- acend gmbh

Labs

- acend gmbh

1. Setting up Prometheus

In this first section we are going to set up our first parts of the Prometheus stack. Each trainee will have their own stack installed and configured.

Working mode (GitOps)

During the labs you will deploy and update several resources on your Kubernetes environment. ArgoCD will be your primary interface to interact with the cluster and will simplify the GitOps process for you.

> **Note**
>
> Argo CD is a part of the Argo Project and affiliated under the Cloud Native Computing Foundation (CNCF) . The project is just under three years old, completely open source, and primarily implemented in Go.
>
> As the name suggests, Argo CD takes care of the continuous delivery aspect of CI/CD. The core of Argo CD consists of a Kubernetes controller, which continuously compares the live-state with the desired-state. The live-state is tapped from the Kubernetes API, and the desired-state is persisted in the form of manifests in YAML or JSON in a Git repository. Argo CD helps to point out deviations of the states, to display the deviations or to autonomously restore the desired state.

The configuration and deployments needed for you are already in a git repository. Navigate to your Gitea and look for a project called 'prometheus-training-lab-setup'. The repository consists of two Helm Charts you will further use in this lab. In this first section we will no setup your Prometheus instance step by step.

We're going to use two main Namespaces for the lab

- `<user>` - where the user workload (Demo application, Webshell) is deployed
- `<user>-monitoring` - where we deploy our monitoring stack to

How do metrics end up in Prometheus?

Since Prometheus is a pull-based monitoring system, the Prometheus server maintains a set of **targets** to scrape. This set can be configured using the `scrape_configs` option in the Prometheus configuration file. The `scrape_configs` consist of a list of jobs defining the targets as well as additional parameters (path, port, authentication, etc.) which are required to scrape these targets. As we will be using the Prometheus Operator on Kubernetes, we will never actually touch this configuration file by ourselves. Instead, we rely on the abstractions provided by the Operator, which we will look at closer in the next section.

There are two basic types of targets that we can add to our Prometheus server:

Static targets

In this case, we define one or more targets statically. In order to make changes to the list, you need to change the configuration file. As the name implies, this way of defining targets is inflexible and not suited to monitor workloads inside of Kubernetes as these are highly dynamic.

Dynamic configuration

Besides the static target configuration, Prometheus provides many ways to dynamically add/remove targets. There are builtin service discovery mechanisms for cloud providers such as AWS, GCP, Hetzner, and many more. In addition, there are more versatile discovery mechanisms available which allow you to implement Prometheus in your environment (e.g. DNS service discovery or file service discovery). Most

importantly, the Prometheus Operator makes it very easy to let Prometheus discover targets dynamically using the Kubernetes API.

Prometheus Operator

The Prometheus Operator is the preferred way of running Prometheus inside of a Kubernetes Cluster. In the following labs you will get to know its CustomResources in more detail, which are the following:

- Prometheus : Manage the Prometheus instances
- Alertmanager : Manage the Alertmanager instances
- ServiceMonitor : Generate Kubernetes service discovery scrape configuration based on Kubernetes service definitions
- PrometheusRule : Manage the Prometheus rules of your Prometheus
- AlertmanagerConfig : Add additional receivers and routes to your existing Alertmanager configuration
- PodMonitor : Generate Kubernetes service discovery scrape configuration based on Kubernetes pod definitions
- Probe : Manage Prometheus blackbox exporter targets
- ThanosRuler : Manage Thanos rulers

Service Discovery

When configuring Prometheus to scrape metrics from containers deployed in a Kubernetes Cluster it doesn't really make sense to configure every single target (Pod) manually. That would be far too static and wouldn't really work in a highly dynamic environment. A container platform is too dynamic. Pods can be scaled, the names are random and so on.

In fact, we tightly integrate Prometheus with Kubernetes and let Prometheus discover the targets, which need to be scraped, automatically via the Kubernetes API.

The tight integration between Prometheus and Kubernetes can be configured with the Kubernetes Service Discovery Config .

The way we instruct Prometheus to scrape metrics from an application running as a Pod is by creating a `ServiceMonitor` .

ServiceMonitors are Kubernetes custom resources, which look like this:

```
# just an example
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    app.kubernetes.io/name: example-web-python
  name: example-web-python-monitor
spec:
  endpoints:
  - interval: 30s
    port: http
    scheme: http
    path: /metrics
  selector:
    matchLabels:
      prometheus-monitoring: 'true'
```

How does it work

The Prometheus Operator watches namespaces for `ServiceMonitor` custom resources. It then updates the

Service Discovery configuration of the Prometheus server(s) accordingly.

The selector part in the `ServiceMonitor` defines which Kubernetes Services will be scraped. Here we are selecting the correct service by defining a selector on the label `prometheus-monitoring: 'true'` .

```
# servicemonitor.yaml
...
  selector:
    matchLabels:
      prometheus-monitoring: 'true'
...
```

The corresponding `Service` needs to have this label set:

```
apiVersion: v1
kind: Service
metadata:
  name: example-web-python
  labels:
    prometheus-monitoring: 'true'
...
```

The Prometheus Operator then determines all `Endpoints` (which are basically the IPs of the Pods) that belong to this `Service` using the Kubernetes API. The `Endpoints` are then dynamically added as targets to the Prometheus server(s).

The `spec` section in the ServiceMonitor resource allows further configuration on how to scrape the targets. In our case Prometheus will scrape:

- Every 30 seconds
- Look for a port with the name `http` (this must match the name in the `Service` resource)
- Scrape metrics from the path `/metrics` using `http`

Best practices

Use the common k8s labels https://kubernetes.io/docs/concepts/overview/working-with-objects/common-labels/

If possible, reduce the number of different `ServiceMonitors` for an application and thereby reduce the overall complexity.

- Use the same `matchLabels` on different `Services` for your application (e.g. Frontend Service, Backend Service, Database Service)
- Also make sure the ports of different `Services` have the same name
- Expose your metrics under the same path

1.1 Tasks: Setup

Task 1.1.1: Getting Started - Web IDE

The first thing we're going to do is to explore our lab environment and get in touch with the different components.

The namespace with the name corresponding to your username is going to be used for all the hands-on labs. And you will be using the folling tools during the lab

- Gitea Git Server: https://gitea.training.cluster.acend.ch - Login with `<user>` and the provided password
- Argo CD Server: https://argocd.training.cluster.acend.ch - Login via Gitea
- git - Login with `<user>` and the provided password
- kubectl - already logged in

> **Note**
> The URL and Credentials to the Web IDE will provided by the teacher. Use Chrome for the best experience.

Once you're successfully logged into the web IDE open a new Terminal by hitting `CTRL + SHIFT + ¨` or clicking the Menu button –> Terminal –> new Terminal and check the installed kubectlversion by executing the following command:

```
kubectl version --output=yaml
```

The Web IDE Pod consists of the following tools:

- oc
- kubectl
- kustomize
- helm
- kubectx
- kubens
- tekton cli
- argocd

The files in the home directory under `/home/project` are stored in a persistence volume, so please make sure to store all your persistence data in this directory.

Task 1.1.1.1: Local Workspace Directory

During the lab, you'll be using local files (eg. YAML resources) which will be applied in your lab project.

Create a new folder for your `<workspace>` in your Web IDE (for example `prometheus-training` under `/home/project/prometheus-training`). Either you can create it with `right-mouse-click -> New Folder` or in the Web IDE terminal

```
mkdir prometheus-training && cd prometheus-training
```

In the Web IDE we set the `USER` environment variable to your personal `<username>`.

Verify that with the following command:

```
echo $USER
```

The `USER` variable will be used as part of the commands to make the lab experience more comfortable for you.

Clone the forked repository to your local workspace:

```
git clone https://$USER@gitea.training.cluster.acend.ch/$USER/prometheus-training-lab-setup.git
```

Change the working directory to the cloned git repository:

```
cd prometheus-training-lab-setup
```

For convenience let's configure the git client:

```
git config user.name "$USER"
git config user.email "$USER@gitea.training.cluster.acend.ch"
```

And we also want git to store our Password for two days so that we don't need to login every single time we push something.

```
git config credential.helper 'cache --timeout=172800'
```

Then use the following command to verify whether the git config for username and email were correctly added:

```
git config --local --list
```

Explore the cloned repository.

Task 1.1.2: Install Prometheus

As explained in the previous section, we're going to use ArgoCD to deploy our Kubernetes resources for our lab, therefore the first thing we do is to create the ArgoCD configuration.

Configure ArgoCD correctly

In order for ArgoCD to monitor and synchronize your applications correctly, we need first to make ArgoCD aware of the applications to be deployed. ArgoCD application resources (`Application`) create a logical connection for ArgoCD between a git repository and a kubernetes namespace. In your cloned repository you will have already have two ArgoCD applications prepared for you (`apps/user-demo.yaml` and `apps/user-prom-stack.yaml`). The first application `apps/user-demo.yaml` will synchronize and deploy your user workload examples and the second application `apps/user-prom-stack.yaml` will be used to deploy the prometheus infrastructure resources.

Open these two files (`apps/user-demo.yaml` and `apps/user-prom-stack.yaml`) in your editor and replace all the `<user>` placeholders with your correct username. Safe the files and push them to your git repository:

```
git add .
git commit -m "Replace userid"
git push
```

> **Note**
> After `git push`, a popup will appear in the top section of your WebIDE. Enter your password and confirm.

So far, nothing has happened yet. We create a third ArgoCD application to synchronize the other two applications. In ArgoCD we call this the app-of-apps pattern . In order to do that, we create a file `user-app-of-apps.yaml` in the root directory of your git repository. Add the following content and replace all `<user>` placeholder with your user:

```
---
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: <user>-app-of-apps
  namespace: argocd
spec:
  destination:
    namespace: argocd
    server: https://kubernetes.default.svc
  project: default
  source:
    repoURL: 'https://gitea.training.cluster.acend.ch/<user>/prometheus-training-lab-setup'
    path: apps/
    targetRevision: main
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    syncOptions:
      - Replace=true
```

```
kubectl -n argocd create -f user-app-of-apps.yaml
```

Deploy Prometheus

As mentioned our Prometheus Stack will be deployed in the `<user>-monitoring` namespace. The instance itself will be deployed via ArgoCD with the defined application in your git repository. In order to create our prometheus instance we need to alter the configuration of the Helm chart in `charts/user-monitoring` . Open the `charts/user-monitoring/values.yaml` file in your editor and change the `<user>` placeholder correctly and update the value of `prometheus.enabled` to `true` .

```
user: <user> # Replace me
# prometheus
prometheus:
  enabled: true
# thanos-query
query:
  enabled: false
# grafana
grafana:
  enabled: false
# blackboxexporter
blackboxexporter:
  enabled: false
# pushgateway
pushgateway:
  enabled: false
# alertmanager
alertmanager:
  enabled: false
# thanos-ruler
ruler:
  enabled: false
```

If you are curious to see what ArgoCD will do for you, you can render the helmchart locally:

```
helm template charts/user-monitoring/.
```

When you are confident that the changes are done correctly, simply commit and push the files to your git repository and from there ArgoCD will take over and start synchronizing.

```
git add .
git commit -m "Enable Prometheus"
git push
```

Head over to the ArgoCD UI and verify that the synchronization process of your application is synced and healthy. As soon as your application is healthy and synced (green status on top) you are good to go and have your prometheus instance ready.

When all has finished syncing, you can inspect your prometheus installation in your namespace:

```
kubectl -n $USER-monitoring get prometheus prometheus -oyaml
```

Configure Prometheus

As mentioned in the introduction, configuring a Prometheus on Kubernetes can be done using the Prometheus Operator. We basically need to specify a Prometheus custome resouce and the Operator will do its work. In the prometheus custom resource's spec block you can find various configuration options:

```
spec:
  enableAdminAPI: true
  evaluationInterval: 30s
  externalLabels:
    monitoring: <user>
  podMonitorNamespaceSelector:
    matchLabels:
      user: <user>
  podMonitorSelector: {}
  portName: web
  probeNamespaceSelector:
    matchLabels:
      user: <user>
  probeSelector: {}
  resources:
    requests:
      memory: 400Mi
  scrapeInterval: 60s
  serviceAccountName: prometheus-<user>
  serviceMonitorNamespaceSelector:
    matchLabels:
      user: <user>
  serviceMonitorSelector: {}
```

So far we do not have configured anything special yet. We instruct prometheus to automatically discover Probes, PodMonitors and ServiceMonitors in all namespaces with the label `user` matching your username.

> **Note**
> We will learn more about other configuration options (`evaluation_interval`) later in this training.

Check Prometheus

Is your prometheus running? Use your browser to navigate to https://<user>-prometheus.training.cluster.acend.ch . You should now see the Prometheus web UI.

Task 1.1.3: Deploy example application

In the next section we are going to deploy our own application and try to monitor their metrics. As stated earlier the folder `user-demo` is automatically watched and synchronized by ArgoCD from your git repository. Deploy the Acend example Python application, which provides application metrics at `/metrics` by creating the following file (`user-demo/deployment.yaml`) in your repo:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: example-web-python
  name: example-web-python
spec:
  replicas: 1
  selector:
    matchLabels:
      app: example-web-python
  template:
    metadata:
      labels:
        app: example-web-python
    spec:
      containers:
      - image: quay.io/acend/example-web-python
        name: example-web-python
        resources:
          requests:
            memory: "32Mi"
            cpu: "10m"
          limits:
            memory: "128Mi"
            cpu: "100m"
```

We can simply deploy the application by pushing the resource into our git repository in the `user-demo` folder:

```
git add .
git commit -m "Deploy Demo App"
git push
```

Use the following command to verify whether pod `example-web-python` is Ready and Running in your `<user>` namespace. (use CTRL C to exit the command) Or you can also check in the ArgoCD UI .

```
kubectl -n $USER get pod -w
```

To access the deployed resources we also need to create a Service for the new application. Create a file (`user-demo/service.yaml`) with the following content:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: example-web-python
    prometheus-monitoring: 'true'
  name: example-web-python
spec:
  ports:
    - name: http
      port: 5000
      protocol: TCP
      targetPort: 5000
  selector:
    app: example-web-python
  type: ClusterIP
```

We can add and push the resource to our git repository to have it synchronized to our namespace:

```
git add .
git commit -m "Deploy Demo Service"
git push
```

This created a so-called Kubernetes Service which allows communication via an internal network abstraction.

```
kubectl -n $USER get services
```

Task 1.1.4: Create a ServiceMonitor

So far we did not have any interaction with Prometheus at all. Kubernetes-based Prometheus installation use PodMonitors and ServiceMonitors as a service-discovery mechanism. PodMonitors and ServiceMonitors select via labels pods / services for prometheus to scrape at a given endpoint. Check whether the application metrics are actually exposed by opening a shell within the container and curling the metrics endpoint.

```
# get the pod name
kubectl -n $USER get pod
```

```
# exec curl within the pod
kubectl -n $USER exec -it <pod-name> -- curl http://localhost:5000/metrics
```

Should result in something like:

```
# HELP python_gc_objects_collected_total Objects collected during gc
# TYPE python_gc_objects_collected_total counter
python_gc_objects_collected_total{generation="0"} 541.0
python_gc_objects_collected_total{generation="1"} 344.0
python_gc_objects_collected_total{generation="2"} 15.0
...
```

Since our newly deployed application now exposes metrics, the next thing we need to do, is to tell our Prometheus server to scrape metrics from the Kubernetes deployment. In a highly dynamic environment like Kubernetes this is done with so called Service Discovery.

Let us create a ServiceMonitor for the example application, which will configure Prometheus to scrape metrics from the example-web-python application every 30 seconds.

For this to work, you need to ensure:

- The example-web-python Service is labeled correctly and matches the labels you've defined in your ServiceMonitor.
- The port name in your ServiceMonitor configuration matches the port name in the Service definition.
  - hint: check with `kubectl -n $USER get service example-web-python -o yaml`

- Verify the target in the Prometheus user interface.

Create the following ServiceMonitor `user-demo/servicemonitor.yaml` in your git repository:

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    app.kubernetes.io/name: example-web-python
  name: example-web-python-monitor
spec:
  endpoints:
    - interval: 60s
      port: http
      scheme: http
      path: /metrics
  selector:
    matchLabels:
      prometheus-monitoring: 'true'
```

Then commit and push the resource again for ArgoCD to synchronize.

```
git add .
git commit -m "Add ServiceMonitor"
git push
```

In the Prometheus UI we can check our targets to be scaped. So far no target should appear from your demo application and ServiceMonitor we just deployed. This is happening because we tell Prometheus to look for ServiceMonitors in our `<user>` namespace, but it is not yet allowed by the Kubernetes API to see resources in this namespace. Therefore we need to add RBAC rules to let Prometheus see and scrape these resources.

We do this again by creating a Role and RoleBinding for the Prometheus' ServiceAccount. Create the following two files in your git repository:

`user-demo/role.yaml` Replace the `<user>` placeholder with your user:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: prometheus-<user>
rules:
- apiGroups: [""]
  resources:
  - services
  - endpoints
  - pods
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources:
  - configmaps
  verbs: ["get"]
- apiGroups:
  - networking.k8s.io
  resources:
  - ingresses
  verbs: ["get", "list", "watch"]
```

`user-demo/rolebinding.yaml` Replace the `<user>` placeholder with your user:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: prometheus-<user>
subjects:
- kind: ServiceAccount
  name: prometheus-<user>
  namespace: <user>-monitoring
```

Then again add and push these files to your git repository and let ArgoCD synchronize them for you.

```
git add .
git commit -m "Add role and rolebinding"
git push
```

> **Note**
> This part usually is done on a cluster level, but is needed for our current Lab setup.

Verify that the target gets scraped in the Prometheus user interface . Target name: `serviceMonitor/<user>/example-web-python-monitor/0 (1/1 up)` (it may take up to a minute for Prometheus to load the new configuration and scrape the metrics).

Task 1.1.5: Your first PromQL Query

Switch back to the `Graph` Tab in the Prometheus UI and enter the following query into the input field:

```
python_info
```

hit enter and explore the result should look similar to

```
python_info{endpoint="http", implementation="CPython", instance="10.244.18.95:5000", job="example-web-python", major="3
", minor="11", namespace="<user>", patchlevel="5", pod="example-web-python-7c8b9984d4-z9b92", service="example-web-pyth
on", version="3.11.5"}  1
```

- acend gmbh

2. Metrics

In this lab you are going to learn about the Prometheus exposition format and how metrics and their values are represented withing the Prometheus ecosystem.

Prometheus exposition format

Prometheus consumes metrics in Prometheus text-based exposition format and plans to adopt the OpenMetrics standard: https://prometheus.io/docs/introduction/roadmap/#adopt-openmetrics .

Optionally check Prometheus Exposition Format for a more detailed explanation of the format.

All metrics within Prometheus are scraped, stored and queried in the following format:

```
# HELP <metric name> <info>
# TYPE <metric name> <metric type>
<metric name>{<label name>=<label value>, ...} <sample value>
```

The Prometheus server exposes and collects its own metrics too. You can easily explore the metrics with your browser under (http://LOCALHOST:9090/metrics).

Metrics similar to the following will be shown:

```
...
# HELP prometheus_tsdb_head_samples_appended_total Total number of appended samples.
# TYPE prometheus_tsdb_head_samples_appended_total counter
prometheus_tsdb_head_samples_appended_total 463
# HELP prometheus_tsdb_head_series Total number of series in the head block.
# TYPE prometheus_tsdb_head_series gauge
prometheus_tsdb_head_series 463
...
```

Metric Types

There are 4 different metric types in Prometheus

- Counter, (Basic use cases, always goes up)
- Gauge, (Basic use cases, can go up and down)
- Histogram, (Advanced use cases)
- Summary, (Advanced use cases)

For now we focus on Counter and Gauge.

Find additional information in the official Prometheus Metric Types docs.

Special labels

As you have already seen in several examples, a Prometheus metric is defined by one or more labels with the corresponding values. Two of those labels are special, because the Prometheus server will automatically generate them for every metric:

- instance

- acend gmbh

The instance label describes the endpoint where Prometheus scraped the metric. This can be any application or exporter. In addition to the IP address or hostname, this label usually also contains the port number. Example: `10.0.0.25:9100` .

- job

  This label contains the name of the scrape job as configured in the Prometheus configuration file. All instances configured in the same scrape job will share the same job label. In a Kubernetes environment this relates to the `Service` -Name.

> **Note**
> Prometheus will append these labels dynamically before sample ingestion. Therefore you will not see these labels if you query the metrics endpoint directly (e.g. by using `curl`).

Let's take a look at the following `ServiceMonitor` (example, no need to apply this to the cluster):

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    app.kubernetes.io/name: example-web-python
  name: example-web-python-monitor
spec:
  endpoints:
  - interval: 30s
    port: http
    scheme: http
    path: /metrics
  selector:
    matchLabels:
      name: example-web-python-monitor
```

In the example above we instructed Prometheus to scrape all Pods that are matched by the `Service` named `example-web-python-monitor` . After ingestion into Prometheus, every metric scraped by this job will have the label: `job="example-web-python-monitor"` . In addition, metrics scraped by this job from the Pod with IP `10.0.0.25` will have the label `instance="10.0.0.25:80"`

Node Exporter

The tasks of this chapter will all be based on metrics that are provided by the `node_exporter` . An exporter is generally used to expose metrics from an application or system that would otherwise not expose metrics natively in the Prometheus exposition format. You will learn more about other exporters in the lab *4* .

In case of the `node_exporter` , the system we're interested in are Linux machines. It gathers the necessary information from different files and folders (e.g. `/proc/net/arp` , `/proc/sys/fs/file-nr` , etc.) and therefore is able to expose information about common metrics like CPU, Memory, Disk, Network, etc., which makes it very useful for expanding Prometheus' monitoring capabilities into the infrastructure world.

On our Lab Setup there are several `node_exporters` deployed. On each of our Kuberentes Nodes runs a `node_exporter` - Container deployed in a daemonset.

2.1 Tasks: Thanos Querier

To have centralized access to the metrics of both the shared infrastructure Prometheus instance as well as your own userwokload Prometheus instance, we will install the Thanos Querier in this lab. This will give us a

central point of view for the following prometheus instances:

- Infrastructure Prometheus (K8S Metrics, Node Metrics, ...) in the global `monitoring` namespace
- Userworkload Prometheus (the stack we can use for our application metrics) in the `<user>-monitoring` namespace

Task 2.1.1: Install Thanos Querier

To install the Thanos Querier, change `query.enabled` to `true` in the `values.yaml` of your user-monitoring Helm release. ArgoCD will automagically install all needed components in your namespace.

`charts/user-monitoring/values.yaml` :

```
user: <user> # Replace me
# prometheus
prometheus:
  enabled: true
# thanos-query
query:
  enabled: true
# grafana
grafana:
  enabled: false
# blackboxexporter
blackboxexporter:
  enabled: false
# pushgateway
pushgateway:
  enabled: false
# alertmanager
alertmanager:
  enabled: false
# thanos-ruler
ruler:
  enabled: false
```

Commit and push the changes to your git repository and let ArgoCD synchronize the changes.

Task 2.1.1: Verify the installation

Make sure that the Thanos Querier is running and ready.

```
kubectl -n $USER-monitoring get pods -l app.kubernetes.io/name=thanos-query
```

Have a look at the Thanos Query deployment in detail to see the relevant configuration parameters.

```
kubectl -n $USER-monitoring get deploy -l app.kubernetes.io/name=thanos-query -o yaml
```

Open the web UI to check whether the querier is up and running and accessible.

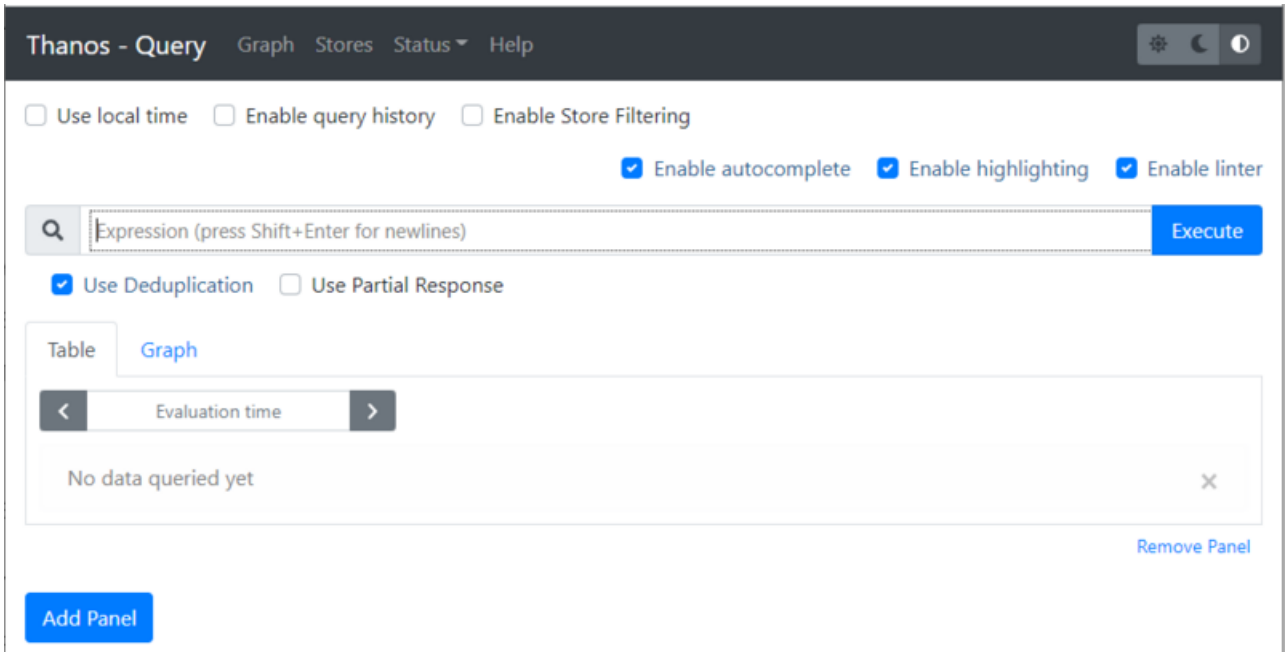
2.2 Tasks: Explore metrics

In this lab you are going to explore various metrics, which your Prometheus server is collecting.

Task 2.2.1: Prometheus/Thanos web UI

As we have now attached the Thanos Querier to your own Prometheus as well as the infrastructure Prometheus, we will use its UI to have a global view.

Get a feel for how to use the Thanos Querier web UI. Open the web UI and navigate to the **Graph** menu (right on top in the grey navigation bar next to Stores).



Let's start and find a memory related metric. The best way to start is by typing `node_memory` in the expression bar.

> **Note**
> As soon as you start typing a dropdown with matching metrics is shown.

Select a metric such as `node_memory_MemFree_bytes` and click the `Execute` button.

The result of your first Query will be available under the two tabs:

1. Table
2. Graph

Explore those two views on your results. Shrink the time range in the Graph tab.

Task 2.2.2: Metric Prometheus server version

Prometheus collects its own metrics, so information such as the current build version of your Prometheus server is displayed as a metric.

Let's find a metric that shows you the version of your Prometheus server.

Start typing `prometheus_...` in the expression browser, choose the `prometheus_build_info` metric and click the `Execute` Button.

Something similar to the following will be displayed

Sice we have two Prometheus Servers connected to our querier, there will be two results

```
prometheus_build_info{branch="HEAD", container="prometheus", endpoint="http-web", goarch="amd64", goos="linux", goversi
on="go1.21.0", instance="10.244.18.50:9090", job="kube-prometheus-stack-prometheus", namespace="monitoring", pod="prome
theus-kube-prometheus-stack-prometheus-0", prometheus="monitoring/kube-prometheus-stack-prometheus", revision="efa34a58
40661c29c2e362efa76bc3a70dccb335", service="kube-prometheus-stack-prometheus", tags="netgo,builtinassets,stringlabels",
 version="2.47.0"}
prometheus_build_info{branch="HEAD", container="prometheus", endpoint="web", goarch="amd64", goos="linux", goversion="g
o1.21.0", instance="10.244.12.41:9090", job="prometheus-operated", namespace="user6-monitoring", pod="prometheus-promet
heus-0", prometheus="monitoring/kube-prometheus-stack-prometheus", revision="efa34a5840661c29c2e362efa76bc3a70dccb335",
 service="prometheus-operated", tags="netgo,builtinassets,stringlabels", version="2.47.0"}
```

The actual Version of your Prometheus Server will be available as label `version`

```
{version="2.47.0"}
```

Task 2.2.3: Metric TCP sockets

Let’s explore a `node exporter` metric in this lab.

1. Find a metric that shows you the number of TCP sockets in use
2. Display the number 5 minutes ago
3. Display the numbers in a graph over the last 15 minutes

The node exporter metrics are all available in the `node` namespace .

The number of TCP sockets in use are available in the following metric.

```
node_sockstat_TCP_inuse
```

If you want to display the value 5 minutes ago, you’ll have to add the correct timestamp in the **Evaluation time** field.

Switch to the **Graph** tab and change the value of the timepicker from `1h` to `15m` to display the graph over the last 15 minutes.

Task 2.2.4: Metric network interfaces

Most virtual Linux machines nowadays have network interfaces. The `node exporter` you have enabled and configured in the previous lab also exposes metrics about network components.

Show all disk devices where the device name starts with `sd`

The network interfaces are available in the following series:

```
node_network_info
```

The result includes all sorts of network interface. If you need to **filter** the result by a label you will have to alter your query:

```
node_disk_filesystem_info{device="sdc"}
```

But this will only show results for the exact `sdc` device. The Task was to show all interfaces that start with `sd`.

In this case we have to use Time series Selectors to create a matching filter:

```
node_disk_filesystem_info{device=~"sd.*"}
```

There will be a lot more about queries and filtering in the next Labs

2.3 Tasks: PromQL

In this lab you are going to learn a bit more about PromQL (Prometheus Query Language) .

PromQL is the query language that allows you to select, aggregate and filter the time series data collected by prometheus in real time.

> **Note**
> Execute the queries in the Thanos Querier UI .
>
> PromQL can seem overwhelming. It may take a little time to get used to it. There may be different approaches to solve the tasks. Our solution is just one possibility.

Task 2.3.1: Explore Examples

In this first task you are going to explore some querying examples.

Get all time series with the metric `prometheus_http_requests_total` .

```
prometheus_http_requests_total
```

The result represents the time series for the http requests sent to your Prometheus server as an **instant vector**.

Get all time series with the metric `prometheus_http_requests_total` and the given `code` and `handler` labels.

```
prometheus_http_requests_total{code="200", handler="/api/v1/targets"}
```

The result will show you the time series for the http requests sent to the query endpoint of your Prometheus Server, which were successful (HTTP status code 200).

Get a whole range of time (5 minutes) for the same vector, making it a **range vector**:

```
prometheus_http_requests_total{code="200", handler="/api/v1/targets"}[5m]
```

A range vector can not be graphed directly in the Prometheus UI, use the table view to display the result.

With regular expressions you can filter time series only for `handlers` whose name matches a certain pattern, in this case all `handlers` starting with `/api` :

```
prometheus_http_requests_total{handler=~"/api.*"}
```

All regular expressions in Prometheus use the RE2 syntax . To select all HTTP status codes except 2xx, you would execute:

```
prometheus_http_requests_total{code!~"2.."}
```

Task 2.3.2: Sum Aggregation Operator

The Prometheus Aggregation operators help us to aggregate time series in PromQL.

There is a Prometheus metric that represents all samples scraped by Prometheus. Let's sum up the metrics returned.

The metric `scrape_samples_scraped` represents the total of scraped samples by `job` and `instance`. To get the total amount of scraped samples, we use the Prometheus aggregation operators `sum` to sum the values.

```
sum(scrape_samples_scraped)
```

Task 2.3.3: Rate Function

Use the `rate()` function to display the current CPU **idle** usage per CPU core of the server in % based on data of the last 5 minutes.

> **Hint**
> Read the documentation about the `rate()` function.

The CPU metrics are collected and exposed by the `node_exporter` therefore the metric we're looking for is under the `node` namespace.

```
node_cpu_seconds_total
```

To get the `idle` CPU seconds, we add the label filter `{mode="idle"}`.

Since the `rate` function calculates the per-second average increase of the time series in a **range vector**, we have to pass a range vector to the function.

To get the idle usage in % we therefore have to multiply it with 100.

```
rate(
  node_cpu_seconds_total{mode="idle"}[5m]
  )
* 100
```

Task 2.3.4: Arithmetic Binary Operator

In the previous lab, we created a query that returns the CPU **idle** usage. Now let's reuse that query to create a query that returns the current CPU usage per core of the server in %. The usage is the total (100%) minus the CPU usage **idle**.

To get the CPU usage we can simply substract **idle** CPU usage from 1 (100%) and then multiply it by 100 to get percentage.

```
(
  1 -
  rate(
      node_cpu_seconds_total{mode="idle"}[5m]
      )
)
* 100
```

Task 2.3.5: How much free memory

Arithmetic Binary Operator can not only be used with constant values `eg. 1` , it can also be used to evaluate to other instant vectors.

Write a Query that returns how much of the memory is free in %.

The `node exporter` exposes these two metrics:

- `node_memory_MemTotal_bytes`
- `node_memory_MemAvailable_bytes`

We can simply divide the available memory metric by the total memory of the node and multiply it by 100 to get percent.

```
sum by(instance) (node_memory_MemAvailable_bytes)
/
sum by(instance) (node_memory_MemTotal_bytes)
* 100
```

Task 2.3.6: Comparison Binary Operators

In addition to the Arithmetic Binary Operator, PromQL also provides a set of Comparison binary operators

- `==` (equal)
- `!=` (not-equal)
- `>` (greater-than)
- `<` (less-than)
- `>=` (greater-or-equal)
- `<=` (less-or-equal)

Check if the server has more than 20% memory available using a Comparison binary operators

We can simply use the greater-than-binary operator to compare the instant vector from the query with 20 (In our case, this corresponds to 20% memory usage).

```
sum by(instance) (node_memory_MemAvailable_bytes)
/
sum by(instance) (node_memory_MemTotal_bytes)
* 100
> 20
```

The query only has a result when more than 20% of the memory is available.

Change the value from 20 to 90 or more to see the result, when the operator doesn’t match.

- acend gmbh

Task 2.3.7: Histogram (optional)

So far we've been using gauge and counter metric types in our queries.

Read the documentation about the `histogram` metric type.

There exists a histogram for the http request durations to the Prometheus sever. It basically counts requests that took a certain amount of time and puts them into matching buckets (`le` label).

We want to write a query that returns

- the total numbers of requests
- to the Prometheus server
- on `/metrics`
- below 0.1 seconds

A metric name has an application prefix relevant to the domain the metric belongs to. The prefix is sometimes referred to as namespace by `client libraries` . As seen in previous labs, the http metrics for the Prometheus server are available in the `prometheus_` namespace.

By filtering the `le` label to 0.1 we get the result for our query.

```
prometheus_http_request_duration_seconds_bucket{handler="/metrics",le="0.1"}
```

Tip: Analyze the query in PromLens

Advanced: You can calculate how many requests in % were below 0.1 seconds by aggregating above metric. See more information about Apdex score at Prometheus documentation

Example

```
sum(
  rate(
    prometheus_http_request_duration_seconds_bucket{handler="/metrics",le="0.1"}[5m]
  )
) by (job, handler)
/
sum(
  rate(
    prometheus_http_request_duration_seconds_count{handler="/metrics"}[5m]
  )
) by (job, handler)
* 100
```

Task 2.3.8: Quantile (optional)

We can use the histogram_quantile function to calculate the request duration quantile of the requests to the Prometheus server from a histogram metric. To archive this we can use the metric `prometheus_http_request_duration_seconds_bucket` , which the Prometheus server exposes by default.

Write a query, that returns the per-second average of the 0.9th quantile under the `metrics` handler using the metric mentioned above.

Expression

```
histogram_quantile(
  0.9,
  rate(
    prometheus_http_request_duration_seconds_bucket{handler="/metrics"}[5m]
  )
)
```

Explanation: `histogram_quantile` will calculate the 0.9 quantile based on the samples distribution in our buckets by assuming a linear distribution within a bucket.

Task 2.3.9: predict_linear function (optional)

We could simply alert on static thresholds. For example, notify when the file system is more than 90% full. But sometimes 90% disk usage is a desired state. For example, if our volume is very large. (e.g. 10% of 10TB would still be 1TB free, who wants to waste that space?) So it is better to write queries based on predictions. Say, a query that tells me that my disk will be full within the next 24 hours if the growth rate is the same as the last 6 hours.

Let's write a query, that exactly makes such predictions:

- Find a metric that displays you the available disk space on filesystem mounted on `/`
- Use a function that allows you to predict when the filesystem will be full in 4 hours
- Predict the usage linearly based on the growth over the last 1 hour

Expression

```
predict_linear(node_filesystem_avail_bytes{mountpoint="/"}[1h], 3600 * 4) < 0
```

Explanation: based on data over the last `1h`, the disk will be `< 0` bytes in `3600 * 4` seconds. The query will return `no data` because the file system will not be full in the next 4 hours. You can check how much disk space will be available in 4 hours by removing the `< 0` part.

```
predict_linear(node_filesystem_avail_bytes{mountpoint="/"}[1h], 3600 * 4)
```

Task 2.3.10: Many-to-one vector matches (optional)

Prometheus provides built-in metrics that can be used to correlate their values with metrics exposed by your exporters. One such metric is `date()`. Prometheus also allows you to add more labels from different metrics if you can correlate both metrics by labels. See Many-to-one and one-to-many vector matches for more examples.

Write a query that answers the following questions:

- What is the uptime of the server in minutes?
- Which kernel is currently active?

Expression

```
(
  (
    time() - node_boot_time_seconds
  ) / 60
)
* on(instance) group_left(release) node_uname_info
```

- **time()**: Use the current UNIX Epoch time
- **node_boot_time_seconds**: Returns the UNIX epoch time at which the VM was started
- **on(instance) group_left(release) node_uname_info**: Group your metrics result with the metric `node_uname_info` which contains information about your kernel in the `release` label.

Alternative solution with `group_right` instead of `group_left` would be:

```
node_uname_info * on(instance) group_right(release)
(
  (
    time() - node_boot_time_seconds
  ) / 60
)
```

- acend gmbh

3. Visualization

Our goal with this lab is to give you a brief overview how to visualize your Prometheus time series produced in the previous labs. For a more detailed tutorial, please refer to the Grafana tutorials .

Useful links and guides

- Prometheus data source
- Grafana dashboards
- Grafana provisioning

3.1 Tasks: Grafana intro

Task 3.1.1: Install Grafana

Similar to the basic setup, we are just going to update our configuration of the ArgoCD application to install and create our Grafana instance. Update your monitoring application (`charts/user-monitoring/values.yaml`) and update the `grafana.enabled` flag to `true` :

`charts/user-monitoring/values.yaml` :

```
user: <user> # Replace me
# prometheus
prometheus:
  enabled: true
# thanos-query
query:
  enabled: true
# grafana
grafana:
  enabled: true
# blackboxexporter
blackboxexporter:
  enabled: false
# pushgateway
pushgateway:
  enabled: false
# alertmanager
alertmanager:
  enabled: false
# thanos-ruler
ruler:
  enabled: false
```

Verify the installation and sync process in the ArgoCD UI . When the application is synchronized successfully navigate to your freshly created Grafana instance: https://<user>-grafana.training.cluster.acend.ch . Use the admin account to login in. Skip the password change.

> **Note**
> This grafana setup does not have persistence storage attached, therefore all compontents (dashboards, datasources, plugins, and so on) need to be provisioned with configuration using our gitops approach. It also means if we change something manually and the pod gets restarted, all changes will be lost.

Task 3.1.2: Add additional datasource to Grafana

By default, our setup adds the grafana datasource `thanos-querier`. This will allow us to use the combined view of metrics in the future to create dashbaords based on this data.

Navigate to the Datasouce Section in your Grafana installation and verify whether the datasources exists or not.

https://<user>-grafana.training.cluster.acend.ch/connections/datasources

Eventhough the `thanos-querier` datasource will be the main datasource to use in our lab, we're going to add an additional datasource to Grafana, for you to understand how that can be configured and to have your `<user>-monitoring` prometheus server dircetly accessible in grafana as well.

Datasources can be added via config files to Grafana, in our case this will be handled with a secret containing the datasources in the following format:

```
apiVersion: 1

datasources:
  - name: Graphite
    url: http://localhost:$PORT
    user: $USER
    secureJsonData:
      password: $PASSWORD
```

In our example the datasources are already handled by the Helm Chart. Update your ArgoCD application and add the datasource config block to the `values.yaml` like the following:

```
# grafana
grafana:
  enabled: true
  datasources:
  - name: prometheus
    access: proxy
    editable: false
    type: prometheus
    url: http://prometheus-operated:9090
```

Commit your changes in the ArgoCD application to your git repository and let it synchronize.

Again go to https://<user>-grafana.training.cluster.acend.ch/connections/datasources and check for the newly added datasource.

Note

While Grafana can discover dashboards at runtime, preprovisioned datasources, like the one we just added, are only discovered during the startup.

Therefore we need to restart our grafana pod manually for the datasource to appear.

```
kubectl -n $USER-monitoring get pod
kubectl -n $USER-monitoring delete pod <grafana-pod>
```

Task 3.1.3: Configure Prometheus to scrape Grafana metrics

This is repetition. Grafana instruments the Prometheus client library and provides a variety of metrics at the `/metrics` endpoint: http://<user>-grafana.training.cluster.acend.ch/metrics

This endpoint can be configured as target in prometheus, with the same methode like we did with our example application.

We simply configure a Service Monitor, which tells the prometheus server where to scrape the metrics from.

The Grafana Service Monitor has already been deployed together with grafana itself.

```
kubectl -n $USER-monitoring get servicemonitor grafana-monitor -oyaml
```

Check if the Grafana instance appears in the targets section of Prometheus (http://<user>-prometheus.training.cluster.acend.ch/targets). In addition you can use the following query to show list all metrics of the new target:

```
{job="grafana"}
```

Task 3.1.4: Use datasources in grafana

Since we have our datasouces now configured in Grafana, we can use the explore tab in grafana, to check, whether it's working correctly.

- Open your grafana https://<user>-grafana.training.cluster.acend.ch
- Use the navigation to open the `Explore` site.
- Select the `prometheus` datasource
- Switch to code mode in your query input section
- Execute the following query

```
python_info
```

The result will be the same like the one from Lab 1.

As we are most likely always more interested in the global view, we are again going to use the `thanos-querier` datasource instead of the `prometheus` one.

- Change the datasource to `thanos-querier`
- Execute the same query again.

Again the same result.

But, since the querier now combines the metrics from our two prometheus stacks (Cluster infrastructure and user-monitoring) we now have the possibility to also query cluster metrics:

```
kube_node_info
```

Which will give you info about our underlying kubernetes cluster nodes. Such metrics are only available on the `thanos-querier` datasource.

> **Note**
> From now on we'll be using primarily the `thanos-querier` datasource.

3.2 Tasks: Grafana dashboards

Task 3.2.1: Import a dashboard

As a first Task we import an existing dashboard to our grafana. There is a huge collection of predefined dashboards available on https://grafana.com/grafana/dashboards .

In this task we learn how to import existing dashboards, which we can also use as reference to write our own queries and dashboards.

Choose one or more of the preexisting dashboards from https://grafana.com/grafana/dashboards and import them into your grafana. Use the daterange, interval picker on the top right to change the timerange (between `5 minutes` and `10 minutes`) of the displayed mertics.

> **Note**
> You can import the following dashboards
>
> - Node Exporter Full dashboard, which will present you useful metrics about your linux servers
> - Prometheus Overview . This gives you an overivew of your prometheus instance.
> - Kubestate Metrics Information about your Kubernetes Cluster
> - ...

- Navigate to dashboard site and copy the dashboard ID
- On your Grafana web UI
  - Navigate to **Dashboards** (Icon with the four squares on the left navigation menu) > New > **Import**
  - Add the copied ID to the **Import via grafana.com** field
  - Hit **Load**
- Choose your **thanos-querier** data source and hit **Import**
- Open the dashboard time control (to the upper right)
  - Set **From** to `now-10m`
  - Set **To** to `now-5m`
  - Hit **Apply time range**

Task 3.2.2: Create your first dashboard

In this task you're going to create your first own dashboard `happy_little_dashboard` . You will add the panel `CPU Utilisation` based on the following query:

```
sum(rate(container_cpu_usage_seconds_total{container="example-web-python", image!="", namespace="<user>"}[$__rate_interval])) by (pod)
```

- Navigate to Dashboards (Icon with the four squares on the left navigation menu)> New > **New Dashboard**
  - Select **Add visualization**
- Select the **thanos-querier** data source
- In general, metrics can be built using the Grafana Query Builder or using "plain" PromQL queries. You can easily switch between these two at the top right of the query window. Going forward, we will use plain PromQL queries.

- Add the expression from above into the text field right next to the **Metrics Browser** dropdown
- Set the panel title to `CPU Utilisation` under **Panel options > Title** (you may need to open the options pane with the **<** button on the right hand side just below the **Apply** button)
- Save the dashboard and give it the name `happy_little_dashboard`

Task 3.2.3: Add a Gauge panel to the dashboard

Task description:

Add another panel to the existing `happy_little_dashboard` with the panel name `Memory Used`. Display the following query:

```
sum(container_memory_working_set_bytes{namespace="<user>", pod=~"example-web-python.*", image!=""}) / sum(kube_pod_cont
ainer_resource_limits{namespace="<user>", pod=~"example-web-python.*", resource="memory"})

Also, change the panel type to `Gauge` and display it in %. Define the following thresholds:
```bash
0% (green)
60% (orange)
80% (red)
```

- Hit **Add** (top navigation menu) **> Visualization**
- Select the **thanos-querier** data source
- Add the expression from above into the text field right next to the **Metrics Browser** dropdown
- Set the panel title to `Memory Used` under **Panel options > Title** (you may need to open the options pane with the **<** button on the right hand side just below the **Apply** button)
- Define unit under **Standard options > Unit > Misc / Percent (0-100)**
- Choose **Gauge** in the dropdown menu just below the **Apply** button
- Add `60` and `80` thresholds under **Thresholds** and switch to Precentage
  - Choose **Greem** for **Base**
  - Choose **Orange** for **60**
  - Choose **Red** for **80**
- Save the dashboard

Task 3.2.4: Add a Stat panel that uses a variable to the dashboard

Add another panel to the existing `happy_little_dashboard` with the panel name `Disk Available` that uses a variable. Name the variable `disk` and label it `Select disk`. To calculate the available disk space of a certain mountpoint in percent, use the following query:

```
100 - ((node_filesystem_avail_bytes{mountpoint="$disk", instance="10.0.0.10:9100"} * 100) / node_filesystem_size_bytes{
mountpoint="$disk", instance="10.0.0.10:9100"})
```

Also, change the panel type to `Stat` and display the value in %. Define the following thresholds:
```

- acend gmbh

```
0% (red)
10% (orange)
25% (green)
```

- First, we create the variable. Hit the little gear icon on the top right corner of the dashboard
  - Select **Variables** in the menu on the left **> Add Variable**
  - As we want the available values for the variable to be calculated dynamically, we will use a PromQL query for this as well. Thus, choose **Query** in the dropdown menu for `variable type`
  - Set `Name` to `disk`
  - Set `Label` to `Select disk`
  - Select the **thanos-querier** data source
  - Set `Query Type` to `Label values`
  - Choose the label `mountpoint`
  - As we are only interested in the mountpoint label of our linux VM, enter the metric `node_filesystem_avail_bytes{instance="10.0.0.10:9100"}`
  - The preview at the bottom should now already show the different mountpoints of the server
  - Leave everything else as is, apply and navigate back to the dashboard
- Now, let's use the variable in a new panel. Hit **Add** (top navigation menu) **> Visualization**
  - Select the **thanos-querier** data source
  - Add the query `100 - ((node_filesystem_avail_bytes{mountpoint="$disk", instance="10.0.0.10:9100"} * 100) / node_filesystem_size_bytes{mountpoint="$disk", instance="10.0.0.10:9100"})` to the **Metrics browser** field
  - Set the panel title to `Disk Available` under **Panel options > Title** (you may need to open the options pane with the **<** button on the right hand side just below the **Apply** button)
  - Define unit under **Standard options > Unit > Misc / Percent (0-100)**
  - Choose **Stat** in the dropdown menu just below the **Apply** button
  - Add `10` and `25` thresholds under **Thresholds**
    - Choose **Red** for **Base**
    - Choose **Orange** for **10**
    - Choose **Green** for **25**
- Save the dashboard

Task 3.2.5: Save your dashboard to GIT

In a git provisioned Grafana Setup the dashboards will only be persisted in your GIT repo. Manually clicked dashboards get deleted everytime Grafana restarts. Therefore, let us make sure that your dashboard will not be lost.

- Save your dashboard in a configmap to your `user-demo` directory
- On your Grafana web UI
  - Navigate to your Dashboard `happy_little_dashboard`
  - Click the share icon, to the right of the dashboard name
  - Select export, then view JSON
  - And hit *Copy to Clipboard*
  - Copy the JSON content and save the file `charts/user-monitoring/templates/training_dashboard.yaml` :

```
apiVersion: v1
kind: ConfigMap
metadata:
  labels:
    dashboard: 'true'
  name: training-dashboard
data:
  training-dashboard.json: |-
    {
      Your dashboard JSON definition
    }
```

> **Note**
> Pay attention to YAML syntax, especially indentation.

```
   training-dashboard.json: |-
     {
       "annotations": {
         "list": [
           {
...
```

Commit and push the dashboard.

To ensure that the dashboard provisioning works as specified, try deleting the dashboard using the Grafana user interface.

- On your Grafana web UI
  - Navigate to your Dashboard `happy_little_dashboard`
  - Select **Dashboard settings** (Icon on the top navigation menu that looks like a gear)
  - Click on **Delete Dashboard**

You should see a warning message that prevents deleting `provisioned dashboards` .

- acend gmbh

4. Prometheus exporters

An increasing number of applications directly instrument a Prometheus metrics endpoint. This enables applications to be scraped by Prometheus out of the box. For all other applications, an additional component (the Prometheus exporter) is needed to close the gap between Prometheus and the application which should be monitored.

> **Note**
> There are lots of exporters available for many applications, such as MySQL/MariaDB, Nginx, Ceph, etc. Some of these exporters are maintained by the Prometheus GitHub organization while others are maintained by the community or third-party vendors. Check out the list of exporters on the Prometheus website for an up-to-date list of exporters.

One example of a Prometheus exporter is the `node_exporter` we used in the second chapter of this training. This exporter collects information from different files and folders (e.g., `/proc/net/arp`, `/proc/sys/fs/file-nr`, etc.) and uses this information to create the appropriate Prometheus metrics. In the tasks of this chapter we will configure two additional exporters.

Special exporters

Blackbox exporter

This is a classic example of a so-called multi-target exporter which uses relabeling to pass the targets to the exporter. This exporter is capable of probing the following endpoints:

- HTTP
- HTTPS
- DNS
- TCP
- ICMP

By using the TCP prober you can create custom checks for almost any service including services using STARTTLS. Check out the example.yml file in the project's GitHub repository.

Prometheus Pushgateway

The Pushgateway allows jobs (e.g., Kubernetes Jobs or CronJobs) to push metrics to an exporter where Prometheus will collect them. This can be required since jobs only exist for a short amount of time and as a result, Prometheus would fail to scrape these jobs most of the time. In addition, it would require all these jobs to implement a webserver in order for Prometheus to collect the metrics.

> **Note**
> The Pushgateway should only be used for for this specific use case. It simply acts as cache for short-lived jobs and by default does not even have any persistence. It is not intended to convert Prometheus into a push-based monitoring system

4.1 Tasks: Blackbox exporter

Task 4.1.1: Install Blackbox exporter

Similar to the basic setup, we are just going to update our configuration of the ArgoCD application to install the Blackbox exporter. Update your monitoring application (`charts/user-monitoring/values.yaml`) and update the `blackboxexporter.enabled` flag to `true` :

`charts/user-monitoring/values.yaml` :

```
user: <user> # Replace me
# prometheus
prometheus:
  enabled: true
# thanos-query
query:
  enabled: true
# grafana
grafana:
  enabled: true
  datasources:
  - name: prometheus
    access: proxy
    editable: false
    type: prometheus
    url: http://prometheus-operated:9090
# blackboxexporter
blackboxexporter:
  enabled: true
# pushgateway
pushgateway:
  enabled: false
# alertmanager
alertmanager:
  enabled: false
# thanos-ruler
ruler:
  enabled: false
```

Commit and push the changes.

Verify the installation and sync process in the ArgoCD UI . Or execute the following command:

```
kubectl -n $USER-monitoring get pod
```

Task 4.1.1: Add a blackbox target

We will use the blackbox exporter to create a new probe which accepts a `2xx` return code as a valid http return code. This will return the `probe_success` metric from the blackbox exporter with the value `1` , if the http status code is `2xx` .

Task description:

- Create a probe (`user-demo/training_blackbox_target.yaml`) which uses the HTTP prober and expects a `2xx` return code as a valid status code
- Define `https://www.acend.ch` as a single static target, which the blackbox should probe

To configure the blackbox exporter you have to add the following file `user-demo/training_blackbox_target.yaml` to your directory, commit and push the changes:

```
apiVersion: monitoring.coreos.com/v1
kind: Probe
metadata:
  name: acend-2xx
spec:
  module: http_2xx
  prober:
    url: blackbox:9115
  targets:
    staticConfig:
      static:
      - https://www.acend.ch
```

The Prometheus server translates this Probe Custom Resource into a prometheus target, which gets scraped accordingly to its scrape interval.

Url: `http://<blackboxexportertservice>?target=https://www.acend.ch&module=http_2xx`

Check the prometheus target configuration in the Prometheus UI .

- Open a new Terminal and execute the following command

```
kubectl -n $USER-monitoring port-forward service/blackbox 9115:9115
```

This will open a port-forward to the blackbox exporter.

- Switch back to the initial terminal and execute this command

```
curl localhost:9115/probe?target=https://www.acend.ch&module=http_2xx
```

The blackbox exporter checks the given target url and returns metrics, the metric `probe_success` metric should have the value `1` .

```
...
# HELP probe_success Displays whether or not the probe was a success
# TYPE probe_success gauge
probe_success 1
...
```

Task 4.1.2: Query blackbox metrics

Let’s now create a query which selects all metrics belonging to the blackbox exporter target `https://www.acend.ch` and display them in the Thanos Querier UI .

We can select all metrics for the target with the following query:

```
{instance="https://www.acend.ch"}
```

Warning

In the list of metrics you will find one metric with the name `up`. In the case of a multi-target exporter such as the blackbox exporter this metric will always be up as long as Prometheus is able to successfully scrape the exporter even if the actual target (website, TCP service, etc.) is down. To monitor the state of the targets always use the `probe_success` metric.

Task 4.1.3 (optional): Add a protocol label to your blackbox target

Add the new label `protocol` to every blackbox exporter target by updating the relabel config. The new label should contain the protocol (HTTP or HTTPS) extracted from the target URL.

To configure the blackbox exporter you have to updates the following file `training_blackbox_target.yaml` in your directory:

```
apiVersion: monitoring.coreos.com/v1
kind: Probe
metadata:
  name: acend-2xx
spec:
  module: http_2xx
  prober:
    url: blackbox:9115
  targets:
    staticConfig:
      static:
      - https://www.acend.ch
  metricRelabelings:
  - sourceLabels: [instance] #1
    targetLabel: protocol #2
    regex: '^(.+):.+' #3
    replacement: $1 #4
```

- **1**: Use the value from the label `instance`. This label contains all targets defined at `.spec.targets.staticConfig.static`
- **2**: We will call the new label `protocol`
- **3**: Capture the first part of your url until `:`. In our case `https` from `https://acend.ch/`
- **4**: Replace `target_label` value with the regex match from `source_labels` value

Task 4.1.4 (optional): PromQL Query, ssl certificate expire

The blackbox exporter provides a lot of metrics. One very common usecase is, to create a dashboard or alert for expiring SSL certificates.

Wrtie a query, that returns the days until the ssl certificate of https://www.acend.ch expires.

Info

Hint: Use the `probe_ssl_earliest_cert_expiry` as a starting point. This metric returns the timestamp in seconds, when the certificate will expire. The result of the `time()` function, is the current timestamp.

```
# solution
(probe_ssl_earliest_cert_expiry - time()) / (3600 *24)
```

4.2 Tasks: Pushgateway

Task 4.2.1 - Install and configure Pushgateway

Update your monitoring application (`charts/user-monitoring/values.yaml`) and update the `blackboxexporter.enabled` flag to `true` :

`charts/user-monitoring/values.yaml` :

```
user: <user> # Replace me
# prometheus
prometheus:
  enabled: true
# thanos-query
query:
  enabled: true
# grafana
grafana:
  enabled: true
  datasources:
  - name: prometheus
    access: proxy
    editable: false
    type: prometheus
    url: http://prometheus-operated:9090
# blackboxexporter
blackboxexporter:
  enabled: true
# pushgateway
pushgateway:
  enabled: true
# alertmanager
alertmanager:
  enabled: false
# thanos-ruler
ruler:
  enabled: false
```

Commit and push the changes and verify whether the deployment worked correctly.

Task 4.2.2 - Push metrics to Pushgateway

In this task you're going to push metrics to the Pushgateway. This is what you would normally do, after a cronjob has completed successfully.

In order to push metrics to the Pushgateway, you can simply send an HTTP `POST` or `PUT` request, with the actual metric we want to push as content.

When pushing metrics to the Pushgateway, you always have to specify the job, therefore the URL Path looks like this:

```
http://localhost:9091/metrics/job/<JOB_NAME>{/<LABEL_NAME>/<LABEL_VALUE>}
```

If we want to push the metric `prometheus_training_labs_completed_total` with the value `4` and the job `prometheus_training` to the Pushgateway, we can do that by creating the following Kubernetes Job:

```
kubectl -n $USER-monitoring create job --image=quay.io/acend/example-web-python pushgw-example1 -- \
sh -c "echo 'prometheus_training_labs_completed_total 3' | curl --data-binary @- http://pushgateway:9091/metrics/job/prometheus_training"
```

Command Explanation

If you are not very familiar with `kubectl create job`. The above command does the following:

- `kubectl -n ... create job` creates an adhoc kubernetes job
- `--image=` specifies, which image the container will use. We will use the toolkit container because it provides bash and curl.
- `pushgw-example1` is the name of the job
- `bash -c "..."` is the command, the job should execute

Verify the metric in the Thanos Querier web UI. It may take up to 30s (Depending on the `scrape_interval`) to be available in Prometheus.

Push the following metric (notice the `instance` label) to the Pushgateway and make sure the metric gets scraped by Prometheus

```
# TYPE some_metric_total counter
# HELP This is just an example metric.
some_metric_total{job="prometheus_training",instance="myinstance"} 42
```

To push a metric to the Pushgateway, which will then be scraped by Prometheus, we can simply create the following job. Note the actual content of the HTTP request, is exactly the metric we want Prometheus to scrape.

Execute the following command to push the metric to your Pushgateway:

```
kubectl -n $USER-monitoring create job --image=quay.io/acend/example-web-python pushgw-example2 -- \
sh -c "cat <<EOF | curl --data-binary @- http://pushgateway:9091/metrics/job/prometheus_training/instance/myinstance
# TYPE some_metric_total counter
# HELP This is just an example metric.
some_metric_total 42
EOF"
```

Command Explanation

If you are not very familiar with the Linux shell, the above command does the following:

- the `cat` command reads the actual metric and pipes it to `stdin`
- curl sends a HTTP POST request to the URL http://pushgateway:9091/metrics/job/prometheus_training/instance/myinstance with the –data-binary parameter set to `stdin` (the actual metric)

Verify the metric in the Thanos Querier web UI. It may take up to 30s (depending on the `scrape_interval`) to be available in Prometheus.

Task 4.2.3 - Delete Pushgateway metrics

By sending HTTP `delete` requests to the same endpoint, we can delete metrics from the Pushgateway.

> **Note**
> Metrics pushed to the Pushgateway are not automatically purged until you manually delete them via the API or the process restarts. If you persist the metrics with `--persistence.file`, you should ensure that you have set up a job that cleans up the metrics on a regular basis.

According to the official Pushgateway documentation you can delete either metrics for specific label combinations (exact match required) or all metrics.

Delete the pushed metrics from the Pushgateway.

To delete the metrics for the job `prometheus_training`, you can simply execute the following command:

```
kubectl -n $USER-monitoring create job --image=quay.io/acend/example-web-python pushgw-delete -- \
curl -X DELETE http://pushgateway:9091/metrics/job/prometheus_training
```

> **Note**
> This will delete metrics with the label set `{job="prometheus_training"}` but not `{job="prometheus_training",another_label="value"}` since the delete methode requires an exact label match.

The Pushgateway pod has no persistence, so you can delete all metrics stored in Pushgateway by deleting the pod.

```
kubectl -n $USER-monitoring delete pod -l app.kubernetes.io/name=pushgateway
```

Remove the created examples jobs.

```
kubectl -n $USER-monitoring delete jobs pushgw-delete pushgw-example1 pushgw-example2
```

4.3 Tasks: Exporter as a sidecar

Task 4.3.1: Deploy a database and use a sidecar container to expose metric

As we've learned in Lab 4 - Prometheus exporters when applications do not expose metrics in the Prometheus format, there are a lot of exporters available to convert metrics into the correct format. In Kubernetes this is often done by deploying so called sidecar containers along with the actual application.

This lab demonstrates that with a `mariadb` database. In the first step, we are going to install a plain `mariadb` with a Secret (username password to access the database), a Service and the Deployment .

Create the following three files with the corresponding filenames and content:

`user-demo/mariadb-secret.yaml` :

```
---
apiVersion: v1
kind: Secret
metadata:
  name: mariadb
  labels:
    app: mariadb
data:
  database-name: YWNlbmRleGFtcGxlZGI=
  database-password: bXlzcWxwYXNzd29yZA==
  database-root-password: bXlzcWxyb290cGFzc3dvcmQ=
  database-user: YWNlbmQtdXNlcg==
```

`user-demo/mariadb-deployment.yaml` :

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mariadb
  labels:
    app: mariadb
spec:
  selector:
    matchLabels:
      app: mariadb
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mariadb
    spec:
      containers:
        - image: mariadb:10.5
          imagePullPolicy: IfNotPresent
          name: mariadb
          args:
            - --ignore-db-dir=lost+found
          env:
            - name: MYSQL_USER
              valueFrom:
                secretKeyRef:
                  key: database-user
                  name: mariadb
            - name: MYSQL_PASSWORD
              valueFrom:
                secretKeyRef:
                  key: database-password
                  name: mariadb
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  key: database-root-password
                  name: mariadb
            - name: MYSQL_DATABASE
              valueFrom:
                secretKeyRef:
                  key: database-name
                  name: mariadb
          livenessProbe:
            tcpSocket:
              port: 3306
          ports:
            - containerPort: 3306
              name: mariadb
```

user-demo/mariadb-service.yaml :

```
---
apiVersion: v1
kind: Service
metadata:
  name: mariadb
  labels:
    app: mariadb
    prometheus-monitoring: 'true'
spec:
  ports:
    - name: mariadb
      port: 3306
      protocol: TCP
      targetPort: 3306
  selector:
    app: mariadb
```

Add, commit and push the changes to your git repository and ArgoCD will deploy the `mariadb`.

This will create a Secret (username password to access the database), a Service and the Deployment.

MariaDB is not able to expose prometheus metrics out of the box, we need to deploy the mariadb exporter from https://quay.io/prometheus/mysqld-exporter/ as a sidecar container

For that we must alter the existing MariaDB deployment and service definition to contain the side car.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mariadb
  labels:
    app: mariadb
spec:
  selector:
    matchLabels:
      app: mariadb
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mariadb
    spec:
      containers:
        - image: quay.io/prometheus/mysqld-exporter:v0.14.0
          imagePullPolicy: IfNotPresent
          name: mariadb-exporter
          env:
            - name: MYSQL_USER
              valueFrom:
                secretKeyRef:
                  key: database-user
                  name: mariadb
            - name: MYSQL_PASSWORD
              valueFrom:
                secretKeyRef:
                  key: database-password
                  name: mariadb
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  key: database-root-password
                  name: mariadb
            - name: MYSQL_DATABASE
              valueFrom:
                secretKeyRef:
                  key: database-name
                  name: mariadb
```

```
            - name: DATA_SOURCE_NAME
              value: $(MYSQL_USER):$(MYSQL_PASSWORD)@(127.0.0.1:3306)/$(MYSQL_DATABASE)
          ports:
            - containerPort: 9104
              name: mariadb-exp
        - image: mariadb:10.5
          name: mariadb
          args:
            - --ignore-db-dir=lost+found
          env:
            - name: MYSQL_USER
              valueFrom:
                secretKeyRef:
                  key: database-user
                  name: mariadb
            - name: MYSQL_PASSWORD
              valueFrom:
                secretKeyRef:
                  key: database-password
                  name: mariadb
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  key: database-root-password
                  name: mariadb
            - name: MYSQL_DATABASE
              valueFrom:
                secretKeyRef:
                  key: database-name
                  name: mariadb
          livenessProbe:
            tcpSocket:
              port: 3306
          ports:
            - containerPort: 3306
              name: mariadb
```

```
---
apiVersion: v1
kind: Service
metadata:
  name: mariadb
  labels:
    app: mariadb
    prometheus-monitoring: 'true'
spec:
  ports:
    - name: mariadb
      port: 3306
      protocol: TCP
      targetPort: 3306
    - name: mariadb-exp
      port: 9104
      protocol: TCP
      targetPort: 9104
  selector:
    app: mariadb
```

Then we also need to create a new ServiceMonitor (`user-demo/mariadb-servicemonitor.yaml`) to instruct Prometheus to scrape the sidecar container on the given port:

```
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    app.kubernetes.io/name: mariadb
  name: mariadb
spec:
  endpoints:
    - interval: 60s
      port: mariadb-exp
      scheme: http
      path: /metrics
  selector:
    matchLabels:
      prometheus-monitoring: 'true'
```

Explore the changes we did to the resources and verify that the target gets scraped in the Prometheus user interface . Target name: `serviceMonitor/<user>/mariadb/0` (It may take up to two minutes for Prometheus to load the new configuration and scrape the metrics).

Task 4.3.2: (Optional) Explore mariadb Metrics

Explore the newly added metrics using the following PromQL Query:

```
{job="mariadb"}
```

Import the mariadb dashboard and explore how those metrics can be used.

5. Prometheus in Kubernetes

kube-prometheus

The kube-prometheus stack already provides an extensive Prometheus setup and contains a set of default alerts and dashboards from Prometheus Monitoring Mixin for Kubernetes . The following targets will be available.

kube-state-metrics: Exposes metadata information about Kubernetes resources. Used, for example, to check if resources have the expected state (deployment rollouts, pods CrashLooping) or if jobs fail.

```
# Example metrics
kube_deployment_created
kube_deployment_spec_replicas
kube_daemonset_status_number_misscheduled
...
```

cAdvisor: cAdvisor exposes usage and performance metrics about running container. Commonly used to observe memory usage or CPU throttling .

```
# Example metrics
container_cpu_cfs_throttled_periods_total
container_memory_working_set_bytes
container_fs_inodes_free
...
```

kubelet: Exposes general kubelet related metrics. Used to observe if the kubelet and the container engine is healthy.

```
# Example metrics
kubelet_runtime_operations_duration_seconds_bucket
kubelet_runtime_operations_total
...
```

apiserver: Metrics from the Kubernetes API server. Commonly used to catch errors on resources or problems with latency.

```
# Example metrics
apiserver_request_duration_seconds_bucket
apiserver_request_total{code="200",...}
...
```

probes: Expose metrics about Kubernetes liveness, readiness and startup probes Normally you would not alert on Kubernetes probe metrics, but on container restarts exposed by `kube-state-metrics` .

```
# Example metrics
prober_probe_total{probe_type="Liveness", result="successful",...}
prober_probe_total{probe_type="Startup", result="successful",...}
...
```

blackbox-exporter: Exposes default metrics from blackbox-exporter. Can be customized using the Probe custom resource.

```
# Example metrics
probe_http_status_code
probe_http_duration_seconds
...
```

node-exporter: Exposes the hardware and OS metrics from the nodes running Kubernetes.

```
# Example metrics
node_filesystem_avail_bytes
node_disk_io_now
...
```

alertmanager-main/grafana/prometheus-k8s/prometheus-operator/prometheus-adapter: Exposes all monitoring stack component metrics.

```
# Example metrics
alertmanager_alerts_received_total
alertmanager_config_last_reload_successful
...
grafana_build_info
grafana_datasource_request_total
...
prometheus_config_last_reload_successful
prometheus_rule_evaluation_duration_seconds
...
prometheus_operator_reconcile_operations_total
prometheus_operator_managed_resources
...
```

pushgateway: Exposes metrics pushed to your pushgateway.

```
# Example metrics
pushgateway_build_info
pushgateway_http_requests_total
...
```

5.1 Tasks: kube-prometheus metrics

Task 5.1.1: Memory usage of Prometheus

Task description:

- acend gmbh

- Display the memory usage of both Prometheus pods
- Use a filter to just display metrics from your `prometheus` containers

Use the [Thanos Querier web UI](https://<user>-thanos-query.training.cluster.acend.ch to execute the queries.

> **Note**
> Search for a metric with `memory_working_set` in its name

```
container_memory_working_set_bytes{pod=~"prometheus-prometheus-0", container="prometheus",namespace=~"<user>.*"}
```

Task 5.1.2: Kubernetes pod count

Task description:

- Display how many pods are currently running on your Kubernetes platform

There are different ways to archive this. You can for example query all running containers and group them by `pod` and `namespace`.

```
count(sum(kube_pod_container_status_running == 1) by (pod,namespace,cluster))
```

You may also sum() all running pods on your Kubernetes nodes

```
sum(kubelet_running_pods)
```

Task 5.1.3: Display metrics in Kubernetes Grafana

Let's try to inspect some cluster metrics in Grafana. Of course we could build our dashboard from scratch again and feel free to try, but in most cases there will always be someone who has already built a dashboard for your purpose. As we already have seen grafana has a searchable dashboard database to get prebuilt dashboards from. Currently we are interested in resource usage of our kubernetes namespace.

Task description:

- Open and copy the id or json of the Kubernetes / Views / Namespaces dashboard
- Navigate to your Kubernetes Grafana
- Create a new dashboard in your Grafana instance and choose 'Import'
- Paste the Json or Id from the dashboard and choose 'thanos-querier' as your datasource

Inspect the dashboard for your namespace `<user>`, you will find metrics and visualizations for most important metrics. You can inspect and edit panels for your purpose.

Task 5.1.4: (Optional) Kubernetes Dashboards

There are a whole bunch of Kubernetes Dashbaords available, which you can find here: https://github.com/dotdc/grafana-dashboards-kubernetes/tree/master/dashboards

- acend gmbh

If you have enough time or are interested in more Kubernetes Dashboards go a head and import those as well.

5.2 Tasks: Prometheus Operator

5.2.1: Prometheus storage

By default, the Prometheus operator stack does not persist the data of the deployed monitoring stack. Therefore, any pod restart would result in a reset of all data. Let's learn about persistence for Prometheus.

Task description:

- See this example of how to configure storage for Prometheus

To get the current values, either look at the file in git or use `kubectl -n $USER-monitoring get prometheus prometheus -oyaml`.

To enable storage for our Prometheus we would simply add the following section to our Prometheus definition.

```
...
spec:
  ...
  securityContext:
    fsGroup: 65534
  storage:
    disableMountSubPath: true
    volumeClaimTemplate:
      spec:
        resources:
          requests:
            storage: 10Gi
  ...
```

5.2.2: Prometheus Retention

By default, the Prometheus operator stack will set the retention of your metrics to `24h`. When metrics are persisted in Prometheus we can define the retention. Read about retention operational-aspects for options to manage retention.

Task description:

- Read the Operator's documentation about retention and size based retention.

> **Note**
> Check documentation for available options

To configure a retention of two days with a maximum of 9Gi of storage, we could provide the following configuration to our Prometheus resource:

```
...
spec:
  ...
  retention: 2d
  retentionSize: 9GB
  ...
```

5.2.3: (Optional) Configure storage and retention in your Prometheus instance

Since we don't always trust written documentation without testing it ourselves, we are going to configure storage and retention in our `<user>-monitoring` namespace.

The Prometheus resource is provided by the `user-monitoring` chart in your git repository. To edit the resource we need to edit the file `charts/user-monitoring/templates/_user-prometheus.yaml .`

Enable storage:

As we have read in the documentation, we can add the following properties to the Prometheus resource:

```
...
spec:
  ...
  securityContext:
    fsGroup: 65534
  storage:
    disableMountSubPath: true
    volumeClaimTemplate:
      spec:
        resources:
          requests:
            storage: 10Gi
  ...
```

Configure retention:

To configure the retention and retention limits we can add the following properties to the Prometheus resource:

```
...
spec:
  ...
  retention: 2d
  retentionSize: 9GB
  ...
```

Verify the changes:

Commit and push the changes to your git repository and let ArgoCD sync your application. As soon as the pod is up and running again, we can verify that the PVC was created and bound with:

```
kubectl -n $USER-monitoring get pvc
```

Check if the volume is available inside the pod with running `df -h /prometheus` inside the first Prometheus pod.

```
kubectl -n $USER-monitoring exec prometheus-prometheus-0 -c prometheus -- df -h /prometheus
```

Verifying the retention config can be done by inspecting the Prometheus pod's arguments:

```
kubectl -n $USER-monitoring describe pods prometheus-prometheus-0
```

The output should contain the following lines:

```
...
Containers:
  ...
  prometheus:
    ...
    Args:
      ...
      --storage.tsdb.retention.size=9GB
      --storage.tsdb.retention.time=2d
      ...
```

6. Instrumenting with client libraries

While an exporter is an adapter for your service to adapt a service specific value into a metric in the Prometheus format, it is also possible to export metric data programmatically in your application code.

Client libraries

The Prometheus project provides client libraries which are either official or maintained by third-parties. There are libraries for all major languages like Java, Go, Python, PHP, and .NET/C#.

Even if you don’t plan to provide your own metrics, those libraries already export some basic metrics based on the language. For Go , default metrics about memory management (heap, garbage collection) and thread pools can be collected. The same applies to Java .

Specifications and conventions

Application metrics or metrics in general can contain confidential information, therefore endpoints should be protected from unauthenticated users. This can be achieved either by exposing the metrics on a different port, which is only reachable by Prometheus or by protecting the metrics endpoints with some sort of authentication.

There are some guidelines and best practices how to name your own metrics. Of course, the specifications of the datamodel must be followed and applying the best practices about naming is not a bad idea. All those guidelines and best practices are now officially specified in openmetrics.io .

Following these principles is not (yet) a must, but it helps to understand and interpret your metrics.

Best practices

Though implementing a metric is an easy task from a technical point of view, it is not so easy to define what and how to measure. If you follow your existing log statements and if you define an error counter to count all errors and exceptions , then you already have a good base to see the internal state of your application.

The four golden signals

Another approach to define metrics is based on the four golden signals :

- Latency
- Traffic
- Errors
- Saturation

There are other methods like RED or USE that go into the same direction.

Collecting Application Metrics

When running applications in production, a fast feedback loop is a key factor. The following reasons show why it’s essential to gather and combine all sorts of metrics when running an application in production:

- To make sure that an application runs smoothly
- To be able to see production issues and send alerts

- acend gmbh

- To debug an application
- To take business and architectural decisions
- Metrics can also help to decide when to scale applications

6.1 Tasks: Instrumenting

Task 6.1.1: Spring Boot Example Instrumentation

Using the micrometer metrics facade in Spring Boot Applications lets us collect all sort of metrics within a Spring Boot application. Those metrics can be exported for Prometheus to scrape by a few additional dependencies and configuration.

Let's have a deeper look at how the instrumentation of a Spring Boot application works. For that we can use the prometheus-training-spring-boot-example application located at https://github.com/acend/prometheus-training-spring-boot-example . To make the application collect metrics and provide a Prometheus endpoint we now need to simply add the following two dependencies in the `pom.xml` file, where it says `<!-- Add Dependencies here-->` :

> **Note**
> For your convenience, the changes mentioned below are already implemented in the `solution` subfolder of the git repository. You therefore do not have to make any changes in the code.

```
....
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
    <scope>runtime</scope>
</dependency>
....
```

Additionally to those dependencies we also need to configure the metrics endpoints to be exposed.

This can be done in the file `src/main/resources/application.properties` by adding the following line:

```
management.endpoints.web.exposure.include=prometheus,health,info,metric
```

As mentioned above, these changes have already been implemented in the `solution` subfolder of the repository. A pre-built docker image is also available under https://quay.io/repository/acend/prometheus-training-spring-boot-example?tab=tags .

> **Note**
> In the next step we will deploy our application to our OpenShift Cluster for demonstration purposes in our monitoring namespace. This should **never** be done for production use cases. If you are familiar with deploying on OpenShift, you can complete this lab by deploying the application on our test cluster.

- Add the following resource `user-demo/training_springboot_example.yaml` to your git directory, commit and push

your changes.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: example-spring-boot
  name: example-spring-boot
spec:
  replicas: 1
  selector:
    matchLabels:
      app: example-spring-boot
  template:
    metadata:
      labels:
        app: example-spring-boot
    spec:
      containers:
      - image: quay.io/acend/prometheus-training-spring-boot-example
        imagePullPolicy: Always
        name: example-spring-boot
      restartPolicy: Always
---
apiVersion: v1
kind: Service
metadata:
  name: example-spring-boot
  labels:
    app: example-spring-boot
spec:
  ports:
    - name: http
      port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app: example-spring-boot
  type: ClusterIP
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    app.kubernetes.io/name: example-spring-boot
  name: example-spring-boot-monitor
spec:
  selector:
    matchLabels:
      app: example-spring-boot
  endpoints:
  - interval: 30s
    port: http
    scheme: http
    path: /actuator/prometheus
```

Note

This will create a `Deployment`, a `Service` and a `ServiceMonitor` resource in our monitoring namespace. We will learn about `ServiceMonitors` later in labs 8. For now, we only need to know, that a `ServiceMonitor` resource will configure Prometheus targets based on the pods linked to the service.

Verify in the web UI whether the target has been added and is scraped. This might take a while until the target appears.

And you should also be able to find your custom metrics:

```
{job="example-spring-boot"}
```

Explore the spring boot metrics.

Task 6.1.2: Metric names

Study the following metrics and decide if the metric name is ok

```
http_requests{handler="/", status="200"}

http_request_200_count{handler="/"}

go_memstats_heap_inuse_megabytes{instance="localhost:9090",job="prometheus"}

prometheus_build_info{branch="HEAD",goversion="go1.15.5",instance="localhost:9090",job="prometheus",revision="de1c1243f
4dd66fbac3e8213e9a7bd8dbc9f38b2",version="2.32.1"}

prometheus_config_last_reload_success_timestamp{instance="localhost:9090",job="prometheus"}

prometheus_tsdb_lowest_timestamp_minutes{instance="localhost:9090",job="prometheus"}
```

- The `_total` suffix should be appended, so `http_requests_total{handler="/", status="200"}` is better.
- There are two issues in `http_request_200_count{handler="/"}` : The `_count` suffix is foreseen for histograms, counters can be suffixed with `_total` . Second, status information should not be part of the metric name, a label `{status="200"}` is the better option.
- The base unit is `bytes` not `megabytes` , so `go_memstats_heap_inuse_bytes` is correct.
- Everything is ok with `prometheus_build_info` and its labels. It's a good practice to export such base information with a gauge.
- In `prometheus_config_last_reload_success_timestamp` , the base unit is missing, correct is `prometheus_config_last_reload_success_timestamp_seconds` .
- The base unit is `seconds` for timestamps, so `prometheus_tsdb_lowest_timestamp_seconds` is correct.

Task 6.1.3: Metric names (optional)

What kind of risk do you have, when you see such a metric

```
http_requests_total{path="/etc/passwd", status="404"} 1
```

There is no potential security vulnerability from exposing the `/etc/passwd` path, which seems to be handled appropriately in this case: no password is revealed.

From a Prometheus point of view, however, there is the risk of a DDoS attack: An attacker could easily make requests to paths which obviously don't exist. As every request and therefore path is registered with a label, many new time series are created which could lead to a cardinality explosion and finally to out-of-memory errors.

It's hard to recover from that!

For this case, it's better just to count the 404 requests and to lookup the paths in the log files.

```
http_requests_total{status="404"} 15
```

Task 6.1.4: Custom metric (optional)

In this lab you're going to create your own custom metric in the java Spring Boot application.

> **Note**
> This tasks requires that you have docker and git installed on your local machine. This counter is just a simple example for the sake of this lab. Those kind of metrics are provided by the micrometer Prometheus Spring Boot integration out of the box.

First we need to clone the repository to our local machine:

```
git clone https://github.com/acend/prometheus-training-spring-boot-example && \
cd prometheus-training-spring-boot-example
```

and then configure the dependencies and `application.properties` as described in Task 6.1.1.

Next, create a new CustomMetrics RestController class in your Spring Boot application `src/main/java/ch/acend/prometheustrainingspringbootexample/CustomMetricController.java` :

```
package ch.acend.prometheustrainingspringbootexample;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import io.micrometer.core.instrument.Counter;
import io.micrometer.core.instrument.MeterRegistry;

@RestController
public class CustomMetricController {

    private final Counter myCounter;
    private final MeterRegistry meterRegistry;

    @Autowired
    public CustomMetricController(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
        this.myCounter = meterRegistry.counter("my.prometheus.instrumentation.counter");
    }

    @GetMapping(value = "/api")
    public String getAll() {
        myCounter.increment();
        return "ok";
    }
}
```

We register our custom counter `myCounter` on the `MeterRegistry` in the constructor of the RestController.

Then we simply increase the counter every time the endpoint `/api` is hit. (just an example endpoint)

To build the application we will use the `Dockerfile` provided in the root folder of the repository.

```
docker build -t gitea.training.cluster.acend.ch/<user>/prometheus-training-spring-boot-example:latest .
```

Log into the docker registry of Gitea with your Gitea User credentials:

```
docker login gitea.training.cluster.acend.ch
```

And push the created image into the docker registry:

```
docker push gitea.training.cluster.acend.ch/<user>/prometheus-training-spring-boot-example:latest
```

Update the deployment in your `user-demo` folder:

```
---
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  [...]
  template:
    [...]
    spec:
      containers:
      - image: gitea.training.cluster.acend.ch/<user>/prometheus-training-spring-boot-example:latest
      [...]
```

Commit and push your changes.

Verify the metrics in your Prometheus UI .

7. Alerting with Alertmanager

Installation

Setup

You will install Alertmanager in the following labs. We will have a look at the default configuration in the next chapter.

Configuration in Alertmanager

Alertmanager's configuration is done using a YAML config file. There are two main sections for configuring how Alertmanager is dispatching alerts: receivers and routing.

Receivers

With a receiver , one or more notifications can be defined. There are different types of notifications types, e.g. mail, webhook, or one of the message platforms like Slack or PagerDuty.

Routing

With routing blocks , a tree of routes and child routes can be defined. Each routing block has a matcher which can match one or several labels of an alert. Per block, one receiver can be specified, or if empty, the default receiver is taken.

amtool

As routing definitions might be very complex and hard to understand, amtool becomes handy as it helps to test the rules. It can also generate test alerts and has even more useful features. More about this in the labs.

More (advanced) options

For more insights of the configuration options, study the following resources:

- Example configuration provided by Alertmanager on GitHub
- General overview of Alertmanager

Alerting rules in Prometheus

Prometheus alerting rules are configured very similarly to recording rules which you will get to know later in this training. The main difference is that the rules expression contains a threshold (e.g., `query_expression >= 5`) and that an alert is sent to the Alertmanager in case the rule evaluation matches the threshold. An alerting rule can be based on a recording rule or be a normal expression query.

> **Note**
> Sometimes the community or the maintainer of your Prometheus exporter already provide generic Prometheus alerting rules that can be adapted to your needs. For this reason, it makes sense to do some

research before writing alerting rules from scratch. Before implementing such a rule, you should always understand and verify the rule. Here are some examples:

- MySQL: mysqld-mixin
- Strimzi Kafka Operator: strimzi/strimzi-kafka-operator
- General rules for Kubernetes: kubernetes-mixin-ruleset
- General rules for various exporters: samber/awesome-prometheus-alerts

Templates for awesome rules

Whenever creating PrometheusRules you can always expect other people having the same problem as you. On this site you can find a collection of different PrometheusRules for a big amount of cloud native technology.

7.1 Tasks: Enable and configure Alertmanager

Task 7.1.1: Install Alertmanager and Thanosruler

Update your monitoring application (`charts/user-monitoring/values.yaml`) and update the `alertmanager.enabled` and `ruler.enabled` flag to `true` :

`charts/user-monitoring/values.yaml` :

```
user: <user> # Replace me
# prometheus
prometheus:
  enabled: true
# thanos-query
query:
  enabled: true
# grafana
grafana:
  enabled: true
  datasources:
  - name: prometheus
    access: proxy
    editable: false
    type: prometheus
    url: http://prometheus-operated:9090
# blackboxexporter
blackboxexporter:
  enabled: true
# pushgateway
pushgateway:
  enabled: true
# alertmanager
alertmanager:
  enabled: true
# thanos-ruler
ruler:
  enabled: true
```

Commit and push the changes.

Verify the installation and sync process in the ArgoCD UI . Or execute the following command:

```
kubectl -n $USER-monitoring get pod
```

This will install two Custom Resources (CR):

```
...
apiVersion: monitoring.coreos.com/v1
kind: Alertmanager
metadata:
  labels:
    app.kubernetes.io/name: alertmanager
  name: apps-monitoring
  namespace: <user>-monitoring
spec:
  alertmanagerConfigNamespaceSelector:
    matchNames:
    - <user>-monitoring
  alertmanagerConfigSelector:
  image: quay.io/prometheus/alertmanager:v0.25.0
  replicas: 2
  resources:
    requests:
      cpu: 4m
      memory: 40Mi
  storage:
    volumeClaimTemplate:
      spec:
        resources:
          requests:
            storage: 100Mi
```

```
---
apiVersion: monitoring.coreos.com/v1
kind: ThanosRuler
metadata:
  labels:
    app.kubernetes.io/name: thanos-ruler
  name: thanos-ruler
spec:
  image: quay.io/thanos/thanos:v0.28.1
  evaluationInterval: 10s
  queryEndpoints:
  - dnssrv+_http._tcp.thanos-query:10902
  ruleSelector: {}
  ruleNamespaceSelector:
    matchLabels:
      user: {{ .Values.user }}
  alertmanagersConfig:
    key: alertmanager-configs.yaml
    name: thanosruler-alertmanager-config
```

Task 7.1.3: Enable Alertmanager in Thanos Ruler

We connceted the thanos ruler to the Alertmanager instance with the following config.

```
---
apiVersion: v1
kind: Secret
metadata:
  name: thanosruler-alertmanager-config
stringData:
  alertmanager-configs.yaml: |-
    alertmanagers:
    - static_configs:
      - "dnssrv+_web._tcp.alertmanager-operated.<user>-monitoring.svc.cluster.local"
      api_version: v2
```

Task 7.1.4: Add Alertmanager as monitoring target in Prometheus

> **Note**
> This setup is only suitable for our lab environment. In real life, you must consider how to monitor your monitoring infrastructure: Having an Alertmanager instance as an Alertmanager AND as a target only in the same Prometheus is a bad idea!

This is repetition: The Alertmanagers (`alertmanager-operated.<user>-monitoring.svc:9093`) also exposes metrics, which can be scraped by Prometheus.

The ServiceMonitor telling Prometheus where to scrape the metrics of Alertmanager was already created by enabling the alertmanager.

Check in the Prometheus user interface if the target can be scraped.

Task 7.1.5: Query an Alertmanager metric

After you add the Alertmanager metrics endpoint, you will have huge bunch of different values and identifiers.

Use Querier UI to get the list of all available metrics. `{job="alertmanager-operated"}`

Then you get all metrics as follows (shortened), and you can pick whatever you're interested in.

```
# HELP alertmanager_alerts How many alerts by state.
# TYPE alertmanager_alerts gauge
alertmanager_alerts{state="active"} 0
alertmanager_alerts{state="suppressed"} 0
# HELP alertmanager_alerts_invalid_total The total number of received alerts that were invalid.
# TYPE alertmanager_alerts_invalid_total counter
alertmanager_alerts_invalid_total{version="v1"} 0
alertmanager_alerts_invalid_total{version="v2"} 0
# HELP alertmanager_alerts_received_total The total number of received alerts.
# TYPE alertmanager_alerts_received_total counter
alertmanager_alerts_received_total{status="firing",version="v1"} 0
alertmanager_alerts_received_total{status="firing",version="v2"} 0
alertmanager_alerts_received_total{status="resolved",version="v1"} 0
alertmanager_alerts_received_total{status="resolved",version="v2"} 0
...
```

Task 7.1.6: Alertmanager UI

Open the Alertmanager UI and explore its capabilities.

7.2 Tasks: Alertmanager

Task 7.2.1: Configure Slack as alert receiver

As part of our lab setup we already configured an alert receiver for Alertmanager alerts.

We deployed the Mail catcher , which is a very simple component. We can send Emails to the server, and those are then displayed in the Web UI. The Emails are not send anywhere, and this setup should only be used for demo purposes.

When we enabled the alertmanager configuration in the `charts/user-monitoring/values.yaml` we also deployed the following AlertmanagerConfig Custom Resource

```
---
apiVersion: monitoring.coreos.com/v1alpha1
kind: AlertmanagerConfig
metadata:
  name: <user>-mailcatcher
  labels:
    alertmanagerConfig: <user>-alertmanager
spec:
  route:
    groupBy: ['job']
    groupWait: 30s
    groupInterval: 5m
    repeatInterval: 12h
    receiver: 'mailcatcher'
  receivers:
    - name: 'mailcatcher'
      emailConfigs:
        - to: alert@localhost
          from: prometheus-operator@localhost
          smarthost: mailcatcher:1025
          requireTLS: false
```

When an alert is firing it will send an email to the Mail catcher.

Task 7.2.2: Send a test alert

In this task you can use the amtool command to send a test alert.

To send a test alert with the labels `alertname=Up` and `node=bar` you can simply execute the following command.

```
kubectl --namespace $USER-monitoring exec -it statefulsets/alertmanager-$USER-alertmanager -c alertmanager -- sh
amtool alert add --alertmanager.url=http://localhost:9093 alertname=Up node=bar
```

Check in the Alertmanager UI if you see the test alert with the correct labels set. **Info:** this will not yet send an email to the Mail catcher. You'll find out why in the end of this lab.

Task 7.2.3: Show the routing tree

Show routing tree:

```
kubectl --namespace $USER-monitoring exec -it statefulsets/alertmanager-$USER-alertmanager -c alertmanager -- sh
amtool config routes --alertmanager.url=http://localhost:9093
```

Depending on the configured receivers your output might vary.

If you only configured the email receiver, the output will look similar to this:

```
kubectl --namespace $USER-monitoring exec -it statefulsets/alertmanager-$USER-alertmanager -c alertmanager -- sh
amtool config routes --config.file /etc/alertmanager/alertmanager.yml
Routing tree:

└── default-route  receiver: default
    ├── {namespace="<user>-monitoring"}  continue: true  receiver: <user>-monitoring/<user>-mailcatcher/mailcatcher
```

Task 7.2.5: Test your alert receivers

Add a test alert and check if alert has been sent to the Mail catcher . It can take up to 5 minutes as the alarms are grouped together based on the group_interval .

```
kubectl --namespace $USER-monitoring exec -it statefulsets/alertmanager-$USER-alertmanager -c alertmanager -- sh
amtool alert add --alertmanager.url=http://localhost:9093 alertname=snafu env=dev severity=critical
```

It is also advisable to validate the routing configuration against a test dataset to avoid unintended changes. With the option `--verify.receivers` the expected output can be specified:

```
kubectl --namespace $USER-monitoring exec -it statefulsets/alertmanager-$USER-alertmanager -c alertmanager -- sh
amtool config routes test --alertmanager.url=http://localhost:9093 --verify.receivers=<user>-monitoring/<user>-mailcatc
her/mailcatcher env=dev severity=info
```

```
default
WARNING: Expected receivers did not match resolved receivers.
```

Only alerts with the `namespace=<user>-monitoring` will be routed to our mailcatcher.

```
kubectl --namespace $USER-monitoring exec -it statefulsets/alertmanager-$USER-alertmanager -c alertmanager -- sh
amtool config routes test --config.file /etc/alertmanager/alertmanager.yml --verify.receivers=<user>-monitoring/<user>-
mailcatcher/mailcatcher env=dev namespace=<user>-monitoring
```

7.3 Tasks: Alertrules and alerts

> **Note**
> For doing the alerting lab it's useful to have a "real" application so that alerts can be provoked. You will use the demo app installed in your monitoring-demo namespace for this purpose.
>
> The example app exposes metrics which are already scraped from lab one.

```
{job="example-web-python"}
```

The Prometheus Operator allows you to configure Alerting Rules (PrometheusRules). This enables Kubernetes users to configure and maintain alerting rules for their projects. Furthermore it is possible to treat Alerting Rules like any other Kubernetes resource and lets you manage them in Helm or Kustomize. A PrometheusRule has the following form:

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: <resource-name>
spec:
  <rule definition>
```

See the Alertmanager documentation for `<rule definition>`

Task 7.3.1: Add an alerting rule for crashlooping pods

- Define an alerting rule which sends an alert when a pod in your namespace is crashlooping at least once in the last 5 minutes
- New alarms should be in `pending` state for 15 minutes before they transition to firing
- Add a label `severity` with the value `info`
- Add an annotation `summary` with information about which pods and job is down

To add an Alerting rule, create a PrometheusRule resource `user-demo/training_testrules.yaml` in the monitoring folder of your repository.

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: testrules
spec:
  groups:
    - name: pod-rules
      rules:
        - alert: kubePodCrashLooping
          expr: rate(kube_pod_container_status_restarts_total{job="kube-state-metrics",namespace="<user>-monitoring"}[5
m]) * 60 * 5 > 0
          for: 15m
          annotations:
            message: Pod {{ $labels.namespace }}/{{ $labels.pod }} ({{ $labels.container }}) is restarting {{ printf "%
.2f" $value }} times / 5 minutes.
          labels:
            severity: info
```

You can build/verify your Query in your Thanos Querier UI . As soon, as you apply the PrometheusRule resource, you should be able to see the alert in your Thanos Ruler implementation.

Task 7.3.2: Add a an alert for absent targets

Task description:

- Define an alerting rule which sends an alert when a target is down. Remember the `up` metric?
- New alarms should be in `pending` state for 2 minutes before they transition to firing
- Add a label `user` with the value `<user>`
- Add an annotation `summary` with information about which instance and job is down

Either create a separate PrometheuRule resource as you did before, or add the group definition below to your existing PrometheusRule resource.

```
groups:
  - name: job-rules
    rules:
      - alert: target-down
        expr: up == 0
        for: 2m
        labels:
          user: <user>
        annotations:
          summary: Instance {{ $labels.instance }} of job {{ $labels.job }} is down
```

The value in field `for` is the wait time until the active alert gets in state `FIRING`. Before that, the alert is `PENDING` and not yet sent to Alertmanager.

The alert is instrumented with the labels from the metric (e.g. `job` and `instance`). Additional labels can be defined in the rule. Labels can be used in Alertmanager for the routing.

With annotations, additional human-readable information can be attached to the alert.

- In the Prometheus web UI there is an **Alerts** menu item which shows you information about the alerts.

Task 7.3.3: Verify the target down alert

In this task you're going to explore what happens, when a rule fires and sends an alert.

First we add the following rule to our rules, commit and push the changes to git:

```
groups:
  - name: test-rules
    rules:
      - alert: memory-usage-to-high
        expr: container_memory_usage_bytes{namespace="user6", pod=~"example-spring-boot.*", image!="", container!=""} /
 1024 / 1004 > 256
        for: 1m
        labels:
          user: <user>
        annotations:
          summary: Java App {{ $labels.namespace }}/{{ $labels.pod }} ({{ $labels.container }}) uses more than 256 Mi M
emory
```

We know that our Spring boot example from the Lab 6 uses more than that. If not, we should decrease the threshold.

- The Thanos Ruler UI **Alerts** [menu item](Thanos Ruler /alerts) shows you information about inactive and active alerts.
- As soon as an alert is in state `FIRING` the alert is sent to Alertmanager. You should see the alert in its web UI . This might take some time.

Task 7.3.4: Silencing alerts

Sometimes the huge amount of alerts can be overwhelming, or you're currently working on fixing an issue, which triggers an alert. Or you're simply testing something that fires alerts.

In such cases alert **silencing** can be very helpful.

Let's now silence our test alert.

Open the Alertmanger web UI and search for the test alert.

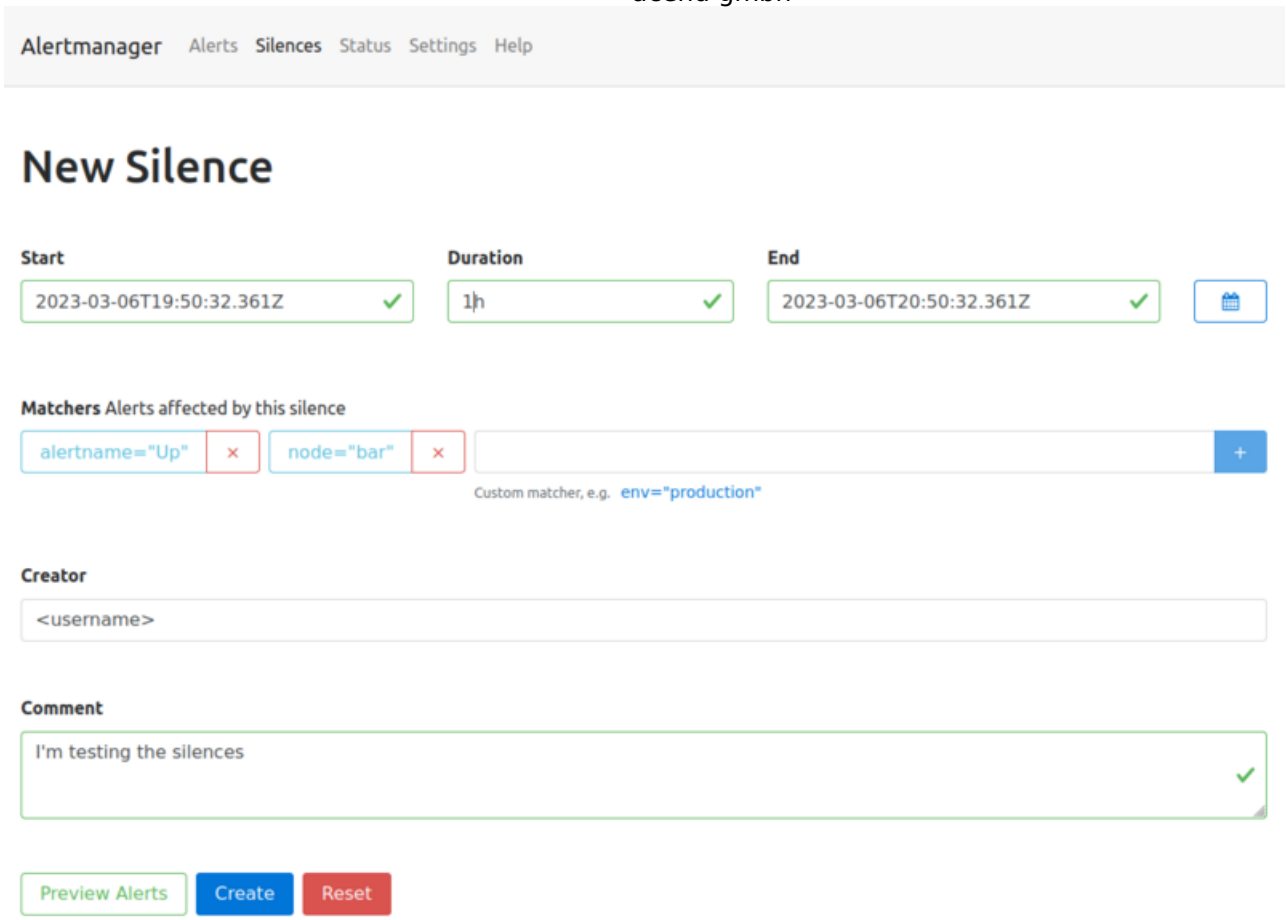
> **Note**
> The alert might have been resolved already, use the following command to re-trigger it again:

```
kubectl --namespace $USER-monitoring exec -it statefulsets/alertmanager-$USER-alertmanager -c alertmanager -- sh
amtool alert add --alertmanager.url=http://localhost:9093 alertname=Up node=bar namespace=<user>-monitoring
```

You can either silence the specific alert by simply clicking on the `Silcence` button next to the alert, or create a new silence by clicking the `New Silence` button in the top menu on the right. Either way, you'll end up on the same form. The button next to the alert will conveniently fill out the matchers, so that the alert will be affected by the new silence.

- Click the `Silence` button next to the test alert.
- Make sure the matchers contains the two labels (`alertname="Up"` , `node="bar"`) of the test alert.
- Set the duration to 1h
- Add your username to the creator form field.
- Fill out the description with the reason you're creating a silence.

You can then use the `Preview Alerts` button to check your matchers and create the alert by clicking `create` .



All alerts, which match the defined labels of the matcher will be silenced for the defined time slot.

Go back to the Alerts page, the silenced alert disappeared and will only be visible when checking the silenced alerts checkbox.

The top menu entry silence will show you a list of the created silences. Silences can also be created programmatically using the API or the amtool (`amtool silence --help`).

The following command is exactly the same you just did via the Web UI:

```
kubectl --namespace $USER-monitoring exec -it statefulsets/alertmanager-$USER-alertmanager -c alertmanager -- sh
amtool silence add alertname=Up node=bar --author="<username>" --comment="I'm testing the silences" --alertmanager.url=
http://localhost:9093
```

8. Advanced Topics

This section covers different advanced topis.

8.1 Tasks: Relabeling

Relabeling in Prometheus can be used to perform numerous tasks using regular expressions, such as

- adding, modifying or removing labels to/from metrics or alerts,
- filtering metrics based on labels, or
- enabling horizontal scaling of Prometheus by using `hashmod` relabeling.

It is a very powerful part of the Prometheus configuration, but it can also get quite complex and confusing. Thus, we will only take a look at some basic/simple examples.

There are four types of relabelings:

- `relabel_configs` (target relabeling)

  Target relabeling is defined in the job definition of a `scrape_config`. When using the Prometheus Operator, custom `relabel_configs` can be added to the `ServiceMonitor`. This concept is also used to configure scraping of a multi-target exporter (e.g., `blackbox_exporter` or `snmp_exporter`) where one single exporter instance is used to scrape multiple targets. Check out the Prometheus docs for a detailed explanation and example configurations of `relabel_configs`.

- `metric_relabel_configs` (metrics relabeling)

  Metrics relabeling is applied to scraped samples right before ingestion. It allows adding, modifying, or dropping labels or even dropping entire samples if they match certain criteria.

- `alert_relabel_configs` (alert relabeling)

  Alert relabeling is similar to `metric_relabel_configs`, but applies to outgoing alerts.

- `write_relabel_configs` (remote write relabeling)

  Remote write relabeling is similar to `metric_relabel_configs`, but applies to `remote_write` configurations.

8.2 Tasks: Recording Rules

Prometheus recording rules allow you to precompute queries at a defined interval (`global.evaluation_interval` or `interval` in `rule_group`) and save them to a new set of time series.

In this lab you are going to create your first own recording rules. Recording rules are very useful when it comes to queries, which are very complex and take a long time to compute. The naming convention dictates to use the following format when naming recording rules `level:metric:operation` . Additional information regarding naming best-practices can be found here .

> **Warning**
> Recording rules store the result in a new series and they can add additional complexity.

Task 8.2.1: Memory usage recording rule

With the following recording rule, we create a new metric that represents the available memory on a node as a percentage. A metric the `node exporter` doesn't expose when running on a machine with an older Linux kernel and needs to be calculated every time.

- Query the recording rule in the Prometheus web UI
- Add the following recording rule file `user-demo/training_prometheusrule_avail_memory.yaml` to your directory, commit and push your changes.

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: prometheusrule-avail-memory
spec:
  groups:
    - name: node_memory
      rules:
        - record: :node_memory_MemAvailable_bytes:sum
          expr: |
            (1 - (
              sum by(instance) (node_memory_MemFree_bytes
              +
              node_memory_Cached_bytes
              +
              node_memory_Buffers_bytes
            )
            )
            /
              sum by(instance) (node_memory_MemTotal_bytes))
            * 100
```

Commit and push the changes.

After configuring the recording rule and reloading the configuration, Prometheus provides those metrics accordingly.

> **Note**
> It may take up to one minute for the recording rule to become available.

Use your `recording_rule` definition in the expression browser:

```
:node_memory_MemAvailable_bytes:sum
```

or hit the following link

> **Note**
> If you take a look at the historical metrics, you will notice that there is no backfilling (by default) of your data. Only data since activation of the recording rule is available. Optional backfilling can be accomplished by using the `promtool` utility

> **Note**
> Perhaps you have noticed that the rule name starts with a colon. While this may seem odd at first sight, this is actually the result of following the naming convention mentioned above. The rule does not aggregate over a certain `level` and therefore the first field of `level:metric:operation` remains empty.

Task 8.2.2: CPU utilization recording rule

In this lab you are going to create a CPU utilization recording rule.

- Create a rule to record the **CPU utilization** of your server
- Make sure that Prometheus evaluates this rule every **60 seconds**
- Verify in the web UI that you can query your recording rule

As you saw in a previous exercise, the `node_cpu_seconds_total` metric contains the CPU utilization of a node. We can use the `mode` label on this metric to filter for `idle` cpu time.

All other modes than `idle` indicate, that the CPU is used. Therefore we can simply subtract the idle percentage from 100 % and get the value we want.

- Add the following recording rule file `user-demo/training_recording_rule_cpu_usage.yaml` to your directory, commit and push your changes.

```
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: recording-rule-cpu-usage
spec:
  groups:
    - name: node_cpu
      interval: 60s
      rules:
        - record: instance:node_cpu_utilisation:rate5m
          expr: |
            100 - (
              avg by (instance) (rate(node_cpu_seconds_total{mode="idle"}[5m]))
              * 100
            )
```

Commit and push the changes.

Query your recording rule using the expression browser

8.3 Tasks: Troubleshoot Kubernetes Service Discovery

Task 8.3.1: Troubleshooting Kubernetes Service Discovery

We will now deploy an application with an error in the monitoring configration.

- Deploy Loki in your namespace by adding the following files to your git repo in the `user-demo/` folder:

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: loki
  name: loki
spec:
  replicas: 1
  selector:
    matchLabels:
      app: loki
  template:
    metadata:
      labels:
        app: loki
    spec:
      containers:
      - image: mirror.gcr.io/grafana/loki:latest
        name: loki
```

```
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: loki
  name: loki
spec:
  ports:
    - name: http
      port: 3100
      protocol: TCP
      targetPort: 3100
  selector:
    app: loki
  type: NodePort
```

```
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
    app.kubernetes.io/name: loki
  name: loki
spec:
  endpoints:
    - interval: 30s
      port: http
      scheme: http
      path: /metrics
  selector:
    matchLabels:
      prometheus-monitoring: 'true'
```

Add, commit and push changes to your git repository and let ArgoCD sync your app again.

- When you visit the Prometheus user interface you will notice, that the Prometheus Server does not scrape metrics from Loki. Try to find out why.

Troubleshooting: Prometheus is not scrapping metrics

The cause that Prometheus is not able to scrape metrics is usually one of the following.

- The configuration defined in the ServiceMonitor does not appear in the Prometheus scrape configuration
  - Check if the label of your ServiceMonitor matches the label defined in the `serviceMonitorSelector` field of the Prometheus custom resource
  - Check the Prometheus operator logs for errors (Permission issues or invalid ServiceMonitors)
- The Endpoint appears in the Prometheus scrape config but not under targets.
  - The namespaceSelector in the ServiceMonitor does not match the namespace of your app
  - The label selector does not match the Service of your app
  - The port name does not match the Service of your app
- The Endpoint appears as a Prometheus target, but no data gets scraped
  - The application does not provide metrics under the correct path and port
  - Networking issues
  - Authentication required, but not configured

The quickest way to do this is to follow the instructions in the info box above. So let’s first find out which of the following statements apply to us

- The configuration defined in the ServiceMonitor does not appear in the Prometheus scrape configuration
  - Let’s check if Prometheus reads the configuration defined in the ServiceMonitor resource. To do so navigate to Prometheus configuration and search if `loki` appears in the scrape_configuration. You should find a job with the name `serviceMonitor/loki/loki/0`, therefore this should not be the issue in this case.
- The Endpoint appears in the Prometheus configuration but not under targets.
  - The namespaceSelector in the ServiceMonitor does not match the namespace of your app
  - The label selector does not match the Service of your app
  - The port name does not match the Service of your app
- The Endpoint appears as a Prometheus target, but no data gets scraped
  - The application does not provide metrics under the correct path and port
  - Networking issues
  - Authentication required, but not configured

The quickest way to do this is to follow the instructions in the info box above. So let’s first find out which of

- acend gmbh

the following statements apply to us

- The configuration defined in the ServiceMonitor does not appear in the Prometheus scrape configuration
  - Let's check if Prometheus reads the configuration defined in the ServiceMonitor resource. To do so navigate to Prometheus configuration and search if `loki` appears in the scrape_configuration. You should find a job with the name `serviceMonitor/loki/loki/0` , therefore this should not be the issue in this case.
- The Endpoint appears in the Prometheus configuration but not under targets.
  - Lets check if the application is running

```
kubectl get pod
```

You should see a loki Pod in the `Running` state:

```
NAME                   READY   STATUS    RESTARTS   AGE
loki-5846d87f4c-tthsr   1/1     Running   0          34m
```

  - Lets check if the application is exposing metrics

```
PODNAME=$(kubectl get pod -l app=loki -o name)
kubectl exec $PODNAME -it -- wget -O - localhost:3100/metrics
...
```

  - The application exposes metrics and Prometheus generated the configuration according to the defined servicemonitor. Let's verify, if the ServiceMonitor matches the Service.

```
kubectl get svc loki -o yaml
```

```
apiVersion: v1
kind: Service
metadata:
  ...
  labels:
    app: loki
  name: loki
spec:
  ...
  ports:
  - name: http
    ...
```

We see that the Service has the port named `http` and the label `app: loki` set. Let's check the ServiceMonitor

```
kubectl get servicemonitor loki -o yaml
```

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
...
spec:
  endpoints:
  - interval: 30s
    ...
    port: http
    ...
  selector:
    matchLabels:
      prometheus-monitoring: "true"
```

We see that the ServiceMonitor expect the port named `http` and a label `prometheus-monitoring: "true"` set. So the culprit is the missing label. Let's adjust the service manifest, commit and push.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: loki
    prometheus-monitoring: "true"
  name: loki
spec:
  ports:
    - name: http
      port: 3100
      protocol: TCP
      targetPort: 3100
  selector:
    app: loki
  type: NodePort
```

Verify that the target now gets scraped in the Prometheus user interface .