kubernetes

Lev Epshtein
lev@opsguru.io

NAYA College

Kubernetes

Basic part II

- Helm
- Role-based Access Control
- Prometheus
- Istio

NAYA College

Helm Chart

K8S Helm Charts

A Helm chart is a package that contains all the necessary resources to deploy an application to a Kubernetes cluster. This includes YAML configuration files for deployments, services, secrets, and config maps that define the desired state of your application.

A Helm chart packages together YAML files and templates that can be used to generate additional configuration files based on **parametrized values**. This allows you to customize configuration files to suit different environments and to create reusable configurations for use across multiple deployments. Additionally, each Helm chart can be versioned and managed independently, making it easy to maintain multiple versions of an application with different configurations.

Releases

A running instance of a chart is known as a release. When you run the **helm install** command, it pulls the config and chart files and deploys all the Kubernetes resources.

helm install bitnami/mysql --generate-name

Architecture

Helm’s architecture has two main components: client and library.

A Helm client is a command-line utility for end users to control local chart development and manage repositories and releases. Just like using the MySQL database client to run MySQL commands, you use the Helm client to run Helm commands.

The Helm library does all the heavy lifting. It contains the actual code to perform the operations specified in the Helm command. The combination of config and chart files to create any release is handled by the Helm library.

How Helm works

The Helm application library uses charts to define, create, install, and upgrade Kubernetes applications. Helm charts allow you to manage Kubernetes manifests without using the Kubernetes command-line interface (CLI) or remembering complicated Kubernetes commands to control the cluster.

How Helm works

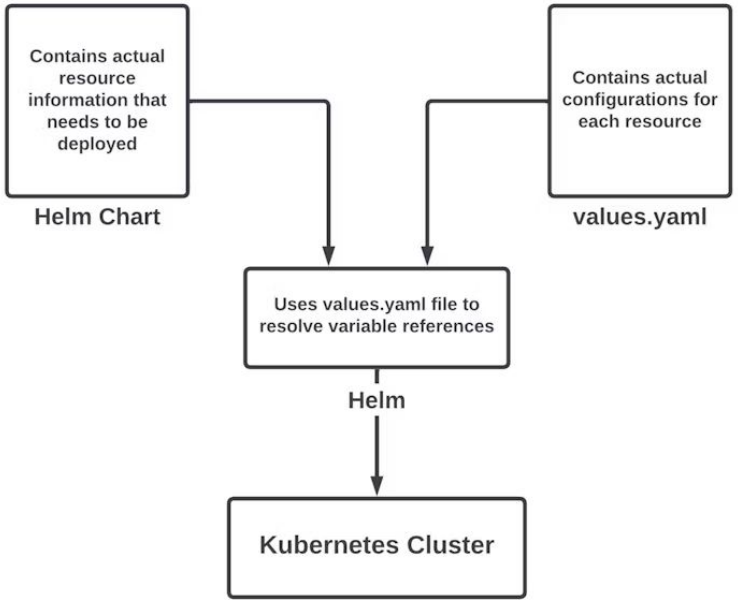
Consider a practical scenario where Helm is helpful. Suppose you want to deploy your application in a production environment with ten replicas. You specify this in the deployment YAML file for the application and run the deployment using the kubectl command.

Now, run the same application in a staging environment. Assume that you need three replicas in staging and that you will run an internal application build in the staging environment. To do this, update the replicas count and the Docker image tag in the deployment YAML file and then use it in the staging Kubernetes cluster.

As your application becomes more complex, the number of YAML files increases. Eventually, the configurable fields in the YAML file also increase. Soon, updating many YAML files to deploy the same app in different environments becomes hard to manage.

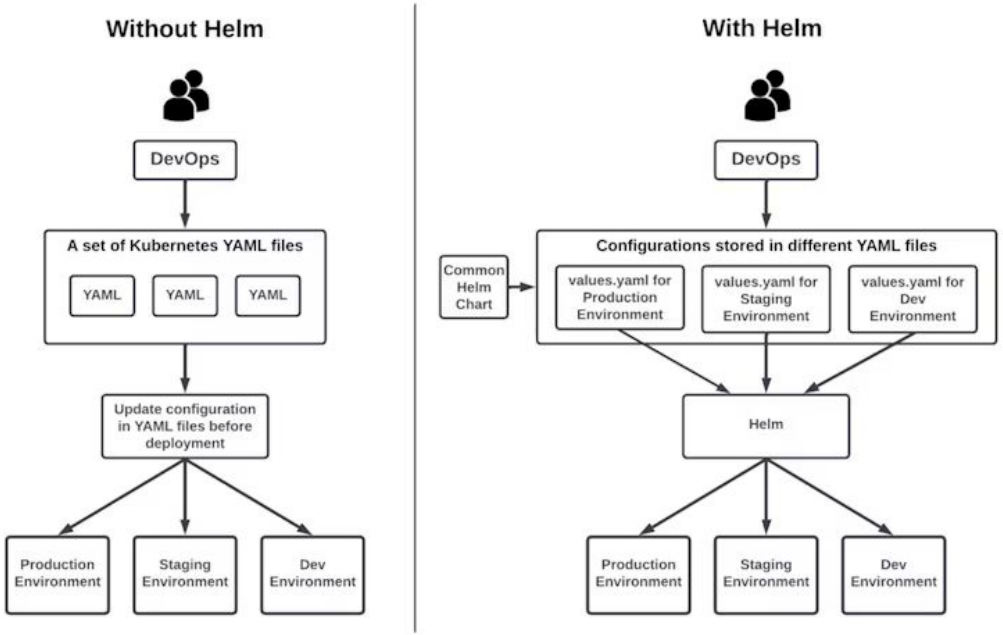
How Helm works

Using Helm, you can parameterize the fields depending on the environment. In the previous example, instead of using a static value for replicas and Docker images, you can take the value for these fields from another file. This file is called values.yaml.



How Helm works

Now, you can maintain a values file for each environment with the proper values for each. Helm helps you decouple the configurable field values from the actual YAML configuration.



Helm repositories

The Helm repository is where you can upload Helm charts. You can also create a private repository to share charts within your organization. Artifact Hub is a global Helm repository that features searchable charts that you can install for numerous purposes. In short, Artifact Hub does for Helm charts what Docker Hub does for Docker images.

Using Helm chart

Common actions for Helm:

- helm search: search for charts
- helm pull: download a chart to your local directory to view
- helm install: upload the chart to Kubernetes
- helm list: list releases of charts

The Chart File Structure

A chart is organized as a collection of files inside of a directory. The directory name is the name of the chart (without versioning information). Thus, a chart describing WordPress would be stored in a wordpress/ directory.

```
wordpress/
  Chart.yaml          # A YAML file containing information about the chart
  LICENSE             # OPTIONAL: A plain text file containing the license for the chart
  README.md           # OPTIONAL: A human-readable README file
  values.yaml         # The default configuration values for this chart
  values.schema.json  # OPTIONAL: A JSON Schema for imposing a structure on the values.yaml file
  charts/             # A directory containing any charts upon which this chart depends.
  crds/               # Custom Resource Definitions
  templates/          # A directory of templates that, when combined with values,
                      # will generate valid Kubernetes manifest files.
  templates/NOTES.txt # OPTIONAL: A plain text file containing short usage notes
```

The Chart.yaml File

The `Chart.yaml` file is required for a chart. It contains the following fields:

```yaml
apiVersion: The chart API version (required)
name: The name of the chart (required)
version: A SemVer 2 version (required)
kubeVersion: A SemVer range of compatible Kubernetes versions (optional)
description: A single-sentence description of this project (optional)
type: The type of the chart (optional)
keywords:
  - A list of keywords about this project (optional)
home: The URL of this projects home page (optional)
sources:
  - A list of URLs to source code for this project (optional)
dependencies: # A list of the chart requirements (optional)
  - name: The name of the chart (nginx)
    version: The version of the chart ("1.2.3")
    repository: (optional) The repository URL ("https://example.com/charts") or alias ("@repo-name")
    condition: (optional) A yaml path that resolves to a boolean, used for enabling/disabling charts (e.g. subchart1.enabled )
    tags: # (optional)
      - Tags can be used to group charts for enabling/disabling together
    import-values: # (optional)
      - ImportValues holds the mapping of source values to parent key to be imported. Each item can be a string or pair of child/parent sublist items.
    alias: (optional) Alias to be used for the chart. Useful when you have to add the same chart multiple times
maintainers: # (optional)
  - name: The maintainers name (required for each maintainer)
    email: The maintainers email (optional for each maintainer)
    url: A URL for the maintainer (optional for each maintainer)
icon: A URL to an SVG or PNG image to be used as an icon (optional).
appVersion: The version of the app that this contains (optional). Needn't be SemVer. Quotes recommended.
deprecated: Whether this chart is deprecated (optional, boolean)
annotations:
  example: A list of annotations keyed by name (optional).
```

NAYA College

Templates and Values

- Helm Chart templates are written in the Go template language, with the addition of 50 or so add-on template functions from the Sprig library and a few other specialized functions.
- All template files are stored in a chart's templates/ folder. When Helm renders the charts, it will pass every file in that directory through the template engine.
- Values for the templates are supplied two ways:
  - Chart developers may supply a file called values.yaml inside of a chart. This file can contain default values.
  - Chart users may supply a YAML file that contains values. This can be provided on the command line with **helm install**.

Template Files

- Template files follow the standard conventions for writing Go templates (see the text/template Go package documentation for details). An example template file might look something like this

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: deis-database
  namespace: deis
  labels:
    app.kubernetes.io/managed-by: deis
spec:
  replicas: 1
  selector:
    app.kubernetes.io/name: deis-database
  template:
    metadata:
      labels:
        app.kubernetes.io/name: deis-database
    spec:
      serviceAccount: deis-database
      containers:
        - name: deis-database
          image: {{ .Values.imageRegistry }}/postgres:{{ .Values.dockerTag }}
          imagePullPolicy: {{ .Values.pullPolicy }}
          ports:
            - containerPort: 5432
          env:
            - name: DATABASE_STORAGE
              value: {{ default "minio" .Values.storage }}
```

Chart Hooks

Helm provides a *hook* mechanism to allow chart developers to intervene at certain points in a release's life cycle. For example, you can use hooks to:

- Load a ConfigMap or Secret during install before any other charts are loaded.
- Execute a Job to back up a database before installing a new chart, and then execute a second job after the upgrade in order to restore data.
- Run a Job before deleting a release to gracefully take a service out of rotation before removing it.

Annotation Value	Description
pre-install	Executes after templates are rendered, but before any resources are created in Kubernetes
post-install	Executes after all resources are loaded into Kubernetes
pre-delete	Executes on a deletion request before any resources are deleted from Kubernetes
post-delete	Executes on a deletion request after all of the release's resources have been deleted
pre-upgrade	Executes on an upgrade request after templates are rendered, but before any resources are updated
post-upgrade	Executes on an upgrade request after all resources have been upgraded
pre-rollback	Executes on a rollback request after templates are rendered, but before any resources are rolled back
post-rollback	Executes on a rollback request after all resources have been modified
test	Executes when the Helm test subcommand is invoked (view test docs)

NAYA College

Pros and cons of using Helm

When to use Helm

Helm is helpful when your project uses Kubernetes to run complex applications with many microservices. Using Helm, you can easily automate the deployment and management of the application, reducing the amount of manual work and improving the reliability and stability of the system. Helm also provides access to an extensive repository of preconfigured packages, making it easy to add new features and functionality to the application.

Pros and cons of using Helm

When to use Helm

By organizing the application’s components into modular charts that you can easily install and upgrade, Helm simplifies the process of managing application components. It can reduce the amount of manual work required to maintain the application and helps you avoid errors and inconsistencies that can arise when managing complex systems manually.

Pros and cons of using Helm

When to use Helm

Helm also supports the deployment of containers across multiple environments, such as development, staging, and production, making it easy to manage the lifecycle of containers throughout the development process.

Pros and cons of using Helm

When Helm does not excel

Helm is not well-suited to projects where a single container needs to be deployed on a server. In this case, using Helm to manage the deployment of the container would be unnecessary and could even add complexity to the process. Since Helm is designed to manage multiple container deployments as a single unit, it would not be helpful in this scenario.

Pros and cons of using Helm

When Helm does not excel

If you have a small number of Kubernetes applications and can manage them manually without needing a package manager, using Helm may not provide significant benefits.

Finally, if your organization has strict security policies that prevent using third-party tools like Helm, then it may not be possible to use Helm in your environment.

Demo/Labs

K8S Helm Chart:

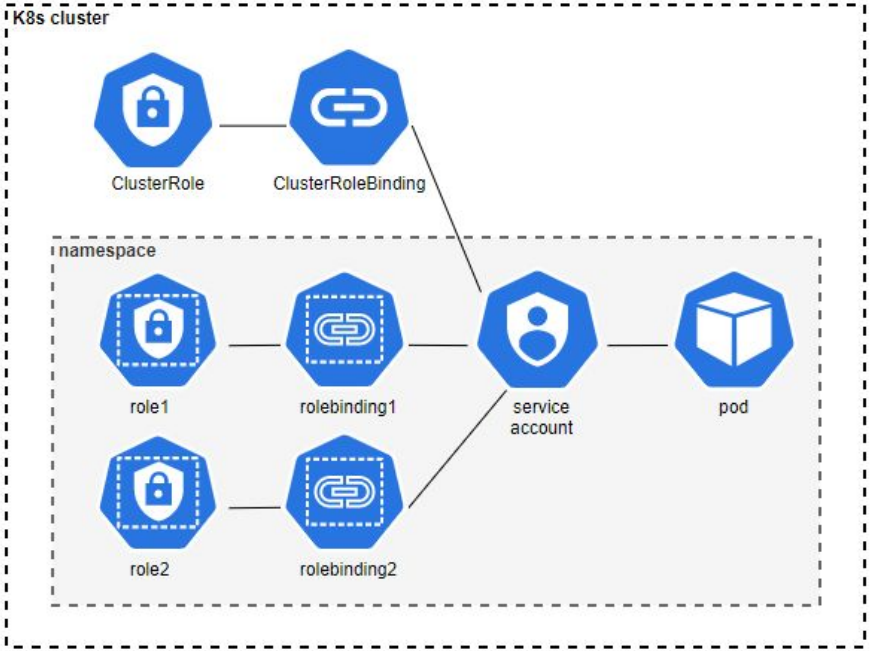
In our repository

$ cd solaredge-k8s-course/k8s-3/helm-tutorial/

Role-based Access Control

Role-based Access Control

In Kubernetes, granting roles to a user or an application-specific service account is a best practice to ensure that your application is operating in the scope that you have specified.



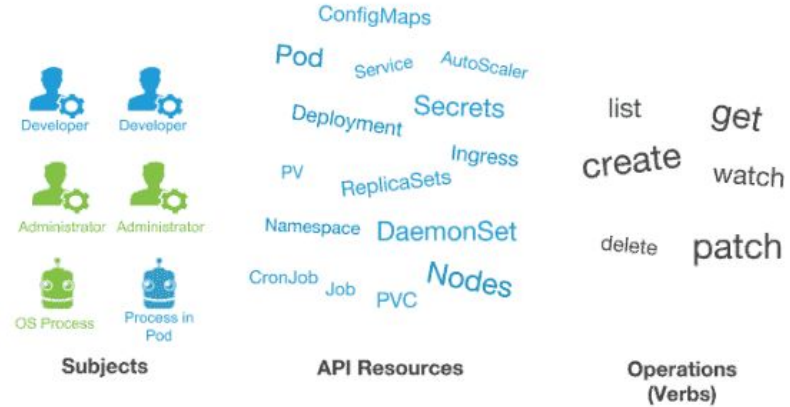
The key to understanding RBAC in Kubernetes

In order to fully grasp the idea of RBAC, we must understand that three elements are involved:

Subjects: The set of users and processes that want to access the Kubernetes API.

Resources: The set of Kubernetes API Objects available in the cluster. Examples include Pods, Deployments, Services, Nodes, and PersistentVolumes, among others.

Verbs: The set of operations that can be executed to the resources above. Different verbs are available (examples: get, watch, create, delete, etc.), but ultimately all of them are Create, Read, Update or Delete (CRUD) operations.



NAYA College

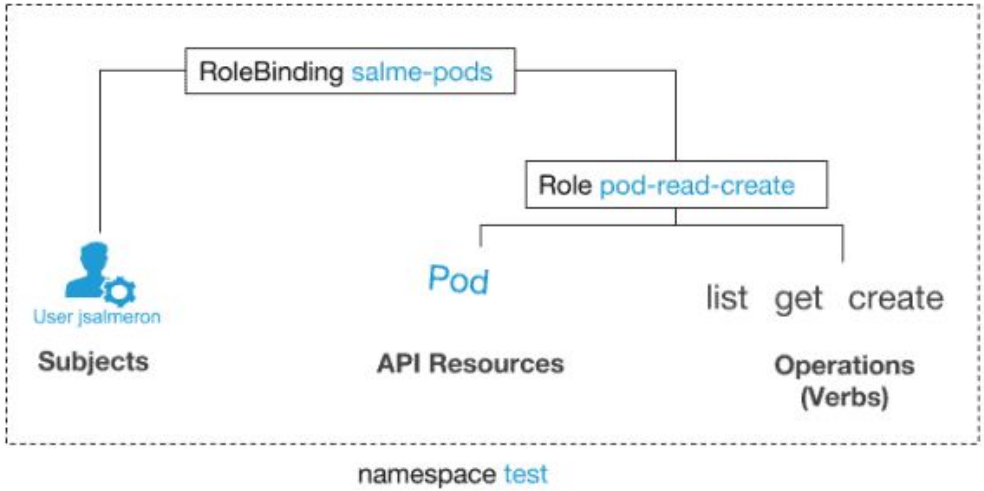
Understanding RBAC API objects

Roles: Will connect API Resources and Verbs. These can be reused for different subjects. These are binded to one namespace (we cannot use wildcards to represent more than one, but we can deploy the same role object in different namespaces). If we want the role to be applied cluster-wide, the equivalent object is called **ClusterRoles**.

RoleBinding: Will connect the remaining entity-subjects. Given a role, which already binds API Objects and verbs, we will establish which subjects can use it. For the cluster-level, non-namespaced equivalent, there are **ClusterRoleBindings**.

```yaml
kind: Role
apiVersion:
rbac.authorization.k8s.io/v1beta1
metadata:
   name: pod-read-create
   namespace: test
rules:
 - apiGroups: [""]
   resources: ["pods"]
   verbs: ["get", "list", "create"]
```

```yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: salme-pods
  namespace: test
subjects:
- kind: User
  name: jsalmeron
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: ns-admin
  apiGroup: rbac.authorization.k8s.io
```



NAYA College

Users and… ServiceAccounts?

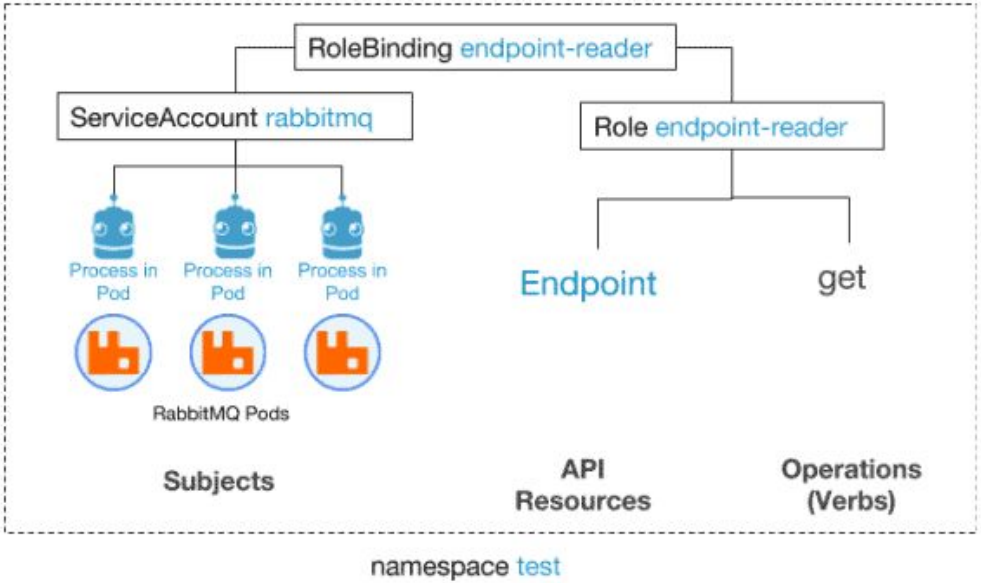
Users: These are global, and meant for humans or processes living outside the cluster.

ServiceAccounts: These are namespaced and meant for intra-cluster processes running inside pods.

Have **ServiceAccounts** per deployment with the **minimum set of privileges** to work.

A ServiceAccount for the RabbitMQ pods - Example

The diagram shows how we enabled the processes running in the RabbitMQ pods to perform “get” operations over Endpoint objects. This is the minimum set of operations it requires to work. So, at the same time, we are ensuring that the deployed chart is secure and will not perform unwanted actions inside the Kubernetes cluster.



A ServiceAccount for the RabbitMQ pods - Example

```
{{- if .Values.rbacEnabled }}
apiVersion: v1
kind: ServiceAccount
metadata:
  name: {{ template "rabbitmq.fullname" . }}
  labels:
    app: {{ template "rabbitmq.name" . }}
    chart: {{ template "rabbitmq.chart" .  }}
    release: "{{ .Release.Name }}"
    heritage: "{{ .Release.Service }}"
{{- end }}
```

```
{{- if .Values.rbacEnabled }}
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: {{ template "rabbitmq.fullname" . }}-endpoint-reader
  labels:
    app: {{ template "rabbitmq.name" . }}
    chart: {{ template "rabbitmq.chart" .  }}
    release: "{{ .Release.Name }}"
    heritage: "{{ .Release.Service }}"
rules:
- apiGroups: [""]
  resources: ["endpoints"]
  verbs: ["get"]
{{- end }}
```

```
{{- if .Values.rbacEnabled }}
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: {{ template "rabbitmq.fullname" . }}-endpoint-reader
  labels:
    app: {{ template "rabbitmq.name" . }}
    chart: {{ template "rabbitmq.chart" .  }}
    release: "{{ .Release.Name }}"
    heritage: "{{ .Release.Service }}"
subjects:
- kind: ServiceAccount
  name: {{ template "rabbitmq.fullname" . }}
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: {{ template "rabbitmq.fullname" . }}-endpoint-reader
{{- end }}
```

NAYA College

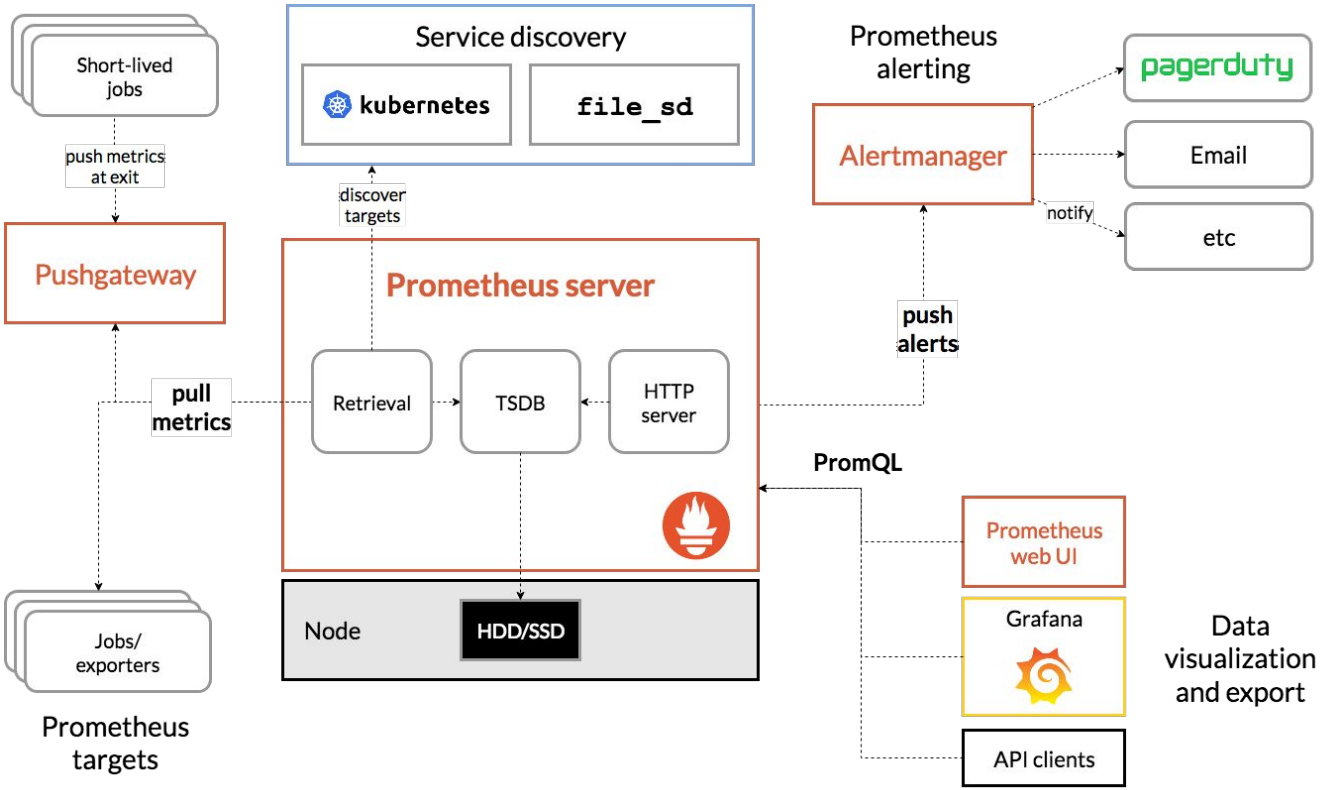
Prometheus

What is Prometheus

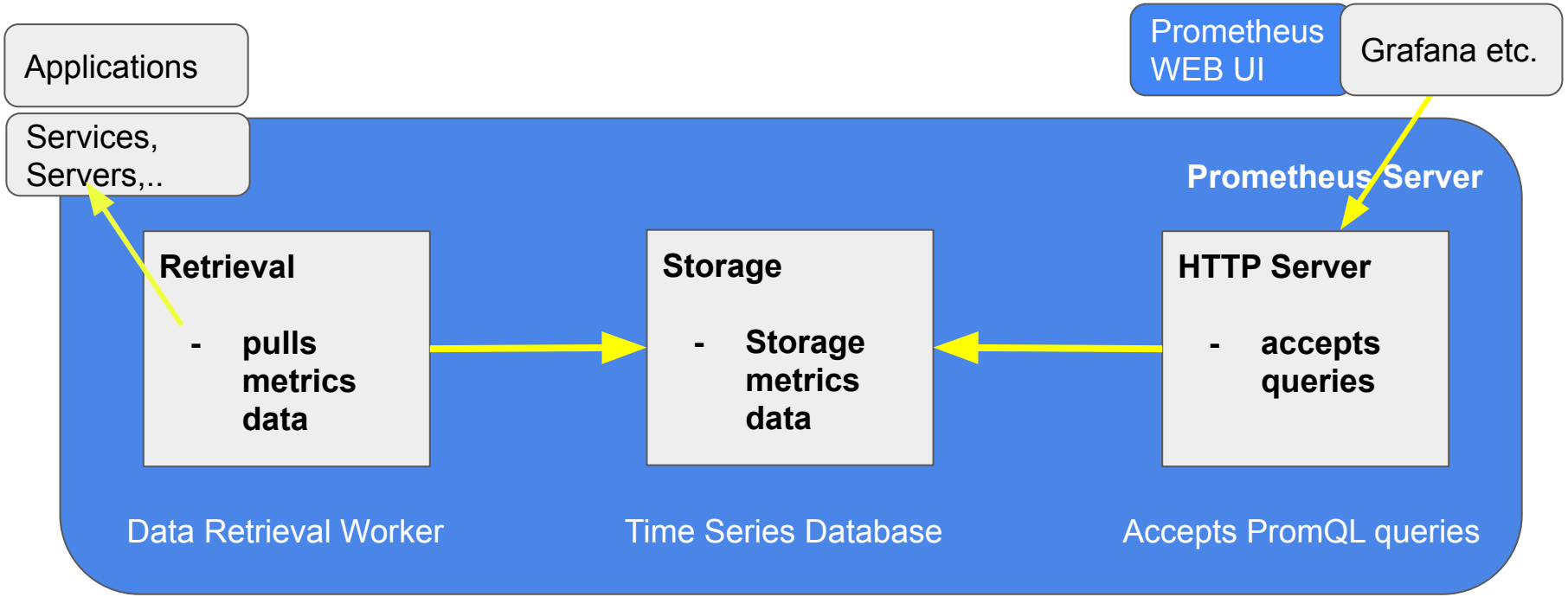
Prometheus is a high-scalable open-source monitoring framework. It provides out-of-the-box monitoring capabilities for the Kubernetes **container orchestration platform**. Also, In the observability space, it is gaining huge popularity as it helps with metrics and alerts.

NAYA College

Prometheus architecture



Main Component: Prometheus Server



NAYA College

Prometheus key points

- **Metric Collection**: Prometheus uses the pull model to retrieve metrics over HTTP. There is an option to push metrics to Prometheus using **Pushgateway** for use cases where Prometheus cannot Scrape the metrics.
- **Metric Endpoint**: The systems that you want to monitor using Prometheus should expose the metrics on an **/metrics** endpoint. Prometheus uses this endpoint to pull the metrics in regular intervals.
- **PromQL**: Prometheus comes with PromQL, a very flexible query language that can be used to query the metrics in the Prometheus dashboard. Also, the PromQL query will be used by Prometheus UI and Grafana to visualize metrics.

Prometheus key points

- **Prometheus Exporters**: Exporters are libraries that convert existing metrics from third-party apps to Prometheus metrics format. There are many official and community Prometheus exporters. One example is, the Prometheus node exporter. It exposes all Linux system-level metrics in Prometheus format.
- **TSDB** (time-series database): Prometheus uses TSDB for storing all the data efficiently. By default, all the data gets stored locally. However, to avoid a single point of failure, there are options to integrate remote storage for Prometheus TSDB.

Prometheus helm chart Installation

https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack

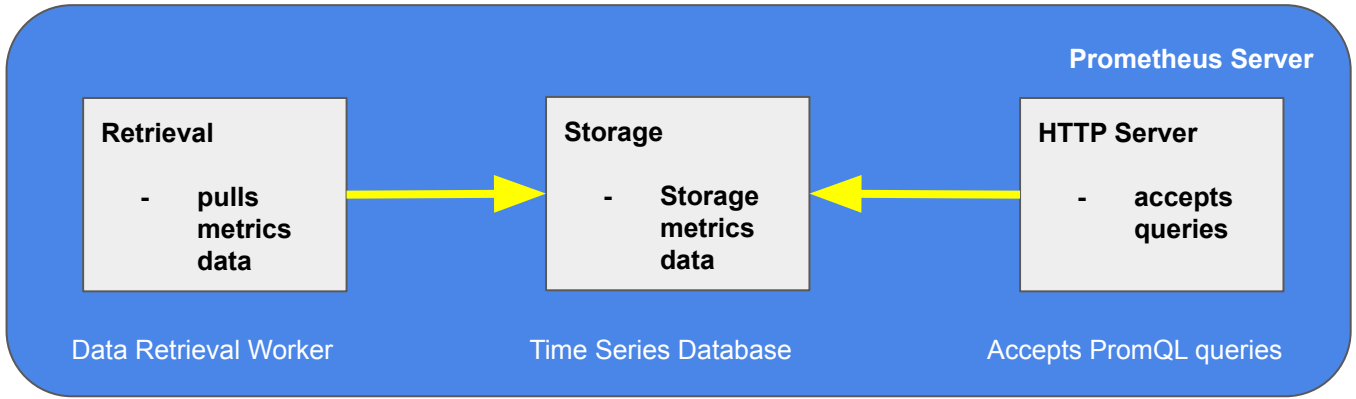
Targets and Metrics

What does Prometheus monitor?

- Linux/Windows Server
- Single Application
- Apache Server
- Service, like Database

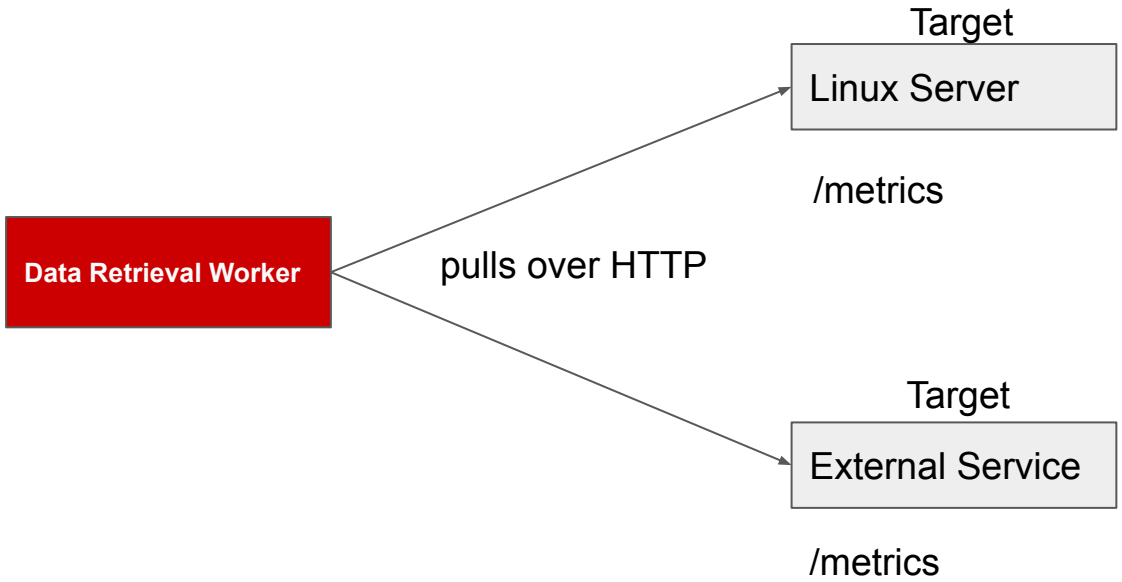
Which units are monitored of those targets?

- CPU Status
- Memory/Disk Space Usage
- Request Count
- Exceptions Count
- Request Duration



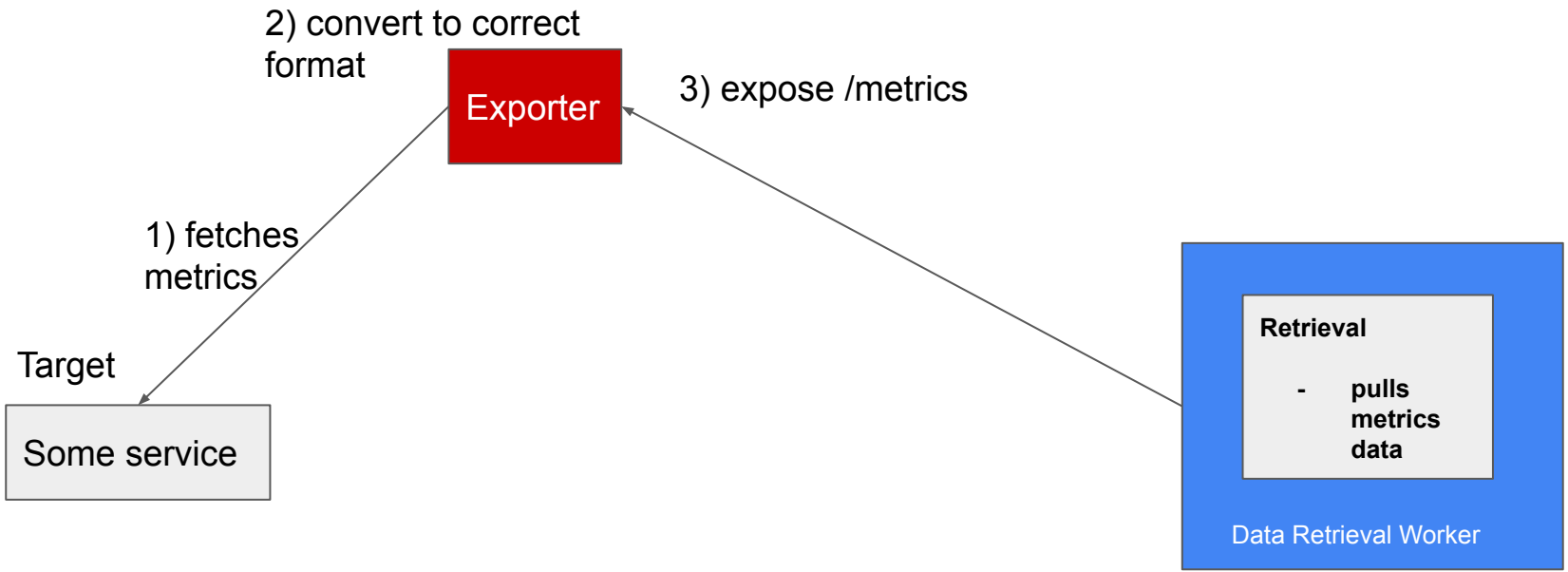
NAYA College

Collecting Metrics Data from Targets

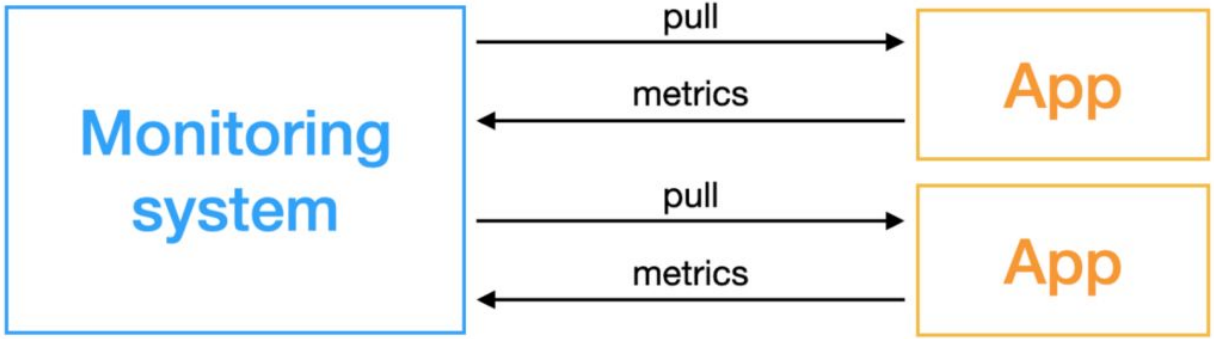


- Pulls from HTTP endpoints
- hostaddress/metrics
- must be in correct format

Target Endpoints and Exporters

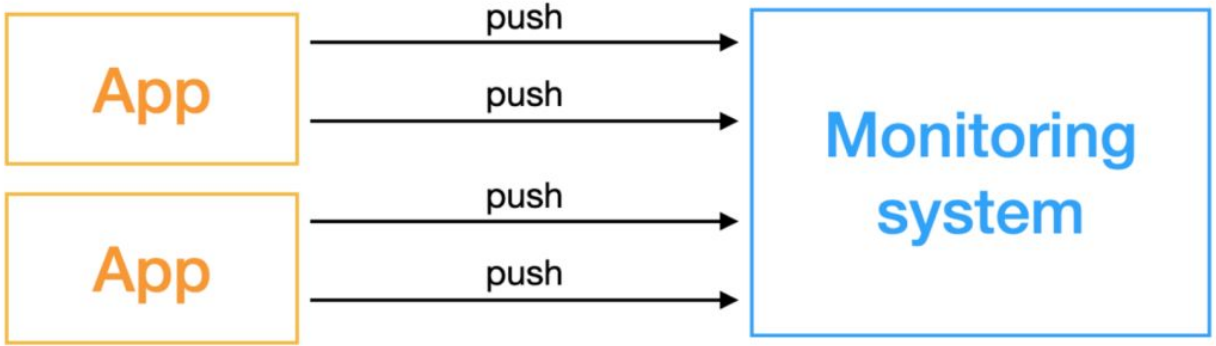


Pull-based system



- multiple Prometheus instances can pull metrics data
- better detection/insight if service is up and running

Push-based monitoring system



- high load of network traffic
- monitoring can become your bottleneck
- install additional software or tool to push metrics

EXPORTERS AND INTEGRATIONS

There are a number of libraries and servers which help in exporting existing metrics from third-party systems as Prometheus metrics. This is useful for cases where it is not feasible to instrument a given system with Prometheus metrics directly (for example, HAProxy or Linux system stats).

Third-party exporters

Some of these exporters are maintained as part of the official Prometheus GitHub organization, those are marked as *official*, others are externally contributed and maintained.

We encourage the creation of more exporters but cannot vet all of them for best practices. Commonly, those exporters are hosted outside of the Prometheus GitHub organization.

The exporter default port wiki page has become another catalog of exporters, and may include exporters not listed here due to overlapping functionality or still being in development.

The JMX exporter can export from a wide variety of JVM-based applications, for example Kafka and Cassandra.

How Does Prometheus Integrate With Your Workloads?

When using client libraries, you get a lot of default metrics from your application. For example, in Go, you get the number of bytes allocated, number of bytes used by the GC, and a lot more. See the below

```
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.13.3"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 626792
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 626792
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.442982e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 124
# HELP go_memstats_gc_cpu_fraction The fraction of this program's available CPU time used by the GC
# TYPE go_memstats_gc_cpu_fraction gauge
go_memstats_gc_cpu_fraction 0
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata.
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 2.240512e+06
```

NAYA College

Metrics

- Format: **Human-readable** text-based
- Metrics entries: TYPE and HELP attributes

```
# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.13.3"} 1
# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 626792
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 626792
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.442982e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 124
# HELP go_memstats_gc_cpu_fraction The fraction of this program's available CPU time used by the GC
# TYPE go_memstats_gc_cpu_fraction gauge
go_memstats_gc_cpu_fraction 0
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system metadata.
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 2.240512e+06
```

NAYA College

Data Model

Prometheus fundamentally stores all data as time series: streams of timestamped values belonging to the same metric and the same set of labeled dimensions.

Metric names and labels

Every time series is uniquely identified by its metric name and optional key-value pairs called labels.

The metric name specifies the general feature of a system that is measured (e.g. *http_requests_total* - the total number of HTTP requests received). It may contain ASCII letters and digits, as well as underscores and colons. It must match the regex *[a-zA-Z_:][a-zA-Z0-9_:]*.*

Data Model

Samples

Samples form the actual time series data. Each sample consists of:

- a float64 value
- a millisecond-precision timestamp

Notation

Given a metric name and a set of labels, time series are frequently identified using this notation:

<metric name>{<label name>=<label value>, ...}

For example, a time series with the metric name api_http_requests_total and the labels method="POST" and handler="/messages" could be written like this:

api_http_requests_total{method="POST", handler="/messages"}

Metric Types

The Prometheus client libraries offer four core metric types:

Gauge - Gauges are a snapshot of state, and usually when aggregating them you want to take a sum, average, minimum, or maximum.

Counter - Counters track the number or size of events, and the value your applications expose on their /metrics is the total since it started.

Summary - A summary metric will usually contain both a _sum and _count, and sometimes a time series with no suffix with a quantile label. The _sum and _count are both counters.

Histogram - Histogram metrics allow you to track the distribution of the size of events, allowing you to calculate quantiles from them.

Matchers

You almost always will want to limit by job label, and depending on what you are up to, you might want to only look at one instance or one handler, for example

node_filesystem_size_bytes{mountpoint="/etc/hosts"}

Matchers:

- = is the equality matcher; for example, job="node".
- != is the negative equality matcher; for example, job!="node".
- =~ is the regular expression matcher; for example, job=~"n.*".
- !~ is the negative regular expression matcher.

You can have multiple matchers:

node_filesystem_size_bytes{job="node",mountpoint=~"/run/.",mountpoint!~"/run/user/.*"}*

Offset

There is a modifier you can use with either type of vector selector called offset. Offset allows you to take the evaluation time for a query, and on a per-selector basis put it further back in time. For example:

process_resident_memory_bytes{job="node"} offset 1h

would get memory usage an hour before the query evaluation time. Offset is not used much in simple queries like this, as it would be easier to change the evaluation time for the whole query instead. Where this can be useful is when you only want to adjust one selector in a query expression. For example:

process_resident_memory_bytes{job="node"} - process_resident_memory_bytes{job="node"} offset 1h

would give the change in memory usage in the Node exporter over the past hour.

Aggregation

Prometheus supports the following built-in aggregation operators that can be used to aggregate the elements of a single instant vector, resulting in a new vector of fewer elements with aggregated values:

- sum (calculate sum over dimensions)
- min (select minimum over dimensions)
- max (select maximum over dimensions)
- avg (calculate the average over dimensions)
- stddev (calculate population standard deviation over dimensions)
- stdvar (calculate population standard variance over dimensions)
- count (count number of elements in the vector)
- count_values (count number of elements with the same value)
- bottomk (smallest k elements by sample value)
- topk (largest k elements by sample value)
- quantile (calculate φ-quantile ($0 \leq \varphi \leq 1$) over dimensions)

These operators can either be used to aggregate over all label dimensions or preserve distinct dimensions by including a without or by clause. These clauses may be used before or after the expression.

Grouping

Example:

If the metric http_requests_total had time series that fan out by application, instance, and group labels, we could calculate the total number of seen HTTP requests per application and group over all instances via:

sum without (instance) (http_requests_total)

Which is equivalent to:

sum by (application, group) (http_requests_total)

To get the 5 largest HTTP requests counts across all instances we could write:

topk(5, http_requests_total)

Binary operators

Prometheus query language supports basic logical and arithmetic operators. For operations between two instant vectors, the matching behavior can be modified.

Arithmetic binary operators

The following binary arithmetic operators exist in Prometheus:

- + (addition)
- - (subtraction)
- * (multiplication)
- / (division)
- % (modulo)
- ^ (power/exponentiation)

Binary arithmetic operators are defined between scalar/scalar, vector/scalar, and vector/vector value pairs.

Demo/Labs

K8S Prometheus:

In our repository

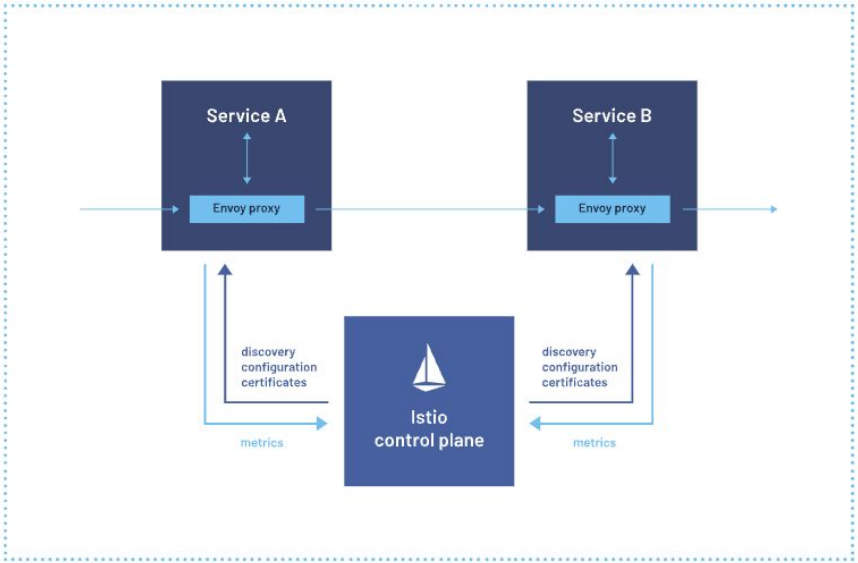
$ cd solaredge-k8s-course/k8s-3/prometheus-tutorial/

Follow README.md

Istio

The Istio service mesh

Istio addresses the challenges developers and operators face with a distributed or microservices architecture. Whether you're building from scratch or migrating existing applications to cloud native, Istio can help.



What is Istio?

Istio is an open source service mesh that layers transparently onto existing distributed applications. Istio's powerful features provide a uniform and more efficient way to secure, connect, and monitor services. Istio is the path to load balancing, service-to-service authentication, and monitoring – with few or no service code changes. Its powerful control plane brings vital features, including:

- Secure service-to-service communication in a cluster with TLS encryption, strong identity-based authentication and authorization
- Automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic
- Fine-grained control of traffic behavior with rich routing rules, retries, failovers, and fault injection
- A pluggable policy layer and configuration API supporting access controls, rate limits and quotas
- Automatic metrics, logs, and traces for all traffic within a cluster, including cluster ingress and egress

How Istio work

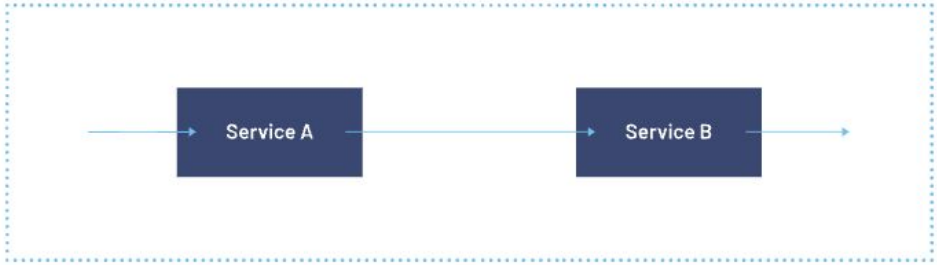
Istio has two components: the data plane and the control plane.

The data plane is the communication between services. Without a service mesh, the network doesn’t understand the traffic being sent over, and can’t make any decisions based on what type of traffic it is, or who it is from or to.

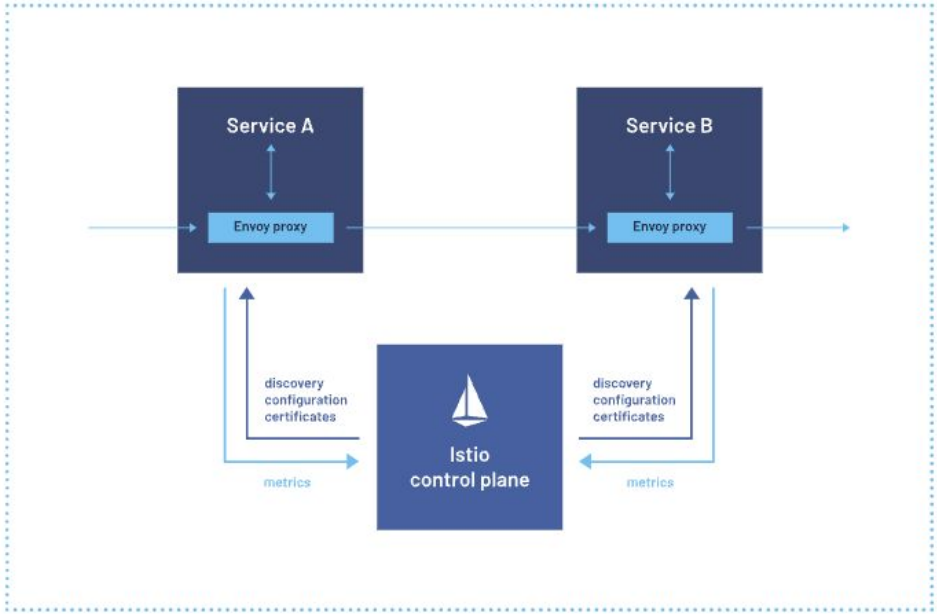
Service mesh uses a proxy to intercept all your network traffic, allowing a broad set of application-aware features based on configuration you set.

An Envoy proxy is deployed along with each service that you start in your cluster, or runs alongside services running on VMs.

The control plane takes your desired configuration, and its view of the services, and dynamically programs the proxy servers, updating them as the rules or the environment changes.

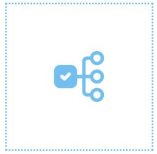


Before utilizing Istio



After utilizing Istio

Istio Concepts



- Traffic management

- Observability

- Security capabilities

NAYA College

Observability

As services grow in complexity, it becomes challenging to understand behavior and performance. Istio generates detailed telemetry for all communications within a service mesh. This telemetry provides observability of service behavior, empowering operators to troubleshoot, maintain, and optimize their applications. Even better, you get almost all of this instrumentation without requiring application changes. Through Istio, operators gain a thorough understanding of how monitored services are interacting.

Istio’s telemetry includes detailed metrics, distributed traces, and full access logs. With Istio, you get thorough and comprehensive service mesh observability.

Security capabilities

Microservices have particular security needs, including protection against man-in-the-middle attacks, flexible access controls, auditing tools, and mutual TLS. Istio includes a comprehensive security solution to give operators the ability to address all of these issues. It provides strong identity, powerful policy, transparent TLS encryption, and authentication, authorization and audit (AAA) tools to protect your services and data.

Istio’s security model is based on security-by-default, aiming to provide in-depth defense to allow you to deploy security-minded applications even across distrusted networks.

Traffic management

Routing traffic, both within a single cluster and across clusters, affects performance and enables better deployment strategy. Istio's traffic routing rules let you easily control the flow of traffic and API calls between services. Istio simplifies configuration of service-level properties like circuit breakers, timeouts, and retries, and makes it easy to set up important tasks like A/B testing, canary deployments, and staged rollouts with percentage-based traffic splits.

https://istio.io/latest/docs/concepts/traffic-management/

Demo/Labs

K8S Istio:

In our repository

$ cd solaredge-k8s-course/k8s-3/istio-tut/

Follow README.md