● JANUARY 2026 SERIES

FROM GO BUILD
TO GO RUN

GOLANG 2026 - NIV RAVE

#50

DISTRIBUTED TRACING & OPENTELEMETRY

CONNECTING THE DOTS ACROSS A DISTRIBUTED LANDSCAPE

The Anatomy of a Trace

The Trace, The Span, and The Context

A Trace represents the entire journey of a request. It is made up of multiple Spans. A Span represents a single unit of work - like a database query or an internal function call.

Spans are parent-child relationships. If your *PlaceOrder* span contains a *CheckStock* span and a *ChargeCard* span, you can instantly see which part of the process is the bottleneck. The "glue" that holds these together is the Trace ID.

Why OpenTelemetry?

One Standard to Rule Them All

In the past, you had to choose between vendor-specific libraries (like Jaeger or Datadog). OpenTelemetry (OTel) is a vendor-neutral standard. You instrument your Go code once, and you can send that data to any backend you choose.

The Strategy:
Think of OTel as the "USB-C" of observability. It provides a unified set of APIs for traces, metrics, and logs, ensuring your instrumentation doesn't become technical debt if you switch monitoring providers.

Instrumenting a Function

Starting a Span

To trace a function, you "start" a span using a Tracer. This span is automatically linked to the incoming context, maintaining the parent-child relationship.

```go
func (s *Service) ProcessOrder(ctx context.Context, orderID string) error {
    // Start a new span for this specific operation
    ctx, span := s.tracer.Start(ctx, "ProcessOrder")
    defer span.End() // Always end the span!

    // Add metadata for debugging
    span.SetAttributes(attribute.String("order.id", orderID))

    return s.repo.Save(ctx, orderID)
}
```

Tracing Across the Wire

Injectors and Extractors

As we touched on in Day 20, the Trace ID must travel across HTTP or gRPC boundaries. OTel handles this using "Propagators".

The Logic:
The Client injects the Trace ID into the headers, and the Server extracts it. This ensures that the span started in Service B correctly identifies the span in Service A as its parent. This is how you get a single, continuous "Flame Graph" for a request.

Capturing Errors in Spans

Visibility Into Failure

A trace isn't just for timing; it's for status. When a function returns an error, you should record that error on the span. This turns the span "red" in your visualization tool, making it easy to spot failing nodes in a complex graph.

```go
if err != nil {
    // Record the error and set the span status
    span.RecordError(err)
    span.SetStatus(codes.Error, "failed to save order")
    return err
}
```

The Cost of Observability

Sampling Strategies

Tracing every single request in a high-traffic system can generate massive amounts of data and increase network costs. This is where Sampling comes in.

The Strategy:
A common pattern - you might choose to only trace 1% of successful requests but 100% of errors. OTel allows you to define these rules at the "Head" (when the request starts) or the "Tail" (after the request finishes), allowing you to balance deep visibility with resource efficiency.

The Collector Pattern

Offloading the Heavy Lifting

You don't want your Go application to spend CPU cycles compressing and sending traces to a remote server. Instead, your app sends traces to a local OTel Collector.

The Benefit:
The Collector acts as a buffer and a processor. It can scrub sensitive data, aggregate spans, and export them to multiple backends simultaneously. This keeps your application lean and focused purely on business logic.

Summary:

- **Traces vs. Logs**: Logs tell you what happened; Traces tell you where it slowed down.
- **OpenTelemetry**: Standardize your instrumentation to avoid vendor lock-in.
- **Context**: Use *context.Context* to carry the Span through your call stack.
- **Attributes**: Enrich spans with business data (IDs, Status) for powerful filtering.

Question: When looking at a trace, what do you look for first: the longest span (latency) or the presence of error attributes (reliability)? 👇