Prometheus

Prometheus

Course Introduction

Prometheus

- Thank you for taking this **Prometheus Course**
  - **Open-source** monitoring solution & time series database
  - Was built by **Soundcloud**
  - **Very active** developer and user community
  - Now its a **standalone** open source project
  - Joined the **Cloud Native Computing Foundation** in 2016
  - Ideal for monitoring on premise as well as cloud workloads

Course Overview

Introduction	Monitoring	Alerting	Internals	Use cases
What is Prometheus	Client Libraries	Introduction	Storage	Grafana Provisioning
Installing Prometheus & Grafana	Pushing metrics	Setting up alerts	Security	Cloudwatch Integration
Concepts	Querying			
Configuration	Service Discovery			
Monitoring Nodes	Exporters			
Architecture				

Course Objectives

- To be able to **use Prometheus**
- To get familiar with the **Prometheus ecosystem**
- To setup a monitoring **platform using**
    - **Prometheus**
- To **create alerts** in Prometheus
- To be able to query Prometheus data

Who is Edward Viaene

- My name is Edward Viaene
- I am a **consultant** and **trainer** in Cloud and Big Data technologies
- I’m a big advocate of **Agile** and **DevOps techniques**
- I held various roles from **banking** to **startups**
- I have a **background** in Unix/Linux, Networks, Security, Risk, and distributed computing
- Nowadays I specialize in everything that has to do with **Cloud** and **DevOps**

Who is Jorn Jambers

- My name is Jorn Jambers
- I am a **freelance DevOps consultant** and **trainer**
- DevOps **advocate**
- Worked in banks, consultancy companies and startups
    - In the latter I found my **passion** for **DevOps**
- I have a **background** in Unix/Linux, Hadoop, DBA, Networks, automations
- Today I help companies **succeed** on the public cloud

Online Training

- **Online training** on Udemy
  - **DevOps**, **Distributed Computing**, **Cloud, Big Data**
  - Using online video lectures
  - 40,000+ enrolled students in 100+ countries

Prometheus

Introduction

Prometheus

- Prometheus is an **Open source monitoring** solution
- Started at SoundCloud around 2012-2013, and was made public in early 2015
- Prometheus provides **Metrics & Alerting**
- It is inspired by Google’s **Borgmon**, which uses time-series data as a datasource, to then send alerts based on this data
- It fits very well in the **cloud native infrastructure**
    - Prometheus is also a member of the **CNCF (Cloud Native Foundation)**

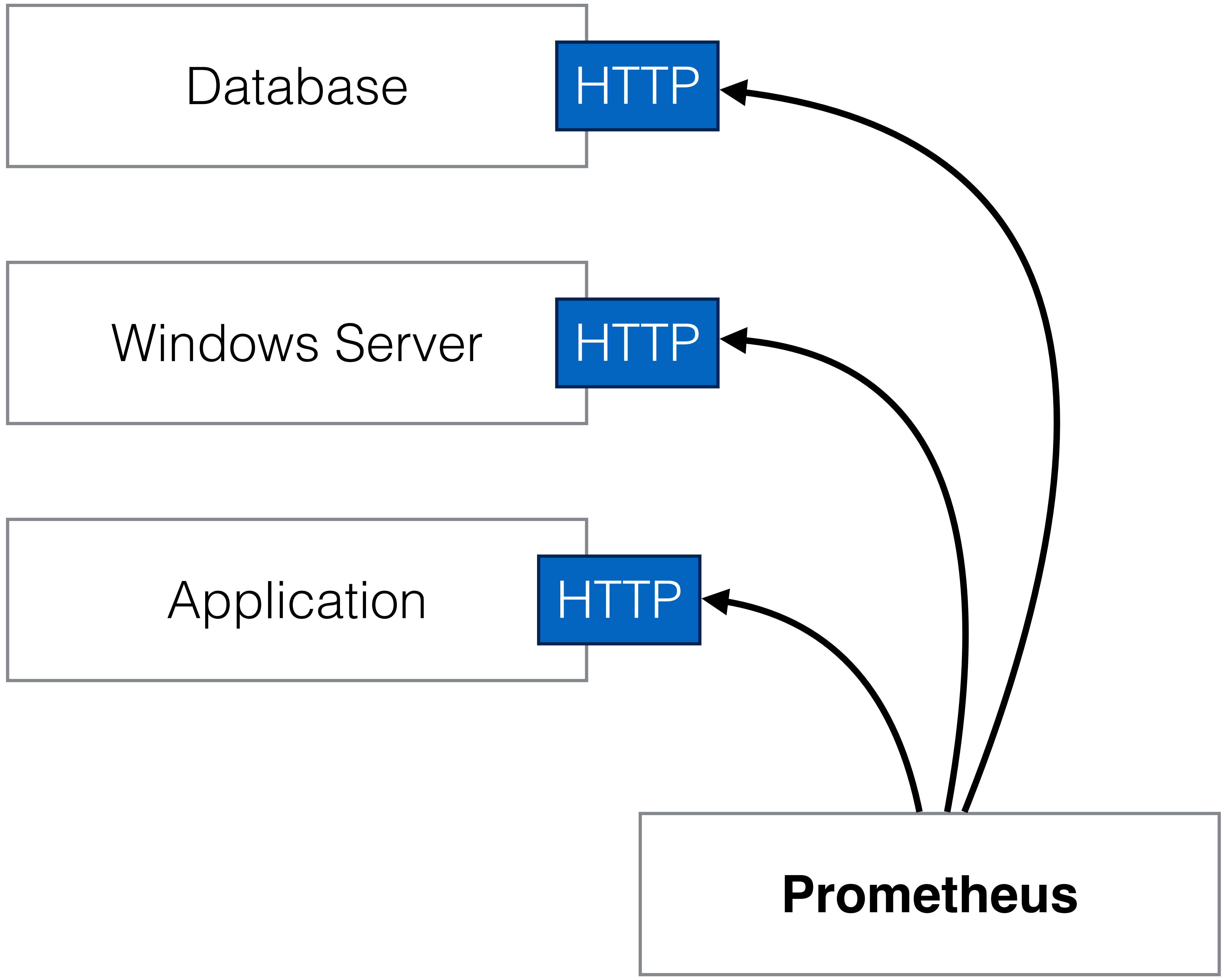
Prometheus

- In Prometheus we talk about **Dimensional Data**: time series are identified by metric name and a set of key/value pairs

Metric name	Label	Sample
Temperature	location=outside	90

- Prometheus includes a Flexible **Query Language**
- **Visualizations** can be shown using a built-in expression browser or with integrations like Grafana
- It stores metrics in **memory** and **local disk** in an own **custom, efficient format**
- It is written in **Go**
- Many **client libraries** and **integrations available**

How does Prometheus work?



- Prometheus collects metrics from monitored targets by **scraping metrics HTTP endpoints**
  - This is fundamentally different than other monitoring and alerting systems, (except this is also how Google's Borgmon works)
  - **Rather than using custom scripts** that check on particular services and systems, the **monitoring data itself is used**
- **Scraping endpoints** is much more efficient than other mechanisms, like 3rd party agents
  - A single prometheus server is able to ingest up to one million samples per second as several million time series

Prometheus

Installation

Prometheus Installation

- I will install **Prometheus using scripts** from our **GitHub** repository (https://github.com/in4it/prometheus-course)
    - They will work on any modern **Linux distribution**
- I'll install it on a **DigitalOcean droplet**
    - Feel free to use the scripts with any **Cloud Provider**, **Virtual Machine**, or **Docker image**, as long as it's a recent Linux distribution
    - To get a free **$100 coupon on DigitalOcean**, valid for 60-days with a valid payment method added, use the following link:

      https://m.do.co/c/b71b388ab76f
    - $10 is enough to run a 2 **GB memory droplet** for one month

Prometheus Installation

- If you do not want to use the provided scripts, you can download the **full distribution** from https://github.com/prometheus/prometheus/releases
- **MacOS**, **Windows**, **Linux**, and some **Unix** distributions are supported
- After extracting you'll get a **prometheus executable** (prometheus.exe for windows), which you can use to run prometheus, for example:
  - ./prometheus --config.file /path/to/prometheus.yaml
- It's best to **use the scripts we provided** so that **your environment** is the **same** as ours when you follow the demos

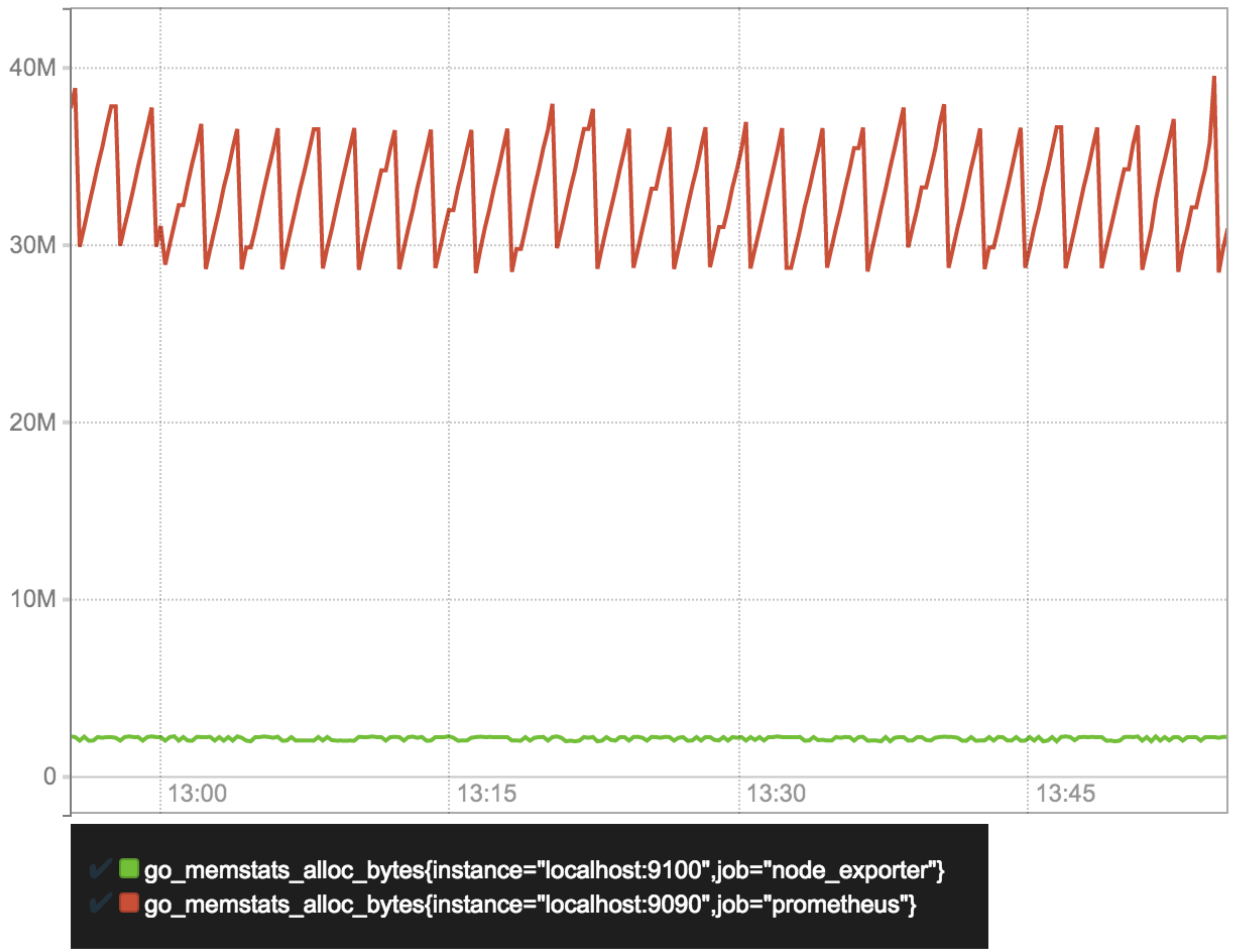
Demo

Installing Prometheus & Grafana

Prometheus

Basic Concepts

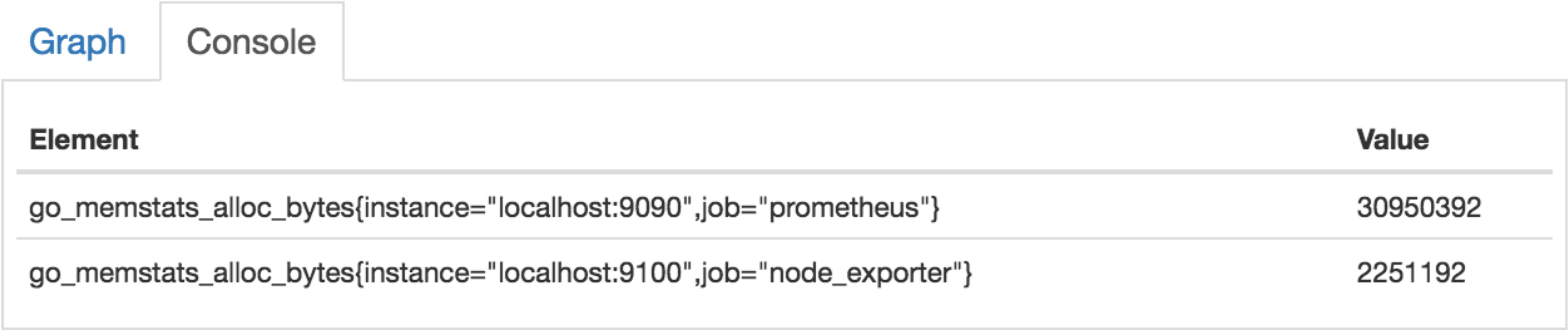
Concepts



- All data is stored as time series
  - Every time series is identified by the “**metric name**” and a set of **key-value pairs,** called **labels**
    - **metric:** go_memstat_alloc_bytes
      - instance=localhost:9090
      - job=prometheus

Prometheus

- The time series data also consists of the **actual data**, called **Samples**:
  - It can be a **float64** value
  - or a **millisecond-precision timestamp**



Element	Value
go_memstats_alloc_bytes{instance="localhost:9090",job="prometheus"}	30950392
go_memstats_alloc_bytes{instance="localhost:9100",job="node_exporter"}	2251192

Prometheus

- The notation of time series is often using this **notation**:
    - <metric name>{<label name>=<label value>, …}
    - For example:
        - node_boot_time{instance="localhost:9100",job="node_exporter"}

Prometheus

Prometheus Configuration file

Prometheus Configuration

- The **configuration** is stored in the Prometheus configuration file, in yaml format
    - The configuration file can be **changed and applied**, without having to restart Prometheus
        - A **reload** can be done by executing kill -SIGHUP <pid>
- You can also pass parameters (flags) at **startup time** to ./prometheus
    - Those parameters cannot be changed without restarting Prometheus
- The configuration file is passed using the flag --config.file

Prometheus Configuration

- The default configuration looks like this:

```
# my global config
global:
  scrape_interval:     15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
  - static_configs:
    - targets:
      # - alertmanager:9093

# Load rules once and periodically evaluate them according to the global 'evaluation_interval'.
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"
```

Prometheus Configuration

- To scrape metrics, you need to add configuration to the prometheus config file

- For example, to scrape metrics from prometheus itself, the following code block is added by default

```
# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: 'prometheus'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ['localhost:9090']
```

Demo

Prometheus Configuration

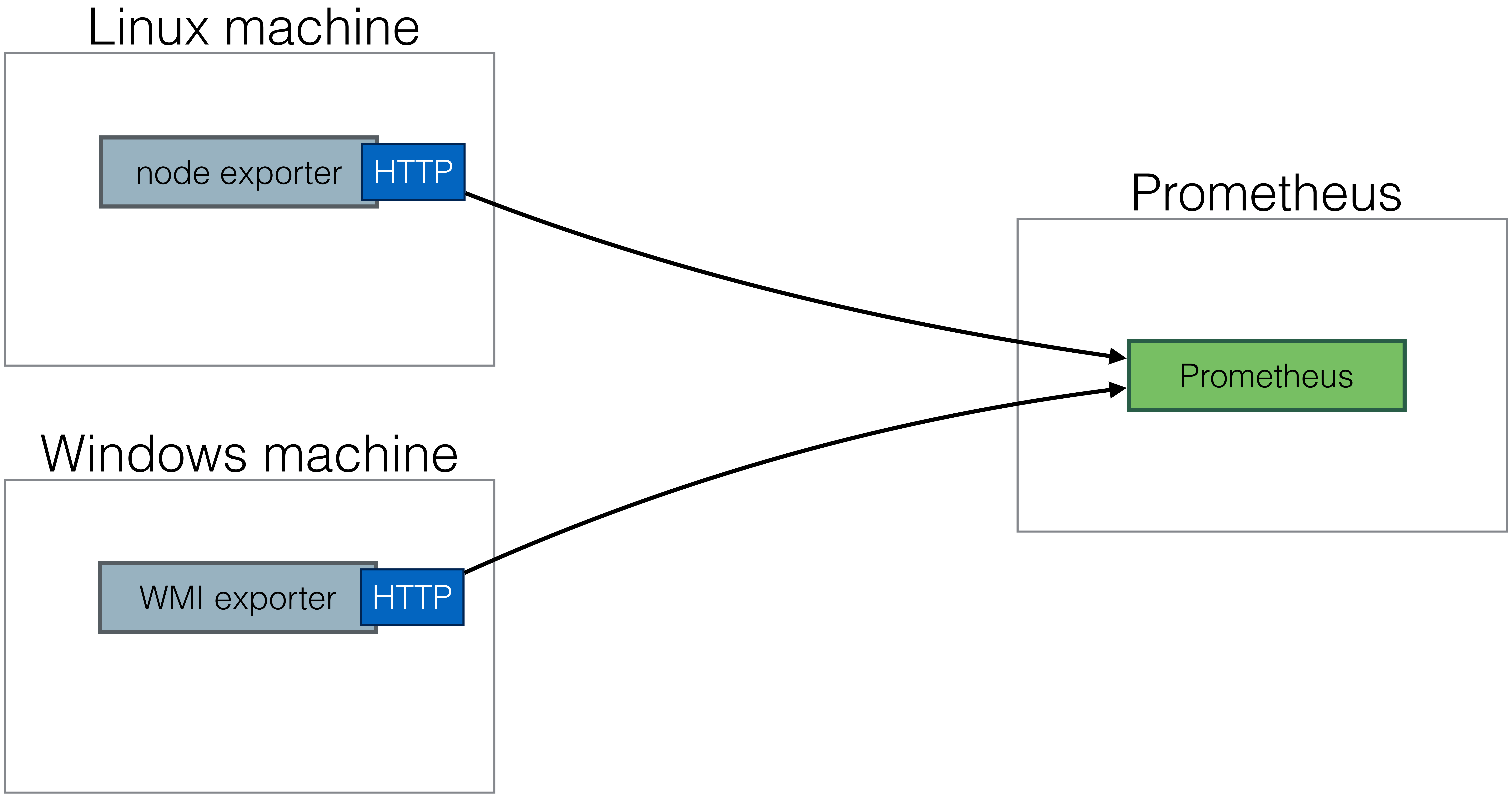
Prometheus

Monitoring Nodes

Monitor nodes

- To monitor nodes, you need to install the node-exporter
- The node exporter will expose machine metrics of Linux / *Nix machines
  - For example: cpu usage, memory usage
- The node exporter can be used to monitor machines, and later on, you can **create alerts based on these ingested metrics**
- For Windows, there's a WMI exporter (see https://github.com/martinlindhe/wmi_exporter)

Monitor nodes



Demo

Node exporter

Demo

WMI exporter

Prometheus

Monitoring

Monitoring - Introduction

- Client Libraries
- Pushing Metrics
- Querying
- Service Discovery
- Exporters

Prometheus

Client Libraries

Client Libraries - Introduction

- Instrumenting your code
- Libraries
  - Official: Go, Java/Scala, Python, Ruby
  - Unofficial: Bash, C++, Common Lisp, Elixir, Erlang, Haskell, Lua for Nginx, Lua for Tarantool, .NET / C#, Node.js, PHP, Rust
- No client library available?
  - Implement it yourself in one of the supported exposition formats

Client Libraries - Introduction

- **Exposition** formats:
  - Simple text-based format
  - Protocol-buffer format **(Prometheus 2.0 removed support for the protocol-buffer format)**

```
metric_name [
  "{" label_name "=" `"` label_value `"` { "," label_name "=" `"` label_value `"` } [ "," ] "}"
] value [ timestamp ]


node_filesystem_avail_bytes{device="/dev/vda1",fstype="ext4",mountpoint="/"} 4.9386491904e+10
node_filesystem_avail_bytes{device="/dev/vda15",fstype="vfat",mountpoint="/boot/efi"} 1.05903104e+08
node_filesystem_avail_bytes{device="lxcfs",fstype="fuse.lxcfs",mountpoint="/var/lib/lxcfs"} 0
node_filesystem_avail_bytes{device="tmpfs",fstype="tmpfs",mountpoint="/run"} 2.01273344e+08
node_filesystem_avail_bytes{device="tmpfs",fstype="tmpfs",mountpoint="/run/lock"} 5.24288e+06
```

Client Libraries - Introduction

- 4 types of metrics
  - **Counter**
  - **Gauge**
  - **Histogram**
  - **Summary**

Client Libraries - Introduction

- **Counter**

  A value that only goes up (e.g. Visits to a website)

Client Libraries - Introduction

- **Gauge**

  Single numeric value that can go up and down (e.g. CPU load, temperature)

Client Libraries - Introduction

- **Histogram**

  Samples observations (e.g. request durations or response sizes) and these observations get counted into **buckets**. Includes (_count and _sum)

  Main purpose is calculating quantiles

Client Libraries - Introduction

- **Summary**

  Similar to a ***histogram***, a ***summary*** samples observations (e.g. request durations or response sizes). A summary also provides a total count of observations and a sum of all observed values, it calculates configurable quantiles over a sliding time window.

  Example: You need 2 counters for calculating the latency
  1) total request(_**count**)
  2) the total latency of those requests (_**sum**)

  Take the rate() and divide = average latency

Prometheus

Instrumentation- Python

Client Libraries - Python Example

- https://github.com/prometheus/client_python
- Officially supported language
- pip install prometheus_client
- Supported metrics: Counter, Gauge, Summary and Histogram

Client Libraries - Python Example

- Python example:

```
import random, time

from flask import Flask, render_template_string, abort
from prometheus_client import generate_latest, REGISTRY, Counter, Gauge, Histogram

app = Flask(__name__)

REQUESTS = Counter('http_requests_total', 'Total HTTP Requests (count)', ['method', 'endpoint', 'status_code'])
IN_PROGRESS = Gauge('http_requests_inprogress', 'Number of in progress HTTP requests')
TIMINGS = Histogram('http_request_duration_seconds', 'HTTP request latency (seconds)')

@app.route('/')
@TIMINGS.time()
@IN_PROGRESS.track_inprogress()
def hello_world():
    REQUESTS.labels(method='GET', endpoint="/", status_code=200).inc()  # Increment the counter
    return 'Hello, World!'

@app.route('/prometheus-course/<name>')
@IN_PROGRESS.track_inprogress()
@TIMINGS.time()
def index(name):
    REQUESTS.labels(method='GET', endpoint="/prometheus-course/<name>", status_code=200).inc()
    return render_template_string('<b>Hello {{name}} welcome!</b>!', name=name)

@app.route('/metrics')
@IN_PROGRESS.track_inprogress()
```

Client Libraries - Python Example

- Python example:

```
import random, time

from flask import Flask, render_template_string, abort
from prometheus_client import generate_latest, REGISTRY, Counter, Gauge, Histogram

app = Flask(__name__)

REQUESTS = Counter('http_requests_total', 'Total HTTP Requests (count)', ['method', 'endpoint', 'status_code'])
IN_PROGRESS = Gauge('http_requests_inprogress', 'Number of in progress HTTP requests')
TIMINGS = Histogram('http_request_duration_seconds', 'HTTP request latency (seconds)')

@app.route('/')
@TIMINGS.time()
@IN_PROGRESS.track_inprogress()
def hello_world():
    REQUESTS.labels(method='GET', endpoint="/", status_code=200).inc()  # Increment the counter
    return 'Hello, World!'

@app.route('/prometheus-course/<name>')
@IN_PROGRESS.track_inprogress()
@TIMINGS.time()
def index(name):
    REQUESTS.labels(method='GET', endpoint="/prometheus-course/<name>", status_code=200).inc()
    return render_template_string('<b>Hello {{name}} welcome!</b>!', name=name)

@app.route('/metrics')
@IN_PROGRESS.track_inprogress()
```

Client Libraries - Python Example

- Python example:

```
import random, time

from flask import Flask, render_template_string, abort
from prometheus_client import generate_latest, REGISTRY, Counter, Gauge, Histogram

app = Flask(__name__)

REQUESTS = Counter('http_requests_total', 'Total HTTP Requests (count)', ['method', 'endpoint', 'status_code'])
IN_PROGRESS = Gauge('http_requests_inprogress', 'Number of in progress HTTP requests')
TIMINGS = Histogram('http_request_duration_seconds', 'HTTP request latency (seconds)')

@app.route('/')
@TIMINGS.time()
@IN_PROGRESS.track_inprogress()
def hello_world():
    REQUESTS.labels(method='GET', endpoint="/", status_code=200).inc()  # Increment the counter
    return 'Hello, World!'

@app.route('/prometheus-course/<name>')
@IN_PROGRESS.track_inprogress()
@TIMINGS.time()
def index(name):
    REQUESTS.labels(method='GET', endpoint="/prometheus-course/<name>", status_code=200).inc()
    return render_template_string('<b>Hello {{name}} welcome!</b>!', name=name)

@app.route('/metrics')
@IN_PROGRESS.track_inprogress()
```

Client Libraries - Python Example

- Python example:

```
import random, time

from flask import Flask, render_template_string, abort
from prometheus_client import generate_latest, REGISTRY, Counter, Gauge, Histogram

app = Flask(__name__)

REQUESTS = Counter('http_requests_total', 'Total HTTP Requests (count)', ['method', 'endpoint', 'status_code'])
IN_PROGRESS = Gauge('http_requests_inprogress', 'Number of in progress HTTP requests')
TIMINGS = Histogram('http_request_duration_seconds', 'HTTP request latency (seconds)')

@app.route('/')
@TIMINGS.time()
@IN_PROGRESS.track_inprogress()
def hello_world():
    REQUESTS.labels(method='GET', endpoint="/", status_code=200).inc()  # Increment the counter
    return 'Hello, World!'

@app.route('/prometheus-course/<name>')
@IN_PROGRESS.track_inprogress()
@TIMINGS.time()
def index(name):
    REQUESTS.labels(method='GET', endpoint="/prometheus-course/<name>", status_code=200).inc()
    return render_template_string('<b>Hello {{name}} welcome!</b>!', name=name)

@app.route('/metrics')
@IN_PROGRESS.track_inprogress()
```

Client Libraries - Python Example

- Python example:

```
import random, time

from flask import Flask, render_template_string, abort
from prometheus_client import generate_latest, REGISTRY, Counter, Gauge, Histogram

app = Flask(__name__)

REQUESTS = Counter('http_requests_total', 'Total HTTP Requests (count)', ['method', 'endpoint', 'status_code'])
IN_PROGRESS = Gauge('http_requests_inprogress', 'Number of in progress HTTP requests')
TIMINGS = Histogram('http_request_duration_seconds', 'HTTP request latency (seconds)')

@app.route('/')
@TIMINGS.time()
@IN_PROGRESS.track_inprogress()
def hello_world():
    REQUESTS.labels(method='GET', endpoint="/", status_code=200).inc()  # Increment the counter
    return 'Hello, World!'

@app.route('/prometheus-course/<name>')
@IN_PROGRESS.track_inprogress()
@TIMINGS.time()
def index(name):
    REQUESTS.labels(method='GET', endpoint="/prometheus-course/<name>", status_code=200).inc()
    return render_template_string('<b>Hello {{name}} welcome!</b>!', name=name)

@app.route('/metrics')
@IN_PROGRESS.track_inprogress()
```

Client Libraries - Python Example

- Python example:

```python
import random, time

from flask import Flask, render_template_string, abort
from prometheus_client import generate_latest, REGISTRY, Counter, Gauge, Histogram

app = Flask(__name__)

REQUESTS = Counter('http_requests_total', 'Total HTTP Requests (count)', ['method', 'endpoint', 'status_code'])
IN_PROGRESS = Gauge('http_requests_inprogress', 'Number of in progress HTTP requests')
TIMINGS = Histogram('http_request_duration_seconds', 'HTTP request latency (seconds)')

@app.route('/')
@TIMINGS.time()
@IN_PROGRESS.track_inprogress()
def hello_world():
    REQUESTS.labels(method='GET', endpoint="/", status_code=200).inc()  # Increment the counter
    return 'Hello, World!'

@app.route('/prometheus-course/<name>')
@IN_PROGRESS.track_inprogress()
@TIMINGS.time()
def index(name):
    REQUESTS.labels(method='GET', endpoint="/prometheus-course/<name>", status_code=200).inc()
    return render_template_string('<b>Hello {{name}} welcome!</b>!', name=name)

@app.route('/metrics')
@IN_PROGRESS.track_inprogress()
```

Client Libraries - Python Example

- Python example:

```
import random, time

from flask import Flask, render_template_string, abort
from prometheus_client import generate_latest, REGISTRY, Counter, Gauge, Histogram

app = Flask(__name__)

REQUESTS = Counter('http_requests_total', 'Total HTTP Requests (count)', ['method', 'endpoint', 'status_code'])
IN_PROGRESS = Gauge('http_requests_inprogress', 'Number of in progress HTTP requests')
TIMINGS = Histogram('http_request_duration_seconds', 'HTTP request latency (seconds)')

@app.route('/')
@TIMINGS.time()
@IN_PROGRESS.track_inprogress()
def hello_world():
    REQUESTS.labels(method='GET', endpoint="/", status_code=200).inc()  # Increment the counter
    return 'Hello, World!'

@app.route('/prometheus-course/<name>')
@IN_PROGRESS.track_inprogress()
@TIMINGS.time()
def index(name):
    REQUESTS.labels(method='GET', endpoint=“/prometheus-course/jorn", status_code=200).inc()
    return render_template_string('<b>Hello {{name}} welcome!</b>!', name=name)

@app.route('/metrics')
@IN_PROGRESS.track_inprogress()
```

Client Libraries - Python Example

- Python example:

```
import random, time

from flask import Flask, render_template_string, abort
from prometheus_client import generate_latest, REGISTRY, Counter, Gauge, Histogram

app = Flask(__name__)

REQUESTS = Counter('http_requests_total', 'Total HTTP Requests (count)', ['method', 'endpoint', 'status_code'])
IN_PROGRESS = Gauge('http_requests_inprogress', 'Number of in progress HTTP requests')
TIMINGS = Histogram('http_request_duration_seconds', 'HTTP request latency (seconds)')

@app.route('/')
@TIMINGS.time()
@IN_PROGRESS.track_inprogress()
def hello_world():
    REQUESTS.labels(method='GET', endpoint="/", status_code=200).inc()  # Increment the counter
    return 'Hello, World!'

@app.route('/prometheus-course/<name>')
@IN_PROGRESS.track_inprogress()
@TIMINGS.time()
def index(name):
    REQUESTS.labels(method='GET', endpoint="/prometheus-course/jorn", status_code=200).inc()
    return render_template_string('<b>Hello {{name}} welcome!</b>!', name=name)

@app.route('/metrics')
@IN_PROGRESS.track_inprogress()
```

Client Libraries - Python Example

- Python example:

```
@app.route('/metrics')
@IN_PROGRESS.track_inprogress()
@TIMINGS.time()
def metrics():
    REQUESTS.labels(method='GET', endpoint="/metrics", status_code=200).inc()
    return generate_latest(REGISTRY)


if __name__ == "__main__":
    app.run(host='0.0.0.0')
```

Prometheus

Instrumentation - Go

Client Libraries - Golang Example

- https://github.com/prometheus/client_golang

- Officially supported language

- Easy to implement:

```
package main

import (
    "github.com/prometheus/client_golang/prometheus/promhttp"
    "net/http"
)

func main() {
    http.Handle("/metrics", promhttp.Handler())
    panic(http.ListenAndServe(":8080", nil))
}
```

- Supported metrics: Counter, Gauge, Summary and Histogram

Client Libraries - Golang Example

- Gauge

```
import "github.com/prometheus/client_golang/prometheus"

var jobsInQueue = prometheus.NewGauge(
    prometheus.GaugeOpts{
        Name: "jobs_queued",
        Help: "Current number of jobs queued",
    },
)

func init(){
    promtheus.MustRegister(jobsQueued)
}

func enqueueJob(job Job) {
    queue.Add(job)
    jobsInQueue.Inc()
}

func runNextJob() {
    job := queue.Dequeue()
    jobsInQueue.Dec()

    job.Run()
}
```

Client Libraries - Golang Example

- Gauge

```
import "github.com/prometheus/client_golang/prometheus"

var jobsQueued = prometheus.NewGauge(
    prometheus.GaugeOpts{
        Name: "jobs_queued",
        Help: "Current number of jobs queued",
    },
)

func init(){
    promtheus.MustRegister(jobsQueued)
}

func enqueueJob(job Job) {
    queue.Add(job)
    jobsQueued.Inc()
}

func runNextJob() {
    job := queue.Dequeue()
    jobsInQueue.Dec()

    job.Run()
}
```

Client Libraries - Golang Example

- Adding labels

```
import "github.com/prometheus/client_golang/prometheus"

var jobsQueued = prometheus.NewGaugeVec(
    prometheus.GaugeOpts{
        Name: "jobs_queued",
        Help: "Current number of jobs in the queue",
    },
    []string{"job_type"},
)

func init(){
    promtheus.MustRegister(jobsQueued)
}

func enqueueJob(job Job) {
    queue.Add(job)
    jobsInQueue.WithLabelValues(job.Type()).Inc()
}

func runNextJob() {
    job := queue.Dequeue()
    jobsInQueue.WithLabelValues(job.Type()).Dec()

    job.Run()
}
```

Client Libraries - Golang Example

- Adding labels

```
import "github.com/prometheus/client_golang/prometheus"

var jobsQueued = prometheus.NewGaugeVec(
    prometheus.GaugeOpts{
        Name: "jobs_queued",
        Help: "Current number of jobs in the queue",
    },
    []string{"job_type"},
)

func init(){
    promtheus.MustRegister(jobsQueued)
}

func enqueueJob(job Job) {
    queue.Add(job)
    jobsQueued.WithLabelValues(job.Type()).Inc()
}

func runNextJob() {
    job := queue.Dequeue()
    jobsQueued.WithLabelValues(job.Type()).Dec()

    job.Run()
}
```

Client Libraries - Golang Example

- Histogram

```
import "github.com/prometheus/client_golang/prometheus"

var jobsDurationHistogram = prometheus.NewHistogramVec(
    prometheus.HistogramOpts{
        Name:    "jobs_duration_seconds",
        Help:    "Jobs duration distribution",
        Buckets: []float64{1, 2, 5, 10, 20, 60},
    },
    []string{"job_type"},
)

start := time.Now()
job.Run()
duration := time.Since(start)
jobsDurationHistogram.WithLabelValues(job.Type()).Observe(duration.Seconds())
```

Client Libraries - Golang Example

- Histogram

```
import "github.com/prometheus/client_golang/prometheus"

var jobsDurationHistogram = prometheus.NewHistogramVec(
    prometheus.HistogramOpts{
        Name:    "jobs_duration_seconds",
        Help:    "Jobs duration distribution",
        Buckets: []float64{1, 2, 5, 10, 20, 60},
    },
    []string{"job_type"},
)

start := time.Now()
job.Run()
duration := time.Since(start)
jobsDurationHistogram.WithLabelValues(job.Type()).Observe(duration.Seconds())
```

Client Libraries - Golang Example

- Summary

```
prometheus.NewSummary()
```

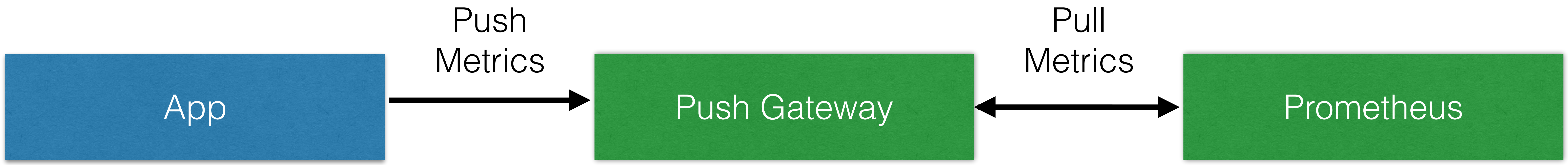
Prometheus

Pushing metrics

Pushing Metrics - Introduction

- https://github.com/prometheus/pushgateway

- Diagram



Pushing Metrics - Introduction

- Sometimes metrics cannot be scraped
  Example: **batch jobs, servers are not reachable due to NAT, firewall**

- **Pushgateway** is used as an intermediary service which allows you to push metrics.

- **Pitfalls**

  - Most of the times this is a single instance so this results in a **SPOF**

  - Prometheus's automatic instance health monitoring is not possible

  - The **Pushgateway** never forgets the metrics unless they are deleted via the api example:

```
curl -X DELETE http://localhost:9091/metrics/job/prom_course/instance/localhost
```

Pushing Metrics - Introduction

- Only 1 valid use case for the Pushgateway
  - Service-level batch jobs and not related to a specific machine
- If NAT or/both firewall is blocking you from using the pull mechanism
  - Move the Prometheus server on the same network

Pushing Metrics - Python Example

- Python example:

```
from prometheus_client import CollectorRegistry, Gauge, push_to_gateway

registry = CollectorRegistry()
g = Gauge('job_last_success_unixtime', ‘Last time the course batch job has finished', registry=registry)
g.set_to_current_time()
push_to_gateway('localhost:9091', job='batchA', registry=registry)
```

- **Pushgateway** functions take a grouping key.
    - **push_to_gateway** replaces metrics with the same grouping key
    - **pushadd_to_gateway** only replaces metrics with the same name and grouping key
    - **delete_from_gateway** deletes metrics with the given job and grouping key.

Prometheus

Pushing Metrics - Go

Pushing Metrics - Go Example

- Go example:

```
package main
import (
    "flag"
    "log"
    "net/http"
    "github.com/prometheus/client_golang/prometheus/promhttp"
    "github.com/prometheus/client_golang/prometheus/push"
)

gatewayUrl:="http://localhost:9091/"

throughputGuage := prometheus.NewGauge(prometheus.GaugeOpts{
        Name: “throughput”,
        Help: "Throughput in Mbps",
})
throughputGuage.Set(800)

if err := push.Collectors(
        "throughput_job", push.HostnameGroupingKey(),
        gatewayUrl, throughputGuage
); err != nil {
    fmt.Println("Could not push completion time to Pushgateway:", err)
}
```

Prometheus

Querying

Querying Metrics - Introduction

- Prometheus provides a functional expression language called PromQL
  - Provides built in operators and functions
  - Vector-based calculations like Excel
  - Expressions over time-series vectors
- PromQL is read-only
- Example:

```
100 - (avg by (instance) (irate(node_cpu_seconds_total{job='node_exporter',mode="idle"}[5m])) * 100)
```

Prometheus

Querying - Expressions

Querying Metrics - Introduction

- **Instant vector** - a set of time series containing a single sample for each time series, all sharing the same timestamp
  Example: `node_cpu_seconds_total`

- **Range vector** - a set of time series containing a range of data points over time for each time series
  Example: `node_cpu_seconds_total[5m]`

- **Scalar** - a simple numeric floating point value
  Example: `-3.14`

- **String** - a simple string value; currently unused
  Example: `foobar`

Prometheus

Querying - Operators

Querying Metrics - Introduction

- **Arithmetic binary operators**
  Example: - (subtraction), * (multiplication), / (division), % (modulo), ^ (power/exponentiation)

- **Comparison binary operators**
  Example: == (equal), != (not-equal), > (greater-than), < (less-than) ,>= (greater-or-equal), <= (less-or-equal)

- **Logical/set binary operators**
  Example: and (intersection), or (union), unless (complement)

- **Aggregation operators**
  Example:**sum** (calculate sum over dimensions), **min** (select minimum over dimensions) ,**max** (select maximum over dimensions), **avg** (calculate the average over dimensions), **stddev** (calculate population standard deviation over dimensions), **stdvar** (calculate population standard variance over dimensions), **count** (count number of elements in the vector), **count_values** (count number of elements with the same value), **bottomk** (smallest k elements by sample value), **topk** (largest k elements by sample value), **quantile** (calculate ϕ-quantile (0 ≤ ϕ ≤ 1) over dimensions)

Demo

Querying

Prometheus

Service Discovery

Service Discovery - Introduction

- Definition:
  Service discovery is the automatic detection of devices and services offered by these devices on a computer network.

- Not really a service discovery mechanism

```
static_configs:
  - targets: ['localhost:9090']
```

- Cloud support for (AWS, Azure, Google,…)

- Cluster managers (Kubernetes, Marathon, …)

- Generic mechanisms (DNS, Consul, Zookeeper, …)

Prometheus

Service Discovery - Example AWS

Service Discovery - Introduction

- EC2 Example:
  Add following config to */etc/prometheus/prometheus.yml*

```
global:
  scrape_interval: 1s
  evaluation_interval: 1s

scrape_configs:
  - job_name: 'node'
    ec2_sd_configs:
      - region: eu-west-1
        access_key: PUT_THE_ACCESS_KEY_HERE
        secret_key: PUT_THE_SECRET_KEY_HERE
        port: 9100
```

- Make sure the user has the following IAM role: **AmazonEC2ReadOnlyAccess**

- Make sure you security groups allow access to port (9100, 9090)

Service Discovery - Introduction

- EC2 Example:
  Only monitor instances started with the name PROD

```
global:
  scrape_interval: 1s
  evaluation_interval: 1s

scrape_configs:
  - job_name: 'node'
    ec2_sd_configs:
      - region: eu-west-1
        access_key: PUT_THE_ACCESS_KEY_HERE
        secret_key: PUT_THE_SECRET_KEY_HERE
        port: 9100
    relabel_configs:
        # Only monitor instances with a tag Name starting with “PROD"
      - source_labels: [__meta_ec2_tag_Name]
        regex: PROD.*
        action: keep
        # Use the instance ID as the instance label
      - source_labels: [__meta_ec2_instance_id]
        target_label: instance
```

Service Discovery - Introduction

- EC2 Example:
  Relabel ip adress to instance id for convenience

```
global:
  scrape_interval: 1s
  evaluation_interval: 1s

scrape_configs:
  - job_name: 'node'
    ec2_sd_configs:
      - region: eu-west-1
        access_key: PUT_THE_ACCESS_KEY_HERE
        secret_key: PUT_THE_SECRET_KEY_HERE
        port: 9100
    relabel_configs:
        # Only monitor instances with a tag Name starting with “PROD"
      - source_labels: [__meta_ec2_tag_Name]
        regex: PROD.*
        action: keep
        # Use the instance ID as the instance label
      - source_labels: [__meta_ec2_instance_id]
        target_label: instance
```

Prometheus

Service Discovery - Example Kubernetes

Service Discovery - Introduction

- Kubernetes Example:
  Add following config to */etc/prometheus/prometheus.yml*

```
  - job_name:'kubernetes'

     kubernetes_sd_configs:
       -
         api_servers:
           - https://kubernetes.default.svc

         in_cluster: true

         basic_auth:
         username: prometheus
         password: secret
         retry_interval:5s

   - job_name:’kubernetes-service-endpoints'

     kubernetes_sd_configs:
       -
         api_servers:
           - https://kube-master.prometheuscourse.com

         in_cluster: true
```

Prometheus

Service Discovery - Example DNS

Service Discovery - DNS

- DNS Example:
  Add following config to */etc/prometheus/prometheus.yml*

```
- job_name: mysql
  dns_sd_configs:
    - names:
      - metrics.mysql.example.com
- job_name: haproxy
  dns_sd_configs:
    - names:
      - metrics.haproxy.example.com
```

Prometheus

Service Discovery - Example using file

Service Discovery - Using file

- File Example:
  Add following config to */etc/prometheus/prometheus.yml*

```
scrape_configs:
  - job_name: 'dummy'  # This is a default value, it is
mandatory.
    file_sd_configs:
      - files:
        - targets.json
```

- Format target.json

```
[
  {
    "targets": [ "myslave1:9104", "myslave2:9104" ],
    "labels": {
      "env": "prod",
      "job": "mysql_slave"
    }
  },
  {
    "targets": [ "mymaster:9104" ],
    "labels": {
      "env": "prod",
      "job": "mysql_master"
    }
  }
]
```

Prometheus

Exporters

Exporters - Introduction

- Build for exporting prometheus metrics from existing third-party metrics

- When Prometheus is not able to pull metrics directly(Linux sys stats, haproxy, …)

- Examples:
  MySQL server exporter
  Memcached exporter
  Consul exporter
  Node/system metrics exporter
  MongoDB
  Redis
  Many more….

- https://prometheus.io/docs/instrumenting/exporters/

Exporters - Introduction

- We are already using one :-)
  Check */etc/prometheus/prometheus.yml*

```
  - job_name: 'node_exporter'
    scrape_interval: 5s
    static_configs:
      - targets: ['localhost:9100']
```

Prometheus

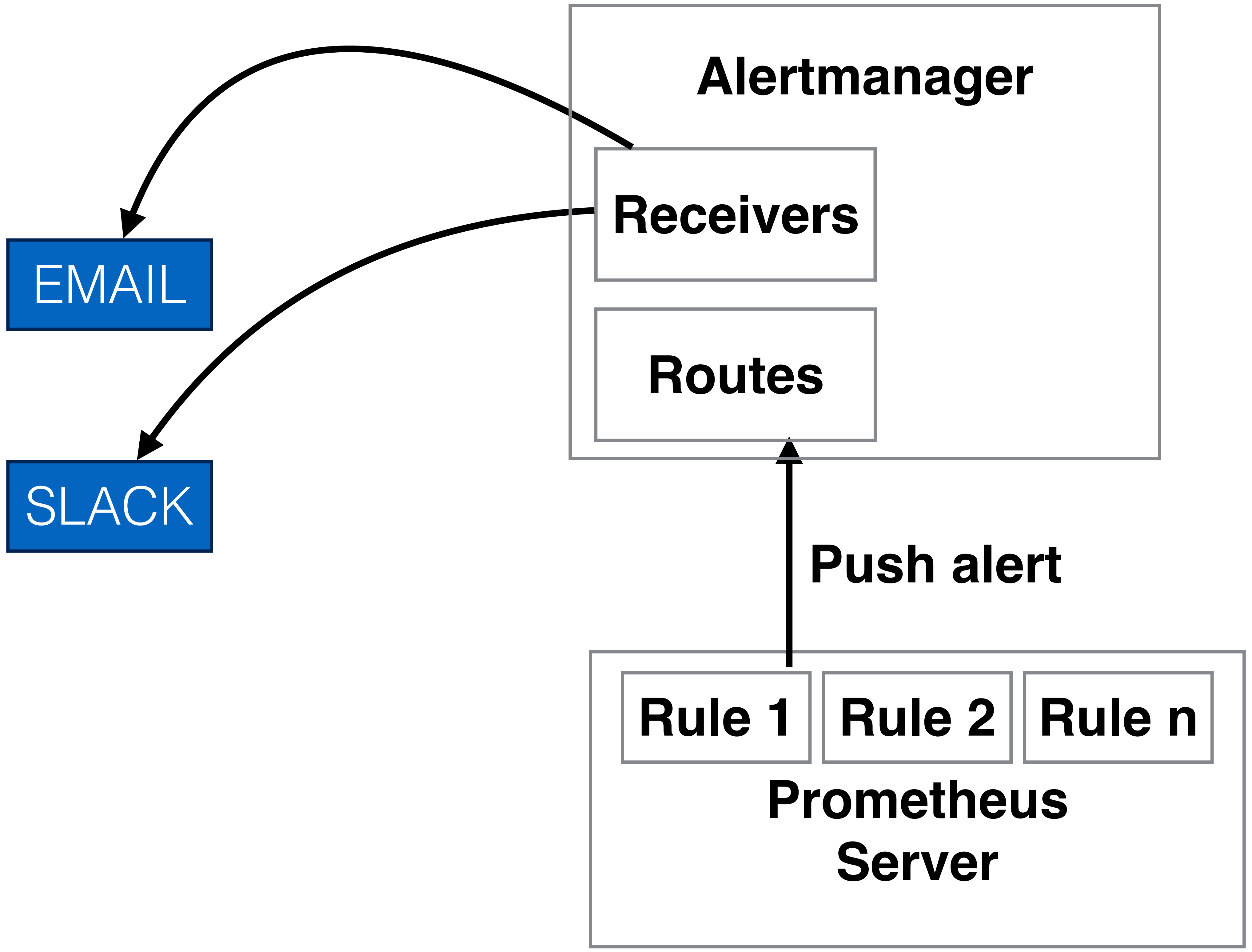
Alerting

Prometheus

Alerting - Introduction

Alerting - Introduction

- Alerting in Prometheus is separated into 2 parts
  - **Alerting rules** in Prometheus server
  - **Alertmanager**



Prometheus

Alerting - Alerting rules

Alerting Rules

- Rules live in Prometheus server config
- Best practice to separate the alerts from the prometheus config
  - Add an include to */etc/prometheus/prometheus.yml*

```
rule_files:
- "/etc/prometheus/alert.rules"
```

- Alert format:

```
ALERT <alert name>
  IF <expression>
  [ FOR <duration> ]
  [ LABELS <label set> ]
  [ ANNOTATIONS <label set> ]
```

- Alert example:

```
groups:
- name: example
  rules:
  - alert: cpuUsge
    expr: 100 - (avg by (instance) (irate(node_cpu_seconds_total{job='node_exporter',mode="idle"}[5m])) * 100) >
95
    for: 1m
    labels:
      severity: critical
    annotations:
      summary: Machine under healvy load
```

Alerting Rules

- Alerting rules allow you to define the alert conditions
- Alerting rules sent the alerts being fired to an external service
- The format of these alerts is in the Prometheus expression language
- Example:

```
groups:
- name: Important instance
  rules:

  # Alert for any instance that is unreachable for >5 minutes.
  - alert: InstanceDown
    expr: up == 0
    for: 5m
    labels:
      severity: page
    annotations:
      summary: "Instance {{ $labels.instance }} down"
      description: "{{ $labels.instance }} of job {{ $labels.job }} has been down for more than 5 minutes."
```

Prometheus

Alerting - Alertmanager

Alertmanager

- **Alertmanager** handles the alerts fired by the prometheus server
- Handles **deduplication**, **grouping** and **routing** of alerts
- **Routes** alerts to **receivers** (Pagerduty, Opsgenie, email, Slack,…)

Alertmanager

- Alertmanager Configuration (*/etc/alertmanager/alertmanager.yml*):

```
global:
  smtp_smarthost: 'localhost:25'
  smtp_from: 'alertmanager@prometheus.com'
  smtp_auth_username: ''
  smtp_auth_password: ''

templates:
- '/etc/alertmanager/template/*.tmpl'

route:
  repeat_interval: 1h
  receiver: operations-team

receivers:
- name: 'operations-team'
  email_configs:
  - to: 'operations-team+alerts@example.org'
  slack_configs:
  - api_url: https://hooks.slack.com/services/XXXXXX/XXXXXX/XXXXXX
    channel: '#prometheus-course'
    send_resolved: true
```

Alertmanager

- Prometheus Configuration (*/etc/prometheus/prometheus.yml*):

```
# my global config
global:
  scrape_interval:     15s # Set the scrape interval to every 15 seconds. Default is every 1 minute.
  evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minute.
  # scrape_timeout is set to the global default (10s).

# Alertmanager configuration
alerting:
  alertmanagers:
  - static_configs:
    - targets:
       - localhost:9093

# Load rules once and periodically evaluate them according to the global 'evaluation_interval'.
rule_files:
  # - "first_rules.yml"
  # - "second_rules.yml"

# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries scraped from this config.
  - job_name: 'prometheus'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
      - targets: ['localhost:9090']
```

Alertmanager

- Concepts:
  - **Grouping**: Groups similar alerts into 1 notification
  - **Inhibition**: Silence other alerts if one specified alert is already fired
  - **Silences**: A simple way to mute certain notifications

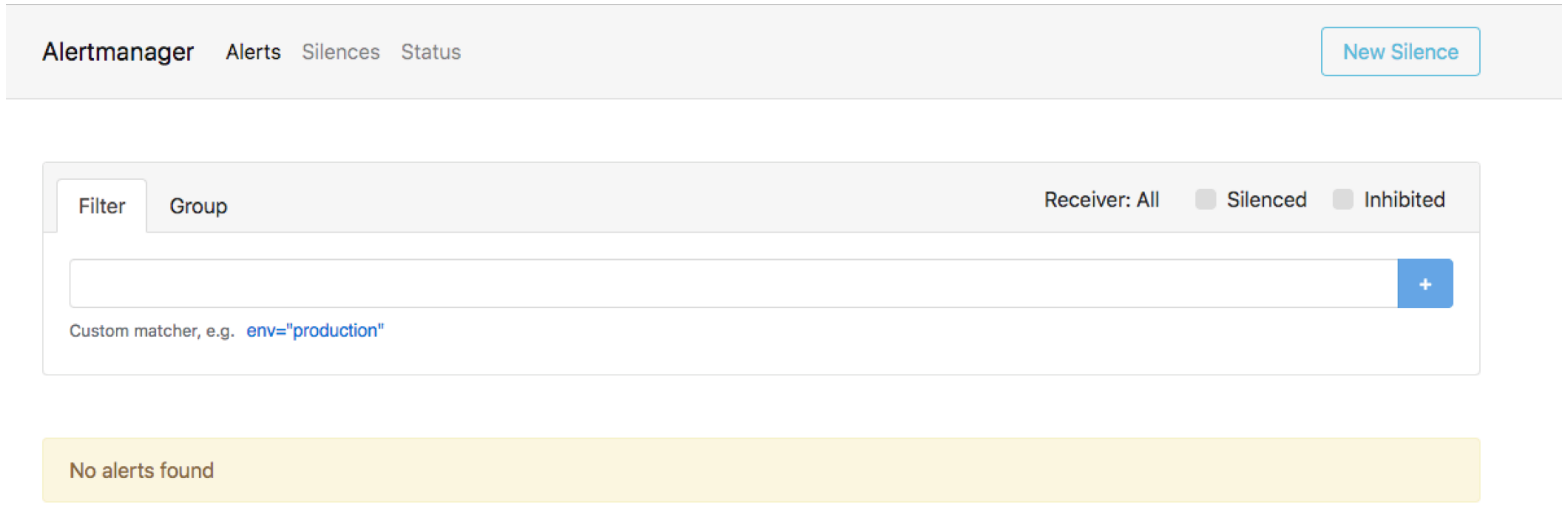
Alertmanager

- **High availability**
  - You can create a high available **Alertmanager** cluster using mesh config
  - Do **not** load balance this service!
    - Use a list of Alertmanager nodes in Prometheus config
  - All alerts are sent to all known Alertmanager nodes
  - No need to monitor the monitoring
  - Guarantees the notification is at least send once

Alertmanager

- **Alert states:**
  Inactive - No rule is met
  Pending - Rule is met but can be suppressed due to validations
  Firing - Alert is sent to the configured channel(mail,Slack,…)

- Runs on port **:9093**



Custom matcher, e.g. env="production"

No alerts found

Alertmanager

- **Notifying multiple destinations**

```
route:
  repeat_interval: 1h
  receiver: operations-team

receivers:
- name: 'operations-team'
  email_configs:
  - to: 'operations-team+alerts@example.org'
  slack_configs:
  - api_url: https://hooks.slack.com/services/XXXXXX/XXXXXX/XXXXXX
    channel: '#prometheus-course'
    send_resolved: true
```

Prometheus

Setting up alerts

Prometheus

Setting up alerts - Demo

Setting up alerts

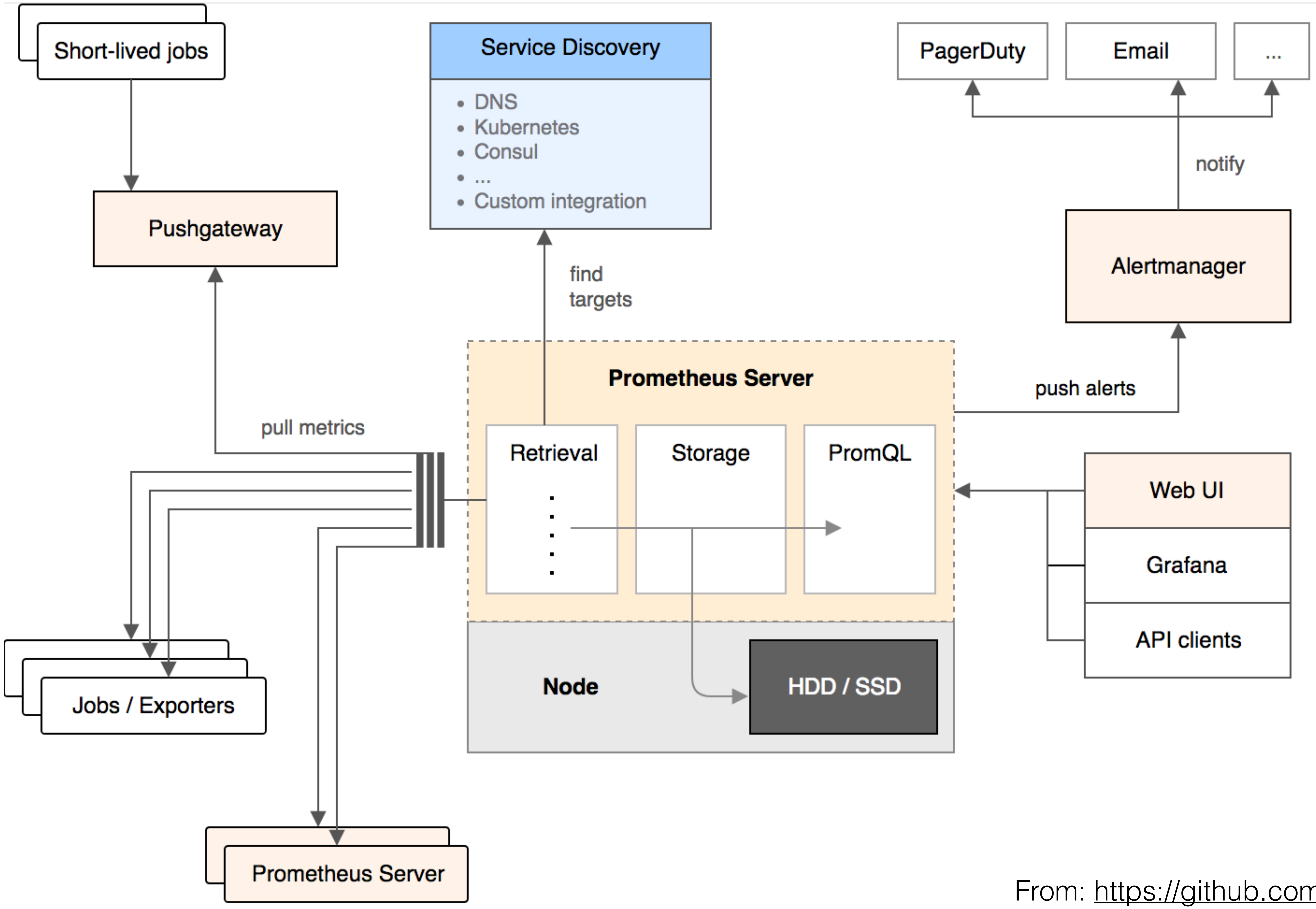
- Install Alertmanager
- Create config for the Alertmanager
  - Mail
  - Slack
- Alter prometheus config
- Setup an alert
  - See the notification coming in when an alert is fired

Prometheus internals

Prometheus

Architecture

Architecture



From: https://github.com/prometheus/prometheus

Prometheus

Storage

Storage

- You can use the default local **on-disk storage,** or optionally the **remote storage system**
    - **Local storage**: a local **time series database** in a custom Prometheus format
    - **Remote storage**: you can read/write samples to a **remote system** in a **standardized format**
        - Currently it uses a **snappy-compressed protocol buffer encoding** over HTTP, but might change in the future (to use gRPC or HTTP/2)

Remote Storage

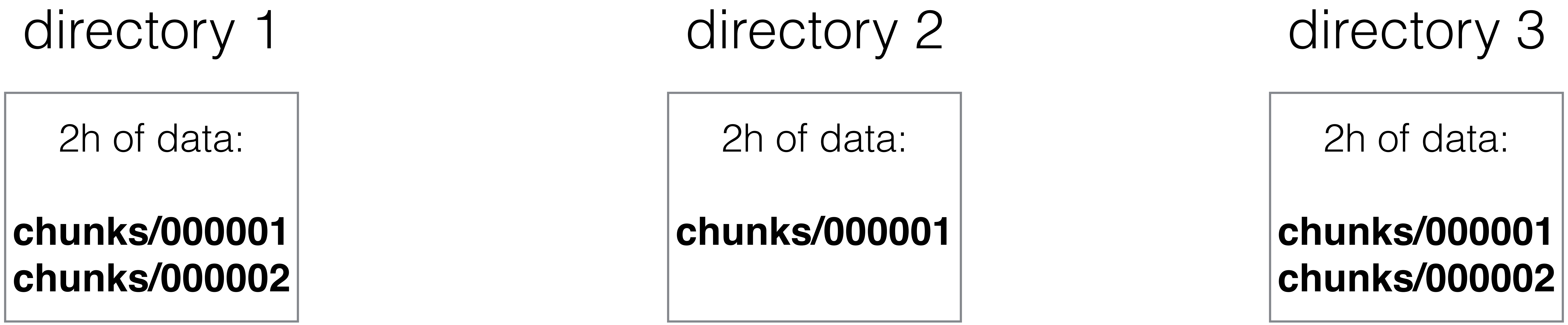
- Remote storage is primarily focussed at long term storage
- Currently there are adapters available for the following solutions:

AppOptics: write	Graphite: write
Chronix: write	InfluxDB: read and write
Cortex: read and write	OpenTSDB: write
CrateDB: read and write	PostgreSQL/TimescaleDB: read and write
Gnocchi: write	SignalFx: write

Source: https://prometheus.io/docs/operating/integrations/#remote-endpoints-and-storage

Local Storage

- Prometheus >=2.0 uses a **new storage engine** which dramatically increases scalability
- Ingested samples are **grouped in blocks of two hours**
- Those 2h samples are stored in **separate directories** (in the data directory of prometheus)
- Writes are batched and written to disk in **chunks**, containing multiple data points



directory 1

2h of data:

chunks/000001
chunks/000002

directory 2

2h of data:

chunks/000001

directory 3

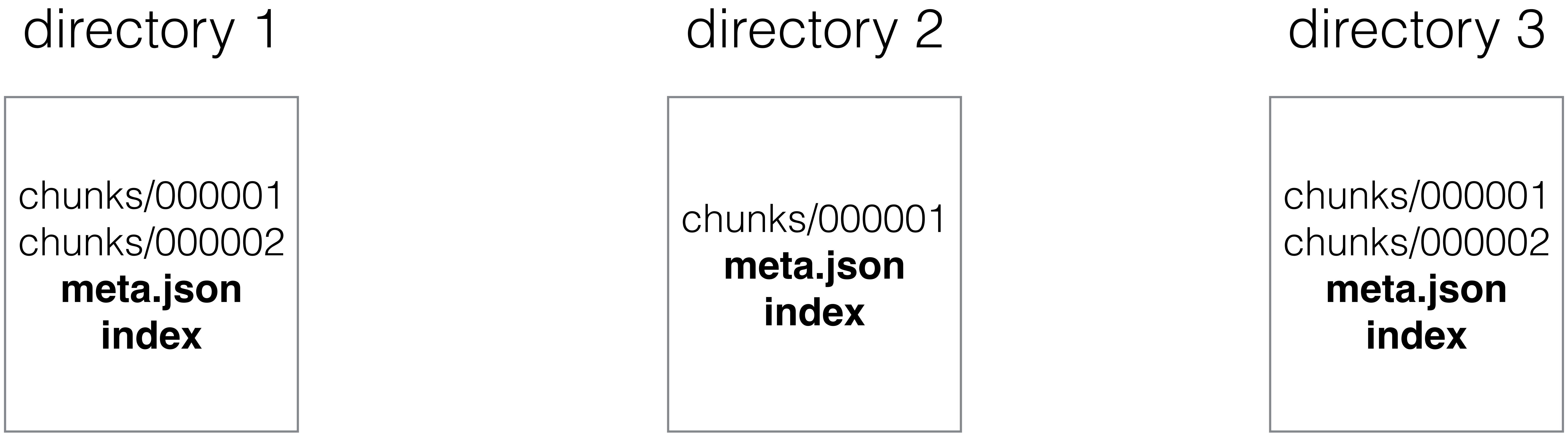
2h of data:

chunks/000001
chunks/000002

Local Storage

- Every directory also has an **index file** (index) and a **metadata file** (meta.json)

- It stores the **metric names** and the **labels**, and **provides an index** from the metric names and labels to the series in the chunk files



directory 1

chunks/000001
chunks/000002
meta.json
index

directory 2

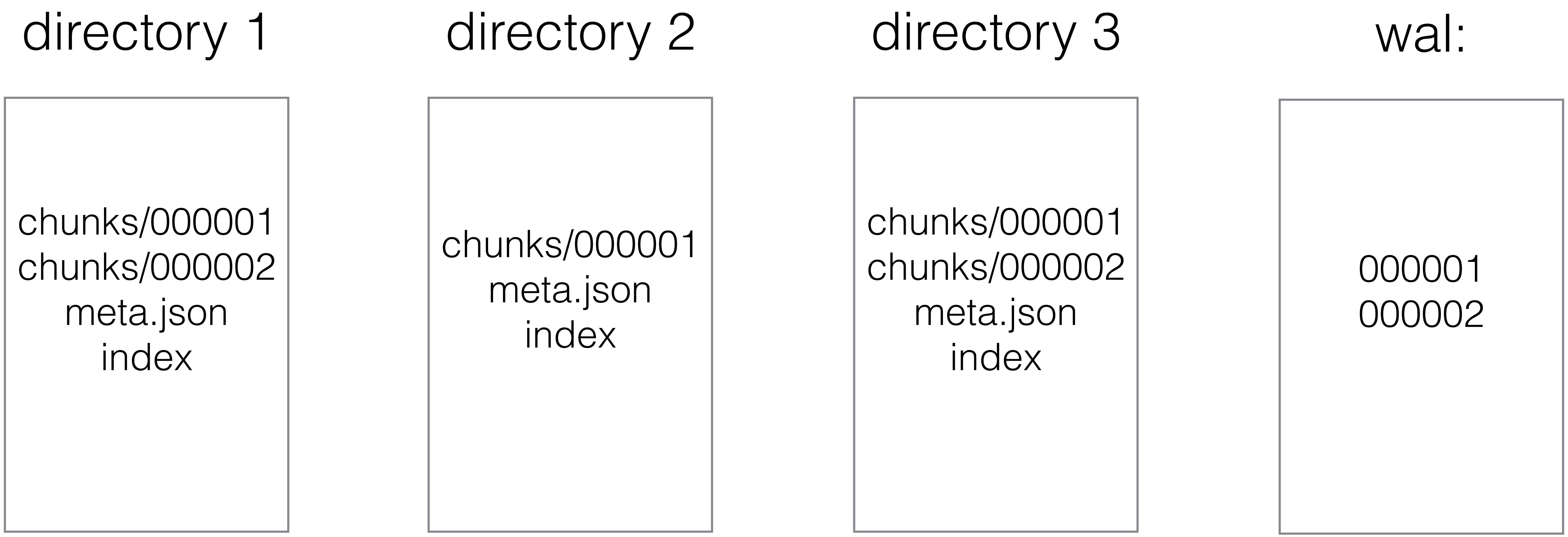
chunks/000001
meta.json
index

directory 3

chunks/000001
chunks/000002
meta.json
index

Local Storage

- The most recent data is kept **in memory**
- You don’t want to loose the in-memory data during a crash, so the **data** also **needs to be persisted to disk**. This is done using a **write-ahead-log** (WAL)



directory 1

chunks/000001
chunks/000002
meta.json
index

directory 2

chunks/000001
meta.json
index

directory 3

chunks/000001
chunks/000002
meta.json
index

wal:

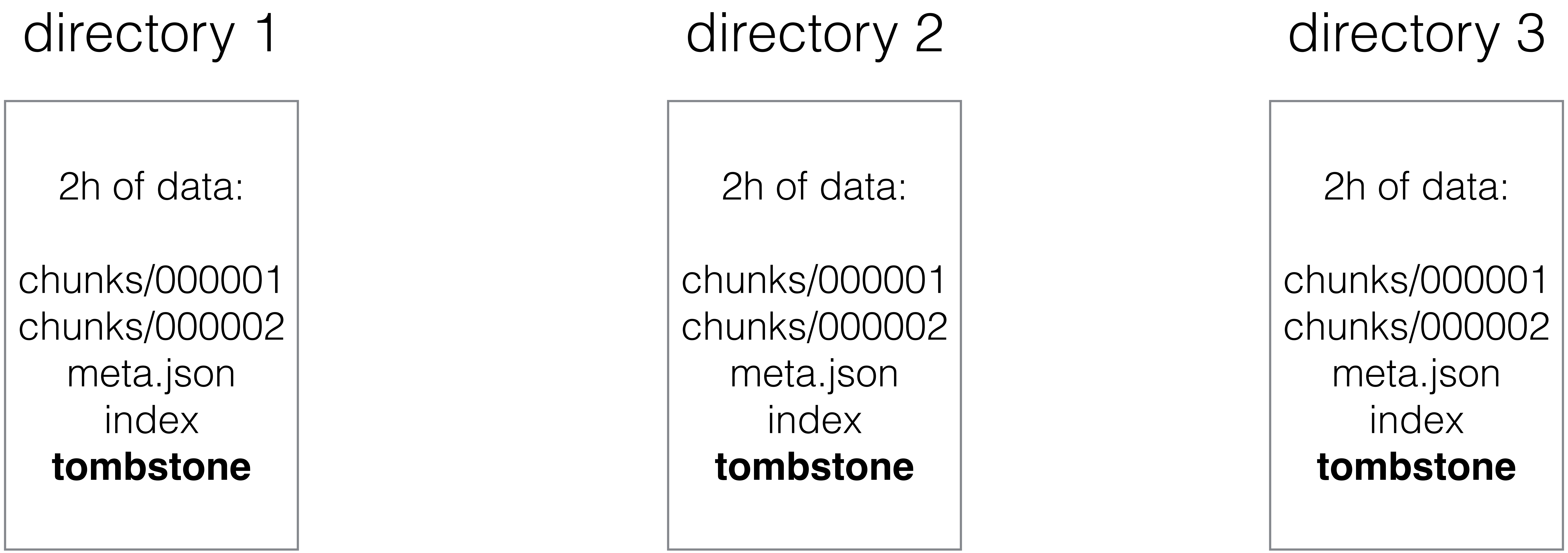
000001
000002

Local Storage

- **Write Ahead Log** (WAL)
  - It's quicker to **append** to a file (like a log) than making (multiple) random read/writes
  - If there's a server crash and the data from memory is lost, then the WAL will be **replayed**
  - This way, **no data will be lost** or corrupted during a crash

Local Storage

- When series gets deleted, a **tombstone file** gets created
- This is **more efficient** than immediately deleting the data from the chunk files, as the **actual delete can happen at a later time** (e.g. when there's not a lot of load)



directory 1

2h of data:

chunks/000001
chunks/000002
meta.json
index
tombstone

directory 2

2h of data:

chunks/000001
chunks/000002
meta.json
index
tombstone

directory 3

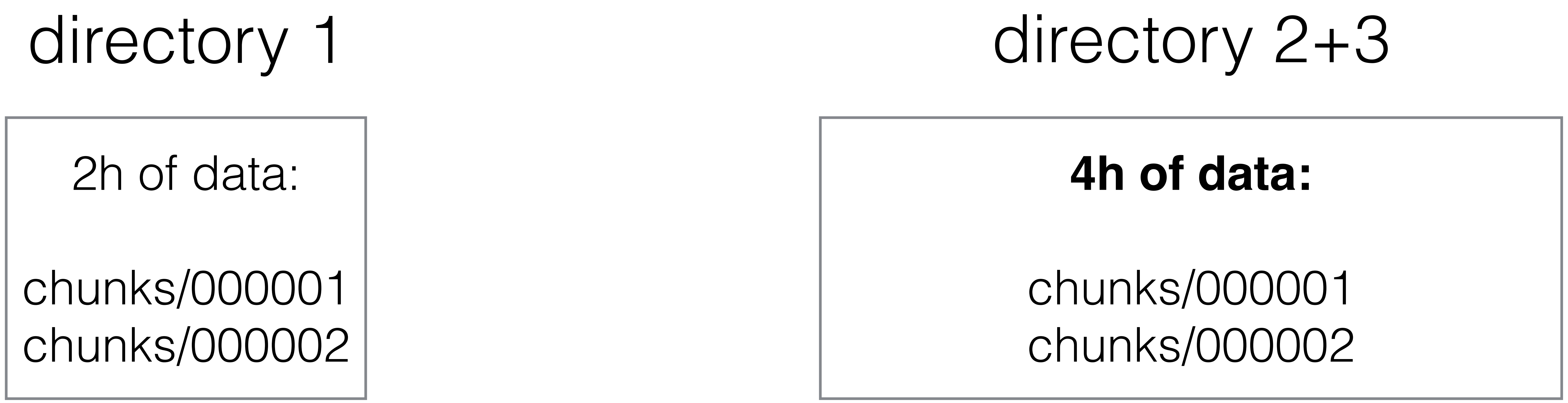
2h of data:

chunks/000001
chunks/000002
meta.json
index
tombstone

Local Storage

- The initial 2-hour blocks are **merged** in the **background** to form longer blocks

- This is called **compaction**



directory 1

2h of data:

chunks/000001
chunks/000002

directory 2+3

4h of data:

chunks/000001
chunks/000002

Local Storage

- **Block characteristics**:
  - A block on the filesystem is a **directory with chunks**
  - You can see each block as a **fully independent database** containing all time series for the window
  - Every block of data, except the current block, is **immutable** (no changes can be made)
  - These non-overlapping blocks are actually a **horizontal partitioning** of the ingested time series data

Local Storage

- This **horizontal partitioning** gives a lot of benefits:
  - When querying, the **blocks not in the time range** can be **skipped**
  - When **completing a block**, data only needs to be **added**, and not modified (avoids write-amplification)
  - **Recent data is kept in memory**, so can be queried quicker
  - **Deleting old data** is only a matter of **deleting directories** on the filesystem

Local Storage

- **Compaction**:
  - **When querying**, blocks have to be **merged** together to be able to **calculate the results**
  - Too many blocks could cause **too much merging overhead**, so blocks are compacted
    - 2 blocks are merged and **form a newly created** (often larger) **block**
    - Compaction can also **modify data**: **dropping deleted data** or **restructuring the chunks** to increase the query performance

Local Storage

- The index:
  - Having **horizontal partitioning** already makes **most queries quicker**, but not those that need to go through all the data to get the result
  - The index is an **inverted index** to provide better query performance, also in cases where all data needs to be queried
    - Each series is assigned a unique ID (e.g. ID 1, 2, and 3)
    - The index will contain an inverted index for the labels, for example for label env=production, it'll have 1 and 3 as IDs if those series contain the label env=production

Local Storage

- What about Disk size?
- On average, Prometheus needs **1-2 bytes per sample**
- You can use the following formula to calculate the disk space needed:

```
needed_disk_space = retention_time_seconds * ingested_samples_per_second * bytes_per_sample
```

Local Storage

- How to reduce disk size?
  - You can increase the scrape interval, which will get you less data
  - You can decrease the targets or series you scrape
  - Or you can can reduce the retention (how long you keep the data)

--storage.tsdb.retention: This determines when to remove old data. Defaults to 15d.

References

- To read the full story of Prometheus time series database, read the blog post from **Fabian Reinartz** at https://fabxc.org/tsdb/

Prometheus

Security

Security

- At the moment Prometheus **doesn’t offer any support for authentication** or encryption (TLS) on the server components
    - They argue that they’re **focussing on building a monitoring solution**, and want to avoid having to implement complex security features
    - You can still enable **authentication** and **TLS**, using a reverse proxy
- This is **only valid for server components**, prometheus can **scrape TLS and authentication** enabled targets
    - See **tls_config** in the prometheus configuration to configure a CA certificate, user certificate and user key
    - You’d still need to setup a reverse proxy for the targets itself

Demo

Prometheus TLS and authentication

Demo

Prometheus mutual TLS for targets

Prometheus Use Cases

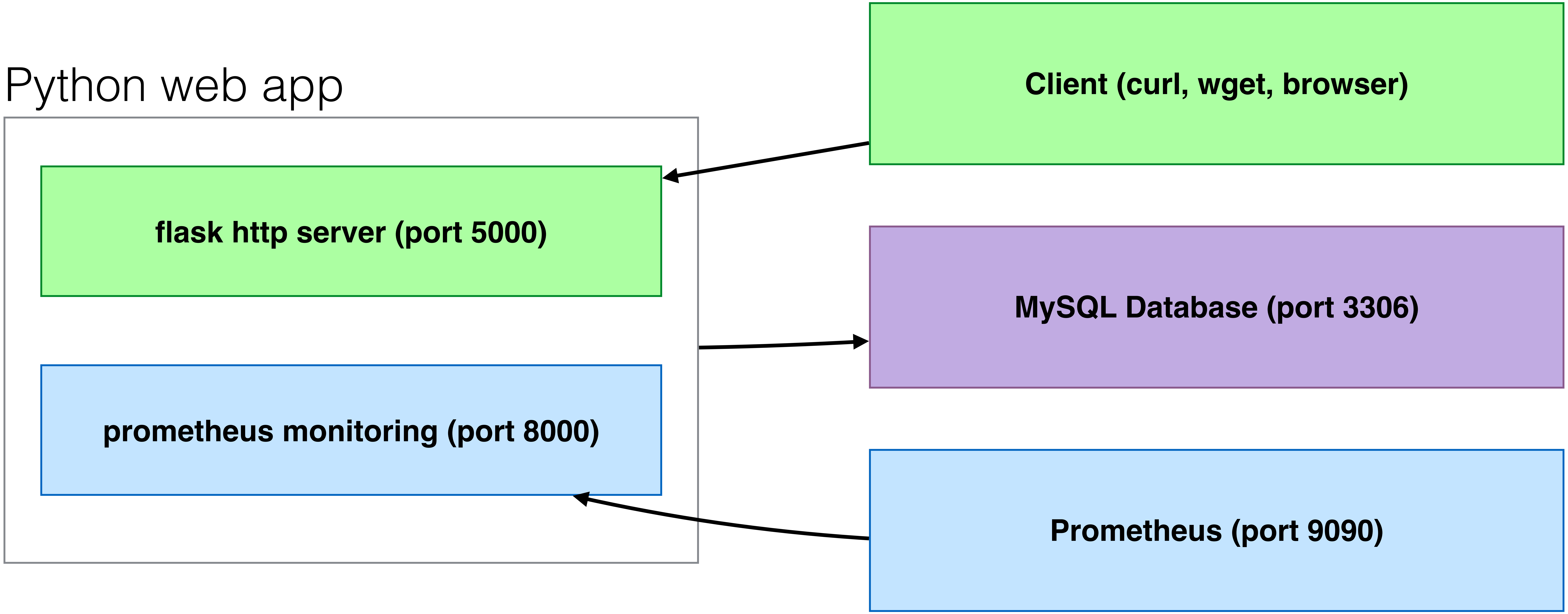
Monitoring a web app

Prometheus with python-flask and MySQL

Monitoring a web app

- I’m going to integrate **prometheus monitoring** with a **web application** based on **python**
    - I’ll use the official **prometheus_client** library for Python
    - **Flask** is the web framework I’m going to use
        - It will create an **http server** and I’ll able to configure routes (e.g. /query)
    - I’ll use **mysqlclient** for python to query a **MySQL** database
        - I’ll include one normal query and one **“bad behaving” query** that will take between 0 and 10 seconds to execute

The web app



Python web app

flask http server (port 5000)

prometheus monitoring (port 8000)

Client (curl, wget, browser)

MySQL Database (port 3306)

Prometheus (port 9090)

Monitoring a web app

- I’m going to use the **Counter** and the **Histogram metric types** to capture the data:
    - A **Counter** to capture the amount of times an **http endpoint** is hit + to capture the amount of times a **MySQL query** is executed
        - The value of the **Counter must always increase**, that’s why you should take the Counter type for these types of data
  - A Histogram to **capture the latency of the HTTP requests** and the MySQL Queries
      - A Histogram **samples observations** (like latencies) and **counts them** in configurable **buckets**. It also provides a **sum of all observed values**.
      - The **default buckets** are intended to **cover a typical web/rpc request** from milliseconds to seconds

Monitoring a web app

- This is how I’m going to define the data types in Python for Prometheus:

```
from prometheus_client import Counter, Histogram

FLASK_REQUEST_LATENCY = Histogram('flask_request_latency_seconds', 'Flask Request Latency',
                      ['method', 'endpoint'])
FLASK_REQUEST_COUNT = Counter('flask_request_count', 'Flask Request Count',
                      ['method', 'endpoint', 'http_status'])

MYSQL_REQUEST_LATENCY = Histogram('mysql_query_latency_seconds', 'MYSQL Query Latency',
                      ['query'])
MYSQL_REQUEST_COUNT = Counter('mysql_query_count', 'Flask Request Count',
                      ['query'])
```

Monitoring a web app

- This is how we can calculate the latency of a query:

```
start_time = time.time()

sql = “select * from table”
# do the query

query_latency = time.time() - start_time

MYSQL_REQUEST_LATENCY.labels(sql[:50]).observe(query_latency)
MYSQL_REQUEST_COUNT.labels(sql[:50]).inc()
```

Demo

Monitor a web application with Prometheus

Demo

Monitor a web application with Prometheus - apdex score

Monitoring a web app

Prometheus with Java Spring boot

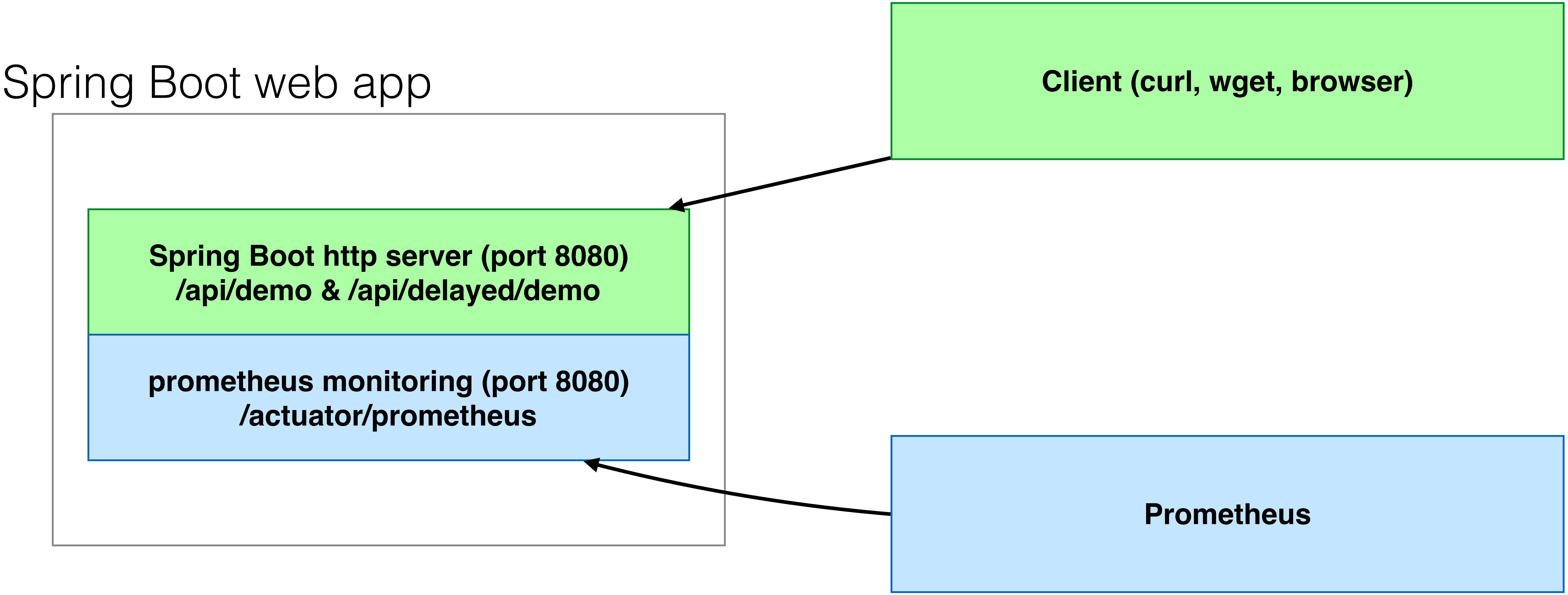
Monitoring a web app

- Introduction
  - Application's **health + Metrics**
  - Notice **unwanted** behaviour
  - Monoliths as well as microservices
    - Crucial in microservices architecture
  - “To measure is to know”

Monitoring a web app

- We are going to integrate **prometheus monitoring** with a **web application** based on **Java Spring Boot**
  - We will use following :
    - **Spring Boot**
    - **Spring Boot Actuator**
    - **Micrometer**
- We are also going to do a demo with an example.

The web app


Spring Boot web app
Client (curl, wget, browser)
Spring Boot http server (port 8080)
/api/demo & /api/delayed/demo
prometheus monitoring (port 8080)
/actuator/prometheus
Prometheus

Monitoring a web app

- **Spring Boot Actuator**
    - Sub-Project of Spring boot
    - **Production** ready endpoints
    - **/actuator** is the common **prefix** of these endpoints
        - **Protected** by default
        - Adjustable in **application.properties**
            - Expose all: **management.endpoints.web.exposure.include=***

Monitoring a web app

- **Micrometer**
  - **Vendor-Neutral** application metrics facade
  - Support for **Prometheus** and many others:
    - AWS Cloudwatch, Datadog, InfluxDB/Telegraf, New Relic, …
  - **Transforms** /actuator/metrics data into data your monitoring system understands
  - Only a **vendor-specific** micrometer dependency in your application is required

Monitoring a web app

- Micrometer
  - pom.xml example

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-core</artifactId>
</dependency>

<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

Monitoring a web app

- Spring Boot
  - Code example

```
…
import io.micrometer.core.instrument.Metrics;
…
private Counter runCounter = Metrics.counter("runCounter");
…
@GetMapping("/api/demo")
@Timed
public String apiUse() throws InterruptedException {
  runCounter.increment();
    log.info("Hello world app accessed on /api/demo");
    return "Hello world";
}
```

Demo

Monitor and instrument a Spring boot application

Grafana

Grafana Provisioning

Grafana Provisioning

- In one of the first lectures I showed you how to **setup Grafana using the UI**
- Rather than using the UI, you can also **use yaml and json files** to provision Grafana with datasources and dashboards
- This is a much more **powerful way** of using Grafana, as you can test new dashboards first on a **dev / test server**, then **import** the newly created dashboards to **production**
  - You can do the import manually through the UI, or **using yaml and json files**
  - When using files, you can keep files within **version control** to keep changes, revisions and backups

Grafana Provisioning

- The configuration of Grafana is all kept in /etc/grafana:

```
/etc/grafana/:
-rw-r----- 1 root grafana  14K Jul 17 12:30 grafana.ini
-rw-r----- 1 root grafana 3.4K Jul 17 12:30 ldap.toml
drwxr-xr-x 4 root grafana 4.0K Jul 17 13:15 provisioning/
/etc/grafana/provisioning/:
drwxr-xr-x 2 root grafana 4.0K Jul 17 14:56 dashboards/
drwxr-xr-x 2 root grafana 4.0K Jul 17 15:34 datasources/
```

- The data is kept in /var/lib/grafana:

```
/var/lib/grafana:
drwxr-xr-x 2 root    root    4.0K Jul 17 15:47 dashboards/
-rw-r----- 1 grafana grafana 500K Jul 17 15:48 grafana.db
drwxr-x--- 2 grafana grafana 4.0K Jul 17 12:31 plugins/
drwx------ 5 grafana grafana 4.0K Jul 17 12:40 sessions/
```

Grafana Provisioning

- You can change the database & paths in /etc/grafana/grafana.ini

```
[paths]
# Path to where grafana can store temp files, sessions, and the sqlite3 db (if that is used)
;data = /var/lib/grafana

# Directory where grafana can store logs
;logs = /var/log/grafana

# Directory where grafana will automatically scan and look for plugins
;plugins = /var/lib/grafana/plugins

# folder that contains provisioning config files that grafana will apply on startup and while running.
;provisioning = conf/provisioning
…
[database]
# Either "mysql", "postgres" or "sqlite3", it's your choice
;type = sqlite3
;host = 127.0.0.1:3306
;name = grafana
;user = root
# If the password contains # or ; you have to wrap it with triple quotes. Ex """#password;"""
;password =
```

Demo

Grafana Provisioning

Prometheus Use Cases

Cloudwatch exporter

Use Cases - Cloudwatch

- Cloudwatch exporter
  - Installation
- Configuration (exporter + AWS)
- Charges + measuring them
- Querying metrics

Prometheus Use Cases

Consul integration

Consul integration

- Consul is a **distributed**, **highly available** solution providing:
    - A Service Mesh
    - Service Discovery
    - Health checks for your services
    - A Key-Value store
    - Secure Service Communications
    - Multi-datacenter support
- Consul is often deployed **in conjunction with Docker**

Consul integration

- There are **2 integrations** that are interesting to use:
    - 1) Prometheus can **scrape** Consul’s metrics and provide you with all sorts of information about your running services
        - Consul provides **Service Discovery**, so it knows where services are running and what the **current state** of it is
    - 2) Consul can be integrated within Prometheus to **automatically add the services as targets**
        - Consul will discover your services, and these can then be automatically added to Prometheus as a target

Consul integration

- In the next demo I'll focus on the **Prometheus integration** with Consul, not really on implementing consul itself
    - I'll show you the **installation of consul**, but not how to integrate consul with your infrastructure (it's out of scope for this Prometheus course)
    - I'll **manually register a service to consul**, rather than setting up service discovery
        - In production environments, where you have a lot of services, service discovery using consul will allow you to register your services automatically
    - If you are interested in Consul, have a look at the documentation at https://www.consul.io/ and you can find the service registrator for docker at https://github.com/gliderlabs/registrator

Demo

Consul integration

Prometheus Use Cases

EC2 auto discovery

EC2 auto discovery

- **Service discovery** is the automatic detection of devices and services offered by these devices on a computer network.
- In this Use Case we will:
  - Create prerequisites in AWS (IAM role, Security Groups, EC2 instances)
  - Alter Prometheus config (*/etc/prometheus/prometheus.yml*)
  - Query the data in Grafana
  - https://github.com/in4it/prometheus-course/blob/master/use-cases/ec2-auto-discovery/lab.txt

Prometheus on Kubernetes

Getting Kubernetes metrics