LFS241:
MONITORING SYSTEMS AND SERVICES WITH PROMETHEUS

- V.05.02.2022 -

Lab 4.1 - Downloading Prometheus

Let's download and set up Prometheus.

In a new terminal, change to your home directory and download Prometheus version 2.26.0 for Linux:

```
wget
https://github.com/prometheus/prometheus/releases/download/v2.26.0/prometheus-2.
26.0.linux-amd64.tar.gz
```

Extract the archive:

```
tar xvfz prometheus-2.26.0.linux-amd64.tar.gz
```

Change into the extracted directory:

```
cd prometheus-2.26.0.linux-amd64
```

Lab 4.2 - Configuring Prometheus

Before starting Prometheus, you need to create a configuration file for it.

Prometheus collects metrics from monitored targets by scraping metrics HTTP endpoints on these targets. Since Prometheus also exposes data in the same way about itself, it can also scrape and monitor its own health. While a Prometheus server that only collects data about itself is not very useful, it is a good starting example.

Save the following Prometheus configuration as a file named **`prometheus.yml`** in the current directory (overwrite the existing example **`prometheus.yml`** file):

```
global:
  scrape_interval: 5s
  evaluation_interval: 5s

scrape_configs:
  - job_name: 'prometheus'
    static_configs:
    - targets: ['localhost:9090']
```

This configures Prometheus to scrape metrics from itself every 5 seconds. It also configures recording and alerting rules to be evaluated every 5 seconds, which will become relevant later on.

Note: A 5-second scrape interval is quite aggressive, but useful for demo purposes where you want data to be available quickly. In real-world scenarios, intervals between 10 and 60 seconds are most common.

For a complete specification of configuration options, see the configuration documentation.

Besides the configuration file, Prometheus also has flag-based configuration settings. You can see these by running:

```
./prometheus -h
```

While flag-based settings can only be set on server startup and are restricted to simple options, any settings in the configuration file can be reloaded during runtime without requiring a server restart (either by sending a **`HUP`** signal or via the web API).

Lab 4.3 - Starting Prometheus

Start Prometheus with your newly created configuration file:

```
./prometheus
```

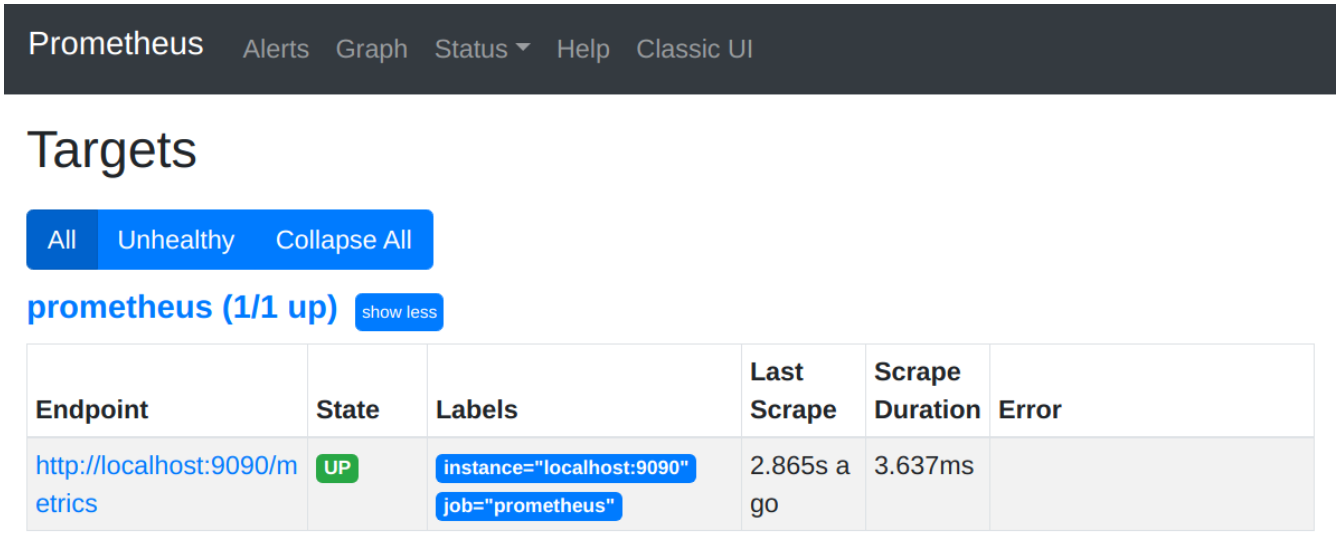
By default, Prometheus stores its database in the **`./data`** directory relative to the current working directory (flag **`--storage.tsdb.path`**) and reads its configuration file from **`prometheus.yml`** (flag **`--config.file`**).

Note: *When starting long-running processes (like the Prometheus server) throughout this course, we assume that you keep them running for the entire course duration unless noted otherwise. In production setups, you would typically use a supervisor software like systemd or a cluster manager like Kubernetes to keep server processes running in the background. In this course, we will not assume a particular deployment system and run components manually from the command-line instead. To facilitate running multiple server processes over a single SSH session when working on a remote machine (even across logouts and reconnects), you can use terminal multiplexer tools like screen, tmux, or byobu, that allow you to create and manage multiple virtual terminals over the same connection. If you are new to terminal multiplexers, we recommend byobu, as it is the most modern and easiest to use.*

Prometheus should start up and show a status page about itself at **`http://<machine-ip>:9090/status`**. Give it a couple of seconds to collect data about itself from its own HTTP metrics endpoint. If needed, you can shut down Prometheus gracefully by pressing **`Ctrl-C`** (but we assume that you keep it running).

You can also verify that Prometheus is serving metrics about itself by navigating to its metrics endpoint at **`http://<machine-ip>:9090/metrics`**.

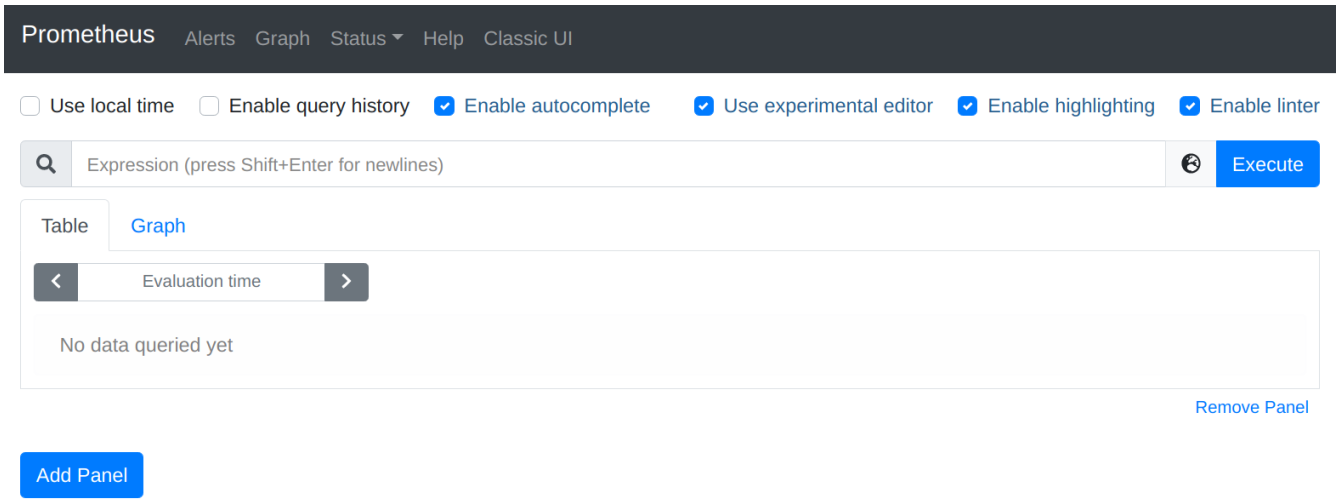
Navigate to **`http://<machine-ip>:9090/targets`** to verify that Prometheus is able to scrape itself - the one endpoint for the prometheus scrape job should be shown in green as **`UP`**:



Prometheus targets page

Lab 4.4 - Using the Expression Browser

In this step, you will explore Prometheus's built-in expression browser and run some test queries. To use the expression browser, navigate to **`http://<machine-ip>:9090/`** and choose the **`Table`** view. It should look like this:



Prometheus expression browser

We recommend activating the "Use experimental editor" checkbox, as the experimental expression editor supports syntax highlighting, contextual autocompletion, and inline-linting features, that make the editing experience much more pleasant.

The **`Table`** tab shows the most recent value of each output series of an expression, while the **`Graph`** tab plots values over time. The latter can be expensive (for both the server and the browser). It is in general a good idea to try potentially expensive expressions in the **`Table`** tab first and switch to the **`Graph`** tab once an expression is sufficiently narrowed down. Take a bit of time to play with the expression browser.

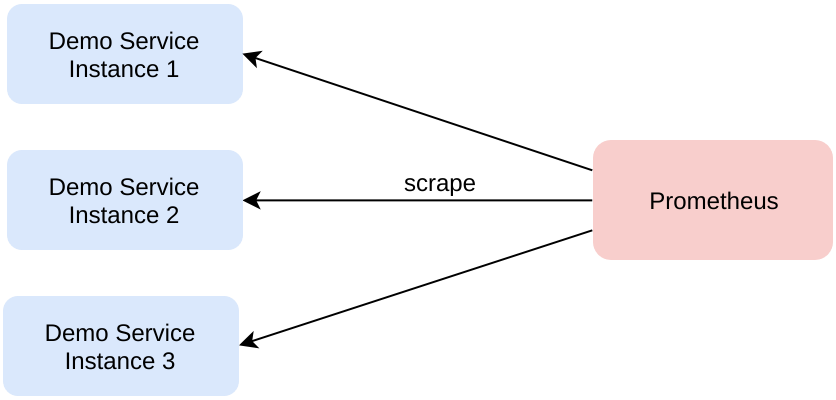
Suggestions:

- **`prometheus_tsdb_head_samples_appended_total`** shows you the total number of samples ingested into Prometheus's local storage since process start time.
- **`rate(prometheus_tsdb_head_samples_appended_total[1m])`** shows you the number of samples ingested per second as averaged over a 1-minute window.
- **`up{job="prometheus"}`** shows the value of the synthetic **`up`** metric for the Prometheus target, which is set to **`0`** when a scrape fails and to **`1`** when it succeeds.

Try viewing both expressions in the **`Table`** view as well as the **`Graph`** view and experiment with the graph range settings.

Lab 5.1 - Monitoring a Demo Service

To learn more about querying, you are going to run and monitor three instances of a demo service that outputs synthetic (artificially generated) metrics and let Prometheus scrape them:



Prometheus scraping three demo service instances

You will then be able to experiment with the data that Prometheus collects from it.

In a new terminal, change to your home directory and download the demo service:

```
wget
https://github.com/juliusv/prometheus_demo_service/releases/download/0.0.4/prome
theus_demo_service-0.0.4.linux.amd64.tar.gz
```

Extract the archive:

```
tar xvfz prometheus_demo_service-0.0.4.linux.amd64.tar.gz
```

Run three instances of the demo service in the background, each on its own port:

```
./prometheus_demo_service -listen-address=:10000 &
./prometheus_demo_service -listen-address=:10001 &
./prometheus_demo_service -listen-address=:10002 &
```

Each of the instances now serves synthetic metrics about itself on **`http://<machine-ip>:<instance-port>/metrics`**. Explore the metrics of one instance a bit. The demo service generates four different types of simulated metrics:

- Metrics about an HTTP API server (request counts, durations, errors)
- CPU usage metrics
- Disk usage and total size metrics
- Metrics about a batch job that runs periodically and can fail with a certain probability.

You will learn more about these metrics in detail when querying them.

Add the following list entry to the **`scrape_configs`** section in your **`prometheus.yml`** to make Prometheus scrape the demo service instances:

```
  - job_name: 'demo'
    static_configs:
      - targets:
        - 'localhost:10000'
        - 'localhost:10001'
        - 'localhost:10002'
```

Make sure that Prometheus reloads its configuration file:

```
killall -HUP prometheus
```

The Prometheus server's targets page at **`http://<machine-ip>:9090/targets`** should now show the three new targets in an **`UP`** state.

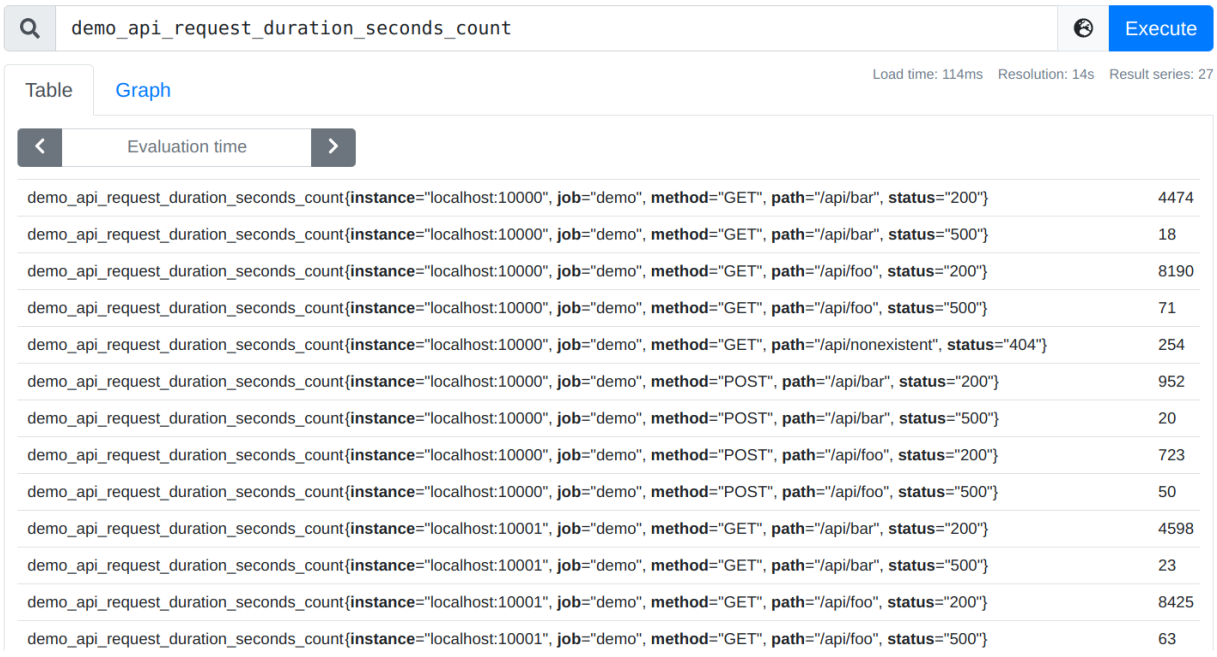
Note that so far, you have only configured targets statically via the configuration file. Later you will learn how to use service discovery integrations to discover targets dynamically in rapidly changing environments.

Lab 5.2 - Selecting Series

Before you can do anything more useful in PromQL, you have to learn how to select series data from the TSDB. The simplest way to do so is to ask for all series that have a certain metric name. The query for this is simply the metric name itself. For example:

```
demo_api_request_duration_seconds_count
```

Try executing this query (and subsequent ones) both in the **`Table`** view and the **`Graph`** view of the expression browser. The **`Table`** view will show you the latest value for each series, while the **`Graph`** view will show you their development over time. The **`Table`** view should look like this:



Selecting request counts in the expression browser

Note: The **`demo_api_request_duration_seconds_count`** metric is a counter that tells you the total number of handled API requests. The reason why it awkwardly includes

`request_duration_seconds` in its name is that it is created as a by-product of a histogram tracking request durations with the base name **`demo_api_request_durations_seconds`**. A stand-alone counter metric might otherwise be more aptly named **`demo_api_requests_total`**.

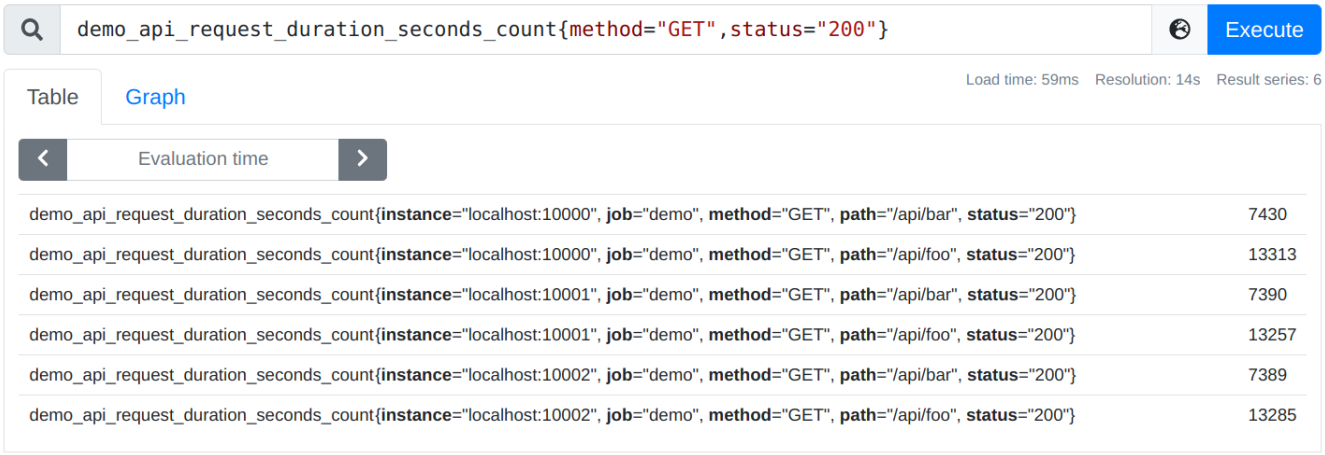
To narrow down the set of series to select, you can add filter conditions based on the label values on each series. For example, the following query only selects requests that resulted in a **`200`** HTTP status code:

```
demo_api_request_duration_seconds_count{status="200"}
```

You can also filter by multiple label conditions like this:

```
demo_api_request_duration_seconds_count{method="GET",status="200"}
```

This is what the result should look like:



Narrowing down request counts in the expression browser

Each individual label matcher needs to apply for a series to be selected, so the query above selects only series that have both the **`method="GET"`** and the **`status="200"`** labels.

There are several other label matcher types that Prometheus supports besides equality matching:

- **`!=`:** Match only labels that have a different value than the one provided.
- **`=~`:** Match only labels whose value matches a provided regular expression.
- **`!~`:** Match only labels whose value does not match a provided regular expression.

For example, to only select requests for the paths **`/api/foo`** and **`/api/bar`**, run the following query:

```
demo_api_request_duration_seconds_count{path=~"/api/(foo|bar)"}
```

Note: Regular expressions in Prometheus are always fully anchored. That means that they always try to match the entire string instead of just a substring. Thus you do not need to prefix the regular expression with **"^"** to match the beginning of the **`path`** label value, or suffix it with **"$"** to match the end. In cases where you want to allow matching substrings, you can prepend and append **".*"** to the regular expression as necessary.

In Prometheus, metric names are internally represented as just another label with the special label name **`__name__`**. So the query:

```
demo_api_request_duration_seconds_count
```

...is equivalent to selecting the metric name by its internal **`__name__`** label:

```
{__name__="demo_api_request_duration_seconds_count"}
```

This can sometimes be useful to select multiple metric names at once using regular expressions for debugging purposes.

Note: So far you have selected metric names without regard for the scrape job they originated from. In larger settings it is possible for multiple services (jobs) to expose the same metric name, but with slightly different semantics. To be on the safe side and avoid naming collisions, always include a selector for the **`job`** label in your series selectors. For example, instead of:

```
demo_api_request_duration_seconds_count
```

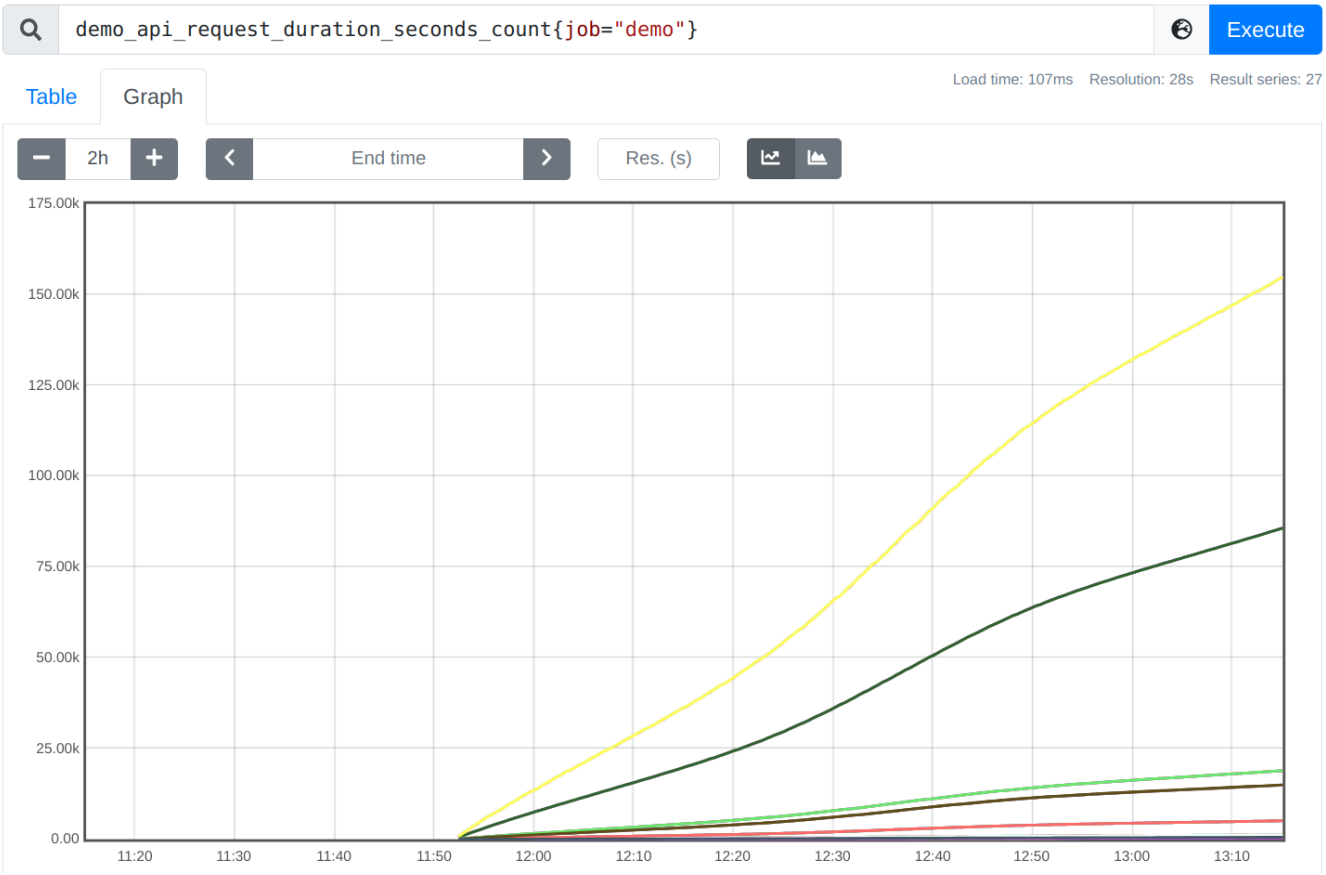
...query for:

```
demo_api_request_duration_seconds_count{job="demo"}
```

The query examples in the following sections will follow this recommendation.

Lab 5.3 - Computing Rates of Increase

Graphing raw counter metrics (like the total number of HTTP requests handled by a process) is usually not very useful, as they only ever go up and you almost never care about a counter's absolute value. For example, if you just graph **`demo_api_request_duration_seconds_count{job="demo"}`**, it will look something like this:



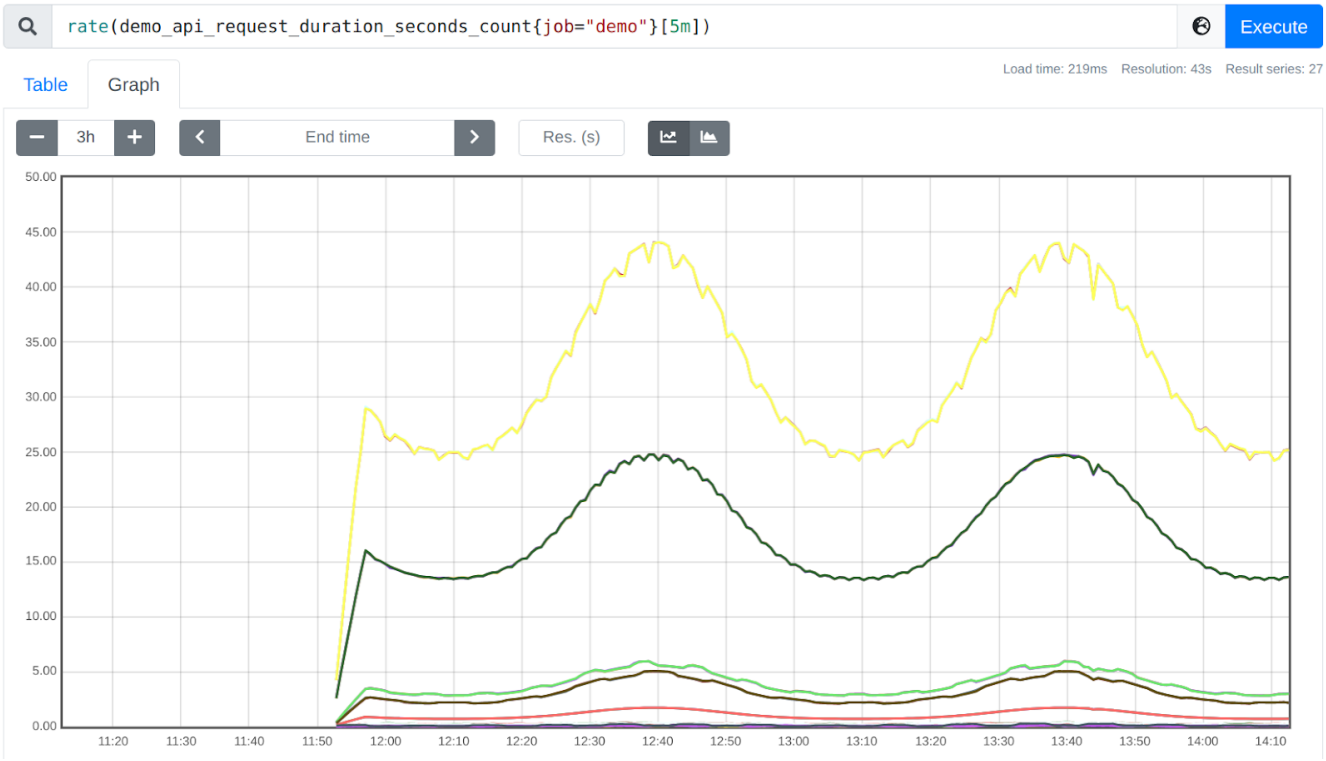
Graphing raw counter values

Instead, you typically want to know how fast a counter is increasing. PromQL has multiple distinct functions for this.

The most common function for calculating the rate of increase is **`rate()`**. **`rate()`** calculates the per-second increase of a counter as averaged over a specified window of time. To tell **`rate()`** the window of time to average over, you have to add a range selector after the series selector, like **`[5m]`** for a five-minute window. For example, the following query tells you the per-second increase of all the **`demo_api_request_duration_seconds_count`** series as averaged over a five-minute window at each point in the graph:

```
rate(demo_api_request_duration_seconds_count{job="demo"}[5m])
```

The result now looks more useful:



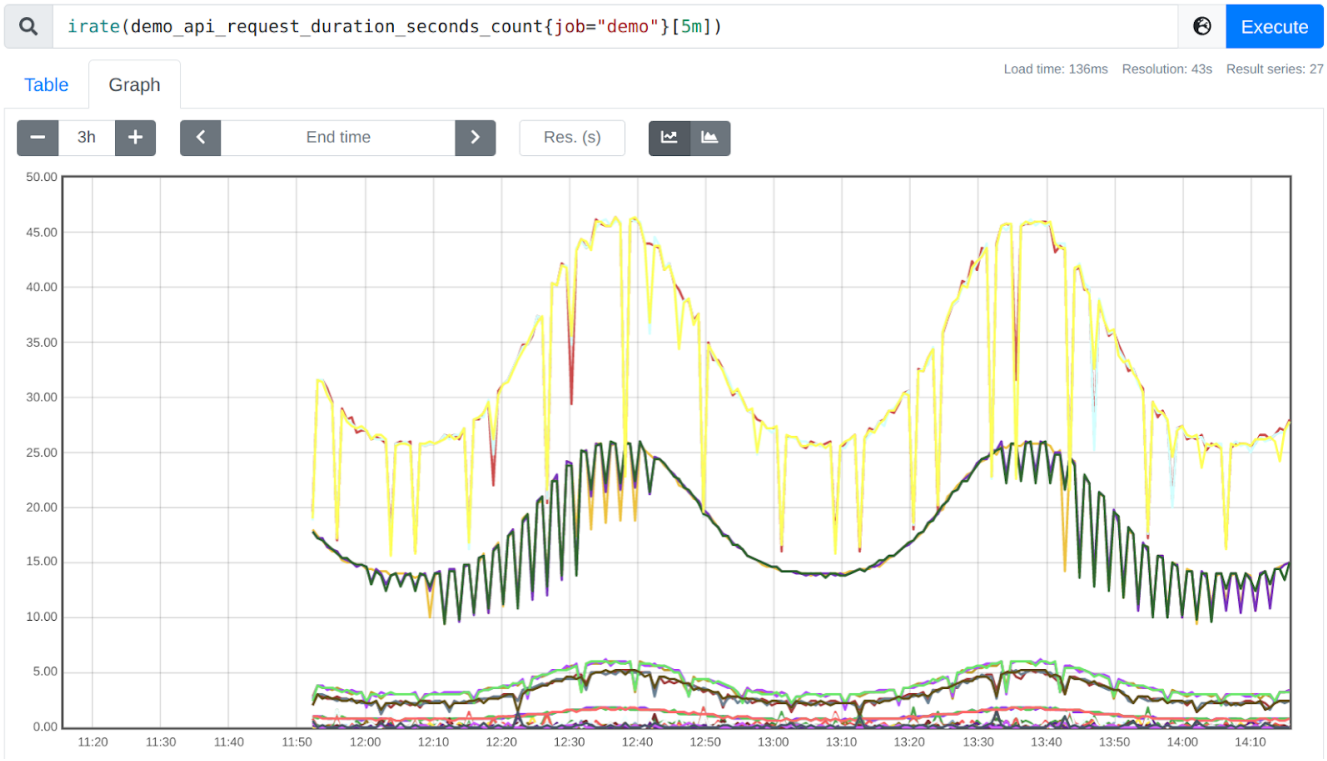
Graphing per-second counter increase rates

Counter metrics can reset to 0 when a scraped process restarts and loses its counter state, but **`rate()`** automatically corrects for this by assuming that any decrease in a counter value is a reset and pretending that the new lower value represents another increase starting from 0 that occurred between scrapes.

A variant of **`rate()`** is **`irate()`** (short for "instantaneous rate"). Instead of averaging over the entire provided time window, it only considers the last two points under the provided window and calculates

an instantaneous rate from them. The provided window duration still tells **`irate()`** how far to maximally look back for the last two samples. **`irate()`** reacts faster to counter changes and is thus useful for high-resolution, zoomed-in graphs. Since **`rate()`** provides smoother results, it is recommended over **`irate()`** in alerting expressions that should not fire when there is only a short spike in a rate of increase.

Using **`irate()`**, the above expression will show details (short drops) in the rate that weren't visible in the smoothed rate before:



Graphing instantaneous counter increase rates

Instead of calculating the per-second rate, you can also query the total increase over a given time window using the **`increase()`** function:

```
increase(demo_api_request_duration_seconds_count{job="demo"}[1h])
```

Generally it's a good idea to stick to **`rate()`** and per-second units everywhere, as that makes comparisons and arithmetics between expressions easier later on.

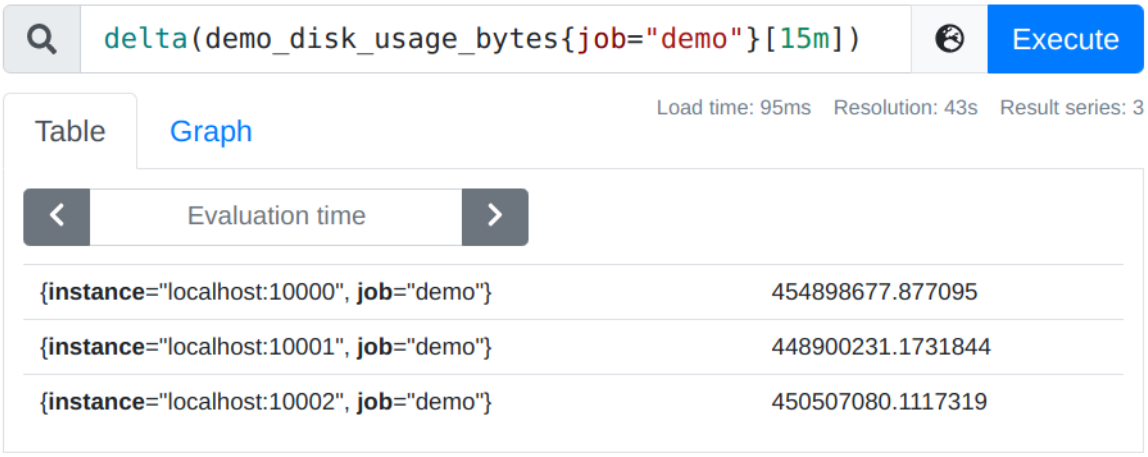
Lab 5.4 - Computing Derivatives

The **`rate()`**, **`irate()`**, and **`increase()`** functions only work for counter metrics, since they treat any decrease in value as a counter reset and can only output non-negative results. For gauge metrics that track a value that can go up or down (like temperatures or memory usage or free disk space), use the **`delta()`** and **`deriv()`** functions instead to see how they develop over time.

To get the raw increase of **`demo_disk_usage_bytes{job="demo"}`** over 15 minutes, use **`delta()`**:

```
delta(demo_disk_usage_bytes{job="demo"}[15m])
```

In the console view, this should show an increase of hundreds of MB in disk usage over the last 15 minutes:



Showing the difference in disk usage over time

Note that **`delta()`** only considers the first and the last data point under the provided time window and disregards overall trends in the intermediate data points. To calculate the per-second derivative over the same time window using full linear regression over all available data points under the same

window, use the **`deriv()`** function. For example, to calculate by how much the disk usage is going up or down per-second when looking at a 15-minute window, query for:

```
deriv(demo_disk_usage_bytes{job="demo"}[15m])
```

The result should look somewhat like this:

Graphing the derivative of the disk usage

The **`predict_linear()`** function is an extension to this that allows you to predict what the value of a gauge will be in a given amount of time in the future. For example, the following query will try to predict what the disk usage is in one hour, based on its development in the last 15 minutes:

```
predict_linear(demo_disk_usage_bytes{job="demo"}[15m], 3600)
```

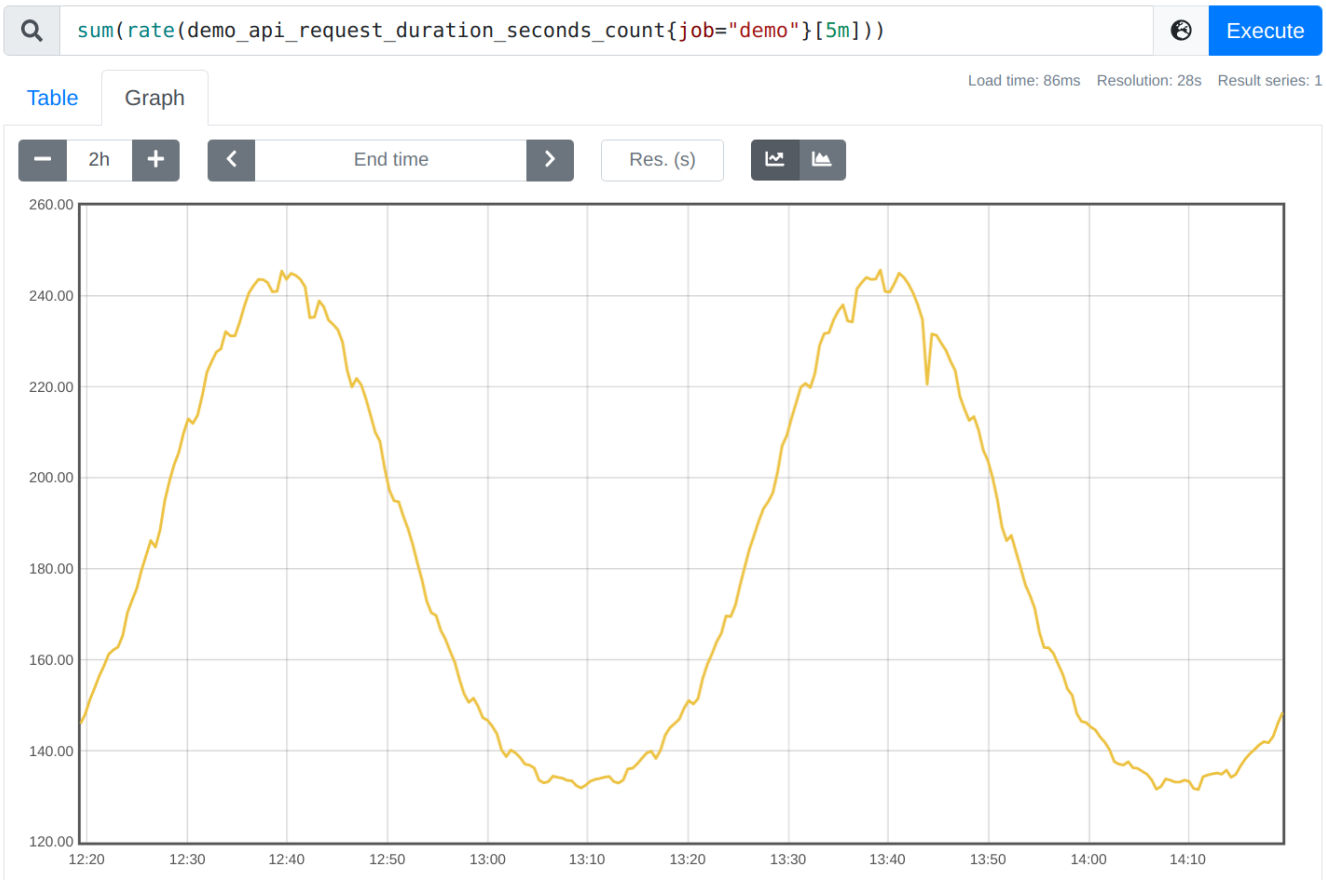
This can be useful for building alerts that tell you if the disk is about to fill up in several hours.

Lab 5.5 - Aggregations

Prometheus's time series data can be highly dimensional. This is helpful for drilling down, but often you want to aggregate over dimensions and show overall results. For example, if you wanted to know the total number of requests our demo service is handling per second, you could sum up the individual rates:

```
sum(rate(demo_api_request_duration_seconds_count{job="demo"}[5m]))
```

This will only give you one output series:



Graphing an aggregated sum of all request rates

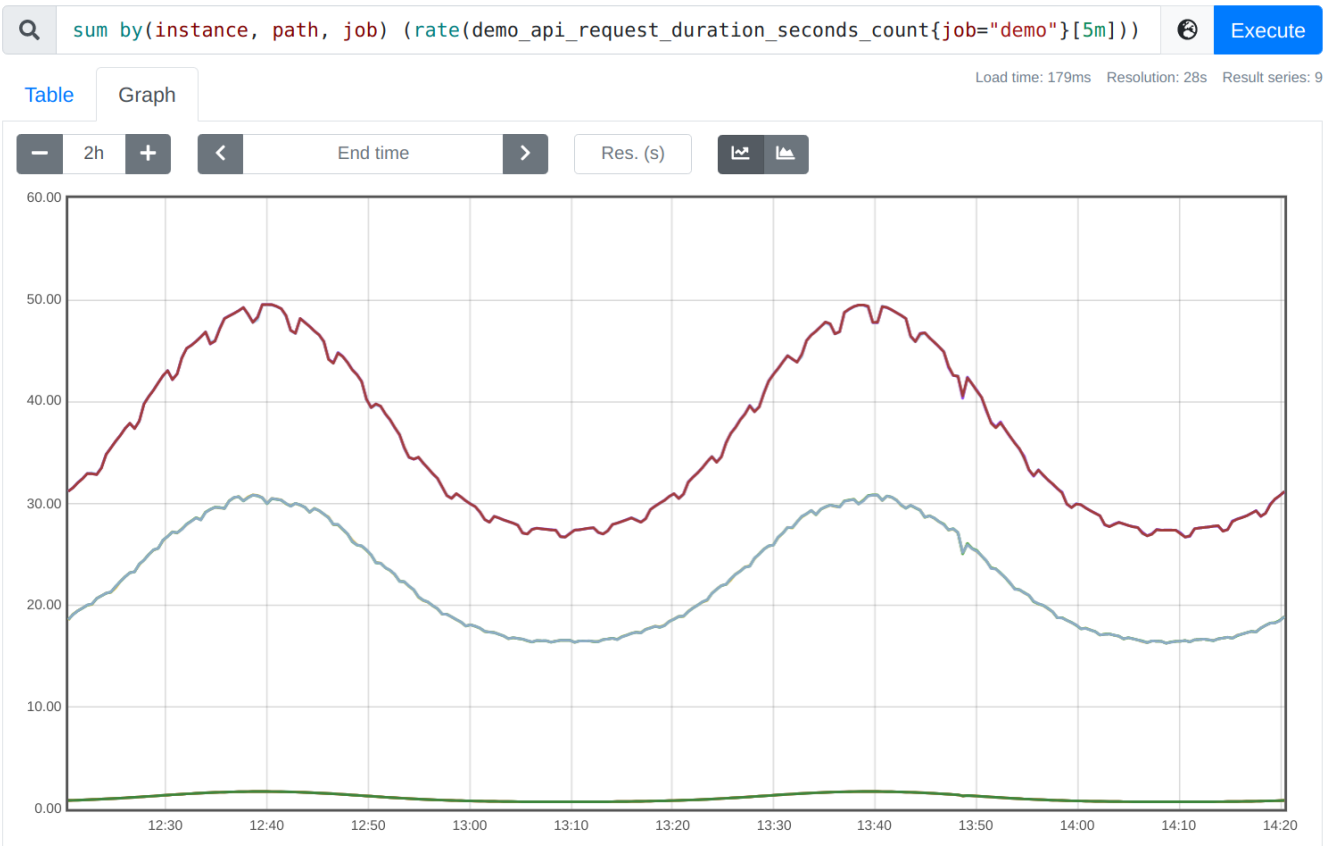
However, usually you will want to preserve *some* of the dimensions of a metric and only aggregate over those that you do not care about in a particular context. For example, you might want to calculate total rates per **`instance`** and **`path`**, but not care about individual **`method`** or **`status`** results. To do that, you can add a **`without()`** modifier to the **`sum()`** aggregator:

```
sum without(method, status)
(rate(demo_api_request_duration_seconds_count{job="demo"}[5m]))
```

This is equivalent to using the **`by()`** modifier with the opposite set of labels (the ones you want to keep):

```
sum by(instance, path, job)
(rate(demo_api_request_duration_seconds_count{job="demo"}[5m]))
```

The resulting sum is now grouped by **`instance`**, **`path`**, and **`job`**:



Graphing summed request rates while preserving some dimensions

While **`sum()`** is the most commonly used aggregation operator, Prometheus supports many kinds of aggregation operators, where some take extra arguments:

- **`sum()`**: Calculates the sum over dimensions.
- **`min()`**: Selects the minimum value across dimensions.
- **`max()`**: Selects the maximum value across dimensions.
- **`avg()`**: Calculates the average over dimensions.
- **`stddev()`**: Calculates the population standard deviation over dimensions.
- **`stdvar()`**: Calculates the population standard variance over dimensions.
- **`count()`**: Counts number of series in the vector.
- **`count_values(value_label, ...)`**: Counts the number of series for each distinct sample value.
- **`bottomk(k, ...)`**: Selects the smallest k elements by sample value.
- **`topk(k, ...)`**: Selects the largest k elements by sample value.
- **`quantile(φ, ...)`**: Calculates the φ-quantile (0 ≤ φ ≤ 1) over dimensions.

Similarly to **`sum()`**, these operators each support the **`by()`** and **`without()`** modifiers to preserve or aggregate over certain dimensional groupings in the output.

Lab 5.6 - Arithmetic

One of the most powerful features of PromQL is its ability to easily perform arithmetic between entire sets of time series. But to start simple, let's first look at how you can use PromQL to simulate a calculator. For example, you can do basic arithmetic between scalar (non-vector) numbers by querying for:

```
(2 + 3 / 6) * 2^2
```

This returns a scalar-typed result with the value 10, as expected. Similarly to how you can multiply a vector in math with a scalar number, you can also multiply a vector (list) of time series with a scalar value in PromQL. Let's say you wanted to display the number of bytes that the last demo batch job run processed in GiB instead of raw bytes. You could query for:

```
demo_batch_last_run_processed_bytes{job="demo"} / 1024^3
```

The scalar operation is automatically applied to every element (series) in the vector and propagated into the output, with the new value in GiB:

Converting a number of processed bytes to GiB

Prometheus supports all the usual mathematical operators that you would expect. For details, see the Operators documentation.

Where Prometheus really starts to shine is when doing arithmetic between entire sets (vectors) of time series. When you ask PromQL to calculate binary operations between time series vectors, it automatically finds corresponding series with identical label sets on the left-hand side and the right-hand side, and performs the binary operation between each of the pairs. Each of the sub-results is propagated as an equivalently labeled value into the output vector.

A common example where this is useful is when wanting to calculate the average HTTP response time from a histogram's **`_count`** series (total count of requests) and **`_sum`** series (total time spent in requests), while preserving all dimensions in the output. The following query calculates the average response time as averaged over the last five minutes, by dividing the increase in request counts by the total time spent in requests during that time window:

```
  rate(demo_api_request_duration_seconds_count{job="demo"}[5m])
/
  rate(demo_api_request_duration_seconds_sum{job="demo"}[5m])
```

The result will look like this:

Graphing average request latencies using binary vector arithmetic

As you can see, PromQL preserved all existing dimensions and has produced a binary operation result for each unique set of labels.

The example above works nicely because labeled series match up exactly between the two input vectors. The only difference is in the metric name, which is ignored for vector matching purposes. When label sets on both sides don't match completely, you will need to introduce further constructs to make things work. One example where this comes up is when trying to calculate the ratio of the `status="500"` error rate vs. the total rate of all handled requests. You can *not* simply query for:

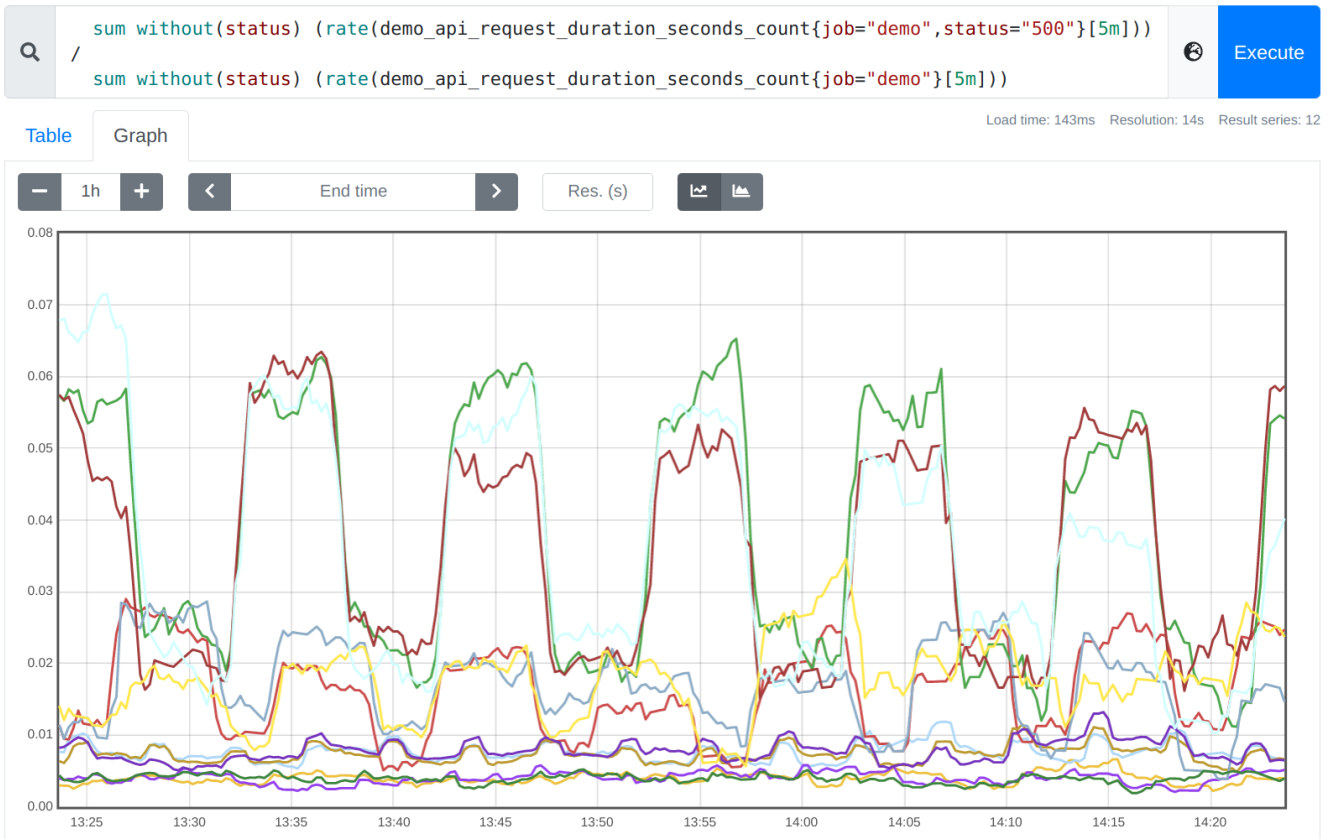
```
  rate(demo_api_request_duration_seconds_count{job="demo",status="500"}[5m])
/
  rate(demo_api_request_duration_seconds_count{job="demo"}[5m])
```

...as this would only select `status="500"` series on the left-hand side, and the `status!="500"` series on the right-hand side would find no matches at all and are thus simply dropped from the output.

So you have to make the vectors match somehow. For example, you can aggregate away the `status` label on both sides, as you know that it can only have the value `500` on the left-hand side and you want to sum over all its possible values on the right-hand side:

```
  sum without(status)
(rate(demo_api_request_duration_seconds_count{job="demo",status="500"}[5m]))
/
  sum without(status)
(rate(demo_api_request_duration_seconds_count{job="demo"}[5m]))
```

This will now produce the correct output:



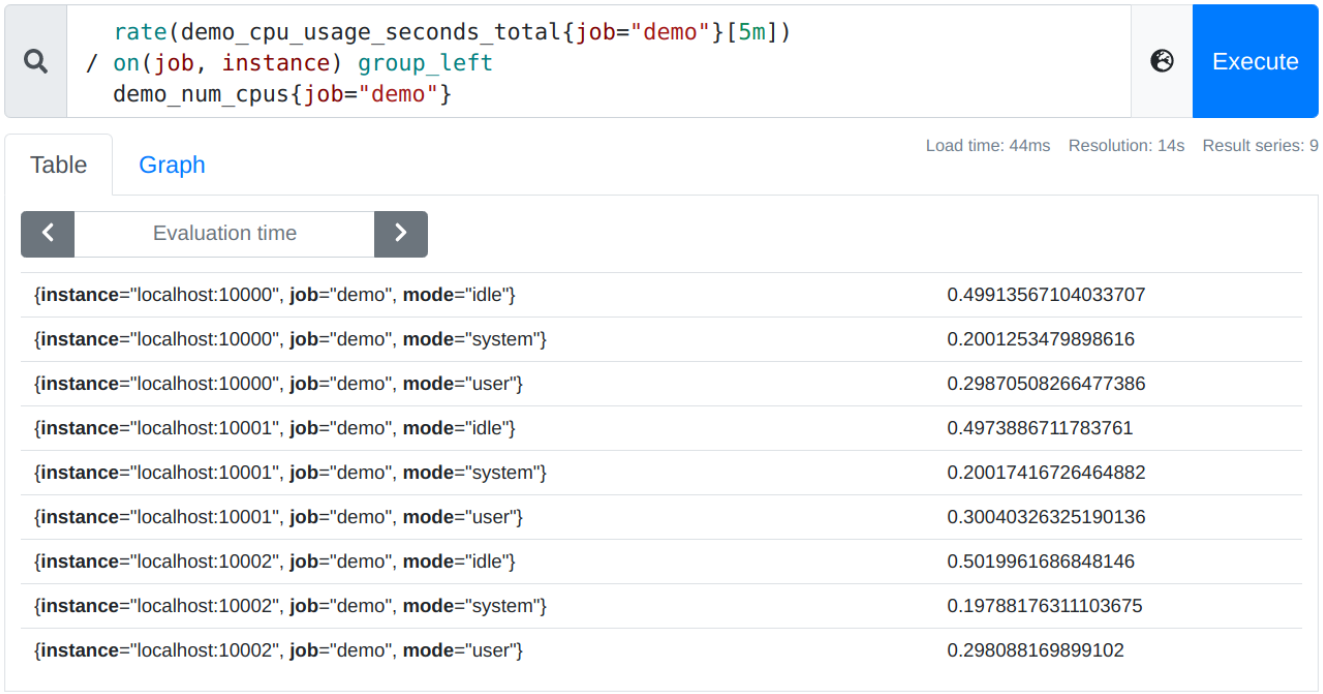
Graphing the ratio of `status="500"` error rates vs. total request rates

Note that you could include further labels in the `without()` or `by()` clause to produce the error rate ratios for exactly the dimensional combinations you are interested in.

There are also cases where the vector on one side of a binary operation naturally has more dimensions (labels) than the other vector. For example, the **`demo_num_cpus`** metric tells you the number of CPU cores of each instance, while the **`demo_cpu_usage_seconds_total`** metric has an additional **`mode`** label dimension that splits up the CPU usage per mode (**`idle`**, **`system`**, **`user`**, etc.). To calculate the per-mode CPU usage divided by the number of cores (to arrive at per-core usage from 0 to 1), you will need to tell the division binary operator to group its results by the extra **`mode`** label dimension present on the **`demo_cpu_usage_seconds_total`** metric. You can do this using the **`group_left`** modifier. At the same time, you also need to exclude this label from the vector matching requirements by explicitly matching only on the mutually existing label dimensions via an **`on()`** modifier:

```
  rate(demo_cpu_usage_seconds_total{job="demo"}[5m])
/ on(job, instance) group_left
  demo_num_cpus{job="demo"}
```

This will give you:



Calculating the per-mode CPU usage ratio, normalized for the number of cores

In addition to **`on()`**, there is also an opposite **`ignoring()`** matching modifier that can be used instead to exclude some label dimensions from the binary operation matching criteria. If there are extra dimensions on the right instead of the left side of the operation, use the **`group_right`** modifier instead of **`group_left`**.

To learn more about vector matching details, see the Operators documentation.

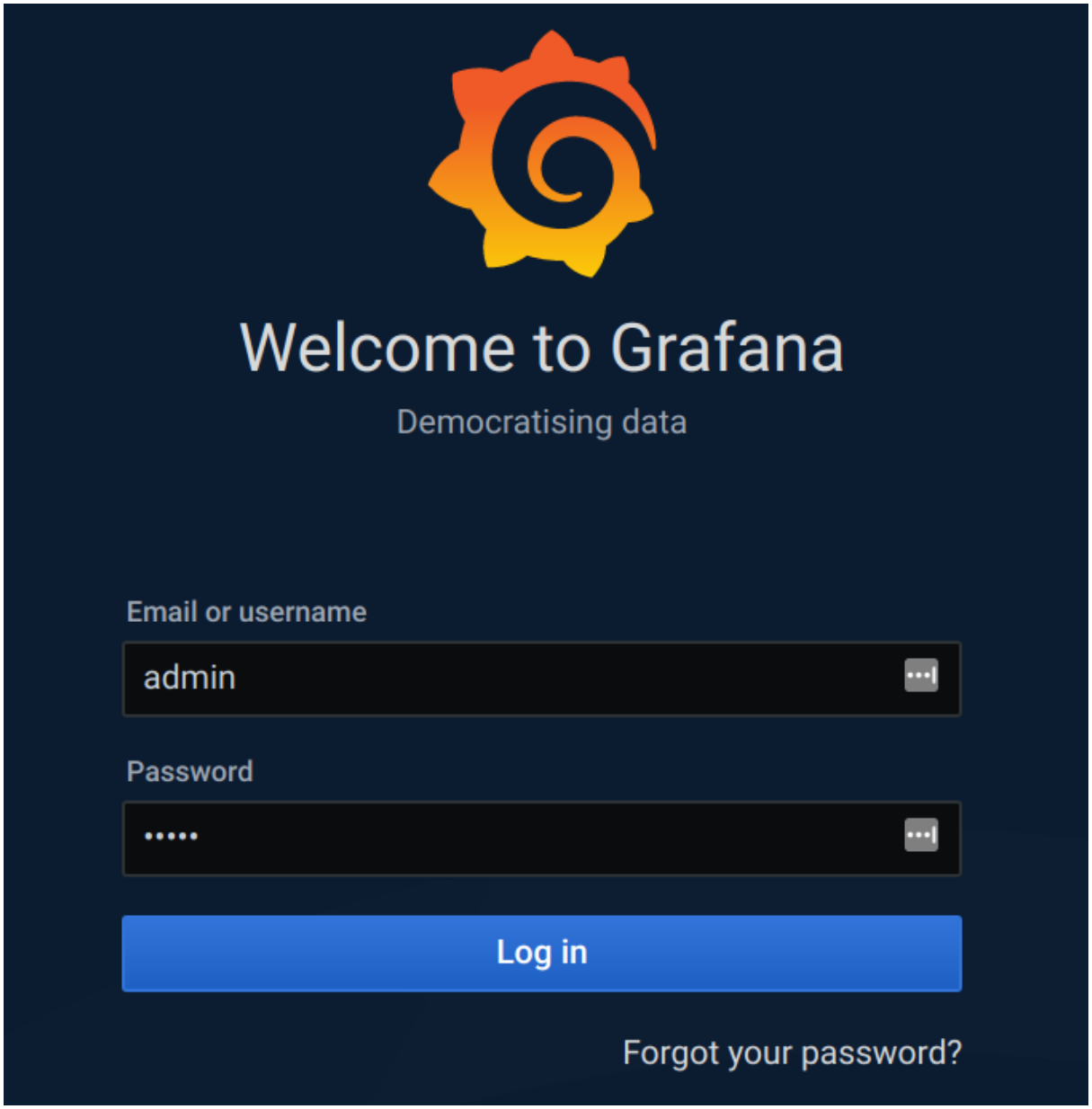
Lab 6.1 - Setting Up Grafana

Grafana can be deployed and configured in multiple ways, and can use different backends for storing dashboard metadata. We will not discuss operational details of Grafana here, but you can find these in the Grafana documentation. To keep things simple, you are going to run Grafana on Docker with default configuration parameters.

To run Grafana on Docker on port 3000, run:

```
docker run -d -p 3000:3000 --name grafana grafana/grafana:7.5.4
```

Head to **`http://<machine-ip>:3000/`** and log in with the default username **`admin`** and the password **`admin`**:



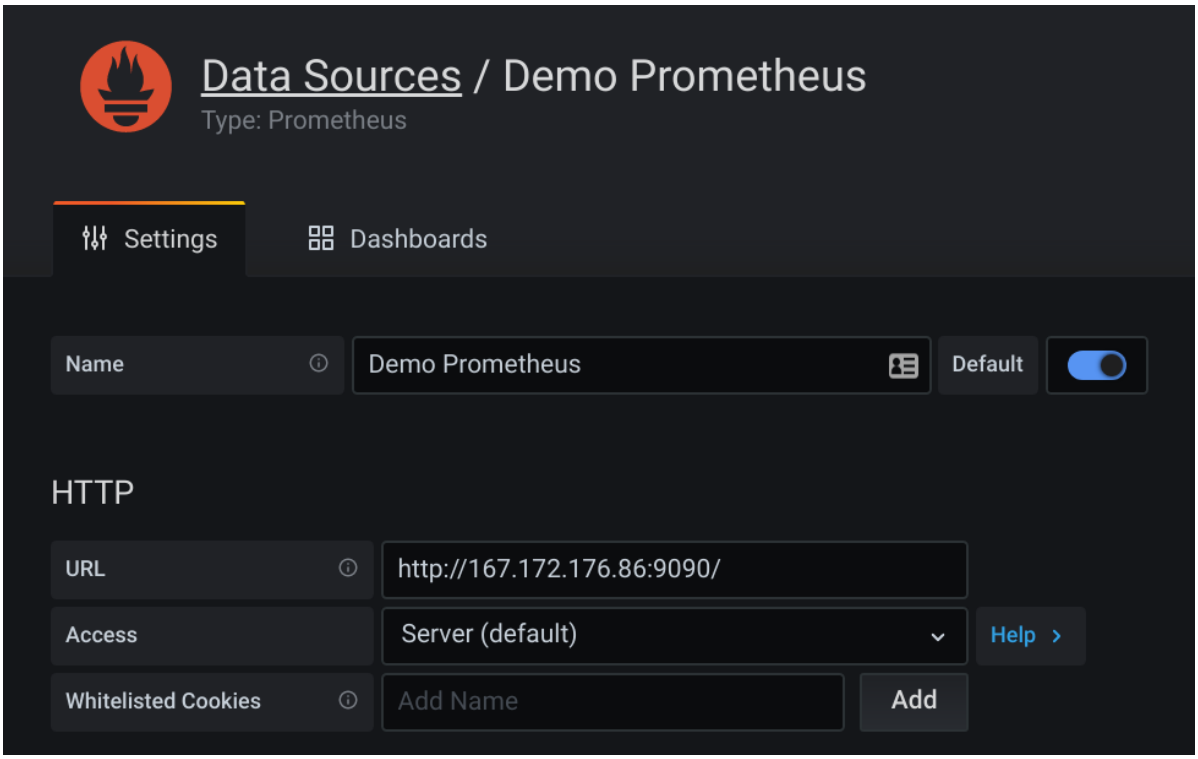
Grafana login screen

You will be prompted to update this password when logging in for the first time. Choose any password of your liking or skip the step.

Lab 6.2 - Creating a Prometheus Datasource

Before you can create a dashboard, you need to configure a data source in Grafana that knows how to query Prometheus:

- Click on *Add your first data source* in the *Home* view that is shown after logging in.
- Choose “Prometheus” as a datasource type.
- Fill out the data source fields:
  - Enter *Demo Prometheus* into the *Name* field.
  - Set the *URL* to *http://<machine-ip>:9090*.
  - Leave the other fields at their defaults.
- Click *Save & test* and verify that the data source was saved and tested successfully.



Demo Prometheus data source configuration in Grafana

This data source can now be selected in Grafana's dashboard panels in order to fetch data via PromQL queries from our demo Prometheus server.

Lab 6.3 - Creating a Dashboard

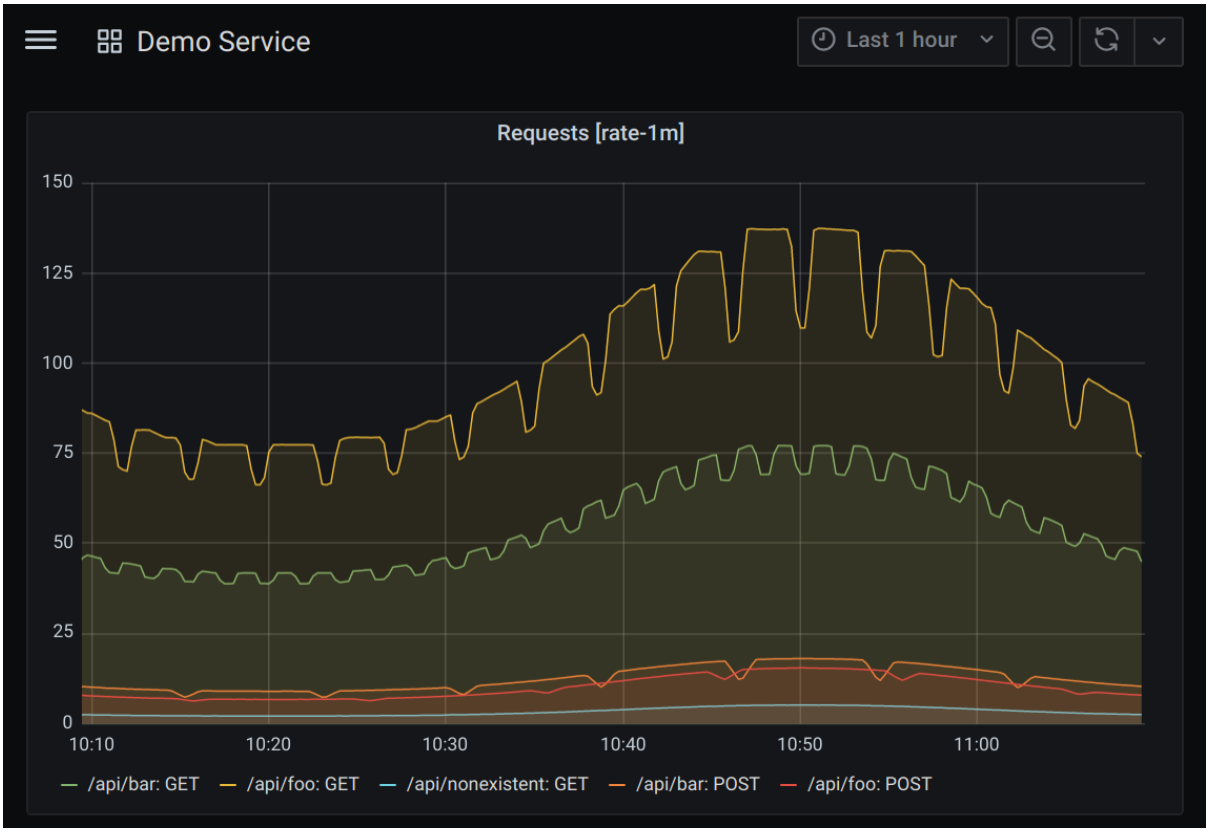
Now we can create our first Grafana dashboard:

- Click on the + (Create) icon in the left menu, which takes you to an empty new dashboard.
- Click on *Add an empty panel* to add a new graph panel to the dashboard.
- Set the time range (at the top right of the graph panel) to *Last 1 hour*.
- Enter the following expression into the query expression field (with the placeholder text “Enter a PromQL query”):

  ```
  sum without(instance, status)
  (rate(demo_api_request_duration_seconds_count{job="demo"}[1m]))
  ```

  This graphs the demo service HTTP request rate, aggregated over all instances and response status codes.
- Set the *Legend* field to *{{path}}: {{method}}* to display briefer legend strings that only contain the label values that are actually relevant to the displayed result.
- In the panel settings on the right, set the *Panel Title* field to *Requests [rate-1m]*.
- Click *Save* at the top to save the dashboard under the name *Demo Service*.

You have now created your first Grafana dashboard based on data from Prometheus. It should look somewhat like this:



Demo service dashboard with initial graph

Also add graphs for the following request statistics of the demo service:

- The total error rate, broken up by method and status code:
  - Title: *Errors [rate-5m]*
  - Expression:

```
sum without(instance, path)
(rate(demo_api_request_duration_seconds_count{status=~"5..",job="demo"}[5m]))
```

- 90th-percentile latency, broken up by path and method:
  - Title: *Latency [90th]*
  - Expression:

```
histogram_quantile(0.9, sum without(instance, status)
(rate(demo_api_request_duration_seconds_bucket{job="demo"}[5m]))
)
```

    - On the *Axes* settings in the right menu, set the *Unit* to *seconds (s)* under the *Time* category

Grafana also supports other panel types than graphs. For example, you can show a list of series and their values in a table, show only the current value of a single series, or display histogram data in a heatmap.

Let's show all service instances in a table whose internal batch job hasn't completed successfully in over 60 seconds:

- Add a new panel to your dashboard.
- Under *Visualization*, choose *Table*.
- Set the title to *Delayed batch jobs [seconds since last success]*.
- Under the *Query* settings, enable the *Instant* toggle, set the *Format* to *Table*, and set the query to:

  ```
  max by (instance) (time() -
  demo_batch_last_success_timestamp_seconds{job="demo"} > 60)
  ```

- Under *Transform*, choose *Filter by name*, then select *Time* to hide the time column.

Save the dashboard again. It should now look something like this:

Full demo service dashboard

Feel free to add more graphs and other panel types to the dashboard or change the graphing options.

Lab 7.1 - Install and Run the Node Exporter

Let's install the Node Exporter.

In a new terminal, change to your home directory and download Node Exporter version 1.1.2 for Linux:

```
wget
https://github.com/prometheus/node_exporter/releases/download/v1.1.2/node_export
er-1.1.2.linux-amd64.tar.gz
```

Extract the archive:

```
tar xvfz node_exporter-1.1.2.linux-amd64.tar.gz
```

Change into the extracted directory:

```
cd node_exporter-1.1.2.linux-amd64
```

The Node Exporter has a variety of collector modules that can be configured or turned off and on via flags (see the Node Exporter's collector module documentation). However, by default it will start up with a reasonable configuration without providing any flags.

Start the Node Exporter:

```
./node_exporter
```

In a browser, go to **`http://<machine-ip>:9100/metrics`** to see the metrics that the Node Exporter exposes.

Lab 7.2 - Monitor the Node Exporter

To scrape the Node Exporter, add the following scrape configuration to the **`scrape_configs`** list in your **`prometheus.yml`**:

```
  - job_name: 'node'
    static_configs:
      - targets: ['localhost:9100']
```

Reload the Prometheus configuration:

```
killall -HUP prometheus
```

Verify that **`http://<machine-ip>:9090/targets`** lists the **`node`** target with an **`UP`** status.

Lab 7.3 - Explore Node Exporter Metrics

Let's explore some of the metrics that the Node Exporter exposes.

The metric **`node_cpu_seconds_total`** tracks how many CPU seconds have been used since boot time per core (**`cpu`** label) and per mode (**`mode`** label). To see how many cores are used in each core and mode, calculate the rate over this counter:

```
rate(node_cpu_seconds_total{job="node"}[1m])
```

To only see actual CPU usage, filter out the **`idle`** mode. To see the actual CPU usage over all cores, query for:

```
sum without(cpu) (rate(node_cpu_seconds_total{mode!="idle",job="node"}[1m]))
```

To see how much memory (in GiB) is available on the machine, add the free, buffers, and cached memory amounts:

```
(
    node_memory_MemFree_bytes{job="node"}
  +
    node_memory_Buffers_bytes{job="node"}
  +
    node_memory_Cached_bytes{job="node"}
) / 1024^3
```

To see the free bytes on each filesystem, evaluate the following query:

```
node_filesystem_free_bytes
```

Together with the **`node_filesystem_size_bytes`** metric (which represents the total size of each filesystem in bytes), you can compute the usage in percent of each filesystem:

```
(node_filesystem_free_bytes / node_filesystem_size_bytes) * 100
```

You can also see the sum of the incoming and outgoing network traffic on the machine, grouped by network interface:

```
  rate(node_network_receive_bytes_total[1m])
+
  rate(node_network_transmit_bytes_total[1m])
```

The Node Exporter also exposes metrics that are not related to resource usage, but provide other kinds of information about the system. For example, query the system's boot time as a Unix timestamp:

```
node_boot_time_seconds
```

You could use this to detect nodes that are rebooting frequently by detecting frequent changes in this metric. The following query finds nodes that have rebooted more than 3 times in the last 30 minutes:

```
changes(node_boot_time_seconds[30m]) > 3
```

Another such metric provides the current system time of the monitored node:

```
node_time_seconds
```

Using the **`timestamp()`** function to get the scrape timestamp of a sample, you can compare Prometheus's idea of the current time with the scraped node's system time to debug potential time-related problems:

```
node_time_seconds - timestamp(node_time_seconds)
```

There are many more metrics that the Node Exporter exposes. Study its metrics endpoint at **`http://<machine-ip>:9100/metrics`** for more details.

Lab 8.1 - Run cAdvisor

Let's run cAdvisor. The easiest way to deploy it is using Docker. In a new terminal, run:

```
docker run \
  --volume=/:/rootfs:ro \
  --volume=/var/run:/var/run:ro \
  --volume=/sys:/sys:ro \
  --volume=/var/lib/docker/:/var/lib/docker:ro \
  --volume=/dev/disk/:/dev/disk:ro \
  --publish=8080:8080 \
  --detach=true \
  --name=cadvisor \
  gcr.io/google-containers/cadvisor:v0.36.0
```

You should now be able to reach the cAdvisor metrics endpoint at **`http://<machine-ip>:8080/metrics`**.

Lab 8.2 - Monitor cAdvisor

To scrape cAdvisor, add the following scrape configuration to the **`scrape_configs`** list in your **`prometheus.yml`**:

```
  - job_name: 'cadvisor'
    static_configs:
      - targets: ['localhost:8080']
```

Reload the Prometheus configuration:

```
killall -HUP prometheus
```

Verify that **`http://<machine-ip>:9090/targets`** lists the **`cadvisor`** target with an **`UP`** status.

Lab 8.3 - Explore Container Metrics

Let's have a look at some of the metrics that cAdvisor exposes. Besides some metrics about cAdvisor itself, it exposes per-container resource usage metrics prefixed with **`container_`**. These metrics include CPU, memory, network, and disk usage, as well as others.

Each of these metrics has an **`id`** label that corresponds to the container's full path name in the host's virtual **`cgroups`** filesystem. For example, an **`id`** value of **`/docker/7230bbfcad8a1ca963822fb28a06c25df7ba17a801c3c13cf1790238e53cb630`** would map to the cgroups path **`/sys/fs/cgroup/memory/docker/7230bbfcad8a1ca963822fb28a06c25df7ba17a801c3c13cf1790238e53cb630`** (for other cgroup resource types than memory, replace the **`memory`** in this path with the appropriate cgroup resource name, like **`cpu`** or **`pids`**).

The per-container metrics also expose an **`image`** label, although this is only set in the case of Docker containers that have an image name (like **`image="gcr.io/google-containers/cadvisor:v0.36.0"`**). The same is true for the **`name`** label. Many of the per-container metrics will have empty **`image`** and **`name`** labels, as they correspond to cgroups that were created by the host's systemd init system, which does not provide this information.

For example, **`container_cpu_usage_seconds_total`** is a metric that exposes the number of CPU seconds used by each container (cgroup) on the machine, split out by CPU core. Try querying for it:

```
container_cpu_usage_seconds_total
```

Since it's counting the number of seconds that a container has used the CPU for so far, you can calculate the per-second increase rate of this metric to arrive at a usage ratio between 0 and 1 per core:

```
rate(container_cpu_usage_seconds_total[1m])
```

To get the total CPU usage in number of cores for the Grafana container that you started earlier, you could limit this query to containers with the **`name="grafana"`** and sum the usage over all CPU cores:

```
sum without(cpu) (rate(container_cpu_usage_seconds_total{name="grafana"}[1m]))
```

To get this container's memory usage in bytes, you could query for:

```
container_memory_usage_bytes{name="grafana"}
```

cAdvisor outputs many more metrics about the containers running on a machine. The full list, including documentation strings, is available on its metrics endpoint at **`http://<machine-ip>:8080/metrics`**. You can also find a table documenting the meaning of each metric in the cAdvisor documentation about Prometheus metrics.

Lab 9.1 - Instrumenting an Example Service

Let's instrument an example service. You can either choose to instrument an example based on Go or based on Python. Both implement a demo API on port **`12345`** which processes dummy API requests at the HTTP paths **`/api/foo`** and **`/api/bar`** with different latencies. They also run a periodic batch job in the background which can fail with a certain probability. Try adding metrics for the following:

- The request latency distribution for each path of the API.
- The number of total vs. failed batch job runs.
- The timestamp of the last successful batch job run.
- Other metrics that you can think of.

In a new terminal, clone the git repository containing both the Go and the Python example code:

```
git clone https://github.com/juliusv/instrumentation-exercise
```

Change into the cloned directory:

```
cd instrumentation-exercise
```

There are two branches in the git repository:

- **`master`**: Contains the uninstrumented demo API code.
- **`instrumented`**: An example of an instrumented version of the same code. Note that the example instrumentation is not necessarily ideal or complete.

Option A: Go

Follow this section if you prefer instrumenting the Go example. We assume that you have a working Go environment set up for this step. Otherwise, see the instructions for installing Go.

Change into the **`go`** directory in the example repository:

```
cd go
```

In there, you will find a **`main.go`** file to edit and add instrumentation to.

To fetch missing package dependencies, run:

```
go get -d .
```

You may have to repeat this step when adding new package imports like the Prometheus client library.

To start the example server, run:

```
go run main.go
```

You can stop the server by pressing **`<Ctrl>-C`**.

The following links will be helpful in instrumenting the service:

- Prometheus instrumentation library documentation
- Prometheus HTTP exposition and tooling library documentation

Option B: Python

Follow this section if you prefer instrumenting the Python example. We assume that you have a working Python 2 environment set up for this step. Otherwise, see the instructions for installing Python.

Install the Prometheus Python client library:

```
export LC_ALL=C
sudo pip install prometheus_client
```

Change into the **`python`** directory in the example repository:

```
cd python
```

In there, you will find a **`server.py`** file to edit and add instrumentation to.

To start the example server, run:

```
python2 server.py
```

You can stop the server by pressing **<Ctrl>-C**.

The Python client library README.md file will be helpful in instrumenting the service.

Lab 9.2 - Monitoring Your Instrumented Code

Let's monitor your own instrumented code with Prometheus so that you can better explore its metrics. Make sure that your demo API server is running for this section.

Add the following to the **`scrape_configs`** section in your **`prometheus.yml`**:

```
  - job_name: 'instrumentation-example'
    static_configs:
      - targets: ['localhost:12345']
```

Reload the Prometheus configuration by sending a **`HUP`** signal to the Prometheus process:

```
killall -HUP prometheus
```

Head to your Prometheus server's status page at **`http://<machine-ip>:9090/targets`** and verify that the instrumented example service target is being scraped successfully.

You can now explore its metrics in the Prometheus expression browser at **`http://<machine-ip>:9090/`**. Look for series with the **`job="instrumentation-example"`** label.

Lab 10.1 - Studying and Running an Example Exporter

In this lab, you will study and run a simple example exporter written in Python that exposes CPU metrics about the Linux machine it is running on. These metrics will be roughly equivalent to the CPU usage metrics of the Node Exporter, although they only split up CPU usage by mode, not additionally by CPU core. To retrieve the host's CPU metrics, the exporter uses the **`psutil`** Python library so that it does not need to parse the underlying virtual file (**`/proc/stat`**) on its own. Writing exporters works similarly in other languages, but Python allows us to illustrate a much shorter example than in a language like Go.

First, install the necessary Python dependencies:

```
export LC_ALL=C
sudo pip install prometheus_client
sudo pip install psutil
```

Change into your home directory and create a file named **`cpu_exporter.py`** with the following contents:

```
import time
import psutil

from prometheus_client import start_http_server
from prometheus_client.core import CounterMetricFamily, REGISTRY

class CPUCollector(object):
    def collect(self):
        c = CounterMetricFamily('cpu_exporter_cpu_usage_seconds_total', 'The
total number of CPU seconds used per mode.', labels=['mode'])
        for mode, usage_seconds in psutil.cpu_times()._asdict().items():
                c.add_metric([mode], usage_seconds)
```

```
        yield c

REGISTRY.register(CPUCollector())

if __name__ == '__main__':
    start_http_server(8000)
    while True:
        time.sleep(1)
```

Study the code above. The Prometheus client library's HTTP server is started on port **`8000`** by calling **`start_http_server(8000)`**. The HTTP server calls the **`collect()`** method of the **`CPUCollector`** class once per scrape, fetches the current CPU usage stats using the **`psutil`** library, and uses a **`CounterMetricFamily`** to translate those values into a "throw-away" Prometheus metric family (a group of series with the same metric name, but different labels) that is split out by the **`mode`** label dimension.

Start the demo CPU exporter:

```
python2 cpu_exporter.py
```

Add the following scrape configuration to the **`scrape_configs`** stanza of your **`prometheus.yml`** file:

```
  - job_name: 'cpu-exporter'
    static_configs:
      - targets:
        - 'localhost:8000'
```

Reload the Prometheus configuration by sending an **`HUP`** signal to the Prometheus process:

```
killall -HUP prometheus
```

The Prometheus server should now automatically start scraping your demo CPU exporter. Head to your Prometheus server's status page at **`http://<machine-ip>:9090/targets`** and verify that the targets for the demo job are now showing an **`UP`** state.

You can now query CPU usage from your own exporter instead of using the Prometheus Node Exporter:

```
rate(cpu_exporter_cpu_usage_seconds_total{job="cpu-exporter"}[1m])
```

Compare this with the equivalent expression based on the Node Exporter's CPU metrics:

```
sum without(cpu) rate(node_cpu_seconds_total{job="node"}[1m])
```

The CPU usage should look identical.

You now know how to build simple exporters in Python. You can build exporters for any other third-party system in a similar way.

Lab 11.1 - Histograms and Calculating Quantiles

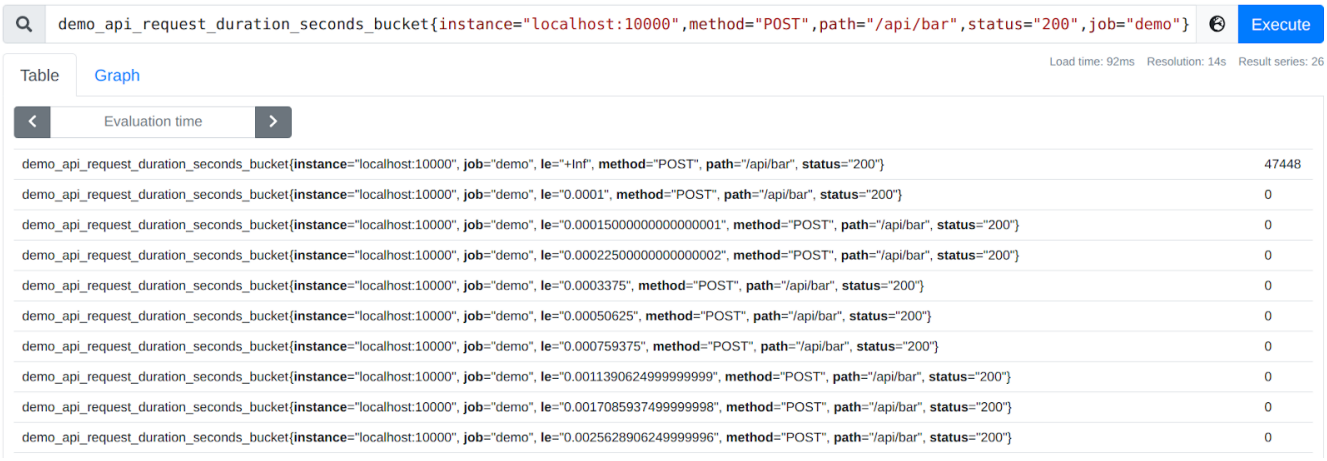
We already briefly mentioned histograms when instrumenting an example service. Histograms are a metric type that allow you to track the distribution of a set of values (like request durations) by categorizing the values into range buckets and counting how many events of each bucket type an application has observed so far. Prometheus histograms are cumulative, meaning that each bucket also contains the counts of previous buckets with lower range boundaries. Most frequently, histograms are used to track durations of operations like HTTP requests, so the first bucket might count request durations from 0 to 50ms, the next one 0 to 100ms, and so on. Finally, Prometheus histograms always contain a bucket that ranges from 0 to infinite, effectively counting *all* observed values (this is equivalent to just counting all requests that have occurred).

In Prometheus, histogram metrics are exposed as multiple time series (one for each bucket) with a **`_bucket`** metric name suffix and an **`le`** label that indicates the upper bucket boundary (**`le`** stands for "less than or equal"). Additionally, you may see unrelated label dimensions on the series that were added by the instrumentation user. Each combination of other labels will contain its own "sub-histogram", or full set of possible **`le`** bucket label values.

To see how this looks in practice, use the **`Table`** view in the expression browser to query for only the sub-histogram of the request durations for a particular combination of **`instance`**, **`method`**, **`path`**, and **`status`** labels:

```
demo_api_request_duration_seconds_bucket{instance="localhost:10000",method="POST
",path="/api/bar",status="200",job="demo"}
```

This will give you 26 output time series since the histogram has 26 buckets. The **`le`** label values will look a bit erratic, as they were auto-generated by a library helper function:



Querying a sub-histogram for a given path, method, instance, and status code

While a histogram gives you an idea of the distribution of request durations or can tell you how many of your requests take longer than e.g. 50ms to complete (provided you have a bucket with an upper boundary of 50ms), you often want to calculate quantiles (a more general form of percentiles) from them. For example, you might want to know in which time at least 99% of your requests complete.

In PromQL, you can compute quantiles from a histogram using the **`histogram_quantile()`** function. This function takes a desired quantile value as the first parameter and a histogram (set of series labeled with **`le`** labels) as the second parameter. Note that the quantile value has to be between 0 and 1, which would correspond to the percentiles 0 and 100.

To calculate the 90th percentile request latency in the demo service as measured over the last 5 minutes, query for:

```
histogram_quantile(0.9,
rate(demo_api_request_duration_seconds_bucket{job="demo"}[5m]))
```

You should see a detailed breakdown:

Graphing the 90th percentile latency for every sub-dimension

Note: It is important to take the **`rate()`** of a histogram before calculating quantiles from it. This ensures that only the bucket increases of the last e.g. 5 minutes are taken into account. Otherwise, you would get quantiles averaged over the entire lifetime of the bucket counters, not the latency "right now".

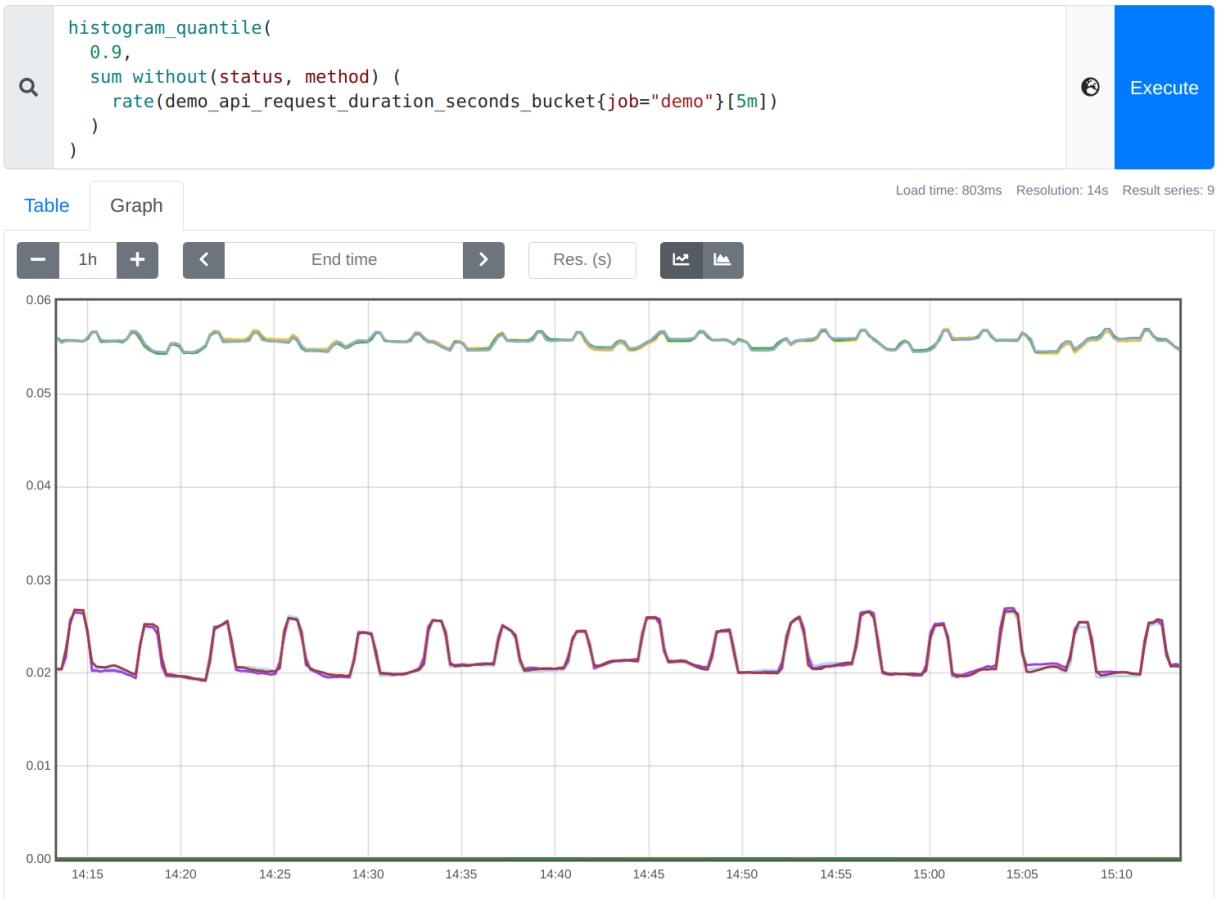
The above still gives you a detailed drill-down of 90th percentiles for every sub-dimension (path, method, etc.). Often you want to aggregate some dimensions away and just see an overall system latency. The good news is that all sub-histograms of a given histogram metric have the same bucket configuration, and each bucket is just a counter. This means that you can just sum up the corresponding bucket increase rates over a given set of dimensions to arrive at a valid aggregated histogram (which we can then feed into **`histogram_quantile()`**).

For example, to calculate the 90th percentile latency with the **`status`** and **`method`** dimensions aggregated away, query for:

```
histogram_quantile(
  0.9,
  sum without(status, method) (
```

```
    rate(demo_api_request_duration_seconds_bucket{job="demo"}[5m])
  )
)
```

You should now see an aggregated view:



Graphing an aggregated 90th percentile latency

Note: Be careful to not aggregate away the `le` label. The `histogram_quantile()` function requires this label to be present to interpret the histogram buckets.

This kind of query allows you to choose the quantile, the aggregation level, as well as the time to average over when computing quantiles from histograms.

Since Prometheus histograms have a limited number of buckets (since each bucket incurs a cost in the number of time series), there will always be an error associated when converting histograms into quantile values. You can find details about this in the Prometheus documentation on errors of quantile estimation.

Lab 11.2 - Filtering by Sample Value

You already learned earlier how to select series by their metric name and label values. Sometimes you will want to filter series by their sample value as well. The most obvious use case for this is for thresholds in alerting expressions.

PromQL allows you to do this by adding a binary comparison operator between a time series vector and a scalar number. For example, to select only the request rates for **`status="500"`** errors that are above **`0.2`** per second, you can query for:

```
rate(demo_api_request_duration_seconds_count{status="500",job="demo"}[5m]) > 0.2
```

The result should look like this:

Filtering error rates by sample value

This works similarly for other comparison operators you may be used to from programming languages: **`<`**, **`<=`**, **`>=`**, **`!=`**, and **`==`**.

You can also compare entire sets of time series with the same label-matching behavior that arithmetic binary operators have. This can be useful to correlate commonly labeled series. For example, the following query selects all **`status="500"`** error rates that are at least 50 times larger than the total traffic for a given **`path`**, **`method`**, and **`instance`** combination:

```
    rate(demo_api_request_duration_seconds_count{status="500",job="demo"}[5m]) *
50
> ignoring(status)
    sum without(status)
(rate(demo_api_request_duration_seconds_count{job="demo"}[5m]))
```

The result should look like this (you may see more or fewer series, depending on current request and error rates):

Filtering error rates by total request rates

Lab 11.3 - Performing Set Operations

You just learned how to correlate and filter two time series vectors based on their sample values. Sometimes it can be useful to correlate two series vectors based on the mutual *presence* of certain label sets in each of the vectors and modify the output set accordingly. PromQL offers three set operators for this: **`and`**, **`or`**, and **`unless`**.

The **`and`** operator allows you to select only series from a vector that also have an equivalently labeled series in a second vector (set intersection). This is useful for expressing more complex filter conditions in queries. For example, to graph the **`status="500"`** error rates only for those **`path`**, **`method`**, and **`instance`** combinations that have a total request rate larger than 10 per second, query for:

```
  sum without(status)
(rate(demo_api_request_duration_seconds_count{job="demo",status="500"}[5m]))
and
  sum without(status)
(rate(demo_api_request_duration_seconds_count{job="demo"}[5m])) > 10
```

This will give you a filtered-down list of error rates:

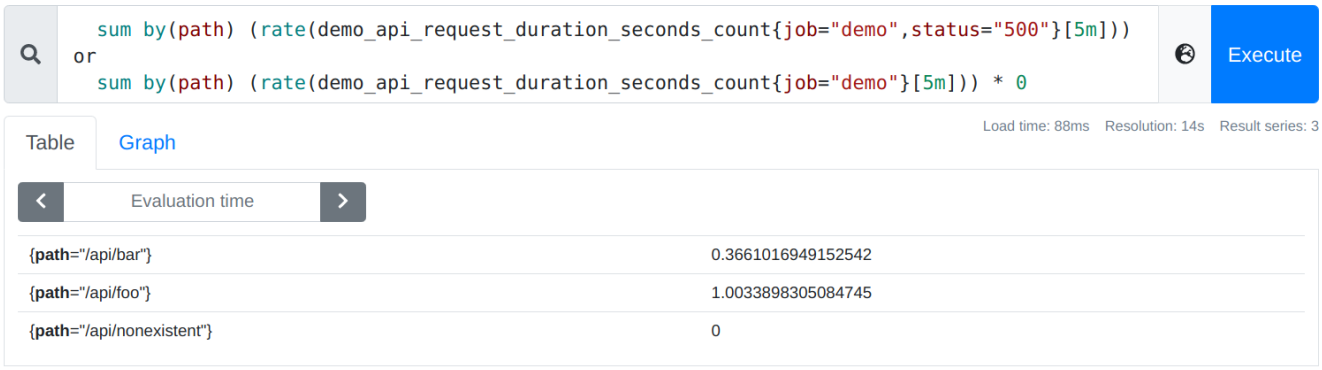
*Filtering error rates using the **`and`** operator*

The **`or`** operator allows you to select label sets from two time series vectors which are in *either* of the two vectors (set union). Each resulting sample value will be taken from the left-hand side of the operation if a given label set is present there, and from the right-hand side otherwise. This is often useful when you are missing certain label combinations in one vector and want to fill them in with alternative values originating from another vector which you know has all label combinations.

A somewhat contrived example: If you wanted to query the **`status="500"`** error rate for all paths that have ever received HTTP requests (even ones that never encountered a **`500`** error), you would typically run into the problem that no series exist at all for those paths that never produced a **`500`** error (so you would not see a rate of **`0`** for these paths, but no series at all). Using the **`or`** operator, you can fill in the paths that have ever handled any requests at all and set their rate value to `0` using a multiplication trick:

```
  sum by(path)
(rate(demo_api_request_duration_seconds_count{job="demo",status="500"}[5m]))
or
  sum by(path) (rate(demo_api_request_duration_seconds_count{job="demo"}[5m])) *
0
```

You should now also see a **`500`** rate for the **`/api/nonexistent`** path, which only ever results in **`404`** status codes since it does not exist:



*Filling in missing series using the **`or`** operator*

Finally, there is also an **`unless`** operator that only propagates label sets from one vector into the output if they are *not* found in the second vector (set complement).

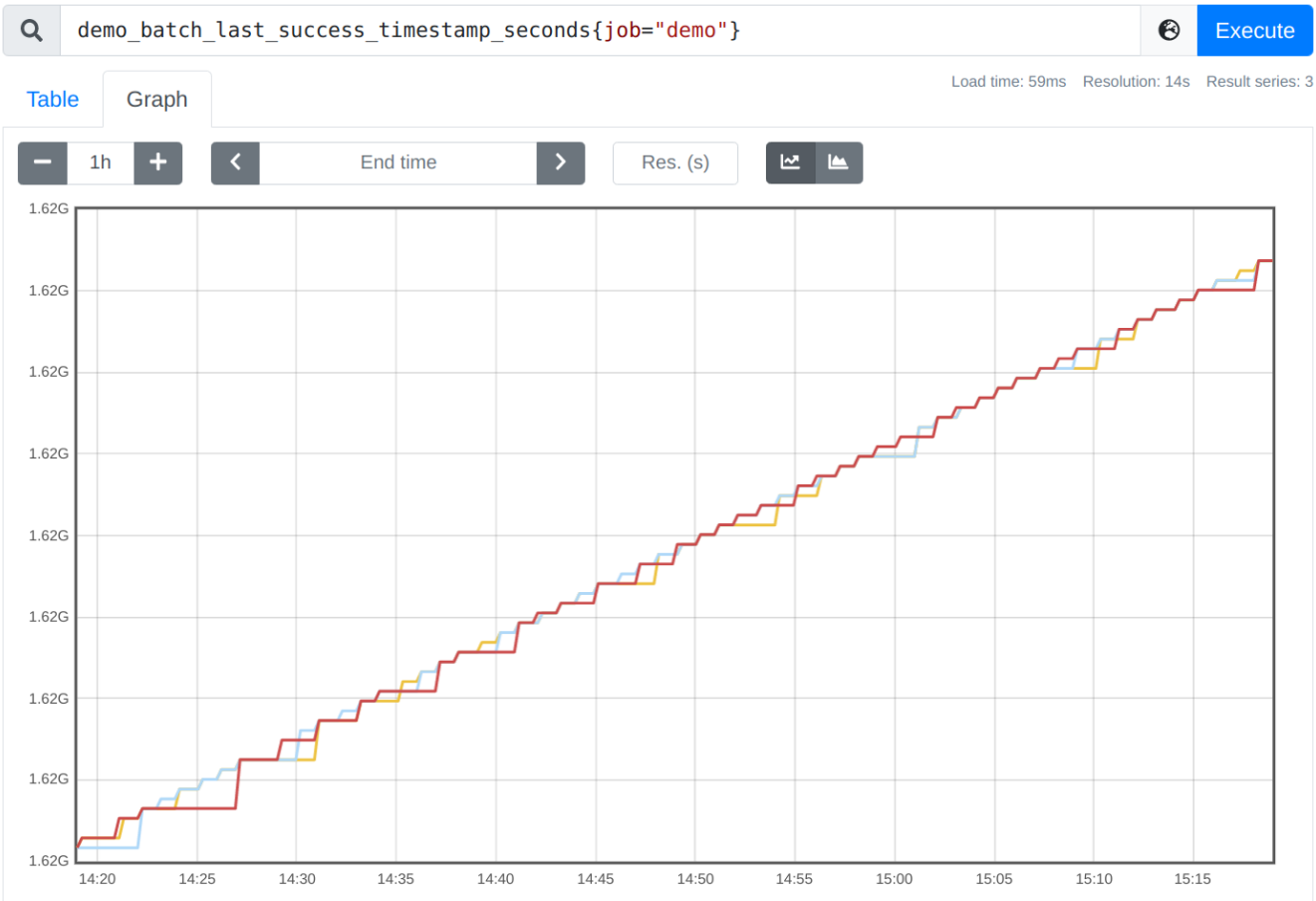
Lab 11.4 - Working with Timestamp Metrics

For tracking the time when some event has happened, services and exporters frequently encode a Unix timestamp in seconds into the *sample value* (not the sample timestamp!) of a metric. This could be the boot time of a machine, the last time a batch job finished a successful run, or when the last garbage collection cycle happened.

For example, the `demo` service exposes a metric that tells you when the last successful run of its internal batch job happened:

```
demo_batch_last_success_timestamp_seconds{job="demo"}
```

When you graph this directly, you will see a "staircase" graph in which the timestamp remains constant for a while and then jumps up every time that the batch job run succeeds:



Graphing the raw timestamp of each last successful batch job run

In monitoring, you are often more interested in the duration since something has happened, rather than the absolute timestamp. To calculate the age of a timestamp in PromQL, simply subtract the timestamp metric from the current time as provided by the **`time()`** function:

```
time() - demo_batch_last_success_timestamp_seconds{job="demo"}
```

This will give you the number of seconds since the last successful batch job run:

Graphing the age of each last successful batch job run

This kind of sawtooth graph is easy to analyze with the eye. Lines that go up too far represent batch jobs which are far overdue, while any drop in a line to `0` represents a successful batch job completion.

Combining this expression with a value threshold allows us to detect batch jobs that have not finished within the last 1.5 minutes:

```
time() - demo_batch_last_success_timestamp_seconds{job="demo"} > 1.5 * 60
```

You could use this kind of filter expression to alert on batch jobs that have not run successfully in a while.

Lab 11.5 - Offsetting Data

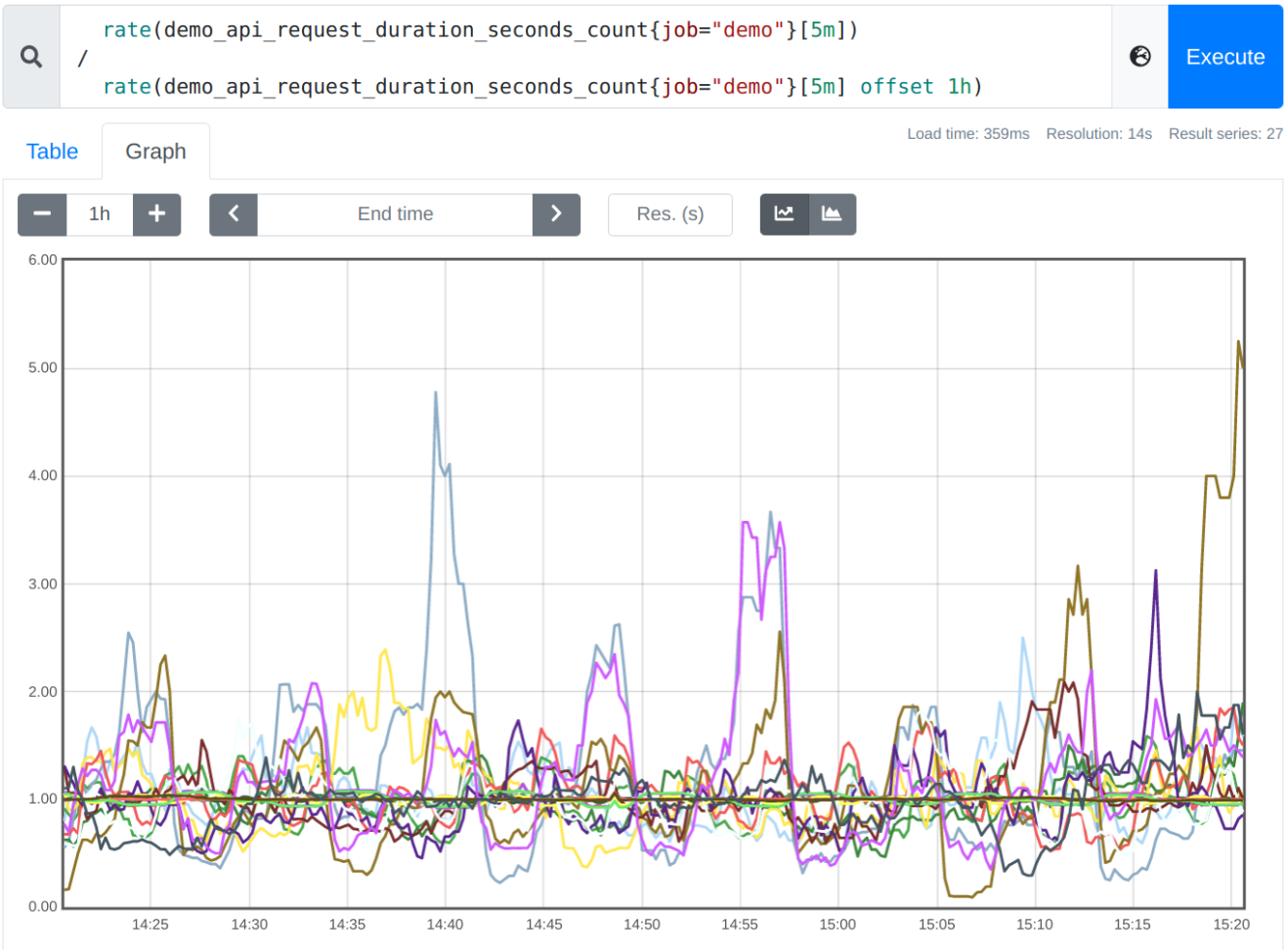
Sometimes you want to compare current system behavior to past behavior in one PromQL expression. For example, you might want to see what the difference (or ratio) is between the current request rate and that of a week ago on the same day. To do this, PromQL offers an **`offset <duration>`** modifier that you can add to any series selector, which time-shifts the requested data range into the past. For example, to get the **`demo`** service request rate one hour ago (relative to each evaluation point in the graph), you can query for:

```
rate(demo_api_request_duration_seconds_count{job="demo"}[5m] offset 1h)
```

You could now calculate a ratio between each current rate and the rate one hour ago:

```
  rate(demo_api_request_duration_seconds_count{job="demo"}[5m])
/
  rate(demo_api_request_duration_seconds_count{job="demo"}[5m] offset 1h)
```

It will look something like this, hovering around `1`:



Graphing the ratio of current and past request rates

In a real service, you might use an expression like this to get an idea of whether a request rate is significantly different from the rate at the same time and day of the last week (using **`offset 7d`**).

Note that you can only use the **`offset`** modifier directly after an instant or range vector selector (**`foo`** or **`foo[5m]`**), not after any arbitrary sub-expression.

Lab 11.6 - Sorting and Limiting

In the **`Table`** view of the expression browser it's sometimes useful to sort query results by their sample value. For example, you may want to show the total memory usage of every service on a cluster, sorted descendingly, so you can easily spot the biggest memory users. You can use the **`sort()`** or **`sort_desc()`** functions to achieve this. The former sorts ascendingly, while the latter sorts descendingly.

To see the request rates for each path in our **`demo`** service, sorted descendingly by the magnitude of the rate, query for:

```
sort_desc(sum by(path)
(rate(demo_api_request_duration_seconds_count{job="demo"}[5m])))
```

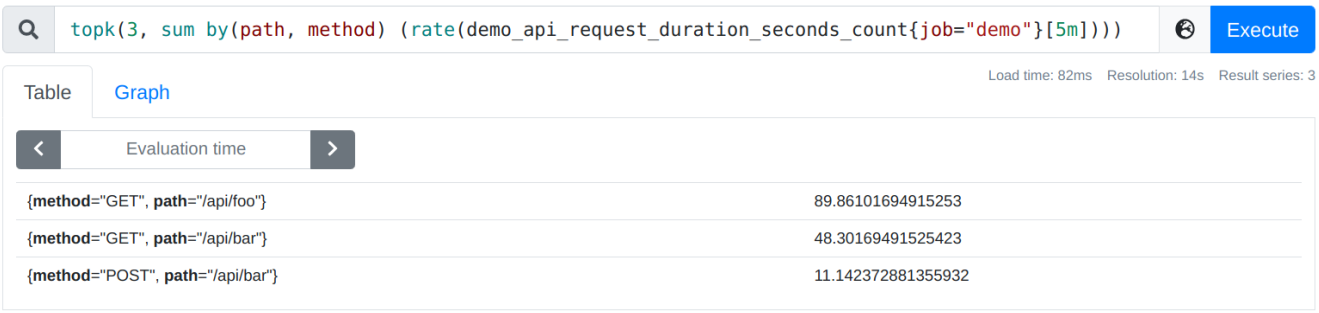
You should give you a sorted result:

Note: Sorting does not have any effect on graphs, since the X and Y values of each series are determined independently of the order in which each time series is returned in a query result.

To show only a certain number of top **`k`** or bottom **`k`** results of a query (where **`k`** is an integer), you can use the **`topk(k, ...)`** or **`bottomk(k, ...)`** aggregation operators. These operators take the number of output values you want to show as their first parameter and an instant vector expression to limit as their second parameter.

For example, to show the top 3 request rates for each **`path`** and **`method`** combination, query for:

```
topk(3, sum by(path, method)
(rate(demo_api_request_duration_seconds_count{job="demo"}[5m])))
```



Graphing the top 3 request rates

Note: One confusing aspect when graphing **`topk()`** or **`bottomk()`** is that the total number of series displayed in the graph can be different from the specified **`k`** value. For one, it can show fewer than **`k`** series if there were not enough input series to begin with. Since the entire PromQL expression is also evaluated independently at every evaluation step of the graph (without any knowledge of the total range of the graph), it may also return a different set of top or bottom **`k`** series at each step in the graph, in which case you will see *more* than **`k`** series in total.

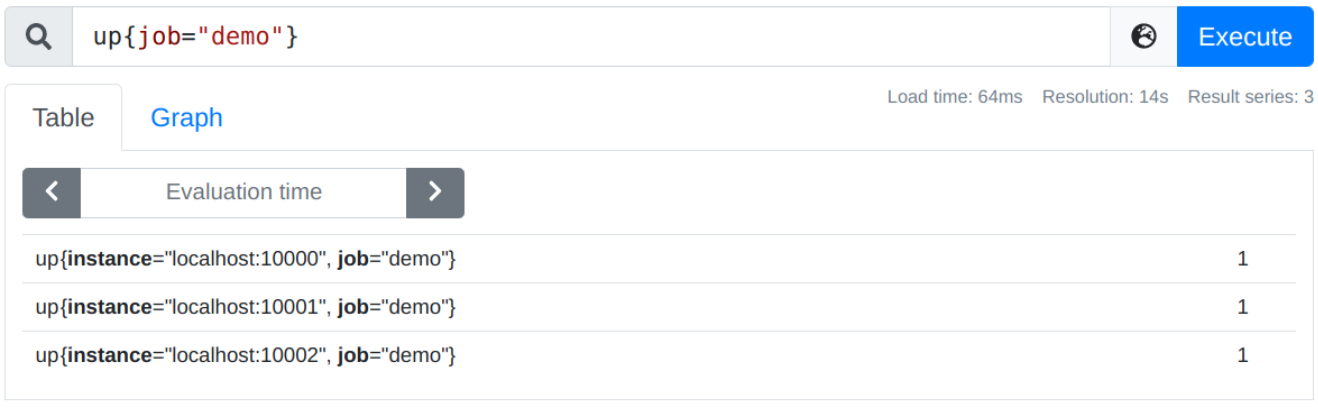
Lab 11.7 - Inspecting Scrape Health

Since Prometheus is a pull-based system, it notices when it cannot scrape a target and records this fact in a metric. For every scrape, Prometheus records a synthetic **`up`** metric with the labels of the scrape target. It sets the sample value to `1` if the scrape succeeded and to `0` if the scrape failed for whatever reason. You can use this fact to query basic availability information about each job's targets.

Verify that all instances of the **`demo`** service are up by querying:

```
up{job="demo"}
```

All three instances should have a sample value of `1`, meaning that they could all be scraped:

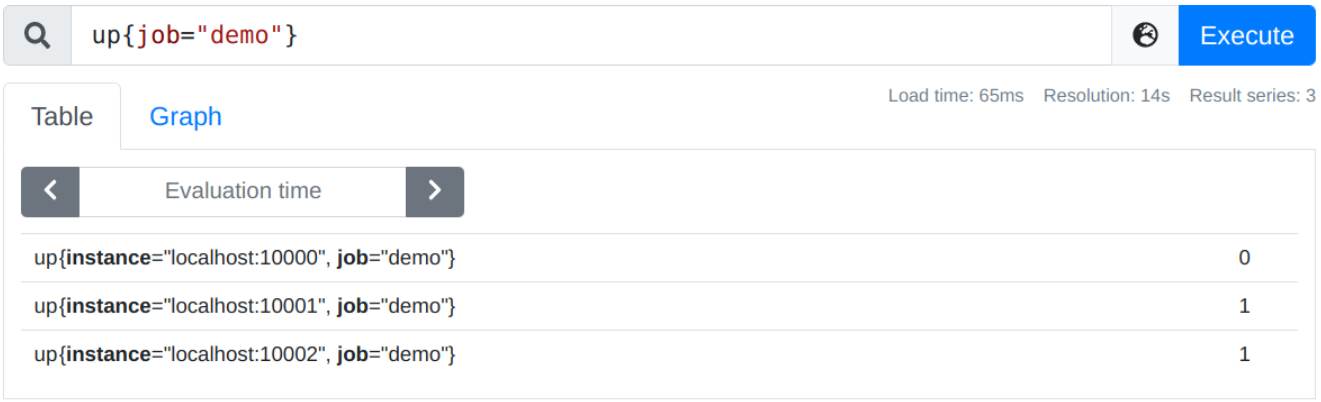


*Querying the scrape health of the **`demo`** job*

Let's kill one of the three instances and see how the result changes:

```
pkill -f -- -listen-address=:10000
```

You should now see one down instance:



Querying the scrape health of the `demo` job with one instance down

You could also filter this list to see only the down instances:

```
up{job="demo"} == 0
```

You can use the `up` metric to formulate different variants of basic scrape availability alerts. For example, if you wanted to alert when more than half of all demo instances are down, you could use the following expression:

```
  count by(job) (up{job="demo"} == 0)
/
  count by(job) (up{job="demo"})
> 0.5
```

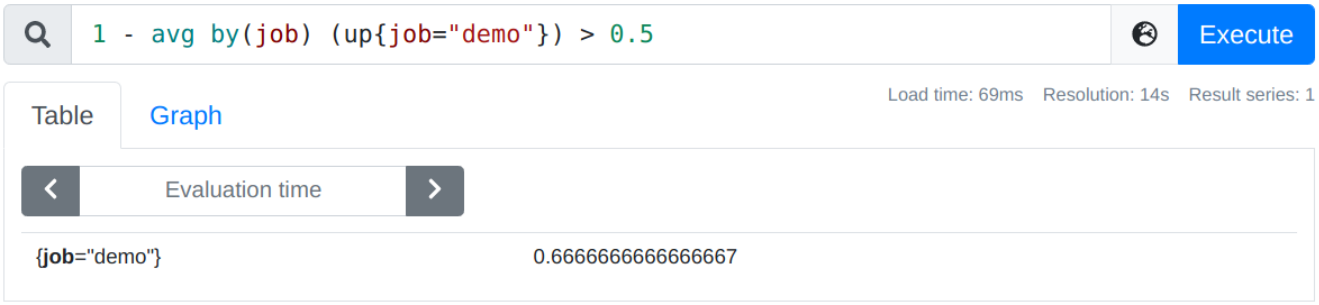
Since the `up` metric can only take the values `0` or `1`, the following (simpler) query is equivalent:

```
1 - avg by(job) (up{job="demo"}) > 0.5
```

Either of those queries should give you no output. Try killing one more demo service instance:

```
pkill -f -- -listen-address=:10001
```

Now, the expression should start returning a result for a `demo` job:



Querying whether more than half of all instances are down

Finally, start the killed instances again:

```
./prometheus_demo_service -listen-address=:10000 &
./prometheus_demo_service -listen-address=:10001 &
```

Note: The `count()` operator returns an empty result instead of `0` if there are no matching `up` series at all for a given job, since any result has to be an aggregation of existing series.

Lab 12.1 - Using Relabeling to Drop Samples

Relabeling is a wide field. Instead of diving in too deep into different relabeling use cases in this chapter, we will study more examples of relabeling in later chapters, when we learn about blackbox monitoring, service discovery, alerting, and using Prometheus with Kubernetes.

For now, let's try a simple example in which we will use relabeling to selectively drop some metrics of the **`demo`** service that we do not want to store. The **`demo`** service exposes four classes of metrics:

- Application-specific metrics, prefixed with **`demo_`**.
- Go runtime-specific metrics, prefixed with **`go_`**.
- Generic process metrics, prefixed with **`process_`**.
- Generic HTTP-level metrics, prefixed with **`http_`**.

Let's say you only wanted to store the application-specific and HTTP-level metrics. We could then use metric relabeling to throw away any samples where the metric name does not start with either **`demo_`** or **`http_`** by matching a regular expression against the special **`__name__`** label that contains a sample's metric name.

Add the following **`metric_relabel_configs`** section to the **`demo`** service's **`scrape_config`** section in your **`prometheus.yml`**:

```
    metric_relabel_configs:
      - action: keep
        source_labels: [__name__]
        regex: '(demo_|http_).*'
```

This keeps only the samples whose metric name matches the regular expression **`(demo_|http_).*`** and thus only stores application-level and HTTP-level metrics.

Make sure that Prometheus reloads its configuration file:

```
killall -HUP prometheus
```

Query the following expression in the Table view of the Prometheus expression browser:

```
{job="demo"}
```

This should now only show series with the expected matching metric names.

Note: Using relabeling on individual samples like this still requires producing and transfering all samples to the Prometheus server before they can be filtered. If you are using this to filter large groups of metrics, it is often better to fix the metrics source to only expose the metrics you are interested in.

Lab 13.1 - Setting Up Consul-Based Discovery

A common tool that organizations use to register and discover hosts and services is Hashicorp's Consul. In the following lab, you will use Prometheus's native support for Consul-based service discovery to monitor your existing three **`demo`** service instances.

In a new terminal, change to your home directory and download Consul version 1.9.3:

```
wget https://releases.hashicorp.com/consul/1.9.3/consul_1.9.3_linux_amd64.zip
```

Extract the archive:

```
unzip consul_1.9.3_linux_amd64.zip
```

This should extract a single binary executable file named **`consul`**.

Create a configuration file named **`demo-service.json`** in the same directory that defines each of your three **`demo`** service instances for Consul:

```
{
  "services": [
    {
      "id": "demo1",
      "name": "demo",
      "address": "127.0.0.1",
      "port": 10000
    },
    {
      "id": "demo2",
      "name": "demo",
      "address": "127.0.0.1",
```

```
      "port": 10001
    },
    {
      "id": "demo3",
      "name": "demo",
      "address": "127.0.0.1",
      "port": 10002
    }
  ]
}
```

Run the Consul agent in development mode while providing the above file as its configuration:

```
./consul agent -dev -config-file=demo-service.json
```

In addition to the three **`demo`** service targets, the Consul agent will also register itself as a single-instance service with the name **`consul`**.

To make Prometheus discover the **`demo`** service targets from Consul, add the following scrape configuration to the **`scrape_configs`** stanza of your **`prometheus.yml`** file:

```
  - job_name: 'consul-sd-demo'
    consul_sd_configs:
      - server: 'localhost:8500'
    relabel_configs:
      - action: keep
        source_labels: [__meta_consul_service]
        regex: demo
```

Note that this is also adding a relabeling step to only keep targets where the Consul service name is **`demo`**. Otherwise, Prometheus would try to scrape the Consul agent itself as well, which doesn't have a **`/metrics`** endpoint.

Reload the Prometheus configuration by sending a **`HUP`** signal to the Prometheus process:

```
killall -HUP prometheus
```

Head to your Prometheus server's targets page at **`http://<machine-ip>:9090/targets`** and verify that the targets defined above are present and healthy under the **`consul-sd-demo`** job name.

Lab 13.2 - Configuring File-Based Discovery

To make Prometheus read targets from a file named **`targets.yml`**, add the following scrape configuration to the **`scrape_configs`** stanza of your **`prometheus.yml`** file:

```
  - job_name: 'file-sd-demo'
    file_sd_configs:
      - files:
        - 'targets.yml'
```

In the same directory as your **`prometheus.yml`**, create a **`targets.yml`** file with the following contents:

```
- targets:
    - 'localhost:10000'
    - 'localhost:10001'
  labels:
    env: production
- targets:
    - 'localhost:10002'
  labels:
    env: staging
```

This "discovers" your existing three **`demo`** services under a second scrape job named **`file-sd-demo`** and labels the first two instances with the **`env="production"`** label and the third one with **`env="staging"`**.

Reload the Prometheus configuration by sending a **`HUP`** signal to the Prometheus process:

```
killall -HUP prometheus
```

Head to your Prometheus server's targets page at **`http://<machine-ip>:9090/targets`** and verify that the targets defined above are present, healthy, and have the expected labels under the **`file-sd-demo`** job name.

You can now try changing the contents of **`targets.yml`** (such as changing the **`env`** label for one of the target groups) without reloading the Prometheus configuration. Prometheus will watch the file and should pick up any changes automatically.

Note: When updating a **`file_sd`** targets file of a production Prometheus server, ensure that you make any file changes atomically to avoid Prometheus seeing partially updated contents. The best way to ensure this is to create the updated file in a separate location and then rename it to the destination file name (using the **`mv`** shell command or the **`rename()`** system call).

You now have a generic mechanism that lets you dynamically change the targets that Prometheus monitors without having to restart or explicitly reload the Prometheus server. Instead of making manual changes to the **`targets.yml`** file, you would now build an integration that reads target information from your infrastructure and updates the **`targets.yml`** file accordingly.

Lab 14.1 - Setting Up Blackbox Exporter

In a new terminal, change into your home directory and download Blackbox Exporter 0.18.0 for Linux:

```
wget
https://github.com/prometheus/blackbox_exporter/releases/download/v0.18.0/blackb
ox_exporter-0.18.0.linux-amd64.tar.gz
```

Extract the tarball:

```
tar xvfz blackbox_exporter-0.18.0.linux-amd64.tar.gz
```

Change into the extracted directory:

```
cd blackbox_exporter-0.18.0.linux-amd64
```

Replace the existing **`blackbox.yml`** configuration file with the following simplified contents (see the Blackbox Exporter configuration documentation for more details):

```
modules:
  http_2xx:
    prober: http
    timeout: 2s
    http:
      method: GET
      preferred_ip_protocol: "ip4"
```

This configures the Blackbox Exporter to allow probing targets over HTTP. The module name (**`http_2xx`**) can be freely chosen, but needs to match the value that Prometheus sends in the **`module`** HTTP parameter during a scrape to the Blackbox Exporter. This is how Prometheus chooses which type of probe to run through the exporter.

Note: The timeout for a probe from the Blackbox Exporter to a backend target needs to be smaller than Prometheus's own timeout when scraping the exporter. Otherwise, Prometheus will see the Blackbox Exporter as down (with an **`up`** metric value of **`0`**), instead of receiving information about the backend target being down.

Start the Blackbox Exporter:

```
./blackbox_exporter
```

By default, the Blackbox Exporter will read its configuration from **`blackbox.yml`** and it will listen on port **`9115`**. Try heading to **`http://<machine-ip>:9115/probe?target=https://prometheus.io&module=http_2xx`** to see an example metrics output for an HTTP probe against https://prometheus.io.

Lab 14.2 - Probing Example Services

In this example, you will configure Prometheus to probe some websites using HTTP(S) through the Blackbox Exporter. This is where you will apply the aforementioned relabeling rules to always scrape the Blackbox Exporter, but pass the service targets to probe to it via a **`target`** query parameter.

Add the following to the **`scrape_configs`** section in your **`prometheus.yml`**:

```
  - job_name: 'blackbox'
    metrics_path: /probe
    params:
      module:
        - http_2xx # Look for an HTTP 200 response.
    static_configs:
      - targets: # Targets to probe through the Blackbox Exporter.
        - 'http://prometheus.io'
        - 'https://prometheus.io'
        - 'http://example.com:8080'
    relabel_configs:
      - source_labels: [__address__]
        target_label: __param_target
      - source_labels: [__param_target]
        target_label: instance
      - target_label: __address__
        replacement: 127.0.0.1:9115  # Blackbox Exporter address.
```

Reload the Prometheus configuration by sending a **`HUP`** signal to the Prometheus process:

```
killall -HUP prometheus
```

The Prometheus server should now automatically start scraping the Blackbox Exporter. Head to your Prometheus server's targets page at **`http://<machine-ip>:9090/targets`** and verify that the discovered service probe targets are visible there.

Note: You should see all three Blackbox Exporter probe targets in the **`UP`** state, but that does not mean that the target probes succeeded. It only means that the scrape of the Blackbox Exporter itself worked, as the exporter can even return useful metrics about the service probe if the probe failed. The exact metrics returned about the backend probe depend on the Blackbox Exporter module in either case.

In the case of the **`http`** prober module, several metrics related to the probe are returned, for example:

- **`probe_duration_seconds`**: How many seconds the backend probe took
- **`probe_success`**: Whether the probe was overall successful (**`0`** or **`1`**)
- **`probe_http_status_code`**: The backend's HTTP status code
- **`probe_http_content_length`**: The length in bytes of the backend HTTP response content
- **`probe_http_redirects`**: How many redirects were followed
- **`probe_http_ssl`**: Whether the probe used SSL in the final redirect (**`0`** or **`1`**)
- **`probe_ssl_earliest_cert_expiry`**: In the case of an SSL request, the Unix timestamp of the certificate expiry
- **`probe_ip_protocol`**: The IP protocol version used for the backend probe (**`4`** or **`6`**)

Head to **`http://<machine-ip>:9090/graph`** and evaluate the following expression:

```
probe_success{job="blackbox"}
```

You should see a value of **`0`** (unsuccessful probe) for the **`http://example.com:8080`** target and a **`1`** for the other targets. Note that **`http://prometheus.io`** redirects permanently to **`https://prometheus.io`**. The Blackbox exporter follows redirects, and thus even the **`http://prometheus.io`** target probe will include SSL/TLS-related metrics like certificate expiry.

You can find out in how many days the SSL certificate for prometheus.io will expire by querying for:

```
(
    probe_ssl_earliest_cert_expiry{instance="https://prometheus.io"}
  -
    time()
) / 86400
```

This kind of expression is useful for alerting on soon-to-expire certificates.

Try querying for the other metrics mentioned above to get more insight about probe durations and other probe-related metadata.

For more information on configuring the Blackbox Exporter and using its other modules, see the Blackbox Exporter documentation.

Lab 15.1 - Running the Pushgateway

Let's download and run the Pushgateway.

In a new terminal, change to your home directory and download Pushgateway version 1.4.0 for Linux:

```
wget
https://github.com/prometheus/pushgateway/releases/download/v1.4.0/pushgateway-1
.4.0.linux-amd64.tar.gz
```

Extract the archive:

```
tar xvfz pushgateway-1.4.0.linux-amd64.tar.gz
```

Change into the extracted directory:

```
cd pushgateway-1.4.0.linux-amd64
```

The Pushgateway does not require further configuration. If you wanted to persist pushed metrics across restarts in a file, you could provide a value to the **`--persistence.file`** command-line flag. For now, you can omit this flag to only keep metrics in memory.

Run the Pushgateway:

```
./pushgateway
```

The Pushgateway listens on port 9091 by default. Head to **`http://<machine-ip>:9091/`** to inspect its web interface. There should not be any pushed metrics visible yet.

Lab 15.2 - Monitoring the Pushgateway

The Pushgateway exposes metrics about itself, as well as about the groups of metrics that other jobs have pushed to it under the **`/metrics`** HTTP path. Thus, you can simply scrape it like any other target.

Add the following list entry to the **`scrape_configs`** section in your **`prometheus.yml`** to scrape the Pushgateway:

```
  - job_name: 'pushgateway'
    honor_labels: true
    static_configs:
      - targets: ['localhost:9091']
```

Note the **`honor_labels: true`** scrape option. Because the Pushgateway proxies metrics from other jobs that usually already attach their own **`job`** label to a group of metrics, you will want to prevent Prometheus from overwriting any such labels with the target labels from the scrape configuration. The Pushgateway documentation explains this in more detail.

Make sure that Prometheus reloads its configuration file:

```
killall -HUP prometheus
```

Head to your Prometheus server's targets page at **`http://<machine-ip>:9090/targets`** to verify that it is scraping the Pushgateway correctly.

Lab 15.3 - Pushing Metrics from a Batch Job

Rather than deploying a real batch job, let's simulate one using **`curl`**. This batch job simulates deleting a random number of users, and then pushes the timestamp of its last successful run, the timestamp of its last overall run (successful or not), as well as the number of deleted users to the Pushgateway. Exposing such timestamps proves to be useful for monitoring and alerting on the overall health of periodic batch job runs, as you will see later.

Execute a simulated batch job run using the following command:

```
cat <<EOF | curl --data-binary @-
http://localhost:9091/metrics/job/demo_batch_job
# TYPE demo_batch_job_last_successful_run_timestamp_seconds gauge
# HELP demo_batch_job_last_successful_run_timestamp_seconds The Unix timestamp
in seconds of the last successful batch job run.
demo_batch_job_last_successful_run_timestamp_seconds $(date +%s)
# TYPE demo_batch_job_last_run_timestamp_seconds gauge
# HELP demo_batch_job_last_run_timestamp_seconds The Unix timestamp in seconds
of the last successful batch job run.
demo_batch_job_last_run_timestamp_seconds $(date +%s)
# TYPE demo_batch_job_users_deleted gauge
# HELP demo_batch_job_users_deleted How many users were deleted in the last
batch job run.
demo_batch_job_users_deleted $RANDOM
EOF
```

Note that the command embeds the **`$(date +%s)`** command output above to generate up-to-date Unix timestamps for the timestamp-based metrics on each run. It also uses the special **`$RANDOM`** shell variable to generate random numbers between 0 and 32767 for the number of deleted users.

Head to **`http://<machine-ip>:9091/`** to verify that the metrics are visible in the Pushgateway's web interface. There should now be one metrics group with the grouping label **`job="demo_batch_job"`**. Click on the group to expand it and view details about its metrics.

Run the above command multiple times to simulate successive runs of the batch job and verify that the data in the Pushgateway is updated.

Lab 15.4 - Working with Batch Job Metrics

Now you should be able to see the batch job metrics in Prometheus. Run the following query in Prometheus:

```
time() - demo_batch_job_last_successful_run_timestamp_seconds
```

This tells you how many seconds ago the last successful run happened.

You could now use this to find batch jobs that have not run successfully within the last hour:

```
time() - demo_batch_job_last_successful_run_timestamp_seconds > 60*60
```

Finally, when a batch job is deprovisioned for good, you will want to delete its metrics from the Pushgateway so that Prometheus stops scraping them. You can either do this from the Pushgateway's web interface, or you can delete a metrics group using the Pushgateway's web API and the **`DELETE`** HTTP method. Here, you will use the web API.

Run the following command to delete the **`demo_batch_job`** metrics from the Pushgateway:

```
curl -XDELETE http://localhost:9091/metrics/job/demo_batch_job
```

Verify in the Pushgateway web interface that the metrics have been deleted.

Lab 16.1 - Downloading and Configuring Alertmanager

In a new terminal, change to your home directory and download Alertmanager 0.21.0 for Linux:

```
wget
https://github.com/prometheus/alertmanager/releases/download/v0.21.0/alertmanage
r-0.21.0.linux-amd64.tar.gz
```

Extract the tarball:

```
tar xvfz alertmanager-0.21.0.linux-amd64.tar.gz
```

Change into the extracted directory:

```
cd alertmanager-0.21.0.linux-amd64
```

Before starting the Alertmanager, create a simple configuration file for it. Save the following in a file named **`alertmanager.yml`** (replace the existing default file contents):

```
route:
  group_by: ['alertname', 'job']
  group_wait: 30s
  group_interval: 5m
  repeat_interval: 3h

  receiver: test-receiver
```

This configuration groups all incoming alerts by their **`alertname`** and **`job`** labels and waits 30 seconds (**`group_wait`**) before sending a notification for a grouping to see if other alerts for the same group will still arrive. It then sends a notification containing all alerts in that group. In case new alerts for the same group arrive after that, it will send another notification after five more minutes

(**`group_interval`**). Finally, if any alerts in the group are still firing three hours later (**`repeat_interval`**), it will resend the notification another time.

In this simple demo configuration, there is only one top-level route that routes all alerts to a single receiver called **`test-receiver`**. You are going to configure this receiver in the following sections.

Alternative A: Sending Notifications to Slack

If you have a Slack account (or would like to create a Slack workspace for this course), you can send alert notifications to a channel on Slack. Otherwise, follow the *Alternative B* section below to send notifications to a generic webhook receiver, which does not require any external accounts.

Create a Slack channel with an incoming webhook integration:

- Create a Slack channel called **`#alertmanager`** in your Slack workspace.
- Head to Slack's "Incoming WebHooks" app page.
- Click the "Add Configuration" button.
- Choose the **`#alertmanager`** channel in the "Post to Channel" dropdown list.
- Click "Add Incoming WebHooks Integration".
- Slack should now give you a webhook URL of the form: **`https://hooks.slack.com/services/T00000000/B00000000/XXXXXXXXXXXXXXXXXXXXXXXX`**

Add the following receiver configuration to the top level of your **`alertmanager.yml`** file:

```
receivers:
- name: test-receiver
  slack_configs:
  - api_url: '<slack webhook URL>'
    username: 'Demo Alertmanager'
    channel: '#alertmanager'
    send_resolved: true
```

IMPORTANT: Replace the **`<slack webhook URL>`** placeholder with the webhook URL you just received from Slack.

This receiver configuration sends all notifications to the `#alertmanager` Slack channel as the user "Demo Alertmanager". Besides sending messages when alerts are firing, it will also notify you when a group of alerts have been marked as resolved.

Alternative B: Sending Notifications to a Webhook

If you do not have access to a Slack workspace, you can configure Alertmanager to send notifications to a generic webhook receiver as a JSON object. The webhook receiver can then choose to implement arbitrary actions based on the incoming data. In this example, you are going to run a simple Python program that receives webhook notifications and just prints information about all received alert notifications to the terminal it is running on.

In a new terminal, change to your home directory and create a file called **`webhook_receiver.py`** with the following contents:

```
import json
from BaseHTTPServer import HTTPServer, BaseHTTPRequestHandler

class AlertHandler(BaseHTTPRequestHandler):
  def do_POST(self):
    content_len = int(self.headers.getheader('Content-Length', 0))
    data = json.loads(self.rfile.read(content_len))
    print "Received %u %s alerts:" % (len(data["alerts"]), data["status"])
    print "\tGrouping labels:"
    for k, v in data['groupLabels'].items():
      print "\t\t%s = %r" % (k, v)
    print "\tCommon labels:"
    for k, v in data['commonLabels'].items():
      print "\t\t%s = %r" % (k, v)
    print "\tCommon annotations:"
    for k, v in data['commonAnnotations'].items():
      print "\t\t%s = %r" % (k, v)
    print "\t\tAlert details:"
    for idx, alert in enumerate(data['alerts']):
      print "\t\t\tAlert %u:" % idx
      print "\t\t\t\tLabels: %r" % alert['labels']
      print "\t\t\t\tAnnotations: %r" % alert['annotations']

    self.send_response(200)
    self.end_headers()

if __name__ == '__main__':
   httpd = HTTPServer(('', 9595), AlertHandler)
   httpd.serve_forever()
```

Start the webhook receiver script on port **`9595`**:

```
python2 webhook_receiver.py
```

Add the following receiver configuration to the top level of your **`alertmanager.yml`** file:

```
receivers:
- name: test-receiver
  webhook_configs:
  - url: http://localhost:9595/
```

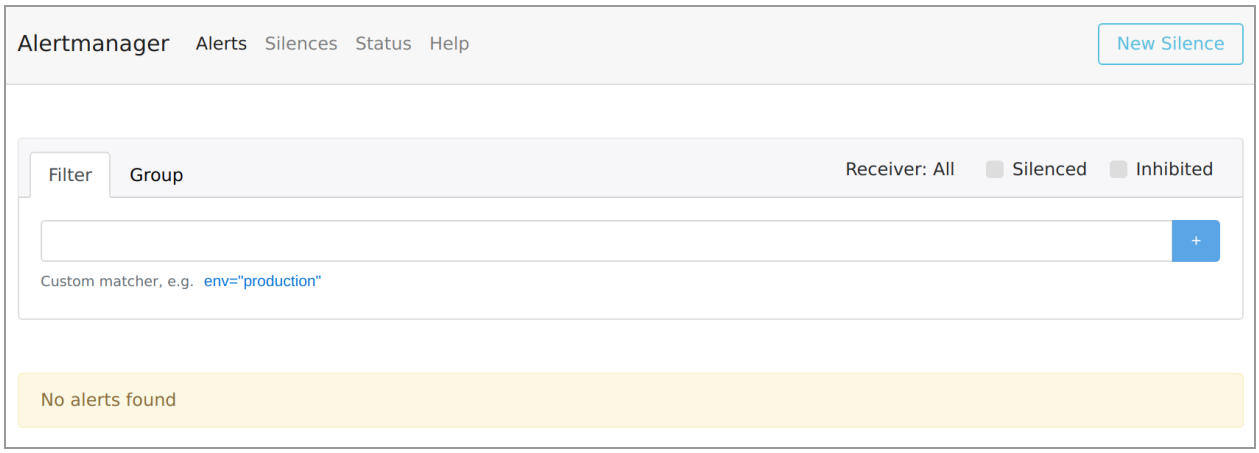
Lab 16.2 - Starting Alertmanager

In the terminal in which you downloaded and configured Alertmanager, start Alertmanager:

```
./alertmanager --cluster.listen-address="" --log.level=debug
```

This disables the cluster port that Alertmanager listens on for peer gossip notifications, as this is only relevant when running Alertmanager in a highly available cluster (you will learn about high availability later). It also enables debug logging, as certain notification failures are only logged at that level.

In a browser, head to **`http://<machine-ip>:9093/`** to view the Alertmanager's web interface. It should look like this:



Alertmanager web interface without any active alerts present

The web interface should not show any active alerts, as you have not configured any Prometheus server to send alerts to it yet.

Lab 16.3 - Configuring Alerting Rules

Let's configure some example alerting rules in Prometheus based on the metrics of the **`demo`** service that your Prometheus server is already monitoring.

Let's say you wanted to alert on HTTP paths that have a **`5xx`** error rate percentage larger than **`0.5%`**. You can try graphing the following expression to see that some paths actually sometimes have a higher error rate than **`0.5%`**:

```
  sum by(path, instance, job) (
    rate(demo_api_request_duration_seconds_count{status=~"5..",job="demo"}[1m])
  )
/
  sum by(path, instance, job) (
    rate(demo_api_request_duration_seconds_count{job="demo"}[1m])
  )
* 100
```

To tell Prometheus to load alerting rules from a file called **`alerting_rules.yml`**, add the following section to the top level of your **`prometheus.yml`**:

```
rule_files:
  - alerting_rules.yml
```

In the same directory, create a file **`alerting_rules.yml`** with the following content:

```
groups:
- name: demo-service-alerts
  rules:
  - alert: Many5xxErrors
    expr: |
```

```
      sum by(path, instance, job) (

rate(demo_api_request_duration_seconds_count{status=~"5..",job="demo"}[1m])
      )
      /
      sum by(path, instance, job) (
        rate(demo_api_request_duration_seconds_count{job="demo"}[1m])
      ) * 100 > 0.5

    for: 30s
    labels:
      severity: critical
    annotations:
      description: "The 5xx error rate for path {{$labels.path}} on
{{$labels.instance}} is {{$value}}%."
```

This example alerting rule instructs Prometheus to send alerts for any **`instance`** and **`path`** label combinations that have an error rate of more than **`0.5%`** for more than 30 seconds. Note that this **`for`** duration is quite short, in order to quickly produce visible results. In real alerting rules, a more common value would be 5 minutes (**`"for: 5m"`**). This rule will also attach an extra **`severity="critical"`** label to any generated alerts. This extra label could be used to route alerts to a pager or other critical notification mechanism via the Alertmanager routing configuration. Finally, a human-readable **`description`** annotation is sent along with each alert, which you could choose to include in notification templates on the Alertmanager side.

Lab 16.4 - Configuring Prometheus to Send Alerts to Alertmanager

To tell Prometheus to send alerts to your Alertmanager, configure its address statically in an **`alerting`** section at the top level of the **`prometheus.yml`** configuration file (note that you could also use service discovery here):

```
alerting:
  alertmanagers:
    - static_configs:
      - targets: ['localhost:9093']
```

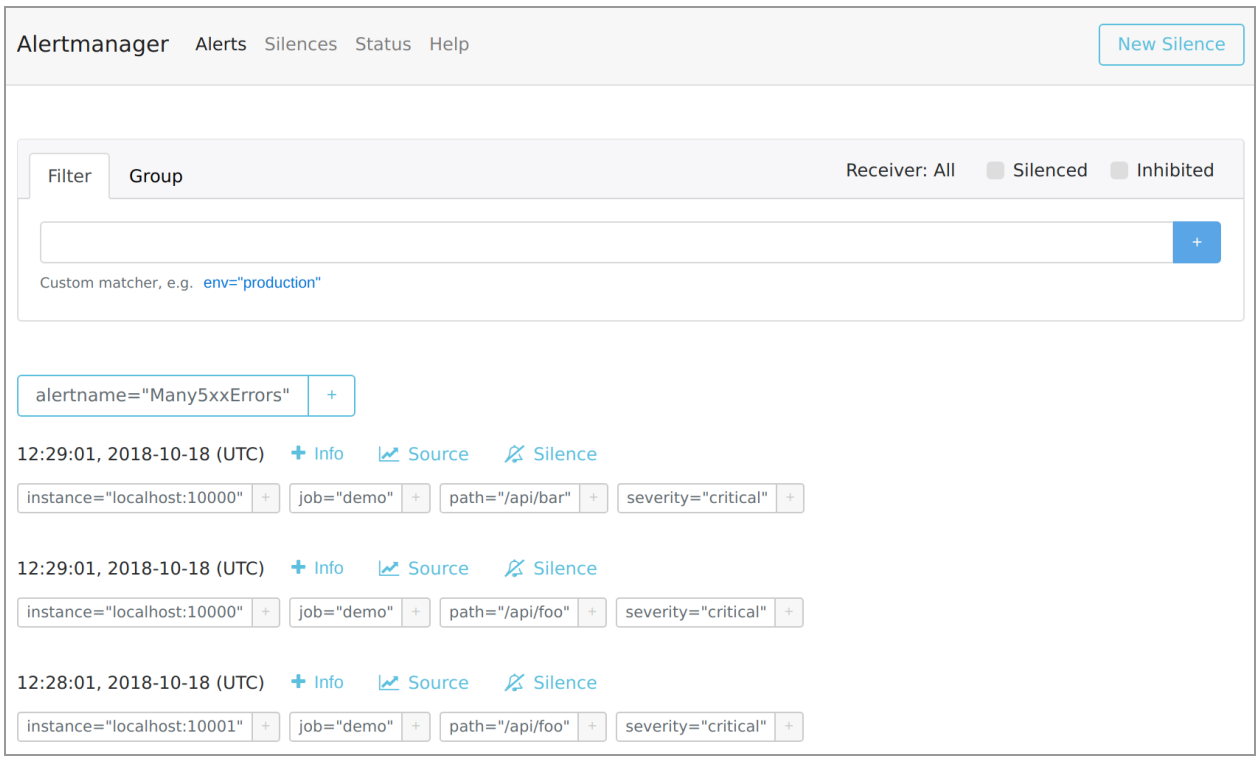
Reload the Prometheus configuration by sending a **`HUP`** signal to the Prometheus process:

```
killall -HUP prometheus
```

Now head to **`http://<machine-ip>:9090/alerts`** to verify that an alerting rule is defined and eventually shows firing alert entries when some of the error rates are higher than 0.5% (click on the **`Many5xxErrors`** alert name to expand the details):

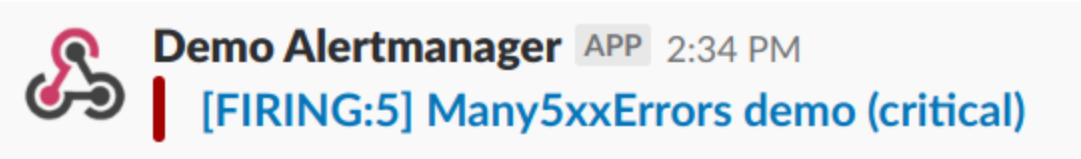
Prometheus `/alerts` page view

Once an alert is firing, you should also be able to see it in the Alertmanager web interface at **`http://<machine-ip>:9093/`**:



Alertmanager web interface showing active alerts

If you are sending alerts to Slack, you should now be seeing notifications arriving in your Slack channel:



Alert notification in Slack

Note that the default notification template is quite simple and only mentions the number of firing alerts, the alert name, and the label values that are common to all alerts of the alert grouping that caused a notification. This is customizable, as you will learn later.

If you are sending alerts to the custom webhook receiver, the output should look somewhat like this:

```
Received 2 firing alerts:
      Grouping labels:
            job = u'demo'
            alertname = u'Many5xxErrors'
      Common labels:
            path = u'/api/bar'
            job = u'demo'
            severity = u'critical'
            alertname = u'Many5xxErrors'
      Common annotations:
            Alert details:
                  Alert 0:
                        Labels: {u'instance': u'localhost:10001', u'job':
u'demo', u'severity': u'critical', u'alertname': u'Many5xxErrors', u'path':
u'/api/bar'}
                        Annotations: {u'description': u'The 5xx error rate for
path /api/bar on localhost:10001 is 0.7981755986316988%.'}
                  Alert 1:
                        Labels: {u'instance': u'localhost:10002', u'job':
u'demo', u'severity': u'critical', u'alertname': u'Many5xxErrors', u'path':
u'/api/bar'}
                        Annotations: {u'description': u'The 5xx error rate for
path /api/bar on localhost:10002 is 0.9090909090909092%.'}
```

Note that you may see a different number of combination of alerts depending on exact timing.

Lab 16.5 - Setting Up Alertmanager in HA Mode

In this lab, you will set up Alertmanager in HA mode.

Stop your existing Alertmanager with **<Ctrl>+C** if it's still running.

In two separate terminals, start two Alertmanager instances **A** and **B** that are configured as a meshed highly available cluster:

```
./alertmanager --cluster.listen-address="0.0.0.0:19093"
--cluster.peer="localhost:29093" --storage.path=data-a
```

...and:

```
./alertmanager --cluster.listen-address="0.0.0.0:29093"
--cluster.peer="localhost:19093" --web.listen-address=:9094
--storage.path=data-b
```

Note how both instances configure the respective other instance as a cluster peer using the **`--cluster.peer`** flag. This causes them to automatically gossip notification and silence state information to the other instance.

Change the alerting section in your **`prometheus.yml`** to the following to send alerts to both of the new Alertmanagers:

```
alerting:
  alertmanagers:
    - static_configs:
      - targets:
        - 'localhost:9093'
        - 'localhost:9094'
```

Reload the Prometheus configuration by sending a **HUP** signal to the Prometheus process:

```
killall -HUP prometheus
```

The alerts should now appear in both Alertmanager instances, but you should only get one notification for each alert group. Also try creating a silence in the web interface of one of the Alertmanager instances. The silence should then automatically also appear in the second instance.

Try taking down one of the Alertmanager nodes (using **<Ctrl>+C**) and verify that alerting still works. If you delete the terminated Alertmanager's data directory (**data-a** or **data-b**) and start it up again, it should re-replicate both the notification states and silences.

Lab 18.1 - Configure an Example Recording Rule

In this lab exercise, you will configure an example recording rule. Let us pretend that you are monitoring many **demo** service instances (not just three) and that it is expensive to calculate the total number of requests across all instances and other label sub-dimensions (**path**, **method**, etc.):

```
sum by(job) (rate(demo_api_request_duration_seconds_count[5m]))
```

You will create a recording rule to pre-record this expression into a new series with the name **job:demo_api_request_duration_seconds_count:rate5m**. The recording rule will be evaluated (and recorded) every 5 seconds, as per our initially configured global **evaluation_interval** setting.

Create a new file in the same directory as your **prometheus.yml** called **recording_rules.yml**:

```
groups:
- name: demo-service
  rules:
  - record: job:demo_api_request_duration_seconds_count:rate5m
    expr: |
      sum by(job) (rate(demo_api_request_duration_seconds_count[5m]))
```

You then have to tell Prometheus to load this rule file by adding an entry to the **rule_files** list in the **prometheus.yml** (which already references the **alerting_rules.yml** file). Update the **rule_files** list to look like this:

```
rule_files:
  - alerting_rules.yml
  - recording_rules.yml
```

Reload the Prometheus configuration by sending a **HUP** signal to the Prometheus process:

```
killall -HUP prometheus
```

In the Prometheus server, run the following query:

```
job:demo_api_request_duration_seconds_count:rate5m
```

This should give you the same result as the original query, but only requires loading pre-computed series. In a large setup, this could now be much faster than running the original query. You could now also pull this aggregated metric into higher-level Prometheus servers using federation, without burdening those servers with instance-level detail. You will learn how to do this in the next chapter.

Note: Recording rules only start writing out results starting from the time they were configured. They do not support back-filling of historical data with the results of an expression.

Lab 19.1 - Scaling via Hierarchical Federation

Let's set up an example of hierarchical federation. Pretend that you have two clusters (although you really only have one machine to run this lab on), with a local Prometheus server in each cluster. Each local Prometheus server monitors three **`demo`** service instances and uses recording rules to create per-job aggregated metrics for their cluster. A third, global, Prometheus server uses federation to pull in those aggregate metrics about both clusters from the per-cluster **`/federate`** Prometheus endpoints. These endpoints return the last value of all series selected via a set of **`match[]`** HTTP parameters and the federating Prometheus then ingests them as normal metrics.

Create two configuration files named **`prometheus-cluster-a.yml`** and **`prometheus-cluster-b.yml`**, both with the following content:

```
global:
  scrape_interval: 5s
  evaluation_interval: 5s
  external_labels:
    cluster: <cluster-name>

rule_files:
  - recording_rules.yml

scrape_configs:
  - job_name: 'demo'
    static_configs:
      - targets:
        - 'localhost:10000'
        - 'localhost:10001'
        - 'localhost:10002'
```

Important: Replace the **`<cluster-name>`** placeholder with "**`a`**" in the first file and "**`b`**" in the second file.

Note that you will simply monitor each of the three **`demo`** service instances twice for this lab instead of truly running separate service instances for each cluster.

In separate terminals, start both Prometheus servers with different data directories and ports:

```
./prometheus \
  --config.file=prometheus-cluster-a.yml \
  --storage.tsdb.path=data-cluster-a \
  --web.listen-address=:19090
```

…and:

```
./prometheus \
  --config.file=prometheus-cluster-b.yml \
  --storage.tsdb.path=data-cluster-b \
  --web.listen-address=:29090
```

Create a file **`prometheus-global.yml`** for the federating Prometheus with the following content:

```
global:
  scrape_interval: 5s
  evaluation_interval: 5s

scrape_configs:
  - job_name: 'federate'
    honor_labels: true
    metrics_path: '/federate'
    params:
      'match[]':
        - '{__name__=~"job:.*"}'
    static_configs:
      - targets:
        - 'localhost:19090'
        - 'localhost:29090'
```

Note the **`honor_labels: true`** scrape option. Because the metrics from lower-level Prometheus servers already include **`job`** and **`instance`** labels, this prevents Prometheus from overwriting any such labels with the target labels from the scrape configuration.

In a new terminal, start the global Prometheus server:

```
./prometheus \
  --config.file=prometheus-global.yml \
  --storage.tsdb.path=data-global \
  --web.listen-address=:39090
```

Head to the global Prometheus server's graphing view at **`http://machine-ip:39090/`** and verify that the metric **`job:demo_api_request_duration_seconds_count:rate5m`** from both per-cluster Prometheus servers is queryable and distinguishes metrics from both clusters via a **`cluster`** label.

Lab 20.1 - Setting Up a Kubernetes Cluster

In this step, you will set up a single-node Kubernetes cluster on your machine. The following instructions have been tested on **Ubuntu 20.04**, but should work similarly on other Ubuntu versions. Installation steps on other Linux distributions will vary. See the Kubernetes installation instructions for details if you are not using Ubuntu 20.04 or want to use a different way of installing Kubernetes.

First, install the Kubelet server, as well as the **`kubeadm`** and **`kubectl`** CLI tools.

As **`root`**, execute the following commands:

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt-get update
apt-get install -y kubelet kubeadm kubectl
```

The Kubelet should now be restarting every few seconds, as it waits in a crashloop for **`kubeadm`** to tell it what to do.

As **`root`**, initialize a Kubernetes cluster using **`kubeadm`**:

```
kubeadm init
```

As **`root`**, copy over the administrator-level **`kubectl`** configuration to your normal user's home directory (**IMPORTANT:** be sure to replace the **`<username>`** placeholder with your normal user's user name!):

```
USERNAME=<username>
mkdir /home/$USERNAME/.kube
```

```
cp /etc/kubernetes/admin.conf /home/$USERNAME/.kube/config
chown $USERNAME:$USERNAME -R /home/$USERNAME/.kube
```

Note that this means that your user will have full administrator-level access to the Kubernetes cluster.

Switch back to your **normal user**.

Since this is a single-node cluster, you have to allow running user pods on the same node that the Kubernetes master is running on:

```
kubectl taint nodes --all node-role.kubernetes.io/master-
```

Install a pod networking plugin on the cluster (Weave Net in this case) so that pods can talk to each other:

```
kubectl apply -n kube-system -f
"https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr
-d '\n')"
```

List the nodes of the Kubernetes cluster:

```
kubectl get nodes
```

After a minute or two, you should see one node in the **`Ready`** state, meaning that your Kubernetes cluster is ready to use.

Lab 20.2 - Setting Up Prometheus on Kubernetes

Prometheus needs to be granted permissions to perform service discovery and to scrape Kubernetes's own service endpoints. To keep things simple for this lab, create a very permissive authorization policy for the Kubernetes cluster:

```
kubectl create clusterrolebinding permissive-binding \
  --clusterrole=cluster-admin \
  --user=admin \
  --user=kubelet \
  --group=system:serviceaccounts
```

On a real Kubernetes cluster, you will want to set more granular permissions. More on that later.

Since the required example Kubernetes resource files to configure and set up Prometheus are relatively long, download a repository containing them to ease the process:

```
git clone https://github.com/juliusv/prometheus-k8s-files
```

Change into the repository:

```
cd prometheus-k8s-files
```

First, store the Prometheus server's configuration in a Kubernetes **`ConfigMap`** with the name **`prometheus-config`** (in setups where the Prometheus configuration contains confidential credentials, you may want to use a Kubernetes **`Secret`** instead of a **`ConfigMap`**):

```
kubectl apply -f prometheus-config.yml
```

Study the configuration file contents to understand how this configuration uses Kubernetes service discovery and relabeling rules to automatically discover and scrape the Kubernetes API Server, the

Kubelets, and all endpoints of services on the cluster that have a **`prometheus.io/scrape: "true"`** annotation. Note that this and similar annotation names do not have a hard-coded meaning in Prometheus, and depending on your situation, you might want to choose a different way of structuring your Prometheus-related Kubernetes annotations and how they are acted upon in relabeling rules.

Now, create a corresponding Prometheus deployment and service that uses the **`ConfigMap`** that you created above:

```
kubectl apply -f prometheus-deployment.yml
```

The resulting Prometheus server should now be reachable at **`http://<machine-ip>:30000/`**.

To get some web service metrics, run three **`demo`** service instances on Kubernetes as well:

```
kubectl apply -f demo-service.yml
```

Verify that both the Prometheus server and three instances of your **`demo`** service have been started successfully:

```
kubectl get pods
```

This should show all pods in the **`Running`** state, at least eventually.

Lab 20.3 - Monitoring Kubernetes Components with Prometheus

Now, let's look at the metrics that Prometheus collects from the Kubernetes components. Head to the Prometheus server's targets page at **`http://<machine-ip>:30000/targets`**. Although the Prometheus configuration didn't contain any static target configurations, it should have discovered targets for:

- Job **`kubelet`**: This monitors the health of the Kubelets themselves (you only have one in your test cluster).
- Job **`kubelet-cadvisor`**: This monitors the built-in cAdvisor endpoint of the Kubelets, which exposes per-container resource usage metrics for the containers running on each machine.
- Job **`kubernetes-apiserver`**: This monitors the health of the Kubernetes API Server instances (of which you also only have one).
- Endpoints for services that have the Kubernetes annotation **`prometheus.io/scrape: "true"`** set:
    - Job **`demo-service`**: This is the **`demo`** service.
    - Job **`kube-dns`**: Kubernetes's DNS server. Although you didn't set a scrape annotation on this one yourself, it comes with the right annotation out of the box when setting up a Kubernetes cluster.
    - Job **`prometheus`**: This is the Prometheus server monitoring itself.

Lab 20.4 - Exploring Container Resource Usage Metrics

The container resource usage metrics that the Kubelet's built-in cAdvisor exposes are the same as the metrics from a stand-alone cAdvisor. The only difference is that Kubernetes attaches additional Kubernetes-specific labels to the containers it starts.

For example, query for:

```
sum by(namespace) (rate(container_cpu_usage_seconds_total[1m]))
```

This shows the aggregate CPU usage for all containers, summed up by Kubernetes namespace.

You can also have a look at the memory usage of the three demo service containers:

```
container_memory_usage_bytes{pod=~"demo-service-.*",container="demo-service"}
```

This works analogously for the other resource usage metrics that the Kubelet exposes.

Lab 20.5 - Exploring Kubernetes API Server Metrics

The Kubernetes API Server exposes metrics about the inner events and states of the API Server itself.

For example, this contains a metric **`apiserver_request_total`** that counts requests to the API Server broken out for each verb (**`GET`**, **`LIST`**, etc.), API resource, resource group, and HTTP response content type and code.

To show how many service-discovery-related requests (most of which will come from the Prometheus server in our test cluster) the Kubernetes API Server is handling, query for:

```
rate(apiserver_request_total{group="discovery.k8s.io",
job="kubernetes-apiserver"}[5m])
```

This should not be a high number, but you could use a similar query to diagnose overload situations, where a client is sending too many requests to the API Server.

You can list all time series that the API Server exposes by querying in the **`Console`** (not **`Graph`**!) view for:

```
{job="kubernetes-apiserver"}
```

Lab 20.6 - Querying Service Metrics

You can query the metrics of the demo service instances that were discovered through the **`kubernetes-service-endpoints`** scrape configuration in the same way that you queried them earlier on in this course, when you were running the demo service outside of Kubernetes. The only difference is that the target labels will be slightly different. As one example, the following query should give you the total HTTP request rate:

```
sum by(job) (rate(demo_api_request_duration_seconds_count[1m]))
```

You can query the metrics of any other job whose service was annotated for scraping as well. Note that this means you don't have to update your Prometheus configuration every time you want to monitor another service's endpoints. Simply add the **`prometheus.io/scrape: "true"`** annotation and the Prometheus server will discover the targets automatically.

Lab 20.7 - Kubernetes Deployment Object Metrics

So far, you have looked at container resource usage metrics, service metrics, and metrics about the Kubernetes API Server itself. But you don't yet have a way to monitor the state of domain-level Kubernetes API objects such as **`Deployment`**, **`DaemonSet`**, **`StatefulSet`** etc. objects. For this, Kubernetes provides an exporter called **`kube-state-metrics`** that queries the API Server for this information and exposes it as Prometheus metrics. In this step, you will set up **`kube-state-metrics`** and explore its metrics.

The **`kube-state-metrics`** repository contains configuration files for deploying the exporter on Kubernetes, without needing to build it ourselves.

In a new terminal, clone the **`kube-state-metrics`** git repository:

```
git clone https://github.com/kubernetes/kube-state-metrics
```

Change into the cloned directory:

```
cd kube-state-metrics
```

Check out version 2.0.0:

```
git checkout v2.0.0
```

Edit the file **`examples/standard/service.yaml`** in this repository to add a **`prometheus.io/scrape: "true"`** annotation to the service definition, so that Prometheus will automatically start monitoring **`kube-state-metrics`**. In the **`metadata`** section of the file, add the following lines:

```
  annotations:
    prometheus.io/scrape: "true"
```

Deploy the exporter on Kubernetes:

```
kubectl apply -f examples/standard
```

You should now see two targets of the exporter on **`http://<machine-ip>:30000/targets`** grouped under the **`kube-state-metrics`** job, since the service exposes two ports (one for the deployment object metrics and one for the health metrics of **`kube-state-metrics`** itself).

You now have metrics about the state of various Kubernetes API objects. For example, to see the desired number of replicas for each **`Deployment`** object on Kubernetes, query for:

```
kube_deployment_spec_replicas
```

To only see how many replicas the **`kube-state-metrics`** exporter itself has, query for:

```
kube_deployment_spec_replicas{deployment="kube-state-metrics"}
```

The time series value should reflect the current number of auto-scaled pods for the **`kube-state-metrics`** deployment (usually one).

You can use this to compare the desired number of replicas with the number of actually available replicas:

```
  kube_deployment_status_replicas_available
!=
  kube_deployment_spec_replicas
```

The result of this expression should be empty unless the desired count of replicas does not match the actual count for a deployment (for example, when a pod cannot be started because a container image cannot be pulled). This kind of comparison filter can be used to create an alert on unhealthy deployments.

For more details on what you can do with these metrics, see the full metrics documentation of **`kube-state-metrics`**.

Lab 22.1 - Setting Up Cortex-based Remote Storage

In this lab, you will set up Prometheus to write all samples it receives into Cortex and also to read them back when issuing PromQL queries. Note that for the purposes of this workshop, we will run Cortex in single-process mode. To set up Cortex as a scalable, production-ready cluster, see the Cortex documentation.

Let's first download Cortex 1.8.0 and mark it executable:

```
wget
https://github.com/cortexproject/cortex/releases/download/v1.8.0/cortex-linux-am
d64
chmod a+x cortex-linux-amd64
```

Download an example configuration file that allows running Cortex in single-process mode with local filesystem storage:

```
wget
https://raw.githubusercontent.com/cortexproject/cortex/4e2373b1eecece53ebe6139cd
75e63d14a2a5a11/docs/configuration/single-process-config.yaml
```

Start Cortex:

```
./cortex-linux-amd64 -config.file=single-process-config.yaml
```

Add the following remote write and remote read sections to the top level of your `prometheus.yml`:

```
remote_write:
  - url: "http://localhost:9009/api/prom/push"
remote_read:
  - url: "http://localhost:9009/api/prom/api/v1/read"
```

Reload the Prometheus configuration by sending a `HUP` signal to the Prometheus process:

```
killall -HUP prometheus
```

You should now be able to query data from Cortex via Prometheus's expression browser. You can also try shutting down Prometheus, deleting its local data directory, starting up Prometheus again, and see if you can still query data from before the deletion.

Lab 22.2 - Setting Up Remote Storage via Thanos

In this lab, you will set up Thanos with your existing Prometheus installation to ship data to MinIO, an S3-compatible object storage system, and get an integrated view of Prometheus's most recent data and the data that Thanos has shipped to MinIO.

In order for Thanos to tag which blocks came from which Prometheus server, you will need to configure external labels for your Prometheus server. Add the following to your main Prometheus server's **`global`** config section in the **`prometheus.yml`**:

```
  external_labels:
    prometheus: main
```

In order for Thanos to safely ship TSDB blocks to long-term storage, you will also need to disable the background compaction in Prometheus's own local TSDB. To achieve this, stop your main Prometheus server and restart it with the following flags:

```
./prometheus --storage.tsdb.min-block-duration=2h
--storage.tsdb.max-block-duration=2h
```

Start a local MinIO instance for storing persisted blocks:

```
docker run -d -p 9000:9000 -p 9001:9001 minio/minio:RELEASE.2021-10-27T16-29-42Z
server /data --console-address ":9001"
```

This will bring up the MinIO web interface on port **`9001`** with the default credentials **`minioadmin`** / **`minioadmin`**. Navigate to **`http://<machine-ip>:9001/buckets`** in your browser to bring up the MinIO storage buckets user interface. Click the **`"Create Bucket +"`** button, then create a bucket named "**`thanos`**" (keep all other bucket options at their default values).

In your Prometheus server's directory, download Thanos 0.23.1:

```
wget
https://github.com/thanos-io/thanos/releases/download/v0.23.1/thanos-0.23.1.linu
x-amd64.tar.gz
```

Extract the archive:

```
tar xvfz thanos-0.23.1.linux-amd64.tar.gz
```

Create a file named **`bucket.yml`** in the main Prometheus server's directory. This will hold the object storage (MinIO) configuration that Thanos should use for uploading data:

```
type: S3
config:
  bucket: "thanos"
  endpoint: "localhost:9000"
  access_key: "minioadmin"
  insecure: true
  secret_key: "minioadmin"
```

Start Thanos in Sidecar mode, pointing it at the object storage configuration file and the data directory of your Prometheus server:

```
./thanos-0.23.1.linux-amd64/thanos sidecar --tsdb.path=./data
--objstore.config-file=bucket.yml
```

If all goes well, the Thanos Sidecar should eventually log lines like this, indicating that it uploaded existing TSDB blocks to MinIO:

```
level=info ts=2021-10-29T10:35:31.692179037Z caller=shipper.go:337 msg="upload
new block" id=01FK5S37D9XD5WKTWG903G70VA
level=info ts=2021-10-29T10:35:31.74048867Z caller=shipper.go:337 msg="upload
new block" id=01FK5S37HZ5KNS1H70RQ5H37A0
```

You should also see block objects appearing in the "**`thanos`**" bucket in the MinIO web interface.

To read data back from object storage, you will need to first bring up the Store component of Thanos (which will be accessed by a later Query component), ensuring that its port usage does not conflict

with the already locally running Sidecar Thanos component. In a new terminal, change back into your Prometheus server's directory and run:

```
./thanos-0.23.1.linux-amd64/thanos store \
  --http-address="0.0.0.0:20902" \
  --grpc-address="0.0.0.0:20901" \
  --data-dir=./thanos-cache \
  --objstore.config-file=bucket.yml
```

Eventually, the Store component should log successful block metadata synchronization from MinIO:

```
level=info ts=2021-10-29T10:38:45.692825164Z caller=fetcher.go:476
component=block.BaseFetcher msg="successfully synchronized block metadata"
duration=2.159171ms duration_ms=2 cached=2 returned=2 partial=0
```

Finally, to query over data from MinIO (via the Store component) and the local Prometheus server (via the Sidecar component) in an integrated way, bring up the Query component of Thanos, pointing it at both the Store and Sidecar components via the repeated usage of the **`--store`** flag.

In a new terminal in your Prometheus server's directory, run:

```
./thanos-0.23.1.linux-amd64/thanos query \
  --http-address="0.0.0.0:30902" \
  --grpc-address="0.0.0.0:30901" \
  --store="127.0.0.1:10901" \
  --store="127.0.0.1:20901"
```

Navigate to **`http://<machine-ip>:30902/`** to see the web interface of the Thanos Query component. It offers a querying web interface similar to Prometheus's built-in expression browser. Try out some queries, and confirm that Thanos returns the same data as your local Prometheus server, but with an additional **`prometheus="main"`** label attached to all returned time series, corresponding to the external labels that you configured on your Prometheus server. If your Thanos setup stored the data from multiple Prometheus servers, you could use this label in a selector to ensure you are selecting data only from the intended server.

In this lab, you explored some of Thanos' functionality for long-term storage and integrated querying. Thanos has several more components that may make sense to run in a production setup: for example, a Compact component that compacts blocks in the object storage system in a similar way as Prometheus compacts blocks locally, or a Rule component which can be used to evaluate Prometheus rules outside of the Prometheus server. Thanos can also deduplicate samples originating from two servers of a high availability (HA) setup. For this and more, see the Thanos documentation.

Lab 24.1 - Logging

Prometheus logs particularly important warnings and errors to stderr. Depending on your environment (Kubernetes, Docker, or some other supervisor environment), those logs may be viewable via different means. Since you are directly running Prometheus on the command line in this course, you can see any log lines in the terminal in which it is running.

For example, Prometheus logs an error when it fails to reload its configuration.

Try modifying the **`prometheus.yml`** of one of your Prometheus servers to the following:

```
invalid_key: invalid_value
```

Make sure that Prometheus reloads its configuration file:

```
killall -HUP prometheus
```

Now confirm that the Prometheus server in question has logged an error on **`stderr`**:

```
level=info ts=2018-01-10T13:03:36.686083237Z caller=main.go:511 msg="Loading
configuration file" filename=bad.yml
level=error ts=2018-01-10T13:03:36.68651404Z caller=main.go:383 msg="Error
reloading config" err="couldn't load configuration (--config.file=bad.yml):
parsing YAML file bad.yml: unknown fields in config: invalid_key"
```

Prometheus is generally relatively quiet and does not log every scrape, query, or other individual actions. However, you can increase the logging verbosity somewhat by setting the **`--log.level=debug`** flag.

It is not recommended to use Prometheus's logging output as a primary means of monitoring Prometheus servers. See the following pages about metrics-based meta-monitoring instead.

Lab 24.2 - Profiling a Prometheus Server

The Prometheus server, Alertmanager, and some other Prometheus components are written in Go and include debugging and profiling HTTP endpoints under the **`/debug/pprof`** path. These endpoints are useful when you want to diagnose certain performance problems or bugs. For example:

- Where Prometheus spends the most CPU time (CPU profile under **`/debug/pprof/profile`** - this one returns binary information and is thus not suitable for manual inspection).
- Where Prometheus allocates the most memory, for example to track down a memory leak (memory profile under **`/debug/pprof/heap`**).
- What all goroutines are currently doing and whether they are blocked on something (goroutine stack dump under **`/debug/pprof/goroutine?debug=2`**).

You can either look at these endpoints manually, or you can use Go's **`go tool pprof`** command to analyze them.

For example, you can collect a CPU profile from a running Prometheus server like this:

```
go tool pprof http://localhost:9090/debug/pprof/profile
```

This will wait for 30 seconds (by default) as the Prometheus server samples over that duration in which functions it spends the most CPU time. It does this by periodically triggering a timer that inspects the stack of all goroutines and observes which function they are executing at that time. It then builds statistics out of these observations.

Once the profiling completes, **`go tool pprof`** opens up a special profiling shell:

```
Fetching profile from http://localhost:9090/debug/pprof/profile
Please wait... (30s)
Saved profile in
/home/workshop/pprof/pprof.prometheus.localhost:9090.samples.cpu.002.pb.gz
Entering interactive mode (type "help" for commands)
```

```
(pprof)
```

On this prompt, enter the following to save the CPU profile as an SVG call graph:

```
svg
```

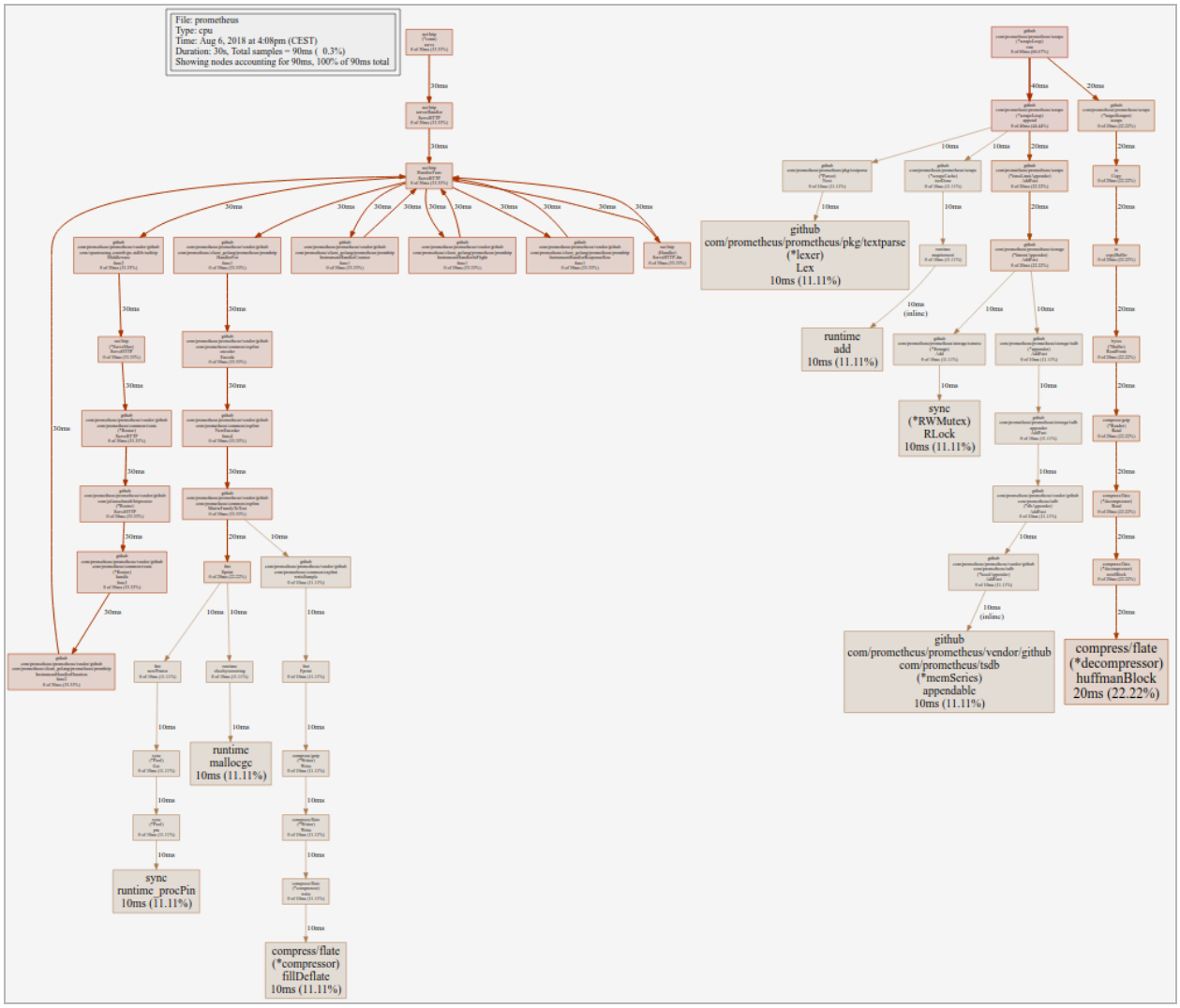
You should see the following:

```
(pprof) svg
Generating report in profile001.svg
```

Or, if you ran **`go tool pprof`** locally against a remote endpoint, you can open the graph directly in a browser or an image viewer by typing:

```
web
```

The resulting graph should look somewhat like this:



CPU profile graph generated using `go tool pprof`

The boxes in the graph form a call graph between functions, with bigger boxes indicating more CPU time spent in that function. You will also see a per-box percentage of time spent.

A memory profile analysis works similarly and can be obtained by running:

```
go tool pprof http://localhost:9090/debug/pprof/heap
```

To learn more about this, see this Go blog post about profiling.