Ch04. Metric Exposition

앞에서 특정 Metric정보를 직접 계측하기 위한 방법을 알아 보았다. 이렇게 계측(생성)된 정보인 메트릭을 프로메테우스가 이용할 수 있도록 하는 것을 “Metric Exposition, 메트릭 노출(게시)”라고 한다.

프로메테우스에서 Metric Exposition은 HTTP를 사용하여 제공된다. 사용중인 개발언어에 따라 준비된 Client Library를 사용하여, 이러한 Metric Exposition을 하는 것이 권장되며, 사용하는 개발언어에 적합한 라이브러리가 없다면 수동으로 지정하거나 Custom이 가능하지만, 뭔 당연히….

- 프로메테우스에서 지원하는 공식 Client Library: Go, Python, Java, Ruby, Unofficial Client LIbrary: Bash, C, C++, Common Lisp, Dart, Elixir, Erlang, Haskell, .NET / C#, Node.js, Perl, PHP, R, Rust …

(Global Chacteristic) Metric Exposition은 해당 Application의 최상위에서 구현되며, 당연히 한 번안 구성하면 된다. 따라서, 계측의 감시가 있는 세부 코드에서도 주의하여야 하며, 해당 Application 전체에서 적용됨에 주의 한다. 또한, 당연히 해당 Application에 대한 Metirc의 정의가 실행되는 순간 필요한 Registry도 함께 Regist되는 점에도 주의 한다.

몇 몇 Client Library 환경에서 나름대로 계측(Instrument) 내용을 Metric화 하고, 이것을 HTTP방식으로 Metric Exposition하는 방법에 대한 소개를 살펴본다. 당연히 연계되어 Library dependency와 Develop Environment는 준비되어 있는 것으로 간주 한다.

4.1. Python

Python환경에서 Instrument를 진행하고 Metric Exposition을 위한 코드를 작성하기 위해, Prometheus Python Client Library를 사용하는 예제를 이미 Chap03에서 진행해 보았다.

Client Library의 `start_http_server`를 사용하여 , 이 함수가 동작하는 서버 OS에서 프로메테우스에 Metric을 제공하는 HTTP서버를 백그라운드로 동작시키고 있다. 그러나, 보통의 경우, Metric을 제공하기 위한 다른 HTTP서버가 이미 있을 수 있다. Python기반에서 어떤 Framework을 사용하는냐에 따라 다양한 방법으로 Metric을 표현할 수 있다.

CASE: WSGI

WSGI(Web Server Gateway Interface)는 Web Application과 관련되어 Python 표준이다. Python으로 구현된 Cient는 WSGI 코드를 사용할 수 있는 WSGI Application을 지원한다.

```
$ more 4-1-wsgi.py
from prometheus_client import make_wsgi_app
from wsgiref.simple_server import make_server

metrics_app = make_wsgi_app()

def my_app(environ, start_fn):
    if environ['PATH_INFO'] == '/metrics':
        return metrics_app(environ, start_fn)
    start_fn('200 OK', [])
    return [b'Hello World']

if __name__ == '__main__':
    httpd = make_server('', 8000, my_app)
    httpd.serve_forever()
$
$ python3 4-1-wsgi.py
$ curl localhost:8000
$ curl localhost:8000/metric
```

- python3로 해당 코드를 서버로 구동하고, URI에 따른 테스트를 진행해 본다.
- 위의 Python 웹서버는 기본 웹 요청에 대한 처리에 대해 `my_app()`으로 구현 하였고, `my_app()`은 요청의 URI가 “/metrics”인 경우, `metric_app()`에 그 처리에 대해 또다시 위임한다.
- 그렇지 않은 경우는 `my_app()` 내부에서 원래 목적의 웹서비스에 필요한 처리를 하면 되는 구조이다.
- 이렇게 WSGI Application과 연결고리를 가지게 함으로서, 단순 WebService가 아닌 WebApplication과 연계되도록 만들 수 있으며, 웹미들웨어(Authentication, Session, Cookie 처리 등)같은 것들을 추가할 수 있다.

CASE: **Twisted Engine (TBD)**

CASE: **Gunicarn for Multi-Process (TBD)**

보통의 경우 Single-Process, Multi-Thread를 사용하지만, 더 나은 성능과 분산을 위해 Multi-Process를 사용하는 환경에서는 당연히, Metric들도 프로세스별로 생성,관리,조회되어야 한다. 그러나, 의미적으로 Multi-Process을 동일한 역할의 Worker개념으로 본다면, Metric들도 당연히 통합된 Metric으로 보여져야 한다. 이러한 요건에 대한 방법이다.

> 💡 **mmap for Multi-Process**
> (책에서는 Client Library에 한정적으로 표현했지만) 일반적으로 Multi-Processs는 성능을 위한 선택이고, 구조이므로, Multi-Process간의 Call-Overhead는 필연이다. 따라서, 프로세스간의 통신(요청, 접근, 지원)의 성능은 곧 전체의 성능이므로, 해당 프로세스간의 통신은 Process가 메모리에서 제공받는 성능 수준이어야 한다.

4.2. Go (TBD)

4.3. Java

CASE: HTTPServer

자바환경에서의 Instrument와 Metric Exposition을 위한 코드를 작성하려면, Prometheus Java Client Library를 사용하여야 한다. Python에서 `start_http_server`를 사용했던 것 처럼, Java Client Library의 Class를 사용하여 HTTP 서버를 구동하고 수행해 본다. (참고로 자바에서 Java Client Library는 Simpleclient로 불리워 진다.)

Simpleclient를 사용하는 현실적인 목적과 개념의 이해는 다음과 같다.

- 일반적인 Java로 만들어진 단독 Application이나, Daemon형태로 서비스를 제공하도록 만들어진 서버용 Application에 Simpleclient를 포함시키고, 이것으로 해당 Application이 프로메테우스 Metric을 Exposition할 수 있도록 하는 것이 목적 이다.

다음은 Prometheus Java Client Library의 HTTPServer Class를 사용하여, HTTP Server를 구동하고 사용하는 예제이다. 이미 알고 있듯이, Java환경에서의 Metric들은 static으로 정의되기 때문에 한 번만 등록되어도 된다.

- 만약 다수의 JVM과 Process환경에서 해당 Metric이 동작하게 하기 위해서는 DefaultExports.initialize()를 호출하여, 모든 Java Application에서 사용할 수 있으며, 해당 Method는 Thread-Safe임을 참고한다.

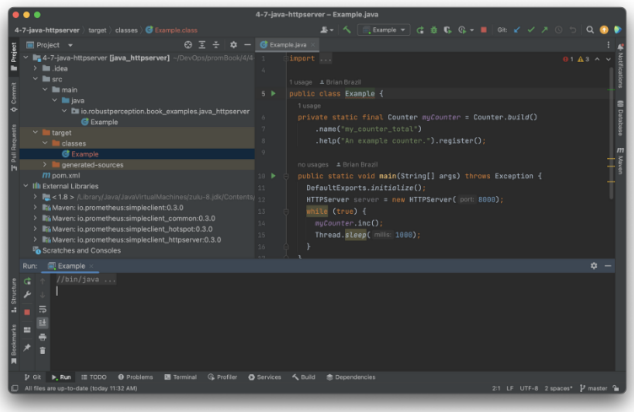
```
$ more
import io.prometheus.client.Counter;
```

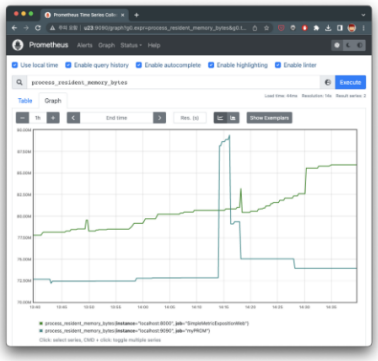
```
import io.prometheus.client.hotspot.DefaultExports;
import io.prometheus.client.exporter.HTTPServer;

public class Example {
  private static final Counter myCounter = Counter.build()
      .name("my_counter_total")
      .help("An example counter.").register();

  public static void main(String[] args) throws Exception {
    DefaultExports.initialize();
    HTTPServer server = new HTTPServer(8000);
    while (true) {
      myCounter.inc();
      Thread.sleep(1000);
    }
  }
}
$
```

- 참고: Ubuntu서버에서는 Java Code 관련된 내용(소스, 컴파일, 패키지들에 대한 것들)을 진행하는 것은 귀찮으므로, macBook에서 Code 및 dependency관리와 컴파일을 하고, Ubuntu를 Remote Deploy 대상으로 하여, 진행한다.
  - 참고: 참고, IntelliJ를 사용하면, 아래의 dependencies들도 다 알아서 준비된다. 또한, 해당 컴파일된 내용을 Ubuntu에서 실행하는 것도 IntelliJ에서 Ubuntu Server에 접근하면 알아서 진행해 준다.
- 위의 Simpleclient 코드와 관련된 dependnecy를 준비하기 위하여, Maven기반의 pom.xml에 포함하여야 하는 dependencies는 다음과 같다.

```
io.prometheus:simpleclient:0.3.0
io.prometheus:simpleclient_common:0.3.0
io.prometheus:simpleclient_hotspot:0.3.0
io.prometheus:simpleclient_httpserver:0.3.0
```

- 해당 소스를 실행 하여, 원격으로 지정된 Ubuntu 환경에서 구동되도록 하고, 해당 Ubuntu환경에서 확인해 보면, 해당 home directory에 로컬의 macBook에서 Deploy한 내용들이 위치하게 되고, 해당 Ubuntu의 JVM으로된 서버가 가동중이다.
- 해당 서버는 tcp 8000 port에서 LISTEN중인데, curl로 확인해 보면, 가동중인 HTTPServer의 JVM관련된 Metric들이 조회될 것이다.





CASE: Servlet (TBD)

- 예상하고 있는대로, Servlet기반의 Web Application Server(e.g. tomcat)의 경우에도 Java Client Library의 `MetricServlet` Class를 사용하여, 해당 Web Applicaiont Server의 중간에 끼워 넣는 개념(like as Jetty)이다. 물론,

Servlet 계층의 취급을 위해, javax.servlet.http 패키지의 `HttpServletRequest`, `HttpServletRequest`가 필요한 것은 당연하다.

4.4. Pushgateway

Batch Job은 보통 지속적으로 동작하는 방식이 아니므로, 프로메테우스에서는 이런 Batch Job을 정확하게 수집하기 위해, Pushgateway가 필요하다. 일반적으로 Pushgateway는 Prometheus실행시 함께 실행되며, Prometheus는 Pushgateway에 있는 Metric을 수집하는 방식이다.

- 예를 들어, 하나의 Batch Job의 결과를 Pushgateway에 Push하게 되는 경우, 그 데이터들은 Pushgateway내에서는 마지막 데이터만 Cache하게 된다.
- Pushgateway는 서비스 레벨에서의 Batch Job에 대한 Metric Cache(Cached Matric)으로 보아야 한다. Metric이 취급되고 적용되는 실제 범위는 하나의 Machine(OS) Instance 또는 Process Instance에 연계되거나 종속되거나 한정되는 개념은 아닌 것이다.
- 프로메테우스의 관심범위(의미적 역할범위) 내에서 Batch Job의 결과가 어떻게 되었는지에만 집중하여야 하기 때문이다. 그래서 프로메테우스에서 Batch Job은 “**Service-Level Batch Job”** (=like as global scope at code)으로 간주 된다.

아래의 내용으로 Pushgateway를 설치해 보고, Pushgateway에 Metric 하나를 Push해 보고, 프로메테우스에서 감시되는지 확인해 보자.

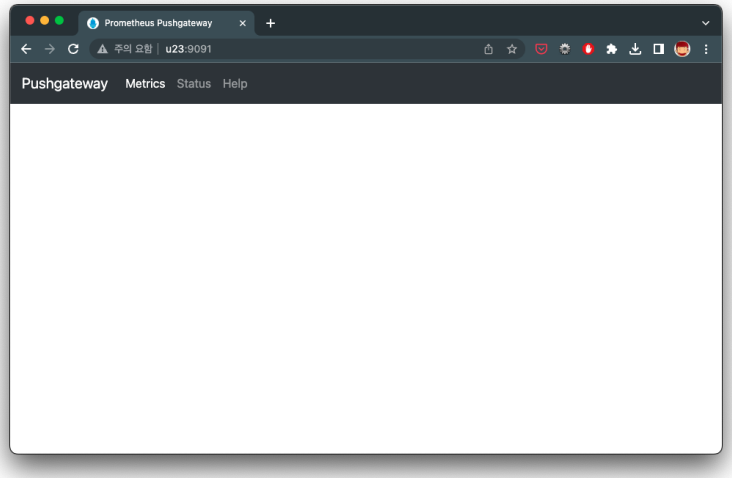
- (1) **Pushgateway Download, Install & Run**
  - 프로메테우스 다운로드 페이지에서 pushgateway를 다운로드 받아, Ubuntu에 설치한다. (현재 버젼은 1.5.1 / 2022-11-29 이다.)
  - Pushgateway는 Port 9091에서 동작하는 일종의 Exporter(like as Agency Exporter)이며, 아래 설정을 프로메테우스에 있는 prometheus.yml 의 scrape_config 섹션에 추가하여, 구성한다. Pushgateway에는 `honor_labels: true`로 설정을 하여, 특정 instance에 종속되지 않는 Pushgateway동작하도록 한다.

```
scrape_configs:
 - job_name: pushgateway
   honor_labels: true
   static_configs:
    - targets:
      - localhost:9091
```

  - 설정내용을 확인해 보고, 프로메테우스와 Pushgateway를 기동한다. Listen Port는 9091이다. 필요시 ufw, NAT 등을 사전에 정리해 준다. 마지막의 예처럼 확인도 해 본다.

```
$ cd /svc/prometheus
$ ./promtool check config prometheus.yml
$ ./prometheus
$ cd /svc/pushgateway
$ ./pushgateway
$ curl localhost:9091/metrics
```

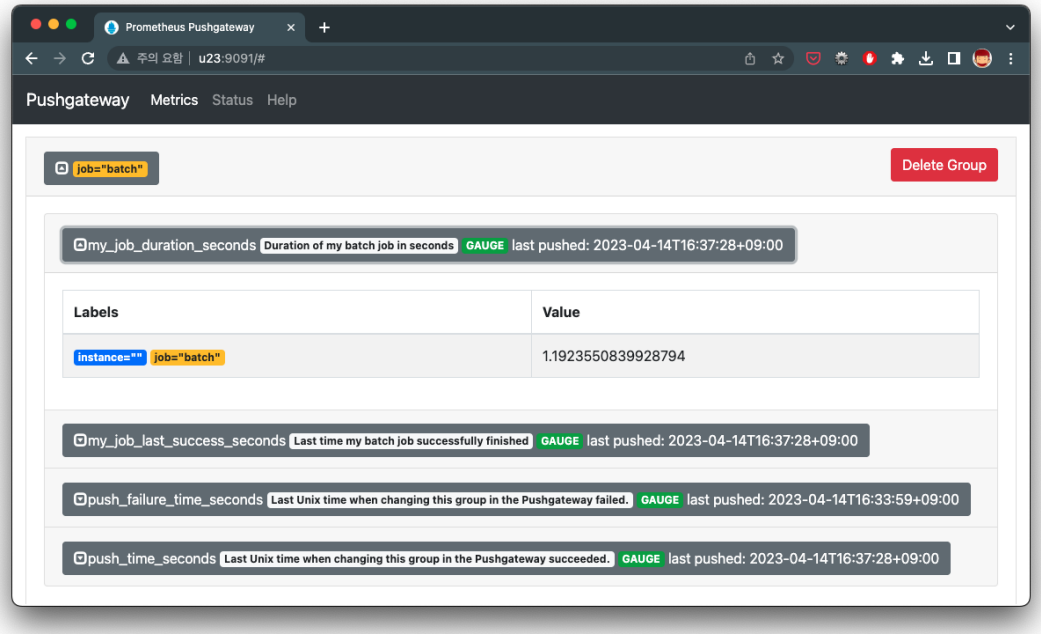
  - 이제 웹 브라우져로 `http://localhost:9091`에 접근하여, Pushgateway를 확인해 보자. 처음이라면, 다음과 같이 화면에 아무것도 없다. (localhost는 pushgateway가 실중인 host) - (좀 불친절하다. 0개라고 하던가. 처음에는 오류인줄 알았음)



- **(2) Push a Metric to Pushgateway**
  - Batch Job에 의해 생성된 Metric을 Pushgateway에 Push하는 코드는 다음과 같다. Pushgateway에 Metric을 Push하려면, Client Library를 사용한다. 아래의 예제 코드를 만들고 python으로 실행해 본다.

```
$ more 4-12-pushgateway.py
import time
import random
from prometheus_client import CollectorRegistry, Gauge, pushadd_to_gateway

registry = CollectorRegistry()
duration = Gauge('my_job_duration_seconds',
        'Duration of my batch job in seconds', registry=registry)

try:
    with duration.time():
        # Your code here.
        pass
        aSleepTime =  random.uniform(1,3)
        time.sleep( aSleepTime )
        print("aSleepTime: ", aSleepTime)

    # This only runs if there wasn't an exception.
    g = Gauge('my_job_last_success_seconds',
            'Last time my batch job successfully finished', registry=registry)
#    g.set_to_current_time()
finally:
    pushadd_to_gateway('localhost:9091', job='batch', registry=registry)
$ python3 4-12-pushgateway.py
aSleepTime:  2.956427826257024
$
```

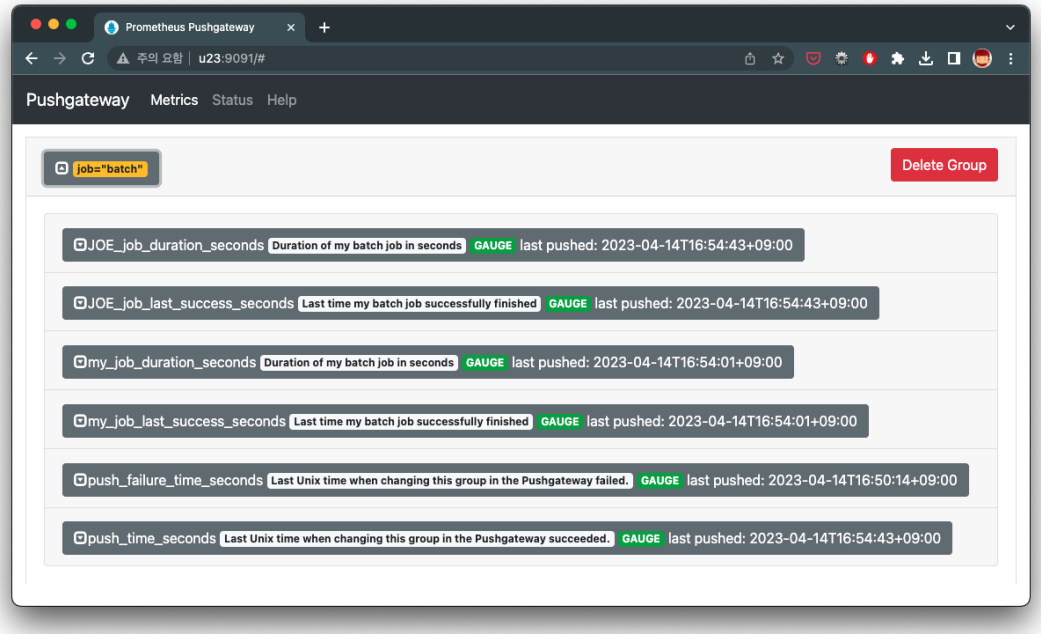
  - 다시 웹 브라우져로 pushgateway에 접근해 본다. 위의 예제가 Pushgateway에 기록한 Batch Job에 대한 기록이 남아 있음을 알 수 있다.



- 우리가 정의하여 생성한 Metric 2개가 있으며, (1) `my_job_duration`은 Batch Job이 실행이 지속된 시간(코드의 `Guage duration`)을 항상 Pushgateway에 반영하게 되며, (화면 처럼 Dropdown하여 보면, 실행에 소요된 시간이 기록되어 있음을 알 수 있다.) (2) `my_job_last_success_seconds`는 Batch Job이 성공한 시각(코드의 `Guage g`)을 Batch Job이 끝나고, 명시적으로 코드에 의해 Pushgateway에 Push한 시각이다.
- 또한, Pushgateway는 스스로 2개의 Metric을 추가로 생성하는데, (3) `push_failure_time_seconds`는 마지막으로 기록된 실패(실패가 없어도 첫 실패 기록은 첫 기록 시각으로)에 대한 시각이 반영되고, (4) `push_time_seconds`는 마지막으로 성공적으로 기록된 시각이 기록되어 있다.

◦ 추가 확인을 위해 앞의 코드와 유사하지만, Metric이름을 다르게 한 다음의 코드를 실행해 보고, Pushgateway의 내용을 살펴 본다.

```
$ more 4-12-pushgateway_more.py
import time
import random
from prometheus_client import CollectorRegistry, Gauge, pushadd_to_gateway

registry = CollectorRegistry()
duration = Gauge('JOE_job_duration_seconds',
        'Duration of my batch job in seconds', registry=registry)

try:
    with duration.time():
        # Your code here.
        pass
        aSleepTime =  random.uniform(1,3)
        time.sleep( aSleepTime )
        print("aSleepTime: ", aSleepTime)

    # This only runs if there wasn't an exception.
    g = Gauge('JOE_job_last_success_seconds',
            'Last time my batch job successfully finished', registry=registry)
#    g.set_to_current_time()
finally:
    pushadd_to_gateway('localhost:9091', job='batch', registry=registry)
$ python3 4-12-pushgateway_more.py
aSleepTime:  1.3637385927527448
$
```



    - 새로 Push한 기록이 추가 되어 있는 것을 확인 할 수 있으며, `push_time_seconds`에는 Pushgateway가 마지막으로 성공적으로 기록된 마지막 시각이 기록되어 있다.
- (Service-Level Batch Job) 위의 내용에서 `instance=""` 로 되어 있는 것이 있다. 앞서 설명한 대로 해당 Batch Job이 Instance(OS machine)에 속하지 않은 상위의 서비스 개념으로서 **"Service-Level Batch Job"**으로 취급하기 때문이다. 필요시 이 Label을 가지고 데이터를 취급할 수 있을 것이다.
- (Grouping key) 위의 내용에서 `job="batch"` 처럼 주어진 Label은 다른 Label을 추가할 수 있으며, Label을 사용하여 Job들을 그룹화하여 취급할 수 있다. 이때, `job=""`로 주어진 Label을 **"Grouping Key"**라고 한다.

4.5. Bridge

Go, Java, Python에는 Graphite Bridge가 있다. Bridge의 역할은 Client Library의 Registry로 부터 Metric을 전달받아, 프로메테우스 방식이 아닌, 다른 형태로 Metric을 출력 할 수 있다.

이러한 개념을 확장하여, 프로메테우스가 아닌 다른 계측 라이브러리(여기서는 다른 Platform이나 Framework)에게 전달할 수도 있다. 또한, 이 방법은 양방향을 지원하므로, 다른 계측 라이브러리가 프로메테우스쪽으로 데이터를 전달하는 것도 가능하다.

- Chapter 12: "익스포터 작성하기 - 사용자 정의 수집기"에서 상세 내용을 다룸

4.6. Parser & OpenMetrics

앞의 대부분의 예제들이 보여 주듯이, Metric결과에 접근하기 위해서는 Client Library를 사용하여, 필요한 Registry에 접근하였다. 그러나, Go, Python의 경우에는 Parser(Reference Implementation으로 구현된)를 제공하는데, 이런 방법으로 프로메테우스 메트릭에 접근하여 원하는 데이터를 얻어내고, 이것을 다른 모니터링 시스템이나 다른 App에 사용할 수 있도록 데이터를 변형(Parse)하거나 전달 할 수도 있다.

아래 Python코드를 살펴 보면, 프로메테우스 스타일의 Metric출력형 정보를 나름의 원하는 형식으로 바꾸어 출력하고 있다.

```
$ more 4-14-parse.py
from prometheus_client.parser import text_string_to_metric_families
```

```
for family in text_string_to_metric_families(u"counter_total 1.0\n"):
  for sample in family.samples:
    print("Name: {0} Labels: {1} Value: {2}".format(*sample))
$ python3 4-14-parse.py
Name: counter_total Labels: {} Value: 1.0
$
```

“OpenMetrics”라는 프로젝트의 목적은 프로메테우스의 Metric 표시 형식으로 Metric을 처리하고 표준화하는 것이다. 모니터링 시스템들은 유사한 방식과 유사한 정보들을 수집하여, 나름의 목적에 맞도록 모니터링을 하는데, 공통된 부분으로 진행할 수 있는 Metric에 대한 표준화는 많은 이점이 있을 듯…

- DataDog, InfluxDB, Sensu, Metricbeat(at ELK) 같은 모니터링 시스템은 텍스트 형식을 파싱할 수 있는 컴포넌트를 지원한다. 프로메테우스가 아니더라도 이러한 모니터링 시스템들은 Metric에 직접 접근하여 정보를 얻어 자신의 모니터링에 해당 정보를 사용할 수 있다.

4.7. Metric Exposition 사용

프로메테우스의 텍스트 기반 형태의 Metric Exposition 형식은 그 내용(Metric Context)를 Make하거나, 의미를 알아내어 처리하기 위해 Parsing의 구현과 취급이 쉽다.

준비되어 있는 Client Library를 사용하여, 프로메테우스 시스템을 구성하고 관리할 수 있지만, 우리가 직접 textfile을 Node Exporter로 연계하여 Metric을 구성해 내는 Textfile Collector처럼 사용할 수도 있다.

결국 Metrics 정보들도 Text형태의 문서이다. 가장 기본이 되는 형식은 예제는 다음과 같다. Metric이름과 그 뒤에는 64bit Float 형식의 값이 따라오는 형식이다.

```
my_counter_total 14
a_small_guage 8.3e-96
```

Metrics Type & Formation

앞의 예제 보다 조금더 완전한 형태의 Metric 출력은 다음과 같다. 프로메테우스의 주요 데이터 형식인 Counter, Guage, Summary, Histogram에 대한 예시이다.

```
# HELP example_guage An example guage
# TYPE example_guage guage
example_guage -0.7
# HELP my_counter_total An example my counter
# TYPE my_counter_total counter
my_counter_total 14
# HELP my_summary An example summary
# TYPE my_summary summary
my_summary_sum 0.6
my_summary_count 19
# HELP my_histogram_example An example for histogram
# TYPE my_histogram_example histogram
latency_seconds_bucket{le="0.1"} 7
latency_seconds_bucket{le="0.2"} 18
latency_seconds_bucket{le="0.4"} 24
latency_seconds_bucket{le="0.8"} 28
latency_seconds_bucket{le="+Inf"} 29
latency_seconds_sum 0.6
latency_seconds_count 29
```

- HELP는 수집할 때마다 변경되어서는 안된다.

- TYPE은 Guage, Counter, Summary, Histogram, untyped 중 하나 이다. untyped는 Metric TYPE이 지정되지 않은 경우 기본값으로 사용된다.
- HELP, TYPE은 프로메테우스 내부에서는 현실적으로 무시될 수 있지만, 나중에 그라파나 같은 도구에서 쿼리를 작성하는 등의 목적으로 사용 될 수 있다.
- Metric은 중복될 수 없으며, 시계열을 포함하여 그룹화 될 수 있도록 정리 되어야 한다.
- Histogram에서는 `_count`는 `+Inf`와 일치하여야 하며, Bucket도 변경되어서는 안된다. PromQL등에서 `historgram_quantile()` 함수등의 동작에 문제를 일으킨다.

Label

레이블은 여러개를 부여할 수 있으며, 가능한 순서를 지키도록 한다. 기본적인 형태는 다음과 같다. 다음 Chapter의 주제이다. `LabelName="LabelKeyValue",`

Escape Character Code

프로메테우스에서 사용되는 기본 문자 형식은 UTF-8로 인코딩되며, Escape Character는 \(backslash)를 사용한다.

- HELP statement: \n, \\
- LabelValue Statement: \n, \\, \"

TimeStamp

시계열이 필요한 곳에는 Timestamp로 명시하게 된다. 프로메테우스에서 Timestamp는 Unix Epoch 방식을 사용한다. 또한, 프로메테우스에서 Timestamp는 메트릭 값 다음에 위치하는 것이 보통이다.

Timestamp는 프로메테우스 내부적으로 시계열 구조에 자동으로 사용되거나, Federation 같은 특별한 구성에서만 사용되며, Label 또는 Grouping로 활용하기 위한 목적이나, 데이터 그 자체로서 직접적으로 사용되는 것은 권장되지 않는듯…하다.

> 💡 Federation at Prometheus
> 프로메테우스 서버에서 다른 프로메테우스 서버의 시계열 데이터를 스크랩하는 기법으로, 유연한 모니터링 Use Side의 확장에 의미가 있다. Hierachical Federation, Cross-service Federation의 모델이 있다.
> REF: https://prometheus.io/docs/prometheus/latest/federation/#federation

Checkup **Metric**

프로메테우스 2.0부터 사용자 정의 Parser가 도입되었다. 이런 저런 이유로 이제는 프로메테우스가 수집한 /metrics의 자료의 내용이 항상 유효하다고 볼 수 없다. 아래와 같이 `promtool`을 사용하여 메트릭 결과가 유효한지 검사를 해 볼 수 있다.

```
$ curl http://localhost:8000/metrics | promtool check-metrics
```

+==