Prometheus

Prometheus is the de-facto standard for monitoring in the cloud age. In a classic monitoring scenario the monitoring system

Prometheus brings with it a *modular* approach. It employs *service discovery* to find the IP addresses and ports of the servic
a *time-series database* and can be queried with a specialized query language called PromQL.

Running Prometheus

The easiest way to run Prometheus is using a container. As a first step you need to create a configuration file. Let's say, you

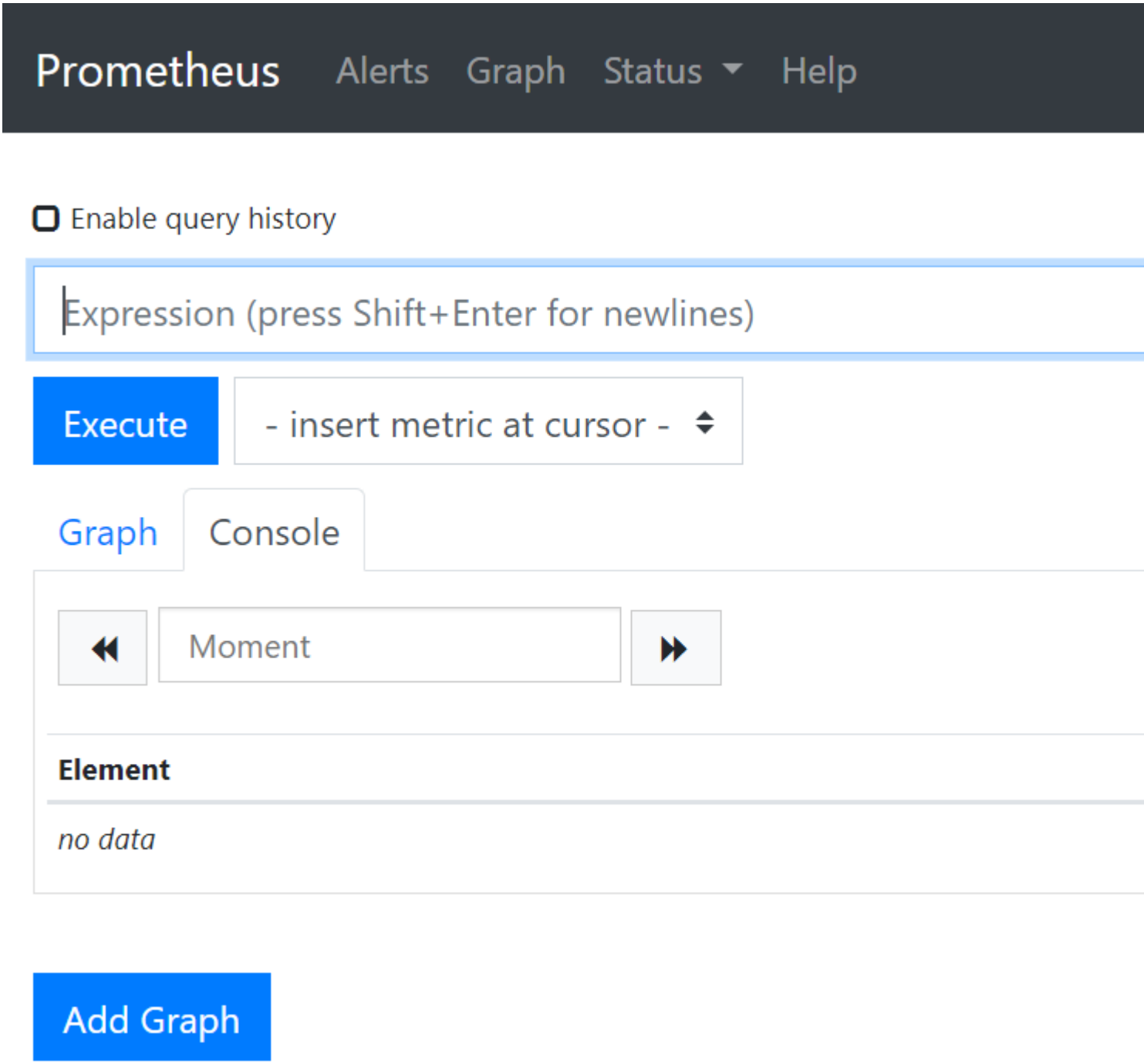
```
global:
  scrape_interval: 15s
scrape_configs:
  - job_name: 'prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:9090']
```

This configuration file will tell Prometheus to gather its own metrics. Having this configuration file, we can launch Prometheu

```
docker run \
    -d \
    -p 9090:9090 \
    -v /srv/prometheus.yml:/etc/prometheus/prometheus.yml \
    prom/prometheus
```

> **Warning**
>
> This will run Prometheus on your host machine on port 9000. Prometheus itself does not have any authentication. Make sure to

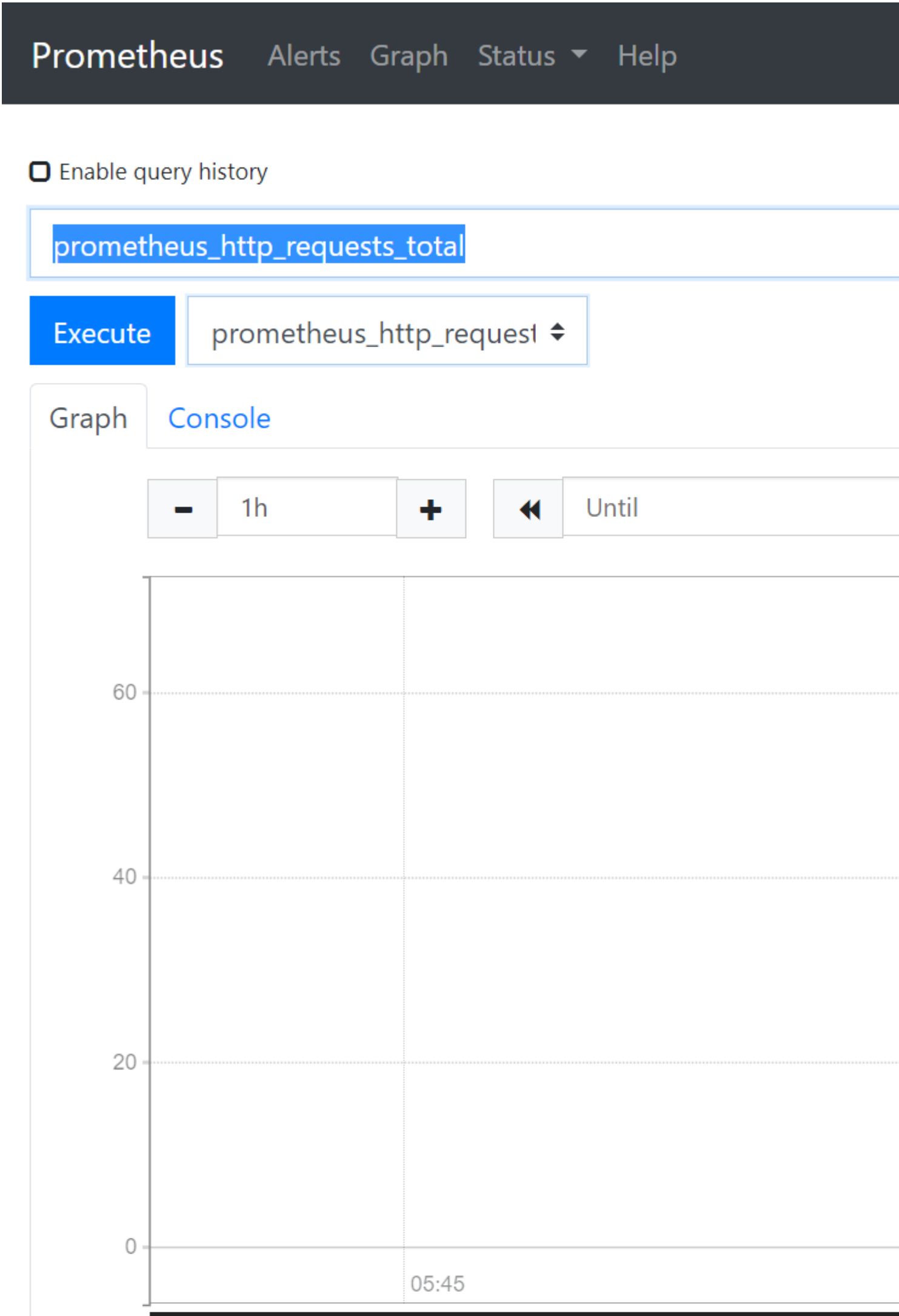
After running the previous command you can now enter the IP address of your server into your browser and access Promet



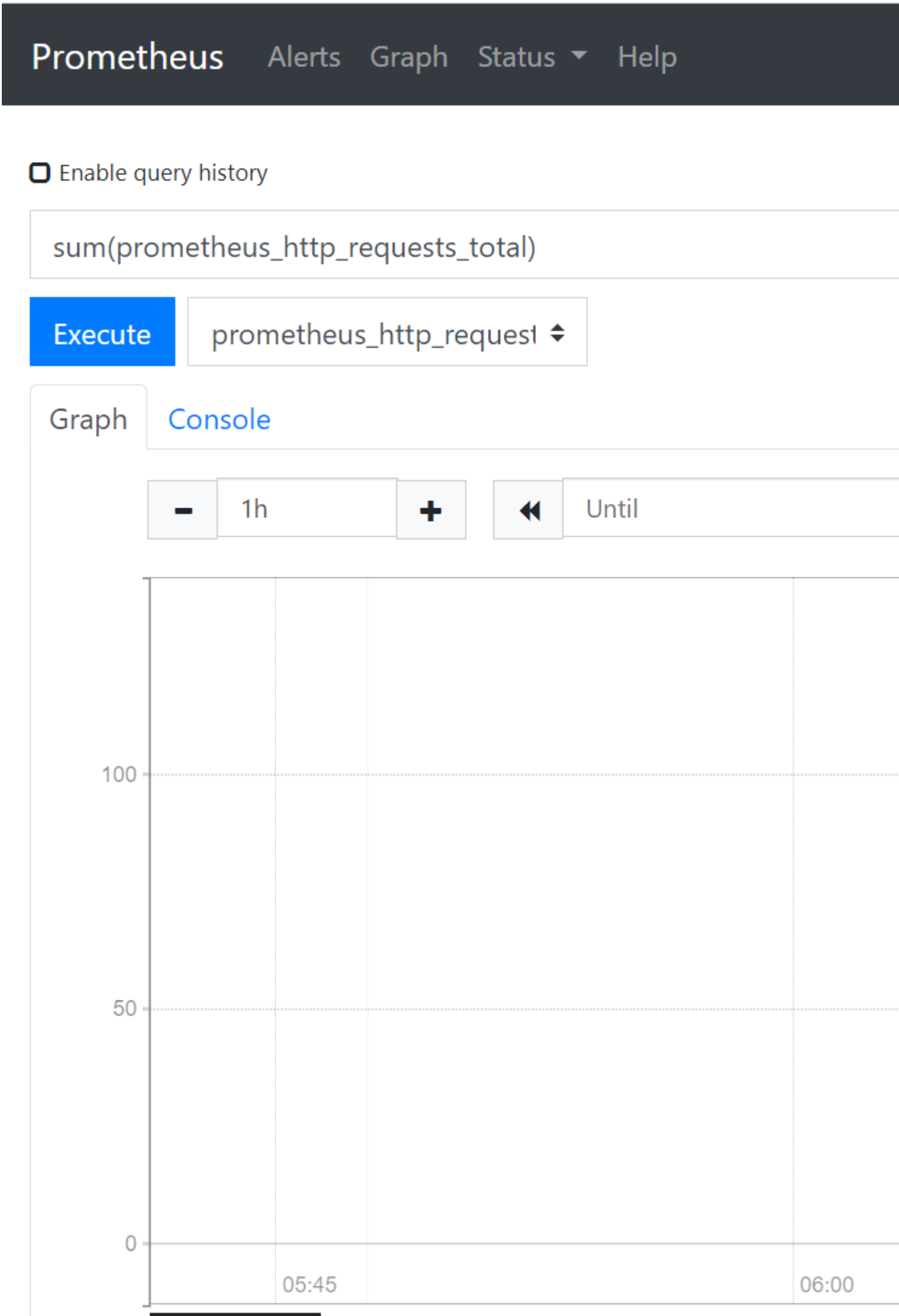
Querying Prometheus

Prometheus is running and gathering its own metrics. You can check the successful metrics scraping by navigating to Statu

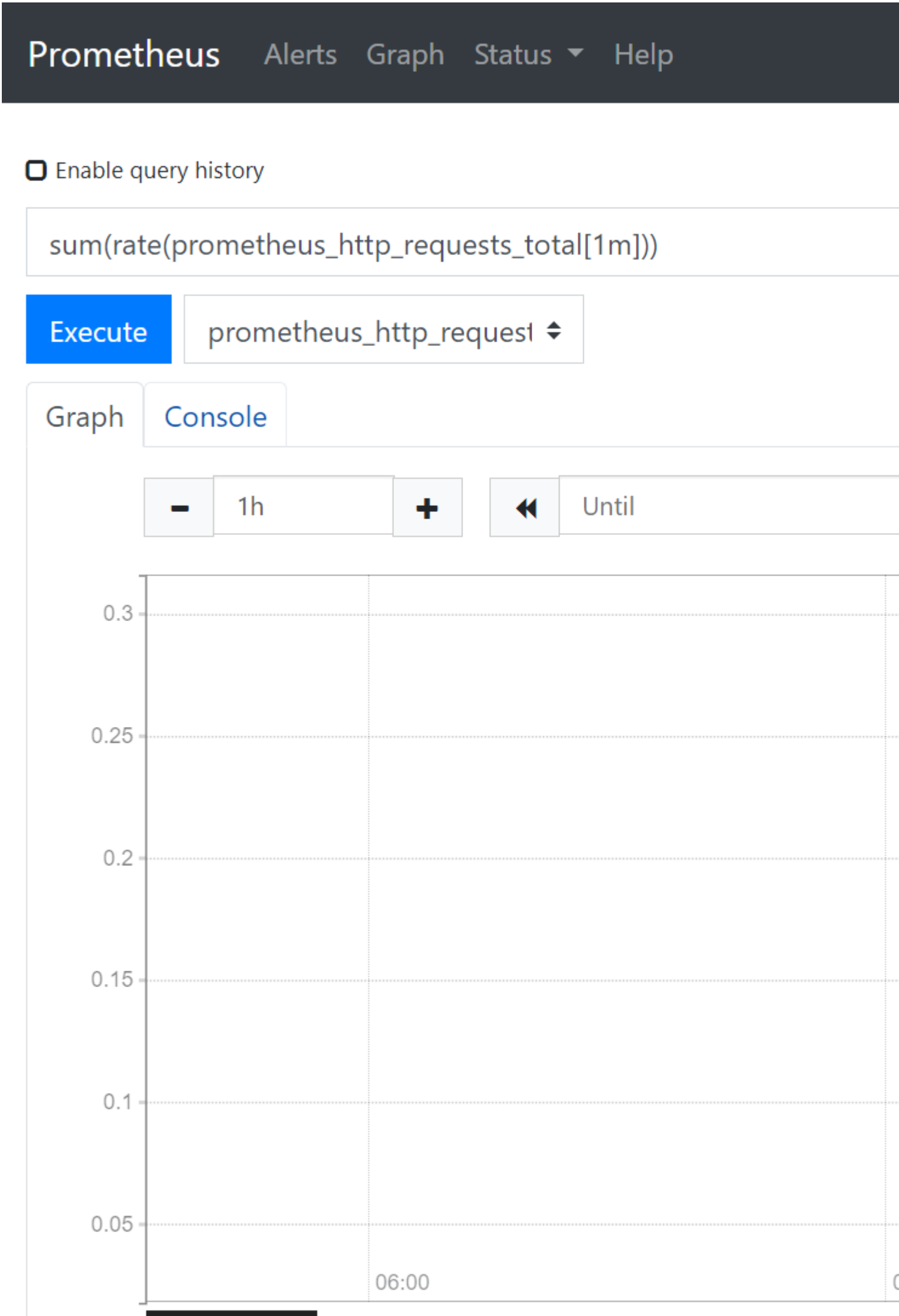
`prometheus_http_requests_total` metric to show the HTTP requests Prometheus has received on each endpoint:


Prometheus
Alerts
Graph
Status
Help
Enable query history
prometheus_http_requests_total
Execute
prometheus_http_request
Graph
Console
1h
Until
60
40
20
0
05:45

You can also summarize the HTTP requests by using the `sum(prometheus_http_requests_total)` formula:


Prometheus
Alerts
Graph
Status
Help
Enable query history
sum(prometheus_http_requests_total)
Execute
prometheus_http_request
Graph
Console
1h
Until
100
50
0
05:45
06:00

As you can see, this graph is ever-growing. This is because the graph type is a `counter`. Counters always increase. Let's g


Prometheus
Alerts
Graph
Status
Help
Enable query history
sum(rate(prometheus_http_requests_total[1m]))
Execute
prometheus_http_request
Graph
Console
1h
Until
0.3
0.25
0.2
0.15
0.1
0.05
06:00

You can read more about the available query functions in PromQL in the official documentation.

Configuring a node exporter

Prometheus in and of itself does not do anything. It needs to read from an exporter. The easiest and probably most useful e

Let's start by launching the node exporter on the same host as Prometheus:

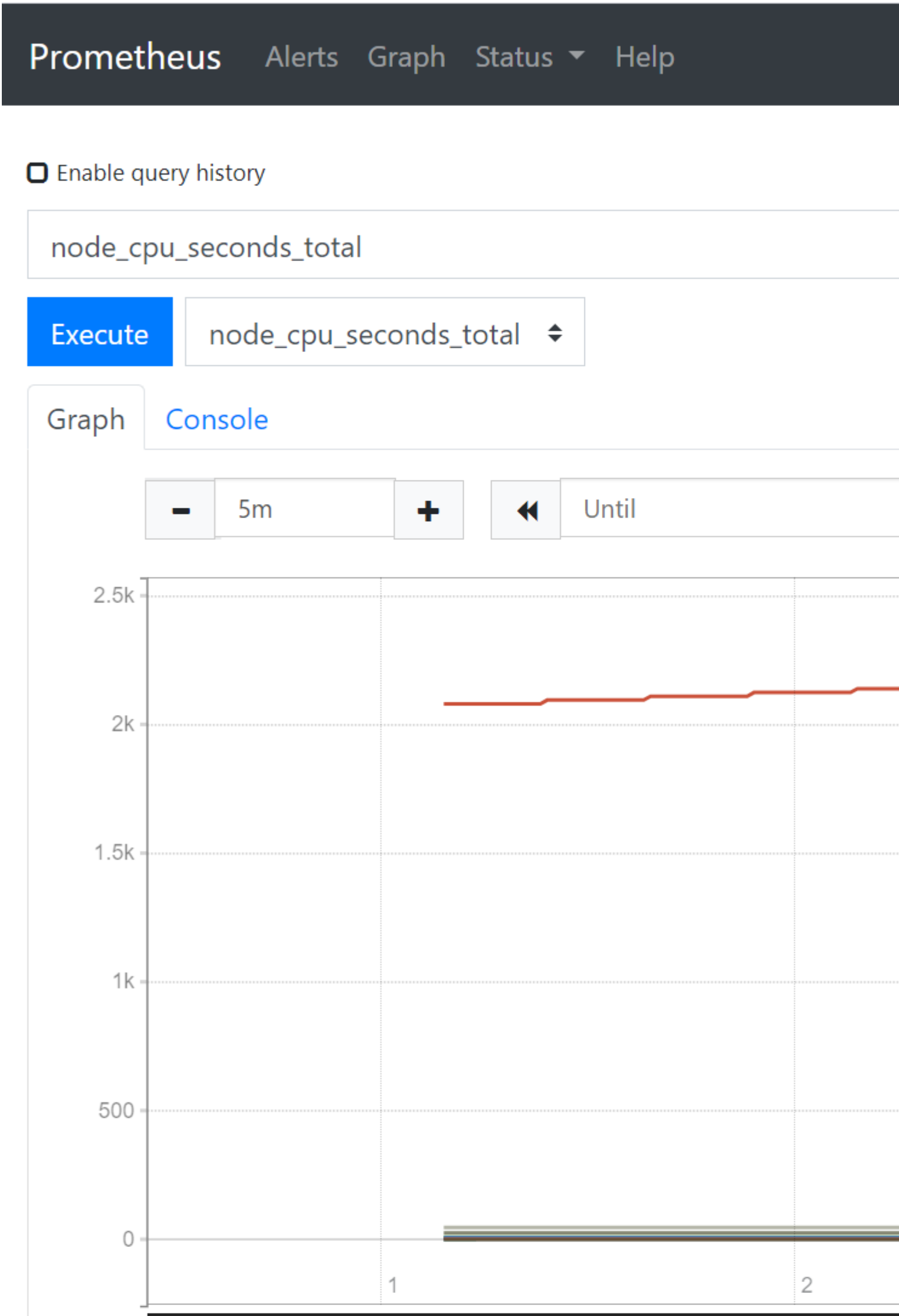
```
docker run -d \
  --net="host" \
  --pid="host" \
  -v "/:/host:ro,rslave" \
  quay.io/prometheus/node-exporter \
  --path.rootfs=/host
```

In the next step you can configure the *public IP* of your Prometheus server to be scraped for metrics in your `prometheus.ym`

```
global:
  scrape_interval: 15s
scrape_configs:
  - job_name: 'prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:9090']
  - job_name: Monitoring Server Node Exporter
    static_configs:
      - targets:
          - '1.2.3.4:9100'
```

> **Note**
>
> The `localhost:9090` and the `1.2.3.4:9100` are *not* the same! Remember, Prometheus runs in a container and has its own ne

Let's restart Prometheus and then check Status → Targets to see if the node exporter shows up as a target. We can also lo


Prometheus
Alerts
Graph
Status
Help
Enable query history
node_cpu_seconds_total
Execute
node_cpu_seconds_total
Graph
Console
5m
Until
2.5k
2k
1.5k
1k
500
0
1
2

As you can see, the CPU query results in several counters. This is the result of how CPU metrics gathering works. These se
applications and our kernel. This can be done by filtering out the `idle` state and then comparing it to the whole:

```
sum(rate(node_cpu_seconds_total{mode!="idle"}[1m])) /
sum(rate(node_cpu_seconds_total[1m]))
```

Dynamic scraping

There is a wide range of service discovery options available. In our case we will implement the file SD. The file service disc

Let's extend our configuration my the file SD configuration:

```
global:
  scrape_interval: 15s
scrape_configs:
  - job_name: 'prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:9090']
  - job_name: Monitoring Server Node Exporter
    static_configs:
      - targets:
          - '1.2.3.4:9100'
  - job_name: Custom
    file_sd_config:
      - files:
          - /custom_servers.json
        refresh_interval: 10s
```

To make this work you will have to mount an additional volume for Prometheus:

```
docker run \
    -d \
    -p 9090:9090 \
    -v /srv/custom_servers.json:/custom_servers.json \
    -v /srv/prometheus.yml:/etc/prometheus/prometheus.yml \
    prom/prometheus
```

Finally, let's place the following file in `/srv/custom_servers.json` on the host:

```
[
  {
    "targets": [ "1.2.3.4:9100" ]
  }
]
```

You can enter the IP address of a second server in this file to enable scraping and restart Prometheus this one last time to e
which you should then be able to query.

Let's repeat the PromQL query from the example above to get the CPU usage. You will see that it won't give you two graphs

```
sum by (instance) (rate(node_cpu_seconds_total{mode!="idle"}[1m])) /
sum by (instance) (rate(node_cpu_seconds_total[1m]))
```

The only thing left to do is to write an automatic cron job that fetches the IP address list from the cloud provider (Exoscale ir

Tip

Prometheus would be able to do the alerting behavior required for the project work itself without Grafana using the Alert Manage