kubernetes

Lev Epshtein
lev@opsguru.io

NAYA College

Kubernetes

Basic part II

- K8S operator
- Custom Resource Definition (CRD)
- Helm
- Role-based Access Control
- Istio

NAYA College

K8S Operator pattern

K8S operator

Operators are software extensions to Kubernetes that make use of **custom resources** to **manage applications and their components**. Operators follow Kubernetes principles, notably the **control loop**.

K8S Operator

The operator pattern aims to capture the key aim of a human operator who is managing a service or set of services. Human operators who look after specific applications and services have deep knowledge of how the **system ought to behave, how to deploy it, and how to react if there are problems.**

People who run workloads on Kubernetes often like to use automation to take care of repeatable tasks. The operator pattern captures how you can write code to automate a task beyond what Kubernetes itself provides.

An example operator

Some of the things that you can use an operator to automate include:

- deploying an application on demand
- taking and restoring backups of that application's state
- handling upgrades of the application code alongside related changes such as database schemas or extra configuration settings
- publishing a Service to applications that don't support Kubernetes APIs to discover them
- simulating failure in all or part of your cluster to test its resilience
- choosing a leader for a distributed application without an internal member election process

What might an operator look like in more detail? Here's an example:

- A custom resource named SampleDB, that you can configure into the cluster.
- A Deployment that makes sure a Pod is running that contains the controller part of the operator.
- A container image of the operator code.
- Controller code that queries the control plane to find out what SampleDB resources are configured.
- The core of the operator is code to tell the API server how to make reality match the configured resources.
    - If you add a new SampleDB, the operator sets up PersistentVolumeClaims to provide durable database storage, a StatefulSet to run SampleDB and a Job to handle initial configuration.
    - If you delete it, the operator takes a snapshot, then makes sure that the StatefulSet and Volumes are also removed

What might an operator look like in more detail? Here's an example:

- The operator also manages regular database backups. For each SampleDB resource, the operator determines when to create a Pod that can connect to the database and take backups. These Pods would rely on a ConfigMap and / or a Secret that has database connection details and credentials.
- Because the operator aims to provide robust automation for the resource it manages, there would be additional supporting code. For this example, code checks to see if the database is running an old version and, if so, creates Job objects that upgrade it for you.

https://operatorhub.io/

Custom Resource Definition (CRD)

What is a Kubernetes Resource

A Kubernetes resource is an endpoint in the Kubernetes API that stores a collection of API objects of a certain kind. For example, the built-in Pods resource contains a collection of Pod objects.

In Kubernetes, there are many built-in resources available. Some examples of Kubernetes resources include:

- Deployments
- Pod
- Services
- ConfigMap

What is a Custom Resource(CR)

A Custom Resource is an object that extends the Kubernetes API or allows you to introduce your own API into a cluster.

Custom resources can appear and disappear in a running cluster through dynamic registration, and cluster admins can update custom resources independently of the cluster itself.

Once a custom resource is installed, users can create and access its objects using kubectl, just as they do for built-in resources like Pods.

What is a CustomResourceDefinition(CRD)

A CustomResourceDefinition (CRD) defines the name and structure of your custom resource that you want to add to the cluster.

CRDs allow users to create new types of custom resources without adding another API server. Defining a CRD object creates a new custom resource with a name and schema that you specify.

When you create a new CRD, the Kubernetes API Server creates a new RESTful resource path to serve and handle the storage of your custom resource. This frees you from writing your own API server to handle the custom resource.

Custom controllers

On their own, **custom resources let you store and retrieve structured data**. When you combine a c**ustom resource** with a **custom controller**, custom resources provide a **true declarative API.**

The Kubernetes declarative API enforces a separation of responsibilities. **You declare the desired state of your resource.** The Kubernetes controller keeps the current state of Kubernetes objects in sync with your declared desired state. This is in contrast to an imperative API, where you instruct a server what to do.

Custom controllers

You can deploy and update a custom controller on a running cluster, independently of the cluster's lifecycle. Custom controllers can work with any kind of resource, but th**ey are especially effective when combined with custom resources.** The **Operator pattern** combines **custom resources and custom controllers.** You can use custom controllers to encode domain knowledge for specific applications into an extension of the Kubernetes API.

Create a Custom Resource Definition

In repository:

$ cd cognyte-devops-32/k8s-3/CRD

README.md

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

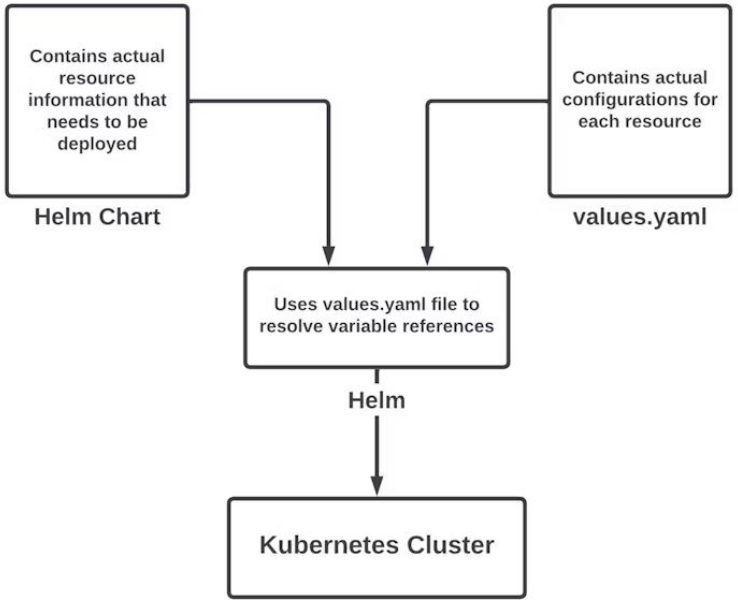
Consider a practical scenario where Helm is helpful. Suppose you want to deploy your application in a production environment with ten replicas. You specify this in the deployment YAML file for the application and run the deployment using the kubectl command.

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.

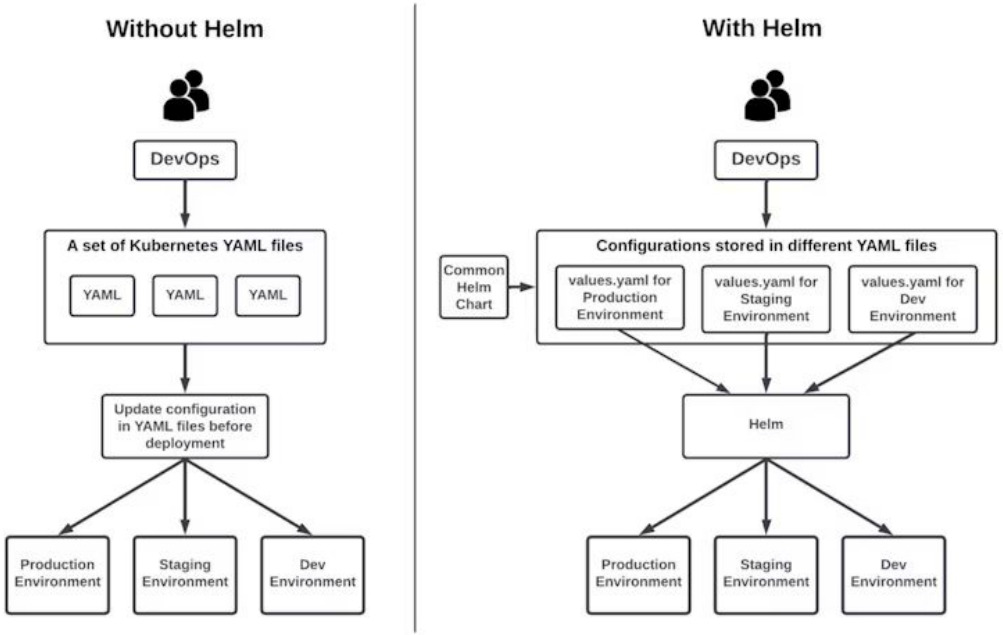
How Helm works

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:

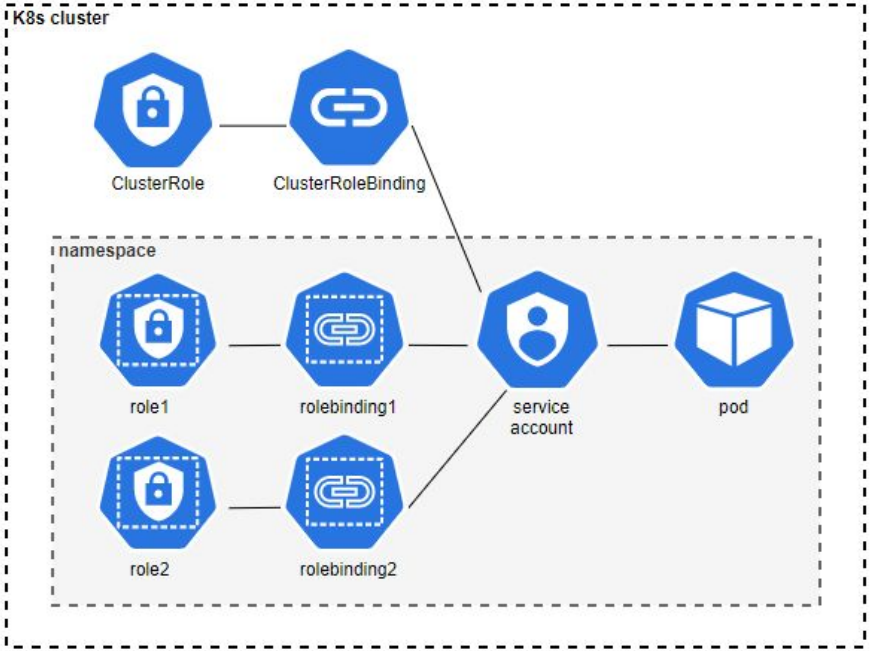
In our repository

$ cd cognyte-devops-32/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.



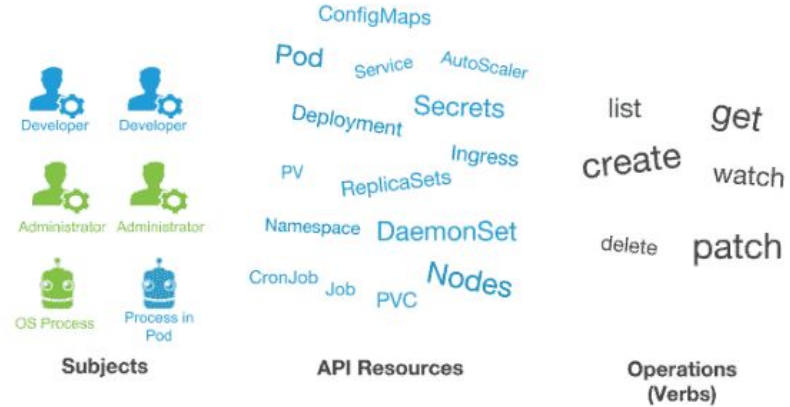
The key to understanding RBAC in Kubernetes

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

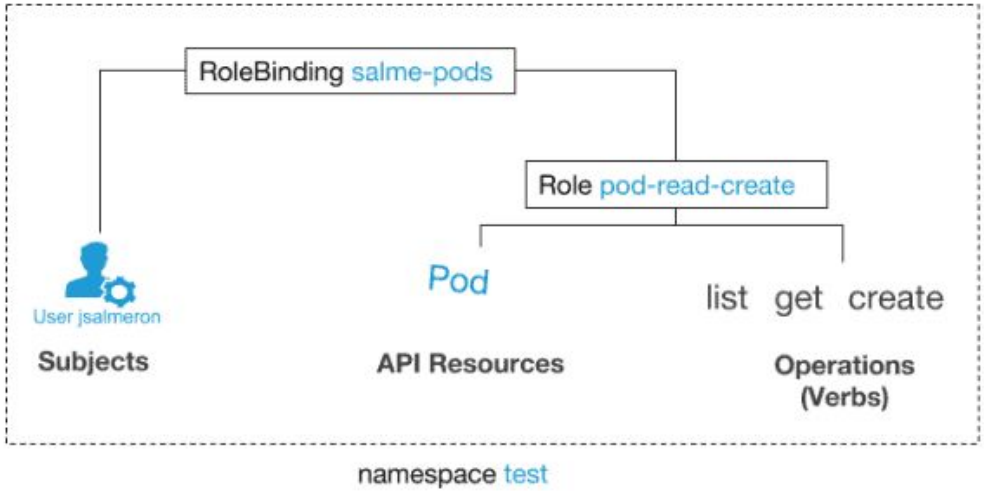
Understanding RBAC API objects

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?

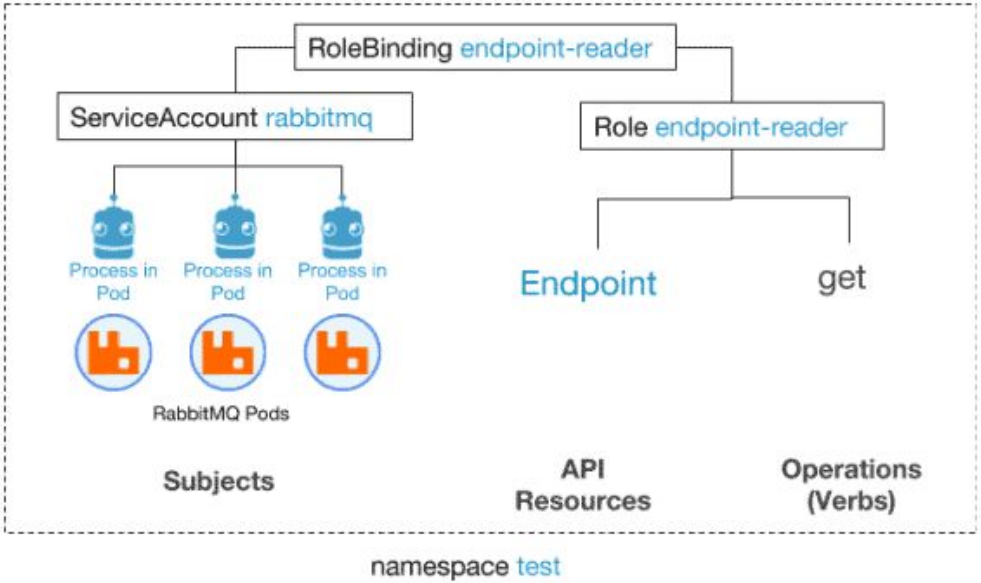
Users: These are global, and meant for humans or processes living outside the cluster.

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 }}
```

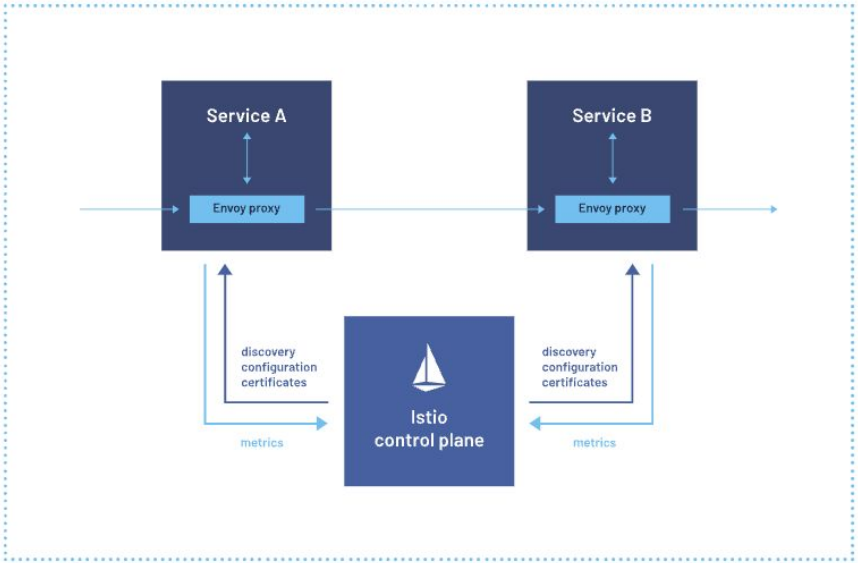
```
{{- if .Values.rbacEnabled }}
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: {{ template "rabbitmq.fullname" . }}-endpoint-reader
  labels:
    app: {{ template "rabbitmq.name" . }}
    chart: {{ template "rabbitmq.chart" .  }}
    release: "{{ .Release.Name }}"
   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

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.

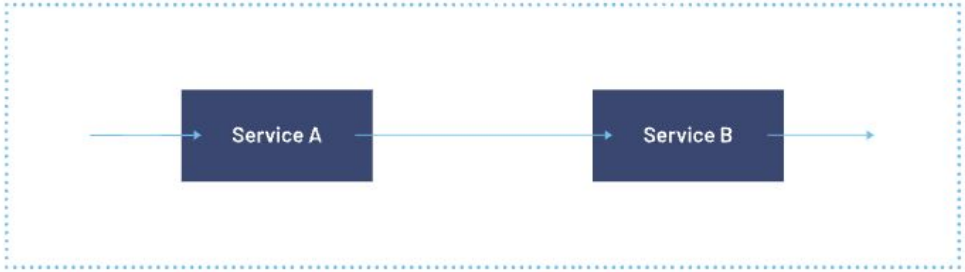


What is Istio?

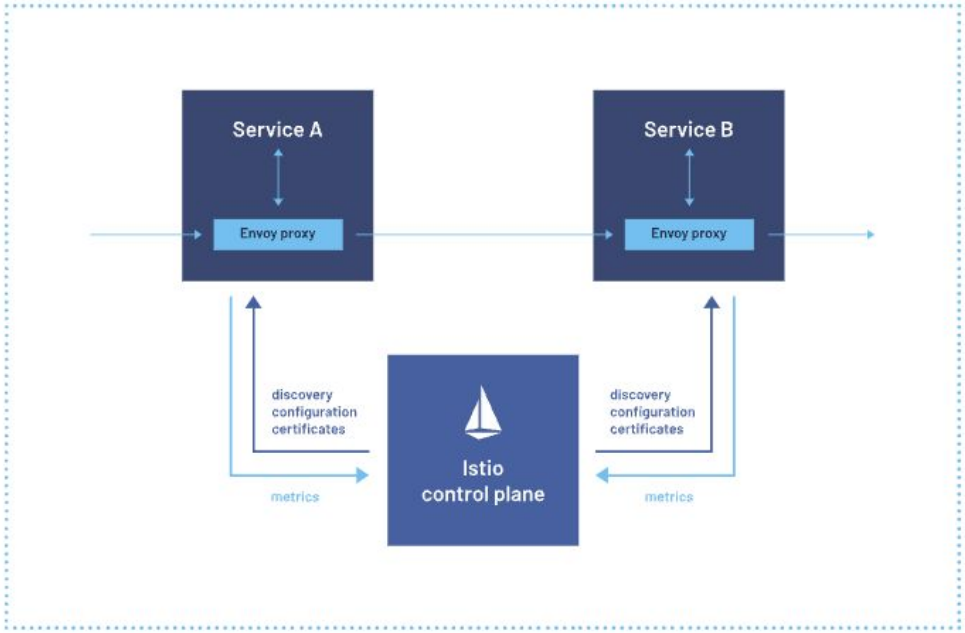
Istio is an open source service mesh that layers transparently onto existing distributed applications. Istio's powerful features provide a uniform and more efficient way to secure, connect, and monitor services. Istio is the path to load balancing, service-to-service authentication, and monitoring – 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



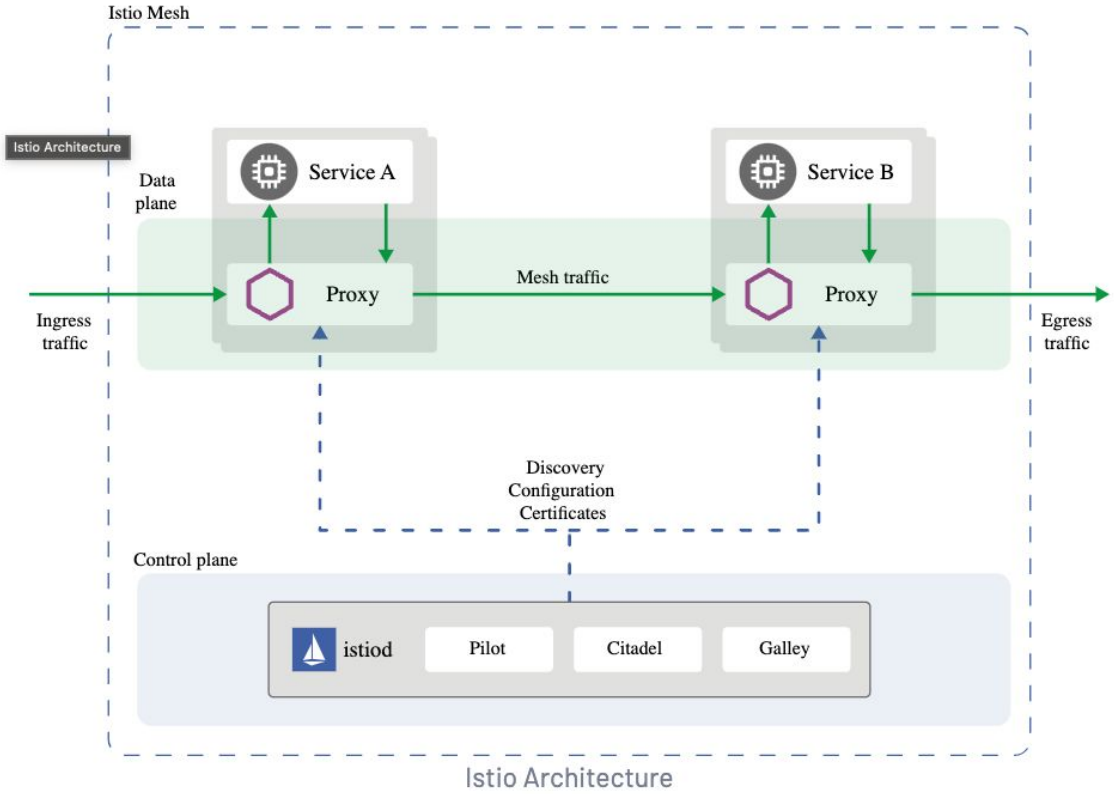
Before utilizing Istio



After utilizing Istio

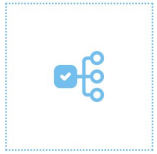
NAYA College

How Istio work



NAYA College

Istio Concepts



- Traffic management

- Observability

- Security capabilities

NAYA College

Observability

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 cognyte-devops-32/k8s-3/istio-tut/

Follow README.md