# TOKIO-TRACE

scoped, **structured**, async-aware diagnostics

# WHOIS?

**eliza weisman**

→ systems engineer at Buoyant
→ tokio, tower, linkerd 2, etc
→ @mycoliza on twitter
→ cat liker

so, who am I? my name is Eliza Weisman; I'm a systems software engineer at
Buoyant here in San Francisco.

I've been writing Rust since 2015, and I've been doing it professionally for
almost two years now.

I contribute to the tokio, tower, and linkerd 2 open source projects.

Some of you have probably seen my twitter where I post bad programming jokes
and pictures of my cats...

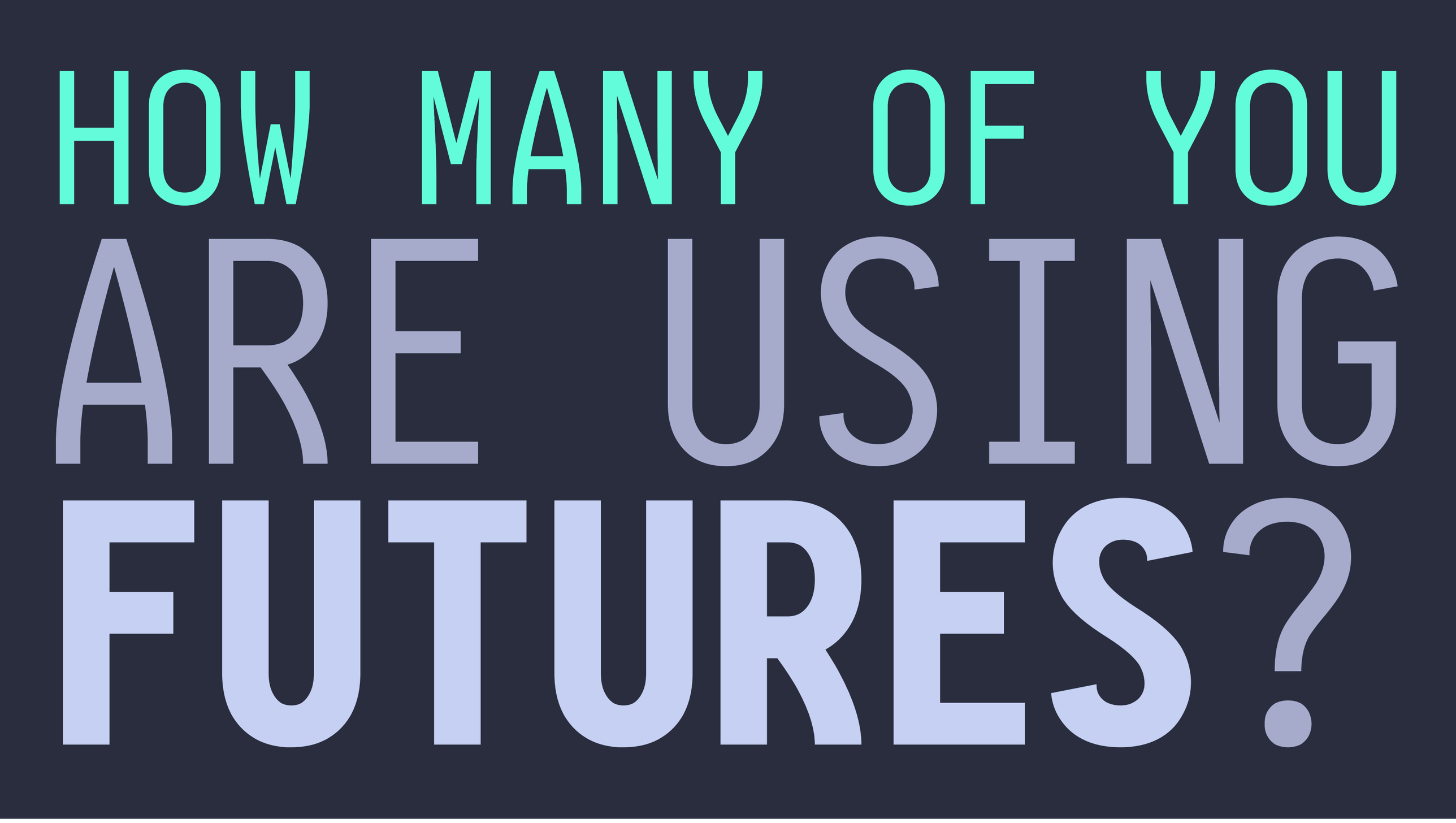
# WHY I MADE ANOTHER LOGGING LIBRARY

A lot of you are probably wondering why we made another logging library. We
already have logging.

Well first of all, I don't like to call it "logging". I prefer to call it "GNU
slash logging".

Yes, that was a joke. But what I really prefer to call it is "in-process
tracing". We'll talk about what that actually means a little later.

To answer this question, I'm going to start by asking some questions of my
own.




alright, show of hands...how many of you are using futures?

# AND DOES YOUR LOGGING MAKE ANY SENSE?

great. do your logs make any sense? like at all?

# ASYNCHRONY IS HARD

Yeah. Async is hard, right? *pause for laughter*

If you're writing a high-performance network application, you probably need to use asynchronous programming. Async presents some unique challenges to
diagnostics.

Execution is multiplexed between tasks. When a task is blocked on IO or
another task, it yields, and we start executing another task. The task will wake
back up when IO is ready.

Because of this, log messages can end up interleaved, or we can't tell what
context a message happened in.

# HOW DO WE GET USABLE DIAGNOSTICS FROM ASYNC SYSTEMS?

What do we need in order to get usable diagnostics from async software?

The way I see it, there are three main things we want our diagnostics to capture: context, causality, and structure.

# CONTEXT

Context. When we record that an event occurred, we don't just want to know
where in the *source code* it happened, but in what *runtime context* as well.

For example, if we have a server that's processing requests, the context might
include: what client did this request come from? what where the request's
method, path, and headers?

In synchronous code, we can infer context from the order that log records
appear in. But in async code, we switch between contexts, so our diagnostics
need to track them.

# CAUSALITY

Second, we want to capture *causality*. What other events caused this event to occur?

If some task is running in the background, say, a DNS resolution, or a database connection, what caused that task to start? Which request required that DNS resolution?

In async systems we can't rely on ordering to determine causality. So again, we need to record it.

# STRUCTURE

Traditional logging is based on human-readable text messages. We'd prefer our diagnostics to record machine-readable structured data.

This lets us interact with our diagnostic data programmatically. You can record typed values and interact with them as numbers, booleans, and so on.

TIME FOR A

# DEMO

# HOW TOKIO-TRACE ACTUALLY WORKS

So how does `tokio-trace` actually *work*?

# SO WHAT DID I MEAN BY IN-PROCESS TRACING?

Is anyone here familiar with distributed tracing systems? Like OpenTracing,
OpenCensus or Zipkin?

Okay, great. These are diagnostic tools for distributed systems. They're designed for tracking contexts as they move from node to node, so that you can
correlate events on one node with events on another.

A key insight behind `tokio-trace` is that asynchronous programs are kind of
like distributed systems writ small. You have concurrently running tasks that
communicate through fallible message passing. The only difference is that
everything lives in one address space.

# CORE PRIMITIVES SPANS AND

Our core primitives instrumentation primitives are *spans* and *events*.

# SPANS

## PERIODS OF TIME

```
span!("my_great_span").enter(|| {
    // do some stuff *inside* the span...
})
```

A span represents a period of time where the program is executing in a context.

Spans have beginnings and ends, and we can *enter* and *exit* them as we
switch between contexts.

# EVENTS

## MOMENTS IN TIME

```
event!(Level::Info, "something happened!");
```

Events, on the other hand, represent singular instants in time where something happened.

They're analagous to log records in conventional logging. But unlike log records, they exist in a span context.

# FIELDS

## ADD STRUCTURED DATA

```
event!(Level::Info, foo = 3, bar = false);
```

Fields are how we attach typed, structured data to spans and events. A field
is a key-value pair.

Tokio-trace subscribers can consume field values as a subset of Rust
primitive
types.


To put it all together, here's a little example. We're shaving some yaks.

```
span!("shaving_yaks", yak_count = yaks.len()).enter(|| {

//  for yak in yaks {
//      span!("shave", current_yak = yak).enter(|| {
//          match shave_yak(yak) {
//              Ok(_) => debug!(message = "yak shaved successfully"),
//              Err(e) => warn!(message = "yak shaving failed!", error = field::debug(e)),
//          }
//      })
//  }
})
```

So we create a span called "shaving yaks". We're going to do all the work in there. We annotate that span with the number of yaks we're shaving.

```
span!("shaving_yaks", yak_count = yaks.len()).enter(|| {

    for yak in yaks {
        span!("shave", current_yak = yak).enter(|| {
//          match shave_yak(yak) {
//              Ok(_) => debug!(message = "yak shaved successfully"),
//              Err(e) => warn!(message = "yak shaving failed!", error = field::debug(e)),
//          }
        })
    }
})
```

Then we loop over all the yaks, and we create a new span, "shave", for each one. The new span is *inside* the "shaving yaks" span. We record which yak we're currently shaving as a field on that span.

```
span!("shaving_yaks", yak_count = yaks.len()).enter(|| {

    for yak in yaks {
        span!("shave", current_yak = yak).enter(|| {
            match shave_yak(yak) {
                Ok(_) => debug!(message = "yak shaved successfully"),
                Err(e) => warn!(message = "yak shaving failed!", error = field::debug(e)),
            }
        })
    }
})
```

We call this "shave yak" function on the current yak. Anything that happens
in that function is *also* inside the "shave" span, which is nested inside the
"shaving yaks" span.

Then, we match on the return value of "shave_yak", and record if it's Ok or an Error. Since those events are inside the "shave" span, they're annotated with
the yak we're shaving automatically.

# SUBSCRIBERS

## COLLECT TRACE

Finally, we have a component called a `Subscriber`. Subscribers are the
component that actually collects and records the trace data generated by our
instrumentation.

You can think of a subscriber as being kind of like a logger. And like loggers, `Subscribers` are pluggable. This is tokio-trace's main extension point.

Libraries can provide subscribers that implement different behavior. One might
print traces to standard out, another might record metrics, and third might send
events to some distributed tracing system.

# HOW TO USE IT

We've tried to make `tokio-trace` as easy to adopt as possible. This includes compatibility with other libraries you might already be using.

Here are some examples of stuff you can do.

## plays nice with futures

```
my_future
    .and_then(|result| {
        debug!("doing something...");
        do_something(result)
    })
    .map_err(|e| {
        warn!(error = field::debug(e));
    })
    .instrument(span!("my_future"));
```

It plays nice with futures. Here we're composing a future with some
combinators.

We provide this new `instrument` combinator which lets you attach a span to a
future. Whenever we poll this future, we'll enter the span for the duration of
the poll.

This means that everything that happens in `my_future`, or in the `and_then`
and `map_err` here, will be inside of the "my_future" span.

**this compiles**

```
#[macro_use]
extern crate log;

info!("log-style logging! foo={}; bar={}", 42, true);
```

We also have drop in compatibility with the `log` crate. Here I'm importing log and using its `info` macro to log a message. If I want to switch to tokio-trace...

## ...and so does this

```
#[macro_use]
extern crate tokio_trace;

info!("log-style logging! foo={}; bar={}", 42, true);
```

All I have to do is change which crate I'm importing.

Tokio-trace has macros that are a superset of log's macros. They can do more stuff than log, but they support all the same syntax.

We also have adapters to let you convert between log records and trace events.

# ONLY **PAY** FOR WHAT YOU **USE**

Any runtime instrumentation has performance costs. Tokio-trace's goal is to ensure you don't pay any costs you don't *have* to.

What does that mean?

# DISABLED INSTRUMENTATION IS (NEARLY)

First of all, we've made sure that a subscriber filters
out spans or events you don't want to record, the overhead is basically a single load and a branch --- under one nanosecond.

We cache filter evaluations when possible --- if something is always
disabled, we never need to re-filter it.

# SUBSCRIBERS DON'T PAY COSTS BY DEFAULT

Furthermore, we've left all the real overhead up to subscriber implementations.

Since different use-cases have different requirements --- some have to allocate to track data, others need to make syscalls to get timestamps --- `tokio-trace` doesn't require that *all* subscribers pay those costs.

let's see some

# DEMOS

# BOOTSTRAPPING AN ECOSYSTEM

It's worth noting that we're trying to bootstrap a whole ecosystem here. We released the core library on crates.io today, but that's just the beginning.

There's a whole lot of neat stuff we can build on top of tokio-trace together. I'm sure I haven't even thought of all of it yet.

# GET INVOLVED

- → crates.io/crates/tokio-trace-core
- → github.com/tokio-rs/tokio
- → github.com/tokio-rs/tokio-trace-nursery

So here's how you can get involved. The first thing you can do is just try it
out. I love bug reports and feature requests, and I love PRs even more.

Second, if there's anything you want to see in the ecosystem, maybe you want a
subscriber for your favorite metrics lib, or a different way of formatting trace
logs, please share it! I can't wait to see what people build using `tokio-trace`.

The core crates live in the `tokio` repo, and we have a "nursery" repo for
less stable libraries. A lot of the utility and compatibility crates live there.

# thanks <3

- → Carl Lerche (@carllerche)
- → David Barsky (@davidbarsky)
- → Ashley Mannix (@KodrAus)
- → Lucio Franco (@LucioFranco)

These are some of the folks who have already helped out a lot.

Carl Lerche, of course, is the original author of `tokio`, and he's given me so much guidance throughout the whole process of writing tokio-trace.

I'd like to thank David Barsky for all the conversations we had during the design and development of tokio-trace, and for the work he did on the tokio-trace macros.

Ashley Mannix is working on adding structured logging to the `log` crate, and we had some great discussions about how to ensure `tokio-trace` is compatible with log. He also had some great advice for the design of the `Value` system.

Lucio Franco helped out with the nursery crates a lot, especially the format subscriber.

Also, thanks to my partner Tristan, who listened to me practice this talk several times even though they didn't really understand what it was about.

Finally, thanks to all of you for giving your time to listen to me speak about `tokio-trace` today!'

# QUESTIONS?

→ email: eliza@buoyant.io
→ twitter: @mycoliza
→ slides: elizas.website/slides/
→ ...or, see me after class!

Before I open the floor for questions, here's how you can contact me if you want to chat about `tokio-trace`, `tokio` or `linkerd`.

Also, you can find the slides (and a recording of this talk) at elizas.website/slides. Feel free to take a picture of this slide if you want to.