Telemetry

codecov 86%

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:

```elixir
:telemetry.execute(
  [:web, :request, :done],
  %{latency: latency},
  %{request_path: path, status_code: status}
)
```

In Erlang:

```erlang
telemetry:execute(
  [web, request, done],
  #{latency => Latency},
  #{request_path => Path, status_code => Status}
)
```

Then you can create a module to be invoked whenever the event happens.

In Elixir:

```elixir
defmodule LogResponseHandler do
  require Logger

  def handle_event([:web, :request, :done], measurements, metadata,
      _config) do
    Logger.info(
      "[#{metadata.request_path}] #{metadata.status_code} sent in #{
         measurements.latency}"
    )
  end
end
```

In Erlang:

```
-module(log_response_handler).

-include_lib("kernel/include/logger.hrl").

handle_event([web, request, done], #{latency := Latency}, #{
    request_path := Path,
                                                              status_code
                                                                  :=
                                                                Status},
                                                                 _Config
                                                                ) ->
  ?LOG_INFO("[~s] ~p sent in ~p", [Path, Status, Latency]).
```

Important note:

The `handle_event` callback of each handler is invoked synchronously on each `telemetry:execute` call. Therefore, it is extremely important to avoid blocking operations. If you need to perform any action that is not immediate, consider offloading the work to a separate process (or a pool of processes) by sending a message.

Finally, all you need to do is to attach the module to the executed event.

In Elixir:

```
:ok =
  :telemetry.attach(
    # unique handler id
    "log-response-handler",
    [:web, :request, :done],
    &LogResponseHandler.handle_event/4,
    nil
  )
```

In Erlang:

```
ok = telemetry:attach(
  %% unique handler id
  <<"log-response-handler">>,
  [web, request, done],
  fun log_response_handler:handle_event/4,
  []
)
```

You might think that it isn't very useful, because you could just as well write a log statement instead of calling `telemetry:execute/3` – and you would be right! But now imagine that each Elixir library would publish its own set of events with information useful for introspection. Currently each library rolls their own instrumentation layer – Telemetry aims to provide a single interface for these use cases across the whole ecosystem.

Spans

In order to provide uniform events that capture the start and end of discrete events, it is recommended that you use the `telemetry:span/3` call. This function will generate a start event and a stop or exception event depending on whether the provided function executed successfully or raised an error. Under the hood, the `telemetry:span/3` function leverages the `telemetry:execute/3` function, so all the same usage patterns apply. If an exception does occur, an `EventPrefix ++ [exception]` event will be emitted and the caught error will be re-raised.

The measurements for the `EventPrefix ++ [start]` event will contain a key called `system_time` which is derived by calling `erlang:system_time()`. For `EventPrefix ++ [stop]` and `EventPrefix ++ [exception]` events, the measurements will contain a key called `duration` and its value is derived by calling `erlang:monotonic_time() - StartMonotonicTime`. All events include a `monotonic_time` measurement too. All of them represent time as native units.

To convert the duration from native units you can use:

```
milliseconds = System.convert_time_unit(duration, :native, :millisecond
  )
```

To create span events you would do something like this:

In Elixir:

```
def process_message(message) do
  start_metadata = %{message: message}

  result =
    :telemetry.span(
      [:worker, :processing],
      start_metadata,
      fn ->
        result = ... # Process the message
        {result, %{metadata: "Information related to the processing of
          the message"}}
      end
    )
end
```

In Erlang:

```
process_message(Message) ->
  StartMetadata =  #{message => Message},
  Result = telemetry:span(
    [worker, processing],
    StartMetadata,
    fun() ->
      Result = % Process the message
      {Result, #{metadata => "Information related to the processing of
        the message"}}
    end
  ).
```

To then attach to the events that `telemetry:span/3` emits you would do the following:

In Elixir:

```
:ok =
  :telemetry.attach_many(
    "log-response-handler",
    [
      [:worker, :processing, :start],
      [:worker, :processing, :stop],
      [:worker, :processing, :exception]
    ],
    &LogResponseHandler.handle_event/4,
    nil
  )
```

In Erlang:

```
ok = telemetry:attach_many(
  <<"log-response-handler">>,
  [
    [worker, processing, start],
    [worker, processing, stop],
    [worker, processing, exception]
  ],
  fun log_response_handler:handle_event/4,
  []
)
```

With the following event handler module defined:

In Elixir:

```elixir
defmodule LogResponseHandler do
  require Logger

  def handle_event(event, measurements, metadata, _config) do
    Logger.info("Event: #{inspect(event)}")
    Logger.info("Measurements: #{inspect(measurements)}")
    Logger.info("Metadata: #{inspect(metadata)}")
  end
end
```

In Erlang:

```erlang
-module(log_response_handler).

-include_lib("kernel/include/logger.hrl").

handle_event(Event, Measurements, Metadata, _Config) ->
  ?LOG_INFO("Event: ~p", [Event]),
  ?LOG_INFO("Measurements: ~p", [Measurements]),
  ?LOG_INFO("Metadata: ~p", [Metadata]).
```

Installation

Telemetry is available on Hex. To install, just add it to your dependencies in `mix.exs`:

```elixir
defp deps() do
  [
    {:telemetry, "~> 1.0"}
  ]
end
```

or `rebar.config`:

```erlang
{deps, [{telemetry, "~> 1.0"}]}.
```

Copyright and License