Prometheus Deep Dive - English

Created	@May ,14 2025 4:01 PM
Tags	Course
URL Course	https://learn.acloud.guru/course/0eaae074-9914-47d1-9239-3d6f267d302b/dashboard
URL Documentation	https://prometheus.io/docs

Introduction

Overview

What is Prometheus?

`Prometheus` **- An open-source monitoring and alerting tool. It collects data about applications and systems and allows to visualize the data and issue alerts based on the data.**

`Prometheus` is an open-source systems monitoring and alerting toolkit originally built at `SoundCloud` . Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user `community` . It is now a standalone open source project and maintained independently of any company. To emphasize this, and to clarify the project's governance structure, Prometheus joined the `Cloud Native Computing Foundation` in 2016 as the second hosted project, after `Kubernetes` .

Prometheus collects and stores its metrics as time series data, i.e. metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels.

Features

Prometheus's main features are:

- A multi-dimensional `data model` with time series data identified by **metric name** and **key/value pairs**
- PromQL, a `flexible query language` to leverage this dimensionality
- No reliance on distributed storage; single server nodes are autonomous
- Time series collection happens via a pull model over HTTP
- `Pushing time series` is supported via an intermediary gateway
- Targets are discovered via service discovery or static configuration
- Multiple modes of graphing and dashboarding support

What are components in Prometheus

The two most basic components of a Prometheus systems are:

- `Prometheus server`: A central server that gathers metrics and makes them available.
- `Exporters`: Agents that expose data about systems and applications for collection by the Prometheus server.

`Pull model` - Prometheus server pulls metric data from exporters. Agents do not push data to the Prometheus server.

Prometheus Components:

- `Prometheus Server:` Collects metric data.
- `Exporters:` Provide metric data for Prometheus to consume.
- `Prometheus Pushgateway:` Allows pushing metrics to Prometheus for certain specific use cases.

- `Alertmanager:` Sends alerts triggered by metric data.
- `Visualization tools:` Provide useful ways to view metric data. These are not necessarily part of Prometheus.

Prometheus Background - Use Case - Strengths and Limitations

Background:

- **Language:** Prometheus is written primarily in Go, with some components implemented in Java, Python, and Ruby.
- **License:** Prometheus uses the open-source Apache 2.0 license.
- **History:** Matt T. Proud and Julius Volz started Prometheus development with initial sponsorship from SoundCloud. Today, it is a fully open-source project maintained by many individuals and organizations.
- **Website:** More information and full documentation can be found at `prometheus.io`

Strengths (Good Use Cases):

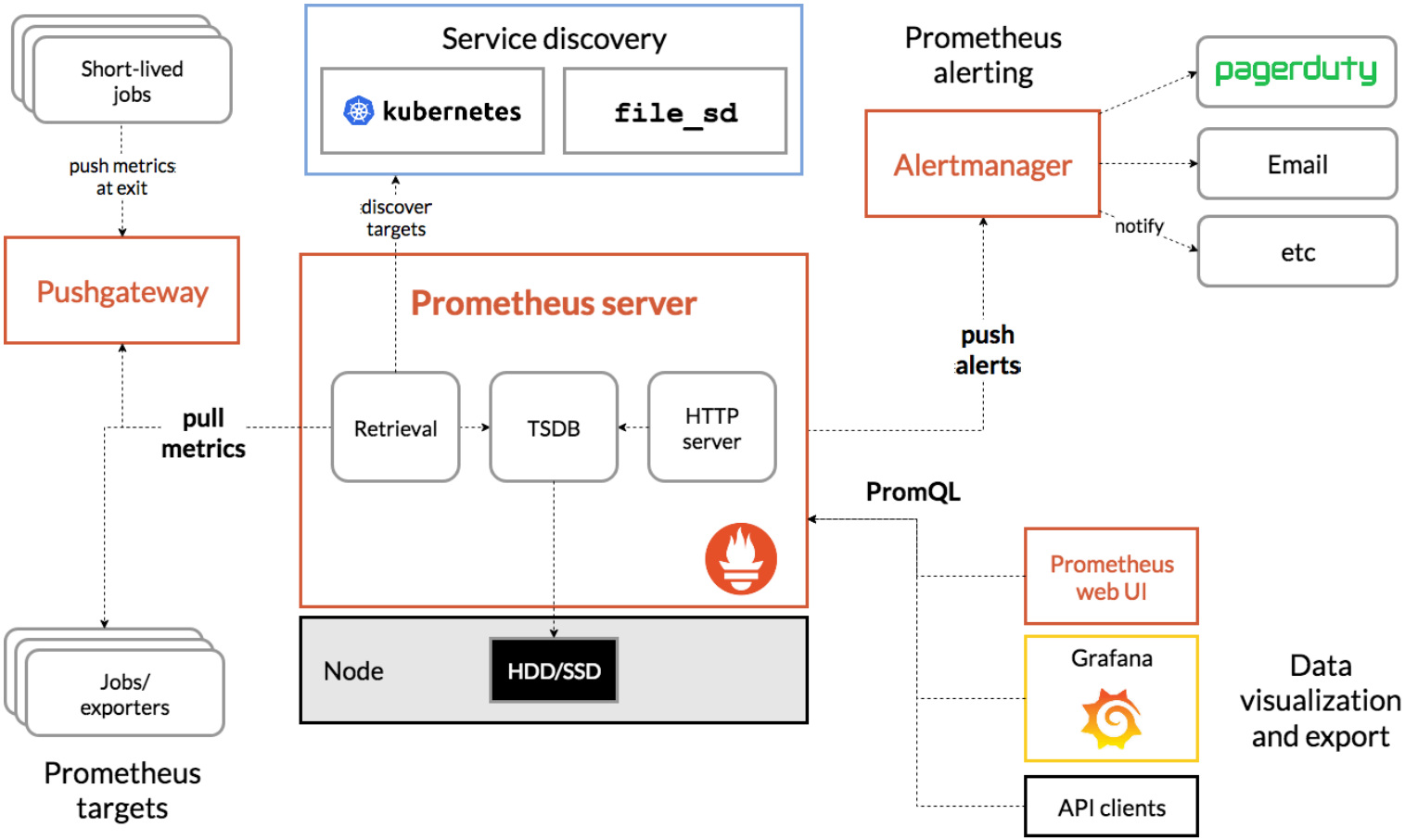
- **Metric collection:** Collect important metrics about your systems and applications in one place.
- **Visualization:** Build dashboards that provide an overview of the health of your systems.
- **Alerting**: Receive an email when something is broken.

Limitations (Not-So-Good Use Cases):

- 100% accuracy (e.g., per-request billing): Prometheus prioritizes system availability over perfect accuracy. It continues operating during failures and outages, which means some data points may be missing or delayed. For use cases requiring absolute precision—like per-request billing—Prometheus isn't the ideal choice.

- Non time-series data (e.g., log aggregation): Prometheus specializes in collecting numerical, time-series metrics. It's not designed for handling other types of data, such as system logs.

Architecture



Concepts

What are metrics?

Metrics are numerical measurements in layperson terms. The term time series refers to the recording of changes over time. What users want to measure differs from application to application. For a web server, it could be request times; for a database, it could be the number of active connections or active queries, and so on.

Metrics play an important role in understanding why your application is working in a certain way. Let's assume you are running a web application and discover that it is slow. To learn what is happening with your application, you will need some information. For example, when the number of requests is high, the application may become slow. If you have the request count metric, you can determine the cause and increase the number of servers to handle the load.

`A metric in Prometheus` is a named set of `time series`, optionally `with labels` that describe additional dimensions.

Introduction to Prometheus Data

Introduction to Prometheus Data Model

What is Time-Series Data?

Prometheus is built around storing `time-series data`.

`Time-series data` consists of a series of values associated with different points in time.

All Prometheus data is fundamentally stored as time-series data. This means Prometheus **not only** tracks the current value of each metric, **but also** changes to each metric over time.

Metrics and Labels

Every time series is uniquely identified by its metric name and optional key-value pairs called labels.

- **Metrics**

`Metric Names` - Every metric in Prometheus has a metric name. The metric name refers to the general feature of a system or application that is being measured.

A example of a metric name: `node_cpu_seconds_total` **. It measures the total amount of CPU seconds.**

- **Labels**

`Metric Labels` - Prometheus uses labels to provide a dimensional data model. This means we can use labels to specify additional things, such as which node's CPU usage is being represented.

A unique combination of a metric name and a set of labels identifies a particular set of time-series data. This example uses a label called CPUT to refer to usage of a specific CPU: `node_cpu_seconds_total{cpu="0"}` **or** `node_cpu_seconds_total{cpu="1", instance="192.168.60.136:9100", job="Linux Server", mode="idle"}`

Metric Types

- **Counter**

`A counter` **is a single number that can only increase or be reset to 0. Counters represent cumulative values.**

When querying `http_requests_total` **:**

Total HTTP requests served: 0 → 15 → 85 → 276 → 0

Examples:

- Number of HTTP requests served by an application
- Number of records processed
- Number of application restarts
- Number of errors

- **Gauge**

`A gauge` **is a single number that can increase and decrease over time.**

Current HTTP requests active: 76 → 82 → 24 → 56

Examples:

- Number of concurrent HTTP requests
- CPU usage
- Memory usage
- Current active threads

- **Histogram**

`A histogram` counts the number of observations/events that fall into a set of configurable buckets, each with its own separate time series. A histogram will use labels to differentiate between buckets.

- **Summary**

`A Summary` is a Prometheus metric type that samples observations (e.g., request durations, response sizes) and provides a total count, a sum of all observed values, and configurable quantiles (like the 95th percentile) calculated over a sliding time window.

Introduction to Prometheus Querying

What is Querying?

`Querying` allows you to access and work with your metric data in Prometheus.

Use `PromQL` (Prometheus Query Language) to write queries and retrieve useful information from the metric data collected by Prometheus.

Use Prometheus queries in a variety of ways to obtain and work with data:

- Expression Browser
- Prometheus HTTP API
- Visualization tools such as Grafana

In the Prometheus UI, the "**Table**" tab is for instant queries and the "**Graph**" tab is for range queries.

Other programs can fetch the result of a PromQL expression via the HTTP API.

Query Basics

- `Selectors`

The most basic component of the `PromQL` query is a time-series selector. A time-series selector is simply a metric name, optionally combined with labels.

Both of the following are valid time-series selectors:(Example) `node_cpu_seconds_total` or `node_cpu_seconds_total{cpu="0"}`

- `Label Matching`

Use a variety of operators to perform advanced matches based upon label values:

- `=`**: Equals**
- `!=`**: Does not equal**
- `=~`**: Regex match**
- `!~`**: Does not regex match**

Example:

```
node_cpu_seconds_total{cpu="0"}
node_cpu_seconds_total{cpu!="0"}
node_cpu_seconds_total{cpu=~"1*"}
node_cpu_seconds_total{cpu!~"1*"}
```

- `Range Vector Selectors`

Range vector literals work like instant vector literals, except that they select a range of samples back from the current instant. Syntactically, a float literal is appended in square brackets (`[]`) at the end of a vector selector to specify for how many seconds back in time values should be fetched for each resulting range vector element. Commonly, the float literal uses the syntax with one or more time units, e.g. `[5m]`.

```
http_requests_total{job="prometheus"}[5m]
```

- `Offset Modifier`

The `offset` modifier allows changing the time offset for individual instant and range vectors in a query.

For example, the following expression returns the value of `http_requests_total` 5 minutes in the past relative to the current query evaluation time:

```
http_requests_total{job="prometheus"}[5m]
```

Operators

Arithmetic Binary Operators

The following operators perform basic arithmetic on numeric values:
+: Addition

-: Subtraction

*: Multiplication

/: Division

%: Modulo

^: Exponentiation

```
node_cpu_seconds_total * 2
```

Matching Rules

Use the following keywords to control matching behavior:

- ignoring(<label list>) - Ignore the specified labels when matching.
- on(<label list>) - Use only the specified labels when matching.

```
node_cpu_seconds_total{mode="system"} + ignoring(mode) node_cpu_seconds_total{mode="user"}
node_cpu_seconds_total{mode="system"} + on(cpu) node_cpu_seconds_total{mode="user"}
```

Comparison Binary Operators

By default, comparison operators filter results to only those where the comparison evaluates as true:
==: Equal
!=: Not equal

<: Less than

=: Greater than or equal
<=: Less than or equal

```
node_cpu_seconds_total == 0
```

Use the bool modifier to instead return the `boolean` result of the comparison:

```
node_cpu_seconds_total == bool 0
```

Logical/Set Binary Operators

These operators combine sets of results in various ways:

- `and`: Intersection
- `or`: Union
- `unless`: Complement

These operators use labels to compare records. For example, and returns only records where the set of labels in one set of results is matched by labels in the other set:

```
node_cpu_seconds_total and node_cpu_guest_seconds_total
```

Aggregation Operators

Aggregation operators combine multiple values into a single value:

`sum`: Add all values together.

`min`: Select the smallest value.

`max`: Select the largest value.

`avg`: Calculate the average of all values.

`stddev`: Calculate population standard deviation over all values.

`stdvar`: Calculate population standard variance over all values.

`count`: Count number of values.

`count_values`: Count the number of values with the same value.

`bottomk`: Smallest number (k) of elements.

`topk`: Largest number (k) of elements.

`quantile`: Calculate the quantile for a particular dimension.

This query uses avg to get the average idle time between all CPUs:

```
avg(node_cpu_seconds_total{mode="idle"})
```

Functions

Functions provide a wide array of built-in functionality to aid in the process of writing queries.

Consult the Prometheus documentation for a full list of available functions.

For example:

- abs(): Calculates absolute value
- clamp_max(): Returns values, but replaces them with a maximum value if they exceed that value

- clamp_min(): Returns values, but replaces them with a minimum value if they are less than that value
- rate() is a particularly useful function for tracking the average per-second rate of increase in a time-series value.

For example, this function is useful for alerting when a particular metric "`spikes`," or increases abnormally quickly.

Prometheus HTTP API

Prometheus provides an HTTP API you can use to execute queries and obtain results using HTTP requests.

This API is a useful way to interact with Prometheus, especially if you are building your own custom tools that require access to Prometheus data.

You can run queries via the API at `/api/v1/query` on the Prometheus server.

Use the query parameter to specify your query:

```
/api/v1/query?query=node_cpu_seconds_total{cpu="0"}
```

Introduction to Visualization

What is Visualization?

`Visualization` - The creation of visual representations of your metric data, such as charts, graphs, dashboards, etc. There are multiple tools that can help you visualize your metric data.

- Expression Browser

- Console Templates

Other external tools can be integrated with Prometheus to visualize Prometheus data:

- Grafana

Native Visualization Methods

Expression Browser

> Expression browser is a built-in web UI for executing queries and viewing simple graphs. Expression browser is mainly used for ad-hoc queries and basic debugging. More advanced visualization tools will likely be needed for day-to-day monitoring. You can access the expression browser at the `/graph` endpoint on your Prometheus server

Console Templates

Console templates allow you to create visualization consoles using Go templating language. Prometheus server serves consoles based upon these templates. Template files are stored at the location defined by the `--web.console.templates argument` (`/etc/prometheus/consoles)`. You can view templates by accessing the /consoles/

Console Templates Graph Library

Graph Library allows you to render graphs within your console templates:

Grafana

What is Grafana?

Grafana is an open-source analytics and monitoring tool. Grafana can connect to Prometheus, allowing you to build visualizations and dashboards to display your Prometheus metric data.

With Grafana, you can:

- Access Prometheus data using queries.
- Display query results using a variety of different panels (graphs, gauges, tables, etc.)
- Collect multiple panels into a single dashboard

Introduction to Collecting Metrics

Introduction to Exporters

What is Exporters?

Exporters provide the metric data that is collected by the Prometheus server. An exporter is any application that exposes data in a format Prometheus can read.

The `scrape_config` in `prometheus.yml` configures the Prometheus server to regularly connect to each exporter and

Application Monitoring

We have already set up monitoring for a Linux server using Node Exporter. However, you can also use Prometheus to monitor specific applications using other exporters. This is known as application monitoring.

Jobs and Instances

Instances - Prometheus automatically adds an instance label to metrics.

Instances are individual endpoints Prometheus scrapes. Usually, an instance is a single application or process being monitored.

Jobs - Prometheus also automatically adds labels for jobs.

A job is a collection of instances, all sharing the same purpose. For example, a job might refer to a collection of multiple replicas for a single application:

```
- job: api-server
  - instance 1: 1.2.3.4:5670
  - instance 2: 1.2.3.4:5671
  - instance 3: 5.6.7.8:5670
```

Introduction to `Pushgateway`

Prometheus server uses a pull method to collect metrics, meaning Prometheus reaches out to exporters to pull data.
Exporters do not reach out to Prometheus.

However, there are some use cases where a push method is necessary, such as monitoring of batch job processes.

Prometheus `Pushgateway` serves as a middle-man for these use cases:

- Clients push metric data to `Pushgateway`.
- Prometheus server pulls metrics from `Pushgateway`, just like any other exporter.

The Prometheus documentation recommends using `Pushgateway` only for very specific use cases.

Introduction to Recording Rules

Recording rules allow you to pre-compute the values of expressions and queries and save the results as their own separate set of time-series data.

Recording rules are evaluated on a schedule, executing an expression and saving the result as a new metric.

Introduction to Alerting

What is `Alertmanager`?

Alertmanager is an application that runs in a separate process from the Prometheus server. It is responsible for handling alerts sent to it by clients such as Prometheus server.

Alerts are notifications that are triggered automatically by metric data.

For example: A server goes down, and the on-call administrator automatically gets an email notifying them of the problem so they can take action.

`Alertmanager` does the following:

- Deduplicating alerts when multiple clients send the same alert
- Grouping multiple alerts together when they happen around the same time
- Routing alerts to the proper destination such as email, or another alerting application such as PagerDuty or `OpsGenie`

Prometheus Alerts - Alerting Rules

Alerting rules provide a way to define the conditions and content of Prometheus alerts.

They allow you to use Prometheus expressions to define conditions that will trigger an alert based upon your metric data.

Alerting rules are created and configured in the same way as recording rules.

Specify one or more locations for rule files in `prometheus.yml` under `rule_files`.

Advanced Concepts