I know on the program it says this talk is about "tokio-trace". That's a lie, this talk is not actually about tokio-trace. We renamed the library, and it's called "tracing" now. "tracing" is a set of libraries for instrumenting Rust programs with scoped, structured, and async-aware diagnostics.
you're probably wondering, who am I, and why am I an important logging thought leader? my name is Eliza Weisman; I'm a systems software engineer at Buoyant in San Francisco.
I've been writing Rust since 2015, and I've been doing it professionally for over 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...
A lot of you are probably wondering why we made another logging library. We already have a pretty robust ecosystem for 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 (or async-await)?
AND DOES YOUR LOGGING MAKE ANY SENSE?
great. do your logs make any sense? like at all?
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 requires something from IO or another task, instead of blocking, it yields. Then, 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.
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. 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.
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.
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.
Enter tracing. Tracing is a framework for instrumenting Rust programs, with contextual, causal, structured diagnostics.

As I said before, this used to be called tokio-trace. Tracing is part of the Tokio project, but it doesn't require the Tokio runtime.
TRACING

→ formerly tokio-trace
→ part of the Tokio project
→ doesn't require the Tokio runtime
→ works with no-std*

With some caveats.
TIME FOR A DEMO
So how does tracing actually work?
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 tracing 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.
Our core primitives instrumentation primitives are spans and events
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. A program can enter and exit a span multiple times over its lifespan. This is especially important for asynchronous code. In particular, it gives us the ability to record both how long a task existed for, and how long we were actively executing it.
let span = span!(Level::TRACE, "my_great_span");

So here is how you use spans. You create a span with the `span!` macro, and you get back this span object. Here's a span called "my great span".
Then you can enter the span. You get back a RAII guard, so you can enter the span in a scope.

```rust
let span = span!(Level::TRACE, "my_great_span");
let _guard = span.enter();
```
let span = span!(Level::TRACE, "my_great_span");
let _guard = span.enter();
// do some stuff *inside* the span...

Anything that happens while that guard is held is considered inside the span.
We also have this instrument attribute. You can put it on a function & it will automatically create and enter a span when the function is called.
And this works with async functions too.
Events, on the other hand, represent singular instants in time where something happened. They're analogous to log records in conventional logging. But unlike log records, they exist in a span context.
Fields are how we attach typed, structured data to spans and events. A field is a key-value pair. Tracing subscribers can consume field values as a subset of Rust primitive types.
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 tracing'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.
BUT HOW DO I USE IT?
To put it all together, here's a little example.
We're shaving some yaks.

```python
for yak in yaks {
    shave_yak(yak);
}
```
...asynchronously. So this shave_yak function is asynchronous, it's returning a future.
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.
let span = span!(Level::INFO, "shaving_yaks", yak_count = yaks.len());
let _enter = span.enter();

for yak in yaks {
    tokio::spawn(shave_yak(yak));
}
Every iteration of the loop, we create a new span "shave", annotated with the yak we're shaving.

```rust
let span = span!(Level::INFO, "shaving_yaks", yak_count = yaks.len());
let _enter = span.enter();

for yak in yaks {
    let span = span!(Level::DEBUG, "shave", current_yak = yak);
    tokio::spawn(shave_yak(yak));
}
```
This instrument combinator attaches a span to a future so that it's entered whenever we poll the future.

```rust
let span = span!(Level::INFO, "shaving_yaks", yak_count = yaks.len());
let _enter = span.enter();

for yak in yaks {
    let span = span!(Level::DEBUG, "shave", current_yak = yak);
    tokio::spawn(shave_yak(yak).instrument(span));
}
```
We record an event after the yak is shaved. Since this is inside the async block that's implemented with the shave span, we don't need to record which yak we're shaving --- the span does that for us.
Let's say shaving the yak is fallible. Here we can record an event depending on the return value. And here we're recording that error as a structured field on this span, so the subscriber can record it as a typed value.

Again, we don't need to record what yak we were shaving --- that's added by the span.
WORKS WITH
LOG
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 tracing...

this compiles

use log::info;

info!("log-style logging! foo={}; bar={}", 42, true);
All I have to do is change which crate I'm importing.

tracing 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.
Any runtime instrumentation has performance costs. Tracing's goal is to ensure you don't pay any costs you don't have to. What does that mean?

ONLY PAY FOR WHAT YOU USE
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.
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 --- tracing doesn't require that all subscribers pay those costs.
It's worth noting that we're trying to bootstrap a whole ecosystem here. Tracing consists of two core libraries, tracing and tracing-core, that provide the instrumentation APIs and the plumbing that makes everything work. But this doesn't do a whole lot on its own. There are so many different ways to consume the data that tracing exposes, and we have some other libraries providing subscriber implementations. For example, the one I used in the demo is from the "tracing-fmt" crate, which logs tracing events to standard out. Jon Gjengset is working on a "tracing-timing" crate that uses tracing spans for time profiling. There's a whole lot of neat stuff we can build on top of tracing together. I'm sure I haven't even thought of all of it yet.
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 tracing.

The core crates live in the tokio-rs/tracing repo
Before I open the floor for questions, here's how you can contact me if you want to chat about tracing, 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.