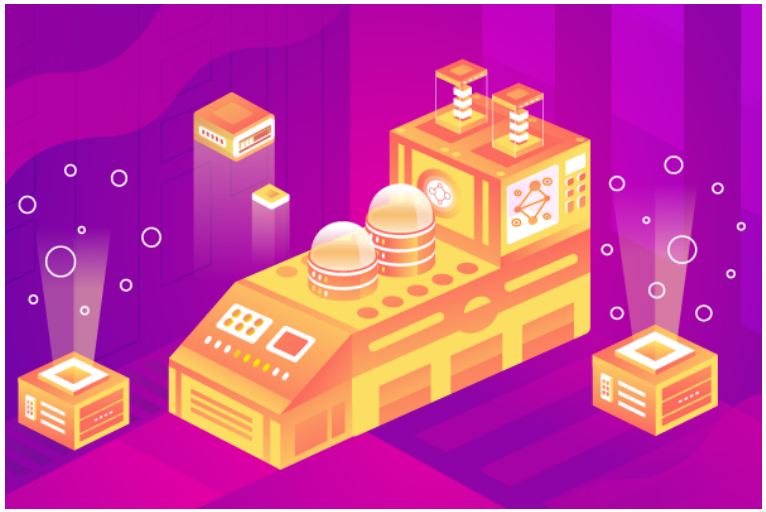
Study Guide



Prometheus Deep Dive Study Guide

Introduction to Prometheus

Will Boyd
williamb@linuxacademy.com
July 7, 2020

A CLOUD GURU

Contents

Prometheus Basics

What Is Prometheus?

Documentation:

- **Prometheus Overview**

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

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.

Background:

- Language: Prometheus is primarily written in *Go*. Some components are written in *Java*, *Python*, and *Ruby*.
- License: Prometheus uses the open-source *Apache 2.0* license.
- History: Prometheus development was started by *Matt T. Proud* and *Julius Volz*. It was initially sponsored by *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**.

Prometheus Architecture - Bird's-Eye View

Documentation:

- **Prometheus Overview**

The two most basic components of a Prometheus system 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
- Client libraries: Easily turn your custom application into an exporter that exposes metrics in a format Prometheus can 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 Use Cases — Strengths and Limitations

Documentation:

- **Prometheus Overview**

Strengths (Good Use Cases):

- Metric Collection
- Visualization
- Alerting

Limitations (Not-So-Good Use Cases):

- 100% accuracy (e.g., per-request billing): Prometheus is designed to operate even under failure conditions. This means it will continue to provide data even if new data is not available due to failures and outages. If you need 100% up-to-the-minute accuracy, such as in the case of per-request billing, Prometheus may not be the best tool to use.
- Non time-series data (e.g., log aggregation): Prometheus is built to monitor time-series metrics, especially data that is numeric. It is not the best choice for collecting more generic types of data, such as system logs.

Installation and Configuration

Building a Prometheus Server

Documentation:

- **Prometheus Installation**

Create a Cloud Playground server:

- Distribution: `Ubuntu 18.04 Bionic Beaver LTS`
- Size: `Small`
- Tag: `Prometheus`

Create a user, group, and directories for Prometheus:

```
sudo useradd -M -r -s /bin/false prometheus

sudo mkdir /etc/prometheus /var/lib/prometheus
```

Download and extract the pre-compiled binaries:

```
wget https://github.com/prometheus/prometheus/releases/download/v2.16.0/prometheus-2.16.0.linux-amd64.tar.gz

tar xzf prometheus-2.16.0.linux-amd64.tar.gz prometheus-2.16.0.linux-amd64/
```

Move the files from the downloaded archive to the appropriate locations and set ownership:

```
sudo cp prometheus-2.16.0.linux-amd64/{prometheus,promtool} /usr/local/bin/

sudo chown prometheus:prometheus /usr/local/bin/{prometheus,promtool}

sudo cp -r prometheus-2.16.0.linux-amd64/{consoles,console_libraries} /etc/prometheus/

sudo cp prometheus-2.16.0.linux-amd64/prometheus.yml /etc/prometheus/prometheus.yml

sudo chown -R prometheus:prometheus /etc/prometheus

sudo chown prometheus:prometheus /var/lib/prometheus
```

Briefly test your setup by running Prometheus in the foreground:

```
prometheus --config.file=/etc/prometheus/prometheus.yml
```

Create a systemd unit file for Prometheus:

```
sudo vi /etc/systemd/system/prometheus.service
```

Define the Prometheus service in the unit file:

```
[Unit]
Description=Prometheus Time Series Collection and Processing Server
Wants=network-online.target
After=network-online.target

[Service]
User=prometheus
Group=prometheus
Type=simple
ExecStart=/usr/local/bin/prometheus \
```

```
    --config.file /etc/prometheus/prometheus.yml \
    --storage.tsdb.path /var/lib/prometheus/ \
    --web.console.templates=/etc/prometheus/consoles \
    --web.console.libraries=/etc/prometheus/console_libraries

[Install]
WantedBy=multi-user.target
```

Start and enable the Prometheus service:

```
sudo systemctl daemon-reload
sudo systemctl start prometheus
sudo systemctl enable prometheus
```

Make an HTTP request to Prometheus to verify that it is able to respond:

```
curl localhost:9090
```

You can also access Prometheus in a browser using the server's public IP address: `http://<Prometheus server public IP>:9090`.

Configuring Prometheus

Documentation:

- **Prometheus Configuration**
- **Example Prometheus Config File**

Prometheus can be configured using a configuration file. Run Prometheus with the `--config.file` flag to specify the location of this file. We set up our systemd service to use the configuration file at `/etc/prometheus/prometheus.yml`.

The configuration file used the *YAML* format.

Refer to the Prometheus documentation for detailed information on all the available configuration options.

Restarting Prometheus will load the new configuration.

Prometheus can also reload its configuration at runtime without the need for a restart. One way to do this is to send a *SIGHUP* signal to the Prometheus process (e.g., `sudo killall -HUP prometheus`).

Configuring an Exporter

Documentation:

- **Scrape Config**
- **Monitoring a Linux Host**

Exporter - A Prometheus exporter is any application that exposes metric data in a format that can be collected (or "scraped") by the Prometheus server.

For example, *Node Exporter* runs on a Linux machine and collects a variety of system metrics. It then exposes them to Prometheus.

The *scrape_config* section of the Prometheus config file provides a list of targets the Prometheus server will scrape, such as a Node Exporter running on a Linux machine.

Prometheus server will scrape these targets periodically to collect metric data.

Prometheus Data

Prometheus Data Model

What Is Time-Series Data?

Documentation:

- **Wikipedia - Time Series**
- **Prometheus Data Model**

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

Documentation:

- **Prometheus Metric Names and Labels**

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.

An example of a metric name:

```
node_cpu_seconds_total
```

`node_cpu_seconds_total` measures the total amount of CPU time being used in CPU seconds.

Note that the metric name merely refers to the feature being measured. Metric names do not point to a specific data value but potentially a collection of many values.

Simply querying `node_cpu_seconds_total` would likely return a list of multiple data points, such as CPU usage for multiple CPUs on a server, or even multiple servers.

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 `cpu` to refer to usage of a specific CPU:

```
node_cpu_seconds_total{cpu="0"}
```

Most Prometheus metrics have multiple labels:

```
node_cpu_seconds_total{cpu="0",instance="10.0.1.102:9100",mode="idle"} 287.75
node_cpu_seconds_total{cpu="0",instance="10.0.1.102:9100",mode="user"}  51.45
```

Using metric names and labels, you can write queries that do things like average CPU usage across a whole data center as well as drill down into the CPU usage of a single CPU on a single node.

Metric Types

Documentation:

- **Metric Types**
- **Histograms and Summaries**

Metric types refer to different ways in which exporters represent the metric data they provide.

Metric types are not represented in any special way in a Prometheus server, but it is important to understand them in order to properly interpret your metrics.

Counter

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

Total HTTP requests served:

- 0
- 12
- 85
- 276

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. The below example provides the number of HTTP requests whose duration falls into each bucket:

```
http_request_duration_seconds_bucket{le="0.3"}
http_request_duration_seconds_bucket{le="0.6"}
http_request_duration_seconds_bucket{le="1.0"}
```

Histograms also include separate metric names to expose the `_sum` of all observed values and the total `_count` of events:

```
http_request_duration_seconds_sum
http_request_duration_seconds_count
```

Summary

A *summary* is similar to a histogram, but it exposes metrics in the form of *quantiles* instead of buckets. While buckets divide values based on specific boundaries, quantiles divide values based on the percentiles into which they fall.

This value represents the number of HTTP requests whose duration falls within the 95th percentile of all requests or the top 5% longest requests:

```
http_request_duration_seconds{quantile="0.95"})
```

Like histograms, summaries also expose the `_sum` and `_count` metrics.

Querying

Introduction to Prometheus Querying

Documentation:

- **Querying Prometheus**

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

You can use *PromQL (Prometheus Query Language)* to write queries and retrieve useful information from the metric data collected by Prometheus.

You can use Prometheus queries in a variety of ways to obtain and work with data:

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

Query Basics

Documentation:

- **Querying Prometheus**

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:

```
node_cpu_seconds_total

node_cpu_seconds_total{cpu="0"}
```

When multiple values exist for a label in the Prometheus data and you do not specify the label in your query, Prometheus will simply return data for all the values of that label.

Label Matching

You can use a variety of operators to perform advanced matches based upon label values:

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

```
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

Since Prometheus data is fundamentally time-series data, you can select data points over a particular time range.

This query selects all values for the metric name and label recorded over the last two minutes:

```
node_cpu_seconds_total{cpu="0"}[2m]
```

Offset Modifier

You can use an *offset modifier* to provide a time offset to select data from the past, with or without a range selector.

Select the CPU usage from one hour ago:

```
node_cpu_seconds_total offset 1h
```

Select CPU usage values over a five-minute period one hour ago:

```
node_cpu_seconds_total[5m] offset 1h
```

Query Operators

Documentation:

- **Operators**

Operators allow you to perform calculations based upon your metric data.

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

When using operators, Prometheus uses matching rules to determine how to combine or compare records from two sets of data. By default, records only match if all of their labels match.

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
- `>`: Greater than
- `<`: 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"})
```

Query Functions

Documentation:

- **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

Documentation:

- **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"}
```

Visualization

Introduction to Visualization

Prometheus Visualization Methods

Documentation:

- **Expression Browser**
- **Grafana**
- **Console Templates**

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.

Some tools are built into Prometheus:

- Expression Browser
- Console Templates

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

- Grafana
- Others

Native Visualization Methods

Expression Browser

Documentation:

- **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

Documentation:

- **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/<template file name>` endpoint on your Prometheus server.

You can find some example template files located in `/etc/prometheus/consoles`.

View the example console at `/consoles/index.html.example`.

The Prometheus documentation also contains examples.

Console templates contain HTML with *Go template expressions* that will be dynamically replaced with Prometheus data when the console is rendered.

Console Template Graph Library

Documentation:

- **Graph Library**

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

```
<div id="queryGraph"></div>
<script>
new PromConsole.Graph({
  ...
})
</script>
```

Grafana

What Is Grafana?

Documentation:

- **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.

Installing and Configuring Grafana

Documentation:

- **Install on Debian or Ubuntu**

Create a Grafana server.

Recommended cloud playground settings:

- Distribution: *Ubuntu 18.04 Bionic Beaver LTS*
- Size: *Small*
- Tag: *Grafana*

Log in to your new server.

Install some required packages:

```
sudo apt-get install -y apt-transport-https software-properties-common wget
```

Add the GPG key for the Grafana OSS repository, then add the repository:

```
wget -q -O - https://packages.grafana.com/gpg.key | sudo apt-key add -

sudo add-apt-repository "deb https://packages.grafana.com/oss/deb stable main"
```

Install the Grafana package:

```
sudo apt-get update

sudo apt-get install grafana=6.6.2
```

Enable and start the `grafana-server` service:

```
sudo systemctl enable grafana-server

sudo systemctl start grafana-server
```

Make sure the service is in the `Active (running)` state:

```
sudo systemctl status grafana-server
```

You can also verify that Grafana is working by accessing it in a web browser at `http://<Grafana server Public IP>:3000`.

Log in to Grafana with the username `admin` and password `admin`.

Reset the password when prompted.

Click **Add data source**.

Select **Prometheus**.

For the `URL`, enter `http://<Prometheus server Private IP>:9090`. Be sure to supply the unique private IP address of your Prometheus server.

Click **Save & Test**. You should see a banner that says *Data source is working*.

Test your setup by running a query to get some Prometheus data. Click the **Explore** icon on the left.

In the PromQL Query input, enter a simple query, such as `up`.

Execute the query. You should see some data appear.

Building Prometheus Dashboards in Grafana

Documentation:

- **Using Grafana with Prometheus**
- **Grafana Pre-Built Dashboards**

Once Grafana is up and running, the process of using it to visualize Prometheus data is relatively simple:

- Configure a Prometheus data source in Grafana.
- Create a Prometheus graph in Grafana.
- Build Grafana dashboards to display your Prometheus graphs.

Grafana allows you to import pre-built dashboards, many of which are built with Prometheus data in mind.

You can find these at https://grafana.com/grafana/dashboards.

Collecting Metrics

Exporters

Introduction to Exporters

Documentation:

- **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 collect metric data.

Application Monitoring

Documentation:

- **Exporters**
- **Apache Exporter for Prometheus**

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*.

There are a variety of ways to monitor applications:

- Use an existing exporter.
- Use Prometheus Pushgateway for batch processing and short-lived jobs.
- Use client libraries to build an exporter into your custom applications.
- Code your own client libraries or exporters.

For example, you can monitor Apache with *Apache Exporter for Prometheus*.

Jobs and Instances

Documentation:

- **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
```

Prometheus automatically creates metric data about scrapes for each instance.

For example:

- `up{job="<job-name>", instance="<instance-id>"}` — Indicates whether or not the scrape was successful.
- `scrape_duration_seconds{job="<job-name>", instance="<instance-id>"}` — The number of seconds the scrape took to complete.

Prometheus Pushgateway

Introduction to Pushgateway

Documentation:

- **Pushing Metrics**
- **Prometheus Pushgateway**
- **When to Use 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.

These usually involve service-level batch jobs. A batch job's process exits when processing is finished. It is unable to serve metrics once the job is complete. It should not need to wait for a scrape from Prometheus server in order to provide metrics; therefore, such jobs need a way to push metrics at the appropriate time.

Installing Pushgateway

Documentation:

- **Pushing Metrics**
- **Prometheus Pushgateway**

Log in to the Prometheus server.

Create a user and group for Pushgateway:

```
sudo useradd -M -r -s /bin/false pushgateway
```

Download and install the Pushgateway binary:

```
wget https://github.com/prometheus/pushgateway/releases/download/v1.2.0/pushgateway-1.2.0.linux-amd64.tar.gz

tar xvfz pushgateway-1.2.0.linux-amd64.tar.gz

sudo cp pushgateway-1.2.0.linux-amd64/pushgateway /usr/local/bin/

sudo chown pushgateway:pushgateway /usr/local/bin/pushgateway
```

Create a systemd unit file for Pushgateway:

```
sudo vi /etc/systemd/system/pushgateway.service
```

```
[Unit]
Description=Prometheus Pushgateway
Wants=network-online.target
After=network-online.target

[Service]
User=pushgateway
Group=pushgateway
Type=simple
ExecStart=/usr/local/bin/pushgateway

[Install]
WantedBy=multi-user.target
```

Start and enable the `pushgateway` service:

```
sudo systemctl enable pushgateway

sudo systemctl start pushgateway
```

Verify that the service is running and that it is serving metrics:

```
sudo systemctl status pushgateway

curl localhost:9091/metrics
```

Configure Prometheus to scrape metrics from Pushgateway

Edit the Prometheus config:

```
sudo vi /etc/prometheus/prometheus.yml
```

Under the `scrape_configs` section, add a scrape configuration for Pushgateway. Be sure to set `honor_labels: true`:

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

Restart Prometheus to load the new configuration:

```
sudo systemctl restart prometheus
```

Use Expression Browser to verify that you can see Pushgateway metrics in Prometheus. You can access Expression Browser in a web browser at `http://<Prometheus Server Public IP>:9090`.

Run a query to view some Pushgateway metric data:

```
pushgateway_build_info
```

Pushing Data to Pushgateway

Documentation:

- **Pushing Metrics**
- **Prometheus Pushgateway**

To push metrics to Pushgateway, simply send the metric data over HTTP with the *Pushgateway API*:

```
/metrics/job/some_job/instance/some_instance
```

The request body should contain data formatted like any other Prometheus exporter metric:

```
# TYPE some_metric counter
# HELP Example metric
some_metric{label="val1"} 42
# TYPE another_metric gauge
another_metric 12
```

The Prometheus client libraries also include functionality for pushing data via Pushgateway.

Recording Rules

Introduction to Recording Rules

Documentation:

- **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.

Recording rules are especially useful when you have complex or expensive queries that are run frequently. For example, by saving pre-computed results using a recording rule, the expression does not need to be re-evaluated every time someone opens a dashboard.

Recording rules are configured using YAML. Create them by placing YAML files in the location specified by `rule_files` in `prometheus.yml`.

When creating or changing recording rules, reload their configuration the same way you would when changing `prometheus.yml`:

```
groups:
name: my_rule_group
  rules:
  - record: my_custom_metric
    expr: up{job="My Job"}
```

Configuring Recording Rules

Documentation:

- **Recording Rules**

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

Create rules by defining them within YAML files in the appropriate location.

You can define a recording rule like so:

```
# The name of the group. Must be unique within a file.
name: <string>
# How often rules in the group are evaluated.
[ interval: <duration> | default = global.evaluation_interval ]
rules:
# The name of the metric where results will be stored.
- recording: <string>
  # The expression or query used to calculate the value.
  expr: <string>
```

Alerting

Alertmanager Setup and Configuration

What Is Alertmanager?

Documentation:

- **Alertmanager**
- **Alerting Overview**

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

Alertmanager does not create alerts or determine when alerts needs to be sent based on metric data. Prometheus handles that step and forwards the resulting alerts to Alertmanager:

Installing Alertmanager

Documentation:

- **Alertmanager**
- **Alertmanager Github**

Log in to your Prometheus server.

Create a user and group for Alertmanager:

```
sudo useradd -M -r -s /bin/false alertmanager
```

Download and install the Alertmanager binaries, move the files into the appropriate locations, and set ownership:

```
wget https://github.com/prometheus/alertmanager/releases/download/v0.20.0/
alertmanager-0.20.0.linux-amd64.tar.gz

tar xvfz alertmanager-0.20.0.linux-amd64.tar.gz

sudo cp alertmanager-0.20.0.linux-amd64/{alertmanager,amtool} /usr/local/bin/

sudo chown alertmanager:alertmanager /usr/local/bin/{alertmanager,amtool}

sudo mkdir -p /etc/alertmanager

sudo cp alertmanager-0.20.0.linux-amd64/alertmanager.yml /etc/alertmanager

sudo chown -R alertmanager:alertmanager /etc/alertmanager
```

Create a data directory for Alertmanager:

```
sudo mkdir -p /var/lib/alertmanager

sudo chown alertmanager:alertmanager /var/lib/alertmanager
```

Create a configuration file for amtool:

```
sudo mkdir -p /etc/amtool

sudo vi /etc/amtool/config.yml
```

Enter the following content in the amtool config file:

```
alertmanager.url: http://localhost:9093
```

Create a systemd unit file for Alertmanager:

```
sudo vi /etc/systemd/system/alertmanager.service
```

```
[Unit]
Description=Prometheus Alertmanager
Wants=network-online.target
After=network-online.target

[Service]
User=alertmanager
Group=alertmanager
Type=simple
ExecStart=/usr/local/bin/alertmanager \
  --config.file /etc/alertmanager/alertmanager.yml \
  --storage.path /var/lib/alertmanager/
```

```
[Install]
WantedBy=multi-user.target
```

Start and enable the `alertmanager` service:

```
sudo systemctl enable alertmanager

sudo systemctl start alertmanager
```

Verify that the service is running and that you can reach it:

```
sudo systemctl status alertmanager

curl localhost:9093
```

You can also access Alertmanager in a web browser at `http://<Prometheus server public IP>:9093`.

Verify that amtool is able to connect to Alertmanager and retrieve the current configuration.

```
amtool config show
```

Configure Prometheus to Connect to Alertmanager

Edit the Prometheus config:

```
sudo vi /etc/prometheus/prometheus.yml
```

Under `alerting`, add your Alertmanager as a target:

```
alerting:
  alertmanagers:
```

```
  - static_configs:
    - targets: ["localhost:9093"]
```

Restart Prometheus to reload the configuration:

```
sudo systemctl restart prometheus
```

Verify that Prometheus is able to reach the Alertmanager. Access Prometheus in a web browser at `http://<Prometheus server public IP>:9090`.

Click **Status** > **Runtime & Build Information**.

Verify that your Alertmanager instance appears under the `Alertmanagers` section.

Alertmanager Configuration

Documentation:

- **Alertmanager Configuration**

Alertmanager is configured in much the same way as Prometheus server.

The location of a configuration file is defined in the `--config.file` command-line flag when running Alertmanager.

This configuration file is a YAML file containing configuration data for Alertmanager.

Alertmanager configurations are reloaded similarly to Prometheus server configurations.

There are multiple ways to reload the Alertmanager configuration:

- Restart Alertmanager.
- Send a *SIGHUP* signal to the Alertmanager process (e.g., `sudo killall -HUP alertmanager`).

- Send an *HTTP POST* request to the `/-/reload` endpoint.

High Availability and Alertmanager

Documentation:

- **Alertmanager - High Availability**
- **Alertmanager Github - High Availability**

You can run Alertmanager in a high-availability configuration with multiple Alertmanager instances. These instances will work with one another to de-duplicate and group alerts, even if the alerts are sent to different instances.

Use `--cluster` flags to configure a multi-instance Alertmanager cluster.

Note that Prometheus server should be aware of each Alertmanager instance. Do not load-balance traffic between Prometheus server and Alertmanager.

Prometheus Alerts

Alerting Rules

Documentation:

- **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`.

Create rules by defining them within YAML files in the appropriate location.

An example alerting rule:

```
groups:
- name: example
  rules:
  - alert: HighRequestLatency
    expr: job:request_latency_seconds:mean5m{job="myjob"} > 0.5
    for: 10m
    labels:
      severity: page
    annotations:
      summary: High request latency
```

You can view the current status of your alerts in your browser at `http://<Prometheus Address>:9090/alerts`.

Managing Alerts

Documentation:

- **Alertmanager**
- **Alertmanager Configuration**

Prometheus simply triggers alerts based on data. Once you have created some alerting rules, you can use Alertmanager to provide more sophisticated management of your alerts.

Routing

Alertmanager implements a *routing tree*, which is represented in the `route` block in your Alertmanager config file (`alertmanager.yml`).

The routing tree controls the logic of how and when alerts will be sent.

Grouping

Grouping allows you to combine multiple alerts into a single notification.

Example: Multiple servers just went down. Instead of getting an email for each server, you get one alert email telling you how many servers went down.

Use the `group_by` block in your routes to set which labels will be used to group alerts together.

Inhibition

Inhibition allows you to suppress an alert if another alert is already firing.

Example: Your data center just lost network connectivity. Instead of getting alerts for every application that is unreachable, the network connectivity alert suppresses all of these other alerts.

Use the `inhibit_rule` block in your Alertmanager config to set matchers for which alerts will inhibit which other alerts.

Silences

Silences are a simple way to temporarily turn off certain notifications.

Example: Your infrastructure is having widespread issues you are already aware of. You use silences to temporarily stop the alerts while you fix the issue.

Silences are configured through the Alertmanager web UI.

Advanced Concepts

Using Multiple Prometheus Servers

High Availability

Documentation:

- **Can Prometheus be made highly-available?**

High-availability systems are systems that are resilient and durable. They are capable of operating for long periods of time without failure.

Can Prometheus be made highly-available?

Making Prometheus highly-available is actually a fairly simple process. You can run multiple Prometheus servers so that if one fails, there are others still working.

You can simply run multiple Prometheus servers with the same configuration.

Each server will scrape metrics separately from all the exporters based on the same configuration. This simplicity is one advantage of the pull-based method of gathering metrics!

Federation

Documentation:

- **Federation**

- **Scaling and Federating Prometheus**

Federation is the process of one Prometheus server scraping some or all time-series data from another Prometheus server.

Hierarchical Federation - Higher-level Prometheus servers collect time-series data from multiple lower-level servers.

Use case: There are multiple data centers, each with its own internal Prometheus server(s) monitoring things within the data center. A higher-level Prometheus server scrapes and aggregates data across all data centers.

Cross-Service Federation - A Prometheus server monitoring one service or set of services scrapes selected data from another server monitoring a different set of services, so queries and alerts can be run on the combined data set.

Use case: A Prometheus server monitors server metrics, and another Prometheus server monitors application metrics. The application-monitoring Prometheus server scrapes CPU usage data from the server-monitoring Prometheus server, allowing queries to be run that take into account both CPU usage and application-based metrics.

Federation can be set up by configuring a Prometheus server to scrape from the `/federate` endpoint on another Prometheus server:

- The `/federate` endpoint allows you to retrieve the current value for selected time series.
- Narrow the time-series data you are retrieving with the `match[]` parameter.
- Match on metrics names and/or labels to retrieve a subset of the data.

Security

Prometheus Security Assumptions

Documentation:

- **Prometheus Security Model**

Prometheus Server security considerations:

- Prometheus server does not provide authentication out of the box. Anyone who can access the server's HTTP endpoints has access to all your time-series data.
- Prometheus server does not provide TLS encryption. Non-encrypted communications between clients and Prometheus are vulnerable to unencrypted data being read and to man-in-the middle attacks.
- If your Prometheus endpoints are open to a network with potentially untrusted clients, you can add your own security layer on top of Prometheus server using a reverse proxy.

Be sure to consider all potentially unsecured traffic in your Prometheus setup. If your network configuration would allow untrusted users to gain access to sensitive components or data, you may need to take steps to secure your Prometheus setup.

A *reverse proxy* acts as a middleman between clients and a server. You can implement security features on top of Prometheus using a reverse proxy. You can use any simple web server, such as Apache or Nginx, for this purpose.

Alertmanager security considerations:

- Like Prometheus server, Alertmanager does not provide authentication or TLS encryption.
- Use a reverse proxy to add your own security layer, if needed.

Pushgateway security considerations:

- Pushgateway likewise does not provide authentication or TLS encryption. Again, you can add your own security layer with a reverse proxy.

Exporter security considerations:

- Every exporter is different.
- Many exporters do provide authentication and/or TLS encryption.
- Check the documentation for your exporters to learn more about basic security features.

- Without security, data provided by exporters can be read by anyone with access to the /metrics endpoint.

Client Libraries

Introduction to Prometheus Client Libraries

Documentation:

- **Client Libraries**
- **Writing Client Libraries**

Prometheus *client libraries* provide an easy way to add instrumentation to your code in order to monitor your applications with Prometheus.

Client libraries provide functionality that allows you to:

- Collect and record metric data in your code
- Provide a `/metrics` endpoint, turning your application into a Prometheus exporter so Prometheus can scrape metrics from your application

There are existing client libraries for many popular programming languages and frameworks. You can also code your own client libraries.

Prometheus supports the following official client libraries, although there are many third-party client libraries for other languages:

- Go
- Java/Scala
- Python
- Ruby

Using the Prometheus Java Client Library

Documentation:

- **Java Client Library**
- **Client Libraries**

The *Java client library* allows you to instrument your Java applications for Prometheus monitoring.

The Java client library provides:

- An API for registering metrics using any of the standard Prometheus metric types
- Several collectors that automatically collect certain metrics out of the box
- An HTTP exporter that exposes a standard Prometheus `/metrics` endpoint

Check out the documentation on GitHub for more information about what the Java client library can do!