Observable Elixir

Agenda

- What is Observability
- Event Logging
- Metrics
- Traces

And we will be talking alot about `:telemetry`

Bernardo Amorim

Software Engineer @ Slab.com

What is Observability?

According to Wikipedia

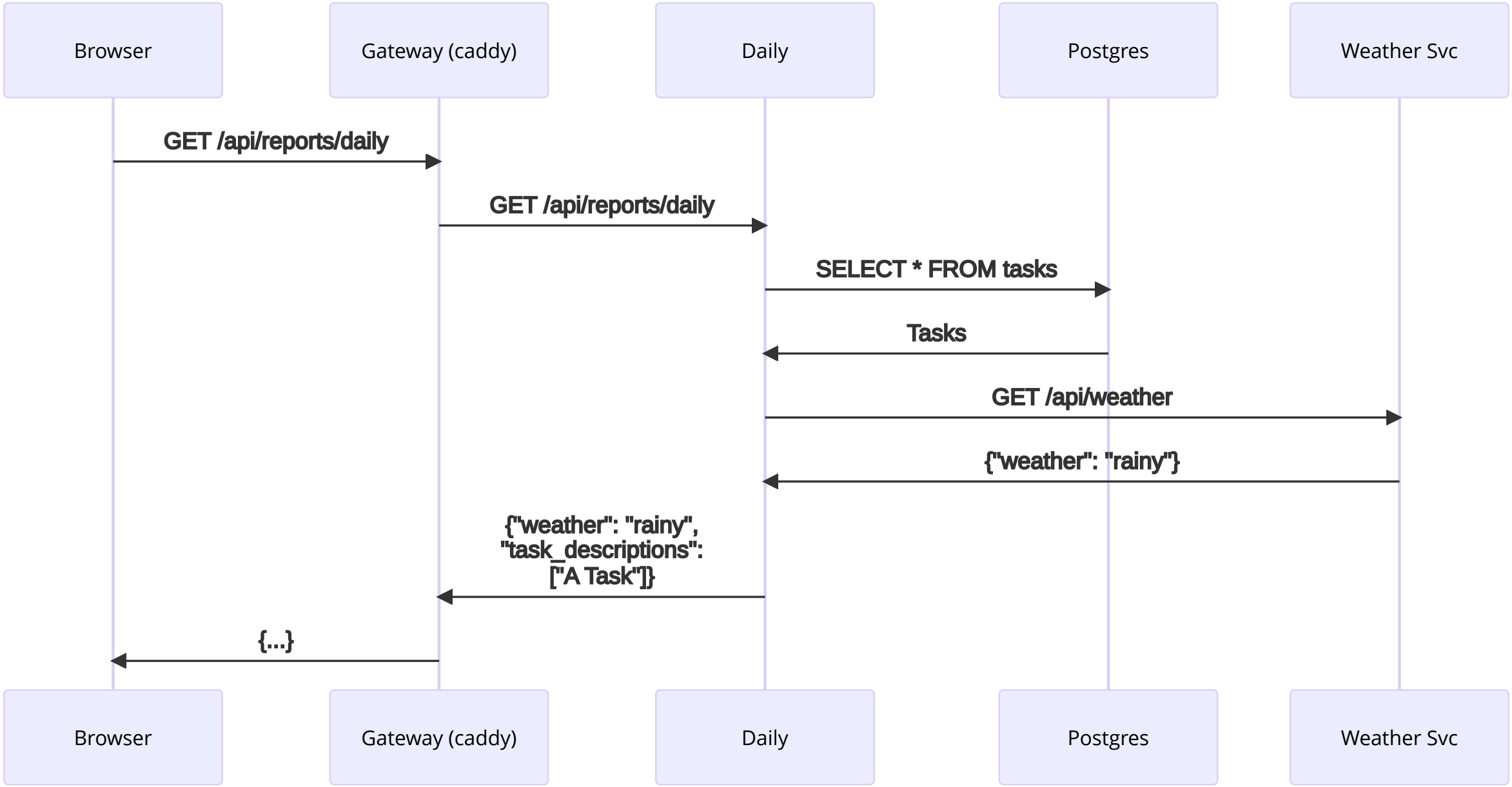
> Observability is a measure of how well internal states of a system can be inferred from knowledge of its external outputs.

We want to

- Understand whether the system is healthy or not
- Debug what went or is going wrong during crisis
- Find areas for improvement

The Demo App for this presentation

github.com/bamorim/observable-elixir-daily


Browser
Gateway (caddy)
Daily
Postgres
Weather Svc
GET /api/reports/daily
GET /api/reports/daily
SELECT * FROM tasks
Tasks
GET /api/weather
{"weather": "rainy"}
{"weather": "rainy",
"task_descriptions":
["A Task"]}
{...}
Browser
Gateway (caddy)
Daily
Postgres
Weather Svc

<video src="./videos/demo-observable-elixir.webm" controls width="100%"></video>

Your observability stack

Operational dashboards for your data here, there, or anywhere

✨🧰☁️✨

The (actually useful) free forever plan

Grafana, of course + 10K series Prometheus metrics + 50GB logs + 50GB traces

Create free account

(No credit card required)

MILLIONS OF USERS ACROSS 800K+ GLOBAL INSTANCES

ROBLOX SIEMENS salesforce PayPal ebay Pernod Ricard verizon

Success stories →

Three Pillars of Observability

- Event Logs
- Metrics
- Traces

Event Logs

- Things happen in your system
- Logs are a way for externalization
- Examples:
  - HTTP Request Handled
  - Database Query Executed
  - Background Job Executed

Metadata

Common data

- Duration
- Response Status Code
- Path / Route

Similar to `:telemetry` events

Telemetry

Documentation

Telemetry is a lightweight library for dynamic dispatching of events, with a focus on metrics and instrumentation. Any Erlang or Elixir library can use `telemetry` to emit events. Application code and other libraries can then hook into those events and run custom handlers.

> *Note: this library is agnostic to tooling and therefore is not directly related to OpenTelemetry. For OpenTelemetry in the Erlang VM, see opentelemetry-erlang, and check opentelemetry_telemetry to connect both libraries.*

Usage

In a nutshell, you register a custom module and function to be invoked for certain events, which are executed whenever there is such an event. The event name is a list of atoms. Each event is composed of a numeric value and can have metadata attached to it. Let's look at an example.

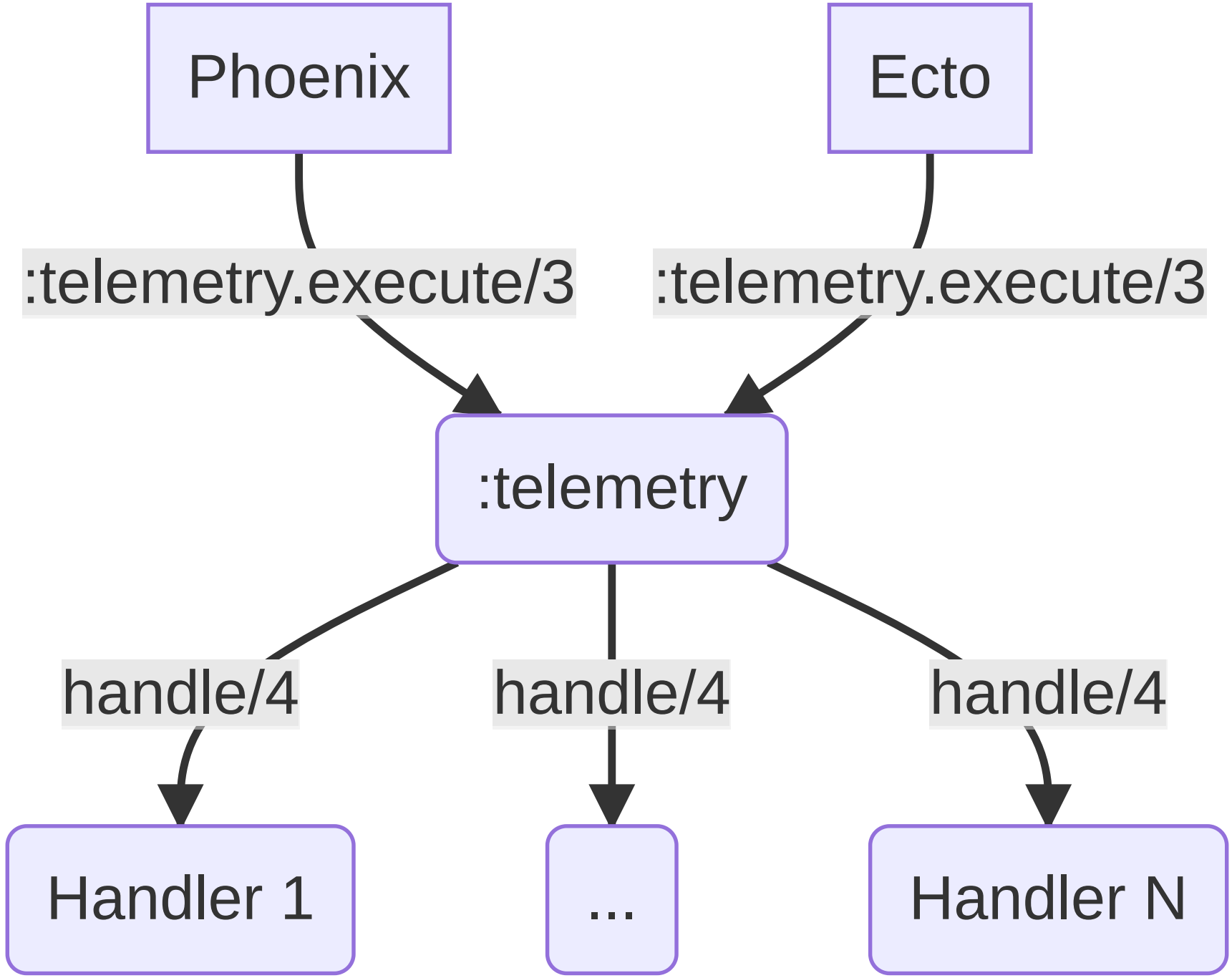
Imagine that you have a web application and you'd like to log latency and response status for each incoming request. With Telemetry, you can build a module which does exactly that whenever a response is sent. The first step is to execute a measurement.

In Elixir:

`:telemetry` events

```
@type event() :: {event_name(), metadata(), measurements()}

@type event_name() :: [atom(), ...]
@type metadata() :: map()
@type measurements() :: map()
```


Phoenix
Ecto
:telemetry.execute/3
:telemetry.execute/3
:telemetry
handle/4
handle/4
handle/4
Handler 1
...
Handler N

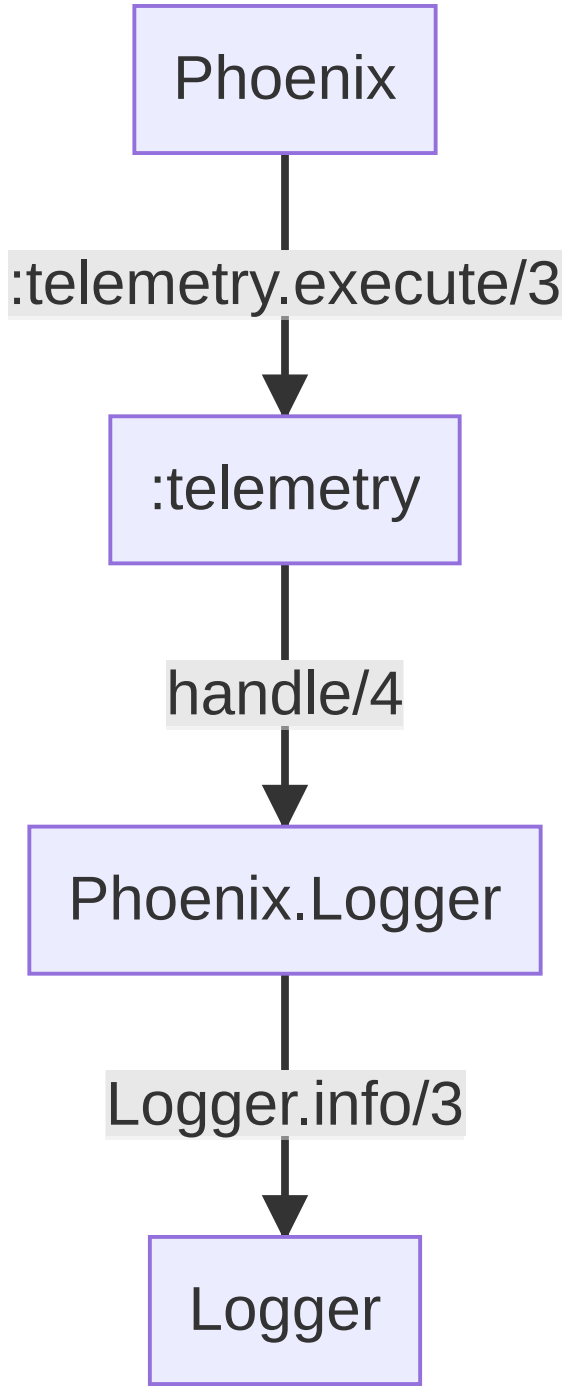
`:telemetry` spans

- Has a start and an end
- `:telemetry` defines a standard convention with 3 events:
  - `:start` with `:system_time` measurement
  - `:stop` and `:exception` with `:duration` measurement

`:telemetry` span example

- `[:phoenix, :endpoint, :start]` when the request starts
- `[:phoenix, :endpoint, :stop]` when the requests finishes successfully
- `[:phoenix, :endpoint, :exception]` when an exception happens

Logs are just `:telemetry` events that are externalized

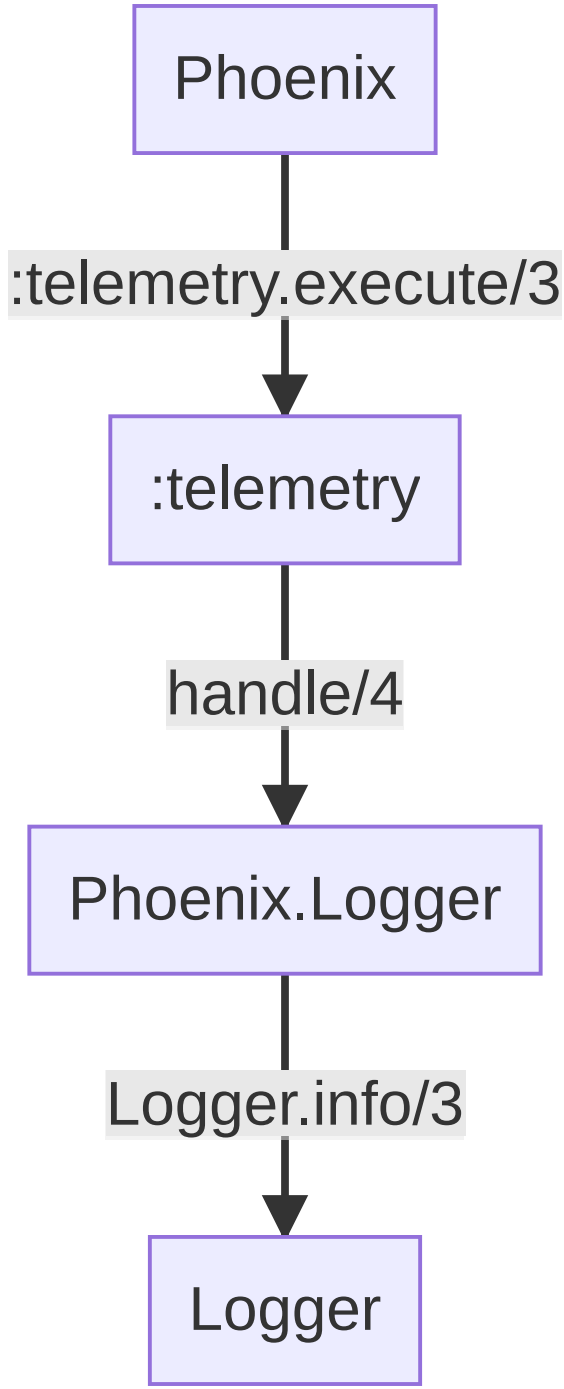


My problem with Phoenix default logging

```
[info] GET /api/tasks/
[info] Sent 200 in 23ms
[info] GET /api/tasks/
[info] Sent 200 in 2ms
[info] GET /api/tasks/
[info] Sent 200 in 2ms
```

We can fix it

- Disable default Phoenix Logger
- Implement custom logger by listening to telemetry events
- Implement custom log formatter using a structured format (Logfmt or JSON)



Implement in Elixir with Libraries

Many libraries for JSON Logging

- LoggerJSON
- Ink

Implement in Elixir with Libraries

I'm open sourcing some internal libraries for logging we were using:

- github.com/bamorim/telemetry_logger
- github.com/bamorim/structured_logger

Install the libraries

```
defp deps do
  [
    # Add the following deps
    {:telemetry_logger, github: "bamorim/telemetry_logger"},
    {:structured_logger, github: "bamorim/structured_logger"}
  ]
end
```

Then run `mix deps.get`

Switch to the new logger and formatter

```
# Disable Phoenix Logger
config :phoenix, logger: false

# Set the formatter and allow all metadata
config :logger, :console,
  format: {StructuredLogger, :format},
  metadata: :all
```

```
# Add to your MyApp.Application.start/2
TelemetryLogger.attach_loggers([
  {TelemetryLogger.PlugLogger, router: MyAppWeb.Router}
])
```

Explore | loki | Split | Add to dashboard | Last 1 hour | Run query

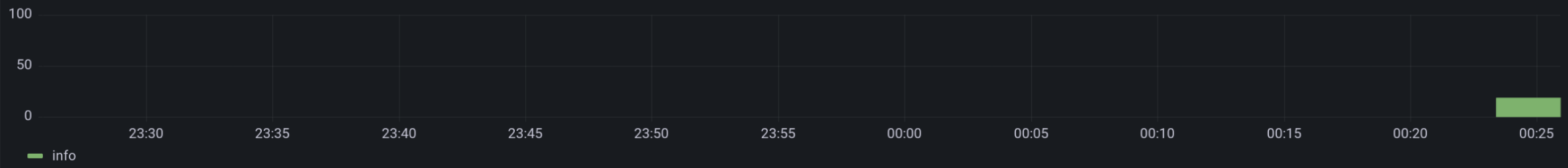
A (loki)

Log browser > {compose_service="todos"} | logfmt | path="/api/tasks"

Query type: Range | Instant | Line limit: auto | Resolution: 1/1

+ Add query | Query history | Inspector

Log volume



Logs

Time | Unique labels | Wrap lines | Prettify JSON | Dedup: None | Exact | Numbers | Signature | Display results: Newest first | Oldest first

Common labels: daily todos daily-todos-1 docker-desktop GET GET /api/tasks -> 200 /api/tasks /api/tasks Elixir.DailyWeb.TaskController index stdout 200 Line limit: 1000 (19 returned)

Total bytes processed: 13.1 kB

> level=info msg="GET /api/tasks -> 200" duration_us=6475 method=GET path=/api/tasks request_id=FvWOfyw0Ah6jffcAAAKC route=/api/tasks route_plug=Elixir.DailyWeb.TaskController route_plug_opts=index status=200 time=1654385114988774

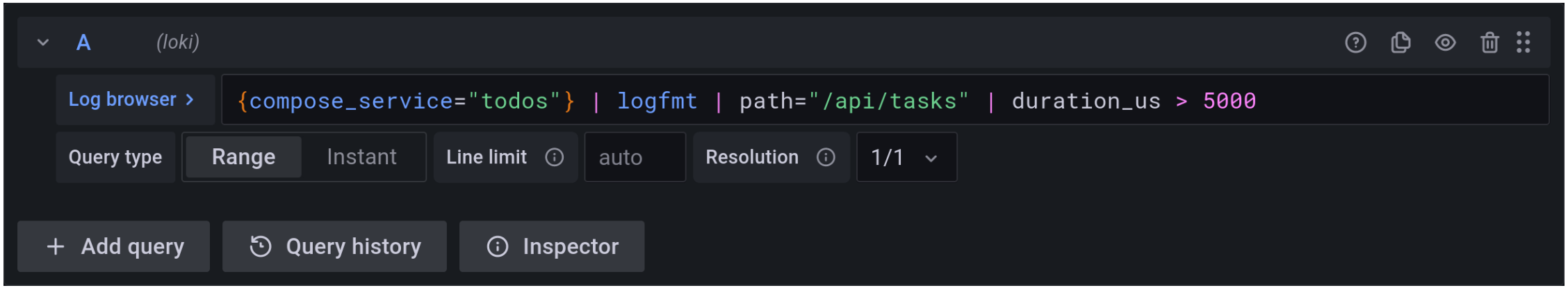
> level=info msg="GET /api/tasks -> 200" duration_us=1030 method=GET path=/api/tasks request_id=FvWOfxkRi7-DIKUAAAJi route=/api/tasks route_plug=Elixir.DailyWeb.TaskController route_plug_opts=index status=200 time=1654385114662229

> level=info msg="GET /api/tasks -> 200" duration_us=2628 method=GET path=/api/tasks request_id=FvWOfwKJFOmlK50AAAJC route=/api/tasks route_plug=Elixir.DailyWeb.TaskController route_plug_opts=index status=200 time=1654385114285703

Start of range

00:25:55

Grafana, Loki and LogQL are Awesome



Logs Tips

- **Do** use structured logging
- **Don't do** "print-debugging"
- **Do** take advantage of log levels
- **Do** allow your system to change log level without redeploying
- **Don't** nest fields in your logs

Logs help with debugging, but

How to check if the system is healthy?

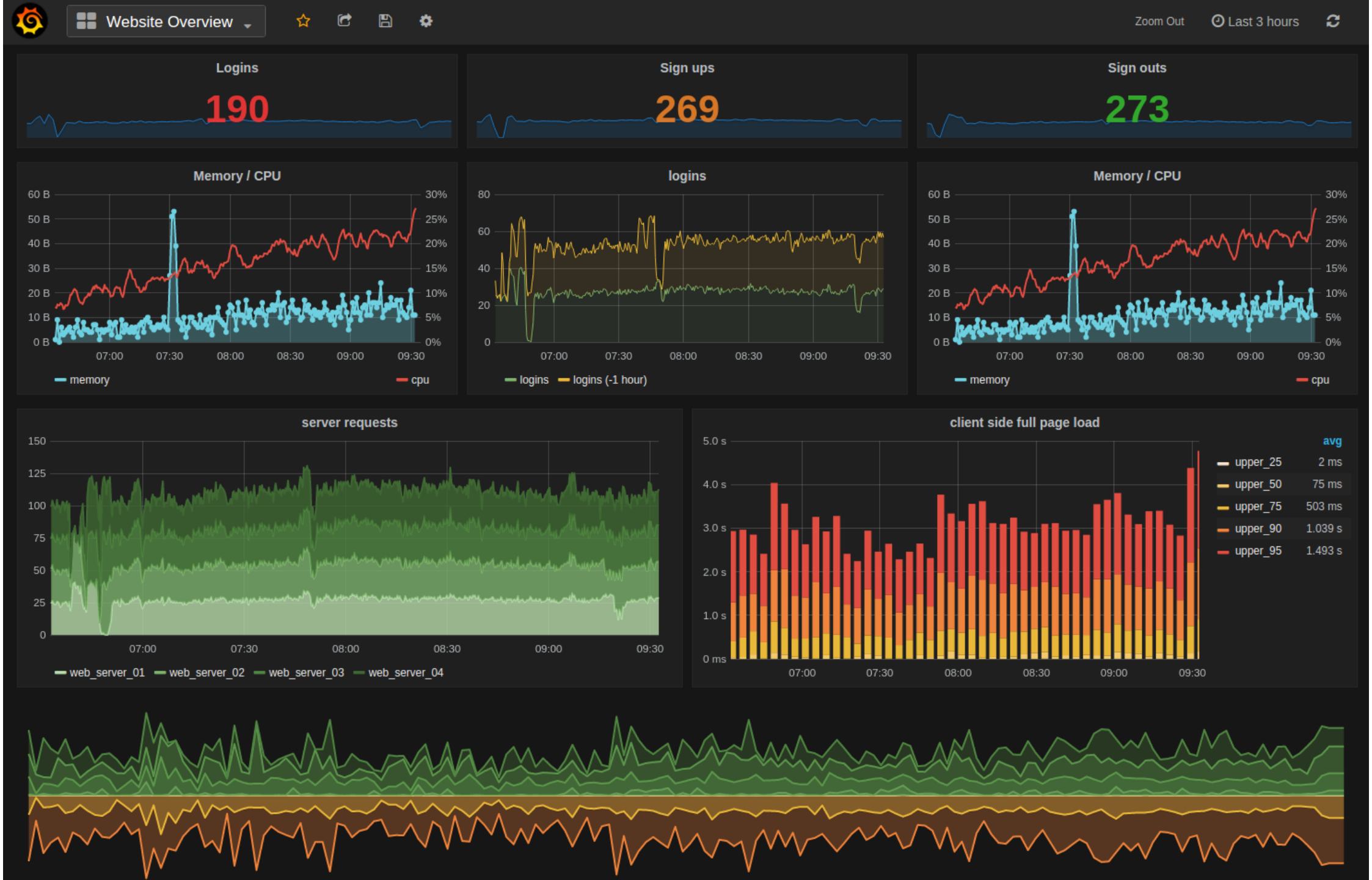
- Requests/second
- Average (and other percentiles) latency
- Memory and CPU usage

Metrics

Numerical values sampled over time

Metrics give you a high-level view of your system

- Useful both on a technical level (e.g. memory usage) or domain level (e.g. total count of payments processed)
- Great for visualizations


Website Overview
Zoom Out
Last 3 hours
Logins
190
Sign ups
269
Sign outs
273
Memory / CPU
60 B
50 B
40 B
30 B
20 B
10 B
0 B
30%
25%
20%
15%
10%
5%
0%
07:00
07:30
08:00
08:30
09:00
09:30
memory
cpu
logins
80
60
40
20
0
logins (-1 hour)
Memory / CPU
memory
cpu
server requests
150
125
100
75
50
25
0
web_server_01
web_server_02
web_server_03
web_server_04
client side full page load
5.0 s
4.0 s
3.0 s
2.0 s
1.0 s
0 ms
avg
upper_25 2 ms
upper_50 75 ms
upper_75 503 ms
upper_90 1.039 s
upper_95 1.493 s

Metrics are cheap and fast to process

- Complexity is only dependent on number of timeseries and sample frequency
- Great for alerting
- Great for long term storage

Metrics - Data Model

```
@type timeseries() :: {metric_id(), [sample()]}

@type metric_id() :: {metric_name(), metric_labels()}
@type sample() :: {sample_value(), timestamp()}

@type metric_name() :: String.t()
@type metric_labels() :: %{String.t() => String.t()}
@type sample_value() :: float()
@type sample_timestamp() :: integer()
```

Computing metrics from `:telemetry` events

- `Plug.Telemetry` emits `[:phoenix, :endpoint, :stop]` events
- We can count the number of events emitted and aggregate into the "total number of requests"

Telemetry.Metrics

Common interface for defining metrics based on `:telemetry` events.

Metrics are aggregations of Telemetry events with specific name, providing a view of the system's behaviour over time.

To give a more concrete example, imagine that somewhere in your code there is a function which sends an HTTP request, measures the time it took to get a response, and emits an event with the information:

```
:telemetry.execute([:http, :request, :stop], %{duration: duration})
```

You could define a counter metric, which counts how many HTTP requests were completed:

```
Telemetry.Metrics.counter("http.request.stop.duration")
```

or you could use a summary metric to see statistics about the request duration:

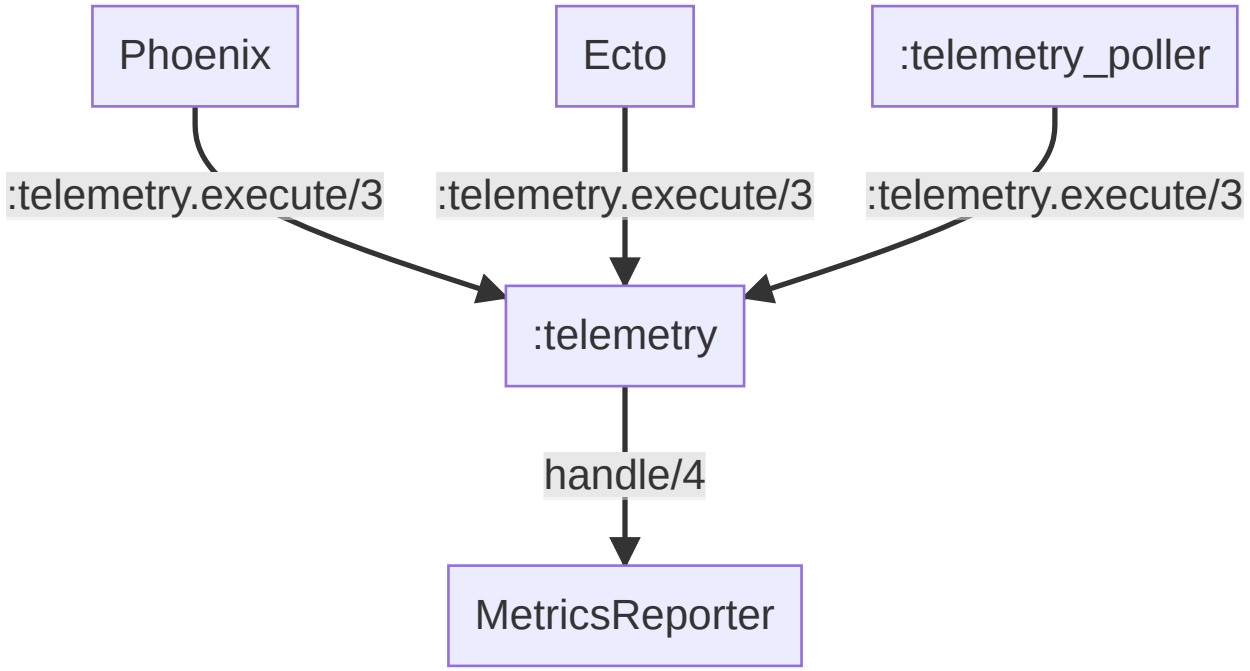
```
Telemetry.Metrics.summary("http.request.stop.duration")
```

This documentation is going to cover all the available metrics and how to use them, as well as options, and how to integrate those metrics with reporters.

Metrics

`Telemetry.Metrics`

- Language for defining `:telemetry` based metrics
- Define 5 different metric types (counter, distribution, last value, sum and summary)
- Metric Reporters attach to events and aggregate them



Updates automatically

Phoenix LiveDashboard

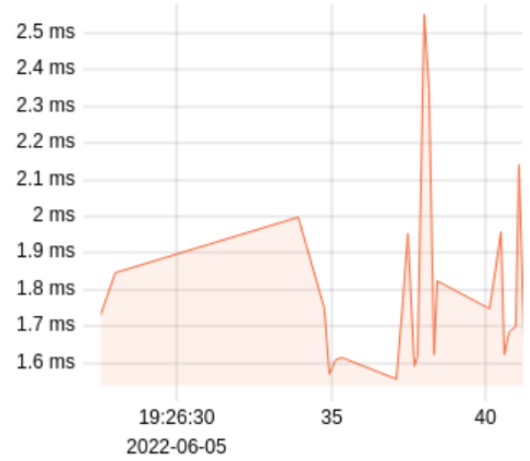
Selected node nonode@nohost

Home | OS Data | Metrics | Request Logger | Applications | Processes | Ports | Sockets | ETS | Ecto Stats

Enable

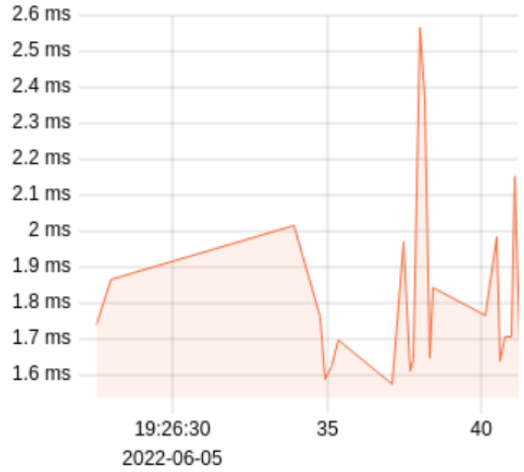
Daily | Phoenix | VM

phoenix.endpoint.stop.duration



	Value	Min	Max	Avg
Duration	--	--	--	--

phoenix.router_dispatch.stop.duration (route)



	Value	Min	Max	Avg
/api/tasks	--	--	--	--

```elixir
defmodule DailyWeb.Telemetry do
  use Supervisor
  import Telemetry.Metrics

  # ...

  def metrics do
    [
      # Phoenix Metrics
      summary("phoenix.endpoint.stop.duration",
        unit: {:native, :millisecond}
      ),
      summary("phoenix.router_dispatch.stop.duration",
        tags: [:route],
        unit: {:native, :millisecond}
      ),
      # ...
    ]
  end
end
```

Limitations of LiveDashboard

- Metrics are not persisted
- If you have multiple apps it will be hard to consolidate visualizations and data
- Only works for Elixir

From metrics to insight

Power your metrics and alerting with the leading open-source monitoring solution.

GET STARTED DOWNLOAD

The Prometheus Team strongly condemns Russia's illegal invasion of Ukraine. Please consider donating to a humanitarian aid organization such as Aktion Deutschland Hilft or Care in Action to provide relief.

Dimensional data

Prometheus implements a highly dimensional data model. Time series are identified by a metric name and a set of key-value pairs.

Powerful queries

PromQL allows slicing and dicing of collected time series data in order to generate ad-hoc graphs, tables, and alerts.

Great visualization

Prometheus has multiple modes for visualizing data: a built-in expression browser, Grafana integration, and a console template language.

Efficient storage

Prometheus stores time series in memory and on local disk in an efficient custom format. Scaling is achieved by functional sharding and federation.

Simple operation

Each server is independent for reliability, relying only on local storage. Written in Go, all binaries are statically linked and easy to deploy.

Precise alerting

Alerts are defined based on Prometheus's flexible PromQL and maintain dimensional information. An alertmanager handles notifications and silencing.

Many client libraries

Client libraries allow easy instrumentation of services. Over ten languages are supported already and custom libraries are easy to implement.

Many integrations

Existing exporters allow bridging of third-party data into Prometheus. Examples: system statistics, as well as Docker, HAProxy, StatsD, and JMX metrics.

«Even though Borgmon remains internal to Google, the idea of treating time-series data as a data source for generating alerts is now accessible to everyone through those open source tools like Prometheus [...]»

— **Site Reliability Engineering:** How Google Runs Production Systems (O'Reilly Media)

Prometheus

- Open-source monitoring and alerting system
- A multi-dimensional data model for time-series data
- PromQL: query language
- Data collection via **pull** over simple HTTP protocol

Pull vs Push

- Prometheus is Pull, that is, Prometheus controls when to ask for metrics
  - Improves back-pressure (if Prometheus is overloaded it can delay the sampling)
- Your app just need to:
  - Keep last values for metrics
  - Be able to report them when Prometheus request (in a specific format)

Prometheus Exposition Format

```
my_metric{label1=value1} 101
my_metric{label1=value2} 42

other_metric{label=value} 3.14
```

Integrating Prometheus with Elixir

TelemetryMetricsPrometheus

Prometheus Reporter for `Telemetry.Metrics` definitions.

Provide a list of metric definitions to the `init/2` function. It's recommended to run TelemetryMetricsPrometheus under a supervision tree, usually under Application.

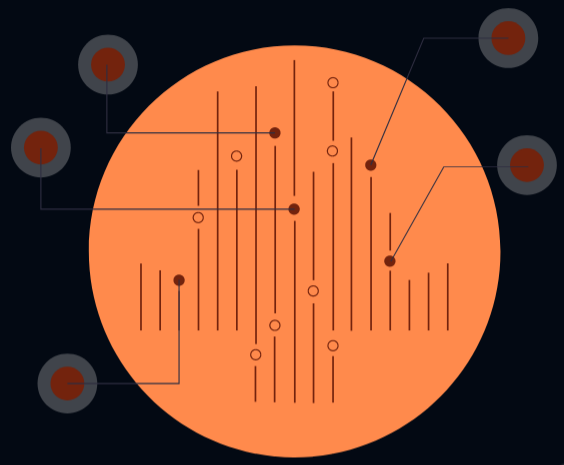
```
def start(_type, _args) do
  # List all child processes to be supervised
  children = [
    {TelemetryMetricsPrometheus, [metrics: metrics()]}
  ...
  ]

  opts = [strategy: :one_for_one, name: ExampleApp.Supervisor]
  Supervisor.start_link(children, opts)
end

defp metrics, do:
  [
    counter("http.request.count"),
    sum("http.request.payload_size", unit: :byte),
    last_value("vm.memory.total", unit: :byte)
  ]
```

By default, metrics are exposed on port `9568` at `/metrics`. The port number can be configured if necessary. You are not required to use the included server, though it is recommended. `https` is not supported yet, in

README



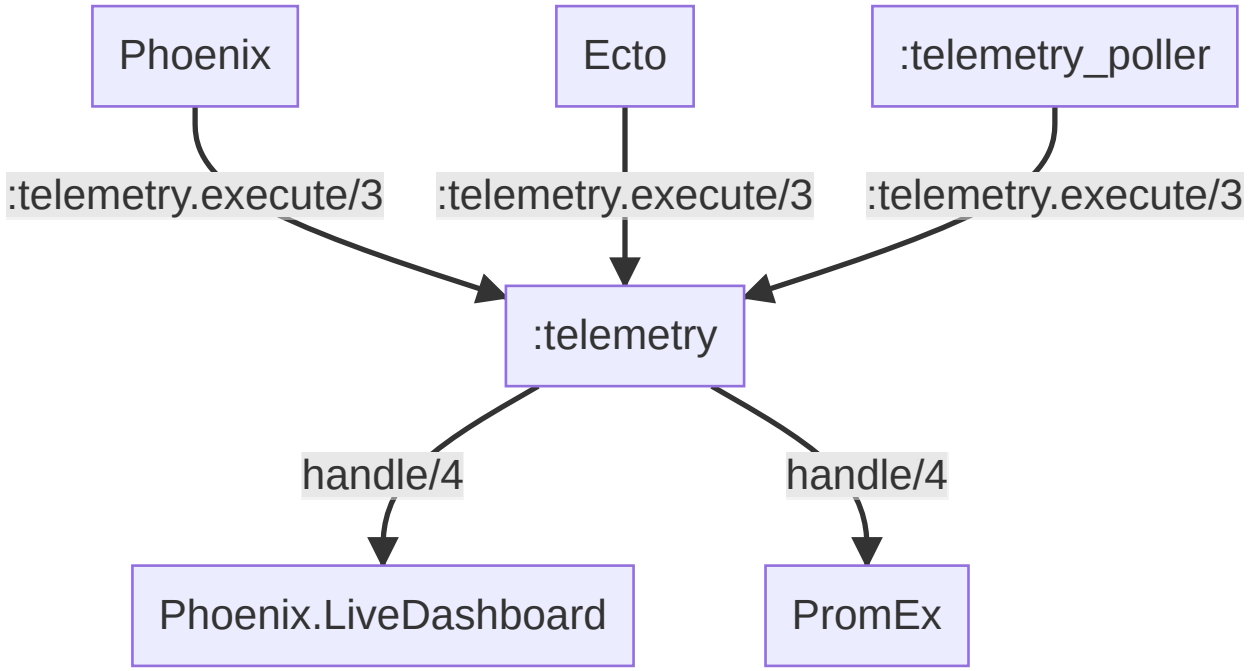
PromEx

Prometheus metrics and Grafana dashboards for all of your favorite Elixir libraries

Contents

`PromEx`

- Just another handler
- Shared core with `TelemetryMetricsPrometheus`
- Nice library of ready-made metrics and Grafana dashboards
- For something more minimalist, `TelemetryMetricsPrometheus` is probably your best bet.



Install PromEx

Add the dependency

```
  defp deps do
    [
      {:prom_ex, "~> 1.7.1"}
    ]
  end
```

Run `mix prom_ex.gen.config --datasource prometheus`

You should end up with something like this:

```elixir
defmodule Daily.PromEx do
  use PromEx, otp_app: :daily

  @impl true
  def plugins do
    #...
  end

  @impl true
  def dashboard_assigns do
    # ...
  end

  @impl true
  def dashboards do
    # ...
  end
end
```

If you follow the instructions on the generated file you can then enable the relevant plugins, for example:

```
def plugins do
  [
    Plugins.Application,
    Plugins.Beam,
    {Plugins.Phoenix, router: DailyWeb.Router, endpoint: DailyWeb.Endpoint},
    Plugins.Ecto
  ]
end
```

The instructions will also tell you to set some configs and

```
defmodule DailyWeb.Endpoint do
  use Phoenix.Endpoint, otp_app: :daily

  # Add this line
  plug PromEx.Plug, prom_ex_module: Daily.PromEx
end
```

```
defmodule Daily.Application do
  def start(_type, _args) do
    children = [
      # Add PromEx to the supervision tree
      Daily.PromEx,
      # ...
    ]
    # ...
  end
end
```

Now you can get metrics in prometheus format at `http://localhost:4000/metrics`

```
# HELP daily_prom_ex_phoenix_endpoint_port_info The configured port of the Endpoint module.
# TYPE daily_prom_ex_phoenix_endpoint_port_info gauge
daily_prom_ex_phoenix_endpoint_port_info{endpoint="DailyWeb.Endpoint",port="4000"} 1
# HELP daily_prom_ex_phoenix_endpoint_url_info The configured URL of the Endpoint module.
# TYPE daily_prom_ex_phoenix_endpoint_url_info gauge
daily_prom_ex_phoenix_endpoint_url_info{endpoint="DailyWeb.Endpoint",url="http://localhost:4000"} 1
# HELP daily_prom_ex_beam_system_schedulers_online_info The number of scheduler threads that are online.
# TYPE daily_prom_ex_beam_system_schedulers_online_info gauge
daily_prom_ex_beam_system_schedulers_online_info 6
# HELP daily_prom_ex_beam_system_schedulers_info The number of scheduler threads in use by the BEAM.
# TYPE daily_prom_ex_beam_system_schedulers_info gauge
daily_prom_ex_beam_system_schedulers_info 6
# HELP daily_prom_ex_beam_system_dirty_io_schedulers_info The total number of dirty I/O schedulers used to execute I/O bound native functions.
# TYPE daily_prom_ex_beam_system_dirty_io_schedulers_info gauge
daily_prom_ex_beam_system_dirty_io_schedulers_info 10
# HELP daily_prom_ex_beam_system_dirty_cpu_schedulers_online_info The total number of dirty CPU schedulers that are online.
# TYPE daily_prom_ex_beam_system_dirty_cpu_schedulers_online_info gauge
daily_prom_ex_beam_system_dirty_cpu_schedulers_online_info 6
# HELP daily_prom_ex_beam_system_dirty_cpu_schedulers_info The total number of dirty CPU scheduler threads used by the BEAM.
# TYPE daily_prom_ex_beam_system_dirty_cpu_schedulers_info gauge
daily_prom_ex_beam_system_dirty_cpu_schedulers_info 6
# HELP daily_prom_ex_beam_system_word_size_bytes_info The size of Erlang term words in bytes.
# TYPE daily_prom_ex_beam_system_word_size_bytes_info gauge
daily_prom_ex_beam_system_word_size_bytes_info 8
# HELP daily_prom_ex_beam_system_time_correction_support_info Whether the BEAM instance has time correction support.
# TYPE daily_prom_ex_beam_system_time_correction_support_info gauge
daily_prom_ex_beam_system_time_correction_support_info 1
# HELP daily_prom_ex_beam_system_thread_support_info Whether the BEAM instance has been compiled with threading support.
# TYPE daily_prom_ex_beam_system_thread_support_info gauge
daily_prom_ex_beam_system_thread_support_info 1
# HELP daily_prom_ex_beam_system_jit_support_info Whether the BEAM instance is running with the JIT compiler.
# TYPE daily_prom_ex_beam_system_jit_support_info gauge
daily_prom_ex_beam_system_jit_support_info 1
# HELP daily_prom_ex_beam_system_smp_support_info Whether the BEAM instance has been compiled with SMP support.
# TYPE daily_prom_ex_beam_system_smp_support_info gauge
daily_prom_ex_beam_system_smp_support_info 1
# HELP daily_prom_ex_beam_system_version_info The OTP release major version.
# TYPE daily_prom_ex_beam_system_version_info gauge
daily_prom_ex_beam_system_version_info 25
# HELP daily_prom_ex_beam_system_atom_limit_info The maximum number of atoms allowed.
```

Instructing Prometheus to scrape metrics

This will vary depending on your setup, but it should be something as easy as configuring this:

```
scrape_configs:
  - job_name: 'daily'
    scrape_interval: 5s
    static_configs:
      - targets: ['daily:4000']
```

And now you can query through Grafana (or other frontends)

Explore | prometheus | Split | Add to dashboard | Last 1 hour | Run query

A (prometheus)

Run query | Explain | Builder Beta | Code

Metrics browser > daily_prom_ex_phoenix_http_requests_total{}

Options Legend: Verbose Format: Time series Type: Both

+ Add query | Query history | Inspector

Graph

Lines | Bars | Points | Stacked lines | Stacked bars

daily_prom_ex_phoenix_http_requests_total{action="Unknown", controller="Unknown", instance="daily:4000", job="daily", method="GET", path="Unknown", status="404"}

daily_prom_ex_phoenix_http_requests_total{action="daily", controller="DailyWeb.ReportController", instance="daily:4000", job="daily", method="GET", path="/api/reports/daily", status="200"}

daily_prom_ex_phoenix_http_requests_total{action="index", controller="DailyWeb.TaskController", instance="daily:4000", job="daily", method="GET", path="/api/tasks", status="200"}

And let's say you have something like

```
def dashboards
  [
    {:prom_ex, "application.json"},
    {:prom_ex, "beam.json"},
    {:prom_ex, "phoenix.json"},
    {:prom_ex, "ecto.json"}
  ]
end
```

And you run something like

```
for dashboard in application beam phoenix ecto; do
  mix prom_ex.dashboard.export -m 'Daily.PromEx' -d "$dashboard.json" -s > ./docker/grafana/$dashboard.json
done
```

Each will generate a JSON file you import into Grafana.

Prometheus Job: daily | Application Instance: daily:4000 | Ecto Repo: Daily.Repo | Interval: 30s

Sponsor PromEx | Ecto Plugin Docs

Overview

Connection Idle Time (Average)

590 ms

Connection Queue Time (Average)

0.0613 ms

Connection Decode Time (Average)

0.0117 ms

Connection Execution Time (Average)

0.530 ms

Ecto Repo

Daily.Repo

Database Host

postgres

Database Name

daily_prod

Pool Size

10

Timeout Duration

15 s

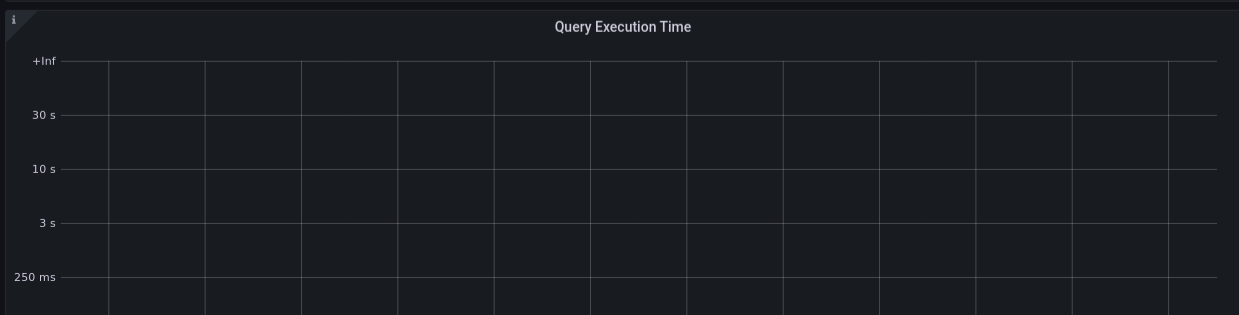
Query Details

Average Total Execution Time

Total Execution Time

Average Query Execution Time

Query Execution Time



Traces

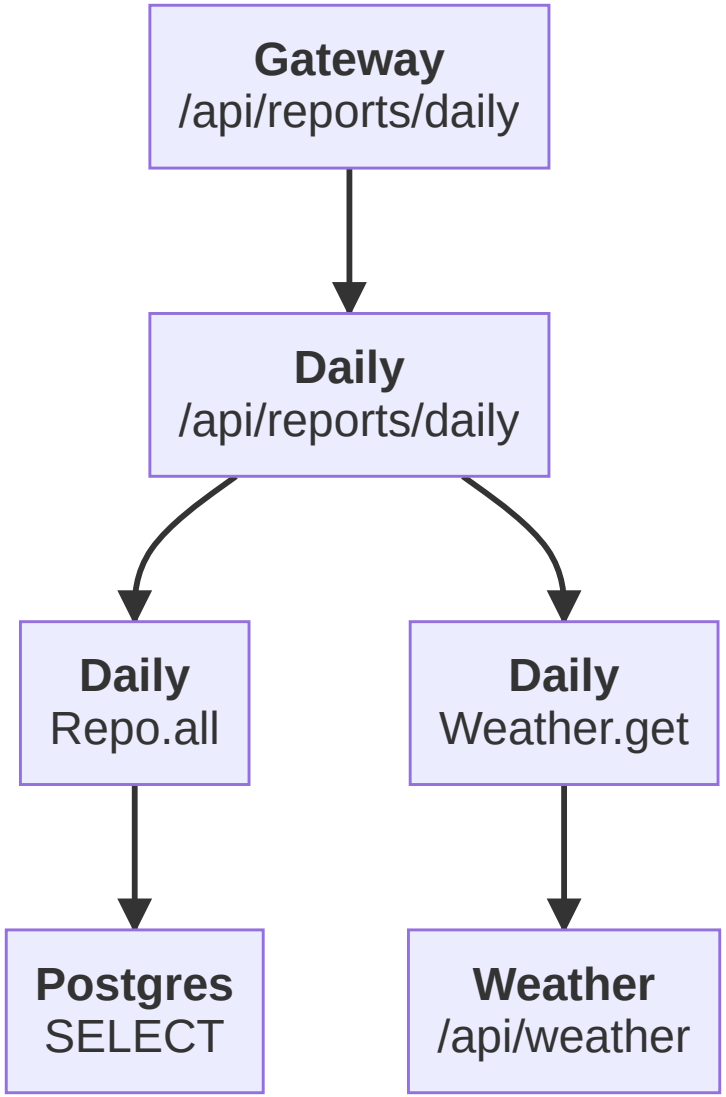
A trace is a collection of **correlated events** that captures information about a program execution.

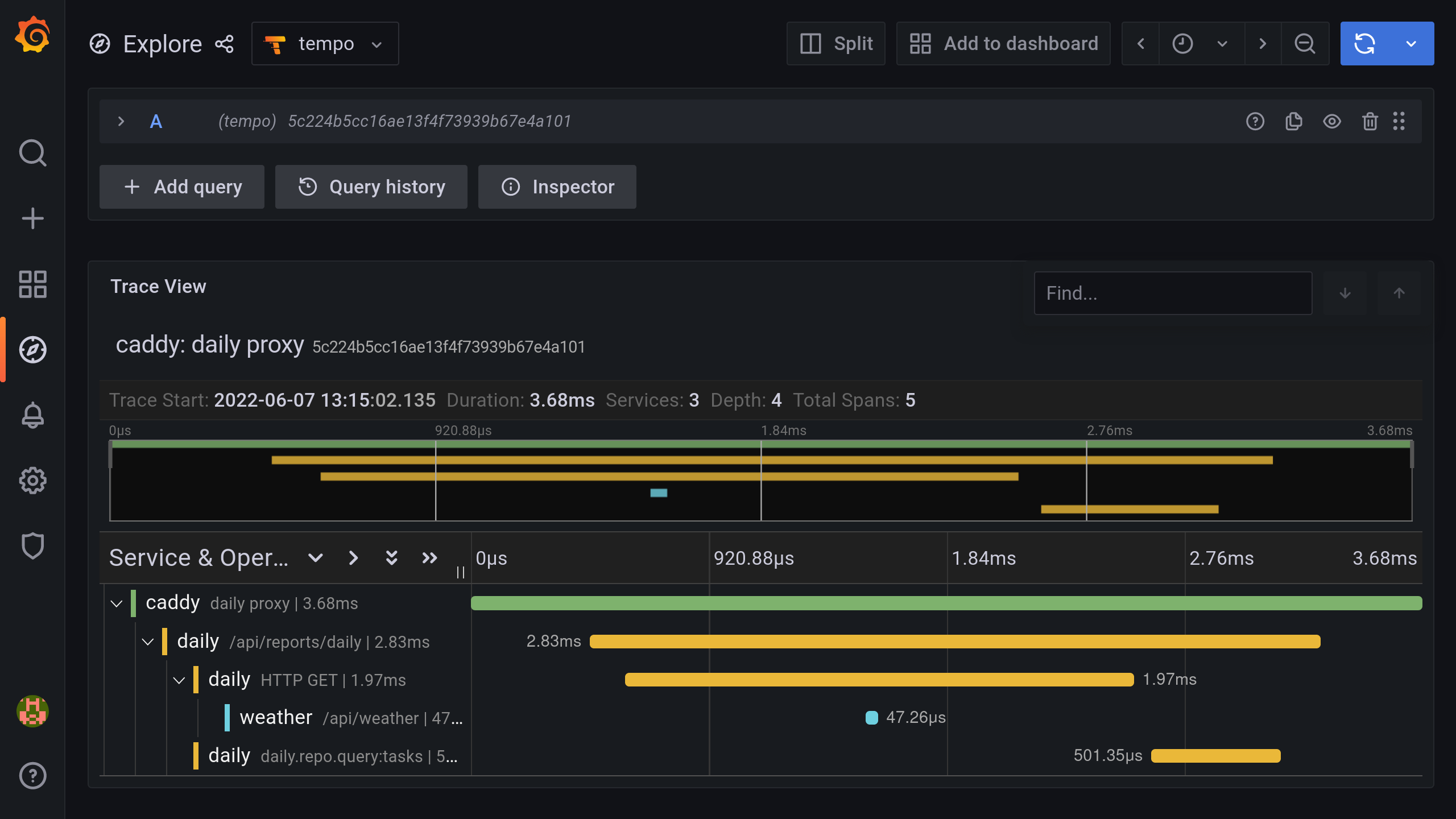
Distributed Tracing

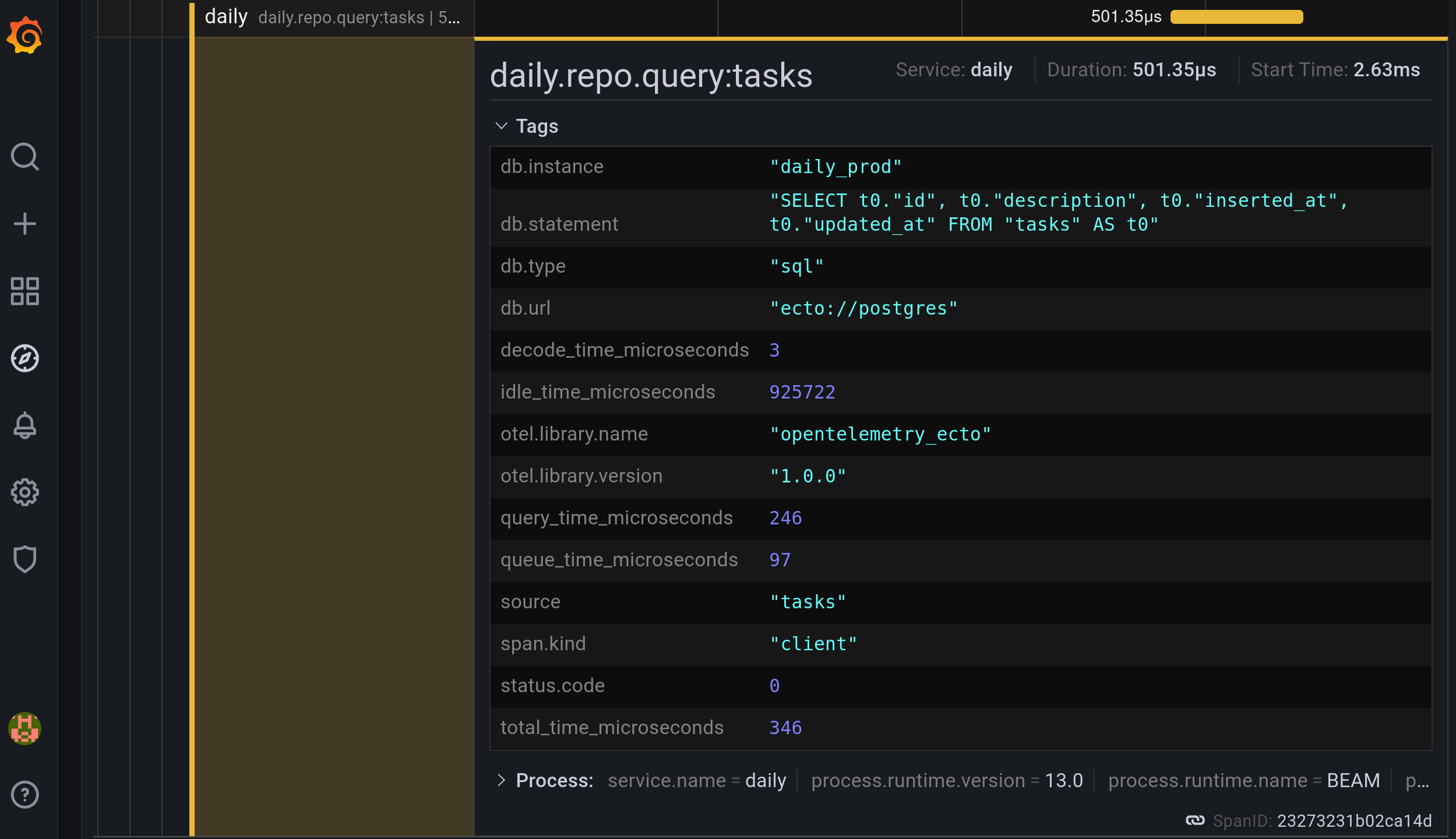
A trace where spans are executed in multiple different services.

Trace Model

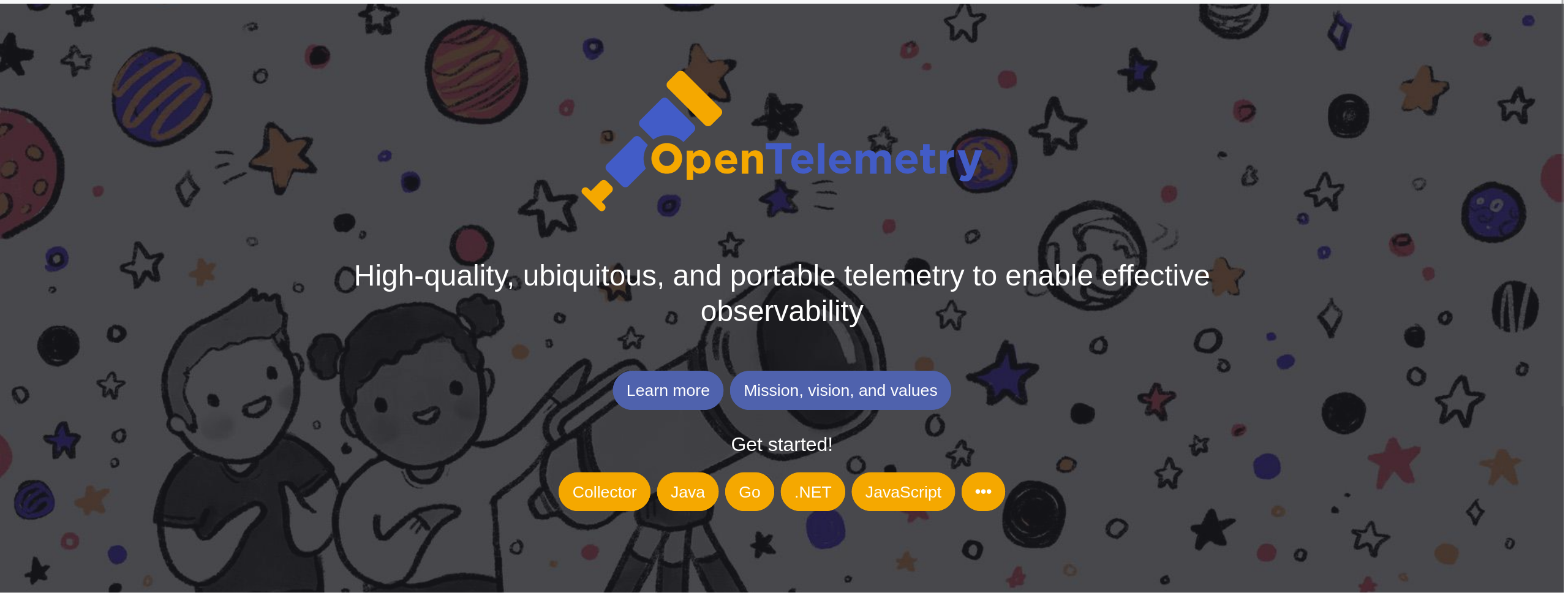
- A **trace** is a tree of **spans**
- A **span**:
  - Has a start and end timestamps
  - Contained to one service
  - Contains some metadata
  - Belogns to a trace
  - Can be either root or child of another span on the same trace




Explore
tempo
Split
Add to dashboard
A (tempo) 5c224b5cc16ae13f4f73939b67e4a101
Add query
Query history
Inspector
Trace View
Find...
caddy: daily proxy 5c224b5cc16ae13f4f73939b67e4a101
Trace Start: 2022-06-07 13:15:02.135 Duration: 3.68ms Services: 3 Depth: 4 Total Spans: 5
0µs
920.88µs
1.84ms
2.76ms
3.68ms
Service & Oper...
caddy daily proxy | 3.68ms
daily /api/reports/daily | 2.83ms
2.83ms
daily HTTP GET | 1.97ms
1.97ms
weather /api/weather | 47...
47.26µs
daily daily.repo.query:tasks | 5...
501.35µs


daily daily.repo.query:tasks | 5...
501.35µs
daily.repo.query:tasks
Service: daily
Duration: 501.35µs
Start Time: 2.63ms
Tags
db.instance "daily_prod"
db.statement "SELECT t0."id", t0."description", t0."inserted_at", t0."updated_at" FROM "tasks" AS t0"
db.type "sql"
db.url "ecto://postgres"
decode_time_microseconds 3
idle_time_microseconds 925722
otel.library.name "opentelemetry_ecto"
otel.library.version "1.0.0"
query_time_microseconds 246
queue_time_microseconds 97
source "tasks"
span.kind "client"
status.code 0
total_time_microseconds 346
Process: service.name = daily | process.runtime.version = 13.0 | process.runtime.name = BEAM | p...
SpanID: 23273231b02ca14d

Implementation



OpenTelemetry is a collection of tools, APIs, and SDKs. Use it to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) to help you analyze your software's performance and behavior.

OpenTelemetry is **generally available** across several languages and is suitable for use.

Erlang/Elixir OpenTelemetry API

EEF Observability hex v1.0.5

This is the API portion of OpenTelemetry for Erlang and Elixir Applications, implementing the API portion of the specification.

This is a library, it does not start any processes, and should be the only OpenTelemetry dependency of Erlang/Elixir Applications.

Use

There are both Erlang and Elixir macros that make use of the current module's name to lookup a Named Tracer -- a Named Tracer is created for each Application loaded in the system at start time -- for you and can be used for Trace and Span operations:

```
-include_lib("opentelemetry_api/include/otel_tracer.hrl").

some_fun() ->
    ?with_span(<<"some_fun/0">>, #{},
        fun(_SpanCtx) ->
            ...
            ?set_attribute(<<"key">>, <<"value">>),
            ...
        end),
```

Erlang/Elixir OpenTelemetry SDK

The SDK is an implementation of the OpenTelemetry API and should be included in your final deployable artifact (usually an OTP Release).

Configuration

The SDK starts up its supervision tree on boot, so initial configuration must be done through the Application or OS environment variables. The following example configurations show configuring the SDK to use the batch span processor which then exports through the OpenTelemetry Protocol over HTTP to `http://localhost:4318`, encoding the Spans with protobufs.

```
[
 {opentelemetry,
  [{span_processor, batch},
   {traces_exporter, otlp}]},

 {opentelemetry_exporter,
  [{otlp_protocol, http_protobuf},
   {otlp_endpoint, "http://localhost:4318"}]}]}
].
```

```
config :opentelemetry,
```

OpentelemetryTelemetry

`OpentelemetryTelemetry` provides conveniences for leveraging `telemetry` events for `OpenTelemetry` bridge libraries.

OpenTelemetry Contexts

`opentelemetry` does not automatically set current span context when ending another span. Since `telemetry` events are executed in separate handlers with no shared context, correlating individual events requires a mechanism to do so. Additionally, when ending telemetry-based spans, the user must set the correct parent context back as the current context. This ensures sibling spans are correctly correlated to the shared parent span.

This library provides helper functions to manage contexts automatically with `start_telemetry_span/4`, `set_current_telemetry_span/2`, and `end_telemetry_span/2` to give bridge library authors a mechanism for working with these challenges. Once `start_telemetry_span/4` or `set_current_telemetry_span/2` are called, users can use all of `OpenTelemetry` as normal. By providing the application tracer id and the event's metadata, the provided span functions will identify and manage span contexts automatically.

Example Telemetry Event Handlers

```
  def handle_event(_event,
                   %{system_time: start_time},
                   metadata,
                   %{type: :start, tracer_id: tracer_id, span_name: name}) do
    start_opts = %{start_time: start_time}
```

OpentelemetryPhoenix

OpentelemetryPhoenix uses telemetry handlers to create `OpenTelemetry` spans.

Current events which are supported include endpoint start/stop, router start/stop, and router exceptions.

Usage

In your application start:

```
def start(_type, _args) do
  OpentelemetryPhoenix.setup()

  children = [
    {Phoenix.PubSub, name: MyApp.PubSub},
    MyAppWeb.Endpoint
  ]

  opts = [strategy: :one_for_one, name: MyStore.Supervisor]
  Supervisor.start_link(children, opts)
end
```

Summary

OpentelemetryEcto

Telemetry handler for creating OpenTelemetry Spans from Ecto query events. Any relation preloads, which are executed in parallel in separate tasks, will be linked to the span of the process that initiated the call. For example:

```
Tracer.with_span "parent span" do
  Repo.all(Query.from(User, preload: [:posts, :comments]))
end
```

this will create a span called "parent span" with three child spans for each query: users, posts, and comments.

> **Note**
>
> Due to limitations with how Ecto emits its telemetry, nested preloads are not represented as nested spans within a trace.

Summary

Functions

setup(event_prefix, config \\ [])

OpenTelemetryTesla

Tesla middleware that creates OpenTelemetry spans and injects tracing headers into HTTP requests for Tesla clients.

Installation

If available in Hex, the package can be installed by adding `opentelemetry_tesla` to your list of dependencies in `mix.exs`:

```
def deps do
  [
    {:opentelemetry_tesla, "~> 2.0.1"}
  ]
end
```

Setup

Whilst using this middleware is as simple as adding it to your Tesla middlewares configuration, **It's very important to set the correct order of the middlewares**

The is crucial to correctly get the parameterized version of the URL, something like `/api/users/:id` instead of `/api/users/3`.

`OpenTelemetry` comes **first**, `PathParams` (if you're using it) comes after.

Add relevant libraries

```
def deps do
  [
    {:opentelemetry, "~> 1.0"},
    {:opentelemetry_exporter, "~> 1.0"},
    {:opentelemetry_phoenix, "~> 1.0"},
    {:opentelemetry_ecto, "~> 1.0"},
    {:opentelemetry_tesla, "~> 2.0"}
  ]
end
```

Setup instrumentation

On your `Application.start`

```
  OpentelemetryEcto.setup([:daily, :repo])
  OpentelemetryPhoenix.setup()
```

On your Tesla client, add:

```
Tesla.client([
  # ...
  Tesla.Middleware.OpenTelemetry
])
```

Configure the SDK and Exporter

Easiest way is to set the following environment variables

```
OTEL_EXPORTER_OTLP_TRACES_ENDPOINT: "http://tempo:4317"
OTEL_EXPORTER_OTLP_TRACES_PROTOCOL: grpc
OTEL_SERVICE_NAME: "daily"
```

Extra: incluide trace_id in logs

Add to your `Phoenix.Endpoint`:

```
plug :set_logger_trace_id

def set_logger_trace_id(conn, _opts) do
  span_ctx = OpenTelemetry.Tracer.current_span_ctx()

  if span_ctx != :undefined do
    Logger.metadata(trace_id: OpenTelemetry.Span.hex_trace_id(span_ctx))
  end

  conn
end
```