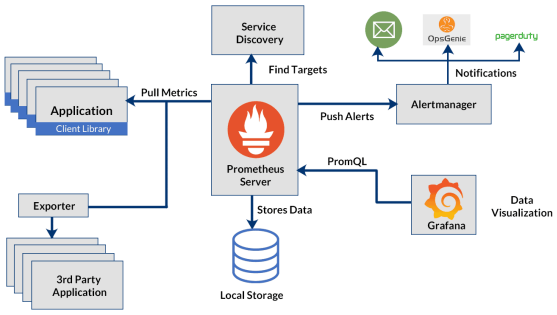
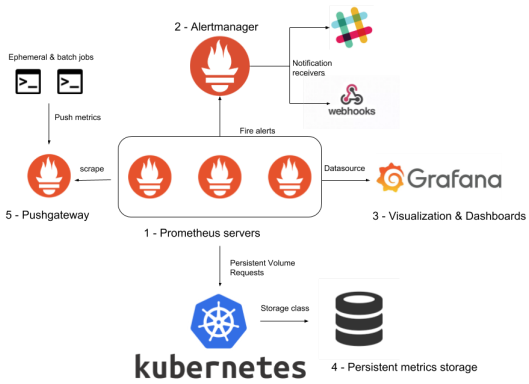
# Ch03. 계측/Instrumentation

### 3.0. Intro

- 프로메테우스의 계측은 감시 하고자 하는 시스템의 상태를 모니터링하고 분석할 수 있도록, 해당 시스템의 메트릭을 계측(측정,생성)하는 방법을 사용한다.
  - 감시하고자 하는 시스템에 대한 Exporter가 만들어져 있는 경우, Exporter를 대상으로 Pull Scrap을 하게됨
  - 보통의 경우 감시하고 싶은 우리가 만드는 Applicaation에 대해서는 해당 Application에 맞는 Client Library를 사용하여, 감시할 세부 Metric을 만들어야 한다. (직접계측, Direct Instrumentation)
  - 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
- “메트릭”을 “계측한다” 라는 것을 구현 관점에서 바꾸어 이야기 하면,
  - 특정 원하는 메트릭 정보를, 프로메테우스에서 취급하는 메트릭 형태와 방식(feat 프로메테우스 Client Library)으로 측정하여 얻어내고, 해당 계측한 특정 메트릭 정보를 해당 노드의 종합 메트릭 정보에 추가하는 것이다. 이렇게 프로메테우스에서 해당 노드의 메트릭 조회(Pull Scrap)시 해당 계측한 메트릭정보도 같이 포함되어 운용된다.
  - Instrumentation Code는 계측하고자 하는 Application의 Code에 함께 넣어야 한다. Metric Class Instance는 Instance를 사용하는 Code File에 같이 넣어야 한다. 그래야 에러추적과 개발관리에 유리하다.
  - 모니터링할 Application의 서비스 타입을 고려한다.
- 이러한 메트릭들은 프로메테우스에서 취급하는 **Counter, Gauge, Summary, Histogram**의 4가지 형태로 생성됩니다. 이러한 메트릭은 간단한 예제부터 복잡한 시스템까지 다양한 상황에서 사용될 수 있으며, 단위 테스팅과 같은 다양한 방법으로 계측 결과를 검증할 수 있습니다. 이러한 계측 결과는 시스템의 성능 문제를 감지하고 해결하는 데 매우 유용합니다. (사실 이것 밖에 안됨)
- Metric의 사용과 관련된 몇가지 권고 특징들
  - Counter와 Guage의 쉬운 선택 방법: 값이 내려갈 수 있다면, Guage이다. Counter는 보통 `rate()`를 사용하며, 단위시간당 증가 비율을 계산한다. Guage는 `rate()`로 계산하면 안 된다.
  - Timestamps, not Timesince: 어떤 일이 발생하고, 그 지난 시간을 추적하고 싶다면, 발행 이후의 시간(Timesince)을 측정하지 마라. 그냥 Unix Timestamp를 남기고, 두 개의  Timestamp의 차이를 사용해라.
  - Inner Loops: Inner Loops에 포함된 Metric, 특히 Lebel이 있는 경우는 그 리소스 비용에 부담이 될 수 있다. 가능하면, Inner Loops내의 Metric수를 제한하고, Label은 피해라. 차라리 Cache형태로 구성하고, 그 값을 읽어 Label 처리해라.
  - Avoid Missing Metric: 특정 이벤트가 발생하기 전까지 Metric이 없다면, 특히나 시계열에서는 그 처리가 어렵다. 존재해야될 Metric이라면, 기본값을 Export하도록 한다.
- Prometheus Architecture: Remind




- 실습환경 준비: 나는 개인적으로 macBook에 VM으로 Ubuntu 20.04 서버를 설치 했다. 그리고, Ubuntu에서 Python Application을 만들고, 이에 대한 프로메테우스를 위한 App Instrumentaion을 만들기 위한 준비를 아래와 같이 진행 했다.
  - Ubuntu 기준 Python, pip 설치

```
$ sudo apt install python3
$ sudo apt install python3-pip
$ python3 --version
Python 3.10.6
$ pip --version
pip 22.0.2 from /usr/lib/python3/dist-packages/pip (python 3.10)
$
```

  - pip로 python을 위한 prometheus-clinet설치

```
$ pip install prometheus-client
Defaulting to user installation because normal site-packages is not writeable
Collecting prometheus-client
  Downloading prometheus_client-0.16.0-py3-none-any.whl (122 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 122.5/122.5 KB 3.0 MB/s eta 0:00:00
Installing collected packages: prometheus-client
Successfully installed prometheus-client-0.16.0
$
```

  - 도서의 예제 다운로드

```
$ git clone http://github.com/prometheus-up-and-running/examples
```

### 3.1. 간단한 예제

python으로 간단한 웹서버를 만들어서 해당 웹서버를 프로메테우스로 감시해보는 예제이다. 당연히, python 서버용 코드에 프로메테우스를 위한 metric 관련된 라이브러리가 포함되어야 한다. 아래는 python 웹서버 코드이며, python으로 기동해 보면 된다.

```
$ more 3-1-example.py
import http.server
from prometheus_client import start_http_server

class MyHandler(http.server.BaseHTTPRequestHandler):
```

```
    def do_GET(self):
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Hello World")

if __name__ == "__main__":
    start_http_server(8000)
#    server = http.server.HTTPServer(('localhost', 8001), MyHandler)
    server = http.server.HTTPServer(('0.0.0.0', 8001), MyHandler)
    server.serve_forever()
$ python3 3-1-example.py
```

- 위의 Code는 8000에서 시작된 서버는 프로메테우스 Metric을 제공하고 있고, 8001에서는 다른 Handler에 의해 원래 웹서비스를 제공할 code block을 유지하는 경우를 모델링 한 것이다.
- `http://localhost:8000` 또는 `http://localhost:8000/metrics` 로 접근해 보면 해당 웹서버의 프로메테우스 메트릭을 확인 할 수 있고, `http://localhost:8001` 로 접근하면, 일반적인 웹서버의 response를 받을 수 있다.

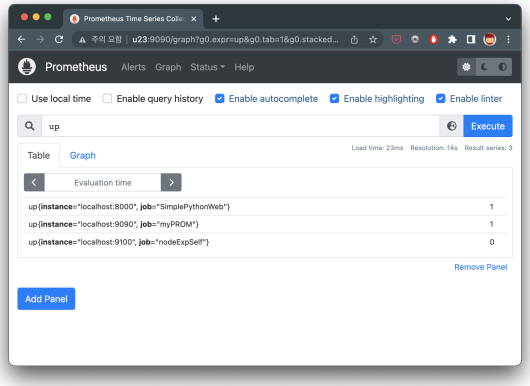
해당 간단한 웹서버를 프로메테우스가 감시하도록 아래 내용을 참고하여, prometheus.yml 파일에 job_name 관련 내용을 추가하여 설정을 추가하고, 프로메테우스를 기동하여 확인해 본다.

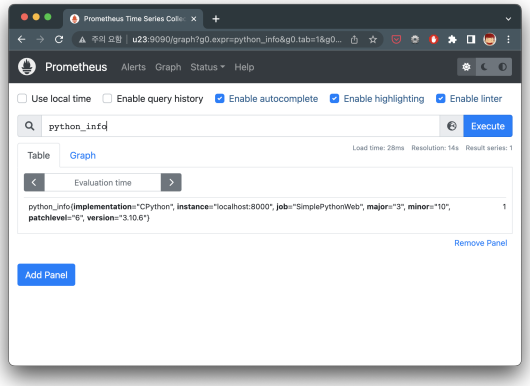
```
scrape_configs:
  - job_name: SimplePythonWeb
    static_configs:
      - targets:
        - localhost:8000
```

prometheus.yml 파일이 정상인지 확인할 필요가 있다면, promtool을 아래와 같이 사용한다. 설정파일에 문제가 있는지 미리 상세한 내용을 확인 할 수 있다.

```
$ ./promtool check config prometheus.yml
```

아래와 같이 python_info를 수식 브라우져에서 조회해 본다.



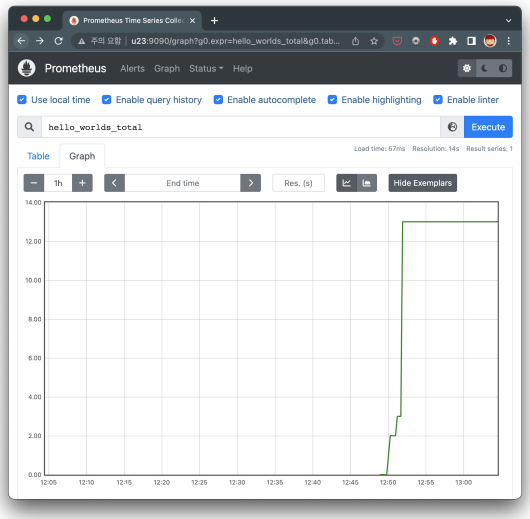
### 3.2. Counter

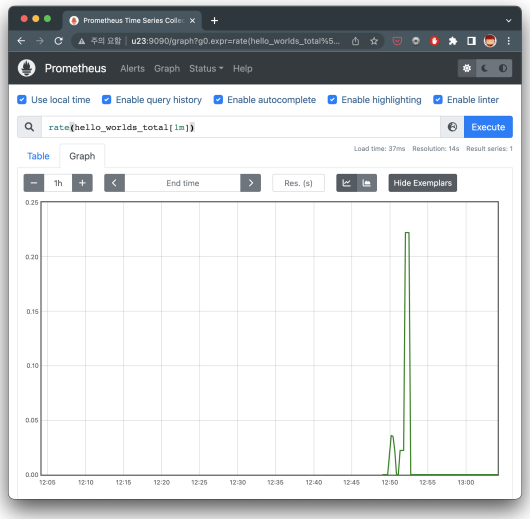
앞의 python으로된 간단한 웹서버에 사용자가 수집할 새로운 항목을 위한 새로운 메트릭을 추가해 본다. 사용할 Client Library 모듈과 Counter Type의 REQUEST라는 메트릭을 정의하고, 해당 메트릭이 계측되도록 구성한 것이다.

```
$ more 3-3-example.py
import http.server
from prometheus_client import start_http_server
from prometheus_client import Counter

REQUESTS = Counter('hello_worlds_total',
        'Hello Worlds requested.')

class MyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        REQUESTS.inc()
        self.send_response(200)
        self.end_headers()
        self.wfile.write(b"Hello World")

if __name__ == "__main__":
    start_http_server(8000)
    server = http.server.HTTPServer(('localhost', 8001), MyHandler)
    server.serve_forever()
$
```

- Counter 정의: REQUESTS라는 Counter 메트릭을 정의 하였고, 해당 부분의 설명은 메트릭에 포함되어 보여질 설명 내용이다.
- Instrument(계측) 구성: REQUESTS.inc()로 해당 Counter 메트릭을 계측하고록 하였다. 서비스를 제공하는 code block인 “`MyHandler()`”에서 계측을 수행하게 되고, Global로 정의된 프로메테우스 Metric에 추가한 `REQUESTS`에 그 값을 저장하고 노출하게 된다.

이제 해당 python으로된 간단한 웹서버를 다시 기동하고, 프로메테우스에서 확인해 본다. Expression Browser에서 `hello_worlds_total`, `rate(hello_worlds_total[1m])` 으로 조회해 본다. 아래는 두가지 경우에 대한 조회 결과 이다.




- (TBD) 카운트의 예외처리
- (TBD) 크기에 의한 카운트 처리

### 3.3. Guage

Guage는 현재 상태에 대한 스냅샷 개념의 측정이다. Counter가 얼마나 빨리 증가(변화)하는에 관심을 두지만, Guage는 실제 해당 값에 관심을 둔다.

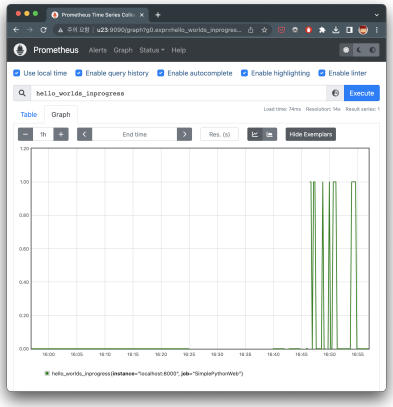
- e.g. Queue의 항목 갯수, Cache사용량, Active Thread수, 마지막 처리 시각, 마지막 1분간의 초당 평균 요청 갯수 (이 경우는 Counter를 PromQL의 rate()를 사용하여, Guage로 전환한 것이다.)

Guage에서 사용 가능한 Method는 Counter와 마찬가지로, inc, dec, set이 있다. Counter의 경우 그 값이 감소되는 경우가 권장되지 않지만, Guage는 상관 없다. 참고로, `@INPROGRESS.track_inprogress()` 코드는 해당 요청을 처리하는 코드블럭의 앞과 뒤에서 `INPROGRESS.inc()`, `INPROGRESS.dec()` 를 수행한 것과 동일한 내용이다. 즉, 아래의 코드의 실행시작과 끝 사이에서만 INPROGRESS임을 누적개념으로 표현한다.

```
$ more 3-7-example.py
import http.server
import time
import random
from prometheus_client import start_http_server
from prometheus_client import Gauge

INPROGRESS = Gauge('hello_worlds_inprogress',
        'Number of Hello Worlds in progress.')
LAST = Gauge('hello_world_last_time_seconds',
        'The last time a Hello World was served.')

class MyHandler(http.server.BaseHTTPRequestHandler):
    @INPROGRESS.track_inprogress()
    def do_GET(self):
        self.send_response(200)
        self.end_headers()
        time.sleep( random.uniform(1,10) )
        self.wfile.write(b"Hello World - 3.7ex")
        LAST.set_to_current_time()

if __name__ == "__main__":
    start_http_server(8000)
    server = http.server.HTTPServer(('localhost', 8001), MyHandler)
    server.serve_forever()
$
```

- 빠르게 진행 되므로 일부러 timer와 random을 사용하였다. 측정할 Metric이 inprogress()이므로 기록된 값은 해당 서버가 작동중인 상태에 대해서 1.0으로 표현된다.

```
$ curl localhost:8001 & curl localhost:8001 &
$ curl localhost:8000/metrics
...
hello_worlds_inprogress 1.0
...
$
```



Prometheus Naming Convention
Guage에는 Suffix(접미어)가 없다. Counter는 모두 `_total` 접미사로 끝난다. 또 다른 접미어로 `_count`, `_sum`, `_bucket` 등 이 있다. 또한 Metric Name 끝에는 단위를 명시하여야 한다. 내가 만든 App이 처리한 byte에 대한 처리량 Counter는 다음과 같은 패턴으로 표현될 것이다. e.g. `myapp_requests_processed_bytes_total`

- (TBD) Callback: Inner Guage handle

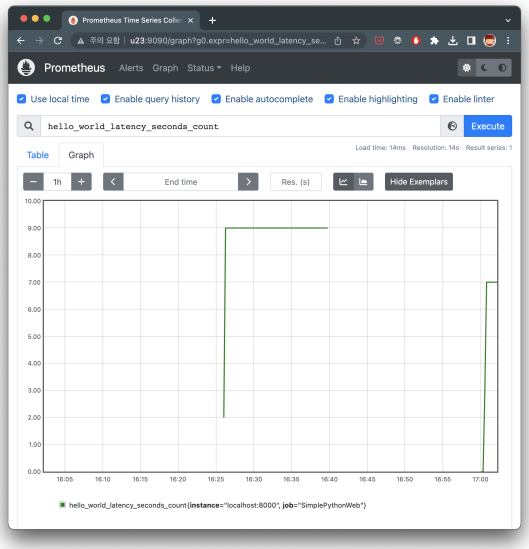
### 3.4. Summary

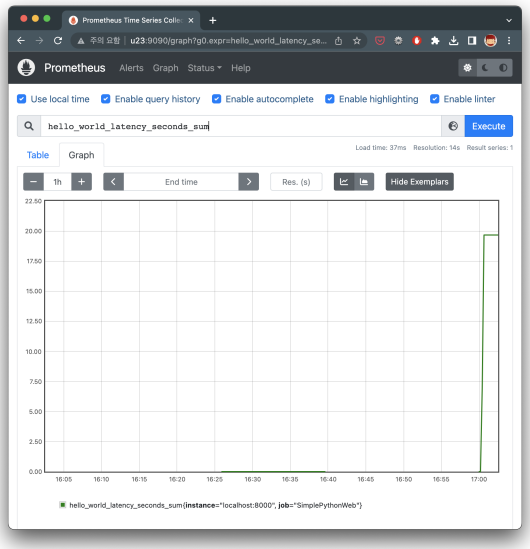
단일 분위수(quantile) 형태의 계측에 사용된다. Summary Type의 주요 메소드는 observe()이며, observe()는 해당 이벤트들의 모음에 대한 정보를 표현한다. (모음 개념으로 측정한 데이터는 다음과 같은 경우가 그 예이다.)

- e.g. 마지막 1분동안 각각 2초, 4초, 9초가 걸리는 3개의 요청이 처리되었다면, 전체 Count는 3, 전체 Sum은 15(초)가 될 것이다. 이 경우, 평균 평균 대기 시간은 5초가 될 것이다.

아래의 예제를 실행해 본다. observe()는 주어진 이름인 `hello_world_latency_seconds` 에 대해, 기본적으로 생성 시점에 대한 timestamp인 `hello_world_latency_seconds_created`를 기준으로 `hello_world_latency_seconds_count`와 `hello_world_latency_seconds_sum`이 LATENCY.observe()에 의해 기록된다.

```
$ more 3-9-example.py
import http.server
import time
import random
from prometheus_client import start_http_server
from prometheus_client import Summary

LATENCY = Summary('hello_world_latency_seconds',
        'Time for a request Hello World.')

class MyHandler(http.server.BaseHTTPRequestHandler):
    def do_GET(self):
        start = time.time()
        self.send_response(200)
        self.end_headers()
        time.sleep( random.uniform(1,5) )
        self.wfile.write(b"Hello World-3.9ex")
        LATENCY.observe(time.time() - start)

if __name__ == "__main__":
    start_http_server(8000)
    server = http.server.HTTPServer(('localhost', 8001), MyHandler)
    server.serve_forever()
$
```




- 실행은 모두 7건, 소요시간의 합은 20초로 보여진다. 해당 예제의 경우, 계산상의 평균 소요(대기)시간은 $20/7 \approx 3$ 이다.

## 3.5. Histogram

Summary Metric Type으로 평균 대기시간을 제공했다면, 분위수(quantile)을 제공하려면 어떻게 해야 할까? 예로서, 특정 이벤트(감시, 측정대상)이 300ms, 0.95분위수라면, 이것은 요청의 95%는 300ms미만의 시간이 걸린다는 의미로 해석할 수 있다.

Histogram Metric의 계측을 위한 Method는 Summary와 같은 observe()를 사용하며, 시계열로 구별된 카운터의 집합인 Bucket을 만들어 낸다. 예를 들어 Histogram에는 1ms, 10ms, 25ms 같은 Bucket의 집합이 있는 것이다.

Bucket은 보통 1ms~10s 범위를 다룬다. 일반적인 웹 애플리케이션의 범주이다. 필요하다면, 아래 처럼, Override하여, 자체적인 Bucket을 만들어 사용할 수도 있다.

```
LATENCY = Histogram('hello_world_latency_seconds',
          'Time for a request Hello World.',
          bucket=[0.0001, 0.0002, 0.0005, 0.001, 0.01, 0.1, 1])
```

아래에 예제를 참고하여 실행해 본다. 참고로 `2**random.randint(1,10) / 2**10 * 5` 에 의해 0~5사이에 exponential 분포의 난수를 구해 보았다. 분위수는 보통 10개 정도의 Bucket으로만 해도 보통의 경우 충분하다. 다만 변별력이 있도록 잘 배열을 분포 시키는게 중요하겠다.

```
$ more 3-11-example.py
import http.server
import time
import random
from prometheus_client import start_http_server
from prometheus_client import Histogram

LATENCY = Histogram('hello_world_latency_seconds',
        'Time for a request Hello World.')

class MyHandler(http.server.BaseHTTPRequestHandler):
    @LATENCY.time()
    def do_GET(self):
        self.send_response(200)
        self.end_headers()
```

```
        XX =  2**random.randint(1,10) / 2**10 * 5
        time.sleep( XX )
        self.wfile.write(b"Hello World-3.11:%f" % XX)

if __name__ == "__main__":
    start_http_server(8000)
    server = http.server.HTTPServer(('localhost', 8001), MyHandler)
    server.serve_forever()
$
```

- 위의 예제 화면들을 확인해 본다. 이후, 해당 metric을 조회해 보자. Summary와 다르게 Bucket Metric이 수집되고 있으며, “+Inf”라는 Metric도 추가되어 있으며, 집계된 수치가 누적형으로 되어 있음에 주의 한다.
- 이것은 Bucket이 제거되거나 줄어들어도, 전체적인 분위수는 문제가 없도록 하기 위함이다. 또한 분위수는 보통 10개 정도의 Bucket으로만 해도 보통의 경우 충분하다. 다만 변별력이 있도록 잘 배열을 분포 시키는게 중요 하겠다.
- 분위수에 분포하는 것은 디버깅이나, 재계산을 위해서는 적합하지 않으나, 직관적이고, End-User측면의 서술하는데는 적합한 점에 주의한다.

```
$ curl localhost:8000
...
# HELP hello_world_latency_seconds Time for a request Hello World.
# TYPE hello_world_latency_seconds histogram
hello_world_latency_seconds_bucket{le="0.005"} 0.0
hello_world_latency_seconds_bucket{le="0.01"} 0.0
hello_world_latency_seconds_bucket{le="0.025"} 4.0
hello_world_latency_seconds_bucket{le="0.05"} 6.0
hello_world_latency_seconds_bucket{le="0.075"} 6.0
hello_world_latency_seconds_bucket{le="0.1"} 8.0
hello_world_latency_seconds_bucket{le="0.25"} 9.0
hello_world_latency_seconds_bucket{le="0.5"} 9.0
hello_world_latency_seconds_bucket{le="0.75"} 10.0
hello_world_latency_seconds_bucket{le="1.0"} 10.0
hello_world_latency_seconds_bucket{le="2.5"} 12.0
hello_world_latency_seconds_bucket{le="5.0"} 14.0
hello_world_latency_seconds_bucket{le="7.5"} 16.0
hello_world_latency_seconds_bucket{le="10.0"} 16.0
hello_world_latency_seconds_bucket{le="+Inf"} 16.0
hello_world_latency_seconds_count 16.0
hello_world_latency_seconds_sum 18.628900371997588
...
$
```

### 3.6. Unit Testing 계측

코드에 대한 단위 테스트를 위해, Prometheus가 사용될 수 있다. 코드의 주요한 부분에 대한 Metric을 측정하고, 테스트의 결과로 해당 코드에 대한 평가에 사용해 볼 수 있다.

그러나, 코드에 대한 완벽성을 검증하는 테스트라면, 모든 발생 가능한 케이스에 대해 시도하고, 해당 로그를 찾아 명확하게 동작되었는지 확인하는 것이 우선되는 원칙이라는 사실은 변함이 없다.

### 3.7. 계측의 사용 방법

(서비스에 대한 계측의 적용) 서비스에 대한 계측은 보통 3가지 Type의 형상이 있다. (feat. Daemon vs Job atOS)

- (1) Online-Serving System: 사람이나, 다른 시스템에 대한 서비스를 제공하는 패턴의 시스템에 대한 계측이 있다.
    - 보통 RED(Request Rate, Error, Duration) 메소드가 주요한 계측 요소이다. 주요 Metric은 수행한 쿼리수, 오류 횟수, 지연시간, 현재진행중인 수 등이다.

    - 대부분의 HTTP요청, DB요청은 이 카테고리로 분류된다. 요청이 완료된 마지막 시점에 처리를 하는게 유리하다. (Dangle Zombi ??)
- (2) Offline-Serving System: 사용자나, 다른 시스템과 관계 없이 운용되는 시스템에 대한 계측이 있다.
    - 보통 USE(Utilization, Saturation, Error) 메소드가 주요한 계측 요소이다. Utilization: 서비스제공을 위해 현재 자원이 얼마나 잘 사용되고 있는가, Saturation: 서비스 제공에서 처리 대기가 얼마나 있는가의 정보이다.
    - 처리 단계별, 대기수, 처리수, 마지막 시간을 남기고 추적하게 할 수도 있지만, 차라리, Hearbeat개념으로 Timestamp를 포함하여, 그 시점의 상태에 대해 남기는 방법도 있다.
- (3) Batch System: 앞에서 설명한 Online-Serving과 Offline-Serving은 지속적으로 수행되는 작업환경에 대한 것이지만, Batch는 정기적으로 필요한 시점에만 동작하는 차이가 있다.
    - 항상 실행되는 것이 아니므로, 계측에 주의가 필요하다. **보통 Pushgateway같은 기법들이 사용되며**, 실행에 걸린 시간, 마지막으로 성공한 시각, 사용한 자원정보들이 주요 계측 요소일 것이다.
    - 그러나, 해당 Job이 오래 걸리는 것들이라면, Pull Scrap방식도 괜찮다. Job의 진행중의 상태를 알수 있도록 하여, 추적과 디버깅에 유리해 질 것이다.
    - 15분 이하 간격으로 자주 실행하는 Batch Job은 Daemon형태로 변환하고 Offline-Serving 방식으로 하는 것이 좋을 것이다.

(Subsystem에 대한 계측의 적용) 서비스 보다 작은 단위의 서비스로 해당 서비스를 구성하는 라이브러리(또는 Sub-system)를 계측, 모니터링 하는 것을 생각해 볼 수 있다.

- Library in Application
    - Code내에서 Library를 썼다면, 해당 사용한 코드에 대해 계측할 수 있어야 한다.
    - 해당 LIbrary가 프로세스 외부의 자원에 의존하여, 영향을 받는 경우라면, 최소한의 지연시간을 가지고 유용하고, 필수적인 요소만 추적해라
    - Library는 Application의 다른 곳에서 다르게 사용할 수 있으므로, Label을 사용해라
- Logging in Application
    - 로그를 남기는 Code 부분에서 적절하게 Logging 내용 자체가 아닌 특정 경우에 대한 Logging 자체(로그처리수 등)에 대한 Counter 정보 등을 남기면, 그 내용만으로도 직관적인 계측이 될 수 있다. (e.g.특정 경우의 처리량, 단위 시간당 처리량)
- Failure in Application
    - Logging과 비슷하게, 실패에 대해 Counter 정보 등을 총 시도횟수 등과 같이 남기면, 그 내용만으로도 직관적인 계측이 될 수 있다. (e.g. 실패율, 단위시간당 실패율)
- Threadpools in Application
    - 어떤 방식의 Threadpool을 사용하건, 결국 Pool(Queue, Buffer, Map도 마찬가지)이다. 이 경우의 핵심 Metric은 Pool에 대기중인 요청수, 사용중인 Thread 수, 전체 Thread수, 처리한 Task수, Task별 처리시간, Pool에 대기한 시간 등이다.
- Collectors in Application
    - 중요한 Custom Collector를 구현한 Code라면, 수집하는데 걸린 시간과 발생한 에러수 등에 대해 각각 Guage형식의 Metric을 구성하고, Exportgk하는 것이 권장된다.

(적정한 계측의 빈도와 적정한 Metric규모에 대한 고려)

(메트릭 명명 규칙)

- Character Set Rule, Snake Case, Suffix, Unit Scale, Naming & Naming Space

- Metric에 해당 Application의 이름을 사용하면 안된다. 이런 경우는 Label을 사용한다.

+==