How to test the performance of microservices

TLDR: use locust and Jaeger to understand why your software is slow.

Andrea Janes
Freie Universität Bozen, Italien | ajanes@unibz.it

locust.io

Hi!

- My name is Andrea Janes.
- I studied in Vienna and Klagenfurt.
- I am a passionate about software creation, architectures, and many things around it.
- I work at the Free University of Bozen-Bolzano as an Associate Professor in Computer Science.
- The best way to reach me is at ajanes@unibz.it.
- On the weekends you find me on some mountain or at some lake.

A simple performance test

```
import requests as rq, time

if __name__ == "__main__":
    print("Start")
    start = time.time()

    for i in range(10):
        print(f"Testing {i}")
        rq.post("http://localhost:1234/", data=f"Hi! {i}")

    print(f"Response time: {int((time.time() - start) * 1000)}")
```

A bit more complicated performance test

```
import requests as rq
import threading
import time

def worker(iterations: int) -> None:
    for _ in range(iterations):
        rq.post("http://localhost:1234/", data="Hi!")

if __name__ == "__main__":
    threads = []
    start = time.time()

    for _ in range(10):
        t = threading.Thread(target=worker, args=(10,))
        threads.append(t)
```

A bit more complicated performance test

```
        t.start()

    for t in threads:
        t.join()

    print(f"Total␣time:␣{int((time.time()␣-␣start)␣*␣1000)}␣ms")
```

Performance Testing

- Performance testing is the process of evaluating how well a system or application performs under various load conditions.
- It involves measuring key performance indicators (KPIs) such as response time, throughput, and resource utilization.
- Performance testing is typically done to identify bottlenecks, capacity limits, and scalability issues.

Types of Performance Testing

- Load testing: simulates user activity to measure how well the system handles a specific number of users and transactions.
- Stress testing: determines the system's ability to handle extreme loads beyond its capacity.
- Spike testing: tests the system's ability to handle sudden spikes in traffic.
- Endurance testing: evaluates the system's performance over an extended period of time to determine its stability and reliability.

Key Performance Indicators (KPIs)

- Response time: the time it takes for the system to respond to a user request.
- Throughput: the number of transactions that the system can handle per unit of time.
- Concurrency: the number of users or transactions that can be processed simultaneously.
- CPU usage: the percentage of CPU capacity being used by the system.
- Memory usage: the amount of memory being used by the system.
- Disk I/O: the speed at which the system reads and writes data to disk.

Locust.io[1]

- Locust.io is an open-source load testing tool written in Python.
- It allows you to write test scenarios in Python code and then run them to simulate user activity.
- Locust.io is designed to be highly scalable, so you can simulate thousands or even millions of users with ease.
- It also provides real-time monitoring and reporting of key performance metrics such as response time and throughput.

[1] https://locust.io

Writing a Locust.io Test Scenario

- To write a Locust.io test scenario, you need to define a class that inherits from the `HttpUser` class.
- This class should define a set of tasks that represent user actions, such as making HTTP requests.
- Each task should be decorated with the `task` decorator and should use the `self.client` attribute to make HTTP requests.
- You can also define task sequences and use the `@seq_task` decorator to specify the order in which tasks should be executed.

example.py

```
from locust import HttpUser, between, task

class MyUser(HttpUser):

    @task
    def go_to_homepage(self):
        self.client.get("/")

    # Define a task to perform a search (executed 3 times as often ↘
       as the other tasks)
    @task(3)
    def search(self):
        self.client.get("/search?q=locust")

    @task(2)
```

example.py

```
    def view_product(self):
        product_id = self.client.get("/products").json()[0]["id"]
        self.client.get(f"/products/{product_id}")

    @task(3)
    def add_to_cart(self):
        product_id = self.client.get("/products").json()[0]["id"]
        self.client.post(f"/cart", json={"product_id": product_id, "↘
            quantity": 1})
```

Running locust manually

- Install Python
- `pip install locust`
- Run `locust -f my_locust_file.py`
- Open the page `http://127.0.0.1:8089`

Exercise: execute a performance test

- Download https://github.com/ajanes/tutorial_sfscon
- Open a terminal and change path to example1
- `python -m venv .venv`
- `source ./.venv/bin/activate.fish`
- `pip install -r requirements.txt`
- `python server.py`

```
 * Serving Flask app 'server'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a produ
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```

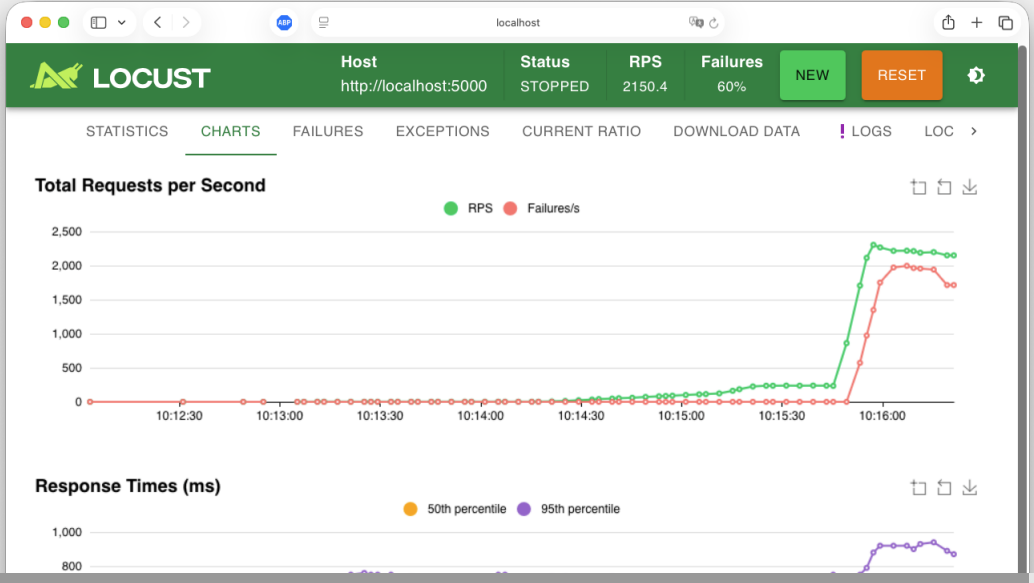
server.py

```
import random, time
from flask import Flask, jsonify

app = Flask(__name__)

@app.route("/")
def home():
    return "Hi!"

@app.route("/users")
def get_users():
    time.sleep(random.uniform(0.25, 0.75))
    return jsonify(["Alice", "Bob", "Charlie"])

if __name__ == "__main__":
```

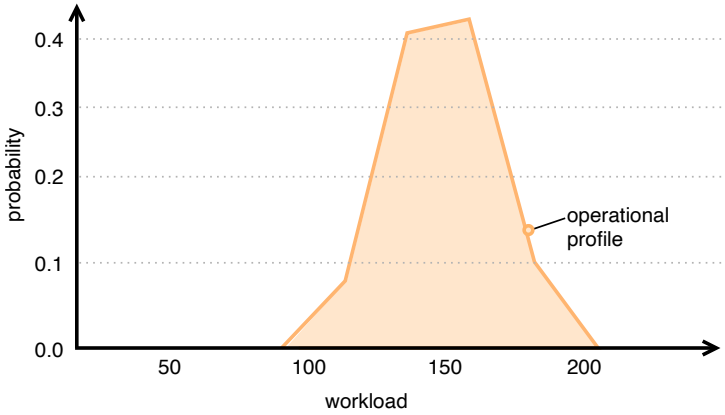
server.py

```
app.run(debug=True)
```

http://localhost:8089



Operational profile



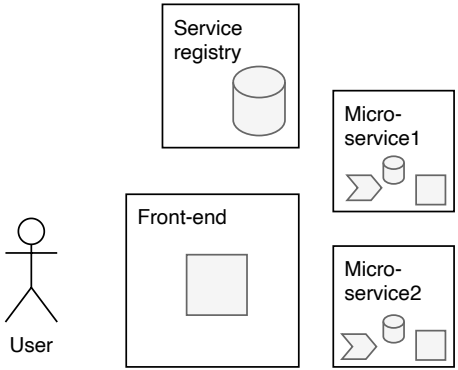
jaegertracing.io

Contents

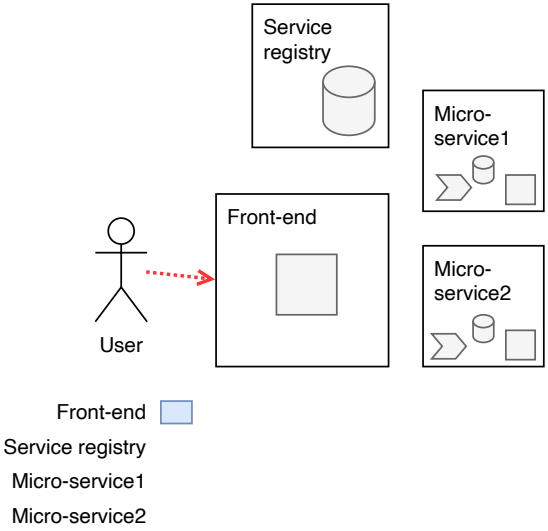
Distributed Tracing Overview

- Definition: the practice of following requests end-to-end across services to understand latency and dependencies.
- Why is it important?
  - Improves user experience by revealing slow hops
  - Prevents downtime and data loss via rapid diagnosis
  - Increases efficiency by showing where to optimize
  - Facilitates root cause analysis with concrete evidence

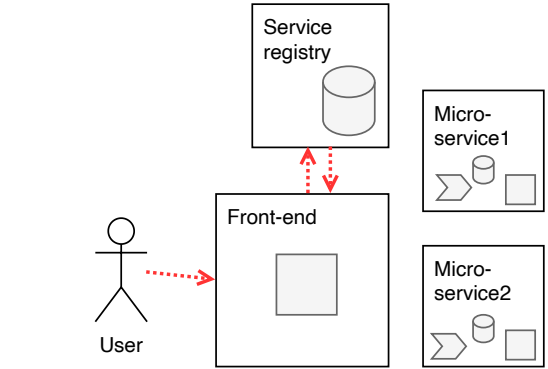
Tracing Workflow



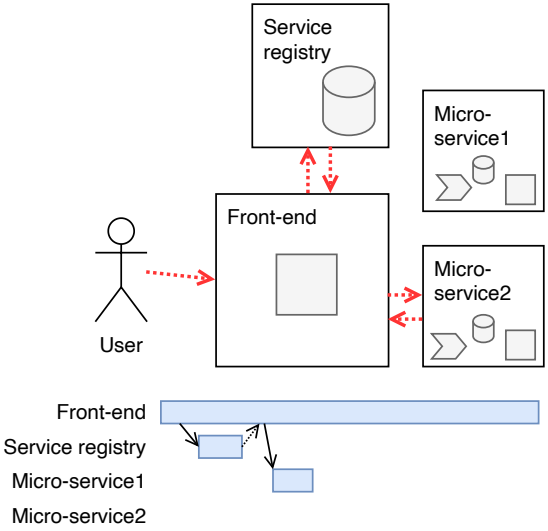
Tracing Workflow



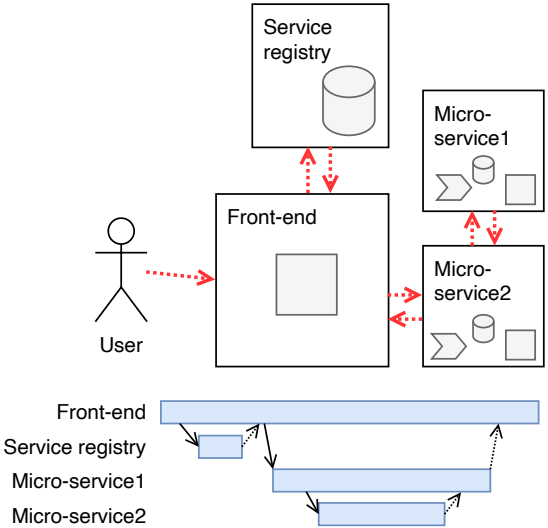
Tracing Workflow



Tracing Workflow



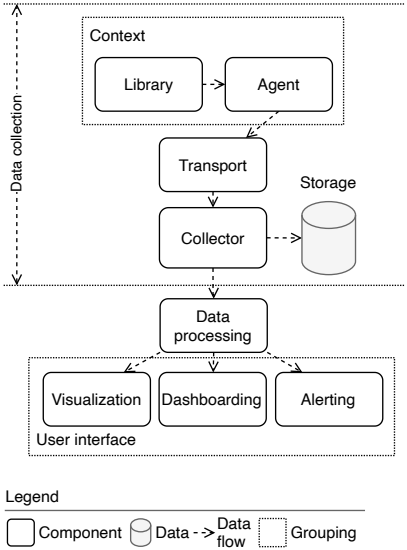
Tracing Workflow



Core Observability Building Blocks

- Metrics: data points that are collected and analyzed to measure application performance (e.g., response time, CPU usage, memory usage).
- Alerts: notifications that are triggered when certain thresholds are exceeded (e.g., if response time exceeds a certain threshold, an alert is triggered).
- Dashboards: graphical representations of application performance metrics that allow users to easily monitor performance over time.
- Tracing: the process of tracking requests as they move through an application, showing how long each hop or component takes to process a request.

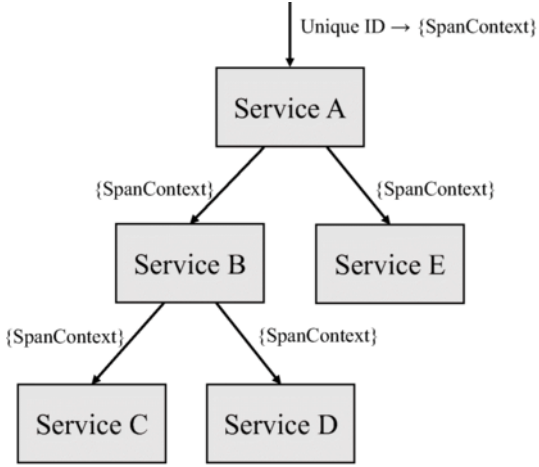
Typical tracing architecture



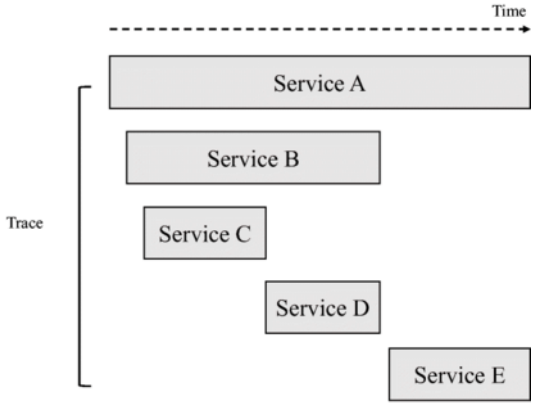
How a tracing backend works

- Modern systems use “spans” to represent a single unit of work in a distributed system.
- Spans are grouped together into “traces” to represent a full request or transaction.
- Each span contains timing information, including the start time, duration, and any associated metadata (such as HTTP headers).
- A typical backend exposes a collector API to receive trace data and a query component to search and visualize traces.

Spans vs Traces



(a) Trace Topology



(b) Trace and Span

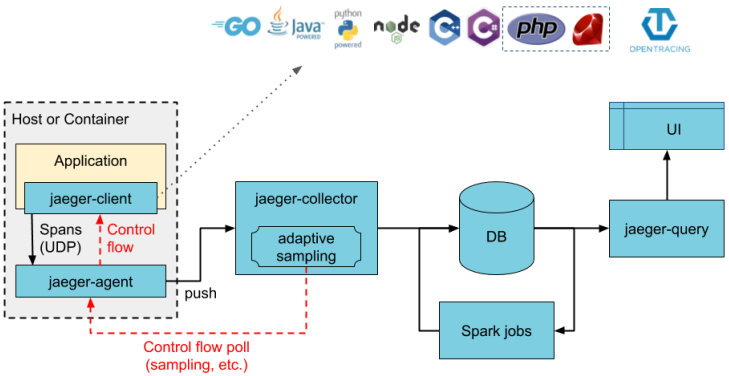
Key tracing backend features

- Distributed tracing: allows users to trace requests as they move through a distributed system.
- Sampling: allows users to control the amount of data that is collected and stored, reducing the impact on system performance.
- Instrumentation: ecosystem libraries help instrument applications written in various programming languages, including Java, Ruby, and Go.
- Integration: tracing backends often connect with tools such as Prometheus, Grafana, and Kubernetes.

Contents

How does Jaeger Tracing work?

- Jaeger works by instrumenting your code with tracing libraries that create and propagate trace context across service boundaries. When a request comes in, each service creates a span and sends it to Jaeger via a collector.



Getting started with Jaeger Tracing

To get started with Jaeger, you'll need to do the following:

- Instrument your code with Jaeger tracing libraries.
- Configure your services to send trace data to Jaeger via a collector.
- Start Jaeger and configure it to collect and store trace data.
- Use the Jaeger user interface to view and analyze trace data.

Let’s have a look at an example application

- Two Docker containers
- service1 has an endpoint /, which calles service2 to work
- service2 has just an endpoint /

docker-compose.yml

```
services:
  service2:
    build:
      context: ./service2
    container_name: service2-without
    expose:
      - "5000"
    ports:
      - "5101:5000"

  service1:
    build:
      context: ./service1
    container_name: service1-without
    environment:
```

docker-compose.yml

```
    SERVICE2_URL: http://service2:5000/
  depends_on:
    - service2
  ports:
    - "5100:5000"
```

service1/app.py

```
import os

import requests
from flask import Flask, jsonify

SERVICE2_URL = os.environ.get("SERVICE2_URL", "http://service2:5000/↘
    ")

app = Flask(__name__)


@app.get("/")
def index():
    """Call service2 and wrap its response."""
    upstream = requests.get(SERVICE2_URL, timeout=5)
```

service1/app.py

```
    upstream.raise_for_status()
    return jsonify(service="service1", upstream=upstream.json())


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

docker-compose.yml with Jaeger

```
services:

  service2:
    build:
      context: ./service2
    container_name: service2
    environment:
      JAEGER_AGENT_HOST: jaeger
      JAEGER_AGENT_PORT: "6831"
    expose:
      - "5000"
    ports:
      - "5001:5000"

  service1:
```

docker-compose.yml with Jaeger

```
    build:
      context: ./service1
    container_name: service1
    environment:
      SERVICE2_URL: http://service2:5000/
      JAEGER_AGENT_HOST: jaeger
      JAEGER_AGENT_PORT: "6831"
    depends_on:
      - service2
    ports:
      - "5000:5000"

  jaeger:
    image: jaegertracing/all-in-one:1.54
    container_name: jaeger
```

docker-compose.yml with Jaeger

```
environment:
  COLLECTOR_ZIPKIN_HOST_PORT: ":9411"
ports:
  - "16686:16686" # Jaeger UI
  - "14268:14268" # Collector HTTP
  - "14250:14250" # Collector gRPC
  - "4317:4317" # OTLP gRPC
  - "4318:4318" # OTLP HTTP
  - "6831:6831/udp" # Agent UDP
  - "6832:6832/udp" # Agent UDP
```

service1/app.py with manual instrumentation

```
import os

import requests
from flask import Flask, jsonify, request
from opentelemetry import propagate, trace
from opentelemetry.exporter.jaeger.thrift import JaegerExporter
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor

SERVICE2_URL = os.environ.get("SERVICE2_URL", "http://service2:5000/↘
    ")

app = Flask(__name__)
```

service1/app.py with manual instrumentation

```
_tracing_configured = False

if not _tracing_configured:
    resource = Resource.create({"service.name": "service1"})
    provider = TracerProvider(resource=resource)
    trace.set_tracer_provider(provider)
    jaeger_exporter = JaegerExporter(
        agent_host_name=os.environ.get("JAEGER_AGENT_HOST", "jaeger"
            ),
        agent_port=int(os.environ.get("JAEGER_AGENT_PORT", "6831")),
    )
    provider.add_span_processor(BatchSpanProcessor(jaeger_exporter))
    _tracing_configured = True

tracer = trace.get_tracer("service1")
```

service1/app.py with manual instrumentation

```
@app.get("/")
def index():
    """Fetch a response from service2 and wrap it with service1 ↘
       metadata."""
    context = propagate.extract(request.headers)
    with tracer.start_as_current_span("service1.index", context=↘
       context):
        carrier = {}
        propagate.inject(carrier)
        with tracer.start_as_current_span("service1.call_service2"):
            upstream = requests.get(SERVICE2_URL, timeout=5, headers↘
               =carrier)
            upstream.raise_for_status()
```

service1/app.py with manual instrumentation

```
        return jsonify(service="service1", upstream=upstream.json())


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

service1/app.py with automatic instrumentation

```
import os

import requests
from flask import Flask, jsonify
from opentelemetry import trace
from opentelemetry.exporter.jaeger.thrift import JaegerExporter
from opentelemetry.instrumentation.flask import FlaskInstrumentor
from opentelemetry.instrumentation.requests import ↘
    RequestsInstrumentor
from opentelemetry.sdk.resources import Resource
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor

SERVICE2_URL = os.environ.get("SERVICE2_URL", "http://service2:5000/↘
    ")
```

service1/app.py with automatic instrumentation

```
app = Flask(__name__)

_tracing_configured = False

if not _tracing_configured:
    resource = Resource.create({"service.name": "service1"})
    provider = TracerProvider(resource=resource)
    trace.set_tracer_provider(provider)
    jaeger_exporter = JaegerExporter(
        agent_host_name=os.environ.get("JAEGER_AGENT_HOST", "jaeger"),
        agent_port=int(os.environ.get("JAEGER_AGENT_PORT", "6831")),
    )
    provider.add_span_processor(BatchSpanProcessor(jaeger_exporter))
```

service1/app.py with automatic instrumentation

```
    FlaskInstrumentor().instrument_app(app)
    RequestsInstrumentor().instrument()
    _tracing_configured = True


@app.get("/")
def index():
    """Fetch a response from service2 and wrap it with service1 ↘
       metadata."""
    upstream = requests.get(SERVICE2_URL, timeout=5)
    upstream.raise_for_status()
    return jsonify(service="service1", upstream=upstream.json())


if __name__ == "__main__":
```

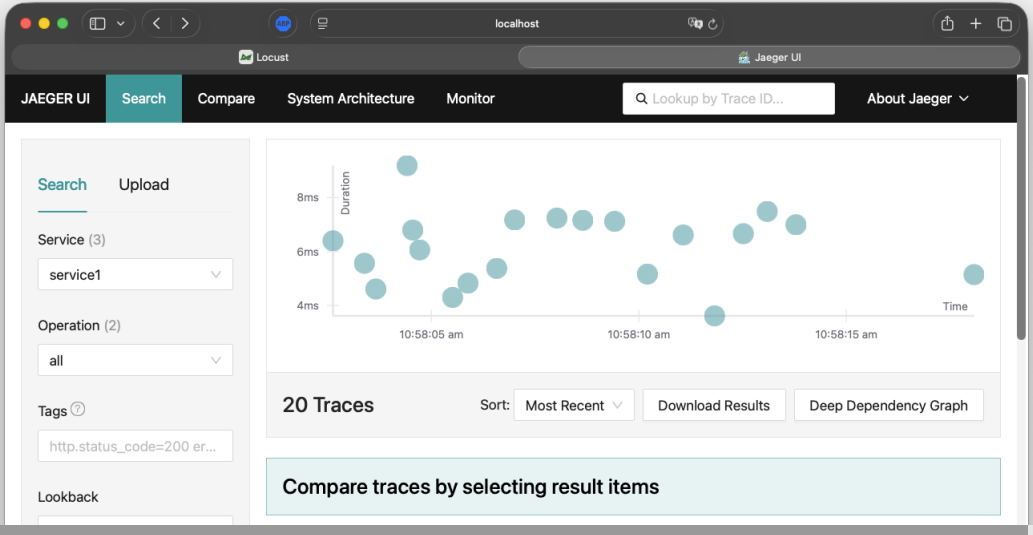
service1/app.py with automatic instrumentation

```
app.run(host="0.0.0.0", port=5000)
```

Exercise: collect traces

- Download https://github.com/ajanes/tutorial_sfscon
- Open a terminal and change path to example2/automatic
- `python -m venv .venv`
- `source ./.venv/bin/activate.fish`
- `pip install -r requirements.txt`
- `docker-compose up -build`
- `locust`
- Navigate to http://localhost:8089
- Run locust for a bit stress testing http://localhost:5000
- Navigate to http://localhost:16686

http://localhost:16686



http://localhost:16686