Kubernetes Packages: Multi-environment package managment with Helm vs Kustomize comparison, Helmfile SOPS integration

Check GitHub for helpful DevOps tools:

Michael Robotics
Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics

Ask Personal AI Document assistant to learn interactively (FASTER)!

1 Download PDF

2 Go to website

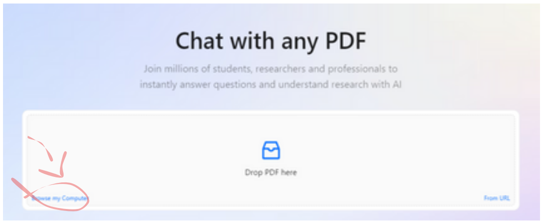
3 Browse file

4 Chat with Document

1 https://github.com/MichaelRobotics/DevOpsTools/blob/main/KubernetesPackages.pdf

Click there to go to ChatPdf website 2

3



Chat with any PDF

Join millions of students, researchers and professionals to instantly answer questions and understand research with AI

Ask questions about document! 4

Complety new to Linux and Networking?

Essential for this PDF is a thorough knowledge of networking. I highly recommend the HTB platform's networking module, which offers extensive information to help build a comprehensive understanding.

HTB - Your Cyber Performance Center

We provide a human-first platform creating and maintaining high performing cybersecurity individuals and organizations.

https://www.hackthebox.com/

What is Kubernetes?

Kubernetes is an open-source platform that automates the deployment, scaling, and management of containerized applications. It helps manage clusters of nodes running containers, ensuring efficient and reliable operation.

How Kubernetes clusters are made?

Kubernetes clusters consist of a control plane and multiple worker nodes. The control plane manages cluster operations, while worker nodes run the actual container workloads.

Why and When use Kubernetes

Kubernetes is ideal for deploying scalable, resilient, and automated containerized applications. It is used when managing multiple containers across different environments is necessary.

Example: Running a microservices-based e-commerce platform that scales up during peak hours.

System Requirements

- RAM: 2 GB per node (1 GB can work for testing but may lead to limited performance)
- 10 GB free storage
- Ubuntu

Kubernetes: Main components & packages

- **kube-apiserver:** Central management component that exposes the Kubernetes API; acts as the front-end for the cluster.
- **etcd:** Distributed key-value store for storing all cluster data, ensuring data consistency across nodes.
- **kube-scheduler:** Assigns pods to available nodes based on resource requirements and policies.
- **kube-controller-manager:** Manages core controllers that handle various functions like node status, replication, and endpoints.
- **kubelet:** Agent that runs on each node, responsible for managing pods and their containers.
- **kube-proxy:** Manages networking on each node, ensuring communication between pods and services within the cluster.

Kubernetes Packages: Helm charts

1) Helm intro

What is Helm

Helm is a package manager for Kubernetes that simplifies the deployment and management of applications in Kubernetes clusters. It allows you to organize all Kubernetes YAML configuration files into reusable and configurable packages called charts.

What is Helm chart

Helm charts group all YAML files in a templates folder and include metadata like name, description, and version. They can be reused for different apps by injecting custom parameters via the CLI for quick overrides or a values.yaml file for larger configurations.

2) Environment configuration

Create kind cluster or use Killerkoda:

```
kind create cluster --name helm --image kindest/node:latest
```

run alpine linux container for isolated testing environemt and install dependencies:

```
docker run -it --rm -v ${HOME}:/root/ -v ${PWD}:/work -w /work --net host alpine sh
apk add --no-cache curl nano git tree
curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s
https://storage.googleapis.com/kubernetes-
release/release/stable.txt`/bin/linux/amd64/kubectl
chmod +x ./kubectl
mv ./kubectl /usr/local/bin/kubectl
export KUBE_EDITOR="nano"
```

Install Helm CLI

```
curl -LO https://get.helm.sh/helm-v3.4.0-linux-amd64.tar.gz
tar -C /tmp/ -zxvf helm-v3.4.0-linux-amd64.tar.gz
rm helm-v3.4.0-linux-amd64.tar.gz
mv /tmp/linux-amd64/helm /usr/local/bin/helm
chmod +x /usr/local/bin/helm
```

Setup directories

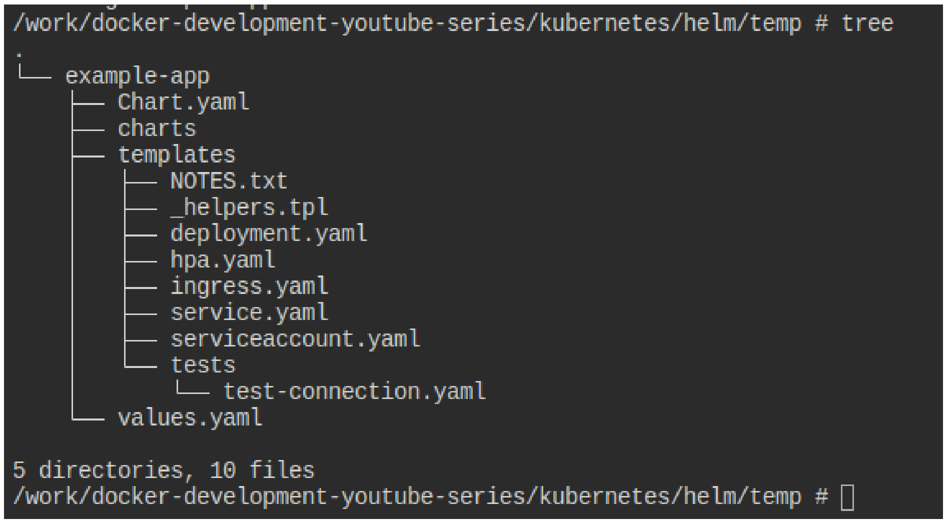
download repo with k8s yamls used in this helm introduction

```
git clone https://github.com/marcel-dempers/docker-development-youtube-series.git
cd docker-development-youtube-series/kubernetes/helm
```

create example helmchart template named example-app

```
mkdir temp && cd temp
helm create example-app
```

tree structure of helmchart template should look like this:



```
/work/docker-development-youtube-series/kubernetes/helm/temp # tree
.
└── example-app
    ├── Chart.yaml
    ├── charts
    ├── templates
    │   ├── NOTES.txt
    │   ├── _helpers.tpl
    │   ├── deployment.yaml
    │   ├── hpa.yaml
    │   ├── ingress.yaml
    │   ├── service.yaml
    │   ├── serviceaccount.yaml
    │   └── tests
    │       └── test-connection.yaml
    └── values.yaml

5 directories, 10 files
/work/docker-development-youtube-series/kubernetes/helm/temp #
```

delete files and folders in /templates, keeping only _helpers.tpl

mv all files from except _helpers.tpl _/kubernetes/helm/example-app/templates to /kubernetes/helm/temp/example-app/templates

Now lets understand Helmchart structure:

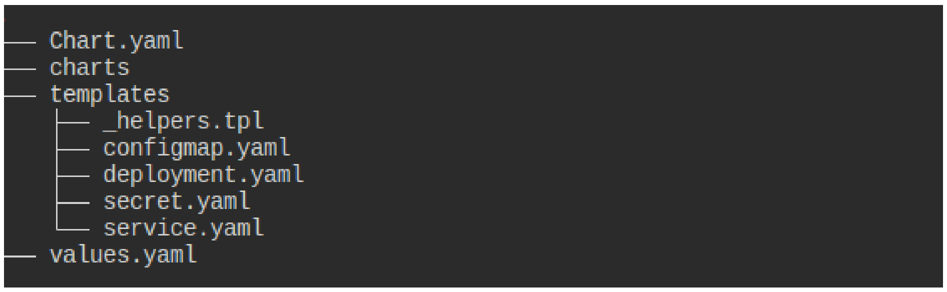

```
├── Chart.yaml
├── charts
├── templates
│   ├── _helpers.tpl
│   ├── configmap.yaml
│   ├── deployment.yaml
│   ├── secret.yaml
│   └── service.yaml
└── values.yaml
```

Chart.yaml: Contains metadata about the Helm chart, such as its name, version, and description.

values.yaml: Defines default configuration values for the chart that can be overridden by the user.

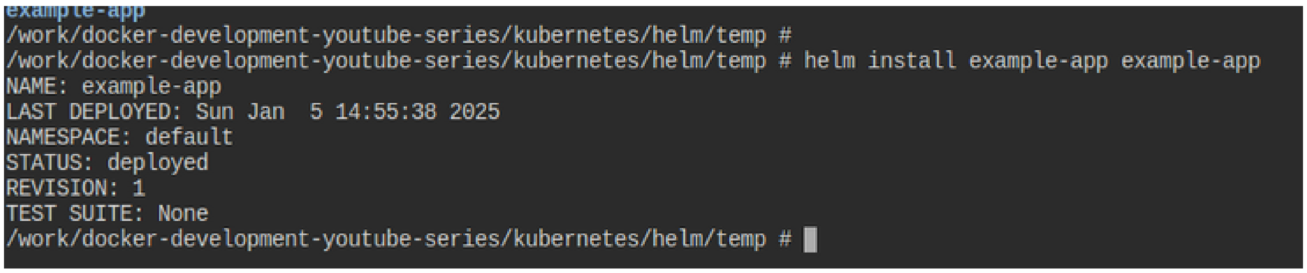
charts/: Contains any dependent charts that the main chart requires, often as packaged .tgz files.

templates/: Holds Kubernetes YAML templates that are rendered into manifests during chart installation.

Delete values.yaml and create it again. By default it goes with some predefined values and we want a blank file.

Now install app from helm chart.

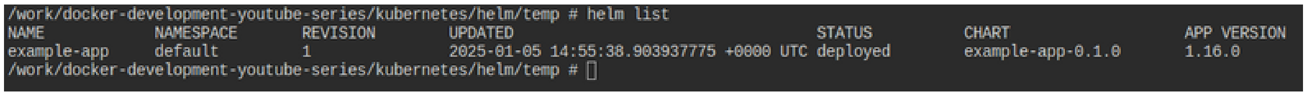
```
helm install example-app example-app
```



```
example-app
/work/docker-development-youtube-series/kubernetes/helm/temp #
/work/docker-development-youtube-series/kubernetes/helm/temp # helm install example-app example-app
NAME: example-app
LAST DEPLOYED: Sun Jan  5 14:55:38 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
/work/docker-development-youtube-series/kubernetes/helm/temp #
```

check deployments created by helm:

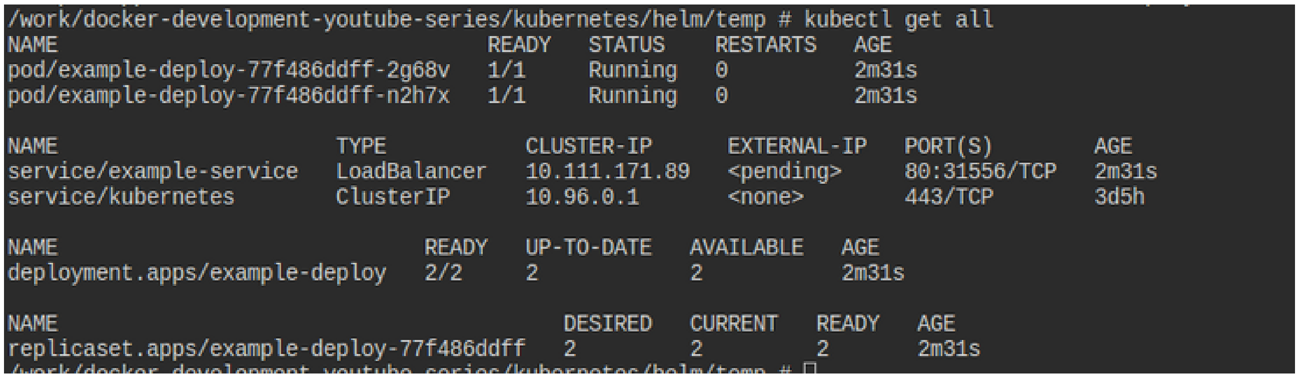
```
helm list
```



```
/work/docker-development-youtube-series/kubernetes/helm/temp # helm list
NAME            NAMESPACE       REVISION        UPDATED                                 STATUS          CHART                   APP VERSION
example-app     default         1               2025-01-05 14:55:38.903937775 +0000 UTC deployed        example-app-0.1.0       1.16.0
/work/docker-development-youtube-series/kubernetes/helm/temp #
```

No check deployment:

```
kubectl get all
```



```
/work/docker-development-youtube-series/kubernetes/helm/temp # kubectl get all
NAME                                  READY   STATUS    RESTARTS   AGE
pod/example-deploy-77f486ddff-2g68v   1/1     Running   0          2m31s
pod/example-deploy-77f486ddff-n2h7x   1/1     Running   0          2m31s

NAME                      TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
service/example-service   LoadBalancer   10.111.171.89   <pending>     80:31556/TCP   2m31s
service/kubernetes        ClusterIP      10.96.0.1       <none>        443/TCP        3d5h

NAME                             READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/example-deploy   2/2     2            2           2m31s

NAME                                        DESIRED   CURRENT   READY   AGE
replicaset.apps/example-deploy-77f486ddff   2         2         2       2m31s
/work/docker-development-youtube-series/kubernetes/helm/temp #
```

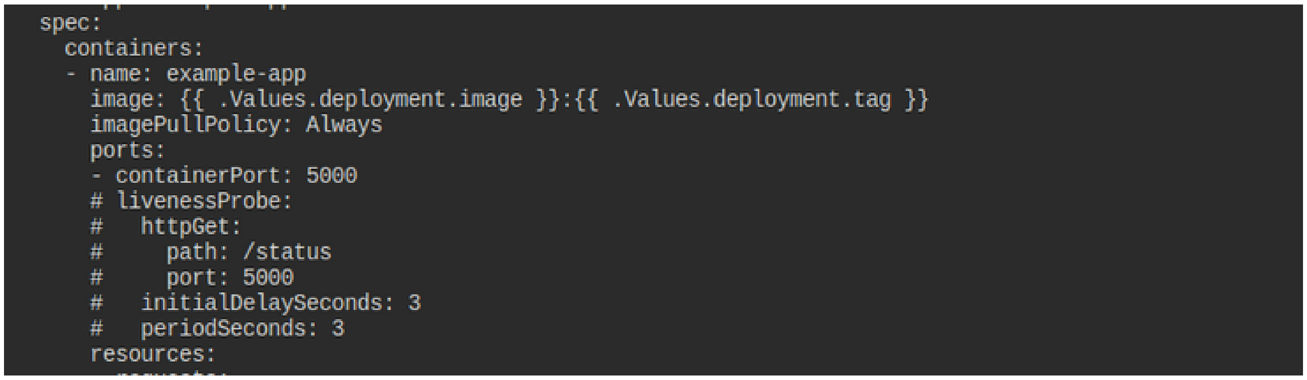
3) Inject values for simple upgrade

Helmchart we created is not reusable. It just deploys all yaml files we specified. Now its time to make it configurable, so it will create deployment with some customized values injected from values.yaml. We will start with parametrizing deployment image name.

Modify values.yaml and add:

```
deployment:
  image: "aimvector/python"
  tag: "1.0.4"
name: example-app
```

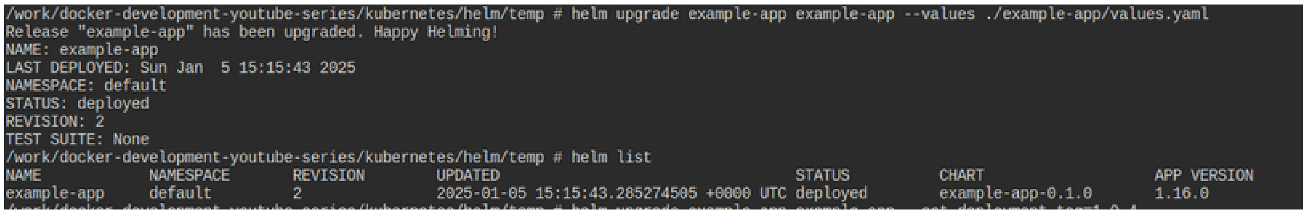
In deployment, parametrize Image fileds, to let values.yaml know, when to inject those:



```
  spec:
    containers:
    - name: example-app
      image: {{ .Values.deployment.image }}:{{ .Values.deployment.tag }}
      imagePullPolicy: Always
      ports:
      - containerPort: 5000
      # livenessProbe:
      #   httpGet:
      #     path: /status
      #     port: 5000
      #   initialDelaySeconds: 3
      #   periodSeconds: 3
      resources:
```

now upgrade helmchart from values.yaml:

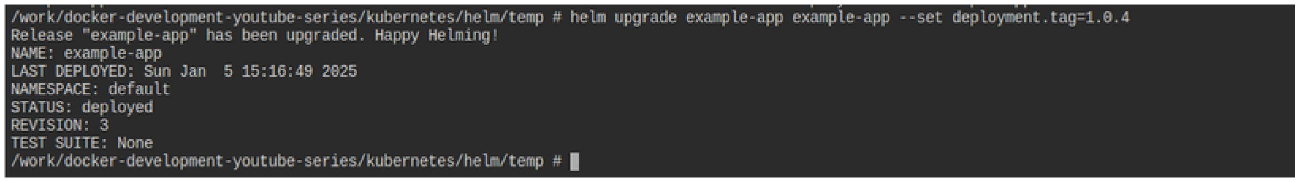
```
helm upgrade example-app example-app --values ./example-app/values.yaml
```



```
/work/docker-development-youtube-series/kubernetes/helm/temp # helm upgrade example-app example-app --values ./example-app/values.yaml
Release "example-app" has been upgraded. Happy Helming!
NAME: example-app
LAST DEPLOYED: Sun Jan  5 15:15:43 2025
NAMESPACE: default
STATUS: deployed
REVISION: 2
TEST SUITE: None
/work/docker-development-youtube-series/kubernetes/helm/temp # helm list
NAME            NAMESPACE       REVISION        UPDATED                                 STATUS          CHART                   APP VERSION
example-app     default         2               2025-01-05 15:15:43.285274505 +0000 UTC deployed        example-app-0.1.0       1.16.0
```

You can update chart from CLI too:

```
helm upgrade example-app example-app --set deployment.tag=1.0.4
```



```
/work/docker-development-youtube-series/kubernetes/helm/temp # helm upgrade example-app example-app --set deployment.tag=1.0.4
Release "example-app" has been upgraded. Happy Helming!
NAME: example-app
LAST DEPLOYED: Sun Jan  5 15:16:49 2025
NAMESPACE: default
STATUS: deployed
REVISION: 3
TEST SUITE: None
/work/docker-development-youtube-series/kubernetes/helm/temp #
```

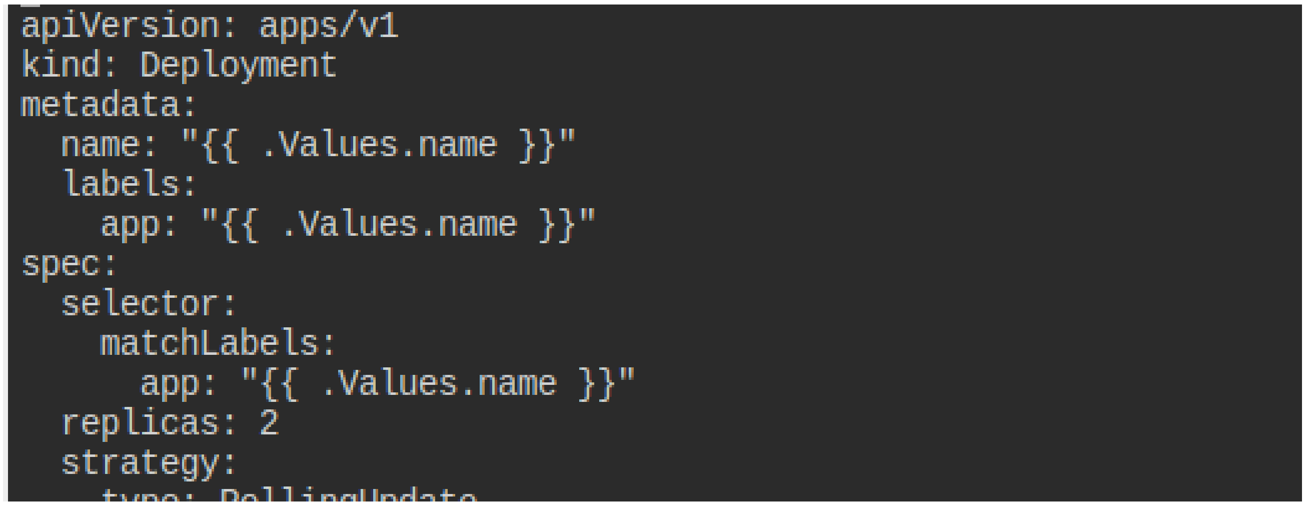
4) Make chart generic

Our values file let us only upgrade existing deployment image name. What if we want to create separate deployments? Lets copy values.yaml and name it values-app-02.yaml

Edit values-app-02.yaml

```
deployment:
  image: "aimvector/python"
  tag: "1.0.4"
name: example-app
```

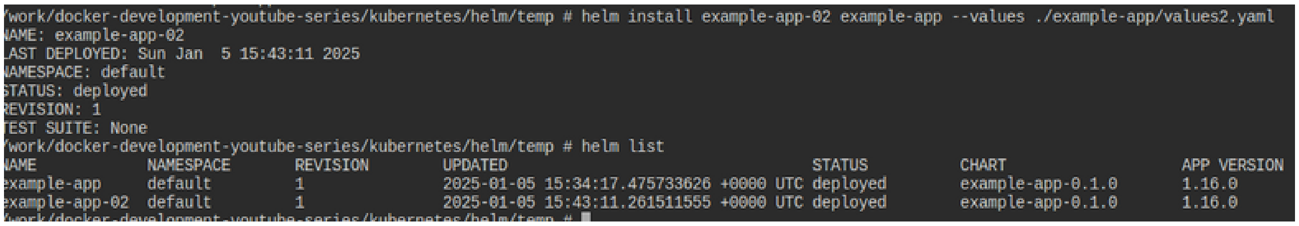
Edit all yaml “name” and “app” keys to point at Values.name parameter stored in values-app02.yaml



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: "{{ .Values.name }}"
  labels:
    app: "{{ .Values.name }}"
spec:
  selector:
    matchLabels:
      app: "{{ .Values.name }}"
  replicas: 2
  strategy:
```

Install new deployment with helm using values from values-app-02.yaml. Lets name it example-app-02

```
helm install example-app-02 example-app --values ./example-app/values.yaml
```



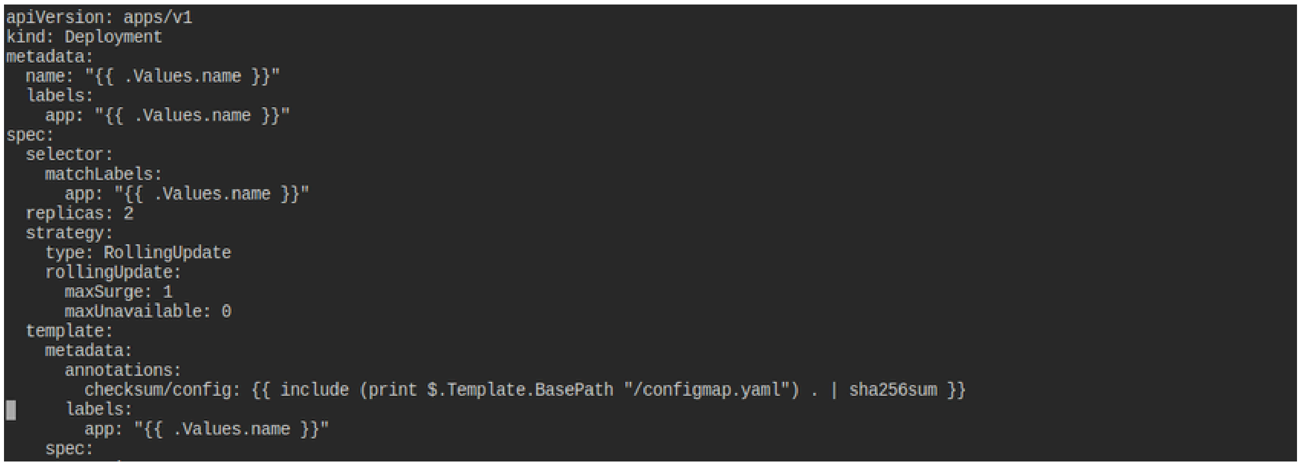
```
/work/docker-development-youtube-series/kubernetes/helm/temp # helm install example-app-02 example-app --values ./example-app/values2.yaml
NAME: example-app-02
LAST DEPLOYED: Sun Jan  5 15:43:11 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
/work/docker-development-youtube-series/kubernetes/helm/temp # helm list
NAME            NAMESPACE       REVISION        UPDATED                                 STATUS          CHART                   APP VERSION
example-app     default         1               2025-01-05 15:34:17.475733626 +0000 UTC deployed        example-app-0.1.0       1.16.0
example-app-02  default         1               2025-01-05 15:43:11.261511555 +0000 UTC deployed        example-app-0.1.0       1.16.0
/work/docker-development-youtube-series/kubernetes/helm/temp #
```

As you can see, helm created new application.

By default, deployments don't roll out new pods when a configmap changes. To trigger a rollout, we'll configure it to update pods when the configmap changes.

add annotation to deployment.yaml

checksum/config: {{ include (print $.Template.BasePath "/configmap.yaml") . | sha256sum }}



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: "{{ .Values.name }}"
  labels:
    app: "{{ .Values.name }}"
spec:
  selector:
    matchLabels:
      app: "{{ .Values.name }}"
  replicas: 2
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    metadata:
      annotations:
        checksum/config: {{ include (print $.Template.BasePath "/configmap.yaml") . | sha256sum }}
      labels:
        app: "{{ .Values.name }}"
    spec:
```

5) Controll flows

You can also set default values in case they are not supplied by the values.yaml file.

This may help you keep the values.yaml file small. Add those to your deployment.yaml

```
{{- if .Values.deployment.resources }}
  resources:
    {{- if .Values.deployment.resources.requests }}
    requests:
      memory: {{ .Values.deployment.resources.requests.memory | default "50Mi" | quote }}
      cpu: {{ .Values.deployment.resources.requests.cpu | default "10m" | quote }}
    {{- else}}
    requests:
      memory: "50Mi"
      cpu: "10m"
    {{- end}}
    {{- if .Values.deployment.resources.limits }}
    limits:
      memory: {{ .Values.deployment.resources.limits.memory | default "1024Mi" | quote }}
      cpu: {{ .Values.deployment.resources.limits.cpu | default "1" | quote }}
    {{- else}}
    limits:
      memory: "1024Mi"
      cpu: "1"
    {{- end }}
  {{- else }}
  resources:
    requests:
      memory: "50Mi"
      cpu: "10m"
    limits:
      memory: "1024Mi"
      cpu: "1"
  {{- end}}
```

Kubernetes Packages: Helmfiles

1) Intro & Setup

Helmfile is a declarative tool for managing Kubernetes Helm charts. It simplifies deploying and managing multiple Helm releases by organizing configurations in a single YAML file, ensuring consistent and repeatable deployments across environments. It automatically adds helm repositories and installs them.

Download my helmfile examples repo:

```
git clone https://github.com/MichaelRobotics/Kubernetes.git
cd Kubernetes/Helm/Helmfile
```

Install helmfile with script. When prompted, choose “yes”:

```
./helmfile.sh
```

2) Basics

Lets have a look at example helmfile:

```
cat helmfile-prometheus.yaml
```

```
repositories:
 - name: prometheus-community
   url: https://prometheus-community.github.io/helm-charts

releases:
- name: prom-norbac-ubuntu
  namespace: prometheus
  chart: prometheus-community/prometheus
```

1. Repositories Section

The repositories section lists Helm chart repositories to fetch charts from. Here, prometheus-community is the alias for the repository at https://prometheus-community.github.io/helm-charts.

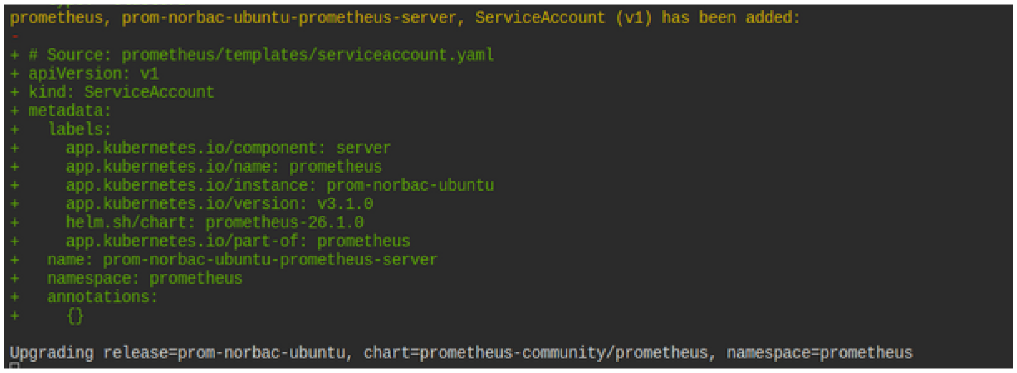
2. Releases Section

The releases section defines Helm chart deployments. In this case, the release prom-norbac-ubuntu installs the prometheus chart from the prometheus-community repository in the prometheus namespace.

Apply helmfile:

```
helmfile \
--file helmfile-prometheus.yaml \
apply \
--wait
```

The command combines helmfile diff and helmfile sync to ensure your Helm releases align with the desired state in helmfile-prometheus.yaml. It first shows the differences (diff) between the current state and the desired configuration, then applies (sync) those changes while waiting (--wait) for resources to stabilize.



```
prometheus, prom-norbac-ubuntu-prometheus-server, ServiceAccount (v1) has been added:
-
+ # Source: prometheus/templates/serviceaccount.yaml
+ apiVersion: v1
+ kind: ServiceAccount
+ metadata:
+   labels:
+     app.kubernetes.io/component: server
+     app.kubernetes.io/name: prometheus
+     app.kubernetes.io/instance: prom-norbac-ubuntu
+     app.kubernetes.io/version: v3.1.0
+     helm.sh/chart: prometheus-26.1.0
+     app.kubernetes.io/part-of: prometheus
+   name: prom-norbac-ubuntu-prometheus-server
+   namespace: prometheus
+   annotations:
+     {}

Upgrading release=prom-norbac-ubuntu, chart=prometheus-community/prometheus, namespace=prometheus
```

Similar to Git, where git diff shows changes and git merge applies them, this command highlights Kubernetes configuration differences and ensures the cluster matches the Helmfile.

3) Updating releases

Lets check what will happen if we modfiy existing chart. Have a look:

```
cat helmfile-prometheus.yaml
```

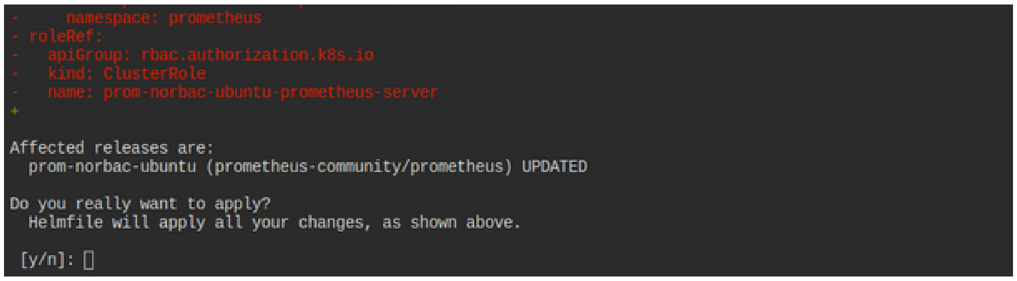
```
repositories:
 - name: prometheus-community
   url: https://prometheus-community.github.io/helm-charts
```

```
releases:
- name: prom-norbac-ubuntu
  namespace: prometheus
  chart: prometheus-community/prometheus
  set:
  - name: rbac.create
    value: false
```

Modifying the chart to disable RBAC (rbac.create=false) and running helmfile apply updates the release to match the new configuration. Similar to Git, it ensures changes are tracked and applied to the cluster.

apply

```
helmfile \
    --interactive \
    --file helmfile-prometheus-rbac.yaml \
    apply \
    --wait
```



```
-      namespace: prometheus
- roleRef:
-   apiGroup: rbac.authorization.k8s.io
-   kind: ClusterRole
-   name: prom-norbac-ubuntu-prometheus-server
+

Affected releases are:
  prom-norbac-ubuntu (prometheus-community/prometheus) UPDATED

Do you really want to apply?
  Helmfile will apply all your changes, as shown above.

 [y/n]: 
```

As you can see, helmfile detected changes and asks you if you want to apply update.

4) Hooks

We can add scripts to execute at different stages of kubernetes resources deployment.

```
cat helmfile-hooks.yaml
```

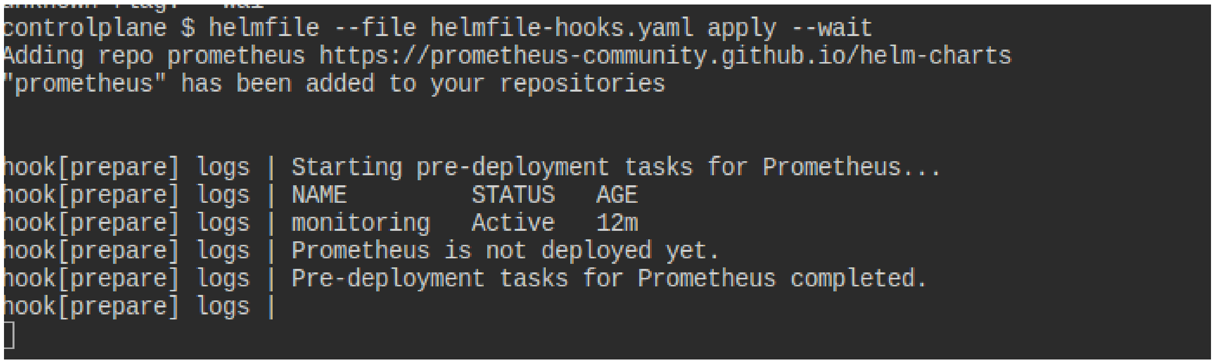
```
# helmfile.yaml
repositories:
  - name: prometheus
    url: https://prometheus-community.github.io/helm-charts

releases:
  - name: prometheus
    namespace: monitoring
    chart: prometheus-community/prometheus
    version: "14.8.0"
    values:
      - values-prometheus.yaml
    set:
      - name: replicaCount
        value: 3
    hooks:
      - events: ["prepare"]
        command: "./pre-deploy-prometheus.sh"
        showlogs: true
```

The set field overrides specific values in the Helm chart, like setting replicaCount to 3 for Prometheus. The hooks section allows custom scripts to run at specified stages, such as executing ./pre-deploy-prometheus.sh during the prepare event before deployment. This ensures pre-deployment tasks are handled, and showlogs: true ensures script output is displayed.

apply

```
helmfile \
--file helmfile-hooks.yaml \
apply \
--wait
```



```
controlplane $ helmfile --file helmfile-hooks.yaml apply --wait
Adding repo prometheus https://prometheus-community.github.io/helm-charts
"prometheus" has been added to your repositories


hook[prepare] logs | Starting pre-deployment tasks for Prometheus...
hook[prepare] logs | NAME         STATUS   AGE
hook[prepare] logs | monitoring   Active   12m
hook[prepare] logs | Prometheus is not deployed yet.
hook[prepare] logs | Pre-deployment tasks for Prometheus completed.
hook[prepare] logs |
```

As you can see, hooks initialized pre-deployment tasks.

More About Hooks:

Hooks in Helmfile allow custom logic to run at different stages of the deployment, such as before, during, or after installation. They are useful for tasks like environment setup, configuration validation, or resource cleanup. Hooks automate workflows, making deployments more flexible and customizable.

6) Secrets

We can define values.yaml files as secrets.

```
cat helmfile-hooks.yaml
```

```
repositories:
 - name: bitnami
 url: https://charts.bitnami.com/bitnami

releases:
 - name: myapp
 chart: bitnami/nginx
 namespace: default
 values:
 - values.yaml # Regular values file
 secrets:
 - ./values-secrets-encrypted.yaml
```

In this example, secrets is used to reference an encrypted values-secrets-encrypted.yaml file, typically encrypted using tools like SOPS to securely store sensitive data. When applying the Helmfile, Helmfile will decrypt the secrets and inject them into the release alongside the regular values.yaml file.

Install SOPS. Script will prompt for Key name and your email addres.

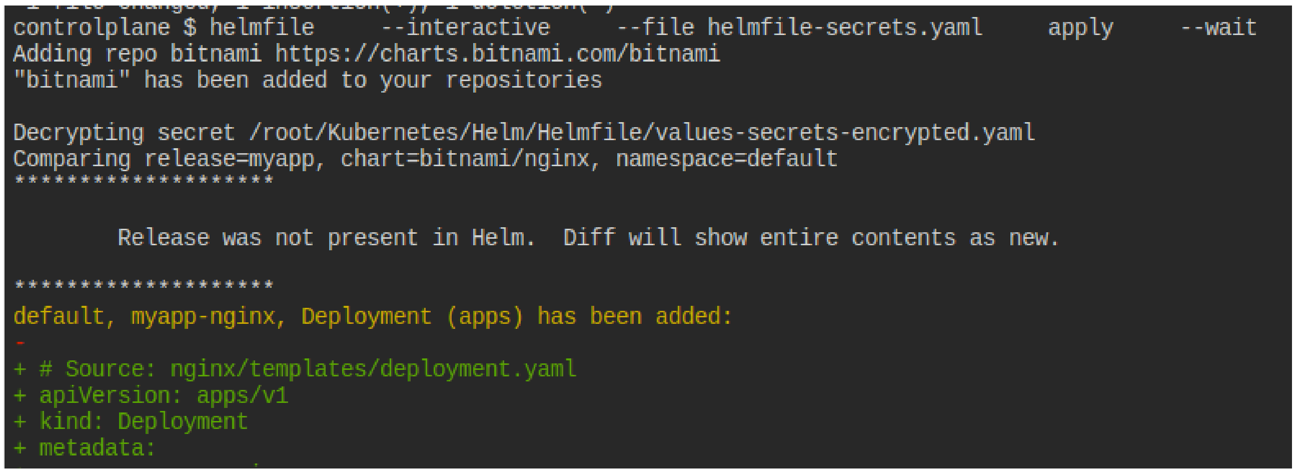
```
/.installsops.sh
```

Generate encrypted yaml file. Check Bash script if you want to.

```
./encrypt.sh
```

Now we can apply helmfile with secrets:

```
helmfile \
   --interactive \
   --file helmfile-secrets.yaml \
   apply \
   --wait
```



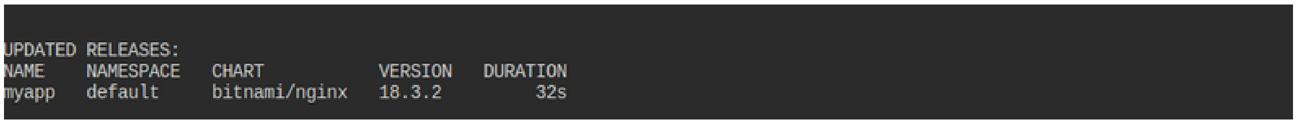
```
controlplane $ helmfile     --interactive     --file helmfile-secrets.yaml     apply     --wait
Adding repo bitnami https://charts.bitnami.com/bitnami
"bitnami" has been added to your repositories

Decrypting secret /root/Kubernetes/Helm/Helmfile/values-secrets-encrypted.yaml
Comparing release=myapp, chart=bitnami/nginx, namespace=default
********************

        Release was not present in Helm.  Diff will show entire contents as new.

********************
default, myapp-nginx, Deployment (apps) has been added:
-
+ # Source: nginx/templates/deployment.yaml
+ apiVersion: apps/v1
+ kind: Deployment
+ metadata:
```

App created with helmfile:



```
UPDATED RELEASES:
NAME    NAMESPACE   CHART           VERSION   DURATION
myapp   default     bitnami/nginx   18.3.2         32s
```

7) Multiple releases

We can deploy multiple helmcharts in 1 helmfile:

```
cat helmfile-secrets.yaml
```

```
# helmfile.yaml
repositories:
  - name: prometheus-community
    url: https://prometheus-community.github.io/helm-charts
  - name: bitnami
    url: https://charts.bitnami.com/bitnami

releases:
  - name: prometheus
    namespace: monitoring
    chart: prometheus-community/prometheus
    values:
      - values-prometheus.yaml
    set:
      - name: replicaCount
        value: 3
    hooks:
      - events: ["prepare"]
        command: "./pre-deploy-prometheus.sh"
        showlogs: true

  - name: myapp
    chart: bitnami/nginx
    namespace: default
    values:
      - values-nginx.yaml           # Regular values file
    hooks:
      - events: ["prepare"]
        command: "./pre-deploy-nginx.sh"
        showlogs: true
```

In this Helmfile example, two Helm charts—Prometheus and Nginx—are deployed in separate namespaces (monitoring and web). Each release has its own configuration, including values files (values-prometheus.yaml and values-nginx.yaml), specific settings like replica counts and image tags, and pre-deployment hooks for custom actions.

This setup allows you to manage multiple applications in a single Helmfile, ensuring each chart is deployed with its unique configuration and lifecycle events.

Now we can apply helmfile with secrets:

```
helmfile \
 --interactive \
 --file helmfile-multi.yaml \
 apply \
 --wait
```

8) Templates

When we deploy multiple helmcharts, some labels repeat themselves. To make helmfile smaller, think about using templates:

```
cat helmfile-templates.yaml
```

```
repositories:
  - name: bitnami
    url: https://charts.bitnami.com/bitnami

# Define a template for Nginx releases
templates:
  default: &default
    chart: bitnami/nginx



      - values-nginx.yaml  # Custom values file
    set:
      - name: image.tag
        value: "1.21.0"  # Override image tag
    hooks:
      - events: ["prepare"]
        command: "./pre-deploy-nginx.sh"
        showlogs: true
```

```
# Use the template to define multiple releases
releases:
  - name: nginx-main
    namespace: web
    <<: *default  # Use the template for the main Nginx deployment

  - name: nginx-backup
    namespace: backup
    <<: *default  # Use the template for a backup Nginx deployment
```

Templates in Helmfile centralize shared configurations like chart details and values, reducing redundancy. The default template is reused across releases using <<: *default, simplifying the Helmfile. This approach makes deployments easier to manage and maintain.

6) Managed dev, stage, prod environments

We can manage charts accordingly to their environemnts:

```
cat helmfile-dev-prod-stage.yaml
```

```
environments:
  prod:
    values:
      - values-prod.yaml  # Values specific to the prod environment
  stage:
    values:
      - values-stage.yaml  # Values specific to the stage environment
  dev:
    values:
      - values-dev.yaml  # Values specific to the dev environment
```

You can specify the environment to deploy in Helmfile using the -e or --environment flag.

Here’s how to define and deploy to a specific environment:

for dev

```
helmfile -e dev --file helmfile-dev-prod-stage.yaml apply --wait
```

for stage

```
helmfile -e stage --file helmfile-dev-prod-stage.yaml apply --wait
```

for prod

```
helmfile -e prod --file helmfile-dev-prod-stage.yaml apply --wait
```

values form environment yaml, override base helmfile yaml:

```
cat helmfile-dev-prod-stage.yaml
```

```
repositories:
  - name: bitnami
    url: https://charts.bitnami.com/bitnami
  - name: prometheus-community
    url: https://prometheus-community.github.io/helm-charts

releases:
  - name: prometheus
    namespace: monitoring
    chart: prometheus-community/prometheus
    version: "14.8.0"
    values:
      - values-common.yaml  # Common values shared across environments
    hooks:
      - events: ["prepare"]
        command: "./pre-deploy-prometheus.sh"
        showlogs: true

  - name: myapp
    chart: bitnami/nginx
    namespace: default
    values:
      - values-common.yaml  # Common values shared across environments
    set:
      - name: image.tag
        value: "1.21.0"
    hooks:
      - events: ["prepare"]
        command: "./pre-deploy-nginx.sh"
        showlogs: true
```

Using environment-specific values in Helmfile allows tailored configurations for dev, stage, and prod, overriding shared settings in values-common.yaml. The -e flag selects the appropriate environment, ensuring the corresponding values-dev.yaml, values-stage.yaml, or values-prod.yaml is applied during deployment

Kubernetes Packages: Kustomize

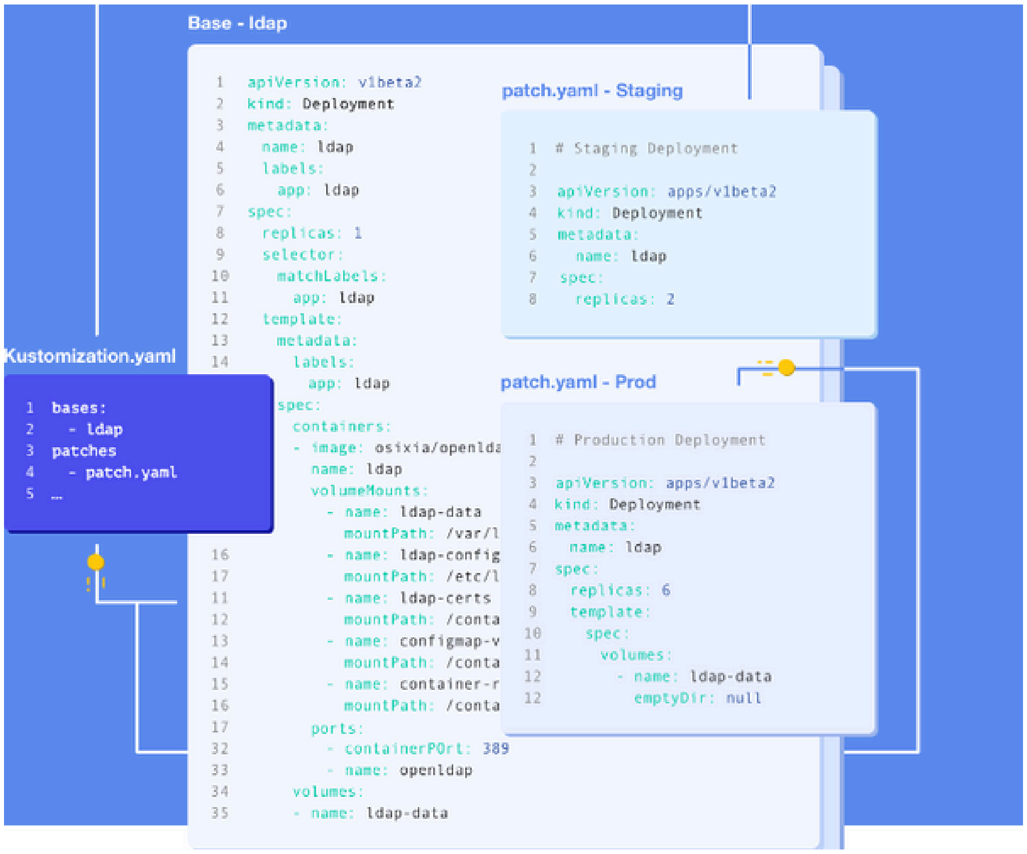
1) Intro & Setup

The best way to understand Kustomize is to visit its official website directly:

Kubernetes native configuration management

Kustomize introduces a template-free way to customize application configuration that simplifies the use of off-the-shelf applications. Now, built into `kubectl` as `apply -k`.

Kustomize is a template-free configuration management tool for Kubernetes applications.[5]



```
bases:
  - ldap
patches
  - patch.yaml
…
```

```
apiVersion: v1beta2
kind: Deployment
metadata:
  name: ldap
  labels:
    app: ldap
spec:
  replicas: 1
  selector:
    matchLabels:
      app: ldap
  template:
    metadata:
      labels:
        app: ldap
    spec:
      containers:
      - image: osixia/openlda
        name: ldap
        volumeMounts:
          - name: ldap-data
            mountPath: /var/l
          - name: ldap-config
            mountPath: /etc/l
          - name: ldap-certs
            mountPath: /conta
          - name: configmap-v
            mountPath: /conta
          - name: container-r
            mountPath: /conta
        ports:
          - containerPOrt: 389
          - name: openldap
      volumes:
      - name: ldap-data
```

```
# Staging Deployment

apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: ldap
 spec:
  replicas: 2
```

```
# Production Deployment

apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: ldap
spec:
  replicas: 6
  template:
    spec:
      volumes:
        - name: ldap-data
          emptyDir: null
```

In Kustomize, bases and patching help manage Kubernetes resources across environments.

Base: A base is a set of common Kubernetes resources (like Deployments or ConfigMaps) stored in a directory, reusable across environments (e.g., dev, prod).
Patching: Patching allows modifying specific fields in base resources (e.g., replica count, image tag) without changing the base itself, using methods like strategicMerge or jsonPatch.

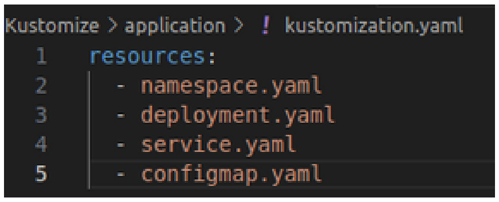
Download git repo

```
git clone https://github.com/MichaelRobotics/Kubernetes.git
cd Kubernetes/Kustomize
```

A kustomization.yaml lists all the YAMLs that define your application.

5

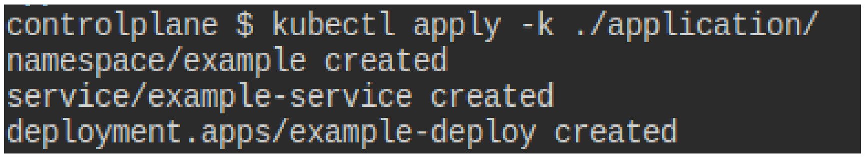
```
cat application/kustomization.yaml
```



```yaml
resources:
  - namespace.yaml
  - deployment.yaml
  - service.yaml
  - configmap.yaml
```

To deploy with Kustomize, simply point kubectl to the kustomization.yaml location.

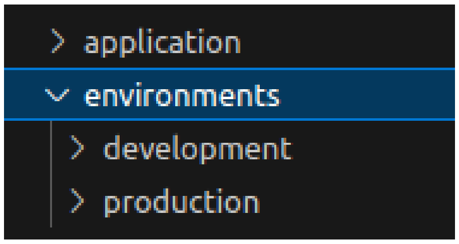
```
kubectl apply -k ./application/
```



```
controlplane $ kubectl apply -k ./application/
namespace/example created
service/example-service created
deployment.apps/example-deploy created
```

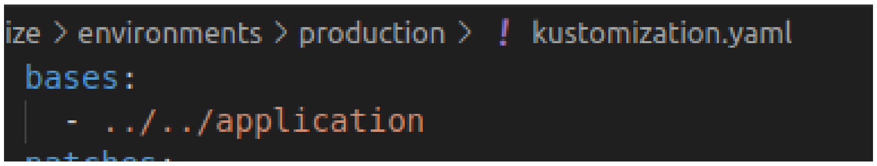
2) Multi-environment setup with overlays

Each environment will be stored in its own folder within the environment directory.


```
> application
v environments
  > development
  > production
```

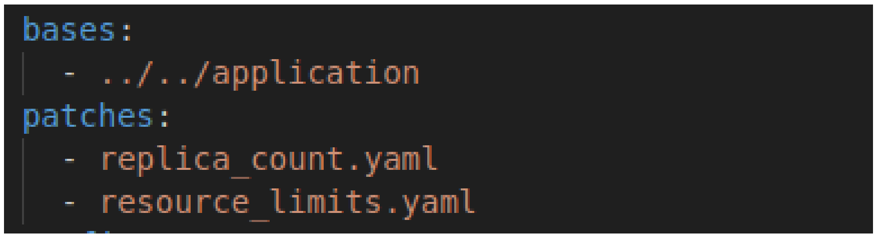
Each environment has its own kustomization.yaml, which points to the "base". The contents of this file will override the base YAMLs. Start by checking the production folder.

```
cat environments/production/kustomization.yaml
```

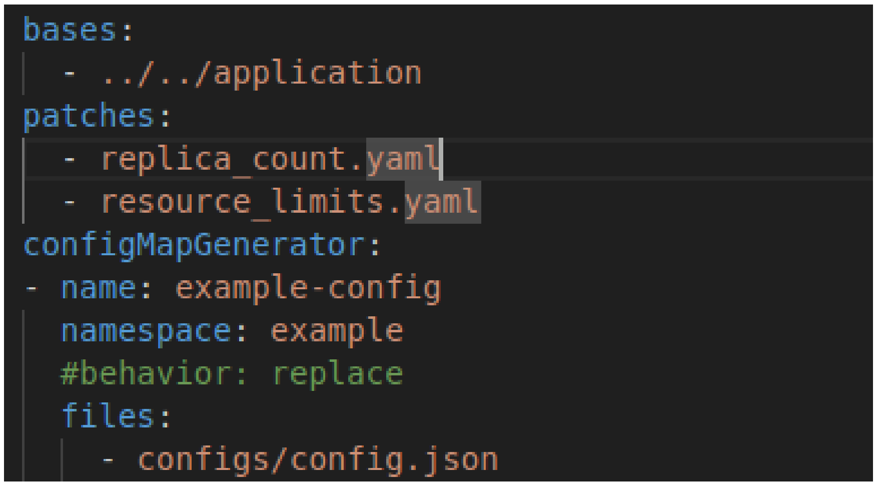

```
ize > environments > production > ! kustomization.yaml
bases:
  - ../../application
```

5

As mentioned, the bases label points to the base application YAML. The patches parameter defines the YAMLs we want to add or overwrite in our app.


```
bases:
  - ../../application
patches:
  - replica_count.yaml
  - resource_limits.yaml
```

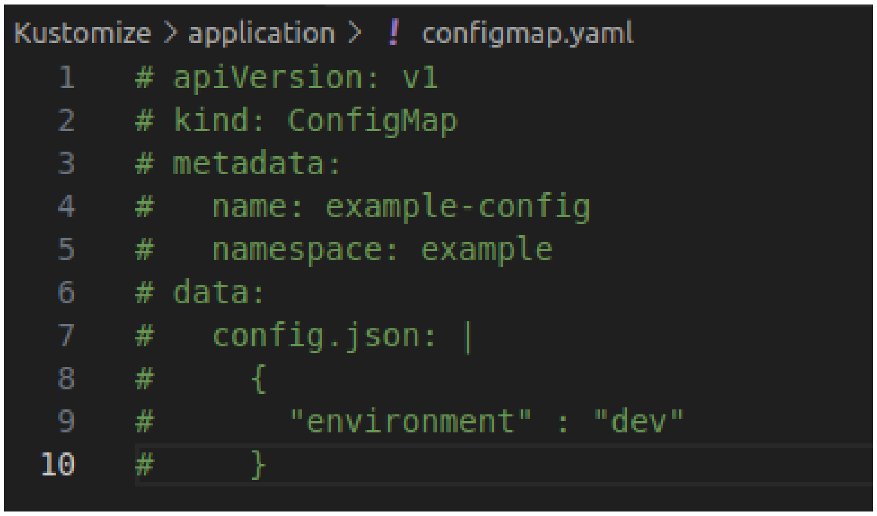
The configMapGenerator points to a JSON file, which contains the configuration we want to replace. Check the behavior parameter, which is commented out with #, as it defines how the old configuration should be replaced.



```yaml
bases:
  - ../../application
patches:
  - replica_count.yaml
  - resource_limits.yaml
configMapGenerator:
- name: example-config
  namespace: example
  #behavior: replace
  files:
    - configs/config.json
```

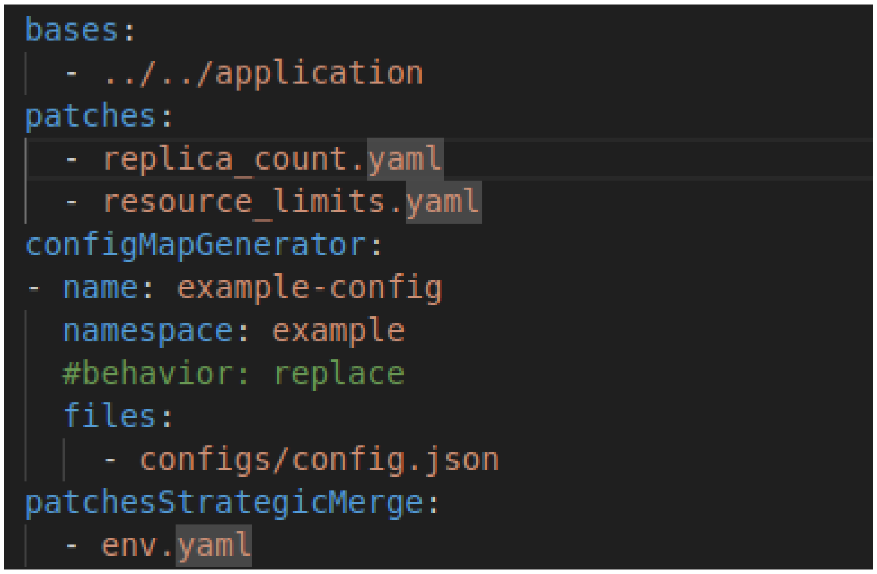
When you overwrite a ConfigMap, changes won't be applied until the pods are restarted, and this doesn't happen automatically. To make Kustomize restart the pods, the behavior[5] label must be uncommented, which will ensure the ConfigMap is replaced and trigger the restart in the same way as the base application ConfigMap.

```
cat application/configmap.yaml
```



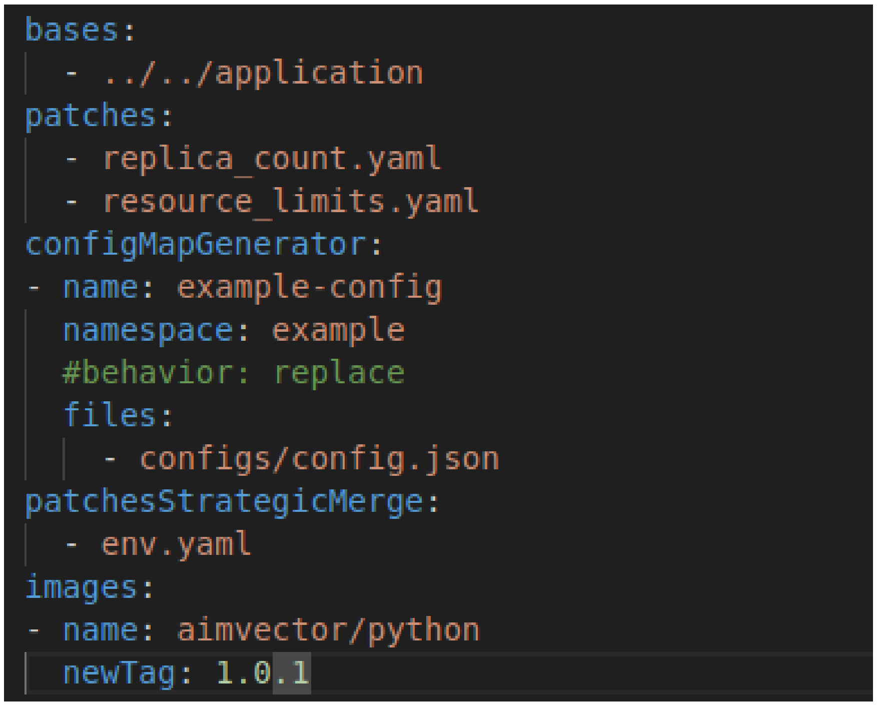
```yaml
# apiVersion: v1
# kind: ConfigMap
# metadata:
#   name: example-config
#   namespace: example
# data:
#   config.json: |
#     {
#       "environment" : "dev"
#     }
```

Next, we can modify or add app environment variables with patchesStrategicMerge. All environment variables are defined in env.taml in production directory.



```yaml
bases:
  - ../../application
patches:
  - replica_count.yaml
  - resource_limits.yaml
configMapGenerator:
- name: example-config
  namespace: example
  #behavior: replace
  files:
    - configs/config.json
patchesStrategicMerge:
  - env.yaml
```

We can modify or add app environment variables using patchesStrategicMerge. All[5] environment variables are defined in env.yaml within the production directory.



```yaml
bases:
  - ../../application
patches:
  - replica_count.yaml
  - resource_limits.yaml
configMapGenerator:
- name: example-config
  namespace: example
  #behavior: replace
  files:
    - configs/config.json
patchesStrategicMerge:
  - env.yaml
images:
- name: aimvector/python
  newTag: 1.0.1
```

Kubernetes Packages: Final Thoughts Helm vs Kustomize

Helm: Best for Complex Deployments

Helm is a powerful tool designed for complex and large-scale deployments. It allows you to package Kubernetes manifests into reusable, shareable charts, complete with dynamic templating for custom configurations. Helm excels when managing intricate applications with multiple services, dependencies, and environment-specific configurations.

Kustomize: Built for Lightweight Applications

Kustomize, on the other hand, is a leaner, more straightforward tool. It shines when[5] managing lightweight applications where configuration customization is minimal. Kustomize uses overlays and patches to modify existing Kubernetes manifests declaratively, avoiding the need for templates.

A Simple Alternative for Minor Changes

If your only goal is to make a small change, such as updating the image name in a deployment, using Helm or Kustomize might be overkill. Instead, you can simply modify your YAML files directly using a tool like sed:

```
sed -i 's|old-image-name:tag|new-image-name:tag|' deployment.yaml
```

common troubleshooting

1) Helm Chart Rendering Issue

Cause: Errors in values.yaml or Helm templates.
Solution: Run helm lint and helm template to debug and verify chart output.

2) Kustomize Patch Not Applying

Cause: Target resource mismatch (name/namespace).
Solution: Use kubectl kustomize to preview and verify patch alignment.

3) SOPS Decryption Fails

Cause: Missing GPG keys or incorrect AWS KMS config.
Solution: Test with sops -d <file> and check environment variables for key access.

4) Check my Kubernetes Troubleshooting series:

Michael Robotics
Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics

Learn more about Kubernetes

Check Kubernetes and piyushsachdeva - great docs!

Setup a Multi Node Kubernetes Cluster

kubeadm is a tool to bootstrap the Kubernetes cluster

https://github.com/piyushsachdeva/CKA-2024/tree/main/Resources/Day27

Kubernetes Documentation

This section lists the different ways to set up and run Kubernetes

https://kubernetes.io/docs/setup/

Share, comment, DM and check GitHub for scripts & playbooks created to automate process.

Check my GitHub

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics

PS.

If you need a playbook or bash script to manage KVM on a specific Linux distribution, feel free to ask me in the comments or send a direct message!