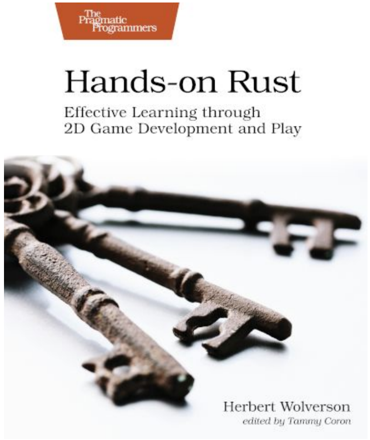
Async Rust

A 1-Hour Deep Dive

herbert.wolverson@ardanlabs.com

About Herbert Wolverson

- Ardan Labs Rust Trainer & Consultant
- Author of *Hands-on Rust* and *Rust Brain Teasers*
- Author of the *Rust Roguelike Tutorial*
- Lead developer, *LibreQoS*, *bracket-lib*.
- Contributor to many open source projects.



All code used in this presentation is available here:

https://github.com/thebracket/Ardan-1HourAsync

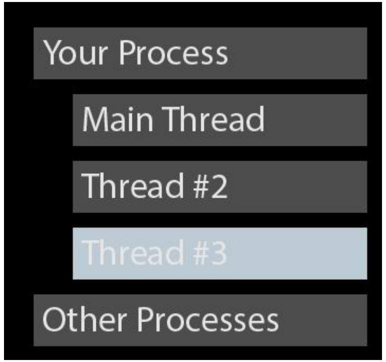
What We're Going to Cover

- Threads vs. Async - Together or Separate
- Async Runtimes (Executors)
- What does "block on" really mean - and the Tokio macros
- Running Async Code
- Blocking, and Sending Blocking Tasks to Threads
- Inter-Task Communication: Channels
- Streams: Async Iterators
- Tokio + Axum: High-Performance Web Services with Dependency Injection and Ergonomic Development
- Tracing & Performance Metrics
- Q&A

Threads and Asynchronous Code

System Threads

- Created with std::thread
- Map directly to Operating System Threads
- Heavy-weight:
  - Limited number (60,000 on my Linux system).
  - Each thread gets a full stack.
  - Each thread is scheduled by the Operating System as a full entity.
  - Many thousands of threads don't scale.
- Acts like “normal” code - it runs from end to end, the OS interrupts and switches threads when it decides to do so.
- Great for CPU intensive work, no need to “yield” control.



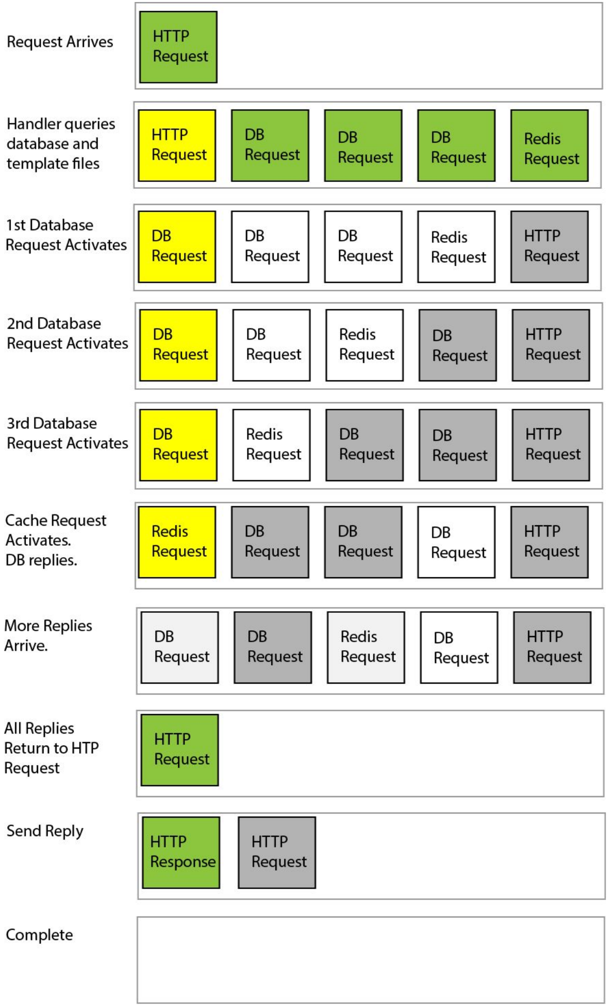
So Why not Use Threads for Everything?

- With a simple model of one thread per network connection, a busy server could have *thousands* of threads.
- Each thread will spend most of its time idle:
  - Waiting for the Network
  - Waiting for Storage
  - Waiting for Database Queries
- Scheduler thread polling and RAM usage quickly adds up to a sluggish server.

- You can *select* from a group of sockets to see which are ready for you.
- This was an early form of async programming.
- Your async runtime is probably doing something similar for you!

Asynchronous

- Async can be single-threaded or run across multiple threads.
- Whenever you create an async *task* it is:
  - Lightweight (very little memory usage)
  - Cooperatively Scheduled by Your Runtime
  - Easy to Cancel
- Async queues run *one* task at a time.
- Tasks run until they *yield*, or *await* another task.
- Your runtime thread will still be scheduled by the Operating System.



Async and Threading - Why Not Both?

The larger/more powerful runtimes for Rust combine threading and async.

- Each CPU core gets a thread (or you can customize the number of threads).
- Each thread maintains a task list.
- Threads can “steal work” from one another - if a core finds itself idle, it can reach into another queue and “steal” a job.
- The result? You are benefiting from network/storage/database latency, maximizing CPU utilization while not wasting time polling idle threads.

You can divide cores as needed:

- *x* cores running one more more async runtimes.
- *y* cores dedicated to commute.
- Communicate between the groups with high-performance channels.
- Result:
  - No I/O wait thread starvation on the tasks receiving work and sending results.
  - High-performance cores perform CPU intensive tasks.

Asynchronous Runtimes (Executors)

Rust is Agnostic - and Flexible

Many languages chose a predetermined async strategy.

- This is great, so long as you are writing the type of application for which the async environment was designed.
- This can leads to jumping through hoops if your goals don’t align with the language design.
- Most languages offer some tuning to mitigate this.

Rust:

- Implemented the plumbing for async in the language core (designed so that the core won’t allocate memory) - and leaves the implementation to runtimes / executors.
- This gives flexible choice of runtimes - you can even build your own.
- Runtimes are available for small embedded projects, all the way up to high-performance targets.
- 5 of TechEmpower’s 10 fastest webservers are built on Rust.

Choosing a Runtime

- **Tokio**: a “do everything” high-performance runtime.
- **Async-std**: Also high in features, not as popular but more focused on standardization.
- **Futures**: A partial runtime, many useful utilities.
- **Smol**: A small runtime, kept simple.
- **Pasts**: A simple runtime that runs without the standard library and minimal memory allocations - works with embedded and WASM.

For this talk, we’re going to use **Tokio**.

Tokio provides a broad ecosystem, and high performance. It pairs nicely with an existing stack of Hyper (for HTTP), Tower (for service management) and web servers (Axum and Actix being two popular ones).

Launching into Asynchronous Code

All Programs Start Synchronously

- You can't run `async` functions outside of an async runtime.
- Most runtimes are started with a call to `block_on(<<your async function>>)`
- This grants flexibility:
    - For an all async app, you can use one runtime.
    - You can launch threads, and spawn runtimes inside them.
    - You can even spawn a runtime just to access a single async service.

Cargo.toml:

```toml
[dependencies]
futures = "0.3.28"
```

main.rs:

```rust
use futures::executor::block_on;

async fn do_work() {
    println!("Hello, async world!");
}

▶ Run | Debug
fn main() {
    println!("Hello, sync world");
    block_on(do_work());
}
```

Tokio Startup with Block On

Tokio also starts with `block_on` - and lets you tailor it to what you need (it also picks good defaults).

```
fn main() {
    let rt = runtime::Builder::new_multi_thread()
        // YOU DON'T HAVE TO SPECIFY ANY OF THESE
        .worker_threads(4)  // 4 threads in the pool
        .thread_name_fn(thread_namer) // Name the threads.
                                      // This helper names them "my-pool-#" for debugging assistance.
        .thread_stack_size(3 * 1024 * 1024) // You can set the stack size
        .event_interval(61) // You can increase the I/O polling frequency
        .global_queue_interval(61) // You can change how often the global work thread is checked
        .max_blocking_threads(512) // You can limit the number of "blocking" tasks
        .max_io_events_per_tick(1024) // You can limit the number of I/O events per tick
        // YOU CAN REPLACE THIS WITH INDIVIDUAL ENABLES PER FEATURE
        .enable_all()
        // Build the runtime
        .build()
        .unwrap();

    rt.block_on(hello());
}
```

Or: Quick-Start Tokio with a Macro

Tokio’s #[tokio::main] macro lets you just write an async main function.

You can specify parameters (e.g. flavor=“current_thread”) to apply the same customization options.

You can even still spawn threads - and runtimes inside them.

```rust
#[tokio::main]
async fn main() {
    hello().await;
}
```

Writing Asynchronous Code

Hello Async/Await

- Decorating a function with `async` changes the return type to a `Future`: a handle for the task, and a link to the result when it becomes available.
- Futures don't run until you tell them to.
- The most common way to launch an async task is to `await` it.
- When you `await` a task:
    - Your task enters a paused state.
    - The task you are waiting for is added to the task list.
    - When the task finishes (it may, in turn, await) control returns to your function with the result available.

```rust
async fn hello() {
    println!("Hello from async");
}

#[tokio::main]
async fn main() {
    hello().await;
}
```

Joining: Simultaneous tasks, wait for all

You can use Tokio's `join!` macro to spawn several tasks at once, and wait for all of them.

- Results are returned in a tuple, one per task.
- Your task waits for *all* of the joined tasks to complete.

You can use `join_all` from futures, or Tokio's `JoinSet` to join an arbitrary vector of futures.

```rust
async fn hello() {
    println!("Hello from async");

    // Use the tokio::join! macro
    let result: (i32, i32) = tokio::join!(double(2), double(3));
    println!("{result:?}");

    // You can still use futures join_all
    let futures: Vec<impl Future<Output = …>> = vec![double(2), double(3)];
    let result: Vec<i32> = futures::future::join_all(iter: futures).await;
    println!("{result:?}");

    // Using Tokio JoinSet
    let mut set: JoinSet<i32> = JoinSet::new();
    for i: i32 in 0..10 {
        set.spawn(task: double(i));
    }
    while let Some(res: Result<i32, JoinError>) = set.join_next().await {
        println!("{res:?}");
    }
}
```

Spawn: detached tasks

Calling tokio::spawn launches an async task detached. Your task remains active, and the spawned task joins the task queue.

If you are in a multi-threaded context, the task is likely to start on another thread.

```rust
async fn ticker() {
    for i: i32 in 0..10 {
        println!("tick {i}");
    }
}

async fn tocker() {
    for i: i32 in 0..10 {
        println!("tock {i}");
    }
}

#[tokio::main]
▶ Run | Debug
async fn main() {
    let _ = tokio::join!(
        tokio::spawn(ticker()),
        tokio::spawn(tocker()),
    );
}
```

Yielding Control

If your async function is doing too much, you can explicitly yield control to another task.

When it's your function's turn to run again, it will resume where it left off.

Yielding puts your task at the back of the work queue. If there are no other tasks, you'll keep running.

Yielding can be a good band-aid, but if you are doing enough to regularly slow down other tasks - you need blocking.

```rust
async fn ticker() {
    for i: i32 in 0..10 {
        println!("tick {i}");
        tokio::task::yield_now().await;
    }
}

async fn tocker() {
    for i: i32 in 0..10 {
        println!("tock {i}");
        tokio::task::yield_now().await;
    }
}

#[tokio::main]
▶ Run | Debug
async fn main() {
    let _ = tokio::join!(
        tokio::spawn(ticker()),
        tokio::spawn(tocker()),
    );
}
```

Select: Process Whichever Returns First

Tokio's `select!` Macro launches all of the listed futures - and waits for *any one* of them to return. The other futures are canceled.

This can be useful for:

- Adding timeouts to operations.
- Listening to multiple data sources (channels & streams), and responding to whichever has an event ready.
- Retrieving data from several sources, and using whichever one answers first (the DNS or NTP model).

```rust
use rand::Rng;

async fn sleep_random() {
    let mut rng = rand::thread_rng();
    let secs = rng.gen_range(0..5);
    tokio::time::sleep(tokio::time::Duration::from_secs(secs)).await;
}

#[tokio::main]
▶ Run | Debug
async fn main() {
    for _ in 0..10 {
        tokio::select! {
            _ = sleep_random() => println!("Task 1 Returned"),
            _ = sleep_random() => println!("Task 2 Returned"),
            _ = sleep_random() => println!("Task 3 Returned"),
        }
    }
}
```

The Problem with Blocking

Async functions are cooperatively multitasked. So if you call a synchronous function that takes a while to run, you risk “blocking” the whole task system.

Calling `std::thread::sleep` can be really bad, you might put the whole runtime to sleep!

Fortunately, Tokio includes `tokio::time::sleep` for async sleeping.

```
use std::time::Duration;

async fn hello_delay(task: u64, time: u64) {
    println!("Task {task} has started");
    std::thread::sleep(dur: Duration::from_millis(time));
    //tokio::time::sleep(Duration::from_millis(time)).await;
    println!("Task {task} is done.");
}

#[tokio::main]
▶ Run | Debug
async fn main() {
    let mut futures: Vec<impl Future<Output = …>> = Vec::new();
    for i: u64 in 0..5 {
        futures.push(hello_delay(task: i, time: 500 * i));
    }
    futures::future::join_all(futures).await;
}
```

Spawning Blocking Tasks

Tokio includes a command named `spawn_blocking` for launching synchronous blocking tasks.

Blocking tasks run in a system thread, and don't pause the async runtime. You `await` the spawned task and your task is idle until it returns.

You can configure the size of the blocking thread pool on runtime startup.

```rust
use tokio::task::spawn_blocking;

fn is_prime(n: u32) -> bool {
    (2 ..= n/2).all(|i: u32| n % i != 0 )
}

async fn slow_counter() -> usize {
    spawn_blocking(move || {
        (2 .. 100_000).filter(|&x: u32| is_prime(x)).count()
    }).await.unwrap()
}

async fn ticker() {
    loop {
        println!("Still alive!");
        tokio::time::sleep(tokio::time::Duration::from_secs(1)).await;
    }
}

#[tokio::main]
 ▶ Run | Debug
async fn main() {
    tokio::spawn(ticker());
    let counted_primes: usize = slow_counter().await;
    println!("{counted_primes}");
}
```

Communicating with Channels

Inter-task Communication

Let’s create an asynchronous MPSC (Multi-Producer, Single Consumer) channel.

We’ll send the producer to one task that periodically sends out tick messages.

We’ll also spawn a consumer that sits and waits for messages.

Identical functionality is provided in the standard library for inter-thread communication.

```rust
enum Message {
    Tick,
}

async fn sender(tx: tokio::sync::mpsc::Sender<Message>) {
    loop {
        tx.send(Message::Tick).await.unwrap();
        tokio::time::sleep(tokio::time::Duration::from_secs(1)).await;
    }
}

async fn receiver(mut rx: tokio::sync::mpsc::Receiver<Message>) {
    while let Some(message: Message) = rx.recv().await {
        match message {
            Message::Tick => println!("Tick"),
        }
    }
}

#[tokio::main]
▶ Run | Debug
async fn main() {
    let (tx: Sender<Message>, rx: Receiver<Message>) = tokio::sync::mpsc::channel::<Message>(100);
    tokio::spawn(sender(tx));
    receiver(rx).await;
}
```

Listening to Multiple Channels

Let's create a second channel, and a second message producer on a different cadence.

Now we'll use `select!` to listen to both channels, and process whichever one has a message for us.

Channels queue data - up to the specified maximum length - you won't lose data.

```rust
enum Message {
    Tick(u32),
}

async fn sender(tx: tokio::sync::mpsc::Sender<Message>, n: u32) {
    loop {
        tx.send(Message::Tick(n)).await.unwrap();
        tokio::time::sleep(tokio::time::Duration::from_secs(1)).await;
    }
}

async fn receiver(
    mut rx: tokio::sync::mpsc::Receiver<Message>,
    mut rx2: tokio::sync::mpsc::Receiver<Message>
)
{
    loop {
        tokio::select! {
            Some(Message::Tick(n)) = rx.recv() => println!("Received message {n}"),
            Some(Message::Tick(n)) = rx2.recv() => println!("Received message {n}"),
        }
    }
}

#[tokio::main]
▶ Run | Debug
async fn main() {
    let (tx, rx) = tokio::sync::mpsc::channel::<Message>(100);
    let (tx2, rx2) = tokio::sync::mpsc::channel::<Message>(100);
    tokio::spawn(sender(tx, n: 1));
    tokio::spawn(sender(tx: tx2, n: 2));
    receiver(rx, rx2).await;
}
```

Channels Between Threads and Tasks

You can mix-and-match channel communication between threads and tasks - or even async runtimes - by using standard library channels and obtaining a “handle” to the runtime.

Let’s create a standard library channel, and pass the sender to a system thread along with a handle for connecting to a runtime from the outside.

The thread sends data into the channel.

This can work in either direction. It’s useful for:

- System threads managing intensive/time critical tasks and notifying the runtime of updates.
- CPU intensive tasks can run on a dedicated thread pool, and receive tasks from (and send results to) the async system.

This is a great way to have the best of both worlds.

```rust
use std::time::Duration;

#[tokio::main]
▶ Run | Debug
async fn main() {
    let (tx, mut rx) = tokio::sync::mpsc::channel::<u32>(100);
    let handle = tokio::runtime::Handle::current();

    std::thread::spawn(move || {
        let mut n: i32 = 0;
        loop {
            std::thread::sleep(dur: Duration::from_secs(1));
            let my_tx = tx.clone();
            handle.spawn(async move {
                my_tx.send(n).await.unwrap();
            });
            n += 1;
        }
    });

    while let Some(n) = rx.recv().await {
        println!("Received {n} from the system thread");
    }
}
```

Asynchronous Iteration with Streams

Streams: Async Iterators

Iterators work by:

- Storing a type they will yield.
- Returning the next item with `next()` or None if the iterator is done.

Streams are the same, but `next()` is an async call.

This gives some benefits:

- Iterating a huge data-set yields on each element, ensuring smooth task flow.
- Streams become self-pacing. Other parts of the data pipeline are awaiting too - so the stream advances at the speed of the slowest item, completely idle while paused.

```rust
use tokio_stream::StreamExt;

0 implementations
struct MyStream {
    counter: u32,
}

impl tokio_stream::Stream for MyStream {
    type Item = u32;

    fn poll_next(
        mut self: std::pin::Pin<&mut Self>,
        cx: &mut std::task::Context<'_>
    ) -> std::task::Poll<Option<Self::Item>> {
        self.counter += 1;
        // Pretend there's some work here
        for _ in 0..1000 {
        }
        if self.counter < 100 {
            std::task::Poll::Ready(Some(self.counter))
        } else {
            std::task::Poll::Ready(None)
        }
    }
}

async fn ticker() {
    loop {
        print!("T");
        tokio::task::yield_now().await;
    }
}

async fn streamer() {
    let mut stream: MyStream = MyStream { counter: 0 };
    while stream.next().await.is_some() {
        print!(".");
    }
}

#[tokio::main]
▶ Run | Debug
async fn main() {
    tokio::spawn(ticker());
    streamer().await;
}
```

Streams as Generators

You can use a stream as a generator - it doesn't have to be streaming pre-existing content.

Each call to poll_next() generates a new item.

You can use this for anything from random number generation - when you need random numbers, subscribe to the stream - to unique identifiers or iterative math.

```rust
use tokio_stream::StreamExt;

0 implementations
struct Doubler {
    counter: u32,
}

impl tokio_stream::Stream for Doubler {
    type Item = u32;

    fn poll_next(
        mut self: std::pin::Pin<&mut Self>,
        _cx: &mut std::task::Context<'_>
    ) -> std::task::Poll<Option<Self::Item>> {
        self.counter *= 2;
        if self.counter < u32::MAX/2 {
            std::task::Poll::Ready(Some(self.counter))
        } else {
            std::task::Poll::Ready(None)
        }
    }
}

#[tokio::main]
▶ Run | Debug
async fn main() {
    let mut stream: Doubler = Doubler { counter: 1 };
    while let Some(n) = stream.next().await {
        println!("{n}");
    }
}
```

Convert a Reader into a Stream

Tokio's stream helpers can convert readers into streams (and vice versa).

So you can open a file, stream the content, and File IO becomes self-pacing - and no need to read the whole file at once.

```rust
async fn async_line_count(n: u32, filename: String) -> anyhow::Result<()> {
    use tokio::io::AsyncBufReadExt;
    use tokio::io::BufReader;
    use tokio::fs::File;

    println!("Reading {filename}...");

    let file: ! = File::open(filename).await?;
    let reader = BufReader::new(file);
    let mut lines = reader.lines(); // Create a stream of lines
    while let Some(line) = lines.next_line().await? {
        println!("{n}: {line}");
    }
    Ok(())
}

#[tokio::main(flavor="current_thread")]
▶ Run | Debug
async fn main() {
    let _ = tokio::join!(
        tokio::spawn(async_line_count(1, "warandpeace.txt".to_string())),
        tokio::spawn(async_line_count(2, "warandpeace.txt".to_string())),
        tokio::spawn(async_line_count(3, "warandpeace.txt".to_string())),
    );
}
```

Tracing & Metrics

Simple Trace Logging

The `tracing` crate has become the standard way for async and synchronous libraries to issue log messages.

The program receiving the messages needs a tracing subscriber.

Subscribers can print to the console, write to log files, write structured data to databases, send it to analytics.

The default “fmt” provider prints to the console.

```rust
#[tokio::main]
▶ Run | Debug
async fn main() {
    // Start configuring a `fmt` subscriber
    let subscriber = tracing_subscriber::fmt()
        // Use a more compact, abbreviated log format
        .compact()
        // Display source code file paths
        .with_file(true)
        // Display source code line numbers
        .with_line_number(true)
        // Display the thread ID an event was recorded on
        .with_thread_ids(true)
        // Don't display the event's target (module path)
        .with_target(false)
        // Build the subscriber
        .finish();

    // Set the subscriber as the default
    tracing::subscriber::set_global_default(subscriber).unwrap();

    // Log some events
    tracing::info!("Starting up");
    tracing::warn!("Are you sure this is a good idea?");
    tracing::error!("This is an error!");
}
```

```
2023-07-29T15:26:10.436640Z  INFO ThreadId(01) tracing_log\src\main.rs:22: Starting up
2023-07-29T15:26:10.436934Z  WARN ThreadId(01) tracing_log\src\main.rs:23: Are you sure this is a good idea?
2023-07-29T15:26:10.437174Z ERROR ThreadId(01) tracing_log\src\main.rs:24: This is an error!
```

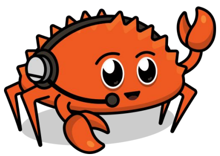
Structured Logging with Tracing

Changing log output to easily parsable JSON is easy. Change the subscriber builder.

```rust
#[tokio::main]
▶ Run | Debug
async fn main() {
    // Start configuring a `fmt` subscriber
    let subscriber: Subscriber<JsonFields, Format<…>> = tracing_subscriber::fmt().json()
        .finish();

    // Set the subscriber as the default
    tracing::subscriber::set_global_default(subscriber).unwrap();

    // Log some events
    tracing::info!("Starting up");
    tracing::warn!("Are you sure this is a good idea?");
    tracing::error!("This is an error!");
}
```




```
{"timestamp":"2023-07-29T15:30:12.066733Z","level":"INFO","fields":{"message":"Starting up"},"target":"tracing_json"}
{"timestamp":"2023-07-29T15:30:12.066967Z","level":"WARN","fields":{"message":"Are you sure this is a good idea?"},"target":"tracin
g_json"}
{"timestamp":"2023-07-29T15:30:12.067099Z","level":"ERROR","fields":{"message":"This is an error!"},"target":"tracing_json"}
```

Instrumenting Spans

The `tracing` system includes instrumentation.

Enable span events in your output.

Decorate functions to trace with `#[instrument]`

You now have call-time information and parameters logged. You can change what it logged with parameters to the macro.

You can also manually create spans inside your code.

```rust
use tracing_subscriber::fmt::format::FmtSpan;

#[tracing::instrument]
async fn hello() {
    println!("Hello World");
    tokio::time::sleep(tokio::time::Duration::from_secs(1)).await;
}

#[tokio::main]
▶ Run | Debug
async fn main() -> anyhow::Result<()> {
    // Applications that receive events need to subscribe
    //let subscriber = tracing_subscriber::FmtSubscriber::new();

    // Start configuring a `fmt` subscriber
    let subscriber = tracing_subscriber::fmt()
        // Use a more compact, abbreviated log format
        .compact()
        // Display source code file paths
        .with_file(true)
        // Display source code line numbers
        .with_line_number(true)
        // Display the thread ID an event was recorded on
        .with_thread_ids(true)
        // Don't display the event's target (module path)
        .with_target(false)
        // Add span events
        .with_span_events(FmtSpan::ENTER | FmtSpan::CLOSE)
        // Build the subscriber
        .finish();

    // Set the subscriber as the default
    tracing::subscriber::set_global_default(subscriber)?;

    hello().await;
    Ok(())
} fn main
```

```
2023-07-29T15:34:03.688526Z  INFO ThreadId(01) hello: tracing_spans\src\main.rs:3: enter
2023-07-29T15:34:03.688764Z  INFO ThreadId(01) hello: tracing_spans\src\main.rs:3: close time.busy=802µs time.idle=1.01s
```

Tokio-Console: It’s like “htop” for async

Install Tokio Console with `cargo install tokio-console`

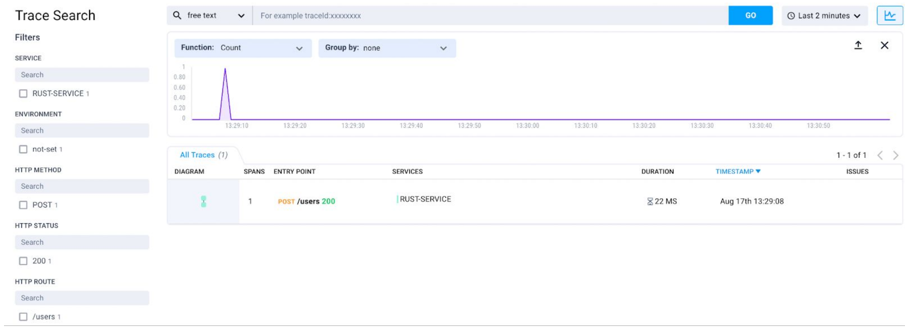
Subscribe to the console with:

```
console_subscriber::init();
```

OpenTelemetry & More

With the `tracing_opentelemetry` crate, you can connect to an *OpenTelemetry* system - and send your spans, trace points, etc.

You can then view your distributed telemetry in the various OpenTelemetry tools. Here's a screenshot of Aspecto.



Pinning

What is Pinning?

Rust’s memory model strictly ensures that references must still exist, won’t move, and won’t be dropped while still in use.

That’s *great* for avoiding common memory bugs.

It’s *tricky* in a highly asynchronous environment, tasks may depend upon other tasks - which typically move around quite a bit.

Pinning lets you tell Rust that a variable needs to stick around - in the same place - until you unpin it.

- A stream that relies upon another stream will typically pin its access to the previous stream.
- A select operation may need to pin entries for the same reason.
- Asynchronously calling yourself - recursion - requires pinning the iterations.

Pinning and Select

(Code from the Tokio examples)

Of you want to create multiple futures as variables, and operate on them - you need to make sure that they will remain valid.

Tokio's `pin!` macro makes this straightforward.

```
use tokio::{pin, select};

async fn my_async_fn() {
    // async logic here
}

#[tokio::main]
async fn main() {
    pin! {
        let future1 = my_async_fn();
        let future2 = my_async_fn();
    }

    select! {
        _ = &mut future1 => {}
        _ = &mut future2 => {}
    }
}
```

Pinning and Recursion

Async recursion is difficult, because you need to pin the futures in turn.

The `async_recursion` crate offers an easy way.

This won't compile:

```rust
async fn fib(n : u32) -> u32 {
   match n {
       0 | 1 => 1,
       _ => fib(n-1).await + fib(n-2).await
   }
}
```

Let the crate do its magic for you:

```rust
use async_recursion::async_recursion;

#[async_recursion]
async fn fib(n : u32) -> u32 {
   match n {
       0 | 1 => 1,
       _ => fib(n-1).await + fib(n-2).await
   }
}
```

Adapting a Stream In-Flight

Tokio supports *adapters* - streams that read other streams, mutate them, and output the new stream.

This can be a great way to modify data in-flight, and take advantage of stream self-pacing and yielding/awaiting.

Unfortunately, using a parent stream requires some pinning gymnastics. Use the `pin_project_lite` crate to make it easier.

```rust
pin_project! {
    struct ToUpper {
        #[pin]
        stream: tokio_stream::wrappers::LinesStream<BufReader<tokio::fs::File>>,
    }
}

impl ToUpper {
    fn new(stream: tokio_stream::wrappers::LinesStream<BufReader<tokio::fs::File>>) -> Self {
        Self { stream }
    }
}

impl tokio_stream::Stream for ToUpper {
    type Item = std::io::Result<String>;

    fn poll_next(self: std::pin::Pin<&mut Self>, cx: &mut std::task::Context<'_>) -> std::task::Poll<Option<Self::Item>> {
        self.project().stream.poll_next(cx).map(|opt: Option<Result<String, Error>>| {
            opt.map(|res: Result<String, Error>| {
                res.map(op: |line: String| {
                    line.to_uppercase() + "\n"
                })
            })
        })
    }
}
```

Async Traits

Traits can't - yet (it's being stabilized) - contain async functions by default.

This won't compile:

```
trait MyTrait {
    async fn f() {}
}
```

Examples taken from the `async_trait` crate documentation.

Using the `async_trait` crate does the hard work for you:

```
use async_trait::async_trait;

#[async_trait]
trait Advertisement {
    async fn run(&self);
}

struct Modal;

#[async_trait]
impl Advertisement for Modal {
    async fn run(&self) {
        self.render_fullscreen().await;
        for _ in 0..4u16 {
            remind_user_to_join_mailing_list().await;
        }
        self.hide_for_now().await;
    }
}
```

Questions?

All code used in this presentation is available here:

https://github.com/thebracket/Ardan-1HourAsync