# Ch10. General Exporter

앞서 대표적인 Exporter로 Node_Exporter를 살펴 보았었다. 그외 다른 Exporter들도 간단히 살펴본다.

Exporter는 보통 아무런 구성이나 설정없이 '작업'만 할 뿐이다. 그러나, 경우에 따라서는 최소한의 설정이 Exporter도 있다. (e.g. 대상 Application을 지정하려는 경우 등) 또는 이것 저것 다 '작업'하는 광범위한 Exporter도 있다.

Application Instance의 경우, 일반적으로 Application Instance마다 하나의 Exporter를 갖는다. (1 instance, 1 exporter) 이러한 Application을 프로메테우스와 연동되기 위한 방법(모델)은 몇 가지 형태를 생각해 볼 수 있다.

- (A) Application자체가 직접 계측(Instrumentation)한 다음, 이것을 프로메테우스가 검색하고, 수집하는 방법. (현실적으로 Custom된 Application에 대해서는 이 방법이 선호될 수 있을 것이다.)
- (B) Application이 직접 처리하지 않는 경우, Exporter가 사용되며, Application Instance와 함께 가능한한 일반화된 형태의 Exporter가 실행되는 방법
- (C) 여러개의 Application Instance에서 정보를 수집하는 Exporter로 구성하는 방법 (이 경우 앞서 설명된 것 처럼 `metrics_relabel_configs`에 의해 필요없는 label들은 drop되어야 한다. e.g. Pushgateway)

> 💡 현재 기준 Prometheus에서 제공하는 Exporter(Officail) 이외에 prometeus.io에서 알고 있는 third-party Exporter들을 포함한 모든 Exporter들은 아래에서 확인 가능하다.
>
> - Exporter and Integrations at prometheus.io

## 10.1. Consul Exporter (TBD)

## 10.2. HAProxy Exporter (TBD)

**(added) What is HAProxy?**


HAProxy는 기존의 HW기반의 Loadbanacer Switch를 대체하는 SW기반의 Loacbalancer이다. HW LB가 제공하는 L4, L7의 기능을 제공한다. HAProxy는 유사 SW들중에서 HA(High-Avaiability)를 쉽고 빠르게 제공하므로 인기가 있다.

LB의 기본적인 구성과 동작 형태는 VIP로 접근하는 접속에 대해 NAT를 기반으로 Reverse Proxy형태로 뒤쪽의 실제 서버들에게 접근을 Tunelling하며, Return되는 경로에 대해서는 다시 Proxy하거나, DSR(Dynamic Source Routing)에 의해 실제 요청을 진행한 Client로 돌아간다. (이것 때문에 Client측에서 IP handling에 조심…)

이런 방식으로 HAProxy는 VIP를 기준으로 서버를 Scale-out할 수 있으며, HAProxy자체도 HA를 위해 VRRP기반으로 해당 VIP를 사용하는 Active-Standby HAProxy를 구성하여 운영 할 수 있다.

참고로 HAProxy가 Loadbalancing하는 방식은 다음과 같은 것들이 있다.

- RR, Static-RR, LeastConn-RR, Source-Weight-RR
- URI, URL_Pattern, HDR, RDP-Cookie

책의 소개 내용은 간략히 설명하면 다음과 같다.

HAProxy가 구성되어 있는 서버에 서버 자체를 위한 Node Exporter와 HAProxy를 위한  HAProxy Exporter를 설치한다. 관련된 설정의 주요 내용을 요약하면 다음과 같다.

- Node Exporter는 :9100을 Metric 조회로 사용하지만 이것을 HAProxy Exporter의 :1234로 Proxy구성함
- HAProxy Exporter는 :1235에서 CSV문서 형식으로 HAProxy가 처리하고 운영된 내용에 대한 감시 내역을 확인 할 수 있다.
- HAProxy Exporter자체에 대한 Metric은 :9101/metrics에서 확인할 수 있다.

## 10.3. Grok Exporter

모든 Application(Framework, Platform-Instance포함) 들이 프로메테우스를 위한 Metrics를 제공한다면 좋겠지만, 그렇지 못하다. Application들은 보통 자신들만의 Log를 생성하는데, Grok Exporter는 이 로그를 매트릭으로 변환하여 사용할 때 사용한다. (눈치 빠르게, 저장된 Log 또는 현재 진행중인 Log등 어떤 것일지, Log의 이름이 바뀌는 경우등에 대비는 어떨지 궁금하긴 하다…)

Grok Exporter는 Log의 비정형적인 내용물을 취급하기 위하여, Logstach(feat. ELK)에서 사용하는 Log Parser방법을 사용하여, 이 방법으로 Log에서 패턴을 추출하여 처리한다. 불행히도, Grok은 Prometheus의 official이 아니므로 설치하려면, github에서 직접 다운로드 받아 설치하고 사용하여야 한다.


- https://github.com/fstab/grok_exporter (https://github.com/fstab/grok_exporter)

참고로 grok으로 ngnix의 Log를 모니터링 하는 방법을 소개한 괜찮은 링크가 있다. 여기를 참고하는게 좋을 듯.

- github-gurumee92's grok Exporter자료: https://github.com/gurumee92/getting-started-prometheus/blob/master/docs/part2/05_service_metric_monitoring_02/README.md

이렇게 grok Exporter는 Log파일의 내용을 자신이 원하는 Metric으로 만들기 위해 grok처리를 모두 정의해 주어야 하기 때문에 거의 Direct Instrumentation에 가깝다. 그렇기 때문에 grok Exporter는 다른 Exporter보다, 설정에 많은 노력이 필요하고 설정과 밀접한 관련이 있다.

Test로 확인해 보기위해, 우선은 grok exporter를 github에서 다운로드 받아 설치한다. (Release는 2020년 9월이 마지막이다.)

```
$ wget https://github.com/fstab/grok_exporter/releases/download/v1.0.0.RC5/grok_exporter-1.0.0.RC5.linux-amd64.zip
$ unzip grok_exporter-1.0.0.RC5.linux-amd64.zip
$ ln -s grok_exporter-1.0.0.RC5.linux-amd64/ grok
$ cd grok
```

grok할 대상 Log가 다음과 같이 예제로 있다고 가정하고, 이 Log를 grok처리하는 grok Exporter를 사용해 본다.

```
$ more example.log
GET /foo 1.23
GET /bar 3.2
POST /foo 4.6
$
```

다음으로 grok이 사용할 설정파일을 준비한다. 설정 내용은 간단한 로그파일을 파싱해서 Metric을 생성하는 예제이다.

```
$ more 10-6-grok.yml
global:
  config_version: 3
input:
  type: file
```

```
  path: example.log
  readall: true
grok_patterns:
- METHOD [A-Z]+
- PATH [^ ]+
- NUMBER [0-9.]+
metrics:
- type: counter
  name: log_http_requests_total
  help: HTTP requests
  match: '%{METHOD} %{PATH:path} %{NUMBER:latency}'
  labels:
    path: '{{.path}}'
- type: histogram
  name: log_http_request_latency_seconds_total
  help: HTTP request latency
  match: '%{METHOD} %{PATH:path} %{NUMBER:latency}'
  value: '{{.latency}}'
server:
  protocol: http
  port: 9144
$
```

- 책에서는 config_version 2로 되어 있는데, 최근에는 config_version 3이다. 다음 명령을 참고하면, version 2를 3으로 확인 할 수 있다: `./grok_exporter -showconfig -config 10-6-grok.yml`
- grok Exporter가 취급해야할 Log File에 대한 정의에서 `readall:` 를 `true`로 설정한것은 해당 Log File 전체를 읽기 위한 설정이고, 실제 운영환경에서 Log File의 마지막 Tail 처리를 위해서는 `false`로 설정 하여야 한다.
- grok Exporter 에서 사용하는 패턴 표현식은 '정규표현식'으로 해도 되고, grok 방식으로 해도 된다. 예제에서는 `additional_patterns:`를 정규표현식으로 정의하였고, Metric으로 사용할 `log_http_requests_total`, `log_http_request_latency_seconds_total`은 grok 표현방식으로 정의 하였다.
- 참고로 grok을 설치한 위치의 `~/patterns`를 보면, grok으로 catch 하여야 할 기본적인 pattern들의 예제가 다양하게 있다. 또한 `~/example`의 예시도 참고해 본다.

이제 grok Exporter를 실행하고, grok Exporter가 해당 example.log파일을 어떻게 처리하여, Metric으로 제공하는지 확인해 본다. 추가 테스트를 위해, Log File 뒤에 기록을 추가해 보고, Metric을 다시 확인해 보면, update되어 있음을 확인 할 수 있다.

```
$ ./grok_exporter -config 10-6-grok.yml &
$ curl http://localhost:9144/metrics
$ echo "GET /foo 0.2" >> example.log
$ curl http://localhost:9144/metrics
```

## 10.4. Blackbox Exporter (official)

먼저 용어 설명이 필요할것 같다. "Blackbox Monitoring"이란 무엇일까? 모니터링은 어떤 방법을 체택하는지에 따라 크게 다음과 같이 분류할 수 있다.

- Whitebox Monitoring
- Blackbox Monitoring

일반적으로 Whitebox Monitoring은 에이전트 혹은 소프트웨어를 설치해서 모니터링 하는 것을 말한다. 예를 들어, `Zabbix`, `PRTG`, `Prometheus` 등이 이에 속한다. 일반적인 모니터링 기술들이 이에 속한다고 보면 된다. "Whitebox Monitoring"은 접근성은 좋으나 실제 모니터링 시스템을 구축할 때 모니터링 "소스"가 되는 기술에 대한 이해도가 높아야 한다.

반면 "Blackbox Monitoring"은 시스템 내부가 어떻게 동작하는지보다 **관측 가능한 행위**에 초점을 두는 모니터링 기법이다. 쉽게 예를 들면, CPU가 어떻게 동작하는지, Memory가 어떻게 동작하는지 몰라도 되며, 그냥 CPU 사용량이 얼마인지, Memory 사용량이 얼마인지가 중요할 뿐인 것이다. 프로세스가 사용하는 포트 모니터링도 마찬가지이다. TCP가 어떻게 동작하는지 알 필요가 없다. 실행되는 프로세스에 TCP 연결이 가능한지 그 여부가 중요할 뿐인 것이다.

Blackbox Monitoring쪽에서 유명한 기술은 `Nagios`가 있다. (참고로 Blackbox방식과 유사한 것으로 SNMP스타일도 있다. 어찌보면, 오래된 기법인 MiB기반의 SNMP방식이 프로메테우스가 개념적으로 차용한 것으로 볼수도 있겠다.)

프로메테우스는 Whitebox Monitoring에 속하지만,  Blackbox Monitoring을 지원하기 위해서, 공식적으로 Blackbox Exporter를 지원한다. `blackbox-exporter`는 HTTP, HTTPS는 물론 TCP, ICMP, DNS 등의 프로토콜 위에서 동작하는 엔드포인트들에 대한 Blackbox Monitoring을 할 수 있게 만들어준다. (이것 이외에도 grpc, pop3, ssh, irc 등을 지원하는 것으로 알려져 있다.)

특이한 점은 프로메테우스쪽에서 관측해야할 해당 IP:PORT 정보를 제공해야 한다는 점이다.

우선은 Blackbox Exporter를 설치하고 실행해 본다.  (prometheus.io의 Download에서 구해도 되고, github에서 다운로드 받아도 된다.) 실행할 때는 black Exporter가 취급하여야 할 OS 자원의 권한문제로 sudo 명령을 사용한 것에 주의 한다.

- https://prometheus.io/download/
- github releases: https://github.com/prometheus/blackbox_exporter/releases

```
$ wget https://github.com/prometheus/blackbox_exporter/releases/download/v0.23.0/blackbox_exporter-0.23.0.linux-amd64.tar.gz
$ tar zxvf blackbox_exporter-0.23.0.linux-amd64.tar.gz
$ ln -s blackbox_exporter-0.23.0.linux-amd64/ black
$ cd black
$ sudo ./back_exporter
```

Backbox Exporter를 간단히 살펴보기 위해, 웹 브라우져에서 `http://localhost:9115`로 접근해 본다. Blackbox Exporter의 상태를 확인해 볼 수 있는 페이지를 확인할 수 있다. 또한 아래의 몇 가지 방법으로 접근해 본다. (웹 브라우져가 아니라면, URL부분에 escape character에 주의한다.)

```
// blackbox_exporter self metrics
$ curl http://localhost:9115/metrics

// http
$ curl http://u23:9115/probe\?target\=localhost
$ curl http://u23:9115/probe\?target\=localhost\&module\=http_2xx

// icmp
$ curl http://u23:9115/probe\?target\=localhost\&module\=icmp
$ curl http://u23:9115/probe\?module\=icmp\&target\=localhost
$ curl http://u23:9115/probe\?module\=icmp\&target\=www.google.com
```

- `/metrics`는 blackbox exporter자신에 대한 정보이고, `/probe`는 blackbox가 탐색한 정보들이다.

> 💡 프로메테우스와 blackbox exporter에서 사용하는 host이름 해석 방법은 `getbyhostname()`이 아니라, DNS Resolving에 의한 결과인 것에 주의 한다.

### ICMP

앞의 예제에서 “www.google.com”에 대한 blackbox probe결과는 다른 결과를 보인다. 결과 내용중 일부 관심 있는 부분은 다음과 같다. DNS Resolving은 기본적으로 ipv4/6를 모두 반환하는데, 해당 테스트 host가 ipv6에 대한 설정과 사용이 가능하지 않았기 때문에 아래와 같은 결과가 확인된다.

```
$ curl http://u23:9115/probe\?module\=icmp\&target\=www.google.com
...
probe_ip_protocol 6
probe_success 0
```

```
// with debug mode
$ curl http://u23:9115/probe\?module\=icmp\&target\=www.google.com\&debug\=true
```

- 해당 접속에 대해, debug모드로 추가 정보를 조회해 볼 수 있다. 해당 처리에 대한 상세 내용이 확인되고, Ipv6로 처리한 내역과 상세 결과를 확인 할 수 있다.

보통의 host들은 아직도 ipv4를 주 사용 모드로 사용하고 있으므로, 위의 문제를 해결하기 위해서는 DNS Resolving이 ipv4에 의해서만 가능하도록 `blackbox.yml`의 설정 내용을 아래와 같이 수정한다. `icmp:` 에서 사용할 준비가된 프로토콜이 무엇인지 명시적으로 추가 정보를 설정해 준다.

```
$ more blackbox.yml
...
icmp:
    prober: icmp
    icmp:
      preferred_ip_protocol: ip4
$
```

- 다시 “www.google.com”에 대한 blackbox probe결과를 확인해 보면, 제대로 DNS Resolving이 이루어 졌고, 결과는 “1”로서 true임을 나타내고 있다.

### TCP

TCP기반의 서비스들의 Port에 대해 확인해 본다. 이것은 단순하게 TCP기반의 서비스들중 흔히 사용하는 HTTP, SMTP, telnet, ssh, irc등의 서비스들이 Listen하고 있는 tcp port가 반응하는지 확인하는 것이다. debug mode에서 상세 내용도 확인해 본다.

```
$ curl http://u23:9115/probe\?module\=tcp_connect\&target\=localhost:22
$ curl http://u23:9115/probe\?module\=tcp_connect\&target\=localhost:22\&debug\=true
```

tcp connect 를 위해서는 다음의 내용이 `blackbox.yml`에 정의되어 있어야 한다.

```
$ more blackbox.yml
...
tcp_connect:
    prober: tcp
$
```

TCP기반의 서비스들중 몇몇은 추가적인 확인도 가능하다. 예를 들어 ssh의 경우, 해당 Port로 접근시, 최초에는 Text로 된 Banner로 응답하는 규격인데, 이에 맞도록 확인도 가능하다. 아래와 같이 ssh connect 를 위한 설정을 `blackbox.yml`에 추가하고, test를 진행해 본다.

```
$ more blackbox.yml
...
ssh_banner:
    prober: tcp
    tcp:
      query_response:
      - expect: "^SSH-2.0-"
      - send: "SSH-2.0-blackbox-ssh-check"
$
$ curl http://u23:9115/probe\?module\=ssh_banner\&target\=localhost:22
$ curl http://u23:9115/probe\?module\=ssh_banner\&target\=localhost:22\&debug\=true
```

앞의 2가지 `tcp_connect`와 `ssh_banner` 테스트 결과는 거의 유사하지만, `ssh_banner`가 ssh이 응답하는 것까지 확인하였으므로 `tcp_connect`의 결과보다는 더 의미를 포함한다고 볼 수 있다.

`ssh_banner` 테스트의 경우, 가령 expect에 설정한 정규표현식과 일치하지 않는다면, `probe_success`의 값은 0이 되고, `probe_failed_due_to_regex`는 1이 될 것이다.

### HTTP

HTTP는 현대의 대부분의 웹, 앱에서 제공되는 서비스 방식이다. 프로메테우스에서 HTTP를 통해 metrics를 수집하는 것이 보통의 구성이지만, 경우에 따라 HTTP서비스에 대해 blackbox 모니터링을 수행 할 수도 있다.

아래의 예시는 `http_2xx` (prober:http) 방식의 탐색(관측)에 대한 설정과 실제로 "www.skbroadband.com"서버에 대해 http 프로토콜로 확인한 결과를 얻을 수 있다. 앞의 `tcp_connect`, `ssh_banner` 보다 `http_2xx`는 조금 더 많은 정보들을 포함하고 있다.

```
$ more blackbox.yml
...
http_2xx:
    prober: http
$
$ curl http://u23:9115/probe\?module\=http_2xx\&target\=www.skbroadband.com
```

http probe방식의 탐색(관측)에는 많은 추가 옵션들과 방법들이 있는데, 자세한 것은 아래를 참고한다.

- https://github.com/prometheus/blackbox_exporter/blob/master/CONFIGURATION.md

http probe의 다영한 옵션과 방법들로 HTTP의 header내용 또는 POST body의 내용에서 String pattern을 찾는 등의 조건을 추가할 수도 있다. 간단한 예시로 아래의 설정을 blackbox.yml에 추가하고, 조회한 결과들을 확인해 보자. 이런 식으로 Front Application Instance가 원하는 올바른 결과를 반환하는지 확인하는 것으로 Instance상태에 대한 확인도 가능하다.

```
$ more blackbox.yml
...
http_200_ssl_prom:
    prober: http
    http:
      valid_status_codes: [200]
      fail_if_body_matches_regexp:
      - prometheus
      preferred_ip_protocol: ip4
$
$ curl http://u23:9115/probe\?module\=http_200_ssl_prom\&target\=naver.com
$ curl http://u23:9115/probe\?module\=http_200_ssl_prom\&target\=www.skbroadband.com
$ curl http://u23:9115/probe\?module\=http_200_ssl_prom\&target\=prometheus.io
```

- body에 "prometheus"가 있으면, fail이 되는 조건이다. 명시적으로 사용할 프로토콜이 ipv4로 지정하였다.
- 테스트 결과는 "prometheus.io" 사이트의 결과만, `probe_success` 값이 "0"으로 해당 단어를 포함하고 있음을 알 수 있다. 다른 사이트들은 `probe_success` 의 값이 "1"로 확인된다.
- 각기 다른 Test Case가 필요하다면, `blackbox.yml`에서 `http_200_ssl_prom` 같은 방식으로 각각의 Test Case를 위한 모듈을 추가 설정하면 된다.

### DNS

앞의 다른 Probe와 유사하게 사용할 수 있다. 일반적인 DNS Resolving만을 위한 탐색(관측)보다는 http, tcp, icmp등의 probe 사용시 dns resolving확인이 포함되어 확인하는게 더 상식적이다.

DNS probe에 의한 탐색은 DNS 서비스에 포함되어 있는 MX Record의 확인등 세부적 내용의 확인이 필요한 경우 사용하는 것이 보통이다.

### Prometheus (for blackbox)

앞에서 보았듯이 blackbox exporter는 `/probe` endpoint에서 `module`과 `target URL`을 파라메터로 사용하여 원하는 수집값을 얻어내는 방식이다. 또한, 그 보다 앞선 Service Discovery의 scrape방법에서 보았듯이, `__metrics_path`, `__param_<name>`을 사용하여, 프로메테우스가 scrape하는 대상에 대해 parameter를 포함하는 URL방식으로 감시 대상을 설정 할 수 있었다.

3개의 웹 사이트에 대해, 프로메테우스가 지정하는 서버에 있는 blackbox exporter에 해당 내용을 의뢰하여, 지정된 3개 웹 사이트가 동작중임을 확인하고 Target label로 프로메테우스에서 관리되도록 `prometheus.yml`을 설정한 예제를 살펴 보자

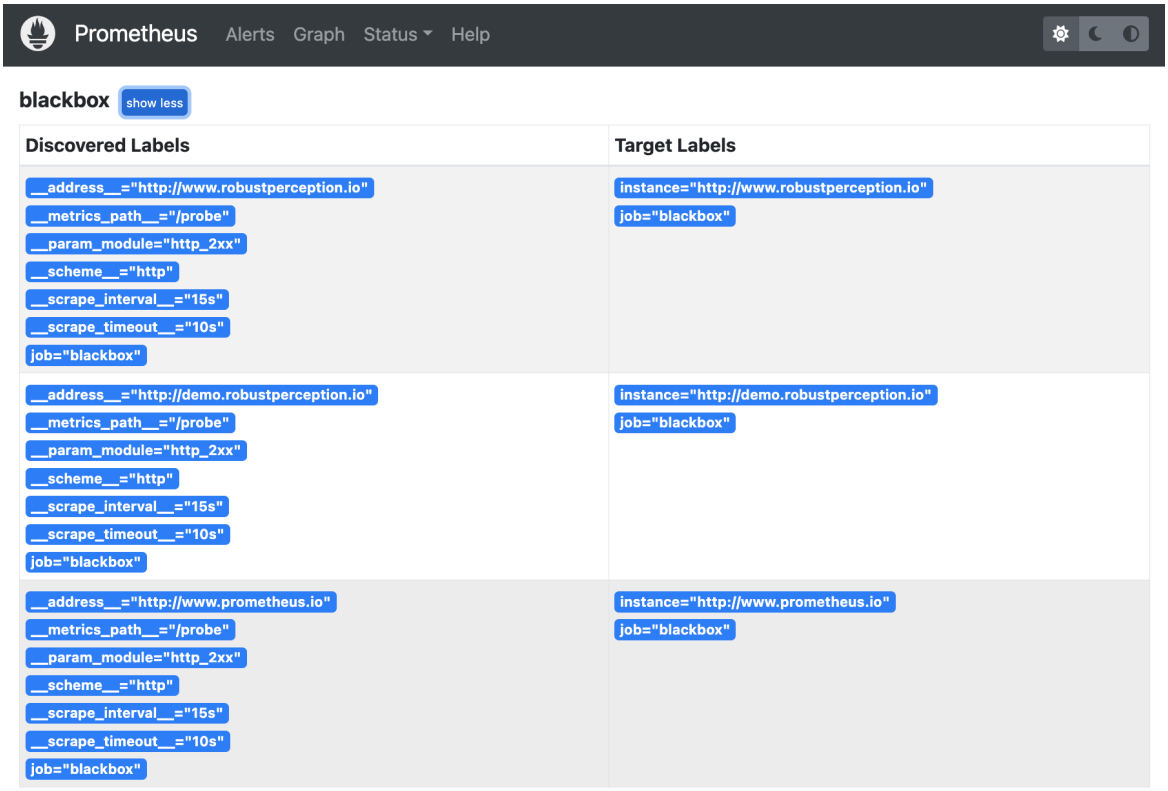
```
scrape_configs:
 - job_name: blackbox
   metrics_path: /probe
   params:
     module: [http_2xx]
   static_configs:
    - targets:
       - http://www.prometheus.io
       - http://www.robustperception.io
       - http://demo.robustperception.io
   relabel_configs:
    - source_labels: [__address__]
      target_label: __param_target
    - source_labels: [__param_target]
      target_label: instance
    - target_label: __address__
      replacement: 127.0.0.1:9115
```

- `metrics_path: /probe`, `module: [http_2xx]` 설정으로 blackbox exporter로 탐색(관측)에 사용할 모듈과 endpoint를 지정하였다. 그러면, 사용할 blackbox exporter는 어디에 있는가? 그것은 `relabel_configs:`에 설정한 내용에서 참조할 수 있다. 여기에서 Target Label에서 알고 있어야할 대상을 지정하는 `__address__` label을 “127.0.0.1:9115”로 설정한 것에서 알 수 있다.
- `relabel_configs:`에서 현재의 `prometheus.yml`에서 소스로 지정한 Source Labels측의 `__address__`, 즉 `taragets:`에 지정된 웹 사이트들의 리스트 내용을 `__param_target`이라는 이름으로 Target Labels측에 relabel하여, blackbox exporter가 URL Parameter로 사용하게 하였다. 동시에 Target Labels측의 `instance`에도 동일하게 relabel하여, 관측하는 대상이 무엇인지 프로메테우스에 남아 있도록 하였다.

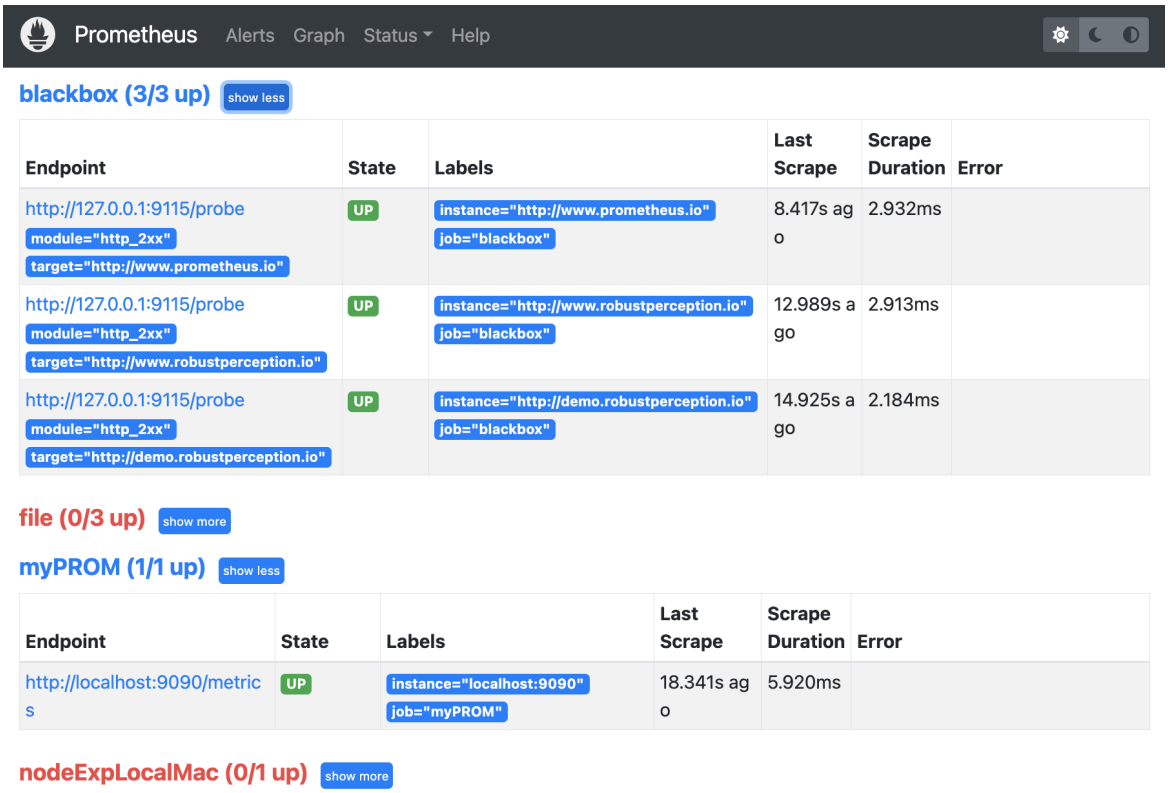
> 💡 개인적인 생각이지만, blackbox와 관련된 Target Labels로의 구성 방법은 차기 버젼에서 바뀌어야 할 것 같다. 이런식으로는 이해하기도 힘들고, 설정과 관리도 힘들 것이다.
>
> 조금더 생각해 본다면, Relabel 개념은 좋은데, Label이 계측대상에 대한 것이 아닌, 설정과 구성관리에서 Label이 사용되며, Relabel 하여 설정을 확장하는 것 자체가 좀 개선이 되어야 할 것 같다.

위의 프로메테우스 설정을 추가하고, 프로메테우스를 재기동한다. 동일 서버에서 backbox exporter도 실행해 놓는다. 이후, 프로메테우스 웹 화면에서 몇 가지 기본적인 화면들을 살펴 보자

- 프로메테우스의 service-discovery 내용: prometheus.yml에서 설정한 감시 대상 3개의 웹사이트가 Discovered Labels에는 `__address__` label로, Target Lebels에는 `instance` label로 구성되어 있다. 해당 내용은 prometheus.yml에서 `taragets:`에 설정한 내용이다.




- 프로메테우스의 Targets 상세 내용: blackbox Exporter에 의한 3개의 웹사이트(`job_name`="blackbox") 에 대한 설정된 내용이 프로메테우스의 Targets 로 등록되어 운용되고 있음을 알 수 있다.
  - blackbox Exporter의 경우, State가 "up"으로 되어 있다는 것은 blackbox Exporter가 up이라는 의미이지, 탐색(관측)한 대사의 상태 결과가 "up"라는 의미가 아니다. 이 정보는 각각의 `/probe`에서 `probe_success`가 "1"인지 확인해야 된다.



- Blackbox의 timeout (우선 순위 또는 적용 룰은 ….)
  - 프로메테우스의 scrape_timeout을 기반으로 자동으로 설정됨
  - +) 프로메테우스는 HTTP 사용시, `X-Prometheus-Scrape-Timeout-Seconds` 라는 header를 사용하는데, blackbox Exporter는 이 값을 자신의 값으로 사용한다.
  - +) blackbox.yml의 timeout필드의 값을 사용

+==