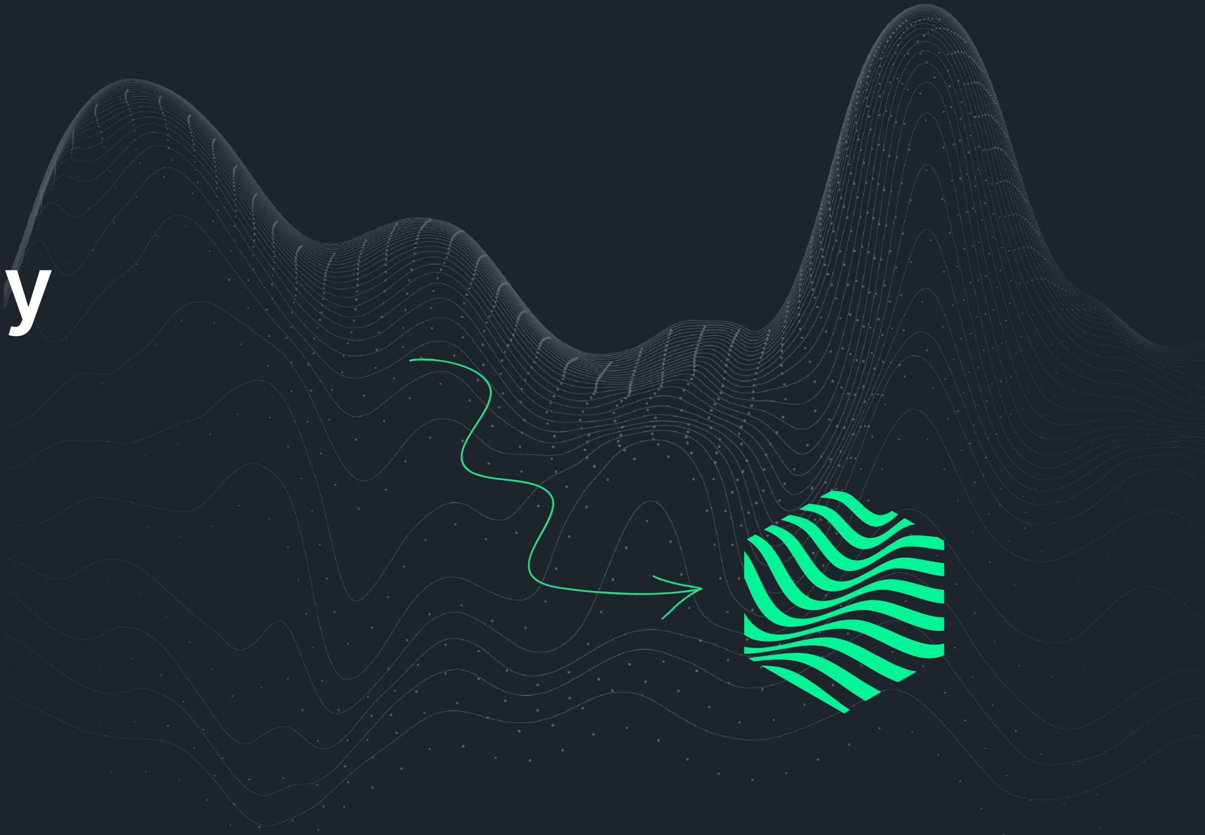
OpenTelemetry

Rust with OpenTelemetry

Tirslen Pather

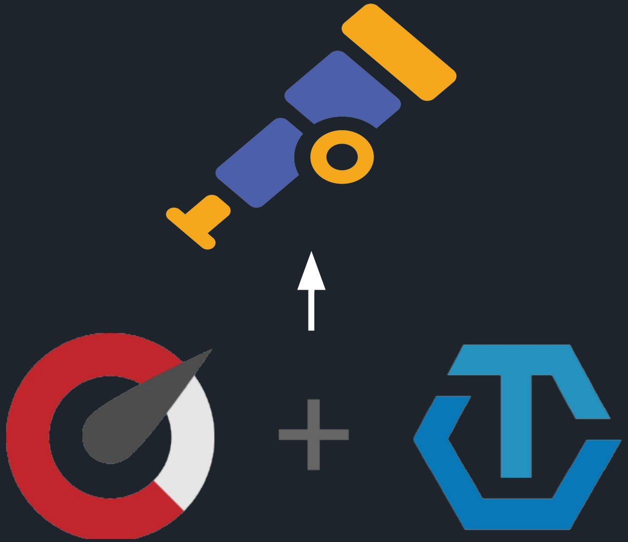
Discord: @tirslo

GitHub: Tirslo

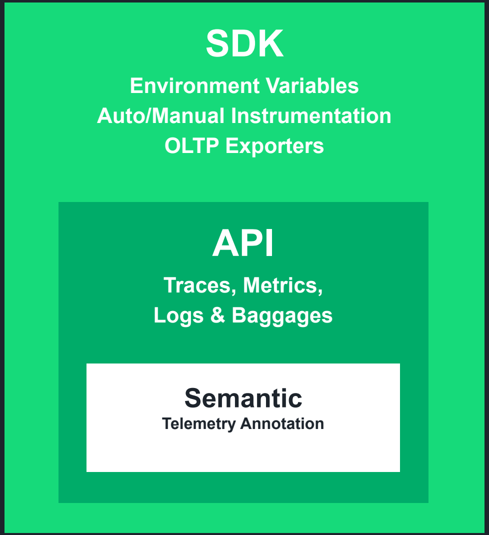


Key Facts on OpenTelemetry

- **OpenTelemetry** is a member of CNCF.
- Formed through a merger of the **OpenTracing** and **OpenCensus** projects.
- Vendor agnostic - set of APIs, libraries, integrations, and a collector service for telemetry.
- **Standardizes** how you collect telemetry data from your applications and services.
- Send it to an Observability platform of your choice.



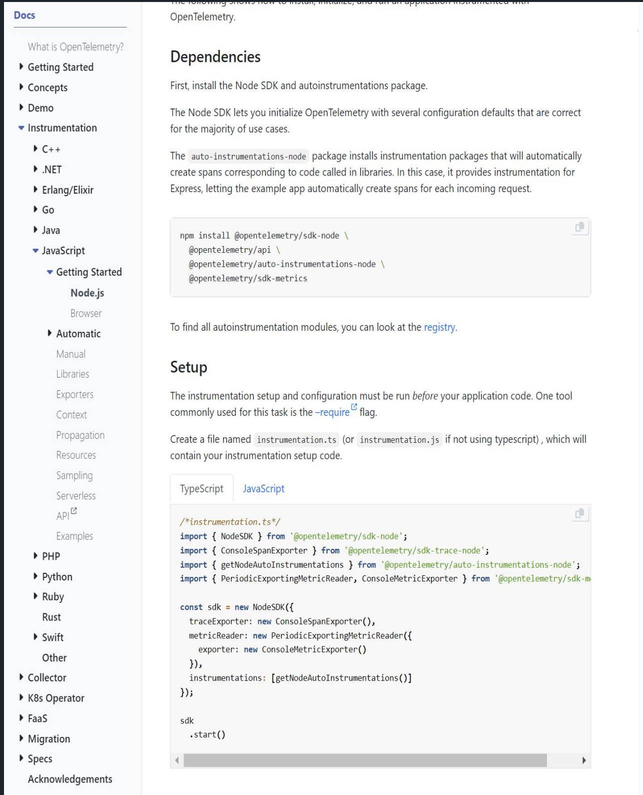
OpenTelemetry Instrumentation



Core Concepts on Instrumentation

- **Semantic Conventions** - annotate telemetry with attributes specific to the represented operation, such as HTTP calls.
- **API** - data types for tracing, metrics, and logging data.
- **SDK** - language-specific implementation of the API.
- SDKs incorporate automatic instrumentation for common libraries and frameworks for your application.
- **OpenTelemetry Protocol (OTLP)** - used to send data to your backend Observability platform of choice.

Documentation???



Docs

- What is OpenTelemetry?
- Getting Started
- Concepts
- Demo
- Instrumentation
  - C++
  - .NET
  - Erlang/Elixir
  - Go
  - Java
  - JavaScript
    - Getting Started
      - Node.js
      - Browser
    - Automatic
    - Manual
    - Libraries
    - Exporters
    - Context
    - Propagation
    - Resources
    - Sampling
    - Serverless
    - API
    - Examples
  - PHP
  - Python
  - Ruby
  - Rust
  - Swift
  - Other
- Collector
- K8s Operator
- FaaS
- Migration
- Specs
- Acknowledgements

The following shows how to install, initialize, and run an application instrumented with OpenTelemetry.

Dependencies

First, install the Node SDK and autoinstrumentations package.

The Node SDK lets you initialize OpenTelemetry with several configuration defaults that are correct for the majority of use cases.

The `auto-instrumentations-node` package installs instrumentation packages that will automatically create spans corresponding to code called in libraries. In this case, it provides instrumentation for Express, letting the example app automatically create spans for each incoming request.

```
npm install @opentelemetry/sdk-node \
  @opentelemetry/api \
  @opentelemetry/auto-instrumentations-node \
  @opentelemetry/sdk-metrics
```

To find all autoinstrumentation modules, you can look at the registry.

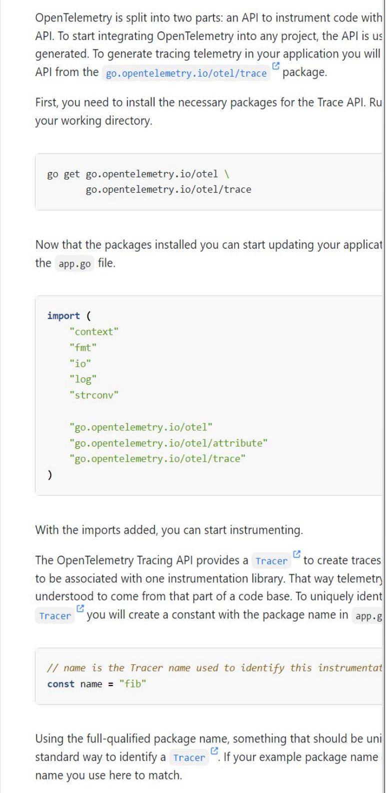
Setup

The instrumentation setup and configuration must be run *before* your application code. One tool commonly used for this task is the –require flag.

Create a file named `instrumentation.ts` (or `instrumentation.js` if not using typescript) , which will contain your instrumentation setup code.

TypeScript JavaScript

```
/*instrumentation.ts*/
import { NodeSDK } from '@opentelemetry/sdk-node';
import { ConsoleSpanExporter } from '@opentelemetry/sdk-trace-node';
import { getNodeAutoInstrumentations } from '@opentelemetry/auto-instrumentations-node';
import { PeriodicExportingMetricReader, ConsoleMetricExporter } from '@opentelemetry/sdk-m

const sdk = new NodeSDK({
  traceExporter: new ConsoleSpanExporter(),
  metricReader: new PeriodicExportingMetricReader({
    exporter: new ConsoleMetricExporter()
  }),
  instrumentations: [getNodeAutoInstrumentations()]
});

sdk
  .start()
```



OpenTelemetry is split into two parts: an API to instrument code with API. To start integrating OpenTelemetry into any project, the API is us generated. To generate tracing telemetry in your application you will API from the `go.opentelemetry.io/otel/trace` package.

First, you need to install the necessary packages for the Trace API. Ru your working directory.

```
go get go.opentelemetry.io/otel \
       go.opentelemetry.io/otel/trace
```

Now that the packages installed you can start updating your applicat the `app.go` file.

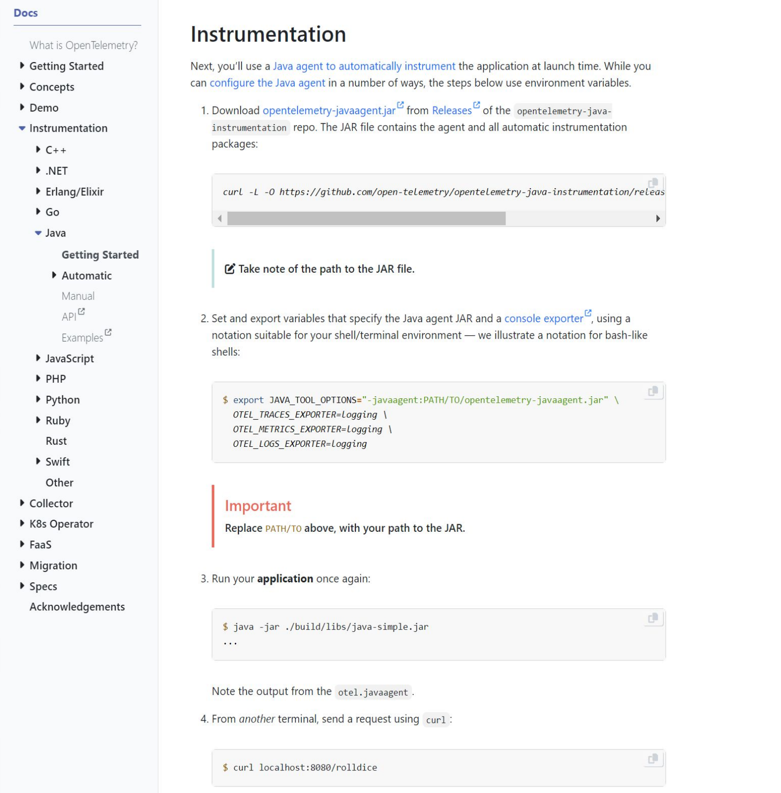
```
import (
    "context"
    "fmt"
    "io"
    "log"
    "strconv"

    "go.opentelemetry.io/otel"
    "go.opentelemetry.io/otel/attribute"
    "go.opentelemetry.io/otel/trace"
)
```

With the imports added, you can start instrumenting.

The OpenTelemetry Tracing API provides a `Tracer` to create traces to be associated with one instrumentation library. That way telemetry understood to come from that part of a code base. To uniquely ident `Tracer` you will create a constant with the package name in `app.g`

```
// name is the Tracer name used to identify this instrumentat
const name = "fib"
```

Using the full-qualified package name, something that should be uni standard way to identify a `Tracer`. If your example package name name you use here to match.



Docs

- What is OpenTelemetry?
- Getting Started
- Concepts
- Demo
- Instrumentation
  - C++
  - .NET
  - Erlang/Elixir
  - Go
  - Java
    - Getting Started
    - Automatic
    - Manual
    - API
    - Examples
  - JavaScript
  - PHP
  - Python
  - Ruby
  - Rust
  - Swift
  - Other
- Collector
- K8s Operator
- FaaS
- Migration
- Specs
- Acknowledgements

Instrumentation

Next, you'll use a Java agent to automatically instrument the application at launch time. While you can configure the Java agent in a number of ways, the steps below use environment variables.

1. Download opentelemetry-javaagent.jar from Releases of the `opentelemetry-java-instrumentation` repo. The JAR file contains the agent and all automatic instrumentation packages:

   ```
   curl -L -O https://github.com/open-telemetry/opentelemetry-java-instrumentation/releas
   ```

   > **Take note of the path to the JAR file.**

2. Set and export variables that specify the Java agent JAR and a console exporter, using a notation suitable for your shell/terminal environment — we illustrate a notation for bash-like shells:

   ```
   $ export JAVA_TOOL_OPTIONS="-javaagent:PATH/TO/opentelemetry-javaagent.jar" \
     OTEL_TRACES_EXPORTER=logging \
     OTEL_METRICS_EXPORTER=logging \
     OTEL_LOGS_EXPORTER=logging
   ```

   > Important
   >
   > **Replace `PATH/TO` above, with your path to the JAR.**

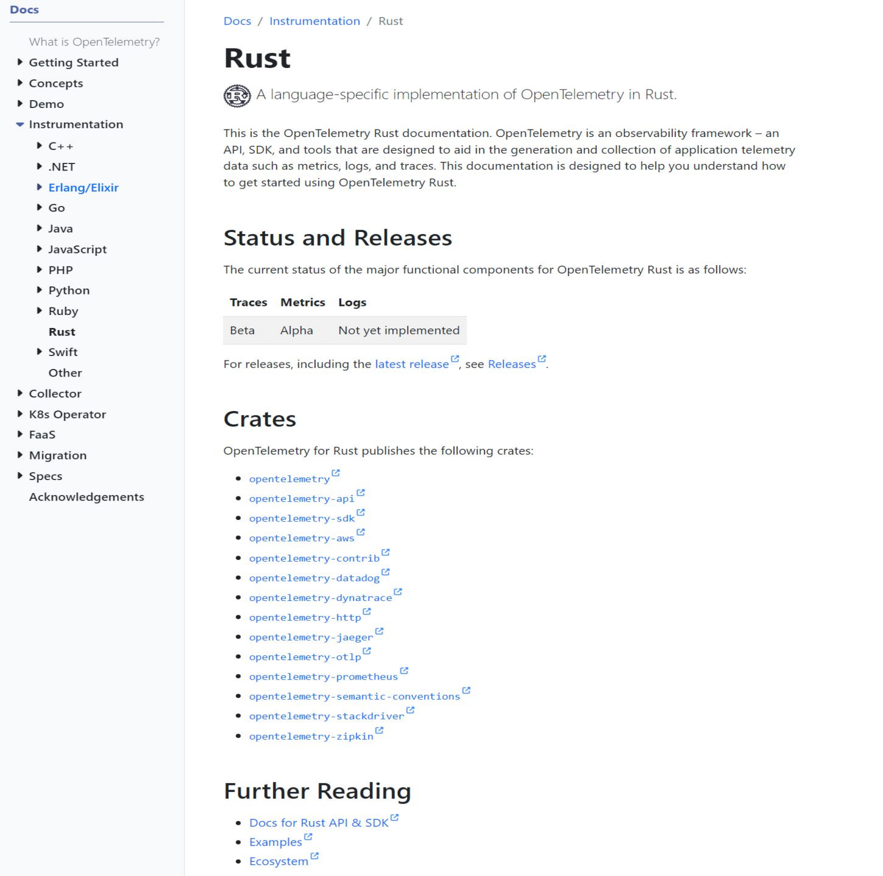
3. Run your **application** once again:

   ```
   $ java -jar ./build/libs/java-simple.jar
   ...
   ```

   Note the output from the `otel.javaagent`.

4. From *another* terminal, send a request using `curl`:

   ```
   $ curl localhost:8080/rolldice
   ```



Docs / Instrumentation / Rust

Rust

A language-specific implementation of OpenTelemetry in Rust.

This is the OpenTelemetry Rust documentation. OpenTelemetry is an observability framework – an API, SDK, and tools that are designed to aid in the generation and collection of application telemetry data such as metrics, logs, and traces. This documentation is designed to help you understand how to get started using OpenTelemetry Rust.

Status and Releases

The current status of the major functional components for OpenTelemetry Rust is as follows:

Traces	Metrics	Logs
Beta	Alpha	Not yet implemented

For releases, including the latest release, see Releases.

Crates

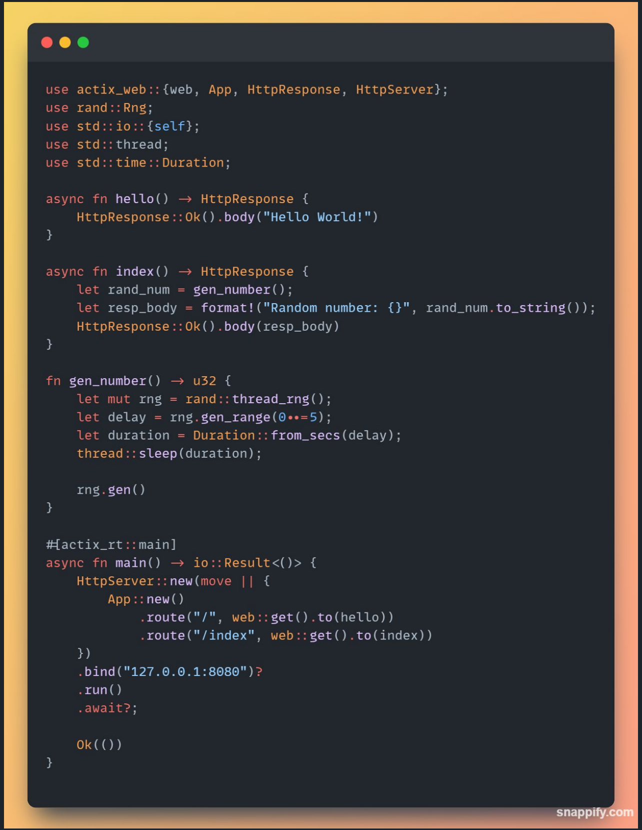
OpenTelemetry for Rust publishes the following crates:

- `opentelemetry`
- `opentelemetry-api`
- `opentelemetry-sdk`
- `opentelemetry-aws`
- `opentelemetry-contrib`
- `opentelemetry-datadog`
- `opentelemetry-dynatrace`
- `opentelemetry-http`
- `opentelemetry-jaeger`
- `opentelemetry-otlp`
- `opentelemetry-prometheus`
- `opentelemetry-semantic-conventions`
- `opentelemetry-stackdriver`
- `opentelemetry-zipkin`

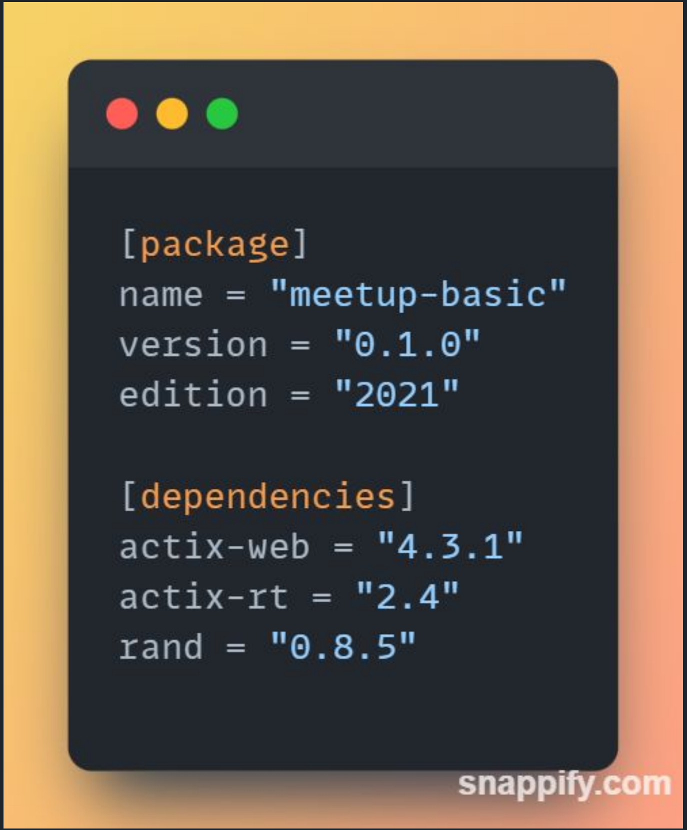
Further Reading

- Docs for Rust API & SDK
- Examples
- Ecosystem

Rust App

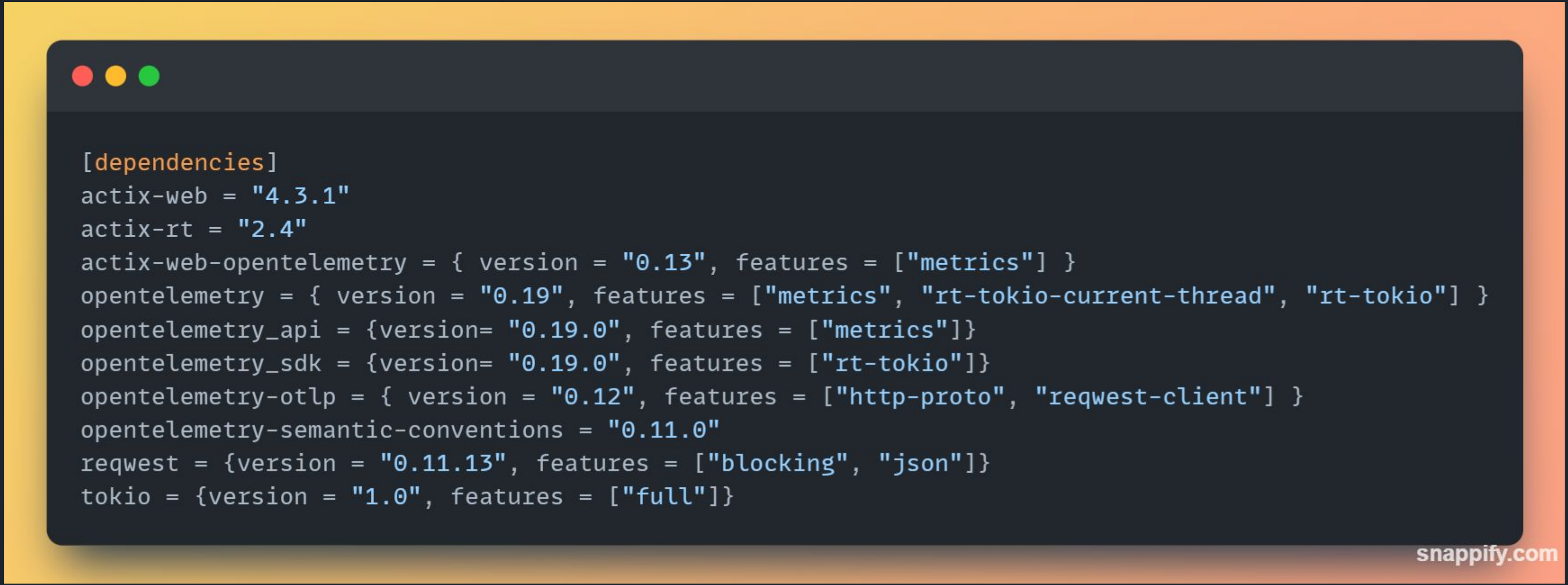


```rust
use actix_web::{web, App, HttpResponse, HttpServer};
use rand::Rng;
use std::io::{self};
use std::thread;
use std::time::Duration;

async fn hello() -> HttpResponse {
    HttpResponse::Ok().body("Hello World!")
}

async fn index() -> HttpResponse {
    let rand_num = gen_number();
    let resp_body = format!("Random number: {}", rand_num.to_string());
    HttpResponse::Ok().body(resp_body)
}

fn gen_number() -> u32 {
    let mut rng = rand::thread_rng();
    let delay = rng.gen_range(0..=5);
    let duration = Duration::from_secs(delay);
    thread::sleep(duration);

    rng.gen()
}

#[actix_rt::main]
async fn main() -> io::Result<()> {
    HttpServer::new(move || {
        App::new()
            .route("/", web::get().to(hello))
            .route("/index", web::get().to(index))
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await?;

    Ok(())
}
```



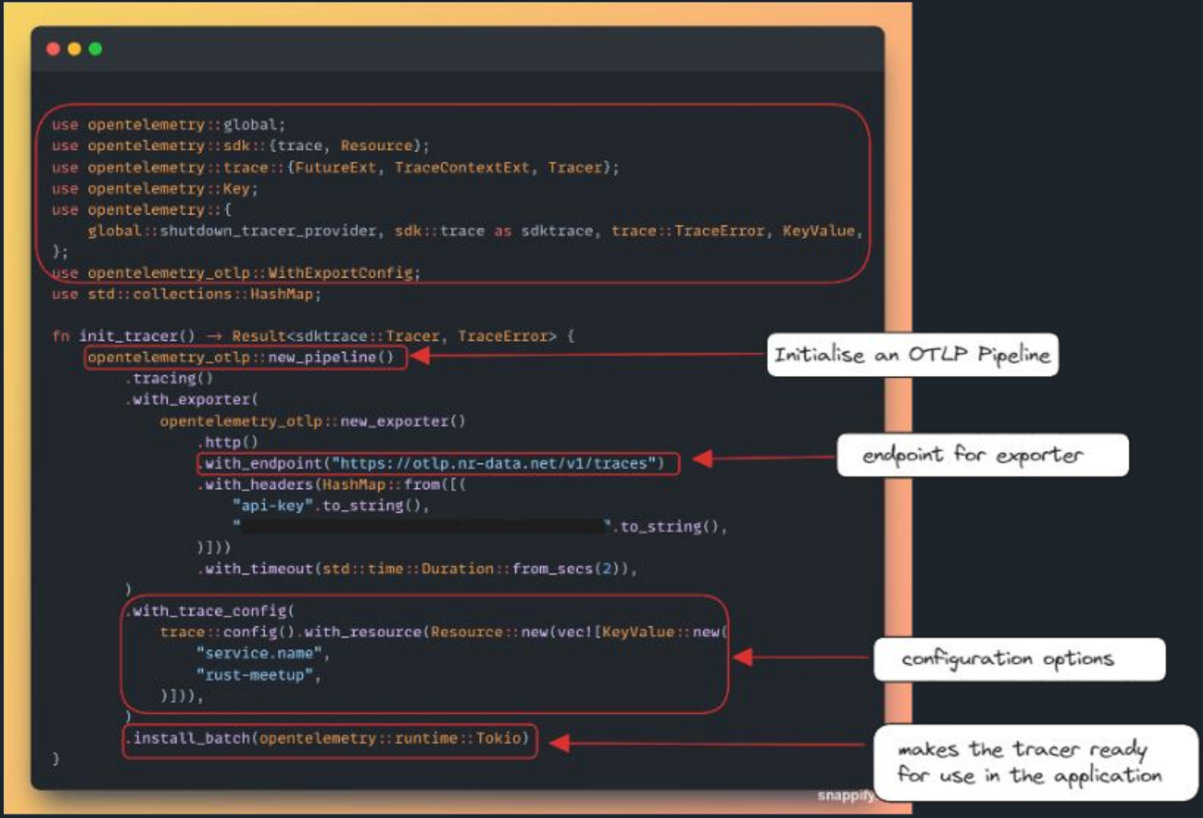
```toml
[package]
name = "meetup-basic"
version = "0.1.0"
edition = "2021"

[dependencies]
actix-web = "4.3.1"
actix-rt = "2.4"
rand = "0.8.5"
```

What's needed for Otel



```toml
[dependencies]
actix-web = "4.3.1"
actix-rt = "2.4"
actix-web-opentelemetry = { version = "0.13", features = ["metrics"] }
opentelemetry = { version = "0.19", features = ["metrics", "rt-tokio-current-thread", "rt-tokio"] }
opentelemetry_api = {version= "0.19.0", features = ["metrics"]}
opentelemetry_sdk = {version= "0.19.0", features = ["rt-tokio"]}
opentelemetry-otlp = { version = "0.12", features = ["http-proto", "reqwest-client"] }
opentelemetry-semantic-conventions = "0.11.0"
reqwest = {version = "0.11.13", features = ["blocking", "json"]}
tokio = {version = "1.0", features = ["full"]}
```

Adding Instrumentation → Tracer



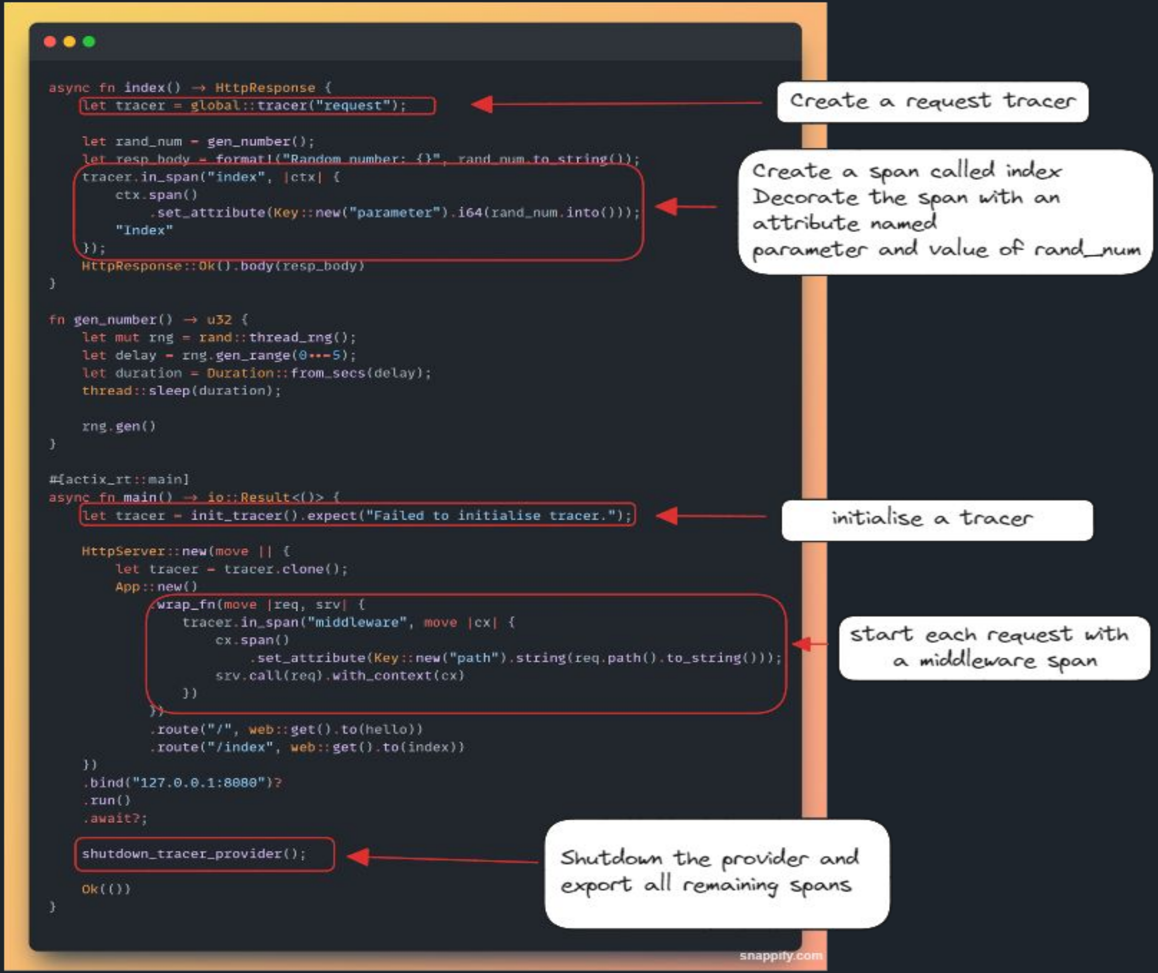
```rust
use opentelemetry::global;
use opentelemetry::sdk::{trace, Resource};
use opentelemetry::trace::{FutureExt, TraceContextExt, Tracer};
use opentelemetry::Key;
use opentelemetry::{
    global::shutdown_tracer_provider, sdk::trace as sdktrace, trace::TraceError, KeyValue,
};
use opentelemetry_otlp::WithExportConfig;
use std::collections::HashMap;

fn init_tracer() → Result<sdktrace::Tracer, TraceError> {
    opentelemetry_otlp::new_pipeline()
        .tracing()
        .with_exporter(
            opentelemetry_otlp::new_exporter()
                .http()
                .with_endpoint("https://otlp.nr-data.net/v1/traces")
                .with_headers(HashMap::from([(
                    "api-key".to_string(),
                    "[redacted]".to_string(),
                )]))
                .with_timeout(std::time::Duration::from_secs(2)),
        )
        .with_trace_config(
            trace::config().with_resource(Resource::new(vec![KeyValue::new(
                "service.name",
                "rust-meetup",
            )])),
        )
        .install_batch(opentelemetry::runtime::Tokio)
}
```

Initialise an OTLP Pipeline

endpoint for exporter

configuration options

makes the tracer ready
for use in the application



```rust
async fn index() -> HttpResponse {
    let tracer = global::tracer("request");

    let rand_num = gen_number();
    let resp_body = format!("Random number: {}", rand_num.to_string());
    tracer.in_span("index", |ctx| {
        ctx.span()
            .set_attribute(Key::new("parameter").i64(rand_num.into()));
        "Index"
    });
    HttpResponse::Ok().body(resp_body)
}

fn gen_number() -> u32 {
    let mut rng = rand::thread_rng();
    let delay = rng.gen_range(0..=5);
    let duration = Duration::from_secs(delay);
    thread::sleep(duration);

    rng.gen()
}

#[actix_rt::main]
async fn main() -> io::Result<()> {
    let tracer = init_tracer().expect("Failed to initialise tracer.");

    HttpServer::new(move || {
        let tracer = tracer.clone();
        App::new()
            .wrap_fn(move |req, srv| {
                tracer.in_span("middleware", move |cx| {
                    cx.span()
                        .set_attribute(Key::new("path").string(req.path().to_string()));
                    srv.call(req).with_context(cx)
                })
            })
            .route("/", web::get().to(hello))
            .route("/index", web::get().to(index))
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await?;

    shutdown_tracer_provider();

    Ok(())
}
```

Create a request tracer

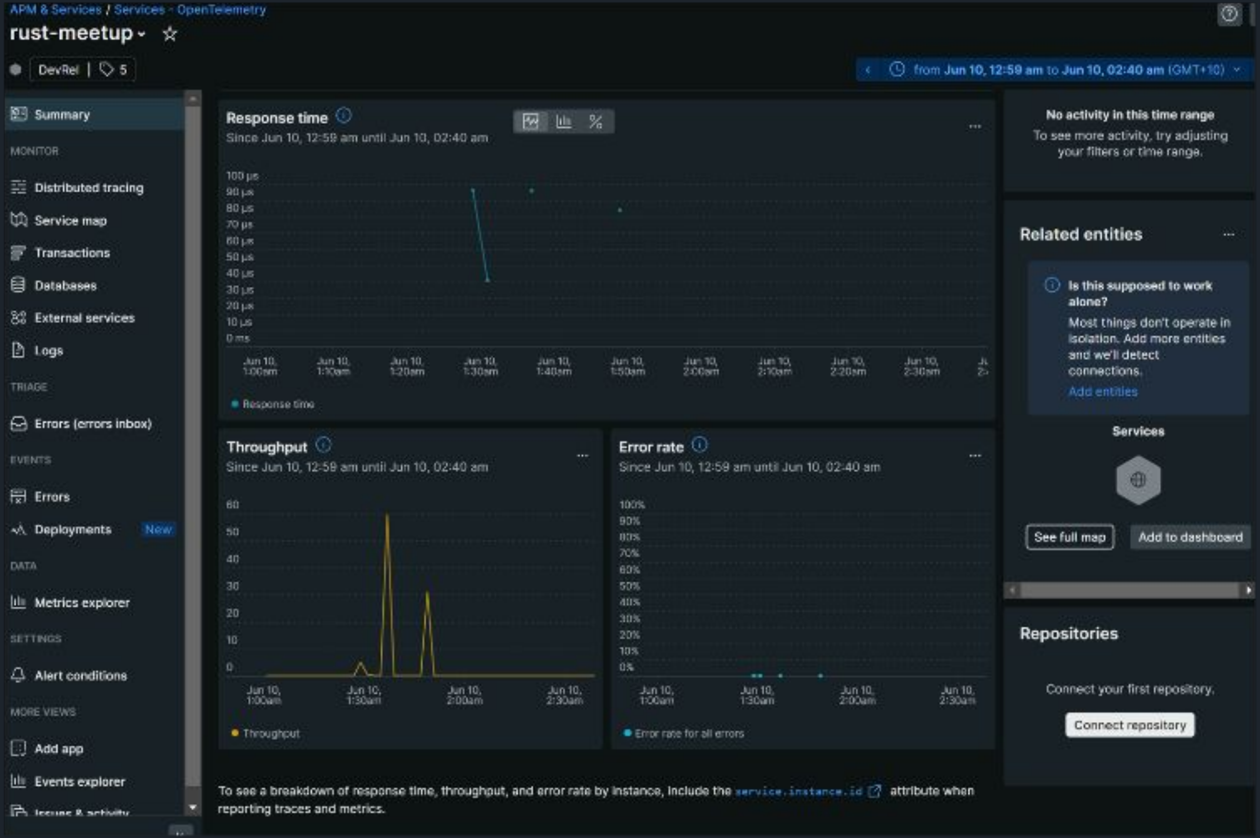
Create a span called index
Decorate the span with an
attribute named
parameter and value of rand_num

initialise a tracer

start each request with
a middleware span

Shutdown the provider and
export all remaining spans

What's it look like?



APM & Services / Services - OpenTelemetry

rust-meetup

from Jun 10, 12:59 am to Jun 10, 02:40 am (GMT+10)

Response time

Since Jun 10, 12:59 am until Jun 10, 02:40 am

Throughput

Since Jun 10, 12:59 am until Jun 10, 02:40 am

Error rate

Since Jun 10, 12:59 am until Jun 10, 02:40 am

No activity in this time range

To see more activity, try adjusting your filters or time range.

Related entities

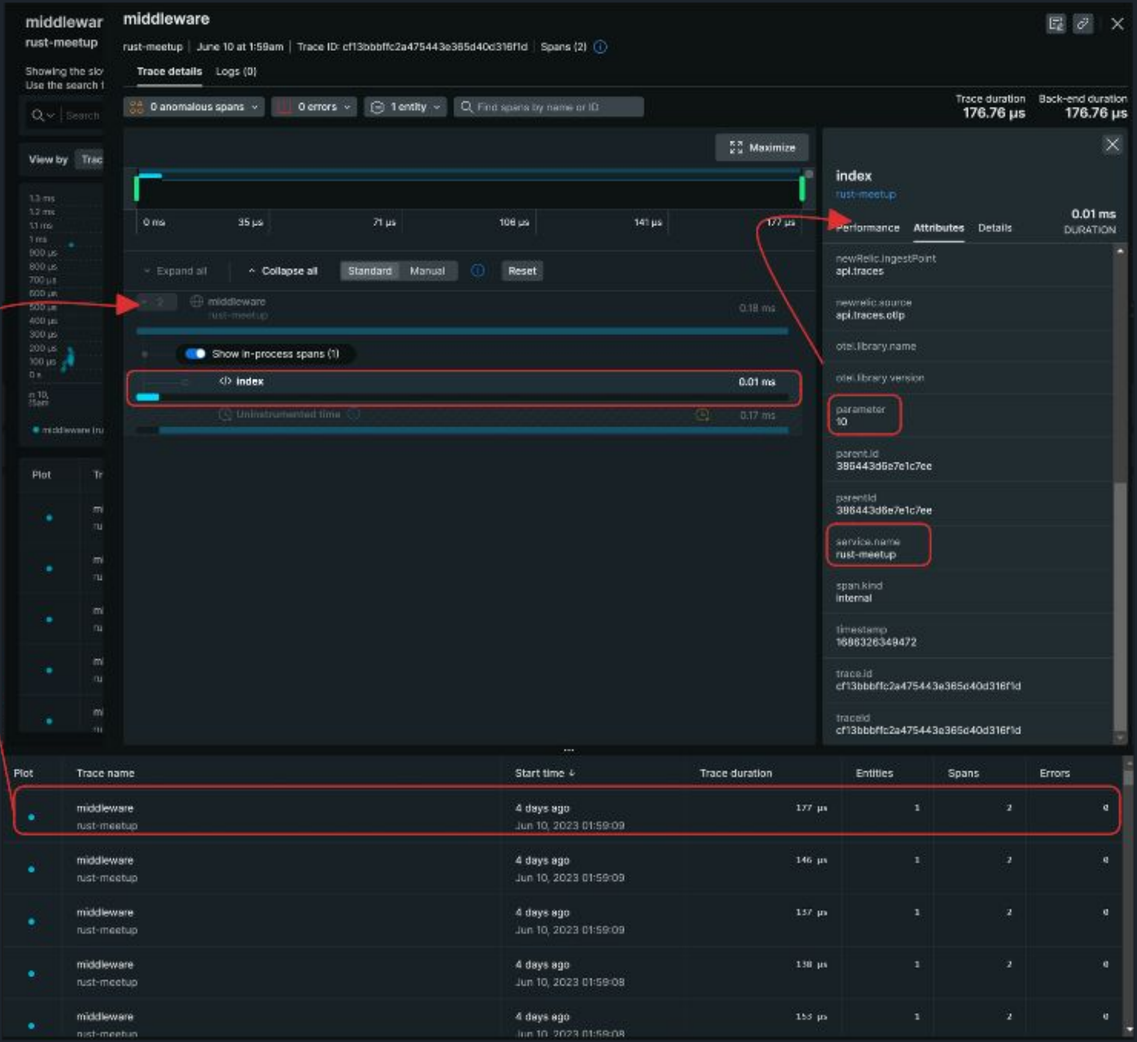
Is this supposed to work alone?

Most things don't operate in isolation. Add more entities and we'll detect connections.

Repositories

Connect your first repository.

To see a breakdown of response time, throughput, and error rate by instance, include the `service.instance.id` attribute when reporting traces and metrics.



middleware

rust-meetup | June 10 at 1:59am | Trace ID: cf13bbbffc2a475443e365d40d316f1d | Spans (2)

Trace duration 176.76 µs
Back-end duration 176.76 µs

Span	Duration
middleware rust-meetup	0.18 ms
index	0.01 ms
Uninstrumented time	0.17 ms

index
rust-meetup

0.01 ms DURATION

newRelic.ingestPoint
api.traces

newrelic.source
api.traces.otlp

otel.library.name

otel.library.version

parameter
10

parent.id
386443d6e7e1c7ee

parentId
386443d6e7e1c7ee

service.name
rust-meetup

span.kind
internal

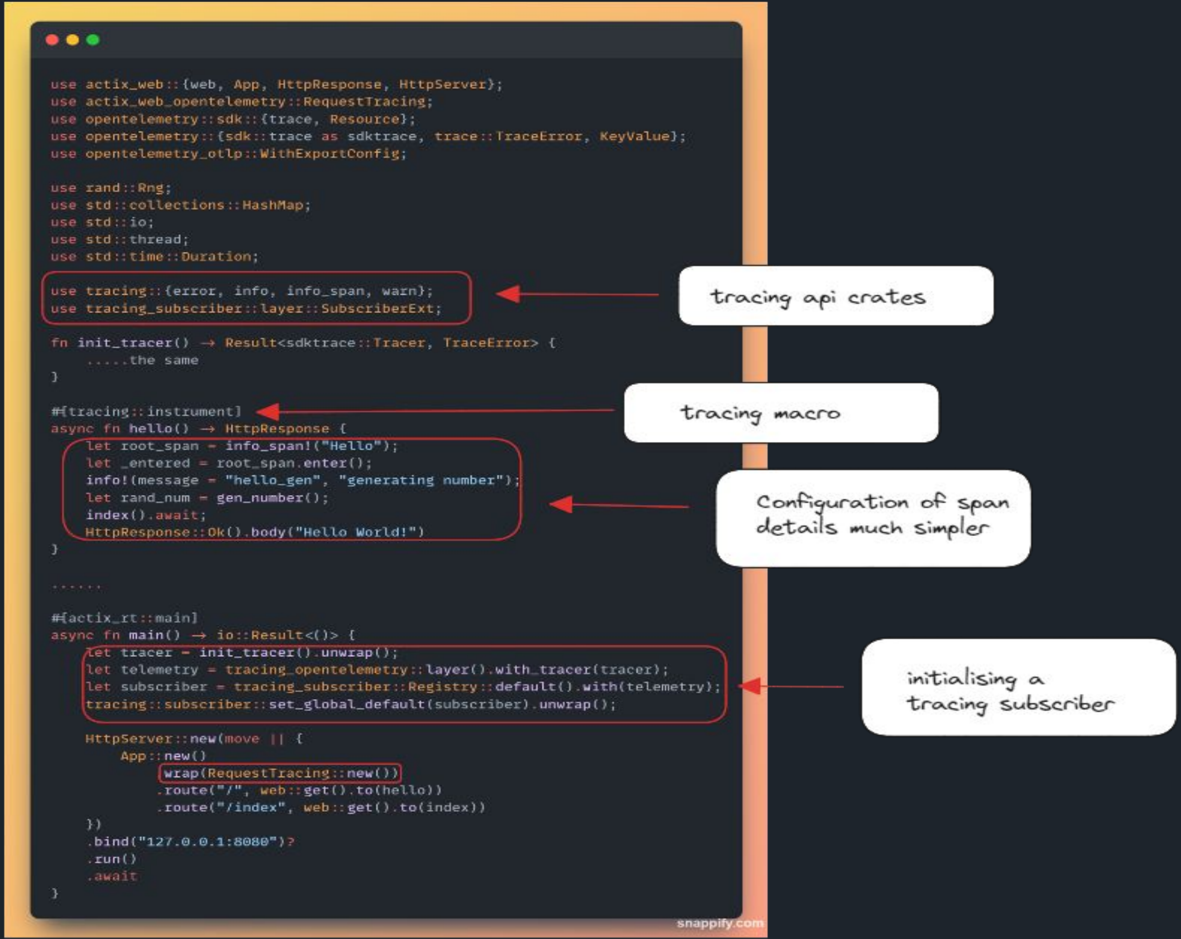
timestamp
1686326349472

trace.id
cf13bbbffc2a475443e365d40d316f1d

traceId
cf13bbbffc2a475443e365d40d316f1d

Plot	Trace name	Start time ↓	Trace duration	Entities	Spans	Errors
●	middleware rust-meetup	4 days ago Jun 10, 2023 01:59:09	177 µs	1	2	0
●	middleware rust-meetup	4 days ago Jun 10, 2023 01:59:09	146 µs	1	2	0
●	middleware rust-meetup	4 days ago Jun 10, 2023 01:59:09	137 µs	1	2	0
●	middleware rust-meetup	4 days ago Jun 10, 2023 01:59:08	138 µs	1	2	0
●	middleware rust-meetup	4 days ago Jun 10, 2023 01:59:08	153 µs	1	2	0

Tracing API



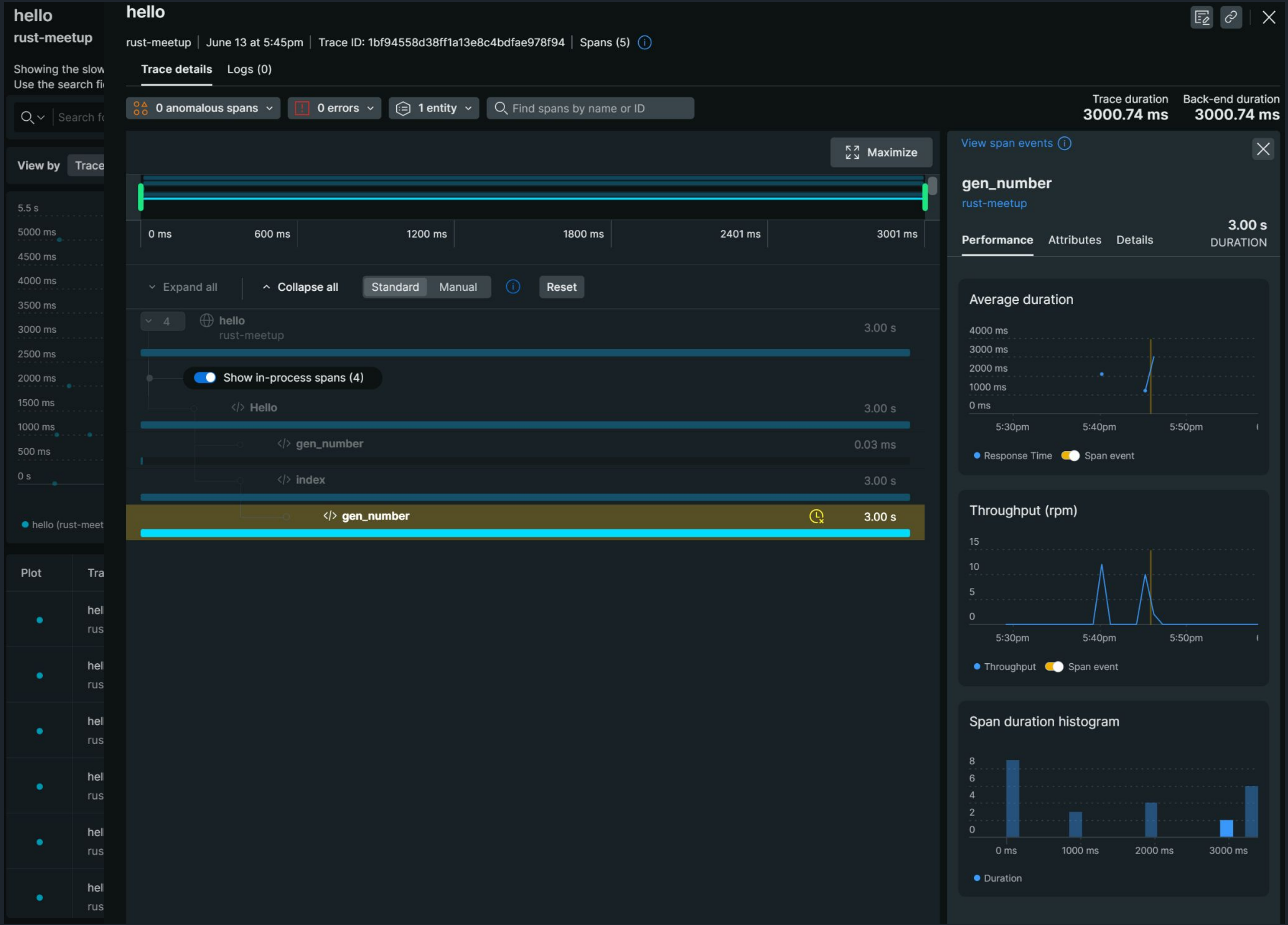
```rust
use actix_web::{web, App, HttpResponse, HttpServer};
use actix_web_opentelemetry::RequestTracing;
use opentelemetry::sdk::{trace, Resource};
use opentelemetry::{sdk::trace as sdktrace, trace::TraceError, KeyValue};
use opentelemetry_otlp::WithExportConfig;

use rand::Rng;
use std::collections::HashMap;
use std::io;
use std::thread;
use std::time::Duration;

use tracing::{error, info, info_span, warn};
use tracing_subscriber::layer::SubscriberExt;

fn init_tracer() -> Result<sdktrace::Tracer, TraceError> {
    .....the same
}

#[tracing::instrument]
async fn hello() -> HttpResponse {
    let root_span = info_span!("Hello");
    let _entered = root_span.enter();
    info!(message = "hello_gen", "generating number");
    let rand_num = gen_number();
    index().await;
    HttpResponse::Ok().body("Hello World!")
}

......

#[actix_rt::main]
async fn main() -> io::Result<()> {
    let tracer = init_tracer().unwrap();
    let telemetry = tracing_opentelemetry::layer().with_tracer(tracer);
    let subscriber = tracing_subscriber::Registry::default().with(telemetry);
    tracing::subscriber::set_global_default(subscriber).unwrap();

    HttpServer::new(move || {
        App::new()
            .wrap(RequestTracing::new())
            .route("/", web::get().to(hello))
            .route("/index", web::get().to(index))
    })
    .bind("127.0.0.1:8080")?
    .run()
    .await
}
```

tracing api crates

tracing macro

Configuration of span details much simpler

initialising a tracing subscriber



hello

rust-meetup | June 13 at 5:45pm | Trace ID: 1bf94558d38ff1a13e8c4bdfae978f94 | Spans (5)

Trace duration 3000.74 ms

Back-end duration 3000.74 ms

```
hello                    3.00 s
rust-meetup
  Show in-process spans (4)
  </> Hello              3.00 s
    </> gen_number       0.03 ms
    </> index            3.00 s
      </> gen_number     3.00 s
```

gen_number

rust-meetup

3.00 s DURATION

Documentation

- https://tokio.rs/tokio/topics/tracing
- https://opentelemetry.io/docs/instrumentation/rust
- https://github.com/open-telemetry/opentelemetry-rust