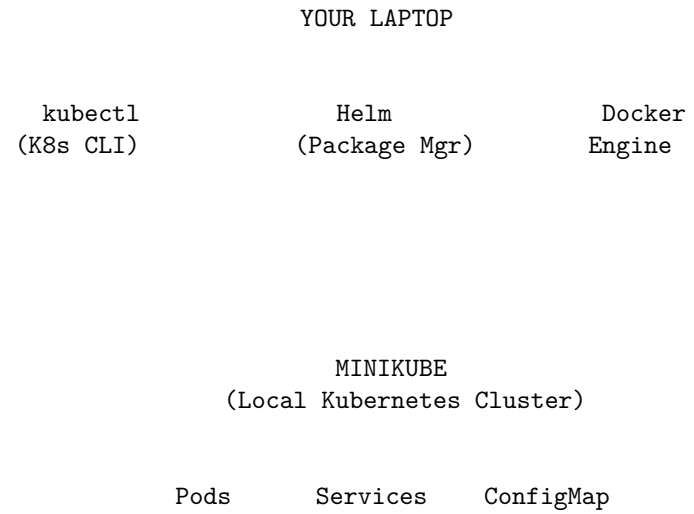
00 – Environment Setup (08:30–09:00)

Objective: Validate kubectl, Minikube, and Helm installations; confirm cluster is ready for labs.

Estimated duration: 20–30 minutes

Prerequisites: Internet access; Docker or other container runtime installed.

Environment Architecture



What You Will Learn

Before we can use Helm, we need three tools installed and working together:

1. **kubectl** – The command-line tool that talks to Kubernetes clusters. Think of it as the remote control for your cluster.
2. **Minikube** – A tool that creates a mini Kubernetes cluster on your laptop. It's perfect for learning because you don't need expensive cloud servers.
3. **Helm** – The package manager we'll use all day. It installs apps into Kubernetes using "charts" (pre-made templates).

Operating System Installation Guide

This section covers installation for **Ubuntu 24.04 LTS** (primary), **macOS**, and **Windows**.

Ubuntu 24.04 LTS Installation (PRIMARY)

Ubuntu 24.04 LTS (Noble Numbat) is our primary training platform. Follow these steps carefully.

Step 1: Install Docker on Ubuntu 24.04

Update your system first:

```
sudo apt update && sudo apt upgrade -y
```

Install required packages:

```
sudo apt install -y apt-transport-https ca-certificates curl software-properties-common gnupg lsb-release
```

Add Docker's official GPG key:

```
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gp
sudo chmod a+r /etc/apt/keyrings/docker.gpg
```

Add the Docker repository for Ubuntu 24.04:

```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.c
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Install Docker:

```
sudo apt update
sudo apt install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin
```

Add your user to the docker group (so you don't need sudo):

```
sudo usermod -aG docker $USER
```

Important: Log out and log back in for group changes to take effect!

```
# Or use this to apply changes in current session:
newgrp docker
```

Verify Docker installation:

```
docker --version
docker run hello-world
```

Expected output:

```
Docker version 24.0.x, build ...
Hello from Docker!
This message shows that your installation appears to be working correctly.
```

Step 2: Install kubectl on Ubuntu 24.04

Download the latest stable kubectl:

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kube
```

Validate the binary (optional but recommended):

```
curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kube
echo "$(cat kubectl.sha256)  kubectl" | sha256sum --check
```

Install kubectl:

```
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

Verify installation:

```
kubectl version --client
```

Expected output:

```
Client Version: v1.31.x
Kustomize Version: v5.x.x
```

Alternative: Using apt repository:

```
# Add Kubernetes apt repository
curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.31/deb/Release.key | sudo gpg --dearmor -o /etc/apt/keyring
echo 'deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.31/

# Install kubectl
sudo apt update
sudo apt install -y kubectl
```

Step 3: Install Minikube on Ubuntu 24.04

Download the latest Minikube:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
```

Install Minikube:

```
sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Verify installation:

```
minikube version
```

Expected output:

```
minikube version: v1.34.0
commit: ...
```

Start Minikube with Docker driver:

```
minikube start --driver=docker
```

Expected output:

```
  minikube v1.34.0 on Ubuntu 24.04
  Using the docker driver based on user configuration
  Starting control plane node minikube in cluster minikube
  Pulling base image ...
  Creating docker container (CPUs=2, Memory=3900MB) ...
  Preparing Kubernetes v1.31.0 on Docker 27.2.0 ...
     Generating certificates and keys ...
     Booting up control plane ...
     Configuring RBAC rules ...
  Verifying Kubernetes components...
     Using image gcr.io/k8s-minikube/storage-provisioner:v5
  Enabled addons: storage-provisioner, default-storageclass
  Done! kubectl is now configured to use "minikube" cluster
```

Set Docker as the default driver:

```
minikube config set driver docker
```

Step 4: Install Helm on Ubuntu 24.04

Method 1: Using the official install script (recommended):

```
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
```

Method 2: Using Snap:

```
sudo snap install helm --classic
```

Method 3: Using apt repository:

```
curl https://baltocdn.com/helm/signing.asc | gpg --dearmor | sudo tee /usr/share/keyrings/helm.gpg > /dev/n
sudo apt install apt-transport-https --yes
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/helm.gpg] https://baltocdn.com/h
sudo apt update
sudo apt install helm
```

Verify Helm installation:

```
helm version
```

Expected output:

```
version.BuildInfo{Version:"v3.16.x", GitCommit:"...", GitTreeState:"clean", GoVersion:"go1.22.x"}
```

---

macOS Installation

Install with Homebrew (Recommended)

Install Homebrew if not already installed:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

Install Docker Desktop:

Download from https://www.docker.com/products/docker-desktop/

Or use Homebrew:

```
brew install --cask docker
```

Then open Docker Desktop from Applications and wait for it to start.

Install kubectl:

```
brew install kubectl
```

Verify:

```
kubectl version --client
```

Install Minikube:

```
brew install minikube
```

Start Minikube:

```
minikube start --driver=docker
```

Install Helm:

```
brew install helm
```

Verify:

```
helm version --short
```

Windows Installation

Prerequisites

- Windows 10/11 (64-bit) with virtualization enabled in BIOS
- Administrator access

Install with Chocolatey (Recommended)

Install Chocolatey (open PowerShell as Administrator):

```
Set-ExecutionPolicy Bypass -Scope Process -Force; [System.Net.ServicePointManager]::SecurityProtocol = [Sys
```

Install Docker Desktop:

```
choco install docker-desktop -y
```

Restart your computer after installation, then open Docker Desktop.

Install kubectl:

```
choco install kubernetes-cli -y
```

Verify:

```
kubectl version --client
```

Install Minikube:

```
choco install minikube -y
```

Start Minikube:

```
minikube start --driver=docker
```

Install Helm:

```
choco install kubernetes-helm -y
```

Verify:

```
helm version --short
```

Alternative: Using winget (Windows 11)

```
winget install Docker.DockerDesktop
winget install Kubernetes.kubectl
winget install Kubernetes.minikube
winget install Helm.Helm
```

Universal Post-Installation Steps (All Operating Systems)

Verify Your Complete Setup

Run these commands on any OS to verify everything is working:

Check kubectl:

```
kubectl version --client
```

Check your contexts:

```
kubectl config get-contexts
```

What this shows: A "context" is a saved connection to a cluster. You might see nothing yet (that's okay—Minikube will create one), or you might see existing contexts from previous work.

Check Minikube status:

```
minikube status
```

Expected output:

```
minikube
type: Control Plane
host: Running
```

```
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Verify kubectl can talk to the cluster:

```
kubectl get nodes
```

Expected output:

```
NAME       STATUS   ROLES           AGE   VERSION
minikube   Ready    control-plane   2m    v1.31.0
```

Check Helm:

```
helm version --short
```

Expected output:

```
v3.16.x+g...
```

Explore Helm's help:

```
helm help
```

This shows all available commands. Don't worry about memorizing them—we'll learn the important ones throughout the day.

Step 4: Test DNS and Networking Inside the Cluster

Why test this? Kubernetes apps need to find each other by name (like "my-database" or "my-api"). This uses internal DNS. If DNS is broken, nothing works properly.

Run a quick DNS test:

```
kubectl run dns-test --image=busybox:1.36 --restart=Never -- nslookup kubernetes.default
```

Breaking down this command: - `kubectl run dns-test` = "Create a pod named dns-test" - `--image=busybox:1.36` = "Use the busybox image (a tiny Linux with basic tools)" - `--restart=Never` = "Don't restart if it exits (one-time test)" - `-- nslookup kubernetes.default` = "Run this command inside the pod"

Wait a few seconds, then check the result:

```
kubectl logs dns-test
```

Expected output:

```
Server:    10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:      kubernetes.default
Address 1: 10.96.0.1 kubernetes.default.svc.cluster.local
```

This means DNS is working! The pod successfully looked up "kubernetes.default".

Clean up the test pod:

```
kubectl delete pod dns-test
```

Expected Results Summary

After completing this setup, you should have:

Check	Command	Expected
kubectl installed	`kubectl version --client`	Shows version v1.2x+
Minikube running	`minikube status`	All items "Running"
Node ready	`kubectl get nodes`	One node, status "Ready"
Helm installed	`helm version --short`	Shows v3.x
DNS working	Logs from dns-test pod	Shows resolved addresses

Troubleshooting Common Problems

Problem: "minikube start" fails with Docker errors

Symptoms:

```
Exiting due to PROVIDER_DOCKER_NOT_RUNNING: docker is not running
```

Solution: 1. Open Docker Desktop application 2. Wait for it to say "Docker Desktop is running" 3. Try `minikube start --driver=docker` again

---

Problem: kubectl cannot connect to cluster

Symptoms:

```
The connection to the server localhost:8080 was refused
```

Solution: 1. Make sure Minikube is running: `minikube status` 2. If stopped, start it: `minikube start` 3. Set the context: `kubectl config use-context minikube`

Problem: "helm: command not found"

Solution: Helm isn't installed. Run the installer:

```
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
```

Then close and reopen your terminal, or run:

```
source ~/.bashrc    # or source ~/.zshrc on macOS
```

Problem: Minikube is very slow or runs out of memory

Solution: Give Minikube more resources:

```
minikube stop
minikube delete
minikube start --driver=docker --cpus=4 --memory=8192
```

This allocates 4 CPU cores and 8GB RAM to the cluster.

Additional Validation Commands

These commands help you verify everything is healthy:

Check Docker is available:

```
docker info | head -n 5
```

Should show Docker version and running status.

Check Minikube addons:

```
minikube addons list | head -n 10
```

Look for `default-storageclass` and `storage-provisioner` marked as "enabled".

Check all system pods are running:

```
kubectl get pod -A
```

All pods should show "Running" or "Completed" status.

Check Helm environment:

```
helm env | head -n 5
```

Shows where Helm stores its cache and configuration.

Setting Up Training Namespaces

To make this tutorial more realistic and organized, we will create several Namespaces. A Namespace is a logical unit which we use to organize different environments. Think of namespaces as separate folders on your computer—each one keeps things organized and isolated.

Why use Namespaces? - **Organization**: Keep dev, test, and prod resources separate - **Security**: Apply different permissions to different namespaces - **Resource Limits**: Set quotas per namespace - **Clarity**: Easily see what belongs to what environment

Create the training namespaces:

Create a file called `namespaces.yaml`:

```
cat << 'EOF' > namespaces.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: dev
  labels:
    environment: development
---
apiVersion: v1
kind: Namespace
metadata:
  name: test
  labels:
    environment: testing
---
apiVersion: v1
kind: Namespace
metadata:
  name: prod
  labels:
    environment: production
---
apiVersion: v1
kind: Namespace
metadata:
  name: tools
  labels:
```

```
    environment: tooling
EOF
```

Apply the namespaces:

```
kubectl apply -f namespaces.yaml
```

Expected output:

```
namespace/dev created
namespace/test created
namespace/prod created
namespace/tools created
```

Verify the namespaces were created:

```
kubectl get namespaces
```

Expected output:

```
NAME              STATUS   AGE
default           Active   10m
dev               Active   5s
kube-node-lease   Active   10m
kube-public       Active   10m
kube-system       Active   10m
prod              Active   5s
test              Active   5s
tools             Active   5s
```

Understanding the Namespace Structure:

```
                    KUBERNETES CLUSTER


       dev           test          prod         tools

      Your          Testing     Production      Helm
      apps          before         apps         repos,
      in            prod          (live)      monitoring
      dev



     kube-system         default       ← System namespaces
     (K8s core)        (your work)
```

Deploy to a specific namespace:

When deploying with Helm, you'll use the `--namespace` flag:

```
# Example: Deploy to dev namespace
helm install myapp bitnami/nginx --namespace dev

# Example: Deploy to tools namespace
helm install monitoring prometheus-community/prometheus --namespace tools
```

Installation Summary Table by OS

Tool	Ubuntu 24.04	macOS	Windows
Docker	`apt install docker-ce`	Docker Desktop / `brew install --cask docker`	`choco install docker-desktop`
kubectl	`curl -LO` + `install` or apt	`brew install kubectl`	`choco install kubernetes-cli`
Minikube	`curl -LO` + `install`	`brew install minikube`	`choco install minikube`
Helm	Script / snap / apt	`brew install helm`	`choco install kubernetes-helm`

Optional Challenge

If you finish early, try changing the Minikube driver:

```
minikube stop
minikube delete
minikube start --driver=hyperkit  # or --driver=virtualbox
```

Then run `minikube status` to see how the output differs. This teaches you that Minikube can use different virtualization backends.

Quick Reference: Essential Commands

```
# Start your environment
minikube start --driver=docker

# Check cluster status
minikube status
kubectl get nodes
kubectl get namespaces

# Stop your environment (saves resources)
minikube stop

# Delete and start fresh
minikube delete
minikube start --driver=docker

# Check tool versions
kubectl version --client
helm version --short
minikube version
```

Environment setup complete! You're ready for Module 1. # 01 – Helm Fundamentals (Theory) 09:00–10:00

Learning objectives: - Define Helm, chart, release, repository in plain terms - Recognize chart folder structure at a high level - Understand how Helm compares to package managers you already know

---

What is Helm? (The Big Picture)

Imagine you want to install a web server on your laptop. On macOS, you might type `brew install nginx`. On Ubuntu, you'd use `apt install nginx`. These package managers handle downloading the software, putting files in the right places, and configuring defaults.

Helm does the same thing, but for Kubernetes.

Without Helm, installing an application in Kubernetes means writing many YAML files by hand: - A Deployment file (tells Kubernetes how to run your app) - A Service file (lets other apps or users reach your app) - A ConfigMap file (configuration settings) - A Secret file (passwords, API keys) - Maybe an Ingress file (expose to the internet)

For a simple app, that's 5+ files with 200+ lines of YAML. For a database like PostgreSQL, it could be 500+ lines. Copying and editing all that by hand is slow and error-prone.

Helm solves this with "charts"—pre-made packages that bundle all those files together.

Key Terms Explained Simply

Chart

A **chart** is a folder containing templates and default values. It's like an installer package.

Real-world analogy: A chart is like a recipe. It has ingredients (values) and instructions (templates). You can follow the recipe exactly, or adjust the ingredients to your taste.

Example: The `bitnami/nginx` chart contains everything needed to run nginx in Kubernetes—Deployment, Service, ConfigMap, and more. Someone else wrote it; you just install it.

Release

A **release** is what you get when you install a chart into your cluster. It's a running instance with a name you choose.

Real-world analogy: If a chart is a recipe, a release is the actual cake you baked. You can bake the same recipe multiple times and give each cake a different name.

Example: You install `bitnami/nginx` twice: - First release named `web-frontend` (serves your website) - Second release named `api-docs` (serves documentation)

Same chart, two separate releases, running independently.

Repository

A **repository** (or "repo") is a server that stores charts. You add repos to Helm, then search and install charts from them.

Real-world analogy: A repo is like an app store. Apple's App Store is a repo. Google Play is another repo. Each has different apps (charts) available.

Example repos: - `https://charts.bitnami.com/bitnami` – Thousands of production-ready charts - `https://prometheus-community.github.io/helm-charts` – Monitoring tools - Your company might have a private repo for internal apps

---

Values

Values are the settings you can customize when installing a chart. They're stored in a file called `values.yaml`.

Real-world analogy: Values are like the options when ordering a pizza—size, toppings, crust type. The pizza shop (chart) has defaults, but you can customize.

Example values in an nginx chart:

```yaml
replicaCount: 2          # Run 2 copies for reliability
image:
  repository: nginx      # Which container image to use
  tag: "1.25"            # Which version
service:
  type: NodePort         # How to expose the service
  port: 80               # Which port to listen on
```

How Helm Works (Step by Step)

Here's what happens when you run `helm install my-nginx bitnami/nginx`:

```
Step 1: Helm downloads the chart from the repository
        (bitnami/nginx → your local cache)

                              ↓

Step 2: Helm reads values.yaml (defaults) and any overrides
        you provided (like --set replicaCount=3)

                              ↓

Step 3: Helm "renders" templates-fills in placeholders with
        actual values, producing final YAML files

                              ↓

Step 4: Helm sends those YAML files to Kubernetes
        (just like running kubectl apply)

                              ↓

Step 5: Kubernetes creates the resources (pods, services, etc.)
        Helm saves a record called a "release"
```

What's Inside a Chart Folder?

When you create or download a chart, you get a folder like this:

```
my-chart/
   Chart.yaml          # Metadata: name, version, description
   values.yaml         # Default configuration values
   templates/          # YAML templates with placeholders
      deployment.yaml
      service.yaml
      configmap.yaml
      _helpers.tpl    # Reusable template snippets
      NOTES.txt       # Message shown after install
   charts/             # Dependencies (other charts this one needs)
```

Let's look at each file:

Chart.yaml

Contains metadata about the chart:

```yaml
apiVersion: v2
name: my-chart
description: A simple web application
version: 1.0.0        # Chart version (your packaging version)
appVersion: "1.25"    # App version (the software inside)
```

values.yaml

Default values that users can override:

```yaml
replicaCount: 1

image:
  repository: nginx
  tag: "1.25"
  pullPolicy: IfNotPresent

service:
  type: ClusterIP
  port: 80

resources:
  limits:
    cpu: 100m
    memory: 128Mi
```

templates/deployment.yaml (simplified example)

A template with placeholders (the `{{ }}` parts):

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-app
spec:
  replicas: {{ .Values.replicaCount }}
  template:
    spec:
      containers:
        - name: app
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
```

When Helm renders this with `replicaCount: 2` and `image.tag: "1.25"`, it becomes:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx-app
spec:
  replicas: 2
  template:
    spec:
      containers:
        - name: app
          image: "nginx:1.25"
```

Why Use Helm Instead of Plain YAML?

Without Helm	With Helm
Copy-paste YAML between projects	Reuse charts from repos
Edit 10 files to change one setting	Change one value, re-render
No history of what you deployed	Helm tracks every release and revision
Deleting means finding all related resources	`helm uninstall` removes everything
Upgrades are manual and risky	`helm upgrade` handles it; rollback if needed

---

Essential Commands Overview

Command	What it does	Example
`helm help`	Shows all commands	`helm help`
`helm version`	Shows Helm version	`helm version --short`
`helm search hub <term>`	Searches Artifact Hub (public)	`helm search hub nginx`
`helm search repo <term>`	Searches your added repos	`helm search repo nginx`
`helm show readme <chart>`	Displays chart documentation	`helm show readme bitnami/nginx`
`helm show values <chart>`	Shows default values	`helm show values bitnami/nginx`

Walkthrough Example: Exploring an nginx Chart

Let's walk through discovering and inspecting a chart:

1. Search for nginx charts on the public hub:

```
helm search hub nginx
```

This queries Artifact Hub and returns dozens of nginx charts from different publishers.

2. Add a trusted repository:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update
```

Now you can search within Bitnami's catalog.

3. Search the local repo:

```
helm search repo nginx
```

Output:

```
NAME                    CHART VERSION   APP VERSION   DESCRIPTION
bitnami/nginx           15.0.0          1.25.0        NGINX is a high-performance...
bitnami/nginx-ingress   9.0.0           1.8.0         NGINX Ingress Controller...
```

4. Read the chart's README:

```
helm show readme bitnami/nginx | head -n 50
```

This shows installation instructions, configuration options, and examples.

5. View default values:

```
helm show values bitnami/nginx | head -n 30
```

This shows what you can customize when installing.

Review Questions

Test your understanding:

1. **What is the difference between a chart and a release?**
   - A chart is the package/template; a release is an installed instance of that chart with a specific name.
2. **Where do charts come from?**
   - Charts are stored in repositories. You add repos with `helm repo add`, then search and install from them.
3. **Why use Helm instead of writing YAML by hand?**
   - Reusability, easier configuration, version tracking, simple upgrades and rollbacks, one-command uninstall.
4. **What file contains the default configuration for a chart?**
   - `values.yaml`
5. **What command shows what a chart would install without actually installing it?**
   - `helm template <chart>` or `helm install <release> <chart> --dry-run`

You now understand the core concepts of Helm. Next, we'll actually install and use it! # 01 – Helm Fundamentals (Lab)

Objective: Explore Helm commands hands-on, search for charts, and inspect chart contents without installing anything.

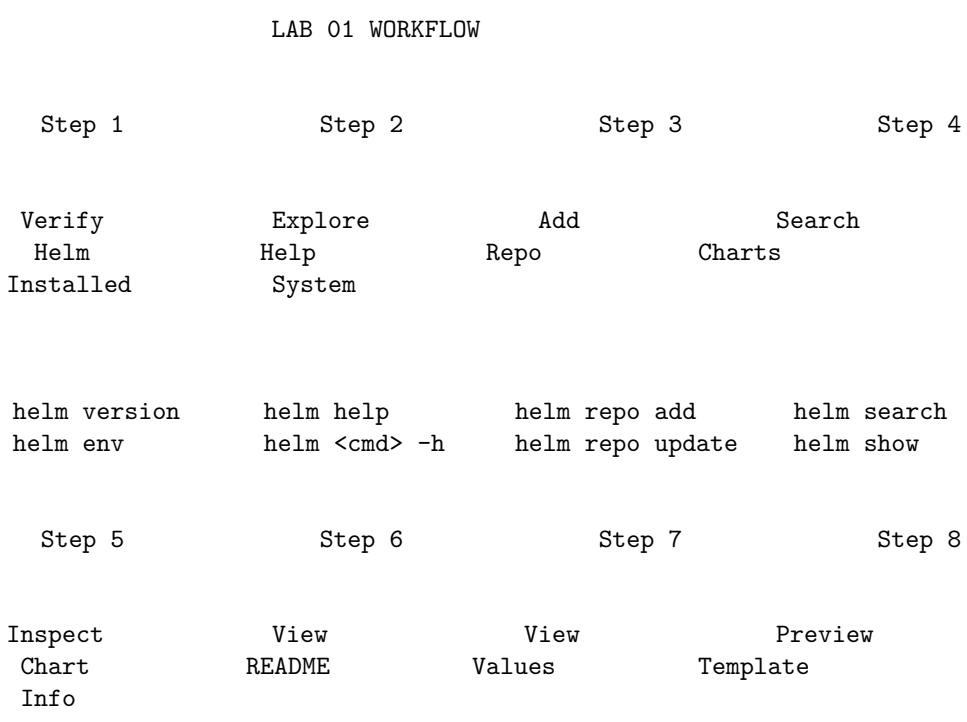
Estimated duration: 20–25 minutes

Prerequisites: Completed environment setup; Minikube running; Helm installed.

What You Will Practice

In this lab, you will: - Run basic Helm commands to understand the tool - Search for charts in public repositories - Inspect a chart's documentation and default values - Preview what a chart would create (without actually installing)

Lab Workflow Overview



```
helm show chart   helm show readme   helm show values   helm template
```

Step-by-Step Instructions

Step 1: Verify Helm is Working

First, let's confirm Helm is installed and see what version we have.

Run this command:

```
helm version
```

What you should see:

```
version.BuildInfo{Version:"v3.13.0", GitCommit:"...", GitTreeState:"clean", GoVersion:"go1.21.0"}
```

What this means: - `Version:"v3.13.0"` – You have Helm 3 (the current major version). Any v3.x is good. - The other fields show build details that you can ignore.

For a shorter output:

```
helm version --short
```

Output: `v3.13.0+g...`

Step 2: Explore Helm's Help System

Helm has built-in documentation for every command.

```
                    HELM HELP HIERARCHY


                       helm help



      helm install      helm repo       helm create
        --help           --help           --help


    [detailed usage]  [repo subcommands]  [create usage]
                          helm repo add
                          helm repo list
                          helm repo update
                          helm repo remove
```

See all available commands:

```
helm help
```

What you'll see: A list of command groups: - `completion` – Shell auto-completion - `create` – Create a new chart - `install` – Install a chart - `list` – List releases - `repo` – Manage repositories - …and many more

Get help for a specific command:

```
helm help install
```

What you'll see: Detailed documentation for the `install` command, including all flags and examples.

Why this matters: You don't need to memorize every flag. Use `helm help <command>` whenever you forget.

Step 3: Search for Charts on Artifact Hub

Artifact Hub is a public website that indexes thousands of Helm charts.

Search for nginx charts:

```
helm search hub nginx
```

What you should see:

```
URL                                                 CHART VERSION   APP VERSION   DESCRIPTION
https://artifacthub.io/packages/helm/bitnami/nginx  15.0.0          1.25.0        NGINX Open Source is a.
https://artifacthub.io/packages/helm/nginx/nginx    0.14.0          1.5.0         An NGINX HTTP server...
... (many more results)
```

Understanding the output: - **URL** – Link to the chart's page on Artifact Hub - **CHART VERSION** – Version of the Helm chart (packaging version) - **APP VERSION** – Version of the actual software (nginx itself) - **DESCRIPTION** – Brief description

Note: This searches the public hub, not charts you've downloaded locally.

Step 4: Add a Repository

To install charts, you first add their repository to Helm.

Add the Bitnami repository:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

What this command does: - `helm repo add` – "I want to add a new chart source" - `bitnami` – Local nickname for this repo (you can call it anything) - `https://charts.bitnami.com/bitnami` – URL where charts are hosted

Expected output:

```
"bitnami" has been added to your repositories
```

Update the repository index:

```
helm repo update
```

What this does: Downloads the latest list of available charts from all your repos. Like refreshing an app store.

Expected output:

```
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "bitnami" chart repository
Update Complete. Happy Helming!
```

Step 5: Search Your Added Repositories

Now search within the repos you've added (not the public hub).

Search for nginx in your repos:

```
helm search repo nginx
```

What you should see:

```
NAME                    CHART VERSION   APP VERSION   DESCRIPTION
bitnami/nginx           15.0.0          1.25.0        NGINX Open Source is a web server...
bitnami/nginx-ingress   9.0.0           1.8.0         NGINX Ingress Controller is...
```

Difference from `helm search hub`: - `helm search hub` – Searches the entire public hub (internet) - `helm search repo` – Searches only repos you've added locally (faster, works offline after update)

Step 6: Inspect a Chart's README

Before installing a chart, you should read its documentation.

Show the README for bitnami/nginx:

```
helm show readme bitnami/nginx
```

What you'll see: The full README file, which typically includes: - Description of what the chart installs - Prerequisites - Installation commands - Configuration options - Troubleshooting tips

To see just the first part:

```
helm show readme bitnami/nginx | head -n 50
```

This shows the first 50 lines, which usually covers the introduction.

Step 7: View a Chart's Default Values

Every chart has a `values.yaml` with default settings. You can customize these.

Show default values:

```
helm show values bitnami/nginx | head -n 40
```

What you'll see:

```yaml
## @section Global parameters
global:
  imageRegistry: ""
  imagePullSecrets: []
  storageClass: ""

## @section Common parameters
nameOverride: ""
fullnameOverride: ""

## @section NGINX parameters
image:
  registry: docker.io
  repository: bitnami/nginx
  tag: 1.25.0-debian-11-r0
...
```

Understanding this: - Each line is a setting you can override - Comments (lines starting with `#` or `##`) explain each setting - Nested values use indentation (like `image.tag`)

Step 8: View Chart Metadata

Show chart metadata:

```
helm show chart bitnami/nginx
```

What you'll see:

```
annotations:
  category: Infrastructure
apiVersion: v2
appVersion: 1.25.0
description: NGINX Open Source is a web server...
home: https://nginx.org
icon: https://...
keywords:
- nginx
- http
- web
maintainers:
- name: Bitnami
name: nginx
version: 15.0.0
```

Key fields: - `name` – Chart name - `version` – Chart version (the package) - `appVersion` – Version of the software inside (nginx 1.25.0) - `description` – What it does - `maintainers` – Who maintains it

---

Step 9: Preview What Would Be Installed (Dry Run)

You can see exactly what Kubernetes resources a chart would create without actually installing.

Render templates locally:

```
helm template my-nginx bitnami/nginx | head -n 60
```

What this command does: - `helm template` – "Show me the rendered YAML" - `my-nginx` – The release name (what you'd call this installation) - `bitnami/nginx` – The chart to render

What you'll see: Actual Kubernetes YAML files with all placeholders filled in:

```
---
# Source: nginx/templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
  labels:
    app.kubernetes.io/name: nginx
    ...
```

Why this is useful: - You can review exactly what will be created - Catch configuration mistakes before installing - Save the output to a file for auditing: `helm template my-nginx bitnami/nginx > rendered.yaml`

Step 10: Check Your Repository List

List all configured repositories:

```
helm repo list
```

Expected output:

```
NAME      URL
bitnami   https://charts.bitnami.com/bitnami
```

Check Helm environment:

```
helm env | head -n 5
```

What you'll see:

```
HELM_CACHE_HOME="/Users/you/.cache/helm"
HELM_CONFIG_HOME="/Users/you/.config/helm"
HELM_DATA_HOME="/Users/you/.local/share/helm"
...
```

These are the directories where Helm stores cached charts and configuration.

Expected Results Summary

After this lab, you should have:

Task	Command	Result
Helm version confirmed	`helm version --short`	v3.x displayed
Hub search works	`helm search hub nginx`	Multiple results
Repo added	`helm repo list`	Shows "bitnami"
Can view README	`helm show readme bitnami/nginx`	Documentation displayed
Can view values	`helm show values bitnami/nginx`	YAML values displayed
Can render templates	`helm template my-nginx bitnami/nginx`	Kubernetes YAML displayed

Troubleshooting

Problem: "helm search hub" returns nothing

Possible causes: - No internet connection - Artifact Hub might be temporarily unavailable

Solution: Try again in a minute, or use `helm search repo` after adding a repo.

Problem: "Error: repo bitnami not found"

Cause: You haven't added the bitnami repo yet.

Solution:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo update
```

Problem: "helm show values" shows nothing useful

Cause: Chart name might be wrong.

Solution: First search for the correct name:

```
helm search repo nginx
```

Use the exact name from the NAME column.

Optional Challenges

If you finish early, try these:

1. **Find a PostgreSQL chart:**

   ```
   helm search hub postgresql
   helm search repo postgresql  # after adding bitnami repo
   ```

2. **Compare two charts:**

   ```
   helm show chart bitnami/nginx
   helm show chart bitnami/postgresql
   ```

   Notice the different app versions and descriptions.

3. **Save rendered output to a file:**

   ```
   helm template my-app bitnami/nginx > my-app-rendered.yaml
   cat my-app-rendered.yaml | head -n 100
   ```

4. **Render with custom values:**

   ```
   helm template my-nginx bitnami/nginx --set replicaCount=3 | grep "replicas:"
   ```

   You should see `replicas: 3` in the output.

Key Takeaways

- `helm search hub` finds charts on the public internet
- `helm search repo` finds charts in repos you've added locally
- `helm show readme/values/chart` lets you inspect before installing
- `helm template` renders YAML so you can preview what will be created
- Always read the README and check values before installing a new chart

---

Lab complete! You've explored Helm without installing anything. Next, we'll actually install a chart! # 02 – Installing & Using Helm (Theory) 10:00–11:00

Learning objectives: - Understand how repositories work and how to manage them - Learn the difference between searching, pulling, and installing - Install your first Helm release and understand what happens

---

The Repository System Explained

What is a Repository?

A Helm repository is simply a web server that hosts chart packages and an index file. Think of it like this:

- **Apple App Store** = A repository of iOS apps
- **npm** = A repository of JavaScript packages
- **Bitnami Helm Repo** = A repository of Kubernetes charts

When you add a repository, Helm downloads an **index file** that lists all available charts, their versions, and where to download them.

Repository Lifecycle

1. ADD REPO
   helm repo add bitnami https://charts.bitnami.com/bitnami
   → Helm saves the URL and downloads the index

```
                              ↓

2. UPDATE REPO
   helm repo update
   → Helm re-downloads the index to get latest chart versions

                              ↓

3. SEARCH REPO
   helm search repo nginx
   → Helm searches the local index (fast, works offline)

                              ↓

4. INSTALL FROM REPO
   helm install my-nginx bitnami/nginx
   → Helm downloads chart, renders templates, applies to cluster
```

Search Commands Explained

Helm has two search commands that work differently:

`helm search hub <term>`

What it does: Searches Artifact Hub, a public website indexing charts from many sources.

When to use: When you're looking for a chart and don't know which repo has it.

Example:

```
helm search hub redis
```

Pros: Finds charts from everywhere **Cons:** Requires internet; you still need to add the repo before installing

---

`helm search repo <term>`

What it does: Searches the index files of repositories you've already added.

When to use: When you've added repos and want to find specific charts.

Example:

```
helm search repo redis
```

Pros: Fast; works offline (after initial repo add/update) **Cons:** Only searches repos you've added

---

Installing a Chart: What Actually Happens

When you run `helm install`, Helm performs several steps:

Example Command:

```
helm install my-nginx bitnami/nginx
```

Step-by-Step Breakdown:

Step 1: Parse the command - `my-nginx` = The release name (your choice, must be unique in the namespace) - `bitnami/nginx` = The chart reference (repo-name/chart-name)

Step 2: Download the chart Helm downloads the chart archive from the repository to your local cache.

Step 3: Read values Helm reads `values.yaml` from the chart. If you provided overrides (`--set` or `-f`), those are merged on top.

Step 4: Render templates Helm processes each file in `templates/`, replacing placeholders like `{{ .Values.replicaCount }}` with actual values.

Step 5: Validate the YAML Helm checks that the rendered YAML is valid Kubernetes syntax.

Step 6: Send to Kubernetes Helm sends the rendered manifests to the Kubernetes API server (like `kubectl apply`).

Step 7: Create release record Helm stores a record of this installation as a Kubernetes Secret in the namespace. This enables upgrades and rollbacks later.

Understanding Release Names

The release name is important:

```
helm install my-nginx bitnami/nginx
#            ^^^^^^^^
#            This is your release name
```

Rules for release names:

- Must be unique within a namespace
- Use lowercase letters, numbers, and hyphens
- Maximum 53 characters
- Should be descriptive (e.g., `frontend-prod`, `api-staging`)

Why release names matter:

- All resources created will include this name
- You use it for upgrades: `helm upgrade my-nginx bitnami/nginx`
- You use it to check status: `helm status my-nginx`
- You use it to uninstall: `helm uninstall my-nginx`

Example of multiple releases:

```
# Same chart, different releases for different purposes
helm install web-prod bitnami/nginx -n production
helm install web-staging bitnami/nginx -n staging
helm install docs-site bitnami/nginx -n documentation
```

The `helm pull` Command: Downloading Without Installing

Sometimes you want to download a chart without installing it:

```
helm pull bitnami/nginx --untar -d /tmp/nginx-chart
```

Breaking down this command:

- `helm pull` = Download the chart archive
- `bitnami/nginx` = Which chart to download
- `--untar` = Extract the archive (otherwise you get a .tgz file)
- `-d /tmp/nginx-chart` = Where to put it

Why pull instead of install?

1. **Inspect the templates** before installing
2. **Modify the chart** for your specific needs
3. **Store charts** in your own version control
4. **Work offline** after pulling

What you get:

```
/tmp/nginx-chart/nginx/
  Chart.yaml
  values.yaml
  templates/
      deployment.yaml
      service.yaml
      ...
  README.md
```

Preview Before Installing: Dry Run

Never install blind! Always preview first:

Option 1: `helm template` (render locally)

```
helm template my-nginx bitnami/nginx
```

Shows rendered YAML without contacting the cluster.

Option 2: `helm install --dry-run` (server-side validation)

```
helm install my-nginx bitnami/nginx --dry-run --debug
```

Renders templates AND validates against the Kubernetes API (catches more errors).

What `--debug` adds:

- Shows the computed values
- Shows hook manifests
- More detailed output

Listing and Checking Releases

After installing, you'll want to check your releases:

List all releases in current namespace:

```
helm list
```

Output:

```
NAME      NAMESPACE   REVISION   UPDATED                   STATUS     CHART          APP VERSION
my-nginx  default     1          2024-01-15 10:30:00       deployed   nginx-15.0.0   1.25.0
```

List releases in all namespaces:

```
helm list --all-namespaces
# or shorter:
helm list -A
```

Check status of a specific release:

```
helm status my-nginx
```

Output shows: - Deployment status (deployed, failed, pending) - Last deployed time - Namespace - Notes from the chart (often includes access instructions)

Uninstalling: Cleaning Up

When you're done with a release:

```
helm uninstall my-nginx
```

What this does:

1. Finds all Kubernetes resources created by this release
2. Deletes them (Deployments, Services, ConfigMaps, etc.)
3. Removes the release record

What it does NOT delete:

- PersistentVolumeClaims (by default, to protect data)
- Resources created outside of Helm

Keep history after uninstall:

```
helm uninstall my-nginx --keep-history
```

This lets you see the old release with `helm history my-nginx` and even rollback to it.

Common Flags for `helm install`

Flag	Purpose	Example
`--set key=value`	Override a single value	`--set replicaCount=3`
`-f values.yaml`	Use a custom values file	`-f my-values.yaml`
`-n namespace`	Install in specific namespace	`-n production`
`--create-namespace`	Create namespace if missing	`-n new-ns --create-namespace`
`--dry-run`	Preview without installing	`--dry-run --debug`
`--wait`	Wait until pods are ready	`--wait --timeout 5m`
`--atomic`	Rollback if install fails	`--atomic`

Practical Example: Full Workflow

Let's walk through a complete example:

```
# 1. Add repository
helm repo add bitnami https://charts.bitnami.com/bitnami
```

```
# 2. Update to get latest charts
helm repo update

# 3. Search for what we want
helm search repo nginx

# 4. Check the chart's documentation
helm show readme bitnami/nginx | head -n 30

# 5. Check default values
helm show values bitnami/nginx | head -n 20

# 6. Preview what will be installed
helm install demo-nginx bitnami/nginx --dry-run --debug | head -n 100

# 7. Actually install it
helm install demo-nginx bitnami/nginx

# 8. Verify installation
helm list
helm status demo-nginx

# 9. Check Kubernetes resources
kubectl get pods,svc

# 10. Clean up when done
helm uninstall demo-nginx
```

Beginner Tips

1. **Always run `helm repo update` before installing** – You want the latest chart versions.
2. **Use `--dry-run --debug` liberally** – Preview before you commit.
3. **Choose meaningful release names** – `test1` and `test2` become confusing; use `nginx-frontend-dev` instead.
4. **Check the README first** – Charts often have required values or prerequisites.
5. **Start with defaults** – Install with no overrides first, then customize in subsequent upgrades.

Review Questions

1. **What's the difference between `helm search hub` and `helm search repo`?**
   - `hub` searches the internet; `repo` searches locally added repositories.
2. **What does `helm repo update` do?**
   - Re-downloads the index file from each repo to get the latest chart list.
3. **Why would you use `helm pull` instead of `helm install`?**
   - To download and inspect a chart without installing it, or to modify it.
4. **What information does `helm status <release>` show?**
   - Deployment status, last update time, namespace, and chart notes.
5. **How do you preview an installation without actually creating resources?**
   - Use `--dry-run --debug` flag with `helm install`.

You now understand how Helm repositories and installations work. Let's practice in the lab! # 02 – Installing & Using Helm (Lab)

Objective: Add a repository, search for charts, install your first release, verify it works, and uninstall cleanly.

Estimated duration: 25–30 minutes

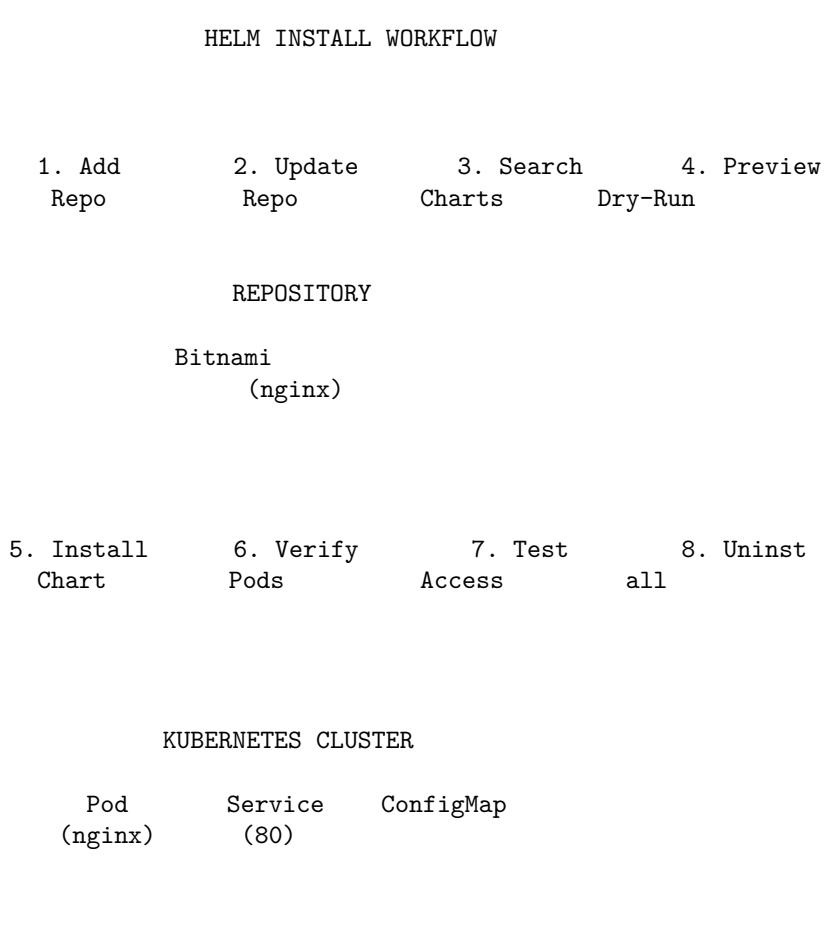
Prerequisites: Environment setup complete; Minikube running; Helm installed.

What You Will Do

This is your first hands-on experience installing something with Helm. You will: 1. Add a chart repository 2. Search for an nginx chart 3. Preview what it would install 4. Actually install it 5. Verify the pods and services are running 6. Access the running nginx 7. Uninstall and clean up

---

Lab Workflow Diagram



Step-by-Step Instructions

Step 1: Add the Bitnami Repository

Bitnami maintains high-quality, production-ready Helm charts. Let's add their repository.

Run this command:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

Breaking down the command: - `helm repo add` = "Add a new chart repository" - `bitnami` = The nickname you'll use to reference this repo - `https://charts.bitnami.com/bitnami` = The URL where charts are hosted

Expected output:

```
"bitnami" has been added to your repositories
```

What just happened: Helm saved the URL and downloaded an index file listing all available charts.

Step 2: Update the Repository Index

Always update after adding a repo to ensure you have the latest chart versions.

Run this command:

```
helm repo update
```

Expected output:

```
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "bitnami" chart repository
Update Complete. Happy Helming!
```

What this does: Downloads the latest index file, which lists all charts and their versions. Run this periodically to get new chart versions.

Step 3: Search for nginx Charts

Now let's find an nginx chart.

Run this command:

```
helm search repo nginx
```

Expected output:

```
NAME                    CHART VERSION   APP VERSION   DESCRIPTION
bitnami/nginx           15.0.0          1.25.0        NGINX Open Source is a web server...
bitnami/nginx-ingress   9.0.0           1.8.0         NGINX Ingress Controller is...
```

Understanding the columns: - **NAME**: Full chart reference (repo/chart) - you'll use this to install - **CHART VERSION**: Version of the Helm chart package - **APP VERSION**: Version of nginx inside the chart - **DESCRIPTION**: What the chart does

---

Step 4: (Optional) Pull the Chart Locally

Before installing, let's download the chart to inspect it.

Run these commands:

```
helm pull bitnami/nginx --untar -d /tmp/helm-nginx
ls /tmp/helm-nginx/nginx
```

Expected output:

```
Chart.yaml  README.md  charts  templates  values.yaml  values.schema.json
```

What you downloaded: - `Chart.yaml` – Metadata (name, version) - `values.yaml` – Default configuration - `templates/` – YAML templates that become Kubernetes resources - `README.md` – Documentation

Inspect the default values (first 20 lines):

```
head -n 20 /tmp/helm-nginx/nginx/values.yaml
```

This shows you what settings you can customize.

Step 5: Preview the Installation (Dry Run)

Before installing, preview what Kubernetes resources will be created.

Run this command:

```
helm install lab-nginx bitnami/nginx --dry-run --debug 2>&1 | head -n 80
```

Breaking down the command: - `helm install` = "Install a chart" - `lab-nginx` = Your chosen release name - `bitnami/nginx` = The chart to install - `--dry-run` = Don't actually install, just show what would happen - `--debug` = Show extra details (computed values, hooks) - `2>&1 | head -n 80` = Show first 80 lines of output

What you'll see: - The computed values (defaults merged with any overrides) - The rendered Kubernetes YAML (Deployment, Service, etc.) - Hook information

Why this matters: You can catch configuration mistakes BEFORE they affect your cluster.

Step 6: Install the Chart

Now let's actually install nginx into your cluster.

Run this command:

```
helm install lab-nginx bitnami/nginx
```

Expected output:

```
NAME: lab-nginx
LAST DEPLOYED: Thu Jan 15 10:30:00 2024
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
CHART NAME: nginx
CHART VERSION: 15.0.0
APP VERSION: 1.25.0

** Please be patient while the chart is being deployed **

NGINX can be accessed through the following DNS name from within your cluster:

    lab-nginx.default.svc.cluster.local (port 80)

To access NGINX from outside the cluster, follow the steps below:

1. Get the NGINX URL by running these commands:
...
```

What just happened: 1. Helm downloaded the chart (if not cached) 2. Rendered templates with default values 3. Sent the YAML to Kubernetes 4. Kubernetes created the Deployment, Service, etc. 5. Helm recorded this as "revision 1" of release "lab-nginx"

Step 7: Verify the Release is Listed

Run this command:

```
helm list
```

Expected output:

```
NAME       NAMESPACE   REVISION   UPDATED                   STATUS     CHART          APP VERSION
lab-nginx  default     1          2024-01-15 10:30:00       deployed   nginx-15.0.0   1.25.0
```

Understanding the columns: - **NAME**: Your release name - **NAMESPACE**: Where it's installed - **REVISION**: How many times it's been installed/upgraded (1 = first install) - **STATUS**: Current state (deployed, failed, pending-install, etc.) - **CHART**: Which chart and version - **APP VERSION**: The application version (nginx 1.25.0)

Step 8: Check the Release Status

Run this command:

```
helm status lab-nginx
```

Expected output: Same as the install output, plus updated status information.

When to use this: - Check if a release is healthy - Re-read the NOTES (often contain access instructions) - See when it was last updated

---

Step 9: Verify Kubernetes Resources

Let's check what Kubernetes resources were created.

Check pods:

```
kubectl get pods
```

Expected output:

```
NAME                          READY   STATUS    RESTARTS   AGE
lab-nginx-7d4f8b7b9c-xk2j4    1/1     Running   0          2m
```

What this shows: - One pod is running (STATUS: Running) - 1/1 containers are ready - Name includes your release name (`lab-nginx-`)

Check services:

```
kubectl get svc
```

Expected output:

```
NAME         TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP   1h
lab-nginx    ClusterIP   10.96.123.45    <none>        80/TCP    2m
```

What this shows: - A Service named `lab-nginx` was created - Type is `ClusterIP` (internal only by default) - Listening on port 80

Get more details about the service:

```
kubectl describe svc lab-nginx
```

This shows: - Full configuration - Endpoints (which pod IPs receive traffic) - Labels and selectors

---

Step 10: Access the Running nginx

Since the service type is ClusterIP (internal), we need port-forwarding to access it from our laptop.

```
                    PORT FORWARDING


  YOUR LAPTOP                          KUBERNETES CLUSTER
```

Run this command:

```
kubectl port-forward svc/lab-nginx 8080:80
```

Breaking down the command: - `kubectl port-forward` = Forward a local port to a cluster resource - `svc/lab-nginx` = The service to forward to - `8080:80` = Local port 8080 → Service port 80

Expected output:

```
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

Now open a new terminal (leave port-forward running) **and test:**

```
curl http://localhost:8080
```

Expected output:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

You're seeing the nginx welcome page served from your Kubernetes cluster!

Stop port-forward: Press `Ctrl+C` in the terminal running port-forward.

Step 11: See What Was Installed (Manifests)

View the actual Kubernetes YAML that was applied:

```
helm get manifest lab-nginx | head -n 50
```

This shows: The exact YAML that Helm sent to Kubernetes. Useful for debugging or auditing.

View the values that were used:

```
helm get values lab-nginx
```

Expected output:

```
USER-SUPPLIED VALUES:
null
```

Since we didn't override any values, it shows null. The chart used all defaults.

Step 12: Uninstall the Release

When you're done, clean up the installation.

Run this command:

```
helm uninstall lab-nginx
```

Expected output:

```
release "lab-nginx" uninstalled
```

What this does: 1. Finds all Kubernetes resources created by this release 2. Deletes them (Deployment, Service, etc.) 3. Removes the release record from Helm's history

Verify it's gone:

```
helm list
kubectl get pods
kubectl get svc
```

The `lab-nginx` release, pod, and service should no longer appear.

Expected Results Summary

Step	Command	Expected Result
Add repo	`helm repo add bitnami ...`	"bitnami" has been added
Update repo	`helm repo update`	Update Complete
Search	`helm search repo nginx`	Shows bitnami/nginx
Install	`helm install lab-nginx bitnami/nginx`	STATUS: deployed
List	`helm list`	Shows lab-nginx, revision 1
Check pods	`kubectl get pods`	1 pod Running
Port-forward	`curl localhost:8080`	nginx welcome page
Uninstall	`helm uninstall lab-nginx`	release uninstalled

Troubleshooting

Problem: Pod is stuck in "Pending" state

Check what's wrong:

```
kubectl describe pod lab-nginx-xxxxx
```

Common causes: - Not enough cluster resources (try `minikube stop && minikube start --cpus=4 --memory=8192`) - Image pull issues (check your internet connection)

Problem: "port is already in use" during port-forward

Solution: Use a different local port:

```
kubectl port-forward svc/lab-nginx 9090:80
# Then access http://localhost:9090
```

Problem: Install fails with "INSTALLATION FAILED"

Get more details:

```
helm install lab-nginx bitnami/nginx --debug
```

Common causes: - Chart not found (did you `helm repo add` and `helm repo update`?) - Release name already exists (uninstall first or choose different name)

Optional Challenges

Challenge 1: Install with Custom Service Type

Install nginx with NodePort so Minikube can provide a URL:

```
helm install nginx-nodeport bitnami/nginx --set service.type=NodePort
```

Then get the URL:

```
minikube service nginx-nodeport --url
```

Open that URL in your browser!

Don't forget to uninstall when done:

```
helm uninstall nginx-nodeport
```

Challenge 2: Install with Multiple Replicas

```
helm install nginx-ha bitnami/nginx --set replicaCount=3
kubectl get pods
```

You should see 3 pods running. This demonstrates high availability.

Uninstall:

```
helm uninstall nginx-ha
```

Challenge 3: Inspect the Computed Values

```
helm install nginx-debug bitnami/nginx --dry-run --debug 2>&1 | grep -A 20 "COMPUTED VALUES"
```

This shows all the final values after merging defaults with your overrides.

Key Takeaways

1. **`helm repo add` + `helm repo update`** = Set up your chart sources
2. **`helm search repo`** = Find charts in your repos
3. **`helm install --dry-run --debug`** = Preview before installing
4. **`helm install <release> <chart>`** = Install the chart
5. **`helm list` and `helm status`** = Check your releases
6. **`kubectl get pods,svc`** = Verify Kubernetes resources
7. **`helm uninstall`** = Clean up when done

Congratulations! You've installed your first Helm release. Next, we'll create our own chart! # 03 – Creating Helm Charts (Theory)

Estimated reading time: 15 minutes

What is a Helm Chart?

The Simple Explanation

Think of a Helm chart like a **recipe in a cookbook**:

- A **recipe** tells you what ingredients you need and how to combine them
- A **Helm chart** tells Kubernetes what resources you need and how to configure them

When you follow a recipe, you might adjust it (less salt, more garlic). Similarly, with a Helm chart, you can adjust configuration values (different port, more replicas).

The Technical Definition

A Helm chart is a **collection of files** that describe a set of Kubernetes resources. It packages all the YAML templates, default values, and metadata needed to deploy an application.

Why Create Your Own Charts?

Scenario 1: No Existing Chart

You've built a custom application. There's no public chart for it. You need to create one to deploy it to Kubernetes.

Scenario 2: Company Standards

Your organization has specific requirements (labels, annotations, security settings) that aren't in public charts.

Scenario 3: Simplified Deployment

Instead of managing 10 separate YAML files, you can bundle them into one chart that's easy to install and upgrade.

Scenario 4: Reusability

Create once, deploy many times with different configurations.

Chart Directory Structure

When you create a chart, Helm generates a specific folder structure:

```
mychart/
  Chart.yaml          # Required: Chart metadata
  values.yaml         # Required: Default configuration values
  charts/             # Optional: Dependencies (subcharts)
  templates/          # Required: Template files
      deployment.yaml
      service.yaml
      ingress.yaml
      NOTES.txt       # Optional: Post-install instructions
      _helpers.tpl    # Optional: Reusable template snippets
      tests/          # Optional: Test files
          test-connection.yaml
  .helmignore         # Optional: Patterns to ignore when packaging
  README.md           # Optional: Documentation
```

Let's Understand Each File:

---

1. Chart.yaml (Required)

The **identity card** of your chart. Contains metadata about the chart itself.

Example:

```yaml
apiVersion: v2                    # Helm 3 uses v2
name: mychart                     # Chart name (must match folder name)
description: A Helm chart for my application
type: application                 # "application" or "library"
version: 0.1.0                    # Chart version (you control this)
appVersion: "1.0.0"               # Version of the app being deployed
keywords:
  - web
  - nginx
maintainers:
  - name: Your Name
    email: you@example.com
```

Key fields explained: - `apiVersion: v2` – Always "v2" for Helm 3 charts - `name` – How users reference your chart - `version` – The chart package version (follows semantic versioning) - `appVersion` – The version of the actual application (nginx 1.25, your-app 2.1, etc.)

Why two versions? - Chart version: Changes when you modify the chart (new features, bug fixes) - App version: Changes when you update the application inside

---

2. values.yaml (Required)

The **default settings** for your chart. Users can override any of these.

Example:

```yaml
# Container image configuration
image:
  repository: nginx       # Docker image name
  tag: "1.25"             # Image version
  pullPolicy: IfNotPresent

# Deployment configuration
replicaCount: 1           # Number of pods

# Service configuration
service:
  type: ClusterIP         # ClusterIP, NodePort, or LoadBalancer
  port: 80                # Port the service listens on

# Resource limits
resources:
  limits:
    cpu: 100m
    memory: 128Mi
  requests:
    cpu: 50m
    memory: 64Mi
```

```yaml
# Feature toggles
ingress:
  enabled: false          # Set to true to create an Ingress
```

Why values.yaml matters: 1. Documents all configurable options in one place 2. Provides sensible defaults 3. Users can override any value without editing templates

---

3. templates/ Directory (Required)

Contains **Go template files** that generate Kubernetes YAML.

How templating works:

Instead of hardcoding values, you use placeholders:

Regular Kubernetes YAML (hardcoded):

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
```

Helm Template (dynamic):

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Release.Name }}-app
spec:
  replicas: {{ .Values.replicaCount }}
```

The `{{ ... }}` syntax inserts values dynamically: - `{{ .Release.Name }}` → The name of the release (e.g., "prod-nginx") - `{{ .Values.replicaCount }}` → The value from values.yaml (e.g., 3)

---

4. _helpers.tpl (Common Patterns)

A special file containing **reusable template snippets**. The underscore prefix tells Helm not to output this file directly.

Example:

```
{{/*
Generate a standard name for resources
*/}}
{{- define "mychart.name" -}}
{{- default .Chart.Name .Values.nameOverride | trunc 63 | trimSuffix "-" }}
{{- end }}

{{/*
Generate standard labels
*/}}
{{- define "mychart.labels" -}}
helm.sh/chart: {{ .Chart.Name }}-{{ .Chart.Version }}
app.kubernetes.io/name: {{ include "mychart.name" . }}
app.kubernetes.io/instance: {{ .Release.Name }}
{{- end }}
```

Usage in other templates:

```yaml
metadata:
  name: {{ include "mychart.name" . }}
  labels:
    {{- include "mychart.labels" . | nindent 4 }}
```

Why use helpers: - DRY (Don't Repeat Yourself) – Define once, use everywhere - Consistency – All resources get the same labels - Maintenance – Change in one place, updates everywhere

---

5. NOTES.txt (Post-Install Instructions)

Displayed after a successful install. Uses the same templating syntax.

Example:

```
Congratulations! {{ .Release.Name }} has been deployed.

To access your application:

{{- if eq .Values.service.type "NodePort" }}
  Get the NodePort:
  kubectl get svc {{ .Release.Name }} -o jsonpath='{.spec.ports[0].nodePort}'
{{- else if eq .Values.service.type "LoadBalancer" }}
  Wait for the external IP:
  kubectl get svc {{ .Release.Name }} -w
{{- else }}
  Run port-forward:
  kubectl port-forward svc/{{ .Release.Name }} 8080:{{ .Values.service.port }}
{{- end }}
```

---

6. .helmignore

Like `.gitignore`, specifies files to exclude when packaging the chart.

Example:

```
# Patterns to ignore when building packages
.git
.gitignore
*.swp
*.bak
*.tmp
.DS_Store
```

Template Syntax Deep Dive

Basic Syntax

Syntax	Description	Example
`{{ .Values.x }}`	Access values.yaml	`{{ .Values.replicaCount }}`
`{{ .Release.Name }}`	Release name	`{{ .Release.Name }}`
`{{ .Chart.Name }}`	Chart name	`{{ .Chart.Name }}`
`{{ .Release.Namespace }}`	Target namespace	`{{ .Release.Namespace }}`

Built-in Objects

Available in every template:

```
.Values      → Contents of values.yaml (and any overrides)
.Release     → Information about the release
  .Name      → Release name
  .Namespace → Namespace
  .Revision  → Revision number
  .IsUpgrade → true if this is an upgrade
  .IsInstall → true if this is an install
.Chart       → Contents of Chart.yaml
  .Name      → Chart name
  .Version   → Chart version
.Files       → Access non-special files in the chart
.Capabilities → Info about the Kubernetes cluster
```

Conditional Logic

if/else:

```
{{- if .Values.ingress.enabled }}
apiVersion: networking.k8s.io/v1
kind: Ingress
# ... ingress configuration
{{- end }}
```

if/else if/else:

```
{{- if eq .Values.env "production" }}
replicas: 3
{{- else if eq .Values.env "staging" }}
replicas: 2
{{- else }}
replicas: 1
{{- end }}
```

Loops

range (for-each):

```
{{- range .Values.ports }}
- port: {{ .port }}
  name: {{ .name }}
{{- end }}
```

With values.yaml:

```
ports:
  - port: 80
    name: http
  - port: 443
    name: https
```

Output:

```
- port: 80
  name: http
- port: 443
  name: https
```

Whitespace Control

The `-` inside `{{- }}` and `{{ -}}` controls whitespace:

- `{{-` removes whitespace/newlines before
- `-}}` removes whitespace/newlines after

Without whitespace control:

```
metadata:
  labels:
{{ include "mychart.labels" . }}
```

May produce extra blank lines.

With whitespace control:

```
metadata:
  labels:
{{- include "mychart.labels" . | nindent 4 }}
```

Produces clean output.

Functions and Pipelines

Helm includes many useful functions.

Common Functions

Function	Description	Example
`default`	Provide default value	`{{ default "nginx" .Values.image }}`
`quote`	Wrap in quotes	`{{ .Values.name | quote }}`
`upper`	Uppercase	`{{ .Values.name | upper }}`
`lower`	Lowercase	`{{ .Values.name | lower }}`
`trunc`	Truncate string	`{{ .Values.name | trunc 63 }}`
`trim`	Remove whitespace	`{{ .Values.name | trim }}`
`indent`	Add indentation	`{{ .Values.config | indent 4 }}`
`nindent`	Newline + indent	`{{ .Values.config | nindent 4 }}`
`toYaml`	Convert to YAML	`{{ .Values.resources | toYaml }}`

Pipeline Syntax

Functions can be chained with `|` (pipe):

```
# Start with a value, then apply functions left-to-right
name: {{ .Values.name | default "myapp" | upper | quote }}

# Equivalent to: quote(upper(default("myapp", .Values.name)))
```

Visual: Chart Creation Flow

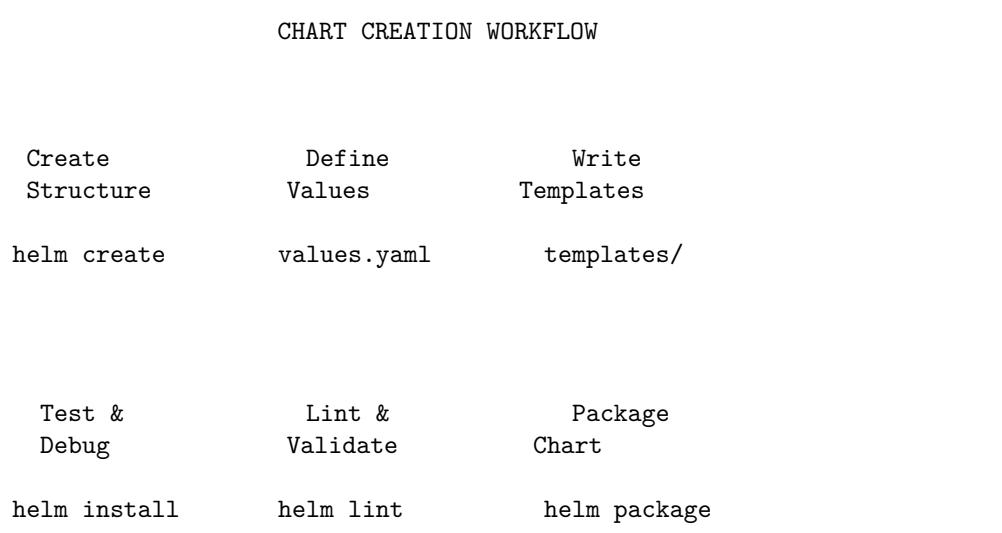


Chart Design Best Practices

1. Use Meaningful Defaults

```yaml
# Good: Works out of the box
replicaCount: 1
service:
  type: ClusterIP
  port: 80

# Bad: Requires user to set everything
replicaCount: null   # Forces user to set this
```

2. Make Things Toggleable

```yaml
# Allow users to enable/disable features
ingress:
  enabled: false     # Disabled by default

autoscaling:
  enabled: false     # Disabled by default
  minReplicas: 1
  maxReplicas: 10
```

3. Use Consistent Naming

```yaml
# Use kebab-case for chart names: my-app
# Use camelCase for values: replicaCount, servicePort
```

4. Add Comments

```yaml
# -- Number of replicas for the deployment
replicaCount: 1

# -- Container image repository
image:
```

```
  repository: nginx
  # -- Image tag (overrides Chart appVersion)
  tag: ""
```

5. Validate with Schema

Create `values.schema.json` to validate user inputs:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "properties": {
    "replicaCount": {
      "type": "integer",
      "minimum": 1
    }
  }
}
```

Common Patterns

Pattern 1: Full Resource Name

```
{{- define "mychart.fullname" -}}
{{- if .Values.fullnameOverride }}
{{- .Values.fullnameOverride | trunc 63 | trimSuffix "-" }}
{{- else }}
{{- printf "%s-%s" .Release.Name .Chart.Name | trunc 63 | trimSuffix "-" }}
{{- end }}
{{- end }}
```

Result: `prod-nginx-mychart` → Unique names per release

Pattern 2: Standard Labels

```
{{- define "mychart.labels" -}}
helm.sh/chart: {{ printf "%s-%s" .Chart.Name .Chart.Version | replace "+" "_" }}
app.kubernetes.io/name: {{ include "mychart.name" . }}
app.kubernetes.io/instance: {{ .Release.Name }}
app.kubernetes.io/version: {{ .Chart.AppVersion | quote }}
app.kubernetes.io/managed-by: {{ .Release.Service }}
{{- end }}
```

Pattern 3: Optional Resources

```
{{- if .Values.serviceAccount.create -}}
apiVersion: v1
kind: ServiceAccount
metadata:
  name: {{ include "mychart.serviceAccountName" . }}
{{- end }}
```

Key Takeaways

1. **Chart = Package** – Contains everything needed to deploy an application
2. **values.yaml = Configuration** – Sensible defaults users can override
3. **templates/ = Dynamic YAML** – Go templates that generate Kubernetes resources
4. **Templating = Power** – Conditionals, loops, functions make charts flexible

5. **helpers = Reusability** – Define once, use throughout the chart
6. **NOTES.txt = UX** – Help users know what to do after installation

---

Chart Dependencies

What are Chart Dependencies?

When your application needs other services (like a database), you can include them as dependencies. Instead of creating one massive chart, you can: - Split applications into separate charts - Reuse existing charts (like MySQL, Redis) - Manage complex deployments as a single unit

Why Use Dependencies?

```
                    WITHOUT DEPENDENCIES


  Manual Installation:

  1. helm install mysql bitnami/mysql -n myapp
  2. helm install redis bitnami/redis -n myapp
  3. helm install myapp ./myapp-chart -n myapp

  Problems:
  - Must remember the order
  - Must manually track which releases belong together
  - Cleanup is error-prone


                     WITH DEPENDENCIES


  Single Command:

  helm install myapp ./myapp-chart -n myapp

  Benefits:
  - One command installs everything
  - All resources tracked as one release
  - One command to upgrade/rollback/uninstall
```

Defining Dependencies in Chart.yaml

Modern approach (Helm 3):

```yaml
# Chart.yaml
apiVersion: v2
name: my-app
version: 1.0.0
dependencies:
  - name: mysql
    version: "9.0.0"
    repository: "https://charts.bitnami.com/bitnami"
  - name: redis
    version: "18.0.0"
    repository: "https://charts.bitnami.com/bitnami"
    condition: redis.enabled    # Optional: only include if enabled
```

Managing Dependencies

Download dependencies:

```
helm dependency update ./my-app
```

This downloads the dependency charts to `charts/` folder and creates `Chart.lock`:

```
my-app/
   Chart.yaml
   Chart.lock              # Locks dependency versions
   charts/                 # Downloaded dependencies
       mysql-9.0.0.tgz
       redis-18.0.0.tgz
   values.yaml
   templates/
```

Build dependencies (from lock file):

```
helm dependency build ./my-app
```

Configuring Dependencies via values.yaml

Override subchart values by nesting under the dependency name:

```yaml
# values.yaml for parent chart
replicaCount: 1
image:
  repository: my-app
  tag: "1.0"

# Configuration for mysql subchart
mysql:
  auth:
    rootPassword: "secretpassword"
    database: "myapp"
  primary:
    persistence:
      size: 10Gi

# Configuration for redis subchart
redis:
  enabled: true    # Controls the condition
  auth:
    password: "redispassword"
```

Dependency Workflow Diagram

```
                    DEPENDENCY WORKFLOW


  1. Define Dependencies


         Chart.yaml

        dependencies:
         - mysql
         - redis
```

2. Download Dependencies

helm dependency update

charts/
mysql-9.0.tgz
redis-18.0.tgz

3. Install Parent Chart

helm install myapp ./myapp

Kubernetes Cluster

my-app

mysql

redis

Dependency Conditions and Tags

Conditionally include dependencies:

```yaml
# Chart.yaml
dependencies:
  - name: redis
    version: "18.0.0"
    repository: "https://charts.bitnami.com/bitnami"
    condition: redis.enabled
    tags:
      - cache
```

```yaml
# values.yaml
redis:
  enabled: false    # Set to true to include redis

# Or use tags
tags:
  cache: true       # Enable all charts tagged "cache"
```

Accessing Subchart Values in Templates

To reference subchart services from your parent chart templates:

```yaml
# In parent chart's deployment.yaml
env:
  - name: DATABASE_HOST
    value: "{{ .Release.Name }}-mysql"
  - name: DATABASE_USER
    value: "{{ .Values.mysql.auth.username }}"
  - name: REDIS_HOST
    value: "{{ .Release.Name }}-redis-master"
```

Dependency Best Practices

1. **Always pin versions** - Use exact versions, not ranges

   ```
   version: "9.0.0"      # Good
   version: ">=9.0.0"    # Risky
   ```

2. **Use conditions** - Don't force dependencies

   ```
   condition: redis.enabled
   ```

3. **Document subchart values** - Show users what they can configure
4. **Test with and without dependencies**
5. **Keep dependencies minimal** - Only include what you truly need

---

Next: We'll create our own chart from scratch! # 03 – Creating Helm Charts (Lab)

Objective: Create a custom Helm chart from scratch, customize it, lint it, and install it.

Estimated duration: 30–35 minutes

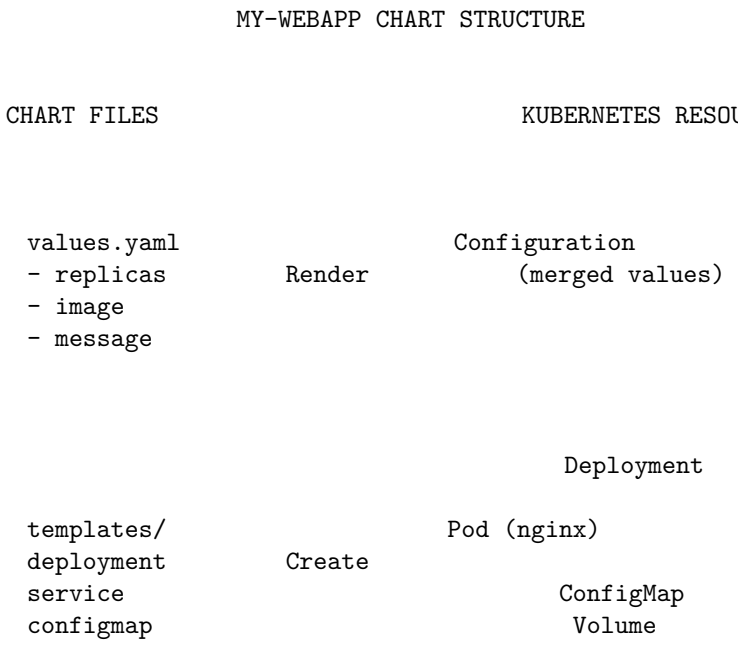
Prerequisites: Helm installed, Minikube running, completed Module 02.

What You Will Build

In this lab, you'll create a chart for a simple web application: - Deployment (runs nginx containers) - Service (exposes the application) - ConfigMap (custom welcome page) - Customizable through values

---

Architecture Diagram



```
Service
(ClusterIP:80)
```

Step-by-Step Instructions

Step 1: Create the Chart Scaffold

Helm can generate a starter chart structure for you.

Navigate to a working directory:

```
cd ~/
mkdir helm-training && cd helm-training
```

Create the chart:

```
helm create my-webapp
```

Expected output:

```
Creating my-webapp
```

What was created:

```
my-webapp/
   Chart.yaml
   values.yaml
   charts/
   templates/
       deployment.yaml
       service.yaml
       ingress.yaml
       hpa.yaml
       serviceaccount.yaml
       NOTES.txt
       _helpers.tpl
       tests/
           test-connection.yaml
   .helmignore
```

Helm generates a complete, working chart! Let's explore it.

Step 2: Explore the Generated Files

View the chart metadata:

```
cat my-webapp/Chart.yaml
```

Expected output:

```
apiVersion: v2
name: my-webapp
description: A Helm chart for Kubernetes
type: application
version: 0.1.0
appVersion: "1.16.0"
```

What this tells us: - Chart name: my-webapp - Version: 0.1.0 (chart version) - AppVersion: 1.16.0 (the app version, we'll change this)

View the default values:

```
cat my-webapp/values.yaml
```

This is a long file. Let's look at key sections:

```
head -n 30 my-webapp/values.yaml
```

Key values you'll see:

```
replicaCount: 1

image:
  repository: nginx
  pullPolicy: IfNotPresent
  tag: ""

service:
  type: ClusterIP
  port: 80
```

View a template file:

```
cat my-webapp/templates/deployment.yaml
```

What you'll see: Go template syntax with: - `{{ .Values.replicaCount }}` – pulls from values.yaml - `{{ include "my-webapp.fullname" . }}` – uses a helper function - Conditional blocks with `{{- if ... }}`

Step 3: Customize Chart.yaml

Let's update the chart metadata.

Edit Chart.yaml:

```
cat > my-webapp/Chart.yaml << 'EOF'
apiVersion: v2
name: my-webapp
description: A custom web application chart for Helm training
type: application
version: 0.1.0
appVersion: "1.25.0"
keywords:
  - nginx
  - web
  - training
maintainers:
  - name: Helm Training Student
    email: student@example.com
EOF
```

What we changed: - Better description - Updated appVersion to 1.25.0 (current nginx) - Added keywords and maintainer info

---

Step 4: Simplify values.yaml

The generated values.yaml has many options. Let's simplify it for our training.

Replace values.yaml:

```
cat > my-webapp/values.yaml << 'EOF'
# Number of pod replicas
replicaCount: 1

# Container image settings
image:
  repository: nginx
  tag: "1.25"
  pullPolicy: IfNotPresent

# Service settings
service:
  type: ClusterIP
  port: 80

# Resource limits (important for production)
resources:
  limits:
    cpu: 100m
    memory: 128Mi
  requests:
    cpu: 50m
    memory: 64Mi

# Custom welcome message (we'll use this!)
welcomeMessage: "Welcome to Helm Training!"
EOF
```

What we configured: - 1 replica (can be overridden) - nginx 1.25 image - ClusterIP service on port 80 - Resource limits (good practice) - Custom welcome message (we'll use this in a ConfigMap)

Step 5: Create a Custom ConfigMap Template

Let's add a ConfigMap that creates a custom welcome page.

Create the ConfigMap template:

```
cat > my-webapp/templates/configmap.yaml << 'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "my-webapp.fullname" . }}-html
  labels:
    {{- include "my-webapp.labels" . | nindent 4 }}
data:
  index.html: |
    <!DOCTYPE html>
    <html>
    <head>
      <title>{{ .Values.welcomeMessage }}</title>
      <style>
        body { font-family: Arial, sans-serif; text-align: center; padding: 50px; }
        h1 { color: #326CE5; }
        .info { background: #f0f0f0; padding: 20px; border-radius: 10px; margin: 20px auto; max-width: 600p
      </style>
    </head>
    <body>
```

```
      <h1>{{ .Values.welcomeMessage }}</h1>
      <div class="info">
        <p><strong>Release Name:</strong> {{ .Release.Name }}</p>
        <p><strong>Namespace:</strong> {{ .Release.Namespace }}</p>
        <p><strong>Chart Version:</strong> {{ .Chart.Version }}</p>
        <p><strong>App Version:</strong> {{ .Chart.AppVersion }}</p>
        <p><strong>Replicas:</strong> {{ .Values.replicaCount }}</p>
      </div>
    </body>
    </html>
EOF
```

What this does: - Creates a ConfigMap with a custom `index.html` - Uses template values to show release information - The HTML is dynamically generated with Helm values

---

Step 6: Update the Deployment to Use the ConfigMap

We need to mount the ConfigMap into nginx so it serves our custom page.

View the current deployment template:

```
head -n 50 my-webapp/templates/deployment.yaml
```

Replace the deployment template:

```
cat > my-webapp/templates/deployment.yaml << 'EOF'
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "my-webapp.fullname" . }}
  labels:
    {{- include "my-webapp.labels" . | nindent 4 }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      {{- include "my-webapp.selectorLabels" . | nindent 6 }}
  template:
    metadata:
      labels:
        {{- include "my-webapp.selectorLabels" . | nindent 8 }}
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          ports:
            - name: http
              containerPort: 80
              protocol: TCP
          # Mount the custom HTML from ConfigMap
          volumeMounts:
            - name: html-volume
              mountPath: /usr/share/nginx/html
          resources:
            {{- toYaml .Values.resources | nindent 12 }}
          livenessProbe:
            httpGet:
```

```
            path: /
            port: http
          initialDelaySeconds: 5
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /
            port: http
          initialDelaySeconds: 5
          periodSeconds: 10
      volumes:
        - name: html-volume
          configMap:
            name: {{ include "my-webapp.fullname" . }}-html
EOF
```

What we changed: - Added a volumeMount to mount our ConfigMap - Added a volume referencing the ConfigMap - Added liveness and readiness probes (best practice) - Used `toYaml` to include resources from values

---

Step 7: Update the Service Template

Let's ensure the service template is clean:

Replace the service template:

```
cat > my-webapp/templates/service.yaml << 'EOF'
apiVersion: v1
kind: Service
metadata:
  name: {{ include "my-webapp.fullname" . }}
  labels:
    {{- include "my-webapp.labels" . | nindent 4 }}
spec:
  type: {{ .Values.service.type }}
  ports:
    - port: {{ .Values.service.port }}
      targetPort: http
      protocol: TCP
      name: http
  selector:
    {{- include "my-webapp.selectorLabels" . | nindent 4 }}
EOF
```

Step 8: Update NOTES.txt

Update the post-install instructions:

```
cat > my-webapp/templates/NOTES.txt << 'EOF'
 Congratulations! {{ .Release.Name }} has been deployed!

 Release Information:
   - Name: {{ .Release.Name }}
   - Namespace: {{ .Release.Namespace }}
   - Replicas: {{ .Values.replicaCount }}

 To access your application:
```

```
{{- if eq .Values.service.type "NodePort" }}
   Run: minikube service {{ include "my-webapp.fullname" . }} --url
{{- else if eq .Values.service.type "LoadBalancer" }}
   Run: kubectl get svc {{ include "my-webapp.fullname" . }} -w
{{- else }}
   Run: kubectl port-forward svc/{{ include "my-webapp.fullname" . }} 8080:{{ .Values.service.port }}
   Then open: http://localhost:8080
{{- end }}

 Useful commands:
   - Check status: helm status {{ .Release.Name }}
   - View pods: kubectl get pods -l app.kubernetes.io/instance={{ .Release.Name }}
   - View logs: kubectl logs -l app.kubernetes.io/instance={{ .Release.Name }}
   - Uninstall: helm uninstall {{ .Release.Name }}
EOF
```

Step 9: Delete Unnecessary Templates

The generated chart has files we don't need. Let's remove them:

```
rm my-webapp/templates/ingress.yaml
rm my-webapp/templates/hpa.yaml
rm my-webapp/templates/serviceaccount.yaml
rm -rf my-webapp/templates/tests
```

Why remove these: - ingress.yaml – We're not using Ingress in this lab - hpa.yaml – Horizontal Pod Autoscaler not needed - serviceaccount.yaml – Default service account is fine - tests/ – We'll cover testing in a later module

---

Step 10: Lint Your Chart

Before installing, check for errors:

Run helm lint:

```
helm lint my-webapp
```

Expected output:

```
==> Linting my-webapp
[INFO] Chart.yaml: icon is recommended

1 chart(s) linted, 0 chart(s) failed
```

What this means: - 0 charts failed – No errors - INFO: icon is recommended – Just a suggestion, not an error

---

Step 11: Preview the Installation (Template Rendering)

See what Kubernetes YAML will be generated:

```
helm template my-release my-webapp
```

This shows all the rendered YAML. Let's pipe it through head:

```
helm template my-release my-webapp | head -n 60
```

What you'll see: - ConfigMap with your custom HTML - Service definition - Deployment with volume mounts

Check for your welcome message:

```
helm template my-release my-webapp | grep -A 5 "welcomeMessage"
```

You should see your custom welcome message in the HTML.

Step 12: Install Your Chart

Now let's install it!

```
helm install my-release my-webapp
```

Expected output:

```
NAME: my-release
LAST DEPLOYED: Thu Jan 15 11:00:00 2024
NAMESPACE: default
STATUS: deployed
REVISION: 1

 Congratulations! my-release has been deployed!
...
```

Step 13: Verify the Installation

Check the release:

```
helm list
```

Expected output:

```
NAME         NAMESPACE   REVISION   UPDATED                   STATUS     CHART           APP VERSION
my-release   default     1          2024-01-15 11:00:00       deployed   my-webapp-0.1.0 1.25.0
```

Check pods:

```
kubectl get pods
```

Expected output:

```
NAME                                  READY   STATUS    RESTARTS   AGE
my-release-my-webapp-7d4f8b7b9c-xxxxx   1/1     Running   0          30s
```

Check services:

```
kubectl get svc
```

Check the ConfigMap:

```
kubectl get configmap
kubectl describe configmap my-release-my-webapp-html
```

You should see your custom HTML in the ConfigMap.

Step 14: Test Your Application

Start port-forward:

```
kubectl port-forward svc/my-release-my-webapp 8080:80
```

In a new terminal, test with curl:

```
curl http://localhost:8080
```

Expected output:

```
<!DOCTYPE html>
<html>
<head>
  <title>Welcome to Helm Training!</title>
...
```

You should see your custom welcome page with release information!

Stop port-forward: Press `Ctrl+C`

Step 15: Install with Custom Values

Let's install another release with different values:

```
helm install custom-release my-webapp \
  --set replicaCount=2 \
  --set welcomeMessage="Hello from Custom Release!" \
  --set service.type=NodePort
```

Verify two releases exist:

```
helm list
```

Expected output:

```
NAME            NAMESPACE   REVISION    STATUS      CHART
my-release      default     1           deployed    my-webapp-0.1.0
custom-release  default     1           deployed    my-webapp-0.1.0
```

Check pods (should have 3 total):

```
kubectl get pods
```

Access the custom release via Minikube:

```
minikube service custom-release-my-webapp --url
```

Open that URL in your browser and see your custom message!

Step 16: Clean Up

Uninstall both releases:

```
helm uninstall my-release
helm uninstall custom-release
```

Verify:

```
helm list
kubectl get pods
```

Everything should be gone.

Expected Results Summary

Step	What You Did	Expected Result
Create chart	`helm create my-webapp`	Chart structure created
Customize	Edited values.yaml, added configmap.yaml	Custom configuration
Lint	`helm lint my-webapp`	0 charts failed

Step	What You Did	Expected Result
Template	`helm template my-release my-webapp`	Rendered YAML shown
Install	`helm install my-release my-webapp`	STATUS: deployed
Test	Port-forward + curl	Custom welcome page
Custom install	`--set` flags	Different configuration
Uninstall	`helm uninstall`	Releases removed

Troubleshooting

Problem: "error parsing" when running helm template

Cause: YAML syntax error in templates

Solution: Check for: - Missing colons - Incorrect indentation - Unclosed template brackets `{{ }}`

Debug:

```
helm template my-release my-webapp --debug
```

Problem: Pod stuck in CrashLoopBackOff

Cause: Container can't start

Solution: Check logs:

```
kubectl logs -l app.kubernetes.io/instance=my-release
kubectl describe pod -l app.kubernetes.io/instance=my-release
```

Problem: ConfigMap not mounting

Cause: Volume mount path or name mismatch

Solution: Verify volume and volumeMount names match:

```
kubectl describe pod -l app.kubernetes.io/instance=my-release
```

Look for Events at the bottom.

Chart Files Summary

After this lab, your chart should have:

```
                    CHART DIRECTORY STRUCTURE


    my-webapp/

       Chart.yaml            Identity (name, version)

       values.yaml           Default configuration

       templates/            Kubernetes resource templates

           deployment.yaml       Pod specification
```

```
            volumeMounts    Mounts ConfigMap

    service.yaml         Network exposure

    configmap.yaml        Custom HTML content

    NOTES.txt           Post-install instructions

    _helpers.tpl         Reusable template functions

  charts/              Dependencies (empty)

  .helmignore           Files to exclude from package
```

Key Takeaways

1. **`helm create`** generates a working starter chart
2. **values.yaml** contains all configurable options
3. **Template syntax** (`{{ .Values.x }}`) makes charts dynamic
4. **ConfigMaps** are great for injecting configuration
5. **Volume mounts** connect ConfigMaps to containers
6. **`helm lint`** catches errors before installation
7. **`helm template`** previews rendered YAML
8. **`--set`** overrides values at install time

Congratulations! You've created your first custom Helm chart!

Keep the `my-webapp` chart – we'll use it in the next modules! # 04 – Deploying & Managing Releases (Theory)

Estimated reading time: 15 minutes

What is a Release?

The Simple Explanation

Think of a **release** like a **running instance** of a recipe:

- A **chart** is the recipe (instructions for making a cake)
- A **release** is the actual cake you made (running in your kitchen)

You can make the same recipe multiple times: - Birthday cake for Alice → Release "alice-cake" - Birthday cake for Bob → Release "bob-cake"

Same recipe, different instances.

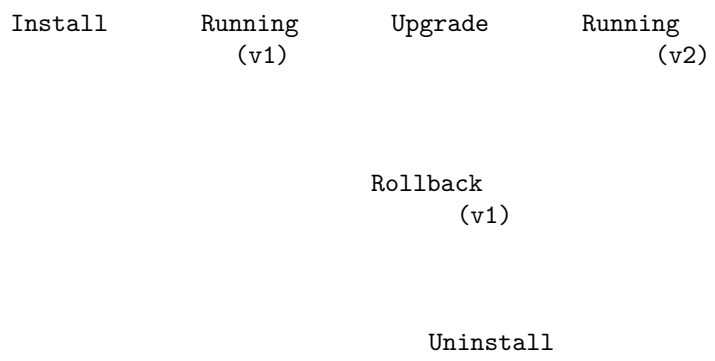
The Technical Definition

A **release** is a chart deployed to Kubernetes with a specific configuration. Each release: - Has a unique name within a namespace - Tracks its own revision history - Manages its own set of Kubernetes resources - Can be upgraded, rolled back, or uninstalled independently

---

Release Lifecycle

A release goes through several states:



Release States

State	Description
`deployed`	Current release is active and running
`superseded`	Previous release version (replaced by newer)
`failed`	Release failed to install or upgrade
`uninstalling`	Release is being deleted
`pending-install`	Install in progress
`pending-upgrade`	Upgrade in progress
`pending-rollback`	Rollback in progress

The Install Command

Basic Installation

```
helm install <release-name> <chart>
```

Example:

```
helm install my-app bitnami/nginx
```

What happens: 1. Helm fetches the chart (from repo or local) 2. Merges default values with any overrides 3. Renders all templates 4. Sends resources to Kubernetes API 5. Records the release in cluster (as a Secret) 6. Displays NOTES.txt

---

Installation Options

Option	Description	Example
`--namespace`	Target namespace	`--namespace prod`
`--create-namespace`	Create namespace if missing	`--create-namespace`
`--values` / `-f`	Use values file	`-f custom-values.yaml`
`--set`	Override single value	`--set replicas=3`
`--set-string`	Override as string	`--set-string version="1.0"`
`--set-file`	Set value from file	`--set-file cert=cert.pem`
`--dry-run`	Preview without installing	`--dry-run`
`--debug`	Show debug information	`--debug`

Option	Description	Example
`--wait`	Wait for pods ready	`--wait`
`--timeout`	Set wait timeout	`--timeout 5m`
`--atomic`	Auto-rollback on failure	`--atomic`

Ways to Override Values

Method 1: Using –set (single values)

```
helm install my-app ./mychart --set replicaCount=3
```

Method 2: Using –set for nested values (dot notation)

```
helm install my-app ./mychart --set image.tag=v2.0
```

Method 3: Using –set for lists

```
helm install my-app ./mychart --set "hosts[0]=host1.com,hosts[1]=host2.com"
```

Method 4: Using a values file

```
# Create a values file
cat > custom-values.yaml << EOF
replicaCount: 3
image:
  tag: v2.0
EOF

# Use it
helm install my-app ./mychart -f custom-values.yaml
```

Method 5: Combining multiple sources

```
# Values are merged in order (later values win)
helm install my-app ./mychart \
  -f base-values.yaml \
  -f prod-values.yaml \
  --set image.tag=latest
```

Priority (lowest to highest): 1. Chart's values.yaml 2. First values file (-f) 3. Second values file (-f) 4. –set values

---

The Upgrade Command

Upgrades update an existing release to a new chart version or configuration.

Basic Upgrade

```
helm upgrade <release-name> <chart>
```

Example:

```
helm upgrade my-app bitnami/nginx --set replicaCount=3
```

What happens: 1. Fetches the chart 2. Merges new values with previous release values 3. Renders templates 4. Computes diff with current release 5. Applies changes to Kubernetes 6. Increments revision number 7. Records new revision in cluster

---

Important: Value Persistence

By default, upgrade **resets values to chart defaults**, then applies your new values.

Problem scenario:

```
# Install with replica=3
helm install my-app ./mychart --set replicaCount=3

# Upgrade without --reuse-values
helm upgrade my-app ./mychart --set image.tag=v2.0
# replicaCount is now back to default (1)!
```

Solution 1: Use –reuse-values

```
helm upgrade my-app ./mychart --reuse-values --set image.tag=v2.0
# Keeps replicaCount=3, updates image.tag
```

Solution 2: Always specify all values

```
helm upgrade my-app ./mychart -f values.yaml
# values.yaml contains ALL your configuration
```

Best practice: Always use a values file for predictable upgrades.

Upgrade Options

Option	Description
`--reuse-values`	Keep existing values, merge new ones
`--reset-values`	Reset to chart defaults (default behavior)
`--install` / `-i`	Install if release doesn't exist
`--atomic`	Rollback on failure
`--wait`	Wait for pods ready
`--force`	Force resource update (delete + recreate)

Install-or-Upgrade Pattern

Use `upgrade --install` to handle both cases:

```
helm upgrade --install my-app ./mychart -f values.yaml
```

What this does: - If "my-app" exists → Upgrade it - If "my-app" doesn't exist → Install it

This is useful in CI/CD pipelines where you don't know the current state.

Revision History

Each install/upgrade creates a new **revision**.

View History

```
helm history <release-name>
```

Example output:

```
REVISION   UPDATED                   STATUS       CHART          APP VERSION   DESCRIPTION
1          Thu Jan 15 10:00:00 2024   superseded   mychart-0.1.0  1.0.0         Install complete
2          Thu Jan 15 11:00:00 2024   superseded   mychart-0.1.0  1.0.0         Upgrade complete
3          Thu Jan 15 12:00:00 2024   deployed     mychart-0.2.0  1.1.0         Upgrade complete
```

Understanding the output: - Revision 1: Original install - Revision 2: First upgrade (same chart version, different values?) - Revision 3: Current (deployed), new chart version

---

How Helm Stores History

Helm stores release information as **Secrets** in the release namespace:

```
kubectl get secrets -l owner=helm
```

Output:

```
NAME                            TYPE                 DATA   AGE
sh.helm.release.v1.my-app.v1    helm.sh/release.v1   1      1h
sh.helm.release.v1.my-app.v2    helm.sh/release.v1   1      30m
sh.helm.release.v1.my-app.v3    helm.sh/release.v1   1      5m
```

Each Secret stores the complete state of that revision.

The Rollback Command

Rollback reverts to a previous revision.

Basic Rollback

```
helm rollback <release-name> <revision>
```

Example:

```
# Current: revision 3
# Rollback to revision 2
helm rollback my-app 2
```

What happens: 1. Helm reads the state from revision 2's Secret 2. Applies that configuration 3. Creates a NEW revision (4) with revision 2's content 4. Updates the deployment

The history after rollback:

```
REVISION   STATUS       DESCRIPTION
1          superseded   Install complete
2          superseded   Upgrade complete
3          superseded   Upgrade complete
4          deployed     Rollback to 2
```

Notice: Rollback creates a **new** revision, not a true "undo".

Rollback Options

```
helm rollback my-app 2 --wait       # Wait for pods ready
helm rollback my-app 2 --dry-run    # Preview without applying
helm rollback my-app 2 --force      # Force resource recreation
```

The Uninstall Command

Removes a release and its resources.

Basic Uninstall

```
helm uninstall <release-name>
```

What happens: 1. Helm finds all resources created by the release 2. Deletes them from Kubernetes 3. Removes the release record from cluster

---

Keep History After Uninstall

If you might want to rollback after uninstall:

```
helm uninstall my-app --keep-history
```

Then later:

```
# See the uninstalled release
helm list --all

# Rollback to resurrect it
helm rollback my-app 3
```

Status and Information Commands

helm list

Shows all releases:

```
helm list                    # Current namespace
helm list --all-namespaces   # All namespaces
helm list --all              # Include failed/pending
helm list --filter 'prod-*'  # Filter by name pattern
```

helm status

Shows detailed release status:

```
helm status my-app
```

Output includes: - Release name, namespace, revision - Current status - Last deployment time - NOTES.txt content

helm get

Get specific information:

```
helm get values my-app          # User-supplied values
helm get values my-app --all    # All values (merged)
helm get manifest my-app        # Rendered Kubernetes YAML
helm get notes my-app           # Just the NOTES
helm get hooks my-app           # Hook definitions
helm get all my-app             # Everything
```

Release Best Practices

1. Use Meaningful Release Names

```
# Bad
helm install r1 ./mychart
```

```
# Good
helm install prod-web-frontend ./mychart
helm install dev-api-backend ./mychart
```

Pattern: `<environment>-<component>-<service>`

---

2. Always Use Namespaces

```
helm install my-app ./mychart --namespace prod --create-namespace
```

Benefits: - Isolation between environments - Easier resource management - Better security (RBAC per namespace)

---

3. Use Values Files for Each Environment

```
values/
  values-dev.yaml
  values-staging.yaml
  values-prod.yaml
```

```
helm install my-app ./mychart -f values/values-prod.yaml -n prod
```

---

4. Use –wait and –timeout in CI/CD

```
helm upgrade --install my-app ./mychart \
  --wait \
  --timeout 5m \
  --atomic
```

- `--wait`: Don't return until pods are ready
- `--timeout`: Fail after 5 minutes
- `--atomic`: Auto-rollback if it fails

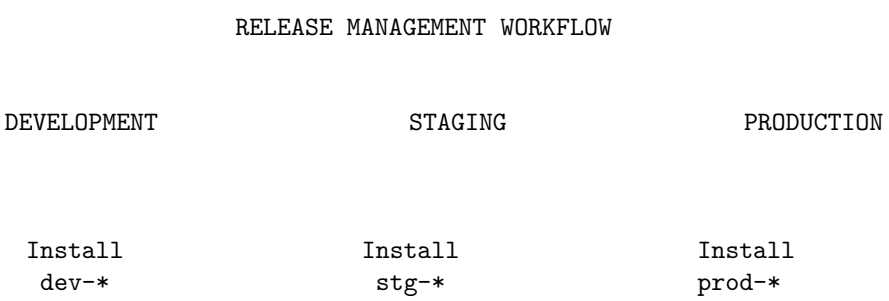
---

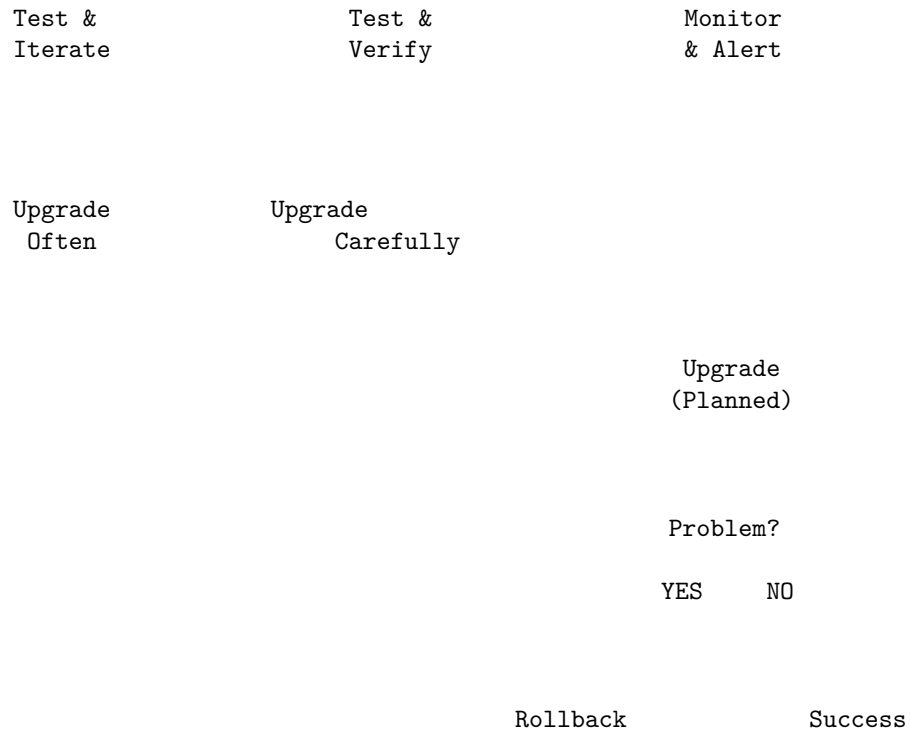
5. Document Your Releases

Create a README with: - Release naming conventions - Values file purpose - Upgrade procedures - Rollback procedures

---

Visual: Release Management Workflow





Key Takeaways

1. **Release = Running Instance** – A chart deployed with specific configuration
2. **Revisions = History** – Every install/upgrade creates a revision
3. **Upgrade carefully** – Use `--reuse-values` or values files
4. **Rollback creates new revision** – It's not a true undo
5. **Use namespaces** – Isolate environments
6. **Use –atomic in CI/CD** – Auto-rollback on failure
7. **Keep release history** – For auditing and recovery

Next: Hands-on release management! # 04 – Deploying & Managing Releases (Lab)

Objective: Practice installing, upgrading, rolling back, and uninstalling releases with various configuration options.

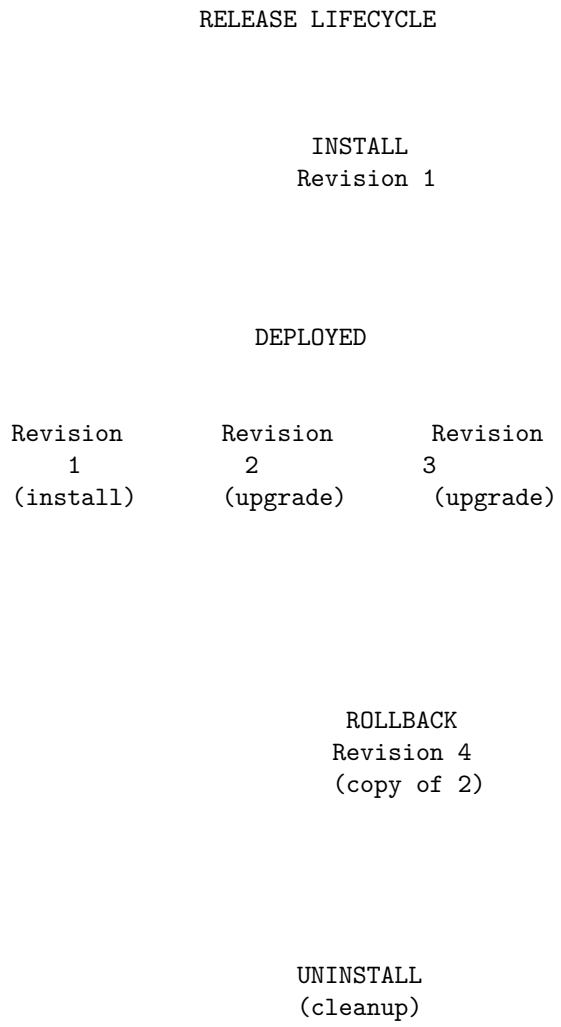
Estimated duration: 25–30 minutes

Prerequisites: Completed Module 03 (you should have the `my-webapp` chart).

What You Will Do

1. Install a release with default values
2. Check release status and information
3. Upgrade with new values
4. View revision history
5. Rollback to a previous version
6. Install with a values file
7. Use upgrade –install pattern
8. Clean up

Release Lifecycle Diagram



Preparation

Make sure you have the my-webapp chart from Module 03:

```
cd ~/helm-training
ls my-webapp
```

Expected output:

```
Chart.yaml  charts  templates  values.yaml
```

If you don't have it, go back and complete Module 03 first.

Step-by-Step Instructions

Step 1: Install with Default Values

Let's install our chart with all defaults.

Run this command:

```
helm install webapp-v1 my-webapp
```

Breaking down the command: - `helm install` = Install a new release - `webapp-v1` = The release name you chose - `my-webapp` = Path to your chart directory

Expected output:

```
NAME: webapp-v1
LAST DEPLOYED: Thu Jan 15 14:00:00 2024
NAMESPACE: default
STATUS: deployed
REVISION: 1
...
```

Verify it's running:

```
kubectl get pods
```

Wait until the pod is Running (1/1 Ready).

Step 2: Check Release Information

List all releases:

```
helm list
```

Expected output:

```
NAME       NAMESPACE  REVISION  UPDATED                   STATUS    CHART           APP VERSION
webapp-v1  default    1         2024-01-15 14:00:00       deployed  my-webapp-0.1.0 1.25.0
```

Get detailed status:

```
helm status webapp-v1
```

This shows: - Release information - Current status - The NOTES.txt output

---

Step 3: View the Current Values

See what values were used:

```
helm get values webapp-v1
```

Expected output:

```
USER-SUPPLIED VALUES:
null
```

Since we didn't override anything, it shows null. The chart used all defaults.

See ALL computed values (defaults + overrides):

```
helm get values webapp-v1 --all | head -n 30
```

This shows all the values from values.yaml.

Step 4: Test the Current Installation

Start port-forward:

```
kubectl port-forward svc/webapp-v1-my-webapp 8080:80 &
```

The & runs it in the background.

Test with curl:

```
curl -s http://localhost:8080 | grep -o "<h1>.*</h1>"
```

Expected output:

```
<h1>Welcome to Helm Training!</h1>
```

Stop the background port-forward:

```
kill %1 2>/dev/null || true
```

Step 5: Upgrade with New Values

Now let's upgrade the release with different values.

Run the upgrade:

```
helm upgrade webapp-v1 my-webapp \
  --set replicaCount=2 \
  --set welcomeMessage="Version 2 - Upgraded!"
```

Expected output:

```
Release "webapp-v1" has been upgraded. Happy Helming!
NAME: webapp-v1
LAST DEPLOYED: Thu Jan 15 14:05:00 2024
NAMESPACE: default
STATUS: deployed
REVISION: 2
...
```

Notice: Revision is now 2!

Step 6: Verify the Upgrade

Check the release:

```
helm list
```

Expected output:

```
NAME       NAMESPACE  REVISION  ...
webapp-v1  default    2         ...
```

Check pods (should see 2 now):

```
kubectl get pods
```

Expected output:

```
NAME                                 READY   STATUS    RESTARTS   AGE
webapp-v1-my-webapp-xxxxx-aaaaa      1/1     Running   0          30s
webapp-v1-my-webapp-xxxxx-bbbbb      1/1     Running   0          30s
```

Verify the new message:

```
kubectl port-forward svc/webapp-v1-my-webapp 8080:80 &
sleep 2
curl -s http://localhost:8080 | grep -o "<h1>.*</h1>"
kill %1 2>/dev/null || true
```

Expected output:

```
<h1>Version 2 - Upgraded!</h1>
```

The ConfigMap was updated with the new message!

Step 7: View Release History

Check the revision history:

```
helm history webapp-v1
```

Expected output:

```
REVISION  UPDATED                   STATUS      CHART           APP VERSION  DESCRIPTION
1         Thu Jan 15 14:00:00 2024  superseded  my-webapp-0.1.0 1.25.0       Install complete
2         Thu Jan 15 14:05:00 2024  deployed    my-webapp-0.1.0 1.25.0       Upgrade complete
```

Understanding this: - Revision 1: Original install (now "superseded" = replaced) - Revision 2: Current version (status "deployed")

Step 8: Upgrade Again (Version 3)

Let's make another upgrade:

```
helm upgrade webapp-v1 my-webapp \
  --set replicaCount=3 \
  --set welcomeMessage="Version 3 - High Availability!"
```

Check history:

```
helm history webapp-v1
```

Expected output:

```
REVISION  STATUS      DESCRIPTION
1         superseded  Install complete
2         superseded  Upgrade complete
3         deployed    Upgrade complete
```

Now we have 3 revisions.

Step 9: Rollback to Previous Version

Something's wrong with version 3! Let's rollback to version 2.

Run the rollback:

```
helm rollback webapp-v1 2
```

Expected output:

```
Rollback was a success! Happy Helming!
```

Check history:

```
helm history webapp-v1
```

Expected output:

```
REVISION  STATUS      DESCRIPTION
1         superseded  Install complete
2         superseded  Upgrade complete
3         superseded  Upgrade complete
4         deployed    Rollback to 2
```

Important: Rollback created revision 4, which is a copy of revision 2.

Verify values rolled back:

```
kubectl get pods    # Should be 2 pods (from revision 2)
```

Test the message:

```
kubectl port-forward svc/webapp-v1-my-webapp 8080:80 &
sleep 2
curl -s http://localhost:8080 | grep -o "<h1>.*</h1>"
kill %1 2>/dev/null || true
```

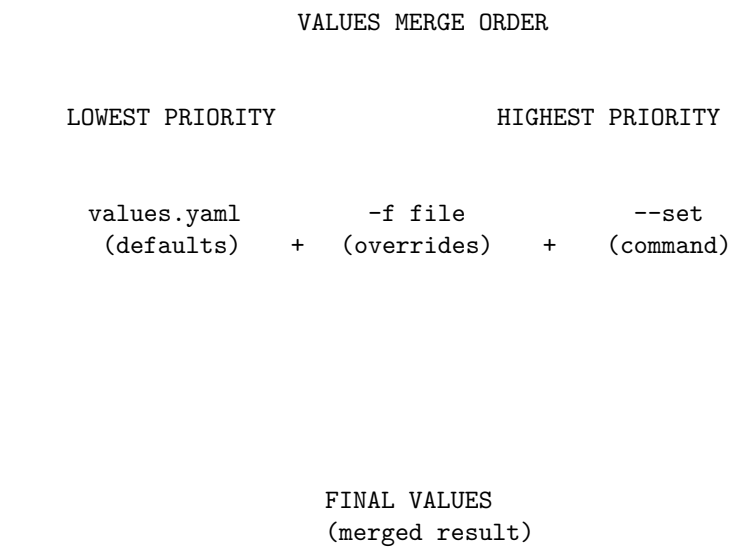
Expected output:

```
<h1>Version 2 - Upgraded!</h1>
```

The message is back to "Version 2"!

Step 12: Using Values Files

Let's install a new release using a values file (best practice).



Create a production values file:

```
cat > prod-values.yaml << 'EOF'
replicaCount: 3

welcomeMessage: "Production Environment"

service:
  type: NodePort

resources:
  limits:
    cpu: 200m
    memory: 256Mi
  requests:
    cpu: 100m
    memory: 128Mi
EOF
```

Install with the values file:

```
helm install prod-webapp my-webapp -f prod-values.yaml
```

Verify:

```
helm list
kubectl get pods | grep prod
```

You should see: - Two releases: webapp-v1 and prod-webapp - 3 pods for prod-webapp (replicaCount: 3)

Step 11: Check What Values Were Used

View user-supplied values for prod-webapp:

```
helm get values prod-webapp
```

Expected output:

```
USER-SUPPLIED VALUES:
replicaCount: 3
resources:
  limits:
    cpu: 200m
    memory: 256Mi
  requests:
    cpu: 100m
    memory: 128Mi
service:
  type: NodePort
welcomeMessage: Production Environment
```

This shows exactly what you provided via the values file.

Step 12: Upgrade with –reuse-values

Important concept: By default, upgrades reset to chart defaults.

Let's see this:

```
# Current: prod-webapp has replicaCount=3

# Upgrade without --reuse-values (WRONG WAY)
helm upgrade prod-webapp my-webapp --set image.tag="1.24"

# Check pods
kubectl get pods | grep prod
```

Problem: You might see only 1 pod because replicaCount reset to default (1)!

Fix it with –reuse-values:

```
helm upgrade prod-webapp my-webapp --reuse-values --set image.tag="1.25"
```

Now replicaCount stays at 3.

Better approach: Always use a values file:

```
helm upgrade prod-webapp my-webapp -f prod-values.yaml
```

Step 13: The upgrade –install Pattern

This pattern is great for CI/CD: install if new, upgrade if exists.

Test with a new release:

```
helm upgrade --install staging-webapp my-webapp \
  --set welcomeMessage="Staging Environment" \
  --set replicaCount=2
```

Output indicates install:

```
Release "staging-webapp" does not exist. Installing it now.
NAME: staging-webapp
...
```

Run the same command again:

```
helm upgrade --install staging-webapp my-webapp \
  --set welcomeMessage="Staging Environment - Updated" \
  --set replicaCount=2
```

Output indicates upgrade:

```
Release "staging-webapp" has been upgraded.
...
```

This pattern handles both cases automatically!

Step 14: List All Releases

See all releases:

```
helm list
```

Expected output:

```
NAME            NAMESPACE  REVISION  STATUS    CHART
prod-webapp     default    3         deployed  my-webapp-0.1.0
staging-webapp  default    2         deployed  my-webapp-0.1.0
webapp-v1       default    4         deployed  my-webapp-0.1.0
```

You have 3 releases, each with its own configuration.

Step 15: Clean Up All Releases

Uninstall all three releases:

```
helm uninstall webapp-v1
helm uninstall prod-webapp
helm uninstall staging-webapp
```

Verify everything is gone:

```
helm list
kubectl get pods
```

Both should be empty.

Clean up the values file:

```
rm prod-values.yaml
```

Expected Results Summary

Step	Action	Expected Result
Install	`helm install webapp-v1 my-webapp`	REVISION: 1, STATUS: deployed
Upgrade	`helm upgrade webapp-v1 --set ...`	REVISION: 2
History	`helm history webapp-v1`	Shows all revisions
Rollback	`helm rollback webapp-v1 2`	Creates new revision (4) with v2 values
Values file	`helm install -f prod-values.yaml`	Values from file used
upgrade –install	`helm upgrade --install`	Install or upgrade as needed
Uninstall	`helm uninstall`	Release removed

Troubleshooting

Problem: "cannot re-use a name that is still in use"

Cause: Trying to install a release name that already exists.

Solution:

```
# Either uninstall first
helm uninstall existing-release

# Or use upgrade --install
helm upgrade --install existing-release ./mychart
```

Problem: Upgrade resets my values

Cause: Default behavior resets to chart defaults.

Solution:

```
# Use --reuse-values
helm upgrade my-app ./mychart --reuse-values --set newKey=newValue

# Or always use a values file
helm upgrade my-app ./mychart -f my-values.yaml
```

Problem: Rollback doesn't seem to work

Check: Make sure you're specifying the correct revision number.

```
# View history first
helm history my-app

# Then rollback to the specific revision
helm rollback my-app <revision-number>
```

Problem: "Error: UPGRADE FAILED: another operation is in progress"

Cause: Previous operation didn't complete cleanly.

Solution:

```
# Check the release status
helm status my-app

# If stuck, you may need to force the operation
helm upgrade my-app ./mychart --force
```

Key Takeaways

1. **`helm install`** creates a new release with revision 1
2. **`helm upgrade`** creates new revisions
3. **`helm history`** shows all revisions
4. **`helm rollback`** creates a NEW revision with old content
5. **Use values files** for production deployments
6. **Use `--reuse-values`** if upgrading with –set
7. **Use `upgrade --install`** in CI/CD pipelines
8. **`helm uninstall`** removes the release and resources

Commands Reference

```
# Install
helm install <name> <chart>
helm install <name> <chart> -f values.yaml
helm install <name> <chart> --set key=value

# Upgrade
helm upgrade <name> <chart>
helm upgrade <name> <chart> --reuse-values
helm upgrade --install <name> <chart>

# Information
helm list
helm status <name>
helm history <name>
helm get values <name>
helm get manifest <name>

# Rollback
helm rollback <name> <revision>

# Uninstall
helm uninstall <name>
helm uninstall <name> --keep-history
```

Excellent! You now know how to manage the full lifecycle of Helm releases! # 05 – Repositories & Namespaces (Theory)

Estimated reading time: 15 minutes

Part 1: Helm Repositories

What is a Helm Repository?

The Simple Explanation

Think of a Helm repository like an **app store** for Kubernetes:

- **Apple App Store** has thousands of apps you can download
- **Helm Repository** has hundreds of charts you can install

Just like the App Store: - Someone maintains the repository - Apps/Charts are versioned - You search, download, and install

---

The Technical Definition

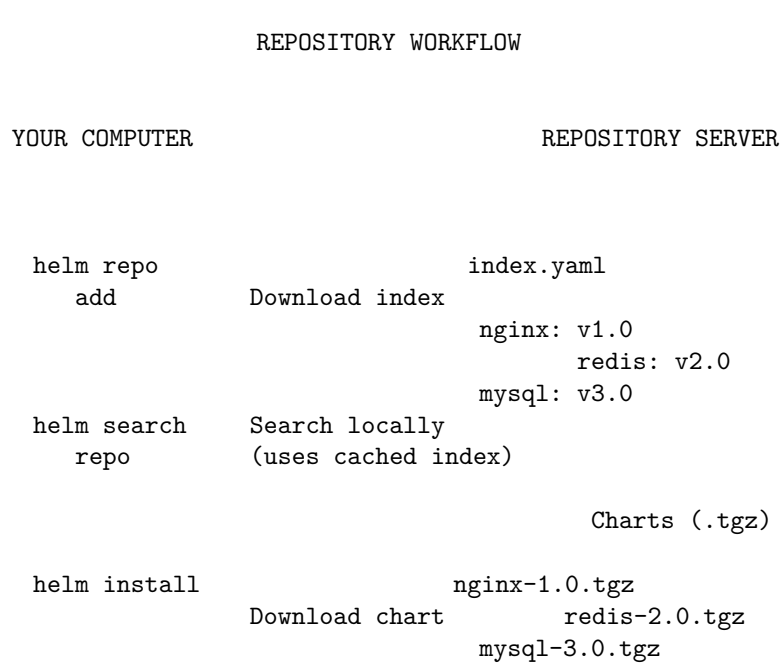
A Helm repository is an **HTTP server** that hosts: 1. An `index.yaml` file listing all available charts and versions 2. Packaged chart files (`.tgz` archives)

When you run `helm repo add`, you're telling Helm where to look for charts.

Repository Types

Type	Description	Example
Public	Open to everyone	Bitnami, Artifact Hub
Private	Company-internal	Your corporate Helm repo
OCI Registry	Container registry	Docker Hub, AWS ECR, Azure ACR

How Repositories Work



The index.yaml File

The `index.yaml` is the catalog of a repository:

```yaml
apiVersion: v1
entries:
  nginx:
    - name: nginx
      version: 15.0.0
      appVersion: "1.25.0"
      description: NGINX Open Source is a web server...
      urls:
        - https://charts.bitnami.com/bitnami/nginx-15.0.0.tgz
      digest: sha256:abc123...
    - name: nginx
      version: 14.2.0
      appVersion: "1.24.0"
      ...
  redis:
    - name: redis
      version: 18.0.0
      ...
generated: "2024-01-15T10:00:00Z"
```

Key fields: - `entries`: Map of chart-name → list of versions - `urls`: Where to download the chart - `digest`: Checksum for verification - `generated`: When the index was last updated

---

Repository Commands

Command	Description
`helm repo add <name> <url>`	Add a repository
`helm repo list`	Show configured repositories
`helm repo update`	Download latest index files
`helm repo remove <name>`	Remove a repository

Finding Charts: Artifact Hub

Artifact Hub (https://artifacthub.io) is like Google for Helm charts:

- Search across many repositories
- View chart documentation
- See version history
- Check security reports
- Find installation instructions

Before adding random repositories, search Artifact Hub to find official/verified charts.

Popular Public Repositories

Repository	URL	Description
Bitnami	`https://charts.bitnami.com/bitnami`	High-quality, production-ready
Prometheus	`https://prometheus-community.github.io/helm-charts`	Monitoring
Grafana	`https://grafana.github.io/helm-charts`	Visualization

Repository	URL	Description
Jetstack	`https://charts.jetstack.io`	cert-manager
Ingress-nginx	`https://kubernetes.github.io/ingress-nginx`	Ingress controller

OCI Registries (Modern Approach)

Helm 3 supports storing charts in OCI (container) registries:

```
# Login to registry
helm registry login registry.example.com

# Push a chart
helm push mychart-0.1.0.tgz oci://registry.example.com/charts

# Install from OCI
helm install my-release oci://registry.example.com/charts/mychart --version 0.1.0
```

Benefits of OCI: - Use existing container registry infrastructure - Single registry for images AND charts - Better access control - Immutable versions

---

Part 2: Kubernetes Namespaces

What is a Namespace?

The Simple Explanation

Think of namespaces like **folders on your computer**:

- You can have a file called `report.txt` in both `/work/` and `/personal/`
- They don't conflict because they're in different folders
- You can delete everything in `/personal/` without affecting `/work/`

In Kubernetes: - You can have a release called `nginx` in both `dev` and `prod` namespaces - They don't conflict because they're in different namespaces - You can delete everything in `dev` without affecting `prod`

The Technical Definition

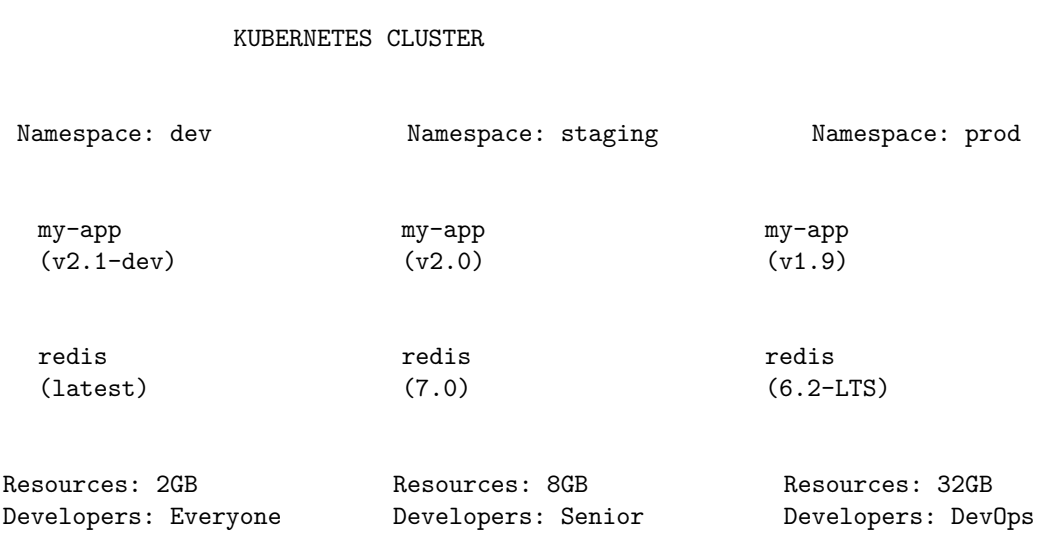
A **namespace** is a virtual cluster within a physical cluster. It provides: - Isolation between environments/teams - Resource quotas and limits - Network policies - RBAC (who can do what)

Default Namespaces

Every Kubernetes cluster starts with these:

Namespace	Purpose
`default`	Default for resources with no namespace specified
`kube-system`	Kubernetes system components
`kube-public`	Publicly readable resources
`kube-node-lease`	Node heartbeats

Why Use Namespaces?



Benefits shown: 1. Same names (my-app, redis) in different environments 2. Different versions per environment 3. Different resource limits 4. Different access permissions

---

Helm and Namespaces

Helm releases are **namespace-scoped**. The same release name can exist in different namespaces.

Installing to a specific namespace:

```
# Install to 'dev' namespace (create it if needed)
helm install my-app ./mychart --namespace dev --create-namespace

# Install to 'prod' namespace
helm install my-app ./mychart --namespace prod --create-namespace
```

Listing releases by namespace:

```
# Current namespace only
helm list

# Specific namespace
helm list --namespace prod

# All namespaces
helm list --all-namespaces
```

Namespace Best Practices

1. Environment-based Namespaces

```
development
staging
production
```

Helm commands:

```
helm install my-app ./mychart -n development --create-namespace
helm install my-app ./mychart -n staging --create-namespace
helm install my-app ./mychart -n production --create-namespace
```

2. Team-based Namespaces

```
team-frontend
team-backend
team-data
```

3. Application-based Namespaces

```
app-payment
app-inventory
app-shipping
```

Cross-Namespace Communication

Services in one namespace can reach services in another using **full DNS names**:

`<service-name>.<namespace>.svc.cluster.local`

Example: - In `dev` namespace: Service `redis` - In `staging` namespace: Service `my-app`

From `my-app` in staging, connect to Redis in dev:

`redis.dev.svc.cluster.local:6379`

Setting Default Namespace

Instead of typing `--namespace` every time:

Temporary (current shell):

```
export HELM_NAMESPACE=dev
helm install my-app ./mychart  # Uses 'dev'
```

kubectl context (persistent):

```
kubectl config set-context --current --namespace=dev
helm install my-app ./mychart  # Uses 'dev'
```

Part 3: Best Practices

Repository Management

1. **Use official repositories when available**
   - Bitnami, Prometheus-community, etc.
   - Check Artifact Hub for verified publishers
2. **Keep repositories updated**

   `helm repo update`

   Run this regularly, especially before installing charts.
3. **Consider OCI for production**
   - Better security
   - Integrated with container workflow
   - Immutable versions
4. **Version pin your chart dependencies**

```yaml
# Chart.yaml
dependencies:
  - name: redis
    version: "18.0.0"  # Pin exact version
    repository: https://charts.bitnami.com/bitnami
```

Namespace Strategy

1. **Always specify namespace**

   ```
   helm install my-app ./mychart --namespace myns
   ```

   Don't rely on default namespace.

2. **Use –create-namespace**

   ```
   helm install my-app ./mychart -n myns --create-namespace
   ```

   Especially in CI/CD where namespace might not exist.

3. **Match namespace to environment**

   ```
   # Environment-specific values
   helm install my-app ./mychart \
     -n production \
     -f values-production.yaml
   ```

4. **Use Resource Quotas** Prevent one team/app from consuming all resources:

   ```yaml
   apiVersion: v1
   kind: ResourceQuota
   metadata:
     name: compute-quota
     namespace: dev
   spec:
     hard:
       requests.cpu: "10"
       requests.memory: 20Gi
       limits.cpu: "20"
       limits.memory: 40Gi
   ```

Security Considerations

1. **Use RBAC** Limit who can deploy to which namespaces.
2. **Network Policies** Control traffic between namespaces.
3. **Chart Verification**

   ```
   # Verify chart signature (if available)
   helm verify mychart-0.1.0.tgz
   ```

4. **Private Repositories** For proprietary charts, use private repos with authentication.

Part 4: Hosting Your Own Chart Repository

Why Host Your Own Repository?

- **Share charts** with your team or organization
- **Private charts** that shouldn't be public

- **CI/CD integration** for automated chart publishing
- **Version control** over what charts are available

Repository Structure

A Helm repository is surprisingly simple. It's just an HTTP server with:

```
charts/
  index.yaml           # Catalog of all charts
  nginx-1.0.0.tgz      # Packaged chart
  nginx-1.0.0.tgz.prov # Optional: Signature file
  mysql-2.0.0.tgz
  myapp-3.0.0.tgz
```

The index.yaml File

The `index.yaml` is the heart of a repository. Helm downloads this file to know what charts are available:

```yaml
apiVersion: v1
entries:
  nginx:
    - name: nginx
      version: 1.0.0
      appVersion: "1.25.0"
      description: NGINX web server
      urls:
        - https://my-repo.example.com/charts/nginx-1.0.0.tgz
      digest: sha256:abc123...
      created: "2024-01-15T10:00:00Z"
  mysql:
    - name: mysql
      version: 2.0.0
      ...
generated: "2024-01-15T10:00:00Z"
```

Packaging Charts

Before uploading, you must package your chart:

```
# Package a chart
helm package ./mychart

# Output: mychart-0.1.0.tgz
```

Generating index.yaml

```
# In your charts directory
helm repo index .

# Or with a URL prefix
helm repo index . --url https://my-repo.example.com/charts
```

Repository Hosting Options

Option	Complexity	Best For
Static web server	Low	Simple setups
GitHub Pages	Low	Open source projects
ChartMuseum	Medium	Teams needing API
Cloud storage	Medium	S3, GCS, Azure Blob

Option	Complexity	Best For
OCI Registry	Medium	Container-centric teams
Harbor	High	Enterprise features

ChartMuseum: Self-Hosted Repository

ChartMuseum is a popular open-source Helm chart repository with: - Simple API for uploading charts - Multiple storage backends (local, S3, GCS, Azure) - Multi-tenancy support - Basic authentication

Quick Start with Helm:

```
# Add ChartMuseum chart repo
helm repo add chartmuseum https://chartmuseum.github.io/charts

# Install ChartMuseum
helm install chartmuseum chartmuseum/chartmuseum \
  --namespace tools \
  --set env.open.STORAGE=local \
  --set env.open.DISABLE_API=false \
  --set persistence.enabled=true
```

Upload a chart to ChartMuseum:

```
# Using curl
curl --data-binary "@mychart-0.1.0.tgz" http://localhost:8080/api/charts

# Using helm-push plugin
helm plugin install https://github.com/chartmuseum/helm-push
helm push mychart-0.1.0.tgz my-repo
```

GitHub Pages Repository

Free hosting for public charts:

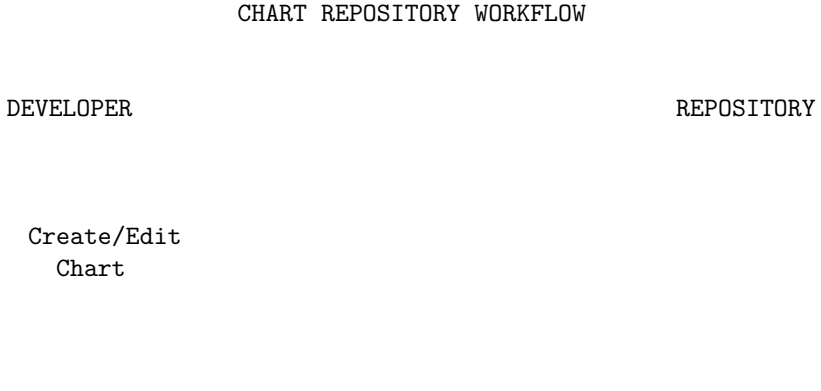
1. Create a GitHub repo with a `charts/` folder
2. Package and add charts to the folder
3. Generate `index.yaml`:

   ```
   helm repo index charts/ --url https://username.github.io/repo-name/charts
   ```

4. Enable GitHub Pages for the repo
5. Add the repo:

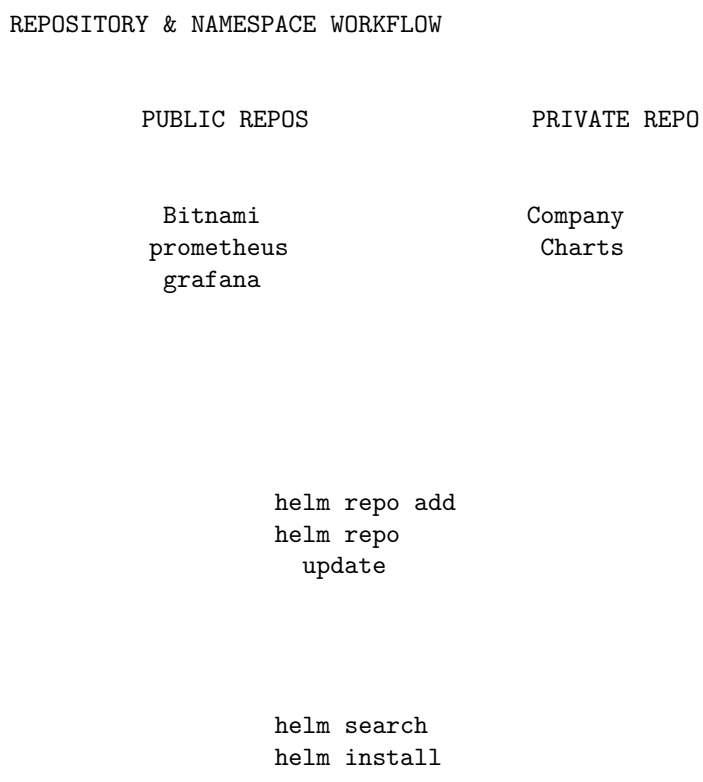
   ```
   helm repo add my-charts https://username.github.io/repo-name/charts
   ```

Repository Workflow Diagram



```
 helm lint      Validate chart


helm package    Create .tgz


 helm push                    ChartMuseum
 (or curl)                              S3 / GCS
                                  GitHub Pages

USER


helm repo add


helm install    Deploy chart
 repo/chart
```

Visual: Repository + Namespace Workflow



```
Namespace: dev          Namespace: stg          Namespace: prod


my-app (v2.1)           my-app (v2.0)           my-app (v1.9)
redis                   redis                   redis
prometheus              prometheus              prometheus
```

Key Takeaways

Repositories

1. **Repository = Chart store** – HTTP server with index.yaml and .tgz files
2. **`helm repo add`** – Add a repository to your local config
3. **`helm repo update`** – Refresh the chart catalog
4. **Artifact Hub** – Search for charts across all repositories
5. **OCI support** – Modern way using container registries

Namespaces

1. **Namespace = Virtual cluster** – Isolation for environments/teams
2. **Always specify namespace** – Don't rely on default
3. **Use `--create-namespace`** – For CI/CD pipelines
4. **Same release name OK** – In different namespaces
5. **Cross-namespace DNS** – `<service>.<namespace>.svc.cluster.local`

Next: Hands-on with repositories and namespaces! # 05 – Repositories & Namespaces (Lab)

Objective: Work with multiple Helm repositories and deploy applications across different namespaces.

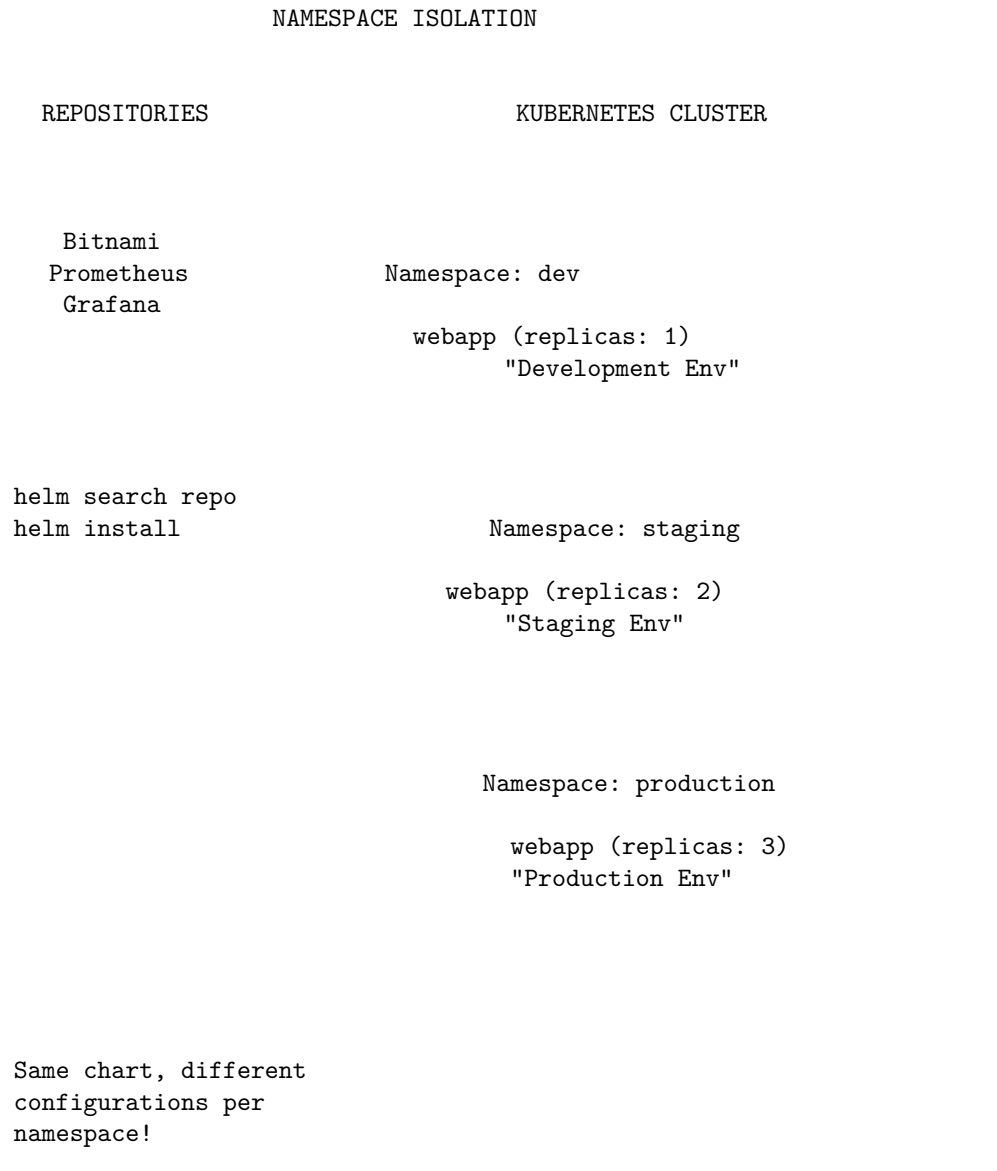
Estimated duration: 25–30 minutes

Prerequisites: Helm installed, Minikube running, my-webapp chart from Module 03.

What You Will Do

1. Explore existing repositories
2. Add and manage multiple repositories
3. Search for charts across repositories
4. Create and use namespaces
5. Deploy the same chart to multiple namespaces
6. Manage releases across namespaces
7. Clean up

Multi-Namespace Deployment Diagram



Step-by-Step Instructions

Step 1: View Current Repositories

List configured repositories:

```
helm repo list
```

Expected output:

```
NAME    URL
bitnami https://charts.bitnami.com/bitnami
```

If you don't see bitnami, add it:

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

Step 2: Add More Repositories

Let's add some popular repositories.

Add Prometheus community charts:

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
```

Expected output:

```
"prometheus-community" has been added to your repositories
```

Add Grafana charts:

```
helm repo add grafana https://grafana.github.io/helm-charts
```

Add ingress-nginx:

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
```

Verify all repositories:

```
helm repo list
```

Expected output:

```
NAME                    URL
bitnami                 https://charts.bitnami.com/bitnami
prometheus-community    https://prometheus-community.github.io/helm-charts
grafana                 https://grafana.github.io/helm-charts
ingress-nginx           https://kubernetes.github.io/ingress-nginx
```

You now have 4 repositories configured!

Step 3: Update All Repositories

Download the latest chart listings:

```
helm repo update
```

Expected output:

```
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "prometheus-community" chart repository
...Successfully got an update from the "grafana" chart repository
...Successfully got an update from the "ingress-nginx" chart repository
...Successfully got an update from the "bitnami" chart repository
Update Complete. Happy Helming!
```

Step 4: Search Across Repositories

Search for monitoring-related charts:

```
helm search repo prometheus
```

Expected output (partial):

```
NAME                                    CHART VERSION   APP VERSION   DESCRIPTION
prometheus-community/prometheus         25.0.0          v2.47.0       Prometheus is a monitoring system...
prometheus-community/kube-prometheus-stack   51.0.0     v0.68.0       kube-prometheus-stack collects...
bitnami/prometheus                      0.3.0           2.47.0        Prometheus is an open source...
```

Notice: Multiple repositories have prometheus charts. The name format is `<repo-name>/<chart-name>`.

Search for grafana:

```
helm search repo grafana
```

Search with version filter:

```
helm search repo prometheus --versions | head -n 10
```

This shows all available versions of each chart.

Step 5: Get Chart Information

View chart details:

```
helm show chart prometheus-community/prometheus
```

Expected output:

```
apiVersion: v2
appVersion: v2.47.0
description: Prometheus is a monitoring system and time series database.
name: prometheus
version: 25.0.0
...
```

View default values:

```
helm show values prometheus-community/prometheus | head -n 50
```

View README (full documentation):

```
helm show readme prometheus-community/prometheus | head -n 100
```

Step 6: Create Namespaces

Let's create namespaces for different environments.

Create dev namespace:

```
kubectl create namespace dev
```

Expected output:

```
namespace/dev created
```

Create staging namespace:

```
kubectl create namespace staging
```

Create production namespace:

```
kubectl create namespace production
```

Verify namespaces:

```
kubectl get namespaces
```

Expected output:

```
NAME              STATUS   AGE
default           Active   1h
dev               Active   30s
kube-node-lease   Active   1h
kube-public       Active   1h
kube-system       Active   1h
production        Active   10s
staging           Active   20s
```

Step 7: Deploy to Dev Namespace

Let's deploy our webapp to the dev namespace.

Navigate to your chart:

```
cd ~/helm-training
```

Install to dev namespace:

```
helm install webapp my-webapp \
  --namespace dev \
  --set welcomeMessage="Development Environment" \
  --set replicaCount=1
```

Expected output:

```
NAME: webapp
LAST DEPLOYED: Thu Jan 15 15:00:00 2024
NAMESPACE: dev
STATUS: deployed
REVISION: 1
...
```

Step 8: Deploy to Staging Namespace

Install to staging namespace:

```
helm install webapp my-webapp \
  --namespace staging \
  --set welcomeMessage="Staging Environment" \
  --set replicaCount=2
```

Expected output:

```
NAME: webapp
NAMESPACE: staging
STATUS: deployed
REVISION: 1
...
```

Notice: Same release name "webapp" works in different namespaces!

Step 9: Deploy to Production Namespace

Install to production namespace:

```
helm install webapp my-webapp \
  --namespace production \
  --set welcomeMessage="Production Environment - Handle with Care!" \
  --set replicaCount=3 \
  --set service.type=NodePort
```

Step 10: List Releases Across Namespaces

List releases in current namespace (default):

```
helm list
```

Expected output:

```
NAME    NAMESPACE   REVISION    ...
(empty - no releases in default namespace)
```

List releases in dev namespace:

```
helm list --namespace dev
```

Expected output:

```
NAME    NAMESPACE   REVISION   STATUS     CHART           APP VERSION
webapp  dev         1          deployed   my-webapp-0.1.0 1.25.0
```

List releases in ALL namespaces:

```
helm list --all-namespaces
```

Expected output:

```
NAME    NAMESPACE    REVISION   STATUS     CHART           APP VERSION
webapp  dev          1          deployed   my-webapp-0.1.0 1.25.0
webapp  staging      1          deployed   my-webapp-0.1.0 1.25.0
webapp  production   1          deployed   my-webapp-0.1.0 1.25.0
```

Three releases, same name, different namespaces!

Step 11: Verify Pods Across Namespaces

Check pods in all namespaces:

```
kubectl get pods --all-namespaces | grep webapp
```

Expected output:

```
dev          webapp-my-webapp-xxxxx    1/1   Running   0   2m
production   webapp-my-webapp-aaaaa    1/1   Running   0   1m
production   webapp-my-webapp-bbbbb    1/1   Running   0   1m
production   webapp-my-webapp-ccccc    1/1   Running   0   1m
staging      webapp-my-webapp-yyyyy    1/1   Running   0   2m
staging      webapp-my-webapp-zzzzz    1/1   Running   0   2m
```

Notice: Dev has 1 pod, Staging has 2, Production has 3 (matching our replicaCount settings).

Step 12: Test Each Environment

Test Dev environment:

```
kubectl port-forward svc/webapp-my-webapp 8081:80 --namespace dev &
sleep 2
curl -s http://localhost:8081 | grep -o "<h1>.*</h1>"
kill %1 2>/dev/null || true
```

Expected output:

```
<h1>Development Environment</h1>
```

Test Staging environment:

```
kubectl port-forward svc/webapp-my-webapp 8082:80 --namespace staging &
sleep 2
curl -s http://localhost:8082 | grep -o "<h1>.*</h1>"
kill %1 2>/dev/null || true
```

Expected output:

```
<h1>Staging Environment</h1>
```

Test Production environment (uses NodePort):

```
minikube service webapp-my-webapp --namespace production --url
```

Copy the URL and open in browser, or:

```
URL=$(minikube service webapp-my-webapp --namespace production --url)
curl -s "$URL" | grep -o "<h1>.*</h1>"
```

Expected output:

```
<h1>Production Environment - Handle with Care!</h1>
```

Step 13: Upgrade in One Namespace

Let's upgrade only the dev release:

```
helm upgrade webapp my-webapp \
  --namespace dev \
  --set welcomeMessage="Development v2 - Updated!" \
  --set replicaCount=1
```

Check that only dev was upgraded:

```
helm list --all-namespaces
```

Expected output:

```
NAME    NAMESPACE    REVISION    ...
webapp  dev          2           ...    # Revision 2!
webapp  staging      1           ...    # Still Revision 1
webapp  production   1           ...    # Still Revision 1
```

Only dev was upgraded to revision 2.

Step 14: View Release History by Namespace

Dev history:

```
helm history webapp --namespace dev
```

Expected output:

```
REVISION   STATUS       DESCRIPTION
1          superseded   Install complete
2          deployed     Upgrade complete
```

Production history:

```
helm history webapp --namespace production
```

Expected output:

```
REVISION   STATUS     DESCRIPTION
1          deployed   Install complete
```

Step 15: Set Default Namespace (Optional)

To avoid typing `--namespace` every time:

Using kubectl:

```
kubectl config set-context --current --namespace=dev
```

Now helm commands use dev by default:

```
helm list   # Shows dev namespace
```

Reset to default namespace:

```
kubectl config set-context --current --namespace=default
```

Step 16: Remove a Repository

Let's remove a repository we don't need:

```
helm repo remove ingress-nginx
```

Expected output:

```
"ingress-nginx" has been removed from your repositories
```

Verify:

```
helm repo list
```

The ingress-nginx repository is gone.

Step 17: Clean Up

Uninstall all webapp releases:

```
helm uninstall webapp --namespace dev
helm uninstall webapp --namespace staging
helm uninstall webapp --namespace production
```

Delete the namespaces:

```
kubectl delete namespace dev
kubectl delete namespace staging
kubectl delete namespace production
```

Verify cleanup:

```
helm list --all-namespaces
kubectl get namespaces | grep -E "(dev|staging|production)"
```

Both should return empty.

Keep repositories for future use (or remove them):

```
# Optional: Remove all added repos except bitnami
helm repo remove prometheus-community
helm repo remove grafana
```

Expected Results Summary

Step	Action	Expected Result
Add repos	`helm repo add ...`	Repository added
Update repos	`helm repo update`	Latest indexes downloaded
Search	`helm search repo prometheus`	Charts from multiple repos
Create namespaces	`kubectl create namespace dev`	Namespaces created

Step	Action	Expected Result
Install to namespace	`helm install webapp my-webapp -n dev`	Release in dev namespace
List all	`helm list --all-namespaces`	Shows all releases
Upgrade one	`helm upgrade ... -n dev`	Only dev updated
Uninstall	`helm uninstall webapp -n dev`	Release removed from dev

Troubleshooting

Problem: "namespace not found"

Solution: Create the namespace first, or use `--create-namespace`:

```
helm install webapp my-webapp \
  --namespace myns \
  --create-namespace
```

Problem: "release already exists"

Cause: A release with that name exists in that namespace.

Check existing releases:

```
helm list --namespace myns
```

Solutions:

```
# Uninstall existing release
helm uninstall webapp --namespace myns

# Or use a different release name
helm install webapp-v2 my-webapp --namespace myns
```

Problem: "Error: repo not found"

Cause: Repository wasn't added or was removed.

Solution:

```
# List current repos
helm repo list

# Add the missing repo
helm repo add myrepo https://...

# Update
helm repo update
```

Problem: Can't find chart after adding repo

Solution:

```
# Make sure to update after adding
helm repo update

# Search again
helm search repo chartname
```

Commands Reference

Repository Commands

```
helm repo add <name> <url>      # Add repository
helm repo list                  # List repositories
helm repo update                # Update all repos
helm repo remove <name>         # Remove repository
helm search repo <keyword>      # Search charts
helm search repo <chart> --versions  # See all versions
helm show chart <repo>/<chart>  # View chart metadata
helm show values <repo>/<chart> # View default values
```

Namespace Commands

```
kubectl create namespace <name>     # Create namespace
kubectl get namespaces              # List namespaces
kubectl delete namespace <name>     # Delete namespace

# Helm with namespaces
helm install ... --namespace <ns>              # Install to namespace
helm install ... -n <ns> --create-namespace    # Create ns if needed
helm list --namespace <ns>                     # List in namespace
helm list --all-namespaces                     # List all
helm upgrade ... --namespace <ns>              # Upgrade in namespace
helm uninstall <name> --namespace <ns>         # Uninstall from namespace
```

Key Takeaways

1. **Multiple repositories** give you access to more charts
2. **`helm repo update`** downloads latest chart versions
3. **Search format** is **`<repo-name>/<chart-name>`**
4. **Namespaces** provide isolation between environments
5. **Same release name** can exist in different namespaces
6. **Always specify namespace** with `--namespace` or `-n`
7. **Use `--create-namespace`** in CI/CD pipelines
8. **`helm list --all-namespaces`** shows everything

Great work! You can now manage repositories and deploy across namespaces! # 06 – Testing & Debugging Charts (Theory)

Estimated reading time: 15 minutes

Why Test and Debug?

The Simple Explanation

Imagine building a house: - Would you wait until it's completely built to check if the walls are straight? - No! You check at every step.

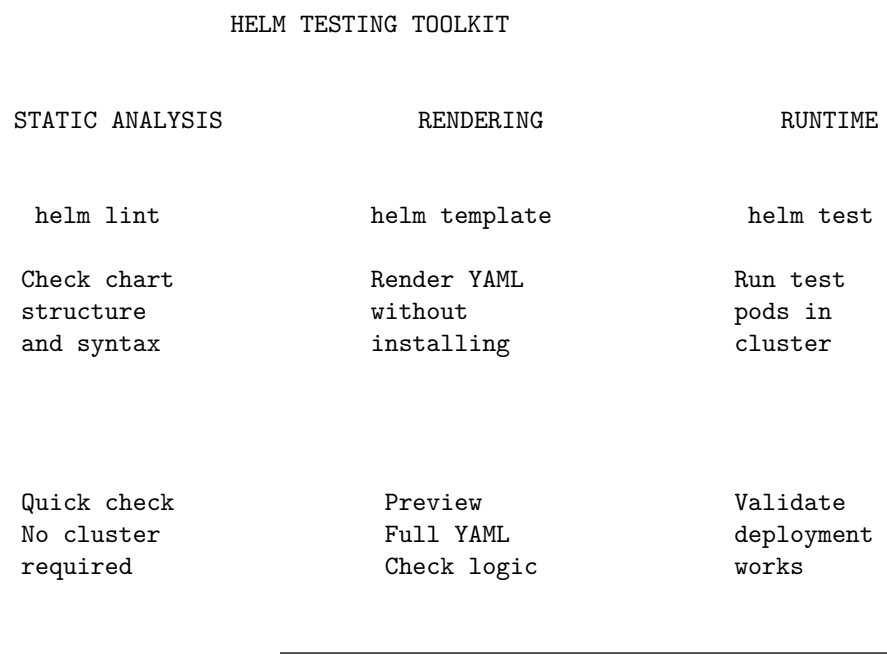
Helm charts are the same: - Test early and often - Catch mistakes before they affect users - Debug problems when they occur

---

The Technical Reasons

1. **Prevent production issues** – Catch configuration errors before deployment
2. **Save time** – Fix problems in development, not at 2 AM
3. **Improve confidence** – Know your charts work before releasing
4. **Enable CI/CD** – Automated testing for chart changes

Testing Tools Overview



Tool 1: helm lint

What It Does

`helm lint` checks your chart for: - Valid chart structure - Required files exist - YAML syntax errors - Best practice violations

Usage

```
helm lint <chart-path>
```

Example

```
helm lint ./my-webapp
```

Possible outputs:

Good chart:

```
==> Linting ./my-webapp
[INFO] Chart.yaml: icon is recommended

1 chart(s) linted, 0 chart(s) failed
```

Chart with errors:

```
==> Linting ./my-webapp
[ERROR] Chart.yaml: version is required
[ERROR] templates/deployment.yaml: unable to parse YAML: ...
[WARNING] templates/service.yaml: object name does not conform to Kubernetes naming requirements

1 chart(s) linted, 1 chart(s) failed
```

Severity Levels

Level	Meaning	Action
`ERROR`	Critical issue, chart won't work	Must fix
`WARNING`	Best practice violation	Should fix
`INFO`	Suggestion for improvement	Optional

Lint with Values

Test with specific values:

```
helm lint ./my-webapp --values prod-values.yaml
```

This validates the chart with your production configuration.

Tool 2: helm template

What It Does

`helm template` renders your templates to YAML without installing anything.

This lets you: - Preview exact output - Verify values substitution - Check conditional logic - Debug template issues

Usage

```
helm template <release-name> <chart-path>
```

Example

```
helm template my-release ./my-webapp
```

Output:

```
---
# Source: my-webapp/templates/configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-release-my-webapp-html
...
---
# Source: my-webapp/templates/service.yaml
```

```
apiVersion: v1
kind: Service
...
---
# Source: my-webapp/templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
...
```

Template with Values Override

```
helm template my-release ./my-webapp --set replicaCount=5
```

Verify that `replicas: 5` appears in the output.

Template Specific Files

Only render one template:

```
helm template my-release ./my-webapp --show-only templates/deployment.yaml
```

Validate Against Kubernetes

Check if the output is valid Kubernetes YAML:

```
helm template my-release ./my-webapp | kubectl apply --dry-run=client -f -
```

Expected output:

```
configmap/my-release-my-webapp-html created (dry run)
service/my-release-my-webapp created (dry run)
deployment.apps/my-release-my-webapp created (dry run)
```

Tool 3: –dry-run (Install/Upgrade)

What It Does

The `--dry-run` flag simulates an install/upgrade: - Connects to cluster - Validates configuration - Shows what would be deployed - Does NOT actually deploy

Usage

```
helm install my-release ./my-webapp --dry-run
```

Difference from helm template

Feature	`helm template`	`helm install --dry-run`
Requires cluster	No	Yes
Validates against API	No	Yes
Shows hooks	Limited	Full
Release name generation	No	Yes
Server-side validation	No	Yes

Use `helm template` for quick local testing. **Use `--dry-run`** for full validation before production deploy.

Debug Mode

Add `--debug` for extra information:

```
helm install my-release ./my-webapp --dry-run --debug
```

This shows: - Computed values - Rendered templates - Hook definitions - Full debug output

---

Tool 4: helm test

What It Does

`helm test` runs test pods defined in your chart to verify the installation works correctly.

How It Works

1. You define test pods in `templates/tests/`
2. These pods run checks (HTTP requests, connections, etc.)
3. If pods succeed (exit 0), test passes
4. If pods fail (exit non-0), test fails

Creating a Test

File: `templates/tests/test-connection.yaml`

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: "{{ include "my-webapp.fullname" . }}-test-connection"
  labels:
    {{- include "my-webapp.labels" . | nindent 4 }}
  annotations:
    "helm.sh/hook": test                    # This makes it a test
spec:
  containers:
    - name: wget
      image: busybox
      command: ['wget']
      args: ['{{ include "my-webapp.fullname" . }}:{{ .Values.service.port }}']
  restartPolicy: Never
```

Key parts: - `helm.sh/hook: test` – Marks this as a test pod - The container runs a command (wget in this case) - `restartPolicy: Never` – Don't restart on failure - If wget succeeds, test passes

Running Tests

```
# First install the chart
helm install my-release ./my-webapp

# Then run tests
helm test my-release
```

Expected output (success):

```
NAME: my-release
LAST DEPLOYED: Thu Jan 15 16:00:00 2024
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE:     my-release-my-webapp-test-connection
```

```
Last Started:   Thu Jan 15 16:00:05 2024
Last Completed: Thu Jan 15 16:00:10 2024
Phase:          Succeeded
```

Expected output (failure):

```
TEST SUITE:     my-release-my-webapp-test-connection
Last Started:   Thu Jan 15 16:00:05 2024
Last Completed: Thu Jan 15 16:00:15 2024
Phase:          Failed
Error: pod my-release-my-webapp-test-connection failed
```

Multiple Tests

You can have multiple test files:

```
templates/tests/
  test-connection.yaml    # Test HTTP connectivity
  test-database.yaml      # Test database connection
  test-config.yaml        # Test configuration loaded
```

Each runs separately and reports pass/fail.

Debugging Techniques

1. Debug Template Rendering

Use `{{ printf }}` for debugging:

```
# Add this temporarily to see a value
{{ printf "DEBUG: replicaCount is %v" .Values.replicaCount }}
```

Run `helm template` to see the output.

Remove before committing!

2. Check Template Logic

Test conditional logic:

```
{{- if .Values.ingress.enabled }}
# This will be rendered
{{- else }}
# This will NOT be rendered
{{- end }}
```

Run `helm template` with different values:

```
# Without ingress
helm template my-release ./my-webapp --set ingress.enabled=false

# With ingress
helm template my-release ./my-webapp --set ingress.enabled=true
```

3. Inspect Running Release

View the manifest (what was deployed):

```
helm get manifest my-release
```

View the values (what configuration was used):

```
helm get values my-release --all
```

View everything:

```
helm get all my-release
```

4. Check Kubernetes Resources

View pods and their status:

```
kubectl get pods -l app.kubernetes.io/instance=my-release
```

Describe a pod (detailed info + events):

```
kubectl describe pod <pod-name>
```

View logs:

```
kubectl logs <pod-name>
kubectl logs -l app.kubernetes.io/instance=my-release
```

5. Common Debugging Commands

```
# See all resources created by a release
kubectl get all -l app.kubernetes.io/instance=my-release

# Watch pods come up
kubectl get pods -w

# Get events (cluster-wide, useful for errors)
kubectl get events --sort-by='.lastTimestamp'

# Check if a value was set correctly
helm get values my-release | grep replicaCount
```

Common Problems and Solutions

Problem 1: YAML Indentation Error

Symptom:

```
Error: YAML parse error: ...
```

Cause: Incorrect whitespace in templates.

Solution: Use `nindent` for proper indentation:

```
# Wrong
labels:
{{ include "mychart.labels" . }}

# Correct
labels:
{{- include "mychart.labels" . | nindent 4 }}
```

Problem 2: Nil Pointer Error

Symptom:

```
Error: template: mychart/templates/deployment.yaml:10:18:
executing "mychart/templates/deployment.yaml" at <.Values.image.tag>:
nil pointer evaluating interface {}.tag
```

Cause: Trying to access a value that doesn't exist.

Solution: Use `default` or check if it exists:

```
# Use default
image: "{{ .Values.image.repository }}:{{ default "latest" .Values.image.tag }}"

# Or check with if
{{- if .Values.image }}
image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
{{- end }}
```

Problem 3: Resource Name Too Long

Symptom:

```
Error: release name "my-very-long-release-name-that-exceeds-limits" is invalid
```

Cause: Kubernetes names must be 63 characters.

Solution: Truncate names in helpers:

```
{{- define "mychart.fullname" -}}
{{- .Release.Name | trunc 63 | trimSuffix "-" }}
{{- end }}
```

Problem 4: Hook Failed

Symptom:

```
Error: pre-install hook failed: Job failed: BackoffLimitExceeded
```

Cause: A pre-install hook (like database migration) failed.

Solution: Check the hook job logs:

```
kubectl logs job/<release-name>-<hook-name>
```

Testing Best Practices

1. Always Lint Before Committing

```
# Add to your workflow
helm lint ./my-webapp
```

2. Template Before Installing

```
# Preview what will be deployed
helm template my-release ./my-webapp | less
```

3. Use –dry-run for Production Deployments

```
# Validate against production cluster
helm upgrade --install my-release ./my-webapp \
  --namespace production \
  -f production-values.yaml \
  --dry-run
```

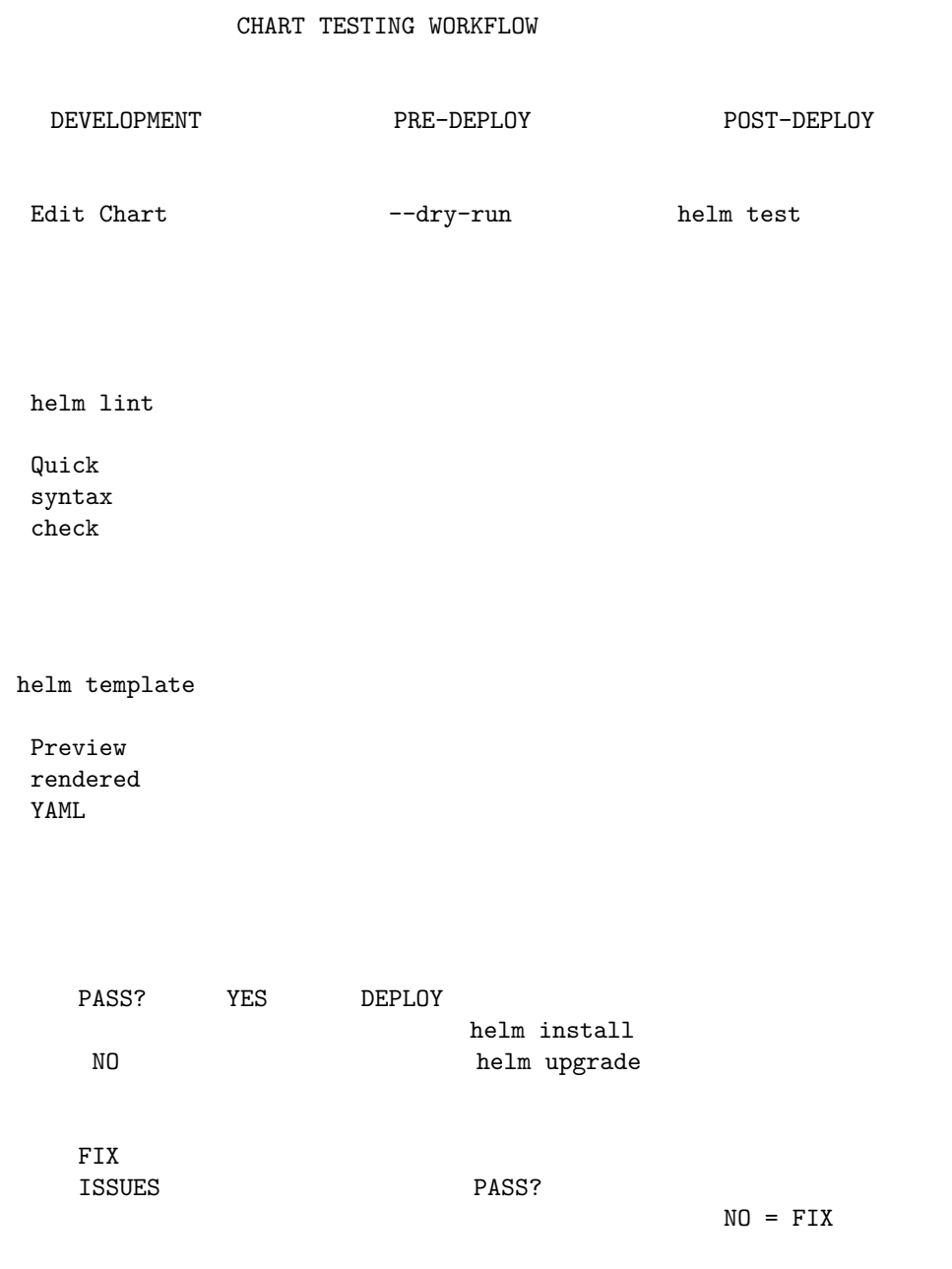
4. Write Test Pods

Create tests for critical functionality: - Connection tests - Configuration validation - Health endpoint checks

5. CI/CD Pipeline Testing

```
# Typical CI pipeline
helm lint ./my-webapp
helm template my-release ./my-webapp | kubectl apply --dry-run=client -f -
# Deploy to test namespace
helm install my-release ./my-webapp -n test
helm test my-release -n test
# Clean up
helm uninstall my-release -n test
```

Visual: Testing Workflow



Key Takeaways

1. **`helm lint`** – Quick syntax and structure check
2. **`helm template`** – Preview rendered YAML without cluster
3. **`--dry-run`** – Full validation with cluster connection
4. **`helm test`** – Runtime tests with test pods
5. **Debug with `--debug`** – See computed values and full output
6. **Inspect with `helm get`** – View manifest, values, notes
7. **Check Kubernetes** – Use kubectl logs, describe, get events
8. **Test early and often** – Catch problems before production

---

Next: Hands-on testing and debugging! # 06 – Testing & Debugging Charts (Lab)

Objective: Practice using Helm's testing and debugging tools to validate charts and troubleshoot issues.

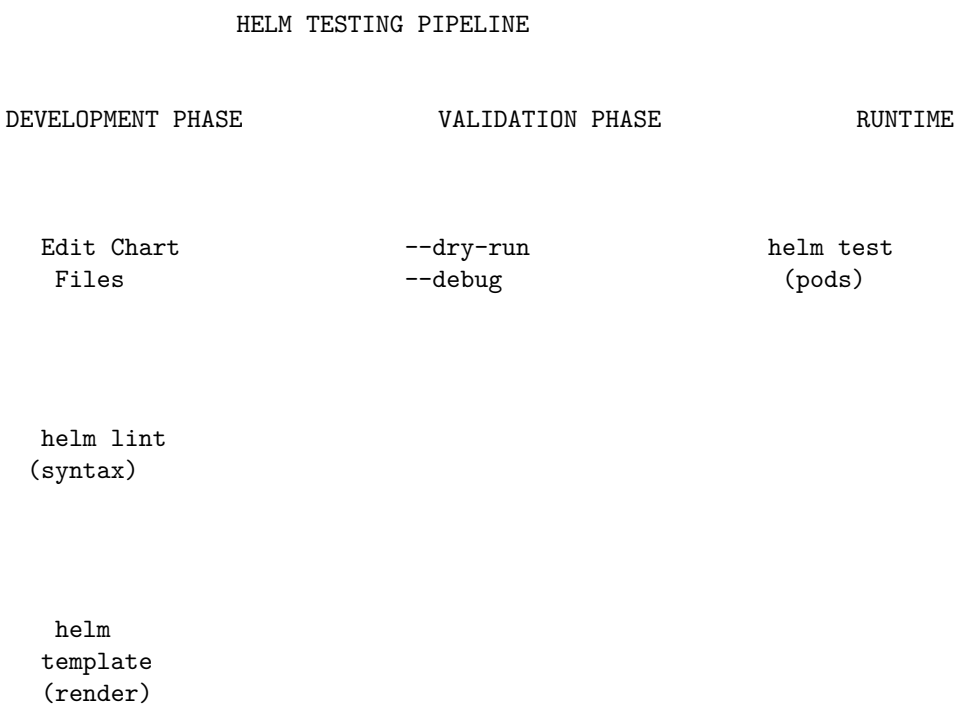
Estimated duration: 25–30 minutes

Prerequisites: Helm installed, Minikube running, my-webapp chart from Module 03.

What You Will Do

1. Use helm lint to check chart structure
2. Use helm template to preview rendered YAML
3. Use –dry-run to validate before installing
4. Create and run test pods
5. Debug intentional errors
6. Practice troubleshooting techniques

Testing Pipeline Diagram



```
PASS?      YES      DEPLOY      PASS?      DONE!

 NO                                      NO


FIX                      FIX
```

Preparation

Navigate to your chart:

```
cd ~/helm-training
ls my-webapp
```

Make sure you have the my-webapp chart from Module 03.

Part 1: Linting Charts

Step 1: Basic Lint

Run lint on your chart:

```
helm lint my-webapp
```

Expected output:

```
==> Linting my-webapp
[INFO] Chart.yaml: icon is recommended

1 chart(s) linted, 0 chart(s) failed
```

What this tells you: - Chart passed (0 failed) - INFO: icon is recommended (optional suggestion)

Step 2: Create a Lint Error (Intentional)

Let's break something to see lint catch it.

Make a backup and corrupt values.yaml:

```
cp my-webapp/values.yaml my-webapp/values.yaml.bak
echo "this is: not: valid: yaml: [" >> my-webapp/values.yaml
```

Run lint again:

```
helm lint my-webapp
```

Expected output:

```
==> Linting my-webapp
[ERROR] values.yaml: unable to parse YAML: yaml: line X: ...

Error: 1 chart(s) linted, 1 chart(s) failed
```

Lint caught the error!

Restore the file:

```
mv my-webapp/values.yaml.bak my-webapp/values.yaml
```

Step 3: Lint with Values File

Create a test values file:

```
cat > test-values.yaml << 'EOF'
replicaCount: 3
welcomeMessage: "Testing Lint"
service:
  type: NodePort
EOF
```

Lint with this values file:

```
helm lint my-webapp -f test-values.yaml
```

Expected output:

```
==> Linting my-webapp
[INFO] Chart.yaml: icon is recommended

1 chart(s) linted, 0 chart(s) failed
```

This validates that your chart works with specific value overrides.

Part 2: Template Rendering

Step 4: Basic Template Rendering

Render all templates:

```
helm template my-release my-webapp
```

This outputs all rendered YAML. Let's examine specific parts:

View only the deployment:

```
helm template my-release my-webapp --show-only templates/deployment.yaml
```

Expected output:

```
---
# Source: my-webapp/templates/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-release-my-webapp
...
spec:
  replicas: 1
...
```

Step 5: Template with Value Overrides

Change replica count:

```
helm template my-release my-webapp \
  --set replicaCount=5 \
  --show-only templates/deployment.yaml | grep "replicas:"
```

Expected output:

```
  replicas: 5
```

The value was correctly substituted!

Step 6: Verify ConfigMap Content

Check the welcome message:

```
helm template my-release my-webapp \
  --set welcomeMessage="Testing Template Render" \
  --show-only templates/configmap.yaml | grep -A 3 "index.html"
```

You should see your custom message in the HTML.

Step 7: Validate Against Kubernetes API

Send rendered YAML to kubectl for validation:

```
helm template my-release my-webapp | kubectl apply --dry-run=client -f -
```

Expected output:

```
configmap/my-release-my-webapp-html created (dry run)
service/my-release-my-webapp created (dry run)
deployment.apps/my-release-my-webapp created (dry run)
```

(dry run) means nothing was actually created - just validated.

Part 3: Dry Run Installation

Step 8: Dry Run with Debug

Run install with –dry-run and –debug:

```
helm install my-release my-webapp --dry-run --debug 2>&1 | head -n 50
```

What you'll see: 1. USER-SUPPLIED VALUES (your overrides) 2. COMPUTED VALUES (all merged values) 3. HOOKS (pre-install, post-install operations) 4. MANIFEST (rendered Kubernetes YAML)

Step 9: Dry Run with Values File

Use your test values:

```
helm install my-release my-webapp \
  -f test-values.yaml \
  --dry-run 2>&1 | head -n 30
```

Check computed values:

```
helm install my-release my-webapp \
  -f test-values.yaml \
  --dry-run --debug 2>&1 | grep -A 10 "COMPUTED VALUES"
```

You should see `replicaCount: 3` from your values file.

Part 4: Creating Tests

Step 10: Create Tests Directory

Create the tests folder:

```
mkdir -p my-webapp/templates/tests
```

Step 11: Create a Connection Test

```
                    HELM TEST POD FLOW


    helm test my-release



       Test Pod                Application
      (busybox)               Service
                      wget
      Exit Code:              Response: 200
        0 = PASS
        1 = FAIL




      Test Result

      Phase: Success
      or
      Phase: Failed
```

Create a test that checks if the service is reachable:

```
cat > my-webapp/templates/tests/test-connection.yaml << 'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: "{{ include "my-webapp.fullname" . }}-test-connection"
  labels:
    {{- include "my-webapp.labels" . | nindent 4 }}
  annotations:
    "helm.sh/hook": test
    "helm.sh/hook-delete-policy": hook-succeeded
spec:
  containers:
    - name: wget
      image: busybox:1.36
      command: ['sh', '-c']
      args:
        - |
          echo "Testing connection to {{ include "my-webapp.fullname" . }}:{{ .Values.service.port }}"
          wget --spider --timeout=5 {{ include "my-webapp.fullname" . }}:{{ .Values.service.port }}
          if [ $? -eq 0 ]; then
            echo "SUCCESS: Service is reachable"
```

```
            exit 0
          else
            echo "FAILURE: Service is not reachable"
            exit 1
          fi
  restartPolicy: Never
EOF
```

What this test does: 1. Runs as a pod with the `test` hook annotation 2. Uses wget to check if the service responds 3. Exits 0 (pass) if reachable, 1 (fail) if not 4. `hook-delete-policy: hook-succeeded` cleans up after success

---

Step 12: Create a Content Test

Create a test that verifies the welcome message appears:

```
cat > my-webapp/templates/tests/test-content.yaml << 'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: "{{ include "my-webapp.fullname" . }}-test-content"
  labels:
    {{- include "my-webapp.labels" . | nindent 4 }}
  annotations:
    "helm.sh/hook": test
    "helm.sh/hook-delete-policy": hook-succeeded
spec:
  containers:
    - name: curl
      image: curlimages/curl:8.4.0
      command: ['sh', '-c']
      args:
        - |
          echo "Checking content from {{ include "my-webapp.fullname" . }}"
          CONTENT=$(curl -s {{ include "my-webapp.fullname" . }}:{{ .Values.service.port }})
          if echo "$CONTENT" | grep -q "{{ .Values.welcomeMessage }}"; then
            echo "SUCCESS: Welcome message found"
            exit 0
          else
            echo "FAILURE: Welcome message not found"
            echo "Expected: {{ .Values.welcomeMessage }}"
            echo "Got: $CONTENT"
            exit 1
          fi
  restartPolicy: Never
EOF
```

Step 13: Lint the Updated Chart

Make sure the tests are valid:

```
helm lint my-webapp
```

Expected output:

```
==> Linting my-webapp
[INFO] Chart.yaml: icon is recommended
```

```
1 chart(s) linted, 0 chart(s) failed
```

Step 14: Install and Run Tests

Install the chart:

```
helm install test-release my-webapp
```

Wait for pod to be ready:

```
kubectl get pods -w
```

Press `Ctrl+C` when the pod shows `1/1 Running`.

Run the tests:

```
helm test test-release
```

Expected output:

```
NAME: test-release
LAST DEPLOYED: Thu Jan 15 16:30:00 2024
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE:     test-release-my-webapp-test-connection
Last Started:   Thu Jan 15 16:30:30 2024
Last Completed: Thu Jan 15 16:30:35 2024
Phase:          Succeeded

TEST SUITE:     test-release-my-webapp-test-content
Last Started:   Thu Jan 15 16:30:36 2024
Last Completed: Thu Jan 15 16:30:40 2024
Phase:          Succeeded
```

Both tests passed!

Step 15: View Test Logs

If you want to see what the tests did:

```
# The pods are deleted on success, but you can disable that
# Let's reinstall without the delete policy for debugging

# First uninstall
helm uninstall test-release
```

Part 5: Debugging Techniques

Step 16: Create an Intentional Error

Let's create a problem and debug it.

Corrupt the deployment template:

```
cp my-webapp/templates/deployment.yaml my-webapp/templates/deployment.yaml.bak

# Add invalid YAML indentation
sed -i '' 's/replicas:/  replicas:/' my-webapp/templates/deployment.yaml 2>/dev/null || \
sed -i 's/replicas:/  replicas:/' my-webapp/templates/deployment.yaml
```

Try to lint:

```
helm lint my-webapp
```

You might see a parsing error or warning.

Try to template:

```
helm template test-release my-webapp 2>&1 | head -n 20
```

You might see an error about YAML structure.

Restore the file:

```
mv my-webapp/templates/deployment.yaml.bak my-webapp/templates/deployment.yaml
```

Step 17: Debug Values Substitution

Install a release:

```
helm install debug-release my-webapp --set replicaCount=3
```

Verify the value was used:

```
helm get values debug-release
```

Expected output:

```
USER-SUPPLIED VALUES:
replicaCount: 3
```

Check the actual deployment:

```
kubectl get deployment debug-release-my-webapp -o yaml | grep replicas
```

Expected output:

```
  replicas: 3
```

Step 18: Debug Running Pods

Get pod information:

```
kubectl get pods -l app.kubernetes.io/instance=debug-release
```

Describe a pod (see events and status):

```
POD=$(kubectl get pods -l app.kubernetes.io/instance=debug-release -o jsonpath='{.items[0].metadata.name}')
kubectl describe pod $POD
```

View pod logs:

```
kubectl logs $POD
```

Step 19: Check Kubernetes Events

See recent cluster events:

```
kubectl get events --sort-by='.lastTimestamp' | tail -n 20
```

This helps identify scheduling issues, image pull errors, etc.

Step 20: Use helm get Commands

View the rendered manifest:

```
helm get manifest debug-release | head -n 40
```

View all values (merged defaults + overrides):

```
helm get values debug-release --all | head -n 30
```

View everything:

```
helm get all debug-release | head -n 50
```

Part 6: Clean Up

Uninstall the release:

```
helm uninstall debug-release
```

Clean up test files:

```
rm test-values.yaml
```

Verify:

```
helm list
kubectl get pods
```

Expected Results Summary

Step	Tool	Expected Result
Lint	`helm lint my-webapp`	0 charts failed
Lint with error	Added invalid YAML	1 chart failed
Template	`helm template my-release my-webapp`	Rendered YAML shown
Template with –set	`--set replicaCount=5`	replicas: 5 in output
kubectl validate	`kubectl apply --dry-run=client`	Resources created (dry run)
Dry run	`helm install --dry-run`	Computed values shown
Create test	Added test-connection.yaml	Test pod defined
Run test	`helm test test-release`	Phase: Succeeded
Debug	`helm get values/manifest`	Configuration shown

Troubleshooting Reference

Problem: Lint shows parse error

Debug:

```
# Check for YAML syntax issues
python3 -c "import yaml; yaml.safe_load(open('my-webapp/values.yaml'))"
```

Common causes: - Missing colons - Incorrect indentation - Invalid characters

---

Problem: Template shows nil pointer

Error message:

```
nil pointer evaluating interface {}.tag
```

Solution:

```
# Use default values
image: "{{ .Values.image.repository }}:{{ default "latest" .Values.image.tag }}"
```

Problem: Test pod fails

Debug:

```
# View test pod logs (before cleanup)
kubectl logs test-release-my-webapp-test-connection

# Or run test with --logs flag
helm test test-release --logs
```

Problem: Pod stuck in Pending

Debug:

```
kubectl describe pod <pod-name>
# Look at Events section at the bottom
```

Common causes: - Insufficient resources - Node selector/affinity issues - PVC not bound

---

Commands Reference

```
# Linting
helm lint <chart>
helm lint <chart> -f values.yaml
helm lint <chart> --strict          # Treat warnings as errors

# Templating
helm template <release> <chart>
helm template <release> <chart> --show-only templates/deployment.yaml
helm template <release> <chart> --set key=value
helm template <release> <chart> | kubectl apply --dry-run=client -f -

# Dry run
helm install <release> <chart> --dry-run
helm install <release> <chart> --dry-run --debug
helm upgrade <release> <chart> --dry-run

# Testing
helm test <release>
```

```
helm test <release> --logs          # Show test pod logs
helm test <release> --timeout 5m    # Set timeout

# Debugging
helm get values <release>           # User-supplied values
helm get values <release> --all     # All values
helm get manifest <release>         # Rendered YAML
helm get notes <release>            # NOTES.txt
helm get all <release>              # Everything

# Kubernetes debugging
kubectl describe pod <pod>
kubectl logs <pod>
kubectl get events --sort-by='.lastTimestamp'
kubectl get all -l app.kubernetes.io/instance=<release>
```

Key Takeaways

1. **`helm lint`** catches syntax errors before deployment
2. **`helm template`** previews output without cluster
3. **`--dry-run`** validates against the actual cluster
4. **Test pods** verify your application works after deployment
5. **`helm get`** helps inspect running releases
6. **`kubectl describe`** shows events and status
7. **Always lint and template** before deploying to production
8. **Write tests** for critical functionality

Excellent! You now have a complete toolkit for testing and debugging Helm charts! # 07 – Final Project: Complete Microservice Deployment

Estimated duration: 40–50 minutes

Project Overview

What You Will Build

In this final project, you'll deploy a complete microservice application with: - A **frontend** web service (nginx-based) - A **backend** API service (nginx simulating an API) - Communication between services - Separate namespaces for different environments - Complete lifecycle management

This project combines everything you've learned: - Creating charts - Deploying releases - Using namespaces - Testing and debugging

---

Skills Integration Map

```
                    SKILLS YOU'LL USE


  Module 01          Module 02          Module 03          Module 04


   Helm            Install          Create           Deploy
```

```
  Basics        Charts        Charts        Releases
```

```
                    FINAL PROJECT
                   (Microservices)
```

```
  Module 05                          Module 06

Namespaces                          Testing
  & Repos                          Debugging
```

Project Architecture

```
                     KUBERNETES CLUSTER

    Namespace: microservices


          FRONTEND              BACKEND
          (nginx)       HTTP      (nginx)

        Port: 80                 Port: 80
        Replicas: 2              Replicas: 2



        NodePort
        (External Access)


         USER BROWSER
```

Part 1: Setup

Step 1: Create Project Directory

```
cd ~/helm-training
mkdir microservice-project && cd microservice-project
```

Step 2: Create Namespace

```
kubectl create namespace microservices
```

Verify:

```
kubectl get namespace microservices
```

Part 2: Create the Backend Chart

Step 3: Create Backend Chart Structure

```
helm create backend
```

Step 4: Configure Backend Chart.yaml

```
cat > backend/Chart.yaml << 'EOF'
apiVersion: v2
name: backend
description: Backend API service for the microservices project
type: application
version: 0.1.0
appVersion: "1.0.0"
keywords:
  - backend
  - api
  - microservices
maintainers:
  - name: Helm Training Student
EOF
```

Step 5: Configure Backend values.yaml

```
cat > backend/values.yaml << 'EOF'
# Replica configuration
replicaCount: 2

# Container image
image:
  repository: nginx
  tag: "1.25-alpine"
  pullPolicy: IfNotPresent

# Service configuration
service:
  type: ClusterIP
  port: 80

# Resource limits
resources:
  limits:
    cpu: 100m
    memory: 128Mi
  requests:
    cpu: 50m
    memory: 64Mi

# Backend-specific configuration
```

```
apiVersion: "v1"
apiName: "Backend API"
EOF
```

Step 6: Create Backend ConfigMap

```
cat > backend/templates/configmap.yaml << 'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "backend.fullname" . }}-config
  labels:
    {{- include "backend.labels" . | nindent 4 }}
data:
  index.html: |
    {
      "service": "{{ .Values.apiName }}",
      "version": "{{ .Values.apiVersion }}",
      "status": "healthy",
      "release": "{{ .Release.Name }}",
      "namespace": "{{ .Release.Namespace }}",
      "timestamp": "{{ now | date "2006-01-02T15:04:05Z07:00" }}"
    }
  nginx.conf: |
    server {
        listen 80;
        server_name localhost;

        location / {
            root /usr/share/nginx/html;
            default_type application/json;
            add_header Content-Type application/json;
        }

        location /health {
            return 200 '{"status": "healthy"}';
            add_header Content-Type application/json;
        }
    }
EOF
```

Step 7: Update Backend Deployment

```
cat > backend/templates/deployment.yaml << 'EOF'
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "backend.fullname" . }}
  labels:
    {{- include "backend.labels" . | nindent 4 }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      {{- include "backend.selectorLabels" . | nindent 6 }}
  template:
    metadata:
      labels:
```

```
        {{- include "backend.selectorLabels" . | nindent 8 }}
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          ports:
            - name: http
              containerPort: 80
              protocol: TCP
          volumeMounts:
            - name: html-volume
              mountPath: /usr/share/nginx/html
            - name: nginx-config
              mountPath: /etc/nginx/conf.d
          resources:
            {{- toYaml .Values.resources | nindent 12 }}
          livenessProbe:
            httpGet:
              path: /health
              port: http
            initialDelaySeconds: 5
            periodSeconds: 10
          readinessProbe:
            httpGet:
              path: /health
              port: http
            initialDelaySeconds: 5
            periodSeconds: 10
      volumes:
        - name: html-volume
          configMap:
            name: {{ include "backend.fullname" . }}-config
            items:
              - key: index.html
                path: index.html
        - name: nginx-config
          configMap:
            name: {{ include "backend.fullname" . }}-config
            items:
              - key: nginx.conf
                path: default.conf
EOF
```

Step 8: Update Backend Service

```
cat > backend/templates/service.yaml << 'EOF'
apiVersion: v1
kind: Service
metadata:
  name: {{ include "backend.fullname" . }}
  labels:
    {{- include "backend.labels" . | nindent 4 }}
spec:
  type: {{ .Values.service.type }}
  ports:
    - port: {{ .Values.service.port }}
```

```
      targetPort: http
      protocol: TCP
      name: http
  selector:
    {{- include "backend.selectorLabels" . | nindent 4 }}
EOF
```

Step 9: Create Backend Test

```
mkdir -p backend/templates/tests

cat > backend/templates/tests/test-connection.yaml << 'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: "{{ include "backend.fullname" . }}-test"
  labels:
    {{- include "backend.labels" . | nindent 4 }}
  annotations:
    "helm.sh/hook": test
    "helm.sh/hook-delete-policy": hook-succeeded
spec:
  containers:
    - name: test
      image: curlimages/curl:8.4.0
      command: ['sh', '-c']
      args:
        - |
          echo "Testing backend service..."
          RESPONSE=$(curl -s {{ include "backend.fullname" . }}:{{ .Values.service.port }})
          echo "Response: $RESPONSE"
          if echo "$RESPONSE" | grep -q "healthy"; then
            echo "  Backend test passed!"
            exit 0
          else
            echo "  Backend test failed!"
            exit 1
          fi
  restartPolicy: Never
EOF
```

Step 10: Remove Unnecessary Backend Files

```
rm backend/templates/ingress.yaml
rm backend/templates/hpa.yaml
rm backend/templates/serviceaccount.yaml
```

Step 11: Lint Backend Chart

```
helm lint backend
```

Expected output:

```
==> Linting backend
[INFO] Chart.yaml: icon is recommended

1 chart(s) linted, 0 chart(s) failed
```

Part 3: Create the Frontend Chart

Step 12: Create Frontend Chart Structure

```
helm create frontend
```

Step 13: Configure Frontend Chart.yaml

```
cat > frontend/Chart.yaml << 'EOF'
apiVersion: v2
name: frontend
description: Frontend web service for the microservices project
type: application
version: 0.1.0
appVersion: "1.0.0"
keywords:
  - frontend
  - web
  - microservices
maintainers:
  - name: Helm Training Student
EOF
```

Step 14: Configure Frontend values.yaml

```
cat > frontend/values.yaml << 'EOF'
# Replica configuration
replicaCount: 2

# Container image
image:
  repository: nginx
  tag: "1.25-alpine"
  pullPolicy: IfNotPresent

# Service configuration
service:
  type: NodePort
  port: 80

# Resource limits
resources:
  limits:
    cpu: 100m
    memory: 128Mi
  requests:
    cpu: 50m
    memory: 64Mi

# Frontend-specific configuration
appTitle: "Microservices Demo"
backendService: "backend-api"

# Environment
environment: "development"
EOF
```

Step 15: Create Frontend ConfigMap

```
cat > frontend/templates/configmap.yaml << 'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "frontend.fullname" . }}-config
  labels:
    {{- include "frontend.labels" . | nindent 4 }}
data:
  index.html: |
    <!DOCTYPE html>
    <html>
    <head>
      <title>{{ .Values.appTitle }}</title>
      <style>
        * { margin: 0; padding: 0; box-sizing: border-box; }
        body {
          font-family: -apple-system, BlinkMacSystemFont, 'Segoe UI', Roboto, sans-serif;
          background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
          min-height: 100vh;
          display: flex;
          align-items: center;
          justify-content: center;
        }
        .container {
          background: white;
          border-radius: 20px;
          padding: 40px;
          box-shadow: 0 20px 60px rgba(0,0,0,0.3);
          max-width: 800px;
          width: 90%;
        }
        h1 {
          color: #333;
          margin-bottom: 20px;
          text-align: center;
        }
        .info-grid {
          display: grid;
          grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
          gap: 20px;
          margin: 30px 0;
        }
        .info-card {
          background: #f8f9fa;
          padding: 20px;
          border-radius: 10px;
          border-left: 4px solid #667eea;
        }
        .info-card h3 {
          color: #667eea;
          margin-bottom: 10px;
          font-size: 14px;
          text-transform: uppercase;
        }
        .info-card p {
```

```
      color: #333;
      font-size: 18px;
      font-weight: bold;
    }
    .backend-status {
      background: #e8f5e9;
      border: 2px solid #4caf50;
      border-radius: 10px;
      padding: 20px;
      margin-top: 20px;
    }
    .backend-status h3 {
      color: #2e7d32;
      margin-bottom: 10px;
    }
    #backend-response {
      font-family: monospace;
      background: #f5f5f5;
      padding: 15px;
      border-radius: 5px;
      white-space: pre-wrap;
    }
    .badge {
      display: inline-block;
      padding: 5px 15px;
      border-radius: 20px;
      font-size: 12px;
      font-weight: bold;
      text-transform: uppercase;
    }
    .badge-{{ .Values.environment }} {
      background: {{ if eq .Values.environment "production" }}#f44336{{ else if eq .Values.environment
      color: white;
    }
  </style>
</head>
<body>
  <div class="container">
    <h1> {{ .Values.appTitle }}</h1>
    <p style="text-align: center; margin-bottom: 20px;">
      <span class="badge badge-{{ .Values.environment }}">{{ .Values.environment }}</span>
    </p>

    <div class="info-grid">
      <div class="info-card">
        <h3>Frontend Release</h3>
        <p>{{ .Release.Name }}</p>
      </div>
      <div class="info-card">
        <h3>Namespace</h3>
        <p>{{ .Release.Namespace }}</p>
      </div>
      <div class="info-card">
        <h3>Chart Version</h3>
        <p>{{ .Chart.Version }}</p>
      </div>
      <div class="info-card">
```

```
            <h3>Replicas</h3>
            <p>{{ .Values.replicaCount }}</p>
          </div>
        </div>

        <div class="backend-status">
          <h3> Backend API Response</h3>
          <div id="backend-response">Loading...</div>
        </div>
      </div>

      <script>
        async function fetchBackend() {
          try {
            const response = await fetch('/api/');
            const data = await response.json();
            document.getElementById('backend-response').textContent = JSON.stringify(data, null, 2);
          } catch (error) {
            document.getElementById('backend-response').textContent =
              'Error connecting to backend: ' + error.message + '\n\n' +
              'This is expected if you are viewing this directly.\n' +
              'The backend connection works through the nginx proxy.';
          }
        }
        fetchBackend();
        setInterval(fetchBackend, 5000);
      </script>
    </body>
    </html>
  nginx.conf: |
    server {
        listen 80;
        server_name localhost;

        location / {
            root /usr/share/nginx/html;
            index index.html;
        }

        location /api/ {
            proxy_pass http://{{ .Values.backendService }}:80/;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }

        location /health {
            return 200 '{"status": "healthy"}';
            add_header Content-Type application/json;
        }
    }
EOF
```

Step 16: Update Frontend Deployment

```
cat > frontend/templates/deployment.yaml << 'EOF'
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
  name: {{ include "frontend.fullname" . }}
  labels:
    {{- include "frontend.labels" . | nindent 4 }}
spec:
  replicas: {{ .Values.replicaCount }}
  selector:
    matchLabels:
      {{- include "frontend.selectorLabels" . | nindent 6 }}
  template:
    metadata:
      labels:
        {{- include "frontend.selectorLabels" . | nindent 8 }}
    spec:
      containers:
        - name: {{ .Chart.Name }}
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          ports:
            - name: http
              containerPort: 80
              protocol: TCP
          volumeMounts:
            - name: html-volume
              mountPath: /usr/share/nginx/html
            - name: nginx-config
              mountPath: /etc/nginx/conf.d
          resources:
            {{- toYaml .Values.resources | nindent 12 }}
          livenessProbe:
            httpGet:
              path: /health
              port: http
            initialDelaySeconds: 5
            periodSeconds: 10
          readinessProbe:
            httpGet:
              path: /health
              port: http
            initialDelaySeconds: 5
            periodSeconds: 10
      volumes:
        - name: html-volume
          configMap:
            name: {{ include "frontend.fullname" . }}-config
            items:
              - key: index.html
                path: index.html
        - name: nginx-config
          configMap:
            name: {{ include "frontend.fullname" . }}-config
            items:
              - key: nginx.conf
                path: default.conf
EOF
```

Step 17: Update Frontend Service

```
cat > frontend/templates/service.yaml << 'EOF'
apiVersion: v1
kind: Service
metadata:
  name: {{ include "frontend.fullname" . }}
  labels:
    {{- include "frontend.labels" . | nindent 4 }}
spec:
  type: {{ .Values.service.type }}
  ports:
    - port: {{ .Values.service.port }}
      targetPort: http
      protocol: TCP
      name: http
  selector:
    {{- include "frontend.selectorLabels" . | nindent 4 }}
EOF
```

Step 18: Create Frontend Test

```
mkdir -p frontend/templates/tests

cat > frontend/templates/tests/test-connection.yaml << 'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: "{{ include "frontend.fullname" . }}-test"
  labels:
    {{- include "frontend.labels" . | nindent 4 }}
  annotations:
    "helm.sh/hook": test
    "helm.sh/hook-delete-policy": hook-succeeded
spec:
  containers:
    - name: test
      image: curlimages/curl:8.4.0
      command: ['sh', '-c']
      args:
        - |
          echo "Testing frontend service..."
          RESPONSE=$(curl -s {{ include "frontend.fullname" . }}:{{ .Values.service.port }})
          if echo "$RESPONSE" | grep -q "{{ .Values.appTitle }}"; then
            echo "  Frontend test passed!"
            exit 0
          else
            echo "  Frontend test failed!"
            exit 1
          fi
  restartPolicy: Never
EOF
```

Step 19: Remove Unnecessary Frontend Files

```
rm frontend/templates/ingress.yaml
rm frontend/templates/hpa.yaml
rm frontend/templates/serviceaccount.yaml
```

Step 20: Lint Frontend Chart

```
helm lint frontend
```

Part 4: Deploy the Application

```
                    DEPLOYMENT ORDER


    Step 1              Step 2              Step 3              Step 4


   Deploy              Test              Deploy              Test
   Backend          Backend           Frontend          Frontend
    First                                Second



helm install        helm test         helm install        helm test
backend-api         backend-api       frontend-web       frontend-web

    WHY THIS ORDER?

    Backend must be running before Frontend
    because Frontend proxies requests to Backend.
```

Step 21: Deploy Backend First

```
helm install backend-api backend \
  --namespace microservices
```

Wait for backend to be ready:

```
kubectl get pods -n microservices -w
```

Press `Ctrl+C` when pods show `2/2 Running`.

Step 22: Test Backend

```
helm test backend-api -n microservices
```

Expected output:

```
TEST SUITE:     backend-api-backend-test
...
Phase:          Succeeded
```

Step 23: Deploy Frontend

```
helm install frontend-web frontend \
  --namespace microservices \
  --set backendService=backend-api-backend
```

Wait for frontend to be ready:

```
kubectl get pods -n microservices
```

Step 24: Test Frontend

```
helm test frontend-web -n microservices
```

Part 5: Verify the Application

Step 25: List All Releases

```
helm list -n microservices
```

Expected output:

```
NAME          NAMESPACE       REVISION  STATUS    CHART           APP VERSION
backend-api   microservices   1         deployed  backend-0.1.0   1.0.0
frontend-web  microservices   1         deployed  frontend-0.1.0  1.0.0
```

Step 26: Check All Resources

```
kubectl get all -n microservices
```

Expected output:

```
NAME                                     READY   STATUS    RESTARTS   AGE
pod/backend-api-backend-xxxxx            1/1     Running   0          5m
pod/backend-api-backend-yyyyy            1/1     Running   0          5m
pod/frontend-web-frontend-aaaaa          1/1     Running   0          3m
pod/frontend-web-frontend-bbbbb          1/1     Running   0          3m

NAME                           TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
service/backend-api-backend    ClusterIP   10.96.xxx.xxx    <none>        80/TCP
service/frontend-web-frontend  NodePort    10.96.yyy.yyy    <none>        80:xxxxx/TCP

NAME                                   READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/backend-api-backend    2/2     2            2           5m
deployment.apps/frontend-web-frontend  2/2     2            2           3m
```

Step 27: Access the Application

```
minikube service frontend-web-frontend -n microservices --url
```

Open the URL in your browser!

You should see: - A beautiful frontend page - Release information - Backend API response (JSON from the backend service)

Part 6: Upgrade the Application

Step 28: Upgrade Backend to v2

```
helm upgrade backend-api backend \
  --namespace microservices \
  --set apiVersion="v2" \
  --set replicaCount=3
```

Step 29: Verify the Upgrade

```
helm history backend-api -n microservices
```

Expected output:

```
REVISION  STATUS      DESCRIPTION
1         superseded  Install complete
2         deployed    Upgrade complete
```

Check pods:

```
kubectl get pods -n microservices | grep backend
```

Should show 3 backend pods now.

Step 30: Upgrade Frontend Environment

```
helm upgrade frontend-web frontend \
  --namespace microservices \
  --set backendService=backend-api-backend \
  --set environment=production \
  --set appTitle="Production Microservices"
```

Refresh your browser to see the changes: - Title changed to "Production Microservices" - Badge shows "PRODUCTION" (red)

Part 7: Rollback (If Needed)

Step 31: Simulate a Problem

Let's say the production upgrade was premature. Roll back:

```
helm rollback frontend-web 1 -n microservices
```

Verify:

```
helm history frontend-web -n microservices
```

Expected output:

```
REVISION  STATUS      DESCRIPTION
1         superseded  Install complete
2         superseded  Upgrade complete
3         deployed    Rollback to 1
```

Part 8: Clean Up

Step 32: Uninstall Everything

```
helm uninstall frontend-web -n microservices
helm uninstall backend-api -n microservices
```

Step 33: Delete Namespace

```
kubectl delete namespace microservices
```

Step 34: Verify Cleanup

```
helm list -n microservices
kubectl get namespace microservices
```

Both should return empty or "not found".

Project Checklist

Task	Completed
Created backend chart	
Created frontend chart	
Deployed to microservices namespace	
Backend tests passed	
Frontend tests passed	
Accessed application in browser	
Upgraded backend to v2	
Upgraded frontend to production	
Performed rollback	
Cleaned up all resources	

What You Learned

1. **Creating charts** – Structured multiple services
2. **ConfigMaps** – Custom nginx configuration
3. **Service communication** – Backend/frontend proxy
4. **Namespaces** – Isolated deployment
5. **Testing** – Validated deployments
6. **Upgrades** – Changed configuration live
7. **Rollbacks** – Reverted changes
8. **Complete lifecycle** – Install → Upgrade → Rollback → Uninstall

---

Bonus Challenges (Optional)

Challenge 1: Add a Database

Create a third chart for Redis or PostgreSQL and connect the backend to it.

Challenge 2: Add Resource Quotas

Add resource quotas to the microservices namespace.

Challenge 3: Create an Umbrella Chart

Create a parent chart that includes both frontend and backend as dependencies.

Challenge 4: CI/CD Simulation

Write a shell script that: 1. Lints both charts 2. Templates and validates 3. Deploys with –atomic 4. Runs tests 5. Rolls back on failure

---

CONGRATULATIONS! You've completed the Helm Training Course!

You now have the skills to: - Create custom Helm charts - Deploy and manage releases - Test and debug charts - Work with repositories and namespaces - Handle the complete release lifecycle

Welcome to the world of Kubernetes package management!