Lab exercise: Scenarios

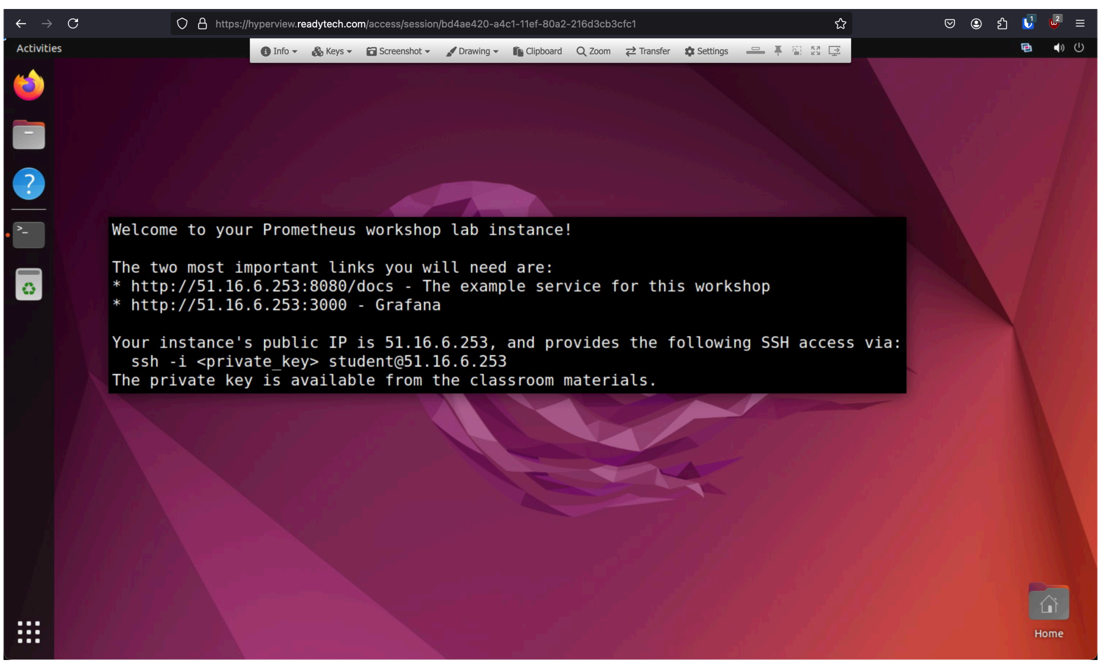
Overview

In this lab you will be presented with several common failure scenarios. Your mission, should you choose to accept it, is to figure out what the failure mode is in each scenario and back your hypotheses with lovely, lovely data.

Assignments

1. Reacquaint yourself with the lab

Your lab is still available at the same IP, and the remote machine is viewable via the classroom application (with the username/password student/student). Your IP address is visible on the lab desktop:



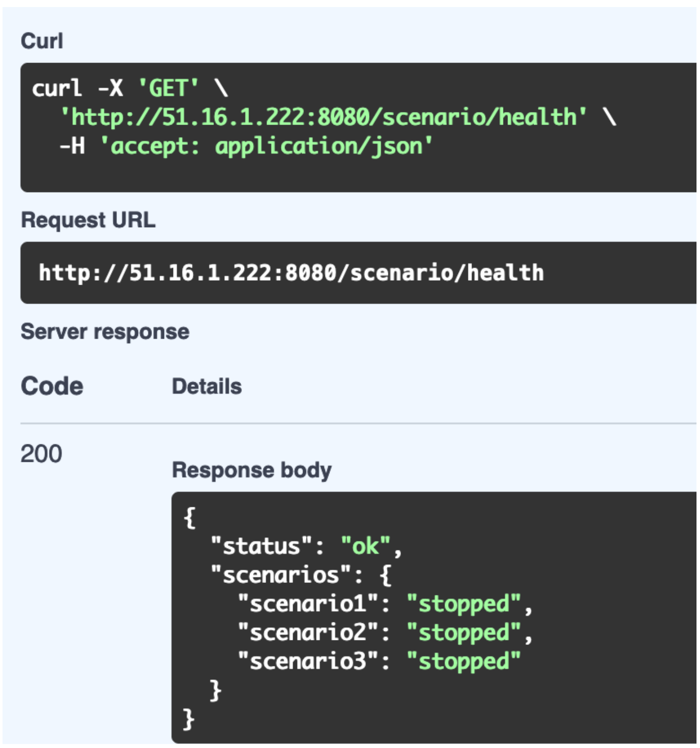
You'll likely need the following two URLs:

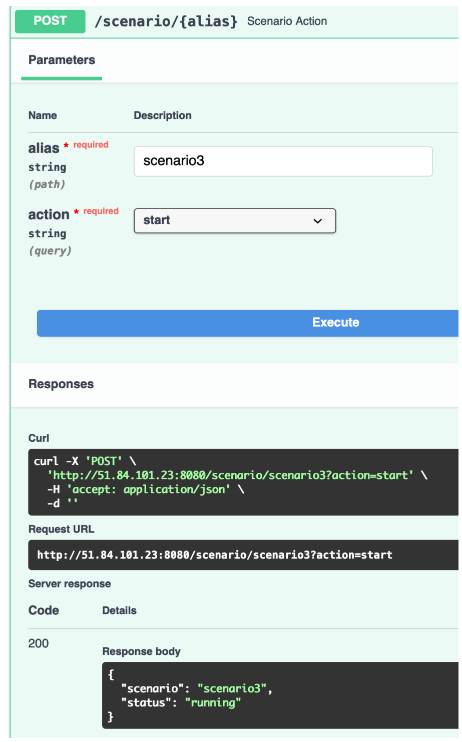
- The Python sample service is available at http://<lab_ip>:8080/docs
- Grafana is available at http://<lab_ip>:3000

2. Scenario runner

The Python-based sample service demonstrates the failure scenarios via a quasi-convenient REST interface. As before, you can use them directly from the FastAPI docs, just don't forget to click on "Try it out" to enable the test UI.

The scenarios are simply named `scenario1` through `scenario3`, and the runner status can be seen via the `/scenario/health` endpoint; the `/scenario/{alias}` endpoint can be used to control a particular scenario using `action="start"` or `action="stop"` accordingly:





3. It's on!

Go through each of the three scenarios in turn:

- Start the scenario
- **Explore the metrics** to figure out what the failure mode is. With no logs or customer complaints this isn't easy, so don't hesitate to look in the solution section below!
- **Find appropriate metrics** in Grafana and chart them to prove/disprove your hypotheses
- Stop the scenario

Stretch goals:

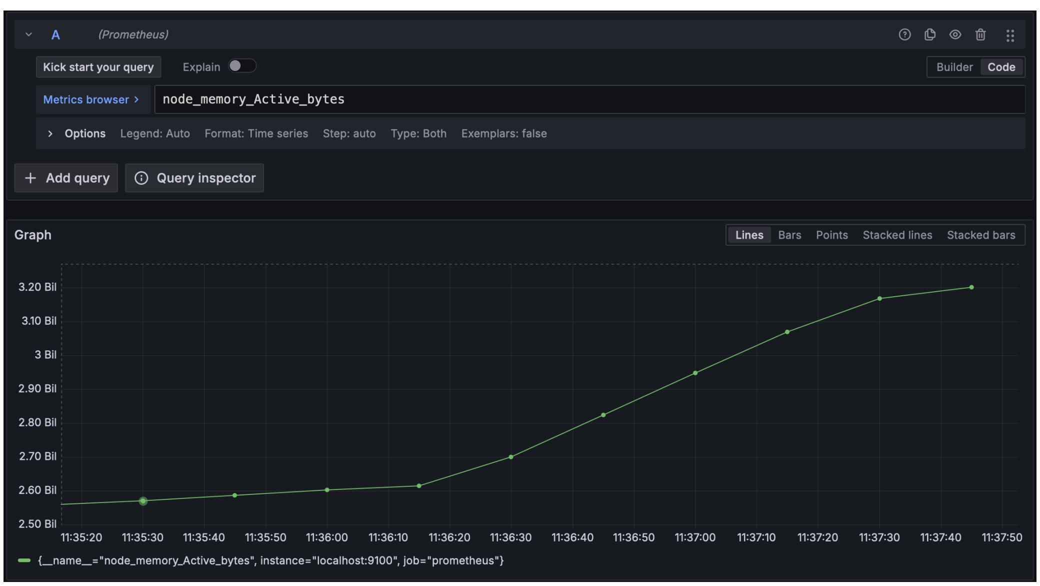
- Build a **Grafana Dashboard** that can showcase the problem to an operator/on-call
- Define an **alert rule** that would trigger if the problem became severe

3.1. Solution: scenario 1

The application is slowly and sadly leaking memory.

You can see this using the `namedprocess_namegroup_memory_bytes` metric, which should show a steady growth under the labels `groupname="python"` with `memtype="resident"` or `memtype="virtual"`:

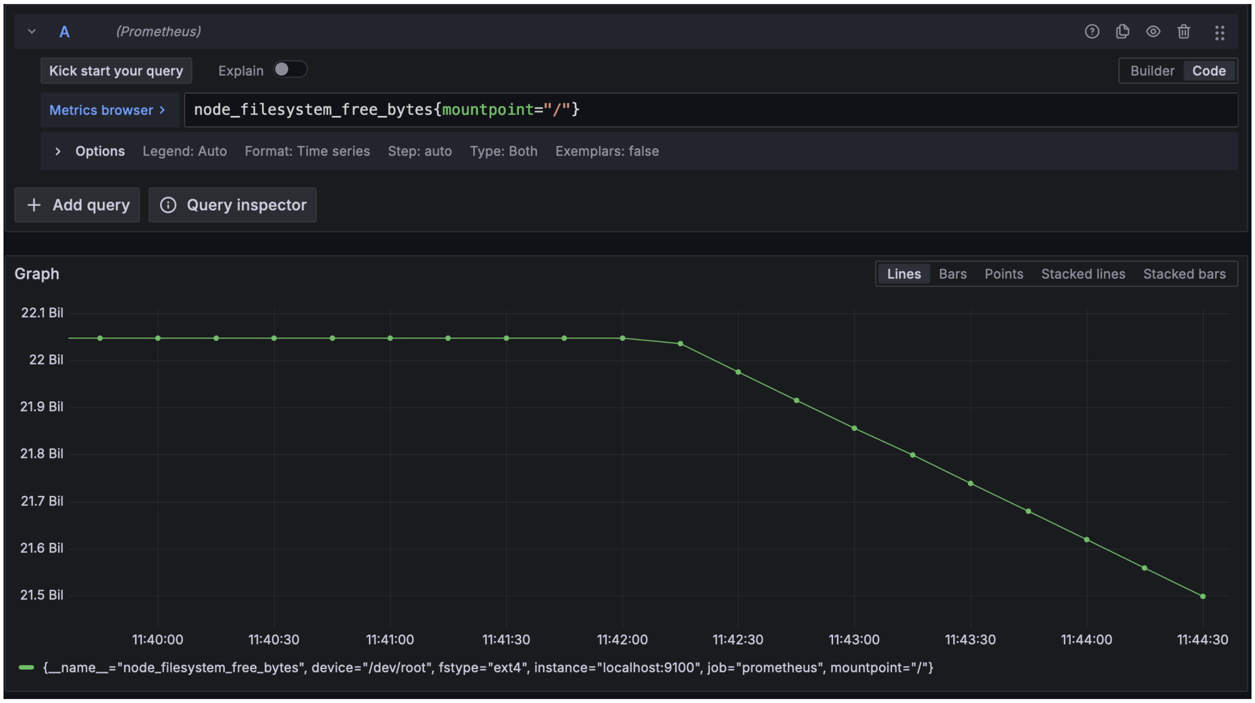
Node-level memory metrics (`node_memory_Active_bytes`, `node_memory_Free_bytes` and `node_memory_Available_bytes`) aren't precise but can still reveal the problem:



3.1. Solution: scenario 2

The application is writing data to disk and will continue to do so until it runs out of space.

You can see this using the `node_filesystem_avail_bytes` or `node_filesystem_free_bytes` metrics with the `mountpoint="/"` label.

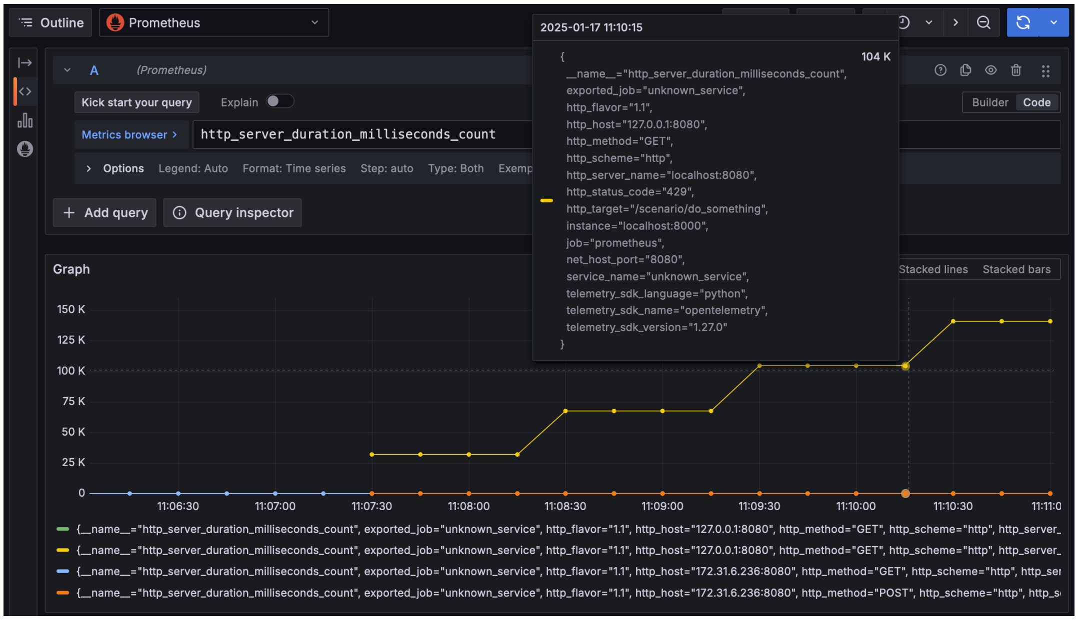


3.1. Solution: scenario 3

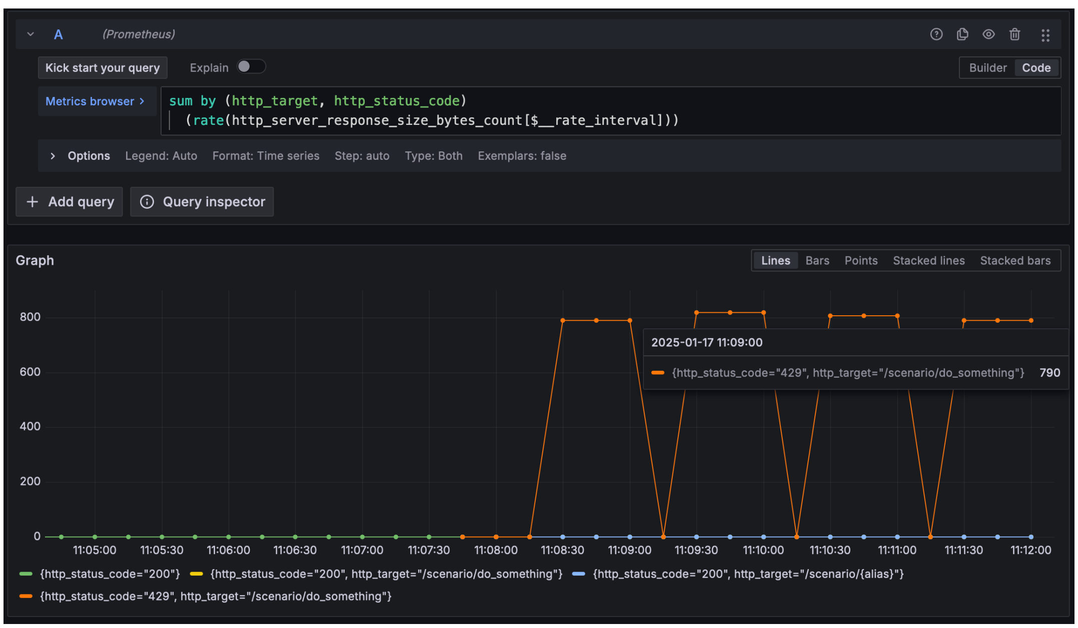
A client is hitting the service's `/do_something` endpoint faster than allowed and is hitting a rate limiter.

This should be easy to show, but picking the right metric can be tricky. We'd ideally like a counter of processed requests, however there aren't any obvious "request/response count" metrics (only an "active requests" gauge, which won't necessarily help). This trick is in realizing that FastAPI instruments both request *durations* and response *sizes*; in other words it maintains histograms for both – meaning you get not only the histogram buckets, but also a counter!

Either `http_server_duration_milliseconds_count` *or* `http_server_response_size_bytes_count` show a steep rise under the `http_status_code="429"` (Too Many Requests).



A different way of approaching this is the visual approach of charting the request rate *by status code*. Since requests can potentially target many handlers, this would require by a rate and an aggregation function:



References

- Workshop presentation in class materials
- SSH private key for connecting to your lab is available in the class materials
- The full sources for everything can always be found here: https://github.com/holograph/prometheus-workshop-service-python