1. 설치하기

- 검증 목표: OpenTelemetry PoC
- OpenTelemetry는 데이터를 모으기 위해 코드를 직접 작성하거나, agent를 사용하거나, Kubernetes operator를 제공한다.
- OpenTelemetry K8S Operator 사용하기
  - 공식 문서

1. cert-manager 설치

- OpenTelemetry Operator for Kubernetes를 사용하기 위해서는 cert-manager가 설치되어 있어야 하므로, 먼저 아래 명령어로 helm chart를 통해 cert-manager를 설치한다. (문서)

```
# namespace
kubectl create ns cert-manager
kubectl config set-context --current --namespace cert-manager

# helm repository
helm repo add jetstack https://charts.jetstack.io
helm repo update

# cert-manager  CRD
kubectl apply -f https://github.com/cert-manager/cert-manager
/releases/download/v1.11.0/cert-manager.crds.yaml

# cert-manager
helm install \
  cert-manager jetstack/cert-manager \
  -n cert-manager \
  --version v1.11.0
```

2. OpenTelemetry Operator for Kubernetes 설치

- OpenTelemetry K8S Operator를 아래 명령어로 설치한다.

```
helm repo add open-telemetry https://open-telemetry.github.io
/opentelemetry-helm-charts
helm repo update

helm install \
  --namespace opentelemetry-operator-system
  opentelemetry-operator open-telemetry/opentelemetry-operator
```

3. OpenTelemetry Collector 설치

- 문서

- Opentelemetry가 제시하는 좋은 접근법은 데이터를 바로 backend 쪽으로 보내는 것이 아니라, 먼저 OpenTelemetry Collector(`OpenTelemetryCollector`) 로 보내는 것이다. 이렇게 하면 민감 정보 관리 및 데이터 추출에 관련된 문제들(재시도 등)을 클라이언트 단에서 수월하게 처리하도록 decoupling할 수 있으며, 추가적인 데이터를 넣어 보낼 수 있다.
- (2)번에서 설치한 operator는 이 collector를 위한 CRD를 포함한다.
- 아래 명령으로 설치해보자.

```
• kubectl create ns otel

  kubectl apply -f - <<EOF
  apiVersion: opentelemetry.io/v1alpha1
  kind: OpenTelemetryCollector
  metadata:
    name: otel-poc
    namespace: otel
  spec:
    config: |
      receivers:
        otlp:
          protocols:
            grpc:
            http:
      processors:
        memory_limiter:
          check_interval: 1s
          limit_percentage: 75
          spike_limit_percentage: 15
        batch:
          send_batch_size: 10000
          timeout: 10s

      exporters:
        logging:

      service:
        pipelines:
          traces:
            receivers: [otlp]
            processors: [memory_limiter, batch]
            exporters: [logging]
          metrics:
            receivers: [otlp]
            processors: [memory_limiter, batch]
            exporters: [logging]
          logs:
            receivers: [otlp]
            processors: [memory_limiter, batch]
            exporters: [logging]
  EOF
```

- 결과는 아래와 같다.

```
$ kubectl get deploy
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
otel-poc-collector    1/1     1            1           11s
```

- 참고: Auto-instrumentation은 지원되는 프레임워크가 정해져 있는데, CloudForet의 경우 python-core 라는 직접 개발한 프레임워크를 사용하기에 Auto-instrumentation을 사용할 수 없다. (관련 Issue)

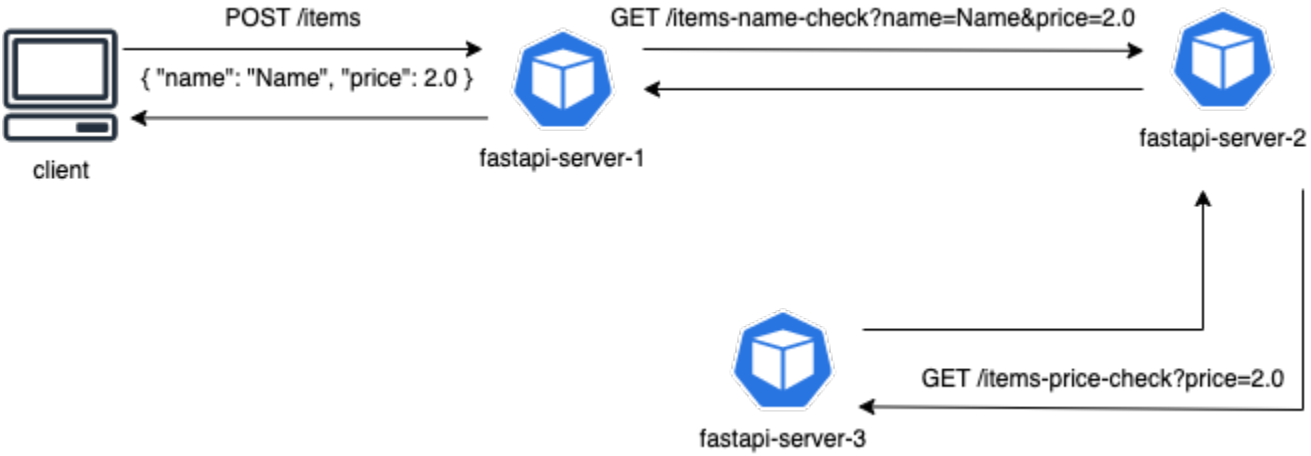
4. 코드에 tracing 설정하기

- 문서
- 이 부분에서는 trace에 대한 설정을 코드를 수정함으로써 적용한다.
- 먼저 아래 명령어로 필요한 패키지를 설치한다.

```
pip install opentelemetry-api
pip install opentelemetry-sdk
pip install opentelemetry-exporter-otlp-proto-grpc
pip install opentelemetry-instrumentation-logging
pip installopentelemetry-exporter-prometheus
```

4.0 예시 아키텍쳐

- 여기서 사용할 코드는 FastAPI로 작성한 web serber 3개로 구성된다.
  - fastapi-server-1
  - fastapi-server-2
  - fastapi-server-3
- 이 세 개 서버가 상호작용하는 과정은 아래와 같다.



4-1. OpenTelemetry tracer 코드

- 아래 코드는 OpenTelemetry가 trace를 수집하기 위해 기본적으로 사용하는 `TraceProvider`를 초기화하는 코드이다. 아래 코드의 `SERVICE_NAME` 을 제외한 나머지는 모두 동일하다.

```
from opentelemetry import trace
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.resources import SERVICE_NAME, Resource
```

```
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import
OTLPSpanExporter

resource = Resource(attributes={
    SERVICE_NAME: "fastapi-server-1"
})
provider = TracerProvider(resource=resource)
processor = BatchSpanProcessor(OTLPSpanExporter(endpoint="
http://otel-poc-collector.otel.svc.cluster.local:4317"))
provider.add_span_processor(processor)
trace.set_tracer_provider(provider)
tracer = trace.get_tracer(__name__)
```

- 위 코드를 보면, 원하는 곳에 trace 관련 설정을 해주고, export할 endpoint로 OpenTelemetry Collector 서비스인 `otel-poc-collector.otel.svc.cluster.local:4317` 을 지정했다. 포트 번호가 4317이면 gRPC로 trace 정보를 보낸다는 것을 뜻한다.(HTTP의 경우 4318, python은 기본적으로 gRPC 사용

4.2 Trace가 시작되는 부분의 코드

- *Trace가 시작되는 부분의 코드*는 첫 번째로 요청을 처리하거나 실행하는 부분에 들어가는 코드이다.
  이 경우, 처음으로 클라이언트로부터 요청을 수신하는 `fastapi-server-1` 에 아래 코드가 들어간다.
- `LoggingInstrumentor` 관련 문서

```
# trace_id    .
#  custom format   .
LoggingInstrumentor().instrument(set_logging_format=True,
                                 logging_format=' trace_id=%
(otelTraceID)s span_id=%(otelSpanID)s - %(message)s')


@app.post("/items", status_code=201)
async def server_1_handler(item: Item, request: Request):
    #  span
    with tracer.start_as_current_span("server-1-handler") as span:
        #  span  attribute .
        span.set_attribute("item_name", item.name)
        span.set_attribute("item_price", item.price)
        span.set_attribute("ip", request.client.host)
        span.set_attribute("path", request.url.path)
        logging.warning(f"[SERVER-1] Received item: {item}")
        # fastapi-servser-2  call_server_2()
        result = call_server_2(item)
        if result.status_code != 200:
            logging.warning(f"[SERVER-1] Server 2 returned non-200
status code. (status_code: {result.status_code})")
            raise HTTPException(status_code=result.status_code,
detail=result.json())
        return JSONResponse(content=jsonable_encoder(result.json()),
status_code=201)
```

```
def call_server_2(item: Item):
    # call_server_2()   span .
    with tracer.start_as_current_span("server-1-call-server-2"):
        carrier = {}
        # carrier  (dict type) key-value  .
        TraceContextTextMapPropagator().inject(carrier)
        # carrier traceparent  key  value  header dict .
        # carrier   key-value  .
        # traceparernt: 00-e6a4a2a2baa6bdd7a9969dba87a80c3c-
5f169a366d4c2057-01
        header = {"traceparent": carrier["traceparent"]}
        logging.warning(f"[SERVER-1] Calling server 2 with item:
{item}")
        # header   fastapi-server-2  .
        response = requests.get(
            f"http://fastapi-server-2-svc.sangwoo-otel-poc.svc.
cluster.local:8000/items-name-check?name={item.name}&price={item.
price}",
            headers=header)
        #        2  sleep.
        time.sleep(2)
        logging.warning(f"[SERVER-1] Received response from server
2: {response}")
        return response
```

4.3 요청을 처리하는 부분의 trace 관련 코드

- 이제 `fastapi-server-1`으로부터 요청을 받아 요청을 처리하는 `fastapi-server-2` 의 코드를 보자.
`TracerProvider` 를 초기화하는 코드는 위와 동일하며, 대신 이번에는 `TraceContextTextMapPropagator.inject()` 가 아니라, 기존에 존재하는 trace의 내용을 빼오는 `extract()` 를 사용한다.

```
def get_trace_parent_header(request: Request):
    return request.headers.get("traceparent")


@app.get("/items-name-check", status_code=200)
async def server_2_handler(name: str = "", price: float = 0.0,
request: Request = None):
    # get_trace_parent_header()  reqeust header
    # key traceparent header value .
    traceparent = get_trace_parent_header(request)
    #  value carrier .
    carrier = {"traceparent": traceparent}
    # carrier   trace context  ctx .
    ctx = TraceContextTextMapPropagator().extract(carrier)
    # ctx span   context argument .
    with tracer.start_as_current_span("server-2-handler",
context=ctx) as span:
        span.set_attribute("item_name", name)
```

```
        span.set_attribute("ip", request.client.host)
        span.set_attribute("path", request.url.path)
        logging.warning(f"[SERVER-2] Received item name: {name},
price: {price}")
        return handle_request_from_server_1(item_name=name,
item_price=price)
```

4.4 `@tracer.start_as_current_span` decorator의 한계

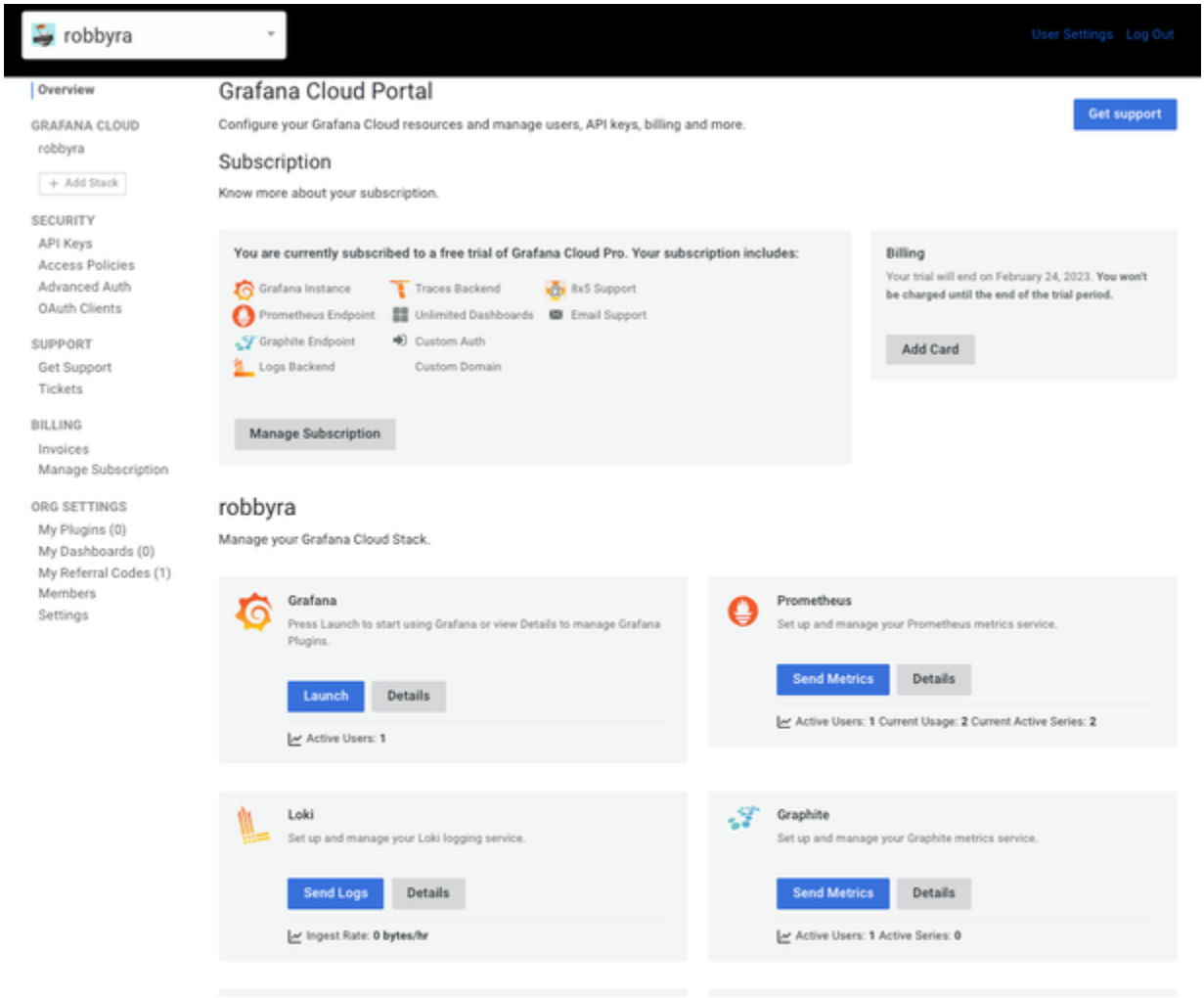
- 문서

- `@tracer.start_as_current_span` 라는 decorator를 함수 선언 위에 적용하면, `with tracer.start_as_current_span() as span` 구문 대신 span을 생성할 수 있다.
  하지만 아래의 단점들이 존재한다.
  - 1. `async def` 로 선언한 함수에는 적용할 수 없다. 적용하면 에러가 발생한다.
  - 2. span에 넣을 attribute들을 함수 실행 전에 미리 지정해야 한다. 예를 들어, 위에서 지정한 ip, path attribute는 `Request` 객체로부터 가져오는 값인데, decorator를 사용하면 이 값을 미리 알 수 없으므로 활용할 수 없다.
  - 3. Distributed tracing이 불가능하다. Attribute와 마찬가지로 distributed tracing을 적용하려면 span 생성 시 context를 만들어 전달해야 하는데, 이는 함수 몸체에서 가져올 수 있으므로 decorator 위치에서 가져와 지정할 수 없다.

Grafana Tempo 설정하기

1. Tempo Backend 생성하기

- 여기서는 클러스터에 Tempo를 설치하는 대신, Grafana Cloud가 제공하는 Tempo 서비스를 사용한다.
  Grafana Cloud에 가입 후 `grafana.com` 접속 → 우측 상단의 `My Account` 를 클릭하면 아래와 같이 Grafana Cloud Stack 관리 페이지가 나온다.



- 우측 아래의 `Tempo` 에서 `Send Traces` 를 클릭해 두 가지 세션 각각에 대해 API Key를 발급한다.
  - `Using Grafana With Tempo`: Grafana가 이 Tempo를 Data source로 활용할 때 사용할 credential 정의. 이 경우, 아래의 `Grafana` 에서 이 credential을 사용할 것이다.
  - `Sending Data to Tempo`: Tempo로 데이터를 보내는 credential 정의. 이 경우, OpenTelemetry Collector가 이 credential을 사용할 것이다.

2. OpenTelemetry Collector 설정 변경하기

- 이제 위에서 생성한 Tempo로 OpenTelemetry가 trace를 전송하도록 설정해야 한다.
  위의 `Sending Data to Tempo` 에서 발급받은 API key와 user id를 base64 encoding해서 OpenTelemetry Collector에 설정해야 하는데, 우선 아래 명령어로 base64 encoding된 값을 얻는다.

```
echo -n "userid:apikey" | base64
```

- 여기서 나온 output을 OpenTelemetry Collector를 위한 yaml 파일에 지정한다.

```
# opentelemetry-collector.yaml
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel-poc
  namespace: otel
spec:
  config: |
    receivers:
      #..
    processors:
      #..
    exporters:
      otlp:
        # endpoint Grafana Cloud  Tempo    .
        endpoint: tempo-us-central1.grafana.net:443
        headers:
          authorization: Basic < base64 encode >
    service:
      #..
```

Grafana 설치 및 설정하기

1. Grafana 설치

- 우선 Grafana를 위한 namespace를 생성한다.

```
kubectl create ns grafana
kubectl config set-context --current --namespace grafana
```

- 다음으로 아래 명령어로 Grafana를 설치한다.

```
helm install grafana grafana/grafana \
    --namespace grafana \
    --set persistence.storageClass="gp2" \
    --set persistence.enabled=true \
    --set adminPassword='' \
    --set service.type=NodePort
```

- 위 명령어로 Helm chart가 정상적으로 설치되면, 아래와 같이 하나의 Pod가 뜨게된다.

```
$ kubectl get po -n grafana
NAME                           READY   STATUS    RESTARTS   AGE
grafana-588759c845-k5xcb   1/1     Running   0          8m42s
```

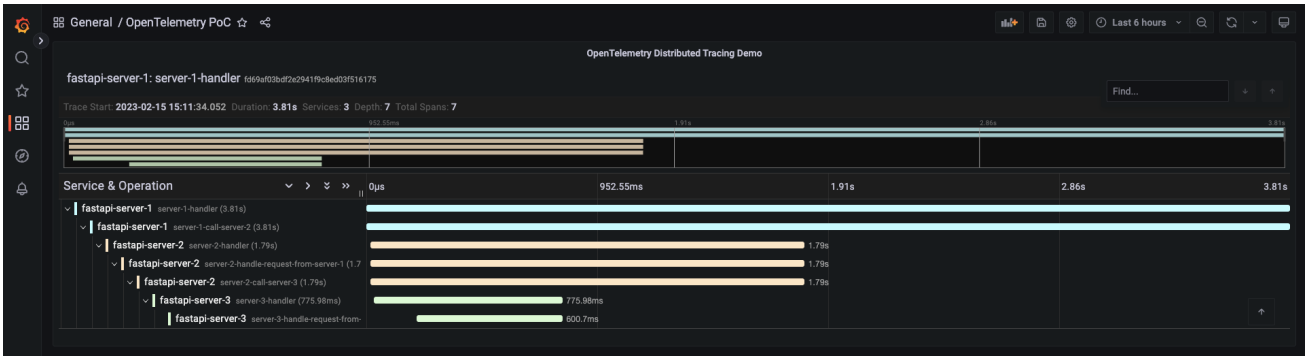
- Kubectl port-forward로 잘 돌아가는지 검증해본다.

```
kubectl port-forward grafana-588759c845-k5xcb 3000:3000
```

- `localhost:3000` 에 들어가면 아래와 같이 로그인 화면이 뜨는데, 아이디는 `admin`, 비밀번호는 `helm install` 시 설정한 값을 입력하면 로그인이 완료된다.

2. Grafana Datasource로 Tempo 설정하기

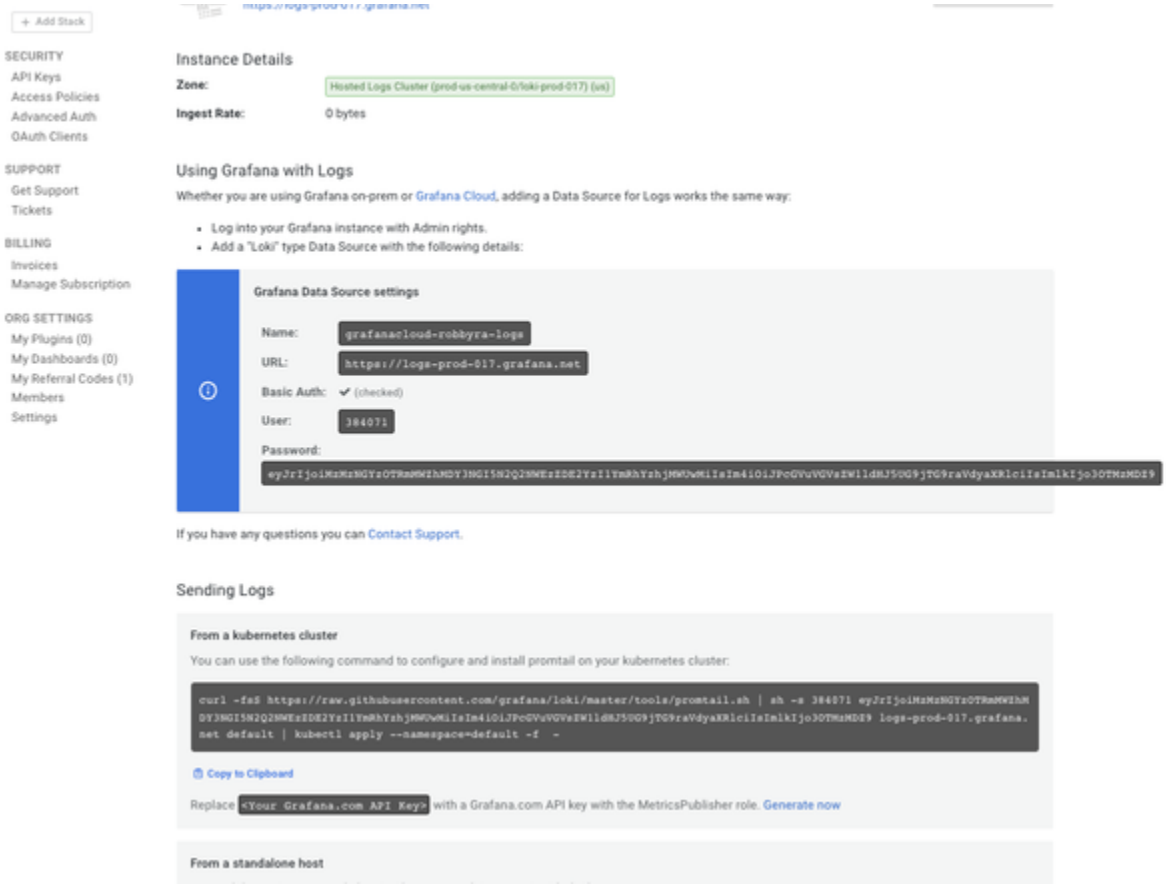
- `http://localhost:3000/datasources` 에 접속해 우측 상단에 `Add data source` 를 클릭하고, 위의 `Grafana Tempo` 단계에서 발급받은 credential(`Using Grafana With Tempo` 부분)을 입력한다.
- 입력 후 `Save & Test` 를 수행했을 때 정상적으로 연결되었다는 다이얼로그가 나타난다.
- 바로 `Explore` 버튼을 눌러 최근 trace를 찾아보면, 아래와 같이 distributed trace가 한눈에 나타나는 모습을 확인할 수 있다.



Loki 설정하기

1. Loki 생성하기

- Loki는 로그 데이터베이스로, Loki 또한 Grafana Cloud에서 생성해 사용할 것이다.
  위의 `Tempo backend` 단계와 마찬가지로 Grafana Cloud Stack 페이지로 이동 후, Loki를 생성한다.
  화면에 나오는 버튼을 눌러 API key를 발급받으면 아래 사진과 같다.



- 맨 아래의 `From a standalone host` 에 써있는 `clients.url` 을 뒤에서 사용할 것이다.

2. Grafana Data source로 Loki 추가하기

- 위의 사진에서 `Using Grafana with Logs` 를 참고해 Grafana 페이지에서 Loki data source를 추가한다.

3. Loki로 log 전달하기

- Loki로 log를 전달하는 주체는 Promtail 이며, 정상 동작을 위해서는 총 5가지의 K8S object를 생성해야 한다.
  - Daemonset: Promtail은 모든 node에 DaemonSet으로 작동해 pod 및 node가 생성하는 로그를 수집해 Loki로 전달한다.
  - ConfigMap: Promtail이 사용할 설정값들을 지종한다.
  - ServiceAccount: Promtail이 사용할 ServiceAccount를 지정한다.
  - ClusterRole: Promtail이 사용하는 ServiceAccount가 가질 권한을 지정한다.
  - ClusterRoleBinding: ClusterRole을 ServiceAccount에 binding하는 과정을 지정한다.
- 각 object를 위한 yaml 파일은 아래와 같다. 참고로, 여기서는 promtail 관련된 모든 설정을 `loki` 라는 namespace에 진행한다고 가정한다.
  - `ServiceAccount`

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: promtail-serviceaccount
  namespace: loki
```

  - `ClusterRole`

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: promtail-clusterrole
rules:
  - apiGroups: [""]
    resources:
      - nodes
      - services
      - pods
    verbs:
      - get
      - watch
      - list
```

- ClusterRoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: promtail-clusterrolebinding
subjects:
  - kind: ServiceAccount
    name: promtail-serviceaccount
    namespace: loki
roleRef:
  kind: ClusterRole
  name: promtail-clusterrole
  apiGroup: rbac.authorization.k8s.io
```

- ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: promtail-config
  namespace: loki
data:
  promtail.yaml: |
    server:
      http_listen_port: 9080
      grpc_listen_port: 0

    clients:
      # url  '1. Loki '  `Sending Logs`  user id
      #  API key    .
      #  url `From a standalone host`   yaml
```

```
      # clients.url   .
    - url: https://384071:
eyJrIjoiMzMzNGYzOTRmMWZhMDY3NGI5N2Q2NWEzZDE2YzI1YmRhYzhjMWUwMiIsIm
4iOiJPcGVuVGVsZW1ldHJ5UG9jTG9raVdyaXRlciIsImlkIjo3OTMzMDZ9@logs-
prod-017.grafana.net/loki/api/v1/push

    positions:
      filename: /tmp/positions.yaml
    target_config:
      sync_period: 10s
    scrape_configs:
    - job_name: pod-logs
      kubernetes_sd_configs:
        - role: pod
      pipeline_stages:
        - docker: {}
      relabel_configs:
        - source_labels:
            - __meta_kubernetes_pod_node_name
          target_label: __host__
        - action: labelmap
          regex: __meta_kubernetes_pod_label_(.+)
        - action: replace
          replacement: $1
          separator: /
          source_labels:
            - __meta_kubernetes_namespace
            - __meta_kubernetes_pod_name
          target_label: job
        - action: replace
          source_labels:
            - __meta_kubernetes_namespace
          target_label: namespace
        - action: replace
          source_labels:
            - __meta_kubernetes_pod_name
          target_label: pod
        - action: replace
          source_labels:
            - __meta_kubernetes_pod_container_name
          target_label: container
        - replacement: /var/log/pods/*$1/*.log
          separator: /
          source_labels:
            - __meta_kubernetes_pod_uid
            - __meta_kubernetes_pod_container_name
          target_label: __path__
```

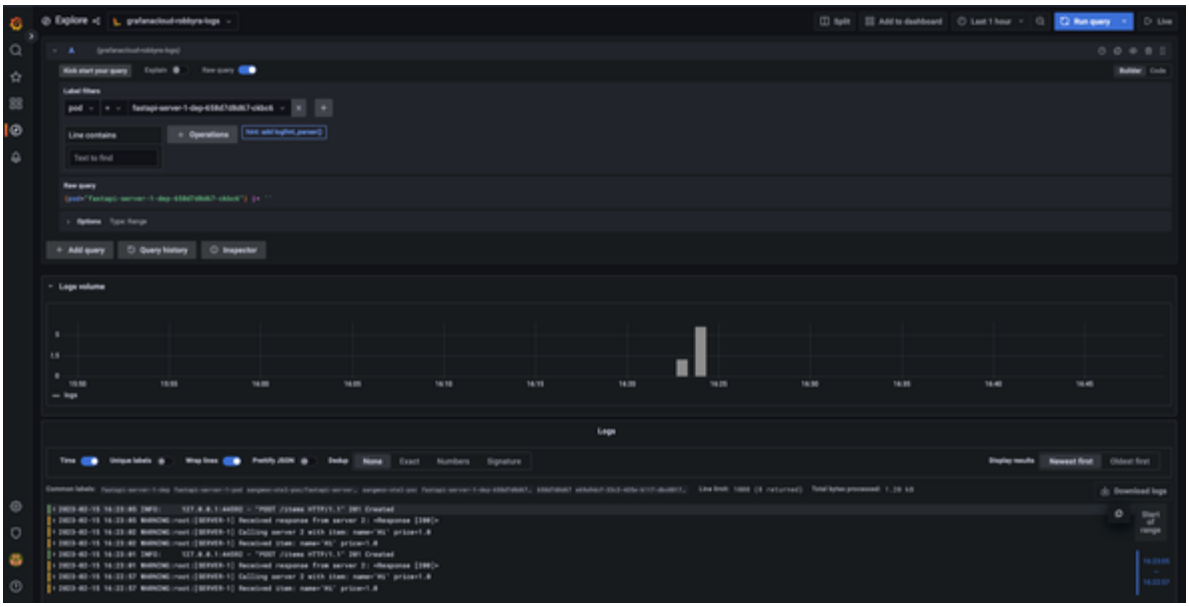
- DaemonSet

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: promtail-daemonset
  namespace: loki
spec:
  selector:
    matchLabels:
      name: promtail
  template:
    metadata:
      labels:
        name: promtail
    spec:
      serviceAccount: promtail-serviceaccount
      containers:
        - name: promtail-container
          image: grafana/promtail
          args:
            - -config.file=/etc/promtail/promtail.yaml
          env:
            - name: "HOSTNAME" # needed when using
kubernetes_sd_configs
              valueFrom:
                fieldRef:
                  fieldPath: "spec.nodeName"
          volumeMounts:
            - name: logs
              mountPath: /var/log
            - name: promtail-config
              mountPath: /etc/promtail
            - mountPath: /var/lib/docker/containers
              name: varlibdockercontainers
              readOnly: true
      volumes:
        - name: logs
          hostPath:
            path: /var/log
        - name: varlibdockercontainers
          hostPath:
            path: /var/lib/docker/containers
        - name: promtail-config
          configMap:
            name: promtail-config
```

- 이렇게 생성한 Promtail Daemonset이 잘 동작하는지는 아래 명령어를 통해 로그를 확인해 알 수 있다.

```
kubectl logs -n loki daemonsets/promtail-daemonset promtail-
container -f
```

- 로그를 수집한 node, pod에 대해서는 별도의 설정을 할 필요가 없으며, Grafana에 Loki를 data source로 잘 추가했다면, 아래와 같이 Grafana에서 로그 조회가 가능해진다.

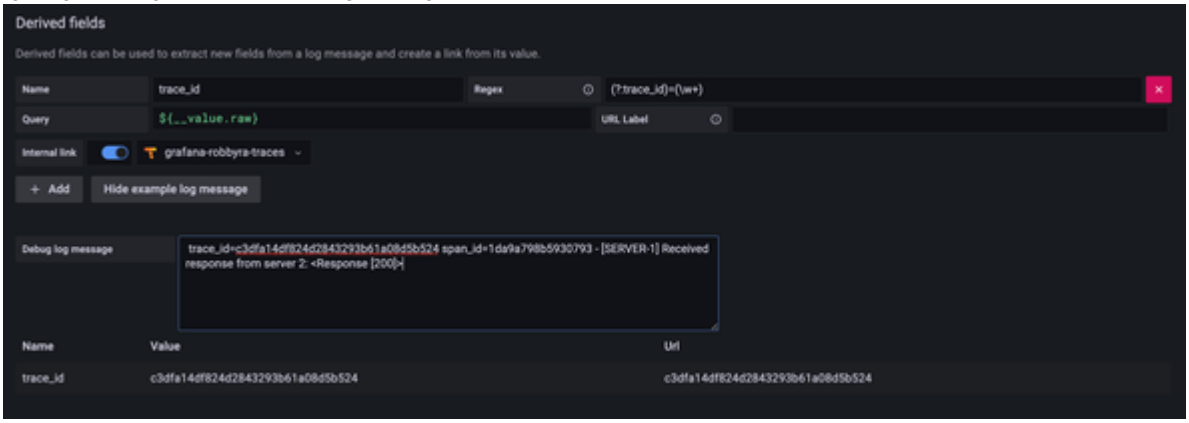


4. Loki와 Tempo 연동하기

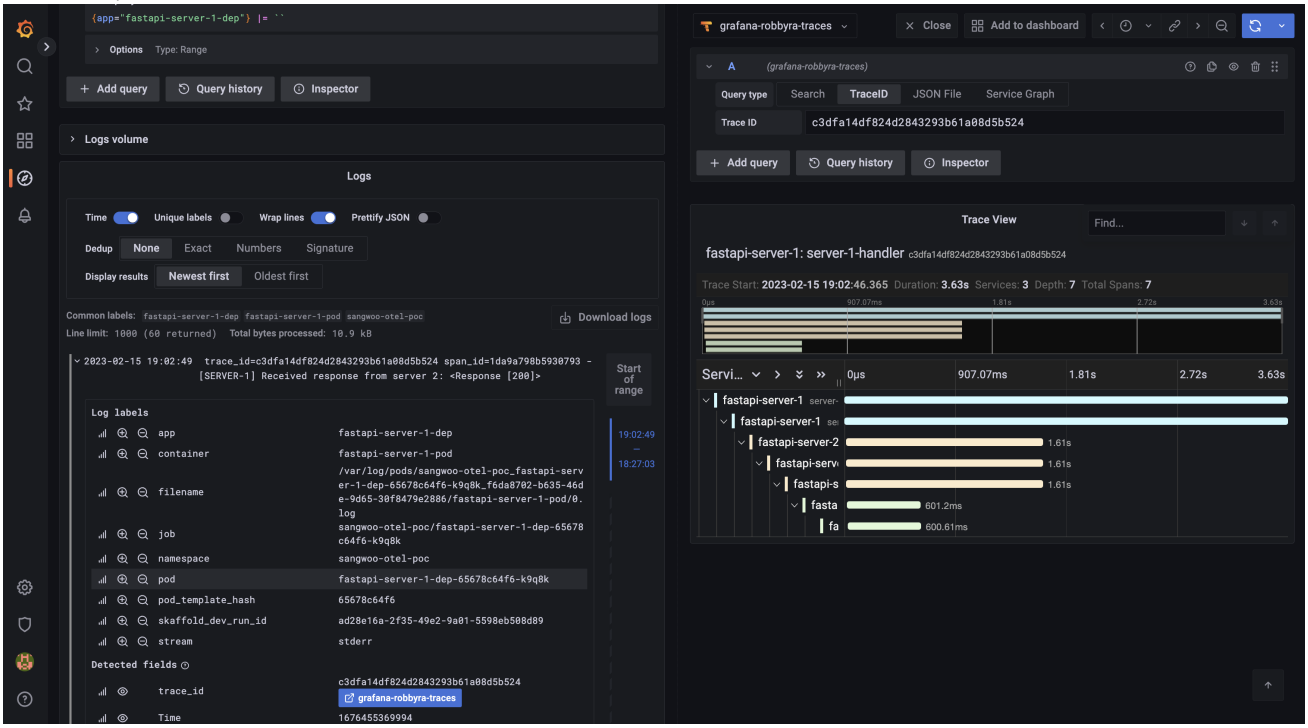
- Loki 2.0 에 등장한 Loki Derived Field를 활용하면, Loki에서 조회한 로그에서 TraceID를 알아내 바로 Tempo trace를 볼 수 있다.
- 먼저 Loki에 쌓이는 로그는 아래와 같은 형식이다.

```
trace_id=c3dfa14df824d2843293b61a08d5b524 span_id=1da9a798b5930793
- [SERVER-1] Received response from server 2: <Response [200]>
```

- 여기서 `trace_id` 를 빼내 Tempo로 연동하는 작업을 해줘야 하는데, 이때 Loki Derived Field를 활용하게 된다.
- Loki Data source 설정에 들어가 하단의 `Derived fields` 를 수정한다.
  - `Name`: `trace_id`
  - `Regex` : `(?:trace_id)=(\w+)`
  - `Query`: `${__value.raw}`
  - `Internal link` : 활성화 후 연결된 tempo로 선택
  - `Debug log message` : 예시로 하나의 log message를 가져와 입력하면, Derived field가 정규식이 원하는대로 추출되는지 확인할 수 있다.



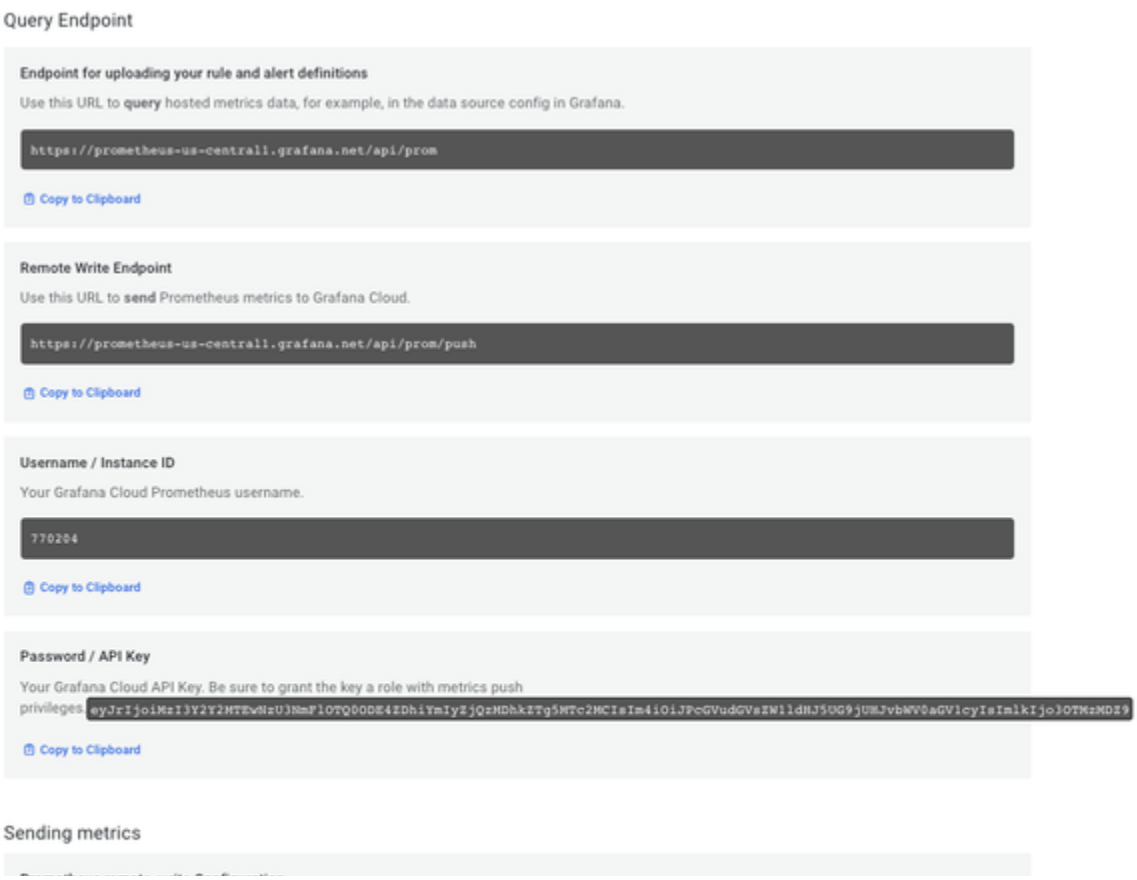
- 설정을 저장하고 다시 Loki로 가서 로그를 확인하면, 아래와 같이 `trace_id` 옆에 파란색 버튼이 보이며, 해당 버튼을 클릭하면 해당 `trace_id`를 들고 Tempo에서 조회한 trace 결과가 우측에 나타나게 된다.



Prometheus 설치하기

1. Prometheus Backend 생성하기

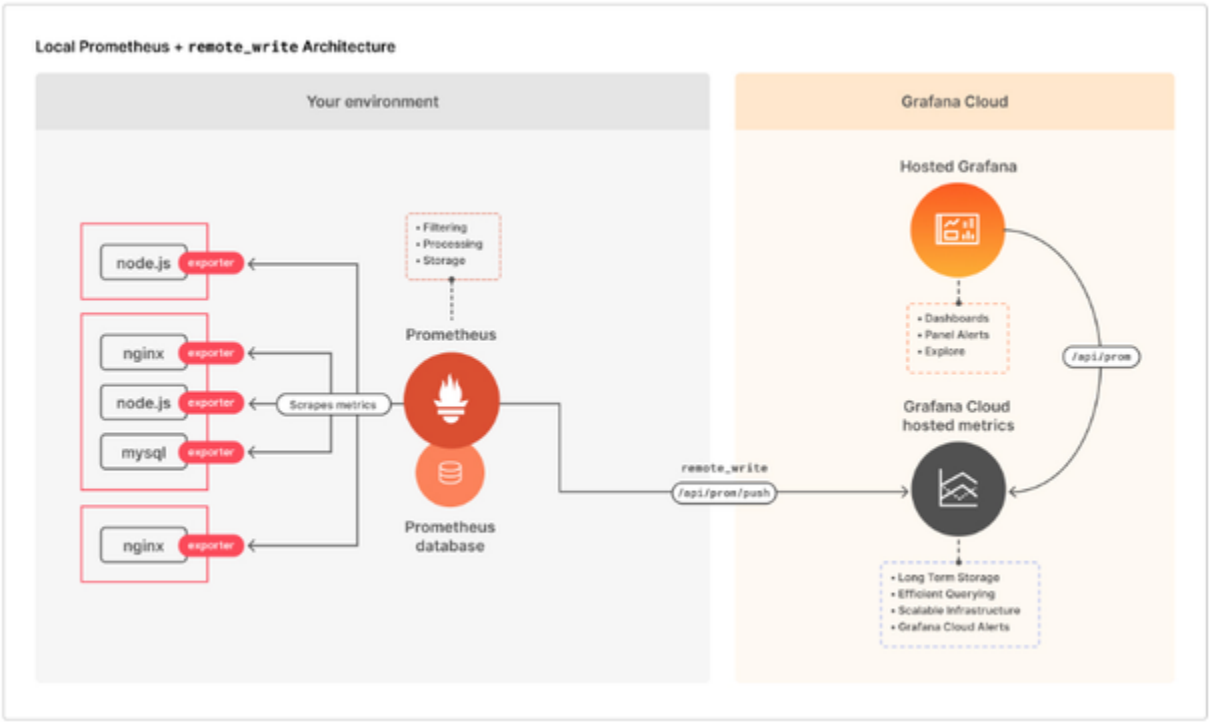
- Tempo, Loki backend를 생성했을 때와 마찬가지로 Prometheus도 Grafana Cloud에서 생성해보자.
아래와 같이 해당 페이지에서 Grafana Cloud의 Prometheus에 접근하기 위해 API Key를 발급받는다.



- 하단에 있는 `Using a self-hosted Grafana instance with Grafana Cloud Metrics` 부분을 나중에 사용할 것이다.

2. Kubernetes cluster에 Prometheus 설치하기

- 이번에는 OpenTelemetry Collector가 수집한 정보를 저장할 시계열 데이터베이스인 Prometheus를 설치한다.
  이렇게 클러스터에 설치된 Prometheus에 먼저 데이터가 저장되고, 해당 데이터가 정제되어 Grafana Cloud에 있는 Prometheus로 전달된다. 아키텍쳐는 아래와 같다.



- 위 아키텍쳐 사진에 대한 문서
- PoC 목적이기에 아래 명령어로 community version을 설치한다.

```
# namespace
kubectl create ns prometheus
kubectl config set-context --current --namespace prometheus

# prometheus  helm chart
helm repo add prometheus-community https://prometheus-community.
github.io/helm-charts
helm repo update
```

- 다음으로 helm value 파일을 수정할 것인데, 여기서는 alert-manager와 node-exporter가 필요하지 않으므로 설치를 제외한다. values file 에서 `alertmanager.enabled` 와 `prometheus-node-exporter.enabled` 를 false로 지정한다. 그리고 위의 Grafana Cloud Prometheus에 데이터를 보내기 위한 설정도 지정한다.

```
# prometeheus-values.yaml
#..

prometheus-node-exporter:
  enabled: false
  #..
alertmanager:
  enabled: false
  #..
remoteWrite:
    # url: Grafana Cloud  Prometheus push URL
  - url: https://prometheus-us-central1.grafana.net/api/prom/push
    basic_auth:
      username: <user name,  6>
      password: < API Key>
```

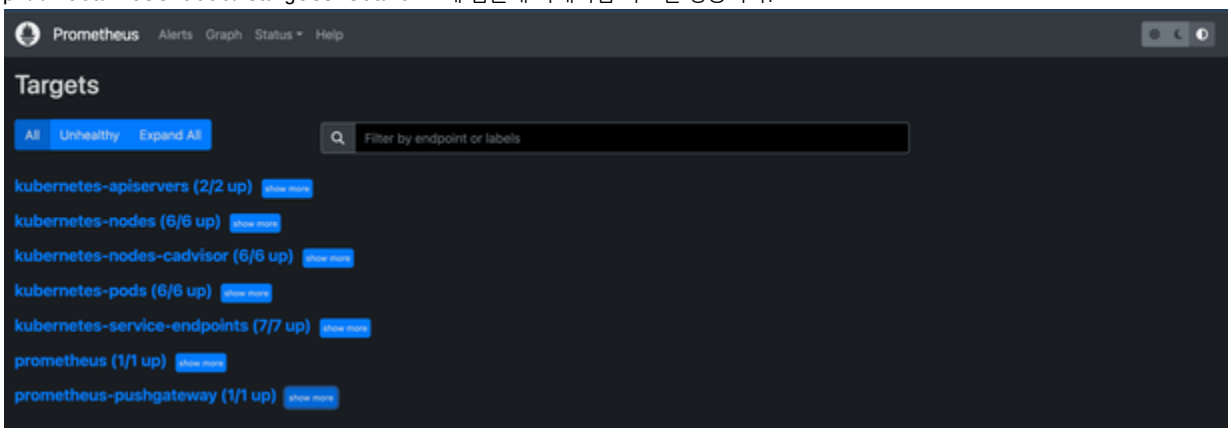
- 위 파일을 `prometheus-values.yaml` 로 저장했다고 가정하고, 다음 명령어로 설치한다.

```
helm upgrade -f prometheus-values.yaml \
  -i prometheus prometheus-community/prometheus \
  --namespace prometheus \
  --set alertmanager.persistentVolume.storageClass="gp2",server.
persistentVolume.storageClass="gp2"
```

- 설치가 잘 되었는지 확인하기 위해 prometheus-server를 port-forward 해본다.

```
kubectl port-forward deploy/prometheus-server 9090:9090
```

- `http://localhost:9090/targets?search=` 에 접근해 아래처럼 나오면 성공이다.

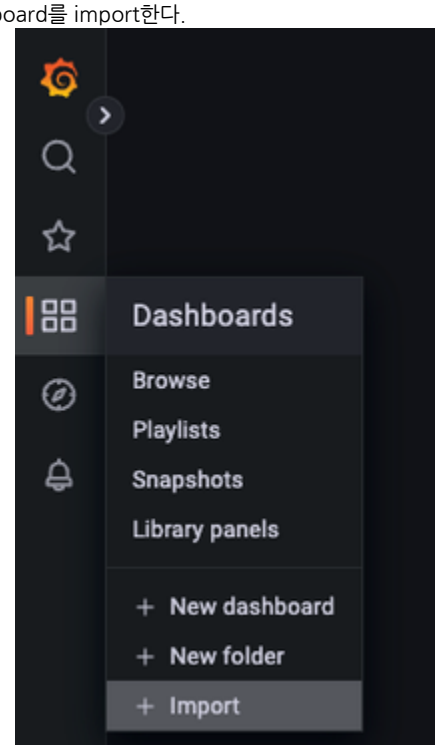


3. Grafana에 Grafana Cloud Prometheus 연결하기

- Doodle 환경의 Grafana에 접속해 data source를 추가한다.
  Prometheus를 선택하고, 입력 값으로는 Grafana Cloud에 Prometheus를 생성할 때 하단에 있던 `Using a self-hosted Grafana instance with Grafana Cloud Metrics` 부분의 정보를 입력하면 Data source가 정상적으로 추가된다.

4. Prometheus 동작을 Dashboard로 테스트하기

- Grafana에 접속해 아래 사진처럼 새로운 dashboard를 import한다.



- 다음으로 아래 사진과 같이 입력창에 3119 를 입력하고 Load를 클릭한다.
  그러면 dashboard가 나오게 되는데, 아래 사진처럼 모든 정보가 제대로 표출되어야 정상적으로 Prometheus가 연동된 것이다.