Understanding and using the multi-target exporter pattern

- The multi-target exporter pattern?
- Running multi-target exporters
- Basic querying of multi-target exporters
- Configuring modules
- Querying multi-target exporters with Prometheus
- Example configuration: Multiple targets and multiple modules in one Prometheus job
- Summary

This guide will introduce you to the multi-target exporter pattern. To achieve this we will:

- describe the multi-target exporter pattern and why it is used,
- run the blackbox exporter as an example of the pattern,
- configure a custom query module for the blackbox exporter,
- let the blackbox exporter run basic metric queries against the Prometheus website,
- examine a popular pattern of configuring Prometheus to scrape exporters using relabeling.

The multi-target exporter pattern?

By multi-target exporter pattern we refer to a specific design, in which:

- the exporter will get the target's metrics via a network protocol.
- the exporter does not have to run on the machine the metrics are taken from.
- the exporter gets the targets and a query config string as parameters of Prometheus' GET request.
- the exporter subsequently starts the scrape after getting Prometheus' GET requests and once it is done with scraping.
- the exporter can query multiple targets.

This pattern is only used for certain exporters, such as the blackbox and the SNMP exporter.

The reason is that we either can't run an exporter on the targets, e.g. network gear speaking SNMP, or that we are explicitly interested in the distance, e.g. latency and reachability of a website from a specific point outside of our network, a common use case for the blackbox exporter.

Running multi-target exporters

Multi-target exporters are flexible regarding their environment and can be run in many ways. As regular programs, in containers, as background services, on baremetal, on virtual machines. Because they are queried and do query over network they do need appropriate open ports. Otherwise they are frugal.

Now let's try it out for yourself!

Use Docker to start a blackbox exporter container by running this in a terminal. Depending on your system configuration you might need to prepend the command with a `sudo`:

```
docker run -p 9115:9115 prom/blackbox-exporter
```

You should see a few log lines and if everything went well the last one should report `msg="Listening on address"` as seen here:

```
level=info ts=2018-10-17T15:41:35.4997596Z caller=main.go:324 msg="Listening on address"
address=:9115
```

Basic querying of multi-target exporters

There are two ways of querying:

1. Querying the exporter itself. It has its own metrics, usually available at `/metrics`.
2. Querying the exporter to scrape another target. Usually available at a "descriptive" endpoint, e.g. `/probe`. This is likely what you are primarily interested in, when using multi-target exporters.

You can manually try the first query type with curl in another terminal or use this link:

```
curl 'localhost:9115/metrics'
```

The response should be something like this:

```
# HELP blackbox_exporter_build_info A metric with a constant '1' value labeled by version,
revision, branch, and goversion from which blackbox_exporter was built.
# TYPE blackbox_exporter_build_info gauge
blackbox_exporter_build_info{branch="HEAD",goversion="go1.10",revision="4a22506cf0cf139d9b2f9c
de099f0012d9fcabde",version="0.12.0"} 1
# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 0
go_gc_duration_seconds{quantile="0.25"} 0
go_gc_duration_seconds{quantile="0.5"} 0
go_gc_duration_seconds{quantile="0.75"} 0
go_gc_duration_seconds{quantile="1"} 0
go_gc_duration_seconds_sum 0
go_gc_duration_seconds_count 0
# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 9

[…]

# HELP process_cpu_seconds_total Total user and system CPU time spent in seconds.
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 0.05
# HELP process_max_fds Maximum number of open file descriptors.
# TYPE process_max_fds gauge
process_max_fds 1.048576e+06
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 7
# HELP process_resident_memory_bytes Resident memory size in bytes.
# TYPE process_resident_memory_bytes gauge
process_resident_memory_bytes 7.8848e+06
# HELP process_start_time_seconds Start time of the process since unix epoch in seconds.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.54115492874e+09
```

```
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 1.5609856e+07
```

Those are metrics in the Prometheus format. They come from the exporter's instrumentation and tell us about the state of the exporter itself while it is running. This is called whitebox monitoring and very useful in daily ops practice. If you are curious, try out our guide on how to instrument your own applications.

For the second type of querying we need to provide a target and module as parameters in the HTTP GET Request. The target is a URI or IP and the module must defined in the exporter's configuration. The blackbox exporter container comes with a meaningful default configuration. We will use the target `prometheus.io` and the predefined module `http_2xx`. It tells the exporter to make a GET request like a browser would if you go to `prometheus.io` and to expect a 200 OK response.

You can now tell your blackbox exporter to query `prometheus.io` in the terminal with curl:

```
curl 'localhost:9115/probe?target=prometheus.io&module=http_2xx'
```

This will return a lot of metrics:

```
# HELP probe_dns_lookup_time_seconds Returns the time taken for probe dns lookup in seconds
# TYPE probe_dns_lookup_time_seconds gauge
probe_dns_lookup_time_seconds 0.061087943
# HELP probe_duration_seconds Returns how long the probe took to complete in seconds
# TYPE probe_duration_seconds gauge
probe_duration_seconds 0.065580871
# HELP probe_failed_due_to_regex Indicates if probe failed due to regex
# TYPE probe_failed_due_to_regex gauge
probe_failed_due_to_regex 0
# HELP probe_http_content_length Length of http content response
# TYPE probe_http_content_length gauge
probe_http_content_length 0
# HELP probe_http_duration_seconds Duration of http request by phase, summed over all
redirects
# TYPE probe_http_duration_seconds gauge
probe_http_duration_seconds{phase="connect"} 0
probe_http_duration_seconds{phase="processing"} 0
probe_http_duration_seconds{phase="resolve"} 0.061087943
probe_http_duration_seconds{phase="tls"} 0
probe_http_duration_seconds{phase="transfer"} 0
```

```
# HELP probe_http_redirects The number of redirects
# TYPE probe_http_redirects gauge
probe_http_redirects 0
# HELP probe_http_ssl Indicates if SSL was used for the final redirect
# TYPE probe_http_ssl gauge
probe_http_ssl 0
# HELP probe_http_status_code Response HTTP status code
# TYPE probe_http_status_code gauge
probe_http_status_code 0
# HELP probe_http_version Returns the version of HTTP of the probe response
# TYPE probe_http_version gauge
probe_http_version 0
# HELP probe_ip_protocol Specifies whether probe ip protocol is IP4 or IP6
# TYPE probe_ip_protocol gauge
probe_ip_protocol 6
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 0
```

Notice that almost all metrics have a value of `0`. The last one reads `probe_success 0`. This means the prober could not successfully reach `prometheus.io`. The reason is hidden in the metric `probe_ip_protocol` with the value `6`. By default the prober uses IPv6 until told otherwise. But the Docker daemon blocks IPv6 until told otherwise. Hence our blackbox exporter running in a Docker container can't connect via IPv6.

We could now either tell Docker to allow IPv6 or the blackbox exporter to use IPv4. In the real world both can make sense and as so often the answer to the question "what is to be done?" is "it depends". Because this is an exporter guide we will change the exporter and take the opportunity to configure a custom module.

Configuring modules

The modules are predefined in a file inside the docker container called `config.yml` which is a copy of blackbox.yml in the github repo.

We will copy this file, adapt it to our own needs and tell the exporter to use our config file instead of the one included in the container.

First download the file using curl or your browser:

```
curl -o blackbox.yml
https://raw.githubusercontent.com/prometheus/blackbox_exporter/master/blackbox.yml
```

Open it in an editor. The first few lines look like this:

```
modules:
  http_2xx:
    prober: http
  http_post_2xx:
    prober: http
    http:
      method: POST
```

YAML uses whitespace indentation to express hierarchy, so you can recognise that two `modules` named `http_2xx` and `http_post_2xx` are defined, and that they both have a prober `http` and for one the method value is specifically set to `POST`. You will now change the module `http_2xx` by setting the `preferred_ip_protocol` of the prober `http` explicitly to the string `ip4`.

```
modules:
  http_2xx:
    prober: http
    http:
      preferred_ip_protocol: "ip4"
  http_post_2xx:
    prober: http
    http:
      method: POST
```

If you want to know more about the available probers and options check out the documentation.

Now we need to tell the blackbox exporter to use our freshly changed file. You can do that with the flag `--config.file="blackbox.yml"`. But because we are using Docker, we first must make this file available inside the container using the `--mount` command.

NOTE: If you are using macOS you first need to allow the Docker daemon to access the directory in which your `blackbox.yml` is. You can do that by clicking on the little Docker whale in menu bar and then on `Preferences`->`File Sharing`->`+`. Afterwards press `Apply & Restart`.

First you stop the old container by changing into its terminal and press `ctrl+c`. Make sure you are in the directory containing your `blackbox.yml`. Then you run this command. It is long, but we will explain it:

```
docker \
  run -p 9115:9115 \
  --mount type=bind,source="$(pwd)"/blackbox.yml,target=/blackbox.yml,readonly \
  prom/blackbox-exporter \
  --config.file="/blackbox.yml"
```

With this command, you told `docker` to:

1. `run` a container with the port `9115` outside the container mapped to the port `9115` inside of the container.
2. `mount` from your current directory (`$(pwd)` stands for print working directory) the file `blackbox.yml` into `/blackbox.yml` in `readonly` mode.
3. use the image `prom/blackbox-exporter` from Docker hub.
4. run the blackbox-exporter with the flag `--config.file` telling it to use `/blackbox.yml` as config file.

If everything is correct, you should see something like this:

```
level=info ts=2018-10-19T12:40:51.650462756Z caller=main.go:213 msg="Starting
blackbox_exporter" version="(version=0.12.0, branch=HEAD,
revision=4a22506cf0cf139d9b2f9cde099f0012d9fcabde)"
level=info ts=2018-10-19T12:40:51.653357722Z caller=main.go:220 msg="Loaded config file"
level=info ts=2018-10-19T12:40:51.65349635Z caller=main.go:324 msg="Listening on address"
address=:9115
```

Now you can try our new IPv4-using module `http_2xx` in a terminal:

```
curl 'localhost:9115/probe?target=prometheus.io&module=http_2xx'
```

Which should return Prometheus metrics like this:

```
# HELP probe_dns_lookup_time_seconds Returns the time taken for probe dns lookup in seconds
# TYPE probe_dns_lookup_time_seconds gauge
probe_dns_lookup_time_seconds 0.02679421
# HELP probe_duration_seconds Returns how long the probe took to complete in seconds
# TYPE probe_duration_seconds gauge
probe_duration_seconds 0.461619124
# HELP probe_failed_due_to_regex Indicates if probe failed due to regex
# TYPE probe_failed_due_to_regex gauge
probe_failed_due_to_regex 0
# HELP probe_http_content_length Length of http content response
```

```
# TYPE probe_http_content_length gauge
probe_http_content_length -1
# HELP probe_http_duration_seconds Duration of http request by phase, summed over all
redirects
# TYPE probe_http_duration_seconds gauge
probe_http_duration_seconds{phase="connect"} 0.062076202999999996
probe_http_duration_seconds{phase="processing"} 0.23481845699999998
probe_http_duration_seconds{phase="resolve"} 0.029594103
probe_http_duration_seconds{phase="tls"} 0.163420078
probe_http_duration_seconds{phase="transfer"} 0.002243199
# HELP probe_http_redirects The number of redirects
# TYPE probe_http_redirects gauge
probe_http_redirects 1
# HELP probe_http_ssl Indicates if SSL was used for the final redirect
# TYPE probe_http_ssl gauge
probe_http_ssl 1
# HELP probe_http_status_code Response HTTP status code
# TYPE probe_http_status_code gauge
probe_http_status_code 200
# HELP probe_http_uncompressed_body_length Length of uncompressed response body
# TYPE probe_http_uncompressed_body_length gauge
probe_http_uncompressed_body_length 14516
# HELP probe_http_version Returns the version of HTTP of the probe response
# TYPE probe_http_version gauge
probe_http_version 1.1
# HELP probe_ip_protocol Specifies whether probe ip protocol is IP4 or IP6
# TYPE probe_ip_protocol gauge
probe_ip_protocol 4
# HELP probe_ssl_earliest_cert_expiry Returns earliest SSL cert expiry in unixtime
# TYPE probe_ssl_earliest_cert_expiry gauge
probe_ssl_earliest_cert_expiry 1.581897599e+09
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 1
# HELP probe_tls_version_info Contains the TLS version used
# TYPE probe_tls_version_info gauge
probe_tls_version_info{version="TLS 1.3"} 1
```

You can see that the probe was successful and get many useful metrics, like latency by phase, status code, ssl status or certificate expiry in Unix time. The blackbox exporter also offers a tiny web interface at localhost:9115 for you to check out the last few probes, the loaded config and debug information. It even offers a direct link to probe `prometheus.io`. Handy if you are wondering why something does not work.

Querying multi-target exporters with Prometheus

So far, so good. Congratulate yourself. The blackbox exporter works and you can manually tell it to query a remote target. You are almost there. Now you need to tell Prometheus to do the queries for us.

Below you find a minimal prometheus config. It is telling Prometheus to scrape the exporter itself as we did before using `curl 'localhost:9115/metrics'`:

NOTE: If you use Docker for Mac or Docker for Windows, you can't use `localhost:9115` in the last line, but must use `host.docker.internal:9115`. This has to do with the virtual machines used to implement Docker on those operating systems. You should not use this in production.

`prometheus.yml` for Linux:

```
global:
  scrape_interval: 5s

scrape_configs:
- job_name: blackbox # To get metrics about the exporter itself
  metrics_path: /metrics
  static_configs:
    - targets:
      - localhost:9115
```

`prometheus.yml` for macOS and Windows:

```
global:
  scrape_interval: 5s

scrape_configs:
- job_name: blackbox # To get metrics about the exporter itself
```

```
  metrics_path: /metrics
  static_configs:
    - targets:
      - host.docker.internal:9115
```

Now run a Prometheus container and tell it to mount our config file from above. Because of the way networking on the host is addressable from the container you need to use a slightly different command on Linux than on MacOS and Windows.:

Run Prometheus on Linux (don't use `--network="host"` in production):

```
docker \
  run --network="host"\
  --mount type=bind,source="$(pwd)"/prometheus.yml,target=/prometheus.yml,readonly \
  prom/prometheus \
  --config.file="/prometheus.yml"
```

Run Prometheus on MacOS and Windows:

```
docker \
  run -p 9090:9090 \
  --mount type=bind,source="$(pwd)"/prometheus.yml,target=/prometheus.yml,readonly \
  prom/prometheus \
  --config.file="/prometheus.yml"
```

This command works similarly to running the blackbox exporter using a config file.

If everything worked, you should be able to go to localhost:9090/targets and see under `blackbox` an endpoint with the state `UP` in green. If you get a red `DOWN` make sure that the blackbox exporter you started above is still running. If you see nothing or a yellow `UNKNOWN` you are really fast and need to give it a few more seconds before reloading your browser's tab.

To tell Prometheus to query `"localhost:9115/probe?target=prometheus.io&module=http_2xx"` you add another scrape job `blackbox-http` where you set the `metrics_path` to `/probe` and the parameters under `params:` in the Prometheus config file `prometheus.yml`:

```
global:
  scrape_interval: 5s

scrape_configs:
```

```
- job_name: blackbox # To get metrics about the exporter itself
  metrics_path: /metrics
  static_configs:
    - targets:
      - localhost:9115   # For Windows and macOS replace with - host.docker.internal:9115

- job_name: blackbox-http # To get metrics about the exporter’s targets
  metrics_path: /probe
  params:
    module: [http_2xx]
    target: [prometheus.io]
  static_configs:
    - targets:
      - localhost:9115   # For Windows and macOS replace with - host.docker.internal:9115
```

After saving the config file switch to the terminal with your Prometheus docker container and stop it by pressing `ctrl+C` and start it again to reload the configuration by using the existing command.

The terminal should return the message `"Server is ready to receive web requests."` and after a few seconds you should start to see colourful graphs in your Prometheus.

This works, but it has a few disadvantages:

1. The actual targets are up in the param config, which is very unusual and hard to understand later.
2. The `instance` label has the value of the blackbox exporter’s address which is technically true, but not what we are interested in.
3. We can’t see which URL we probed. This is unpractical and will also mix up different metrics into one if we probe several URLs.

To fix this, we will use relabeling. Relabeling is useful here because behind the scenes many things in Prometheus are configured with internal labels. The details are complicated and out of scope for this guide. Hence we will limit ourselves to the necessary. But if you want to know more check out this talk. For now it suffices if you understand this:

- All labels starting with `__` are dropped after the scrape. Most internal labels start with `__`.
- You can set internal labels that are called `__param_<name>`. Those set URL parameter with the key `<name>` for the scrape request.
- There is an internal label `__address__` which is set by the `targets` under `static_configs` and whose value is the hostname for the scrape request. By default it is later used to set the value for the label `instance`, which is attached to each metric and tells you where the metrics came from.

Here is the config you will use to do that. Don’t worry if this is a bit much at once, we will go through it step by step:

```
global:
  scrape_interval: 5s

scrape_configs:
- job_name: blackbox # To get metrics about the exporter itself
  metrics_path: /metrics
  static_configs:
    - targets:
      - localhost:9115   # For Windows and macOS replace with - host.docker.internal:9115

- job_name: blackbox-http # To get metrics about the exporter’s targets
  metrics_path: /probe
  params:
    module: [http_2xx]
  static_configs:
    - targets:
      - http://prometheus.io    # Target to probe with http
      - https://prometheus.io   # Target to probe with https
      - http://example.com:8080 # Target to probe with http on port 8080
  relabel_configs:
    - source_labels: [__address__]
      target_label: __param_target
    - source_labels: [__param_target]
      target_label: instance
    - target_label: __address__
      replacement: localhost:9115  # The blackbox exporter’s real hostname:port. For Windows
and macOS replace with - host.docker.internal:9115
```

So what is new compared to the last config?

`params` does not include `target` anymore. Instead we add the actual targets under `static configs: targets`. We also use several because we can do that now:

```
  params:
    module: [http_2xx]
  static_configs:
    - targets:
      - http://prometheus.io    # Target to probe with http
```

```
      - https://prometheus.io   # Target to probe with https
      - http://example.com:8080 # Target to probe with http on port 8080
```

`relabel_configs` contains the new relabeling rules:

```
  relabel_configs:
    - source_labels: [__address__]
      target_label: __param_target
    - source_labels: [__param_target]
      target_label: instance
    - target_label: __address__
      replacement: localhost:9115  # The blackbox exporter's real hostname:port. For Windows
and macOS replace with - host.docker.internal:9115
```

Before applying the relabeling rules, the URI of a request Prometheus would make would look like this: `"http://prometheus.io/probe?module=http_2xx"`. After relabeling it will look like this `"http://localhost:9115/probe?target=http://prometheus.io&module=http_2xx"`.

Now let us explore how each rule does that:

First we take the values from the label `__address__` (which contain the values from `targets`) and write them to a new label `__param_target` which will add a parameter `target` to the Prometheus scrape requests:

```
  relabel_configs:
    - source_labels: [__address__]
      target_label: __param_target
```

After this our imagined Prometheus request URI has now a target parameter: `"http://prometheus.io/probe?target=http://prometheus.io&module=http_2xx"`.

Then we take the values from the label `__param_target` and create a label instance with the values.

```
  relabel_configs:
    - source_labels: [__param_target]
      target_label: instance
```

Our request will not change, but the metrics that come back from our request will now bear a label `instance="http://prometheus.io"`.

After that we write the value `localhost:9115` (the URI of our exporter) to the label `__address__`. This will be used as the hostname and port for the Prometheus scrape requests. So that it queries the exporter and not the target URI directly.

```
  relabel_configs:
    - target_label: __address__
      replacement: localhost:9115  # The blackbox exporter's real hostname:port. For Windows
and macOS replace with - host.docker.internal:9115
```

Our request is now `"localhost:9115/probe?target=http://prometheus.io&module=http_2xx"`. This way we can have the actual targets there, get them as `instance` label values while letting Prometheus make a request against the blackbox exporter.

Often people combine these with a specific service discovery. Check out the configuration documentation for more information. Using them is no problem, as these write into the `__address__` label just like `targets` defined under `static_configs`.

That is it. Restart the Prometheus docker container and look at your metrics. Pay attention that you selected the period of time when the metrics were actually collected.

Example configuration: Multiple targets and multiple modules in one Prometheus job

In the example above we used a single blackbox exporter probe module only. The `module` is specified as a job-level parameter of the `blackbox-http` job. Since it is a list, it can contain multiple elements, but the blackbox exporter only uses the first one. However, we can use multiple blackbox exporter modules in a single job if we need to probe separate groups of targets with appropriate modules for each. Instead of specifying the `module` param of the job, we can specify a `module` label in each `static_config`. Additionally the `relabel_configs` sequence needs to be expanded with a new item, because we have to inject the module (as `__param_module`) in the requests at this point.

```
global:
  scrape_interval: 5s

scrape_configs:
- job_name: blackbox               # To get metrics about the exporter itself
  metrics_path: /metrics
  static_configs:
    - targets:
```

```
      - localhost:9115              # The blackbox exporter's real hostname:port

- job_name: blackbox-probes         # To get metrics about the exporter's targets with
different modules
  metrics_path: /probe
  static_configs:
    - targets:                      # static_config: targets and http_2xx module
      - http://prometheus.io
      - https://prometheus.io
      - http://example.com:8080
      labels:
        module: http_2xx
    - targets:                      # static_config: targets and icmp module
      - prometheus.io
      - example.com
      labels:
        module: icmp
  relabel_configs:
    - source_labels: [__address__]
      target_label: __param_target
    - source_labels: [__param_target]
      target_label: instance
    - source_labels: [module]
      target_label: __param_module
    - target_label: __address__
      replacement: localhost:9115   # The blackbox exporter's real hostname:port
```

Summary

In this guide, you learned how the multi-target exporter pattern works, how to run a blackbox exporter with a customised module, and to configure Prometheus using relabeling to scrape metrics with prober labels.

□□□ #2
□ □□□ □□□ 2026-10-31 13:19:13 UTC
□ □□□ □□□ 2026-10-31 13:19:15 UTC