

# Telemetry for Actionable Ops in Cloud Native Application

## Introduction

Software quality has always been an issue in this software controlled world. Software development teams are struggling to tackle any issues related to software defects. To cover this problem, instrumentation is a technique in software engineering to make a software observable. By using instrumentation technique, an instrumented software can emit data about its behavior. This data is called **telemetry**. This article will discuss telemetry, software for telemetry, and an example of how we can use this technique for actionable ops using Elixir - Phoenix framework and any other related software.

## Table of Contents

1. Understanding Telemetry
2. Telemetry in the Real World
3. Telemetry Data
4. OpenTelemetry Project
5. Architecture of Telemetry Systems Based on OpenTelemetry
6. Implementation
  - 6.1. Development Tools Preparation
  - 6.2. RESTful API Using Phoenix
  - 6.3. OpenTelemetry Configuration for Phoenix
  - 6.4. Embed OpenTelemetry for Erlang/Elixir Libraries Inside Phoenix
  - 6.5. OpenTelemetry Collector
  - 6.6. Docker Compose
  - 6.7. Distributed Tracing Backend: Jaeger and Zipkin
7. Actionable Ops
8. Summary
9. References
10. Abbreviations
11. Tags

## 1. Understanding Telemetry

Telemetry involves any measurement data inside devices or software to gain a better understanding towards the systems under observation. Although this article encompasses software, be advised though that software telemetry is only a part of bigger telemetry systems. Telemetry systems have been around since the late 19th century for electronic devices. The purpose of a telemetry system is to collect data at a place that is remote or inconvenient to relay the data to a point where the data may be evaluated [1]. Current advancement now involves not only devices but also software. In a world where software has a very important role in daily activities of people, software telemetry is needed to understand software behavior, quality, as well as security threats.

In software telemetry, developers put instrumentation codes which enable them to get inside data about software under observation. It is now easier to put instrumentation codes and get telemetry data since there are many services in the cloud which can be used

to monitor and evaluate telemetry data. Even without (public) cloud, we may use many open source solutions for observability purposes. Since cloud native apps are common, we will discuss how we can use telemetry for cloud native apps.

## 2. Telemetry in the Real World

Telemetry has been used in many places. Everything in this world which has a relationship with electronic devices and software may use instrumentation techniques and emit telemetry data for monitoring and evaluation: space shuttle, drilling in oil and gas industry, car racing, transportation, and many others. For software telemetry, especially mass software, there is privacy consideration in emitting telemetry data. Software should ask for explicit consent for telemetry data and explain whatever data is gathered from the user side. Some examples of this are:

- Visual Studio Code (which has another fork to bypass telemetry data: VSCodium [7]).
- .NET SDK and .NET CLI also collect usage data [5].
- Go start to discuss telemetry implementation in their toolchain [4]

However, instrumentation and telemetry are not without their downside. There are two important issues:

- Execution time. Instrumentation means we have to put some configuration and source code which uses telemetry libraries to intercept main flow and propagate telemetry data.
- Privacy. Some telemetry data in a mass public software can be used to propagate and analyze user data. This makes it possible to violate privacy.

## 3. Telemetry Data

In telemetry, emitted data are known as **signals**. There are three kind of signals:

- **Traces**: data about requests made by the application client. The unit of operation in traces is called **span**. A span consists of data about request and request results such as name, attributes, events, status, etc.
- **Metrics**: aggregations of numeric data (about software in observation) over a period of time. Examples of metrics are system error rate, CPU utilization, etc.
- **Logs**: time stamped message about everything that happens inside the software.

Examples of logs are log about changes in data inside the database.

Beside all of those signals, there is also a **Baggage**. A baggage is a place to store and retrieve data across spans. It is meant to be used by developers who want to propagate data from various spans since every span is immutable.

## 4. OpenTelemetry Project

OpenTelemetry project (OTel for short) is the result of a merger of 2 open source projects in May 2019: OpenTracing and OpenCensus. OpenTracing was used to send telemetry data to observability backend, while OpenCensus provided language specific libraries to instrument the software. Currently, OTel is developed under CNCF (Cloud Native Computing Foundation) to provide vendor-agnostic specification, SDK, API, and tools to instrument, collect emitted telemetry data, and export those data to the backend. The backed now agree to use and implement OTel specification for the benefit of software developers.

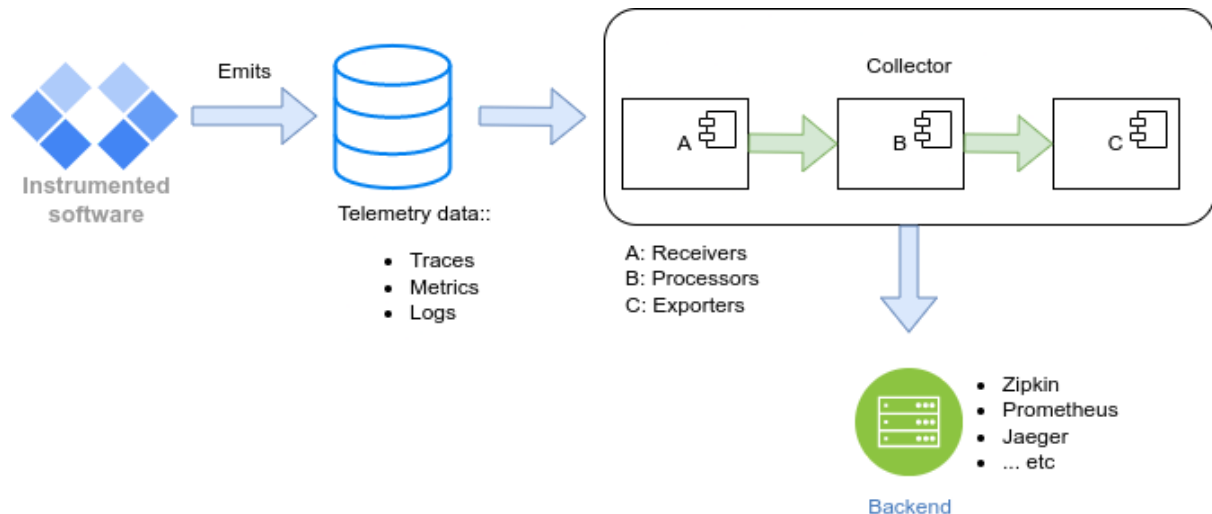
## 5. Architecture of Telemetry Systems Based on OTel

Conceptually, an OTel systems forms a pipeline from an instrumented software to the backend we choose. To put a developed software under observation, we need to put instrumentation codes inside. In this regard, we inject language specific libraries inside the source code of software being observed. Basically, some languages and libraries provides automatic instrumentation and also manual instrumentation. In the real world, we could possibly need to have both of them.

When the instrumentation is done, any activities inside the instrumented software will emit telemetry data. Depending on what kind of observability that we need, we can emit *traces*, *metrics*, and / or *logs* - usually all of those three signals.

The emitted data is then sent to the **OTel collector**. Inside the collector, there are **receivers** (which receive telemetry data), **processors** (which transform telemetry data into whatever we need), and **exporters** (which export processed telemetry data to backend for monitoring and evaluation).

At the end of OTel pipeline, a backend (or more than one if you prefer) is used to gather and visualize telemetry data for actionable operations. By looking at this backend output, developers can understand what is happening inside the software so that they can take necessary actions.



## 6. Implementation

In this part, we will use instrumentation techniques for Phoenix web framework (although in this article we will only use Phoenix for RESTful API). Phoenix has its own telemetry systems and automatically available in LiveView but since we use OpenTelemetry, we will combine Phoenix with OpenTelemetry and have the telemetry data sent to distributed tracing (in this case: Jaeger and Zipkin). We do not cover how to get started with Phoenix, instead you may go to Phoenix documentation [6]. Telemetry is a complex subject, and in

this article, we emit only traces. Other signals worth their own articles. **Traces** component in Erlang/Elixir currently is in stable release while **metrics** and **logs** are still *experimental* [2].

## 6.1 Development Tools Preparation

To follow this part, some development tools are needed:

- Erlang (version 23, 24, or 25)
- Elixir (version 1.14.4 - needed by Phoenix 1.7.2)
- Phoenix (version 1.7.2), also with Ecto.
- Python (needed by **Locust - Load Testing Tool**), you may change to other software if you want.
- Docker (stable version, currently 23.0.5 - no need to have this version, previous version should work as well - at least 19.x.x).
- Docker Compose (stable version, currently 2.17.3 - other version may work as well).

## 6.2 RESTful API Using Phoenix

In this article, we will create a service for the Human Resources department. Phoenix has the facility to create CRUD for a table but for this purpose we will only deal with data query using GET method. The only table for this article is **employee** table which consists of these columns:

- id: UUID, primary key
- name: string, name of employee
- email: email of employee, unique
- occupation: occupation inside the organization, string.
- address: address of employee, string

Here is an excerpt of the steps to create our RESTful API:

1. Install hex package manager: `mix local.hex`
2. Install Phoenix: `mix archive.install hex phx_new 1.7.2`
3. Create Phoenix project without HTML and static assets since we will build only RESTful API: `mix phx.new hr_service --no-html --binary-id`
4. Configure database access in `config/dev.exs`:

---

```
...
...
config :hr_service, HrService.Repo,
  username: "postgres",
  password: "postgres",
  hostname: "localhost",
  database: "hr_service_dev",
  port: "5432",
  stacktrace: true,
  show_sensitive_data_on_connection_error: true,
  pool_size: 10
...
...
```

---

5. Create database: `mix ecto.create`

6. Create schema: `mix phx.gen.context Admin Employee employees name:string email:string:unique occupation:string address:string`
7. Run migration: `mix ecto.setup` (normally we use `ecto.migrate` but since we define `ecto.setup` to include `ecto.migrate`, we can use `ecto.setup`).
8. Create scaffolding: `mix phx.gen.json Admin Employee employees name:string email:string:unique occupation:string address:string --no-context --no-schema`
9. Configure router. Edit file `lib/hr_service_web/router.ex`:

---

```
...
...
scope "/api", HrServiceWeb do
  pipe_through :api

  get "/employees", EmployeeController, :index
  put "/employees", EmployeeController, :edit
  post "/employees", EmployeeController, :create
  delete "/employees", EmployeeController, :delete

end
...
...
```

---

We are going to use a PostgreSQL docker image. For the sake of brevity, whenever we already run docker-compose, use this command to enter **psql** client for aforementioned database:

```
psql -U postgres -W -d hr_service_dev -h localhost.
```

While we are inside, use this example INSERT SQL statement:

```
INSERT INTO EMPLOYEES (id, name, email, occupation, address, inserted_at,
updated_at) VALUES ('fd514a1f-6ae9-480d-b7df-8080927c492c', 'Bambang',
'bpdp@zimeracorp.com', 'Manager', 'Address of bpdp',
'2023-05-01', '2023-05-01');
```

**Note:** see the *docker-compose* below before running database related commands in Phoenix and also the whole HrService app..

10. Run HrService RESTFul API: `mix phx.server`
11. Using your browser, access <http://localhost:4000> to see the routes which can be requested by the client. Try to test <http://localhost/api/employees>.

### 6.3 OTel Configuration for Phoenix

To use OTel with Phoenix, install some OTel packages for Erlang/Elixir and Phoenix-Ecto. Edit ***mix.exs***:

---

```
...
...
defp deps do
  [
    {:phoenix, "~> 1.7.2"},
    ...
    ...
    {:plug_cowboy, "~> 2.5"},
    {:opentelemetry_exporter, "~> 1.4"},
    {:opentelemetry_api, "~> 1.2"},
    {:opentelemetry, "~> 1.3"},
    {:opentelemetry_semantic_conventions, "~> 0.2"},
    {:opentelemetry_cowboy, "~> 0.2"},
    {:opentelemetry_phoenix, "~> 1.1"},
    {:opentelemetry_ecto, "~> 1.1"}
  ]
end
...
...
```

---

Get those packages using this command: `mix deps.get` To configure Phoenix for instrumentation using OTel, proceed below.

## 6.4 Embed OpenTelemetry for Erlang/Elixir Libraries Inside Phoenix

We need to change some files so that our HrService can emit traces.

### config/config.exs

---

```
...
...
# Use Jason for JSON parsing in Phoenix
config :phoenix, :json_library, Jason

# Add OTel configuration below
# Console Exporter for OpenTelemetry Exporter
config :opentelemetry, :processors,
  otel_batch_processor: %{
    exporter: {:otel_exporter_stdout, []}
  }

config :opentelemetry, :resource,
  service: %{name: "hr-service", namespace: "zimera"}
## End of addition for OTel

# Import environment specific config. This must remain at the bottom
# of this file so it overrides the configuration defined above.
import_config "#{config_env()}.exs"
```

...  
...

---

In this configuration file, we need to configure console exporter and resources. In resources, we define service name which will be recognized by Jaeger and Zipkin.

#### config/dev.exs

---

```
...
...
config :swoosh, :api_client, false

config :opentelemetry,
  :processors,
  otel_batch_processor: %{
    # Using `localhost` here since we are starting outside
docker-compose where
    # otel would refer to the hostname of the OpenCollector,
    #
    # If you are running in docker compose, kindly change it to the
correct
    # hostname: `otel`
    exporter: {:opentelemetry_exporter, %{endpoints: [{:http,
"localhost", 4318, []}]}}
  }
...
...
```

---

Pay attention to the hostname. The hostname will different between container environment and non-container environment.

#### config/runtime.exs

---

```
...
...
config :opentelemetry, :processors,
  otel_batch_processor: %{
    # Using `otel` here since we are starting through docker-compose where
    # otel refer to the hostname of the OpenCollector,
    #
    # If you are running it locally, kindly change it to the correct
    # hostname such as `localhost`, `0.0.0.0` and etc.
    exporter: {:opentelemetry_exporter, %{endpoints: ["http://otel:4318"]}}
  }
...
...
```

---

Again, pay attention to the hostname inside container or outside container. If inside the container, see service name at docker-compose configuration - **compose.yaml** file (**otel** for OpenTelemetry collector).

lib/hr\_service/application.ex

---

```
...
...
use Application

@impl true
def start(_type, _args) do
  :opentelemetry_cowboy.setup()
  OpentelemetryPhoenix.setup(adapter: :cowboy2)
  OpentelemetryEcto.setup([:demo, :repo])

  children = [
...
...
```

---

This is where we put the instrumentation setup codes: before the top level supervisor starts. Since we use cowboy2 as our adapter, we use that as the parameter in our setup.

## 6.5 OTel Collector

OTel Collector should be configured so that it knows the receivers - processors - exporters pipeline. We will use OTel Collector as a docker image. This image needs a configuration (usually in */etc/otel-collector-config.yaml*). Therefore, we have to prepare this configuration so that we can copy this file in docker-compose operation.

otel-collector-config.yaml

---

```
receivers:
  otlp:
    protocols:
      grpc:
      http:
processors:
  batch:
    send_batch_size: 1024
    timeout: 5s
exporters:
  otlp:
  zipkin:
    endpoint: "http://zipkin:9411/api/v2/spans"
    tls:
      insecure: true
  jaeger:
    endpoint: "jaeger:14250"
    tls:
      insecure: true
extensions:
  zpages:
    endpoint: :55679
```



```
service:
  extensions: [zpages]
  pipelines:
    traces:
      receivers: [otlp]
      processors: [batch]
      exporters: [jaeger, zipkin]
```

---

## 6.6 Docker Compose

Our RESTful API in Phoenix is developed as a separate component. It will emit telemetry data to OpenTelemetry Collector. At the end of the telemetry pipeline, telemetry data will be sent to Jaeger and Zipkin (you may choose only one or both since both have the same functionalities). All of those infrastructure software are executed using Docker Compose. Here is the **compose.yaml** file.

---

```
version: '3'
services:
  postgres:
    image: 'postgres:15.2'
    ports:
      - 5432:5432

    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: hr_service_dev

  otel:
    image: otel/opentelemetry-collector-contrib-dev:latest
    command: ["--config=/etc/otel-collector-config.yaml"]
    ports:
      - "4318:4318"
      - '55681:55681'
      - '55680:55680'
      - "55679:55679"
    volumes:
      - ./otel-collector-config.yaml:/etc/otel-collector-config.yaml
    depends_on:
      - jaeger
      - zipkin

  zipkin:
    image: openzipkin/zipkin-slim
    container_name: "zipkin"
    ports:
      - '9411:9411'

# Jaeger
jaeger:
```

```
image: jaegertracing/all-in-one:latest
container_name: "jaeger"
ports:
  - "4317:4317"
  - "16686:16686"
  - "14268"
  - "14250:14250"
```

---

Run docker compose: `docker-compose up`. Docker compose will automatically read configuration from `compose.yml`. After docker compose runs all of those services, we may do any DBMS related operations and run HrService RESTful API.

## 6.7 Distributed Tracing Backend: Jaeger and Zipkin

Jaeger and Zipkin have the same functionalities (well, more or less). They both provide backend for distributed tracing. Both now support OTLP (The OpenTelemetry Protocol) specification. Using Jaeger and Zipkin we may visualize telemetry data (traces) for actionable ops. In the docker compose configuration file (**`compose.yml`**), we use both as our services. For Jaeger, we use **`jaegertracing/all-in-one`** image, while for Zipkin we use **`openzipkin/zipkin-slim`** image.

## 7. Actionable Ops

**Actionable** means having sufficient data / information needed to take a specific action. By having instrumentation inside our software being observed, we enable the software to emit telemetry data, sufficient to act towards software improvement or to prevent problems getting worse. To give an idea of how telemetry data works towards actionable ops, we will use a **Locust** for our load testing tool. Using **Locust**, we will simulate around 1000 users to access our RESTful API. Some requests are invalid requests by intention so that we know how we can use Jaeger to investigate the problems and act to resolve problems (in real time if possible).

`locustfile.py`

---

```
from locust import HttpUser, task, between

class LoadTesting(HttpUser):
    wait_time = between(1, 5)

    @task
    def root_employees(self):
        self.client.get("/api/employees", json={})

    @task(2)
    def not_exist_1(self):
        self.client.get("/api/employees/1", json={})

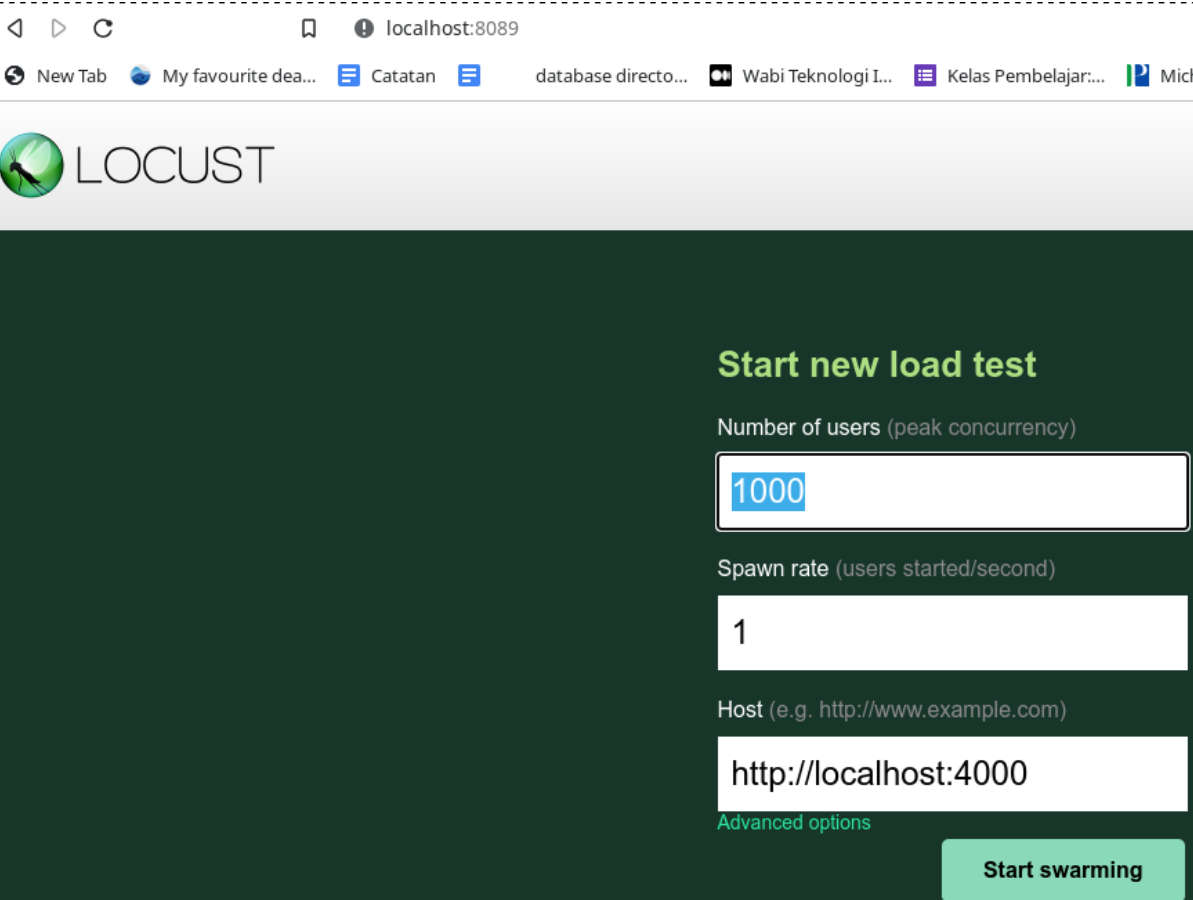
    @task(3)
```

```
def not_exist_2(self):  
    self.client.get("/api/employees", json={})
```

Execute **Locust** using that configuration file:

```
$ locust -f locustfile.py --host=http://localhost:4000  
[2023-05-01 14:00:09,224] dellvuan/INFO/locust.main: Starting web interface  
at http://0.0.0.0:8089 (accepting connections from all network interfaces)  
[2023-05-01 14:00:09,259] dellvuan/INFO/locust.main: Starting Locust 2.15.1
```

Access web UI for **Locust** and start load test using 1000 users:



The screenshot shows a web browser window with the address bar set to `localhost:8089`. The browser's tab bar includes 'New Tab', 'My favourite dea...', 'Catatan', 'database directo...', 'Wabi Teknologi I...', 'Kelas Pembelajaran...', and 'Mich'. The Locust web interface has a header with a green locust logo and the word 'LOCUST'. The main content area has a dark green background and features a 'Start new load test' section. This section contains three input fields: 'Number of users (peak concurrency)' with the value '1000', 'Spawn rate (users started/second)' with the value '1', and 'Host (e.g. http://www.example.com)' with the value 'http://localhost:4000'. Below these fields is a link for 'Advanced options' and a green 'Start swarming' button.

When we have finished our load testing, we may have a look at Jaeger. Jaeger will recognize **hr-service** which we have defined at one of our configurations. Choose **hr-service** and click on **Find Traces**.

localhost:16686/search

New Tab My favourite dea... Catatan database dire

JAEGER UI Search Compare System Architecture Monitor

Search Upload

Service (1)

hr-service

Operation (2)

all

Tags ?

http.status\_code=200 error=true

Lookback

Last Hour

Max Duration

e.g. 1.2s, 100ms, 500us

Min Duration

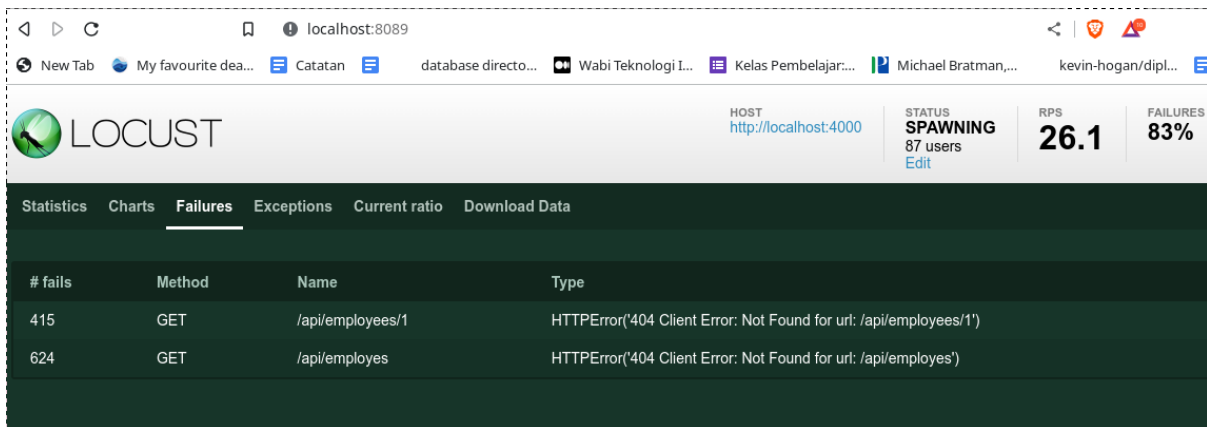
e.g. 1.2s, 100ms, 500us

Limit Results

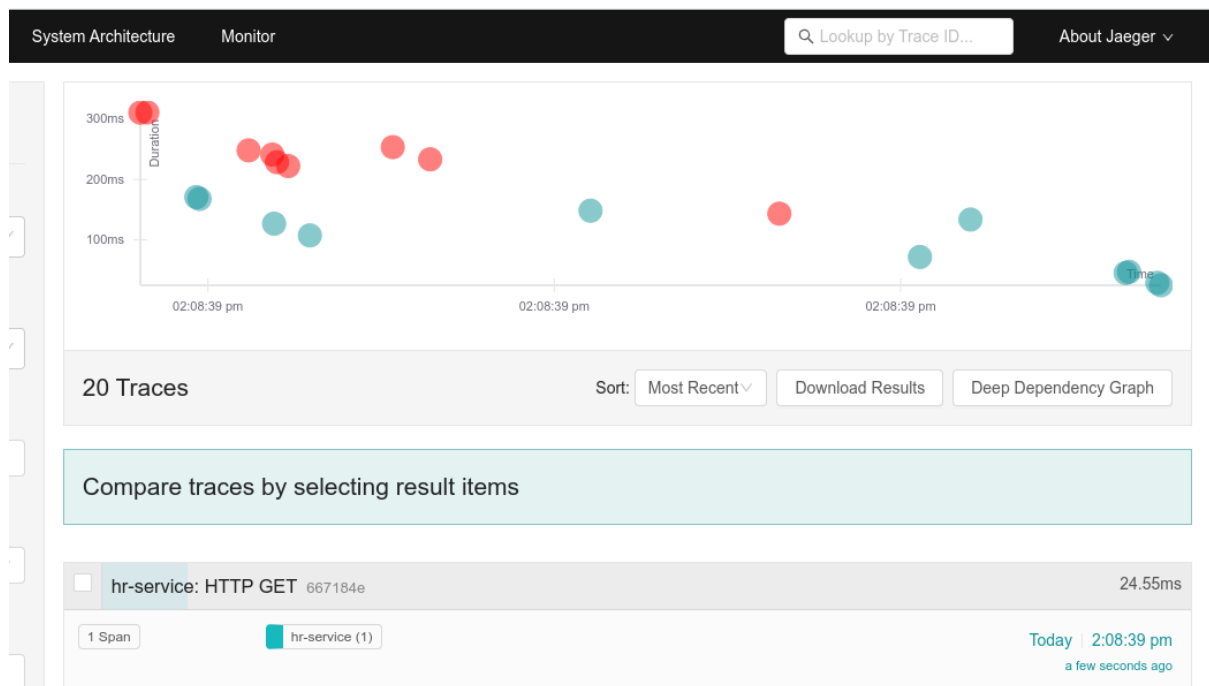
20

Find Traces

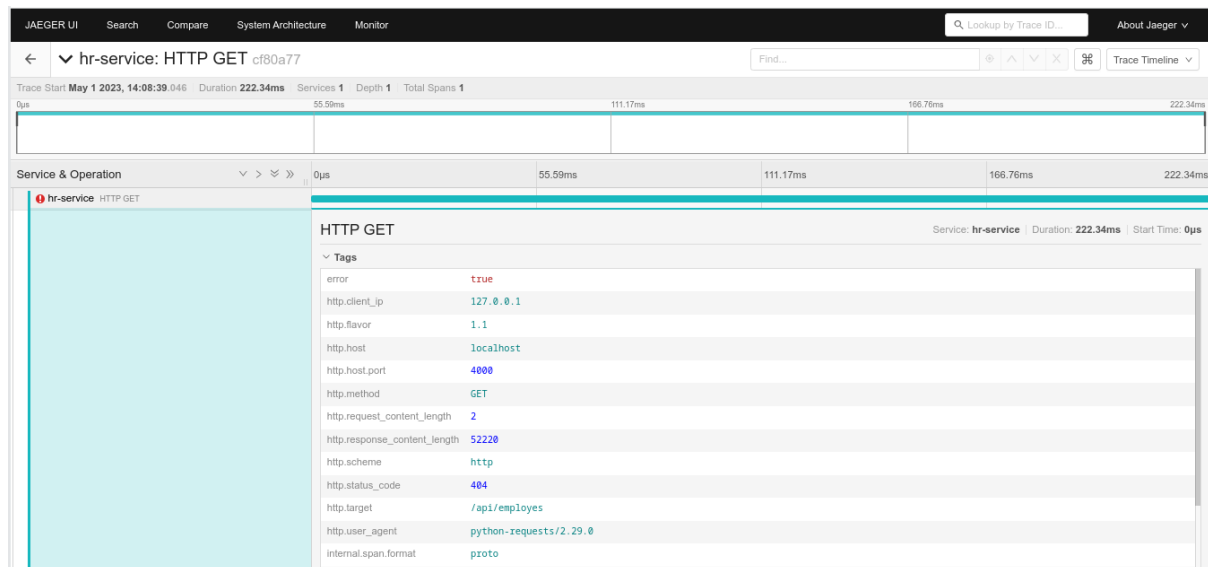
In Locust web UI, we can see also **failures** like this:



Any failures will be displayed red at Jaeger. We can click on the red dot for more information. Here we can get sufficient data and information which enable us to act towards any problems in our software.



Below is an example of Jaeger data and information if we click on one of the red dots. The data and information is enough to resolve problems.



## 8. Summary

Telemetry data is the result of instrumentation at the source code level in a software being observed. Telemetry data (also called *signals*) consist of *traces*, *metrics*, and *logs*. Telemetry data is sent to OTel Collector. Inside OTel Collector, there are receivers, processors, and exporters. The exporter part is responsible to export processed telemetry data to backend (in case of *traces*, we use distributed tracing backend). The backend is able to visualize data for monitoring and evaluation. This makes it possible for actionable ops since actionable means having sufficient data to act towards problem solving in software operations.

## 9. References

- [1] Carden, Frank, Russell P. Jedlicka, and Robert Henry. (2002). *Telemetry Systems Engineering*. Artech House.
- [2] <https://opentelemetry.io/docs/instrumentation/erlang/>
- [3] Riedesel, Jamie. (2022). *Exploring Software Telemetry*. Manning
- [4] <https://github.com/golang/go/discussions/58409>
- [5] <https://learn.microsoft.com/en-us/dotnet/core/tools/telemetry>.
- [6] Phoenix documentation: <https://hexdocs.pm/phoenix/overview.html>
- [7] <https://vscodeium.com/>

## 10. Abbreviations

- **CNCF**: Cloud Native Computing Foundation, a part of Linux Foundation to provide support, oversight and direction for fast-growing, cloud native projects.
- **OTel**: OpenTelemetry, an open source project under CNCF for telemetry specification, SDK, and tools
- **REST**: Representational State Transfer, an architectural style for distributed systems.
- **API**: Application Programming Interface, is a set of defined rules that enable different applications to communicate with each other.

## 11. Tags

Telemetry, OpenTelemetry, Actionable Ops, Cloud Native App, Elixir, Erlang, Distributed Tracing, Jaeger, Zipkin

## Notes

The source code for this article is available at  
<https://github.com/zimera-school/elixir-opentelemetry>