Run query" (button), and "Live" (button). On the left side, there is a sidebar with various icons: a magnifying glass for search, a plus sign for adding queries, a grid for dashboards, a refresh symbol, a bell for notifications, and a gear for settings. The main workspace shows a single query entry under the heading "(Loki)". It contains a "Log browser" input field with placeholder text "Enter a Loki query (run with Shift+Enter)", a "Query type" dropdown set to "Range", a "Range" button, an "Instant" button, a "Line limit" dropdown set to "auto", a "Resolution" dropdown set to "auto", and a "1/1" dropdown. Below the query input are buttons for "+ Add query", "Query history", and "Inspector".

Figure 2.4 – The Grafana web interface

The next application we will check is the OpenTelemetry Collector, which acts as the routing layer for all the telemetry produced by the example application. The Collector exposes a health check endpoint discussed in *Chapter 8, OpenTelemetry Collector*. For now, it's enough to know that accessing the endpoint will give us information about the health of the Collector, using the following curl command:

```
$ curl localhost:13133
{
  "status": "Server available", "upSince": "2021-10-03T15:42:02.734
  5149Z", "uptime": "9.3414709s"
}
```

Lastly, let's ensure the containers forming the grocery store demo application are running. To do this, we use curl again in the following commands to access an endpoint in the applications that returns a status showing the application's health. It's possible to use any other tool capable of making HTTP requests, including the browser, to accomplish this. The following checks the status of the grocery store:

```
$ curl localhost:5000/healthcheck
{
  "service": "grocery-store",
  "status": "ok"
}
```

The same command can be used to check the status of the inventory application by specifying port 5001:

```
$ curl localhost:5001/healthcheck
{
  "service": "inventory",
  "status": "ok"
}
```

The shopper application represents a client application and does not provide any endpoint to expose its health status. Instead, we can look at the logs emitted by the application to get a sense of whether it's doing the right thing or not. The following uses the docker logs command to look at the output from the application. Although it may vary slightly, the output should contain information about the shopper connecting to the grocery store:

```
$ docker logs -n 2 shopper
DEBUG:urllib3.connectionpool:http://grocery-store:5000 "GET /
products HTTP/1.1" 200 107
INFO:shopper:message="add orange to cart"
```

The same `docker logs` command can be used on any of the other containers if you're interested in seeing more information about them. Once you're done with the chapter, you can clean up all the containers by running `stop` to terminate the running containers, and `rm` to delete the containers themselves:

```
$ docker compose stop
$ docker compose rm
```

All the examples in this chapter will expect that the Docker Compose environment is already up and running. When in doubt, come back to this technical requirement section to ensure your environment is still running as expected. Now, let's see what these OpenTelemetry signals are all about, starting with traces.

Traces

Distributed tracing is the foundation behind the tracing signal of OpenTelemetry. A distributed trace is a series of event data generated at various points throughout a system tied together via a unique identifier. This identifier is propagated across all components responsible for any operation required to complete the request, allowing each operation to associate the event data to the originating request. The following diagram gives us a simplified example of what a single request may look like when ordering groceries through an app:

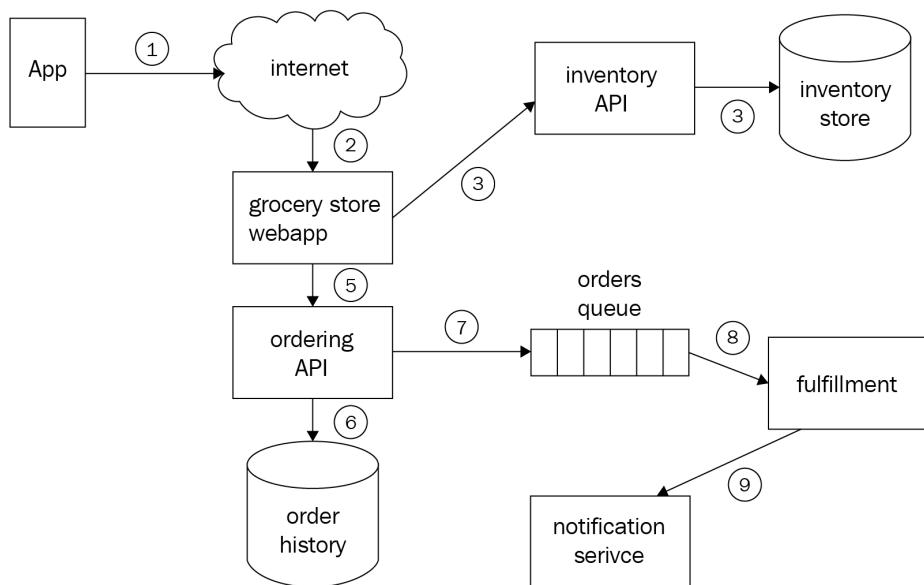


Figure 2.5 – Example request through a simplified ordering system

Each **trace** represents a unique request through a system that can be either synchronous or asynchronous. Synchronous requests occur in sequence with each unit of work completed before continuing. An example of a synchronous request may be of a client application making a call to a server and waiting or blocking until a response is returned before proceeding. In contrast, asynchronous requests can initiate a series of operations that can occur simultaneously and independently. An example of an asynchronous request is a server application submitting messages to a queue or a process that batches operations. Each operation recorded in a trace is represented by a **span**, a single unit of work done in the system. Let's see what the specifics of the data captured in the trace look like.

Anatomy of a trace

The definition of what constitutes a trace has evolved as various systems have been developed to support distributed tracing. The **World Wide Web Consortium (W3C)**, an international group that collaborates to move the web forward, assembled a working group in 2017 to produce a definition for tracing. In February 2020, the first version of the **Trace Context** specification was completed, with its details available on the W3C's website (<https://www.w3.org/TR/trace-context-1/>). OpenTelemetry follows the recommendation from the W3C in its definition of the **SpanContext**, which contains information about the trace and must be propagated throughout the system. The elements of a trace available within a span context include the following:

- A unique identifier, referred to as a **trace ID**, identifies the request through the system.
- A second identifier, the **span ID**, is associated with the span that last interacted with the context. This may also be referred to as the **parent identifier**.
- **Trace flags** include additional information about the trace, such as the sampling decision and trace level.
- Vendor-specific information is carried forward using a **Trace state** field. This allows individual vendors to propagate information necessary for their systems to interpret the tracing data. For example, if a vendor needs an additional identifier to be present in the trace information, this identifier could be inserted as `vendorA=123456` in the trace state field. Other vendors would add their own as needed, allowing traces to be shared across vendors.

A span can represent a method call or a subset of the code being called within a method. Multiple spans within a trace are linked together in a parent-child relationship, with each child span containing information about its parent. The first span in a trace is called the **root span** and is identified because it does not have a parent span identifier. The following shows a typical visualization of a trace and the spans associated with it. The horizontal axis indicates the duration of the entire trace operation. The vertical axis shows the order in which the operations captured by spans took place, starting with the first operation at the top:

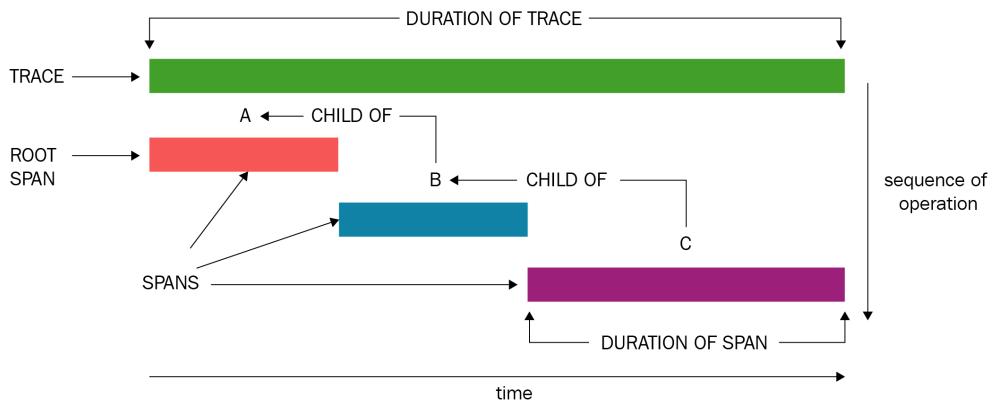


Figure 2.6 – Visual representation of a trace

Let's look closer at a trace by bringing up a sample generated from the telemetry produced by the grocery store application. Access the Jaeger web interface by opening a browser to the following URL: <http://localhost:16686/>.

Search for a trace by selecting a service from the drop-down and clicking the **Find Traces** button. The following screenshot shows the traces found for the shopper service:

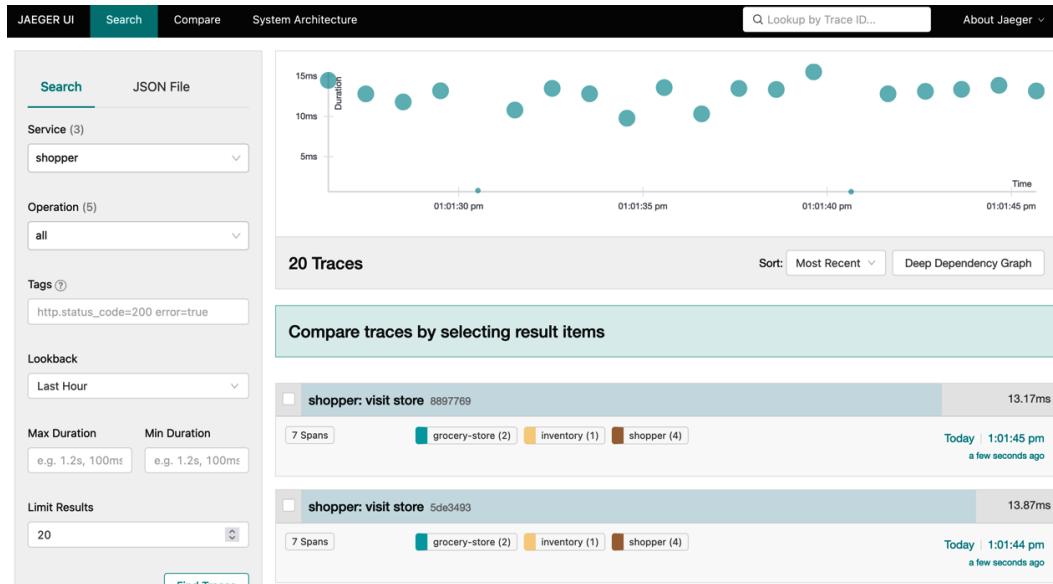


Figure 2.7 – Traces search result

To obtain details about a specific trace, select one of the search results by clicking on the row. The following screenshot, *Figure 2.8*, shows the details of the trace generated by a request through the grocery store applications. It includes the following:

1. The unique trace ID for this request. In OpenTelemetry, this is represented by a 128-bit integer. It's worth noting that other systems may represent this as a 64-bit integer. The integer is encoded into a string containing hexadecimal characters in many systems.
2. The start time for the request.
3. The total duration of the request through the system is calculated by subtracting the time the root span is finished from its start time.
4. A count of the number of services included in this request.
5. A count of spans recorded in this request is shown in Total Spans.

6. A hierarchical view of the spans in the trace.

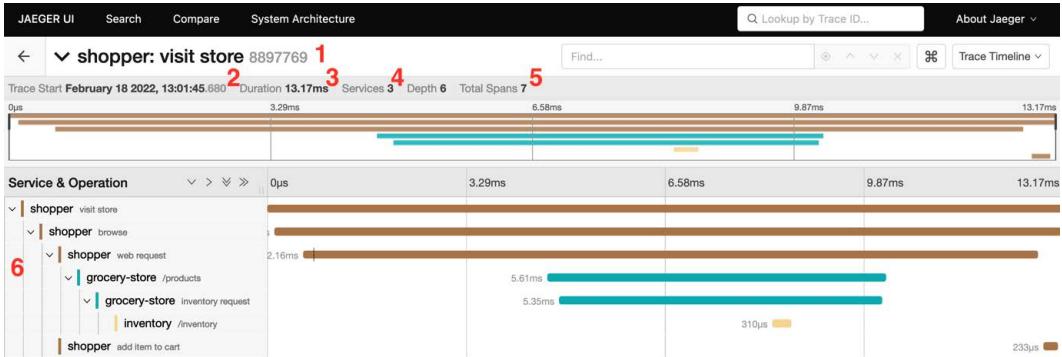


Figure 2.8 – A trace in Jaeger

The preceding screenshot gives us an immediate sense of where time may be spent as the system processes the request. It also provides us with a glimpse into what the underlying code may look like without ever opening an editor. Additional details are captured in spans; let's look at those now.

Details of a span

As mentioned previously, the work captured in a trace is broken into separate units or operations, each represented by a span. The span is a data structure containing the following information:

- A unique identifier
- A parent span identifier
- A name describing the work being recorded
- A start and end time

In OpenTelemetry, a span identifier is represented by a 64-bit integer. The start and end times are used to calculate the operation's duration. Additionally, spans can contain metadata in the form of key-value pairs. In the case of Jaeger and Zipkin, these pairs are referred to as **tags**, whereas OpenTelemetry calls them **attributes**. The goal is to enrich the data provided with the additional context in both cases.

Look for the following details in *Figure 2.9*, which shows the detailed view of a specific span as shown in Jaeger:

1. The name identifies the operation represented by this span. In this case, **/inventory** is the operation's name.
2. **SpanID** is the unique 64-bit identifier represented in hex-encoded formatting.
3. **Start Time** is when the operation recorded its start time relative to the start of the request. In the case shown here, the operation started 8.36 milliseconds after the beginning of the request.
4. **Duration** is the time it took for the operation to complete and is calculated using the start and end times recorded in the span.
5. **The Service name** identifies the application that triggered the operation and recorded the telemetry.
6. **Tags** represent additional information about the operation being recorded.
7. **Process** shows information about the application or process fulfilling the requested operation.



Figure 2.9 – Span details

Many of the tags captured in the span shown previously rely on semantic conventions, which will be discussed further in this chapter.

Additional considerations

When producing distributed traces in a system, it's worth considering the additional visibility's tradeoffs. Generating tracing information can potentially incur performance overhead at the application level. It can result in added latency if tracing information is gathered and transmitted inline. There is also memory overhead to consider, as collecting information inevitably allocates resources. These concerns can be largely mitigated using configuration available in OpenTelemetry, as we'll see in *Chapter 4, Distributed Tracing – Tracing Code Execution*.

Depending on where the data is sent, additional costs, such as bandwidth or storage, can also become a factor. One of the ways to mitigate these costs is to reduce the amount of data produced by sampling only a certain amount of the data. We will dive deeper into sampling in *Chapter 12, Sampling*.

Another challenging aspect of producing distributed tracing data is ensuring that all the services correctly propagate the context. Failing to propagate the trace ID across the system means that requests will be broken into multiple traces, making them difficult to use or not helpful at all.

The last thing to consider is the effort required to instrument an application correctly. This is a non-trivial amount of effort, but as we'll see in future chapters, OpenTelemetry provides instrumentation libraries to make this easier.

Now that we have a deeper understanding of traces, let's look at metrics.

Metrics

Just as distributed traces do, metrics provide information about the state of a running system to developers and operators. The data collected via metrics can be aggregated over time to identify trends and patterns in applications graphed through various tools and visualizations. The term *metrics* has a broad range of applications as they can capture low-level system metrics such as CPU cycles, or higher-level details such as the number of blue sweaters sold today. These examples would be helpful to different groups in an organization.

Additionally, metrics are critical to monitoring the health of an application and deciding when an on-call engineer should be alerted. They form the basis of **service level indicators (SLIs)** (https://en.wikipedia.org/wiki/Service_level_indicator) that measure the performance of an application. These indicators are then used to set **service level objectives (SLOs)** (https://en.wikipedia.org/wiki/Service-level_objective) that organizations use to calculate error budgets.

Important Note

SLIs, SLOs, and **service level agreements (SLAs)** are essential topics in production environments where third-party dependencies can impact the availability of your service. There are entire books dedicated to the issue that we will not cover here. The Google **site reliability engineering (SRE)** book is a great resource for this: <https://sre.google/sre-book/service-level-objectives/>.

The metrics signal of OpenTelemetry combines various existing open source formats into a unified data model. Primarily, it looks to *OpenMetrics*, *StatsD*, and *Prometheus* for existing definitions, requirements, and usage, wanting to ensure the use-cases of each of those communities are understood and addressed by the new standard.

Anatomy of a metric

Just about anything can be a metric; record a value at a given time, and you have yourself a metric. The common fields a metric contains include the following:

- A **name** identifies the metric being recorded.
- A **data point value** may be an integer or a floating-point value. Note that in the case of a histogram or a summary, there is more than one value associated with the metric.
- Additional **dimension** information about the metric. The representation of these dimensions varies depending on the metrics backend. In Prometheus, these dimensions are represented by labels, whereas in StatsD, it is common to add a prefix in the metric's name. In OpenTelemetry, dimensions are added to metrics via attributes.

Let's look at data produced by metrics sent from the demo application. Access the Prometheus interface via a browser and the following URL: `http://localhost:9090`. The user interface for Prometheus allows us to query the time-series database by using the metric's name. The following screenshot contains a table showing the value of the `request_counter` metric. Look for the following details in the resulting table:

1. The name of the metric, in this case, `request_counter`.
2. The dimensions recorded for this metric are displayed in curly braces as key-value pairs with the key emboldened. In the example shown, the `service_name` label caused two different metrics to be recorded, one for the `shopper` service and another for the `store` service.

3. A reported value, in this example, is an integer. This value may be the last received or a calculated current value depending on the metric type.

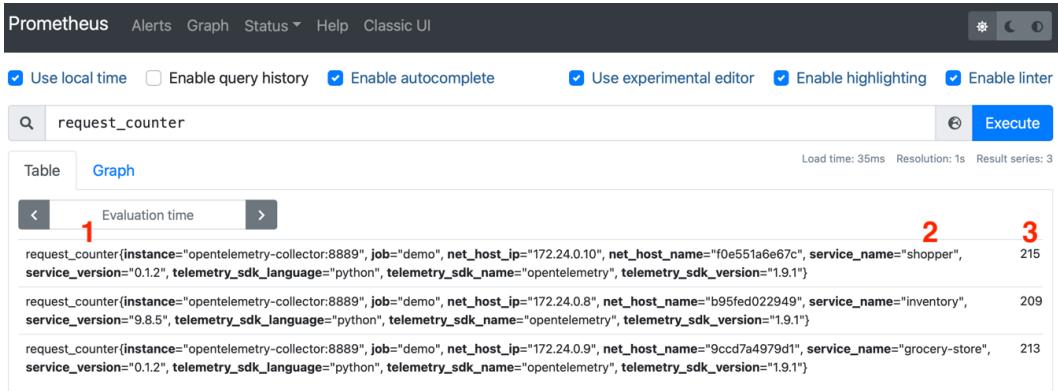


Figure 2.10 – Table view of metric in Prometheus

The table view shows the current value as cumulative. An alternative representation of the recorded metric is shown in the following figure. As the data received by Prometheus is stored over time, a line graph can be generated. Click the **Graph** tab of the interface to see what the data in a chart looks like:

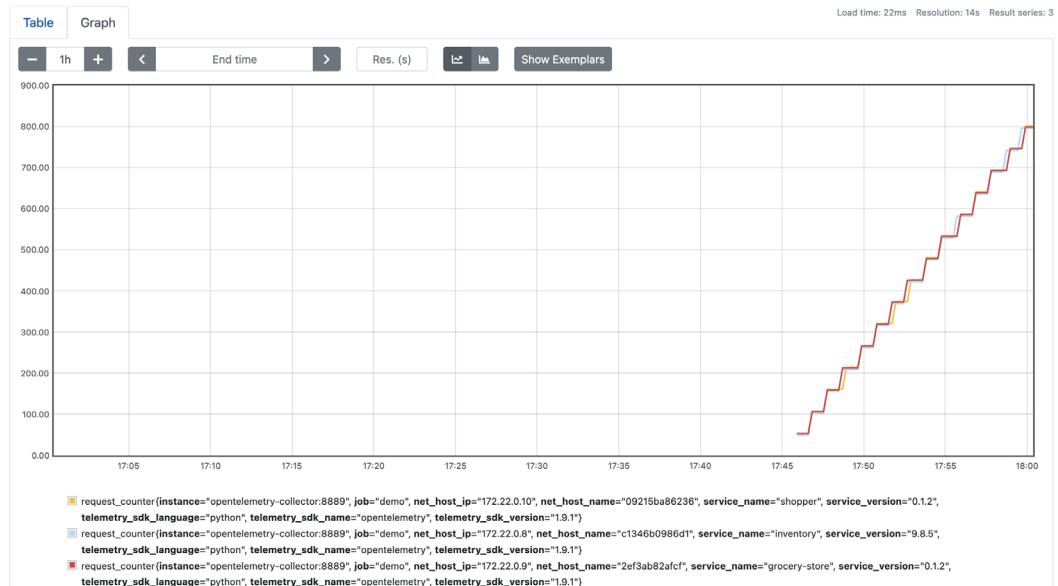


Figure 2.11 – Graph view of the same metric in Prometheus

By looking at the values for the metric over time, we can deduce additional information about the service, for example, its start time or trends in its usage. Visualizing metrics also provides opportunities to identify anomalies.

Data point types

A metric is a more generic term that encapsulates different measurements that can be used to represent a wide array of information. As such, the data is captured using various data point types. The following diagram compares different kinds of data points that can be captured within a metric:

Histogram	Values	Summary	Values
Min	0	Count	10
Max	100	Min	0.99ms
Count	20	Max	3.44ms
Interval	10	Sum	8.56ms
Distribution		Quantiles	
0-10	1	p50	1.37ms
10-20	1	p75	2.82ms
20-30	1	p99	3.44ms
30-40	2	Gauge	Value
40-50	4	Last value	37.5
50-60	4	Counter	Value
60-70	3	Last increment	19
70-80	2		
80-90	1		
90-100	1		

Figure 2.12 – Comparison of counter, gauge, histogram, and summary data points

Each data point type can be used in different scenarios and has slightly different meanings. It's worth noting that even though competing standards provide support for types using the same name, their definition may vary. For example, a counter in StatsD (https://github.com/statsd/statsd/blob/master/docs/metric_types.md#counting) resets every time the value has been flushed, whereas, in Prometheus (https://prometheus.io/docs/concepts/metric_types/#counter), it keeps its cumulative value until the process recording the counter is restarted. The following definitions describe how data point types are represented in the OpenTelemetry specification:

- A **sum** measures incremental changes to a recorded value. This incremental change is either monotonic or non-monotonic and must be associated with an aggregation temporality. The temporality can be either of the following:
 - Delta** aggregation: The reported values contain the change in value from its previous recording.
 - Cumulative** aggregation: The value reported includes the previously reported sum in addition to the delta being reported.

Important Note

A cumulative sum will reset when an application restarts. This is useful to identify an event in the application but may be surprising if it's not accounted for.

The following diagram shows an example of a sum counter reporting the number of visits over a period of time. The table on the right-hand side shows what values are to be expected depending on the type of temporal aggregation chosen:

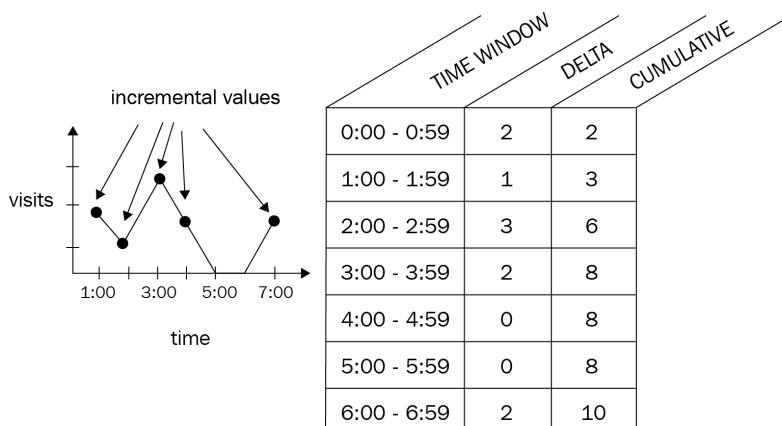


Figure 2.13 – Sum showing delta and cumulative aggregation values

A sum data point also includes the time window for calculating the sum.

- A **gauge** represents non-monotonic values that only measure the last or current known value at observation. This likely means some information is missing, but it may not be relevant. For example, the following diagram represents temperatures recorded at an hourly interval. More specific data points could provide greater granularity as to the rise and fall of the temperature. These incremental changes in the temperature may not be required if the goal is to observe trends over weeks or months.

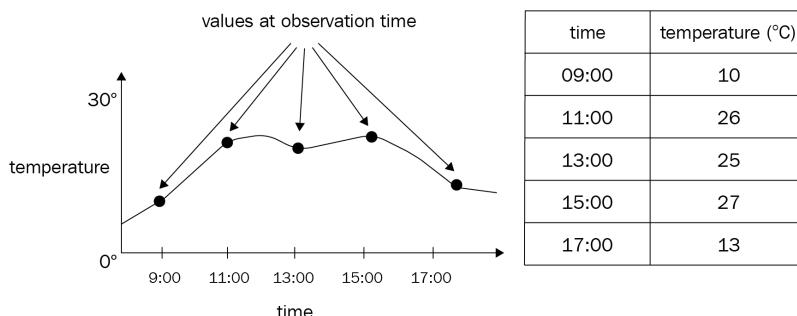


Figure 2.14 – Gauge values recorded

Unlike gauge definitions in other specifications, a gauge in OpenTelemetry is never incremented or decremented; it is only ever set to the value being recorded.

A timestamp of the observation time must be included with the data point.

- A **histogram** data point provides a compressed view into a more significant number of data points by grouping the data into a distribution and summarizing the data, rather than reporting individual measurements for every detail represented. The following diagram shows sample histogram data points representing a distribution of response durations.

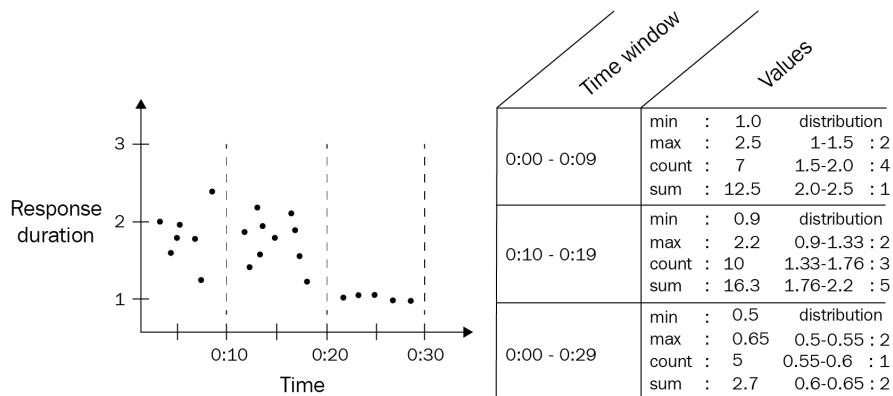


Figure 2.15 – Histogram data points

Like sums, histograms also support a delta or a cumulative aggregation and must contain a time window for the recorded observation. Note that in the case of cumulative aggregation, the data points captured in the distribution will continue to accumulate with each recording.

- The **summary** data type provides a similar capability to histograms, but it's specifically tailored around providing quantiles of a distribution. A **quantile**, sometimes also referred to as **percentile**, is a fraction between zero and one, representing a percentage of the total number of values recorded that falls under a certain threshold. For example, consider the following 10 response times in milliseconds: 1.1, 2.9, 7.5, 8.3, 9, 10, 10, 10, 10, 25. The 0.9-quantile, or the 90th percentile, equals 10 milliseconds.

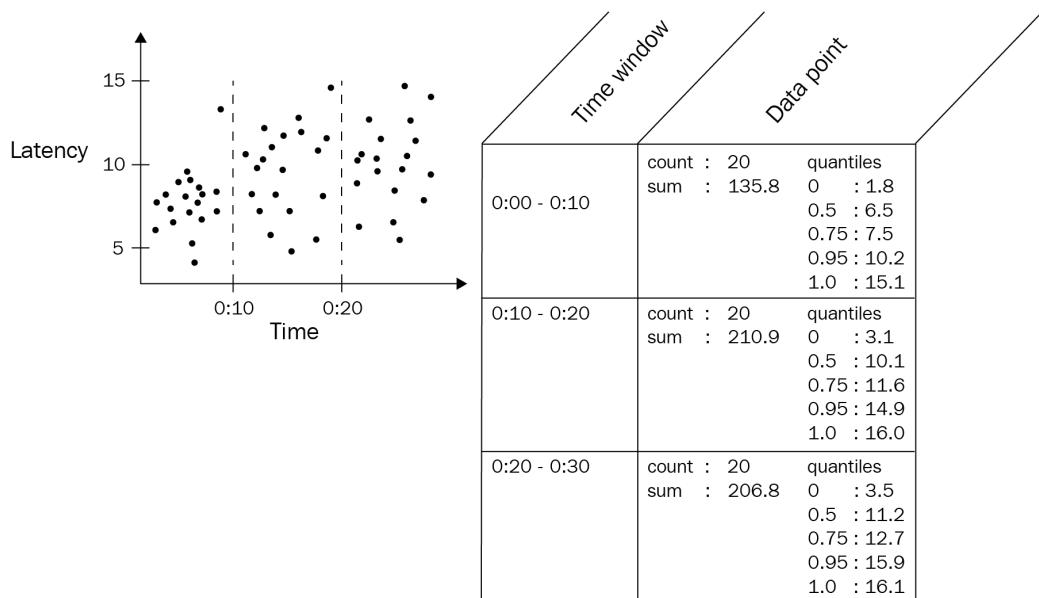


Figure 2.16 – Summary data points

A summary is somewhat similar to a histogram, where the histogram contains a maximum and a minimum value; the summary includes a 1.0-quantile and 0.0-quantile to represent the same information. The 0.5-quantile, also known as median, is often expressed in the summary. For a summary data point, the quantile calculations happen in the producer of the telemetry, which can become expensive for applications. OpenTelemetry supports summaries to provide interoperability with OpenMetrics (<https://openmetrics.io>) and Prometheus and prefers the usage of a histogram, which moves the calculation of quantiles to the receiver of the telemetry. The following screenshot shows histogram values recorded by the inventory service for the `http_request_duration_milliseconds_bucket` metric stored in Prometheus. The data shown represents requests grouped into buckets. Each bucket represents the request duration in milliseconds:

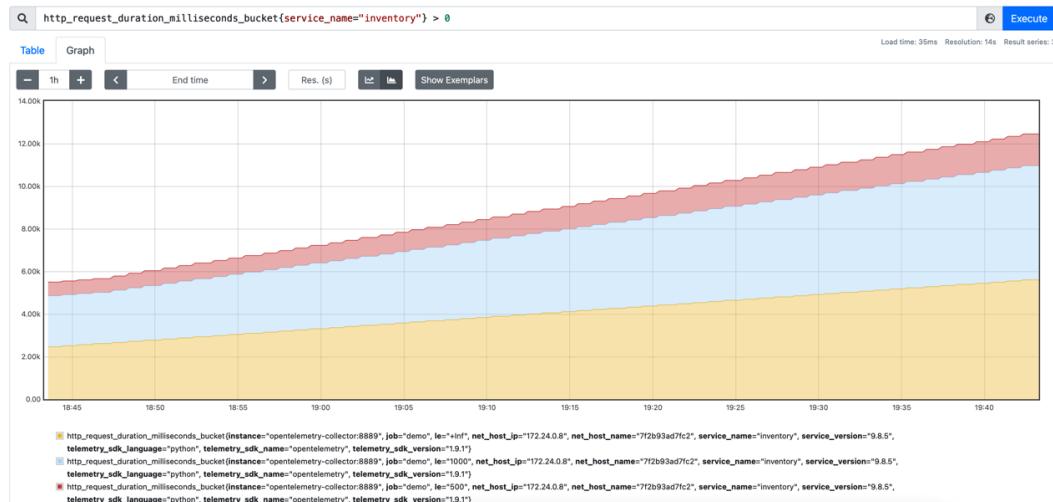


Figure 2.17 – Histogram value in Prometheus

The count of requests per bucket can then calculate quantiles for further analysis. Now that we're familiar with the different types of metric data points, let's see how metrics can be combined with tracing to provide additional insights.

Exemplars

Metrics are often helpful on their own, but when correlated with tracing information, they provide much more context and depth on the events occurring in a system. **Exemplars** offer a tool to accomplish this in OpenTelemetry by enabling a metric to contain information about an active span. Data points defined in OpenTelemetry include an `exemplar` field as part of their definition. This field contains the following:

- A trace ID of the current span in progress
- The span ID of the current span in progress
- A timestamp of the event measured
- A set of attributes associated with the exemplar
- The value being recorded

The direct correlation that exemplars provide replaces the guesswork that involves cobbling metrics and traces with timestamps today. Although exemplars are already defined in the stable metrics section of the OpenTelemetry protocol, the implementation of exemplars is still under active development at the time of writing.

Additional considerations

A concern that often arises with any telemetry is the importance of managing cardinality. **Cardinality** refers to the uniqueness of a value in a set. While counting cars in a parking lot, the number of wheels will likely offer a meager value and low cardinality result as most cars have four wheels. The color, make, and model of cars produces higher cardinality. The license plate, or vehicle identification number, results in the highest cardinality, providing the most valuable data to know in an event concerning a specific vehicle. For example, if the lights have been left on and the owners should be notified, calling out for the person with a four-wheeled car won't work nearly as well as calling for a specific license plate. However, the count of cars with specific license plates will always be one, making the counter itself somewhat useless.

One of the challenges with high-cardinality data is the increased storage cost. Specifically, in the case of metrics, it's possible to significantly increase the number of metrics being produced and stored by adding a single attribute or label. Suppose an application creating a counter for each request processed uses a unique identifier as the metric's name. In that case, the producer or receiver may translate this into a unique time series for each request. This results in a sudden and unexpected increase in load in the system. This is sometimes referred to as **cardinality explosion**.

When choosing attributes associated with produced metrics, it's essential to consider the scale of the services and infrastructure producing the telemetry. Some questions to keep in mind are as follows:

- Will scaling components of the system increase the number of metrics in a way that is understood? When a system scales, the last thing anyone wants is for an unexpected spike in metrics to cause outages.
- Are any attributes specific to instances of an application? This could cause problems in the case of a crashing application.

Using labels with finite and knowable values (for example, countries rather than street names) may be preferable depending on how the data is stored. When choosing a solution, understanding the storage model and limits of the telemetry backend must also be considered.

Logs

Although logs have evolved, what constitutes a **log** is quite broad. Also known as log files, a log is a record of events written to output. Traditionally, logs would be written to a file on disk, searching through as needed. A more recent practice is to emit logs to remote services using the network. This provides long-term storage for the data in a location and improves searchability and aggregation.

Anatomy of a log

Many applications define their formats for what constitutes a log. There are several existing standard formats. An example includes the Common Log Format often used by web servers. It's challenging to identify commonalities across formats, but at the very least, a log should consist of the following:

- A **timestamp** recording the time of the event
- The **message** or payload representing the event

This message can take many forms and include various application-specific information. In the case of structured logging, the log is formatted as a series of key-value pairs to simplify identifying the different fields contained within the log. Other formats record logs in a specific order with a separating character instead. The following shows an example log emitted by the standard formatter in **Flask**, a Python web framework that shows the following:

- A timestamp is enclosed in square brackets.
- A space-delimited set of elements forms the message logged, including the client IP, the HTTP method used to make a request, the request's path, the protocol version, and the response code:

```
172.20.0.9 - - [11/Oct/2021 18:50:25] "GET /inventory HTTP/1.1"
200 -
```

The previous sample is an example of the Common Log Format mentioned earlier. The same log may look something like this as a structured log encoded as JSON:

```
{
  "host": "172.20.0.9",
  "date": "11/Oct/2021 18:50:25",
  "method": "GET",
  "path": "/inventory",
  "protocol": "HTTP/1.1",
  "status": 200
}
```

As you can see with structured logs, identifying the information is more intuitive if you're not already familiar with the type of logs produced. Let's see what logs our demo application produces by looking at the Grafana interface, at `http://localhost:3000/explore`.

This brings us to the `explore` view, which allows us to search through telemetry generated by the demo application. Ensure that **Loki** is selected from the data source drop-down in the top left corner. Filter the logs using the `{job="shopper"}` query to retrieve all the logs generated by the shopper application. The following screenshot shows a log emitted to the Loki backend, which contains the following:

1. The name of the application is under the `job` label.
2. A timestamp of the log is shown both as a timestamp and as a nanosecond value.

3. The body of the logged event.
4. Additional labels and values associated with the event.

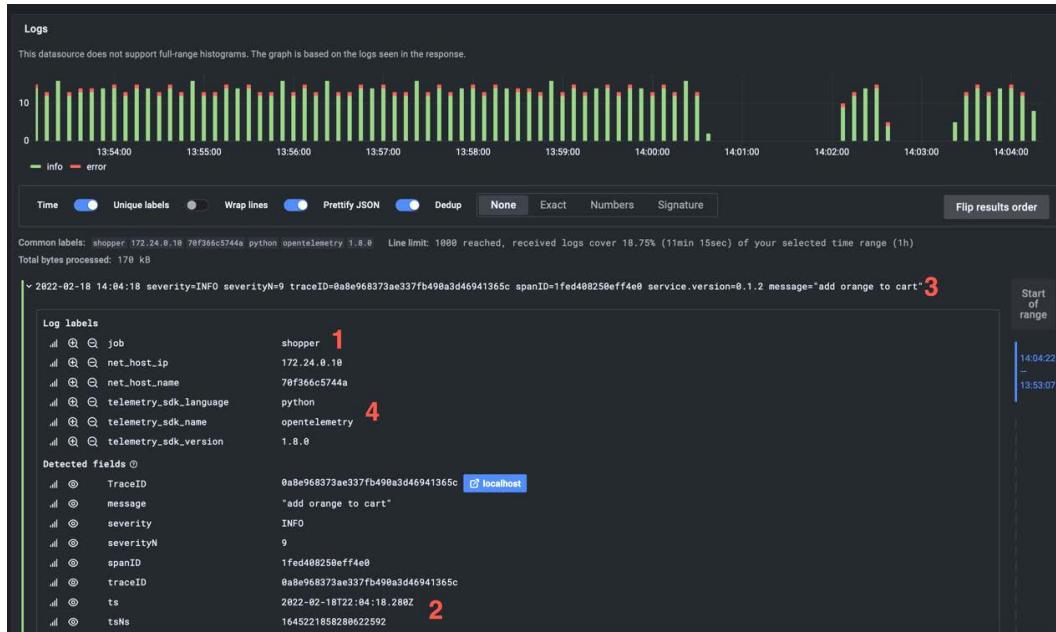


Figure 2.18 – Log shown in Grafana

Now that we can search for logs, let's see how we can combine the information provided by logs with other signals via correlation to give us more context.

Correlating logs

In the same way that information provided by metrics can be augmented by combining them with other signals, logs too can provide more context by embedding tracing information. As we'll see in *Chapter 6, Logging - Capturing Events*, one of the goals of the logging signal in OpenTelemetry is to provide correlation capability to already existing logging libraries. Logs recorded via OpenTelemetry contain the trace ID and span ID for any span active at the time of the event. The following screenshot shows the details of a log record containing the `traceID` and `spanID` attributes:

Log labels	
job	shopper
net_host_ip	172.24.0.10
net_host_name	70f366c5744a
telemetry_sdk_language	python
telemetry_sdk_name	opentelemetry
telemetry_sdk_version	1.8.0
Detected fields	
TraceID	77891c335f8f335fc91869121f5a440a
message	"oops, invalid request"
severity	ERROR
severityN	17
spanID	0bbac7e895961880
traceID	77891c335f8f335fc91869121f5a440a
ts	2022-02-18T22:16:30.944Z
tsNs	1645222590944079872

Figure 2.19 – Log containing trace ID

Using these attributes can then reveal the specific request that triggered this event. The following screenshot demonstrates what the corresponding trace looks like in Jaeger. If you'd like to try for yourself, copy the `traceID` attribute into the **Lookup by Trace ID** field to search for the trace:

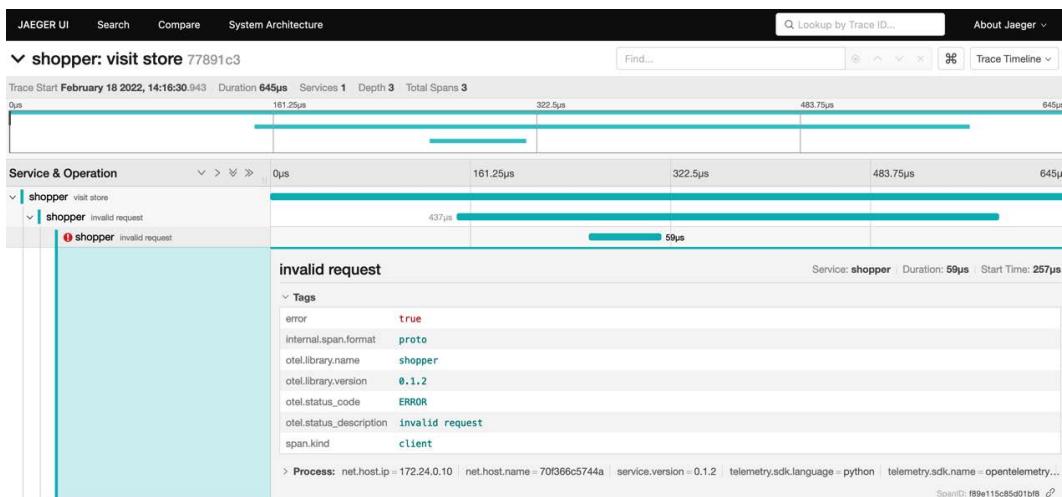


Figure 2.20 – Corresponding trace in Jaeger

The correlation demonstrated in the previous example makes exploring events faster and less error-prone. As we will see in *Chapter 6, Logging - Capturing Events*, the OpenTelemetry specification provides recommendations for what information should be included in logs being emitted. It also provides guidelines for how existing formats can map their values with OpenTelemetry.

Additional considerations

The free form of traditional logs makes them incredibly convenient to use without considering their structure. If you want to add any data to the logs, just call a function and print anything you'd like; it'll be great. However, this can pose some challenges. One of these challenges is the opportunity for leaking potentially private information into the logs and transmitting it to a centralized logging platform. This problem applies to all telemetry, but it's particularly easy to do with logs. This is especially true when logs contain debugging information, which may include data structures with passwords fields or private keys. It's good to review any logging calls in the code to ensure the logged data does not contain information that should not be logged.

Logs can also be overly verbose, which can cause unexpected volumes to be generated. This may make sifting through the logs for useful information difficult, if not impossible, depending on the size of the environment. It can also lead to unanticipated costs when using centralized logging platforms. Specific libraries or frameworks generate much debugging information. Ensuring the correct severity level is configured goes a long towards addressing this concern. However, it's hard to predict just how much data will be needed upfront. On more than one occasion, I've responded to alerts in the middle of the night, wishing for a more verbose log level to be configured.

Semantic conventions

High-quality telemetry allows the data consumer to find answers to questions when needed. Sometimes critical operations can lack instrumentation causing blind spots in the observability of a system. Other times, the processes are instrumented, but the data is not rich enough to be helpful. The OpenTelemetry project attempts to solve this through semantic conventions defined in the specification. These conventions cover the following:

- Attributes that should be present for traces, metrics, and logs.
- Resource attribute definitions for various types of workloads, including hosts, containers, and functions. The resource attributes described by the specification also include characteristics specific to multiple popular cloud platforms.

- Recommendations for what telemetry should be emitted by components participating in various scenarios such as messaging systems, client-server applications, and database interactions.

These semantic conventions help ensure that the data generated when following the OpenTelemetry specification is consistent. This simplifies the work of folks instrumenting applications or libraries by providing guidelines for what should be instrumented and how. It also means that anyone analyzing telemetry produced by standard-compliant code can understand the meaning of the data by referencing the specification for additional information.

Following semantic conventions recommendations from a specification in a Markdown document can be challenging when writing code. Thankfully, OpenTelemetry also provides some tools to help.

Adopting semantic conventions

Semantic conventions are great, but it makes sense to turn the recommendations into code to make it practical for developers to use them. The OpenTelemetry specification repository provides a folder that contains the semantic conventions described as YAML for this specific reason (https://github.com/open-telemetry/opentelemetry-specification/tree/main/semantic_conventions). These are combined with the semantic conventions generator (<https://github.com/open-telemetry/build-tools/blob/v0.7.0/semantic-conventions/>) to produce code in various languages. This code is shipped as independent libraries in some languages, helping guide developers. We will repeatedly rely upon the semantic conventions package in Python in further chapters as we instrument application code.

Schema URL

A challenge of semantic conventions is that as telemetry and observability evolve, so will the terminology used to describe events that we want to observe. An example of this happened when the db.hbase.namespace and db.cassandra.keyspace keys were renamed to use db.name instead. Such a change would cause problems for anyone already using this field as part of their analysis, or even alerting. To ensure the semantic conventions can evolve as needed while remaining backward-compatible with existing instrumentation, the OpenTelemetry community introduced the **schema URL**.

Important Note

The OpenTelemetry community understands the importance of backward compatibility in instrumentation code. Going back and re-instrumenting an application because of a new version of a telemetry library is a pain. As such, a significant amount of effort has gone into ensuring that components defined in OpenTelemetry remain interoperable with previous versions. The project defines its versioning and stability guarantees as part of the specification (<https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/versioning-and-stability.md>).

The schema URL is a field added to the telemetry generated for logs, metrics, resources, and traces tying the emitted telemetry to a version of the semantic conventions. This field allows the producers and consumers of telemetry to understand how to interpret the data. The schema also provides instructions for converting data from one version to another, as per the following example:

1.8.0 schema

```
file_format: 1.0.0
schema_url: https://opentelemetry.io/schemas/1.8.0
versions:
  1.8.0:
    spans:
      changes:
        - rename_attributes:
            attribute_map:
              db.cassandra.keyspace: db.name
              db.hbase.namespace: db.name
  1.7.0:
  1.6.1:
```

Continuing with the previous example, imagine a producer of Cassandra telemetry is emitting `db.cassandra.keyspace` as the name for a Cassandra database and specifying the schema as `1.7.0`. It sends the data to a backend that implements schema `1.8.0`. By reading the schema URL and implementing the appropriate translation, the backend can produce telemetry in its expected version, which is powerful! Schemas decouple systems involved in telemetry, providing them with the flexibility to evolve independently.

Summary

This chapter allowed us to learn or review some concepts that will assist us when instrumenting applications using OpenTelemetry. We looked at the building blocks of distributed tracing, which will come in handy when we go through instrumenting our first application with OpenTelemetry in *Chapter 4, Distributed Tracing – Tracing Code Execution*. We also started analyzing tracing data using tools that developers and operators make use of every day.

We then switched to the metrics signal; first, looking at the minimal contents of a metric, then comparing different data types commonly used to produce metrics and their structures. Discussing exemplars gave us a brief introduction to how correlating metrics with traces can create a more complete picture of what is happening within a system by combining telemetry across signals.

Looking at log formats and searching through logs to find information about the demo application allowed us to get familiar with yet another tool available in the observability practitioner's toolbelt.

Lastly, by leveraging semantic conventions defined in OpenTelemetry, we can begin to produce consistent, high-quality data. Following these conventions removes the painful task of naming things, which everyone in the software industry agrees is hard for producers of telemetry. Additionally, these conventions remove the guesswork when interpreting the data.

Knowing the theory and concepts behind instrumentation and telemetry is excellent to provide us with the tools to do all the instrumentation work ourselves. Still, what if I were to tell you it may not be necessary to instrument every call in every library manually? The next chapter will cover how auto-instrumentation looks to help developers in their quest for better visibility into their systems.

3

Auto- Instrumentation

The purpose of telemetry is to give people information about systems. This data is used to make informed decisions about ways to improve software and prevent disasters from occurring. In the case of an outage, analytics tools can help us investigate the root cause of the interruption by interpreting telemetry. Once the event has been resolved, the recorded traces, metrics, and logs can be correlated retroactively to gain a complete picture of what happened. In all these cases, the knowledge that's gained from telemetry assists in solving problems, be it future, present, or past, in applications within an organization. Being able to see the code is very rarely the bread and butter of an organization, which sometimes makes conversations about investing in observability difficult. Decision-makers must constantly make tradeoffs regarding where to invest. The upfront cost of instrumenting code can be a deterrent to even getting started, especially if a solution is complicated to implement and will fail to deliver any value for a long time. Auto-instrumentation looks to alleviate some of the burdens of instrumenting code manually.

In this chapter, we will cover the following topics:

- What is auto-instrumentation?
- Bytecode manipulation
- Runtime hooks and monkey patching

We will look at some example code in Java and Python, as well as the emitted telemetry, to understand the power of auto-instrumentation. Let's get started!

Technical requirements

The application in this chapter simulates the broken telephone game. If you're not familiar with this game, it is played by having one person think of a phrase and whisper it to the second player. The second player listens to the best of their ability and whispers it to the third player; this continues until the last player shares the message they received with the rest of the group.

Each application represents a player, with the first one printing out the message it is sending, then placing the message in a request object that's sent to the next application. The last application in the game will print out the message it receives. The following diagram shows the data flow of requests and responses through the system:

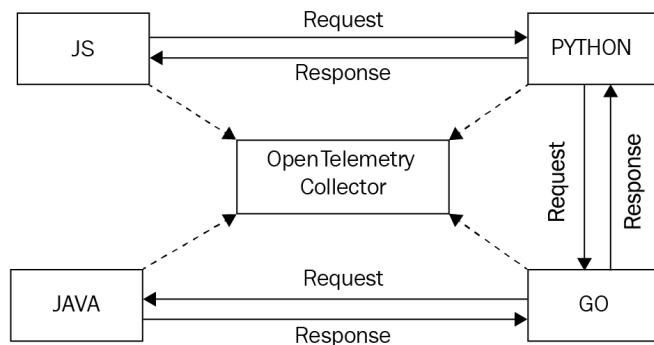


Figure 3.1 – Architectural diagram of the example application

The communication between each service is done via gRPC (<https://grpc.io>), a remote procedure call system developed by Google. For the sake of this chapter, it is enough to know that the applications do the following:

- Share a common understanding of the data structure of a service and a message via the protocol buffer's definition file.
- Send data to each other using the protocol.

The telemetry that's emitted by each application is sent to the OpenTelemetry Collector via the OpenTelemetry exporter that's configured in each service. The collector then forwards it to Jaeger, which we'll use to visualize tracing information collected.

The examples in this chapter are provided within Docker containers to make launching them easier; this also means you don't need to install separate runtime languages and libraries on your system. If you went through the Docker setup steps in the previous chapter, you can skip ahead to *Step 3*:

1. Ensure Docker is installed on your system by following the instructions on the Docker website (<https://docs.docker.com/get-docker/>). The following command shows the version of Docker that's running on your local system:

```
$ docker version
Client:
Cloud integration: 1.0.14
Version:          20.10.6
API version:      1.41
Go version:       go1.16.3 ...
```

2. Verify that `docker compose` is installed on your system using the following command. If it is not installed, follow the directions on the Docker website (<https://docs.docker.com/compose/install/>) to install it:

```
$ docker compose version
Docker Compose version 2.0.0-beta.1
```

3. Download a copy of the companion repository from GitHub and launch the Docker environment that's available in the `chapter3` directory:

```
$ git clone https://github.com/PacktPublishing/Cloud-
Native-Observability
$ cd Cloud-Native-Observability/chapter03
$ docker compose up
```

The applications that form the demo system for this chapter are written in JavaScript, Python, Go, and Java. The code for the application in each language that will be shown in this chapter is also available in this book's GitHub repository, in the `chapter3` directory; each language is in a separate folder. We will look through some of the code in this chapter, but not all of it.

Lastly, although it is not a requirement for this chapter, if you're interested in exploring the trace information that's emitted from the demo application, the best way to see it is through the Jaeger web interface. The **Docker compose** environment launches Jaeger along with the demo app, so you can verify that it is up and running by launching a web browser and visiting `http://localhost:16686`.

What is auto-instrumentation?

In the very early days of the OpenTelemetry project, a proposal was created to support producing telemetry without manual instrumentation. As we mentioned earlier in this book, OpenTelemetry uses **OpenTelemetry Enhancement Proposals** or **OTEPs** to propose significant changes or new work before producing a specification. One of the very first OTEPs to be produced by the project (<https://github.com/open-telemetry/oteps/blob/main/text/0001-telemetry-without-manual-instrumentation.md>) described the need to support users that wanted to produce telemetry without having to modify the code to do so:

Cross-language requirements for automated approaches to extracting portable telemetry data with zero source code modification. –
OpenTelemetry Enhancement Proposal #0001

Being able to get started with OpenTelemetry with very little effort for new users was very much a goal from the start of the project. The hope was to address one of the pain points of producing telemetry: the cost of manual instrumentation.

Challenges of manual instrumentation

Instrumenting an application can be a difficult task. This is especially true for someone who hasn't done so before. Instrumenting applications is a skill that takes time and practice to perfect. Some of the things that can be challenging when instrumenting code are as follows:

- The libraries and APIs that are provided by telemetry frameworks can be hard to learn how to use. With auto-instrumentation, users do not have to learn how to use the libraries and APIs directly; instead, they rely on a simplified user experience that can be tuned via configuration.
- Instrumenting applications can be tricky. This can be especially true for legacy applications where the original author of the code is no longer around. By reducing the amount of code that needs to be modified, auto-instrumentation reduces the surface of the changes that need to be made and minimizes the risks involved.
- Knowing what to instrument and how it should be done takes practice. The authors of auto-instrumentation tooling and libraries ensure that the telemetry that's produced by auto-instrumentation follows the semantic conventions defined by OpenTelemetry.

Additionally, it's not uncommon for systems to contain applications written in different languages. This adds to the complexity of manually instrumenting code as it requires developers to learn how to instrument in multiple languages. Auto-instrumentation provides the necessary tooling to minimize the effort here, as the goal of the OpenTelemetry project is to support the same configuration across languages. This means that, in theory, the auto-instrumentation experience will be *fairly consistent*. I say *fairly* here because the libraries and tools are still changing, so some inconsistencies are being worked through in the project.

Components of auto-instrumentation

In terms of OpenTelemetry, auto-instrumentation is made up of two parts. The first part is composed of instrumentation libraries. These libraries are provided and supported by members of the OpenTelemetry community, who use the OpenTelemetry API to instrument popular third-party libraries and frameworks in each language. The following table lists some of the instrumentation libraries that are provided by OpenTelemetry in various languages at the time of writing:

Language	Libraries
Go	gin, gomemcache, gorilla/mux, net/http
Erlang	Cowboy, Ecto, Phoenix
Java	Akka, gRPC, Hibernate, JDBC, Kafka, Spring, Tomcat
JavaScript	fetch, grpc-js, http, xml-http-request
.NET	AspNetCore, GrpcNetClient, SqlClient, StackExchangeRedis
Python	Boto, Celery, Django, Flask, gRPC, Redis Requests, SQLAlchemy
Ruby	ActiveJob, Faraday, Mongo, Rack, Rails

Figure 3.2 – Some of the available instrumentation libraries in OpenTelemetry

Most of these instrumentation libraries are specific to a particular third-party library of a language. For example, the `Boto` instrumentation library instruments method calls that are specific to the `Boto` library. However, there are cases where multiple instrumentation libraries could be used to instrument the same thing. An example of this is the `Requests` and `urllib3` instrumentation libraries, which would, in theory, instrument the same thing since `Requests` is built on top of `urllib3`. When choosing instrumentation libraries, a good rule of thumb is to find the library that is the most specific to your use case. If more than one fits, inspect the data that's emitted by each library to find the one that fits your needs.

The details of how exactly instrumentation libraries are implemented vary from language to language and sometimes even from library to library. Some of these details will become clearer as we progress through this chapter and look at some of the mechanisms that are used in Java and Python libraries.

As we mentioned at the beginning of this section, two components form auto-instrumentation. The second component is a mechanism that's provided by OpenTelemetry to allow users to automatically invoke the instrumentation libraries without additional work on the part of the user. This mechanism is sometimes called an **agent** or a **runner**. In practice, the purpose of this tool is to configure OpenTelemetry and load the instrumentation libraries that can be used to then generate telemetry.

Important Note

Auto-instrumentation is still being actively developed and the OpenTelemetry specification around auto-instrumentation, its implementation, and how configuration should be specified is still in development. The adoption in different languages is, at the time of writing, in various stages. For the examples in this chapter, the Python and Java examples use full auto-instrumentation with both instrumentation libraries and an agent. The JavaScript and Go code only leverage instrumentation libraries.

Limits of auto-instrumentation

Auto-instrumentation is a good place to start the journey of instrumenting an application and gaining more visibility into its inner workings. However, there are some limitations as to what can be achieved with automatic instrumentation, all of which we should take into consideration.

The first limitation may seem obvious, but it is that auto-instrumentation cannot instrument application-specific code. As such, the instrumentation that's produced via auto-instrumentation is always going to be missing some critical information about your application. For example, consider the following simplified code example of a client application making a web request via the instrumented `requests` HTTP library:

```
def do_something_important():
    # doing many important things

def client_request():
    do_something_important()
    requests.get("https://webserver")
```

If auto-instrumentation were exclusively used to instrument the previous example, telemetry would be generated for the call that was made via the library, but no information would be captured about what happened when the `do_something_important` function was called. This would likely be undesirable as it could leave many questions unanswered.

Another limitation of auto-instrumentation is that it may instrument things you're not interested in. This may result in the same network call being recorded multiple times, or generated data that you're not interested in using. An effort is being made in OpenTelemetry to support configuration to give users fine-grained control over how telemetry is generated via instrumentation libraries.

With this in mind, let's learn how auto-instrumentation is implemented in Java.

Bytecode manipulation

The Java implementation of auto-instrumentation for OpenTelemetry leverages the **Java Instrumentation API** to instrument code (<https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/Instrumentation.html>). This API is defined as part of the Java language and can be used by anyone interested in collecting information about an application.

OpenTelemetry Java agent

The OpenTelemetry Java agent is distributed to users via a single **Java archive (JAR)** file, which can be downloaded from the `opentelemetry-java-instrumentation` repository (<https://github.com/open-telemetry/opentelemetry-java-instrumentation/releases>). The JAR contains the following components:

- The `javaagent` module. This is called by the Java Instrumentation API.
- Instrumenting libraries for various frameworks and third-party libraries.
- The tooling to initialize and configure the OpenTelemetry components.
These will be used to produce telemetry and deliver it to its destination.

The JAR is invoked by passing it to the Java runtime via the `-javaagent` command-line option. The Java OpenTelemetry agent supports configuration via command-line arguments, also known in Java as **system properties**. The following command is an example of how the agent can be used in practice:

```
java -javaagent:/app/opentelemetry-javaagent.jar \
      -Dotel.resource.attributes=service.name=broken-telephone-
      java\
      -Dotel.traces.exporter=otlp \
      -jar broken-telephone.jar
```

Note that the preceding command is also how the demo application is launched inside the container. Using the Java agent to load the OpenTelemetry agent gives the library a chance to modify the bytecode before any other code is executed. The following diagram shows some of the components that are involved in the initialization process when the OpenTelemetry agent is used. `OpenTelemetryAgent` starts the process, while `OpenTelemetryInstaller` uses the configuration provided at invocation time to configure the emitters of telemetry. Meanwhile, `AgentInstaller` loads Byte Buddy, an open source library for modifying Java code at runtime, which is used to instrument the code via bytecode injection:

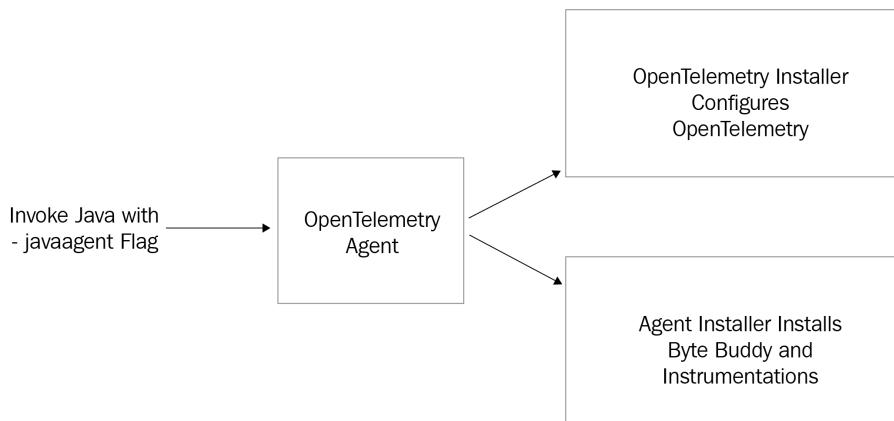


Figure 3.3 – OpenTelemetry Java agent loading order

`AgentInstaller` also loads all the third-party instrumentation libraries that are available in the OpenTelemetry agent.

Important Note

The mechanics of bytecode injection are outside the scope of this book. For the sake of this chapter, it's enough to know that the Java agent injects the instrumentation code at runtime. If you're interested in learning more, I recommend spending some time browsing the Byte Buddy site: <https://bytebuddy.net/#/>.

The following code shows the Java code that handles gRPC requests for the broken telephone server. The specifics of the code are not overly important here, but pay attention to any instrumentation code you can see:

BrokenTelephoneServer.java

```

static class BrokenTelephoneImpl extends
BrokenTelephoneGrpc.BrokenTelephoneImplBase {

    @Override
    public void saySomething(Brokentelephone.
BrokenTelephoneRequest req,
                           StreamObserver<Brokentelephone.
BrokenTelephoneResponse> responseObserver) {
        Brokentelephone.BrokenTelephoneResponse reply =
  
```

```
Brokentelephone.BrokenTelephoneResponse.newBuilder()
        .setMessage("Hello " + req.getMessage()) .
build();
    responseObserver.onNext(reply);
    responseObserver.onCompleted();
}
}
```

As you can see, there is no mention of OpenTelemetry anywhere in the code. The real magic happens when the agent is called at runtime and instruments the application via bytecode injection, as we'll see shortly. With this, we now have an idea of how auto-instrumentation works in Java. Now, let's compare this to the Python implementation.

Runtime hooks and monkey patching

In Python, unlike in Java, where a single archive contains everything that's needed to support auto-instrumentation, the implementation relies on several separate components that must be discussed to help us fully understand how auto-instrumentation works.

Instrumenting libraries

Instrumentation libraries in Python rely on one of two mechanisms to instrument third-party libraries:

- Event hooks are exposed by the libraries being instrumented, allowing the instrumenting libraries to register and produce telemetry as events occur.
- Any intercepting calls to libraries are instrumented and are replaced at runtime via a technique known as **monkey patching** (https://en.wikipedia.org/wiki/Monkey_patch). The instrumenting library receives the original call, produces telemetry data, and then calls the underlying library.

Monkey patching is like bytecode injection in that the applications make calls to libraries without suspecting that those calls have been replaced along the way. The following diagram shows how the `opentelemetry-instrumentation-redis` monkey patch calls `redis.Redis.execute_command` to produce telemetry data before calling the underlying library:

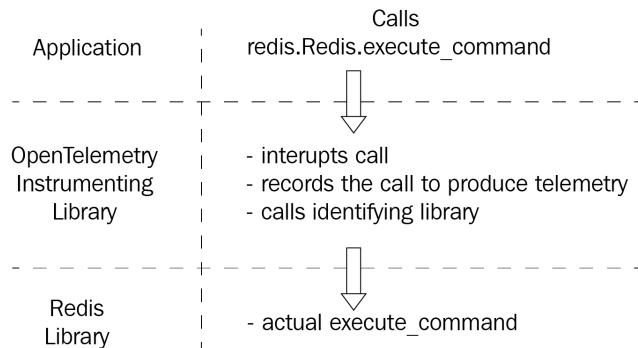


Figure 3.4 – Monkey-caught call to the Redis library

Each instrumentation library adheres to an interface to register and deregister itself. At the time of writing, in Python, unlike in Java, the different instrumentation libraries are packaged independently. This has the advantage of reducing the number of dependencies that are required to install the instrumentation libraries. However, it does have the disadvantage of requiring users to know what packages they will need to install. There are a few ways to work around this, which we'll explore in *Chapter 7, Instrumentation Libraries*.

The Instrumentor interface

To ensure a consistent experience for the users of instrumentation libraries, as well as ensuring the developers of those libraries know what needs to be implemented, the OpenTelemetry Python community has defined the `Instrumentor` (<https://opentelemetry-python-contrib.readthedocs.io/en/latest/instrumentation/base/instrumentor.html>) interface. This interface requires library authors to provide implementations for the following methods:

- `_instrument`: This method contains any initialization logic for the instrumenting library. This is where monkey patching or registering for event hooks takes place.
- `_uninstrument`: This method provides the logic to deregister the library from event hooks or remove any monkey patching. This may also contain any additional cleanup operations.

- `instrumentation_dependencies`: This method returns a list of the library and the versions that the instrumentation library supports.

In addition to fulfilling the Instrumentor interface, if an instrumentation library wishes to be available for auto-instrumentation, it must register itself via an entry point. An entry point is a Python mechanism that allows modules to make themselves discoverable by registering a class or method via a string at installation time.

Important Note

Additional information on entry points is available in the official Python documentation: <https://packaging.python.org/specifications/entry-points/>.

Other Python code can then load this code by doing a lookup for an entry point by name and executing it.

Wrapper script

For those mechanisms to be triggered, the Python implementation ships a script that can be called to wrap any Python application. The `opentelemetry-instrument` script finds all the instrumentations that have been installed in an environment by loading the entry points registered under the `opentelemetry_instrumentor` name.

The following diagram shows two different instrumentation library packages, `opentelemetry-instrumentation-foo` and `opentelemetry-instrumentation-bar`, registering a separate Python class in the `opentelemetry_instrumentor` entry point's catalog. This catalog is globally available within the Python environment and when `opentelemetry-instrument` is invoked, it searches that catalog and loads any instrumentation that's been registered by calling the `instrument` method:

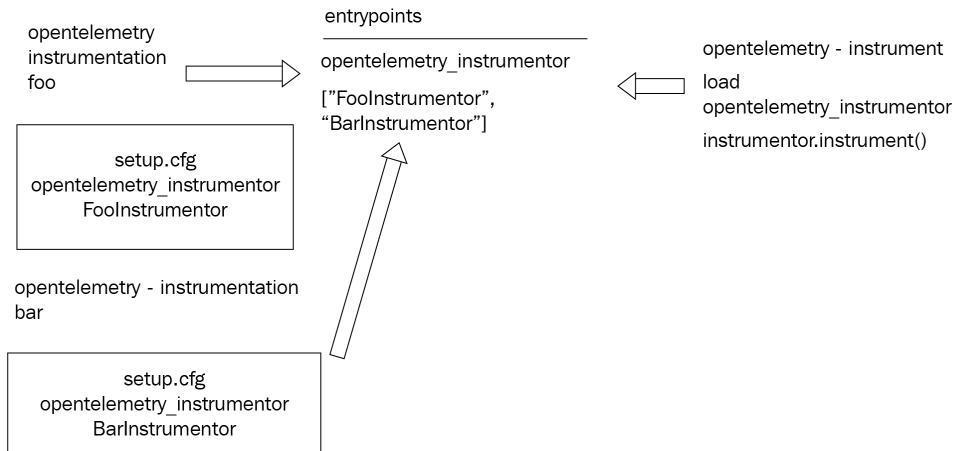


Figure 3.5 – Package registration

The `opentelemetry-instrument` script is made available via the `opentelemetry-instrumentation` Python package. The following code shows the gRPC server implemented in Python. As you can see, as with the previous example, it does not mention OpenTelemetry:

brokentelephone.py

```

#!/usr/bin/env python3

from concurrent import futures

import grpc
import brokentelephone_pb2
import brokentelephone_pb2_grpc

class Player(brokentelephone_pb2_grpc.BrokenTelephoneServicer):
    def SaySomething(self, request, context):
        return brokentelephone_pb2.BrokenTelephoneResponse(
            message="Hello, %s!" % request.message
        )

    def serve():
        server = grpc.server(futures.ThreadPoolExecutor(max_
workers=10))

```

```

brokentelephone_pb2_grpc.add_BrokenTelephoneServicer_to_
server(Player(), server)
server.add_insecure_port("[::]:50051")
server.start()
server.wait_for_termination()

if __name__ == "__main__":
    serve()

```

As we saw in the Java code example, the preceding code is strictly application code – there's no instrumentation in sight. The following command shows an example of how auto-instrumentation is invoked in Python:

```
opentelemetry-instrument ./broken_telephone.py
```

The following screenshot shows a trace that's been generated by our sample application. As we can see, the originating request was made by the brokentelephone-js service to Python, Go, and finally the Java application. The trace information was generated by the gRPC instrumentation library in each of those languages:

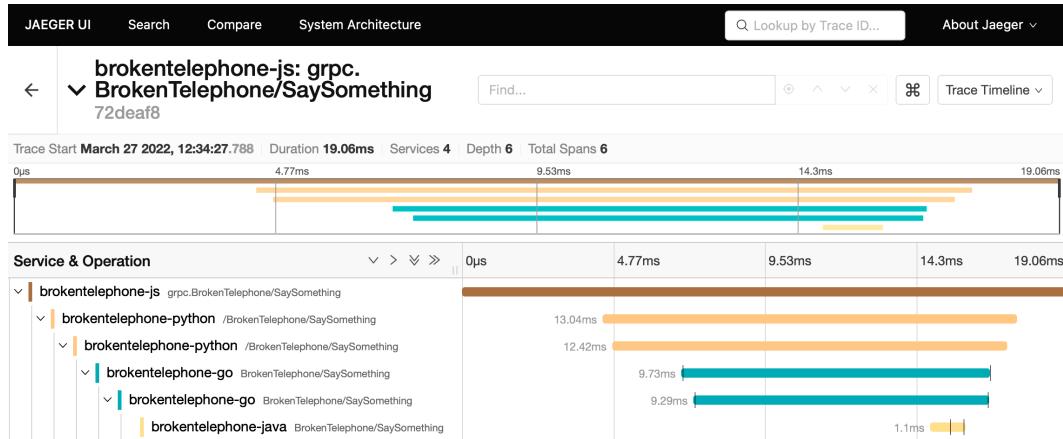


Figure 3.6 – Sample trace generated automatically across all broken telephone services

If you'd like to see a trace for yourself, the demo application should allow you to do so. Just browse to the Jaeger interface at `http://localhost:16686` and search for a trace, as we did in *Chapter 2, OpenTelemetry Signals – Traces, Metrics, and Logs*. The generated trace can give us a glimpse into the data flow through our entire sample application. Although the broken telephone is somewhat trivial, you can imagine how this information would be useful for mapping information across a distributed system. With very little effort, we're able to see where time is spent in our system.

Summary

With auto-instrumentation, it's possible to reduce the time that's required to instrument an existing application. Reducing the friction to get started with telemetry gives users a chance to try it before investing significant amounts of time in manual instrumentation. And although the data that's generated via auto-instrumentation is likely not enough to get to the bottom of issues in complex systems, it's a solid starting point. Auto-instrumentation can also be quite useful when you're instrumenting an unfamiliar system.

The use of instrumentation libraries allows users to gain insight into what the libraries they're using are doing, without having to learn the ins and outs of them. The OpenTelemetry libraries that are available at the time of writing can be used to instrument existing code by following the online documentation that's been made available by each language. As we'll learn in *Chapter 7, Instrumentation Libraries*, using these libraries can be tremendously useful in reducing the code that's needed to instrument applications.

In this chapter, we compared two different implementations of auto-instrumentation by looking at the Java implementation, which utilizes bytecode injection, and the Python implementation, which uses runtime hooks and monkey patching. In each case, the implementation leverages features of the language that allows the implementation to inject telemetry at appropriate times in the code's execution. Before diving into auto-instrumentation, however, it is useful to understand how each signal can be leveraged independently, starting with distributed tracing. We will do this in the next chapter.

Section 2: Instrumenting an Application

In this part, you will walk through instrumenting an application by using the signals offered by OpenTelemetry: distributed tracing, metrics, and logging.

This part of the book comprises the following chapters:

- *Chapter 4, Distributed Tracing – Tracing Code Execution*
- *Chapter 5, Metrics – Recording Measurements*
- *Chapter 6, Logging – Capturing Events*
- *Chapter 7, Instrumentation Libraries*

4

Distributed Tracing – Tracing Code Execution

So, now that we have an understanding of the concepts of OpenTelemetry and are familiar with the different signals it covers, it's time to start instrumenting application code. In *Chapter 2, OpenTelemetry Signals – Tracing, Metrics, and Logging*, we covered the terminology and concepts of those signals by looking at a system that was instrumented with OpenTelemetry. Now, it's time to get hands-on with some code to start generating telemetry ourselves, and to do this, we're going to first look at implementing a tracing signal.

In this chapter, we will cover the following topics:

- Configuring OpenTelemetry
- Generating tracing data
- Enriching the data with attributes, events, and links
- Adding error handling information

By the end of this chapter, you'll have instrumented several applications with OpenTelemetry and be able to trace how those applications are connected via distributed tracing. This will start giving you a sense of how distributed tracing can be used in your own applications going forward.

Technical requirements

At the time of writing, OpenTelemetry for Python supports Python 3.6+. All Python examples in this book will use Python 3.8, which can be downloaded and installed by following the instructions at <https://docs.python.org/3/using/index.html>. The following command can verify which version of Python is installed. It's possible for multiple versions to be installed simultaneously on a single system, which is why both `python` and `python3` are shown here:

```
$ python --version  
$ python3 --version
```

It is recommended to use a virtual environment to run the examples in this book (<https://docs.python.org/3/library/venv.html>). A virtual environment in Python allows you to install packages in isolation from the rest of the system, meaning that if anything goes wrong, you can always delete the virtual environment and start a fresh one. The following commands will create a new virtual environment in a folder called `cloud_native_observability`:

```
$ mkdir cloud_native_observability  
$ python3 -m venv cloud_native_observability  
$ source cloud_native_observability/bin/activate
```

The example code in this chapter will rely on a few different third-party libraries – `Flask` and `Request`. The following command will install all the required packages for this chapter using the package installation for Python, `pip`:

```
$ pip install flask requests
```

Now that we have a virtual environment configured and the libraries needed, we will install the necessary Python packages to use OpenTelemetry. The main libraries we'll need for this section are the API and SDK packages:

```
$ pip install opentelemetry-api opentelemetry-sdk
```

The `pip freeze` command lists all the installed packages in this Python environment; we can use it to confirm whether the correct packages are installed:

```
$ pip freeze | grep opentelemetry
opentelemetry-api==1.3.0
opentelemetry-sdk==1.3.0
opentelemetry-semantic-conventions==0.22b0
```

The version of the packages installed in your environment may differ, as the OpenTelemetry project is still very much under active development, and releases are pretty frequent. It's important to remember this as we work through the examples, as some methods may be slightly different, or the output may vary.

Important Note

The OpenTelemetry APIs should not change unless a major version is released.

Configuring the tracing pipeline

With the packages now installed, we're ready to take our first step to generate distributed traces with OpenTelemetry – configuring the tracing pipeline. The tracing pipeline is what allows the tracing data we explored in *Chapter 2, OpenTelemetry Signals – Traces, Metrics, and Logs*, to be generated when the OpenTelemetry API calls are made. The pipeline also defines where and how the data will be emitted. Without a tracing pipeline, a no-op implementation is used by the API, meaning the code will not generate distributed traces. The tracing pipeline configures the following:

- `TracerProvider` to determine how spans should be generated
- A `Resource` object, which identifies the source of the spans
- `SpanProcessor` to describe how spans will be exported
- `SpanExporter` to describe where the spans will be exported

The following code imports the `TracerProvider`, `ConsoleSpanExporter`, and `SimpleSpanProcessor` classes from the SDK to configure a tracer provider. In this example, `ConsoleSpanExporter` will be used to output traces from the application to the console. The last step to configure the tracer provider is to call the `set_tracer_provider` method, which will set the global tracer provider to the provider we instantiated. The code will be placed in the `configure_tracer` method, which will be called before we do anything else in the code:

shopper.py

```
#!/usr/bin/env python3
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import ConsoleSpanExporter,
SimpleSpanProcessor

def configure_tracer():
    exporter = ConsoleSpanExporter()
    span_processor = SimpleSpanProcessor(exporter)
    provider = TracerProvider()
    provider.add_span_processor(span_processor)
    trace.set_tracer_provider(provider)

if __name__ == "__main__":
    configure_tracer()
```

Throughout this chapter, as we iterate over the application and add more code, each time we do so, we will test the code and inspect its output using the following command, unless specified otherwise:

```
$ python ./shopper.py
```

Running this command for the initial code will not output anything. This allows us to confirm that the modules have been found and imported correctly, and that the code doesn't have any errors in it.

Important Note

A common mistake when first configuring `TracerProvider` is to forget to set the global `TracerProvider`, causing the API to use a default no-op implementation of `TracerProvider`. This default is configured intentionally for the use case where a user does not wish to enable tracing within their application.

Although it may not seem like much, configuring `TracerProvider` for an application is a critical first step before we can start collecting distributed traces. It's a bit like gathering all the ingredients before baking a cake, so let's get baking!

Getting a tracer

With the tracing pipeline configured, we can now obtain the generator for our tracing data, `Tracer`. The `TracerProvider` interface defines a single method to allow us to obtain a tracer, `get_tracer`. This method requires a name argument and, optionally, a version argument, which should reflect the name and version of the instrumenting module. This information is valuable for users to quickly identify what the source of the tracing data is. An example shown in *Figure 4.1* shows how the values passed into `get_tracer` will vary, depending on where the call is made. Inside the library calls to `requests` and `Flask`, the name and version will reflect those libraries, whereas in the `shopper` and `grocery_store` modules, the name and version will reflect those modules.

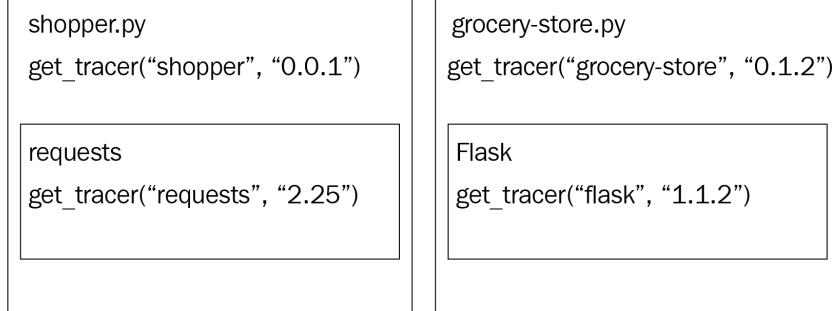


Figure 4.1 – The tracer name and version configuration at different stages of an application

To get the first tracer, the following code will be added to `shopper.py` immediately at the end of `configure_tracer` to return a tracer from the method:

shopper.py

```
def configure_tracer():
    exporter = ConsoleSpanExporter()
    span_processor = SimpleSpanProcessor(exporter)
    provider = TracerProvider()
    provider.add_span_processor(span_processor)
    trace.set_tracer_provider(provider)
    return trace.get_tracer("shopper.py", "0.0.1")

if __name__ == "__main__":
    tracer = configure_tracer()
```

It's important to remember to make the name and version meaningful; the name should be unique within the scope of the application it instruments. The instrumentation scope could be a package, module, or even a class. It's finally time to start using this tracer and trace the application! There are several ways to create a span in OpenTelemetry; let's explore them now.

Generating tracing data

It's finally time to start generating telemetry from the application! There are several ways to create a span in OpenTelemetry; the first one we'll use is to call `start_span` on the `tracer` instance we obtained previously. This will create a `Span` object, using the only required string argument as the name of the span. The `Span` object is the building block of distributed tracing and is intended to represent a unique unit of work in our application. In the following example, we will create a new `Span` object before calling a method that will do some work. Since our application is a *shopper*, the first thing the `shopper` will do is browse the store. In order for the tracing data to be useful, it's important to use a meaningful name in the creation of the span. Once `browse` has returned, we will call `end` on the `Span` object to signal that the work is complete:

shopper.py

```
def browse():
    print("visiting the grocery store")
```

```
if __name__ == "__main__":
    tracer = configure_tracer()
    span = tracer.start_span("visit store")
    browse()
    span.end()
```

Running the code will output our first trace to the console. The `ConsoleSpanExporter` automatically outputs the data as formatted JSON to make it easier to read:

shopper.py output

```
visiting the grocery store
{
  "name": "visit store",
  "context": {
    "trace_id": "0x4c6fd97f286439b1a4bb109f12bf2095",
    "span_id": "0x6ea2219c865f6c4b",
    "trace_state": "[]"
  },
  "kind": "SpanKind.INTERNAL",
  "parent_id": null,
  "start_time": "2021-06-26T20:26:47.176169Z",
  "end_time": "2021-06-26T20:26:47.176194Z",
  "status": {
    "status_code": "UNSET"
  },
  "attributes": {},
  "events": [],
  "links": [],
  "resource": {
    "telemetry.sdk.language": "python",
    "telemetry.sdk.name": "opentelemetry",
    "telemetry.sdk.version": "1.3.0",
    "service.name": "unknown_service"
  }
}
```

Some of the data worth noting in the preceding output is as follows:

- The `name` field of the span we provided.
- The automatically generated trace and span identifiers – `trace_id` and `span_id`.
- The `start_time` and `end_time` timestamps, which can be used to calculate the duration of an operation being traced.
- The `parent_id` identifier is not set. This identifies the span created as the beginning of a trace, otherwise known as `root span`.
- The `status_code` field of the span is set to `UNSET` by default.

With the preceding span information from the JSON output, we now have the first piece of data about the work that our application is doing. One of the most critical pieces of information generated in this data is `trace_id`. This trace identifier is a 128-bit integer that allows operations to be tied together in a distributed trace and represents the single request through the entire system. `span_id` is a 64-bit integer used to identify the specific unit of work in the request and also relationships between different operations. In this next code example, we'll add another operation to our trace to see how this identifier works, but we'll need to take a brief detour to look at the Context API before continuing too far.

Important Note

The examples in this chapter will only use `ConsoleSpanExporter`. We will explore additional exporters in *Chapter 8, The OpenTelemetry Collector*, and *Chapter 10, Configuring a Backend*, when we look at the OpenTelemetry Collector and different backends.

The Context API

In order to tie spans together, we'll need to activate our spans before starting new ones. Activating a span in OpenTelemetry is synonymous with setting the span in the current context object. The `Context` object is a mechanism used across signals to share data about the application either in-process or across API boundaries via propagation. No matter where you are in the application code, it's possible to get the current span by using the Context API. The `Context` object can be thought of as an immutable data store with a consistent API across implementations. In Python, the implementation relies on `ContextVars`, as previously discussed in *Chapter 1, The History and Concepts of Observability*, but not all languages have the notion of a context built into the language itself. The Context API ensures that users will have a consistent experience when using OpenTelemetry. The API definition for interacting with the context is fairly minimal:

- `get_value`: Retrieves a value for a given key from the context. The only required argument is a key and, optionally, a `context` argument. If no context is passed in, the value returned will be pulled from the global context.
- `set_value`: Stores a value for a certain key in the context. The method receives a key, value, and optionally, a context argument to set the value into. As mentioned before, the context is immutable, so the return value is a new `Context` object with the new value set.
- `attach`: Calling `attach` associates the current execution with a specified context. In other words, it sets the current context to the context passed in as an argument. The return value is a unique token, which is used by the `detach` method described next.
- `detach`: To return the context to its previous state, this method receives a token that was obtained by attaching to another context. Upon calling it, the context that was current at the time `attach` was called is restored.

Don't worry if the description doesn't quite make sense yet; the next example will help clarify things. In the following code, we activate the span by setting it in the context via the `set_span_in_context` method, which, under the hood, calls the current context's `set_value` method. The return value of this call is a new immutable `context` object, which we can then attach to before starting the second span:

shopper.py

```
from opentelemetry import context, trace
if __name__ == "__main__":
    tracer = configure_tracer()
    span = tracer.start_span("visit store")
    ctx = trace.set_span_in_context(span)
    token = context.attach(ctx)
    span2 = tracer.start_span("browse")
    browse()
    span2.end()
    context.detach(token)
    span.end()
```

Running the application and looking at the output once again, we can now see that the `trace_id` value for both spans is the same. We can also see that the `browse` span has a `parent_id` field that matches `span_id` of the `visit store` span:

shopper.py output

```
visiting the grocery store
{
    "name": "browse",
    "context": {
        "trace_id": "0x03c197ae7424cc492ab1c92112490be1",
        "span_id": "0xb7396b0e6ccab2fd",
        "trace_state": "[]"
    },
    "kind": "SpanKind.INTERNAL",
    "parent_id": "0x8dd8c60c67518a8d",
}
{
    "name": "visit store",
    "context": {
        "trace_id": "0x03c197ae7424cc492ab1c92112490be1",
        "span_id": "0x8dd8c60c67518a8d",
        "trace_state": "[]"
    },
    "kind": "SpanKind.INTERNAL",
    "parent_id": null,
}
```

Starting and ending spans manually can be useful in many cases, but as demonstrated by the previous code, managing the context manually can be somewhat cumbersome. More often than not, it is easier in Python to use a context manager to wrap the work we want to trace. The `start_as_current_span` convenience method allows us to do exactly this by creating a new Span object, setting it as the current span in a context, and calling the `attach` method. Additionally, it will automatically end the span once the context has been exited. The following code shows us how we can simplify the previous code we wrote:

shopper.py

```
if __name__ == "__main__":
    tracer = configure_tracer()
    with tracer.start_as_current_span("visit store"):
        with tracer.start_as_current_span("browse"):
            browse()
```

This method simplifies the code quite a bit. The automatic management of the context can be used to quickly create hierarchies of spans. In the following code, we will add one new method and one more span. We'll then run the code to observe how each span will use the previous span in the context as the new span's parent:

shopper.py

```
def add_item_to_cart(item):
    print("add {} to cart".format(item))

if __name__ == "__main__":
    tracer = configure_tracer()
    with tracer.start_as_current_span("visit store"):
        with tracer.start_as_current_span("browse"):
            browse()
            with tracer.start_as_current_span("add item to
cart"):
                add_item_to_cart("orange")
```

Running the shopper application, we're starting to see what is appearing to be more and more like a real trace. Looking at the output from the new code, we can see three different operations captured. The order in which output appears in your terminal may vary; we will review operations in the same order in which they appear in the code. The first operation to look at is `visit_store`, as mentioned previously; the root span can be identified by the `parent_id` field being null:

shopper.py output

```
{  
    "name": "visit store",  
    "context": {  
        "trace_id": "0x9251fa73b421a143a7654afb048a4fc7",  
        "span_id": "0x08c9bf4cccd7ba5d",  
        "trace_state": "[]"  
    },  
    "kind": "SpanKind.INTERNAL",  
    "parent_id": null,  
    "start_time": "2021-06-26T21:43:20.441933Z",  
    "end_time": "2021-06-26T21:43:20.442222Z",  
    "status": {  
        "status_code": "UNSET"  
    },  
    "attributes": {},  
    "events": [],  
    "links": [],  
    "resource": {  
        "telemetry.sdk.language": "python",  
        "telemetry.sdk.name": "opentelemetry",  
        "telemetry.sdk.version": "1.3.0",  
        "service.name": "unknown_service"  
    }  
}
```

The next operation to review in the output is the `browse` span. Note that the span's `parent_id` identifier is equal to the `span_id` identifier of the `visit_store` span. `trace_id` also matches, which indicates that the spans are connected in the same trace:

shopper.py output

```
{  
    "name": "browse",  
    "context": {  
        "trace_id": "0x9251fa73b421a143a7654afb048a4fc7",  
        "span_id": "0xa77587668be46030",  
        "trace_state": "[]"  
    },  
    "kind": "SpanKind.INTERNAL",  
    "parent_id": "0x08c9bf4cccd7ba5d",  
    "start_time": "2021-06-26T21:43:20.442091Z",  
    "end_time": "2021-06-26T21:43:20.442212Z",  
    "status": {  
        "status_code": "UNSET"  
    },  
    "attributes": {},  
    "events": [],  
    "links": [],  
    "resource": {  
        "telemetry.sdk.language": "python",  
        "telemetry.sdk.name": "opentelemetry",  
        "telemetry.sdk.version": "1.3.0",  
        "service.name": "unknown_service"  
    }  
}
```

The last span to review is the `add item to cart` span. As with the previous span, its `trace_id` identifier will also match the previous spans. In this case, the `parent_id` identifier of the `add item to cart` span now matches the `span_id` identifier of the `browse` span:

shopper.py output

```
{  
    "name": "add item to cart",  
    "context": {  
        "trace_id": "0x9251fa73b421a143a7654afb048a4fc7",  
        "span_id": "0x6470521265d80512",  
        "trace_state": "[]"  
    },  
    "kind": "SpanKind.INTERNAL",  
    "parent_id": "0xa77587668be46030",  
    "start_time": "2021-06-26T21:43:20.442169Z",  
    "end_time": "2021-06-26T21:43:20.442191Z",  
    "status": {  
        "status_code": "UNSET"  
    },  
    "attributes": {},  
    "events": [],  
    "links": [],  
    "resource": {  
        "telemetry.sdk.language": "python",  
        "telemetry.sdk.name": "opentelemetry",  
        "telemetry.sdk.version": "1.3.0",  
        "service.name": "unknown_service"  
    }  
}
```

Not too bad – the code looks much simpler than the previous example, and we can already see how easy it is to trace code in applications. The last method we can use to start a span is by using a **decorator**. A decorator is a convenient way to instrument code without having to add any tracing specific information to the code itself. This makes the code a bit cleaner.

Important Note

Using the decorator means you will need to keep an instance of a tracer initialized and available globally for the decorators to be able to use it.

Refactoring the `shopper.py` code, we will move the instantiation of the tracer out of the main method and add decorators to each of the methods we've defined previously. Note that the code in main is simplified significantly:

shopper.py

```
tracer = configure_tracer()

@tracer.start_as_current_span("browse")
def browse():
    print("visiting the grocery store")
    add_item_to_cart("orange")

@tracer.start_as_current_span("add item to cart")
def add_item_to_cart(item):
    print("add {} to cart".format(item))

@tracer.start_as_current_span("visit store")
def visit_store():
    browse()

if __name__ == "__main__":
    visit_store()
```

Run the program once again; the spans will be printed as they were before. The output will not have changed with this refactor, but the code looks much cleaner. As with the previous example, context management is handled for us, so we don't need to worry about interacting with the Context API. Reading the code is much simpler with decorators, and it's also easy for someone new to the code to implement new methods with the same pattern when adding code to the application.

Span processors

A quick note about the span processor used in the code so far – the initial configuration of the tracing pipeline used `SimpleSpanProcessor`. This does all of its processing in line with the export happening as soon as the span ends. This means that every span added to the code will add latency in the application, which is generally not what we want. This may be the right choice in some cases – for example, if it's impossible to guarantee that threads other than the main thread will finish before a program is interrupted. However, it's generally recommended that span processing happens out of band from the main thread. An alternative to `SimpleSpanProcessor` is `BatchSpanProcessor`. *Figure 4.2* shows how the execution of the program is interrupted by `SimpleSpanProcessor` to export a span, whereas with `BatchSpanProcessor`, another thread handles the export operation:

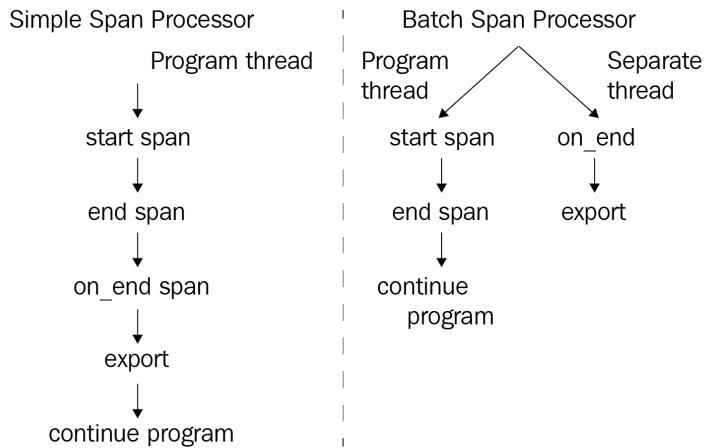


Figure 4.2 – SimpleSpanProcessor versus BatchSpanProcessor

As the name suggests, `BatchSpanProcessor` groups the export of spans. It does this by launching a separate thread that exports spans on a schedule or when there's a certain number of items in the queue. This prevents adding unnecessary latency to the normal application code paths. Configuring `BatchSpanProcessor` is done much like `SimpleSpanProcessor`. For the remainder of the examples in this chapter, we will use this new processor. The following refactor updates the imports and code in the tracer configuration to use `BatchSpanProcessor`:

shopper.py

```
from opentelemetry.sdk.trace.export import BatchSpanProcessor,
ConsoleSpanExporter

def configure_tracer():
    exporter = ConsoleSpanExporter()
    span_processor = BatchSpanProcessor(exporter)
    provider = TracerProvider()
    provider.add_span_processor(span_processor)
    trace.set_tracer_provider(provider)
    return trace.get_tracer("shopper.py", "0.0.1")
```

Run the application now to confirm that the program still works and that the output is the same with the new span processor in place. Although it may not look like much has changed, if you look closely at the `start_time` and `end_time` fields of each span produced, the duration of each span has changed. *Figure 4.3* shows a chart comparing output from running the program with each type of span processor. The duration of the `visit store` span is significantly shorter using `BatchSpanProcessor` because the processing of each span is happening asynchronously:

	<code>start_time</code>	<code>end_time</code>	<code>duration</code>
SimpleSpanProcessor			
visit store	2021-07-02T16:57:56.075531Z	2021-07-02T16:57:56.076188Z	657µs
browse	2021-07-02T16:57:56.075598Z	2021-07-02T16:57:56.076079Z	481µs
add item to cart	2021-07-02T16:57:56.075644Z	2021-07-02T16:57:56.075663Z	19µs
BatchSpanProcessor			
visit store	2021-07-02T17:12:41.748660Z	2021-07-02T17:12:41.748892Z	232µs
browse	2021-07-02T17:12:41.748734Z	2021-07-02T17:12:41.748879Z	145µs
add item to cart	2021-07-02T17:12:41.748828Z	2021-07-02T17:12:41.748855Z	27µs

Figure 4.3 – Span durations using SimpleSpanProcessor and BatchSpanProcessor

Even though microseconds may not seem like much in our example, this type of performance impact is critical to systems in production. `BatchSpanProcessor` is a much better choice for running real-world applications. Now, we have a better sense of how to generate tracing data via the API, but the data we've produced so far could be improved. It doesn't have nearly enough details to make it truly useful, so let's tackle that next.

Enriching the data

You may have noticed in output from the previous examples that each span emitted contains a `resource` attribute. The `resource` attribute provides an immutable set of attributes, representing the entity producing the telemetry. `resource` attributes are not specific to tracing. Any signal that emits telemetry leverages resource attributes by adding them to the data produced at export time. As covered in *Chapter 1, The History and Concepts of Observability*, the resource in an application is associated with the telemetry generator, which, in the case of tracing, is `TracerProvider`. The `resource` attribute on the span output we've seen so far is automatically provided by the SDK with some information about the SDK itself, as well as a default `service.name`. The service name is used by many backends to identify the services sending traces to them; however, as you can see, the default value of `unknown_service` is not a super useful name. Let's fix this. The following code will create a new `Resource` object with a service name and version number, and then pass it in as an argument to the `TracerProvider` constructor:

shopper.py

```
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor,
    ConsoleSpanExporter

def configure_tracer():
    exporter = ConsoleSpanExporter()
    span_processor = BatchSpanProcessor(exporter)
    resource = Resource.create(
        {
            "service.name": "shopper",
            "service.version": "0.1.2",
        }
    )
    provider = TracerProvider(resource=resource)
```

```
provider.add_span_processor(span_processor)
trace.set_tracer_provider(provider)
return trace.get_tracer("shopper.py", "0.0.1")
```

The output from the application will now include the information added in the resource attribute along with the automatically populated data, as shown here:

```
"resource": {
    "telemetry.sdk.language": "python",
    "telemetry.sdk.name": "opentelemetry",
    "telemetry.sdk.version": "1.3.0",
    "service.name": "shopper",
    "service.version": "0.1.2"
}
```

This is much more useful than `unknown_service`; however, we now have the name and version hardcoded in two places. Even worse, the names and versions don't match. Let's fix this before going further by refactoring the `configure_tracer` method to expect the name and version arguments, as follows:

shopper.py

```
def configure_tracer(name, version):
    exporter = ConsoleSpanExporter()
    span_processor = BatchSpanProcessor(exporter)
    resource = Resource.create(
        {
            "service.name": name,
            "service.version": version,
        }
    )
    provider = TracerProvider(resource=resource)
    provider.add_span_processor(span_processor)
    trace.set_tracer_provider(provider)
    return trace.get_tracer(name, version)
tracer = configure_tracer("shopper", "0.1.2")
```

After running the application, the output should remain the same as it was before the change. The code is now less error-prone, as we only have one place to set the service name and version information, and `configure_tracer` can be reused to configure OpenTelemetry for different applications, which will come in handy shortly.

Some additional information you may want to populate in the resource includes things such as the hostname or, in the case of a dynamic runtime environment, an instance identifier of some sort. The OpenTelemetry SDK provides an interface to provide some of the details about a resource automatically; this is known as the `ResourceDetector` interface.

ResourceDetector

As its name suggests, the purpose of the `ResourceDetector` attribute is to detect information that will automatically be populated into a resource. A resource detector is a great way to extract information about a platform running an application, and there are already existing detectors for some popular cloud providers. This information can be a useful way to group applications by region or host when trying to pinpoint application performance issues. The interface for `ResourceDetector` specifies a single method to implement, `detect`, which returns a resource. Let's implement a `ResourceDetector` interface that we can reuse in all the services of the grocery store. This detector will automatically fill in the hostname and IP address of the machine running the code; to accomplish this, Python's `socket` library will come in handy. Place the following code in a new file in the same directory as `shopper.py`:

local_machine_resource_detector.py

```
import socket
from opentelemetry.sdk.resources import Resource,
ResourceDetector

class LocalMachineResourceDetector(ResourceDetector):
    def detect(self):
        hostname = socket.gethostname()
        ip_address = socket.gethostbyname(hostname)
        return Resource.create(
            {
                "net.host.name": hostname,
                "net.host.ip": ip_address,
            }
        )
```

To make use of this new module, let's import it into the shopper application. The code in `configure_tracer` will also be updated to call this new `ResourceDetector` first, before adding the service name and version information. As mentioned earlier, a resource is immutable, meaning that there's no method to call to update a specific resource. Adding new attributes to the resource generated by our resource detector is done via a call to a resource's `merge` method. `merge` creates a new resource from the caller's attributes and then updates that new resource to include all the attributes of the resource passed in as an argument. The following update to the code imports the module we just created, and creates a new resource by calling `LocalMachineResourceDetector` and calls `merge` to ensure that our previous resource information is not lost:

shopper.py

```
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import ConsoleSpanExporter,
SimpleSpanProcessor
from local_machine_resource_detector import
LocalMachineResourceDetector
def configure_tracer(name, version):
    exporter = ConsoleSpanExporter()
    span_processor = SimpleSpanProcessor(exporter)
    local_resource = LocalMachineResourceDetector().detect()
    resource = local_resource.merge(
        Resource.create(
            {
                "service.name": name,
                "service.version": version,
            }
        )
    )
    provider = TracerProvider(resource=resource)
    return trace.get_tracer(name, version)
```

The output from running the code will now contain all the resources seen in the previous example, but it will also include the information generated by `LocalMachineResourceDetector`:

```
"resource": {  
    "telemetry.sdk.language": "python",  
    "telemetry.sdk.name": "opentelemetry",  
    "telemetry.sdk.version": "1.3.0",  
    "net.host.name": "myhost.local",  
    "net.host.ip": "192.168.128.47",  
    "service.name": "shopper",  
    "service.version": "0.1.2"  
}
```

Important Note

If the same resource attribute is included in both the caller and the resource passed into `merge`, the attributes of the argument resource will override the caller. For example, if `resource_one` has an attribute of `foo=one` and `resource_two` has an attribute of `foo=two`, the resulting resource from calling `resource_one.merge(resource_two)` will have an attribute of `foo=two`.

Feel free to play around with `ResourceDetector` and see what other useful information you can add about your machine. Try adding some environment variables or the version of Python running on your system; this can be valuable when troubleshooting applications!

Span attributes

Looking through the tracing data being emitted, we can start to get an idea of what is happening in the code we're writing. Now, let's figure out what data we should add about our shopper to make this trace even more useful. As the shopper application will be used as an HTTP client, we can take a look at the semantic conventions available in the specification to inspire us; *Figure 4.4* (https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/trace/semantic_conventions/http.md#http-client-server-example) shows us the span attributes to add if we want to adhere to OpenTelemetry's semantic conventions, as well as some sample values:

Attribute name	Value
http.method	"GET"
http.flavor	"1.1"
http.url	"https://example.com:8080/webshop/articles/4?s=1"
net.peer.ip	"192.0.2.5"
http.status_code	200

Figure 4.4 – The HTTP client span attributes semantic conventions

A valid attribute must be either a string, a 64-bit integer, a float, or a Boolean. An attribute can also be an array of any of those values, but it must be a homogenous array, meaning the elements of the array must be a single type.

Important Note

Null or None values are not encouraged in attributes, as the handling of null values in backends may differ and thus create unexpected behavior.

In the next example, we will update the `browse` method to include the recommended attributes for a client application. Since we're using decorators here, we'll need to get the current span by calling the `get_current_span` method. Once we have the span, we can call the `set_attribute` method, which requires two arguments – the key to set and the value. Since we have not yet started the server, we'll set a placeholder value for `http.url` and `net.peer.ip`:

shopper.py

```
@tracer.start_as_current_span("browse")
def browse():
    print("visiting the grocery store")
    span = trace.get_current_span()
    span.set_attribute("http.method", "GET")
    span.set_attribute("http.flavor", "1.1")
    span.set_attribute("http.url", "http://localhost:5000")
    span.set_attribute("net.peer.ip", "127.0.0.1")
```

Looking at the output from running the program, we will expect to see the attributes added to the `browse` span; let's take a look:

```
"name": "browse",
"attributes": {
    "http.method": "GET",
    "http.flavor": "1.1",
    "http.url": "http://localhost:5000",
    "net.peer.ip": "127.0.0.1"
},
```

Excellent! The data is there. It's a bit inconvenient to make independent calls to a method when wanting to set multiple attributes; thankfully, there's a convenient method to address this. The code can be simplified by making a single call to `set_attributes` and passing in a dictionary with the same values:

shopper.py

```
span.set_attributes(
{
    "http.method": "GET",
    "http.flavor": "1.1",
    "http.url": "http://localhost:5000",
    "net.peer.ip": "127.0.0.1",
}
)
```

Setting so many attributes, it can be easy for a typo to sneak in. This would, at best, be caught during a review but, at worst, could mean missing some critical data. Imagine a scenario where some alerting is configured to rely on the `url` and `flavor` attributes, but somewhere along the way, `flavor` is spelled as `flavour`. The correctness of the tracing data is critical, and to make setting these attributes more easy, a semantic conventions package provides constants that can be used instead of hardcoding common keys and values. The following is a refactor of the code to make use of the `opentelemetry-semantic-conventions` package:

shopper.py

```
from opentelemetry.semconv.trace import HttpFlavorValues,
SpanAttributes
```

```
@tracer.start_as_current_span("browse")
def browse():
    print("visiting the grocery store")
    span = trace.get_current_span()
    span.set_attributes(
    {
        SpanAttributes.HTTP_METHOD: "GET",
        SpanAttributes.HTTP_FLAVOR: HttpFlavorValues.
HTTP_1_1.value,
        SpanAttributes.HTTP_URL: "http://localhost:5000",
        SpanAttributes.NET_PEER_IP: "127.0.0.1",
    }
)
```

Of course, using semantic conventions alone may not give us enough information about the specifics of the application. One of the powers of attributes is to add meaningful data about the transaction being traced to allow us to understand what happened. One aspect of the shopper application that will likely be unique once we start processing real data is information about the items and quantities added to the cart. The following code adds attributes to the span to record that information:

shopper.py

```
@tracer.start_as_current_span("browse")
def browse():
    print("visiting the grocery store")
    span = trace.get_current_span()
    span.set_attributes(
    {
        SpanAttributes.HTTP_METHOD: "GET",
        SpanAttributes.HTTP_FLAVOR: str(HttpFlavorValues.
HTTP_1_1),
        SpanAttributes.HTTP_URL: "http://localhost:5000",
        SpanAttributes.NET_PEER_IP: "127.0.0.1",
    }
)
add_item_to_cart("orange", 5)
```

```
@tracer.start_as_current_span("add item to cart")
def add_item_to_cart(item, quantity):
    span = trace.get_current_span()
    span.set_attributes({
        "item": item,
        "quantity": quantity,
    })
    print("add {} to cart".format(item))
```

The topic of span attributes will be revisited when we introduce the server later in this chapter. Attributes are also a key component of other signals, so we'll come back to them throughout the book. One last thing to be aware of when thinking of attributes, and really any data being recorded in traces, is to be cognizant of **Personally Identifiable Information (PII)**. Whenever possible, save yourself the trouble and remove all PII from the telemetry. We'll cover more on this topic in *Chapter 8, OpenTelemetry Collector*.

SpanKind

Another piece of information that is useful about a span is `SpanKind`. `SpanKind` is a qualifier that categorizes the span and provides additional information about the relationship between spans in a trace. The following categories for span kinds are defined in OpenTelemetry:

- **INTERNAL**: This indicates that the span represents an operation that is internal to an application, meaning that this specific operation has no external dependencies or relationships. This is the default value for a span when not set.
- **CLIENT**: This identifies the span as an operation making a request to a remote service, which should be identified as a *server* span. The request made by this operation is synchronous, and the client should wait for a response from the server.
- **SERVER**: This indicates that the span is an operation responding to a synchronous request from a *client* span. In a client/server, the *client* is identified as the parent span to the *server*, as it is the originator of the request.
- **PRODUCER**: This identifies the operation as an originator of an asynchronous request. Unlike in the case of the *client* span, the *producer* is not expecting a response from the *consumer* of the asynchronous request.

- CONSUMER: This identifies the operation as a *consumer* of an asynchronous request from a *producer*.

As you may have noticed so far, all the spans that we've created have been identified as *internal*. The following information can be found throughout the output we've generated until now:

```
"kind": "SpanKind.INTERNAL"
```

Now is a good time to start making the shopper application a bit more realistic by adding some calls to a grocery store server. Knowing that this will be a client using HTTP requests to retrieve data from the server, we will set SpanKind to CLIENT on the operation that makes a call to the server. On the receiving side, we will set SpanKind on the operation that is responding to the request to SERVER. The way to set kind is by passing the kind argument when creating the span. The following code adds a web request from the client to the server in the `browse` method. The HTTP request will be facilitated by using the `requests` (<https://docs.python-requests.org/>) library. The request to the server will be wrapped by a context manager, which starts a new span named `web request` with the kind set to CLIENT:

shopper.py

```
import requests
from common import configure_tracer

@tracer.start_as_current_span("browse")
def browse():
    print("visiting the grocery store")
    with tracer.start_as_current_span(
        "web request", kind=trace.SpanKind.CLIENT
    ) as span:
        url = "http://localhost:5000"
        span.set_attributes(
            {
                SpanAttributes.HTTP_METHOD: "GET",
                SpanAttributes.HTTP_FLAVOR:
str(HttpFlavorValues.HTTP_1_1),
                SpanAttributes.HTTP_URL: url,
                SpanAttributes.NET_PEER_IP: "127.0.0.1",
            }
        )
```

```
)  
    resp = requests.get(url)  
    span.set_attribute(SpanAttributes.HTTP_STATUS_CODE,  
    resp.status_code)
```

So far, all the code written was done on the client side; let's talk about the server side. Before starting on the server, in order to reduce the duplication of code, `configure_tracer` has been moved into a separate `common.py` module and placed in the same directory as the rest of the code. In this refactor, we've also updated the previously hardcoded `service.name` and `service.version` attribute keys to use values from the semantic conventions package:

common.py

```
from opentelemetry import trace  
from opentelemetry.sdk.resources import Resource  
from opentelemetry.sdk.trace import TracerProvider  
from opentelemetry.sdk.trace.export import BatchSpanProcessor,  
ConsoleSpanExporter  
from opentelemetry.semconv.resource import ResourceAttributes  
from local_machine_resource_detector import  
LocalMachineResourceDetector  
  
def configure_tracer(name, version):  
    exporter = ConsoleSpanExporter()  
    span_processor = BatchSpanProcessor(exporter)  
    local_resource = LocalMachineResourceDetector().detect()  
    resource = local_resource.merge(  
        Resource.create(  
            {  
                ResourceAttributes.SERVICE_NAME: name,  
                ResourceAttributes.SERVICE_VERSION: version,  
            }  
        )  
    )  
    provider = TracerProvider(resource=resource)
```

```
provider.add_span_processor(span_processor)
trace.set_tracer_provider(provider)
return trace.get_tracer(name, version)
```

This code can now be used in both `shopper.py` and the new server code in `grocery_store.py` to instantiate a tracer. The server code uses Flask (<https://flask.palletsprojects.com/en/1.1.x/>) to provide an API, and the initial code for the application will implement a single route handler. We won't dive too deeply into the nuts and bolts of how Flask works in this book. For the purpose of our application, it's enough to know that the response handler can be configured with a path via the `route` decorator and that the `run` method launches a web server. An additional decorator to create a span on the handler sets the kind to SERVER, as it is the operation that is responding to the CLIENT span instrumented previously. Note that in the code, there are also several attributes being set following the semantic conventions; the Flask library conveniently makes most of the information available quite easily:

grocery_store.py

```
from flask import Flask, request
from opentelemetry import trace
from opentelemetry.semconv.trace import HttpFlavorValues,
SpanAttributes
from opentelemetry.trace import SpanKind
from common import configure_tracer

tracer = configure_tracer("0.1.2", "grocery-store")
app = Flask(__name__)

@app.route("/")
@tracer.start_as_current_span("welcome", kind=SpanKind.SERVER)
def welcome():
    span = trace.get_current_span()
    span.set_attributes(
        {
            SpanAttributes.HTTP_FLAVOR: request.environ.get("SERVER_PROTOCOL"),
            SpanAttributes.HTTP_METHOD: request.method,
            SpanAttributes.HTTP_USER_AGENT: str(request.user_agent),
        }
    )
    return "Welcome to the grocery store!"
```

```
        SpanAttributes.HTTP_HOST: request.host,
        SpanAttributes.HTTP_SCHEME: request.scheme,
        SpanAttributes.HTTP_TARGET: request.path,
        SpanAttributes.HTTP_CLIENT_IP: request.remote_addr,
    }
)
return "Welcome to the grocery store!"

if __name__ == "__main__":
    app.run()
```

Of course, to see the traces we must first run the application; to get the server running, use the following command:

```
python grocery_store.py
```

If another application is already running on the default port that Flask uses, 5000, you may encounter the `Address already in use` error. Ensure only one instance of the server is running at any given time.

Important Note

It's possible to run the server with `debug` mode enabled to have it automatically updated every time the code changes. This is convenient when doing rapid development but should never be left enabled outside of development. Debug mode also causes problems with auto-instrumentation, as we discussed in *Chapter 3, Auto-Instrumentation*. Enabling debug mode is accomplished by calling the `run` method as follows: `run(debug=True)`.

In any future examples, the server will always need to be run before the shopper; otherwise, the client application will throw HTTP connection exceptions. I find it particularly helpful to use a terminal that supports split screens to have both the client and the server running side by side. Let's run both applications now and inspect the output data emitted. The server operation named / will be identified as a SERVER span:

grocery_store.py output

```
{  
    "name": "/",
    "context": {
        "trace_id": "0xe7f562a98f81a36ba81aaf1e239dd718",
        "span_id": "0x51daed87f12f5bc0",
        "trace_state": "[]"
    },
    "kind": "SpanKind.SERVER",
    "parent_id": null,
}
```

On the client side, the operation named `web request` will be identified as a `CLIENT` span:

shopper.py output

```
{  
    "name": "web request",
    "context": {
        "trace_id": "0xc2747c6a8c7f7e12618bf69d7d71a1c8",
        "span_id": "0x88b7afb56d248244",
        "trace_state": "[]"
    },
    "kind": "SpanKind.CLIENT",
    "parent_id": "0xe756587bc381338c",
}
```

This new data is starting to help define the ties between different services and describe the relationships between the components of the system, which is great. By exploring the tracing data alone, we can start getting a clearer idea of the role that each application plays. Oddly enough though, the data we're currently generating doesn't appear to be fully connected yet. The `trace_id` identifier between the client and the server doesn't match, and moreover, the `SERVER` span doesn't contain `parent_id`; it seems we forgot about propagation!

Propagating context

Getting the information from one service to another across the network boundary requires some additional work, namely, propagating the context. Without this context propagation, each service will generate a new trace independently, which means that the backend will not be able to tie the services together at analysis time. As shown in *Figure 4.5*, a trace without propagation between services is missing the link between services, which means the traces will be more difficult to correlate:

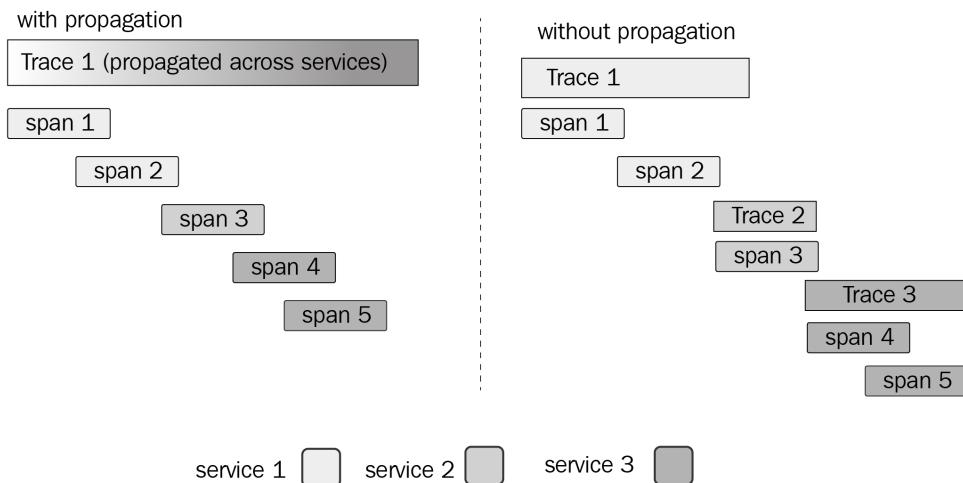


Figure 4.5 – Traces with and without propagation

Specifically, the data needed to propagate the context between services is `span_context`. This includes four key pieces of information:

- `span_id`: The identifier of the current span
- `trace_id`: The identifier of the current trace
- `trace_flags`: The additional configuration flags available to control tracing levels and sampling, as per the W3C Trace Context specification (<https://www.w3.org/TR/trace-context/#trace-flags>)
- `trace_state`: A set of vendor-specific identification data as per the W3C Trace Context specification (<https://www.w3.org/TR/trace-context/#tracestate-header>)

The `span_context` information is used anytime a new span is started. `trace_id` is set as the current new span's trace ID, and `span_id` will be used as the new span's parent ID. When a new span is started in a different service, if the context isn't propagated correctly, the new span has no information from which to pull the data it needs. Context must be serialized and injected across boundaries into a carrier for propagation to occur. On the receiving end, the context must be extracted from the carrier and deserialized. The carrier medium used to transport the context, in the case of our application, is HTTP headers. OpenTelemetry's **Propagators API** provides the methods we'll use in the next example. On the client side, we'll call the `inject` method to set `span_context` in a dictionary that will be passed into the HTTP request as headers:

shopper.py

```
from opentelemetry.propagate import inject
@tracer.start_as_current_span("browse")
def browse():
    print("visiting the grocery store")
    with tracer.start_as_current_span(
        "web request", kind=trace.SpanKind.CLIENT
    ) as span:
        headers = {}
        inject(headers)
        resp = requests.get(url, headers=headers)
```

On the server side, it is a little more complicated, as we need to ensure the context is extracted before the decorator instantiates the span in the request handler. Conveniently, Flask has a mechanism available via decorators to call methods before and after a request is handled. This allows us to extract the context from the request headers and attach to the context before the request handler is called. The call to attach will return a token that will be stored in the context of the request. Once the request has been processed, the call to detach restores the previous context:

grocery_store.py

```
from opentelemetry import context
from opentelemetry.propagate import extract

@app.before_request
def before_request_func():
    context.attach(extract(request.headers))
```

```
    token = context.attach(extract(request.headers))
    request.environ["context_token"] = token

@app.teardown_request
def teardown_request_func(err):
    token = request.environ.get("context_token", None)
    if token:
        context.detach(token)
```

Testing the new code will show that the context is now propagated; remember to restart the server as well as run the client. Take a look at the following output, paying special attention to `trace_id` and `span_id`:

shopper.py output

```
{  
    "name": "web request",  
    "context": {  
        "trace_id": "0x1fe2dc4e2e750e4598463749300277ed",  
        "span_id": "0x5771b0a074e00a5b",  
        "trace_state": "[]"  
    },  
    "kind": "SpanKind.CLIENT",  
}
```

If everything went according to plan, the client and the server should be part of the same trace. The output on the server side shows the span containing a `parent_id` field which matches the client's `span_id` field. As well, note the `trace_id` field which matches on both sides of the request:

grocery_store.py output

```
{  
    "name": "/",  
    "context": {  
        "trace_id": "0x1fe2dc4e2e750e4598463749300277ed",  
        "span_id": "0x26f143d0f8a9c0bd",  
        "trace_state": "[]"  
    },  
}
```

```
"kind": "SpanKind.SERVER",
"parent_id": "0x5771b0a074e00a5b",
}
```

Now that the services are connected, let's explore propagation a bit further!

Additional propagator formats

The propagation we've used so far in the example is the **W3C Trace Context** propagation format. The Trace Context format is fairly recent and is by no means the only propagation format out there. There are additional propagators available to use with OpenTelemetry to support interoperability with other tracing standards. Additional propagators supported by OpenTelemetry and available at the time of writing include B3, Jaeger, and ot-trace. Currently supported propagators implement a `TextMapPropagator` interface with an `inject` method and an `extract` method. A propagator is configured globally using the `set_global_textmap` method. The following code shows an example of configuring a `B3MultiFormat` propagator for an application. This propagator can be found by installing the `opentelemetry-propagator-b3` package:

```
from opentelemetry.propagators.b3 import B3MultiFormat
from opentelemetry.propagate import set_global_textmap

set_global_textmap(B3MultiFormat())
```

Important Note

Troubleshooting propagation issues can be difficult and time-consuming. Services can easily be misconfigured to propagate data using different formats and doing so will result in propagation not working at all.

If you decided to use the previous code in either the shopper or the grocery store applications but not both, you may have noticed propagation breaking. It's not uncommon for applications in the wild to have different propagation formats configured. Thankfully, it's possible to configure multiple propagators simultaneously in OpenTelemetry by using a composite propagator.

Composite propagator

A composite propagator allows users to configure multiple propagators from different cross-cutting concerns. In its current implementation in many languages, the composite propagator can support multiple propagators for the same signal. This functionality provides backward compatibility with older systems while being future-proof.

`CompositePropagator` has the same interface as any propagator but supports passing in a list of propagators at initialization. This list is then iterated through at injection and extraction time. This next example introduces one additional service, a legacy inventory system that is configured to use B3 propagation. *Figure 4.6* shows the flow of the request from the shopper, through the store, and to the inventory system that we will be adding in the next example:

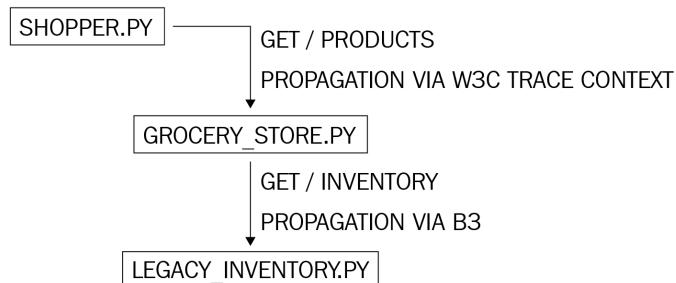


Figure 4.6 – A request to the legacy inventory system

Since the grocery store needs to propagate requests using both W3C Trace Context and B3, we'll need to update the code to configure `CompositePropagator` to support this. The first thing to do before diving into the code is to ensure that the B3 propagator package is installed:

```
pip install opentelemetry-propagator-b3
```

For the sake of simplifying the server code, the following code shows a new method being added to `common.py` to set span attributes in a server handler. This new method, `set_span_attributes_from_flask`, can be used both in `legacy_inventory.py` (as we'll see shortly) and in `grocery_store.py`:

common.py

```
from flask import request
from opentelemetry.semconv.trace import SpanAttributes

def set_span_attributes_from_flask():
```

```
span = trace.get_current_span()
span.set_attributes(
{
    SpanAttributes.HTTP_FLAVOR: request.environ.get("SERVER_PROTOCOL"),
    SpanAttributes.HTTP_METHOD: request.method,
    SpanAttributes.HTTP_USER_AGENT: str(request.user_agent),
    SpanAttributes.HTTP_HOST: request.host,
    SpanAttributes.HTTP_SCHEME: request.scheme,
    SpanAttributes.HTTP_TARGET: request.path,
    SpanAttributes.HTTP_CLIENT_IP: request.remote_addr,
}
)
```

The `legacy_inventory.py` service code is another Flask server application with a single handler that, for now, returns a hardcoded list of items and quantities encoded using JSON. The following code is very similar to the grocery store code. The configuration for both the Flask app and OpenTelemetry should be familiar, the significant difference being how we configure OpenTelemetry in this new service is the propagator, by calling `set_global_textmap`. It's also important to remember to set the port number to a different value than the default Flask port by passing an argument to `app.run`; otherwise, we will run into a socket error when trying to run both `grocery_store.py` and `legacy_inventory.py`:

legacy_inventory.py

```
from flask import Flask, jsonify, request
from opentelemetry import context
from opentelemetry.propagate import extract, set_global_textmap
from opentelemetry.propagators.b3 import B3MultiFormat
from opentelemetry.trace import SpanKind
from common import configure_tracer, set_span_attributes_from_flask

tracer = configure_tracer("legacy-inventory", "0.9.1")
app = Flask(__name__)
set_global_textmap(B3MultiFormat())
```

```
@app.before_request
def before_request_func():
    token = context.attach(extract(request.headers))
    request.environ["context_token"] = token

@app.teardown_request
def teardown_request_func(err):
    token = request.environ.get("context_token", None)
    if token:
        context.detach(token)

@app.route("/inventory")
@tracer.start_as_current_span("/inventory", kind=SpanKind.SERVER)
def inventory():
    set_span_attributes_from_flask()
    products = [
        {"name": "oranges", "quantity": "10"},
        {"name": "apples", "quantity": "20"},
    ]
    return jsonify(products)

if __name__ == "__main__":
    app.run(debug=True, port=5001)
```

In the grocery store application, we will configure CompositePropagator to support both the W3C Trace Context and B3 formats. Add the following code to `grocery_store.py`:

grocery_store.py

```
from opentelemetry.propagate import extract, inject, set_global_textmap
from opentelemetry.propagators.b3 import B3MultiFormat
from opentelemetry.propagators.composite import CompositePropagator
from opentelemetry.trace.propagation import tracecontext
```

```
set_global_textmap(CompositePropagator([tracecontext.  
TraceContextTextMapPropagator(), B3MultiFormat()])))
```

Additionally, the following handler will be added to the store, which will make a request to the legacy inventory service. The key thing to remember here is to ensure the context is present in the headers by calling `inject` and that the headers are passed into the request:

grocery_store.py

```
import requests  
  
from common import set_span_attributes_from_flask  
  
...  
  
@app.route("/")  
@tracer.start_as_current_span("welcome", kind=SpanKind.SERVER)  
def welcome():  
    set_span_attributes_from_flask()  
    return "Welcome to the grocery store!"  
  
  
@app.route("/products")  
@tracer.start_as_current_span("/products", kind=SpanKind.  
SERVER)  
def products():  
    set_span_attributes_from_flask()  
    with tracer.start_as_current_span("inventory request") as  
span:  
        url = "http://localhost:5001/inventory"  
        span.set_attributes(  
            {  
                SpanAttributes.HTTP_METHOD: "GET",  
                SpanAttributes.HTTP_FLAVOR:  
str(HttpFlavorValues.HTTP_1_1),  
                SpanAttributes.HTTP_URL: url,  
                SpanAttributes.NET_PEER_IP: "127.0.0.1",  
            }  
        )  
        headers = {}  
        inject(headers)  
        resp = requests.get(url, headers=headers)  
        return resp.text
```

The last change we need before trying this out is an update to the shopper application's `browse` method to send a request to the new endpoint:

shopper.py

```
def browse():
    print("visiting the grocery store")
    with tracer.start_as_current_span(
        "web request", kind=trace.SpanKind.CLIENT
    ) as span:
        url = "http://localhost:5000/products"
```

Now, we have a third application to launch; the following commands need to be run from separate terminal windows, and remember to ensure no other applications are running on ports 5000 and 5001 to avoid socket errors:

```
$ python ./legacy_inventory.py
$ python ./grocery_store.py
$ python ./shopper.py
```

Once the legacy inventory server is up and running, making a request from the shopper should yield some exciting results. In the output, we'll be looking for `trace_id` to be consistent across all three services, and, as in the previous example of propagation, `parent_id` of the server span should match `span_id` of the corresponding client request span:

shopper.py output

```
"name": "web request",
"context": {
    "trace_id": "0xb2a655bfd008007711903d8a72130813",
    "span_id": "0x3c183afa2640a2bb",
},
```

The following output from the grocery store includes two spans. The span named `/products` represents the request received from the client, and if the context is successfully extracted, `trace_id` will match the previous output. The second span is the request to the inventory service:

grocery_store.py output

```
"name": "/products",
"context": {
    "trace_id": "0xb2a655bfd008007711903d8a72130813",
    "span_id": "0x77883e3459f83fb6",
},
"parent_id": "0x3c183afa2640a2bb",
-----
"name": "inventory request",
"context": {
    "trace_id": "0xb2a655bfd008007711903d8a72130813",
    "span_id": "0x8137dbaaa3f40062",
},
"parent_id": "0x77883e3459f83fb6",
```

The last output is from the inventory service. Remember that this service is using a different propagator format. If the propagation was configured correctly, `trace_id` should remain consistent with the other two services, and `parent_id` should reflect that the parent operation is the `inventory request` span:

legacy_inventory.py output

```
"name": "/inventory",
"context": {
    "trace_id": "0xb2a655bfd008007711903d8a72130813",
    "span_id": "0x3306b21b8000912b",
},
"parent_id": "0x8137dbaaa3f40062",
```

This was a lot of work, but once you get propagation configured and working across a system, it's rare that you'll need to go back and make changes to it. It's a set-it-and-forget-it type of operation. If you happen to be working with a brand-new code base, choose a single propagation format and stick to it; it will save you a lot of headaches. We've now grasped one of the most important concepts in distributed tracing, the propagation of span context across systems. Let's take a look at where else propagation can help us.

Important Note

A possible alternative when working with large code bases and multiple propagator formats is to always configure all available propagation formats. This may seem like overkill, but sometimes, it makes sense to prioritize interoperability over saving a few bytes.

Recording events, exceptions, and status

Quickly identifying when an issue arises is a key aspect of distributed tracing. As demonstrated in *Figure 4.7* with the Jaeger interface, in many backends, traces that contain errors are highlighted to make them easy to find for users of data:

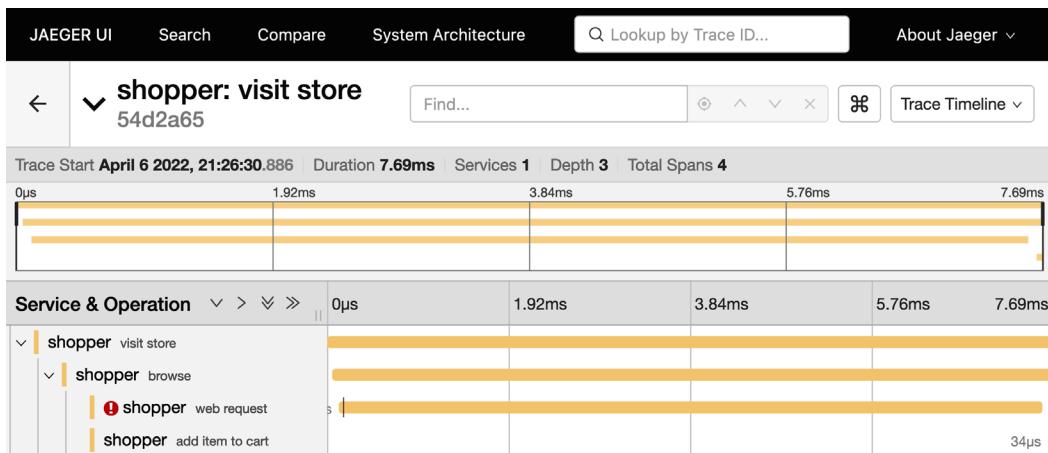


Figure 4.7 – The trace view in Jaeger

In the following sections, we will explore the facilities that OpenTelemetry provides to capture events, record exceptions, and set the status of a span.

Events

In addition to attributes, an **event** provides the facility to record data about a span that occurs at a specific time. Events are similar to logs in OpenTracing in that they contain a timestamp and can contain a list of attributes or key/value pairs. An event is added via an `add_event` method on the span, which accepts a name argument and, optionally, a timestamp and a list of attributes, as shown in the following code:

shopper.py

```
span.add_event("about to send a request")
resp = requests.get(url, headers=headers)
span.add_event("request sent", attributes={"url": url},
timestamp=0)
```

As you'll see in the following output, the list of events is kept in the order in which they are added; they are not ordered by the timestamps they are recorded with:

shopper.py output

```
"events": [
    {
        "name": "about to send a request",
        "timestamp": "2021-07-12T06:38:49.793903Z",
        "attributes": {}
    },
    {
        "name": "request sent",
        "timestamp": "1970-01-01T00:00:00.000000Z",
        "attributes": {
            "url": "http://localhost:5000/products"
        }
    }
],
```

Events differ from attributes in that they have a time dimension to them, which can be helpful to better understand the sequence of things inside a span. There are also events that have a special meaning, as we'll see with exceptions.

Exceptions

In OpenTelemetry, the concepts of exceptions and the status of a span are intentionally kept separate. A span may contain many exceptions, but these exceptions don't necessarily mean that the status of the span should be set as an error. For example, a user may want to record exceptions when a request is made to a specific service, but there may be retry logic that will cause the operation to eventually succeed anyway. Recording those exceptions may be useful to identify areas of the code that can be improved. The initial definition of an **exception** in the OpenTelemetry specification is that an exception is as follows:

- Recorded as an event
- The specific name exception
- Contains the minimum of either `exception.type` or an `exception.message` attribute

The following code records an exception if a request to the grocery store fails by creating one such event. Let's add a `try/except` block in the `browse` method to capture the exception and change `url` to make the request intentionally fail:

shopper.py

```
try:  
    url = "invalid_url"  
    resp = requests.get(url, headers=headers)  
    span.add_event(  
        "request sent",  
        attributes={"url": url},  
        timestamp=0,  
    )  
    span.set_attribute(  
        SpanAttributes.HTTP_STATUS_CODE,  
        resp.status_code  
    )  
except Exception as err:  
    attributes = {  
        SpanAttributes.EXCEPTION_MESSAGE: str(err),  
    }  
    span.add_event("exception", attributes)
```

Running the code will produce an exception that will be caught. This exception will then be recorded as an event and added to the tracing data emitted at the console:

shopper.py output

```
"events": [
    {
        "name": "exception",
        "timestamp": "2021-07-10T04:13:05.287376Z",
        "attributes": {
            "exception.message": "Invalid URL 'invalid_url': No schema supplied. Perhaps you meant http://invalid_url?"
        }
    }
]
```

Although this provides us with more information, it's not practical to have to write so many lines of code every time we want to record an exception. Thankfully, the OpenTelemetry specification has defined a `span` method in the API to address this. The following code replaces the code in the `except` block of the previous example to use the `record_exception` method on the `span`, instead of manually creating an event. Semantically, these are equivalent, but the method is much more convenient. The method accepts an exception as its first argument and supports optional parameters to pass in additional event attributes, as well as a timestamp:

shopper.py

```
try:
    url = "invalid_url"
    resp = requests.get(url, headers=headers)
    ...
except Exception as err:
    span.record_exception(err)
```

Next time the code is run, the exception event is automatically generated for us. Taking a closer look at the output, it's even more useful than before, as we now see the following:

- The message populated as we did before
- The exception type
- A stack trace capturing exactly where in the code the exception was raised

This allows us to immediately find the problematic code and resolve the issue:

shopper.py output

```
"events": [
  {
    "name": "exception",
    "timestamp": "2021-07-10T04:17:07.328665Z",
    "attributes": {
      "exception.type": "MissingSchema",
      "exception.message": "Invalid URL 'invalid_url': No schema supplied. Perhaps you meant http://invalid_url?'",
      "exception.stacktrace": "Traceback (most recent call last):\n  File \"/Users/alex/dev/cloud_native_observability/lib/python3.8/site-packages/opentelemetry/trace/_init_.py\", line 522, in use_span\n    yield span\n  File \"/Users/alex/dev/cloud_native_observability/lib/python3.8/site-packages/opentelemetry/sdk/trace/_init_.py\", line 879, in start_as_current_span\n    yield span\n  context\n  File \"/Users/alex/dev/cloud-native-observability/chapter4/.shopper.py\", line 110, in browse\n    resp =\n    requests.get(\"invalid_url\", headers=headers)\n  File \"/Users/alex/dev/cloud_native_observability/lib/python3.8/site-packages/requests/api.py\", line 76, in get\n    return\n    request('get', url, params=params, **kwargs)\n  File \"/Users/alex/dev/cloud_native_observability/lib/python3.8/site-packages/requests/api.py\", line 61, in request\n    return\n    session.request(method=method, url=url, **kwargs)\n  File \"/Users/alex/dev/cloud_native_observability/lib/python3.8/site-
```

```

packages/requests/sessions.py\", line 528, in request\n    prep
= self.prepare_request(req)\n    File \"/Users/alex/dev/cloud_
native_observability/lib/python3.8/site-packages/requests/
sessions.py\", line 456, in prepare_request\n        p.prepare(\n    File \"/Users/alex/dev/cloud_native_observability/lib/
python3.8/site-packages/requests/models.py\", line 316, in
prepare\n        self.prepare_url(url, params)\n    File \"/Users/
alex/dev/cloud_native_observability/lib/python3.8/site-
packages/requests/models.py\", line 390, in prepare_url\n
raise MissingSchema(error)\nrequests.exceptions.MissingSchema:
Invalid URL 'invalid_url': No schema supplied. Perhaps you
meant http://invalid_url?\n",
            "exception.escaped": "False"
        }
    }
],

```

This type of detail about exceptions in a system is incredibly valuable when debugging, especially when the events may have occurred minutes, hours, or even days ago. It's worth noting that the format of the stack trace is language-specific, as described in *Figure 4.8* (https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/trace/semantic_conventions/exceptions.md#stacktrace-representation):

Language	Format
C#	the return value of Exception.ToString()
Go	the return value of runtime.Stack
Java	the contents of Throwable.printStackTrace()
Javascript	the return value of error.stack as returned by V8
Python	the return value of traceback.format_exc()
Ruby	the return value of Exception.full_message

Figure 4.8 – The stack trace format per language

Additionally, the Python SDK also automatically captures uncaught exceptions and adds an exception event to the span that is active when the exception occurs. We can update the code we just wrote in the previous example to remove the `try/except` block, leaving the invalid URL. The following code has the same effect as calling `record_exception` directly:

shopper.py

```
resp = requests.get("invalid_url", headers=headers)
```

Recording exceptions in spans is valuable, but in the event that it is preferable not to do so, it's possible to set an optional flag when creating a span to disable the functionality. You can try it in the previous example by setting the `record_exception` optional argument, as follows:

shopper.py

```
with tracer.start_as_current_span(
    "web request", kind=trace.SpanKind.CLIENT, record_
exception=False
) as span:
```

Now that we understand how exceptions are recorded, let's further investigate how or even if these exceptions connect to the status of a span.

Status

As mentioned previously in this chapter, the span status has significant benefits to users. Quickly being able to filter through traces based on the span status makes things much easier for operators. The *status* is composed of a status code and, optionally, a description. There are currently three supported span status codes:

- UNSET
- OK
- ERROR

The default status code on any new span is UNSET. This default behavior ensures that when a span status code is set to OK, it has been done intentionally. An earlier version of the specification defaulted a span status code to OK, which left room for misinterpretations – was the span really OK or did the code return before an error status code was set? The decision to set the span status is really up to the application developer or operators of the service. The interface to set a status on a span receives a `Status` object, which is composed of `StatusCode` and a description string. This next example sets the span status code to OK based on the response from the web request. Note that we're using a feature of the Requests library's `Response` object to return `True` if the HTTP status code on the response is between 200 and 400:

shopper.py

```
from opentelemetry.trace import Status, StatusCode
def browse():
    with tracer.start_as_current_span(
        "web request", kind=trace.SpanKind.CLIENT, record_
exception=False
    ) as span:
        url = "http://localhost:5000/products"
        resp = requests.get(url, headers=headers)
        if resp:
            span.set_status(Status.StatusCode.OK)
        else:
            span.set_status(
                Status.StatusCode.ERROR, "status code: {}".
format(resp.status_code))
    )
```

With the code in place, test the application first with the `http://localhost:5000/products` URL to see the following output when a valid URL is used:

shopper.py output with valid URL

```
"status": {
    "status_code": "OK"
}
```

Now, update the URL to an invalid endpoint such as `http://localhost:5000/invalid` to see the following output when the response contains an error code:

shopper.py output with invalid URL

```
"status": {  
    "status_code": "ERROR",  
    "description": "status code: 404"  
}
```

Important Note

The `description` field will only be used if the status code is set to `ERROR`; it is ignored otherwise.

Another thing to note about status codes is that, as per semantic convention, instrumentation libraries should not change the status code to `OK` unless they are providing a configuration option to do this. This is to prevent having an instrumentation library unexpectedly change the outcome of the span. They are, however, encouraged to set the status code to `ERROR` when errors defined in the semantic convention for the type of instrumentation library are encountered.

As with recording exceptions, it's also possible to configure spans to automatically set the status when an exception occurs. This is accomplished via a `set_status_on_exception` argument, available when starting a span:

shopper.py

```
with tracer.start_as_current_span(  
    "web request",  
    kind=trace.SpanKind.CLIENT,  
    set_status_on_exception=True,  
) as span:
```

Play around with the code and see what the status output is when using this setting. Although it may seem like a lot of work, handling errors and setting the status on spans meaningfully will make a world of difference at analysis time. Not only that, but having to work through the different scenarios in the code at instrumentation time is a forcing function to really ensure a solid understanding of what the code is expected to do. And when things go wrong, as they will, having this data will make a world of difference.

Summary

And just like that, you've explored many important concepts of the tracing signal in OpenTelemetry! There was quite a bit to grasp in this chapter, but hopefully, the concepts we've been exploring so far are starting to make more sense now that there's some code behind them. With this knowledge, you now know how to configure different components of the OpenTelemetry tracing pipeline to obtain a tracer and export data to the console. You also have the ability to start spans in various ways, depending on your application's needs. We then spent some time improving the data emitted by enriching it using attributes, resources, and resource detectors. Last but not least, we took a look at the important topic of events, status, and exceptions to capture some important information about errors when they happen in code.

Our understanding of the Context API will allow us to share information across our application, and knowing how to use the Propagation API will allow us to ensure that information is shared across application boundaries.

Although you probably have many more questions, you now know enough to look through some existing applications or plan ahead for instrumenting new applications through distributed tracing. As some of the components we've explored in this chapter are similar across signals, many of the concepts that may not quite make sense yet will become clearer as we take a look at the next chapter, which looks at metrics. Let's go measure some things!

5

Metrics – Recording Measurements

Tracing code execution throughout a system is one way to capture information about what is happening in an application, but what if we're looking to measure something that would be better served by a more lightweight option than a trace? Now that we've learned how to generate distributed traces using OpenTelemetry, it's time to look at the next signal: **metrics**. As we did in *Chapter 4, Distributed Tracing – Tracing Code Execution*, we will first look at configuring the OpenTelemetry pipeline to produce metrics. Then, we'll continue to improve the telemetry emitted by the grocery store application by using the instruments OpenTelemetry puts at our disposal. In this chapter, we will do the following:

- Configure OpenTelemetry to collect, aggregate, and export metrics to the terminal.
- Generate metrics using the different instruments available.
- Use metrics to gain a better understanding of the grocery store application.

Augmenting the grocery store application will allow us to put the different instruments into practice to grasp better how each instrument can be used to record measurements. As we explore other metrics that are useful to produce for cloud-native applications, we will seek to understand some of the questions we may answer using each instrument.

Technical requirements

As with the examples in the previous chapter, the code is written using **Python 3.8**, but OpenTelemetry Python supports **Python 3.6+** at the time of writing. Ensure you have a compatible version installed on your system following the instructions at <https://docs.python.org/3/using/index.html>. To verify that a compatible version is installed on your system, run the following commands:

```
$ python --version  
$ python3 --version
```

On many systems, both `python` and `python3` point to the same installation, but this is not always the case, so it's good to be aware of this if one points to an unsupported version. In all examples, running applications in Python will call the `python` command, but they can also be run via the `python3` command, depending on your system.

The first few examples in this chapter will show a standalone example exploring how to configure OpenTelemetry to produce metrics. The code will require the OpenTelemetry API and SDK packages, which we'll install via the following `pip` command:

```
$ pip install opentelemetry-api==1.10.0 \  
    opentelemetry-sdk==1.10.0 \  
    opentelemetry-propagator-b3==1.10.0
```

Additionally, we will use the **Prometheus** exporter to demonstrate a pull-based exporter to emit metrics. This exporter can be installed via `pip` as well:

```
$ pip install opentelemetry-exporter-prometheus==0.29b0
```

For the later examples involving the grocery store application, you can download the sample from *Chapter 4, Distributed Tracing – Tracing Code Execution*, and add the code along with the examples. The following `git` command will clone the companion repository:

```
$ git clone https://github.com/PacktPublishing/Cloud-Native-  
    Observability
```

The chapter04 directory in the repository contains the code for the grocery store. The complete example, including all the code in the examples from this chapter, is available in the chapter05 directory. I recommend adding the code following the examples and using the complete example code as a reference if you get into trouble. Also, if you haven't read *Chapter 4, Distributed Tracing – Tracing Code Execution*, it may be helpful to skim through the details of how the grocery store application is built in that chapter to get your bearings.

The grocery store depends on the **Requests** library (<https://docs.python-requests.org/>) to make web requests at various points and the **Flask** library (<https://flask.palletsprojects.com>) to provide a lightweight web server for some of the services. Both libraries can be installed via the following pip command:

```
$ pip install flask requests
```

Additionally, the chapter will utilize a third-party open source tool (<https://github.com/raky11/hey>) to generate some load on the web application. The tool can be downloaded from the repository. The following commands download the macOS binary and rename it to hey using curl with the -o flag, then ensure the binary is executable using chmod:

```
$ curl -o hey https://hey-release.s3.us-east-2.amazonaws.com/  
hey_darwin_amd64  
$ chmod +x ./hey
```

If you have a different load generation tool you're familiar with, and there are many, feel free to use that instead if you prefer. This should be everything we need to start; let's start measuring!

Configuring the metrics pipeline

The metrics signal was designed to be conceptually similar to the tracing signal. The metrics pipeline consists of the following:

- A **MeterProvider** to determine how metrics should be generated and provide access to a **meter**.
- The meter is used to create **instruments**, which are used to record **measurements**.
- **Views** allow the application developer to filter and process metrics produced by the **software development kit (SDK)**.

- A **MetricReader**, which collects **metrics** being recorded.
- The **MetricExporter** provides a mechanism to translate metrics into an output format for various protocols.

There are quite a few components, and a picture always helps me grasp concepts more quickly. The following figure shows us the different elements in the pipeline:

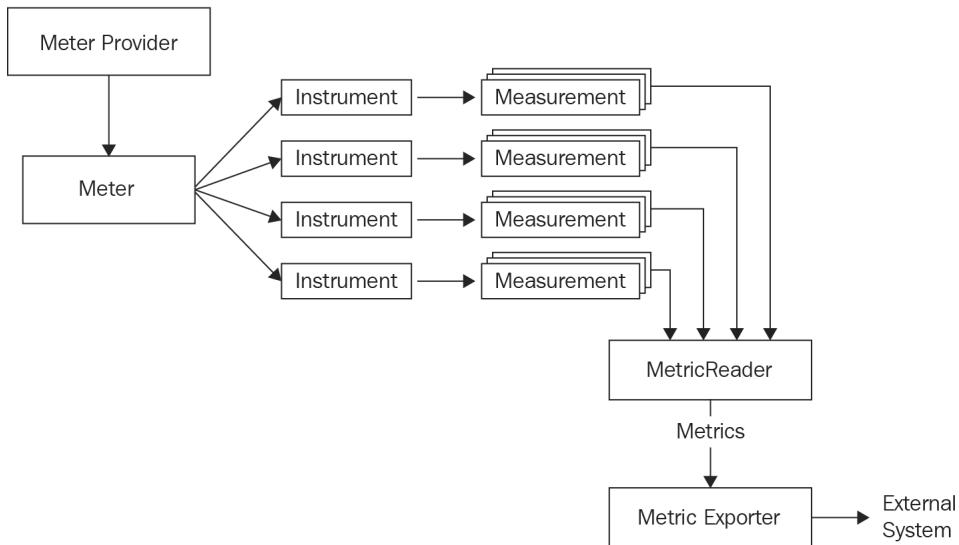


Figure 5.1 – Metrics pipeline

MeterProvider can be associated with a resource to identify the source of metrics produced. We'll see shortly how we can reuse the `LocalMachineResourceDetector` we created in *Chapter 4, Distributed Tracing – Tracing Code Execution*, with metrics. For now, the first example instantiates `MeterProvider` with an empty resource. The code then calls the `set_meter_provider` global method to set the `MeterProvider` for the entire application.

Add the following code to a new file named `metrics.py`. Later in the chapter, we will refactor the code to add a `MeterProvider` to the grocery store, but to get started, the simpler, the better.

metrics.py

```
from opentelemetry._metrics import set_meter_provider
from opentelemetry.sdk._metrics import MeterProvider
from opentelemetry.sdk.resources import Resource
```

```
def configure_meter_provider():
    provider = MeterProvider(resource=Resource.create())
    set_meter_provider(provider)

if __name__ == "__main__":
    configure_meter_provider()
```

Run the code with the following command to ensure it runs without any errors:

```
python ./metrics.py
```

No errors and no output? Well done, you're on the right track!

Important Note

The previous code shows that the metric modules are located at `_metrics`. This will change to `metrics` once the packages have been marked stable. Depending on when you're reading this, it may have already happened.

Next, we'll need to configure an exporter to tell our application what to do with metrics once they're generated. The OpenTelemetry SDK contains `ConsoleMetricExporter` that emits metrics to the console, useful when getting started and debugging. `PeriodicExportingMetricReader` can be configured to periodically export metrics. The following code configures both components and adds the reader to the `MeterProvider`. The code sets the export interval to 5000 milliseconds, or 5 seconds, overriding the default of 60 seconds:

metrics.py

```
from opentelemetry._metrics import set_meter_provider
from opentelemetry.sdk._metrics import MeterProvider
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk._metrics.export import (
    ConsoleMetricExporter,
    PeriodicExportingMetricReader,
)

def configure_meter_provider():
    exporter = ConsoleMetricExporter()
```

```
reader = PeriodicExportingMetricReader(exporter, export_
interval_millis=5000)
provider = MeterProvider(metric_readers=[reader],
resource=Resource.create())
set_meter_provider(provider)

if __name__ == "__main__":
    configure_meter_provider()
```

Run the code once more. The expectation is that the output from running the code will still not show anything. The only reason to run the code is to ensure our dependencies are fulfilled, and there are no typos.

Important Note

Like TracerProvider, MeterProvider uses a default no-op implementation in the API. This allows developers to instrument code without worrying about the details of how metrics will be generated. It does mean that unless we remember to set the global MeterProvider to use MeterProvider from the SDK package, any calls made to the API to generate metrics will result in no metrics being generated. This is one of the most common gotchas for folks working with OpenTelemetry.

We're almost ready to start producing metrics with an exporter, a metric reader, and a MeterProvider configured. The next step is getting a meter.

Obtaining a meter

With MeterProvider globally configured, we can use a global method to obtain a meter. As mentioned earlier, the meter will be used to create instruments, which will be used throughout the application code to record measurements. The meter receives the following arguments at creation time:

- The name of the application or library generating metrics
- An optional version identifies the version of the application or library producing the telemetry

- An optional `schema_url` to describe the data generated

Important Note

The schema URL was introduced in OpenTelemetry as part of the *OpenTelemetry Enhancement Proposal 152* (<https://github.com/open-telemetry/oteps/blob/main/text/0152-telemetry-schemas.md>). The goal of schemas is to provide OpenTelemetry instrumented applications a way to signal to external systems consuming the telemetry what the semantic versioning of the data produced will look like. Schema URL parameters are optional but recommended for all producers of telemetry: meters, tracers, and log emitters.

This information is used to identify the application or library producing the metrics. For example, application *A* making a web request via the `requests` library may contain more than one meter:

- The first meter is created by application *A* with a name identifying it with the version number matching the application.
- A second meter is created by the `requests` instrumentation library with the name `opentelemetry-instrumentation-requests` and the instrumentation library version.
- The `urllib` instrumentation library creates the third meter with the name `opentelemetry-instrumentation-urllib`, a library utilized by the `requests` library.

Having a name and a version identifier is critical in differentiating the source of the metrics. As we'll see later in the chapter, when we look at the *Views* section, this identifying information can also be used to filter out the telemetry we're not interested in. The following code uses the `get_meter_provider` global API method to access the global `MeterProvider` we configured earlier, and then calls `get_meter` with a name, `version`, and `schema_url` parameter:

metrics.py

```
from opentelemetry._metrics import get_meter_provider, set_meter_provider  
...  
...
```

```

if __name__ == "__main__":
    configure_meter_provider()
    meter = get_meter_provider().get_meter(
        name="metric-example",
        version="0.1.2",
        schema_url=" https://opentelemetry.io/schemas/1.9.0",
    )
)

```

In OpenTelemetry, instruments used to record measurements are associated with a single meter and must have unique names within the context of that meter.

Push-based and pull-based exporting

OpenTelemetry supports two methods for exporting metrics data to external systems: push-based and pull-based. A push-based exporter sends measurements from the application to a destination at a regular interval on a trigger. This trigger could be a maximum number of metrics to transfer or a schedule. The push-based method will be familiar to users of **StatsD** (<https://github.com/statsd/statsd>), where a network daemon opens a port and listens for metrics to be sent to it. Similarly, the `ConsoleSpanExporter` for the tracing signal in *Chapter 4, Distributed Tracing – Tracing Code Execution*, is a push-based exporter.

On the other hand, a pull-based exporter exposes an endpoint pulled from or scraped by an external system. Most commonly, a pull-based exporter exposes this information via a web endpoint or a local socket; this is the method popularized by Prometheus (<https://prometheus.io>). The following diagram shows the data flow comparison between a push and a pull model:

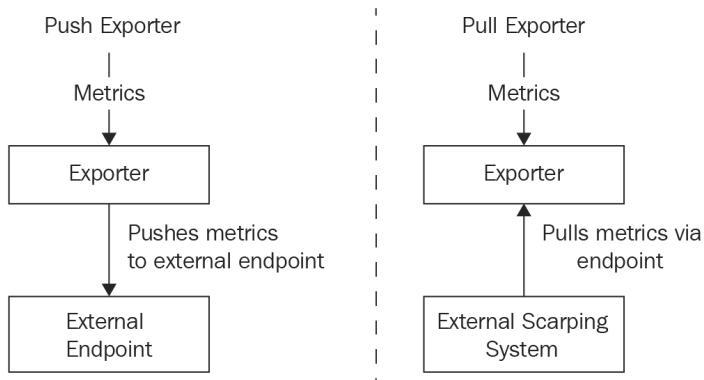


Figure 5.2 – Push versus pull-based reporting

Notice the direction of the arrow showing the interaction between the exporter and an external system. When configuring a pull-based exporter, remember that system permissions may need to be configured to allow an application to open a new port for incoming requests. One such pull-based exporter defined in the OpenTelemetry specification is the Prometheus exporter.

The pipeline configuration for a pull exporter is slightly less complex. The metric reader interface can be used as a single point to collect and expose metrics in the Prometheus format. The following code shows how to expose a Prometheus endpoint on port 8000 using the `start_http_server` method from the Prometheus client library. It then configures `PrometheusMetricReader` with a `prefix` parameter to provide a namespace for all metrics generated by our application. Finally, the code adds a call waiting for input from the user before exiting; this gives us a chance to see the exposed metrics before the application exits:

```
from opentelemetry.exporter.prometheus import
PrometheusMetricReader

from prometheus_client import start_http_server

def configure_meter_provider():
    start_http_server(port=8000, addr="localhost")
    reader = PrometheusMetricReader(prefix="MetricExample")
    provider = MeterProvider(metric_readers=[reader],
    resource=Resource.create())
    set_meter_provider(provider)

if __name__ == "__main__":
    ...
    input("Press any key to exit...")
```

If you run the application now, you can use a browser to see the Prometheus formatted data available by visiting `http://localhost:8000`. Alternatively, you can use the `curl` command to see the output data in the terminal as per the following example:

```
$ curl http://localhost:8000
# HELP python_gc_objects_collected_total Objects collected
during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 1057.0
```

```
python_gc_objects_collected_total{generation="1"} 49.0
python_gc_objects_collected_total{generation="2"} 0.0
# HELP python_gc_objects_uncollectable_total Uncollectable
object found during GC
# TYPE python_gc_objects_uncollectable_total counter
python_gc_objects_uncollectable_total{generation="0"} 0.0
python_gc_objects_uncollectable_total{generation="1"} 0.0
python_gc_objects_uncollectable_total{generation="2"} 0.0
# HELP python_gc_collections_total Number of times this
generation was collected
# TYPE python_gc_collections_total counter
python_gc_collections_total{generation="0"} 55.0
python_gc_collections_total{generation="1"} 4.0
python_gc_collections_total{generation="2"} 0.0
# HELP python_info Python platform information
# TYPE python_info gauge
python_info{implementation="CPython",major="3",minor="8",
patchlevel="0",version="3.9.0"} 1.0
```

The Prometheus client library generates the previous data; note that there are no OpenTelemetry metrics generated by our application, which makes sense since we haven't generated anything yet! We'll get to that next. We'll see in *Chapter 11, Diagnosing Problems*, how to integrate OpenTelemetry with a Prometheus backend. For the sake of simplicity, the remainder of the examples in this chapter will be using the push-based `ConsoleMetricExporter` configured earlier. If you're more familiar with Prometheus, please use this configuration instead.

Choosing the right OpenTelemetry instrument

We're now ready to generate metrics from our application. If you recall, in tracing, the **tracer** produces **spans**, which are used to create distributed traces. By contrast, the meter does not generate metrics; an **instrument** does. The meter's role is to produce instruments. OpenTelemetry offers many different instruments to record measurements. The following figure shows a list of all the instruments available:

Instrument	Synchronicity	Monotonic
Counter	Synchronous	Yes
Asynchronous Counter	Asynchronous	Yes
UpDownCounter	Synchronous	No
Asynchronous UpDownCounter	Asynchronous	No
Histogram	Synchronous	No
Asynchronous Gauge	Asynchronous	No

Figure 5.3 – OpenTelemetry instruments

Each instrument has a specific purpose, and the correct instrument depends on the following:

- The type of measurement being recorded
- Whether the measurement must be done synchronously
- Whether the values being recorded are monotonic or not

For **synchronous** instruments, a method is called on the instrument when it is time for a measurement to be recorded. For **asynchronous** instruments, a callback method is configured at the instrument's creation time.

Each instrument has a name and kind property. Additionally, a unit and a description may be specified.

Counter

A **counter** is a commonly available instrument across metric ecosystems and implementations over the years, although its definition across systems varies.

In OpenTelemetry, a counter is an increasing monotonic instrument, only supporting non-negative value increases. The following diagram shows a sample graph representing a monotonic counter:

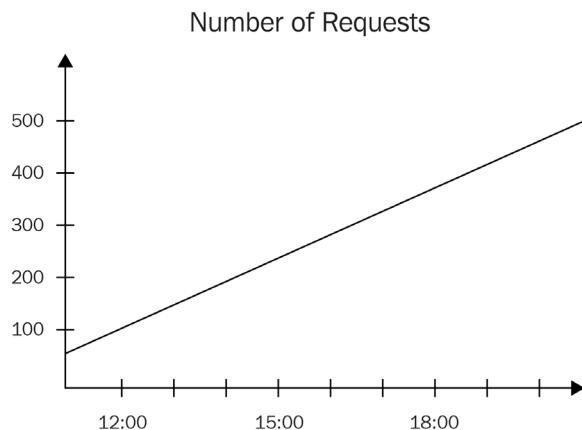


Figure 5.4 – Increasing monotonic counter graph

A counter can be used to represent the following:

- Number of requests received
- Count of orders processed
- CPU time utilization

The following code instantiates a counter to keep a tally of the number of items sold in the grocery store. The code uses the add method to increment the counter and passes the locale of the customer as an attribute:

metrics.py

```
if __name__ == "__main__":
    ...
    counter = meter.create_counter(
        "items_sold",
        unit="items",
        description="Total items sold")
```

```

    )
    counter.add(6, {"locale": "fr-FR", "country": "CA"})
    counter.add(1, {"locale": "es-ES"})

```

Running the code outputs the counter with all its attributes:

output

```

{
  "attributes": {"locale": "fr-FR", "country": "CA"},
  "description": "Total items sold", "instrumentation_info":
    "InstrumentationInfo(metric-example, 0.1.2, https://
    opentelemetry.io/schemas/1.9.0)", "name": "items_sold",
  "resource": "BoundedAttributes({'telemetry.sdk.language':
    'python', 'telemetry.sdk.name': 'opentelemetry', 'telemetry.
    sdk.version': '1.10.0', 'service.name': 'unknown_service'},
    maxlen=None)", "unit": "items", "point": {"start_time_
    unix_nano": 1646535699616146000, "time_unix_nano":
    1646535699616215000, "value": 7, "aggregation_temporality": 2,
    "is_monotonic": true}}
{
  "attributes": {"locale": "es-ES"}, "description": "Total items
  sold", "instrumentation_info": "InstrumentationInfo(metric-
  example, 0.1.2, https://opentelemetry.io/
  schemas/1.9.0)", "name": "items_sold", "resource":
  "BoundedAttributes({'telemetry.sdk.language': 'python',
  'telemetry.sdk.name': 'opentelemetry', 'telemetry.sdk.version':
  '1.10.0', 'service.name': 'unknown_service'}, maxlen=None)",
  "unit": "items", "point": {"start_time_unix_nano":
  1646535699616215001, "time_unix_nano": 1646535699616237000,
  "value": 0, "aggregation_temporality": 2, "is_monotonic":
  true}}

```

Note that the attributes themselves do not influence the value of the counter. They are only augmenting the telemetry with additional dimensions about the transaction. A monotonic instrument like the counter cannot receive a negative value. The following code tries to add a negative value:

```

if __name__ == "__main__":
    ...
    counter.add(6, {"locale": "fr-FR", "country": "CA"})
    counter.add(-1, {"unicorn": 1})

```

This code results in the following warning, which provides the developer with a helpful hint:

output

```
Add amount must be non-negative on Counter items_sold.
```

Knowing to use the right instrument can help avoid generating unexpected data. It's also good to consider adding validation to the data being passed into instruments when unsure of the data source.

Asynchronous counter

The **asynchronous counter** can be used as a counter. Its only difference is that it is used asynchronously. Asynchronous counters can represent data that is only ever-increasing, and that may be too costly to report synchronously or is more appropriate to record on set intervals. Some examples of this would be reporting the following:

- CPU time utilized by a process
- Total network bytes transferred

The following code shows us how to create an asynchronous counter using the `async_counter_callback` callback method, which will be called every time `PeriodExportingMetricReader` executes. To ensure the instrument has a chance to record a few measurements, we've added `sleep` in the code as well to pause the code before exiting:

metrics.py

```
import time
from opentelemetry._metrics.measurement import Measurement

def async_counter_callback():
    yield Measurement(10)

if __name__ == "__main__":
    ...
    # async counter
    meter.create_observable_counter(
        name="major_page_faults",
        callback=async_counter_callback,
```

```
    description="page faults requiring I/O",
    unit="fault",
)
time.sleep(10)
```

If you haven't commented out the output from the instrument, you should see the output from both counters now. The following output omits the previous example's output for brevity:

output

```
{"attributes": "", "description": "page faults requiring
I/O", "instrumentation_info": "InstrumentationInfo(metric-
example, 0.1.2, https://opentelemetry.io/
schemas/1.9.0)", "name": "major_page_faults", "resource":
"BoundedAttributes({'telemetry.sdk.language': 'python',
'telemetry.sdk.name': 'opentelemetry', 'telemetry.sdk.version':
'1.10.0', 'service.name': 'unknown_service'}, maxlen=None)",
"unit": "fault", "point": {"start_time_unix_nano":
1646538230507539000, "time_unix_nano": 1646538230507614000,
"value": 10, "aggregation_temporality": 2, "is_monotonic":
true}}
{"attributes": "", "description": "page faults requiring
I/O", "instrumentation_info": "InstrumentationInfo(metric-
example, 0.1.2, https://opentelemetry.io/
schemas/1.9.0)", "name": "major_page_faults", "resource":
"BoundedAttributes({'telemetry.sdk.language': 'python',
'telemetry.sdk.name': 'opentelemetry', 'telemetry.sdk.version':
'1.10.0', 'service.name': 'unknown_service'}, maxlen=None)",
"unit": "fault", "point": {"start_time_unix_nano":
1646538230507539000, "time_unix_nano": 1646538235507059000,
"value": 20, "aggregation_temporality": 2, "is_monotonic":
true}}
```

These counters are great for ever-increasing values, but measurements go up and down sometimes. Let's see what OpenTelemetry has in store for that.

An up/down counter

The following instrument is very similar to the counter. As you may have guessed from its name, the difference between the counter and the **up/down counter** is that the latter can record values that go up and down; it is non-monotonic. The following diagram shows us what a graph representing a non-monotonic counter may look like:

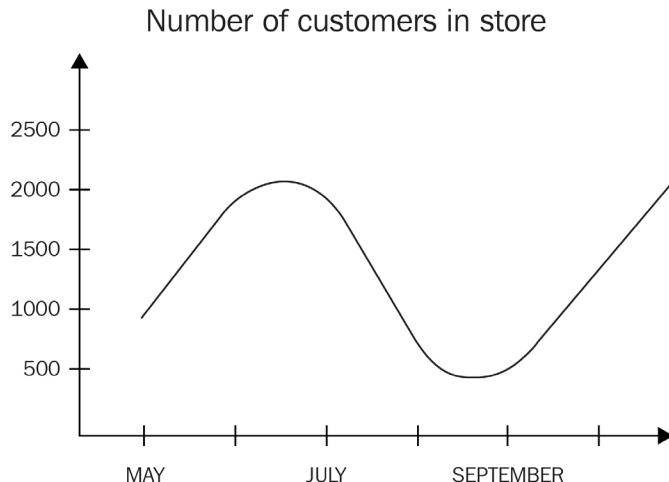


Figure 5.5 – Non-monotonic counter graph

Creating an UpDownCounter instrument is done via the `create_up_down_counter` method. Increment and decrement operations are done via the single `add` method with either positive or negative values:

metrics.py

```
if __name__ == "__main__":
    ...
    inventory_counter = meter.create_up_down_counter(
        name="inventory",
        unit="items",
        description="Number of items in inventory",
    )
    inventory_counter.add(20)
    inventory_counter.add(-5)
```

The previous example's output will be as follows:

output

```
{"attributes": "", "description": "Number of items in inventory", "instrumentation_info": "InstrumentationInfo(metric-example, 0.1.2, https://opentelemetry.io/schemas/1.9.0)", "name": "inventory", "resource": "BoundedAttributes({'telemetry.sdk.language': 'python', 'telemetry.sdk.name': 'opentelemetry', 'telemetry.sdk.version': '1.10.0', 'service.name': 'unknown_service'}, maxlen=None)", "unit": "items", "point": {"start_time_unix_nano": 1646538574503018000, "time_unix_nano": 1646538574503083000, "value": 15, "aggregation_temporality": 2, "is_monotonic": false}}
```

Note the previous example only emits a single metric. This is expected as the two recordings were aggregated into a single value for the period reported.

Asynchronous up/down counter

As you may imagine, as the counter has an asynchronous counterpart, so does UpDownCounter. The **asynchronous up/down counter** allows us to increment or decrement a value on a set interval. As you will see shortly, it is pretty similar in nature to the **asynchronous gauge**. The main difference between the two is that the asynchronous up/down counter should be used when the values being recorded are additive in nature, meaning the measurements can be added across dimensions. Some examples of metrics that could be recorded via this instrument are as follows:

- Changes in the number of customers in a store
- Net revenue for an organization across business units

The following creates an asynchronous up/down counter to keep track of the current number of customers in a store. Note that, unlike its synchronous counterpart, the value recorded in the asynchronous up/down counter is an absolute value, not a delta. As per the previous asynchronous example, an `async_updowncounter_callback` callback method does the work of reporting the measure:

metrics.py

```
def async_updowncounter_callback():
    yield Measurement(20, {"locale": "en-US"})
    yield Measurement(10, {"locale": "fr-CA"})
```

```
if __name__ == "__main__":
    ...
    upcounter_counter = meter.create_observable_up_down_
    counter(
        name="customer_in_store",
        callback=async_updowncounter_callback,
        unit="persons",
        description="Keeps a count of customers in the store"
    )
```

The output will start to look familiar based on the previous examples we've already run through:

output

```
{"attributes": {"locale": "en-US"}, "description": "Keeps
a count of customers in the store", "instrumentation_info":
"InstrumentationInfo(metric-example, 0.1.2, https://
opentelemetry.io/schemas/1.9.0)", "name": "customer_in_
store", "resource": "BoundedAttributes({'telemetry.sdk.
language': 'python', 'telemetry.sdk.name': 'opentelemetry',
'telemetry.sdk.version': '1.10.0', 'service.name': 'unknown_
service'}, maxlen=None)", "unit": "persons", "point": {"start_
time_unix_nano": 1647735390164970000, "time_unix_nano":
1647735390164986000, "value": 20, "aggregation_temporality": 2,
"is_monotonic": false}}
{"attributes": {"locale": "fr-CA"}, "description": "Keeps
a count of customers in the store", "instrumentation_info":
"InstrumentationInfo(metric-example, 0.1.2, https://
opentelemetry.io/schemas/1.9.0)", "name": "customer_in_
store", "resource": "BoundedAttributes({'telemetry.sdk.
language': 'python', 'telemetry.sdk.name': 'opentelemetry',
'telemetry.sdk.version': '1.10.0', 'service.name': 'unknown_
service'}, maxlen=None)", "unit": "persons", "point": {"start_
time_unix_nano": 1647735390164980000, "time_unix_nano":
1647735390165009000, "value": 10, "aggregation_temporality": 2,
"is_monotonic": false}}
```

Counters and up/down counters are suitable for many data types, but not all. Let's see what other instruments allow us to measure.

Histogram

A **histogram** instrument is useful when comparing the frequency distribution of values across large data sets. Histograms use buckets to group the data they represent and effectively identify outliers or anomalies. Some examples of data representable by histograms are as follows:

- Response times for requests to a service
- The height of individuals

Figure 5.6 shows a sample histogram chart representing the response time for requests. It looks like a bar chart, but it differs in that each bar represents a bucket containing a range for the values it contains. The *y* axis represents the count of elements in each bucket:

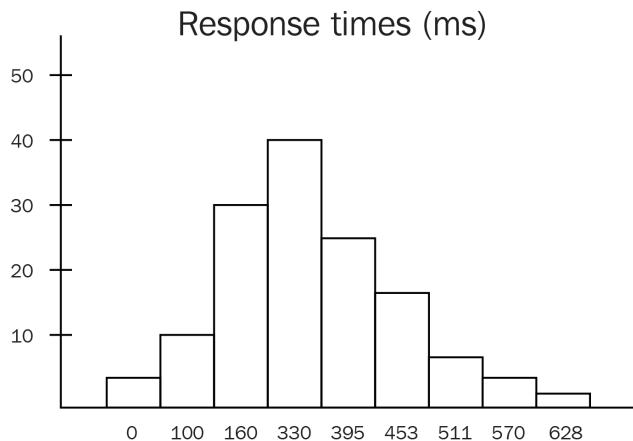


Figure 5.6 – Histogram graph

To capture information in a histogram, the buckets specified must be able to contain all the values it expects to record. For example, take a histogram containing two buckets with explicit upper bounds of 0 ms and 10 ms. Any measurement greater than 10 ms bound would be excluded from the histogram. Both Prometheus and OpenTelemetry address this by capturing any value beyond the maximum upper boundary in an additional bucket. The histograms we'll explore in this chapter all use explicit boundaries, but OpenTelemetry also provides experimental support for exponential histograms (<https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/metrics/datamodel.md#exponentialhistogram>).

Histograms can be, and are often, used to calculate percentiles. The following code creates a histogram via the `create_histogram` method. The method used to produce a metric with a histogram is named `record`:

metrics.py

```
if __name__ == "__main__":
    ...
    histogram = meter.create_histogram(
        "response_times",
        unit="ms",
        description="Response times for all requests",
    )
    histogram.record(96)
    histogram.record(9)
```

In this example, we record two measurements that fall into separate buckets. Notice how they appear in the output:

output

```
{"attributes": "", "description": "Response times for all requests", "instrumentation_info": "InstrumentationInfo(metric-example, 0.1.2, https://opentelemetry.io/schemas/1.9.0)", "name": "response_times", "resource": "BoundedAttributes({'telemetry.sdk.language': 'python', 'telemetry.sdk.name': 'opentelemetry', 'telemetry.sdk.version': '1.10.0', 'service.name': 'unknown_service'}, maxlen=None)", "unit": "ms", "point": {"start_time_unix_nano": 1646539219677439000, "time_unix_nano": 1646539219677522000, "bucket_counts": [0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0], "explicit_bounds": [0.0, 5.0, 10.0, 25.0, 50.0, 75.0, 100.0, 250.0, 500.0, 1000.0], "sum": 105, "aggregation_temporality": 2}}
```

As with the counter and up/down counter, the histogram is synchronous.

Asynchronous gauge

The last instrument defined by OpenTelemetry is the **asynchronous gauge**.

This instrument can be used to record measurements that are non-additive in nature; in other words, which wouldn't make sense to sum together. An asynchronous gauge can represent the following:

- The average memory consumption of a system
- The temperature of a data center

The following code uses Python's built-in resource module to measure the maximum resident set size (https://en.wikipedia.org/wiki/Resident_set_size). This value is set in `async_gauge_callback`, which is used as the callback for the gauge we're creating:

metrics.py

```
import resource

def async_gauge_callback():
    rss = resource.getrusage(resource.RUSAGE_SELF).ru_maxrss
    yield Measurement(rss, {})

if __name__ == "__main__":
    ...
    meter.create_observable_gauge(
        name="maxrss",
        unit="bytes",
        callback=async_gauge_callback,
        description="Max resident set size",
    )
    time.sleep(10)
```

Running the code will show us memory consumption information about our application using OpenTelemetry:

output

```
{"attributes": "", "description": "Max resident set size",  
"instrumentation_info": "InstrumentationInfo(metric-example,  
0.1.2, https://opentelemetry.io/schemas/1.9.0)", "name":  
"maxrss", "resource": "BoundedAttributes({'telemetry.sdk.  
language': 'python', 'telemetry.sdk.name': 'opentelemetry',  
'telemetry.sdk.version': '1.10.0', 'service.name': 'unknown_  
service'}, maxlen=None)", "unit": "bytes", "point": {"time_  
unix_nano": 1646539432021601000, "value": 18341888}}  
  
{"attributes": "", "description": "Max resident set size",  
"instrumentation_info": "InstrumentationInfo(metric-example,  
0.1.2, https://opentelemetry.io/schemas/1.9.0)", "name":  
"maxrss", "resource": "BoundedAttributes({'telemetry.sdk.  
language': 'python', 'telemetry.sdk.name': 'opentelemetry',  
'telemetry.sdk.version': '1.10.0', 'service.name': 'unknown_  
service'}, maxlen=None)", "unit": "bytes", "point": {"time_  
unix_nano": 1646539437018742000, "value": 19558400}}
```

Excellent, we now know about the instruments and have started generating a steady metrics stream. The last topic about instruments to be covered is duplicate instruments.

Duplicate instruments

Duplicate instrument registration conflicts arise if more than one instrument is created within a single meter with the same name. This can potentially produce semantic errors in the data, as many telemetry backends uniquely identify metrics via their names. Conflicting instruments may be intentional when two separate code paths need to report the same metric, or, when multiple developers want to record different metrics but accidentally use the same name; naming things is hard. There are a few ways the OpenTelemetry SDK handles conflicting instruments:

- If the conflicting instruments are identical, the values recorded by these instruments are aggregated. The data generated appears as though a single instrument produced them.
- If the instruments are not identical, but the conflict can be resolved via View configuration, the user will not be warned. As we'll see next, views provide a mechanism to produce unique metric streams, differentiating the instruments.

- If the instruments are not identical and their conflicts are not resolved via views, a warning is emitted, and their data is generated without modification.

Individual meters act as a namespace, meaning two meters can separately create identical instruments without any issues. Using a unique namespace for each meter ensures that application developers can create instruments that make sense for their applications without running the risk of interfering with other metrics generated by underlying libraries. This will also make searching for metrics easier once exported outside the application. Let's see how we can shape the metrics stream to fit our needs with views.

Customizing metric outputs with views

Some applications may produce more metrics than an application developer is interested in. You may have noticed this with the example code for instruments; as we added more examples, it became difficult to find the metrics we were interested in. Recall the example mentioned earlier in this chapter: application *A* represents a client library making web requests that could produce metrics via three different meters. If each of those meters keeps a request counter, duplicate data is highly likely to be generated. Duplicated data may not be a problem on a small scale, but when scaling services up to handling thousands and millions of requests, unnecessary metrics can become quite expensive. Thankfully, **views** provide a way for users of OpenTelemetry to configure the SDK only to generate the metrics they want. In addition to providing a mechanism to filter metrics, views can also configure aggregation or be used to add a new dimension to metrics.

Filtering

The first aspect of interest is the ability to customize which metrics will be processed. To select instruments, the following criteria can be applied to a view:

- `instrument_name`: The name of the instrument
- `instrument_type`: The type of the instrument
- `meter_name`: The name of the meter
- `meter_version`: The version of the meter
- `meter_schema`: The schema URL of the meter

The SDK provides a default view as a catch-all for any instruments not matched by configured views.

Important note

The code in this chapter uses version 1.10.0 which supports the parameter `enable_default_view` to modify to disable the default view. This has changed in version 1.11.0 with the following change: <https://github.com/open-telemetry/opentelemetry-python/pull/2547>. If you are using a newer version, you will need to configure a wildcard view with a `DropAggregation`, refer to the official documentation (<https://opentelemetry-python.readthedocs.io/en/latest/sdk/metrics.html>) for more information.

The following code selects the `inventory` instrument we created in an earlier example. Views are added to the `MeterProvider` as an argument to the constructor.

Another argument is added disabling the default view:

metrics.py

```
from opentelemetry.sdk._metrics.view import View
def configure_meter_provider():
    exporter = ConsoleMetricExporter()
    reader = PeriodicExportingMetricReader(exporter, export_
interval_millis=5000)
    view = View(instrument_name="inventory")
    provider = MeterProvider(
        metric_readers=[reader],
        resource=Resource.create(),
        views=[view],
        enable_default_view=False,
    )
```

The resulting output shows a metric stream limited to a single instrument:

output

```
{"attributes": {"locale": "fr-FR", "country": "CA"},  
"description": "total items sold", "instrumentation_info":  
"InstrumentationInfo(metric-example, 0.1.2, https://  
opentelemetry.io/schemas/1.9.0)", "name": "sold", "resource":
```

```

"BoundedAttributes({'telemetry.sdk.language': 'python',
'telemetry.sdk.name': 'opentelemetry', 'telemetry.sdk.version':
'1.10.0', 'service.name': 'unknown_service'}, maxlen=None)",
"unit": "items", "point": {"start_time_unix_nano":
1647800250023129000, "time_unix_nano": 1647800250023292000,
"value": 6, "aggregation_temporality": 2, "is_monotonic":
true}}}

{"attributes": {"locale": "es-ES"}, "description": "total items
sold", "instrumentation_info": "InstrumentationInfo(metric-
example, 0.1.2, https://opentelemetry.io/schemas/1.9.0)",
"name": "sold", "resource": "BoundedAttributes({'telemetry.
sdk.language': 'python', 'telemetry.sdk.name': 'opentelemetry',
'telemetry.sdk.version': '1.10.0', 'service.name': 'unknown_
service'}, maxlen=None)", "unit": "items", "point": {"start_
time_unix_nano": 1647800250023138000, "time_unix_nano":
1647800250023312000, "value": 1, "aggregation_temporality": 2,
"is_monotonic": true}}

```

The views parameter accepts a list, making adding multiple views trivial. This provides a great deal of flexibility and control for users. An instrument must match all arguments passed into the View constructor. Let's update the previous example and see what happens when we try to create a view by selecting an instrument of the Counter type with the name `inventory`:

metrics.py

```

from opentelemetry._metrics.instrument import Counter

def configure_meter_provider():
    exporter = ConsoleMetricExporter()
    reader = PeriodicExportingMetricReader(exporter, export_
    interval_millis=5000)
    view = View(instrument_name="inventory", instrument_
    type=Counter)
    provider = MeterProvider(
        metric_readers=[reader],
        resource=Resource.create(),
        views=[view],
        enable_default_view=False,
    )

```

As you may already suspect, these criteria will not match any instruments, and no data will be produced by running the code.

Important Note

All criteria specified when selecting instruments are optional. However, if no optional argument is specified, the code will raise an exception as per the OpenTelemetry specification.

Using views to filter instruments based on instrument or meter identification is a great way to reduce the noise and cost of generating too many metrics.

Dimensions

In addition to selecting instruments, it's also possible to configure a view to only report specific dimensions. A dimension in this context is an attribute associated with the metric. For example, a customer counter may record information about customers as per *Figure 5.7*. Each attribute associated with the counter, such as the country the customer is visiting from or the locale their browser is set to, offers another dimension to the metric recorded during their visit:

Customer	Country	Locale
1	Canada	en-US
1	France	fr-FR
1	Canada	fr-FR

Figure 5.7 – Additional dimensions for a counter

Dimensions can be used to aggregate data in meaningful ways; continuing with the previous table, we can obtain the following information:

- Three customers visited our store.
- Two customers visited from Canada and one from France.
- Two had browsers configured to French (fr-FR), and one to English (en-US).

Views allow us to customize the output from our metrics stream. Using the `attribute_keys` argument, we specify the dimensions we want to see in a particular view. The following configures a view to match the Counter instruments and to discard any attributes other than `locale`:

metrics.py

```
def configure_meter_provider():
    exporter = ConsoleMetricExporter()
    reader = PeriodicExportingMetricReader(exporter, export_
    interval_millis=5000)
    view = View(instrument_type=Counter, attribute_
    keys=["locale"])
    ...

```

You may remember that in the code we wrote earlier when configuring instruments, the `items_sold` counter generated two metrics. The first contained `country` and `locale` attributes; the second contained the `locale` attribute. The configuration in this view will produce a metric stream discarding all attributes not specified via `attribute_keys`:

output

```
{"attributes": {"locale": "fr-FR"}, "description": "Total items
sold", ...}
{"attributes": {"locale": "es-ES"}, "description": "Total items
sold", ...}
```

Note that when using `attribute_keys`, all metrics not containing the specified attributes will be aggregated. This is because by removing the attributes, the view effectively transforms the metrics, as per the following table:

Counter operation	Transformed via attribute keys
<code>add(1,{"locale":"fr-FR"})</code>	<code>add(1,{"locale":"fr-FR"})</code>
<code>add(1,{"country":"CA"})</code>	<code>add(1,{})</code>
<code>add(1,{"locale":"en-US", "country":"CA"})</code>	<code>add(1,{"locale":"en-US"})</code>
<code>add(1,{})</code>	<code>add(1,{})</code>

Figure 5.8 – Effect of attribute keys on counter operations

An example of where this may be useful is separating requests containing errors from those that do not, or grouping requests by status code.

In addition to customizing the metric stream attributes, views can also alter their name or description. The following renames the metric generated and updates its description. Additionally, it removes all attributes from the metric stream:

metrics.py

```
def configure_meter_provider():
    exporter = ConsoleMetricExporter()
    reader = PeriodicExportingMetricReader(exporter, export_
interval_millis=5000)
    view = View(
        instrument_type=Counter,
        attribute_keys=[],
        name="sold",
        description="total items sold",
    )
    ...

```

The output now shows us a single aggregated metric that is more meaningful to us:

output

```
{"attributes": "", "description": "total items sold",
"instrumentation_info": "InstrumentationInfo(metric-example,
0.1.2, https://opentelemetry.io/schemas/1.9.0)", "name":
"sold", "resource": "BoundedAttributes({'telemetry.sdk.
language': 'python', 'telemetry.sdk.name': 'opentelemetry',
'telemetry.sdk.version': '1.10.0', 'service.name': 'unknown_
service'}, maxlen=None)", "unit": "items", "point": {"start_
time_unix_nano": 1646593079208078000, "time_unix_nano":
1646593079208238000, "value": 7, "aggregation_temporality": 2,
"is_monotonic": true}}
```

Customizing views allow us to focus further on the output of the metrics generated. Let's see how we can combine the metrics with aggregators.

Aggregation

The last configuration of views we will investigate is aggregation. The `aggregation` option gives the view the ability to change the default aggregation used by an instrument to one of the following methods:

- `SumAggregation`: Add the instrument's measurements and set the current value as the sum. The monotonicity and temporality for the sum are derived from the instrument.
- `LastValueAggregation`: Record the last measurement and its timestamp as the current value of this view.
- `ExplicitBucketHistogramAggregation`: Use a histogram where the boundaries can be set via configuration. Additional options for this aggregation are `boundaries` for the buckets of the histogram and `record_min_max` to record the minimum and maximum values.

The following table, *Figure 5.9*, shows us the default aggregation for each instrument:

Instrument	Default aggregation
Counter	SumAggregation
Asynchronous Counter	SumAggregation
UpDownCounter	SumAggregation
Asynchronous UpDownCounter	SumAggregation
Histogram	ExplicitBucketHistogramAggregation
Asynchronous Gauge	LastValueAggregation

Figure 5.9 – Default aggregation per instrument

Aggregating data in the SDK allows us to reduce the number of data points transmitted. However, this means the data available at query time is less granular, limiting the user's ability to query it. Keeping this in mind, let's look at configuring the aggregation for one of our counter instruments to see how this works. The following code updates the view configured earlier to use `LastValueAggregation` instead of the `SumAggregation` default:

metrics.py

```
from opentelemetry.sdk._metrics.aggregation import
LastValueAggregation
```

```
def configure_meter_provider():
    exporter = ConsoleMetricExporter()
    reader = PeriodicExportingMetricReader(exporter, export_
interval_millis=5000)
    view = View(
        instrument_type=Counter,
        attribute_keys=[],
        name="sold",
        description="total items sold",
        aggregation=LastValueAggregation(),
    )
```

You'll notice in the output now that instead of reporting the sum of all measurements (7) for the counter, only the last value (1) is produced:

output

```
{"attributes": "", "description": "total items sold",
"instrumentation_info": "InstrumentationInfo(metric-example,
0.1.2, https://opentelemetry.io/schemas/1.9.0)", "name":
"sold", "resource": "BoundedAttributes({'telemetry.sdk.
language': 'python', 'telemetry.sdk.name': 'opentelemetry',
'telemetry.sdk.version': '1.10.0', 'service.name': 'unknown_
service'}, maxlen=None)", "unit": "items", "point": {"time_
unix_nano": 1646594506458381000, "value": 1}}
```

Although it's essential to have the ability to configure aggregation, the default aggregation may well serve your purpose most of the time.

Important Note

As mentioned earlier, sum aggregation derives the temporality of the sum reported from its instrument. This temporality can be either *cumulative* or *delta*. This determines whether the reported metrics are to be interpreted as always starting at the same time, therefore, reporting a cumulative metric, or if the metrics reported represent a moving start time, and the reported values contain the delta from the previous report. For more information about temporality, refer to the OpenTelemetry specification found at <https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/metrics/datamodel.md#temporality>.

The grocery store

It's time to go back to the example application from *Chapter 4, Distributed Tracing – Tracing Code Execution*, to get some practical experience of all the knowledge we've gained so far. Let's start by adding a method to retrieve a meter that will resemble `configure_tracer` from the previous chapter. This method will be named `configure_meter` and will contain the configuration code from an example earlier in this chapter. One main difference is the addition of a resource that uses `LocalMachineResourceDetector`, as we already defined in this module. Add the following code to the `common.py` module:

common.py

```
from opentelemetry._metrics import get_meter_provider, set_meter_provider

from opentelemetry.sdk._metrics import MeterProvider
from opentelemetry.sdk._metrics.export import (
    ConsoleMetricExporter,
    PeriodicExportingMetricReader,
)

def configure_meter(name, version):
    exporter = ConsoleMetricExporter()
    reader = PeriodicExportingMetricReader(exporter, export_interval_millis=5000)
    local_resource = LocalMachineResourceDetector().detect()
    resource = local_resource.merge(
        Resource.create(
            {
                ResourceAttributes.SERVICE_NAME: name,
                ResourceAttributes.SERVICE_VERSION: version,
            }
        )
    )
    provider = MeterProvider(metric_readers=[reader],
    resource=resource)
    set_meter_provider(provider)
    schema_url = "https://opentelemetry.io/schemas/1.9.0"
    return get_meter_provider().get_meter(
        name=name,
```

```
    version=version,  
    schema_url=schema_url,  
)  
)
```

Now, update `shopper.py` to call this method and set the return value to a global variable named `meter` that we'll use throughout the application:

shopper.py

```
from common import configure_tracer, configure_meter  
  
tracer = configure_tracer("shopper", "0.1.2")  
meter = configure_meter("shopper", "0.1.2")
```

We will be adding this line to `grocery_store.py` and `legacy_inventory.py` in the following examples, but you may choose to do so now. Now, to start the applications and ensure the code works as it should, launch the three applications in separate terminals using the following commands in the order presented:

```
$ python legacy_inventory.py  
$ python grocery_store.py  
$ python shopper.py
```

The execution of `shopper.py` should return right away. If no errors were printed out because of running those commands, we're off to a good start and are getting closer to adding metrics to our applications!

Number of requests

When considering what metrics are essential to get insights about an application, it can be overwhelming to think of all the things we could measure. A good place is to start is with the *golden signals* as documented in the Google **Site Reliability Engineering (SRE)** book, https://sre.google/sre-book/monitoring-distributed-systems/#xref_monitoring_golden-signals. Measuring the traffic to our application is an easy place to start by counting the number of requests it receives. It can help answer questions such as the following:

- What is the traffic pattern for our application?
- Is the application capable of handling the traffic we expected?

- How successful is the application?

In future chapters, we'll investigate how this metric can be used to determine if the application should be scaled automatically. A metric such as the total number of requests a service can handle is likely a number that would be revealed during benchmarking.

The following code calls `configure_meter` and creates a counter via the `create_counter` method to keep track of the incoming requests to the server application. The `request_counter` value is incremented before the request is processed:

grocery_store.py

```
from common import configure_meter, configure_tracer, set_span_attributes_from_flask

tracer = configure_tracer("grocery-store", "0.1.2")
meter = configure_meter("grocery-store", "0.1.2")
request_counter = meter.create_counter(
    name="requests",
    unit="request",
    description="Total number of requests",
)

@app.before_request
def before_request_func():
    token = context.attach(extract(request.headers))
    request_counter.add(1)
    request.environ["context_token"] = token
```

The updated grocery store code should reload automatically, but restart the grocery store application if it does not. Once the updated code is running, make the following three requests to the store by using curl:

```
$ curl localhost:5000
$ curl localhost:5000/products
$ curl localhost:5000/none-existent-url
```

This should give us output similar to the abbreviated output. Pay attention to the increasing value field, which increases by one with each visit:

```
127.0.0.1 - - [06/Mar/2022 11:44:41] "GET / HTTP/1.1" 200 -
{"attributes": "", "description": "Total number of requests",
... "point": {"start_time_unix_nano": 1646595826470792000,
"time_unix_nano": 1646595833190445000, "value": 1,
"aggregation_temporality": 2, "is_monotonic": true}}
127.0.0.1 - - [06/Mar/2022 11:44:46] "GET /products HTTP/1.1"
200 -
{"attributes": "", "description": "Total number of requests",
... "point": {"start_time_unix_nano": 1646595826470792000,
"time_unix_nano": 1646595883232762000, "value": 2,
"aggregation_temporality": 2, "is_monotonic": true}}
127.0.0.1 - - [06/Mar/2022 11:44:47] "GET /none-existent-url
HTTP/1.1" 404 -
{"attributes": "", "description": "Total number of requests",
... "point": {"start_time_unix_nano": 1646595826470792000,
"time_unix_nano": 1646595888236270000, "value": 3,
"aggregation_temporality": 2, "is_monotonic": true}}
```

In addition to counting the total number of requests, it's helpful to have a way to track the different response codes. In the previous example, if you look at the output, you'll notice the last response's status code indicated a 404 error, which would be helpful to identify differently from other responses.

Keeping a separate counter would allow us to calculate an error rate that could infer the service's health. Alternatively, using attributes can accomplish this, as well. The following moves the code to increment the counter where the response status code is available. This code is then recorded as an attribute on the metric:

grocery_store.py

```
@app.before_request
def before_request_func():
    token = context.attach(extract(request.headers))
    request.environ["context_token"] = token

@app.after_request
def after_request_func(response):
    request_counter.add(1, {"code": response.status_code})
```

```
return response
```

To trigger the new code, use the following curl command:

```
$ curl localhost:5000/none-existent-url
```

The result includes the status code attribute:

output

```
{"attributes": {"code": 404}, "description": "Total number of requests", "instrumentation_info": "InstrumentationInfo(grocery-store, 0.1.2, https://opentelemetry.io/schemas/1.9.0)", "name": "requests", "resource": "BoundedAttributes({'telemetry.sdk.language': 'python', 'telemetry.sdk.name': 'opentelemetry', 'telemetry.sdk.version': '1.10.0', 'net.host.name': 'host', 'net.host.ip': '127.0.0.1', 'service.name': 'grocery-store', 'service.version': '0.1.2'}, maxlen=None)", "unit": "request", "point": {"start_time_unix_nano": 1646598200103414000, "time_unix_nano": 1646598203067451000, "value": 1, "aggregation_temporality": 2, "is_monotonic": true}}
```

Send a few more requests through to obtain different status codes. You can start seeing how this information can calculate error rates. The name given to metrics is significant.

Important Note

It's not possible to generate telemetry where there is no instrumentation. However, it *is* possible to filter out undesired telemetry using the configuration in the SDK and the OpenTelemetry collector. Remember this when instrumenting code. We'll visit how the collector can filter telemetry in *Chapter 8, OpenTelemetry Collector*, and *Chapter 9, Deploying the Collector*.

The data has shown us how to use a counter to produce meaningful data enriched with attributes. The value of this data will become even more apparent once we look at analysis tools in *Chapter 10, Configuring Backends*.

Request duration

The next metric to produce is request duration. The goal of understanding the request duration across a system is to be able to answer questions such as the following:

- How long did the request take?
- How much time did each service add to the total duration of the request?
- What is the experience for users?

Request duration is an interesting metric to understand the health of a service and can often be the symptom of an underlying issue. Collecting the duration is best done via a histogram, which can provide us with the organization and visualization necessary to understand the distribution across many requests. In the following example, we are interested in measuring the duration of operations within each service. We are also interested in capturing the duration of upstream requests and the network latency costs across each service in our distributed application. *Figure 5.10* shows how this will be measured:

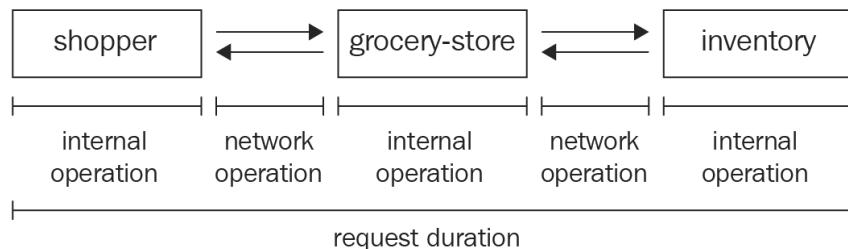


Figure 5.10 – Measuring request duration

We can use the different measurements across the entire request to understand where time is spent. This could help differentiate network issues from application issues. For example, if a request from `shopper.py` to `grocery_store.py` takes 100 ms, but the operation within `grocery_store.py` takes less than 1 ms, we know that the additional 99 ms were spent outside the application code.

Important Note

When a network is involved, unexpected latency can always exist. This common fallacy of cloud-native applications must be accounted for when designing applications. Investment in network engineering and deploying applications within closer physical proximity significantly reduces latency.

In the following example, the `upstream_duration_histo` histogram is configured to record the duration of requests from `shopper.py` to `grocery_store.py`. An additional histogram, `total_duration_histo`, is created to capture the duration of the entire operation within the `shopper` application. The period is calculated using the `time_ns` method from the `time` library, which returns the current time in nanoseconds, which we convert to milliseconds:

shopper.py

```
import time

total_duration_histo = meter.create_histogram(
    name="duration",
    description="request duration",
    unit="ms",
)

upstream_duration_histo = meter.create_histogram(
    name="upstream_request_duration",
    description="duration of upstream requests",
    unit="ms",
)

def browse():
    ...
    start = time.time_ns()
    resp = requests.get(url, headers=headers)
    duration = (time.time_ns() - start)/1e6
    upstream_duration_histo.record(duration)
    ...

def visit_store():
    start = time.time_ns()
    browse()
    duration = (time.time_ns() - start)/1e6
    total_duration_histo.record(duration)
```

The next step is to configure a histogram in `grocery_store.py` to record upstream requests and operation durations. For brevity, I will omit the instantiation of the two histograms to the following code, as the code is identical to the previous example. The following uses methods decorated with Flask's `before_request` and `after_request` to calculate the beginning and end of the entire operation. We also need to calculate the upstream request that occurs in the `products` method:

grocery_store.py

```
@app.before_request
def before_request_func():
    token = context.attach(extract(request.headers))
    request_counter.add(1, {})
    request.environ["context_token"] = token
    request.environ["start_time"] = time.time_ns()

@app.after_request
def after_request_func(response):
    request_counter.add(1, {"code": response.status_code})
    duration = (time.time_ns() - request.environ["start_time"])
    / 1e6
    total_duration_histo.record(duration)
    return response

@app.route("/products")
@tracer.start_as_current_span("/products", kind=SpanKind.SERVER)
def products():
    ...
    inject(headers)
    start = time.time_ns()
    resp = requests.get(url, headers=headers)
    duration = (time.time_ns() - start) / 1e6
    upstream_duration_histo.record(duration)
```

Lastly, for this example, let's add duration calculation for `legacy_inventory.py`. The code will be more straightforward since this service has no upstream requests yet, thus, we'll only need to define a single histogram:

legacy_inventory.py

```
from flask import request
import time

total_duration_histo = meter.create_histogram(
    name="duration",
    description="request duration",
    unit="ms",
)

@app.before_request
def before_request_func():
    token = context.attach(extract(request.headers))
    request.environ["start_time"] = time.time_ns()

@app.after_request
def after_request_func(response):
    duration = (time.time_ns() - request.environ["start_time"])
    / 1e6
    total_duration_histo.record(duration)
    return response
```

Now that we have all these histograms in place, we can finally look at the duration of our requests. The following output combines the output from all three applications to give us a complete picture of the time spent across the system. Pay close attention to the `sum` value recorded for each histogram. As we're only sending one request through, the sum equates the value for that single request:

output

```
{"attributes": "", "description": "duration of
upstream requests", "instrumentation_info":
"InstrumentationInfo(shopper, 0.1.2, https://opentelemetry.io/
schemas/1.9.0)", "name": "upstream_request_duration", "unit":
```

```

"ms", "point": {"start_time_unix_nano": 1646626129420576000,
"time_unix_nano": 1646626129420946000, "bucket_counts": [0, 0,
0, 0, 0, 0, 0, 0, 1], "explicit_bounds": [0.0, 5.0, 10.0,
25.0, 50.0, 75.0, 100.0, 250.0, 500.0, 1000.0], "sum": 18.981,
"aggregation_temporality": 2}},

{"attributes": "", "description": "request duration",
"instrumentation_info": "InstrumentationInfo(shopper,
0.1.2, https://opentelemetry.io/schemas/1.9.0)", "name":
"duration", "unit": "ms", "point": {"start_time_unix_nano": 1646626129420775000, "time_unix_nano": 1646626129420980000,
"bucket_counts": [0, 0, 0, 1, 0, 0, 0, 0, 0, 0], "explicit_
bounds": [0.0, 5.0, 10.0, 25.0, 50.0, 75.0, 100.0, 250.0,
500.0, 1000.0], "sum": 19.354, "aggregation_temporality": 2}},

{"attributes": "", "description": "request duration",
"instrumentation_info": "InstrumentationInfo(grocery-store,
0.1.2, https://opentelemetry.io/schemas/1.9.0)", "name":
"duration", "unit": "ms", "point": {"start_time_unix_nano": 1646626129419257000, "time_unix_nano": 1646626133006672000,
"bucket_counts": [0, 0, 0, 1, 0, 0, 0, 0, 0, 0], "explicit_
bounds": [0.0, 5.0, 10.0, 25.0, 50.0, 75.0, 100.0, 250.0,
500.0, 1000.0], "sum": 10.852, "aggregation_temporality": 2}},

{"attributes": "", "description": "duration of
upstream requests", "instrumentation_info":
"InstrumentationInfo(grocery-store, 0.1.2, https://
opentelemetry.io/schemas/1.9.0)", "name": "upstream_request_
duration", "unit": "ms", "point": {"start_time_unix_nano": 1646626129419136000, "time_unix_nano": 1646626135619575000,
"bucket_counts": [0, 0, 0, 1, 0, 0, 0, 0, 0, 0], "explicit_
bounds": [0.0, 5.0, 10.0, 25.0, 50.0, 75.0, 100.0, 250.0,
500.0, 1000.0], "sum": 10.36, "aggregation_temporality": 2}},

{"attributes": "", "description": "request duration",
"instrumentation_info": "InstrumentationInfo(legacy-inventory,
0.9.1, https://opentelemetry.io/schemas/1.9.0)", "name":
"duration", "unit": "ms", "point": {"start_time_unix_nano": 1646626129417730000, "time_unix_nano": 1646626134436096000,
"bucket_counts": [0, 1, 0, 0, 0, 0, 0, 0, 0, 0], "explicit_
bounds": [0.0, 5.0, 10.0, 25.0, 50.0, 75.0, 100.0, 250.0,
500.0, 1000.0], "sum": 0.494, "aggregation_temporality": 2}}

```

The difference in `upstream_request_duration` and `duration` sums for each application gives us the duration of the operation within each application. Looking closely at the data produced, we can see a significant portion of the request, 93% in this case, is spent communicating between applications.

If you're looking at this and wondering, *Couldn't distributed tracing calculate the duration of the request and latency instead?*, you're right. This type of information is also available via distributed tracing, so long as all the operations along the way are instrumented.

Concurrent requests

Another critical metric is the concurrent number of requests an application is processing at any given time. This helps answer the following:

- Is the application a bottleneck for a system?
- Can the application handle a surge in requests?

Normally, this value is obtained by calculating a rate of the number of requests per second via the counter added earlier. However, since we need practice with instruments and have yet to send our data to a backend that allows for analysis, we'll record it manually.

It's possible to use several instruments to capture this. For the sake of this example, we will use an up/down counter, but we could have also used a gauge as well. We will increment the up/down counter every time a new request begins and decrement it after each request:

grocery_store.py

```
concurrent_counter = meter.create_up_down_counter(  
    name="concurrent_requests",  
    unit="request",  
    description="Total number of concurrent requests",  
)  
  
@app.before_request  
def before_request_func():  
    ...  
    concurrent_counter.add(1)  
  
@app.after_request  
def after_request_func(err):  
    ...  
    concurrent_counter.add(-1)
```

To ensure we can see multiple users connected simultaneously, we will use a different tool than `shopper.py`, which we've used for this far. The `hey` load generation program allows us to generate hundreds of requests in parallel, enabling us to see the up/down counter in action. Run the program now with the following command to generate 300 requests with a maximum concurrency of 10:

```
$ hey -n 3000 -c 10 http://localhost:5000/products
```

That command should have created enough parallel connections. Let's look at the metrics generated; we should expect to see the recorded value going up as the number of concurrent requests increases, and then going back down:

output

```
{"attributes": "", "description": "Total number
of concurrent requests", "instrumentation_info":
"InstrumentationInfo(grocery-store, 0.1.2, https://
opentelemetry.io/schemas/1.9.0)", "name": "concurrent_
requests", "unit": "request", "point": {"start_time_unix_nano": 1646627738799214000, "time_unix_nano": 1646627769865503000,
"value": 10, "aggregation_temporality": 2, "is_monotonic": false}}}

{"attributes": "", "description": "Total number
of concurrent requests", "instrumentation_info":
"InstrumentationInfo(grocery-store, 0.1.2, https://
opentelemetry.io/schemas/1.9.0)", "name": "concurrent_
requests", "unit": "request", "point": {"start_time_unix_nano": 1646627738799214000, "time_unix_nano": 1646627774867317000,
"value": 0, "aggregation_temporality": 2, "is_monotonic": false}}
```

We will come back to using this tool later, but it's worth keeping around if you want to test the performance of your applications. We will be looking at some additional tools to generate load in *Chapter 11, Diagnosing Problems*. Try pushing the load higher to see if you can cause the application to fail altogether by increasing the number of requests or concurrency.

Resource consumption

The following metrics we will capture from our applications are runtime performance metrics. Capturing the performance metrics of an application can help us answer questions such as the following:

- How many resources does my application need?
- What budget will I need to run this service for the next 6 months?

This often helps guide decisions of what resources will be needed as the business needs change. Quite often, application performance metrics, such as memory, CPU, and network consumption, indicate where time could be spent reducing the cost of an application.

Important Note

In the following example, we will focus specifically on runtime application metrics. These do not include system-level metrics. There is an essential distinction between the two. Runtime application metrics should be recorded by each application individually. On the other hand, system-level metrics should only be recorded once for the entire system. Reporting system-level metrics from multiple applications running on the same system is problematic. This will cause system performance metrics to be duplicated, which will require de-duplication either at transport or at analysis time. Another problem is that querying the system for metrics is expensive, and doing so multiple times places an unnecessary burden on the system.

When looking for runtime metrics, there are many metrics to choose from. Let's record the memory consumption that we will measure using an asynchronous gauge. One of the tools available to provide a way to measure memory statistics in Python comes with the standard library. The `resource` package (<https://docs.python.org/3/library/resource.html>) provides usage information about our process. Additional third-party libraries are available, such as `psutil` (<https://psutil.readthedocs.io/>), which provides even more information about the resource utilization of your process. It's an excellent package for collecting information about CPU, disk, and network usage.

As the implementation for capturing this metric will be the same across all the applications in the system, the code for the callback will be placed in `common.py`. The following creates a `record_max_rss_callback` method to record the maximum resident set size for the application. It also defines a convenience method called `start_recording_memory_metrics`, which creates the asynchronous gauge. Add these methods to `common.py` now:

common.py

```
import resource
from opentelemetry._metrics.measurement import Measurement

def record_max_rss_callback():
    yield Measurement(resource.getrusage(resource.RUSAGE_SELF).ru_maxrss)

def start_recording_memory_metrics(meter):
    meter.create_observable_gauge(
        callback=record_max_rss_callback,
        name="maxrss",
        unit="bytes",
        description="Max resident set size",
    )
```

Next, add a call to `start_recording_memory_metrics` in each application in our system. Add the following code to `shopper.py`, `legacy_inventory.py`, and `grocery_store.py`:

shopper.py

```
from common import start_recording_memory_metrics

if __name__ == "__main__":
    start_recording_memory_metrics(meter)
```

After adding this code to each application and ensuring they have been reloaded, each should start reporting the following values:

output

```
{"attributes": "", "description": "Max resident set size",
"instrumentation_info": "InstrumentationInfo(legacy-inventory,
0.9.1, https://opentelemetry.io/schemas/1.9.0)", "name":
"maxrss", "resource": "BoundedAttributes({'telemetry.sdk.
language': 'python', 'telemetry.sdk.name': 'opentelemetry',
'telemetry.sdk.version': '1.10.0', 'net.host.name': 'host',
'net.host.ip': '10.0.0.141', 'service.name': 'legacy-
inventory', 'service.version': '0.9.1'}, maxlen=None)", "unit":
"bytes", "point": {"time_unix_nano": 1646637404789912000,
"value": 33083392}}
```

And just like that, we have memory telemetry about our applications. I urge you to add additional usage metrics to the application and look at the `psutil` library mentioned earlier to expand the telemetry of your services. The metrics we added to the grocery store are by no means exhaustive. Instrumenting the code and gaining familiarity with instruments gives us a starting point from which to work.

Summary

We've covered much ground in this chapter about the metrics signal. We started by familiarizing ourselves with the different components and terminology of the metrics pipeline and how to configure them. We then looked at all the ins and outs of the individual instruments available to record measurements and used each one to record sample metrics.

Using views, we learned to aggregate, filter, and customize the metric streams being emitted by our application to fit our specific needs. This will be handy when we start leveraging instrumentation libraries. Finally, we returned to the grocery store to get hands-on experience with instrumenting an existing application and collecting real-world metrics.

Metrics is a deep topic that goes well beyond what has been covered in this chapter, but hopefully, what you've learned thus far is enough to start considering how OpenTelemetry can be used in your code. The next chapter will look at the third and final signal we will cover in this book – logging.

6

Logging – Capturing Events

Metrics and traces go a long way in helping understand the behaviors and intricacies of cloud-native applications. Sometimes though, it's useful to log additional information that can be used at debug time. Logging gives us the ability to record information in a way that is perhaps more flexible and freeform than either tracing or metrics. That flexibility is both wonderful and terrible. It allows logs to be customized to fit whatever need arises using natural language, which often, but not always, makes it easier to interpret by the reader. But the flexibility is often abused, resulting in a mess of logs that are hard to search through and even harder to aggregate in any meaningful way. This chapter will take a look at how **OpenTelemetry** tackles the challenges of logging and how it can be used to improve the telemetry generated by an application. We will cover the following topics:

- Configuring OpenTelemetry to export logs
- Producing logs via the OpenTelemetry API and a standard logging library
- The logging signal in practice within the context of the grocery store application

Along the way, we will learn about standard logging in Python as well as logging with Flask, giving us a chance to use an instrumentation library as well. But first, let's ensure we have everything we need set up.

Technical requirements

If you've already completed *Chapter 4, Distributed Tracing*, or *Chapter 5, Metrics - Recording Measurements*, the setup here will be quite familiar. Ensure the version of Python in your environment is at least Python 3.6 by running the following commands:

```
$ python --version  
$ python3 --version
```

This chapter will rely on the OpenTelemetry API and SDK packages that are installable via pip with the following command. The examples in this chapter are using the version 1.9.0 opentelemetry-api and opentelemetry-sdk packages:

```
$ pip install opentelemetry-api \  
opentelemetry-sdk \  
opentelemetry-propagator-b3
```

Important Note

The OpenTelemetry examples in this chapter rely on an experimental release of the logging signal for OpenTelemetry. This means it's possible that by the time you're reading this, the updated packages have moved methods to different packages. The release notes available for each release should help you identify where the packages have moved to (<https://github.com/open-telemetry/opentelemetry-python/releases>).

Additionally, in this chapter, we will use an instrumentation package made available by the OpenTelemetry Python community. This instrumentation will assist us when adding logging information to Flask applications that are part of the grocery store. Install the package via pip with the following command:

```
$ pip install opentelemetry-instrumentation-wsgi
```

The code for this chapter is available in the companion repository. The following uses git to copy the repository locally:

```
$ git clone https://github.com/PacktPublishing/Cloud-Native-  
Observability
```

The completed code for the examples in this chapter is available in the chapter06 directory. If you're interested in writing the code yourself, I suggest you start by copying the code in the chapter04 directory and following along.

Lastly, we will need to install the libraries that the grocery store relies on. This can be done via the following pip command:

```
$ pip install flask requests
```

We're now ready to start logging!

Configuring OpenTelemetry logging

Unlike with the two signals we covered in *Chapter 4, Distributed Tracing*, and *Chapter 5, Metrics - Recording Measurements*, the logging signal in OpenTelemetry does not concern itself with standardizing a logging interface. Many languages already have an established logging API, and a decision early on in OpenTelemetry was made to leverage those pre-existing tools. Although OpenTelemetry provides an API capable of producing logging, which we'll use shortly, the signal is intent on hooking into the existing logging facilities. Its focus is to augment the logs produced and provide a mechanism to correlate those logs with other signals. *Figure 6.1* shows us the components of the logging pipeline:

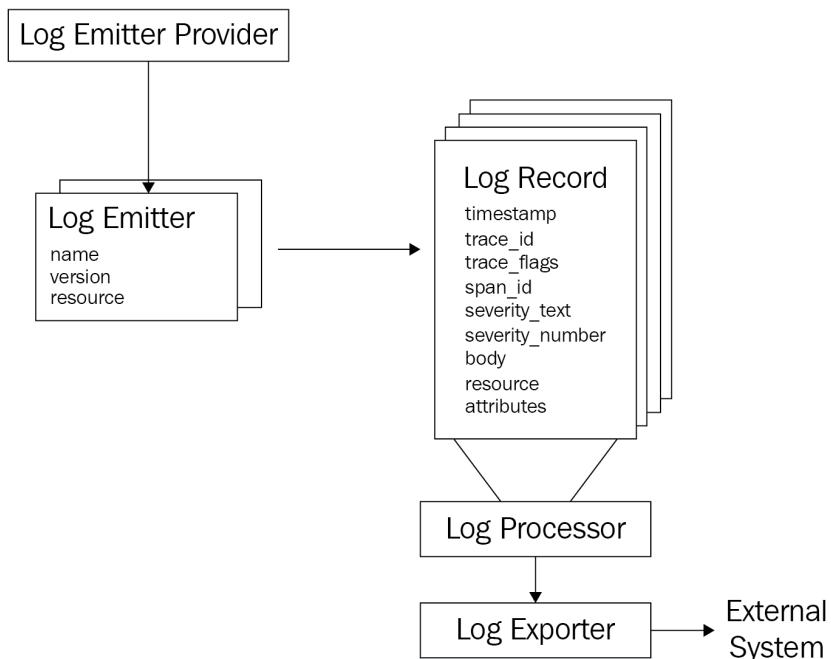


Figure 6.1 – The logging pipeline

These components combine to produce log records and emit them to external systems. The logging pipeline is comprised of the following:

- A `LogEmitterProvider`, which provides a mechanism to instantiate one or more log emitters
- The `LogEmitter`, which produces `LogRecord` data
- The `LogProcessor`, which consumes log records and passes them on to a `LogExporter` for sending the data to a backend

First, as with all the other OpenTelemetry signals, we must configure the provider. The following code instantiates a `LogEmitterProvider` from the SDK, passes in a resource via the `resource` argument, and then sets the global log emitter via the `set_log_emitter_provider` method:

logs.py

```
from opentelemetry.sdk._logs import LogEmitterProvider, set_
log_emitter_provider
from opentelemetry.sdk.resources import Resource

def configure_log_emitter_provider():
    provider = LogEmitterProvider(resource=Resource.create())
    set_log_emitter_provider(provider)
```

Configuring `LogEmitter` alone won't allow us to produce telemetry. We'll need a log processor and an exporter to go a step further. Let's add `BatchLogProcessor`, which, as the name suggests, batches the processing of log records. We will also use a `ConsoleLogExporter` to output logging information to the console:

logs.py

```
from opentelemetry.sdk._logs.export import ConsoleLogExporter,
BatchLogProcessor
from opentelemetry.sdk._logs import LogEmitterProvider, set_
log_emitter_provider
from opentelemetry.sdk.resources import Resource

def configure_log_emitter_provider():
    provider = LogEmitterProvider(resource=Resource.create())
    set_log_emitter_provider(provider)
```

```
exporter = ConsoleLogExporter()
provider.add_log_processor(BatchLogProcessor(exporter))
```

With OpenTelemetry configured, we're now ready to start instrumenting our logs.

Producing logs

Following the pattern from previous signals, we should be ready to get an instance of a log producer and start logging, right? Well, not quite – let's find out why.

Using LogEmitter

Using the same method that we used for metrics and tracing, we can now obtain `LogEmitter`, which will allow us to use the OpenTelemetry API to start producing logs. The following code shows us how to accomplish this using the `get_log_emitter` method:

logs.py

```
from opentelemetry.sdk._logs import (
    LogEmitterProvider,
    get_log_emitter_provider,
    set_log_emitter_provider,
)

if __name__ == "__main__":
    configure_log_emitter_provider()
    log_emitter = get_log_emitter_provider().get_log_emitter(
        "shopper",
        "0.1.2",
)
```

With `LogEmitter` in hand, we're now ready to generate `LogRecord`. The `LogRecord` contains the following information:

- `timestamp`: A time associated with the log record in nanoseconds.
- `trace_id`: A hex-encoded identifier of the trace to correlate with the log record. There will be more on this, the span identifier, and trace flags shortly.
- `span_id`: A hex-encoded identifier of the span to correlate with the log record.

- `trace_flags`: Trace flags associated with the trace active when the log record was produced.
- `severity_text`: A string representation of the severity level.
- `severity_number`: A numeric value of the severity level.
- `body`: The contents of the log message being recorded.
- `resource`: The resource associated with the producer of the log record.
- `attributes`: Additional information associated with the log record in the form of key-value pairs.

Each one of those fields can be passed as an argument to the constructor; note that all those fields are optional. The following creates `LogRecord` with some minimal information and calls `emit` to produce a log entry:

logs.py

```
import time
from opentelemetry.sdk._logs import (
    LogEmitterProvider,
    LogRecord,
    get_log_emitter_provider,
    set_log_emitter_provider,
)

if __name__ == "__main__":
    configure_log_emitter_provider()
    log_emitter = get_log_emitter_provider().get_log_emitter(
        "shopper",
        "0.1.2",
    )
    log_emitter.emit(
        LogRecord(
            timestamp=time.time_ns(),
            body="first log line",
        )
    )
```

After all this work, we can finally see a log line! Run the code, and the output should look something like this:

output

```
{ "body": "first log line", "name": null, "severity_number": "None", "severity_text": null, "attributes": null, "timestamp": 1630814115049294000, "trace_id": "", "span_id": "", "trace_flags": null, "resource": "" }
```

As you can see, there's a lot of information missing to give us a full picture of what was happening. One of the most important pieces of information associated with a log entry is the severity level. The OpenTelemetry specification defines 24 different log levels categorized in 6 severity groups, as shown in the following figure:

SeverityNumber range	Range name	Meaning
1-4	TRACE	A fine-grained debugging event. Typically disabled in default configurations.
5-8	DEBUG	A debugging event.
9-12	INFO	An informational event. Indicates that an event happened.
13-16	WARN	A warning event. Not an error but is likely more important than an informational event.
17-20	ERROR	An error event. Something went wrong.
21-24	FATAL	A fatal error such as an application or system crash.

Figure 6.2 – Log severity levels defined by OpenTelemetry

When defining the severity level, all log levels above that number are reported.

Let's ensure the log record we generate sets a meaningful severity level:

logs.py

```
from opentelemetry.sdk._logs.severity import SeverityNumber

if __name__ == "__main__":
    ...
    log_emitter.emit(
        LogRecord(
            timestamp=time.time_ns(),
            body="first log line",
            severity_number=SeverityNumber.INFO,
```

```
)  
)
```

There – now at least readers of those logs should be able to know how important those log lines are. Run the code and look for the severity number in the output:

output

```
{"body": "first log line", "name": null, "severity_ number": "<SeverityNumber.INFO: 9>", "severity_text": null, "attributes": null, "timestamp": 1630814944956950000, "trace_id": "", "span_id": "", "trace_flags": null, "resource": ""}
```

As mentioned earlier in this chapter, one of the goals of the OpenTelemetry logging signal is to remain interoperable with existing logging APIs. Looking at how much work we just did to get a log line with minimal information, it really seems like there should be a better way, and there is!

The standard logging library

What if we tried using the standard logging library available in Python to interact with OpenTelemetry instead? The logging library has been part of the standard Python library since version 2.3 and is used by many popular frameworks, such as *Django* and *Flask*.

Important Note

The standard logging module in Python is quite powerful and flexible. If you're not familiar with it, it may take some time to get used to it. I recommend reading the Python docs available on [python.org](https://docs.python.org/3/library/logging.html) here: <https://docs.python.org/3/library/logging.html>.

The Python implementation of the OpenTelemetry signal provides an additional component to use, `OTLPHandler`. The following figure shows where `OTLPHandler` fits in with the rest of the logging pipeline:

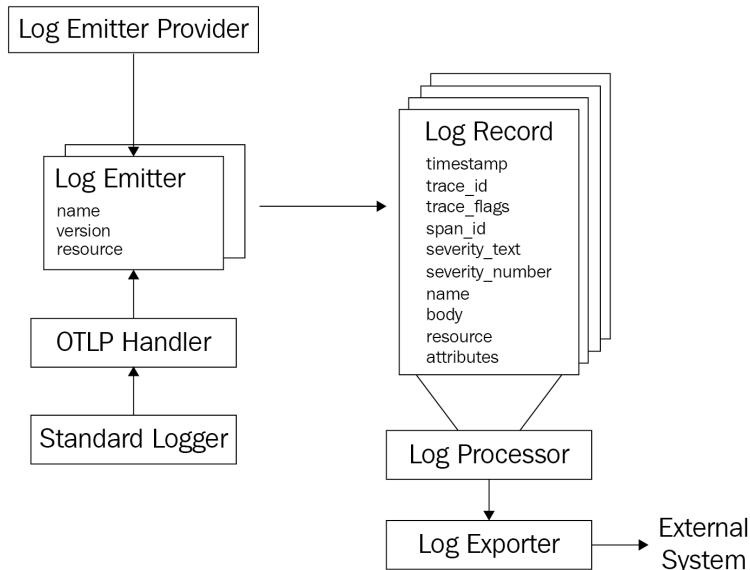


Figure 6.3 – OTLPHandler uses LogEmitter to produce logs

`OTLPHandler` extends the standard logging library's `logging.Handler` class and uses the configured `LogEmitter` to produce log records.

Important note:

The OTLPHandler was renamed LoggingHandler in releases of the opentelemetry-sdk package newer than 1.10.0. Be sure to update any references to it in the examples if you've installed a newer version.

The following code block first imports the logging module. Then, using the getLogger method, a standard Logger object is obtained. This is the object we will use anytime a log line is needed from the application. Finally, OTLPHandler is added to logger, and a warning message is logged:

logs.py

```
import logging
from opentelemetry.sdk._logs import (
    LogEmitterProvider,
```

```
    LogRecord,  
    OTLPHandler,  
    get_log_emitter_provider,  
    set_log_emitter_provider,  
)  
  
if __name__ == "__main__":  
    ...  
    logger = logging.getLogger(__file__)  
    handler = OTLPHandler()  
    logger.addHandler(handler)  
    logger.warning("second log line")
```

Let's see how the information generated differs from the previous example; many of the fields are automatically filled in for us:

- The timestamp is set to the current time.
- The severity number and text are set based on the method used to record a log – in this case, the warning method sets the log severity to WARN.
- Trace and span information is set by pulling information from the current context. As our example does not include starting a trace, we should expect the values in these fields to be invalid.
- Resource data is set via the log emitter provider.

This provides us with a significant improvement in the data generated.

output

```
{"body": "second log line", "name": null, "severity_number":  
"<SeverityNumber.WARN: 13>", "severity_text": "WARNING",  
"attributes": {}, "timestamp": 1630810960785737984,  
"trace_id": "0x0000000000000000000000000000000000000000000000000000000000000000", "span_  
id": "0x00000000000000000000000000000000", "trace_flags": 0, "resource":  
"BoundedAttributes({'telemetry.sdk.language': 'python',  
'telemetry.sdk.name': 'opentelemetry', 'telemetry.sdk.version':  
'1.9.0', 'service.name': 'unknown_service'}, maxlen=None")}
```

Not only does this output contain richer data, but we also didn't need to work nearly as hard to obtain it, and we used a standard library to generate the logs. The `attributes` field doesn't appear to contain anything useful yet – let's fix that. `OTLPHandler` creates the attribute dictionary by looking at any extra attributes defined in the standard `LogRecord`. The following code passes an `extra` argument at logging time:

logs.py

```
if __name__ == "__main__":
    ...
    logger.warning("second log line", extra={"key1": "val1"})
```

As with other attribute dictionaries we may have encountered previously, they should contain information relevant to the specific event being logged. The output should now show us the additional attributes:

output

```
{"body": "second log line", "name": null, "severity_number": "<SeverityNumber.WARN: 13>", "severity_text": "WARNING", "attributes": {"key1": "val1"}, "timestamp": 1630946024854904064, "trace_id": "0x00000000000000000000000000000000", "span_id": "0x0000000000000000", "trace_flags": 0, "resource": "BoundedAttributes({'telemetry.sdk.language': 'python', 'telemetry.sdk.name': 'opentelemetry', 'telemetry.sdk.version': '1.9.0', 'service.name': 'unknown_service'}, maxlen=None")}
```

Let's produce one last example with the standard logger and update the previous code to record a log using the `info` method. This should give us the same severity as the example where we used the log emitter directly:

logs.py

```
import logging
if __name__ == "__main__":
    ...
    logger.info("second log line")
```

Run the code again to see the result. If you're no longer seeing a log with the *second log line* as its body and are perplexed, don't worry – you're not alone. This is due to a feature of the standard logging library. The Python logging module creates a root logger, which is used anytime a more specific logger isn't configured. By default, the root logger is configured to only log messages with a severity of a warning or higher. Any logger instantiated via `getLogger` inherits that severity, which explains why our info level messages are not displayed. Our example can be fixed by calling `setLevel` for the logger we are using in our program:

logs.py

```
if __name__ == "__main__":
    ...
    logger = logging.getLogger(__file__)
    logger.setLevel(logging.DEBUG)
    handler = OTLPHandler()
    logger.addHandler(handler)
    logger.info("second log line")
```

The output should now contain the log line as we expected:

output

```
{"body": "second log line", "name": null, "severity_number": "<SeverityNumber.INFO: 9>", "severity_text": "INFO", "attributes": {}, "timestamp": 1630857128712922112, "trace_id": "0x00000000000000000000000000000000", "span_id": "0x0000000000000000", "trace_flags": 0, "resource": "BoundedAttributes({'telemetry.sdk.language': 'python', 'telemetry.sdk.name': 'opentelemetry', 'telemetry.sdk.version': '1.9.0', 'service.name': 'unknown_service'}, maxlen=None)"}

---


```

An alternative way to configure the log level of the root logger is to use the `basicConfig` method of the logging module. This allows you to configure the severity level, formatting, and so on (<https://docs.python.org/3/library/logging.html#logging.basicConfig>). Another benefit of using the existing logging library means that with a little bit of additional configuration, any existing application should be able to leverage OpenTelemetry logging. Speaking of an existing application, let's return to the grocery store.

A logging signal in practice

Getting familiar with the logging signal theory is great; now it's time to put it into practice. Before using OpenTelemetry logging in the grocery store, let's take a minute to move the configuration code into the `common.py` module:

common.py

```
import logging
from opentelemetry.sdk._logs.export import ConsoleLogExporter,
BatchLogProcessor
from opentelemetry.sdk._logs import (
    LogEmitterProvider,
    OTLPHandler,
    set_log_emitter_provider,
)

def configure_logger(name, version):
    provider = LogEmitterProvider(resource=Resource.create())
    set_log_emitter_provider(provider)
    exporter = ConsoleLogExporter()
    provider.add_log_processor(BatchLogProcessor(exporter))
    logger = logging.getLogger(name)
    logger.setLevel(logging.DEBUG)
    handler = OTLPHandler()
    logger.addHandler(handler)
    return logger
```

With the code in place, we can now obtain a logger in the same fashion as we obtained a tracer and a meter previously. The following code updates the shopper application to instantiate a logger via `configure_logger`. Additionally, let's update the `add_item_to_cart` method to use `logger.info` rather than `print`:

shopper.py

```
from common import configure_tracer, configure_meter,
configure_logger

tracer = configure_tracer("shopper", "0.1.2")
```

```
meter = configure_meter("shopper", "0.1.2")
logger = configure_logger("shopper", "0.1.2")

@tracer.start_as_current_span("add item to cart")
def add_item_to_cart(item, quantity):
    span = trace.get_current_span()
    span.set_attributes(
        {
            "item": item,
            "quantity": quantity,
        }
    )
    logger.info("add {} to cart".format(item))
```

Use the following commands in separate terminals to launch the grocery store, the legacy inventory, and finally, the shopper applications:

```
$ python legacy_inventory.py
$ python grocery_store.py
$ python shopper.py
```

Pay special attention to output running from the previous command; it should include similar output, confirming that our configuration is correct:

output

```
{"body": "add orange to cart", "name": null, "severity_number": "<SeverityNumber.INFO: 9>", "severity_text": "INFO", "attributes": {}, "timestamp": 1630859469283874048, "trace_id": "0x67a8df13b8d5678912a8101bb5724fa4", "span_id": "0x0fc5e89573d7f794", "trace_flags": 1, "resource": "BoundedAttributes({'telemetry.sdk.language': 'python', 'telemetry.sdk.name': 'opentelemetry', 'telemetry.sdk.version': '1.9.0', 'service.name': 'unknown_service'}, maxlen=None)"}
```

This is a great starting point; let's see how we can correlate the information from this log line with the information from our traces.

Distributed tracing and logs

We saw earlier in this chapter that the `LogRecord` class contains fields for span and trace identifiers as well as trace flags. The intention behind this is to allow logs to be correlated with specific traces and spans, permitting the end user to gain a better understanding of what their application is doing when it's running in production. So often, the process of correlating telemetry involves searching tirelessly through events using a timestamp as a mechanism to match up different sources of information. This is not always practical for the following reasons:

- Many operations happen concurrently on the same system, making it difficult to know which operation caused the event.
- The difficulty caused by operations happening simultaneously is exacerbated in distributed systems as even more operations are occurring.
- The clocks across different systems may, and often do, drift. This drift causes timestamps to not match.

A mechanism developed to address this has been to produce a unique event identifier for each event and add this identifier to all logs recorded. One challenge of this is ensuring that this information is then propagated across the entire system; this is exactly what the trace identifier in OpenTelemetry does. As shown in *Figure 6.4*, the trace and span identifiers can pinpoint the specific operation that triggers a log to be recorded:

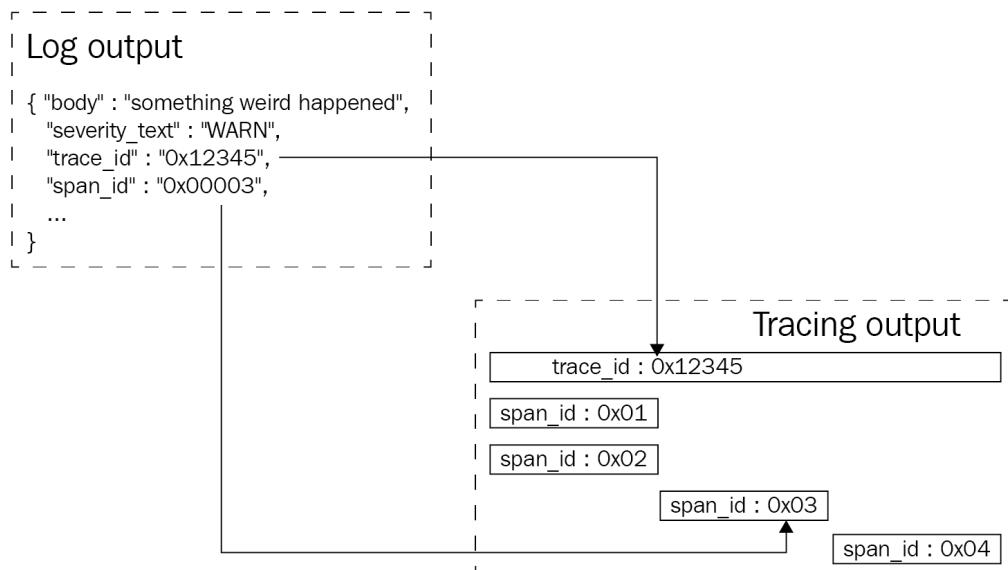


Figure 6.4 – Log and trace correlation

Returning to the output from the previous example, the following shows the logging output as well as a snippet of the tracing output containing the name of the operations and their identifiers. See whether you can determine from the output which operation triggered the log record:

output

```
{ "body": "add orange to cart", "name": null, "severity_number": "<SeverityNumber.INFO: 9>", "severity_text": "INFO", "attributes": {}, "timestamp": 1630859469283874048, "trace_id": "0x67a8df13b8d5678912a8101bb5724fa4", "span_id": "0x0fc5e89573d7f794", "trace_flags": 1, "resource": "BoundedAttributes({'telemetry.sdk.language': 'python', 'telemetry.sdk.name': 'opentelemetry', 'telemetry.sdk.version': '1.9.0', 'service.name': 'unknown_service'}, maxlen=None)" } { "name": "web request", "context": { "trace_id": "0x67a8df13b8d5678912a8101bb5724fa4", "span_id": "0x6e4e03cacd3411b5", }, } { "name": "add item to cart", "context": { "trace_id": "0x67a8df13b8d5678912a8101bb5724fa4", "span_id": "0x0fc5e89573d7f794", }, } { "name": "browse", "context": { "trace_id": "0x67a8df13b8d5678912a8101bb5724fa4", "span_id": "0x5a2262c9dd473b40", }, } { "name": "visit store", }
```

```
    "context": {
        "trace_id": "0x67a8df13b8d5678912a8101bb5724fa4",
        "span_id": "0x504caee882574a9e",
    },
}
```

If you've guessed that the log line was generated by the *add item to cart* operation, you've guessed correctly. Although this particular example is simple since you're already familiar with the code itself, you can imagine how valuable this information can be to troubleshoot an unfamiliar system. Equipped with the information provided by the distributed trace associated with the log record, you're empowered to jump into the source code and debug an issue faster. Let's see how we can use OpenTelemetry logging with the other applications in our system.

OpenTelemetry logging with Flask

As covered previously in the chapter, many frameworks, including Flask, use the standard logging library in Python. This makes configuring OpenTelemetry for the grocery store similar to how any changes to logging in Flask would be done. The following code imports and uses `configure_logger` to set up the logging pipeline. Next, we use the logging module's `dictConfig` method to add `OTLPHandler` to the root logger, and configure the severity level to `DEBUG` to ensure all our logs are output. In a production setting, you will likely want to make this option configurable rather than hardcode it to `debug` level to save costs:

grocery_store.py

```
from logging.config import dictConfig
from common import (
    configure_meter,
    configure_tracer,
    configure_logger,
    set_span_attributes_from_flask,
    start_recording_memory_metrics,
)
tracer = configure_tracer("grocery-store", "0.1.2")
meter = configure_meter("grocery-store", "0.1.2")
logger = configure_logger("grocery-store", "0.1.2")
dictConfig(
```

```
{  
    "version": 1,  
    "handlers": {  
        "otlp": {  
            "class": "opentelemetry.sdk._logs.OTLPHandler",  
        }  
    },  
    "root": {"level": "DEBUG", "handlers": ["otlp"]},  
}  
)  
app = Flask(__name__)
```

Ensure some requests are sent to the grocery store either by running `shopper.py` or via `curl` and see what the output from the server looks like now. The following output shows it before the change on the first line and after the change on the second line:

output

```
127.0.0.1 - - [05/Sep/2021 10:58:28] "GET /products HTTP/1.1"  
200 -  
{ "body": "127.0.0.1 - - [05/Sep/2021 10:58:48] \"GET /  
products HTTP/1.1\" 200 -", "name": null, "severity_  
number": "<SeverityNumber.INFO: 9>", "severity_text":  
"INFO", "attributes": {}, "timestamp": 1630864728996940032,  
"trace_id": "0x00000000000000000000000000000000", "span_  
id": "0x0000000000000000", "trace_flags": 0, "resource":  
"BoundedAttributes({'telemetry.sdk.language': 'python',  
'telemetry.sdk.name': 'opentelemetry', 'telemetry.sdk.version':  
'1.9.0', 'service.name': 'unknown_service'}, maxlen=None)"}
```

We can see the original message is now recorded as the body of the message, and all the additional information is also presented. Although, if we look closely, we can see that the `span_id`, `trace_id`, and `trace_flags` information is missing. It looks like the context for our request is lost somewhere along the way, so let's fix that. What is confusing about this is that we already have hooks defined to handle `before_request` and `teardown_request`, which, in theory, should ensure that the trace information is available. However, the log record we see is generated by Flask's built-in web server (`wsgi`), not the Flask application, and is triggered after the original request has been completed as far as Flask knows. We can address this by creating middleware ourselves, but thankfully, we don't have to.

Logging with WSGI middleware

The OpenTelemetry community publishes a package that provides support for instrumenting an application that uses a wsgi-compatible implementation, such as the built-in Flask server. The `opentelemetry-instrumentation-wsgi` package provides the middleware that hooks into the appropriate mechanisms to make trace information for the duration of the request. The following code imports the middleware and updates the Flask app to use it:

grocery_store.py

```
from opentelemetry.instrumentation.wsgi import
OpenTelemetryMiddleware
...
app = Flask(__name__)
app.wsgi_app = OpenTelemetryMiddleware(app.wsgi_app)
```

With the middleware in place, a new request to our application should allow us to see the `span_id`, `trace_id`, and `trace_flags` components that we expect:

output

```
{"body": "127.0.0.1 - - [05/Sep/2021 11:39:36] \"GET /products HTTP/1.1\" 200 -", "name": null, "severity_number": "<SeverityNumber.INFO: 9>", "severity_text": "INFO", "attributes": {}, "timestamp": 1630867176948227072, "trace_id": "0xf999a4164ac2f20c20549f19abd4b434", "span_id": "0xed5d3071ece38633", "trace_flags": 1, "resource": "BoundedAttributes({'telemetry.sdk.language': 'python', 'telemetry.sdk.name': 'opentelemetry', 'telemetry.sdk.version': '1.9.0', 'service.name': 'unknown_service'}, maxlen=None)"}
```

We will look at how this works in more detail in *Chapter 7, Instrumentation Libraries*, and see how we can simplify the application code using instrumentation libraries. For the purpose of this example, it's enough to know that the middleware enables us to see the tracing information in the log we are recording.

Resource correlation

Another piece of data that OpenTelemetry logging uses when augmenting telemetry is the resource attribute. As you may remember from previous chapters, the resource describes the source of the telemetry. This will allow us to correlate events occurring across separate signals for the same resource. In *Chapter 4, Distributed Tracing*, we defined a `LocalMachineResourceDetector` class that produces an OpenTelemetry resource that includes information about the local machine. Let's update the code in `configure_logger` that instantiates the `LogEmitterProvider` to use this resource, rather than create an empty resource:

common.py

```
def configure_logger(name, version):
    local_resource = LocalMachineResourceDetector().detect()
    resource = local_resource.merge(
        Resource.create(
            {
                ResourceAttributes.SERVICE_NAME: name,
                ResourceAttributes.SERVICE_VERSION: version,
            }
        )
    )
    provider = LogEmitterProvider(resource=resource)
    set_log_emitter_provider(provider)
    ...
```

With the change in place, run `shopper.py` once again to see that the log record now contains more meaningful data about the source of the log entry:

```
{"body": "add orange to cart", "name": null, "severity_number": "<SeverityNumber.INFO: 9>", "severity_text": "INFO", "attributes": {}, "timestamp": 1630949852869427968, "trace_id": "0x2ff0e5c9886f2672c3af4468483d341d", "span_id": "0x40d72ae565b4c19a", "trace_flags": 1, "resource": "BoundedAttributes({'telemetry.sdk.language': 'python', 'telemetry.sdk.name': 'opentelemetry', 'telemetry.sdk.version': '1.9.0', 'net.host.name': 'MacBook-Pro.local', 'net.host.ip': '127.0.0.1', 'service.name': 'shopper', 'service.version': '0.1.2'}, maxlen=None)"}

---


```

Looking at the previous output, we now know the name and version of the service. We also have valuable information about the machine that generated this information. In a distributed system, this information can be used in combination with metrics generated by the same resource to identify problems with a specific system, compute node, environment, or even region.

Summary

With the knowledge of this chapter ingrained in our minds, we have now covered the core signals that OpenTelemetry helps produce. Understanding how to produce telemetry by manually instrumenting code is a building block on the road to improving observability. Without telemetry, the job of understanding what a system is doing is much more difficult.

In this chapter, we learned about the purpose of the logging implementation in OpenTelemetry, as well as how it is intended to co-exist with existing logging implementations. After configuring the logging pipeline, we learned how to use the OpenTelemetry API to produce logs and compared doing so with using a standard logging API. Returning to the grocery store, we explored how logging can be correlated with traces and metrics. This allowed us to understand how we may be able to leverage OpenTelemetry logging within existing applications to improve our ability to use log statements when debugging applications.

Finally, we scratched the surface of how instrumentation libraries can help to make the production of telemetry easier. We will take an in-depth look at this in the next chapter, dedicated to simplifying the grocery store application by leveraging existing instrumentation libraries.

7

Instrumentation Libraries

Understanding the ins and outs of the OpenTelemetry API is quite helpful for manually instrumenting code. But what if we could save ourselves some of that work and still have visibility into what our code is doing? As covered in *Chapter 3, Auto-Instrumentation*, one of the initial objectives of OpenTelemetry is providing developers with tools to instrument their applications at a minimal cost. Instrumentation libraries combined with auto-instrumentation enable users to start with OpenTelemetry without learning the APIs, and leverage the community's efforts and expertise.

This chapter will investigate the components of auto-instrumentation, how they can be configured, and how they interact with instrumentation libraries. Diving deeper into the implementation details of instrumentation libraries will allow us to understand precisely how telemetry data is produced. Although telemetry created automatically may seem like magic, we'll seek to unveil the mechanics behind this illusion. The chapter covers the following main topics:

- Auto-instrumentation configuration and its components
- The Requests library instrumentor
- Automatic configuration

- Revisiting the grocery store
- The Flask library instrumentor
- Finding instrumentation libraries

With this information, we will revisit some of our existing code in the grocery store to simplify our code and manage and improve the generated telemetry. Along the way, we will look at the specifics of existing third-party libraries supported by the OpenTelemetry project. Let's start with setting up our environment.

Technical requirements

The examples in this chapter are provided in this book's companion repository, found here: <https://github.com/PacktPublishing/Cloud-Native-Observability>. The source code can be downloaded via git as per the following command:

```
$ git clone https://github.com/PacktPublishing/Cloud-Native-Observability  
$ cd Cloud-Native-Observability/chapter07
```

The completed examples from this chapter are in the `chapter7` directory. If you'd prefer the refactor along, copy the code from `chapter6` as a starting point. Next, we'll need to ensure the version of Python on your system is at least 3.6. You can verify it with the following commands:

```
$ python --version  
Python 3.8.9  
$ python3 --version  
Python 3.8.9
```

This chapter will use the same `opentelemetry-api`, `opentelemetry-sdk`, and `opentelemetry-propagator-b3` packages we installed in previous chapters. In addition, we will use the `opentelemetry-instrumentation` and `opentelemetry-distro` packages. Install the packages via pip now:

```
$ pip install opentelemetry-api \  
    opentelemetry-sdk \  
    opentelemetry-instrumentation \  
    opentelemetry-propagator-b3 \  
    opentelemetry-distro
```

We will need to install additional packages libraries used by our applications: the Flask and Requests libraries. Lastly, we will install the instrumentation libraries that automatically instrument the calls for those libraries. The standard naming convention for instrumentation libraries in OpenTelemetry is to prefix the library's name being instrumented with `opentelemetry-instrumentation-`. Use pip to install those packages now:

```
$ pip install flask \
    opentelemetry-instrumentation-flask \
    requests \
    opentelemetry-instrumentation-requests
```

Ensure all the required packages have been installed by looking at the output from `pip freeze`, which lists all the packages installed:

```
$ pip freeze | grep opentelemetry
opentelemetry-api==1.9.0
opentelemetry-distro==0.28b0
opentelemetry-instrumentation==0.28b0
opentelemetry-instrumentation-flask==0.28b0
opentelemetry-instrumentation-requests==0.28b0
opentelemetry-instrumentation-wsgi==0.28b0
opentelemetry-propagator-b3==1.9.0
opentelemetry-proto==1.9.0
opentelemetry-sdk==1.9.0
opentelemetry-semantic-conventions==0.28b0
opentelemetry-util-http==0.28b0
```

Throughout the chapter, we will rely on two scripts made available by the `opentelemetry-instrumentation` package: `opentelemetry-instrument` and `opentelemetry-bootstrap`. Ensure these scripts are available in your path with the following commands:

```
$ opentelemetry-instrument --help
usage: opentelemetry-instrument [-h] ...
$ opentelemetry-bootstrap --help
usage: opentelemetry-bootstrap [-h] ...
```

Now that we have all the packages installed and the code available, let's see how auto-instrumentation works in practice.

Auto-instrumentation configuration

Since auto-instrumentation aims to get started as quickly as possible, let's see how fast we can generate telemetry with as little code as possible. The following code makes a web request to <https://www.cloudnativeobservability.com> and prints the HTTP response code:

http_request.py

```
import requests

url = "https://www.cloudnativeobservability.com"
resp = requests.get(url)
print(resp.status_code)
```

When running the code, assuming network connectivity is available and the URL we're requesting connects us to a server that is operating normally, we should see 200 printed out:

```
$ python http_request.py
200
```

Great, the program works; now it's time to instrument it. The following command uses the opentelemetry-instrument application to wrap the application we created. We will look more closely at the command and its options shortly. For now, run the command:

```
$ opentelemetry-instrument --traces_exporter console \
--metrics_exporter console \
--logs_exporter console \
python http_request.py
```

If everything went according to plan, we should now see the following output, which contains telemetry:

output

```
200
{
  "name": "HTTP GET",
  "context": {
    "trace_id": "0x953ca1322b930819077a921a838df0cd",
    "span_id": "0x5b3b72c9c836178a",
    "trace_state": "[]"
  },
  "kind": "SpanKind.CLIENT",
  "parent_id": null,
  "start_time": "2021-11-25T17:38:21.331540Z",
  "end_time": "2021-11-25T17:38:22.033434Z",
  "status": {
    "status_code": "UNSET"
  },
  "attributes": {
    "http.method": "GET",
    "http.url": "https://www.cloudnativeobservability.com",
    "http.status_code": 200
  },
  "events": [],
  "links": [],
  "resource": {
    "telemetry.sdk.language": "python",
    "telemetry.sdk.name": "opentelemetry",
    "telemetry.sdk.version": "1.9.0",
    "telemetry.auto.version": "0.28b0",
    "service.name": "unknown_service"
  }
}
```

Okay, that's exciting, but what just happened? *Figure 7.1* shows how the `opentelemetry-instrument` command is instrumenting the code for our web request by doing the following:

1. Loading the configuration options defined by the `OpenTelemetryDistro` class, which is part of the `opentelemetry-distro` package.
 2. Automatically configuring the telemetry pipelines for traces, metrics, and logs via `OpenTelemetryConfigurator`. The details of how this configuration is set will become clearer shortly.
 3. Iterating through instrumentor classes registered via entry points in the Python environment under `opentelemetry_instrumentor` to find available instrumentation libraries. In doing so, it finds and loads the `RequestsInstrumentor` class defined in the `opentelemetry-instrumentation-requests` package.
 4. With the instrumentation library loaded, the call to `get` is now processed by the `requests` instrumentation library, which creates a span before calling the original `get` method.

The preceding steps are depicted in the following diagram:

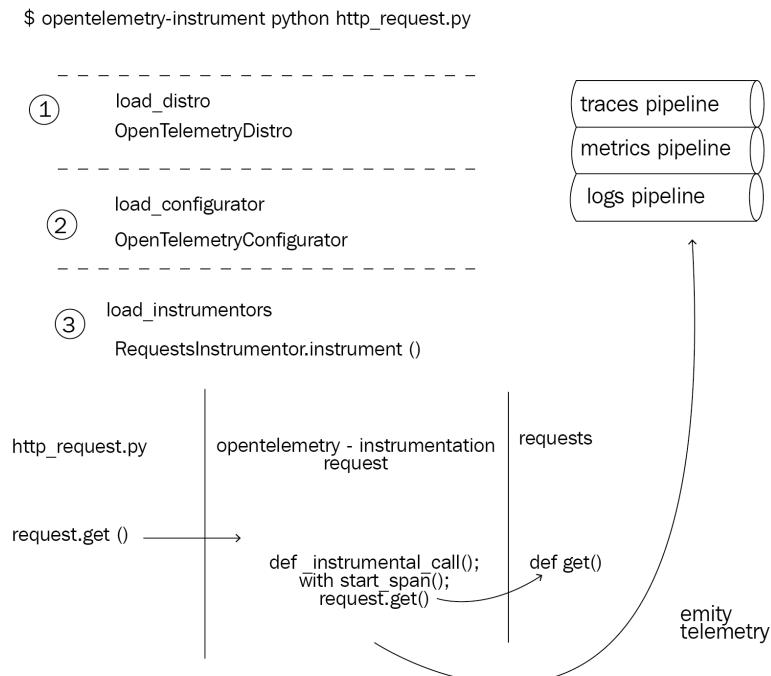


Figure 7.1 – opentelemetry-instrument

The configuration of the telemetry pipeline involves a few different mechanisms loaded via entry points at various times before the application code is executed. Thinking back to *Chapter 3, Auto-Instrumentation*, we introduced entry points (<https://packaging.python.org/specifications/entry-points/>) as a mechanism that allows Python packages to register classes or methods globally. The combination of entry points, interfaces, and options to choose from can make the configuration process a bit complex to understand.

OpenTelemetry distribution

The first step in the configuration process is loading classes registered under the `opentelemetry_distro` entry point. This entry point is reserved for classes adhering to the **BaseDistro** interface, and its purpose is to allow implementors to set configuration options at the earliest possible time. The term *distro* is short for distribution, a concept that is still being officially defined in OpenTelemetry. Essentially, a distro is a way for users to customize OpenTelemetry to fit their needs, allowing them to reduce the complexity of deploying and using OpenTelemetry. For example, the default configuration for OpenTelemetry Python is to configure an OpenTelemetry protocol exporter for all signals. This is accomplished via the `OpenTelemetryDistro` class mentioned previously. The following code shows us how the `OpenTelemetryDistro` class configures the default exporter by setting environment variables:

OpenTelemetryDistro class

```
class OpenTelemetryDistro(BaseDistro):
    """
    The OpenTelemetry provided Distro configures a default
    configuration out of the box.

    """
    def __configure(self, **kwargs):
        os.environ.setdefault(OTEL_TRACES_EXPORTER, "otlp_"
                             proto_grpc")
        os.environ.setdefault(OTEL_METRICS_EXPORTER, "otlp_"
                             proto_grpc")
        os.environ.setdefault(OTEL_LOGS_EXPORTER, "otlp_proto_"
                             grpc")
```

As a user, you could create your distribution to preconfigure all the specific parameters needed to tailor auto-instrumentation for your environment: for example, protocol, destination, and transport options. A list of open source examples extending the `BaseDistro` interface can be found here: <https://github.com/PacktPublishing/Cloud-Native-Observability/tree/main/chapter7#opentelemetry-distro-implementations>. With those options configured, you can then provide an entry point to your implementation of the `BaseDistro` interface, package it up, and add this new package as a dependency in your applications. Therefore, the distribution makes deploying a consistent configuration across a distributed system easier.

OpenTelemetry configurator

The next piece of the configuration puzzle is what is currently known in OpenTelemetry Python as the configurator. The purpose of the configurator is to load all the components defined in the configuration specified by the distro. Another way is to think of the distro as the co-pilot, deciding where the car needs to go, and the configurator as the driver. The configurator is an extensible and declarative interface for configuring OpenTelemetry. It is loaded by auto-instrumentation via, and you may have guessed it, an entry point. The `opentelemetry_configurator` entry point is reserved for classes adhering to the `_BaseConfigurator` interface, whose sole purpose is to prepare the logs, metrics, and traces pipelines to produce telemetry.

Important Note

As you may have noticed, the `_BaseConfigurator` class is preceded by an underscore. This is done intentionally for classes that are not officially part of the supported OpenTelemetry API in Python and warrant extra caution. Methods and classes that are not supported formally can and often do change with new releases.

The implementation of the `_BaseConfigurator` interface loaded in the previous example, the `OpenTelemetryConfigurator` class configures a telemetry pipeline for each signal using components from the standard `opentelemetry-sdk` package. As a user, this configurator is precisely what you want most of the time. However, if a user wishes to provide an alternative SDK, it would be possible to provide their configurator implementation to use this SDK instead.

This covers the two main entry points used by auto-instrumentation. We will continue discussing additional entry points throughout this chapter. As a reference, the following table captures the entry points used by OpenTelemetry Python along with the interface each entry point expects. The table also shows us a brief description of what each entry point is used for:

Entry point identifier	Interface	Purpose
<code>opentelemetry_distro</code>	<code>BaseDistro</code>	Provides configuration to the configurator. The <code>OpenTelemetryDistro</code> sets exporters to use OTLP via environment variables.
<code>opentelemetry_instrumentor</code>	<code>Instrumentor</code>	Registers available instrumentation libraries, all of which are loaded by auto-instrumentation by default.
<code>opentelemetry_configurator</code>	<code>Configurator</code>	Applies the available configuration. The OpenTelemetry SDK provides an implementation which loads configuration from environment variables.
<code>opentelemetry_tracer_provider</code>	<code>TracerProvider</code>	Allows users to load a custom implementation of a <code>TracerProvider</code> .
<code>opentelemetry_meter_provider</code>	<code>MeterProvider</code>	Allows users to load a custom implementation of a <code>MeterProvider</code> .
<code>opentelemetry_log_emitter_provider</code>	<code>LogEmitterProvider</code>	Allows users to load a custom implementation of a <code>LogEmitterProvider</code> .
<code>opentelemetry_traces_exporter</code>	<code>SpanExporter</code>	Registers span exporters available for auto-instrumentation.
<code>opentelemetry_metrics_exporter</code>	<code>MetricExporter</code>	Registers metric exporters available for auto-instrumentation.
<code>opentelemetry_logs_exporter</code>	<code>LogExporter</code>	Registers log exporters available for auto-instrumentation.
<code>opentelemetry_id_generator</code>	<code>IdGenerator</code>	Registers ID generators available for auto-instrumentation.

Figure 7.2 – Entry points used in OpenTelemetry Python

Similar to `OpenTelemetryDistro`, the `OpenTelemetryConfigurator` class and its parent use environment variables to achieve its goal of configuring OpenTelemetry for the end use.

Environment variables

To provide additional flexibility to users, OpenTelemetry supports the configuration of many of its components across all languages via environment variables. These variables are defined in the OpenTelemetry specification, ensuring each compliant language implementation understands them. This allows users to re-use the same configuration options across any language they choose. I recommend reading the complete list of options available in the specification repository found here: <https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/sdk-environment-variables.md>.

We will look more closely at specific variables as we refactor the grocery store further in this chapter. Many, but not all, of the environment variables used by auto-instrumentation are part of the specification linked previously. This is because the implementation details of each language may require additional variables not relevant to others. Language-specific environment variables are supported in the following format:

```
OTEL_{LANGUAGE}_{FEATURE}
```

As we'll see shortly, Python-specific options are prefixed with OTEL PYTHON_. Any option with this prefix will only be found in Python, and the naming convention helps set that expectation with users.

Command-line options

The last tool available to configure OpenTelemetry without editing the application code is the use of command-line arguments, which can be set when invoking `opentelemetry-instrument`. Recall the command we used to call in the earlier example:

```
$ opentelemetry-instrument --traces_exporter console \
                           --metrics_exporter console \
                           --logs_exporter console \
                           python http_request.py
```

This command used command-line arguments to override the traces, metrics, and logs exporters to use the console exporter instead of the configured default. All options available via command line can be listed using the `--help` flag when invoking `opentelemetry-instrument`. These options are the same as those available through environment variables, with a slightly easier name for convenience. The name of the command-line argument is the name of the environment variable in lowercase without the OTEL_ or OTEL PYTHON prefix. The following table shows a few examples:

Environment variable	Command line argument
OTEL_TRACES_EXPORTER	--traces_exporter
OTEL_PYTHON_METER_PROVIDER	--meter_provider
OTEL_PYTHON_ID_GENERATOR	--id_generator

Figure 7.3 – Environment variable to command-line argument translations

With that, we've covered how auto-instrumentation configures OpenTelemetry to generate the telemetry we saw. But what about the instrumented call? Let's see how the Requests library instrumentation works.

Requests library instrumentor

The Instrumentor interface provides instrumentation libraries with the minimum requirements a library must provide to support auto-instrumentation. Implementors must provide an implementation for `_instrument` and `_uninstrument`, that's all. The instrumentation implementation details vary from one library to another depending on whether the library offers any event or callback mechanisms for instrumentation. In the case of the Requests library, the `opentelemetry-instrumentation-requests` library relies on monkey patching the `Session.request` and `Session.send` methods from the `requests` library. This instrumentation library does the following:

1. Provides a wrapper method for the library calls that it instruments, and intercepts calls through those wrappers
2. Upon invocation, creates a new span by calling the `start_as_current_span` method of the OpenTelemetry API, ensuring the span name follows semantic conventions
3. Injects the context information into the request headers via the context API's `attach` method to ensure the tracing data is propagated to the request's destination
4. Reads the response and sets the status code accordingly via the span's `set_status` method

Important Note

Instrumentation libraries must check if the span will be recorded before adding additional attributes to avoid potentially costly operations. This is done to minimize the instrumentation's impact on existing applications when it is not in use.

Additional implementation details can be found in the `opentelemetry-python-contrib` repository: https://github.com/open-telemetry/opentelemetry-python-contrib/blob/main/instrumentation/opentelemetry-instrumentation-requests/src/opentelemetry/instrumentation/requests/__init__.py. The code may inspire you to write and contribute an instrumentation library of your own.

Additional configuration options

The Requests instrumentation library supports the following additional configurable options:

- `span_callback`: A callback mechanism to inject additional information into a span if available via this parameter. For example, this allows users to inject additional information from the response into the span.
- `name_callback`: The default name of a span created by the requests instrumentation library is in the HTTP `{method}` format. The `name_callback` parameter allows users to customize the name of the span as needed.
- `excluded_urls`: There are HTTP destinations for which capturing telemetry may not be desirable, a typical case being requests made to a health check endpoint. The `excluded_urls` parameter supports configuring a comma-separated list of URLs exempt from telemetry. This parameter is also configurable via the `OTEL_PYTHON_REQUESTS_EXCLUDED_URLS` environment variable and is available for use with auto-instrumentation.

As you may have noted by reading the description of each configuration option, not all these options are available for configuration via auto-instrumentation. It's possible to use instrumentation libraries without auto-instrumentation. Let's see how.

Manual invocation

The following code updates the previous example to configure a tracer and instrument the `requests.get` call via the instrumentation library:

http_request.py

```
import requests

from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import (
    BatchSpanProcessor,
    ConsoleSpanExporter,
)
from opentelemetry.instrumentation.requests import
    RequestsInstrumentor

def configure_tracer():
```

```
exporter = ConsoleSpanExporter()
span_processor = BatchSpanProcessor(exporter)
provider = TracerProvider()
provider.add_span_processor(span_processor)
trace.set_tracer_provider(provider)

configure_tracer()
RequestsInstrumentor().instrument()

url = "https://www.cloudnativeobservability.com"
resp = requests.get(url)
print(resp.status_code)
```

This is quite a bit of additional code. Since we're no longer relying on auto-instrumentation, we must configure the tracing pipeline manually. Running this code without invoking opentelemetry-instrument looks like this:

```
$ python http_request.py
```

This should yield very similar telemetry to what we saw earlier. The following shows an excerpt of that output:

output

```
200
{
  "name": "HTTP GET",
  "context": {
    "trace_id": "0xc2ee1f399911a10d361231a46c6fec1b",
  ...
}
```

We can further customize the telemetry produced by configuring additional options we discussed previously. The following code example will customize the name of the span and add other attributes to the data generated. It does so by doing the following:

- Adding a `rename_span` method to replace the HTTP prefix in the name
- Adding the `add_response_attribute` method to append header information from the response object as a span attribute
- Updating the call to `instrument` to utilize the new functionality

http_request.py

```
def rename_span(method, url):
    return f"Web Request {method}"

def add_response_attributes(span, response):
    span.set_attribute("http.response.headers", str(response.headers))

configure_tracer()
RequestsInstrumentor().instrument(
    name_callback=rename_span,
    span_callback=add_response_attributes,
)
```

Running the updated code should give us the slightly updated telemetry as per the following abbreviated sample output:

output

```
200
{
  "name": "Web Request GET",
  "attributes": {
    "http.method": "GET",
    "http.url": "https://www.cloudnativeobservability.com",
    "http.status_code": 200,
    "http.response.headers": "{ 'Connection': 'keep-alive',
'Content-Length': '1864', 'Server': 'GitHub.com'
...
}
```

With this, we've now seen how to leverage the Requests instrumentation library without using auto-instrumentation. The added flexibility of the features not available through auto-instrumentation is nice, but configuring pipelines is tedious. Thankfully, it's possible to get the best of both worlds by using auto-instrumentation and configuring the instrumentor manually. Update the example to remove all the configuration code. The following is all that should be left:

http_request.py

```
import requests

from opentelemetry.instrumentation.requests import
    RequestsInstrumentor

def rename_span(method, url):
    return f"Web Request {method}"

def add_response_attributes(span, response):
    span.set_attribute("http.response.headers", str(response.
        headers))

RequestsInstrumentor().instrument(
    name_callback=rename_span,
    span_callback=add_response_attributes,
)

resp = requests.get("https://www.cloudnativeobservability.com")
print(resp.status_code)
```

Run the new code via the following command we used earlier in the chapter:

```
$ opentelemetry-instrument --traces_exporter console \
                           --metrics_exporter console \
                           --logs_exporter console \
                           python http_request.py
```

Looking at the output, it's clear that something didn't go as planned. The following warning appears at the top of the output:

```
Attempting to instrument while already instrumented
```

Additionally, if you look through the telemetry generated, the span name is back to its original value, and the `response.headers` attribute is missing. Recall that the `opentelemetry-instrument` script iterates through all the installed instrumentors before calling the application code. This means that by the time our application code is executed, the `Request` instrumentor has already instrumented the `Requests` library.

Double instrumentation

Many instrumentation libraries have a safeguard in place to prevent double instrumentation. Double instrumentation in most cases would mean that every piece of telemetry generated is recorded twice. This causes all sorts of problems, from potential added performance costs to making telemetry analysis difficult.

We can ensure that the library isn't instrumented first to mitigate this issue. Add the following method call to your code:

http_request.py

```
import requests

from opentelemetry.instrumentation.requests import
    RequestsInstrumentor
...
    RequestsInstrumentor().uninstrument()
    RequestsInstrumentor().instrument(
        name_callback= rename_span,
        span_callback= add_response_attributes,
    )
```

Running this code once more shows us that the warning is gone and that the telemetry contains the customization we expected. All this with much simpler code. Great! Let's see now how we can apply this to the grocery store.

Automatic configuration

We added new instrumentation in the past three chapters and watched how we could generate more information each time we instrumented the code. We will now see how we can continue to provide the same level of telemetry but simplify our lives by removing some of the code. The first code we will be removing is the configuration code we extracted into the `common.py` module. If you recall from previous chapters, the purpose of the `configure_tracer`, `configure_meter`, and `configure_logger` methods, which we will review in detail shortly, is to do the following:

- Configure the emitter of telemetry.
- Configure the destination and mechanism to output the telemetry.
- Add resource information to identify our service.

As we saw earlier in this chapter, the `opentelemetry-instrument` script enables us to remove the code doing the configuration by interpreting environment variables or command-line arguments that will do the same thing. We will review the configuration code for each signal and look at the flags that can be used to replace the code with environment variables. One of the configurations common to all signals is the resource information; let's start there.

Configuring resource attributes

A resource provides information about the source of the telemetry. If you look through the `common.py` code, you may recall that each method used to configure a signal also called methods to configure the resource. The code looks something like the following:

common.py

```
local_resource = LocalMachineResourceDetector().detect()
resource = local_resource.merge(
    Resource.create(
        {
            ResourceAttributes.SERVICE_NAME: name,
            ResourceAttributes.SERVICE_VERSION: version,
        }
    )
)
```

The code uses a resource detector to fill in the hostname and IP address automatically. A current limitation of auto-instrumentation in Python is the lack of support for configuring resource detectors. Thankfully, since the functionality of our resource detector is somewhat limited, it's possible to replace it, as we'll see shortly.

The code also adds a service name and version information to our resource. Resource attributes can be configured for auto-instrumentation through one of the following options:

Environment variable	Command line argument
OTEL_RESOURCE_ATTRIBUTES	--resource_attributes

Figure 7.4 – Resource configuration

Note that the command-line arguments are shown here for reference only. For the remainder of the chapter, the commands used to run applications will use environment variables. The format of the parameters used for both methods is interchangeable. However, the OpenTelemetry specification only officially supports environment variables. These are consistent across implementations.

The following shows how using only environment variables to configure resources can produce the same result as the previous code. The example uses the `hostname` system utility to retrieve the name of the current host and `ipconfig` to retrieve the IP address. The invocation for these tools may vary depending on your system:

```
$ OTEL_RESOURCE_ATTRIBUTES="service.name=chap7-Requests-app,
  service.version=0.1.2,
  net.host.name='hostname',
  net.host.ip='ipconfig getifaddr en0'" \
opentelemetry-instrument --traces_exporter console \
  --metrics_exporter console \
  --logs_exporter console \
  python http_request.py
```

The resource information in the output from this command now includes the following details:

output

```
"resource": {
  "telemetry.sdk.language": "python",
  "telemetry.sdk.name": "opentelemetry",
```

```
    "telemetry.sdk.version": "1.9.0",
    "service.name": "chap7-Requests-app",
    "service.version": "0.1.2",
    "net.host.name": "cloud",
    "net.host.ip": "10.0.0.141",
    "telemetry.auto.version": "0.28b0"
}
```

We can now start configuring signals with resource attributes out of the way.

Configuring traces

The following code shows the `configure_tracer` method used to configure the tracing pipeline. Note that the code no longer contains resource configuration as we've already taken care of that:

common.py

```
def configure_tracer(name, version):
    exporter = ConsoleSpanExporter()
    span_processor = BatchSpanProcessor(exporter)
    provider = TracerProvider()
    provider.add_span_processor(span_processor)
    trace.set_tracer_provider(provider)
    return trace.get_tracer(name, version)
```

The main components to configure for tracing to emit telemetry are as follows:

- `TracerProvider`
- `SpanProcessor`
- `SpanExporter`

It's possible to set both TracerProvider and SpanExporter via environment variables. This is not the case for SpanProcessor. The OpenTelemetry SDK for Python defaults to using BatchSpanProcessor when auto-instrumentation is used in combination with the opentelemetry-distro package. Options for configuring BatchSpanProcessor are available via environment variables.

Important Note

BatchSpanProcessor will satisfy most use cases. However, if your application requires an alternative SpanProcessor implementation, it can be specified via a custom OpenTelemetry distribution package. Custom span processors can filter or enhance data before it is exported.

Another component we haven't talked about much yet is the sampler, which we'll cover in *Chapter 12, Sampling*. For now, it's enough to know that the sampler is also configurable via environment variables.

The following table shows the options for configuring the tracing pipeline. The acronym BSP stands for BatchSpanProcessor:

Environment variable	Command line argument
OTEL_PYTHON_TRACER_PROVIDER	--tracer_provider
OTEL_TRACES_EXPORTER	--traces_exporter
OTEL_TRACES_SAMPLER	--traces_sampler
OTEL_TRACES_SAMPLER_ARG	--traces_sampler_arg
OTEL_BSP_EXPORT_TIMEOUT	--bsp_export_timeout
OTEL_BSP_MAX_EXPORT_BATCH_SIZE	--bsp_max_export_batch_size
OTEL_BSP_MAX_QUEUE_SIZE	--bsp_max_queue_size
OTEL_BSP_SCHEDULE_DELAY	--bsp_schedule_delay

Figure 7.5 – Tracing configuration

As we continue adding configuration options, the command used to launch the application can get quite unruly. To alleviate this, I recommend exporting each variable as we go along. The following exports the OTEL_RESOURCE_ATTRIBUTES variable we previously set:

```
$ export OTEL_RESOURCE_ATTRIBUTES="service.name=chap7-Requests-
app, service.version=0.1.2, net.host.name='hostname', net.host.
ip='ipconfig getifaddr en0'"
```

We've already configured the exporter via command-line arguments in previous examples. The following shows us configuring the exporter and provider via environment variables. The `console` and `sdk` strings correspond to the name of the entry point for the `ConsoleSpanExporter` and the OpenTelemetry SDK `TracerProvider` classes:

```
$ OTEL_TRACES_EXPORTER=console \
  OTEL_PYTHON_TRACER_PROVIDER=sdk \
  opentelemetry-instrument --metrics_exporter console \
    --logs_exporter console \
    python http_request.py
```

Reading the output from the previous command is uneventful as it is just setting the same configuration in another way. However, we can now move on to metrics with this configuration in place.

Configuring metrics

The configuration for metrics is similar to the configuration for tracing, as we can see from the following code for the `configure_meter` method:

common.py

```
def configure_meter(name, version):
    exporter = ConsoleMetricExporter()
    provider = MeterProvider()
    set_meter_provider(provider)
    return get_meter_provider().get_meter(
        name=name,
        version=version,
    )
```

At the time of writing, the specification for metrics is reaching stability. As such, the support for auto-instrumentation and configuration will likely solidify over the coming months. For now, this section will focus on the options that are available and not likely to change, which covers the following:

- `MeterProvider`
- `MetricExporter`

The following table shows the options available to configure the metrics pipeline:

Environment variable	Command line argument
OTEL_PYTHON_METER_PROVIDER	--meter_provider
OTEL_METRICS_EXPORTER	--metrics_exporter

Figure 7.6 – Metrics configuration

The following command is provided as a reference for configuring MeterProvider and MetricsExporter via environment variables:

```
$ OTEL_METRICS_EXPORTER=console \
  OTEL_PYTHON_METER_PROVIDER=sdk \
  opentelemetry-instrument --logs_exporter console \
    python http_request.py
```

Note that running the previous command as is results in an error as it does not configure the tracing signal. Any signal not explicitly configured defaults to using the **OpenTelemetry Protocol (OTLP)** exporter, which we've not installed in this environment. As the application does not currently produce metrics, we wouldn't expect to see any changes in the telemetry emitted.

Configuring logs

The `configure_logger` method configures the following OpenTelemetry components:

- `LogEmitterProvider`
- `LogProcessor`
- `LogExporter`

common.py

```
def configure_logger(name, version):
    provider = LogEmitterProvider()
    set_log_emitter_provider(provider)
    exporter = ConsoleLogExporter()
    provider.add_log_processor(BatchLogProcessor(exporter))
    logger = logging.getLogger(name)
    logger.setLevel(logging.DEBUG)
```

```

handler = OTLPHandler()
logger.addHandler(handler)
return logger

```

As with metrics, the configuration and auto-instrumentation for the logging signal are still currently under development. The following table can be used as a reference for the environment variables and command-line arguments available to configure logging at the time of writing:

Environment variable	Command line argument
OTEL_PYTHON_LOG_EMITTER_PROVIDER	--log_emitter_provider
OTEL_LOGS_EXPORTER	--logs_exporter

Figure 7.7 – Logging configuration

As with the tracing configuration's span processor, there isn't currently a mechanism for configuring the log processor via auto-instrumentation. This can change in the future. Using those options, we know how to configure the last signal for auto-instrumentation:

```

$ OTEL_LOGS_EXPORTER=console \
  OTEL_PYTHON_LOG_EMITTER_PROVIDER=sdk \
  opentelemetry-instrument python http_request.py

```

We're almost ready to revisit the grocery store code with the signals and resources configured. The last thing left to configure is propagation.

Configuring propagation

Context propagation provides the ability to share context information across distributed systems. This can be accomplished via various mechanisms, as we discovered in *Chapter 4, Distributed Tracing – Tracing Code Execution*. To ensure applications can interoperate with any of the propagation formats, OpenTelemetry supports configuring propagators via the following environment variable:

Environment variable	Command line argument
OTEL_PROPAGATORS	--propagators

Figure 7.8 – Propagator configuration

Later in this chapter, an application will need to configure the B3 and TraceContext propagators. OpenTelemetry makes it possible to configure multiple propagators by specifying a comma-separated list. As mentioned earlier, with so many configuration options, using environment variables can become hard to manage. An effort is underway to add support for configuration files to OpenTelemetry, but the timeline on when that will be available is still in flux.

Recall the code we instrumented in the last three chapters. Let's go through it now and leverage configuration and the instrumentation libraries wherever possible.

Revisiting the grocery store

It's finally time to use all this new knowledge about auto-instrumentation to clean up the grocery store application. This section will showcase the simplified code that continues to produce the telemetry we've come to expect over the last few chapters. The custom decorators have been removed, as has the code configuring the tracer provider, meter provider, and log emitter provider. All we're left with now is the application code.

Legacy inventory

The legacy inventory service is a great place to start. It is a small Flask application with a single endpoint. The Flask instrumentor, installed at the beginning of the chapter via the `opentelemetry-instrumentation-flask` package, will replace the manual instrumentation code we previously added. The following code instantiates the Flask app and provides the `/inventory` endpoint:

legacy_inventory.py

```
#!/usr/bin/env python3
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/inventory")
def inventory():
    products = [
        {"name": "oranges", "quantity": "10"},
        {"name": "apples", "quantity": "20"},
    ]
    return jsonify(products)
```

```
if __name__ == "__main__":
    app.run(port=5001)
```

If you remember from previous chapters, this service was configured to use the B3 format propagator. This will be reflected in the configuration options we pass in when starting the service via auto-instrumentation:

```
$ OTEL_RESOURCE_ATTRIBUTES="service.name=legacy-inventory,
                           service.version=0.9.1,
                           net.host.name='hostname',
                           net.host.ip='ipconfig getifaddr en0'" \
OTEL_TRACES_EXPORTER=console \
OTEL_PYTHON_TRACER_PROVIDER=sdk \
OTEL_METRICS_EXPORTER=console \
OTEL_PYTHON_METER_PROVIDER=sdk \
OTEL_LOGS_EXPORTER=console \
OTEL_PYTHON_LOG_EMITTER_PROVIDER=sdk \
OTEL_PROPAGATORS=b3 \
opentelemetry-instrument python legacy_inventory.py
```

With this service running, let's look at the next one.

Grocery store

The next service to revisit is the grocery store. This service is also a Flask application and will leverage the same instrumentation library. In addition, it will use the Requests instrumentor to add telemetry to the calls it makes to the legacy inventory. The code looks like this:

grocery_store.py

```
#!/usr/bin/env python3
from logging.config import dictConfig
import requests
from flask import Flask
from opentelemetry.instrumentation.wsgi import
OpenTelemetryMiddleware
```

```
dictConfig(  
    {  
        "version": 1,  
        "handlers": {  
            "otlp": {  
                "class": "opentelemetry.sdk._logs.OTLPHandler",  
            }  
        },  
        "root": {"level": "DEBUG", "handlers": ["otlp"]},  
    }  
)  
  
app = Flask(__name__)  
app.wsgi_app = OpenTelemetryMiddleware(app.wsgi_app)  
  
@app.route("/")  
def welcome():  
    return "Welcome to the grocery store!"  
  
@app.route("/products")  
def products():  
    url = "http://localhost:5001/inventory"  
    resp = requests.get(url)  
    return resp.text  
  
if __name__ == "__main__":  
    app.run(port=5000)
```

Running the application will look very similar to running the legacy inventory with only a few different parameters:

- `service.name` and `service.version` will be updated to reflect the different applications.
- The propagators will be configured to use both B3 and TraceContext formats, making it possible for context to be propagated from the shopper through to the legacy inventory.

In a separate terminal window, with the legacy inventory service still running, run the following to start the grocery store:

```
$ OTEL_RESOURCE_ATTRIBUTES="service.name=grocery-store,
    service.version=0.1.2,
    net.host.name='hostname',
    net.host.ip='ipconfig getifaddr en0'" \
OTEL_TRACES_EXPORTER=console \
OTEL_PYTHON_TRACER_PROVIDER=sdk \
OTEL_METRICS_EXPORTER=console \
OTEL_PYTHON_METER_PROVIDER=sdk \
OTEL_LOGS_EXPORTER=console \
OTEL_PYTHON_LOG_EMITTER_PROVIDER=sdk \
OTEL_PROPAGATORS=b3,tracecontext \
opentelemetry-instrument python grocery_store.py
```

The grocery store is up and running. Now we just need to generate some requests via the shopper service.

Shopper

Finally, the shopper application initiates the request through the system.

The RequestsInstrumentor instruments web requests to the grocery store.

Of course, the backend requests don't tell the whole story about what goes on inside the shopper application.

As discussed in *Chapter 3, Auto-Instrumentation*, auto-instrumentation can be pretty valuable. In rare cases, it can even be enough to cover most of the functionality within an application. Applications focused on Create, Read, Update, and Delete operations (<https://en.wikipedia.org/wiki/CRUD>) may not contain enough business logic to warrant manual instrumentation. Operators of applications relying heavily on instrumented libraries may also gain enough visibility from auto-instrumentation.

However, you'll want to add additional details about your code in most scenarios. For those cases, it's crucial to combine auto-instrumentation with manual instrumentation. Such is the case for the last application in our system. The following code shows us the simplified version of the shopper service. As you can see from the code, there is still manual instrumentation code, but no configuration to be seen, as this is all managed by auto-instrumentation. Additionally, you'll note that the `get` call from the `requests` module no longer requires manual instrumentation:

shopper.py

```
#!/usr/bin/env python3
import logging
import requests
from opentelemetry import trace
from opentelemetry.sdk._logs import OTLPHandler

tracer = trace.get_tracer("shopper", "0.1.2")
logger = logging.getLogger("shopper")
logger.setLevel(logging.DEBUG)
logger.addHandler(OTLPHandler())

@tracer.start_as_current_span("add item to cart")
def add_item_to_cart(item, quantity):
    span = trace.get_current_span()
    span.set_attributes(
        {
            "item": item,
            "quantity": quantity,
        }
    )
    logger.info("add {} to cart".format(item))

@tracer.start_as_current_span("browse")
def browse():
    resp = requests.get("http://localhost:5000/products")
    add_item_to_cart("orange", 5)
```

```
@tracer.start_as_current_span("visit store")
def visit_store():
    browse()

if __name__ == "__main__":
    visit_store()
```

It's time to generate some telemetry! Open a third terminal and launch the shopper application with the following command:

```
$ OTEL_RESOURCE_ATTRIBUTES="service.name=shopper,
                           service.version=0.1.3,
                           net.host.name='hostname',
                           net.host.ip='ipconfig getifaddr en0' "
OTEL_TRACES_EXPORTER=console \
OTEL_PYTHON_TRACER_PROVIDER=sdk \
OTEL_METRICS_EXPORTER=console \
OTEL_PYTHON_METER_PROVIDER=sdk \
OTEL_LOGS_EXPORTER=console \
OTEL_PYTHON_LOG_EMITTER_PROVIDER=sdk \
opentelemetry-instrument python shopper.py
```

This command should have generated telemetry from all three applications visible in the individual terminal windows.

Important note

Since the metrics and logging signals are under active development, the instrumentation libraries we use in this chapter only support tracing. Therefore, we will focus on the tracing data being emitted for the time being. It's possible that by the time you're reading this, those libraries also emit logs and metrics.

We will not go through it in detail since the tracing data being emitted is similar to the data we've already inspected for the grocery store. Looking through the distributed trace generated, we can see the following:

- Spans generated for each application; the `service.name` and `service.version` resource attributes should reflect this.
- The trace ID has been propagated correctly across application boundaries. Check the `trace_id` field across all three terminals to confirm.
- The `Requests` and `Flask` instrumentation libraries have automatically populated attributes.

The following diagram offers a visualization of the spans generated across the system. Spans are identified as having been automatically generated (A) or manually generated (M).

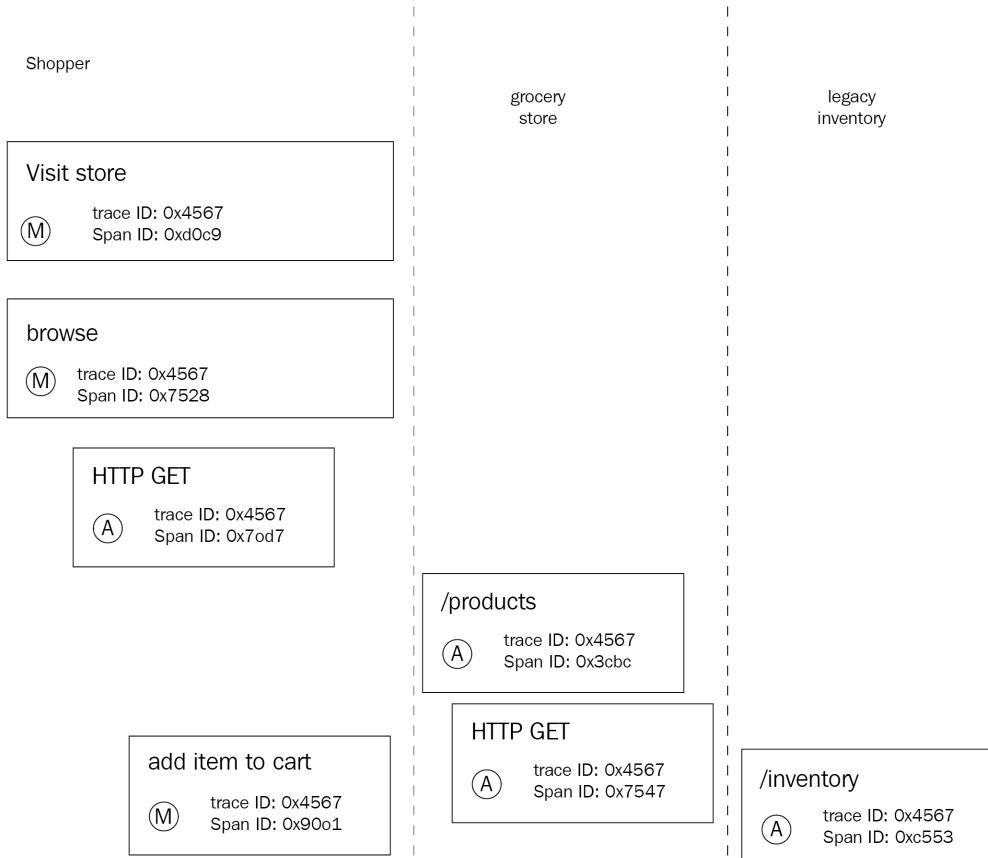


Figure 7.9 – Tracing information generated

This is one of the most exciting aspects of OpenTelemetry. We have telemetry generated by two applications that contain no instrumentation code. The developers of those applications don't need to learn about OpenTelemetry for their applications to produce information about their service, which can be helpful to diagnose issues in the future. Getting started has never been easier. Let's take a quick look at how the Flask instrumentation works.

Flask library instrumentor

Like the Requests library, the Flask instrumentation library contains an implementation of the `BaseInstrumentor` interface. The code is available in the OpenTelemetry Python contrib repository at https://github.com/open-telemetry/opentelemetry-python-contrib/blob/main/instrumentation/opentelemetry-instrumentation-flask/src/opentelemetry/instrumentation/flask/__init__.py. The implementation leverages a few different aspects of the Flask library to achieve instrumentation. It wraps the original Flask app and registers a callback via the `before_request` method. It then provides a middleware to execute instrumentation code at response time. This allows the instrumentation to capture the beginning and the end of requests through the library.

Additional configuration options

The following options are available to configure `FlaskInstrumentor` further:

- `excluded_urls`: Supports a comma-separated list of regular expressions for excluding specific URLs from producing telemetry. This option is also configurable with auto-instrumentation via the `OTEL_PYTHON_FLASK_EXCLUDED_URLS` environment variable.
- `request_hook`: A method to be executed before every Request received by the Flask application.

- `response_hook`: Similar to the `request_hook` argument, the `response_hook` allows a user to configure a method to be performed before a response is returned to the caller.

Important Note

When using the Flask instrumentation library with auto-instrumentation, it's essential to know that the debug mode may cause issues. By default, the debug mode uses a reloader, which causes the auto-instrumentation to fail. For more information on disabling the reloader, see the OpenTelemetry Python documentation: <https://opentelemetry-python.readthedocs.io/en/latest/examples/auto-instrumentation/README.html#instrumentation-while-debugging>.

The Requests and Flask instrumentation libraries are just two of many instrumentation libraries available for Python developers.

Finding instrumentation libraries

A challenge with instrumentation libraries is keeping track of which libraries are available across different languages. The libraries available for Python currently live in the `opentelemetry-collector-contrib` repository (<https://github.com/open-telemetry/opentelemetry-python-contrib>), but that may not always be the case.

OpenTelemetry registry

The official OpenTelemetry website provides a searchable registry (<https://opentelemetry.io/registry/>) that includes packages across languages. This information for this registry is stored in a GitHub repository, which can be updated via pull Requests.

opentelemetry-bootstrap

To make getting started even more accessible, the OpenTelemetry Python community maintains the `opentelemetry-bootstrap` tool, installed via the `opentelemetry-instrumentation` package. This tool looks at all installed packages in an environment and lists instrumentation libraries for that environment. It's possible to use the command also to install instrumentation libraries. The following command shows us how to use `opentelemetry-bootstrap` to list packages:

```
$ opentelemetry-bootstrap
opentelemetry-instrumentation-logging==0.28b0
opentelemetry-instrumentation-urllib==0.28b0
opentelemetry-instrumentation-wsgi==0.28b0
opentelemetry-instrumentation-flask==0.28b0
opentelemetry-instrumentation-jinja2==0.28b0
opentelemetry-instrumentation-requests==0.28b0
opentelemetry-instrumentation-urllib3==0.28b0
```

Looking through that list, there are a few additional packages that we may want to install now that we know about them. Conveniently, the `-a` `install` option installs all the listed packages.

Summary

Instrumentation libraries for third-party libraries are an excellent way for users to use OpenTelemetry with little to no effort. Additionally, instrumentation libraries don't require users to wait for third-party libraries to support OpenTelemetry directly. This helps reduce the burden on the maintainers of those third-party libraries by not asking them to support APIs, which are still evolving.

This chapter allowed us to understand how auto-instrumentation leverages instrumentation libraries to simplify the user experience of adopting OpenTelemetry. By inspecting all the components that combine to make it possible to simplify the code needed to configure telemetry pipelines, we were able to produce telemetry with little to no instrumentation code.

Revisiting the grocery store then allowed us to compare the telemetry generated by auto-instrumented code with manual instrumentation. Along the way, we took a closer look at how different instrumentations are implemented and their configurable options.

Although instrumentation libraries make it possible for users to start using OpenTelemetry today, they require the installation of another library within environments, taking on additional dependencies. As instrumentation libraries have only just started maturing, this may cause users to hesitate to adopt them. Ideally, as OpenTelemetry adoption increases and its API reaches stability across signals, third-party library maintainers will start instrumenting the libraries themselves with OpenTelemetry, removing the need for an additional library. This has already begun with some frameworks, such as Spring in Java and .NET Core libraries.

With the knowledge of OpenTelemetry signals, instrumentation libraries, and auto-instrumentation in our toolbelt, we will now focus on what to do with the telemetry data we're producing. The following few chapters will focus on collecting, transmitting, and analyzing OpenTelemetry data. First, all this data must go somewhere, and the OpenTelemetry Collector is a perfect destination. This will be the topic of the next chapter.

Section 3: Using Telemetry Data

In this part, you will learn how to deploy the OpenTelemetry Collector in conjunction with various backends to visualize the telemetry data as well as identify issues with their cloud-native applications.

This part of the book comprises the following chapters:

- *Chapter 8, OpenTelemetry Collector*
- *Chapter 9, Deploying the Collector*
- *Chapter 10, Configuring Backends*
- *Chapter 11, Diagnosing Problems*
- *Chapter 12, Sampling*

8

OpenTelemetry Collector

So, now that we've learned how to use OpenTelemetry to generate traces, metrics, and logs, we want to do something with all this telemetry data. To make the most of this data, we will need to be able to store and visualize it because, let's be honest – reading telemetry data from the console isn't going to cut it. As we'll discuss in *Chapter 10, Configuring Backends*, many destinations can be used for telemetry data. To send telemetry to a backend, the telemetry pipeline for metrics, traces, and logs needs to be configured to use an exporter that's specific to that signal and the backend. For example, if you wanted to send traces to Zipkin, metrics to Prometheus, and logs to Elasticsearch, each would need to be configured in the appropriate application code. Configuring this across dozens of services written in different languages adds to the complexity of managing the code. But now, imagine deciding that one of the backends must be changed because it no longer suits the needs of your business. Although it may not seem like a lot of work on a small scale, in a distributed system with applications that have been produced over many years by various engineers, the amount of effort to update, test, and deploy all that code could be quite significant, not to mention risky.

Wouldn't it be great if there were a way to configure an exporter once, and then use only configuration files to modify the destination of the data? There is – it's called **OpenTelemetry Collector** and this is what we'll be exploring in this chapter.

In this chapter, we will cover the following topics:

- The purpose of OpenTelemetry Collector
- Understanding the components of OpenTelemetry Collector
- Transporting telemetry via OTLP
- Using OpenTelemetry Collector

Let's start by ensuring we have all the tools in place to work with the collector.

Technical requirements

This chapter will introduce OpenTelemetry Collector as a standalone binary, which can be downloaded from https://github.com/open-telemetry/opentelemetry-collector-releases/releases/download/v0.43.0/otelcol_0.43.0_darwin_amd64.tar.gz. It's also possible to build the collector from the source, but this will not be covered in this chapter. The following commands will download the binary that's been compiled for macOS on Intel processors, extract the `otelcol` file, and ensure the binary can be executed:

```
$ wget -O otelcol.tar.gz https://github.com/open-telemetry/opentelemetry-collector-releases/releases/download/v0.43.0/otelcol_0.43.0_darwin_amd64.tar.gz
$ tar -xzf otelcol.tar.gz otelcol
$ chmod +x ./otelcol
$ ./otelcol --version
otelcol version 0.43.0
```

With the correct binary downloaded, let's ensure that the collector can start by using the following command. It is expected that the process will exit:

```
$ ./otelcol
Error: failed to get config: invalid configuration: no enabled
receivers specified in config
2022/02/13 11:52:47 collector server run finished with error:
failed to get config: invalid configuration: no enabled
receivers specified in config
```

Important Note

The OpenTelemetry Collector project produces a different binary for various operating systems (Windows, Linux, and macOS) and architectures. You must download the correct one for your environment.

The configuration for the collector is written in YAML format (<https://en.wikipedia.org/wiki/YAML>), but we'll try to steer clear of most of the traps of YAML by providing complete configuration examples. The collector is written in Go, so this chapter includes code snippets in Go. Each piece of code will be thoroughly explained, but don't worry if the details of the language escape you – the concept of the code is what we'll be focusing on. To send data to OpenTelemetry Collector from Python applications, we'll need to install the **OTLP** exporter, which can be done via pip:

```
$ pip install opentelemetry-exporter-otlp \
    opentelemetry-propagator-b3 \
    opentelemetry-instrumentation-wsgi \
    flask \
    requests
```

Important Note

The `opentelemetry-exporter-otlp` package itself does not contain any exporter code. It uses dependencies to pull in a different package for each different encoding and transport option that's supported by **OTLP**. We will discuss these later in this chapter.

The completed code and configuration for this chapter is available in this book's GitHub repository in the `chapter08` directory:

```
$ git clone https://github.com/PacktPublishing/Cloud-Native-
Observability
$ cd Cloud-Native-Observability/chapter08
```

As with the previous chapters, the code in these examples builds on top of the previous chapters. If you'd like to follow along with the code changes, copy the code from the `chapter06` folder. Now, let's dive in and figure out what this collector is all about, and why you should care about it.

The purpose of OpenTelemetry Collector

In essence, OpenTelemetry Collector is a process that receives telemetry in various formats, processes it, and then exports it to one or more destinations. The collector acts as a broker between the source of the telemetry, applications, or nodes, for example, and the backend that will ultimately store the data for analysis. The following diagram shows where the collector would be deployed in an environment containing various components:

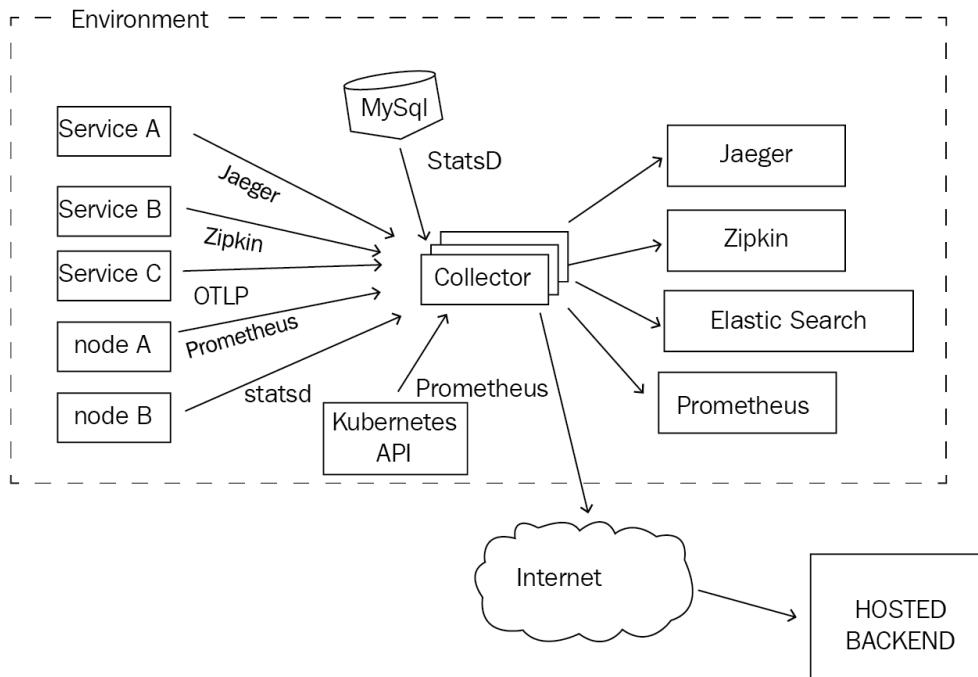


Figure 8.1 – Architecture diagram of an environment with a collector

Deploying a component such as OpenTelemetry Collector is not free as it requires additional resources to be spent on running, operating, and monitoring it. The following are some reasons why deploying a collector may be helpful:

- You can decouple the source of the telemetry data from its destination. This means that developers can configure a single destination for the telemetry data in application code and allow the operators of the collector to determine where that data will go as needed, without having to modify the existing code.
- You can provide a single destination for many data types. The collector can be configured to receive traces, metrics, and logs in many different formats, such as OTLP Jaeger, Zipkin, Prometheus, StatsD, and many more.

- You can reduce latency when sending data to a backend. This mitigates unexpected side effects from occurring when an event causes a backend to be unresponsive. A collector deployment can also be horizontally scaled to increase capacity as required.
- You can modify telemetry data to address compliance and security concerns. Data can be filtered by the collector via processors based on the criteria defined in the configuration. Doing so can stop data leakage and prevent information that shouldn't be included in the telemetry data from ever being stored in a backend.

We will discuss deployment scenarios for the collector in *Chapter 9, Deploying the Collector*. For now, let's focus on the architecture and components that provide the functionality of the collector.

Understanding the components of OpenTelemetry Collector

The collector allows users to configure *pipelines* for each signal separately by combining any number of *receivers*, *processors*, and *exporters* as shown in the following diagram. This gives the collector a lot of flexibility in how and where it can be used:

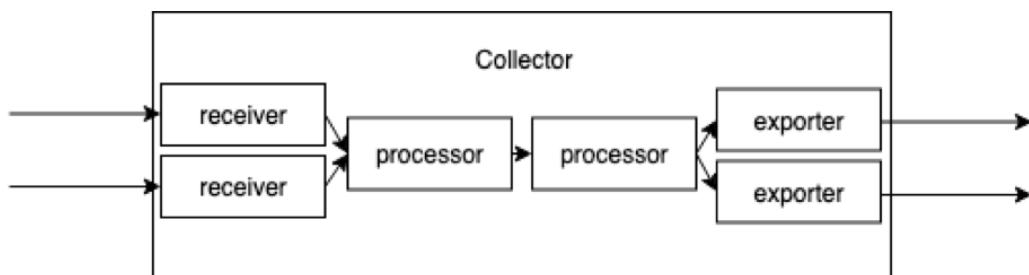


Figure 8.2 – Dataflow through the collector

The initial implementation of the collector was a fork of the OpenCensus Service (<https://opencensus.io/service/>), which served a similar purpose in the OpenCensus ecosystem. The collector supports many open protocols out of the box for inputs and outputs, which we'll explore in more detail as we take a closer look at each component. Each component in the collector implements the Component interface, which is fairly minimal, as shown in the following code:

```

type Component interface {
    Start(ctx context.Context, host Host) error
}
  
```

```
    Shutdown(ctx context.Context) error
```

```
}
```

This interface makes it easy for implementors to add additional components to the collector, making it very extensible. Let's look at each component in more detail.

Receivers

The first component in a pipeline is the **receiver**, a component that receives data in various supported formats and converts this data into an internal data format within the collector. Typically, a receiver registers a listener that exposes a port in the collector for the protocols it supports. For example, the Jaeger receiver supports the following protocols:

- Thrift Binary on port 6832
- Thrift Compact on port 6831
- Thrift HTTP on port 14268
- gRPC on port 14250

Important Note

Default port values can be overridden via configuration, as we'll see later in this chapter.

It's possible to enable multiple protocols for the same receiver so that each of the protocols listed previously will listen on different ports by default. The following table shows the supported receiver formats for each signal type:

	Traces	Metrics	Logs
Host Metrics		✓	
Jaeger	✓		
Kafka	✓	✓	✓
OpenCensus	✓	✓	
OpenTelemetry (OTLP)	✓	✓	✓
Prometheus		✓	
Zipkin	✓		

Figure 8.3 – Receiver formats per signal

Note that all the receivers shown here are receivers that support data in a specific format. However, an exception is the host metrics receiver, which will be discussed later in this chapter. Receivers can be reused across multiple pipelines and it's possible to configure multiple receivers for the same pipeline. The following configuration example enables the OTLP gRPC receiver and the Jaeger Thrift Binary receiver. Then, it configures three separate pipelines named `traces/otlp`, `traces/jaeger`, and `traces/both`, which use those receivers:

```
receivers:
  otlp:
    protocols:
      grpc:
  jaeger:
    protocols:
      thrift_binary:
service:
  pipelines:
    traces/otlp:
      receivers: [otlp]
    traces/jaeger:
      receivers: [jaeger]
    traces/both:
      receivers: [otlp, jaeger]
```

One scenario where it would be beneficial to create separate pipelines for different receivers is if additional processing needs to occur on the data from one pipeline but not the other. As with the component interface, the interface for receivers is kept minimal, as shown in the following code. The `TracesReceiver`, `MetricsReceiver`, and `LogsReceiver` receivers all embed the same `Receiver` interface, which embeds the `Component` interface we saw previously:

```
type Receiver interface {
  Component
}
type TracesReceiver interface {
  Receiver
}
```

The simplicity of the interface makes it easy to implement additional receivers as needed. As we mentioned previously, the main task of a receiver is to translate data that's being received into various formats, but what about the host metrics receiver?

Host metrics receiver

The **host metrics** receiver can be configured to collect metrics about the host running the collector. It can be configured to scrape metrics for the CPU, disk, memory, and various other system-level details. The following example shows how the `hostmetrics` receiver can be configured to scrape load, memory, and network information from a host every 10 seconds (10s):

```
receivers:  
  hostmetrics:  
    collection_interval: 10s  
    scrapers:  
      load:  
      memory:  
      network:  
    service:  
      pipelines:  
        metrics:  
          receivers: [hostmetrics]
```

The receiver supports additional configuration so that you can include or exclude specific devices or metrics. Configuring this receiver can help you monitor the performance of the host without running additional processes to do so. Once the telemetry data has been received through a receiver, it can be processed further via processors.

Processors

It can be beneficial to perform some additional tasks, such as filtering unwanted telemetry or injecting additional attributes, on the data before passing it to the exporter. This is the job of the processor. Unlike receivers and exporters, the capabilities of processors vary significantly from one processor to another. It's also worth noting that the order of the components in the configuration matters for processors, as the data is passed serially from one processor to another. In addition to embedding the component interface, the processor interface also embeds a consumer interface that matches the signal that's being processed, as shown in the following code snippet. The purpose of the consumer interface is to provide a function that consumes the signal, such as `ConsumeMetrics`. It also provides information about whether the processor will modify the data it processes via the `MutatesData` capability:

```
type Capabilities struct {
    MutatesData bool
}

type baseConsumer interface {
    Capabilities() Capabilities
}

type Metrics interface {
    baseConsumer
    ConsumeMetrics(ctx context.Context, md pdata.Metrics) error
}

type MetricsProcessor interface {
    Processor
    consumer.Metrics
}
```

The following example configures an attributes processor called `attributes/add-key` to insert an attribute with the `example-key` key and sets its value to `first`. The second attributes processor, `attributes/update-key`, updates the value of the `example-key` attribute to the `second` value. The traces pipeline is then configured to add the attribute and update its value:

```
processors:
  attributes/add-key:
    actions:
      - key: example-key
        action: insert
```

```
    value: first
  attributes/update-key:
    actions:
      - key: example-key
        action: update
        value: second
  service:
    pipelines:
      traces:
        processors: [attributes/add-key, attributes/update-key]
```

The output that's expected from this configuration is that all the spans that are emitted have an `example-key` attribute set to a value of `second`. Since the order of the processors matters, inverting the processors in the preceding example would set the value to `first`. The previous example is a bit silly since it doesn't make a lot of sense to configure multiple attributes processors in that manner, but it illustrates that ordering the processors matters. Let's see what a more realistic example may look like. The following configuration copies a value from one attribute with the `old-key` key into another one with the `new-key` key before deleting the `old-key` attribute:

```
processors:
  attributes/copy-and-delete:
    actions:
      - key: new-key
        action: upsert
        from_attribute: old-key
      - key: old-key
        action: delete
  service:
    pipelines:
      traces:
        processors: [attributes/copy-and-delete]
```

A configuration like the previous one could be used to migrate values or consolidate data coming in from multiple systems, where different names are used to represent the same data. As we mentioned earlier, processors cover a range of functionality. The following table lists the current processors, as well as the signals they process:

	Traces	Metrics	Logs
Attributes	✓		✓
Batch	✓	✓	✓
Filter		✓	
Memory Limiter	✓	✓	✓
Probabilistic Sampling	✓		
Resource	✓	✓	✓
Span	✓		

Figure 8.4 – Processors per signal

Some of these processors will be familiar to you if you've already used an OpenTelemetry SDK. It's worth taking a moment to explore these processors further.

Attributes processor

As we discussed earlier, the **attributes processor** can be used to modify telemetry data attributes. It supports the following operations:

- `delete`: This deletes an attribute for a specified key.
- `extract`: This uses a regular expression to extract values from the specified attribute and `upsert` new attributes resulting from that extraction.
- `hash`: This computes a SHA-1 hash of the value for an existing attribute and updates the value of that attribute to the computed hash.
- `insert`: This inserts an attribute for a specified key when it does not exist. It does nothing if the attribute exists.
- `update`: This updates an existing attribute with a specified value. It does nothing if the attribute does not exist.
- `upsert`: This combines the functionality of `insert` and `update`. If an attribute does not exist, it will insert it with the specified value; otherwise, it will update the attribute with the value.

The attributes processor, along with the span processor, which we'll see shortly, allows you to include or exclude spans based on `match_type`, which can either be an exact match configured as `strict` or a regular expression configured with `regexp`. The matching is applied to one or more of the configured fields: `services`, `span_names`, or `attributes`. The following example includes spans for the `super-secret` and `secret` services:

```
processors:  
  attributes/include-secret:  
    include:  
      match_type: strict  
      services: ["super-secret", "secret"]  
    actions:  
      - key: secret-attr  
        action: delete
```

The attributes processor can be quite useful when you're scrubbing **personally identifiable information (PII)** or other sensitive information. A common way sensitive information makes its way into telemetry data is via debug logs that capture private variables it shouldn't have, or by user information, passwords, or private keys being recorded in metadata. Data leaks often happen accidentally and are much more frequent than you'd think.

Important Note

It's possible to configure both an `include` and `exclude` rule at the same time. If that is the case, `include` is checked before `exclude`.

Filter processor

The **filter processor** allows you to include or exclude telemetry data based on the configured criteria. This processor, like the attributes and span processors, can be configured to match names with either `strict` or `regexp` matching. It's also possible to use an expression that matches attributes as well as names. Further scoping on the filter can be achieved by specifying `resource_attributes`. In terms of its implementation, at the time of writing, the filter processor only supports filtering for metrics, though additional signal support has been requested by the community.

Probabilistic sampling processor

Although sampling is a topic that we'll cover in more detail in *Chapter 12, Sampling*, it's important to know that the collector provides a sampling processor for traces known as the **probabilistic sampling processor**. It can be used to reduce the number of traces that are exported from the collector by specifying a sampling percentage, which determines what percentage of traces should be kept. The `hash_seed` parameter is used to determine how the collector should hash the trace IDs to determine which traces to process:

```
processors:  
  probabilistic_sampler:  
    sampling_percentage: 20  
    hash_seed: 12345
```

The `hash_seed` configuration parameter becomes especially important when multiple collectors are connected. For example, imagine that a collector (*A*) has been configured to send its data to another collector (*B*) before sending the data to a backend. With both *A* and *B* configured using the previous example, if 100 traces are sent through the two collectors, a total of 20 of those will be sent through to the backend. If, on the other hand, the two collectors use a different `hash_seed`, collector *A* will send 20 traces to collector *B*, and collector *B* will sample 20% of those, resulting in four traces being sent to the backend. Either case is valid, but it's important to understand the difference.

Important Note

The probabilistic sampling processor prioritizes the sampling priority attribute before the trace ID hashing if the attribute is present. This attribute is defined in the semantic conventions and was originally defined in OpenTracing. More information on this will be provided in *Chapter 12, Sampling*, but for now, it's just good to be aware of it.

Resource processor

The **resource processor** lets users modify attributes, just like the attributes processor. However, instead of updating attributes on individual spans, metrics, or logs, the resource processor updates the attributes of the resource associated with the telemetry data. The options that are available for configuring the resource processor are the same as for the attributes processor. This can be seen in the following example, which uses `upsert` for the `deployment.environment` attribute and renames the `runtime` attribute to `container.runtime` using the `insert` and `delete` actions:

```
processors:  
  resource:  
    attributes:  
      - key: deployment.environment  
        value: staging  
        action: upsert  
      - key: container.runtime  
        from_attribute: runtime  
        action: insert  
      - key: runtime  
        action: delete
```

Now, let's discuss the span processor.

Span processor

It may be useful to manipulate the names of spans or attributes of spans based on their names. This is the job of the **span processor**. It can extract attributes from a span and update its name based on those attributes. Alternatively, it can take the span's name and expand it to individual attributes associated with the span. The following example shows how to rename a span based on the `messaging.system` and `messaging.operation` attributes, which will be separated by the `:` character. The second configuration of the span processor shows how to extract the `storeId` and `orderId` attributes from the span's name:

```
processors:  
  span/ rename:  
    name:  
      from_attributes: ["messaging.system", "messaging.operation"]  
      separator: ":"
```

```
span/create-attributes:  
  name:  
  to_attributes:  
  rules:  
    - ^\\/stores\\/(?P<storeId>.*\\/.*$  
    - ^.*\\/orders\\/(?P<orderId>.*\\/.*$
```

As we mentioned previously, the span processor also supports the `include` and `exclude` configurations to help you filter spans. Not all processors are used to modify the telemetry data; some change the behavior of the collector itself.

Batch processor

The **batch processor** helps you batch data to increase the efficiency of transmitting the data. It can be configured both to send batches based on batch size and a schedule. The following code configures a batch processor to send data every 10s or every 10000 records and limits the size of the batch to 11000 records:

```
processors:  
  batch:  
    timeout: 10s # default 200ms  
    send_batch_size: 10000 # default 8192  
    send_batch_max_size: 11000 # default 0 - no limit
```

It is recommended to configure a batch processor for all the pipelines to optimize the throughput of the collector.

Memory limiter processor

To ensure the collector is conscious of resource consumption, the **memory limiter processor** lets users control the amount of memory the collector consumes. This helps ensure the collector does as much as it can to avoid running out of memory. Limits can be specified either via fixed mebibyte values or percentages that are calculated based on the total available memory. If both are specified, the fixed values take precedence. The memory limiter enforces both soft and hard limits, with the difference defined by a spike limit configuration. It is recommended to use the **ballast extension** alongside the memory limiter. The ballast extension allows the collector to pre-allocate memory to improve the stability of the heap. The recommended size for the ballast is between one-third to one-half of the total memory for the collector. The following code configures the memory limiter to use up to 250 Mib of the memory configured via `limit_mib`, with 50 Mib as the difference between the soft and hard limits, which is configured via `spike_limit_mib`:

```
processors:  
  memory_limiter:  
    check_interval: 5s  
    limit_mib: 250  
    spike_limit_mib: 50  
  extensions:  
    memory_ballast:  
      size_mib: 125
```

The memory limiter processor, along with the batch processor, are both recommended if you wish to optimize the performance of the collector.

Important Note

When the processor exceeds soft limits, it returns errors and starts dropping data. If it exceeds hard limits, it will also force garbage collection to free memory.

The memory limiter should be the first processor you configure in the pipeline. This ensures that when the memory threshold is exceeded, the errors that are returned are propagated to the receivers. This allows the receivers to send appropriate error codes back to the client, who can then throttle the requests they are sending. Now that we understand how to process our telemetry data to fit our needs, let's learn how to use the collector to export all this data.

Exporters

The last component of the pipeline is the exporter. The role of the exporter in the collector pipeline is fairly similar to its role in the SDK, as we explored in previous chapters. The exporter takes the data in its internal collector format, marshals it into the output format, and sends it to one or more configured destinations. The interface for the exporter is very similar to the processor interface as it is also a consumer, separated again by a signal. The following code shows us the `LogsExporter` interface, which embeds the interfaces we explored earlier:

```
type LogsExporter interface {
    Exporter
    consumer.Logs
}
```

Multiple exporters of the same type can be configured for different destinations as necessary. It's also possible to configure multiple exporters for the same pipeline to output the data to multiple locations. The following code configures a `jaeger` exporter, which is used for exporting traces, and an `otlp` exporter, which will be used for both traces and metrics:

```
exporters:
  jaeger:
    endpoint: jaeger:14250
  otlp:
    endpoint: otelcol:4317
service:
  pipelines:
    traces:
      exporters: [jaeger, otlp]
    metrics:
      exporters: [otlp]
```

Several other formats are supported by exporters. The following table lists the available exporters, as well as the signals that each supports:

	Traces	Metrics	Logs
File	✓	✓	✓
Jaeger	✓		
Kafka	✓	✓	✓
Logging	✓	✓	✓
OpenCensus	✓	✓	
OpenTelemetry (OTLP)	✓	✓	✓
Prometheus		✓	
Zipkin	✓		

Figure 8.5 – Exporters per signal

Note that in addition to exporting data across different signals to destinations that can be reached over a network, it's also possible to export telemetry data locally to the console via the logs exporter or as JSON to a file via the file exporter. Receivers, processors, and exporters cover the components in the pipeline, but there is yet more to cover about the collector.

Extensions

Although most of the functionality of the collector revolves around the telemetry pipelines, there is additional functionality that is made available via **extensions**. Extensions provide you with another way to extend the collector. The following extensions are currently available:

- **ballast**: This allows users to configure a memory ballast for the collector to improve the overall stability and performance of the collector.
- **health_check**: This makes an endpoint available for checking the health of the collector. This can be useful for service discovery or orchestration of the collector.
- **pprof**: This enables the Go performance profiler, which can be used to identify performance issues within the collector.
- **zpages**: This enables an endpoint in the collector that provides debugging information about the components in the collector.

Thus far, all the components we've explored are part of the core collector distribution and are built into the binary we'll be using in our examples later in this chapter. However, those are far from the only components that are available.

Additional components

As you can imagine, providing this much functionality in an application can become quite complex. To reduce the complexity of the collector's core functionality without impeding progress and enthusiasm in the community, the main collector repository contains components that are defined as part of the OpenTelemetry specification.

With all the flexibility the collector provides, many individuals and organizations are contributing additional receivers, processors, and exporters. These can be found in the `opentelemetry-collector-contrib` repository at <https://github.com/open-telemetry/opentelemetry-collector-contrib>. As the code in this repository is changing rapidly, we won't be going over the components available there, but I strongly suggest browsing through the repository to get an idea of what is available.

Before learning how to use the collector and configuring an application to send data to it, it's important to understand a little bit more about the preferred protocol to receive and export data via the collector. This is known as OTLP.

Transporting telemetry via OTLP

We've mentioned OTLP multiple times in this chapter and this book, so let's look at what it is. To ensure that telemetry data is transmitted as efficiently and reliably as possible, OpenTelemetry has defined OTLP. The protocol itself is defined via protocol buffer (<https://developers.google.com/protocol-buffers>) definition files.

This means that any client or server that's interested in sending or receiving OTLP only has to implement these definitions to support it. OTLP is the recommended protocol of OpenTelemetry for transmitting telemetry data and is supported as a core component of the collector.

Important Note

Protocol buffers or **protobufs** are a language and platform-agnostic mechanism for serializing data that was originally intended for gRPC. Libraries are provided to generate the code from the protobuf definition files in a variety of languages. This is a much deeper topic than we will have time for in this book, so if you're interested in reading the protocol files, I strongly recommended learning more about protocol buffers – they're pretty cool! The Google developer site that was linked previously is a great resource to get started.

The definition for OTLP (<https://github.com/open-telemetry/opentelemetry-proto>) is divided into multiple sections to cover the different signals. Each component of the protocol provides backward compatibility guaranteed via its maturity level, which allows adopters to get a sense of how often they should expect breaking changes. An alpha level makes no guarantees around breaking changes while a stable level guarantees backward-incompatible changes will be introduced no more frequently than every 12 months. The maturity level of each component is available in the project's `README.md` file and the current state, at the time of writing, can be seen in the following table. It's very likely to change by the time you're reading this as progress is being made quite rapidly!

Component	Maturity
Binary Protobuf Encoding	
collector/metrics/*	Stable
collector/trace/*	Stable
collector/logs/*	Beta
common/*	Stable
metrics/*	Stable
resource/*	Stable
trace/trace.proto	Stable
trace/trace_config.proto	Alpha
logs/*	Beta
JSON encoding	
All messages	Alpha

Figure 8.6 – Maturity level of OTLP components

Taking a closer look at the preceding table (<https://github.com/open-telemetry/opentelemetry-proto#maturity-level>), note that it includes a different section for protobuf and JSON encoding. Let's talk about why that is.

Encodings and protocols

The specification defines the encodings and protocols that are supported by OTLP. Initially, the following three combinations are supported:

- protobufs over gRPC
- protobufs over HTTP
- JSON over HTTP

Depending on the requirements of your application or the infrastructure that will be used to deploy your code, certain restrictions may guide the decision of which encoding or protocol to choose. For example, users may be deploying applications in an environment that doesn't support gRPC. This was true for a long time with serverless Python environments in various cloud providers. Similarly, gRPC was not supported in the browser, meaning users of OpenTelemetry for JavaScript cannot use gRPC when instrumenting a browser application. Another tradeoff that may cause users to choose one package over another is the impact of serializing and deserializing data using JSON, which can have some serious performance implications in certain languages compared to using protobufs. The different combinations of encodings and protocols exist to provide additional flexibility for users, depending on the requirements of their environments.

One of the requirements for any OpenTelemetry language implementation is to support at least one of these formats before marking a signal as generally available. This ensures that users can use OTLP to export data across their entire system, from application instrumentation to the backend.

Additional design considerations

Backpressure can happen when clients are generating telemetry data faster than the recipients can receive it. To address this, the specification for OTLP also defines how clients should handle responses from servers to manage backpressure when receiving systems become overloaded. Another design goal of the protocol is to ensure it is load balancer-friendly so that you can horizontally scale various components that could be involved in handling telemetry data using OTLP. Equipped with this knowledge of the protocol, let's start sending data to the collector.

Using OpenTelemetry Collector

Now that we're familiar with the core components of OpenTelemetry Collector and OTLP, it's time to start using the collector with the grocery store. The following diagram gives us an idea of how telemetry data is currently configured and where we are trying to go with this chapter:

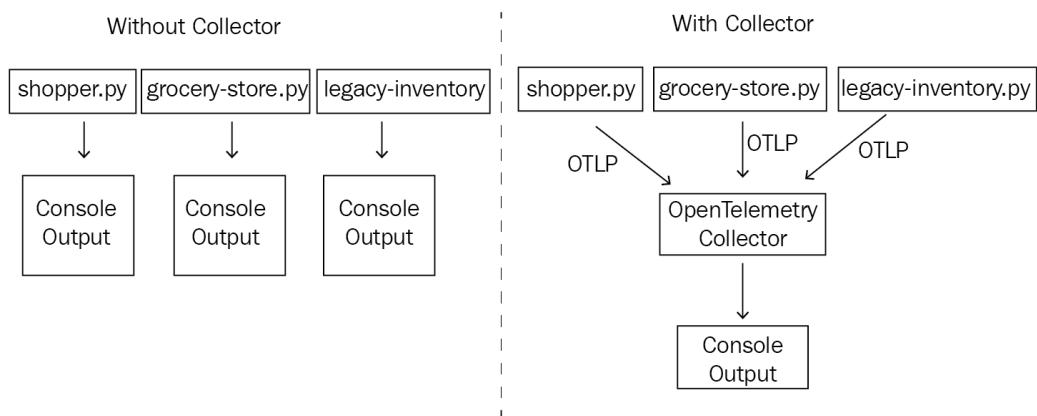


Figure 8.7 – Before and after diagrams of exporting telemetry data

At the beginning of this chapter, we installed the OTLP exporters for Python via the `opentelemetry-exporter-otlp` package. This, in turn, installed the packages that are available for each protocol and encoding:

- `opentelemetry-exporter-otlp-proto-grpc`
- `opentelemetry-exporter-otlp-proto-http`
- `opentelemetry-exporter-otlp-json-http`

The package that includes all the protocols and the encoding is a convenient way to start, but once you're familiar with the requirements for your environment, you'll want to choose a specific encoding and protocol to reduce dependencies.

Configuring the exporter

The following examples will leverage the `otlp-proto-grpc` package, which includes the exporter classes we'll use to export telemetry – `OTLPSpanExporter`, `OTLPMetricExporter`, and `OTLPLogExporter`. The code builds on the example applications from *Chapter 6, Logging — Capturing Events*, by updating the `common.py` module to use the OTLP exporters instead of the control exporters, which we've used so far:

common.py

```
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter
import OTLPSpanExporter

from opentelemetry.exporter.otlp.proto.grpc._metric_exporter
import OTLPMetricExporter

from opentelemetry.exporter.otlp.proto.grpc._log_exporter
import OTLPLogExporter

def configure_tracer(name, version):
    ...
    exporter = OTLPSpanExporter()
    ...

def configure_meter(name, version):
    ...
    exporter = OTLPMetricExporter()
    ...

def configure_logger(name, version):
    ...
    exporter = OTLPLogExporter()
    ...
```

By default, as per the specification, the exporters will be configured to send data to a collector running on `localhost:4317`.

Configuring the collector

The following collector configuration sets up the `otlp` receiver, which will be used to receive telemetry data from our application. Additionally, it configures the logging exporter to output useful information to the console:

config/collector/config.yml

```
receivers:  
  otlp:  
    protocols:  
      grpc:  
exporters:  
  logging:  
service:  
  pipelines:  
    traces:  
      receivers: [otlp]  
      exporters: [logging]  
    metrics:  
      receivers: [otlp]  
      exporters: [logging]  
    logs:  
      receivers: [otlp]  
      exporters: [logging]
```

Important Note

In the following examples, each time `config.yml` is updated, the collector must be restarted for the changes to take effect.

It's time to see whether the collector and the application can communicate. First, start the collector using the following command from the terminal:

```
./otelcol --config ./config/collector/config.yml
```

If everything is going according to plan, the process should be up and running, and the output from it should list the components that have been loaded. It should also contain a message similar to the following:

collector output

```
2021-05-30T16:19:03.088-0700 info service/application.go:197
Everything is ready. Begin running and processing data.
```

Next, we need to run the application code in a separate terminal. First, launch the legacy inventory, followed by the grocery store, and then the shopper application. Note that `legacy_inventory.py` and `grocery_store.py` will remain running for the remainder of this chapter as we will not make any further changes to them:

```
python legacy_inventory.py
python grocery_store.py
python shopper.py
```

Pay close attention to the output from the terminal running the collector. You should see some output describing the traces, metrics, and logs that have been processed by the collector. The following code gives you an idea of what to look for:

collector output

```
2022-02-13T14:35:47.101-0800      INFO    loggingexporter/
logging_exporter.go:69  LogsExporter
                                {"#logs": 1}
2022-02-13T14:35:47.110-0800      INFO    loggingexporter/
logging_exporter.go:40  TracesExporter
                                {"#spans": 4}
2022-02-13T14:35:49.858-0800      INFO    loggingexporter/
logging_exporter.go:40  TracesExporter
                                {"#spans": 1}
2022-02-13T14:35:50.533-0800      INFO    loggingexporter/
logging_exporter.go:40  TracesExporter
                                {"#spans": 3}
2022-02-13T14:35:50.535-0800      INFO    loggingexporter/
logging_exporter.go:69  LogsExporter
                                {"#logs": 2}
```

Excellent – let's do some more fun things with the collector by adding some processors to our configuration! If you look closely at the preceding output, you'll notice that `TracesExporter` is mentioned in three separate instances. Since each of our applications is sending telemetry data, the exporter is called with the new data. The batch processor can improve its efficiency here by waiting a while and sending a single batch containing all the telemetry data simultaneously. The following code configures the batch processor with a timeout of 10 seconds (10s), so the processor will wait up until that time to send a batch. Then, we can add this processor to each pipeline:

config/collector/config.yml

```
processors:  
  batch:  
    timeout: 10s  
...  
pipelines:  
  traces:  
    receivers: [otlp]  
    processors: [batch]  
    exporters: [logging]  
  metrics:  
    receivers: [otlp]  
    processors: [batch]  
    exporters: [logging]  
  logs:  
    receivers: [otlp]  
    processors: [batch]  
    exporters: [logging]
```

Try running the shopper application once again. This time, the output from the collector should show a single line including the sum of all the spans we saw earlier:

collector output

```
2022-02-13T14:40:07.360-0800      INFO      loggingexporter/  
logging_exporter.go:69  LogsExporter  {"#logs": 2}  
2022-02-13T14:40:07.360-0800      INFO      loggingexporter/  
logging_exporter.go:40  TracesExporter {"#spans": 8}
```

If you run the shopper application a few times, you'll notice a 10-second delay in the collector outputting information about the telemetry data that's been generated. This is the batch processor at work. Let's make the logging output slightly more useful by updating the logging exporter configuration:

config/collector/config.yml

```
exporters:  
  logging:  
    loglevel: debug
```

Restarting the collector and running the shopper application again will output the full telemetry data that's been received. What should appear is a verbose list of all the telemetry data the collector is receiving. Look specifically for the span named `add item to cart` as we'll be modifying it in the next few examples:

collector output

```
Span #0  
  Trace ID      : 1592a37b7513b73eaefabde700f4ae9b  
  Parent ID     : 2411c263df768eb5  
  ID            : 8e6f5cdb56d6448d  
  Name          : HTTP GET  
  Kind          : SPAN_KIND_SERVER  
  Start time    : 2022-02-13 22:41:42.673298 +0000 UTC  
  End time      : 2022-02-13 22:41:42.677336 +0000 UTC  
  Status code   : STATUS_CODE_UNSET  
  Status message :  
  
  Attributes:  
    -> http.method: STRING(GET)  
    -> http.server_name: STRING(127.0.0.1)  
    -> http.scheme: STRING(http)  
    -> net.host.port: INT(5000)  
    -> http.host: STRING(localhost:5000)  
    -> http.target: STRING(/products)  
    -> net.peer.ip: STRING(127.0.0.1)
```

So far, our telemetry data is being emitted to a collector from three different applications. Now, we can see all the telemetry data on the terminal running the collector. Let's take this a step further and modify this telemetry data via some processors.

Modifying spans

One of the great features of the collector is its ability to operate on telemetry data from a central location. The following example demonstrates some of the power behind the processors. The following configuration uses two different processors to augment the span we mentioned previously. First, the attributes processor will add an attribute to identify a location attribute. Next, the span processor will use the attributes from the span to rename the span so that it includes the location, item, and quantity attributes. The new processors must also be added to the traces pipeline's processors array:

config/collector/config.yml

```
processors:  
  attributes/add-location:  
    actions:  
      - key: location  
        action: insert  
        value: europe  
  span/rename:  
    name:  
      from_attributes: [location, item, quantity]  
      separator: ":"  
...  
pipelines:  
  traces:  
    processors: [batch, attributes/add-location, span/rename]
```

Important Note

Remember that the order of the processors matters. In this case, the reverse order wouldn't work as the location attribute would not be populated.

Run the shopper and look at the output from the collector to see the effect of the new processors. The new exported span contains a `location` attribute with the `europe` value, which we configured. Its name has also been updated to `location:item:quantity`:

collector output

```
Span #1
  Trace ID      : 47dac26efa8de0cale202b6d64fd319c
  Parent ID     : ee10984575037d4a
  ID            : a4f42124645c4d3b
  Name          : europe:orange:5
  Kind          : SPAN_KIND_INTERNAL
  Start time    : 2022-02-13 22:44:57.072143 +0000 UTC
  End time      : 2022-02-13 22:44:57.07751 +0000 UTC
  Status code   : STATUS_CODE_UNSET
  Status message :
Attributes:
  -> item: STRING(orange)
  -> quantity: INT(5)
  -> location: STRING(europe)
```

This isn't bad for 10 lines of configuration! The final example will explore the `hostmetrics` receiver and how to configure the `filter` processor for metrics.

Filtering metrics

So far, we've looked at how to modify spans, but what about metrics? As we discussed previously, the `hostmetrics` receiver captures metrics about the localhost. Let's see it in action. The following example configures the host metrics receiver to scrape memory and network information every 10 seconds:

config/collector/config.yml

```
receivers:
  hostmetrics:
    collection_intervals: 10s
scrapers:
  memory:
```

```
network:  
...  
service:  
pipelines:  
metrics:  
receivers: [otlp, hostmetrics]
```

After configuring this receiver, just restart the collector – you should see metrics in the collector output, without running `shopper.py`. The output will include memory and network metrics:

collector output

```
InstrumentationLibraryMetrics #0  
InstrumentationLibrary  
Metric #0  
Descriptor:  
    -> Name: system.memory.usage  
    -> Description: Bytes of memory in use.  
    -> Unit: By  
    -> DataType: IntSum  
    -> IsMonotonic: false  
    -> AggregationTemporality: AGGREGATION_TEMPORALITY_  
CUMULATIVE  
IntDataPoints #0  
Data point labels:  
    -> state: used  
StartTimestamp: 1970-01-01 00:00:00 +0000 UTC  
Timestamp: 2022-02-13 22:48:16.999087 +0000 UTC  
Value: 10880851968  
Metric #1  
Descriptor:  
    -> Name: system.network.packets  
    -> Description: The number of packets transferred.  
    -> Unit: {packets}  
    -> DataType: IntSum  
    -> IsMonotonic: true
```

```
-> AggregationTemporality: AGGREGATION_TEMPORALITY_CUMULATIVE
IntDataPoints #0
Data point labels:
-> device: lo0
-> direction: transmit
StartTimestamp: 1970-01-01 00:00:00 +0000 UTC
Timestamp: 2022-02-13 22:48:16.999087 +0000 UTC
Value: 120456
```

Well done – the collector is now generating metrics for you! Depending on the type of system you're running the collector on, you may have many network interfaces available that are generating a lot of metrics. Let's update the configuration to scrape metrics for a single interface to reduce some of the noise. On my host, I will use `lo0` as the interface:

config/collector/config.yml

```
receivers:
hostmetrics:
collection_intervals: 10s
scrapers:
memory:
network:
include:
match_type: strict
interfaces: [lo0]
```

Important Note

Network interface names vary based on the operating system being used. Some common interface names are `lo0`, `eth0`, `en0`, and `wlan0`. If you're unsure, look for the device label in the previous output, which should show you some of the interfaces that are available on your system.

The output will be significantly reduced, but there are still many network metrics to sift through. `system.network.connections` is quite noisy as it collects data points for each `tcp` state. Let's take this one step further and use the `filter` processor to exclude `system.network.connections`:

config/collector/config.yml

```
processors:  
  filter/network-connections:  
    metrics:  
      exclude:  
        match_type: strict  
        metric_names:  
          - system.network.connections  
...  
pipelines:  
  metrics:  
    receivers: [hostmetrics]  
    processors: [batch, filter/network-connections]
```

Restarting the collector one last time will yield a much easier-to-read output. Of course, there are many more scenarios to experiment with when it comes to the collector and its components, but this gives you a good idea of how to get started. I recommend spending some time experimenting with different configurations and processors to get comfortable with it. And with that, we now have an understanding of one of the most critical components of OpenTelemetry – the collector.

Summary

In this chapter, you learned about the fundamentals of OpenTelemetry Collector and its components. You now know what role receivers, processors, exporters, and extensions play in the collector and know about the specifics of individual processors.

Additionally, we looked at the definition of the OTLP, its benefits, and the design decisions behind creating the protocol. Equipped with this knowledge, we configured OpenTelemetry Collector for the first time and updated the grocery store to emit data to it. Using a variety of processors, we manipulated the data the collector was receiving to get a working understanding of how to harness the power of the collector.

The next chapter will expand on this knowledge and take the collector from a component that's used in development to a core component of your infrastructure. We'll explore how to deploy the collector in a variety of scenarios to make the most of it.

9

Deploying the Collector

Now that we've learned about the ins and outs of the collector, it's time to look at how we can use it in production. This chapter will explain how the flexibility of the collector can help us to deploy it in a variety of scenarios. Using Docker, Kubernetes, and Helm, we will learn how to use the OpenTelemetry collector in combination with the grocery store application from earlier chapters. This will give us the necessary knowledge to start using the collector in our cloud-native environment.

In this chapter, we will focus on the following main topics:

- Using the collector as a sidecar to collect application telemetry
- Deploying the collector as an agent to collect system-level telemetry
- Configuring the collector as a gateway

Along the way, we'll look at some strategies for scaling the collector. Additionally, we'll spend some more time with the processors that we looked at in *Chapter 8, OpenTelemetry Collector*. Unlike the previous chapters, which focused on OpenTelemetry components, this chapter is all about using them. As such, it will introduce a number of tools that you might encounter when working with cloud-native infrastructure.

Technical requirements

This chapter will cover a few different tools that we can use to deploy the collector. We will be using containers to run the sample application and collector; all the examples are available from the public Docker container registry (<https://hub.docker.com>). Although we won't dive too deeply into what containers are, just know that containers provide a convenient way to build, package, and deploy self-contained applications that are immutable. For us to run containers locally, we will use Docker, just as we did in *Chapter 2, OpenTelemetry Signals - Traces, Metrics and Logs*. The following is a list of the technical requirements for this chapter:

- If you don't already have Docker installed on your machine, follow the instructions available at <https://docs.docker.com/get-docker/> to get started on Windows, macOS, and Linux. Once you have it installed, run the following command from a Terminal. If everything is working correctly, there should be no errors reported:

```
$ docker ps
CONTAINER ID        IMAGE          COMMAND       CREATED      STATUS
PORTS              NAMES
```

- Shortly, we will be required to run a command using the `kubectl` Kubernetes command-line tool. This tool interacts with the Kubernetes API, which we'll do continuously throughout the chapter to access our applications once they're running in the cluster. Depending on your environment, you might already have a copy of this tool installed. Check whether that is the case by running the following command:

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"17",
GitVersion:"v1.17.0"...
Server Version: version.Info{Major:"1", Minor:"19",
GitVersion:"v1.19.7"...
```

If the output from running the previous command shows command not found, go through the installation steps documented on the Kubernetes website at <https://kubernetes.io/docs/tasks/tools/>.

- In addition to Docker, we will also use Kubernetes (<https://kubernetes.io>) throughout this chapter. This is because it is one of the leading open source tools used in cloud-native infrastructure. Kubernetes will provide the container orchestration for our examples and the collector. It's worth noting that Kubernetes is not the only container orchestration solution that is available; however, it is one of the more popular ones. There are many different tools available to set up a local Kubernetes cluster. For instance, I'll use **kind** to set up my cluster, which runs a local cluster inside Docker. If you already have access to a cluster, then great! You're good to go. Otherwise, head over to <https://kind.sigs.k8s.io/docs/user/quick-start/> and follow the installation instructions for your platform. Once kind is installed, run the following command to start a cluster:

```
$ kind create cluster
Creating cluster "kind" ...
  ✓ Ensuring node image (kindest/node:v1.19.1) 
  ✓ Preparing nodes 
  ✓ Writing configuration 
  ✓ Starting control-plane 
  ✓ Installing CNI 
  ✓ Installing StorageClass 
Set kubectl context to "kind-kind"
You can now use your cluster with:
kubectl cluster-info --context kind-kind
```

The previous command should get the cluster started for you. Getting a cluster up and running is crucial to use the examples in the rest of this chapter. If you're running into issues while setting up a local cluster with kind, you might want to investigate one of the following alternatives:

- A. Minikube: <https://minikube.sigs.k8s.io/docs/start/>
- B. K3s: <https://k3s.io>
- C. Docker Desktop: <https://docs.docker.com/desktop/kubernetes/>

How the cluster is run isn't going to be important; having a cluster is what really matters. Additionally, if running a local cluster isn't feasible, you might want to look at some hosted options:

- A. Google Kubernetes Engine: <https://cloud.google.com/kubernetes-engine>
- B. Amazon Elastic Kubernetes Service: <https://aws.amazon.com/eks/>
- C. Azure Kubernetes Service: <https://azure.microsoft.com/en-us/services/kubernetes-service/>

You should know that there are always costs associated with using a hosted Kubernetes cluster.

- Now, check the state of the cluster using `kubectl`, which we installed earlier. Run the following command to check whether the cluster is ready:

```
$kubectl cluster-info --context kind-kind
Kubernetes master is running at https://127.0.0.1:62708
KubeDNS is running at https://127.0.0.1:62708/api/v1/
namespaces/kube-system/services/kube-dns:dns/proxy
```

- Good job at getting this far! I know there are a lot of tools to install, but it'll be worth it! The last tool that we'll use throughout this chapter is *Helm*. This is a package manager for applications running in Kubernetes. Helm will allow us to install applications in our cluster by using the YAML configuration it calls **charts**; these provide the default configuration for many applications that are available to deploy in Kubernetes. The instructions for installing Helm are available from the Helm website at <https://helm.sh/docs/intro/install/>. Once again, to ensure the tool is working and correctly configured in your path, run the following command:

```
helm version
```

The full configuration for all the examples in this chapter is available in the companion repository at <https://github.com/PacktPublishing/Cloud-Native-Observability>. Please feel free to look in the `chapter9` folder if any of the examples give you trouble. Great! Now that the hard part is done, let's get to the fun stuff and start deploying OpenTelemetry collectors in our cluster!

Collecting application telemetry

Previously, we looked at how to use the collector running as a local binary. This can be useful for development and testing, but it's not how the collector would be deployed in a production environment. Before going further, here are some Kubernetes concepts that we will be using in this chapter:

- **Pod:** This is a container or a group of containers that form an application.
- **Sidecar:** This is a container that is deployed alongside application containers but isn't tightly coupled with the application in the pod.
- **Node:** This is a representation of a Kubernetes worker; it could be a physical host or a virtual machine.
- **DaemonSet:** This is a pod template specification to ensure a pod is deployed to the configured nodes.

Important Note

The concepts of Kubernetes form a much deeper topic than we have time for in this book. For our examples, we will only cover the bare minimum that is necessary for this chapter. There is a lot more to cover and, thankfully, many resources are available on the internet regarding this vast topic.

Figure 9.1 shows three different deployment scenarios that can be used to deploy the OpenTelemetry collector in a production environment, which, in this case, is a Kubernetes cluster:

- The first deployment (1) is alongside the application containers within the same pod. This deployment is commonly referred to as a **sidecar** deployment.
- The second deployment (2) shows the collector running as a container on the same node as the application pod. This **agent** deployment represents a DaemonSet deployment, which means that the collector container will be present in every node in the Kubernetes cluster.
- The third deployment (3) is shown running the collector as a gateway. In practice, the containers in the collector service will run on Kubernetes nodes, which may or may not be the same as the ones running the application pod.

Additionally, the following diagram shows the flow for the telemetry data from one collector to another, which we will configure in this chapter:

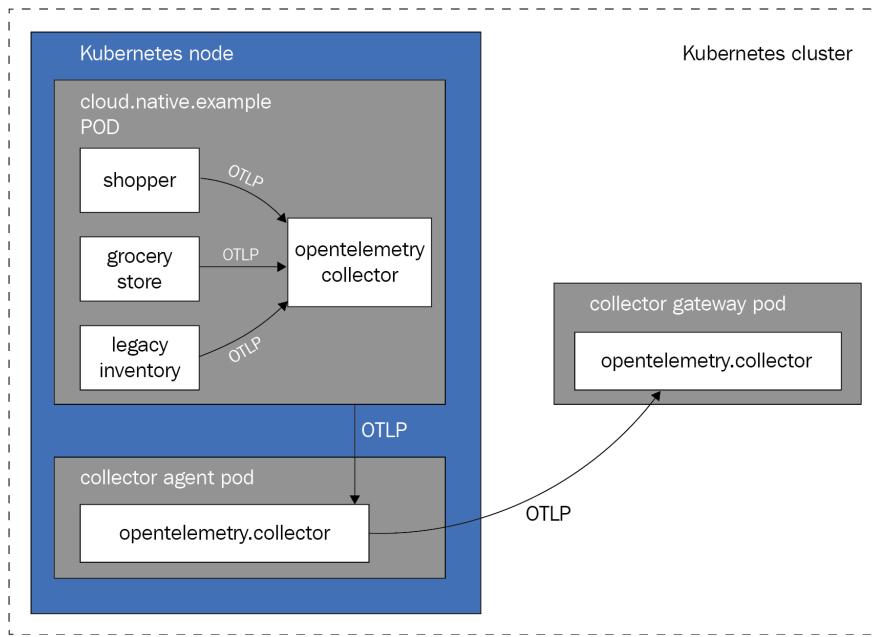


Figure 9.1 – The three deployment options for the collector

In this chapter, we will work through each scenario, starting with collecting application telemetry. We can do this by deploying the collector as close to the application as possible within the same pod. When emitting telemetry from an application, often, it's useful to offload the data as quickly as possible to reduce the resource impact on the application. This allows the application to spend most of its time on what it is meant to do, that is, manage the workloads it was created to manage. To ensure the lowest possible latency while transmitting telemetry, let's look at deploying the collector as close as possible to the application, as a sidecar.

Deploying the sidecar

To reduce that latency and the complexity of collecting telemetry, deploying the collector as a loosely coupled container within the same pod as the application makes the most sense. This ensures the following:

- The application will always have a consistent destination to send its telemetry to since applications within the same pod can communicate with each other via localhost.
- The latency between the application and the collector will not affect the application. This allows the application to offload its telemetry as quickly as possible, preventing unexpected memory loss or CPU pressure for high-throughput applications.

Let's look at how this is done. First, consider the following configuration, which includes the shopper, the grocery store, and the inventory applications. These have been containerized to allow us to deploy them via Kubernetes. In addition to this, the pod configuration contains a collector container. The most important thing to note in the configuration for our use case is the `containers` section, which defines the four containers that make up the application via `name` and `image` containers. Create a YAML file that includes the following configuration:

config/collector/sidecar.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cloud-native-example
  labels:
    app: example
spec:
  replicas: 1
  selector:
    matchLabels:
      app: example
  template:
    metadata:
      labels:
        app: example
```

```
spec:  
  containers:  
    - name: legacy-inventory  
      image: codeboten/legacy-inventory:chapter9  
    - name: grocery-store  
      image: codeboten/grocery-store:chapter9  
    - name: shopper  
      image: codeboten/shopper:chapter9  
    - name: collector  
      image: otel/opentelemetry-collector:0.43.0
```

The default configuration for the collector container configures an OTLP receiver, which you'll remember from *Chapter 8, OpenTelemetry Collector*. Additionally, it configures a logging exporter. We will modify this configuration later in this chapter; however, for now, the default is good enough. Let's apply the previous configuration to our cluster by running the following command. This uses the configuration to pull the container images from the Docker repository and creates the deployment and pod running the application:

```
$ kubectl apply -f config/collector/sidecar.yml  
deployment.apps/cloud-native-example created
```

We can ensure the pod is up and running with the following command, which gives us details about the pod along with the containers that are running within it:

```
$ kubectl describe pod -l app=example
```

We should be able to view all the details about the pod we configured:

kubectl describe output

Name:	cloud-native-example-6bdf8b6d6-
cfhc7	
Namespace:	default
Priority:	0 ...

With the pod running, we should now be able to look at the logs of the collector sidecar and observe the telemetry flowing. The following command lets us view the logs from any container within the pod. The container can be specified via the `-c` flag followed by the name of the container in question. The `-f` flag can be used to tail the logs. You can use the same command to observe the output of the other containers by changing the `-c` flag to the name of different containers:

```
kubectl logs -l app=example -f -c collector
```

The output of the previous command will contain telemetry from the various applications in the grocery store example. It should look similar to the following:

kubectl logs output

```
Span #6
  Trace ID      : 2ca9779b6ad6d5b1a067dd83ea0942d4
  Parent ID     : 09b499899194ba83
  ID            : c8a1d75232eaf376
  Name          : inventory request
  Kind          : SPAN_KIND_INTERNAL
  Start time    : 2021-06-19 22:38:53.3719469 +0000 UTC
  End time      : 2021-06-19 22:38:53.3759594 +0000 UTC
  Status code   : STATUS_CODE_UNSET
  Status message :

Attributes:
  -> http.method: STRING(GET)
  -> http.flavor: STRING(HttpFlavorValues.HTTP_1_1)
  -> http.url: STRING(http://localhost:5001/inventory)
  -> net.peer.ip: STRING(127.0.0.1)
```

Now we have a pod with a collector sidecar collecting telemetry! We will come back to make changes to this pod shortly, but first, let's look at the next deployment scenario.

System-level telemetry

As discussed in *Chapter 8, OpenTelemetry Collector*, the OpenTelemetry collector can be configured to collect metrics about the system it's running on. Often, this can be helpful when you wish to identify resource constraints on nodes, which is a fairly common problem. Additionally, the collector can be configured to forward data. So, it might be beneficial to deploy a collector on each host or node in your environment to provide an aggregation point for all the applications running on that node. As shown in the following diagram, deploying a collector as an agent can reduce the number of connections needed to send telemetry from each node:

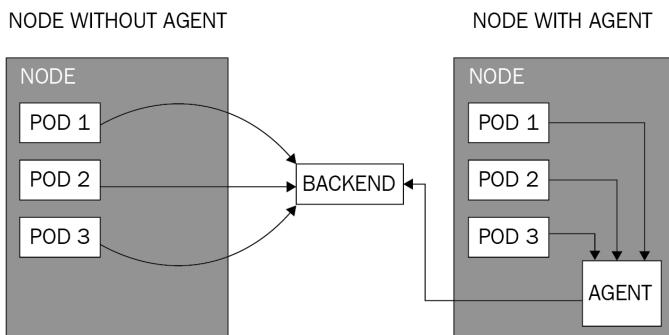


Figure 9.2 – Backend connections from nodes with and without an agent

This can become a significant processing bottleneck if, for example, the backend requires secure connections to be established with some level of frequency and if many applications are running per node.

Deploying the agent

The preferred way to deploy the collector as an agent is by using **Helm Charts**, which is provided by the OpenTelemetry project. You can find this at <https://github.com/open-telemetry/opentelemetry-helm-charts>. The first step to install a Helm chart is to tell Helm where it should look for the chart using the following command. This adds the open-telemetry repository to Helm:

```
$ helm repo add open-telemetry https://open-telemetry.github.io/opentelemetry-helm-charts
"open-telemetry" has been added to your repositories
```

Then, we can launch the collector service using the following command. This will install the `opentelemetry-collector` Helm chart, using all the default options:

```
$ helm install otel-collector open-telemetry/opentelemetry-collector
```

Let's check to see what happened in our Kubernetes cluster because of the previous command. The collector chart should have deployed the collector using **DaemonSet**. As mentioned earlier in the chapter, a **DaemonSet** is a way to deploy an instance of a pod on all nodes in Kubernetes. The following command lists all deployed DaemonSet deployments in our cluster, and you can view the resulting output as follows:

```
$ kubectl get DaemonSet
NAME                                     DESIRED
CURRENT      READY      UP-TO-DATE      AVAILABLE      NODE SELECTOR      AGE
otel-collector-opentelemetry-collector-agent   1           1           1           <none>          3m25s
```

Note that the results might be different depending on how many nodes your cluster has; mine has a single node. Next, let's examine the pods created using the following command:

```
$ kubectl get Pod
NAME                                     READY
STATUS      RESTARTS      AGE
otel-collector-opentelemetry-collector-agent-hhgkk   1/1
Running      0           4m39s
```

With the collector running as an agent on the node, let's learn about how to forward all the data from the collector sidecar to the agent.

Connecting the sidecar and the agent

It's time to update the sidecar collector configuration to use an OTLP exporter to export data. This can be accomplished using a **ConfigMap**, which gives us the ability to have Kubernetes create a file that will be mounted as a volume inside the container. For brevity, the details of ConfigMap and the volumes in Kubernetes will not be described here. Add the following ConfigMap object to the top of the sidecar configuration file:

config/collector/sidecar.yml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: otel-sidecar-conf
  labels:
    app: opentelemetry
    component: otel-sidecar-conf
data:
  otel-sidecar-config: |
    receivers:
      otlp:
        protocols:
          grpc:
          http:
    exporters:
      otlp:
        endpoint: "$NODE_IP:4317"
        tls:
          insecure: true
    service:
      pipelines:
        traces:
          receivers: [otlp]
          exporters: [otlp]
        metrics:
          receivers: [otlp]
          exporters: [otlp]
        logs:
```

```
receivers: [otlp]
exporters: [otlp]
```

The preceding configuration might remind you of the collector-specific configuration we explored in *Chapter 8, OpenTelemetry Collector*. It is worth noting that we will be using the `NODE_IP` environment variable in the configuration of the endpoint for the OTLP exporter.

Following this, we need to update the `containers` section further down. This is so that we can use the `otel-sidecar-conf` ConfigMap and tell the collector container to pass the configuration file at start time via the command option. The following configuration also exposes the node's IP address as an environment variable named `NODE_IP`:

config/collector/sidecar.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cloud-native-example
  labels:
    app: example
spec:
  replicas: 1
  selector:
    matchLabels:
      app: example
  template:
    metadata:
      labels:
        app: example
    spec:
      containers:
        - name: legacy-inventory
          image: codeboten/legacy-inventory:latest
        - name: grocery-store
          image: codeboten/grocery-store:latest
        - name: shopper
          image: codeboten/shopper:latest
```

```
- name: collector
  image: otel/opentelemetry-collector:0.27.0
  command:
    - "/otelcol"
    - "--config=/conf/otel-sidecar-config.yaml"
  volumeMounts:
    - name: otel-sidecar-config-vol
      mountPath: /conf
  env:
    - name: NODE_IP
      valueFrom:
        fieldRef:
          fieldPath: status.hostIP
  volumes:
    - configMap:
        name: otel-sidecar-conf
        items:
          - key: otel-sidecar-config
            path: otel-sidecar-config.yaml
  name: otel-sidecar-config-vol
```

For this new configuration to take effect, we'll go ahead and apply the configuration with the following command:

```
$ kubectl apply -f config/collector/sidecar.yml
```

Looking at the logs for the agent, we can now observe that telemetry is being processed by the collector:

```
kubectl logs -l component=agent-collector -f
2021-06-26T22:57:50.719Z INFO loggingexporter/logging_
exporter.go:327 TracesExporter {"#spans": 20}
2021-06-26T22:57:50.919Z INFO loggingexporter/logging_exporter.
go:327 TracesExporter {"#spans": 10}
2021-06-26T22:57:53.726Z INFO loggingexporter/logging_exporter.
go:375 MetricsExporter {"#metrics": 22}
2021-06-26T22:57:54.730Z INFO loggingexporter/logging_exporter.
go:327 TracesExporter {"#spans": 5}
```

While we're here, we might as well take some time to augment the telemetry processed by the collector. We can do this by applying some of the lessons we learned in *Chapter 8, OpenTelemetry Collector*. Let's configure a processor to provide more visibility inside our infrastructure.

Adding resource attributes

One of the great things about the agent collector is being able to ensure information about the node it's running on is present across all the telemetry processed by the agent. Helm allows us to override the default configuration via YAML; the following can be used in conjunction with the Helm chart to configure a resource attributes processor to inject information into our telemetry. It does the following:

- It makes an environment variable named `NODE_NAME` available for use by the resource attributes processor.
- It sets the `loglevel` parameter of the logging exporter to `debug`. This allows us to observe the data being emitted by the collector in more detail.
- It configures a resource attributes processor to inject the `NODE_NAME` environment variable into an attribute with the `k8s.node.name` key. Additionally, it adds the processor to the pipelines for logs, metrics, and traces.

Create a new `config/collector/config.yml` configuration file that contains the following configuration. We'll use this to update the Helm chart:

config/collector/config.yml

```
extraEnvs:  
  - name: NODE_NAME  
    valueFrom:  
      fieldRef:  
        fieldPath: spec.nodeName  
config:  
  exporters:  
    logging:  
      loglevel: debug  
agentCollector:  
  enabled: true  
  configOverride:  
    processors:
```

```
resource:
  attributes:
    - key: k8s.node.name
      value: ${NODE_NAME}
      action: upsert
  service:
    pipelines:
      metrics:
        processors: [batch, memory_limiter, resource]
      traces:
        processors: [batch, memory_limiter, resource]
      logs:
        processors: [batch, memory_limiter, resource]
```

Apply the preceding configuration via Helm using the following command:

```
$ helm upgrade otel-collector open-telemetry/opentelemetry-
collector -f ./config/collector/config.yml
Release "otel-collector" has been upgraded. Happy Helming!
NAME: otel-collector
LAST DEPLOYED: Sun Sep 19 13:22:10 2021
```

Looking at the logs from the agent, we should observe that the telemetry contains the attributes we added earlier:

```
$ kubectl logs -l component=agent-collector -f
```

Now, we have the collector sidecar sending data to the agent, and the agent is adding attributes via a processor:

kubectl logs output

```
2021-09-19T20:30:20.888Z          DEBUG    loggingexporter/
logging_exporter.go:366 ResourceSpans #0
Resource labels:
  -> telemetry.sdk.language: STRING(python)
  -> telemetry.sdk.name: STRING(opentelemetry)
  -> telemetry.sdk.version: STRING(1.3.0)
  -> net.host.name: STRING(cloud-native-example-5d57799766-
w8rjp)
```

```
-> net.host.ip: STRING(10.244.0.5)
-> service.name: STRING(grocery-store)
-> service.version: STRING(0.1.2)
-> k8s.node.name: STRING(kind-control-plane)
InstrumentationLibrarySpans #0
InstrumentationLibrary 0.1.2 grocery-store
```

Important Note

You might find it confusing that the previous example is not configuring receivers and exporters for the telemetry pipelines. This is because the values we pass into Helm only override some of the default configurations in the chart. Since we only needed to override the processors, the exporters and receivers continued to use the defaults that had already been configured. If you'd like to look at all the configured defaults, I suggest you refer to the repository at <https://github.com/open-telemetry/opentelemetry-helm-charts/blob/main/charts/opentelemetry-collector/values.yaml>.

Having this single point to aggregate and add information to telemetry could be used to simplify our application code. If you recall, in *Chapter 4, Distributed Tracing – Tracing Code Execution*, we created a custom ResourceDetector parameter to add `net.host.name` and `net.host.ip` attributes to all applications. That code could be removed in favor of injecting the same data via the collector. This means that now, any application could get these attributes without the complexity of utilizing custom code. Next, let's look at standalone service deployment.

Collector as a gateway

The last scenario we'll cover is how to deploy the collector as a standalone service, also known as a gateway. In this mode, the collector can provide a horizontally scalable service to do additional processing on the telemetry before sending it to a backend. Horizontal scaling means that if the service comes under too much pressure, we can launch additional instances of it, which, in this case, is the collector, to manage the increasing load. Additionally, the standalone service can provide a central location for the configuring, sampling, and scrubbing of the telemetry. From a security standpoint, it might also be preferable to have a single service sending traffic outside of your network. This is because it simplifies the rules that need to be configured and reduces the risk and blast radius of vulnerabilities.

Important Note

If your backend is deployed within your network, it's possible that a standalone service for the collector will be overkill, as you might be happier sending telemetry directly to the backend and saving yourself the trouble of operating an additional service in your infrastructure.

Conveniently, the same Helm chart we used earlier to deploy the collector as an agent can also be used to configure the gateway. This also provides us with an opportunity to configure the agent to export its data to the standalone collector, and therefore, we can feed two birds with one stone by doing both at the same time. Depending on your Kubernetes cluster, the default value of 2Gi might prevent the service from starting as it did in the case of my kind cluster. The following section can be appended to the bottom of the configuration file from the previous example to enable `standaloneCollector` and limit its memory consumption to 512Mi:

config/collector/config.yml

```
standaloneCollector:  
  enabled: true  
  resources:  
    limits:  
      cpu: 1  
      memory: 512Mi
```

Apply the update to the Helm chart by running the following command again:

```
$ helm upgrade otel-collector open-telemetry/opentelemetry-  
  collector -f ./config/collector/config.yml
```

A nice feature of the OpenTelemetry collector Helm chart is that if both `agentCollector` and `standaloneCollector` are configured, an OTLP exporter is automatically configured on the agent to forward traffic on the standalone collector. The following code depicts a snippet of the Helm chart template to give us an idea of how that will be configured:

config.tpl

```
{%- if .Values.standaloneCollector.enabled %}  
  exporters:  
    otlp:
```

```
    endpoint: {{ include "opentelemetry-collector.fullname" . }}:4317
      insecure: true
{{- end }}
```

It's time to examine the logs from the new service to check whether the data is reaching the standalone collector. The following command should be familiar now; make sure that you use the `standalone-collector` label when filtering the logs:

```
$ kubectl logs -l component=standalone-collector -f
```

Now the output from the logs shows us the same logs that we observed from the agent collector earlier, being processed by the standalone collector:

kubectl logs output

```
Metric #11
Descriptor:
  -> Name: otelcol_processor_accepted_spans
  -> Description: Number of spans successfully pushed into
the next component in the pipeline.
  -> Unit:
  -> DataType: DoubleSum
  -> IsMonotonic: true
  -> AggregationTemporality: AGGREGATION_TEMPORALITY_
CUMULATIVE
DoubleDataPoints #0
Data point labels:
  -> processor: memory_limiter
  -> service_instance_id: b208628b-7b0f-4275-9ea8-
a5c445582cbc
StartTime: 1632083630725000000
Timestamp: 1632083730725000000
Value: 718.000000
```

If you run `kubectl logs` with the `agent-collector` label, you'll find that because the agent collector is now using the `otlp` exporter instead of the `logging` exporter, it no longer emits logs.

Autoscaling

Unlike the sidecar, which relied on an application pod, or the agent deployment, which relied on individual nodes to scale, the standalone service can be automatically scaled based on CPU and memory constraints. It does this using a Kubernetes feature known as **HorizontalPodAutocaling**, which can be configured via the following:

```
autoscaling:  
  enabled: false  
  minReplicas: 1  
  maxReplicas: 10  
  targetCPUUtilizationPercentage: 80  
  targetMemoryUtilizationPercentage: 80
```

Depending on the needs of your environment, combining autoscaling with a load balancer might be worth pursuing to provide a high level of reliability and capacity for the service.

OpenTelemetry Operator

Another option for managing the OpenTelemetry collector in a Kubernetes environment is the OpenTelemetry operator (<https://github.com/open-telemetry/opentelemetry-operator>). If you're already familiar with using operators, they reduce the complexity of deploying and maintaining components in the Kubernetes landscape. In addition to managing the deployment of the collector, the OpenTelemetry operator provides support for auto-instrumenting applications.

Summary

We've only just scratched the surface of how to run the collector in production by looking at very specific use cases. However, you can start thinking about how to apply the lessons you have learned from this chapter to your environments. Whether it be using Kubernetes, bare metal, or another form of hybrid cloud environment, the same principles we explored in this chapter regarding how to best collect telemetry will apply. Collecting telemetry from an application should always be done with minimal impact on the application itself. The sidecar deployment mode provides a collection point as close as possible to the application without adding any dependency to the application itself.

The deployment of the collector as an agent gives us the ability to collect information about the worker running our applications, which could also allow us to monitor the health of the resources in our cluster. Additionally, this serves as a convenient point to augment the telemetry from applications with resource-specific attributes, which can be leveraged at analysis time. Finally, deploying the collector as a gateway allowed us to start thinking about how to deploy and scale a service to collect telemetry within our networks.

This chapter also gave us a chance to become familiar with some of the tools that OpenTelemetry provides to infrastructure engineers to manage the collector. We experimented with the OpenTelemetry collector container alongside the Helm charts provided by the project. Now that we have our environment deployed and primed to send data to a backend, in the next chapter, we'll take a look at options for open source backends.

10

Configuring Backends

So far, what we've been learning about has focused on the tools that are used to generate telemetry data. Although producing telemetry data is an essential aspect of making a system observable, it would be difficult to argue that the data we've generated in the past few chapters has made our system observable. After all, reading hundreds of lines of output in a console is hardly a practical tool for analysis. Data analysis is an essential aspect of observability that we have only briefly discussed thus far. This chapter is all about the tools we can use to analyze our applications' telemetry.

We are going to cover the following topics:

- Open source telemetry backends to analyze traces, metrics, and logs
- Considerations for running analysis systems in production

Throughout this chapter, we will visualize the data we've generated and start thinking about using it in real life. There is a large selection of analysis tools to choose from, but this chapter will only focus on a select few. It's worth noting that many commercial products (<https://opentelemetry.io/vendors/>) support OpenTelemetry; this chapter will focus solely on open source projects. This chapter will also skim the surface of the knowledge that you will need to run these telemetry backends in production.

Technical requirements

This chapter will use Python code to directly configure and use backends from a test application. To ensure your environment is set up correctly, run the following commands and ensure Python 3.6 or greater is installed on your system:

```
$ python --version  
Python 3.8.9  
$ python3 --version  
Python 3.8.9
```

If you do not have **Python 3.6+** installed, go to the Python website (<https://www.python.org/downloads/>) for instructions on installing the latest version.

To test out some of the exporters we'll be using in the chapter, install the following OpenTelemetry packages via pip:

```
$ pip install opentelemetry-distro \  
      opentelemetry-exporter-jaeger \  
      opentelemetry-exporter-zipkin
```

Additionally, we will use Docker (<https://docs.docker.com/get-docker/>) to deploy backends. The following code will ensure Docker is up and running in your environment:

```
$ docker version  
Client:  
  Cloud integration: 1.0.14  
  Version:          20.10.6  
  API version:     1.41  
  Go version:      go1.16.3 ...
```

To launch the backends, we will use Docker Compose once again. Ensure Compose is available by running the following commands:

```
$ docker compose version  
Docker Compose version 2.0.0-beta.1
```

Now, download the code and configuration for this chapter from this book's GitHub repository:

```
$ git clone https://github.com/PacktPublishing/Cloud-Native-Observability
$ cd Cloud-Native-Observability/chapter10
```

With the code downloaded, we're ready to launch the backends using Compose:

```
$ docker compose up
```

The following diagram shows the architecture of the environment that we'll be deploying. Initially, the example for this chapter will connect to the backends directly. After that, we will send data to the OpenTelemetry Collector which we'll connect to the telemetry backends. Grafana is connected to Jaeger, Zipkin, Loki, and Prometheus, as we will discuss later in this chapter.

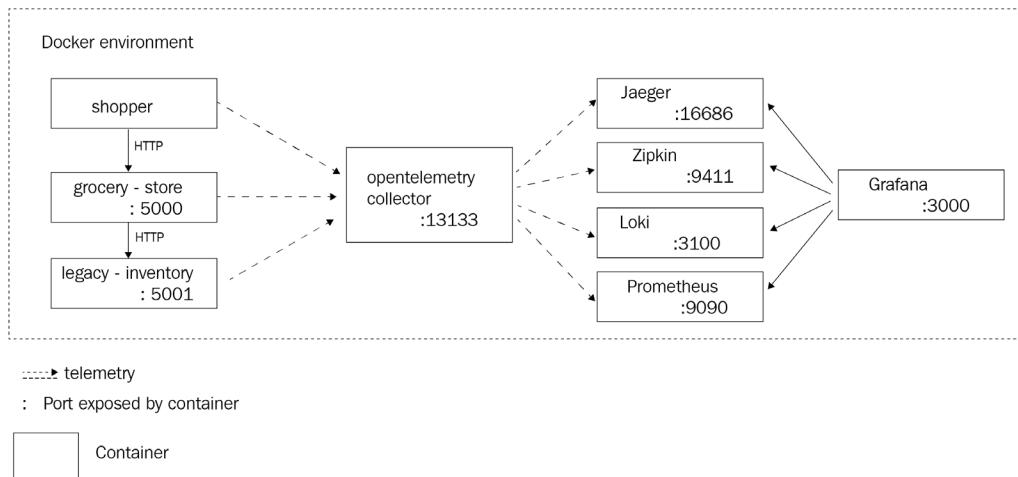


Figure 10.1 – Backend deployment in Docker

The configuration files for all this chapter's examples can be found in the `config` directory. Let's dive in!

Backend options for analyzing telemetry data

The world of observability contains an abundance of tools to provide you with insights into what systems are doing. Within OpenTelemetry, a backend is the destination of the telemetry data and is where it is stored and analyzed. All the telemetry backends that we will explore in this chapter provide the following:

- A destination for the telemetry data. This is usually in the form of a network endpoint, but not always.
- Storage for the telemetry data. The retention period that's supported by the storage is determined by the size of the storage and the amount of data being stored.
- Visualization tooling for the data. All the tools we'll use provide a web interface for displaying and querying telemetry data.

In OpenTelemetry, applications connect to backends via exporters, two of which we've already configured: the *console exporter* and the *OTLP exporter*. Each application can be configured to send data directly via an exporter that's been implemented specifically for that backend. The following table shows a current list of officially supported exporters for the backends by the OpenTelemetry specification, along with their status in the Python implementation:

Exporter	Signal	Status
Jaeger	Tracing	Stable
Zipkin	Tracing	Stable
Prometheus	Metrics	Active development

Figure 10.2 – Status of the exporters in Python for officially supported backends

Each language that implements the OpenTelemetry specification must provide an exporter for these backends. Additional information about the support for each exporter in different languages can be found in the specification repository: <https://github.com/open-telemetry/opentelemetry-specification/blob/main/spec-compliance-matrix.md#exporters>.

Tracing

Starting with the tracing signal, let's look at some options for visualizing traces. As we work through different backends, we'll see how it's possible to use other methods to configure a backend, starting with auto-instrumentation. The following code makes a series of calls to create a table and insert some data into a local database using SQLite (<https://www.sqlite.org/index.html>) while logging some information along the way:

sqlite_example.py

```
import logging
import os
import sqlite3

logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger(__name__)

logger.info("creating database")
con = sqlite3.connect("example.db")
cur = con.cursor()

logger.info("adding table")
cur.execute(
    """CREATE TABLE clouds
        (category text, description text)"""
)

logger.info("inserting values")
cur.execute("INSERT INTO clouds VALUES ('stratus','grey')")
con.commit()
con.close()

logger.info("deleting database")
os.remove("example.db")
```

Run the preceding code to ensure everything is working as expected by running the following command:

```
$ python sqlite_example.py
INFO: __main__:creating database
INFO: __main__:adding table
INFO: __main__:inserting values
INFO: __main__:deleting database
```

Now that we have some working code, let's ensure we can produce telemetry data by utilizing auto-instrumentation. As you may recall from *Chapter 7, Instrumentation Libraries*, Python provides the `opentelemetry-bootstrap` script to detect and install instrumentation libraries for us automatically. The library we're using in our code, `sqlite3`, has a supported instrumentation library that we can install with the following command:

```
$ opentelemetry-bootstrap -a install
Collecting opentelemetry-instrumentation-sqlite3==0.26b1
...

```

The output from the preceding command will produce some logging information that's generated by installing the packages through pip. If the output doesn't quite match mine, `opentelemetry-bootstrap` likely found additional packages to install for your environment.

Using `opentelemetry-instrument`, let's ensure that telemetry data is generated by configuring our trusty console exporter:

```
$ OTEL_RESOURCE_ATTRIBUTES=service.name.sqlite_example \
OTEL_TRACES_EXPORTER=console \
opentelemetry-instrument python sqlite_example.py
```

The output should now contain tracing information that's similar to the following abbreviated output:

output

```
INFO: __main__:creating database
INFO: __main__:adding table
```

```
INFO:__main__:inserting values
INFO:__main__:deleting database
{
    "name": "CREATE",
    "context": {
        "trace_id": "0xf98afa4316b3ac52633270b1e0534ffe",
        "span_id": "0xb52fb818cb0823da",
        "trace_state": "[]"
    },
...
}
```

Now, we're ready to look at our first telemetry backend by using a working example that utilizes instrumentation to produce telemetry data.

Zipkin

One of the original backends for distributed tracing, Zipkin (<https://zipkin.io>) was developed and open sourced by Twitter in 2012. The project was made available for anyone to use under the **Apache 2.0** license, and its community is actively maintaining and developing the project. Its core components are as follows:

- A **collector** to receive and index traces.
- A **storage** component, which provides a pluggable interface for storing data in various databases. The three storage options that are supported by Zipkin natively are *Cassandra*, *Elasticsearch*, and *MySQL*.
- A **query service** or API, which can be used to retrieve data from storage.
- As we'll see shortly, there's a **web UI**, which gives users visualization and querying capabilities.

The easiest way to send data from the sample application to Zipkin is by changing the OTEL_TRACES_EXPORTER environment variable, as per the following command:

```
$ OTEL_RESOURCE_ATTRIBUTES=service.name=sqlite_example \
  OTEL_TRACES_EXPORTER=zipkin \
  opentelemetry-instrument python sqlite_example.py
```

Setting the environment variable to `zipkin` tells auto-instrumentation to load `ZipkinExporter`, which is defined in the `opentelemetry-exporter-zipkin-proto-http` package. This connects to Zipkin via HTTP over port 9411. Launch a browser and access the Zipkin web UI via `http://localhost:9411/zipkin`.

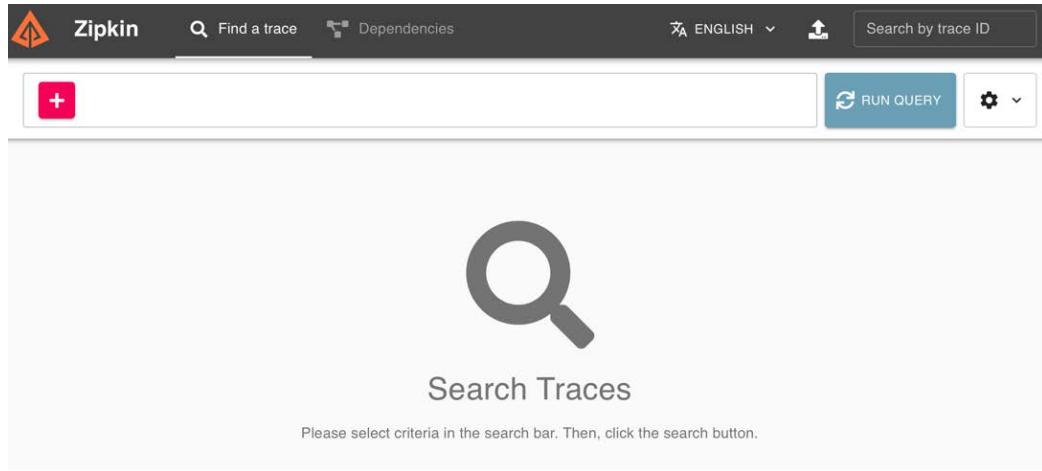


Figure 10.3 – Zipkin UI landing page

Search for traces by clicking the **Run Query** button. The results should show two traces; clicking on the details of one of these will bring up additional span information. This includes the attributes that are automatically populated by the instrumentation library, which are labeled as **Tags** in the Zipkin interface.

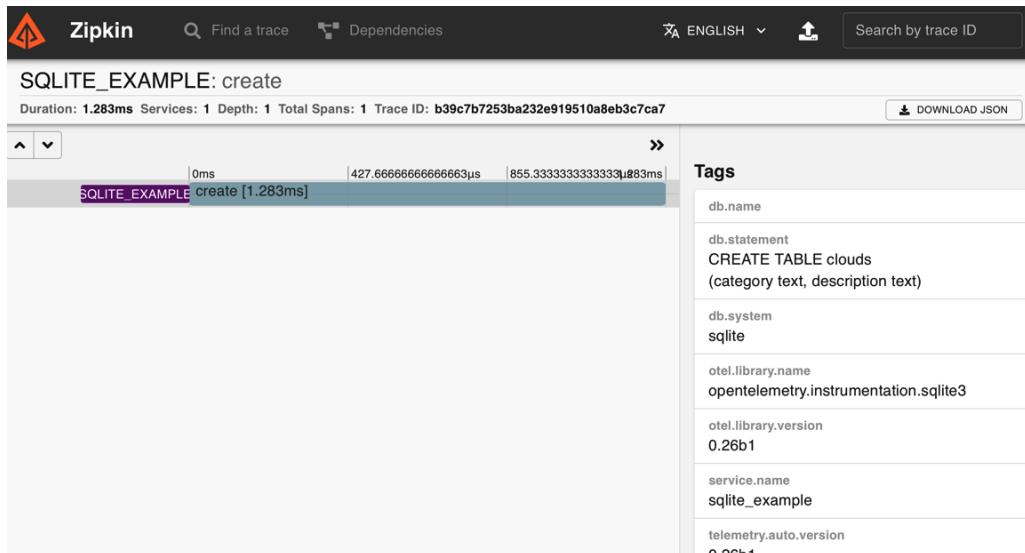


Figure 10.4 – Trace details view

The interface for querying lets you search for traces by trace ID, service name, duration, or tag, among other filters. It's also possible to filter traces by specifying a time window for the query. One last feature of Zipkin we will inspect requires multiple services to produce traces. As it happens, we have the grocery store already making telemetry data in our Docker environment; all we need to do is configure it to send data to Zipkin. Since the grocery store has already been configured to send data to the OpenTelemetry Collector, we'll update the collector's configuration to send data to Zipkin. Add the following configuration to enable the Zipkin exporter for the Collector:

config/collector/config.yml

```
receivers:  
  otlp:  
    protocols:  
      grpc:  
exporters:  
  logging:  
    loglevel: debug  
  zipkin:  
    endpoint: http://zipkin:9411/api/v2/spans  
service:  
  pipelines:  
    traces:  
      receivers: [otlp]  
      exporters: [logging, zipkin]  
    metrics:  
      receivers: [otlp]  
      exporters: [logging]  
  logs:  
    receivers: [otlp]  
    exporters: [logging]
```

For the configuration changes to take effect, the OpenTelemetry Collector container must be restarted. In terminal, use the following command from the chapter10 directory:

```
$ docker compose restart opentelemetry-collector
```

An alternative would be to relaunch the entire Docker Compose environment, but restarting just the `opentelemetry-collector` container is more expedient.

Important Note

Trying to run the `restart` command from other directories will result in an error while trying to find a suitable configuration.

Looking at the Zipkin interface again, searching for traces yields much more interesting results when the traces link spans across services. Try running some queries by searching for specific names or tags and see interesting ways to peruse the data. One more feature worth noting is the dependency graph, as shown in the following screenshot. It provides a service diagram that connects the components of the grocery store.

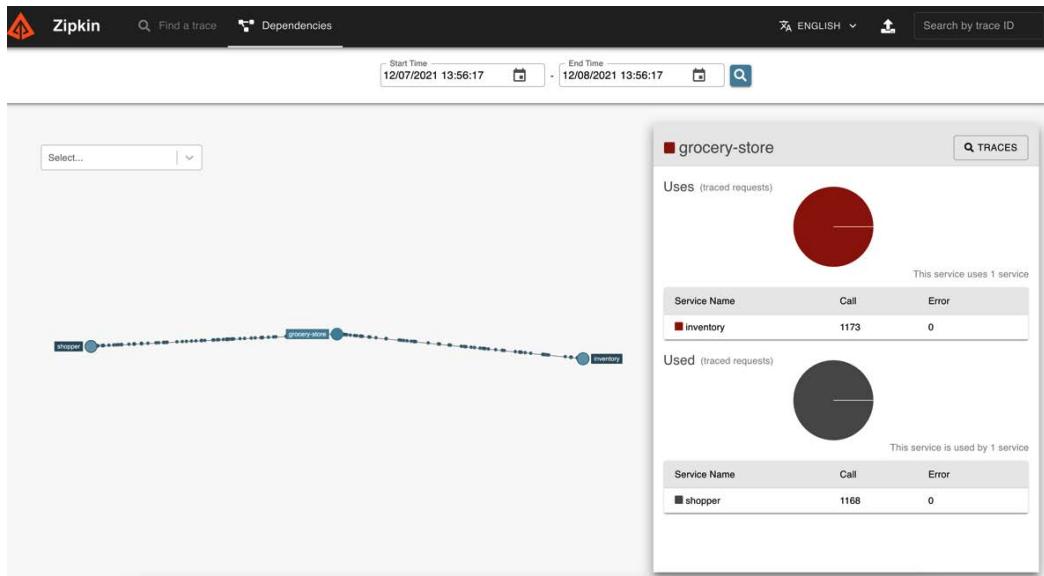


Figure 10.5 – Zipkin dependencies service diagram

The dependencies service diagram can often be helpful if you wish to get a quick overview of a system and understand the flow of information between components. Let's see how this compares with another tracing backend.

Jaeger

Initially developed by engineers at Uber, Jaeger (<https://www.jaegertracing.io>) was open sourced in 2015. It became a part of the **Cloud Native Computing Foundation (CNCF)**, the same organization that oversees OpenTelemetry, in 2017. The Jaeger project provides the following:

- An **agent** that runs as close to the application as possible, often on the same host or inside the same pod.
- A **collector** to receive distributed traces that, depending on your deployment, talks directly to a datastore or Kafka for buffering.
- An **ingester** that is (optionally) deployed. Its purpose is to read Kafka data and output it to a datastore.
- A **query** service that fetches data and provides a web UI for users to view it.

Returning to the sample SQLite application for a moment, the following code uses in-code configuration to configure OpenTelemetry with JaegerExporter. It would be easy to update the OTEL_TRACES_EXPORTER variable to `jaeger` instead of `zipkin` and run `opentelemetry-instrument` to accomplish the same thing. Still, auto-instrumentation may not always be possible for an application. Knowing how to configure these exporters manually will surely come in handy someday.

The code in the following example adds the familiar configuration of the tracing pipeline. The following are a couple of things to note:

- JaegerExporter has been configured to use a secure connection by default. We must pass in the `insecure` argument to change this.
- The code manually invokes `SQLite3Instrumentor` to trace calls via the `sqlite3` library.

Add the following code to the top of the SQLite example code we created previously:

sqlite_example.py

```
...
from opentelemetry import trace
from opentelemetry.exporter.jaeger.proto.grpc import
JaegerExporter
from opentelemetry.instrumentation.sqlite3 import
SQLite3Instrumentor
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace.export import BatchSpanProcessor

def configure_opentelemetry():
    SQLite3Instrumentor().instrument()
    exporter = JaegerExporter(insecure=True)
    provider = TracerProvider(
        resource=Resource.create({"service.name": "sqlite_"
example"})
    )
    provider.add_span_processor(BatchSpanProcessor(exporter))
    trace.set_tracer_provider(provider)

configure_opentelemetry()
...
```

Running the application with the following command will send data to Jaeger:

```
$ python sqlite_example.py
```

Access the Jaeger interface by browsing to `http://localhost:16686/`. Upon arriving on the landing page, searching for traces should yield results similar to what's shown in the following screenshot. Note that in Jaeger, you'll need to select a service from the dropdown on the left-hand side before you can find traces.

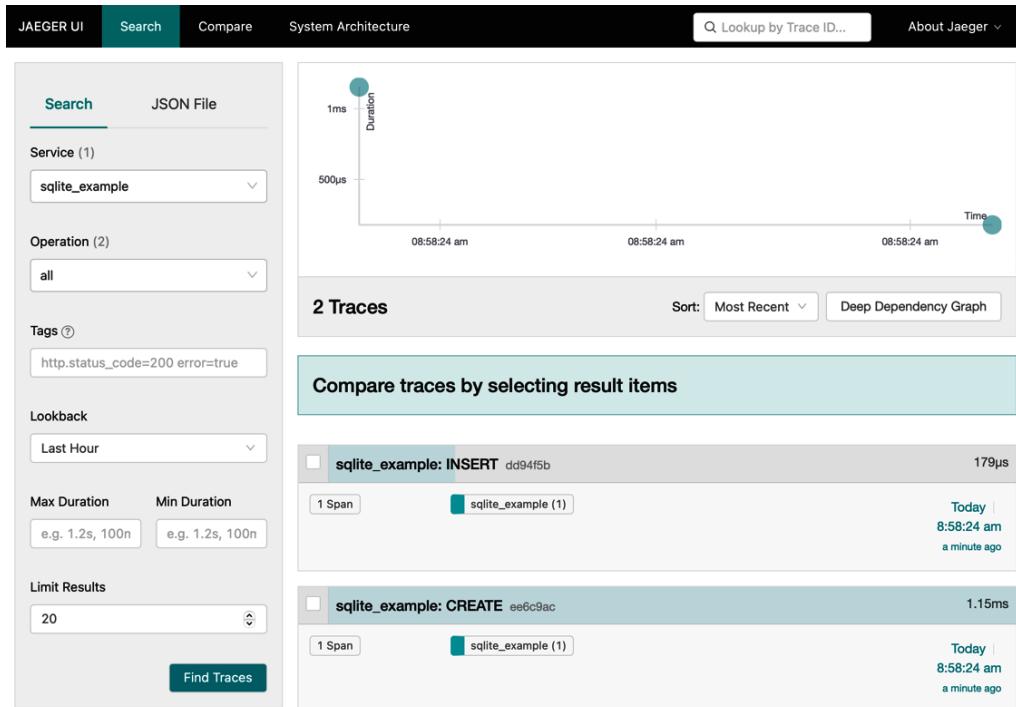


Figure 10.6 – Jaeger search results

Looking through the details for each trace, we can see that the same information we previously found in Zipkin can be seen in Jaeger, although organized slightly differently. Next, let's update the Collector file's configuration to send traces from the grocery store to Jaeger. Add the following `jaeger` section under the `exporters` definition in the Collector configuration file:

config/collector/config.yml

```
...
exporters:
...
jaeger:
  endpoint: jaeger:14250
  tls:
    insecure: true
service:
  pipelines:
```

```
traces:  
    receivers: [otlp]  
    exporters: [logging, zipkin, jaeger]  
...
```

Restart the Collector container to reload the updated configuration:

```
$ docker compose restart opentelemetry-collector
```

The Jaeger web UI starts becoming more interesting when more data comes in. For example, note the scatter plot displayed previously in the search results; it's an excellent way to identify outliers. The chart supports clicking on individual traces to bring up additional details.

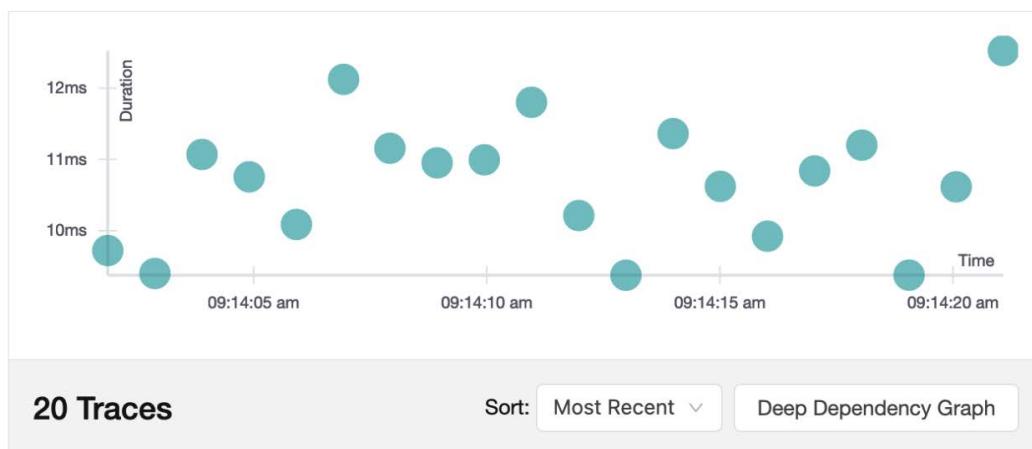


Figure 10.7 – Scatter plot of trace durations

Like Zipkin, Jaeger visualizes the relationship between services via the **System Architecture** diagram. An exciting feature that Jaeger delivers is that you can compare traces by selecting traces of interest from the search results and clicking the **Compare Traces** button. The following screenshot shows a comparison between two traces for the same operation. In one instance, the grocery store failed to connect to the legacy inventory service, resulting in an error and a missing span.

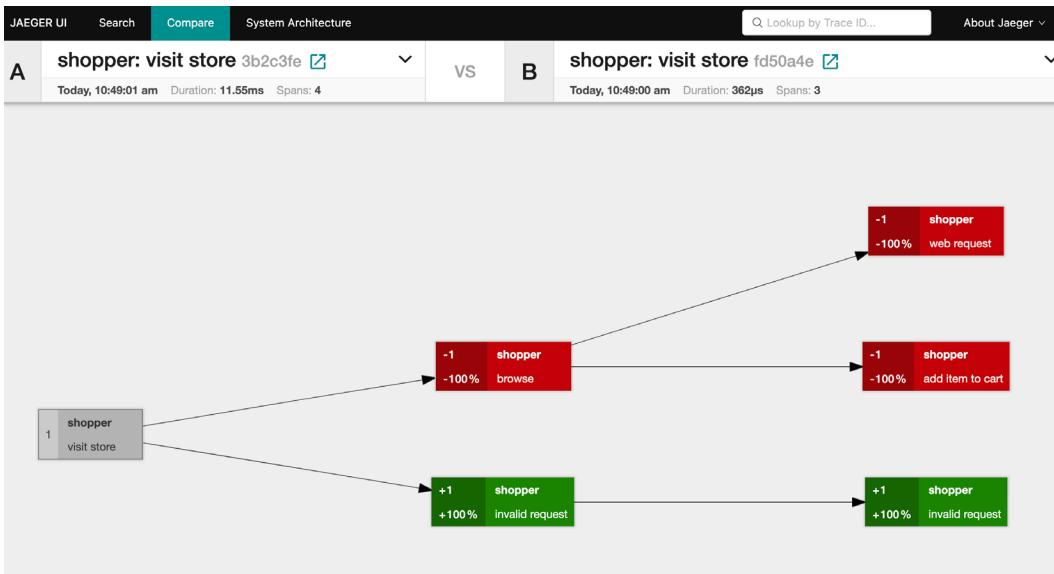


Figure 10.8 – Trace comparison diagram

This visual representation of the trace comparison can help us quickly identify a difference between a typical trace and one where an error occurred, zoning in on where the change was made.

Metrics

As of November 2021, Prometheus is the only officially supported exporter for the metrics signal. Official support for StatsD in the specification was requested some time ago (<https://github.com/open-telemetry/opentelemetry-specification/issues/374>), but the lack of a specification for StatsD has stopped OpenTelemetry from making it a requirement.

Prometheus

A project initially developed in 2012 by engineers at SoundCloud, Prometheus (<https://prometheus.io>) is a dominant open source metrics system. Its support for multi-dimensional data and first-class support for alerting quickly made it a favorite of DevOps practitioners. Initially, Prometheus used a pull model only. Applications that wanted to store metrics exposed them via a network endpoint that had been scraped by the Prometheus server. Prometheus now supports the push model via Prometheus Remote Write, allowing producers to send data to a remote server. The components of interest to us currently are as follows:

- The **Prometheus server** collects data from scrape targets and stores it in its **time-series database (TSDB)**.
- The Prometheus Query Language (**PromQL**) for searching and aggregating metrics.
- Visualization for metrics data via the **Prometheus web UI**.

As the current implementation of the Prometheus exporter for Python is still in development, in this section, we will focus on the data that's produced by the grocery store, which is sent through the Collector. The implementation of the Prometheus exporter in the Collector is also in development at the time of writing, but it is further along. The following configuration can be added to the Collector's configuration to send metrics to Prometheus:

config/collector/config.yml

```
exporters:  
  ...  
  prometheus:  
    endpoint: 0.0.0.0:8889  
    resource_to_telemetry_conversion:  
      enabled: true  
  service:  
    pipelines:  
      ...  
    metrics:  
      receivers: [otlp]  
      exporters: [logging, prometheus]  
  ...
```

Reload the Collector with the following command:

```
$ docker compose restart opentelemetry-collector
```

Bring up the Prometheus web interface by pointing your browser to `http://localhost:9090`. Using PromQL, the following query will return all the metrics that have been produced by the OpenTelemetry Collector:

```
{job="opentelemetry-collector"}
```

This can be seen in the following screenshot:

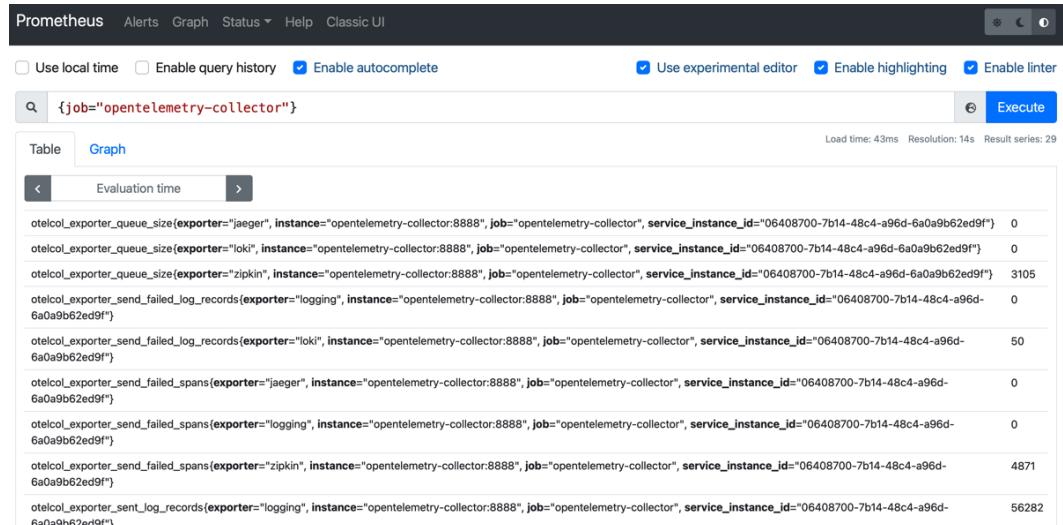


Figure 10.9 – PromQL query results

The pull model makes horizontally scaling Prometheus an easy aspect that makes it a good option for many environments. There are, of course, challenges with running Prometheus at scale, like any other backend. Unfortunately, we don't have the space to dive into data availability across regions and long-term storage, to name just a few challenges. Like Jaeger and OpenTelemetry, Prometheus is also a project under the governance of the CNCF.

Logging

Even with no officially supported backends at the time of writing, it's helpful to have a way to query logs that doesn't require looking at files on disk directly or paying for a service to get started. The tools that we've discussed in this section have exporters available in the OpenTelemetry Collector but may not necessarily have exporters implemented in other languages.

Loki

A project started by Grafana Labs in 2018, Loki is a log aggregation system that's designed to be easy to scale and operate. Its design is inspired by Prometheus and is composed of the following components:

- A **distributor** that validates and pre-processes incoming logging data before sending it off to the ingester
- An **ingester** that writes data to storage and provides a read endpoint for in-memory data
- A **ruler** that interprets configurable rules and triggers actions based on them
- A **querier** that performs queries for both the ingester and storage
- A **query frontend** that acts as a proxy for optimizing requests that are made to the querier

These components can be run in a single deployment or as a separate service to make it easy to deploy them in whichever mode makes the most sense. The OpenTelemetry Collector provides an exporter for Loki, which can be configured as per the following code snippet. The configuration of the Loki exporter supports relabeling attributes and resource attributes before sending the data. In the following example, the `service.name` resource attribute has been relabeled `job`:

config/collector/config.yml

```
exporters:
```

```
...
```

```
loki:  
  endpoint: http://loki:3100/loki/api/v1/push  
  labels:  
    resource:  
      service.name": "job"  
  service:  
    pipelines:  
    ...  
  logs:  
    receivers: [otlp]  
    exporters: [logging, loki]  
  ...
```

Once more, restart the Collector to reload the configuration and start sending data to Loki:

```
$ docker compose restart opentelemetry-collector
```

Now, it's time to review this logging data. You may have noticed that the components we mentioned earlier for Loki lack an interface for visualizing the data. That's because the interface of choice for Loki is Grafana, which is a separate project altogether.

Grafana

Grafana (<https://grafana.com/grafana/>) is an open source tool that's been developed since 2014 by Grafana Labs to allow users to visualize and query telemetry data. Grafana enables users to configure data sources that support various formats for traces, metrics, and logs. This includes Zipkin, Jaeger, Prometheus, and Loki.

Let's see how we can access the logs we sent to our Loki backend. Access the **Explore** section of the Grafana interface via a browser by going to `http://localhost:3000/explore`. In the query field, enter `{job=~"grocery-store|inventory|shopper"}`. This will bring up all the logs for all the grocery store components.

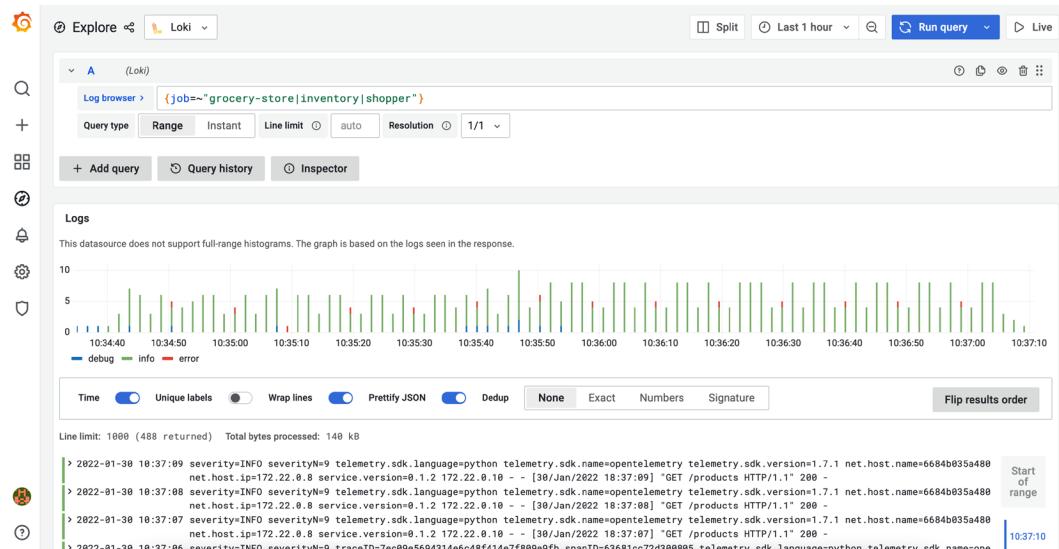


Figure 10.10 – Logs search results

Grafana allows users to create dashboards and alerts for the data that's received via its data sources. Since it's possible to view data from all signals, it's also possible to see data across all signals within a single dashboard. An example of such a dashboard has been preconfigured in the development environment and is accessible via the following URL: `http://localhost:3000/d/otel/opentelemetry`.

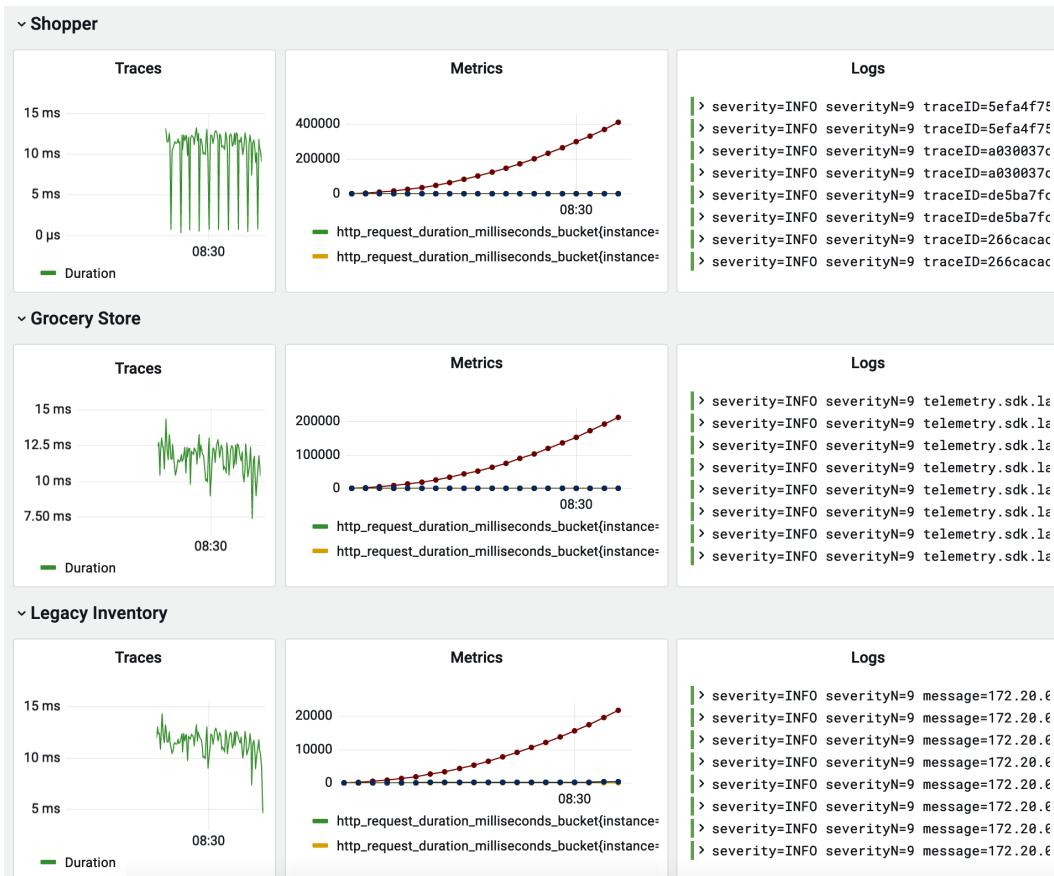


Figure 10.11 – Dashboard combining data across signals

There are many more capabilities to explore regarding each of the tools discussed in this chapter. A vast amount of information on all the features and configuration options is available on the website associated with each project. I strongly recommend spending some time familiarizing yourself with these tools.

Running in production

Using analysis tools in development is one thing; running them in production is another. Running a single container on one machine is not an acceptable strategy for operating a service that provides information that's critical to an organization. It's worth considering the challenges of scaling telemetry backends to meet the demands of the real world. The following subsections highlight areas that require further reading before you run any of the backends mentioned earlier in production.

High availability

The availability of telemetry backends is likely not as critical to end users as that of the applications they are used to monitor. However, having an outage and realizing that the data that's required to investigate is unavailable or missing during the outage causes problems. If an application promises an uptime of 99.99%, the telemetry backend must be available to account for those guarantees. Some aspects to consider when thinking of the high availability in the context of a telemetry backend are as follows:

- Ensuring the telemetry receivers are available to senders. This can be accomplished by placing a load balancer between the senders and the receivers.
- Considering how the backends will be upgraded and how to minimize the impact on the applications being observed.
- Understanding the expectations for being able to query the data.
- Deciding how much of the data needs to be replicated to mitigate the risks of catastrophic failure.

Additionally, geo-distributed environments must consider how the applications will behave if a backend is deployed in distant regions. Many of the backends we've discussed provide recommendations for deploying the backend in a mode that supports high availability.

Scalability

The telemetry backend must be able to grow alongside the applications they support. Whether that's by adding more instances or increasing the number of resources that are given to the backend, knowing what the tools support can help you decide which backend to use. Some questions that are worth asking are as follows:

- Can the components of the backend be scaled independently?
- Will scaling the backend require vertical scaling or horizontal scaling?
- How far will the solution scale? Is there a hard limit somewhere along the way?

When we think about scalability, it's essential to understand the limitations of the tools we're working with, even if we never come close to using them to their full extent.

Data retention

A key challenge in telemetry is the volume of data that's being produced. It's easy to lean toward storing every detail forever, as it is hard to predict when the data may become necessary. It's a bit like holding on to all those old cables and connectors for hardware that hasn't existed since the late 90s; you never know when it will come in handy!

The problem with storing all the data forever is that it becomes costly at scale. On the other hand, the cost tends to cause engineers to lean in the opposite direction too much, where we log or record so little that it becomes hard to find anything of value. Some options to think about are as follows:

- Identify an acceptable data retention period for the quantity of data that's being produced. This will likely change as teams become better at identifying issues within shorter periods.
- If long-term data storage is desirable, use lower-cost storage to reduce operational costs. This may result in longer query times, but the data will still be available.
- Tune a sensible sampling option for the different signals. More on this will be covered in *Chapter 12, Sampling*.

At a minimum, data retention should cover periods when engineers are expected to be away. For example, if no one is watching systems during a 2-day weekend, data should be retained for 3 or more days. Otherwise, events that occur during the weekend will be impossible to investigate.

Whatever you decide regarding the retention method, there are plenty of ways to fine-tune it over time. It's also critical for teams across the organization to be aware of what this data retention is.

Privacy regulations

Depending on the contents of the telemetry data that's produced by applications, the requirements for where and how the data can be stored vary. For example, regulations such as the **General Data Protection Regulation (GDPR)** recommend personally identifiable data to be pseudonymized to ensure nobody can be associated with the data without additional processing. Depending on the requirements in your environment and the telemetry data that's being produced, we have to take the following into account about the data:

- The data may need to remain within a specific country or region.
- The data may need to be processed further before being stored. This could mean many things, from the data being encrypted to scrubbing it of personally identifiable information or pseudonymization.
- The data may need access control and auditing capabilities.

Using the OpenTelemetry Collector as a receiver of telemetry data before sending the data to telemetry backends can alleviate concerns around data privacy. Various processors in the Collector can be configured to facilitate the scrubbing of sensitive information.

Summary

One of the many jobs of software engineers today includes evaluating the new technology and tools that are available to determine whether these tools would improve their ability to accomplish their goals. Leveraging auto-instrumentation, in-code configuration, and the OpenTelemetry Collector, we quickly sent data from one backend to another to help us compare these tools.

All the tools we've discussed in this chapter take much more than a few pages to become familiar with. Entire books have been written about running these in production, and the skills to do so well at scale require practice and experience. Understanding some areas that need additional thinking when those tools are deployed allows us to uncover some of the unknowns.

Looking through the different tools and starting to see how each one provides functionality to visualize the data gave us a sense of how telemetry data can be used to start answering questions about our systems. In the next chapter, we will focus on how these visualizations can identify specific problems.

11

Diagnosing Problems

Finally, after instrumenting application code, configuring a collector to transmit the data, and setting up a backend to receive the telemetry, we have all the pieces in place to observe a system. But what does that mean? How can we detect abnormalities in a system with all these tools? That's what this chapter is all about. This chapter aims to look through the lens of an analyst and see what the shape of the data looks like as events occur in a system. To do this, we'll look at the following areas:

- How leaning on chaos engineering can provide the framework for running experiments in a system
- Common scenarios of issues that can arise in distributed systems
- Tools that allow us to introduce failures into our system

As we go through each scenario, we'll describe the experiment, propose a hypothesis, and use telemetry to verify whether our expectations match what the data shows us. We will use the data and become more familiar with analysis tools to help us understand how we may answer questions about our systems in production. As always, let's start by setting up our environment first.

Technical requirements

The examples in this chapter will use the grocery store application we've used and revisited throughout the book. Since the chapter's goal is to analyze telemetry and not specifically look at how this telemetry is produced, the application code will not be the focus of the chapter. Instead of running the code as separate applications, we will use it as Docker (<https://docs.docker.com/get-docker/>) containers and run it via Compose. Ensure Docker is installed with the following command:

```
$ docker version
Client:
Cloud integration: 1.0.14
Version:          20.10.6
API version:      1.41
Go version:       go1.16.3 ...
```

The following command will ensure Docker Compose is also installed:

```
$ docker compose version
Docker Compose version 2.0.0-beta.1
```

The book's companion repository (<https://github.com/PacktPublishing/Cloud-Native-Observability>) contains the Docker Compose configuration file, as well as the configuration required to run the various containers. Download the companion repository via Git:

```
$ git clone https://github.com/PacktPublishing/Cloud-Native-
Observability
$ cd Cloud-Native-Observability/chapter11
```

With the configuration in place, start the environment via the following:

```
$ docker compose up
```

Throughout the chapter, as we conduct experiments, know that it is always possible to reset the pristine Docker environment by removing the containers entirely with the following commands:

```
$ docker compose stop
$ docker compose rm
```

All the tools needed to run various experiments have already been installed inside the grocery store application containers, meaning there are no additional tools to install. The commands will be executed via `docker exec` and run within the container.

Introducing a little chaos

In normal circumstances, the real world is unpredictable enough that intentionally introducing problems may seem unnecessary. Accidental configuration changes, sharks chewing through undersea cables, and power outages affecting data centers are just a few events that have caused large-scale issues across the world. In distributed systems, in particular, dependencies can cause failures that may be difficult to account for during normal development.

Putting applications through various stress, load, functional, and integration tests before they are deployed to production can help predict their behavior to a large extent. However, some circumstances may be hard to reproduce outside of a production environment. A practice known as **chaos engineering** (<https://principlesofchaos.org>) allows engineers to learn and explore the behavior of a system. This is done by intentionally introducing new conditions into the system through experiments. The goal of these experiments is to ensure that systems are robust enough to withstand failures in production.

Important Note

Although chaos engineers run experiments in production, it's essential to understand that one of the principles of chaos engineering is **not to cause unnecessary pain** to users, meaning experiments must be controlled and limited in scope. In other words, despite its name, chaos engineering isn't just going around a data center and unplugging cables haphazardly.

The cycle for producing experiments goes as follows:

1. It begins with a system under a known good state or steady state.
2. A hypothesis is then proposed to explain the experiment's impact on the system's state.
3. The proposed experiment is run on the system.

4. Verification of the impact on the system takes place, validating that the prediction matches the hypothesis. The verification step provides an opportunity to identify unexpected side effects of the experiment. If something behaved precisely as expected, great! If it acted worse than expected, why? If it behaved better than expected, what happened? It's essential to understand what happened, especially if the results were better than expected. It's too easy to look at a favorable outcome and move right along without taking the time to understand why it happened.
5. Once verification is complete, improvements to the system are made, and the cycle begins anew. Ideally, running these experiments can be automated once the results on the system are satisfactory to guard against future regressions.

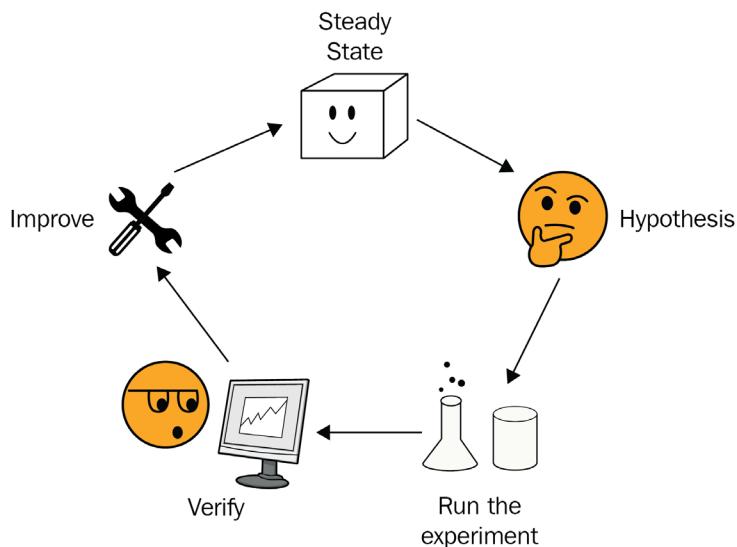


Figure 11.1 – Chaos engineering life cycle

The hypothesis step is crucial because if the proposed experiment will have disastrous repercussions on the system, it may be worth rethinking the experiment.

For example, a hypothesis that turning off all production servers simultaneously will cause a complete system failure doesn't need to be validated. Additionally, this would guarantee the creation of unnecessary pain for all users of the system. There isn't much to learn from running this in production, except seeing how quickly users send angry tweets.

An alternative experiment that may be worthwhile is to shut down an availability zone or a region of a system to ensure load balancing works. This would provide an opportunity to learn how production traffic would be handled in the case of such a failure, validating that those systems in place to manage such a failure are doing their jobs. Of course, if no such mechanisms are in place, it's not worth experimenting either, as this would have the same impact as shutting down all the servers for all users in that zone or region.

This chapter will take a page out of the chaos engineering book and introduce various failure modes into the grocery store system. We will propose a hypothesis and run experiments to validate the assumptions. Using the telemetry produced by the store, we will validate that the system behaved as expected. This will allow us to use the telemetry produced to understand how we can answer questions about our system via this information. Let's explore a problem that impacts all networked applications: latency.

Experiment #1 – increased latency

Latency is the delay introduced when a call is made and the response is returned to the originating caller. It can inject itself into many aspects of a system. This is especially true in a distributed system where latency can be found anywhere one service calls out to another. The following diagram shows an example of how latency can be calculated between two services. Service A calls a remote service (B), the request duration is 25 ms, but a large portion of that time is spent transferring data to and from service B, with only 5 ms spent executing code.

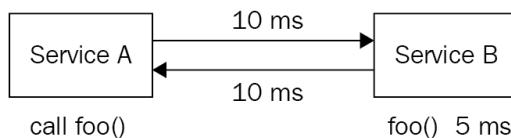


Figure 11.2 – Latency incurred by calling a remote service

If the services are collocated, the latency is usually negligible and can often be ignored. However, latency must be accounted for when services communicate over a network. This is something to think about at development time. It can be caused by factors such as the following:

- The physical distance between the servers hosting services. As even the speed of light requires time to travel distance, the greater the distance between services, the greater the latency.
- A busy network. If a network reaches the limits of how much data it can transfer, it may throttle the data transmitted.
- Problems in any applications or systems connecting the services. Load balancers and DNS services are just two examples of the services needed to connect two services.

Experiment

The first experiment we'll run is to increase the latency in the network interface of the grocery store. The experiment uses a Linux utility to manipulate the configuration on the network interface: Traffic Control ([https://en.wikipedia.org/wiki/Tc_\(Linux\)](https://en.wikipedia.org/wiki/Tc_(Linux))). Traffic Control, or `tc`, is a powerful utility that can simulate a host of scenarios, including packet loss, increased latency, or throughput limits. In this experiment, `tc` will add a delay to inbound and outbound traffic, as shown in *Figure 11.3*:

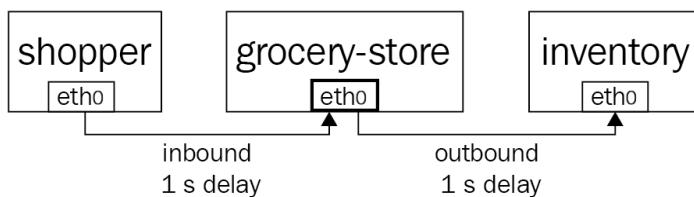


Figure 11.3 – Experiment #1 will add latency to the network interface

Hypothesis

Increasing the latency to the grocery store network interface will incur the following:

- A reduction in the total number of requests processed
- An increase in the request duration time

Use the following Docker command to introduce the latency. This uses the `tc` utility inside the grocery store container to add a 1s delay to all traffic received and sent through interface `eth0`:

```
$ docker exec grocery-store tc qdisc add dev eth0 root netem
delay 1s
```

Verify

To observe the metrics and traces generated, access the **Application Metrics** dashboard in Grafana via the following URL: `http://localhost:3000/d/apps/application-metrics`. You'll immediately notice a drop in the **Request count** time series and an increase in **Request duration** time quantiles. As time passes, you'll also start seeing the **Request duration distribution** histogram change to show an increasing number of requests falling into buckets with longer durations that are as per the following screenshot:

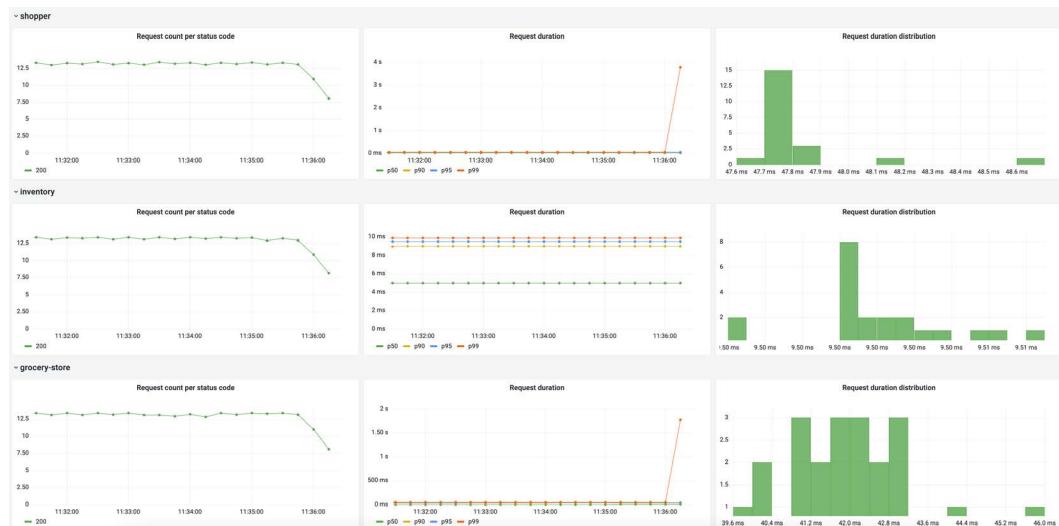


Figure 11.4 – Request metrics for shopper, grocery-store, and inventory services

Note that although the drop in request count is the same across the inventory and grocery store services, the duration of the request for the inventory service remains unchanged. This is a great starting point, but it would be ideal to identify precisely where this jump in the request duration occurred.

Important Note

As discussed earlier in this book, the correlation between metrics and traces provided by exemplars could help us drill down more quickly by giving us specific traces to investigate from the metrics. However, since the implementation of exemplar support in OpenTelemetry is still under development at the time of writing, the example in this chapter does not take advantage of it. I hope that by the time you're reading this, exemplar support is implemented across many languages in OpenTelemetry.

Let's look at the tracing data in Jaeger available at `http://localhost:16686`. From the metrics, we already know that the issue appears to be isolated to the grocery store service. Sure enough, searching for traces for that service yields the following chart:

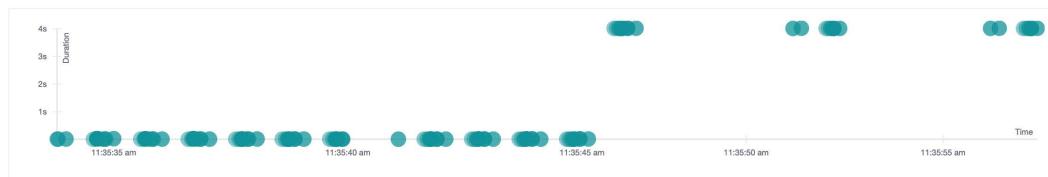


Figure 11.5 – Increased duration results in Jaeger

It's clear from this chart that something happened. The following screenshot shows us two traces; at the top is a trace from before we introduced the latency; at the bottom is a trace from after. Although the two look similar, looking at the duration of the spans named `web request` and `/products`, it's clear that those operations are taking far longer at the bottom than at the top.

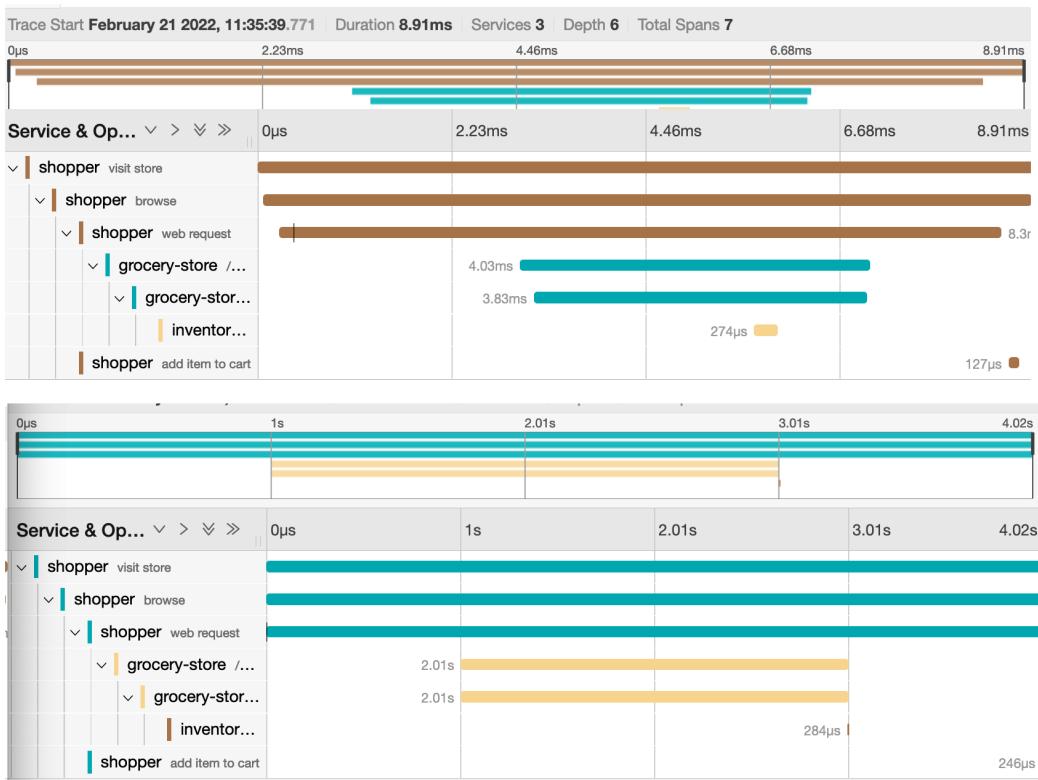


Figure 11.6 – Trace comparison before and after latency was introduced

As hypothesized, the total number of requests processed by the grocery store dropped due to the simulation. This, in turn, reduced the number of calls to the inventory service. The total duration of the request as observed by the shopper client increased significantly.

Remove the delay to see how the system recovers. The following command removes the delay introduced earlier:

```
$ docker exec grocery-store tc qdisc del dev eth0 root netem  
delay 1s
```

Latency is only one of the aspects of networks that can cause problems for applications. Traffic Control's network emulator (<https://man7.org/linux/man-pages/man8/tc-netem.8.html>) functionality can simulate many other symptoms, such as packet loss and rate-limiting, or even the re-ordering of packets. If you're keen on playing with networks, it can be a lot of fun to simulate different scenarios. However, the network isn't the only thing that can cause problems for systems.

Experiment #2 – resource pressure

Although cloud providers make provisioning new computing resources more accessible than ever before, even computing in the cloud is still bound by the physical constraints of hardware running applications. Memory, processors, hard drives, and networks all have their limits. Many factors can contribute to resource exhaustion:

- Misconfigured or misbehaving applications. Crashing and restarting in a fast loop, failing to free memory, or making requests over the network too aggressively can all contribute to a load on resources.
- An increase or spike in requests being processed by the service. This could be good news; the service is more popular than ever! Or it could be bad news, the result of a denial-of-service attack. Either way, more data to process means more resources are required.
- Shared resources cause resource starvation. This problem is sometimes referred to as the noisy neighbor problem, where resources are consumed by another tenant of the physical hardware where a service is running.

Autoscaling or dynamic resource allocation helps alleviate resource pressures to some degree by allowing users to configure thresholds at which new resources should be made available to the system. To know how these thresholds should be configured, it's valuable to experiment with how applications behave under limited resources.

Experiment

We'll investigate how telemetry can help identify resource pressures in the following scenario. The grocery store container is constrained to 50 M of memory via its Docker Compose configuration. Memory pressure will be applied to the container via stress.

The Unix stress utility (<https://www.unix.com/man-page/debian/1/STRESS/>) spins workers that produce loads on systems. It creates memory, CPU, and I/O pressures by calling system functions in a loop; malloc/free, sqrt, and sync, depending on which resource is being pressured.

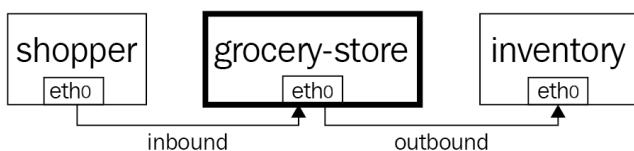


Figure 11.7 – Experiment #2 will apply memory pressure to the container

Hypothesis

As resources are consumed by stress, we expect the following to happen:

- The grocery store processes fewer requests as it cannot obtain the resources to process requests.
- Latency increases across the system, as requests will take longer to process through the grocery store.
- Metrics collected from the grocery store container should quickly identify the increased resource pressure.

The following introduces memory pressure by adding workers that consume a total of 40 M of memory to the grocery store container via `stress` for 30 minutes:

```
$ docker exec grocery-store stress --vm 20 --vm-bytes 2M
--timeout 30m
stress: info: [20] dispatching hogs: 0 cpu, 0 io, 10 vm, 0 hdd
```

Verify

With the pressure in place, let's see whether the telemetry matches what we expected. Looking at the application metrics, we can see an almost immediate increase in request duration as per the following screenshot. The request count is also slightly impacted simultaneously.

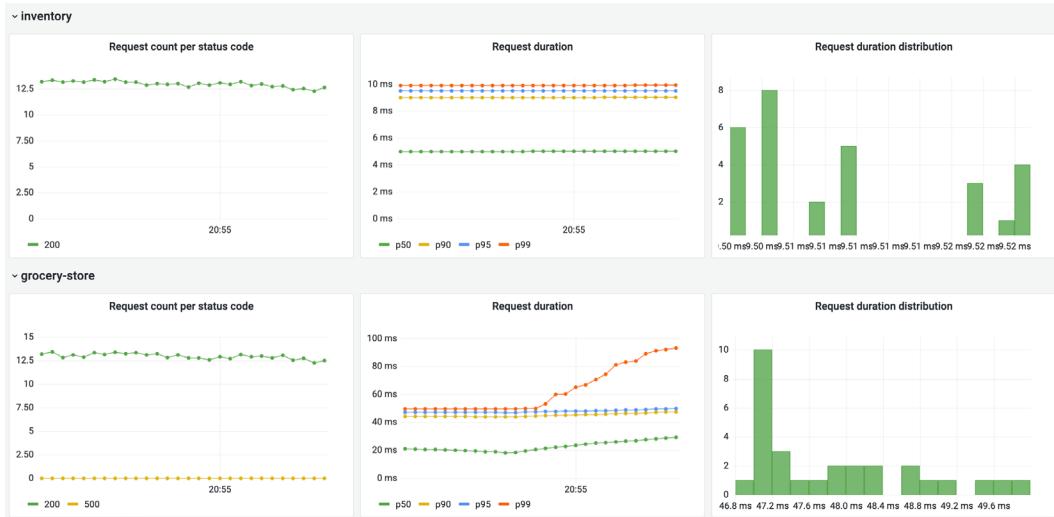


Figure 11.8 – Application metrics

What else can we learn about the event? Searching through traces, an increase in duration similar to what occurred during the first experiment is shown:

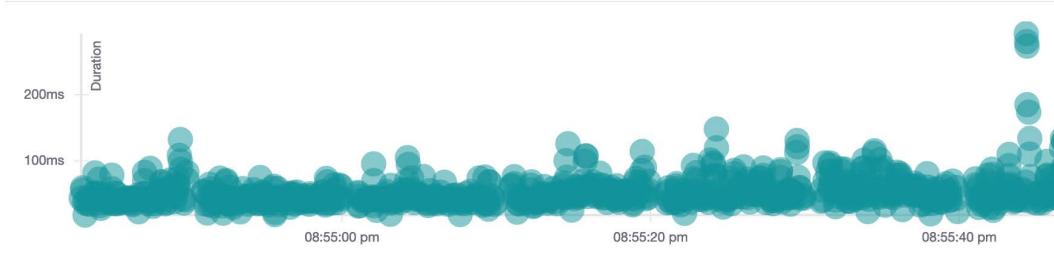


Figure 11.9 – Trace duration increased

Looking in more detail at individual traces, we can identify which paths through the code cause this increase. Not surprisingly, the `allocating memory` span, which locates an operation performing a memory allocation, is now significantly longer, with its time jumping from 2.48 ms to 49.76 ms:

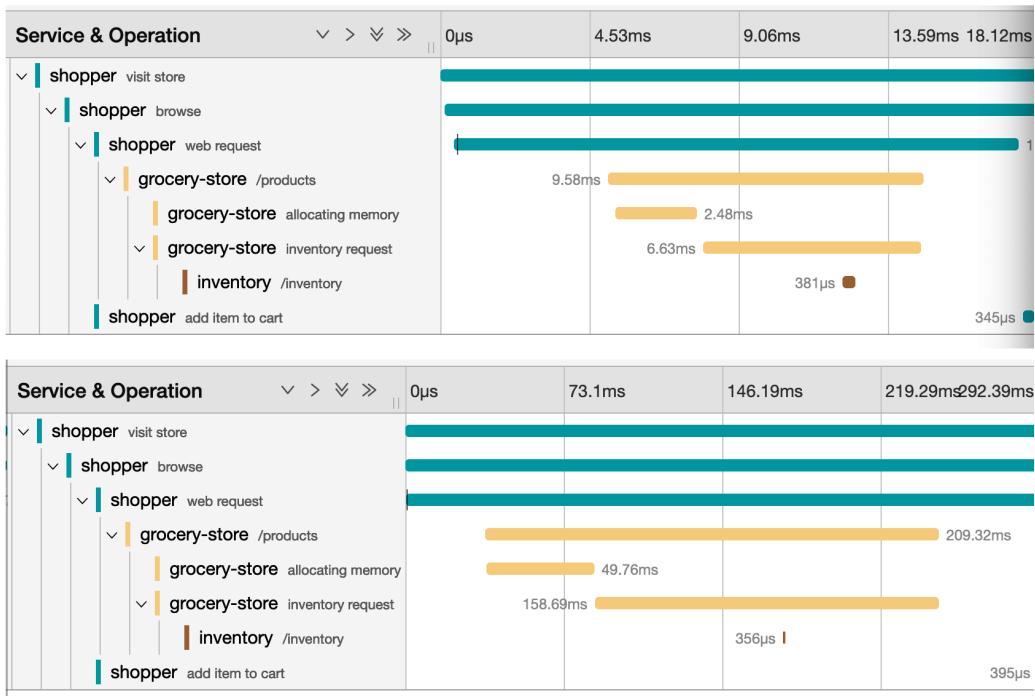


Figure 11.10 – Trace comparison before and after the memory increase

There is a second dashboard worth investigating at this time, the **Container metrics** dashboard (<http://localhost:3000/d/containers/container-metrics>). This dashboard shows the CPU, memory, and network metrics collected directly from Docker by the collector's Docker stats receiver (<https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/receiver/dockerstatsreceiver>). Reviewing the following charts, it's evident that resource utilization increased significantly in one container:

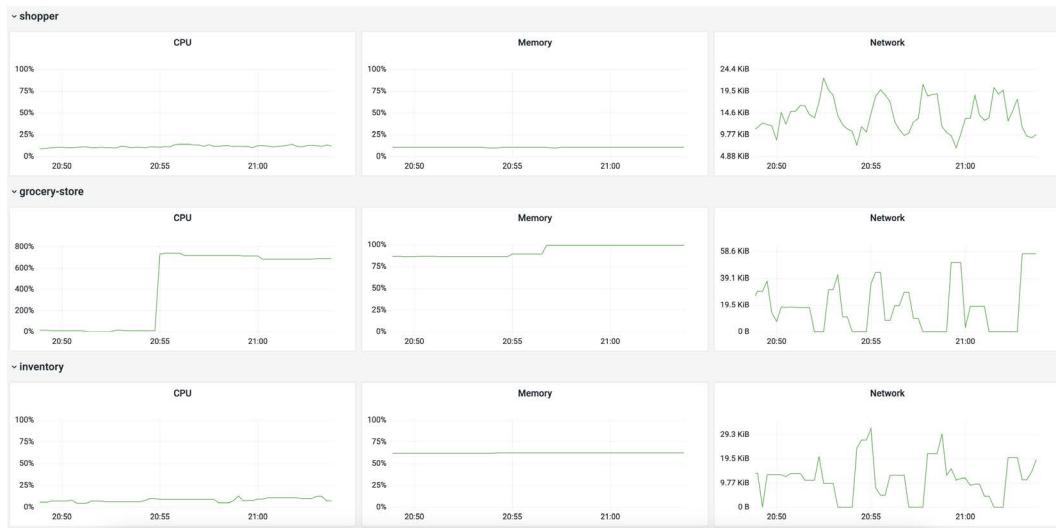


Figure 11.11 – Container metrics for CPU, memory, and network

From the data, it's evident that something happened to the container running the grocery store. The duration of requests through the system increased, as did the metrics showing memory and CPU utilization for the container. Identifying resource pressure in this testing environment isn't particularly interesting beyond this.

Recall that OpenTelemetry specifies resource attributes for all signals, meaning that if multiple services are running on the same resource, host, container, or node, it would be possible to correlate information about those services using this resource information, meaning that if we were running multiple applications on the same host, and one of them triggered memory pressure, it would be possible to verify its impact on other services within the same host by utilizing its resource attributes as an identifier when querying telemetry.

Resource information can help answer questions when, for example, a host has lost power, and there is a need to identify all services impacted by this event quickly. Another way to use this information is when two completely unrelated services are experiencing problems simultaneously. If those two services operate on the same node, resource information will help connect the dots.

Experiment #3 – unexpected shutdown

If a service exits in a forest of microservices and no one is around to observe it, does it make a sound? With the right telemetry and alert configuration in place, it certainly does. Philosophical questions aside, services shutting down happens all the time. Ensuring that services can manage this event gracefully is vital in dynamic environments where applications come and go as needed.

Expected and unexpected shutdowns or restarts can be caused by any number of reasons. Some common ones are as follows:

- An uncaught exception in the code causes the application to crash and exit.
- Resources consumed by a service pass a certain threshold, causing an application to be terminated by a resource manager.
- A job completes its task, exiting intentionally as it terminates.

Experiment

This last experiment will simulate a service exiting unexpectedly in our system to give us an idea of what to look for when identifying this type of failure. Using the `docker kill` command, the inventory service will be shut down unexpectedly, leaving the rest of the services to respond to this failure and report this issue.

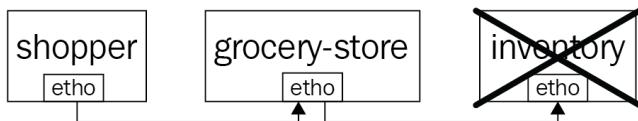


Figure 11.12 – Experiment #3 will terminate the inventory service

In a production environment, issues arising from this scenario would be mitigated by having multiple instances of the inventory service running behind some load balancing, be it a load balancer or DNS load balancing. This would result in traffic being redirected away from the failed instance and over to the others still in operation. For our experiment, however, a single instance is running, causing a complete failure of the service.

Hypothesis

Shutting down the inventory service will result in the following:

- All metrics from the inventory container will stop reporting.
- Errors will be recorded by the grocery store; this should be visible through the request count per status code reporting status code 500.
- Logs should report errors from the shopper container.

Using the following command, send a signal to shut down the inventory service. Note that `docker kill` sends the container a `KILL` signal, whereas `docker stop` would send a `TERM` signal. We use `KILL` here to prevent the service from shutting down cleanly:

```
$ docker kill inventory
```

Verify

With the inventory service stopped, let's head over to the application metrics dashboard one last time to see what happened. The request count graph shows a rapid increase in requests whose response code is `500`, representing an internal server error.

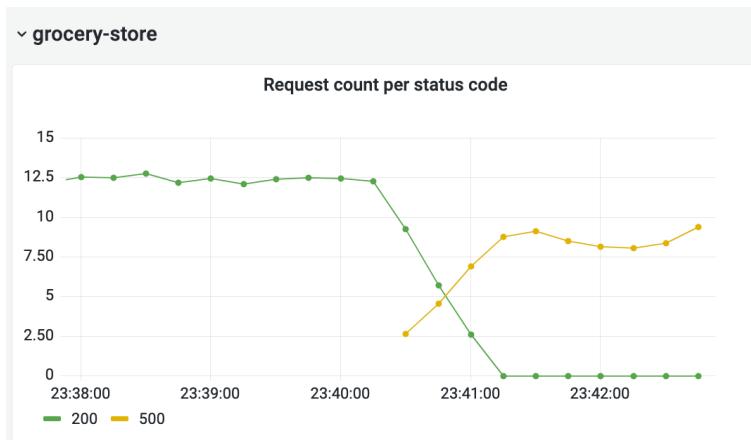


Figure 11.13 – The request counter shows an increase in errors

One signal we've yet to use in this chapter is logging. Look for the **Logs** panel at the bottom of the application metrics dashboard to find all the logs emitted by our system. Specifically, look for an entry reporting a failed request to the grocery store such as the following, which is produced by the shopper application:

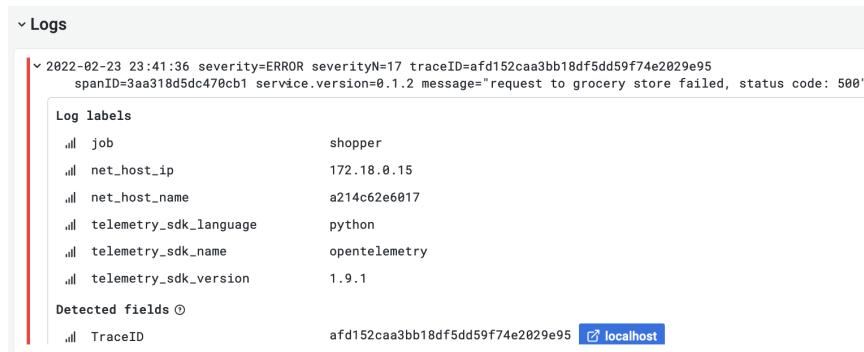


Figure 11.14 – Log entry being recorded

Expanding the log entry shows details about the event that caused an error. Unfortunately, the message `request to grocery store failed` isn't particularly helpful here, although notice that there is a `TraceID` field in the data shown. This field is adjacent to a link. Clicking on the link will take us to the corresponding trace in Jaeger, which shows us the following:



Figure 11.15 – Trace confirms the grocery store is unable to contact the inventory

The trace provides more context as to what error caused it to fail, which is helpful. An exception with the message recorded in the span provides ample details about the **legacy-inventory** service appearing to be missing. Lastly, the container metrics dashboard will confirm the inventory container stopped reporting metrics as per the following screenshot:

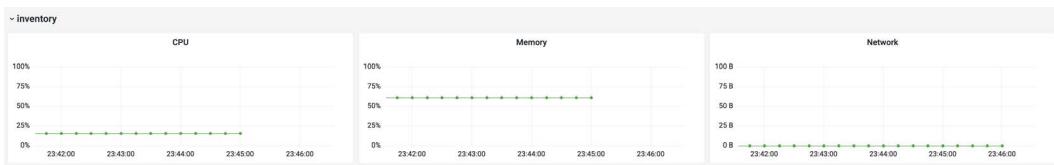


Figure 11.16 – Inventory container stopped reporting metrics

Restore the stopped container via the `docker start` command and observe as the error rate drops and traffic is returned to normal:

```
$ docker start inventory
```

There are many more scenarios that we could investigate in this chapter. However, we only have limited time to cover these. From message queues filling up to caching problems, the world is full of problems just waiting to be uncovered.

Using telemetry first to answer questions

These experiments are a great way to gain familiarity with telemetry. Still, it feels like cheating to know what caused a change before referring to the telemetry to investigate a problem. A more common way to use telemetry is to look at it when a problem occurs without intentionally causing it. Usually, this happens when deploying new code in a system.

Code changes are deployed to many services in a distributed system several times a day. This makes it challenging to figure out which change is responsible for a regression. The complexity of identifying problematic code is compounded by the updates being deployed by different teams. Update the `image` configuration for the `shopper`, `grocery-store`, and `legacy-inventory` services in the Docker Compose configuration to use the following:

docker-compose.yml

```
shopper:  
  image: codeboten/shopper:chapter11-example1  
  ...  
grocery-store:  
  image: codeboten/grocery-store:chapter11-example1  
  ...  
legacy-inventory:  
  image: codeboten/legacy-inventory:chapter11-example1
```

Update the containers by running the following command in a separate terminal:

```
$ docker compose up -d legacy-inventory grocery-store shopper
```

Was the deployment of the new code a success? Did we make things better or worse? Let's look at what the data shows us. Starting with the application metrics dashboard, it doesn't look promising. Request duration has spiked upward, and requests per second dropped significantly.



Figure 11.17 – Application metrics of the deployment

It appears to be impacting both the inventory service and grocery store service, which would indicate something may have gone wrong in the latest deployment of the inventory service. Looking at traces, searching for all traces shows the same increase in request duration as the graphs from the metrics. Selecting a trace and looking through the details points to a likely culprit:

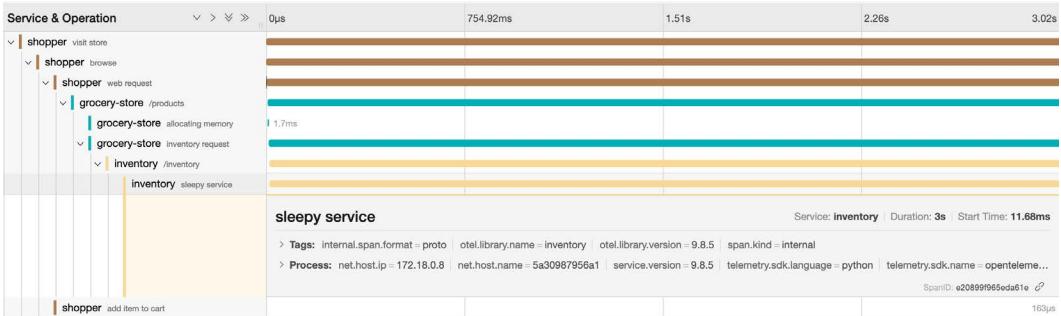


Figure 11.18 – A suspicious span named sleepy service

It appears the addition of an operation called **sleepy service** is causing all sorts of problems in the latest deployment! With this information, we can revert the change and resolve the issue.

In addition to the previous scenario, four additional scenarios are available through published containers to practice your observation skills. They have unoriginal tags: `chapter11-example2`, `chapter11-example3`, `chapter11-example4`, and `chapter11-example5`. I recommend trying them all before looking through the scenarios folder in the companion repository to see whether you can identify the deployed problem!

Summary

Learning to navigate telemetry data produced by systems comfortably takes time. Even with years of experience, the most knowledgeable engineers can still be puzzled by unexpected changes in observability data. The more time spent getting comfortable with the tools, the quicker it will be to get to the bottom of just what caused changes in behavior.

The tools and techniques described in this chapter can be used repeatedly to better understand exactly what a system is doing. With chaos engineering practices, we can improve the resilience of our systems by identifying areas that can be improved upon under controlled circumstances. By methodically experimenting and observing the results from our hypotheses, we can measure the improvements as we're making them.

Many tools are available for experimenting and simulating failures; learning how to use these tools can be a powerful addition to any engineer's toolset. As we worked our way through the vast amount of data produced by our instrumented system, it's clear that having a way to correlate data across signals is critical in quickly moving through the data.

It's also clear that generating more data is not always a good thing, as it is possible to become overwhelmed quickly or overwhelm backends. The last chapter looks at how sampling can help reduce the volume of data.

12

Sampling

One of the challenges of telemetry, in general, is managing the quantity of data that can be produced by instrumentation. This can be problematic at the time of generation if the tools producing telemetry consume too many resources. It can also be costly to transfer the data across various points of the network. And, of course, the more data is produced, the more storage it consumes, and the more resources are required to sift through it at the time of analysis. The last topic we'll discuss in this book focuses on how we can reduce the amount of data produced by instrumentation while retaining the value and fidelity of the data. To achieve this, we will be looking at sampling. Although primarily a concern of tracing, sampling has an impact across metrics and logs as well, which we'll learn about throughout this chapter. We'll look at the following areas:

- Concepts of sampling, including sampling strategies, across the different signals of OpenTelemetry
- How to configure sampling at the application level via the OpenTelemetry **Software Development Kit (SDK)**
- Using the OpenTelemetry collector to sample data

Along the way, we'll look at some common pitfalls of sampling to learn how they can best be avoided. Let's start with the technical requirements for the chapter.

Technical requirements

All the code for the examples in the chapter is available in the companion repository, which can be downloaded using `git` with the following command. The examples are under the `chapter12` directory:

```
$ git clone https://github.com/PacktPublishing/Cloud-Native-  
Observability  
$ cd Cloud-Native-Observability/chapter12
```

The first example in the chapter consists of an example application that uses the OpenTelemetry Python SDK to configure a sampler. To run the code, we'll need Python 3.6 or greater installed:

```
$ python --version  
Python 3.8.9  
$ python3 --version  
Python 3.8.9
```

If Python is not installed on your system, or the installed version of Python is less than the supported version, follow the instructions from the Python website (<https://www.python.org/downloads/>) to install a compatible version.

Next, install the following OpenTelemetry packages via `pip`. Note that through dependency requirements, additional packages will automatically be installed:

```
$ pip install opentelemetry-distro \  
      opentelemetry-exporter-otlp  
$ pip freeze | grep opentelemetry  
opentelemetry-api==1.8.0  
opentelemetry-distro==0.27b0  
opentelemetry-exporter-otlp==1.8.0  
opentelemetry-exporter-otlp-proto-grpc==1.8.0  
opentelemetry-exporter-otlp-proto-http==1.8.0  
opentelemetry-instrumentation==0.27b0  
opentelemetry-proto==1.8.0  
opentelemetry-sdk==1.8.0
```

The second example will use the OpenTelemetry Collector, which can be downloaded from GitHub directly. The example will focus on the tail sampling processor, which currently resides in the `opentelemetry-collector-contrib` repository. The version used in this chapter can be found at the following location: <https://github.com/open-telemetry/opentelemetry-collector-releases/releases/tag/v0.43.0>. Download a binary that matches your current system from the available releases. For example, the following command downloads the macOS for AMD64-compatible binary. It also ensures the executable flag is set and runs the binary to check that things are working:

```
$ wget -O otelcol.tar.gz https://github.com/open-telemetry/opentelemetry-collector-releases/releases/download/v0.43.0/otelcol-contrib_0.43.0_darwin_amd64.tar.gz
$ tar -xzf otelcol.tar.gz otelcol-contrib
$ chmod +x ./otelcol-contrib
$ ./otelcol-contrib --version
otelcol-contrib version 0.43.0
```

If a package matching your environment isn't available, you can compile the collector manually. The source is available on GitHub: <https://github.com/open-telemetry/opentelemetry-collector-contrib>. With this in place, let's get started with sampling!

Concepts of sampling across signals

A method often used in the domain of research, the process of sampling selects a subset of data points across a larger dataset to reduce the amount of data to be analyzed. This can be done because either analyzing the entire dataset would be impossible, or unnecessary to achieve the research goal, or because it would be impractical to do so. For example, if we wanted to record how many doors on average each car in a store parking lot has, it may be possible to go through the entire parking lot and record the data in its entirety. However, if the parking lot contains 20,000 cars, it may be best to select a sample of those cars, say 2,000, and analyze that instead. There are many sampling methods used to ensure that a representational subset of the data is selected, to ensure the meaning of the data is not lost because of the sampling.

Methods for sampling can be grouped as either of the following:

- **Probabilistic** (https://en.wikipedia.org/wiki/Probability_sampling): The probability of sampling is a known quantity, and that quantity is applied across all the data points in the dataset. Returning to the parking lot example, a probabilistic strategy would be to sample 10% of all cars. To accomplish this, we could record the data for every tenth car parked. In small datasets, probabilistic sampling is less effective as the variability between data points is higher.
- **Non-probabilistic** (https://en.wikipedia.org/wiki/Nonprobability_sampling): The selection of data is based on specific characteristics of the data. An example of this may be to choose the 2,000 cars closest to the store out of convenience. This introduces bias into the selection process. The parking area located closest to the store may include designated spots or even spots reserved for smaller cars, therefore impacting the results.

Traces

Specifically, sampling in the context of OpenTelemetry really means deciding what to do with spans that form a particular trace. **Spans** in a **trace** are either processed or dropped, depending on the configuration of the sampler. Various components of OpenTelemetry are involved in carrying the decision throughout the system:

- A **Sampler** is the starting point, allowing users to select a sampling level. Several samplers are defined in the OpenTelemetry specification, more on this shortly.
- The **TracerProvider** class receives a sampler as a configuration parameter. This ensures that all traces produced by the **Tracer** provided by a specific **TracerProvider** are sampled consistently.
- Once a trace is created, a decision is made on whether to sample the trace. This decision is stored in the **SpanContext** associated with all spans in this trace. The sampling decision is propagated to all the services participating in the distributed trace via the **Propagator** configured.
- Finally, once a span has ended, the **SpanProcessor** applies the sampling decision. It passes the spans for all sampled traces to the **SpanExporter**. Traces that are not sampled are not exported.

Metrics

For certain types of data, sampling just doesn't work. Sampling in the case of metrics may severely alter the data, rendering it effectively useless. For example, imagine recording data for each incoming request to a service, incrementing a counter by one with each request. Sampling this data would mean that any increment that is not sampled would result in unaccounted requests. Values recorded as a result would lose the meaning of the original data.

A single metric data point is smaller than a single trace. This means that typically, managing metrics data creates less overhead to process and store. I say *typically* here because this depends on many factors, such as the dimensions of the data and the frequency at which data points are collected.

Reducing the amount of data produced by the metrics signal focuses on aggregating the data, which reduces the number of data points transmitted. It does this by combining data points rather than selecting specific points and discarding others. There is, however, one aspect of metrics where sampling comes into play: exemplars. If you recall from *Chapter 2, OpenTelemetry Signals – Traces, Metrics, and Logs*, exemplars are data points that allow metrics to be correlated with traces. There is no need to produce exemplars that reference unsampled traces. The details of how exemplars and their sampling should be configured are still being discussed in the OpenTelemetry specification as of December 2021. It is good to be aware that this will be a feature of OpenTelemetry in the near future.

Logs

At the time of writing, there is no specification in OpenTelemetry around if or how the logging signal should be sampled. The following shows a couple of ways that are currently being considered:

- OpenTelemetry provides the ability for logs to be correlated with traces. As such, it may make sense to provide a configuration option to only emit log records that are correlated with sampled traces.
- Log records could be sampled in the same way that traces can be configured via a sampler, to only emit a fraction of the total logs (<https://github.com/open-telemetry/opentelemetry-specification/issues/2237>).

An alternative to sampling for logging is aggregation. Log records that contain the same message could be aggregated and transmitted as a single record, which could include a counter of repeated events. As these options are purely speculative, we won't focus any additional efforts on sampling and logging in this chapter.

Before diving into the code and what samplers are available, let's get familiar with some of the sampling strategies available.

Sampling strategies

When deciding on how to best configure sampling for a distributed system, the strategy selected often depends on the environment. Depending on the strategy chosen, the sampling decision is made at different points in the system, as shown in the following diagram:

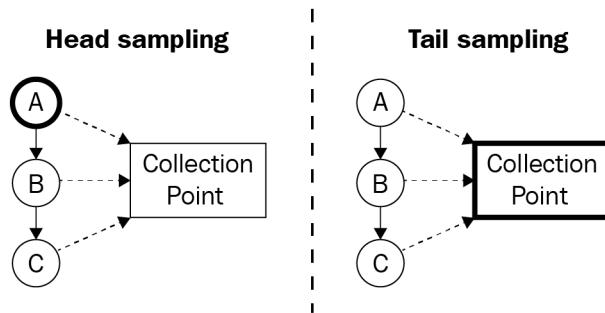


Figure 12.1 – Different points at which sampling decisions can take place

The previous diagram shows where the decisions to sample are made, but before choosing a strategy, we must understand what they are and when they are appropriate.

Head sampling

The quickest way to decide about a trace is to decide at the very beginning whether to drop it or not; this is known as **head sampling**. The application that creates the first span in a trace, the root span, decides whether to sample the trace or not, and propagates that decision via the context to every subsequent service called. This signals to all other participants in the trace whether they should be sending this span to a backend.

Head sampling reduces the overhead for the entire system, as each application can discard unnecessary spans without computing a sampling decision. It also reduces the amount of data transmitted, which can have a significant impact on network costs.

Although it is the most efficient way to sample data, deciding at the beginning of the trace whether it should be sampled or not doesn't always work. As we'll see shortly, when exploring the different samplers available, it's possible for applications to configure sampling differently from one another. This could cause applications to not respect the decision made by the root span, causing broken traces to be received by the backend. *Figure 12.2* shows five applications interacting and combining into a distributed system producing spans. It highlights what would happen if two applications, *B* and *C*, were configured to sample a trace, but the other applications in the system were not:

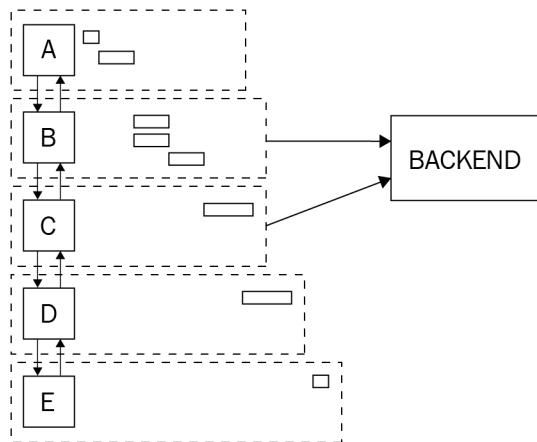


Figure 12.2 – Inconsistent sampling configuration

The backend would receive four spans and some context about the system but would be missing four additional spans and quite a bit of information.

Important Note

Inconsistent sampler configuration is a problem that affects all sampling strategies. Configuring multiple applications in a distributed system introduces the possibility of inconsistencies. Using a consistent sampling configuration across applications is critical.

Making a sampling decision at the very beginning of a trace can also cause valuable information to be missed. Continuing with the example from the previous diagram, if an error occurs in application *D*, but the sampling decision made by application *A* discards the trace, that error would not be reported to the backend. An inherent problem with head sampling is that the decision is made before all the information is available.

Tail sampling

If making the decision at the beginning of a trace is problematic because of a lack of information, what about making the decision at the end of a trace? **Tail sampling** is another common strategy that waits until a trace is complete before making a sampling decision. This allows the sampler to perform some analysis on the trace to detect potentially anomalous or interesting occurrences.

With tail sampling, all the applications in a distributed system must produce and transmit the telemetry to a destination that decides to sample the data or not. This can become costly for large distributed systems. Depending on where the tail sampling is performed, this option may cause significant amounts of data to be produced and transferred over the network, which could have little value.

Additionally, to make sampling decisions, the sampler must buffer in memory or store the data for the entire trace until it is ready to decide. This will inevitably lead to an increase in memory and storage consumed, depending on the size and duration of traces. As mitigation around memory concerns, a maximum trace duration can be configured in tail sampling. However, this leads to data gaps for any traces that never finish within that set time. This is problematic as those traces can help identify problems within a system.

Probability sampling

As discussed earlier in the chapter, **probability sampling** ensures that data is selected randomly, removing bias from the data sampled. Probability sampling is somewhat different from head and tail sampling, as it is both a configuration that can be applied to those other strategies and a strategy in itself. The sampling decision can be made by each component in the system individually, so long as the components share the same algorithm for applying the probability. In OpenTelemetry, the `TraceIdRatioBased` sampler (<https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/trace/sdk.md#traceidratiobased>) combined with the standard random trace ID generator provides a mechanism for probability sampling. The decision to sample is calculated by applying a configurable ratio to a hash of the trace ID. Since the trace ID is propagated across the system, all components configured with the same ratio and the `TraceIdRatioBased` sampler can apply the same logic at decision time independently:

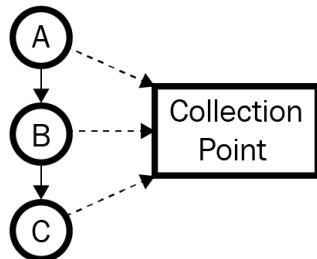


Figure 12.3 – Probabilistic sampling decisions can be applied at every step of the system

There are other sampling strategies available, but these are the ones we'll concern ourselves with for the remainder of this chapter.

Samplers available

There are a few different options when choosing a sampler. The following options are defined in the OpenTelemetry specification and are available in all implementations:

- **Always on:** As the name suggests, the `always_on` sampler samples all traces.
- **Always off:** This sampler does not sample any traces.
- **Trace ID ratio:** The trace ID ratio sampler, as discussed earlier, is a type of probability sampler available in OpenTelemetry.
- **Parent-based:** The parent-based sampler is a sampler that supports the head sampling strategy. The parent-based sampler can be configured with `always on`, `always_off`, or with a trace ID ratio decision as a fallback, when a sampling decision has not already been made for a trace.

Using the OpenTelemetry Python SDK will give us a chance to put these samplers to use.

Sampling at the application level via the SDK

Allowing applications to decide what to sample, provides a great amount of flexibility to application developers and operators, as these applications are the source of the tracing data. Samplers can be configured in OpenTelemetry as a property of the tracer provider. In the following code, a `configure_tracer` method configures the OpenTelemetry tracing pipeline and receives `Sampler` as a method argument. This method is used to obtain three different tracers, each with its own sampling configuration:

- `ALWAYS_ON`: A sampler that always samples.
- `ALWAYS_OFF`: A sampler that never samples.
- `TraceIdRatioBased`: A probability sampler, which in the example is configured to sample traces 50% of the time.

The code then produces a separate trace using each tracer to demonstrate how sampling impacts the output generated by `ConsoleSpanExporter`:

sample.py

```
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor,
    ConsoleSpanExporter
from opentelemetry.sdk.trace.sampling import ALWAYS_OFF,
    ALWAYS_ON, TraceIdRatioBased

def configure_tracer(sampler):
    provider = TracerProvider(sampler=sampler)
    provider.add_span_processor(BatchSpanProcessor(ConsoleSpanExporter()))
    return provider.get_tracer(__name__)

always_on_tracer = configure_tracer(ALWAYS_ON)
always_off_tracer = configure_tracer(ALWAYS_OFF)
ratio_tracer = configure_tracer(TraceIdRatioBased(0.5))

with always_on_tracer.start_as_current_span("always-on") as span:
    span.set_attribute("sample", "always sampled")
```

```
with always_off_tracer.start_as_current_span("always-off") as span:  
    span.set_attribute("sample", "never sampled")  
  
with ratio_tracer.start_as_current_span("ratio") as span:  
    span.set_attribute("sample", "sometimes sampled")
```

Run the code using the following command:

```
$ python sample.py
```

The output should do one of the following:

- Contain a trace with a span named `always-on`.
- Not contain a trace with a span named `always-off`.
- Maybe contain a trace with a span named `ratio`. You may need to run the code a few times to get this trace to produce output.

The following sample output is abbreviated to only show the name of the span and significant attributes:

output

```
{  
    "name": "ratio",  
    "attributes": {  
        "sample": "sometimes sampled"  
    },  
}  
{  
    "name": "always-on",  
    "attributes": {  
        "sample": "always sampled"  
    },  
}
```

Note that although the example configures three different samplers, a real-world application would only ever use one sampler. An exception to this is a single application containing multiple services with separate sampling requirements.

Note

In addition to configuring a sampler via code, it's also possible to configure it via the OTEL_TRACES_SAMPLER and OTEL_TRACES_SAMPLER_ARG environment variables.

Using application configuration allows us to use head sampling, but individual applications don't have the information needed to make tail sampling decisions. For that, we need to go further down the pipeline.

Using the OpenTelemetry Collector to sample data

Configuring the application to sample traces is great, but what if we wanted to use tail sampling instead? The OpenTelemetry Collector provides a natural point where sampling can be performed. Today, it supports both tail sampling and probabilistic sampling via processors. As we've already discussed the probabilistic sampling processor in *Chapter 8, The OpenTelemetry Collector*, we'll focus this section on the tail sampling processor.

Tail sampling processor

In addition to supporting the configuration of sampling via specifying a probabilistic sampling percentage, the tail sampling processor can make sampling decisions based on a variety of characteristics of a trace. It can choose to sample based on one of the following:

- Overall trace duration
- Span attributes' values
- Status code of a span

To accomplish this, the tail sampling processor supports the configuration of policies to sample traces. To better understand how tail sampling can impact the tracing data produced by configuring a variety of policies in the collector, let's look at the following code snippet, which configures a collector with the following:

- The OpenTelemetry protocol listener, which will receive the telemetry from an example application
- A logging exporter to allow us to see the tracing data in the terminal
- The tail sampling processor with a policy to always sample all traces

The following code snippet contains the elements of the previous list:

config/collector/config.yml

```
receivers:  
  otlp:  
    protocols:  
      grpc:  
  
exporters:  
  logging:  
    loglevel: debug  
  
processors:  
  tail_sampling:  
    decision_wait: 5s  
    policies: [{ name: always, type: always_sample }]  
  
service:  
  pipelines:  
    traces:  
      receivers: [otlp]  
      processors: [tail_sampling]  
      exporters: [logging]
```

Start the collector using the following command, which includes the configuration previously shown:

```
$ ./otelcol-contrib --config ./config/collector/config.yml
```

Next, the ensuing code is an application that will send multiple traces to the collector to demonstrate some of the capabilities of the tail sampling processor:

multiple_traces.py

```
import time  
from opentelemetry import trace
```

```
tracer = trace.get_tracer_provider().get_tracer(__name__)
with tracer.start_as_current_span("slow-span"):
    time.sleep(1)

for i in range(0, 20):
    with tracer.start_as_current_span("fast-span"):
        pass
```

Open a new terminal and start the program using OpenTelemetry auto-instrumentation, as per the following command:

```
$ opentelemetry-instrument python multiple_traces.py
```

Looking through the output in the collector terminal, you should see a total of 21 traces being emitted. Let's now update the collector configuration to only sample 10% of all traces. This can be configured via a policy, as per the following:

config/collector/config.yml

```
processors:
  tail_sampling:
    decision_wait: 5s
  policies:
    [
      {
        name: probability,
        type: probabilistic,
        probabilistic: { sampling_percentage: 10 },
      },
    ]
```

Restart the collector and run `multiple_traces.py` once more to see the effects of applying the new policy. The results should show roughly 10% of traces, which in this case would be about two traces. I say *roughly* here because the configuration relies on probabilistic sampling using the trace identifier. Since the trace ID is randomly generated, there is some variance in the results with such a small sample set. Run the command a few times if needed to see the sampling policy in action:

output

Span #0	
Trace ID	: 9581c95ae58bc8368050728f50c32f73
Parent ID	:
ID	: b9c3fb8838eb0f33
Name	: fast-span
Kind	: SPAN_KIND_INTERNAL
Start time	: 2021-12-28 21:29:01.144907 +0000 UTC
End time	: 2021-12-28 21:29:01.144922 +0000 UTC
Status code	: STATUS_CODE_UNSET
Status message	:
Span #0	
Trace ID	: 2a8950f2365e515324c62dfdc23735ba
Parent ID	:
ID	: c5217fb16c4d90ff
Name	: fast-span
Kind	: SPAN_KIND_INTERNAL
Start time	: 2021-12-28 21:29:01.14498 +0000 UTC
End time	: 2021-12-28 21:29:01.144996 +0000 UTC
Status code	: STATUS_CODE_UNSET
Status message	:

Note that in the previous output, only the spans named `fast-span` were emitted. It's unfortunate, because the information about `slow-span` may be more useful to us. It's additionally possible to configure the tail sampling processor to combine policies to create more complex sampling decisions.

For example, you may want to continue capturing only 10% of all traces but always capture traces representing operations that took longer than 1 second to complete. In this case, the following combination of a latency-based policy with a probabilistic policy would make this possible:

config/collector/config.yml

```
processors:  
  tail_sampling:  
    decision_wait: 5s  
    policies:  
      [  
        {  
          name: probability,  
          type: probabilistic,  
          probabilistic: { sampling_percentage: 10 },  
        },  
        { name: slow, type: latency, latency: { threshold_ms: 1000 } },  
      ]
```

Restart the collector one last time and run the example code. You'll notice that both a percentage of traces and the trace containing `slow`-span are visible in the output from the collector. There are other characteristics that can be configured, but this gives you an idea of how the tail sampling processor works. Another example is to base the sampling decision on the status code, which is a convenient way to capture errors in a system. Another yet is to sample custom attributes, which could be used to scope the sampling to specific systems.

Important Note

Choosing to sample traces on known characteristics introduces bias in the selection of spans that could inadvertently hide useful telemetry. Tread carefully when configuring sampling to use non-probabilistic data as it may exclude more information than you'd like. Combining probabilistic and non-probabilistic sampling, as in the previous example, allows us to work around this limitation.

Summary

Understanding the different options for sampling provides us with the ability to manage the amount of data produced by our applications. Knowing the trade-offs of different sampling strategies and some of the methods available helps decrease the level of noise in a busy environment.

The OpenTelemetry configuration and samplers available to configure sampling at the application level can help reduce the load and cost upfront in systems via head sampling. Configuring tail sampling at collection time provides the added benefit of making a more informed decision on what to keep or discard. This benefit comes at the added cost of having to run a collection point with sufficient resources to buffer the data until a decision can be reached.

Ultimately, the decisions made when configuring sampling will impact what data is available to observe what is happening in a system. Sample too little and you may miss important events. Sample too much and the cost of producing telemetry for a system may be too high or the data too noisy to search through. Sample only for known issues and you may miss the opportunity to find abnormalities you didn't even know about.

During development, sampling 100% of the data makes sense as the volume is low. In production, a much smaller percentage of data, under 10%, is often representative of the data as a whole.

The information in this chapter has given us an understanding of the concepts of sampling. It has also given us an idea of the trade-offs in choosing different sampling strategies. In the end, choosing the right strategy requires experimenting and tweaking as we learn more about our systems.

Index

A

agent 62
agent deployment 267
aggregation
 about 14, 155, 156
 methods 155
always off sampler 337
always on sampler 337
Amazon Elastic Kubernetes Service
 URL 266
analysis 13
Apache Flume
 URL 8
application level sampling
 configuring, via OpenTelemetry
 SDK 338-340
application metrics
 reference link 315
application telemetry
 collecting 267, 268
 sidecar, deploying 269-271
asynchronous counter 140, 141
asynchronous gauge 147, 148
asynchronous instruments 137
asynchronous up/down counter 143-145

attributes 37
attributes processor
 about 241
 delete operation 241
 extract operation 241
 hash operation 241
 insert operation 241
 update operation 241
 upsert operation 241
auto-instrumentation
 about 60
 command-line options 204
 components 61, 62
 configuring 198-201
 environment variables 203
 limitations 62, 63
 OpenTelemetry configurator 202, 203
 OpenTelemetry distribution 201, 202
 reference link 226
auto-instrumentation, in Java
 monkey patching 66
 runtime hooks 66
auto-instrumentation, in Python
 Instrumentor interface 67, 68
 libraries, instrumenting 66, 67
 wrapper script 68, 71

automatic configuration
about 211
logs, configuring 216, 217
metrics, configuring 215
propagation, configuring 217, 218
resource attributes, configuring 211, 212
traces, configuring 213-215
Azure Kubernetes Service
 URL 266

B

ballast extension 246, 248
BaseDistro interface
 about 201
 reference link 202
basicConfig method, of logging module
 reference link 184
batch processor 245
BatchSpanProcessor 214
Byte Buddy
 URL 65

C

cardinality 47
cardinality explosion 47
centralized logging 8
chaos engineering
 about 311-313
 latency 313
 URL 311
cloud-based providers 5

cloud-native applications 4
Cloud Native Computing
 Foundation (CNCF) 295
cloud-native software
 observability 15
cloud providers 4
collector
 deploying, benefits 234, 235
command-line options 204
composite propagator 110-115
ConfigMap 274
configuration options
 excluded_urls 206
 name_callback 206
 span_callback 206
conflicting instruments
 handling 148
Context API
 about 82-90
 attach 83
 detach 83
 get_value 83
 set_value 83
context propagation
 about 23, 24, 106-109
 formats 109
ContextVar module 23
counter 138, 139
create, read, update, and delete (CRUD)
 reference link 221
cumulative aggregation 43
cumulative sum 43

D

DaemonSet 267, 273
Dapper
 reference link 9
dashboards
 using 9
data
 enriching 92-94
data point type
 histogram 44, 45
 sum 43
 summary 45, 46
data sampling
 with OpenTelemetry Collector 340
decorator 89
delta aggregation 43
DevOps 6
dimension 152-154
distributed tracing 33, 187, 189
Docker Compose
 about 28
 reference link 28
double instrumentation 210

E

entry points
 reference link 201
environment variables 203
event
 about 117
 recording 116
exception
 about 118-122
 recording 116

exemplars 47
exporters 21, 247, 248
extensions
 about 248
 ballast 248
 Health_check 248
 pprof 248
 zpages 248

F

filter processor 242
Flask
 about 49
 OpenTelemetry logging 189, 190
Flask documentation
 reference link 103
Flask library instrumentor
 about 225
 configuration options 225
Fluentd
 URL 8

G

gauge 44
GDB
 reference link 9
General Data Protection
 Regulation (GDPR) 308
golden signals
 reference link 158
Google Cloud Platform (GCP)
 resource detector 22
Google Kubernetes Engine
 URL 266

Grafana

about 31, 303-305
reference link 303

Graphite

URL 9

grocery store application

about 157, 158, 219-221
concurrent number of requests
metric 167, 168
legacy inventory service 218, 219
number of requests metric 161
number of requests metrics 158-161
request duration metric 162-166
resource consumption metric 169-171
revisiting 218
shopper application 221-225

H

head sampling 334, 335
Health_check extension 248
Helm Charts
reference link 272
Helm website
URL 266
histogram 145, 146
HorizontalPodAutocaling 282
host metrics receiver 238

I

instrumentation libraries
finding 226
opentelemetry-bootstrap 227
OpenTelemetry registry 226

J**Jaeger**

about 30, 295-299
agent 295
Collector 295
ingester 295
query 295
reference link 295

Java archive (JAR) file 64
Java Instrumentation API
reference link 63

K

Kubernetes
URL 264

L

latency
about 313, 314
experiment 314
hypothesis 315
verifying 315-318
legacy inventory service 218, 219
LogEmitter
about 176
using 177-180
LogEmitterProvider 176
log files 48
logging pipeline
components 175, 176
logging signal
about 175
working 185, 186

- LogProcessor 176
LogRecord
 fields 177, 178
logs
 about 48, 187, 302
 anatomy 48-50
 configuring 216, 217
 considerations 52
 correlating 50-52
 Grafana 303-305
 Loki 302, 303
 producing 177
 semantic conventions 52
Logstash
 URL 8
Loki 31, 302, 303
Loki, components
 distributor 302
 ingester 302
 querier 302
 query frontend 302
 ruler 302
- M**
- manual instrumentation
 challenges 60, 61
manual invocation 206-210
measure 13
measurement 13
memory limiter processor 246
message 49
meter
 about 129
 obtaining 132-134
MeterProvider 129
MetricExporter 130
metric outputs, customizing with views
 aggregation 155, 156
 dimension 152-154
 filtering 149-152
MetricReader 130
metrics
 about 39
 anatomy 40, 42
 collecting, from applications 14
 configuring 215
 considerations 47, 48
 data point type 42, 43
 exemplars 47
 using 8, 9
metrics pipeline
 configuring 129-132
 meter, obtaining 132-134
 MeterProvider 129
 MetricExporter 130
 MetricReader 130
 pull-based exporting 134-136
 push-based exporting 134-136
 Views 129
metrics signal
 about 299
 Prometheus 300, 301
monkey patching
 reference link 66
monolith architecture
 versus microservice architecture 5, 6
monoliths, deploying to cloud provider
 challenges 5

N

Node 267
none values 97
non-probabilistic sampling
 about 332
 reference link 332
null values 97

O

observability
 about 3, 7
 history, reviewing 7
OpenCensus
 about 10, 13
 collector data flow 13
 URL 10, 13
OpenCensus Service
 URL 235
OpenMetrics
 reference link 46
Open-source telemetry backends
 exploring 288
 logs, analyzing 302
 metrics, analyzing 299
 traces, analyzing 289
OpenTelemetry
 components 216
 history 10
 log severity levels 179
opentelemetry-bootstrap 227

OpenTelemetry Collector
 collector, configuring 254-258
 exporter, configuring 253
 metrics, filtering 259-262
 need for 234
 spans, modifying 258, 259
 used, for sampling data 340
 using 252
OpenTelemetry Collector, components
 about 235, 236
 additional components 249
 exporters 247, 248
 extensions 248, 249
 processors 239-241
 receivers 236-238
opentelemetry-collector-
 contrib repository
 reference link 249
OpenTelemetry, concepts
 about 16
 context propagation 23, 24
 pipelines 20
 resources 22
 signals 16
OpenTelemetry configurator 202, 203
OpenTelemetry distribution 201, 202
OpenTelemetry Enhancement
 Proposal (OTEPE) 17, 60
OpenTelemetry instrument
 asynchronous counter 140, 141
 asynchronous gauge 147, 148

- asynchronous up/down
 - counter 143, 145
- counter 138, 139
- duplicate instrument 148
- histogram 145, 146
- listing 137
- selecting 137
 - up/down counter 142, 143
- OpenTelemetry Java Agent 64-66
- OpenTelemetry logging
 - configuring 175-177
 - with Flask 189, 190
- OpenTelemetry Protocol (OTLP)
 - design considerations 251
 - encodings 251
 - protocols 251
 - telemetry, transporting via 249, 251
- opentelemetry-python-contrib repository
 - reference link 205
- OpenTelemetry registry
 - about 226
 - reference link 226
- OpenTelemetry SDK
 - used, for configuring application
 - level sampling 338-340
- OpenTracing
 - about 10-12
 - URL 9
- P**
 - parent-based sampler 337
 - parent identifier 34
 - PDB
 - reference link 9
- percentile 45
- Personally Identifiable Information (PII) 100, 242
- pipelines 20
- Pod 267
- pprof extension 248
- probabilistic sampling processor 243
- probability sampling
 - about 332, 336
 - reference link 332
- processors
 - about 21, 239-241
 - attributes processor 241
 - batch processor 245
 - filter processor 242
 - memory limiter processor 246
 - probabilistic sampling processor 243
 - resource processor 244
 - span processor 244
- Prometheus
 - about 31, 300, 301
 - reference link 134
- Prometheus Query Language (PromQL) 300
- Prometheus server 300
- Prometheus web UI 300
- propagation
 - configuring 217, 218
- Propagators API 107
- protobufs
 - about 250
 - reference link 249
- provider 21
- psutil
 - reference link 169

pull-based exporting 134-136
push-based exporting 134-136
Python 3.7+ 23
Python docs
 reference link 180

Q

quantile 45

R

receivers
 about 236-238
 host metrics receiver 238
requests library
 reference link 101
Requests library instrumentor
 about 205
 additional configurable options 206
 configuration options 206
 double instrumentation 210
 manual invocation 206-210
resident set size
 reference link 147
resource attributes
 configuring 211, 212
resource correlation 192, 193
ResourceDetector 94, 95, 96
resource pressure
 about 318
 experiment 319
 hypothesis 319
 verifying 320, 322
resource processor 244

resources 22
resource usage information
 reference link 169
root span 35
runner 62

S

sampler
 options 337
sampling
 about 331
 concepts 331
 methods 332
sampling, across signals of OpenTelemetry
 logging 333
 metrics 333
 traces 332
sampling methods
 non-probabilistic 332
 probabilistic 332
sampling strategies
 about 334
 head sampling 334, 335
 probability sampling 336
 tail sampling 336
schema URL 54, 55, 133
semantic conventions
 about 52, 53
 adopting 53
 schema URL 54, 55
service level agreements (SLAs) 39
service level indicators (SLIs)
 about 39
 reference link 39

service level objectives (SLOs)
about 39
reference link 39

shopper application 221-225

Sidecar 267

sidecar deployment 267

signals
about 16
API 17
data model 17
instrumentation libraries 19, 20
SDK 18
semantic conventions 18, 19
specification 17

site reliability engineering (SRE)
about 39, 158
reference link 39

software development kit (SDK) 13, 129

span 34

span attributes 96-100

span_context
key information 106

SpanContext 34

SpanContext, trace element
span ID 34
trace flags 34
trace ID 34

SpanKind
about 100-105
CLIENT 100
CONSUMER 101
INTERNAL 100
PRODUCER 100
SERVER 100

span processor 90, 91, 244

spans 137

span status codes
ERROR 123
OK 123
UNSET 123

special interest groups (SIGs) 15

SQLite
reference link 289

standalone service
about 279-281
autoscaling 282

standard logging library 180-184

StatsD
about 134
URL 9

status
about 122-124
recording 116

synchronous instruments 137

System Architecture diagram 299

system-level telemetry
about 272
agent, connecting 274-276
agent, deploying 272, 273
resource attributes, adding 277-279
sidecar, connecting 274-276

system properties 64

T

tags 37

tail sampling 336

tail sampling processor 340-344

telemetry
transporting, via OTLP 249, 251
using 326-328

telemetry backends, in production
 data retention 307
 high availability 306
 privacy regulations 308
 running, considerations 306
 scalability 306
telemetry generators 21
telemetry pipeline
 provider 21
time-series database (TSDB) 300
timestamp 48
Trace Context
 reference link 34
trace-flags
 reference link 106
trace ID ratio sampler 337
Tracer 12, 79, 80, 137
TracerProvider interface
 defining 79
traces
 about 33, 34
 anatomy 34-37
 components 213
 configuring 213-215
 considerations 38
 span 37, 38
Trace state field 34
tracestate header
 reference link 106
tracing application
 applying 9, 10
tracing data
 Context API 82-90
 generating 80-82
 span processor 90, 91

tracing pipeline
 configuring 77, 78
tracing signal
 about 289, 290
 Jaeger 295-299
 Zipkin 291-294
traditional monolithic systems
 challenges 6
traffic control
 URL 314

U

unexpected shutdown
 about 323
 experiment 323
 hypothesis 323
 verifying 324, 325
Unix utility stress
 reference link 319
up/down counter 142, 143

V

views
 about 14
 used, for customizing
 metric outputs 149

W

W3C Trace Context 109
World Wide Web Consortium (W3C) 34
WSGI middleware
 using, for logging 191

Y

YAML 277

Z

Zipkin

about 291-294

reference link 291

Zipkin, core components

collector 291

query service or API 291

storage 291

web UI 291

zpages extension 248



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

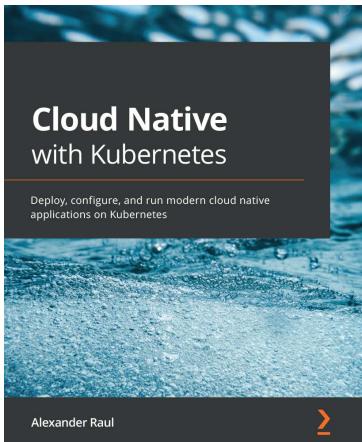
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

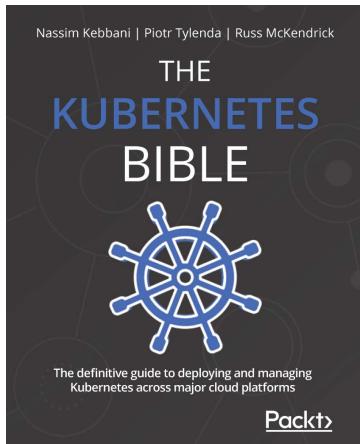


Cloud Native with Kubernetes

Alexander Raul

ISBN: 9781838823078

- Set up Kubernetes and configure its authentication
- Deploy your applications to Kubernetes
- Configure and provide storage to Kubernetes applications
- Expose Kubernetes applications outside the cluster
- Control where and how applications are run on Kubernetes
- Set up observability for Kubernetes
- Build a continuous integration and continuous deployment (CI/CD) pipeline for Kubernetes
- Extend Kubernetes with service meshes, serverless, and more



The Kubernetes Bible

Nassim Kebbani, Piotr Tylenda, Russ McKendrick

ISBN: 9781838827694

- Manage containerized applications with Kubernetes
- Understand Kubernetes architecture and the responsibilities of each component
- Set up Kubernetes on Amazon Elastic Kubernetes Service, Google Kubernetes Engine, and Microsoft Azure Kubernetes Service
- Deploy cloud applications such as Prometheus and Elasticsearch using Helm charts
- Discover advanced techniques for Pod scheduling and auto-scaling the cluster
- Understand possible approaches to traffic routing in Kubernetes

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Cloud-Native Observability with OpenTelemetry*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

