Contents

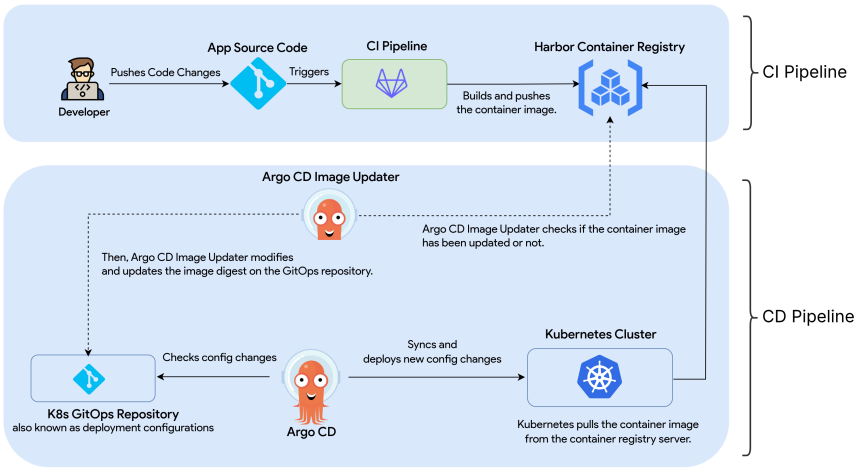

Figure 1: gitops-featured-image

GitOps CI/CD in Kubernetes

A Hands-on Practical Guide to Building a Fully Automated CI/CD Pipeline Using GitLab CI and GitOps Argo CD on Kubernetes

Introduction

This hands-on practical guide is to demonstrate GitOps CI/CD automation in Kubernetes with GitLab CI and Argo CD using the podinfo-sample Python application. It mainly focuses on how to containerize an application, configure Continuous Integration (CI), Continuous Deployment (CD) and fully automated application deployment on Kubernetes.

Summary: Objectives

What you'll learn in this hands-on practical guide:

- Write a Dockerfile to containerize a sample Python application.
- Configure the GitLab CI pipeline to build and push Docker container images using Buildah.
- Setup a Kubernetes Cluster with K3s, Lightweight Kubernetes.

- Write a Helm Chart to deploy the podinfo-sample Python application on Kubernetes.
- Configure Argo CD as GitOps CD to deploy applications automatically on Kubernetes.
- Configure Argo CD Image Updater to automate updating and pulling the Docker container images automatically.

This GitOps hands-on practical guide is based on the GitOps in Kubernetes with GitLab CI and ArgoCD article by Poom Wettayakorn. But, I will share more details and focus on a beginner-friendly guide.

Prerequisites

Make sure you have installed the following:

- Familiar with basic Linux commands
- Docker Engine
- Linux installed VM or server or local machine (e.g., Ubuntu, Fedora, RHEL, etc.)

[1] Containerizing an application

> [!NOTE]
>
> Before You Begin
>
> Make sure you are familiar with Docker before you begin.
>
> - Install Docker: https://www.docker.com/get-started
> - Dockerfile Reference: https://docs.docker.com/reference/dockerfile
>
> If you are not familiar with Docker, please learn it first with the following Docker for Beginners tutorial.
>
> Docker for Beginners: https://docker-curriculum.com/

Introduction to a sample Python application

In this guide, I will use the podinfo-sample Python application to demonstrate building a fully automated GitOps CI/CD pipeline in Kubernetes.

Podinfo is an open-source and simple Python Flask application, originally developed by Poom Wettayakorn that shows the following information in UI:

- Namespace
- Node Name
- Pod Name
- Pop IP Address

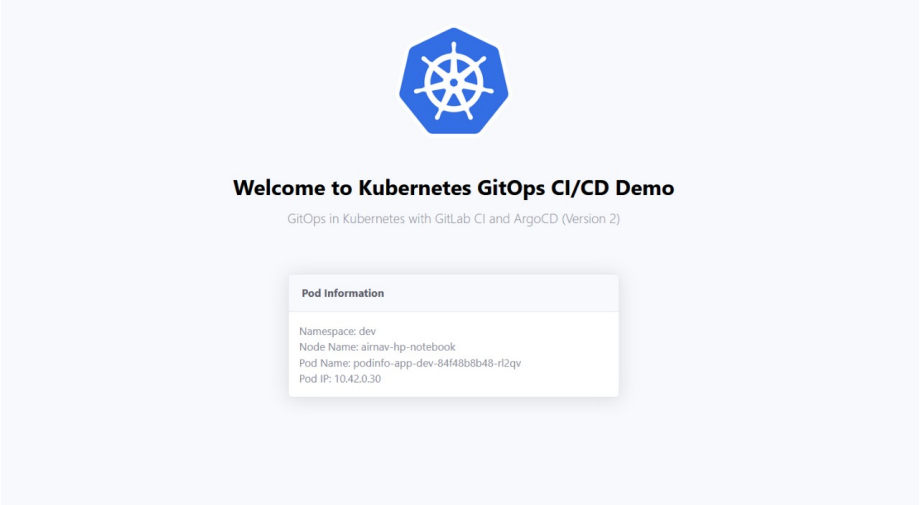

Figure 2: screenshot-podinfo-demo

I've forked Poom Wettayakorn's Podinfo application under my GitLab account and customized it. I will use the following customized version of the Podinfo sample app in this GitOps hands-on practical guide.

Git Repository: https://gitlab.com/thezawzaw/podinfo-sample

Fork this Git repository under your GitLab account and clone with the Git command-line tool.

```
$ git clone git@gitlab.com:<your-username>/podinfo-sample.git
```

For example, replace `gitops-example` with your username.

```
$ git clone git@gitlab.com:gitops-example/podinfo-sample.git
```

Writing a Dockerfile

Firstly, you will need to write Dockerfile to containerize this Python web application. In the podinfo-sample Git repository, I've already written a Dockerfile to containerize the app.

```
#
# Stage (1): Builder Stage
#
# Install the app source code and required Python packages.
#
FROM python:3.12-alpine AS builder

ENV APP_WORKDIR=/app
ENV PATH="${APP_WORKDIR}/venv/bin:$PATH"

WORKDIR ${APP_WORKDIR}

COPY . .

RUN apk add --no-cache \
    gcc musl-dev && \
    python -m venv venv && \
    pip install --upgrade pip && \
    pip install -r requirements.txt

#
# Stage (2): Runtime Stage
#
# The final runtime environment for serving the Podinfo sample application.
#
FROM python:3.12-alpine AS runtime

ENV FLASK_APP=run.py
ENV APP_WORKDIR=/app
ENV APP_USER=zawzaw
ENV APP_GROUP=zawzaw
ENV APP_PORT=5005
ENV PATH="${APP_WORKDIR}/venv/bin:$PATH"

RUN adduser -D ${APP_USER}

WORKDIR ${APP_WORKDIR}

RUN pip uninstall pip -y
COPY --from=builder --chown=${APP_USER}:${APP_GROUP} ${APP_WORKDIR} ${APP_WORK

USER ${APP_USER}

EXPOSE ${APP_PORT}

ENTRYPOINT ["gunicorn", "--config", "gunicorn-cfg.py", "run:app"]
```

Explanation:

In the Stage (1) — Builder Stage:

- Create an application workdir.
- Add the application source code, create the Python virtual environment (venv) and install required packages with pip.

In the Stage (2) — Runtime Stage:

- Copy the created Python venv from the builder stage.
- Then, create and switch to a normal user and serve the Podinfo Python application with the Gunicorn server.

Building and Testing the Container Image Locally

To build the Docker image locally, run the following `docker build` command:

```
$ cd podinfo-sample
$ docker build -t podinfo-sample:local .
```

C To run and test the Podinfo application locally with `docker run`:

```
$ docker run -p 5005:5005 -it --rm --name podinfo podinfo-sample:local
```

To test the Podinfo application locally, open the following localhost address in the web browser:

URL: http://localhost:5005

---

[2] Building GitLab CI Pipelines

Introduction to GitLab CI

- GitLab CI is a Continuous Integration (CI) that automates building and testing the code changes via a `.gitlab-ci.yml` file on your GitLab repository. For Example, in this GitOps CI/CD guide, I will use GitLab CI to build the container image of the Podinfo Python application automatically.

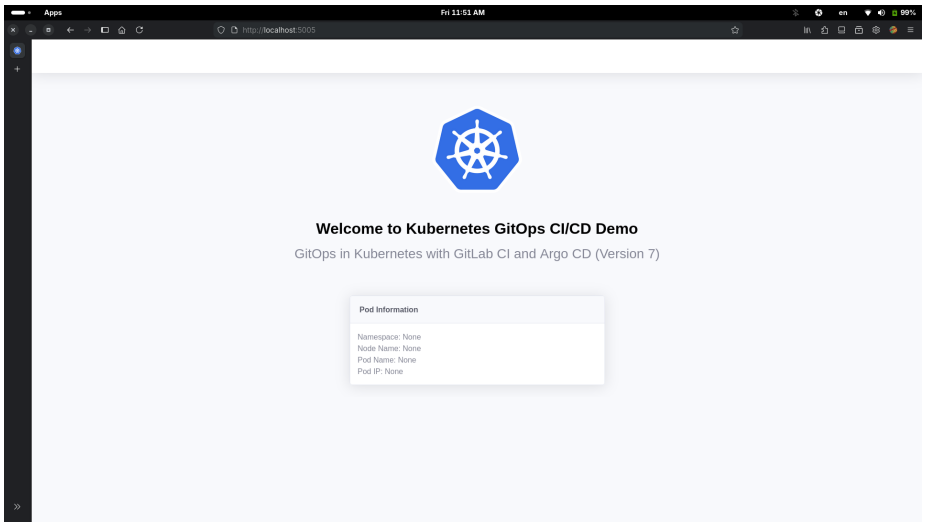

Figure 3: screenshot-podinfo-docker

- GitLab Runner is an application or server that runs GitLab CI jobs in a pipeline. GitLab CI jobs are defined and configured in the `.gitlab-ci.yml` file that automatically triggers when you push the code changes to GitLab. Then, GitLab Runner runs these CI jobs on the server or computing infrastructure. For more information about GitLab Runner, please see https://docs.gitlab.com/runner/#what-gitlab-runner-does.

Before you begin, make sure you have learned the basics of GitLab CI and YAML syntax. Please, start with the following tutorials.

- Get started with GitLab CI: https://docs.gitlab.com/ci/
- CI/CD YAML Syntax Reference: https://docs.gitlab.com/ci/yaml/

Installing a GitLab Runner

In this guide, I will use a self-managed GitLab Runner for running GitLab CI jobs for more control. but it's OPTIONAL. You can also use GitLab-hosted GitLab Runners. If you want to use GitLab-hosted Runners, please see https://docs.gitlab.com/ci/runners/hosted_runners/linux/.

> GitLab provides the GitLab Runner packages for most Linux distributions. But, installation depends on your Linux distribution. In this guide, I will focus on RHEL-based Linux systems.

Add the following GitLab RPM repository (e.g., Fedora Linux):

```
$ curl -L "https://packages.gitlab.com/install/repositories/runner/gitlab-runn
```

Install the GitLab Runner with `yum` or `dnf`:

```
$ sudo yum install gitlab-runner
```

(OR)

```
$ sudo dnf install gitlab-runner
```

For any other Linux distributions, please see https://docs.gitlab.com/runner/install/linux-repository/.

Registering a GitLab Runner

After you install a GitLab Runner, you need to register this for your Podinfo Git repository. Firstly, you need to fork the Podinfo sample application repository that I've mentioned previously. (If you have already forked, you don't need to fork again.)

Podinfo Sample Git Repository: https://gitlab.com/thezawzaw/podinfo-sample

Go to your Podinfo Git repository » Settings » CI/CD » Runners » Three dots menu and then copy your Registration token. (Note: Registration tokens are deprecated, but you can still use them.)

And then, register with the GitLab URL and registration token. Replace with your actual registration token.

```
$ sudo gitlab-runner register --url https://gitlab.com/ --registration-token <
```

And then, you also need to set the `executor`, `docker-image`, and `description` in the interactive shell mode.

(OR)

Alternatively, you can register a GitLab Runner in the non-interactive shell mode.

```
sudo gitlab-runner register \
  --non-interactive \
  --url "https://gitlab.com/" \
  --token <your-registration-token> \
  --executor "docker" \
  --docker-image alpine:latest \
  --description "GitLab Runner for the Podinfo application"
```

And then, you can check GitLab Runner `config.toml` configuration in the `/etc/gitlab-runner/config.toml` file.

```
$ cat /etc/gitlab-runner/config.toml
```

Output:

```
concurrent = 8
check_interval = 0
connection_max_age = "15m0s"
shutdown_timeout = 0

[session_server]
  session_timeout = 1800

[[runners]]
  name = "Podinfo GitLab Runner on Fedora Linux"
  url = "https://gitlab.com/"
  id = 50528940
  token = "<registration-token-example>" # Replace with your registration toke
  token_obtained_at = 2025-11-13T06:27:20Z
  token_expires_at = 0001-01-01T00:00:00Z
  executor = "docker"
  [runners.cache]
    MaxUploadedArchiveSize = 0
    [runners.cache.s3]
    [runners.cache.gcs]
    [runners.cache.azure]
  [runners.docker]
    image = "alpine:latest"
    privileged = false
    tls_verify = false
    pull_policy = "if-not-present"
    disable_entrypoint_overwrite = false
    oom_kill_disable = false
    disable_cache = false
    shm_size = 0
```

```
    network_mtu = 0
    volumes = ["/cache"]
```

Configuring GitLab CI Pipeline to Build and Push Container Images

In this section, I will use Buildah to build and push Docker container images automatically to the Harbor Docker registry.

Buildah is a tool that facilitates building Open Container Initiative (OCI) Container images. Buildah is designed to run in Userspace, also known as Rootless mode and does not require a root-privileged daemon like traditional Docker daemon. This is one of its primary advantages, especially in secured and automated CI/CD environments. Please, see the following GitHub wiki page.

Building Container Images with Buildah in GitLab CI: https://github.com/thezawzaw/platform-wiki/wiki/Building-Container-Images-with-Buildah-in-GitLab-CI

Before you configure GitLab CI pipeline, make sure you add two GitLab CI variables `REGISTRY_HOST` `DOCKER_CFG` on the Podinfo repository:

Go to your Podinfo Git repository » Project Settings » CI/CD » Variables, and add the following key/value GitLab CI variables.

> [!NOTE]
>
> Replace with your container registry credentials.

- Key: `REGISTRY_HOST`, Value: `<your-registry-host>`
- Key: `DOCKER_CFG`, Value: `<your-registry-auth-creds>`

`REGISTRY_HOST` for your container registry host.

`DOCKER_CFG` for login credentials to access your container registry server.

For Example,

Key	Value
`REGISTRY_HOST`	`harbor-repo-example.ops.io`
`DOCKER_CFG`	`{"auths": {"harbor-repo-example.ops.io": {"auth": "YWRtaW46SGFyYm9yMTIzNDU="}}}`

I've already created `.gitlab-ci.yml` GitLab CI configuration on the Podinfo Git repository. But, you can write your own `.gitlab-ci.yml` configuration under your Podinfo sample project's root directory.

GitLab CI Configuration https://gitlab.com/thezawzaw/podinfo-sample/-/blob/main/.gitlab-ci.yml

```yaml
#
# GitLab CI Configuration
#

#
# Define the CI stages here.
#
stages:
  - build
  - scan

# Define global variables here.
variables:
  IMAGE_REPO: "${REGISTRY_HOST}/library/${CI_PROJECT_NAME}"

################################################################################
#
# GitLab CI Templates
#
################################################################################

# Template ---> template_build
# to build and push the Docker container images to the Container Registry serv
.template_build: &template_build
  stage: build
  image: quay.io/buildah/stable
  variables:
    BUILDAH_FORMAT: docker
    TARGET_IMAGE_TAG: ""
  script:
    - echo ${DOCKER_CFG} > /home/build/config.json
    - export REGISTRY_AUTH_FILE=/home/build/config.json
    - echo "Building Docker container image [ $IMAGE_REPO:$TARGET_IMAGE_TAG ]
    - >-
      buildah build
      --file ${CI_PROJECT_DIR}/Dockerfile
      --layers
      --cache-to ${IMAGE_REPO}/cache
      --cache-from ${IMAGE_REPO}/cache
      --tls-verify=false
      --tag ${IMAGE_REPO}:${TARGET_IMAGE_TAG} .
    - buildah push --tls-verify=false ${IMAGE_REPO}:${TARGET_IMAGE_TAG}
    - buildah rmi -f ${IMAGE_REPO}:${TARGET_IMAGE_TAG}
```

```
# Template ---> template_trivy_scan
# to scan and find vulnerabilities of the Docker container images.
.template_trivy_scan: &template_trivy_scan
  stage: scan
  image:
    name: docker.io/aquasec/trivy:0.67.2
    entrypoint: [""]
  variables:
    TRIVY_SEVERITY: "HIGH,CRITICAL"
    TRIVY_EXIT_CODE: "1"
    TARGET_IMAGE_TAG: ""
  script:
    - echo "Scanning Docker container image [ $IMAGE_REPO:$TARGET_IMAGE_TAG ]
    - >-
      trivy --cache-dir "${CI_PROJECT_DIR}/trivy/"
      image
      --image-src remote
      --insecure ${IMAGE_REPO}:${TARGET_IMAGE_TAG}
  cache:
    key: trivy-cache
    paths:
      - "${CI_PROJECT_DIR}/trivy/"
  when: manual


##############################################################################
#
# GitLab CI Jobs
#
##############################################################################

#
# Build CI Job ---> build-image-dev
# to build the Docker container image with the Git branch name as image tag na
#
build-image-dev:
  <<: *template_build
  variables:
    TARGET_IMAGE_TAG: "${CI_COMMIT_REF_SLUG}"
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'

#
# Build CI Job ---> build-image-main
# to build the Docker container image with latest image tag name when you push
```

```
#
build-image-main:
  <<: *template_build
  variables:
    TARGET_IMAGE_TAG: "latest"
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'

#
# Build CI Job ---> build-image-tag
# to build the Docker container image with the Git tag name as image tag when
#
build-image-tag:
  <<: *template_build
  variables:
    TARGET_IMAGE_TAG: "${CI_COMMIT_TAG}"
  rules:
    - if: "$CI_COMMIT_TAG"

#
# Scan CI Job ---> trivy-scan-dev
# to scan and find vulnerabilities of the Docker container image when you push
#
trivy-scan-dev:
  <<: *template_trivy_scan
  variables:
    TARGET_IMAGE_TAG: "${CI_COMMIT_REF_SLUG}"
  rules:
    - if: '$CI_COMMIT_BRANCH == "develop"'

#
# Scan CI Job ---> trivy-scan-main
# to scan and find vulnerabilities of the Docker container image when push cha
#
trivy-scan-main:
  <<: *template_trivy_scan
  variables:
    TARGET_IMAGE_TAG: "latest"
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'

#
# Scan CI Job ---> trivy-scan-tag
# to scan and find vulnerabilities of the Docker container image when you crea
#
trivy-scan-tag:
```

```
  <<: *template_trivy_scan
  variables:
    TARGET_IMAGE_TAG: "${CI_COMMIT_TAG}"
  rules:
    - if: "$CI_COMMIT_TAG"
```

Explanation:

When you push some changes into the Podinfo Git repository, GitLab CI builds and pushes the Docker container image of the Podinfo application to your internal Docker container registry server.

Container image name format → `<your-container-registry>/library/podinfo-sample:<image-tag-name>`

- When you push changes into the `develop` branch, the `<image-tag-name>` will be `develop`.
- When you push changes into the `master` branch, the `<image-tag-name>` will be `latest`.
- When you create a Git tag on the Git repository, the `<image-tag-name>` will be the Git tag number you created.

For Example,

When I push some changes into the `master` branch, the container image name is `harbor-dev-repo.ops.io/library/podinfo-sample:latest`. You can also see the logs in the GitLab CI build job's logs. For reference, please see https://gitlab.com/thezawzaw/podinfo-sample/-/jobs/12120436761

```
...
[2/2]   COMMIT   harbor-dev-repo.ops.io/library/podinfo-
sample:latest
--> Pushing cache [harbor-dev-repo.ops.io/library/podinfo-
sample/cache]:23c4fe872d978253fd66b2a50aa2d6e40da8a09c9f5fd79910fdf7855fc88d7a
--> 048e6533805b
Successfully tagged harbor-dev-repo.ops.io/library/podinfo-
sample:latest
048e6533805b842328e03e7b1b9b2b5efdf2bce11d706283690a8dde4afc78d3
```

---

[3] Creating a Kubernetes Cluster with K3s

[!NOTE]

> If you already have a Kubernetes cluster on your local machine or server, you can skip this step.

In this section, you will learn how to set up a Kubernetes cluster with K3s. I will use K3s in this guide, but you can also use any other Kubernetes distribution.

K3s is a small, minimal, and lightweight Kubernetes distribution, developed and maintained by Rancher. K3s is easy to install, half the memory, all in a single binary of less than 100MB that reduces the dependencies and steps needed to install, run and auto-update a production Kubernetes cluster.

The K3s Official Documentation: https://docs.k3s.io/

Set up K3s Kubernetes Cluster

To bootstrap and set up a K3s Kubernetes cluster, run the following script:

```
#!/usr/bin/env bash

#
# A Shell Script
# to setup and bootstrap the K3s server, also known as Kubernetes control-plan
#
# This script is for setup the single-node K3s Kubernetes cluster.
#

curl -sfL https://get.k3s.io | sh -s - server --write-kubeconfig-mode 644
```

This installation script is for bootstrapping and creating the single-node K3s Kubernetes cluster with proper permissions to the default kubeconfig file. You can also learn how to install on the K8s quickstart guide: https://docs.k3s.io/quick-start

Then, you can check your K3s Kubernetes cluster by running the `kubectl get node` command:

```
$ kubectl get node -o wide
```

Output:

```
AME                     STATUS   ROLES                  AGE   VERSION        IN
airnav-dev-k3s-server   Ready    control-plane,master   22h   v1.33.4+k3s1   1
```

Installing the Dashboard UI to manage Kubernetes Clusters

By default, K3s has built-in kubectl, which is a client command-line tool mainly used to manage and communicate with the Kubernetes clusters. You can use both the kubectl command-line tool and the UI dashboard to manage your Kubernetes clusters.

For a UI dashboard to manage your Kubernetes clusters, I recommend you use Freelens Kubernetes IDE (or) the official Kubernetes Dashboard application.

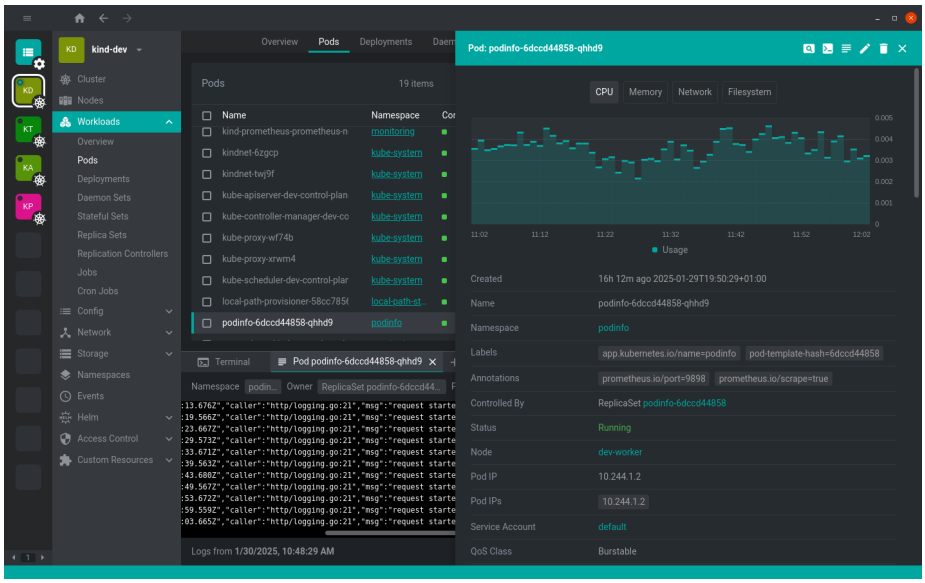

Figure 4: screenshot-freelens-ide

Freelens Kubernetes IDE In this guide, I will use Freelens, a Kubernetes IDE, to manage the K3s Kubernetes cluster.

Freelens is a free and open-source Kubernetes IDE that provides a graphical user interface (UI) for managing and monitoring Kubernetes clusters. Freelens is currently maintained by the community.

- GitHub Repository: https://github.com/freelensapp/freelens
- The Official Website: https://freelensapp.github.io/

Download the Freelens package with curl, for example, RPM-based Linux systems,

```
curl -LO https://github.com/freelensapp/freelens/releases/download/v1.7.0/Free
```

Install the Freelens, for example, RPM-based Linux systems,

```
sudo dnf install ./Freelens-1.7.0-linux-amd64.rpm
```

For more option on installing the Freelens package, please see on GitHub: https://github.com/freelensapp/freelens/blob/main/README.md#downloads

(OR)

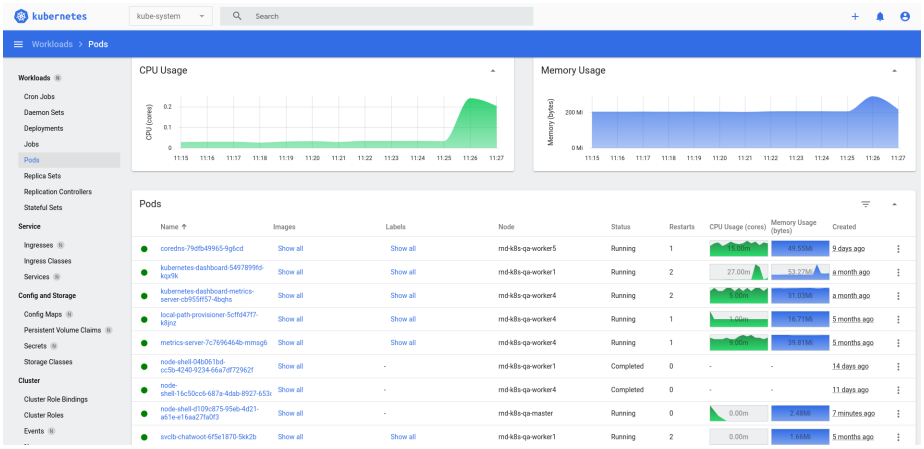

Figure 5: screenshot-k8s-dashboard

Kubernetes Dashboard You can also use the official Kubernetes Dashboard. It is a general-purpose and web-based UI that allows users to manage the Kubernetes clusters and containerized applications running in the cluster and troubleshoot them.

- GitHub Repository: https://github.com/kubernetes/dashboard
- Kubernetes Documentation: https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/

Please, see the detailed documentation on how to install Kubernetes Dashboard: https://github.com/kubernetes/dashboard/blob/master/README.md#installation

[4] Building a Kubernetes Helm Chart from Scratch

Before you write a Kubernetes Helm chart for the Podinfo sample application, make sure you understand Kubernetes core components and resource types. For example, Kubernetes cluster architecture, nodes, services, pods, deployments, ingress, and so on.

Firstly, you must learn to understand Kubernetes basics. If you are a beginner, I would like to recommend the following useful links to learn Kubernetes:

- Learn Kubernetes Basics: https://kubernetes.io/docs/tutorials/kubernetes-basics
- Kubernetes Core Concepts and Components: https://kubernetes.io/docs/concepts/

Introduction to Helm

Helm is a Kubernetes package manager CLI tool that manages and deploys Helm charts.

Helm Charts are collection and packages of pre-configured application ressources which can be deployed as one unit. Helm charts help you define, install, upgrade and deploy applications easily on Kubernetes cluster.

- The Offical Website: https://helm.sh/
- Helm Charts: https://artifacthub.io/

Installation and Setup

To install the Helm command-line tool with script, run the following command:

```
$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/s
```

```
$ chmod 700 get_helm.sh && ./get_helm.sh
```

(OR)

You can install the Helm command-line tool with any other package managers. Please, see on the Helm documentation: https://helm.sh/docs/intro/install#through-package-managers.

Writing a Helm Chart for the Podinfo application

In this section, I will write a Kubernetes Helm chart from scratch for the Podinfo Python application. I had published an article, and you can also learn about how to write a Kubernetes Helm chart from scratch with the following article.

- Writing a Kubernetes Helm Chart from Scratch: https://www.zawzaw.blog/k8s-write-k8s-helm-chart/

For reference, I've already written a Helm chart for the Podinfo Python application. Please, see the following GitOps repository.

- K8s GitOps Repository: https://gitlab.com/thezawzaw/k8s-gitops-airnav-sample/-/tree/main/helm/podinfo-app

Understanding application concepts Before you write a Helm chart for your Podinfo application, make sure you understand the application's concept and how the application works.

In the Podinfo Python application, it will display the following information in the UI:

- Namespace
- Node Name
- Pod Name
- Pop IP Address

For example,

Basically, the Podinfo Python application retrieves the data or information dynamically via the Kubernetes environment variables. So, you need to expose the Pod and Node information to the container via the environment variables in Kubernetes. Then, the app uses these environment variables to retrieve information dynamically.

Reference: Expose Pod Information to Containers Through Environment Variables

For example, you can set these ENV variables with key/value form in your Kubernetes deployment like this:

```
env:
  - name: NODE_NAME
    valueFrom:
```

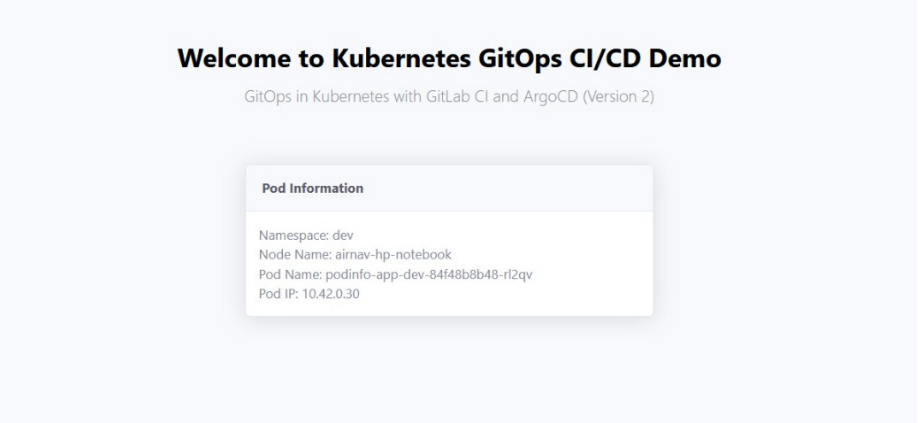

Figure 6: screenshot-podinfo-details

```
      fieldRef:
        fieldPath: spec.nodeName
- name: NAMESPACE
    valueFrom:
      fieldRef:
        fieldPath: metadata.namespace
  - name: POD_NAME
    valueFrom:
      fieldRef:
        fieldPath: metadata.name
  - name: POD_IP
    valueFrom:
      fieldRef:
        fieldPath: status.podIP
```

It's key/value form like this:

- NODE_NAME=spec.nodeName
- NAMESPCE=metadata.namespace
- POD_NAME=metadata.name
- POD_IP=status.podIP

Initializing a Helm Chart with Helm CLI Before you start, make sure you understand Kubernete objects and workloads resources first. If you are not familiar with Kubernetes, you can learn with the Kubernetes Basics tutorial.

Learn Kubernetes Basics: [https://kubernetes.io/docs/tutorials/kubernetes-basics]https://kubernetes.io/docs/tutorials/kubernetes-basics)

Create a Helm chart with the Helm CLI tool.

```
$ helm create pod-info
Creating pod-info
```

Then, Helm automatically generates required Helm templates and values like this:

```
[zawzaw@redhat-linux:~/helm/helm-charts/pod-info]$ tree
.
    Chart.yaml
    templates
        deployment.yaml
        _helpers.tpl
        hpa.yaml
        ingress.yaml
        NOTES.txt
        serviceaccount.yaml
        service.yaml
        tests
            test-connection.yaml
    values.yaml

2 directories, 10 files
```

Customizing and Configuring Helm Chart Basically, Helm Charts have main three categories:

- `Chart.yaml`
  - Define Helm chart name, description, chart revision and so on.
- `templates/`
  - Helm templates are general and dynamic configurations that locate Kubernetes resources written in YAML-based Helm template language. It means that we can pass variables from `values.yaml` file into templates files when we deploy Helm chart. So, values can be changed dynamically based on you configured Helm templates at deployment time.
- `values.yaml`
  - Declare variables to be passed into Helm templates. So, when we run `helm install` to deploy Helm charts, Helm sets this variables into Helm templates files based on you configured templates and values.

In the other words, Helm charts are pre-configured configurations and packages as one unit to deploy applications esaily on Kubernetes cluster.

After initialize a new Helm chart, we need to customize Helm templates and values as you need. It depends on your web application. For pod-info Helm chart, we need to configure the following steps.

Set Docker container image

- In the `values.yaml` file, define variables for the Docker container image that we've built and pushed to your container registry.

  ```
  image:
    repository: harbor-dev-repo.ops.io/library/podinfo-sample:latest
    pullPolicy: IfNotPresent
    tag: "latest"
  ```

- In the `templates/deployment.yaml` file, we can set variables from values.yaml with `.Values.image.repository`, `.Values.image.pullPolicay` and `.Values.image.tag`. It's YAML-based Helm template language syntax. You can learn on The Chart Template Developer's Guide.

  - Get Docker image repository: `.Values.image.repository`
  - Get Docker image pull policy: `.Values.image.pullPolicy`
  - Get Docker image tag: `.Values.image.tag`

So, when need to get variables form `values.yaml` file, we can use `.Values` in Helm templates like this:

```
containers:
  - name: {{ .Chart.Name }}
    image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default .Cha
    imagePullPolicy: {{ .Values.image.pullPolicy }}
```

Set Service Port and Target Port

- In the `values.yaml` file, define variables for sevice type, port and targetPort.

  ```
  service:
    type: NodePort
    port: 80
    targetPort: http
  ```

- In `templates/service.yaml` file, we can set service varibales from values.yaml file like this:
  - Get service type: `.Values.service.type`
  - Get service port: `.Values.service.port`
  - Get service target port: `.Values.service.targetPort`

```
spec:
  type: {{ .Values.service.type }}
  ports:
    - port: {{ .Values.service.port }}
      targetPort: {{ .Values.service.targetPort }}
      protocol: TCP
      name: http
```

Set Target Docker Container Port

- In the `values.yaml` file, define a variable for the Container port number that the Podinfo app is serving and listening to.

```
deployment:
  containerPort: 5005
```

- In the `templates/deployment.yaml` file, set target Docker container port variable from values.yaml file:
  - Get target container port: `.Values.deployment.containerPort`

```
containers:
 - name: {{ .Chart.Name }}
   ports:
    - name: http
      containerPort: {{ .Values.deployment.containerPort }}
      protocol: TCP
```

Set Environment Varibales

- In the `values.yaml` file, define environment variables that the Podinfo application retrieves the data in UI.

```
deployment:
  env:
    - name: NODE_NAME
```

```
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
    - name: NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
    - name: POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: POD_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
```

- In `templates/deployment.yaml`, set environment variables dynamically from the values.yaml file. When you need to pass the array and whole config block into Helm templates, you can use `- with` and `- toYaml`.

```
containers:
  - name: {{ .Chart.Name }}
    {{- with .Values.deployment.env }}
     env:
       {{- toYaml . | nindent 12 }}
    {{- end }}
```

Debugging the Helm Templates After you build Helm chart for the Pod-info application, we can debug and test Helm templates with `helm template` command. So, `helm template` CLI shows passed real values into templates.

Format:

```
helm template <chart_name> <dir_path> --values <values_file_path>
```

For Example:

```
helm template pod-info-dev pod-info --values pod-info/values.yaml
```

If you have syntax errors, Helm shows error messages.

This is automatically generated by Helm Template CLI based on you configured Helm templates and values.

```yaml
---
# Source: pod-info/templates/serviceaccount.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: pod-info-dev
  labels:
    helm.sh/chart: pod-info-0.1.0
    app.kubernetes.io/name: pod-info
    app.kubernetes.io/instance: pod-info-dev
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
---
# Source: pod-info/templates/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: pod-info-dev
  labels:
    helm.sh/chart: pod-info-0.1.0
    app.kubernetes.io/name: pod-info
    app.kubernetes.io/instance: pod-info-dev
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: http
      protocol: TCP
      name: http
  selector:
    app.kubernetes.io/name: pod-info
    app.kubernetes.io/instance: pod-info-dev
---
# Source: pod-info/templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pod-info-dev
  labels:
    helm.sh/chart: pod-info-0.1.0
    app.kubernetes.io/name: pod-info
    app.kubernetes.io/instance: pod-info-dev
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
```

```yaml
spec:
  replicas: 1
  selector:
    matchLabels:
      app.kubernetes.io/name: pod-info
      app.kubernetes.io/instance: pod-info-dev
  template:
    metadata:
      labels:
        app.kubernetes.io/name: pod-info
        app.kubernetes.io/instance: pod-info-dev
    spec:
      serviceAccountName: pod-info-dev
      securityContext:
        {}
      containers:
        - name: pod-info
          securityContext:
            {}
          image: "harbor-dev-repo.ops.io/library/podinfo-sample:latest"
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 8080
              protocol: TCP
          env:
            - name: NODE_NAME
              valueFrom:
                fieldRef:
                  fieldPath: spec.nodeName
            - name: NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: POD_IP
              valueFrom:
                fieldRef:
                  fieldPath: status.podIP
          livenessProbe:
            httpGet:
              path: /
              port: http
```

```
          readinessProbe:
            httpGet:
              path: /
              port: http
          resources:
            limits:
              cpu: 100m
              memory: 128Mi
            requests:
              cpu: 100m
              memory: 128Mi
---
# Source: pod-info/templates/tests/test-connection.yaml
apiVersion: v1
kind: Pod
metadata:
  name: "pod-info-dev-test-connection"
  labels:
    helm.sh/chart: pod-info-0.1.0
    app.kubernetes.io/name: pod-info
    app.kubernetes.io/instance: pod-info-dev
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
  annotations:
    "helm.sh/hook": test
spec:
  containers:
    - name: wget
      image: busybox
      command: ['wget']
      args: ['pod-info-dev:80']
  restartPolicy: Never
```

Deploying the Podinfo Helm Chart Manually on Kubernetes Cluster You can now deploy the Podinfo application with Helm chart manually on your Kubernetes cluster.

Deploy the Podinfo application simply like this:

Format:

```
$ helm install <chart_name> <dir_path> \
 --values <values_file_path> \
 --create-namespace \
 --namespace <namespace>
```

For example:

```
$ helm install pod-info-dev pod-info \
 --values pod-info/values.yaml \
 --create-namespace \
 --namespace dev
```

You have setup `NodePort` service type in the Podinfo Helm chart's service configuration. So, you can access the Podinfo application via Node `Port` from the outside of the Kubernetes cluster. Please, see the service configuration.

```
# Source: pod-info/templates/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: pod-info-dev
  labels:
    helm.sh/chart: pod-info-0.1.0
    app.kubernetes.io/name: pod-info
    app.kubernetes.io/instance: pod-info-dev
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: http
      protocol: TCP
      name: http
  selector:
    app.kubernetes.io/name: pod-info
    app.kubernetes.io/instance: pod-info-dev
```

To get the NodePort URL of the Podinfo application, run the following commands. (Replace with your Namespace and Pod name)

```
$ export NODE_PORT=$(kubectl get --namespace dev -o jsonpath="{.spec.ports[0]
$ export NODE_IP=$(kubectl get nodes --namespace dev -o jsonpath="{.items[0].s
$ echo http://$NODE_IP:$NODE_PORT
```

Then, you can access the Podinfo application with the following URL in your web browser. (Replace with your Kubernete Node IP address.)

```
http://<192.168.x.x>:32431
```

Figure 7: screenshot-podinfo-helm-demo

Now, you can see `Namespace`, `Node Name`, `Pod Name` and `Pod IP` information in the UI of the Podinfo application.