INSTALLING AND CONFIGURING NODE EXPORTER

Copyright © 2020 Anisa Co.

Download Node_Exporter and Extracting

```
# wget \
https://github.com/prometheus/node_exporter/releases/download/v1.0.1/n
ode_exporter-1.0.1.linux-amd64.tar.gz
# tar -xvzf node_exporter-1.0.1.linux-amd64.tar.gz
# useradd -rs /bin/false nodeusr
# mv node_exporter-0.17.0.linux-amd64/node_exporter /usr/local/bin/
```


Copyright © 2020 Anisa Co.

Configuring Systemd

```
# vim /etc/systemd/system/node_exporter.service
[Unit]
Description=Node Exporter
After=network.target

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

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


Copyright © 2020 Anisa Co.

Configuring firewall and start Node_Exporter

```
# systemctl daemon-reload
# systemctl start node_exporter
# systemctl enable node_exporter
# firewall-cmd --add-port=9100/tcp --permanent
# firewall-cmd -reload
# curl http://localhost:9100/metrics
```


Copyright © 2020 Anisa Co.

Configuring Prometheus Server

```
# vim /etc/prometheus/prometheus.yml
```

- Add following lines:

```
  - job_name: 'node_exporter_centos'
      scrape_interval: 5s
      static_configs:
        - targets: ['your_IP_Address:9100']
# systemctl restart prometheus
```


Copyright © 2020 Anisa Co.

Prometheus Metrics Fundamentals

Metrics are the core resources that the Prometheus stack ingests to provide you with useful information. Understanding them correctly is essential to fully utilize, manage, or even extend the realm of possibilities this stack has to offer. From data to information, and finally to knowledge, metrics are here to help you. In brief, the following topics will be covered in this chapter:

- ✓ Understanding the Prometheus data model
- ✓ A tour of the four core metric types
- ✓ Longitudinal and cross-sectional aggregations

UNDERSTANDING THE PROMETHEUS DATA MODEL

To understand the Prometheus data model, we need to go through what makes a **time series** and the storage of such data.

Copyright © 2020 Anisa Co.

Time series data

- Time series data can usually be defined as a sequence of numerical data points that are indexed chronologically from the same source.
- In the scope of Prometheus, these data points are collected at a fixed time interval.
- As such, this kind of data, when represented in graphical form, will most commonly plot the evolution of the data through time, with the x axis being time and the y axis being the data value.

Copyright © 2020 Anisa Co.

Time series databases

- It all starts with the need to collect, store, and query measurements over time.
- When dealing with massive amounts of data from collectors and sensors (such as those that make up the Internet of Things), querying the resulting datasets is extremely slow if the database isn't designed with that use case in mind.
  - ✓ Nothing prevents you from using standard relational or NoSQL databases to store time series data, but the performance penalty and scalability concerns should make you ponder on that decision.
- Prometheus chose to implement a time series database that was tailored to its unique problem space.
- Besides the write-heavy aspect of these types of databases, which in turn implies the storage of a massive volume of measurements, it is also important to understand that a simple query can span over several hours, days, or even months, returning a tremendous amount of data points, but is still expected to return data reasonably fast.

Copyright © 2020 Anisa Co.

Time series databases

- **As such, modern time series databases store the following components:**
  - ✓ A timestamp
  - ✓ A value
  - ✓ Some context about the value, encoded in a metric name or in associated key/value pairs
- **An abstract example of data that fits this time series database specification is as follows:**

```
timestamp=1544978108, company=ACME, location=headquarters,
beverage=coffee, value=40172
```

  - ✓ As you can see, this kind of data can be easily stored into a single table in a database.

Copyright © 2020 Anisa Co.

Prometheus local storage

- Local storage is the standard approach for storing data in Prometheus.
- At a very high level, Prometheus storage design is a combination of an index implementation using posting lists for all currently stored labels with their values, and its own time series data format.
- The way Prometheus stores collected data locally can be seen as a three-part process.

Copyright © 2020 Anisa Co.

Data flow

Memory

- The freshest batch of data is kept in memory for up to two hours.
- This includes one or more chunks of data that are gathered during the two-hour time window.
- This approach dramatically reduces disk I/O two fold;
  - the most recent data is available in memory, making it blazingly fast to query;
  - and the chunks of data are created in memory, avoiding constant disk writes.

Write ahead log

- While in memory, data is not persisted and could be lost if the process terminates abnormally.
- To prevent this scenario, a write-ahead log (WAL) in disk keeps the state of the in-memory data so that it can be replayed if Prometheus, for any reason, crashes or restarts.

Disk

- After the two-hour time window, the chunks get written to disk.
- These chunks are immutable and, even though data can be deleted, it's not an atomic operation.
- Instead, tombstone files are created with the information of the data that's no longer required.

Copyright © 2020 Anisa Co.

Layout

- The way data gets stored in Prometheus, as we can see in the following example, is organized into a series of directories (blocks) containing the data chunks, the LevelDB index for that data, a meta.json file with human readable information about the block, and tombstones for data that's no longer required.
  - ✓ Each one of these blocks represents a database.

Copyright © 2020 Anisa Co.

Layout

```
./data
├── 01BKGV7JBM69T2G1BGBGM6KB12
│   └── meta.json
├── 01BKGTZQ1SYQJTR4PB43C8PD98
│   ├── chunks
│   │   └── 000001
│   ├── tombstones
│   ├── index
│   └── meta.json
├── 01BKGTZQ1HHWHV8FBJXW1Y3W0K
│   └── meta.json
├── 01BKGV7JC0RY8A6MACW02A2PJD
│   ├── chunks
│   │   └── 000001
│   ├── tombstones
│   ├── index
│   └── meta.json
├── chunks_head
│   └── 000001
└── wal
    ├── 000000002
    └── checkpoint.00000001
        └── 00000000
```


Copyright © 2020 Anisa Co.

Prometheus data model

- Prometheus stores data as time series, which includes key/value pairs known as labels, a timestamp, and finally a value.
- A time series in Prometheus is represented as follows:

```
<metric_name>[{<label_1="value_1">,<label_N="value_N">}] <datapoint_numerical_value>
```

- it's represented as a metric name, optionally followed by one or more set of label names/values inside curly brackets, and then the value of the metric.
- Additionally, a sample will also have a timestamp with millisecond precision.

Copyright © 2020 Anisa Co.

Prometheus data model

Metric names

- Even though this is an implementation detail, a metric name is nothing more than the value of a special label called "`__name__`".
- Keep in mind that labels surrounded by "`__`" are internal to Prometheus, and any label prefixed with "`__`" is only available in some phases of the metrics collection cycle.
- The combination of labels (key/values) and the metric name defines the identity of a time series.
- Every metric name in Prometheus must match the following regular expression:

```
"[a-zA-Z_:][a-zA-Z0-9_:]*"
```


Copyright © 2020 Anisa Co.

Prometheus data model

Metric labels

- Labels, or the key/value pairs associated with a certain metric, add dimensionality to the metrics.
  - This is an essential part of what makes Prometheus so good at slicing and dicing time series.
- While label values can be full UTF-8, label names have to match a regular expression to be considered valid; for example:

```
"[a-zA-Z0-9_]*"
```


Copyright © 2020 Anisa Co.

Samples

- **Samples are the collected data points, and they represent the numerical value of time series data.**
- **The components that are required to define a sample are a float64 value, and a timestamp with millisecond precision.**
  - ✓ Samples collected out of order will be discarded by Prometheus.
  - ✓ The same happens to samples with the same metric identity and different sample values.

Copyright © 2020 Anisa Co.

Cardinality

- Depending on the computing resources being assigned to a Prometheus instance (that is, CPU, memory, disk space, and IOPS), it will gracefully handle a number of time series.
- Prometheus will only be able to handle that amount of time series without performance degradation.
- This term is often used to mean the number of unique time series that are produced by a combination of metric names and their associated label names/values.
- As an example, a single metric with no additional dimensions (such as labels) from an application that has one hundred instances will naturally mean that Prometheus will store 100 time series, one for each instance (the instance, here, is a dimension that's added outside of the application); another metric from that application that had a label with ten possible values will translate into 1,000 time series (10 time series per instance, times 100 instances).
- Having multiple dimensions with a large number of possible values in a metric will cause what is called a cardinality explosion in Prometheus, which is the creation of a very large number of time series.

Copyright © 2020 Anisa Co.

Cardinality

- When you have label values that don't have a clear limit, which can increase indefinitely or above hundreds of possible values, you will also have a cardinality problem.
- These metrics might be better suited to be handled in logs-based systems.
- The following are some examples of data with high or unbound cardinality that should not be used as label values (or in metric names, for that matter):
  - ✓ Email addresses
  - ✓ Usernames
  - ✓ Request/process/order/transaction ID

Copyright © 2020 Anisa Co.

Four Core Metric Types

- Prometheus metrics are divided into four main types:
  - ✓ Counters
  - ✓ Gauges
  - ✓ Histograms
  - ✓ Summaries

Copyright © 2020 Anisa Co.

Counter

- This is a strictly **cumulative** metric whose value can only increase.
  - ✓ The only exception for this rule is when the metric is reset, which brings it back to zero.
- This is one of the most useful metric types because even if a scrape fails, you won't lose the cumulative increase in the data, which will be available on the next scrape.
- To be clear, in the case of a failed scrape, granularity would be lost as fewer points will be saved.

```
node_cpu_seconds_total[5m]
```


Copyright © 2020 Anisa Co.

Gauge

- A gauge is a metric that snapshots a given measurement at the time of collection, which can increase or decrease (such as temperature, disk space, and memory usage).
  - ✓ If a scrape fails, you will lose that sample, as the next scrape might encounter the metric on a different value (higher/lower).

```
node_memory_MemAvailable_bytes
```


Copyright © 2020 Anisa Co.

Histogram

- Recording numerical data that's inherent to each event in a system can be expensive, so some sort of pre-aggregation is usually needed to conserve at least partial information about what happened.
- However, by pre-calculating aggregations on each instance (such as average since process start, rolling window, exponentially weighted, and so on), a lot of granularity is lost and some calculations can be computationally costly.
- Adding to this, a lot of pre-aggregations can't generally be re-aggregated without losing meaning — the average of a thousand pre-calculated 95th percentiles has no statistical meaning.
- Similarly, having the 99th percentile of request latency collected from each instance of a given cluster (for example) gives you no indication of the overall cluster's 99th percentile and no way to accurately calculate it.
- Histograms allow you to retain some granularity by counting events into buckets that are configurable on the client side, and also by providing a sum of all observed values.

Copyright © 2020 Anisa Co.

Histogram

- Prometheus histograms produce one time series per configured bucket, plus an additional two that track the **sum** and the **count** of observed events.
- Furthermore, histograms in Prometheus are cumulative, which means each bucket will have the value of the previous bucket, plus the number of its own events.
- This is done so that some buckets can be dropped for performance or storage reasons without losing the overall ability to use the histogram.
- The downside of using histograms is that the selected buckets need to fit the range and distribution of values that are expected to be collected.
- This type of metric is especially useful to track bucketed latencies and sizes (for example, request durations or response sizes) as it can be freely aggregated across different dimensions.
- Another great use is to generate heatmaps.

```
prometheus_http_request_duration_seconds_bucket
```


Copyright © 2020 Anisa Co.

Summaries

- Summaries are similar to histograms in some ways, but present different trade-offs and are generally less useful.
- They are also used to track sizes and latencies, and also provide both a sum and a count of observed events.
- Additionally, summaries can also provide pre-calculated quantiles over a predetermined sliding time window.
- The main reason to use summary quantiles is when accurate quantile estimation is needed, irrespective of the distribution and range of the observed events.
  - ✓ Quantiles in Prometheus are referred to as ϕ-quantiles, where $0 \leq \phi \leq 1$.
  - ✓ Both quantiles and sliding window size are defined in the instrumentation code, so it's not possible to calculate other quantiles or window sizes on an ad hoc basis.
- Doing these calculations on the client side also means that the instrumentation and computational cost is a lot higher.
- The last downside to mention is that the resulting quantiles are not aggregable and thus of limited usefulness.
- One benefit that summaries have is that, without quantiles, they are quite cheap to generate, collect, and store.

`go_gc_duration_seconds`

Copyright © 2020 Anisa Co.

Longitudinal and cross-sectional aggregations

- The last concept to grasp when thinking about time series is how aggregations work on an abstract level.
- One of Prometheus' core strengths is that it makes the manipulation of time series data easy, and this slicing and dicing of data usually boils down to two kinds of aggregations, which are often used together:
  - ✓ longitudinal and cross-sectional aggregations.
- In the context of time series, an aggregation is a process that reduces or summarizes the raw data, which is to say that it receives a set of data points as input and produces a smaller set (often a single element) as output.
- Some of the most common aggregation functions in time series databases are minimum, maximum, average, count, and sum.

Copyright © 2020 Anisa Co.

Longitudinal and cross-sectional aggregations

- Let's pretend we've selected {company=ACME, beverage=coffee} and we're now looking at the raw counters over time per location.
- The data would look something like this:

Location/Time	t=0	t=1	t=2	t=3	t=4	t=5	t=6
Factory	1,045	1	2	3	4	5	6
Warehouse	223	223	223	223	224	224	224
Headquarters	40,160	40,162	40,164	40,166	40,168	40,170	40,172

Copyright © 2020 Anisa Co.

Cross-sectional aggregation

- Cross-sectional aggregations are the easiest to understand.
- As we can see in the following data representation, we take a column of data and apply an aggregation function to it:
  - ✓ If we apply the `max()` aggregation, we can find out which location reported more coffees were dispensed.
  - ✓ Applying `count()` would give us the number of offices reporting data for the dimensions that were selected `({company=ACME, beverage=coffee}): 3`.
- This type of aggregation usually applies to the last data points in the requested set.
  - ✓ The most common case where this is not true is when graphing the aggregation over time, as it needs to be calculated for each point in the graph.

Copyright © 2020 Anisa Co.

Cross-sectional aggregation

Location/Time	t=0	t=1	t=2	t=3	t=4	t=5	t=6
Factory	1,045	1	2	3	4	5	6
Warehouse	223	223	223	223	224	224	224
Headquarters	40160	40,162	40,164	40,166	40,168	40,170	40,172

Copyright © 2020 Anisa Co.

Longitudinal aggregation

- **Longitudinal aggregations are trickier to use because you need to select a time window over which to apply the aggregation.**
- **This means they work over rows, as we can see in the following representation:**
  - ✓ Since the current selectors we're using return three rows of data, this means we'll have three results when applying longitudinal aggregations.
  - ✓ In this example, we've selected the last three minutes of data for aggregation.
  - ✓ If we apply the `max()` aggregation over time, since these are counters and there wasn't a reset in the selected window, we will get the latest values in the selected set: **6 for location=factory, 224 for location=warehouse, and 40,172 for location=headquarters**.
    - `count()` will return the number of points that were selected in the specified time range—in this case, since the collection occurs every minute and we requested it for three minutes, it will return 3 for each location.
- **A more interesting aggregation of this kind that wasn't mentioned before is `rate()`.**
- **It is a particularly useful aggregation to use with counters, as you can calculate the rate of change per unit of time.**
- **In this example, rate() would return 1, 0, and 2 for each location, respectively.**

Copyright © 2020 Anisa Co.

Longitudinal aggregation

Location/Time	t=0	t=1	t=2	t=3	t=4	t=5	t=6
Factory	1,045	1	2	3	4	5	6
Warehouse	223	223	223	223	224	224	224
Headquarters	40160	40,162	40,164	40,166	40,168	40,170	40,172

Copyright © 2020 Anisa Co.

RUNNING A PROMETHEUS SERVER

It's time to get our hands on some Prometheus configurations.

In brief, the following topics will be covered in this chapter:

Deep dive into the Prometheus configuration

Managing Prometheus in a standalone server

Copyright © 2020 Anisa Co.

Deep dive into the Prometheus configuration

- One of the key features of prometheus is, owing to incredibly sane default configurations, that it can scale from a quick test running on a local computer to a production-grade instance, handling millions of samples per second without having to touch almost any of its many knobs and dials.
- Having said that, it is very useful to know what configuration options are available to be able to get the most value out of prometheus.
- There are two main types of configuration on a prometheus server:
  - ✓ Command-line flags and
    - Control the parameters that cannot be changed at runtime, such as the storage path or which TCP port to bind to, and need a full server restart to apply any change done at this level.
  - ✓ Operating logic that provided through configuration files.
    - Control runtime configuration, such as scrape job definitions, rules files locations, or remote storage setup.

Copyright © 2020 Anisa Co.

Prometheus startup configuration

- While running a Prometheus server with no startup configuration can be good enough for local instances, it is advisable to configure a couple of basic command-line flags for any serious deployment.
- At the time of writing, Prometheus has almost 30 command-line flags for tweaking several aspects of its operational configuration, grouped by the following namespaces: **config**, **web**, **storage**, **rules**, **alertmanager**, **query**, **and** **log**.
- The `--help` flag does a good job of describing most options, but it can be a bit terse in a few places, so we're going to highlight the ones that are either important for any deployment or whose function is not readily apparent.

Copyright © 2020 Anisa Co.

The config section

- The first thing that is usually important to set is the Prometheus configuration file path, through the `--config.file` flag.
- By default, Prometheus will look for a file named `prometheus.yml` in the current working directory.
- As a side note, this and a storage directory are the only hard requirements for starting a Prometheus server; without a configuration file, Prometheus refuses to start.

Copyright © 2020 Anisa Co.

The storage section

- Following the same logic from the previous section, the `--storage.tsdb.path` flag should be set to configure the base path to the data storage location.
- This defaults to `data/` on the current working directory, and so it is advisable to point this to a more appropriate path—possibly to a different drive/volume, where data can be safely persisted and I/O contention can be mitigated.
  - To note that NFS (AWS EFS included) is not supported, as it doesn't support the POSIX locking primitives needed for safe database files management.
- Placing the Prometheus data storage directory in a network share is also ill-advised as transient network failures would impact the monitoring system's ability to keep functioning.
- The Prometheus local storage can only be written to by a single Prometheus instance at a time.

Copyright © 2020 Anisa Co.

The storage section

- To make sure this is the case, it uses a lock file in the data directory.
  - ✓ On startup, it tries to lock this file using OS specific system calls, and will refuse to start if the file is already locked by another process.
- There can be an edge case to this behavior; when using persistent volumes to store the data directory, there is a chance that, when relaunching Prometheus as another container instance using the same volume, the previous instance might not have unlocked the database.
- This problem would make a setup of this kind susceptible to race conditions.
- Luckily, there is the `--storage.tsdb.no-lockfile` flag, which can be used in exactly this type of situation.
- Be warned though that, in general (and namely, in most Prometheus deployments), it is a bad idea to disable the lock file, as doing so makes unintended data corruption easier.

Copyright © 2020 Anisa Co.

The web section

- The `--web.external-url` flag sets this base URL so that weblinks generated both in the web user interface and in outgoing alerts link back to the Prometheus server or servers correctly.
  - ✓ This might be the DNS name for a load balancer/reverse proxy, a Kubernetes service, or, in the simplest deployments, the publicly accessible, fully qualified domain name of the host running the server.
- For completeness, and as stated in the official documentation, a URL path can also be supplied here when Prometheus is behind some layer seven reverse proxy with content switching (also referred to as location-based switching or URL prefix routing).

Copyright © 2020 Anisa Co.

The web section

- The Prometheus server behaves as a conventional *nix daemon by reloading its configuration file (along with rules files) when it receives a SIGHUP.
- However, there are situations where sending this signal isn't convenient (for example, when running in a container orchestration system such as Kubernetes or using custom-built automation) or even impossible (when running Prometheus on Windows).
- In these situations, the `--web.enable-lifecycle` flag can be used to enable the /-/reload and /-/quit HTTP endpoints, which can be used to control, reload, and shut down, respectively.
- To prevent accidental triggering of these endpoints, and because a GET wouldn't be semantically correct, a POST request is needed.
- This flag is turned off by default as unfettered access to these endpoints pose a security concern.
- Similarly, the `--web.enable-admin-api` flag is also turned off by default for the same reason.
  - ✓ This flag enables HTTP endpoints that provide some advanced administration actions, such as creating snapshots of data, deleting time series, and cleaning tombstones.

Copyright © 2020 Anisa Co.

The web section

- the official Prometheus tarballs also bring two additional directories, `consoles` and `console_libraries`.
- These are needed to enable the native dashboarding capabilities of Prometheus, which are often overlooked.
- These directories contain some preconfigured dashboards (referred to as consoles) and support template libraries, written in the Go templating language.
- Prometheus can be configured to load these by using the `--web.console.templates` and `--web.console.libraries` flags.
- After that, those dashboards will be available at the /consoles endpoint (a link will be available in the main web UI if an index.html file exists)

Copyright © 2020 Anisa Co.

The query section

- This section is all about tuning the inner workings of the query engine.
- Some are fairly straightforward to understand, such as how long a given query can run before being aborted (`--query.timeout`), or how many queries can run simultaneously (`--query.max-concurrency`).
- However, two of them set limits that can have non-obvious consequences.
- The first is `--query.max-samples`, that sets the maximum number of samples that can be loaded onto memory.
- This was done as a way of capping the maximum memory the query subsystem can use (by using it together with `--query.max-concurrency`) to try and prevent the dreaded query-of-death—a query that loaded so much data to memory that it made Prometheus hit a memory limit and then killing the process.
  - ✓ The behavior post 2.5.0 is that if any query hits the limit set by this flag (which defaults to **50,000,000** samples), the query simply fails.
- The second one is `--query.lookback-delta`.
- Without going into too much detail regarding how PromQL works internally, this flag sets the limit of how far back Prometheus will look for time series data points before considering them stale.
- This implicitly means that if you collect data at a greater interval than what's set here (the default being five minutes), you will get inconsistent results in alerts and graphs, and as such, two minutes is the maximum sane value to allow for failures.

Copyright © 2020 Anisa Co.

Prometheus configuration file walkthrough

- The configuration file we mentioned in the previous section declares the runtime configuration for the Prometheus instance.
- As we will see, everything related to scrape jobs, rule evaluation, and remote read/write configuration is all defined here.
- As we mentioned previously, these configurations can be reloaded without shutting down the Prometheus server by either sending a SIGHUP to the process, or by sending an HTTP POST request to the /-/reload endpoint (when `--web.enable-lifecycle` is used at startup).
- At a high level, we can split the configuration file into the following sections:
  - ✓ `global`
  - ✓ `scrape_configs`
  - ✓ `alerting`
  - ✓ `rule_files`
  - ✓ `remote_read`
  - ✓ `remote_write`

Copyright © 2020 Anisa Co.

Prometheus configuration file walkthrough

- A configuration file with the most comprehensive list of options available can be found in the Prometheus project GitHub repository, located in the following address:

https://github.com/prometheus/prometheus/blob/master/config/testdata/conf.good.yml.

Copyright © 2020 Anisa Co.

Prometheus example configuration

```
global:
  scrape_interval: 1m
...
scrape_configs:
  - job_name: 'prometheus'
    scrape_interval: 15s
    scrape_timeout: 5s
    sample_limit: 1000
    static_configs:
      - targets: ['localhost:9090']
    metric_relabel_configs:
      - source_labels: [ __name__ ]
        regex: expensive_metric_.+
        action: drop
```


Copyright © 2020 Anisa Co.

Global configuration

- The global configuration defines the default parameters for every other configuration section, as well as outlining what labels should be added to metrics going to external systems, as shown in the following code block:

```
global:
  scrape_interval: 1m
  scrape_timeout: 10s
  evaluation_interval: 1m
  external_labels:
    dc: dc1
    prom: prom1
```


Copyright © 2020 Anisa Co.

Global configuration

- Duration can only be integer values and can only have one unit.
  - ✓ This means that trying to use 0.5 minutes instead of 30 seconds or one minute 30 seconds instead of 90 seconds will be considered a configuration error.
- `scrape_interval` sets the default frequency targets that should be scraped.
- This is usually between 10 seconds and one minute, and the default 1m is a good conservative value to start.
- Longer intervals are not advisable as the lost granularity (especially in gauges) starts to impact the ability to properly alert on issues and makes querying finicky as you need to be aware that some shorter intervals might not return data.
- Additionally, considering the default loopback delta of five minutes (mentioned in the command-line flags), any `scrape_interval` longer than 150 seconds (2 minutes 30 seconds) will mean every time series for a given target will be considered stale if a single scrape fails.

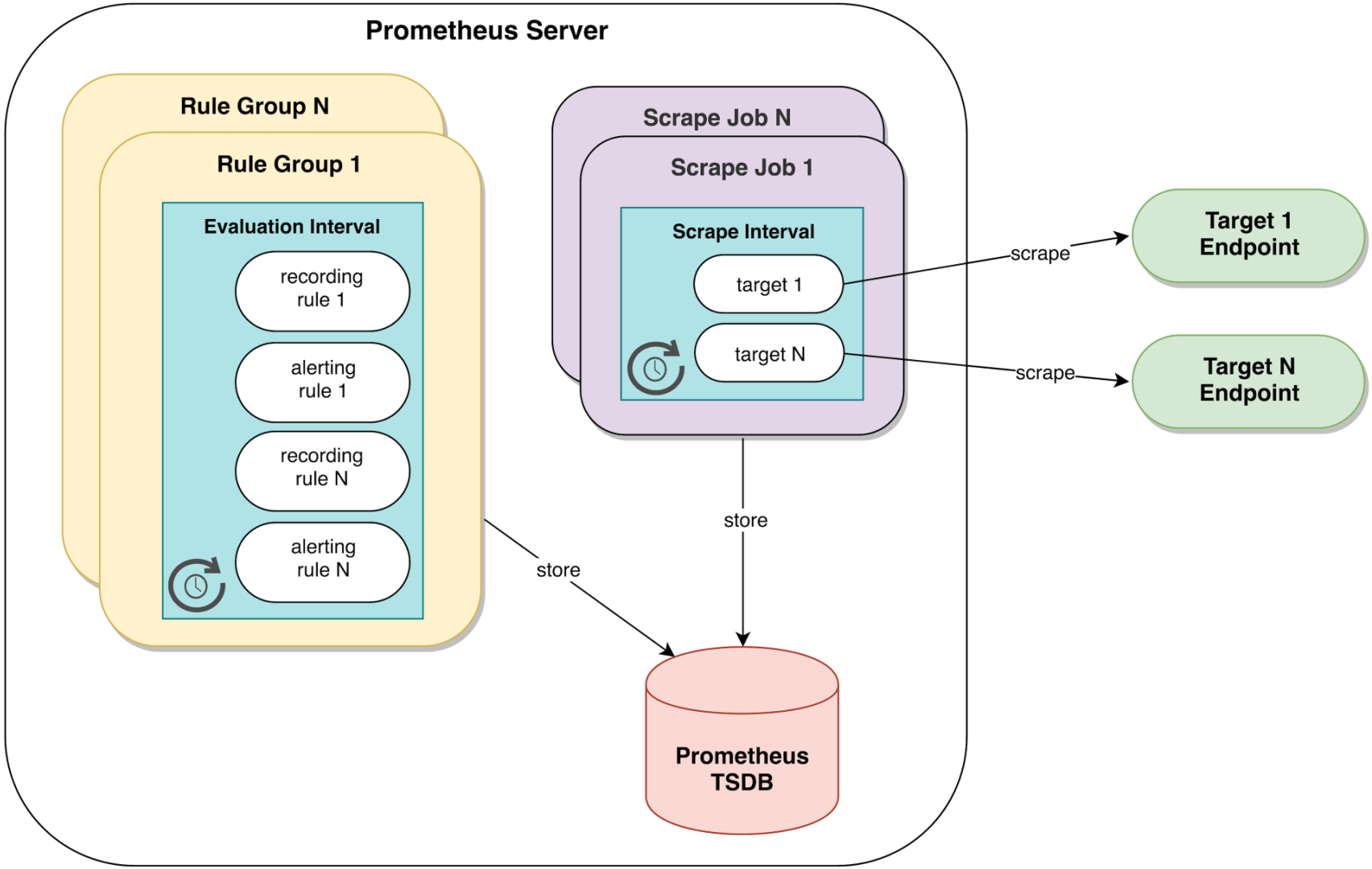
Copyright © 2020 Anisa Co.

Global configuration

- `scrape_timeout` defines how long Prometheus should wait by default for a response from a target before closing the connection and marking the scrape as failed (10 seconds if not declared).
- Bear in mind that even though it is expected that targets respond to scrapes fairly quickly, the guidelines for metrics exposition mandate that collection should happen at scrape time and not cached, which means there can be some exporters that take a bit longer to respond.
- Similar to `scrape_interval`, `evaluation_interval` sets the default frequency recording and alerting rules are evaluated.
- For sanity, both should have the same.

Copyright © 2020 Anisa Co.

Global configuration



Copyright © 2020 Anisa Co.

Global configuration

- Lastly, `external_labels` allows you to set label name/value pairs that are added to time series or alerts going to external systems, such as Alertmanager, remote read and write infrastructure, or even other Prometheis through federation.
- This functionality is usually employed to uniquely identify the source of a given alert or time series; therefore, it is common to identify the region, datacenter, shard, or even the instance identifier of a Prometheus server.

Copyright © 2020 Anisa Co.

Scrape configuration

- Even though Prometheus accepts an empty file as a valid configuration file, the absolute minimum useful configuration needs a `scrape_configs` section.
- This is where we define the targets for metrics collection, and if some post-scrape processing is needed before actual ingestion.
- In Prometheus terms, a scrape is the action of collecting metrics through an HTTP request from a targeted instance, parsing the response, and ingesting the collected samples to storage.
- The default HTTP endpoint used in the Prometheus ecosystem for metrics collection is aptly named **/metrics**.
- A collection of such instances is called a job.

Copyright © 2020 Anisa Co.

Scrape configuration

- The combination of instance and job identify the source of the collected samples, and so these are automatically added as labels to the ingested data, as shown in the following code block:

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

- A scrape job definition needs at least a `job_name` and a set of targets.
- In this example, `static_configs` was used to declare the list of targets for both scrape jobs

Copyright © 2020 Anisa Co.

Scrape configuration

- While Prometheus supports a lot of ways to dynamically define this list, `static_configs` is the simplest and most straightforward method:

```
scrape_configs:
  - job_name: 'prometheus'
    scrape_interval: 15s
    scrape_timeout: 5s
    sample_limit: 1000
    static_configs:
      - targets: ['localhost:9090’]
    metric_relabel_configs:
      - source_labels: [ __name__ ]
        regex: expensive_metric_.+
        action: drop
```


Copyright © 2020 Anisa Co.

Scrape configuration

- We can see that both `scrape_interval` and `scrape_timeout` can be redeclared at the job level, thus overriding the global values.
- As stated before, having varying intervals is discouraged, so only use this when absolutely necessary.
- By setting `sample_limit`, prometheus will ensure that whatever value was set, it will be collected per scrape by not ingesting those samples when their number goes over the limit and marking the scrape as failed.
- This is a great safety net for preventing a cardinality explosion from a target outside of your control impacting the monitoring system.

Copyright © 2020 Anisa Co.