**Bring in light into your Rust Application**

[[](https://medium.com/@patrickkoss?source=post_page-----cb0a46bfc3dc--------------------------------)](https://medium.com/@patrickkoss?source=post_page-----cb0a46bfc3dc--------------------------------)

[Patrick Koss](https://medium.com/@patrickkoss?source=post_page-----cb0a46bfc3dc--------------------------------)

·

Follow

11 min read

·

May 16

9

1



Photo by [Luke Chesser](https://unsplash.com/@lukechesser?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com/?utm_source=medium&utm_medium=referral" \t "_blank)

In today’s complex, interconnected software ecosystems, understanding your application’s internal workings is not only desirable — it’s a mission-critical necessity. This crucial understanding is embodied in the term “observability”, a term borrowed from control theory that refers to how well the internal states of a system can be understood based on its external outputs.

Observability isn’t just about monitoring your application’s health; it’s about illuminating the intricate paths of your system’s behavior. Whether you’re working with a Rust application or any other software system, observability allows you to decipher what’s happening inside your system when it interacts with the world outside.

Let’s consider some of the reasons why observability is vital:  
*Debugging:* Observability is crucial when it comes to finding and fixing bugs in your application. With high observability, you can easily pinpoint the location and cause of a bug, reducing the time and effort required to debug your application. Without observability, you are essentially searching in the dark.  
*Understanding Performance:* Observability allows you to measure your application’s performance in real-time. You can monitor how quickly your application responds to user requests, how efficiently it utilizes resources, and how well it scales under heavy loads. This information can help you identify bottlenecks, optimize your application, and deliver a superior user experience.  
*Understanding Control Flow:* In any non-trivial application, understanding the control flow can be a daunting task. Observability gives you a bird’s eye view of the control flow in your application, enabling you to see how different components interact, how data flows through the system, and how various events trigger different parts of your application.

By investing in observability, you can make your Rust application — or any application — transparent and comprehensible. This transparency not only helps reduce downtime and improve performance, but it also fosters a deeper understanding of your software, making it easier to maintain, upgrade, and evolve your system over time. Observability isn’t just a nice-to-have; it’s a cornerstone of effective, efficient, and resilient software development.

**Logs & Traces**

Logs are timestamped records of events that happened in a system. They are the raw, unfiltered diary of your application, providing detailed context around what your application is doing and how it is performing. Logs can tell you where and when an event took place in your application, providing crucial context when things go wrong.

Traces, on the other hand, are a way to track a request as it propagates through various microservices and functions in your system. In a complex, distributed system, a single request can touch many different services. Tracing allows us to tie together all the logs related to a single operation, giving us a detailed view of the sequence of events related to a specific request.

**Metrics**

While logs and traces provide a qualitative view of your system, metrics provide a quantitative view. Metrics are numerical data that represent the state of a system at a particular point in time. They can be anything from CPU usage, memory consumption, to the number of active users, or the latency of a service.

Metrics are crucial for two main reasons. First, they allow for proactive problem detection. By monitoring metrics, you can identify potential issues before they impact users. For example, if memory usage is steadily increasing, it might indicate a memory leak that needs addressing. Second, they allow for trend analysis and capacity planning. By analyzing metrics over time, you can identify usage patterns, plan system growth, and optimize resource allocation.

**Constructing a Observability-Centric Actix Application**

In this exercise, we will construct a compact Actix REST API. This API will be composed of three endpoints: /healthz for health checks, /metrics for observability data, and /users for user-related operations.  
To enrich our application’s observability, we will integrate a middleware that collects metadata for each request. This middleware will keep track of successful requests, client error requests, server error requests, and the overall duration taken by each request.  
To provide us with deeper insights, we will utilize Jaeger for tracing our application’s internal workings. We will particularly focus on the /users endpoint, leveraging the robust capabilities of Jaeger's tracing to illuminate its operation.  
You can find the full code [here](https://github.com/PatrickKoss/rust-opentelemetry-example).

Let’s kick things off by examining our Cargo.toml:

[package]  
name = "rust-tracing"  
version = "0.1.0"  
edition = "2021"  
  
[dependencies]  
anyhow = "1.0.71"  
tracing = "0.1.37"  
tracing-subscriber = { version = "0.3", features = ["registry", "env-filter"] }  
tracing-opentelemetry = { version = "0.18.0"}  
opentelemetry-jaeger = { version = "0.17.0", features = ["rt-tokio-current-thread", "collector\_client", "rt-tokio", "isahc\_collector\_client"] }  
opentelemetry = { version = "0.18.0", features = ["rt-tokio-current-thread","trace"] }  
tokio = { version = "1", features = ["full"] }  
actix-web = "4.3.1"  
dotenv = "0.15.0"  
serde = { version = "1.0", features = ["derive"] }  
serde\_json = "1.0.96"  
log = "0.4.17"  
env\_logger = "0.10.0"  
serde\_derive = "1.0.163"  
tracing-bunyan-formatter = "0.3.7"  
opentelemetry-prometheus = { version = "0.11.0", features = ["prometheus-encoding"] }  
prometheus = { version = "0.13", features = ["default"] }  
actix-web-opentelemetry = { version = "0.13.0", features = ["metrics", "prometheus", "metrics-prometheus", "opentelemetry-prometheus"] }  
opentelemetry\_sdk = { version = "0.18.0", features = ["metrics", "rt-tokio-current-thread", "trace"] }  
futures-util = "0.3.28"  
opentelemetry\_api = "0.19.0"

Don’t forget to incorporate the necessary features. If you encounter error messages like:

error[E0277]: the trait bound `opentelemetry\_sdk::trace::tracer::Tracer: tracing\_opentelemetry::PreSampledTracer` is not satisfied  
 --> src\main.rs:85:19  
 |  
85 | .with(telemetry)  
 | ---- ^^^^^^^^^ the trait `tracing\_opentelemetry::PreSampledTracer` is not implemented for `opentelemetry\_sdk::trace::tracer::Tracer`  
 | |  
 | required by a bound introduced by this call  
 |  
 = help: the following other types implement trait `tracing\_opentelemetry::PreSampledTracer`:  
 opentelemetry::trace::noop::NoopTracer  
 opentelemetry\_sdk::trace::Tracer  
 = note: required for `tracing\_opentelemetry::OpenTelemetryLayer<tracing\_subscriber::Registry, opentelemetry\_sdk::trace::tracer::Tracer>` to implement `tracing\_subscriber::Layer<tracing\_subscriber::Registry>`  
note: required by a bound in `tracing\_subscriber::prelude::\_\_tracing\_subscriber\_SubscriberExt::with`  
 --> C:\Users\pati9\.cargo\registry\src\github.com-1ecc6299db9ec823\tracing-subscriber-0.3.17\src\layer\mod.rs:1485:12  
 |  
1485 | L: Layer<Self>,  
 | ^^^^^^^^^^^ required by this bound in `tracing\_subscriber::prelude::\_\_tracing\_subscriber\_SubscriberExt::with`  
  
  
error[E0425]: cannot find function `meter` in module `global`  
 --> src\main.rs:100:29  
 |  
100 | let meter = global::meter("my-app");  
 | ^^^^^ not found in `global`

This usually signifies that certain features are absent. I personally ran into this hurdle and let me assure you, debugging wasn’t a walk in the park. I had to delve into the crate’s code to realize I had omitted the required features.

Now, let’s move onto the initialization of traces and metrics. For this purpose, I crafted a file ./src/telemetry.rs.

use actix\_web\_opentelemetry::{PrometheusMetricsHandler};  
use dotenv::dotenv;  
use opentelemetry::global;  
use opentelemetry\_sdk::export::metrics::aggregation;  
use opentelemetry\_sdk::metrics::{controllers, processors, selectors};  
use std::env;  
use tracing\_bunyan\_formatter::{BunyanFormattingLayer, JsonStorageLayer};  
use tracing\_subscriber::{EnvFilter, Registry};  
use tracing\_subscriber::layer::SubscriberExt;  
  
pub fn init\_tracer() {  
 dotenv().ok();  
 let app\_name = env::var("CARGO\_BIN\_NAME").unwrap\_or\_else(|\_| "api".to\_string());  
  
 global::set\_text\_map\_propagator(opentelemetry\_jaeger::Propagator::new());  
  
 let tracer = opentelemetry\_jaeger::new\_agent\_pipeline()  
 .with\_endpoint(std::env::var("JAEGER\_AGENT\_ENDPOINT").unwrap\_or\_else(|\_| "localhost:6831".to\_string()))  
 .with\_service\_name(app\_name.clone())  
 .install\_batch(opentelemetry::runtime::Tokio)  
 .expect("Failed to install OpenTelemetry tracer");  
  
 let telemetry = tracing\_opentelemetry::layer().with\_tracer(tracer);  
 let env\_filter = EnvFilter::try\_from\_default\_env().unwrap\_or\_else(|\_| EnvFilter::new("INFO"));  
 let formatting\_layer = BunyanFormattingLayer::new(app\_name, std::io::stdout);  
 let subscriber = Registry::default()  
 .with(telemetry)  
 .with(JsonStorageLayer)  
 .with(formatting\_layer)  
 .with(env\_filter);  
 tracing::subscriber::set\_global\_default(subscriber)  
 .expect("Failed to install `tracing` subscriber.");  
}  
  
#[derive(Debug, Clone)]  
pub struct Prometheus {  
 metrics\_handler: PrometheusMetricsHandler,  
}  
  
impl Prometheus {  
 pub fn new() -> Self {  
 let controller = controllers::basic(  
 processors::factory(  
 selectors::simple::histogram([0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 0.7, 1.0, 2.0]),  
 aggregation::cumulative\_temporality\_selector(),  
 ).with\_memory(true),  
 ).build();  
 let prometheus\_exporter = opentelemetry\_prometheus::exporter(controller).init();  
 let metrics\_handler = PrometheusMetricsHandler::new(prometheus\_exporter);  
  
 Self {  
 metrics\_handler,  
 }  
 }  
  
 pub fn metrics\_handler(&self) -> PrometheusMetricsHandler {  
 self.metrics\_handler.clone()  
 }  
}  
  
impl Default for Prometheus {  
 fn default() -> Self {  
 Self::new()  
 }  
}

This piece of code creates a tracer that dispatches traces to the localhost:6831 address, which is typically the address of the Jaeger agent. We also globally register the tracer. This enables us to use the #[instrument] attribute, which automatically generates spans for a method marked with this attribute. Lastly, we put together a metrics handler which we will later invoke in the metrics endpoint. By setting this up via opentelemetry\_prometheus, we can register new metrics throughout our application at any later stage.

Next on our agenda is the construction of the request metadata middleware. To do this, I’ve put together a file named ./src/request\_metadata\_middleware.rs:

use actix\_web::{  
 dev::{forward\_ready, Service, ServiceRequest, ServiceResponse, Transform},  
 Error,  
};  
use anyhow::Result;  
use futures\_util::future::LocalBoxFuture;  
use opentelemetry::{Context, global, KeyValue};  
use opentelemetry::metrics::Counter;  
use opentelemetry::metrics::Histogram;  
use std::future::{ready, Ready};  
use std::sync::Arc;  
use std::time::Instant;  
  
pub struct RequestMetadata {}  
  
impl<S, B> Transform<S, ServiceRequest> for RequestMetadata  
 where  
 S: Service<ServiceRequest, Response=ServiceResponse<B>, Error=Error>,  
 S::Future: 'static,  
 B: 'static,  
{  
 type Response = ServiceResponse<B>;  
 type Error = Error;  
 type InitError = ();  
 type Transform = RequestMetadataMiddleware<S>;  
 type Future = Ready<Result<Self::Transform, Self::InitError>>;  
  
 fn new\_transform(&self, service: S) -> Self::Future {  
 let meter = global::meter("api");  
  
 let http\_requests\_total = meter.u64\_counter("http\_requests\_total")  
 .with\_description("Number of total HTTP requests")  
 .init();  
  
 let http\_requests\_4xx\_total = meter.u64\_counter("http\_requests\_4xx\_total")  
 .with\_description("Number of total 4xx HTTP requests")  
 .init();  
  
 let http\_requests\_5xx\_total = meter.u64\_counter("http\_requests\_5xx\_total")  
 .with\_description("Number of total 5xx HTTP requests")  
 .init();  
  
 let http\_requests\_2xx\_total = meter.u64\_counter("http\_requests\_2xx\_total")  
 .with\_description("Number of total 2xx HTTP requests")  
 .init();  
  
 let http\_requests\_duration\_seconds = meter.f64\_histogram("http\_requests\_duration\_seconds")  
 .with\_description("HTTP request duration in seconds for all requests")  
 .init();  
  
 ready(Ok(RequestMetadataMiddleware {  
 service,  
 http\_requests\_total: Arc::new(http\_requests\_total),  
 http\_requests\_duration\_seconds: Arc::new(http\_requests\_duration\_seconds),  
 http\_requests\_2xx\_total: Arc::new(http\_requests\_2xx\_total),  
 http\_requests\_4xx\_total: Arc::new(http\_requests\_4xx\_total),  
 http\_requests\_5xx\_total: Arc::new(http\_requests\_5xx\_total),  
 }))  
 }  
}  
  
pub struct RequestMetadataMiddleware<S> {  
 service: S,  
 http\_requests\_4xx\_total: Arc<Counter<u64>>,  
 http\_requests\_5xx\_total: Arc<Counter<u64>>,  
 http\_requests\_2xx\_total: Arc<Counter<u64>>,  
 http\_requests\_total: Arc<Counter<u64>>,  
 http\_requests\_duration\_seconds: Arc<Histogram<f64>>,  
}  
  
impl<S, B> Service<ServiceRequest> for RequestMetadataMiddleware<S>  
 where  
 S: Service<ServiceRequest, Response=ServiceResponse<B>, Error=Error>,  
 S::Future: 'static,  
 B: 'static,  
{  
 type Response = ServiceResponse<B>;  
 type Error = Error;  
 type Future = LocalBoxFuture<'static, Result<Self::Response, Self::Error>>;  
  
 forward\_ready!(service);  
  
 fn call(&self, req: ServiceRequest) -> Self::Future {  
 let start\_time = Instant::now();  
 let fut = self.service.call(req);  
  
 let http\_requests\_4xx\_total = self.http\_requests\_4xx\_total.clone();  
 let http\_requests\_5xx\_total = self.http\_requests\_5xx\_total.clone();  
 let http\_requests\_2xx\_total = self.http\_requests\_2xx\_total.clone();  
 let http\_requests\_total = self.http\_requests\_total.clone();  
 let http\_requests\_duration\_seconds = self.http\_requests\_duration\_seconds.clone();  
  
 Box::pin(async move {  
 let res = fut.await?;  
 let elapsed = start\_time.elapsed();  
  
 let cx = Context::current();  
 let status = res.response().status();  
 let path = res.request().path().to\_string();  
 let method = res.request().method().to\_string();  
 let status\_key\_value = status.as\_str().to\_string();  
 let key\_value = &[KeyValue::new("path", path), KeyValue::new("method", method), KeyValue::new("status", status\_key\_value)];  
  
 if status.is\_client\_error() {  
 http\_requests\_4xx\_total.add(&cx, 1, key\_value);  
 } else if status.is\_server\_error() {  
 http\_requests\_5xx\_total.add(&cx, 1, key\_value);  
 } else if status.is\_success() {  
 http\_requests\_2xx\_total.add(&cx, 1, key\_value);  
 }  
  
 http\_requests\_total.add(&cx, 1, &[]);  
 let duration = elapsed.as\_millis() as f64 / 1000\_f64;  
 http\_requests\_duration\_seconds.record(&cx, duration, key\_value);  
  
 Ok(res)  
 })  
 }  
}

This forms the skeleton of a standard Actix middleware. In the new\_transform method, we boot up our metrics. As previously highlighted, by utilizing opentelemetry::global::meter, we can contribute our metrics to the registry, which will subsequently be displayed on the /metrics endpoint. When we reach the call() method, we initiate a timer start\_time that will be employed for measuring our request duration. After obtaining the current time, we forward the request using self.service.call(req) to the subsequent middlewares until they arrive at the handler. Once the handler has completed its operation, we can conclude the request by gathering our metrics.

Before we wrap up our application, let’s create a semblance of operational logic. As mentioned before, our aim is to illustrate application tracing using some mock user logic. To this end, I’ve created a file named ./src/user\_service.rs:

use std::time::Instant;  
use opentelemetry::{Context, global, KeyValue};  
use opentelemetry::metrics::{Counter, Histogram};  
use tracing::instrument;  
use tokio::time::{Duration, sleep};  
  
#[derive(Debug)]  
pub struct UserService {  
 user\_repository: UserRepository,  
 user\_repository\_success\_total: Counter<u64>,  
 user\_repository\_duration\_seconds: Histogram<f64>,  
}  
  
impl UserService {  
 pub fn new(user\_repository: UserRepository) -> Self {  
 let meter = global::meter("api");  
 let user\_repository\_success\_total = meter.u64\_counter("user\_repository\_success\_total")  
 .with\_description("Number of total user repository success")  
 .init();  
 let user\_repository\_duration\_seconds = meter.f64\_histogram("user\_repository\_duration\_seconds")  
 .with\_description("UserRepository duration in seconds")  
 .init();  
 Self { user\_repository, user\_repository\_success\_total, user\_repository\_duration\_seconds }  
 }  
  
 #[instrument(name = "UserService::validate")]  
 pub async fn validate(&self) {  
 sleep(Duration::from\_millis(2)).await;  
 }  
  
 #[instrument(name = "UserService::create")]  
 pub async fn create(&self) {  
 let start\_time = Instant::now();  
  
 self.validate().await;  
 self.user\_repository.create().await;  
  
 let cx = Context::current();  
 let elapsed = start\_time.elapsed();  
 self.user\_repository\_success\_total.add(&cx, 1, &[KeyValue::new("action", "create")]);  
 let duration = elapsed.as\_millis() as f64 / 1000\_f64;  
 self.user\_repository\_duration\_seconds.record(&cx, duration, &[KeyValue::new("action", "create")]);  
 }  
}  
  
#[derive(Debug)]  
pub struct UserRepository {}  
  
impl UserRepository {  
 pub fn new() -> Self {  
 Self { }  
 }  
  
 #[instrument(name = "UserRepository::begin")]  
 pub async fn begin(&self) {  
 sleep(Duration::from\_millis(3)).await;  
 }  
  
 #[instrument(name = "UserRepository::commit")]  
 pub async fn commit(&self) {  
 sleep(Duration::from\_millis(4)).await;  
 }  
  
 #[instrument(name = "UserRepository::create")]  
 pub async fn create(&self) {  
 self.begin().await;  
 sleep(Duration::from\_millis(1)).await;  
 self.commit().await;  
 }  
}

In the majority of business applications, there’s typically a service that validates certain data, and then calls upon the repository (often an adapter around a database like PostgreSQL) to save this data. The repository initiates a transaction, inserts the data, and then commits the transaction. It’s worth considering the delegation of transaction handling to a unit of work for better manageability. To gain clearer insights from the spans, I’ve renamed them to match the pattern StructName::functionName. This can be easily achieved by utilizing the attribute #[instrument(name = "StructName::functionName")].

Now, it’s time to bring all our pieces together in the main application file, ./src/main.rs:

use std::env;  
  
use actix\_web::{App, get, Handler, HttpRequest, HttpResponse, HttpServer, middleware, post, Responder, web};  
use actix\_web\_opentelemetry::RequestTracing;  
use anyhow::Result;  
use serde::{Deserialize, Serialize};  
  
mod user\_service;  
mod telemetry;  
mod request\_metadata\_middleware;  
  
#[derive(Debug, Clone, Serialize, Deserialize)]  
struct Message {  
 pub message: String,  
}  
  
#[get("/healthz")]  
#[tracing::instrument]  
async fn health() -> HttpResponse {  
 HttpResponse::Ok().json(Message {  
 message: "healthy".to\_string(),  
 })  
}  
  
#[post("/users")]  
#[tracing::instrument(name = "CreateUserUseCase")]  
async fn users() -> HttpResponse {  
 let user\_repo = user\_service::UserRepository::new();  
 let user\_service = user\_service::UserService::new(user\_repo);  
 user\_service.create().await;  
  
 HttpResponse::Ok().json(Message {  
 message: "healthy".to\_string(),  
 })  
}  
  
async fn not\_found() -> HttpResponse {  
 HttpResponse::NotFound().json(Message {  
 message: "not found".to\_string(),  
 })  
}  
  
#[get("/metrics")]  
async fn metrics(prometheus\_data: web::Data<telemetry::Prometheus>, request: HttpRequest) -> impl Responder {  
 prometheus\_data.metrics\_handler().call(request).await  
}  
  
#[actix\_web::main]  
async fn main() -> Result<()> {  
 env\_logger::init\_from\_env(env\_logger::Env::new().default\_filter\_or("info"));  
  
 let port = env::var("PORT").unwrap\_or\_else(|\_| "8080".to\_string());  
  
 telemetry::init\_tracer();  
  
 let prometheus = telemetry::Prometheus::new();  
 let prometheus\_data = web::Data::new(prometheus.clone());  
  
 HttpServer::new(move || {  
 App::new()  
 .app\_data(prometheus\_data.clone())  
 .wrap(request\_metadata\_middleware::RequestMetadata {})  
 .wrap(middleware::Logger::default())  
 .wrap(RequestTracing::new())  
 .service(health)  
 .service(metrics)  
 .service(users)  
 .default\_service(web::route().to(not\_found))  
 })  
 .bind(format!("0.0.0.0:{}", port))?  
 .run()  
 .await.expect("failed to run server");  
  
 Ok(())  
}

Within the main() function, we kick off by initializing a logger and our tracer. Moving to the application itself, we feed the metrics handler as app\_data, making it accessible at the /metrics endpoint. We incorporate our request metadata middleware using the wrap() method. Here, we employ the third-party package actix\_web\_opentelemetry::RequestTracing to introduce a tracing middleware. This middleware will append a span to each request and tag some request metadata accordingly.

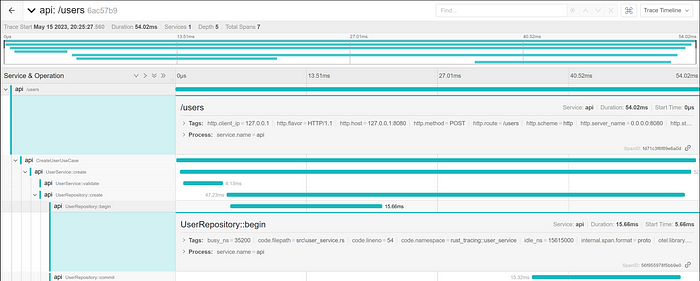
To kickstart Jaeger, we can house it within a Docker container. To do this, we need to create a docker-compose.yaml file in the root directory:

version: '3'  
services:  
 jaeger:  
 image: jaegertracing/all-in-one:latest  
 ports:  
 - "16686:16686"  
 - "14268:14268"  
 - "6831:6831/udp"  
 - "6832:6832/udp"  
 environment:  
 - COLLECTOR\_OTLP\_ENABLED=true  
 - LOG\_LEVEL=debug

To create a trace, we start both Jaeger and our application, then generate a request with the following command:

docker-compose up -d jaeger && cargo run && curl -X POST http://localhost:8080/users

By navigating to the Jaeger UI at http://localhost:16686, we can observe the resulting trace:



Moreover, we can examine some metrics by visiting <http://localhost:8080/metrics.>

http\_requests\_duration\_seconds\_bucket{method="POST",path="/users",service\_name="unknown\_service",status="200",le="0.005"} 0  
http\_requests\_duration\_seconds\_bucket{method="POST",path="/users",service\_name="unknown\_service",status="200",le="0.01"} 0  
http\_requests\_duration\_seconds\_bucket{method="POST",path="/users",service\_name="unknown\_service",status="200",le="0.02"} 0  
http\_requests\_duration\_seconds\_bucket{method="POST",path="/users",service\_name="unknown\_service",status="200",le="0.05"} 0  
http\_requests\_duration\_seconds\_bucket{method="POST",path="/users",service\_name="unknown\_service",status="200",le="0.1"} 1  
http\_requests\_duration\_seconds\_bucket{method="POST",path="/users",service\_name="unknown\_service",status="200",le="0.2"} 1  
http\_requests\_duration\_seconds\_bucket{method="POST",path="/users",service\_name="unknown\_service",status="200",le="0.5"} 1  
http\_requests\_duration\_seconds\_bucket{method="POST",path="/users",service\_name="unknown\_service",status="200",le="0.7"} 1  
http\_requests\_duration\_seconds\_bucket{method="POST",path="/users",service\_name="unknown\_service",status="200",le="1"} 1  
http\_requests\_duration\_seconds\_bucket{method="POST",path="/users",service\_name="unknown\_service",status="200",le="2"} 1  
http\_requests\_duration\_seconds\_bucket{method="POST",path="/users",service\_name="unknown\_service",status="200",le="+Inf"} 1  
http\_requests\_duration\_seconds\_sum{method="POST",path="/users",service\_name="unknown\_service",status="200"} 0.059  
http\_requests\_duration\_seconds\_count{method="POST",path="/users",service\_name="unknown\_service",status="200"} 1  
user\_repository\_duration\_seconds\_bucket{action="create",service\_name="unknown\_service",le="0.005"} 0  
user\_repository\_duration\_seconds\_bucket{action="create",service\_name="unknown\_service",le="0.01"} 0  
user\_repository\_duration\_seconds\_bucket{action="create",service\_name="unknown\_service",le="0.02"} 0  
user\_repository\_duration\_seconds\_bucket{action="create",service\_name="unknown\_service",le="0.05"} 0  
user\_repository\_duration\_seconds\_bucket{action="create",service\_name="unknown\_service",le="0.1"} 1  
user\_repository\_duration\_seconds\_bucket{action="create",service\_name="unknown\_service",le="0.2"} 1  
user\_repository\_duration\_seconds\_bucket{action="create",service\_name="unknown\_service",le="0.5"} 1  
user\_repository\_duration\_seconds\_bucket{action="create",service\_name="unknown\_service",le="0.7"} 1  
user\_repository\_duration\_seconds\_bucket{action="create",service\_name="unknown\_service",le="1"} 1  
user\_repository\_duration\_seconds\_bucket{action="create",service\_name="unknown\_service",le="2"} 1  
user\_repository\_duration\_seconds\_bucket{action="create",service\_name="unknown\_service",le="+Inf"} 1  
user\_repository\_duration\_seconds\_sum{action="create",service\_name="unknown\_service"} 0.057  
user\_repository\_duration\_seconds\_count{action="create",service\_name="unknown\_service"} 1

**Conclusion**

Throughout this post, we’ve explored how to enhance the visibility of your Rust application by incorporating observability principles, with a focus on tracing and metrics.

Tracing is a powerful tool that provides valuable insights into the performance behavior of your application. It allows us to understand where the application is spending its time and resources. Moreover, tracing also reveals useful metadata such as the file, function, and precise line of code where each span is created. This level of detail can be incredibly useful for identifying bottlenecks or problematic areas in your codebase. It is, in a sense, like turning on the lights in a dark room, illuminating the inner workings of your application.

Additionally, we’ve implemented metrics to monitor successful and failed operations, as well as to track performance metrics. These metrics are like the heartbeat of your application, providing a steady stream of information about its health and performance.

A critical aspect of observability is not just collecting this data, but also presenting it in a meaningful way. Using dashboards, we can visualize this data, turning raw numbers into charts and graphs that tell a story about our application. This can be particularly useful in system overviews, making it easier to understand the current state of your application and identify trends over time.

In conclusion, investing in observability is investing in the future maintainability and reliability of your application. It helps developers understand the application’s behavior, improves debugging efficiency, and ultimately leads to a better, more reliable product for your users. As we’ve seen, with Rust and Actix, building an observable application is a straightforward and rewarding process.