OpenTelemetry In Practice

by the OpenTelemetry Authors
presented by [your names here]

Our agenda

- What is observability?
- How does OpenTelemetry relate to observability?
- What concepts do I need to use OpenTelemetry?
- How do I record data using OpenTelemetry?
- Where can I send my data?

Level-setting

- Are you responsible for writing software?
- Are you responsible for operating software?

- Have you used distributed tracing before?
- Have you used OpenCensus?
- Have you used OpenTracing?

Slides for today

- You will need to refer to material from the slides during the interactive work period.
- Please find a copy of the slides here:

[QR code]

Who are we?

- [add your presenter information here]
- [add your presenter information here]

Observability Basics

Why observability?

- Microservices create complex interactions.
- Failures don't exactly repeat.
- Debugging multi-tenancy is painful.
- Monitoring no longer can help us.

What is observability?

- We need to answer questions about our systems.

What characteristics did the queries that timed out at 500ms share in common? Service versions? Browser plugins?

- Instrumentation produces data.
- Querying data answers our questions.

Telemetry aids observability

- Telemetry data isn't observability itself.
- Instrumentation code is *how* we get telemetry.
- Telemetry data can include traces, logs, and/or metrics.

All different *views* into the same underlying truth.

Metrics, logs, and traces, oh my!

- Metrics
  - Aggregated summary statistics.
- Logs
  - Detailed debugging information emitted by processes.
- Distributed Tracing
  - Provides insights into the full lifecycles, aka **traces** of requests to a system, allowing you to pinpoint failures and performance issues.

Structured data can be transmuted into any of these!

Metrics concepts in a nutshell

- Gauges
  - Instantaneous point-in-time value (e.g. CPU utilization)
- Cumulative counters
  - Cumulative sums of data since process start (e.g. request counts)
- Cumulative histogram
  - Grouped counters for a range of buckets (e.g. 0-10ms, 11-20ms)
- Rates
  - The derivative of a counter, typically. (e.g. requests per second)
- Aggregation by tags
  - Data can be joined along shared tags (e.g. hostname, cluster name).

Tracing concepts in a nutshell

- Span
  - Represents a **single unit of work** in a system.
  - Typically encapsulates: operation name, a start and finish timestamp, the parent span identifier, the span identifier, and context items.
- Trace
  - Defined implicitly by its **spans**. A **trace** can be thought of as a directed acyclic graph of **spans** where the edges between **spans** are defined as parent/child relationships.
- DistributedContext
  - Contains the tracing identifiers, tags, and options that are propagated from parent to child **spans**

Add more context to traces with Span Events

- Span Events are context-aware logging.
- An `event` contains timestamped information added to a span. You can think of this as a structured log, or a way to annotate your spans with specific details about what happened along the way.
  - Contains:
    - the name of the event
    - one or more attributes
    - a timestamp

But how do I implement these?

- You need an instrumentation framework!
- and a place to send the data!
- and a way to visualize the data!

About OpenTelemetry

OpenCensus + OpenTracing = OpenTelemetry

- OpenTracing:
  - Provides APIs and instrumentation for distributed tracing
- OpenCensus:
  - Provides APIs and instrumentation that allow you to collect application metrics and distributed tracing.
- OpenTelemetry:
  - An effort to combine distributed tracing, metrics and logging into a single set of system components and language-specific libraries.

OTel Language SDKs

- OpenTelemetry is almost in **beta** -
  - Spec/API is being finalized.
  - Language SDKs are catching up. The following are expected to be in beta on 3/16/20.
    - C#
    - Go
    - JavaScript (Browser/Node)
    - Java
    - Python

OTel API - packages, methods, & when to call

- Tracer
  - A Tracer is responsible for tracking the currently active span.
- Meter
  - A Meter is responsible for accumulating a collection of statistics.

You can have more than one. Why?

Ensures uniqueness of name prefixes.

Code examples: Providers

- A global provider can have a TraceProvider registered.
- Use the TraceProvider to create a named tracer.

```
// Register your provider in your init code
tp, err := sdktrace.NewProvider(...)
global.SetTraceProvider(tp)
// Create the named tracer
tracer = global.TraceProvider().Tracer("workshop/main")
```

Code examples: Providers (Python)

```
// changing in beta to TracerProvider
trace.set_preferred_tracer_source_implementation(lambda T: TracerSource())

// initialize tracer for the process
tracer = trace.get_tracer(__name__)
```

OTel API - Tracer methods, & when to call

- tracer.Start(ctx, name, options)
  - This method returns a child of the current span, and makes it current.
- tracer.WithSpan(name, func() {...})
  - Starts a new span, sets it to be active in the context, executes the wrapped body and closes the span before returning the execution result.
- trace.SpanFromContext(ctx)
  - Used to access & add information to the current span

OTel API (Python) - Tracer methods, & when to call

- tracer.start_span(name, parent=, ...)
  - This method returns a child of the specified span.
- with tracer.start_as_current_span(name)
  - Starts a new span, sets it to be active. Optionally, can get a reference to the span.
- tracer.get_current_span()
  - Used to access & add information to the current span

OTel API - Span methods, & when to call

- span.AddEvent(ctx, msg)
  - Adds structured annotations (e.g. "logs") about what is currently happening.
- span.SetAttributes(core.Key(key).String(value)...)
  - Adds a structured, typed attribute to the current span. This may include a user id, a build id, a user-agent, etc.
- span.End()
  - Often used with defer, fires when the unit of work is complete and the span can be sent

OTel API (Python) - Span methods, & when to call

- span.add_event(name, attributes)
  - Adds structured annotations (e.g. "logs") about what is currently happening.
- span.set_attribute(key, value)
  - Adds an attribute to the current span. This may include a user id, a build id, a user-agent, etc.
- span.end()
  - Manually closes a span.

Code examples: Start/End

```
func (m Model) persist(ctx context.Context) {
    tr := global.TraceProvider().Tracer("me")
    ctx, span := tr.Start(ctx, "persist")
    defer span.End()

    // Persist the model to the database...
    [...]
}
```

Code examples (Python): Start/End

```
def persist(data):
    tracer = trace.get_tracer(__name__)
    tracer.start_as_current_span("persistData")
        // do work...
    return result
```

Code examples: WithSpan

Takes a context, span name, and the function to be called.

```
ret, err := tracer.WithSpan(ctx, "operation",
   func(ctx context.Context) (int, error) {
      // Your function here
      [...]
      return 0, nil
   }
)
```

Code examples: CurrentSpan & Span

- Get the current span
  - `sp := trace.SpanFromContext(ctx)`
- Update the span status
  - `sp.SetStatus(codes.OK)`
- Add events
  - `sp.AddEvent(ctx, "foo")`
- Add attributes
  - ```
    sp.SetAttributes(
        key.New("ex.com/foo").String("yes"))
    ```
    ```

Code examples: Current Span & Span [python]

- Get the current span
  - `span = tracer.get_current_span()`
- Update the span status
  - `span.set_status(Status(StatusCanonicalCode.UNKNOWN, error))`
- Add events
  - `span.add_event("foo", {"customer": "bar"})`
- Add attributes
  - `span.set_attribute("error", True)`

Context Propagation

- Distributed context is an abstract data type that represents collection of entries.
- Each key is associated with exactly one value.
- It is serialized for propagation across process boundaries
- Passing around the context enables related spans to be associated with a single trace.
- W3C TraceContext is the de-facto standard.

Automatic Instrumentation

OpenTelemetry has wrappers around common frameworks to propagate context and make it accessible.

```
import "go.opentelemetry.io/otel/plugin/othttp"

othttp.NewHandler(http.HandlerFunc(h), "h"))

func h(w ResponseWriter, req *Request) {
  ctx := req.Context()
  span := trace.SpanFromContext(ctx)
}
```

Automatic Instrumentation [python]

```
from opentelemetry.ext import http_requests
from opentelemetry.ext.flask import instrument_app

// instrument Requests library
http_requests.enable(trace.tracer_source())

// create flask app, then instrument
app = Flask(__name__)
instrument_app(app)
```

SDKs, Exporters, and Collector Services, Oh My!

- OpenTelemetry's **SDK** implements trace & span creation.
- An **exporter** can be instantiated to send the data collected by OpenTelemetry to the backend of your choice.
  - E.g. Jaeger, Lightstep, Honeycomb, Stackdriver, etc.
- OpenTelemetry **collector** proxies data between instrumented code and backend service(s). The exporters can be reconfigured without changing instrumented code.

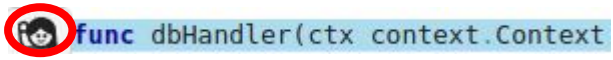
Vendor-neutral exporters

- Jaeger exporter
  - Jaeger was created at Uber and is now an open-source CNCF project
  - Stores and visualizes traces.
- Prometheus exporter
  - Prometheus is a TSDB inspired by Google's Borgmon
  - Exporter not working yet! OpenTelemetry's metrics support is alpha.
- stdout/stderr streaming export
  - Inspect what is actually being sent over the wire.
  - No external setup required!

Our interactive work today

Clone the Glitch repository

glitch.com/edit/#!/opentelemetry-student

- Glitch provisions a container, compiles, and runs code.
  - The first build will take 30 seconds to pull packages.
  - Subsequent builds are automatic and take seconds
- Glitch lets you raise a hand for help! 

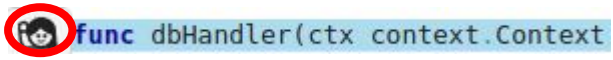
- Create an account to save your work!
- Do *not* use `go get -u` because it will put you on an untested `go.mod` combination.

Our example application

- mux.Handle("/", http.HandlerFunc(rootHandler))
  - Prints "Hello, World!"
- mux.Handle("/favicon.ico", http.NotFoundHandler())
  - 404s
- mux.Handle("/fib", http.HandlerFunc(fibHandler))
  - Returns /fib?i=n-1 + /fib?i=n-2

Clone the Glitch repository [python]

glitch.com/edit/#!/opentelemetry-python-student

- Glitch provisions a container, compiles, and runs code.
  - The first build will take 30 seconds to pull packages.
  - Subsequent builds are automatic and take seconds
- Glitch lets you raise a hand for help! 

- Create an account to save your work!

Our example application (Python)

- `@app.route(“/”)`
  - Returns some informational text.
- `@app.route(“/fib”) and @app.route(“/fibInternal”)`
  - /fib calculates fibonacci sequence by calling /fibInternal

Our job is to instrument this. How?

Golang

Add OTel imports and set up SDK

```
import "go.opentelemetry.io/otel/api/trace"
import "go.opentelemetry.io/otel/global"
import sdktrace "go.opentelemetry.io/otel/sdk/trace"

tp, err := sdktrace.NewProvider(
   sdktrace.WithConfig(sdktrace.Config{
      DefaultSampler: sdktrace.AlwaysSample()}))

global.SetTraceProvider(tp)
```

Add trace spans to the logic

- mux.Handle("/", http.HandlerFunc(rootHandler))
  - Wrap rootHandler with HTTP plugin
  - Add dbHandler internal span.
- mux.Handle("/fib", http.HandlerFunc(fibHandler))
  - Returns /fib?i=n-1 + /fib?i=n-2
    - Wrap the handler
    - Add attributes for the parameters
    - Create spans for each parallel client call
    - Propagate the context to downstream calls.

othttp instrumentation of root handler

```
import "go.opentelemetry.io/otel/plugin/othttp"

func main() {
    mux.Handle("/", othttp.NewHandler(
        http.HandlerFunc(rootHandler), "root"))

func rootHandler([...]) {
    ctx := req.Context()
    trace.SpanFromContext(ctx).AddEvent(ctx, "Ran root
handler.")
```

Internal spans & context propagation

```
func rootHandler([...]) {
    ctx := req.Context()
    dbHandler(ctx, "blue")

func dbHandler(ctx context.Context, color string) int {
    tr := global.TraceProvider().Tracer("dbHandler")
    ctx, span := tr.Start(ctx, "database")
    defer span.End()
```

Configure output to stdout

```
import "go.opentelemetry.io/otel/exporters/trace/stdout"

func main() {
    std, err := stdout.NewExporter(stdout.Options{
        PrettyPrint: true,
    })
    if err != nil {
        log.Fatal(err)
    }
    sdktrace.NewProvider(sdktrace.WithConfig([...]),
        sdktrace.WithSyncer(std))
```

Python

Add OTel imports and set up SDK

```
# server.py
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import (
    SimpleExportSpanProcessor,
    ConsoleSpanExporter
)

from opentelemetry.instrumentation.requests import RequestsInstrumentor
from opentelemetry.instrumentation.flask import FlaskInstrumentor
```

Add OTel imports and set up SDK

```
trace.set_tracer_provider(TracerProvider())
trace.get_tracer_provider().add_span_processor(
    SimpleExportSpanProcessor(
        ConsoleSpanExporter()
    )
)



tracer = trace.get_tracer(__name__)
```

Instrument Flask and Requests

```
app = Flask(__name__)

instrument_app(app)

RequestsInstrumentor().instrument(tracer_provider=trace.get_
tracer_provider())
```

What you should see...

Visit http://[appname].glitch.me, then go Tools -> Logs


```
{
        "SpanContext": {
                "TraceID": "9850b11fa09d4b5fa4dd48dd37f3683b",
                "SpanID": "1113d149cfffa942",
                "TraceFlags": 1
        },
        "ParentSpanID": "e1e1624830d2378e",
        "SpanKind": "internal",
        "Name": "dbHandler/database",
        "StartTime": "2019-11-03T10:52:56.903919262Z",
        "EndTime": "2019-11-03T10:52:56.903923338Z",
        "Attributes": [],
        "MessageEvents": null,
        "Links": null,
        "Status": 0,
        "HasRemoteParent": false,
        …
}
```


⬅️ Golang

Python ⬇️

```
Span(name="root", context=SpanContext(trace_id=0xe2b0888b60ef4828851aa290136d9978, span_id=0x960a301445cd7495,
trace_state={}), kind=SpanKind.SERVER, parent=SpanContext(trace_id=0xe2b0888b60ef4828851aa290136d9978,
span_id=0xa51ce1f847f9b967, trace_state={}), start_time=2020-03-03T00:17:03.789244Z,
end_time=2020-03-03T00:17:03.795240Z)
```

Understanding the output

JSON formatted info, output in order End() was called.


```
"SpanContext": {
      "TraceID": "9850b11fa09d4b5fa4dd48dd37f3683b",
      "SpanID": "1113d149cfffa942",
      "TraceFlags": 1
},
"ParentSpanID": "e1e1624830d2378e",
"SpanKind": "internal",
"Name": "dbHandler/database",
"StartTime": "2019-11-03T10:52:56.903919262Z",
"EndTime": "2019-11-03T10:52:56.903923338Z",
"Attributes": [],
"MessageEvents": null,
"Links": null,
"Status": 0,
"HasRemoteParent": false,
"DroppedAttributeCount": 0,
"DroppedMessageEventCount": 0,
"DroppedLinkCount": 0,
"ChildSpanCount": 0
```

Attributes & MessageEvents


```
"Attributes": [
      {
            "Key": "http.host",
            "Value": {
                  "Type": "STRING",
                  "Value": "opentelemetry-instructor.glitch.me"
            }
      },
      {
            "Key": "http.status_code",
            "Value": {
                  "Type": "INT64",
                  "Value": 200
            }
      }
],
"MessageEvents": [
      {
            "Message": "annotation within span",
            "Attributes": null,
            "Time": "2019-11-03T10:52:56.903914029Z"
      }
],
```

Now, let's instrument /fib.

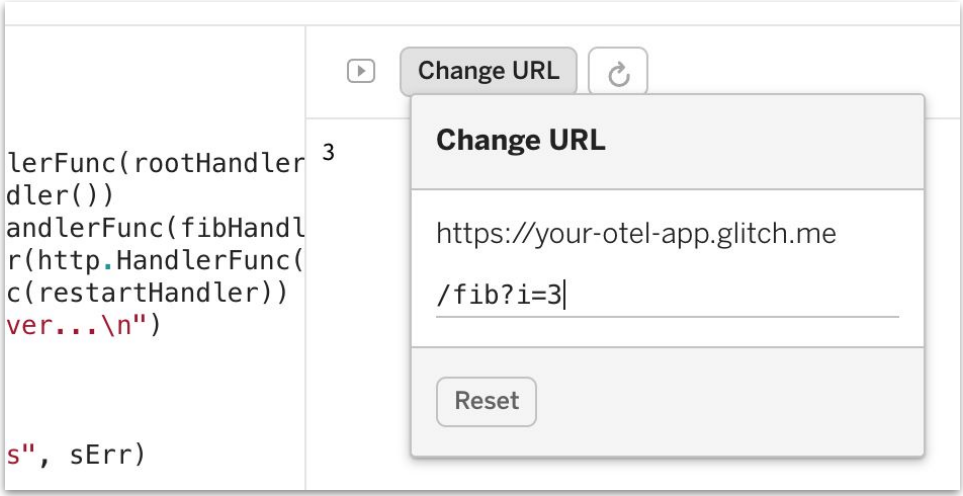
```
import "go.opentelemetry.io/otel/api/key"

func main() {
    mux.Handle("/fib", oththttp.NewHandler(
        http.HandlerFunc(fibHandler), "fib"))
    mux.Handle("/fibinternal", oththttp.NewHandler(
        http.HandlerFunc(fibHandler), "fibInternal"))

func fibHandler([...]) {
    ctx := req.Context()
    // Record the input value.
    trace.SpanFromContext(ctx).SetAttribute(key.Int("input", i))
    [...]
    trace.SpanFromContext(ctx).SetAttribute(key.Int("result", ret))
```

Have you tested the fibonacci output yet?

- Hit `http://[appname].glitch.me/fib?i=3`
  - In a new browser window, or
  - By changing the URL via the “Change URL button”



We'll also want client info.

```
import "go.opentelemetry.io/otel/plugin/httptrace"
func fibHandler([...]) {
    [...]
    clientCall := func(ictx context.Context) {
        trace.SpanFromContext(ictx).SetAttribute(key.Int("req", i))
        req, _ := http.NewRequestWithContext(ictx, "GET", url, nil)
        ictx, req = httptrace.W3C(ictx, req)
        httptrace.Inject(ictx, req)
        res, err := client.Do(req)
    }
    err := tr.WithSpan(ctx, "fibClient", clientCall)
```

Exercise for the reader: record error statuses and results.

Python - Instrument Request Specifically

```
with tracer.start_as_current_span(“getMinusOne”) as span:
    span.set_attribute(“payloadValue”, value - 1)
    respOne = requests.get(‘http://127.0.0.1:5000/fibInternal’,
minusOnePayload)

with tracer.start_as_current_span(“getMinusTwo”) as span:
    span.set_attribute(“payloadValue”, value - 2)
    respTwo = requests.get(‘http://127.0.0.1:5000/fibInternal’,
minusTwoPayload)
```

Getting data out more usefully...

Some motivating challenges:

- How many times is /fibinternal?i=2 called when /fib?i=5 is called?
- Can you find the overhead of DNS compared to the overhead of server HTTP?
- Can you add another parameter to the root HTTP request, and send the value of that parameter as an Attribute to the backend?

Visualize using Jaeger

We've set up Jaeger to receive traces. First set in .env:

```
JAEGER_ENDPOINT=http://[hostname]:14268/api/traces

import "go.opentelemetry.io/otel/exporters/trace/jaeger"

jaegerEndpoint, _ := os.LookupEnv("JAEGER_ENDPOINT")
jExporter, err := jaeger.NewExporter(
    jaeger.WithCollectorEndpoint(jaegerEndpoint),
    jaeger.WithProcess(jaeger.Process{ServiceName: serviceName}),
)
tp, err := sdktrace.NewProvider(
    sdktrace.WithConfig([...]),
    sdktrace.WithSyncer(std), sdktrace.WithSyncer(jExporter)
)
```

Visualize using Jaeger (Python)

We've set up Jaeger in GCP to receive traces.

```
// .env
JAEGER_HOST="34.73.164.21"
// server.py
from opentelemetry.ext.jaeger import JaegerSpanExporter

jaegerExporter = JaegerSpanExporter(
    service_name=serviceName,
    agent_host_name=os.environ['JAEGER_HOST'],
    agent_port=6831,
)

trace.tracer_source().add_span_processor(
    SimpleExportSpanProcessor(jaegerExporter))
```

Go look for your trace!

The Jaeger visualization URL is at (notice the port number):

http://[hostname]:**16686**/search

Put in your `SERVICE_NAME` value into the service name, and search for your recent traces!

Ask your neighbor for their `SERVICE_NAME` and compare!

Trouble in paradise

- Jaeger thinks we're missing the root span.
- And we can verify by checking if our root spans have a `ParentID` (they do!).


```
{
    "SpanContext": {
        "TraceID": "9850b11fa09d4b5fa4dd48dd37f3683b",
        "SpanID": "e1e1624830d2378e",
        "TraceFlags": 1
    },
    "ParentSpanID": "ff33261fd1178603",
    "SpanKind": "server",
    "Name": "go.opentelemtry.io/plugin/othttp/root",
    [...]
```

Configure context propagation

- We're using HTTP headers to propagate context.
- Glitch also has its own headers. (can you find them?)
- We need to mark our public endpoint as a trace boundary.
- To solve this, `othttp.WithPublicEndpoint()` can be passed to `othttp.NewHandler()`
  - `[Instructor note]: this is broken between v0.2.2-v0.2.3, use instead for now the synonym: othttp.WithSpanOptions(trace.WithNewRoot())`

Plugging in your own exporter

- Initialize a custom exporter with an API key
  - examples: Stackdriver, Lightstep, Honeycomb, etc.
  - A current list of vendors working with OpenTelemetry can be found at https://opentelemetry.io/registry/

[vendor] exporter instructions

- Presenters can insert instructional material about their vendor's setup process here, if they so choose.
- [the workshop authors ask that if vendor material is included, that more than one vendor be highlighted to ensure people know they have options.]

Stackdriver exporter

- Prerequisites
  - Client secrets JSON file for the GCP Project
  - Put the JSON file in the path `$GOOGLE_APPLICATION_CREDENTIALS`

```
import "go.opentelemetry.io/otel/exporters/trace/stackdriver"

exporter, _ := stackdriver.NewExporter()
tp, _ := sdktrace.NewProvider(sdktrace.WithSyncer(exporter))
global.SetTraceProvider(tp)
```

Need any hints?

- Instructor code is at https://glitch.com/edit/#!/opentelemetry-instructor?path=src/main.go:1:1
- You can see our solutions there.

Further work if time allows...

Add metrics

- You'll want a metrics export pipeline
  - How about Prometheus?

```
import "go.opentelemetry.io/otel/exporters/metric/prometheus"

func main() {
    prom, metricsHandler, err := prometheus.InstallNewPipeline(
        prometheus.Config{
            DefaultSummaryQuantiles: []float64{0.5, 0.9, 0.99},
    })
    defer prom.Stop()
}
```

Add metrics

- Glitch cares very much about your disk usage etc.
  - So let's track total disk & used disk! And track CPU!
  - See glitch.com/edit/#!/opentelemetry-instructor?path=src/main.go:198:1

```
meter := global.MeterProvider().Meter("container")
mem := meter.NewInt64Gauge("mem_usage",
    metric.WithKeys(appKey, containerKey),
    metric.WithDescription("Amount of memory used."),
)
go func() {
    meter.RecordBatch(ctx, meter.Labels([...]),
        mem.Measurement(mem_used), [...])
}()
```

Instrument your own real applications

- OTel is ~beta right now.
  - Calling the API is safe to do.
  - SDKs may change, but default to no-op.
  - OTel is in use by unicorns and publicly traded companies, but…
  - We offer no prod stability guarantees at the moment.
- Your instructors and TAs are here to help!

Thank you!