Kubernetes Manifest Tools

A Deep Dive into Helm and Kustomize

The Challenge of Plain YAML

Managing raw Kubernetes YAML files across different environments is difficult.

- **Duplication:** Copying and pasting manifests for `dev`, `staging`, and `prod` leads to errors.
- **Configuration Drift:** Manual changes make it hard to know what's actually running. How do you change the replica count for just one environment?
- **Maintenance Burden:** A simple change, like adding a label, requires editing dozens of files.
- **Lack of Abstraction:** No easy way to package and distribute a complex application like a database cluster.

 Introducing Helm

Helm is the **package manager for Kubernetes**. It allows you to package, configure, and deploy applications and services onto your clusters.

- **It's for sharing and reusing applications.** Think of it like `apt`, `yum`, or `Homebrew` for Kubernetes.
- **Core Concepts:**
  - A **Chart** is a package containing all the resource definitions needed to run an application.
  - A **Release** is an instance of a chart running in a Kubernetes cluster.
  - A **Repository** is a place where charts can be collected and shared.

📦 Inside a Helm Chart

A Helm chart is a collection of files in a specific directory structure.

```
my-chart/
├── Chart.yaml          # Metadata: name, version, description
├── values.yaml         # The default configuration values for this chart
├── templates/          # A directory of templates that, when combined
│   │                   # with values, will generate valid Kubernetes manifests.
│   ├── deployment.yaml
│   ├── service.yaml
│   └── _helpers.tpl    # Reusable template snippets (partials)
└── charts/             # Optional directory for dependency charts (subcharts)
```

⚙️ Helm Templating: An Example

Helm uses Go templates to make manifests dynamic.

`templates/deployment.yaml`:

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

`values.yaml`:

🚀 Helm in Action: The Workflow

The typical workflow involves finding, customizing, and installing charts.

1. **Find a chart:**

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

   ```
   helm search repo bitnami/mysql
   ```

2. **Install with custom values:**

   ```
   helm install my-db bitnami/mysql --set auth.rootPassword=secret
   ```

3. **Upgrade a release:**

   ```
   helm upgrade my-db bitnami/mysql --set image.tag=8.0.30
   ```

4. **Roll back to a previous version:**

   ```
   helm rollback my-db 1
   ```

🎨 Introducing Kustomize

Kustomize is a **template-free** way to customize application configuration. It's built into `kubectl`.

- **Philosophy: Patch, Don't Template.** Instead of creating complex templates, you start with a base YAML and apply patches to create variants.
- **Declarative:** All customizations are defined in a `kustomization.yaml` file.
- **YAML-native:** It understands the structure of Kubernetes objects, which makes patching safer and more intuitive than simple text replacement.

📂 Kustomize Structure: Bases and Overlays

Kustomize works by taking a `base` configuration and applying `overlays` to it.

```
base/
├── deployment.yaml
├── service.yaml
└── kustomization.yaml  # Defines the resources for the base

overlays/
├── staging/
│   ├── kustomization.yaml  # Points to the base and defines patches
│   └── replica-patch.yaml  # A patch to change the replica count
└── production/
    ├── kustomization.yaml
    └── cpu-patch.yaml      # A patch to increase CPU resources
```

This structure keeps the configuration for each environment clean and isolated.

⚙️ The `kustomization.yaml` File

This file is the heart of Kustomize. It tells Kustomize what to do.

`overlays/production/kustomization.yaml`:

```
# Inherit from the base configuration
resources:
  - ../../base

# Add a common label to all resources
commonLabels:
  env: production

# Apply patches
patchesStrategicMerge:
  - replica-patch.yaml
  - cpu-patch.yaml

# Change image tags
images:
```

🔧 Kustomize Patching: An Example

Patches are small snippets of YAML that override values in the base manifests.

`base/deployment.yaml` (excerpt):

```
# ...
spec:
  replicas: 1
  template:
    spec:
      containers:
        - name: my-app
          image: my-app:latest
# ...
```

`overlays/production/replica-patch.yaml`:

```
apiVersion: apps/v1
kind: Deployment
```

⚖️ Helm vs. Kustomize: A Deeper Look

Aspect	Helm	Kustomize
Paradigm	**Templating:** Uses a programming language to generate YAML.	**Patching:** Merges YAML files to create variants.
Complexity	Higher. Requires learning Go templating and chart structure.	Lower. It's just YAML.
Use Case	**Packaging & Distribution:** Ideal for complex, configurable apps meant for sharing .	**Customization:** Ideal for managing environment-specific variants of your own applications.

⚖️ Helm vs. Kustomize: A Deeper Look (cont.)

Aspect	Helm	Kustomize
Intrusiveness	You must "Helm-ify" your app. The logic is inside the templates.	Non-intrusive. Works with any standard Kubernetes YAML file.
Ecosystem	Huge. Artifact Hub is a massive repository of pre-built charts.	Smaller, but built directly into `kubectl`.

🤝 Can They Work Together? Yes!

You don't always have to choose. A powerful pattern is to use both:

1. **Use Helm to deploy a third-party chart** (like a database or message queue) and manage its lifecycle.
2. **Use Kustomize to manage your own application's manifests.** Your Kustomize overlays can then configure your app to connect to the Helm-deployed service.

This gives you the best of both worlds: Helm for reusable packages and Kustomize for clean, environment-specific customization of your own code.

🥑 Integration with Argo CD

Argo CD has native support for both Helm and Kustomize.

- When you create an Argo CD Application, you simply tell it the type of your source.
- **For Helm:** Point Argo CD to a Git repo containing a chart. You can provide override values directly in the Argo CD Application manifest.
- **For Kustomize:** Point Argo CD to a Git repo containing a `kustomization.yaml`. Argo CD will automatically run `kustomize build` to generate the final manifests.

This seamless integration makes both tools excellent choices for a GitOps workflow.

✅ Summary

- **Helm** is a **package manager** that uses **templating**. It's great for distributing complex, reusable applications.
- **Kustomize** is a **customization tool** that uses **patching**. It's great for managing environment-specific variants of your own applications.
- Both tools solve the problem of managing raw YAML and are first-class citizens in the Argo CD ecosystem.
- You can even use them together to create a powerful and flexible GitOps workflow.

🙏 Questions?