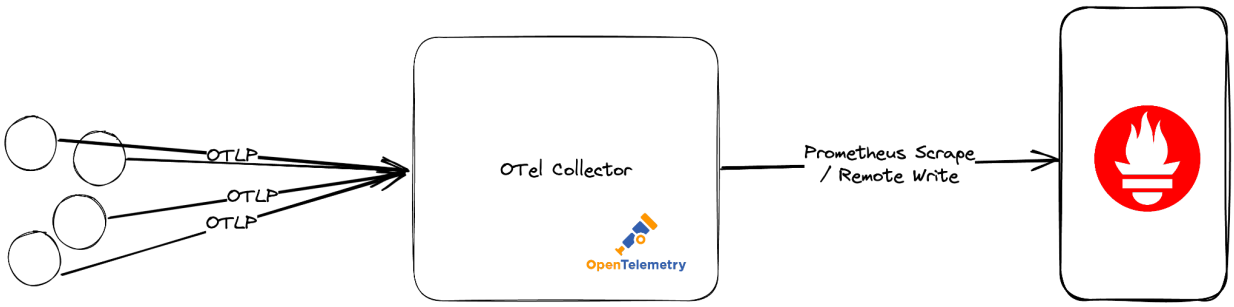
UX of using target_info

Note: This is a public document

Architecture

It assumes the following architecture:



i.e, You have applications emitting OTLP which goes through the OTel Collector and finally to the Prometheus server or Prometheus compatible solutions.

OpenTelemetry Metric Structure

A metric in OTel is made up of several parts. Roughly:

1. Resource attributes which are key-value pairs describing the “source” of the metric. **service.name** is required.

   For example:
   - **service.name**
   - service.namespace
   - service.instance.id
   - service.version
   - deployment.environment
   - k8s.cluster.name
   - k8s.node.name
   - k8s.namespace.name
   - k8s.pod.name
   - k8s.container.name
   - k8s.deployment.name
   - cloud.provider

    - cloud.region
    - cloud.availability_zone
    - …….

2. Metric attributes describe the metric itself. **name and type** are required.
   For example:
    - **name**=http.requests
    - **type**=counter
    - status.code=500
    - path=/login

3. The timestamp and value:
    - value=2000 @timestamp

Prometheus Metric Structure

Prometheus has a flat structure for metrics. A metric is identified by a set of label value pairs, just like OTel Metric, but it doesn't differentiate between labels describing the target and those describing the metric itself:

For example:

```
None
prometheus_http_requests_total{
      code="200",
      handler="/static/*filepath",
      cluster="dev-us-central-0",
      container="prometheus",
      instance="prometheus-0",
      job="default/prometheus",
      namespace="default",
}     1622 1680529110698
```

While this is the final structure that is stored in the database, Prometheus actually arrives at this by merging *target labels* and the actual metric exposed by the application. Brian Brazil has a timeless blog here that describes how labels are attached to metrics here: [Life of a Label – Robust Perception | Prometheus Monitoring Experts]

In the above example, the application only exposes the metric:

```
None
prometheus_http_requests_total{code="200", handler="/static/*filepath"}
1622
```

But when Prometheus discovers the application, it also discovers the metadata associated with the target. And we manually choose (via relabelling) which of this metadata to keep and which to discard. In the above example, we choose to keep: *cluster, container, namespace*.

Note that **job** and **instance** are special labels that are expected on every metric and these labels are supposed to uniquely identify the target. This is to make sure that the same metric exposed by two different targets don't overwrite each other.

Mapping from OTel to Prometheus

When mapping from OTel to Prometheus, the easy way to do it is add all the labels to the metric in Prometheus. Something like this:

```
None
http_requests_total{
      status_code="500",
      path="/login",
      job=<service.namespace/service.name>,
      instance=<service.instance.id>,
      ……resource attrs….
} 2000
```

Note: *job* and *instance* are mapped to the closest primitives in OTel.

However, the problem is that the resource attributes are added to *every single metric* and it could be a lot. A snapshot of the default resource attributes for a Go application running in Kubernetes:

```
None
container_id="fe86a4728728bdd534696a56bf378966f1adf2b5fbce9e76dd638e7fef
92621d",
```

```
host_name="opentelemetry-checkoutservice-6b5dc5954c-q6qcw",
instance="8211de3c-ab65-4909-86b7-236c92d189b1",
job="opentelemetry-demo/checkoutservice",
k8s_namespace_name="otel-demo",
k8s_node_name="gke-dev-us-central-0-main-n2s16-1-1af17bab-mcpn",
k8s_pod_name="opentelemetry-checkoutservice-6b5dc5954c-q6qcw",
os_description="Alpine Linux 3.17.2 (Linux 5.10.162+ #1 SMP Fri Jan 27
10:11:23 UTC 2023 x86_64)",
os_type="linux",
process_command_args="["./checkoutservice"]",
process_executable_name="checkoutservice",
process_executable_path="/usr/src/app/checkoutservice",
process_owner="root"
process_pid="1",
process_runtime_description="go version go1.19.2 linux/amd64",
process_runtime_name="go",
process_runtime_version="go1.19.2",
telemetry_sdk_language="go",
telemetry_sdk_name="opentelemetry",
telemetry_sdk_version="1.10.0",
```

Enter OpenMetric's *target_info*

To help with this problem of capturing contextual metadata, OpenMetrics has a primitive called *target_info*.

> The preferred solution is to provide this target metadata as part of the exposition, but in a way that does not impact on the exposition as a whole. Info MetricFamilies are designed for this. An exposer may include an Info MetricFamily called "target" with a single Metric with no labels with the metadata. An example in the text format might be:

None

```
# TYPE target info
# HELP target Target metadata
target_info{env="prod",hostname="myhost",datacenter="sdc",re
gion="europe",owner="frontend"} 1
```

OpenTelemetry leverages this and moves the resource attributes into ***target_info***.

So when we are converting from OTel to Prometheus, it becomes the following:

```
None
http_requests_total{status_code="500", path="/login",
job=<service.namespace/service.name>,
instance=<service.instance.id>} 2000
target_info{...resource attrs...,
job=<service.namespace/service.name>,
instance=<service.instance.id>} 1
```

Note that *target_info* is created once per target and not for each metric.

Using *target_info*

Great, now we can model from OpenTelemetry to Prometheus, but the job is not done. We still have to use this information.

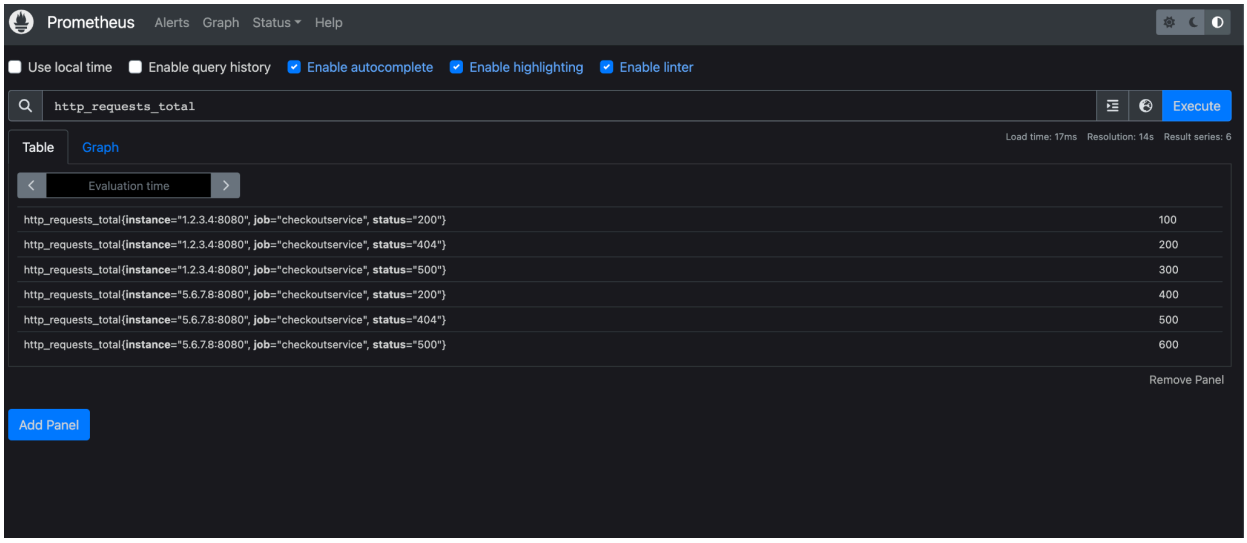
The best way to do this is with an example:

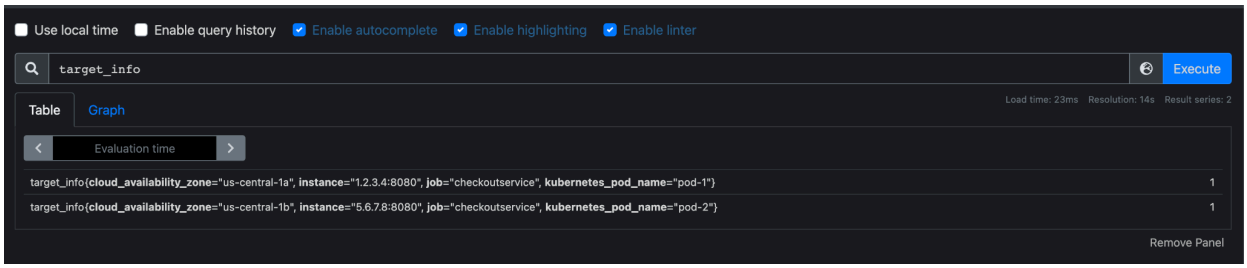
- Let's say we have one service = “checkoutservice”, with two replicas:
    - {service.instance.id = “1.2.3.4:8080”, cloud.availability_zone=”us-central-1a”, kubernetes.pod.name=pod-1}
    - {service.instance.id=”5.6.7.8:8080”, cloud.avalability_zone=”us-central-1b”, kubernetes.pod.name=pod-2}

And each replica is exposing the following metrics:

- http.requests{status=200}
- http.requests{status=404}
- http.requests{status=500}

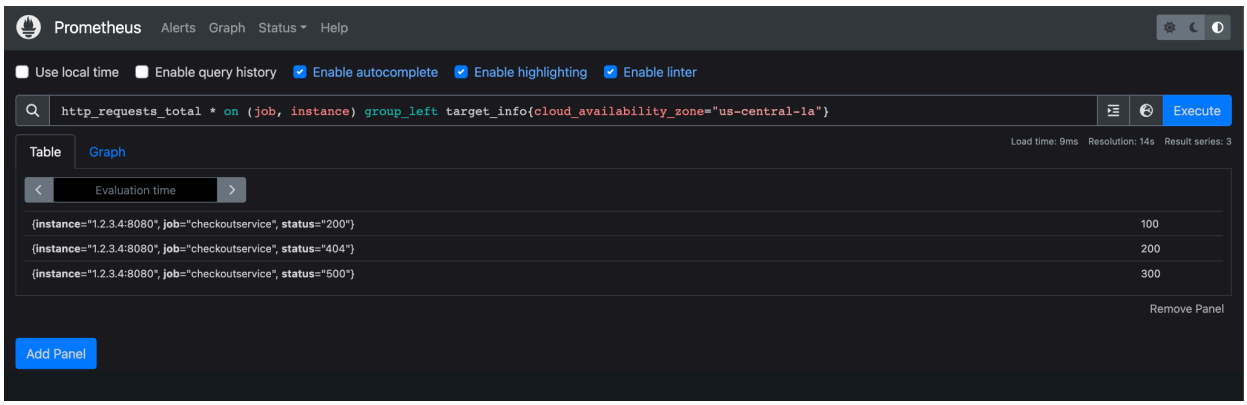
When we apply the default mapping it will look like this:





Now lets says you want to only see the metrics for *cloud_availability_zone="us-central-1a"*, you have to do a join in Prometheus:

```
None
  http_requests_total
* on (job, instance) group_left ()
  target_info{cloud_availability_zone="us-central-1a"}
```



If you want to copy the values of certain resource labels into the result metrics, then you put the labels into group_left():

```
  http_requests_total
* on (job, instance) group_left (kubernetes_pod_name)
  target_info{cloud_availability_zone="us-central-1a"}
```

However, these are trivial queries. If you have a histogram: *traces_span_metrics_duration_seconds_bucket* that you are trying to get the 95%ile broken down by cloud_availability_zone, it just becomes too complex:

```
histogram_quantile(
  0.95,
  sum by (le, cloud_availability_zone) (

rate(traces_span_metrics_duration_seconds_bucket{job="checkoutservice"}[5m])
    * on (job, instance) group_left (cloud_availability_zone)
      target_info
  )
)
```

Also, from the OTel demo, if you want to sum the metric `app_ads_ad_requests` by `namespace`, you need to write:

```
None
sum by (k8s_namespace_name) (app_ads_ad_requests  * on (job, instance)
group_left (k8s_namespace_name)  target_info)
```

Contrast this to how native Prometheus feels like:

```
None
sum by (k8s_namespace_name) (app_ads_ad_requests)
```

What's the alternative?

Let's take a step back and consider what Prometheus does. Prometheus also has labels describing a target, and it doesn't use *target_info*. And it even has more labels that OpenTelemetry, for example, for kubernetes, it has the following: https://prometheus.io/docs/prometheus/latest/configuration/configuration/#kubernetes_sd_config

However, what it does is forces users to pick which target labels are important and apply it to all metrics. For example, at Grafana Labs, we pick the following:

```
None
      cluster="dev-us-central-0",
      container="prometheus",
      instance="prometheus-0",
      job="default/prometheus",
      namespace="default",
      pod="prometheus-0"
```

If we want the 95%ile grouped by *cluster*, the query is the following:

```
None
histogram_quantile(
  0.95,
  sum by (le, cluster)
(rate(traces_span_metrics_duration_seconds_bucket{job="checkoutservice"}[5m]))
)
```

This is the tradeoff we make. We can't select or slice and dice on *every* dimension, but we have to pick ahead of time for the best UX. And it works quite well, because we typically know which dimensions we are interested in!

To achieve this in OTel Collector is quite simple actually. We have to copy the labels over from resource attributes to metric attributes:

```
None
processor:
  transform:
    metric_statements:
      - context: metric
        statements:
        - set(attributes["namespace"],
resource.attributes["k8s_namespace_name"])
        - set(attributes["container"],
resource.attributes["k8s.container.name"])
        - set(attributes["pod"], resource.attributes["k8s.pod.name"])
        - set(attributes["cluster"], resource.attributes["k8s.cluster.name"])
```

By leveraging this, the metrics will have the *namespace, container, pod* and *cluster* labels that you can then select and group on.

This mimics how Prometheus works and would give you the best UX for querying the data back.