

# Monitoring with Prometheus

Estimated Time: 30 minutes

Welcome to the **Monitoring with Prometheus** lab. In this lab, you will become familiar with using Prometheus to monitor sample servers simulated with node exporter. You will use Prometheus to monitor the target `node_exporter` application that is configured by scraping **metrics** endpoints of the `node_exporter`. You will finish the lab by learning how to instrument a Python Flask application to emit metrics and deploy that application so that Prometheus can monitor it.

## Learning Objectives

After completing this exercise, you should be able to:

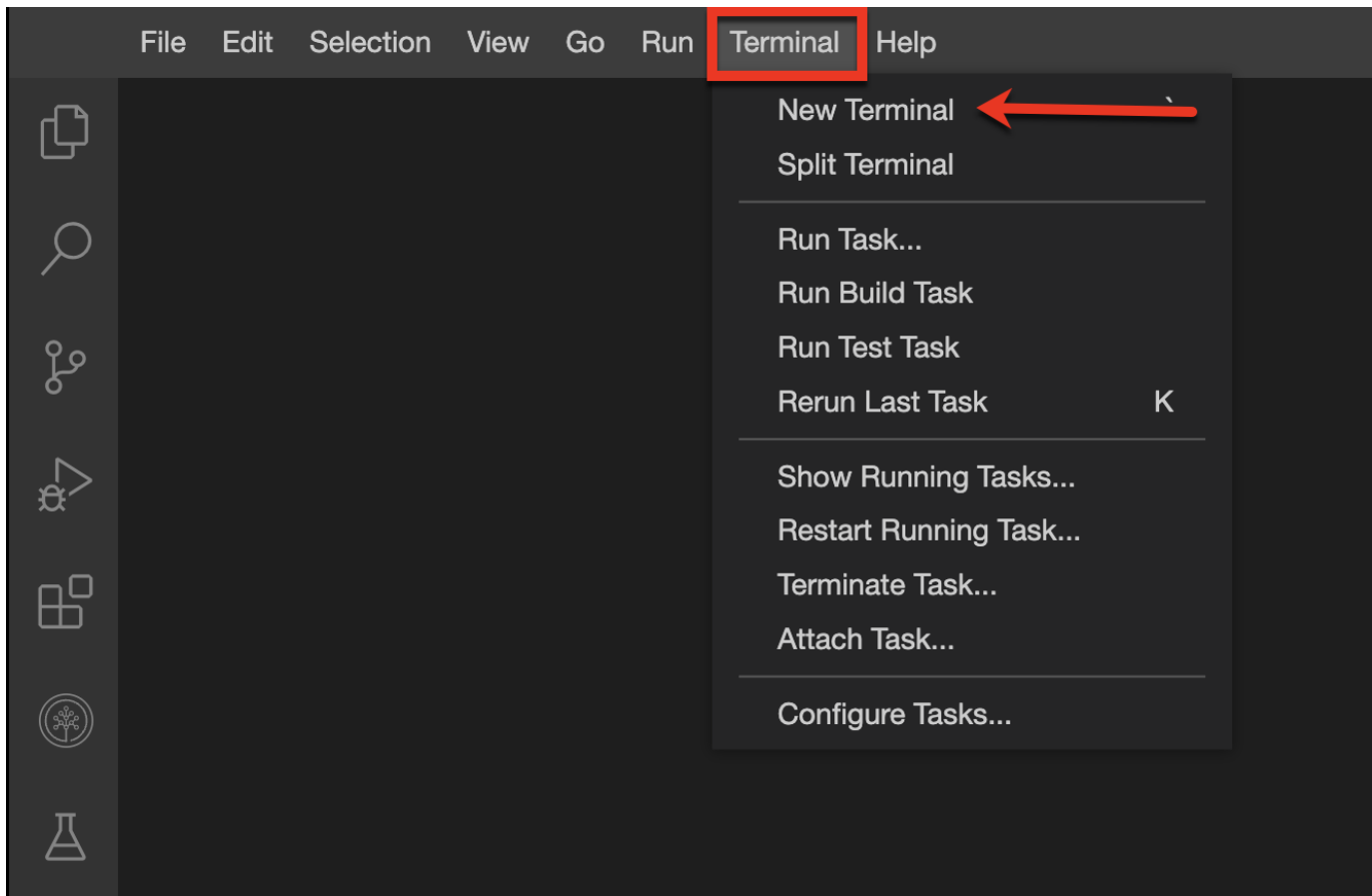
- Configure the targets for Prometheus to monitor
- Create queries to get the metrics about the target
- Determine the status of the targets
- Identify information about the targets and visualize it with graphs
- Instrument a Python Flask application to be monitored by Prometheus

## Prerequisites

This lab uses Docker to run both Prometheus, and special Node Exporters, which will behave like servers that you can monitor. As a prerequisite, you will pull down the `bitnami/prometheus:latest` image and the `bitnami/node-exporter` image from Docker Hub. You will use these images to run Prometheus and create three instances of node exporters to be monitored.

## Your Task

1. To start this lab, you will need a terminal. If a terminal is not already open, you can open one from the top menu. Go to **Terminal** and choose **New Terminal** to open a new terminal window.



2. Next, use the following `docker pull` command to pull down the `bitnami/node-exporter` image from Docker Hub that you will use to simulate three servers being monitored.

```
1. 1
1. docker pull bitnami/node-exporter:latest
```

Copied! Executed!

Your output should look similar to this:

```
theia@theiadosker-rofrano:/home/project$ docker pull bitnami/node-exporter:latest
latest: Pulling from bitnami/node-exporter
1d8866550bdd: Pull complete
8e2055ff5472: Pull complete
Digest: sha256:c306e2c62fa7fee7bf3b24b444dafa75423805fe1cb7aa52bc52af765767c6f1
Status: Downloaded newer image for bitnami/node-exporter:latest
docker.io/bitnami/node-exporter:latest
```

3. Then, pull the Prometheus docker image into your lab environment, by running the following `docker pull` command in the terminal.

```
1. 1
1. docker pull bitnami/prometheus:latest
```

Copied! Executed!

Your output should look similar to this:

```
theia@theiadocker-rofrano:/home/project$ docker pull bitnami/prometheus:latest
latest: Pulling from bitnami/prometheus
1d8866550bdd: Already exists
095e7c5c9312: Pull complete
Digest: sha256:58357c657791c5031ee16429fefef5e50c08e09f4fb50c1a1cce270d11d47901
Status: Downloaded newer image for bitnami/prometheus:latest
docker.io/bitnami/prometheus:latest
```

You are now ready to start the lab.

# Step 1: Start the first node exporter

The first thing you will need is some server nodes to monitor. You will start up three node exporters listening on port 9100 and forwarding to ports 9101, 9102, and 9103, respectively. Each node will need to be started up individually.

In this step, you will create a Docker network for all of the node exporters and Prometheus to communicate on, and start just the first node, 9101, and ensure it is working correctly.

## Your Task

- 1. Start by running the following docker network command to create a network called monitor within which we will run all of the docker containers.

```
1. 1
1. docker network create monitor
```

Copied! Executed!

- 2. Next, run the following docker run command to start a node exporter instance on the monitor network, listening at port 9101 externally and forwarding to port 9100 internally.

```
1. 1
1. docker run -d --name node-exporter1 -p 9101:9100 --network monitor bitnami/node-exporter:latest
```

Copied! Executed!

This will start an instance named node\_exporter1 of node-exporter. The output should look something like this (note: the container id will be different each time):

```
theia@theiadocker-rofrano:/home/project$ docker run -d -p 9101:9100 --name node-exporter1 --
monitor bitnami/node-exporter:latest
a6551f72fdd7dd0a96f28a488ff344a26d184c9a9b493e81a6a2f91c534ef8ee
```

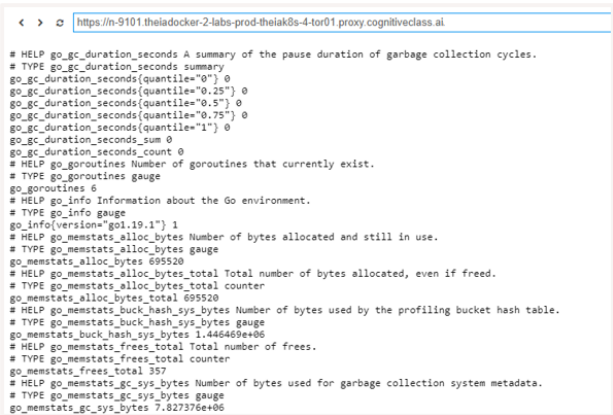
- 3. Next, check if the instance is running by pressing the [Launch Application] button, which will launch the application on port 9101:

Launch Application

- 4. You should see the Node Exporter page open up with a hyperlink to **Metrics**. These are the metrics the Prometheus instance is going to monitor.



- 5. Finally, click the **Metrics** link to see the metrics.



# Step 2: Start two more node exporters

Now that you have one node exporter working, you can start two more so that Prometheus has three nodes to monitor in total. You will do this the same way as you did the first node exporter, except that you will change the external port numbers to 9102 and 9103, respectively.

## Your Task

- 1. In the terminal, run the following commands to start two more instances of node exporter.

```
1. 1
1. docker run -d --name node-exporter2 -p 9102:9100 --network monitor bitnami/node-exporter:latest
```

Copied! Executed!

and

```
1. 1
1. docker run -d --name node-exporter3 -p 9103:9100 --network monitor bitnami/node-exporter:latest
```

Copied! Executed!

2. Now, check if all the instances of node exporter are running by using the `docker ps` command and pipe it through the `grep` command to search for `node-exporter`.

```
1. 1
1. docker ps | grep node-exporter
```

Copied! Executed!

Results

If everything started correctly, you should see output similar to the following coming back from the `docker ps` command:

```
theia@theiadocker-rofrano:/home/project$ docker ps | grep node-exporter
f98cef022805 bitnami/node-exporter:latest "/opt/bitnami/node-e..." 4 seconds ago Up 2 se
0.0.0.0:9103->9100/tcp, :::9103->9100/tcp node-exporter3
48d41798e1be bitnami/node-exporter:latest "/opt/bitnami/node-e..." 18 seconds ago Up 17 s
s 0.0.0.0:9102->9100/tcp, :::9102->9100/tcp node-exporter2
a6551f72fdd7 bitnami/node-exporter:latest "/opt/bitnami/node-e..." 4 minutes ago Up 4 mi
0.0.0.0:9101->9100/tcp, :::9101->9100/tcp node-exporter1
```

You are now ready to configure and run Prometheus.

Step 3: Configure and run Prometheus

Before you can start Prometheus, you need to create a configuration file called `prometheus.yml` to instruct Prometheus on which nodes to monitor.

In this step, you will create a custom configuration file to monitor the three node exporters running internally on the `monitor` network at `node-exporter1:9100`, `node-exporter2:9100`, and `node-exporter3:9100`, respectively. Then you will start Prometheus by passing it the configuration file to use.

Your Task

1. First, use the `touch` command to create a file named `prometheus.yml` in the current directory. This is the file where you will configure the Prometheus to monitor the node exporter instances.

```
1. 1
1. touch /home/project/prometheus.yml
```

Copied! Executed!

2. Next, from **Explorer**, navigate to **Project**, and then select `prometheus.yml` to edit the file, or press the `[Open prometheus.yml in IDE]` button below:

Open prometheus.yml in IDE

3. Then, copy and paste the following configuration contents into the `yml` file and save it:

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
7. 7
8. 8
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
1. # my global config
2. global:
3.   scrape_interval: 15s # Set the scrape interval to every 15 seconds. The default is every 1 minute.
4.
5. scrape_configs:
6.   - job_name: 'node'
7.     static_configs:
8.       - targets: ['node-exporter1:9100']
9.         labels:
10.          group: 'monitoring_node_ex1'
11.       - targets: ['node-exporter2:9100']
12.         labels:
13.          group: 'monitoring_node_ex2'
14.       - targets: ['node-exporter3:9100']
15.         labels:
16.          group: 'monitoring_node_ex3'
```

Copied!

Notice that while you access the node exporters externally on ports `9101`, `9102`, and `9103`, they are internally all listening on port `9100`, which is how Prometheus will communicate them on the `monitor` network.

Take a look at what this file is doing:

- o Globally, you set the `scrape_interval` to 15 seconds instead of the default of 1 minute. This is so that we can see results quicker during the lab, but the 1 minute interval is better for production use.
- o The `scrape_config` section contains all the jobs that Prometheus is going to monitor. These job names have to be unique. You currently have one job called `node`. Later we will add another to monitor a Python application.
- o Within each job, there is a `static_configs` section where you define the targets and define labels for easy identification and analysis. These will show up in the Prometheus UI under the **Targets** tab.
- o The targets you enter here point to the base URL of the service running on each of the nodes. Prometheus will add the suffix `/metrics` and call that endpoint to collect the data to monitor from. (For example, `node-exporter1:9100/metrics`)

You will have an opportunity to create your own Prometheus file to monitor your Python application in the practice exercise.

4. Finally, you can launch the Prometheus monitor in the terminal by executing the following `docker run` command passing the `yml` configuration file as a volume mount with the `-v` parameter.

```
1. 1
2. 2
3. 3
1. docker run -d --name prometheus -p 9090:9090 --network monitor \
2. -v $(pwd)/prometheus.yml:/opt/bitnami/prometheus/conf/prometheus.yml \
3. bitnami/prometheus:latest
```

Copied! Executed!

Note: This Dockerized distribution of Prometheus from Bitnami expects its configuration file to be in the `/opt/bitnami/prometheus/conf/prometheus.yml` file, which is why you are mapping your `prometheus.yml` file to this location. Other distributions may look in other locations. Always check the documentation to be sure of where to mount the configuration file.

Results

You should see just the Prometheus container id returned, indicating that Docker has started Prometheus in the background.

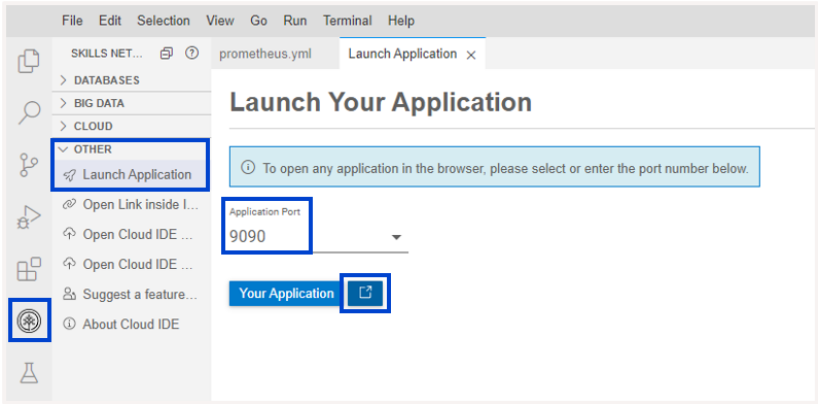
```
theia@theiadocker-rofrano:/home/project$ docker run -d --name prometheus -p 9090:9090
--network monitor \
> -v $(pwd)/prometheus.yml:/opt/bitnami/prometheus/conf/prometheus.yml \
> bitnami/prometheus:latest
9db898b09d03e55b94817b4dc971eaea87e3b02f0ef1517a92273e13fc95941f
theia@theiadocker-rofrano:/home/project$
```

You are now ready to do some monitoring.

Step 4: Open the Prometheus UI

In this step, you will launch the Prometheus web UI in an external browser window and navigate to the page where you start executing queries.

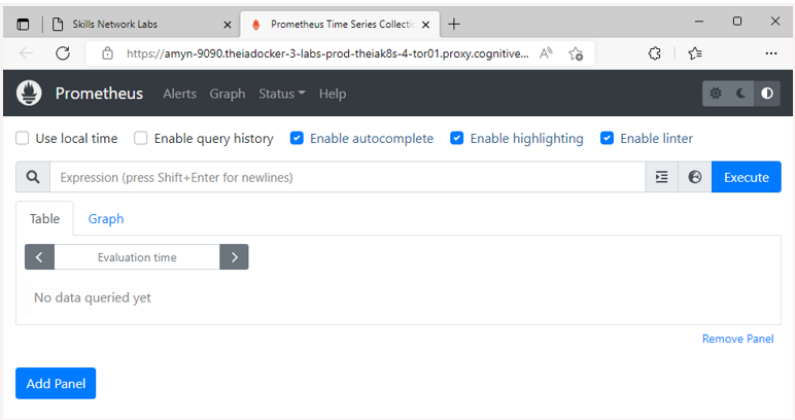
- 1. Open the Prometheus web UI by clicking **Skills Network Toolbox**. Under **Other**, select **Launch Application**, in **Application Port** enter the port number **9090**, and then click the launch URL button.



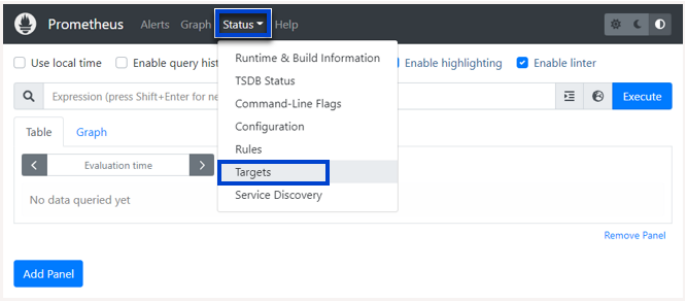
Or click the **Launch Prometheus** button below to launch in an external browser.

[Launch Prometheus](#)

- 2. The Prometheus application UI opens up by default in the graph endpoint.



- 3. Next, in the Prometheus application, click **Status** on the menu and choose **Targets** to see which targets are being monitored.



- 4. View the status of all three node exporters.

# Targets

AllUnhealthyCollapse All

Q

Filter by endpoint or labels

node (3/3 up) 

show less

Endpoint	State	Labels	L
<a href="http://node-exporter3:9100/metrics">http://node-exporter3:9100/metrics</a>	UP	<div>group="monitoring_node_ex3"</div> <div>instance="node-exporter3:9100"</div> <div>job="node"</div>	9
<a href="http://node-exporter1:9100/metrics">http://node-exporter1:9100/metrics</a>	UP	<div>group="monitoring_node_ex1"</div> <div>instance="node-exporter1:9100"</div> <div>job="node"</div>	1
<a href="http://node-exporter2:9100/metrics">http://node-exporter2:9100/metrics</a>	UP	<div>group="monitoring_node_ex2"</div> <div>instance="node-exporter2:9100"</div> <div>job="node"</div>	4

5. Click **Graph** to return to the home page.

# Targets

All Unhealthy Collapse All

Q

Filter by endpoint or labels

node (3/3 up) show less

Endpoint	State	Labels	L
<a href="http://node-exporter3:9100/metrics">http://node-exporter3:9100/metrics</a>	UP	<div>group="monitoring_node_ex3"</div> <div>instance="node-exporter3:9100"</div> <div>job="node"</div>	9
<a href="http://node-exporter1:9100/metrics">http://node-exporter1:9100/metrics</a>	UP	<div>group="monitoring_node_ex1"</div> <div>instance="node-exporter1:9100"</div> <div>job="node"</div>	1
<a href="http://node-exporter2:9100/metrics">http://node-exporter2:9100/metrics</a>	UP	<div>group="monitoring_node_ex2"</div> <div>instance="node-exporter2:9100"</div> <div>job="node"</div>	4

You are now ready to execute queries.

## Step 5: Execute your first query

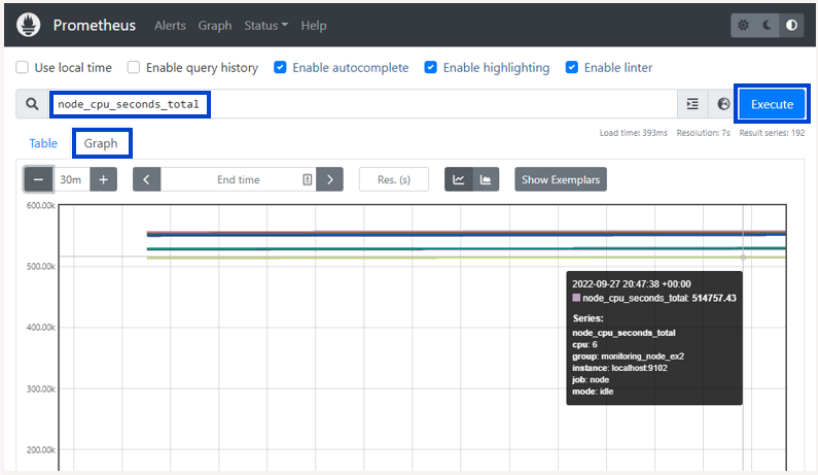
You are now ready to execute your first query. The first query you run will query the nodes for the total CPU seconds. It will show the graph as given in the image. You can observe the details for each instance by hovering the mouse over that instance.

### Your Task

1. Ensure you are on the **Graph** tab, and then *copy-n-paste* the following query and press the blue **Execute** button on the right or press *return* on your keyboard to run it. It will show the graph as given in the image. You can observe the details for each instance by hovering the mouse over that instance.

```
1. 1
1. node_cpu_seconds_total
```

Copied!



2. Next, click **Table** to see the CPU seconds for all the targets in tabular format.

The screenshot shows the Prometheus web interface with the query `node_cpu_seconds_total` entered in the search bar. The 'Table' tab is selected, displaying a table of results. The table has two columns: the query name and the value. The results are as follows:

Query	Value
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex1", instance="node-exporter1:9100", job="node", mode="idle"}	84988.8
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex1", instance="node-exporter1:9100", job="node", mode="iowait"}	571.67
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex1", instance="node-exporter1:9100", job="node", mode="irq"}	0
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex1", instance="node-exporter1:9100", job="node", mode="nice"}	0.31
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex1", instance="node-exporter1:9100", job="node", mode="softirq"}	496.17
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex1", instance="node-exporter1:9100", job="node", mode="steal"}	96.55
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex1", instance="node-exporter1:9100", job="node", mode="system"}	5389.69
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex1", instance="node-exporter1:9100", job="node", mode="user"}	10347.91
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="idle"}	84995.91
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="iowait"}	571.67
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="irq"}	0
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="nice"}	0.31

3. Now, filter the query to get the details for only one instance node-exporter2 using the following query.

```
1. 1
1. node_cpu_seconds_total{instance="node-exporter2:9100"}
```


The screenshot shows the Prometheus web interface with the query `node_cpu_seconds_total{instance="node-exporter2:9100"}` entered in the search bar. The 'Table' tab is selected, displaying a table of results. The table has two columns: the query name and the value. The results are as follows:

Query	Value
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="idle"}	84754.41
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="iowait"}	571.42
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="irq"}	0
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="nice"}	0.31
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="softirq"}	495.03
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="steal"}	96.39
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="system"}	5373.66
node_cpu_seconds_total{cpu="0", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="user"}	10327.11
node_cpu_seconds_total{cpu="1", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="idle"}	88620.25
node_cpu_seconds_total{cpu="1", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="iowait"}	248.13
node_cpu_seconds_total{cpu="1", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="irq"}	0
node_cpu_seconds_total{cpu="1", group="monitoring_node_ex2", instance="node-exporter2:9100", job="node", mode="nice"}	0.9

4. Finally, query for the connections each node has using this query.

```
1. 1
1. node_ipvs_connections_total
```

Copied!

 Prometheus Alerts Graph Status ▾ Help

☐ Use local time

☐ Enable query history

☒ Enable autocomplete

☒ Enable highlighting

☒ Enable linter

Q

node\_ipvs\_connections\_total

⋮

🔄

Execute

Table

Graph

Load time: 716ms

Resolution: 7s

Result series: 3

<

Evaluation time

>

node_ipvs_connections_total{group="monitoring_node_ex1", instance="node-exporter1:9100", job="node"}	0
node_ipvs_connections_total{group="monitoring_node_ex2", instance="node-exporter2:9100", job="node"}	0
node_ipvs_connections_total{group="monitoring_node_ex3", instance="node-exporter3:9100", job="node"}	0

Remove Panel

Step 6: Stop and observe

In this step, we will stop one of the node exporter instances and see how that is reflected in the Prometheus console.

Your Task

1. Stop the node-exporter1 instance by running the following docker stop command and then switch back to the old terminal in which Prometheus is running.

```
1. 1
1. docker stop node-exporter1
```

Copied!

Executed!

2. Now go back to the Prometheus UI on your browser and check the targets by selecting the menu item Status -> Targets.

Launch Prometheus Targets

Results

You should now see that one of the node exporters that are being monitored is down. The nodes might not be displayed in the same order, but the node which is should be node-exporter1, the node that you stopped.

Note: You configured Prometheus to scrape every 15 seconds, so you may have to wait that long and press refresh on your browser to see the status change.



# Targets

AllUnhealthyCollapse All

Q

Filter by endpoint or labels

node (2/3 up) 

show less

Endpoint	State	Labels	Last
<a href="http://node-exporter1:9100/metrics">http://node-exporter1:9100/metrics</a>	DOWN	<div>group="monitoring_node_ex1"</div> <div>instance="node-exporter1:9100"</div> <div>job="node"</div>	11.1s
<a href="http://node-exporter2:9100/metrics">http://node-exporter2:9100/metrics</a>	UP	<div>group="monitoring_node_ex2"</div> <div>instance="node-exporter2:9100"</div> <div>job="node"</div>	4.20
<a href="http://node-exporter3:9100/metrics">http://node-exporter3:9100/metrics</a>	UP	<div>group="monitoring_node_ex3"</div> <div>instance="node-exporter3:9100"</div> <div>job="node"</div>	14.8

## Step 7: Enable your application

Monitoring node exporters is fine for a demonstration, but you are a software engineer. You need to know how to enable your applications to be monitored by Prometheus. There is no magic here. Metrics do not simply appear out of nowhere. You must instrument your application to emit metrics on an endpoint called `/metrics` in order for Prometheus to be able to monitor your application.

Luckily there is a Python package called **Prometheus Flask exporter** for Prometheus that will do this for you. In this step, you will create a simple Python Flask application and enable a metrics endpoint so that you can monitor it.

### Your Task

Below is a code for a Python Flask server with three end points, `/`, `/home`, and `/contact`. The code uses the package `prometheus_flask_exporter` for generating metrics for Prometheus to monitor.

1. First, create a file named `pythonserver.py` in the `/home/project` folder. Press the button below and answer **Create the file** when prompted.

⊗

File not found: /home/project/pythonserver.py

×

Create the file

Open `pythonserver.py` in IDE

2. Then, paste the following code content into it:

1. 1

2. 2

3. 3

4. 4

5. 5

6. 6

7. 7

8. 8

```
9. 9
10. 10
11. 11
12. 12
13. 13
14. 14
15. 15
16. 16
17. 17
18. 18
19. 19
20. 20
21. 21
1. from prometheus_flask_exporter import PrometheusMetrics
2. from flask import Flask
3.
4. app = Flask(__name__)
5. metrics = PrometheusMetrics.for_app_factory()
6. metrics.init_app(app)
7.
8. @app.route('/')
9. def root():
10.     return 'Hello from root!'
11.
12. @app.route('/home')
13. def home():
14.     return 'Hello from home!'
15.
16. @app.route('/contact')
17. def contact():
18.     return 'Contact us!'
19.
20. if __name__ == '__main__':
21.     app.run(host="0.0.0.0", port=8080)
```

Copied!

Notice that you only had to import the `PrometheusMetrics` class from the `prometheus_flask_exporter` package and add two lines of code to instantiate a `PrometheusMetrics.for_app_factory()` as `metrics`, and call `metrics.init_app(app)` to initialize it. That is it! Three total lines of code, and you have Prometheus support!

3. Next, you need to deploy this code on the same docker network as Prometheus. To do this, create a file named `Dockerfile` in the `/home/project` folder:

Open **Dockerfile** in IDE

4. Paste the following contents into `Dockerfile` and save it:

```
1. 1
2. 2
3. 3
4. 4
5. 5
6. 6
1. FROM python:3.9-slim
2. RUN pip install Flask prometheus-flask-exporter
3. WORKDIR /app
4. COPY pythonserver.py .
5. EXPOSE 8080
6. CMD ["python", "pythonserver.py"]
```

Copied!

5. Now, use the `docker build` command to build a Docker image for the service (*Note: You can safely ignore any red output from the docker build command. It just warns about running pip as root*):

```
1. 1
1. docker build -t pythonserver .
```

Copied!

Executed!

6. Finally, run the `pythonserver` Docker container on the `monitor` network exposing port `8080` so that Prometheus will have access to it:

```
1. 1
1. docker run -d --name pythonserver -p 8081:8080 --network monitor pythonserver
```

Copied!

Executed!

7. (Optional) Check that the Python server is running by pressing the `Launch Python Server UI` button:

Launch Python Server UI

You are now ready to add your new application to Prometheus.

# Step 8: Reconfigure Prometheus

Now that you have your application running, it is time to reconfigure Prometheus so that it knows about the new `pythonserver` node to monitor. You can do this by adding the Python server as a target in your `prometheus.yml` file.

## Your Task

1. First, open the `prometheus.yml` file:

Open **prometheus.yml** in IDE

2. Next, create a new job to monitor the `pythonserver` service that is listening on port `8080`. Use the previous job as an example.

- Click here for a hint
- Click here for the solution

3. Check that your complete `prometheus.yml` file looks similar to this one:

► Click here to check your work

4. Restart the `prometheus` server to pick up the new configuration changes:

```
1. 1
1. docker restart prometheus
```

Copied!

Executed!

5. Check the Prometheus UI to see the new Targets.

Note: You may have to click the “show more” button next to `monitorPythonserver` as depicted below:

Targets

All Unhealthy Collapse All

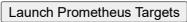
 Filter by endpoint or labels

monitorPythonserver (0/1 up) 



node (1/3 up) 

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
<a href="http://node-exporter1:9100/metrics">http://node-exporter1:9100/metrics</a>	UNKNOWN	<div>group="monitoring_node_ex1"</div> <div>instance="node-exporter1:9100"</div> <div>job="node"</div>	Never	0s	
<a href="http://node-exporter2:9100/metrics">http://node-exporter2:9100/metrics</a>	UNKNOWN	<div>group="monitoring_node_ex2"</div> <div>instance="node-exporter2:9100"</div> <div>job="node"</div>	Never	0s	
<a href="http://node-exporter3:9100/metrics">http://node-exporter3:9100/metrics</a>	UP	<div>group="monitoring_node_ex3"</div> <div>instance="node-exporter3:9100"</div> <div>job="node"</div>	3.243s ago	15.513ms	



Results

If everything went well, when you open the Prometheus targets, you will see the status of your Python server as in the image below.

# Targets

AllUnhealthyCollapse All

Q

Filter by endpoint or labels

monitorPythonserver (1/1 up) 

show less

Endpoint	State	Labels	Last
<a href="http://pythonserver:8080/metrics">http://pythonserver:8080/metrics</a>	UP	<div>group="monitoring_python"</div> <div>instance="pythonserver:8080"</div> <div>job="monitorPythonserver"</div>	9.5s

node (2/3 up) 

show less

Endpoint	State	Labels	Last
<a href="http://node-exporter1:9100/metrics">http://node-exporter1:9100/metrics</a>	DOWN	<div>group="monitoring_node_ex1"</div> <div>instance="node-exporter1:9100"</div> <div>job="node"</div>	10.3s
<a href="http://node-exporter2:9100/metrics">http://node-exporter2:9100/metrics</a>	UP	<div>group="monitoring_node_ex2"</div> <div>instance="node-exporter2:9100"</div> <div>job="node"</div>	3.5s
<a href="http://node-exporter3:9100/metrics">http://node-exporter3:9100/metrics</a>	UP	<div>group="monitoring_node_ex3"</div> <div>instance="node-exporter3:9100"</div> <div>job="node"</div>	14.2s

## Step 9: Monitor your application

In order to see some results of monitoring, you need to generate some network traffic.

1. Make multiple requests to the three endpoints of the Python server you created in the previous task and observe these calls on Prometheus.

```
1. 1
2. 2
3. 3
1. curl localhost:8081
2. curl localhost:8081/home
3. curl localhost:8081/contact
```

Copied!Executed!

Feel free to run these multiple times to simulate real network traffic.

2. Use the Prometheus UI to query for the following metrics.

- flask\_http\_request\_duration\_seconds\_bucket
- flask\_http\_request\_total
- process\_virtual\_memory\_bytes

If you are interested in what other metrics are being emitted by your application, you can view all of the metrics that your application is emitting by opening the /metrics endpoint just like Prometheus does. Feel free to experiment by running queries against other metrics:

Launch PythonServer Metrics

< > ↺

https://rofrano-8081.theiadocker-3-labs-prod-theiak8s-4-tor01.proxy.cognitiveclass.ai/metrics

```
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 329.0
python_gc_objects_collected_total{generation="1"} 76.0
python_gc_objects_collected_total{generation="2"} 0.0
# HELP python_gc_objects_uncollectable_total Uncollectable object found during GC
# TYPE python_gc_objects_uncollectable_total counter
python_gc_objects_uncollectable_total{generation="0"} 0.0
python_gc_objects_uncollectable_total{generation="1"} 0.0
python_gc_objects_uncollectable_total{generation="2"} 0.0
# HELP python_gc_collections_total Number of times this generation was collected
# TYPE python_gc_collections_total counter
python_gc_collections_total{generation="0"} 76.0
python_gc_collections_total{generation="1"} 6.0
python_gc_collections_total{generation="2"} 0.0
# HELP python_info Python platform information
# TYPE python_info gauge
python_info{implementation="CPython",major="3",minor="9",patchlevel="14",version="3.9.14"}
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 1.09953024e+08
# HELP process_resident_memory_bytes Resident memory size in bytes.
# TYPE process_resident_memory_bytes gauge
process_resident_memory_bytes 2.9843456e+07
"-----"
```

Conclusion

Congratulations! You have completed the lab on **Monitoring with Prometheus**. You are now well on your way to monitoring your own applications to ensure they are running properly.

In this lab, you deployed three node exporters and used Prometheus metrics to monitor them. You also learned how to instrument a Python Flask application, deploy it in Docker, and modify the Prometheus configuration to start monitoring this new application.

Next Steps

Your next challenge is to set up Prometheus in your development environment to monitor your applications. Use the **Prometheus Flask exporter** to instrument one of your Python Flask applications. Then use some of the queries you have learned in this lab to check on the health and performance of your application.

Author(s)

[Lavanya T S](#)  
[John J. Rofrano](#)

Other Contributor(s)

Pallavi Rai

Changelog

Date	Version	Changed by	Change Description
2022-09-12	0.1	Lavanaya T S	Initial version created
2022-09-25	0.2	John Rofrano	Added additional content
2022-09-26	0.3	Amy Norton	ID review
2022-09-27	0.4	Amy Norton	Updated images
2022-09-28	0.5	Steve Hord	QA pass edits
2022-09-28	0.7	Lavanaya T S	Added explanation and Python example
2022-09-29	0.8	John Rofrano	Reformatted some content for clarity
2022-09-30	0.9	John Rofrano	Added Prometheus in Docker
2022-10-03	0.9	Steve Hord	QA pass