This guide is part of an ongoing series on observability for engineers and operators of distributed systems. We created Honeycomb to deliver the best observability to your team so you can ship code more quickly and with greater confidence.

www.honeycomb.io

We post frequently about topics related to observability, software engineering, and how to build, manage, and observe complex infrastructures in the modern world of microservices, containers, and serverless systems on our blog:

https://www.honeycomb.io/observability-blog/

This is the third guide in our highly-acclaimed observability series.

- Achieving Observability
- Observability for developers
Distributed Tracing – A Guide for Microservices and More

You don’t need a PHD to understand distributed tracing. Let’s explore.
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Why Trace?</td>
<td>3</td>
</tr>
<tr>
<td>A Bit of History</td>
<td>3</td>
</tr>
<tr>
<td>Tracing from Scratch</td>
<td>5</td>
</tr>
<tr>
<td>What are we looking for out of tracing?</td>
<td>5</td>
</tr>
<tr>
<td>How Do We Modify Our Existing Code To Get There?</td>
<td>7</td>
</tr>
<tr>
<td>Generating Trace Ids</td>
<td>8</td>
</tr>
<tr>
<td>Generating Timing Information</td>
<td>9</td>
</tr>
<tr>
<td>Setting Service Name And Span Name</td>
<td>9</td>
</tr>
<tr>
<td>Propagating Trace Information</td>
<td>10</td>
</tr>
<tr>
<td>Adding Custom Fields</td>
<td>12</td>
</tr>
<tr>
<td>All Together Now</td>
<td>12</td>
</tr>
<tr>
<td>Tracing with Beelines</td>
<td>14</td>
</tr>
<tr>
<td>That's really it?</td>
<td>14</td>
</tr>
<tr>
<td>What about the other services?</td>
<td>15</td>
</tr>
<tr>
<td>What about custom spans?</td>
<td>16</td>
</tr>
<tr>
<td>Querying Traces</td>
<td>16</td>
</tr>
<tr>
<td>What about Standards?</td>
<td>19</td>
</tr>
<tr>
<td>This Must Be the Trace</td>
<td>20</td>
</tr>
</tbody>
</table>
Why Trace?

Very few technologies have caused as much elation and pain for software engineers in the modern era as the advent of computer-to-computer networking. Since the first day we linked two computers together and made it possible for them to “talk”, we have been discovering the gremlins lurking within our programs and protocols. These issues persist in spite of our best efforts to stomp them out, and in the modern era, the rise of complexity from patterns like microservices is only making these problems exponentially more common and more difficult to identify.

Modern microservices architectures in particular exacerbate the well-known problems that any distributed system faces, like lack of visibility into a business transaction across process boundaries and so can especially benefit from the visibility offered via distributed tracing.

Much like a doctor needs high resolution imaging such as MRIs to correctly diagnose illnesses, modern engineering teams need observability over simple metrics monitoring to untangle this Gordian knot of software. Distributed tracing, which shows the relationships among various services and pieces in a distributed system, can play a key role in that untangling.

Sadly, tracing has gotten a bad reputation as something that requires PHD-level knowledge in order to decipher, and hair-yanking frustration to instrument and implement in production. Worse yet, there’s been a proliferation of tooling, standards, and vendors - what’s an engineer to do?

We at Honeycomb believe that tracing doesn't have to be an exercise in frustration. That's why we've made this guide for the rest of us to democratize tracing.
A Bit of History

Distributed tracing first started exploding into the mainstream with the publication of the Dapper paper out of Google in 2010. As the authors themselves say in the abstract, distributed tracing proved itself to be invaluable in an environment full of constantly-changing deployments written by different teams:

Modern Internet services are often implemented as complex, large-scale distributed systems. These applications are constructed from collections of software modules that may be developed by different teams, perhaps in different programming languages, and could span many thousands of machines across multiple physical facilities. Tools that aid in understanding system behavior and reasoning about performance issues are invaluable in such an environment.

Given that tracing systems had already been around for a while, Dapper cited two main innovations as a credit for its particular success:

- The use of **sampling** to keep the volume of traced requests under control
- The use of **common client libraries** to keep the cost of instrumentation under control

Not long after the publication of the Dapper paper, in 2012 Twitter released an open source project called Zipkin which contained their implementation of the system described in the paper. Zipkin functions as both a way to collect tracing data from instrumented programs, and to access information about the collected traces in a browser-based web app. Zipkin allowed many users to get their first taste of the world of tracing.
In 2017 Uber released Jaeger, a tracing system with many similarities to Zipkin, but citing these shortcomings as the reason for writing their own:

_Even though the Zipkin backend was fairly well known and popular, it lacked a good story on the instrumentation side, especially outside of the Java/Scala ecosystem. We considered various open source instrumentation libraries, but they were maintained by different people with no guarantee of interoperability on the wire, often with completely different APIs, and most requiring Scribe or Kafka as the transport for reporting spans._

Since then there has been a proliferation of various implementations, both proprietary and open source. We at Honeycomb naturally think Honeycomb is the best available due to Honeycomb's excellent support for information discovery and high cardinality data. We offer Beelines to make getting tracing data in easier than ever - but what are these doing behind the scenes? To understand the nuts and bolts of tracing, let's take a look at what it's like to build tracing instrumentation from scratch.

**Tracing from Scratch**

Distributed Tracing involves understanding the flow and lifecycle of a unit of work performed in multiple pieces across various components in a distributed system. It can also offer insight into the various pieces of a single program's execution flow without any network hops. To understand how the mechanics of this actually work in practice, we'll walk through an example here of what it might look like to ornament your app's code with the instrumentation needed to collect that data. We'll consider:

- The end result we're looking for out of tracing
- How we _might_ modify our existing code to get there
What are we looking for out of tracing?

In the modern era, we are working with systems that are all interdependent - if a database or a downstream service gets backed up, latency can "stack up" and make it very difficult to identify which component of the system is the root of the misbehavior. Likewise, key service health metrics like latency might mislead us when viewed in aggregate - sometimes systems actually return more quickly when they're misbehaving (such as by handing back rapid 500-level errors), not less quickly. Hence, it's immensely useful to be able to visualize the activity associated with a unit of work as a "waterfall" where each stage of the request is broken into individual chunks based on how long each chunk took, similar to what you might be used to seeing in your browser's web tools.

![Trace example](image)

Each chunk of this waterfall is called a **span** in tracing terminology. Spans are either the **root span**, i.e. the first one in a given trace, or they are nested within other one. You might hear this nesting referred to as a **parent-child** relationship -

---

**Service A's span is the root and takes the longest. It has one child.**

**Service B's span's parent is Service A's span.**

**Service C's span has no children.**
if Service A calls Service B which calls Service C, then in that trace A's spans would be the parent of B’s, which would be the parent of C’s.

Note that a given service call might have multiple spans associated with it - there might be an intense calculation worth breaking into its own span, for instance.

Our efforts in distributed tracing are mostly about generating the right information to be able to construct this view. To that end, there are six variables we need to collect for each span that are absolutely critical:

- An **ID** - so that a unique span can be referenced to lookup a specific trace, or to define parent-child relationships
- A **parent ID** - so we can reference the second field mentioned above to draw the nesting properly
  - For the root span, this is absent. That's how we know it is the root.
- The **timestamp** indicating when a span began
- The **duration** it took a span to finish
- The name of the **service** that generated this span
- The **name** of the span itself - e.g., it could be something like `intense_computation` if it represents an intense unit of work that is not a network hop

We need to generate all of this info and send it to our tracing backend somehow. But how?

**How Do We Modify Our Existing Code To Get There?**

Carl Sagan once said, *"If you wish to make an apple pie from scratch, you must first invent the universe."* The same is true of distributed tracing: a lot of context and instrumentation has to be set up for a tracing effort to be successful. To get a feel for the core component pieces that go into making even a naive tracing system, let's do a thought exercise - we'll write our own example tracing instrumentation from scratch! This will help illustrate why common client libraries are such a key innovation. We won't even cover the back-end/server side
component to collect and query the tracing data itself - we'll just assume one is available for us to write to using HTTP.

Maybe we have a very simple web endpoint. If we issue a GET request to it, it calls a couple of other services to get some data based on what's in the original request, such as whether or not the user is authorized to access the given endpoint, then writes some results back.

```go
func rootHandler(r *http.Request, w http.ResponseWriter) {
    authorized := callAuthService(r)
    name := callNameService(r)

    if authorized {
        w.Write([]byte(fmt.Sprintf(
            `{"message": "waddup %s"},
            name))
    } else {
        w.Write([]byte(
            `{"message": "not cool dawg"}`
        ))
    }
}
```

OK, so we would expect to see a minimum of three spans involved with calling this service in the end -

1. One for the originating root request to `fooHandler`
2. One for the call to the authorization service
3. One for the call to the name service to get the user's name

**Generating Trace Ids**

First things first - let's generate a trace ID to indicate that the span data we generate and send to the back end can be united together later by a shared trace ID. We'll use a [UUID](https://en.wikipedia.org/wiki/UUID) to ensure that collisions of IDs are nigh impossible. We'll
store all of our tracing related data in a map that we intend to serialize as JSON later on when we send the data to our tracing backend. While we're at it, we'll also generate a **span ID** that can be used to uniquely identify that particular span.

```go
func rootHandler(...) {
    traceData := make(map[string]interface{})
    traceData["trace.trace_id"] = uuid.String()
    traceData["trace.span_id"] = uuid.String()

    // ... main work of request ...
}
```

**Generating Timing Information**

OK, so we've got our trace ID that will tie the whole request chain together, and a unique ID for this span. We'll also need to know when this span started and how long it took - so we'll note the **timestamp** from when this request started, and note the difference between that starting timestamp and the timestamp when we're all finished with the request to get the **duration in milliseconds**.

```go
func rootHandler(...) {
    // ... other setup ...

    startTime := time.Now()
    traceData["timestamp"] = startTime.Unix()

    // ... main work of request ...

    traceData["duration_ms"] = time.Now().Sub(startTime)
}
```

**Setting Service Name And Span Name**

We're so close now to having a full complete span for the root! All we need to add is a **name** and **service name** to indicate the service and type of span we're
working with. Additionally, when we're all finished generating the span, we'll send it to our tracing backend using HTTP.

```go
func rootHandler(...) {
    // ... other setup ...

    traceData["name"] = "/"
    traceData["service_name"] = "root"

    // ... main work of request ...
    sendSpan(traceData)
}
```

Phew! That's a bunch of work just to send one little span. But we haven't quite finished yet - we need to somehow indicate to the other services we are calling as a part of this request which trace the calls are a part of (the trace ID generated above).

**Propagating Trace Information**

The most common way to share this information with other services is to set one or more HTTP headers on the outbound request(s) containing this information. For instance, we could expand our helper functions `callAuthService` and `callNameService` to also accept the `traceData` map, so that on their outbound requests, they could set some special headers to be received by those services in their own instrumentation.

We could call these headers anything we want, as long as the programs on the receiving end know what their names are. For instance, maybe our tracing backend is named something wacky like BigBrotherBird, so we might call them things like `X-B3-TraceId`. In this case, we'll send the following to ensure the child spans are able to build and send *their* spans correctly:
1. X-B3-TraceId - Our ID for the whole trace from above
2. X-B3-ParentSpanId - The current span's ID, which will become a trace.parent_id in the child's generated span

```go
func callAuthService(originalRequest *http.Request, traceData map[string]interface{}) {
    req.Header.Set("X-B3-TraceId", traceData["trace.trace_id"])
    req.Header.Set("X-B3-ParentSpanId", traceData["trace.span_id"])

    // ... make the request ...
}
```

Given that information, the two services we call out to can pull these headers off and add them to trace.trace_id and trace.parent_id in their own generation of tracingData. Then, they can also send their generated spans to the tracing backend, which stitches everything together after the fact and enables the lovely waterfall diagrams we see above.
Adding Custom Fields

We might even add some custom fields to the trace data to self-describe further details about the operation encapsulated within the span. That might make it easier to find traces of interest later on, and to have our traces augmented with lots of juicy details. For instance, it’s always useful to know what host the request was served from, and if it was related to a particular user.

```go
gotrace, _ := os.Hostname()
traceData["tags"] = make(map[string]interface{ })
traceData["tags"]["hostname"] = gotrace
traceData["tags"]["user_name"] = name
```

All Together Now

Putting it all together, doing this from scratch would look something like this:

```go
func rootHandler(r *http.Request, w http.ResponseWriter) {
    traceData["tags"] = make(map[string]interface{ })

    gotrace, _ := os.Hostname()
    traceData["tags"]["hostname"] = gotrace

    startTime := time.Now()
    traceData["timestamp"] = startTime.Unix()
    traceData := make(map[string]interface{ })
    traceData["trace.trace_id"] = uuid.String()
    traceData["trace.span_id"] = uuid.String()
    traceData["name"] = "/"
    traceData["service_name"] = "root"

    authorized := callAuthService(r, traceData)
    name := callNameService(r, traceData)
    traceData["tags"]["user_name"] = name

    if authorized {
```
Kind of a lot, huh? It's great that we now have one method instrumented - but we need to spread this instrumentation everywhere. If we're application developers who just want to get stuff done and not worry about littering the leaky abstraction of sending tracing data all over our code, doing all of this from scratch any time we want to get tracing data out of a service is going to be a huge pain. Not to mention that if we want to generate tracing data for a service we use which Kyle's team down the hall develops and operates, we have to convince Kyle to do things our way too, and Kyle is a notorious stick in the mud when it comes to getting with the program. Get it together, Kyle.

But maybe if there was a better, faster way to drop in a shared library and get tracing data we could not only make our own lives easier, we could also convince other teams to instrument and march together in harmony towards our glorious observable future.
Tracing with Beelines

The Dapper paper cites shared client libraries as a key innovation, and Honeycomb Beelines take this kind of tracing instrumentation to the next level. Using Beelines, most of the boilerplate and boring setup work we outlined in our from-scratch example above is handled for you - freeing you to get all the benefits of tracing while being able to get right back to shipping new features and crushing bugs. The Beeline libraries are available for a variety of languages, and often will hook directly into your favorite frameworks such as Rails, Django, and Spring Boot to generate tracing data for your apps with only a few lines of added code.

Let's consider what the above example would look like with the Honeycomb Go Beeline instead.

Once we initialize the Beeline with our Honeycomb write key, we can simply wrap our Go HTTP muxer to create spans whenever an API call is received. This same idea can be used to generate spans when we do things like database queries using the sqlx package as well.

```
http.ListenAndServe(":8080", hynethttp.WrapHandler(mixer))
```

That's really it?

Yes, that's it – with a few lines of code you are sending tracing spans for your HTTP requests to Honeycomb. All of the boilerplate we outlined above is encoded into the Beeline library that Honeycomb provides you.

With Beelines, the only thing that does not come out of the box is the custom “tags” we added in the instrumentation above. To go beyond simple tags, Beelines allow you to augment your tracing spans with any relevant field or variable in your code. The data about which span is currently “active” is passed around in Beelines using things like Go's context package or Python's thread local variables, and you can augment the generated events for rich querying later
on in the Honeycomb web app. **This is extremely powerful because it allows us to easily analyze tracing data per customer, or by any dimension we can imagine.**

Custom fields are your tracing superpower. For instance, in Go we would add custom details like this:

```go
func rootHandler(r *http.Request, w http.ResponseWriter) {
    // ctx contains the Beeline-generated span
    ctx := r.Context()

    // we will be able to execute blazing fast queries
    // over these later
    beeline.AddField(ctx, "hostname", hostname)
    beeline.AddField(ctx, "user_name", name)
}
```

**What about the other services?**

This is **distributed** tracing, after all - so we need to also instrument our client that we use for outbound HTTP calls to the other services. Using a Beeline-based client, we can ensure that the proper headers end up getting passed around and decoded in the other apps. For instance, in Go we could build out a Beeline client and HTTP request that does tracing like this:

```go
client := &http.Client{
    Transport: 
    hynethttp.WrapRoundTripper(http.DefaultTransport),
}
```

Our program therefore does not need to worry about fussy tracing details like which headers need to be set to what value. The Beeline library ensures that this is taken care of.
What about custom spans?

Sometimes our code might do a chunk of work that is not distributed, but it might be something we want to split into its own span anyway. For instance, maybe we find that our program is bottlenecked by JSON unmarshaling or some other CPU-intensive operation and we need to identify when this is causing a problem.

We can wrap these “hot blocks” in their own spans to get an even more detailed waterfall. To do this, we use the context provided (or equivalent in other languages) to call startSpan, then send that span when it’s all done.

```go
ctx, span := beeline.StartSpan("slowCodeBlock", ctx)
if err := slowCodeBlock(ctx); err != nil {
    beeline.AddField(ctx, "error.detail", err)
}
span.Send()
```

This can be used to create traces in non-distributed or non-service-oriented programs as well. For instance, we could create a span for every chunk of work (S3 object, etc.) in a batch job, or for each distinct phase of a Lambda-based pipeline.

Querying Traces

In Honeycomb, one span is simply one event - all of the power of the Honeycomb query engine, including outlier analysis using BubbleUp, is at your fingertips to analyze patterns and trends in the data generated by traces.

For instance, you can go try out the publicly-available Honeycomb Tracing Tour to get a feel for querying over trace data. This dataset represents a finished version of what would be generated by code running in production. We could run a COUNT query where status_code associated with the root span was HTTP 500
which we BREAK DOWN by user_id, which would allow us to rapidly spot that one user was getting a vast amount of HTTP 500 errors.

The query we construct in Honeycomb is based on the properties of these spans/events we want to ask about:

The graph then shows us our answer:

If we want to get a feel for what the lifecycle of what one of these requests looks like as it hops across services, we can navigate to the **Traces** tab which will show us traces associated with the events from our query (the slowest are displayed first).
Clicking a trace ID in the displayed table will pop us into the trace view, where we can analyze the request and figure out in more detail why these HTTP 500s were occurring.

<table>
<thead>
<tr>
<th>user_id</th>
<th>root serviceName</th>
<th>root name</th>
<th>root durationMs</th>
<th>number of spans</th>
<th>span summary</th>
<th>traceId</th>
</tr>
</thead>
<tbody>
<tr>
<td>153228</td>
<td>api</td>
<td>/api/v2/tickets/export</td>
<td>1,449</td>
<td>37</td>
<td></td>
<td>b5fb7ce0c4e0f118</td>
</tr>
<tr>
<td>20109</td>
<td>api</td>
<td>/api/v2/tickets/export</td>
<td>1,380</td>
<td>35</td>
<td></td>
<td>b043ab2e6512013</td>
</tr>
</tbody>
</table>
What about Standards?

“The good thing about standards is that there are so many to choose from.”
— Andrew S. Tanenbaum

There are a few open standard specifications vying for supremacy in tracing-land. The two that come to mind for most folks are:

1. **OpenTracing**, which originally evolved with influence from Dapper and Zipkin to describe a model for tracing independent of implementation, and
2. **OpenCensus**, a project emerging out of Google more recently which seeks to unify metrics and tracing.

Tracing is such a new technology and the standards around it are also so new, that we at Honeycomb do not have any recommendations around standards as there is no clear winner or "best" standard. Here is some information about what we see as the pros and cons of these attempts at standardization.

The **pros** of standards are combatting vendor lock-in, and allowing collaboration to flourish between various entities in the space. Having been burned by software that is difficult to switch out of, many engineers and organizations today are conscientious about choosing software that would be labor intensive to switch out later on. A standard should also allow various participants and in an ecosystem to join forces and collaborate to benefit all the parties involved. Less work is spent re-inventing the wheel and more work is spent on differentiating factors and user happiness.

The **cons** of standards are that they risk diluting the end technology used to get the actual business results, and they tend to slow down innovation. Since everyone must conform to the same format and standard, a “lowest common denominator” factor can potentially take hold. Changing the standard or adding
new features requires sign off from a group composed of entities with highly variable interests and convictions. There is, of course, also the potential peril of choosing a standard which is not successful in the end.

Ultimately, at Honeycomb we find that our users get the best results using our native code integrations directly. That said, if using OpenTracing or OpenCensus is right for your business, we support getting this data into Honeycomb as well.

This Must Be the Trace

Using Honeycomb tracing you can get closer to that holy grail of observability: guessing less and deploying more. We hope that using Honeycomb’s powerful query engine and tracing capabilities, you too can find yourself thinking - “This must be the trace!” and solving your problems faster than ever. Don’t hesitate to sign up for a trial today or to give us a ring at sales@honeycomb.io.
About Honeycomb

Honeycomb provides next-gen APM for modern dev teams to better understand and debug production systems. With Honeycomb teams achieve system observability and find unknown problems in a fraction of the time it takes other approaches and tools. More time is spent innovating and life on-call doesn’t suck. Developers love it, operators rely on it and the business can’t live without it.

Follow Honeycomb on Twitter LinkedIn
Visit us at Honeycomb.io