Distributed Software Architecture Lab 4 Report

# GOALS

Setup observability for services implemented in the previous lab, including:

* Scrape metrics with Prometheus and query with PromQL
* Visualize using Dashboard with Grafana
* Trace with OpenTelemetry & Jaeger

# SOURCE CODE

https://github.com/tvph1996/fourth\_lab

# SYSTEM SETUP

## Prometheus

### REST-service

* Use the same port of the service because the traffic is low and to reduce complexity. This is possible because both Prometheus and REST use plain HTTP.
* Prometheus works as a middleware in FastAPI webserver.
* Metrics scraped:
  + http\_request\_duration\_seconds
  + http\_requests\_total
* Tested PromQL queries:
  + Rate in the last 1 minute
  + Latency p95 in the last 5 minutes
  + Error rate in the last 5 minutes
* gRPC-service
* Use different port since gRPC used HTTP/2 while Prometheus uses HTTP.
* Prometheus works as an Interceptor.
* Metrics scraped:
  + grpc\_server\_handling\_seconds
* Tested PromQL queries:
  + Latency p95 in the last 5 minutes

## Grafana

* Custom Dashboard with panels:
  1. REST-service Rate
  2. REST-service Latency p95
  3. REST-service Error Rate
  4. gRPC-service Latency p95

## 3. OpenTelemetry & Jaeger

* Used protocol: OTLP-gRPC
* Setup:

REST-service & gRPC-service

└─*OTLP-gRPC* → otel-collector:4317 (manually configured)

└─*OTLP-gRPC* → jaeger:4317 (internally supported by framework)

## 4. Testing

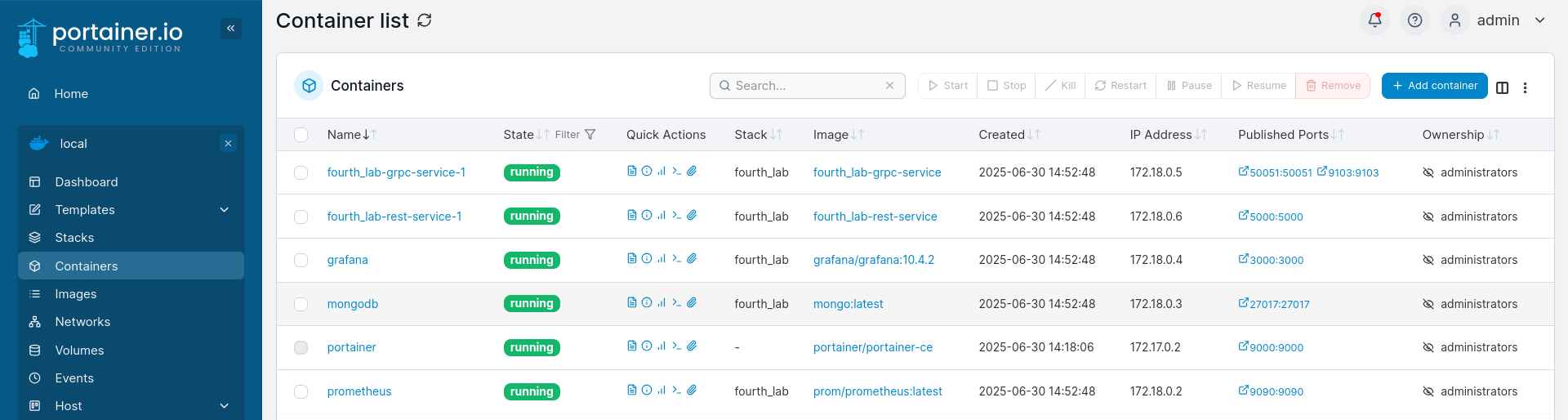
* Testscript to CURL request POST then GET method to REST-service every 2 seconds
* GET method is always successful
* POST method
  + Add new Item with random name from a fixed limited pool
  + Naming duplication is not allowed
* Expectation: Error rate of POST method is increased gradually

## OTHER NOTES

* Environment changed: Windows Subsystem Linux & Docker Desktop (both are VM) => Linux native

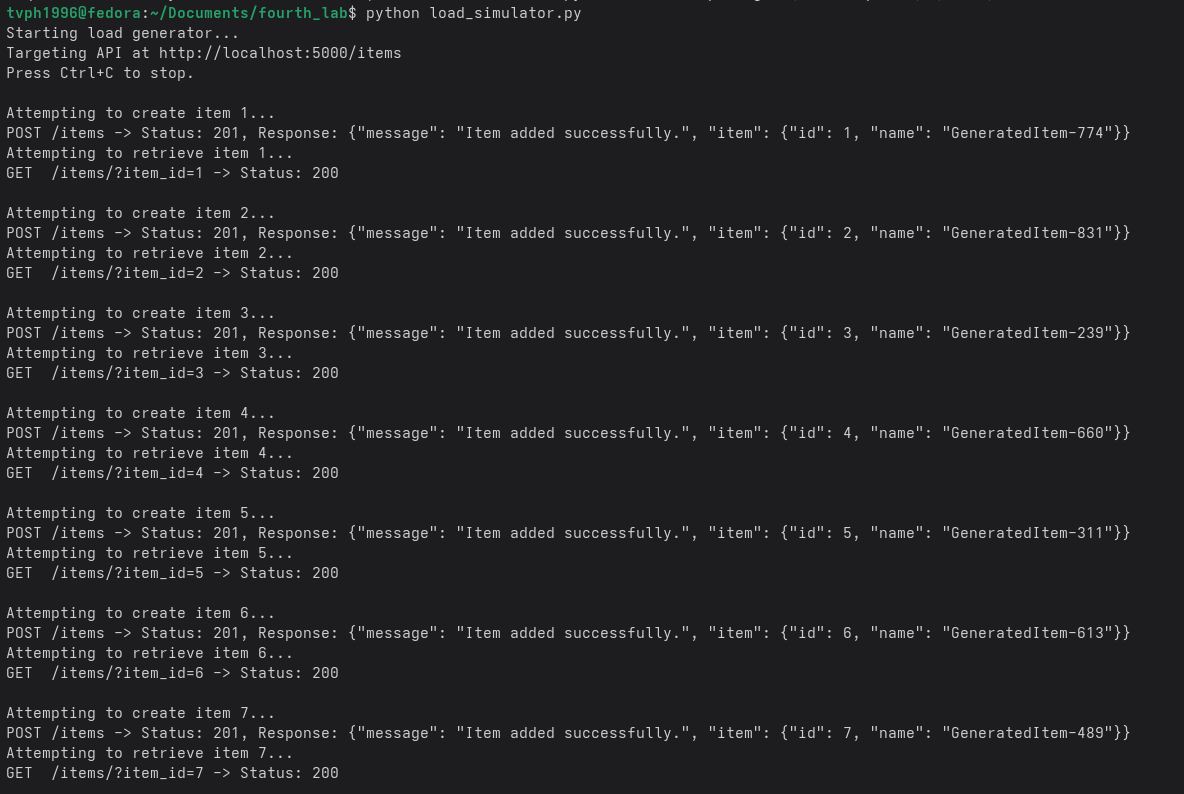
# OUTPUT & OBSERVATION

## docker compose up –d —build

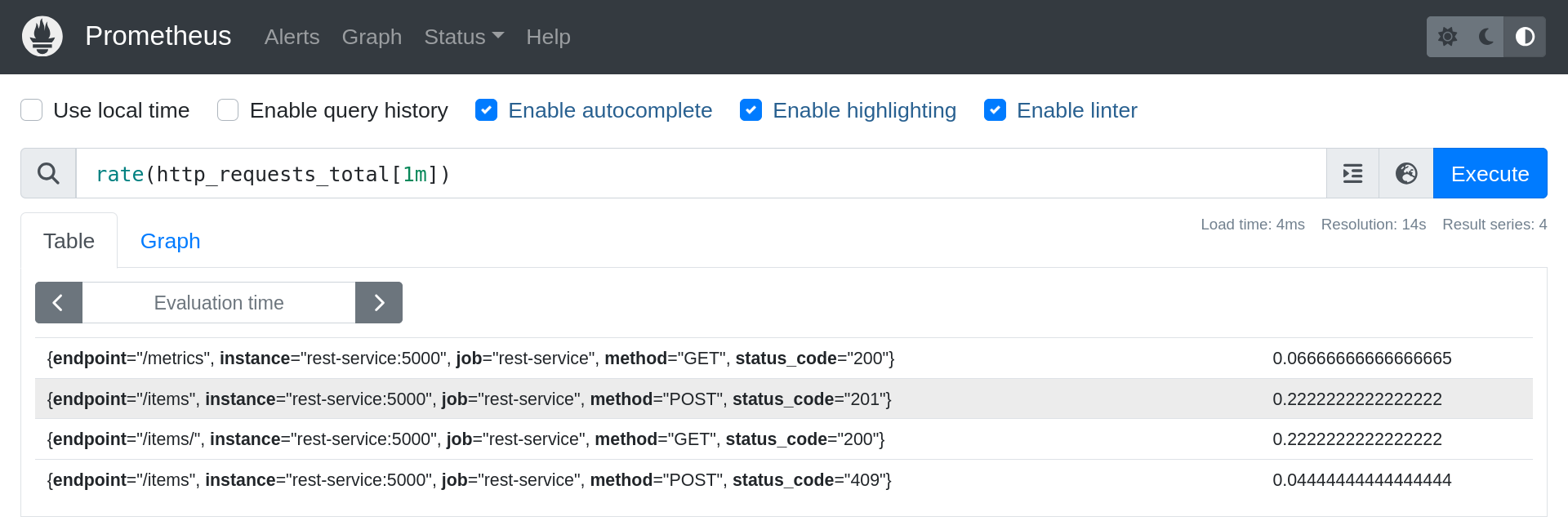


* View from Portainer – a container manager software in Linux
* This was taken before Telemetry was implemented

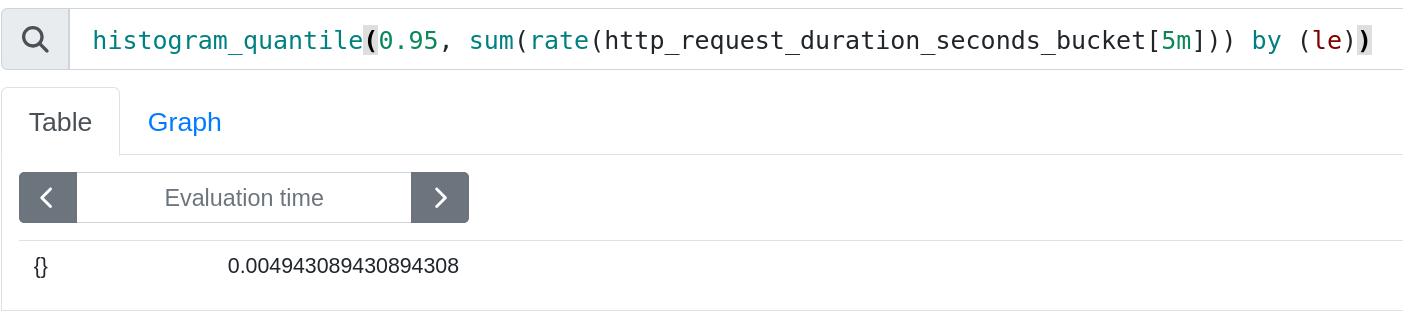
## Testscript started



## Prometheus PromQL: REST-service Rate

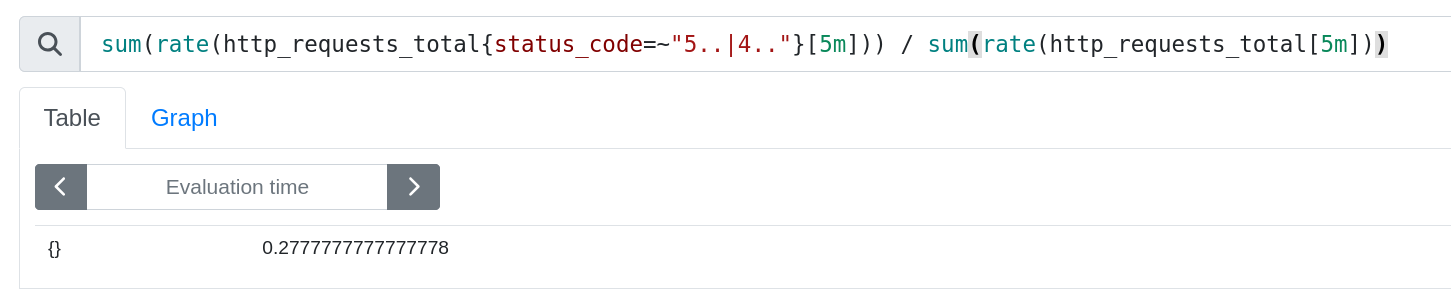


## Prometheus PromQL: REST-service Latency p95

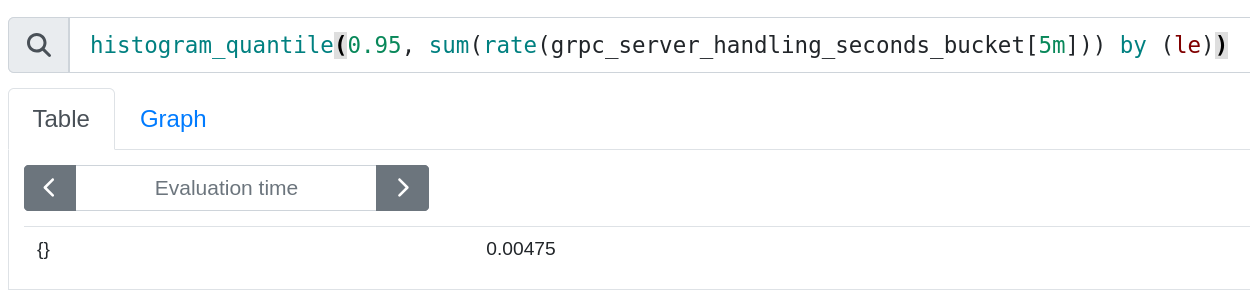


## 

## Prometheus PromQL: REST-service Error Rate



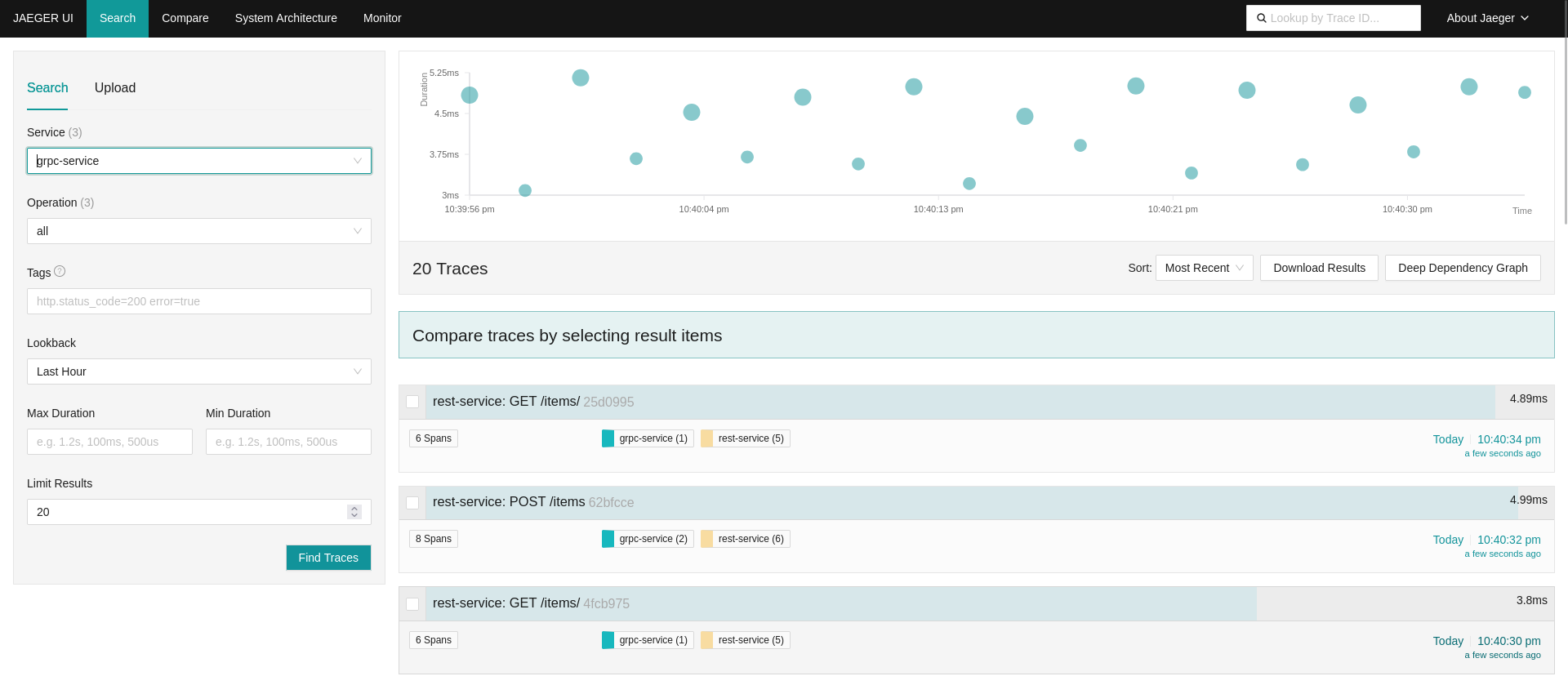
## Prometheus PromQL: gRPC-service Latency p95



## Grafana

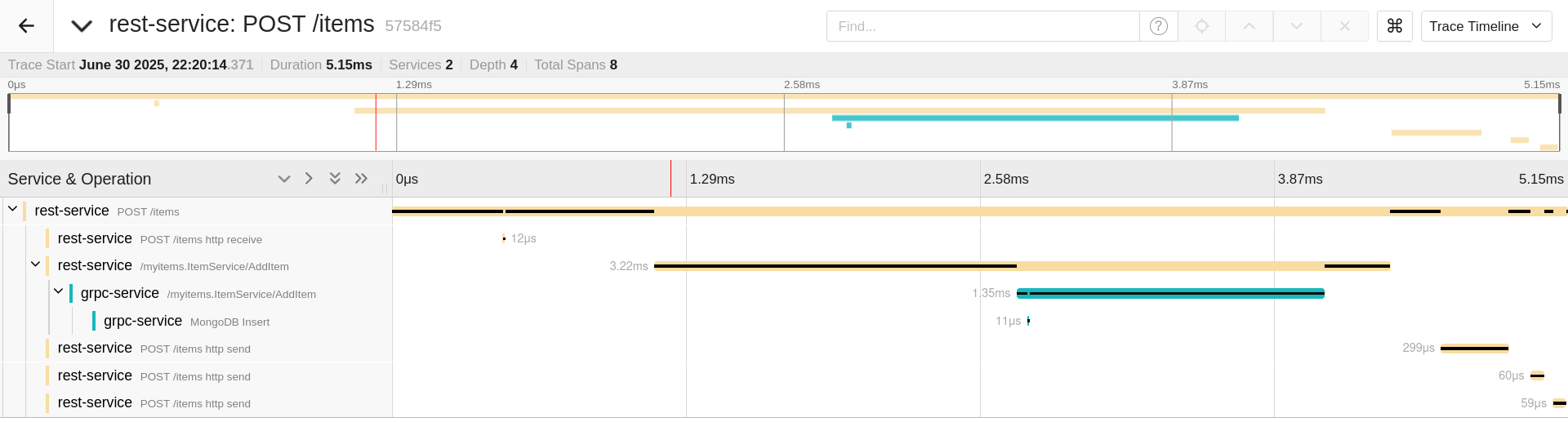


* As expected from the test design, REST-service Error Rate (top right panel) gradually increases
* Latency p95 of POST method (bottom right panel) increases when succeeded because of many involved code operations compared to immediate rejection when fail
* Latency p95 of AddItem method of gRPC-service (bottom left panel) is constant around 5ms. The reason should be because from lab2, gRPC calls can be as low as 0.5ms which is much smaller than the smallest Prometheus bucket allowed in python hence undisplayable. Switching to another Prometheus framework with smaller bucket such as 1ms may solve this dislay problem.
* The request rate (top left panel) of POST method is stable while GET method fluctuates. The reason is the timing required of GET method is much slower (even slower when POST method is successful), which means it must wait for POST method to be completed.



## OpenTelemetry & Jaeger

* Mainpage
* Tracing sample of a POST call from beginning to end



* Unsure why REST-service response consists of 3 small spans, this could be the design of REST API framework.
* There is around 0.5ms increase in latency because of OpenTelemetry instrumentation
  + This means more than 10% of timing in this setup -> Telemetry is useful for important calls or calls that is not ultra low latency, it is not a universal practice.
  + A good practice could be: Only trace a percentage of requests instead of everything (this is supported also by OpenTelemetry)

End of Report