



PUBLISHED · OCTOBER 5, 2026

Kustomize for Declarative Kubernetes Configuration

Build a practical Kustomize mental model, organize reusable bases and focused overlays, render safe variants, and apply them declaratively with kubectl.



This is Learn post 24 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. You already know that Kubernetes resources are declarative API objects and that kubectl apply sends desired configuration to the API server. Kustomize adds a client-side composition step: it builds environment-specific manifests from reusable YAML before Kubernetes sees them.

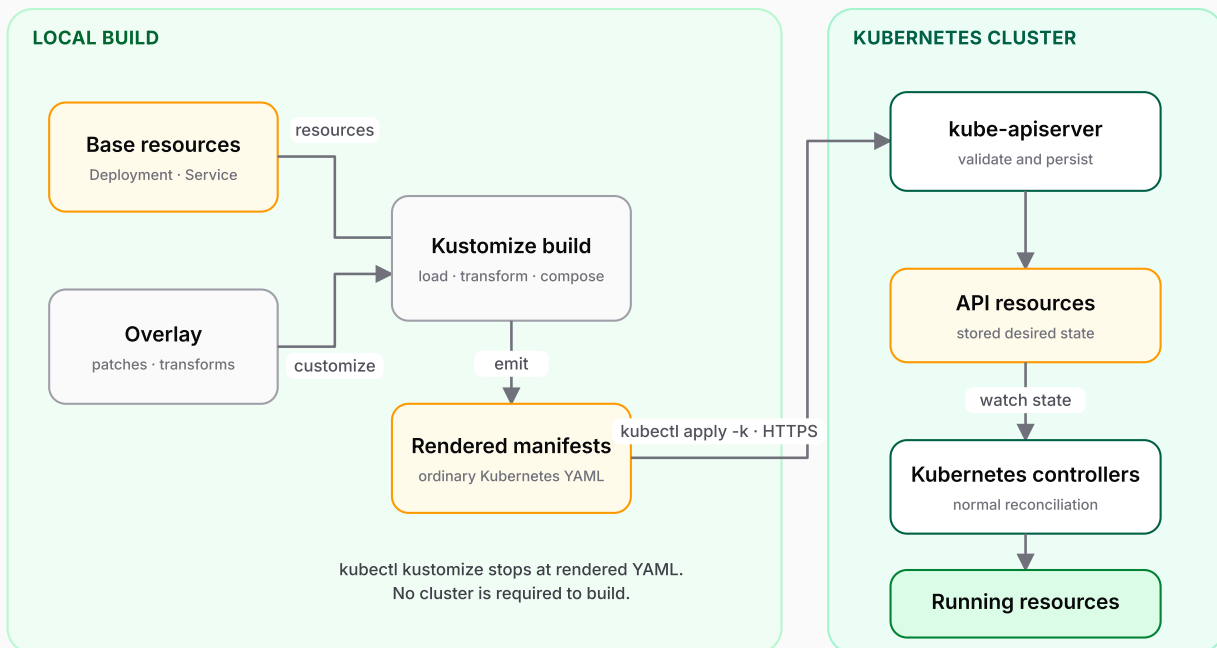
That distinction matters when the same Deployment and Service need small, controlled differences across environments. Instead of copying whole manifests or inserting template expressions into YAML, you keep complete resources in a base and describe each environment as a set of transformations and patches.

What you'll learn

- Explain where Kustomize ends and Kubernetes API processing begins.
- Organize plain Kubernetes resources into a reusable base and a focused overlay.
- Use resources, namespace, labels, images, replicas, patches, and ConfigMap generation deliberately.
- Render, inspect, diff, apply, and observe a customization with kubectl.
- Distinguish Kustomize configuration from Helm releases and from live Kubernetes state.

The mental model: load, transform, emit

Kustomize loads Kubernetes resources, applies declared transformations in memory, and emits ordinary Kubernetes manifests. The API server receives only the emitted resources; it does not run Kustomize or store a Kustomize release.



Kustomize builds locally; Kubernetes reconciles the result

Kustomize assembles ordinary manifests before the API request. The API server and Kubernetes controllers only process the rendered resources; they do not store or run the Kustomization.

Because rendering happens before the API request, `kubectrl kustomize` can run without a cluster. It can reveal the exact YAML that would be submitted, but it cannot prove that the API server will authorize or admit it, or that the resulting Pods will become healthy.

A `kustomization.yaml` file uses `apiVersion` and `kind` fields, but it is build configuration rather than a Kubernetes API object. You will not find a Kustomization resource with `kubectrl get`. `kubectrl` reads the file locally, builds the resources, and submits those resources when you use `-k`.

Base and overlay: stable resources plus intentional differences

A base is a kustomization that describes a useful set of resources. An overlay is another kustomization that includes a base and adds environment-specific changes. Both are ordinary kustomizations; base and overlay describe how you use the directories, not different API kinds.

Keep the base complete enough to render and understand on its own. Keep overlays small enough that a reviewer can see exactly what differs. The rendered output is the truth that will be offered to Kubernetes.

directory layout · text

```
kustomize-demo/  
  base/  
    deployment.yaml  
    service.yaml  
    kustomization.yaml  
  overlays/  
    prod/  
      deployment-patch.yaml  
      namespace.yaml  
      kustomization.yaml
```

Build the reusable base

The base begins with the same valid Deployment and Service manifests you would apply directly. There are no placeholder expressions in them. The Deployment reads application settings from a ConfigMap named web-settings; Kustomize will generate that ConfigMap.

base/deployment.yaml · yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  labels:
    app: web
spec:
  replicas: 1
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: nginx:1.27.3
          ports:
            - name: http
              containerPort: 80
          envFrom:
            - configMapRef:
                name: web-settings
```

base/service.yaml · yaml

```
apiVersion: v1
kind: Service
metadata:
  name: web
spec:
  selector:
    app: web
  ports:
    - name: http
      port: 80
      targetPort: http
```

The kustomization lists its input resources and declares a ConfigMap generator. This file is the entry point for the directory.

```
base/kustomization.yaml · yaml
```

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization

resources:
- deployment.yaml
- service.yaml

configMapGenerator:
- name: web-settings
  literals:
  - APP_MESSAGE=Hello from the base
```

resources composes existing manifests. configMapGenerator creates a ConfigMap from declared literals or files. The generator normally adds a content-derived suffix to the name, so a configuration change produces a new name.

Render before you touch the cluster

Use `kubectl kustomize` when you want to inspect the build product only. Point it at the directory, not at the `kustomization.yaml` filename.

```
bash
```

```
kubectl kustomize kustomize-demo/base
```

rendered excerpt · yaml

```
apiVersion: v1
data:
  APP_MESSAGE: Hello from the base
kind: ConfigMap
metadata:
  name: web-settings-<content-hash>
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
spec:
  template:
    spec:
      containers:
        - envFrom:
            - configMapRef:
                name: web-settings-<content-hash>
          image: nginx:1.27.3
          name: web
```

The actual suffix is deterministic for the generated content and Kustomize version, so it is shown symbolically here rather than as a value to memorize. Notice the important relationship: Kustomize changes both the generated ConfigMap name and the Deployment's ConfigMap reference. Kubernetes later sees two matching names; it does not know that Kustomize performed the rewrite.

Because the generated name appears inside the Pod template, changing the ConfigMap content changes the rendered Deployment template. Applying that change gives the Deployment controller a new Pod template to roll out, while new Pods read the new ConfigMap. This connects configuration generation to the rollout behavior taught earlier in the series.

Layer a production overlay

The production overlay will create its namespace, reuse the base, set that namespace on namespaced resources, add an environment label, scale the Deployment, update the image tag, merge production configuration, and add resource requests and limits

through a patch.

Generate a clean Namespace manifest efficiently, then keep the resulting YAML as source-controlled configuration.

```
bash
```

```
kubect! create namespace web-prod \  
--dry-run=client -o yaml \  
> kustomize-demo/overlays/prod/namespace.yaml
```

```
overlays/prod/namespace.yaml · yaml
```

```
apiVersion: v1  
kind: Namespace  
metadata:  
  name: web-prod
```

The patch contains only the Deployment identity and the fields production needs to add. Kustomize merges it into the matching resource during the build.

```
overlays/prod/deployment-patch.yaml · yaml
```

```
apiVersion: apps/v1  
kind: Deployment  
metadata:  
  name: web  
spec:  
  template:  
    spec:  
      containers:  
        - name: web  
          resources:  
            requests:  
              cpu: 100m  
              memory: 64Mi  
            limits:  
              cpu: 500m  
              memory: 128Mi
```

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
```

```
resources:
- ../../base
- namespace.yaml
```

```
namespace: web-prod
```

```
labels:
- pairs:
  environment: production
  includeSelectors: true
```

```
replicas:
- name: web
  count: 3
```

```
images:
- name: nginx
  newTag: 1.27.4
```

```
configMapGenerator:
- name: web-settings
  behavior: merge
  literals:
  - APP_MESSAGE=Hello from production
```

```
patches:
- path: deployment-patch.yaml
  target:
    kind: Deployment
    name: web
```

Each field answers a narrow question. `resources` chooses the input set. `namespace` sets the destination namespace on namespaced resources. `labels` applies consistent metadata and, because `includeSelectors` is true, updates selectors and Pod template labels together. `replicas` and `images` change common workload fields without copying the Deployment. The generator merges one value, and the patch handles the container resources that need structural YAML.

Use `includeSelectors` deliberately. Adding a label to a Deployment selector after creation can collide with selector immutability, and adding a label to a Service selector changes which Pods receive traffic. Rendering exposes these effects before an API request.

Inspect the production build

Render the overlay and save the output when a full review or another validation tool is useful. The overlay path is the deployable unit.

```
bash
```

```
kubectl kustomize kustomize-demo/overlays/prod \
> prod-rendered.yaml
```

```
less prod-rendered.yaml
```

```
important rendered fields - yaml
```

```
# ConfigMap
metadata:
  name: web-settings-<new-content-hash>
  namespace: web-prod
data:
  APP_MESSAGE: Hello from production
---
# Deployment
metadata:
  labels:
    environment: production
  name: web
  namespace: web-prod
spec:
  replicas: 3
  selector:
    matchLabels:
      app: web
      environment: production
  template:
    metadata:
      labels:
        app: web
        environment: production
    spec:
      containers:
        - image: nginx:1.27.4
          name: web
          resources:
            requests:
              cpu: 100m
              memory: 64Mi
```

What matters is the combined result: every namespaced object targets web-prod; selectors still match the Pod labels; the Deployment has three replicas; the image and resources reflect production; and the ConfigMap reference matches the generated name. Review relationships, not just individual fields.

Kustomize does not create a variable system inside your resource YAML. It transforms structured Kubernetes objects. That keeps the base valid YAML and makes the rendered manifests inspectable with ordinary Kubernetes tools.

Move from local rendering to API validation

Use server-side dry-run when you want the API server to process the built resources without persisting them. This can catch schema, authorization, admission, and policy failures that local rendering cannot see.

```
bash
```

```
kubectl apply --dry-run=server \
-k kustomize-demo/overlays/prod
```

A successful server-side dry-run means the request passed the API processing available at that moment. It does not create resources, wait for a rollout, pull the image, schedule Pods, or prove application health.

When resources already exist, diff shows the proposed live changes. It contacts the API server and therefore needs the relevant read and dry-run update permissions.

```
bash
```

```
kubectl diff -k kustomize-demo/overlays/prod
```

Read the diff as an API-object change, not as a source-file diff. It compares the built configuration with live state, so one small overlay edit may legitimately affect several rendered fields and resources.

Apply and observe the normal Kubernetes objects

Apply the overlay with `-k`. `kubectl` builds it and sends the resulting Namespace, ConfigMap, Service, and Deployment to the API server.

```
bash
```

```
kubectl apply -k customize-demo/overlays/prod
```

```
text
```

```
namespace/web-prod created
configmap/web-settings-<content-hash> created
service/web created
deployment.apps/web created
```

The output lists Kubernetes resources, not a Kustomize release. After the API server stores the Deployment, the Deployment controller performs the same ReplicaSet and Pod reconciliation you learned earlier. Kustomize is no longer in that control loop.

Observe the resource relationships with `kubectl`. The label selector lets you view the production objects together, while rollout status checks the Deployment controller's progress.

bash

```
kubectl get deployment,service,pods,configmap \
--namespace web-prod \
-l environment=production
```

```
kubectl rollout status deployment/web \
--namespace web-prod
```

```
kubectl describe deployment web \
--namespace web-prod
```

text

NAME	READY	UP-TO-DATE	AVAILABLE
deployment.apps/web	3/3	3	3

NAME	TYPE	CLUSTER-IP	PORT(S)
service/web	ClusterIP	10.96.18.42	80/TCP

NAME	READY	STATUS	RESTARTS
pod/web-7c8b6f5d9b-2m8qx	1/1	Running	0
pod/web-7c8b6f5d9b-c7r4p	1/1	Running	0
pod/web-7c8b6f5d9b-vh9kt	1/1	Running	0

NAME	DATA
configmap/web-settings-<content-hash>	1

Names, hashes, Pod suffixes, ClusterIP values, and formatting vary. The stable observations are three available replicas, a Service selecting the labeled Pods, and a generated ConfigMap in the same namespace as the workload.

The administrator workflow: edit source, render, diff, apply, observe

Suppose production should move from nginx:1.27.4 to nginx:1.27.5. Change newTag in the production kustomization, then repeat the same pipeline.

bash

```
kubectl kustomize kustomize-demo/overlays/prod \  
> prod-rendered.yaml  
  
kubectl diff -k kustomize-demo/overlays/prod  
kubectl apply -k kustomize-demo/overlays/prod  
  
kubectl rollout status deployment/web \  
--namespace web-prod  
  
kubectl get deployment web \  
--namespace web-prod \  
-o jsonpath='{.spec.template.spec.containers[0].image}'
```

text

nginx:1.27.5

Because the image field in the Pod template changes, the Deployment controller creates a new ReplicaSet and performs its normal rolling update. Kustomize made the desired-state change reproducible; Kubernetes executed and reported the rollout.

Avoid making a live kubectl edit and then forgetting the source. A later apply from the overlay can replace that field with the rendered value. If an emergency live change is necessary, reconcile it back into the base or overlay so the declared configuration remains authoritative.

Apply is not an inventory garbage collector

kubectl apply -k creates or updates resources present in the current build. If you remove a resource from resources and apply again, an older live object is not automatically deleted by an ordinary apply. Likewise, changing generated ConfigMap content can leave the previous hash-named ConfigMap behind after the Deployment begins using the new one.

Treat deletion as an explicit lifecycle decision. `kubectl delete -k` deletes the resources in the current render, but it cannot identify an old generated object that is no longer rendered. Inventory and pruning strategies require care because an overly broad selection can delete unrelated resources; detailed cleanup workflows belong in the later Practice phase.

```
bash
```

```
# Inspect exactly what the current overlay identifies.
```

```
kubectl get -k kustomize-demo/overlays/prod
```

```
# Remove the resources in the current rendered set when cleanup is intended.
```

```
kubectl delete -k kustomize-demo/overlays/prod
```

Choose the right customization mechanism

Cross-cutting transformations

Use namespace, labels, name prefixes or suffixes, images, and replicas when the same intent should be applied predictably across matching resources. Always render after changing names or selectors because Kustomize may also rewrite references and relationships.

Patches

Use a patch for a focused structural change that does not have a clearer built-in transformer. Keep its target narrow and its content small. A patch target can select by resource identity or selectors; if nothing matches, the build should fail rather than silently inventing a new resource.

Generators

Use ConfigMap and Secret generators when their source material belongs in the customization workflow and reference rewriting is valuable. A Secret manifest's base64 data is not encryption, so do not commit sensitive literals merely because a generator can encode them. Secret handling fundamentals remain those taught in post 9.

Kustomize and Helm solve different configuration problems

The previous lesson introduced Helm charts, values, releases, and revision history. Kustomize starts from Kubernetes resources and applies structured transformations without template expressions. It emits manifests but does not create release history or provide a rollback command.

text

Kustomize	Helm
Plain resources + kustomization	Chart templates + values
Transform and patch YAML	Render template logic
No Kustomize release object	Named release with revisions
Inspect with <code>kubectl kustomize</code>	Inspect with <code>helm template</code>
Apply objects with <code>kubectl</code>	Helm submits and tracks a release

Both ultimately produce Kubernetes API objects, and the same controllers reconcile those objects. Choose based on the ownership and distribution model: Kustomize is strong for maintaining variants of manifests you control; Helm is strong for packaging and configuring an installable application as a release. Some teams use both at different boundaries, but mixing ownership of the same live fields demands discipline.

Read failures in the right layer

text

Build fails? Paths, YAML, generator merge, or patch target
Dry-run/diff fails? API schema, RBAC, admission, or immutable field
Apply succeeds? Objects were accepted, not necessarily healthy
Workload unhealthy? kubectl get, describe, logs, events, rollout status

This layering prevents wasted investigation. A patch that matches no Deployment is a build problem. A forbidden response is an RBAC problem. A Deployment accepted with Pending Pods is a scheduling or resource problem. Kustomize changes how configuration is assembled; it does not replace Kubernetes troubleshooting evidence.

Administrator checklist

- Point -k at the intended overlay directory and verify the current kubeconfig context.
- Keep base resources valid and overlays limited to meaningful differences.
- Render locally and inspect names, namespaces, selectors, references, images, and generated objects.
- Use server-side dry-run and diff when API-aware validation is available.
- Apply the same overlay you reviewed, then observe the resulting Kubernetes resources.
- Plan deletion and stale generated-resource cleanup explicitly.
- Change declared source rather than allowing forgotten live edits to become hidden configuration.

Where this fits in the series

Earlier lessons supplied the YAML, controller, rollout, ConfigMap, Secret, kubeconfig, and RBAC models that explain Kustomize output. Post 23 taught Helm's package-and-release model; this lesson adds resource composition without release state. Extension

interfaces, CustomResourceDefinitions, Operators, networking, storage, and observability follow in posts 25–36. Guided Kustomize repair and cleanup decisions return in Practice post 47.

Official documentation

[Declarative Management of Kubernetes Objects Using Kustomize](#) covers kustomization files, generators, composition, patches, and the `kubectl -k` workflow.

[kubectl kustomize](#) documents local rendering and its available build flags.

[kubectl diff](#) documents API-aware comparison, including the `-k` option.

[kubectl apply](#) documents declarative apply, server-side dry-run, and processing a kustomization directory with `-k`.