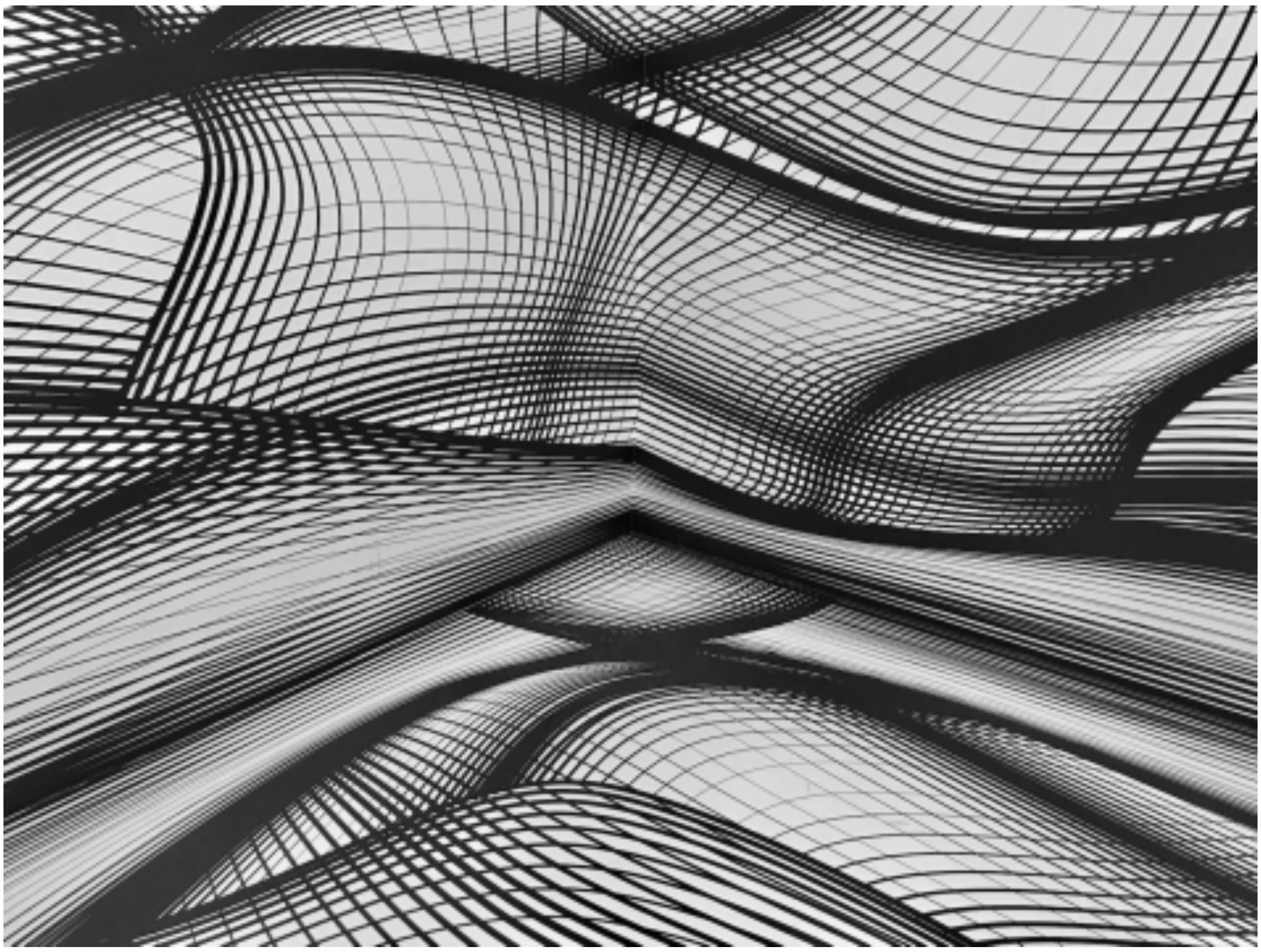
HAMZA.

Software May 1, 2024

HOW TO SETUP AND USE METRICS IN RUST

Leverage Rust metrics for observability and optimization with Grafana/Prometheus.



INTRODUCTION

Metrics provide insights into the system's general performance and specific functionalities. They will also help monitor performance and health.

Effective system monitoring and optimization require detailed metrics. This article will teach you how to use metrics in your Rust application to enhance observability, identify and address performance bottlenecks and security issues, and optimize overall efficiency.

Some standard metrics are **DB Read/Write speed**, **CPU, and RAM usage**.

METRICS TYPES

1. Counter

- **Definition**: A cumulative metric that represents a monotonically increasing value. Usually combined with other functions to give a value per X unit of time (Ex, seconds)
- **Use Case**: Counters help measure an event's total number of occurrences, such as the number of requests processed.

2. Gauge

- **Definition**: A metric representing a single numerical value that can go up or down.
- **Use Case**: Gauges are suitable for measuring fluctuating values, such as the current number of active connections or the available memory.

3. Histogram

- **Definition**: A metric that samples observations and counts them in configurable buckets.
- **Use Case**: Histograms help understand the distribution of values, like response times, allowing you to analyze performance across different percentiles.

TOOLING FOR METRIC VISUALIZATION: GRAFANA AND PROMETHEUS

While collecting metrics is crucial, visualizing and analyzing them is equally important. In the Rust ecosystem, two popular tools, Grafana and Prometheus, stand out for their robust metric visualization capabilities:

1. **Prometheus**: A leading open-source monitoring solution, Prometheus excels at collecting, storing, and querying metric data. With its powerful query language, PromQL, and scalable architecture, Prometheus is well-suited for monitoring modern, dynamic environments.
2. **Grafana**: Grafana complements Prometheus by providing rich visualization and dashboarding capabilities. Developers can create customizable dashboards to visualize metric data in real-time, enabling deep insights into application performance and behaviour.

By integrating Prometheus with **metrics-rs** and visualizing the collected data using Grafana, Rust developers can establish a comprehensive monitoring solution tailored to their specific requirements.

LIBRARIES

OPENTELEMETRY METRICS

This crate is the official implementation of Metrics for OpenTelemetry. It's very verbose. We hav to use opentelemetry::metrics to instrument our app and then opentelemetry_otlp::metrics to export to Prometheus.

The Metrics API consists of these main components:

- MeterProvider is the API entry point. It provides access to Meters.
- Meter is the class responsible for creating Instruments.
- Instrument is accountable for reporting Measurements.

RS-METRICS

In the Rust ecosystem, metrics-rs emerge as a powerful solution for instrumenting and collecting metrics within applications. Developed with simplicity, performance, and flexibility in mind, metrics-rs provides developers with a comprehensive toolkit for effortlessly integrating metrics into their Rust projects.

It has macros that make it very easy to use, and the documentation is very simple and straightforward.

It supports all the metrics we need, has default built-in exporters to Prometheus, and, considering it's widely used in the Rust community, there is a sea of examples and implementations from which to draw inspiration.

In this tutorial, we are going to use the metrics crate, which is easy to use and understand and doesn't require a lot of boilerplate.

GETTING STARTED WITH METRICS-RS

First, we need to add the *metrics* crate to our project. *Quanta* and *Rand Crates* are used to create a demo.

```toml
[dependencies]
metrics = "0.22.3"
metrics-exporter-prometheus = "0.14.0"
metrics-util = "0.16.3"
quanta = "0.12.3"
rand = "0.8.5"
```

cargo.toml hosted with ❤ by **GitHub** **view raw**

We will then create a new module called *our_metrics.rs* that will contain all our setup and configuration configuration. Doing it in a single place makes your code cleaner, and you can quickly know your application's metrics.

```rust
mod our_metrics;

fn main() {
}
```

main_after_module_creation.rs hosted with ❤ by **GitHub** **view raw**

We will create a *Metric struct* in this module with a name and description as properties.

```rust
mod our_metrics;

```

```rust
fn main() {
}
```

main_after_module_creation.rs hosted with ❤ by **GitHub** **view raw**

Using the previously created struct, we instantiate some dummy metrics we will use later in the demo.

```rust
pub const TCP_SERVER_LOOP_DELTA_SECS: Metric = Metric {
    name: "tcp_server_loop_delta_secs",
    description: "",
};

pub const IDLE: Metric = Metric {
    name: "idle_metric",
    description: "",
};

pub const LUCKY_ITERATIONS: Metric = Metric {
    name: "lucky_iterations",
    description: "",
};

pub const TCP_SERVER_LOOPS: Metric = Metric {
    name: "tcp_server_loops",
    description: "",
};
```

dummy_metrics.rs hosted with ❤ by **GitHub** **view raw**

Then, we will have three constants for each Metric type: COUNTERS, GAUGES and HISTOGRAMS, which will be an array of metrics. Think of these as buckets for each metric type.

```rust
pub const COUNTERS: [Metric; 2] = [TCP_SERVER_LOOPS, IDLE];
pub const GAUGES: [Metric; 1] = [LUCKY_ITERATIONS];
pub const HISTOGRAMS: [Metric; 1] = [TCP_SERVER_LOOP_DELTA_SECS];
```

our_metrics_buckets.rs hosted with ❤ by **GitHub** **view raw**

At the end of the file, I like adding utilities that make registering the metrics accessible.

```rust
/// Registers a counter with the given name.
fn register_counter(metric: Metric) {
    metrics::describe_counter!(metric.name, metric.description);
    let _counter = metrics::counter!(metric.name);
}

/// Registers a gauge with the given name.
fn register_gauge(metric: Metric) {
    metrics::describe_gauge!(metric.name, metric.description);
    let _gauge = ::metrics::gauge!(metric.name);
}

/// Registers a histogram with the given name.
fn register_histogram(metric: Metric) {
    metrics::describe_histogram!(metric.name, metric.description);
    let _histogram = ::metrics::histogram!(metric.name);
}
```

utils_metrics.rs hosted with ❤ by **GitHub** **view raw**

We then will have a function called *init_metrics*. This function should ideally be initialized as early as possible in your program. Its job is to initialize the metrics that we want to track.

This function essentially does the following:

1. Initialize the Prometheus builder and configure options like the HTTP listener and the idle time out.
2. We loop through each of the previously created arrays and register those metrics.

Your *our_metrics.rs* should look like the following after the previous steps:

```rust
use std::net::{IpAddr, Ipv4Addr, SocketAddr};
use std::time::Duration;

use metrics_exporter_prometheus::PrometheusBuilder;
use metrics_util::MetricKindMask;


pub struct Metric {
    pub name: &'static str,
    description: &'static str,
}

pub const COUNTERS: [Metric; 2] = [TCP_SERVER_LOOPS, IDLE];
pub const GAUGES: [Metric; 1] = [LUCKY_ITERATIONS];
pub const HISTOGRAMS: [Metric; 1] = [TCP_SERVER_LOOP_DELTA_SECS];

pub const TCP_SERVER_LOOP_DELTA_SECS: Metric = Metric {
    name: "tcp_server_loop_delta_secs",
    description: "The time taken for iterations of the TCP server event loop.",
};

pub const IDLE: Metric = Metric {
    name: "idle_metric",
    description: "",
};

pub const LUCKY_ITERATIONS: Metric = Metric {
    name: "lucky_iterations",
    description: "",
};

pub const TCP_SERVER_LOOPS: Metric = Metric {
    name: "tcp_server_loops",
    description: "The iterations of the TCP server event loop so far.",
};

pub fn init_metrics(port: &u16) {
    println!("initializing metrics exporter");

    PrometheusBuilder::new()
        .idle_timeout(
            MetricKindMask::COUNTER | MetricKindMask::HISTOGRAM,
            Some(Duration::from_secs(10)),
        )
        .with_http_listener(SocketAddr::new(
            IpAddr::V4(Ipv4Addr::new(0, 0, 0, 0)),
            port.to_owned(),
        ))
        .install()
        .expect("failed to install Prometheus recorder");

    for name in COUNTERS {
        register_counter(name)
    }

    for name in GAUGES {
        register_gauge(name)
    }

    for name in HISTOGRAMS {
        register_histogram(name)
    }
}

/******** Utils ********/

/// Registers a counter with the given name.
fn register_counter(metric: Metric) {
    metrics::describe_counter!(metric.name, metric.description);
    let _counter = metrics::counter!(metric.name);
}

/// Registers a gauge with the given name.
fn register_gauge(metric: Metric) {
    metrics::describe_gauge!(metric.name, metric.description);
    let _gauge = ::metrics::gauge!(metric.name);
}

/// Registers a histogram with the given name.
fn register_histogram(metric: Metric) {
    metrics::describe_histogram!(metric.name, metric.description);
    let _histogram = ::metrics::histogram!(metric.name);
}

```

our_metrics.rs hosted with ❤ by **GitHub** **view raw**

Returning to our *main.rs file, we* import the *init_metrics* function in our primary function and call it to initialize the metrics.

To test that our setup works correctly, we will add some demo code that uses the previously created metrics and updates them.

```rust
mod our_metrics;

/// Make sure to run this example with `--features push-gateway` to properly enable push gateway sup
#[allow(unused_imports)]
use std::thread;
use std::time::Duration;

#[allow(unused_imports)]
use metrics::{counter, gauge, histogram};
#[allow(unused_imports)]
use metrics_exporter_prometheus::PrometheusBuilder;

use quanta::Clock;
use rand::{thread_rng, Rng};

use crate::our_metrics::init_metrics;

fn main() {
    init_metrics(&3000);

    let clock = Clock::new();
    let mut last = None;

    counter!(our_metrics::IDLE.name).increment(1);

    // Loop over and over, pretending to do some work.
    loop {
        counter!(our_metrics::TCP_SERVER_LOOPS.name, "system" => "foo").increment(1);

        if let Some(t) = last {
            let delta: Duration = clock.now() - t;
            histogram!(our_metrics::TCP_SERVER_LOOP_DELTA_SECS.name, "system" => "foo").record(delta
        }

        let increment_gauge = thread_rng().gen_bool(0.75);
        let gauge = gauge!(our_metrics::LUCKY_ITERATIONS.name);
        if increment_gauge {
            gauge.increment(1.0);
        } else {
            gauge.decrement(1.0);
        }

        last = Some(clock.now());

        thread::sleep(Duration::from_millis(750));
    }
}
```

main.rs hosted with ❤ by **GitHub** **view raw**

Run *cargo run* and go to *localhost:3000; you should see something like the following*

```
# TYPE idle_metric counter
idle_metric 1

# HELP tcp_server_loops
# TYPE tcp_server_loops counter
tcp_server_loops{system="foo"} 38
tcp_server_loops 0

# HELP lucky_iterations
# TYPE lucky_iterations gauge
lucky_iterations 26

# TYPE testing gauge
testing 42

# HELP tcp_server_loop_delta_secs
# TYPE tcp_server_loop_delta_secs summary
```

```
tcp_server_loop_delta_secs{system="foo",quantile="0"} 0.750220716
tcp_server_loop_delta_secs{system="foo",quantile="0.5"} 0.7549528319395208
tcp_server_loop_delta_secs{system="foo",quantile="0.9"} 0.7551038376064754
tcp_server_loop_delta_secs{system="foo",quantile="0.95"} 0.7551038376064754
tcp_server_loop_delta_secs{system="foo",quantile="0.99"} 0.7551038376064754
tcp_server_loop_delta_secs{system="foo",quantile="0.999"} 0.7551038376064754
tcp_server_loop_delta_secs{system="foo",quantile="1"} 0.755190762
tcp_server_loop_delta_secs_sum{system="foo"} 27.895801044000006
tcp_server_loop_delta_secs_count{system="foo"} 37
tcp_server_loop_delta_secs{quantile="0"} 0
tcp_server_loop_delta_secs{quantile="0.5"} 0
tcp_server_loop_delta_secs{quantile="0.9"} 0
tcp_server_loop_delta_secs{quantile="0.95"} 0
tcp_server_loop_delta_secs{quantile="0.99"} 0
tcp_server_loop_delta_secs{quantile="0.999"} 0
tcp_server_loop_delta_secs{quantile="1"} 0
tcp_server_loop_delta_secs_sum 0
tcp_server_loop_delta_secs_count 0
```

demo.txt hosted with ❤ by **GitHub** **view raw**

CONCLUSION

Tracking these metrics will help to monitor the performance and health of your system and network. Using rs-metrics, we can easily capture and export these metrics to various monitoring tools.

SUBSCRIBE TO THE NEWSLETTER

Building the Future of the Web, One Line of Code at a Time.

Email SUBSCRIBE

READ OTHER ARTICLES

Software January 3, 2023

How to create and use Hooks in Leptos

Creating reusable hooks in Leptos with signals and closures.

Software January 3, 2023

Tracing in Rust: A Comprehensive Guide

Master Rust tracing: synchronous, asynchronous, multithreaded, and distributed scenarios, with practical examples and essential tips.

Software January 3, 2023

Exploring the Latest Features of Javascript 2023 (es2023) in less than 5 minutes

ECMAScript 2023 adds Array methods, Hashbang Grammar, Symbols as WeakMap keys, and non-mutating Array operations.

HAMZA.

Principal Software Engineer at Yahoo!. Tech nerd by heart. Web3 developer by night.

Contact

hello@hamzak.xyz

+44 74 9828 6281