Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · OCTOBER 7, 2026

Custom Resource Definitions and Kubernetes Operators

Understand how CRDs extend the Kubernetes API, how Operators add reconciliation, and how administrators discover, inspect, authorize, and safely manage both layers.



This is Learn post 26 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. You already know the API request flow, built-in controllers, RBAC, Helm, Kustomize, and Kubernetes extension interfaces. This lesson adds a different extension model: new API resource types and controllers that reconcile them.

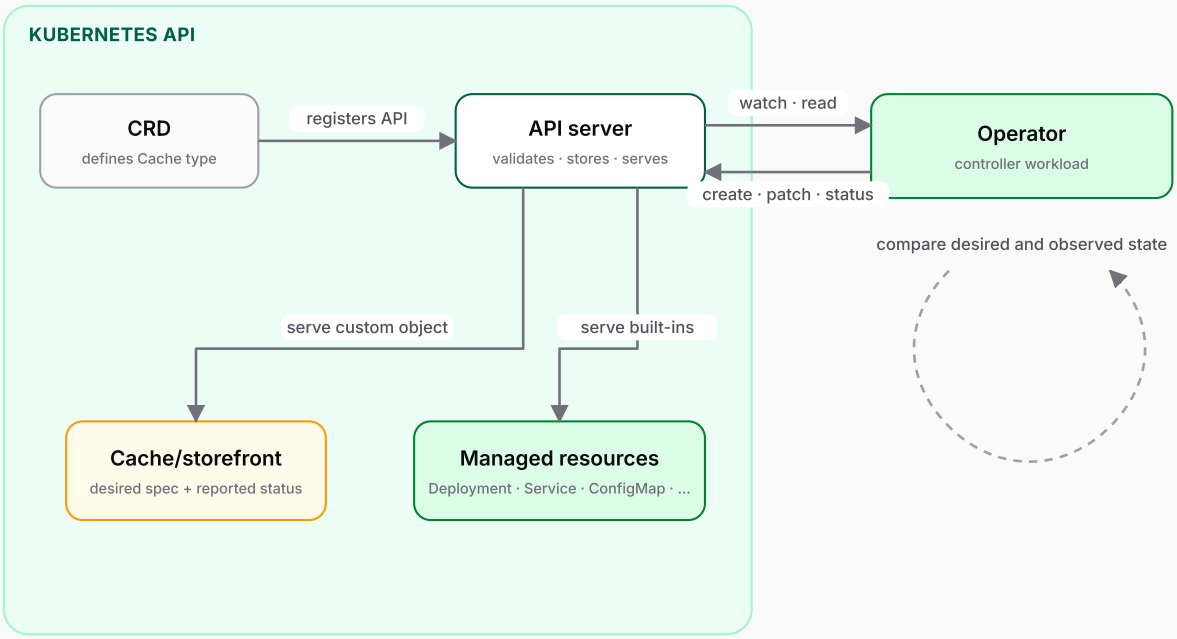
The key distinction is compact: a CustomResourceDefinition (CRD) adds a type to the Kubernetes API; an Operator adds domain-specific behavior. Installing one does not automatically provide the other.

What you'll learn

- Separate a CRD, a custom resource, a custom controller, and an Operator.
- Read a CRD's group, version, kind, plural name, scope, schema, columns, and subresources.
- Install and discover a small custom API, then create and inspect one custom resource.
- Follow an Operator reconciliation loop from desired state to managed resources and reported status.
- Use an administrator's observation order for installed CRDs and Operators without drifting into Operator development.

The mental model: add a noun, then add a control loop

> A CRD teaches the API server a new noun. A custom resource declares one desired instance of that noun. An Operator watches those instances and repeatedly turns their desired state into real resources and status.



A CRD supplies the API; an Operator supplies reconciliation

The custom resource is the contract between the user and the Operator. The CRD makes that contract storable and discoverable; the controller makes it active through API server requests.

This is the same desired-state pattern you saw with Deployments. A Deployment object is reconciled by built-in Kubernetes controllers. A custom resource is usually reconciled by a controller installed separately. Both controllers watch API state, compare desired and observed state, act, and repeat.

Because the API and controller are separate, the API server can accept a valid custom resource even while its controller is absent. The object will be stored, but no domain-specific automation will happen.

Four terms that should not blur together

extension layers · text

```
Part              What it provides                    Example
----------------  ----------------------------------  -------------------------
CRD               API type, schema, names, scope      Cache.apps.example.com
Custom resource   One stored instance of that type    Cache/storefront
Custom controller Reconciliation code for API objects Watches Cache objects
Operator          Controller plus operational knowledge Creates, heals, upgrades
```

A custom controller can reconcile built-in or custom resources. The Operator pattern is a more specific idea: software encodes the knowledge a human operator would use to install, configure, heal, upgrade, back up, or otherwise manage an application. In practice, an Operator commonly includes one or more CRDs plus a controller running in a Deployment.

A CRD is also different from Helm. Helm renders and submits manifests and records a release. An Operator remains active after installation and continues reconciling state. Helm or Kustomize may install an Operator, but they do not become its control loop.

A CRD extends API discovery

The CRD itself is a cluster-scoped apiextensions.k8s.io/v1 resource. Its spec tells the API server which group and versions to serve, how the resource is named, whether instances are namespaced, and which fields are valid. It cannot add a type to an existing built-in API group; custom resources live in their own API group.

cache-crd.yaml · yaml

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: caches.apps.example.com
spec:
  group: apps.example.com
  scope: Namespaced
```

```
  names:
    plural: caches
    singular: cache
    kind: Cache
    shortNames:
      - ca
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              required: [image, replicas]
              properties:
                image:
                  type: string
                replicas:
                  type: integer
                  minimum: 1
            status:
              type: object
              properties:
                readyReplicas:
                  type: integer
                  minimum: 0
                observedGeneration:
                  type: integer
      subresources:
        status: {}
      additionalPrinterColumns:
        - name: Ready
          type: integer
          jsonPath: .status.readyReplicas
        - name: Age
          type: date
          jsonPath: .metadata.creationTimestamp
```

Read the identity from the inside out. group plus version produces apps.example.com/v1. kind is the singular YAML kind Cache. plural is the REST resource name caches. The required CRD metadata name is plural.group, so it becomes caches.apps.example.com.

served means clients may use this version. storage selects the one version used when the API server persists objects; exactly one version must be the storage version. Converting between multiple versions requires additional conversion design, which is outside this administrator-focused lesson.

The structural OpenAPI schema lets the API server validate and prune data. Here image and replicas are required, and replicas cannot be less than one. The status subresource lets a controller update .status separately from the user's desired .spec. Printer columns make important status visible in normal kubectl get output.

Install the API before using it

Apply the CRD, then wait until its Established condition is true. This ordering matters in installation workflows: a custom resource cannot be accepted until the API server serves its type.

```bash
kubectl apply -f cache-crd.yaml

kubectl wait \
  --for=condition=Established \
  crd/caches.apps.example.com \
  --timeout=60s
```

```text
customresourcedefinition.apiextensions.k8s.io/caches.apps.example.com created
customresourcedefinition.apiextensions.k8s.io/caches.apps.example.com condition met
```

Established means the API registration is ready. It says nothing about the health or even the presence of a controller for Cache objects.

Let discovery answer naming and scope questions

After establishment, kubectl discovers the new resource from the API server. Use discovery instead of guessing its short name, group, or scope.

```bash
kubectl api-resources --api-group=apps.example.com

kubectl get crd caches.apps.example.com

kubectl explain cache.spec
kubectl explain cache.spec.replicas
```

```text
NAME     SHORTNAMES   APIVERSION           NAMESPACED   KIND
caches   ca           apps.example.com/v1   true         Cache
```

The discovery row connects CLI vocabulary to API identity. NAMESPACED true came from spec.scope. kubectl explain works because the CRD publishes a schema, so the same field-discovery habit used for built-in resources also works here.

Create one custom resource

A custom resource uses apiVersion and kind exactly as declared by the CRD. Because Cache is namespaced, create a namespace and place the object there.

```bash
kubectl create namespace operators-demo
```

storefront-cache.yaml · yaml

```yaml
apiVersion: apps.example.com/v1
kind: Cache
metadata:
  name: storefront
  namespace: operators-demo
spec:
  image: redis:7.4
  replicas: 2
```

```bash
kubectl apply -f storefront-cache.yaml

kubectl get caches.apps.example.com \
  --namespace operators-demo \
  --output wide

kubectl get cache storefront \
  --namespace operators-demo \
  --output yaml
```

```text
NAME         READY   AGE
storefront           8s
```

The blank READY column is meaningful. The API server validated and stored the object, but nothing has written status yet. A CRD provides storage, discovery, validation, and API behavior—not the application-specific act of creating Redis Pods.

If replicas were 0, the API server would reject the request because the schema minimum is 1. That validation happens before an Operator receives the object, which prevents controllers from having to interpret every malformed request themselves.

The Operator turns stored intent into behavior

Assume a Cache Operator is installed in the cache-operator-system namespace. Its controller uses a ServiceAccount and RBAC permissions to list and watch Cache objects, create or update the built-in resources it manages, and update Cache status.

When storefront appears, the controller's reconciliation loop follows a pattern rather than a one-time script.

1. Watch or list the Cache object from the API.
2. Compare spec.image and spec.replicas with the resources that currently exist.

3. Create or patch a Deployment, Service, ConfigMap, or other resources required by its design.
4. Observe the managed resources and write readyReplicas and observedGeneration to the Cache status subresource.
5. Repeat when the custom resource, a managed resource, or relevant cluster state changes.

Because reconciliation is continuous, deleting a managed Deployment may cause the Operator to recreate it. Editing that Deployment directly may also be overwritten if the custom resource remains the source of desired state. Change the custom resource unless the Operator's documentation says otherwise.

Observe a normal reconciliation

Start with the custom resource, then move to the controller and its managed resources. Names and labels are Operator-specific, so discover what the installed product actually uses.

```bash
kubectl describe cache storefront \
  --namespace operators-demo

kubectl get cache storefront \
  --namespace operators-demo \
  --output jsonpath='{.metadata.generation}{"\t"}{.status.observedGeneration}{"\t"}{.status.readyReplicas}{"\n"}'
```

```text
1   1   2
```

generation increases when desired state changes. observedGeneration is an Operator-defined status convention that commonly records the generation the controller has processed. Matching values plus readyReplicas 2 indicate that this controller observed the current spec and reports both requested replicas ready; always read the installed CRD and Operator documentation for exact status semantics.

```bash
kubectl get deployments,pods \
  --namespace cache-operator-system

kubectl logs \
  --namespace cache-operator-system \
  deployment/cache-operator \
  --tail=50

kubectl get deployment,service,pods \
  --namespace operators-demo \
  --show-labels
```

The first view answers whether the control loop is running. Logs expose its reconciliation decisions. The final view shows the built-in resources that realize the custom intent. Events from kubectl describe can add another time-ordered view when the Operator emits them.

Follow ownership instead of trusting names

Well-designed controllers commonly place an ownerReference on managed objects. This links a child object to the custom resource and can let Kubernetes garbage collection remove dependents when the owner is deleted. A matching name alone is not an ownership relationship.

```bash
kubectl get deployment storefront-cache \
  --namespace operators-demo \
  --output jsonpath='{range .metadata.ownerReferences[*]}{.apiVersion}{"  "}{.kind}{"  "}{.name}{"  controller="}{.controller}
{"\n"}{end}'
```

Expect the owner identity to point to apps.example.com/v1, kind Cache, name storefront when this Operator uses controller ownership. Some Operators intentionally manage external systems or use different relationships, so absence of an ownerReference is evidence to investigate, not automatic proof of failure.

Spec, status, and finalizers divide responsibilities

Users normally write .spec; controllers observe it and write .status. Enabling the status subresource reinforces that boundary and allows RBAC to grant update on caches/status separately from update on caches.

An Operator may add a finalizer to a custom resource when cleanup must finish before deletion—for example, deleting an external service. After a delete request, Kubernetes sets metadata.deletionTimestamp, but the object remains until the responsible controller removes its finalizer.

```bash
kubectl get cache storefront \
  --namespace operators-demo \
  --output jsonpath='{.metadata.deletionTimestamp}{"\n"}{.metadata.finalizers}{"\n"}'
```

A terminating custom resource with a finalizer is waiting for controller-driven cleanup. Check controller health and logs before considering manual finalizer removal; forcing removal can abandon external resources or skip required cleanup.

RBAC applies at both extension layers

Creating or changing a CRD is a cluster-level administrative action. Using namespaced custom resources can be delegated through a Role or ClusterRole and a RoleBinding, just like other namespaced resources. The Operator's ServiceAccount separately needs only the verbs and resources its reconciliation requires.

```bash
kubectl auth can-i create \
  caches.apps.example.com \
  --namespace operators-demo

kubectl auth can-i update \
  caches.apps.example.com \
  --subresource=status \
  --namespace operators-demo \
  --as system:serviceaccount:cache-operator-system:cache-operator
```

RBAC resource names use the plural REST resource and API group. In an RBAC rule, the status resource is written caches/status; kubectl auth can-i selects it with --subresource=status. Reuse the least-privilege model from post 22 rather than granting wildcard access to every present and future custom type.

An administrator's observation order

When working with an installed extension, follow the dependency chain. Each step answers a different question.

1. API: Does the CRD exist, report Established, and serve the expected group/version/resource?
2. Object: Is the custom resource in the intended namespace, and what do spec, status, Events, generation, and finalizers say?
3. Controller: Is the Operator Deployment available, and do its logs show successful reconciliation or authorization errors?
4. Result: Do managed Kubernetes or external resources exist, and are their ownership and health consistent with the custom resource?

```bash
kubectl get crd caches.apps.example.com
kubectl api-resources --api-group=apps.example.com
kubectl get caches.apps.example.com --all-namespaces
kubectl describe cache storefront --namespace operators-demo
kubectl get deployment,pods --namespace cache-operator-system
kubectl logs --namespace cache-operator-system deployment/cache-operator --tail=50
```

This order avoids a common mistake: studying application Pods while the API type is missing, the custom resource is invalid, or the controller itself is unavailable. Move from declared API to active control loop to realized resources.

Installation, upgrades, and deletion deserve care

Install CRDs before the controller and before custom resources that depend on them. For upgrades, follow the Operator vendor's supported version path: a new CRD schema, stored object version, controller image, and managed application version can have compatibility requirements that ordinary manifest ordering cannot infer.

Deleting a namespaced custom resource affects that one declared instance according to controller and garbage-collection behavior. Deleting the cluster-scoped CRD removes the API endpoint and all stored custom resources of that type across the cluster. Treat CRD deletion as data-destructive and inventory instances first.

```bash
kubectl get caches.apps.example.com --all-namespaces

kubectl get crd caches.apps.example.com \
  --output jsonpath='{range .spec.versions[*]}{.name}{"  served="}{.served}{"  storage="}{.storage}{"\n"}{end}'
```

The first command inventories instances. The second exposes served and storage versions before an upgrade. Neither command replaces the extension's backup and upgrade documentation, but both make cluster state explicit.

Common confusions resolved

quick distinctions · text

Confusion	Correct model
CRD = controller	No: it registers an API type only
Custom resource = running app	No: it is stored desired state
Operator = API server plugin	No: it is normally an API client running as Pods
Helm release = Operator loop	No: Helm installs/renders; Operator reconciles
Established CRD = healthy app	No: only the custom API is ready
Same object name = ownership	No: inspect ownerReferences and product behavior
Edit child = durable change	Often no: the Operator may restore declared state

Keep returning to the API-centered model. The CRD defines what may be declared. The custom resource records what is desired. The controller owns the transition toward that state. Status and managed resources are evidence of what happened.

Where this fits in the series

Posts 22–25 supplied the prerequisites: RBAC for controller identities, Helm and Kustomize for installation workflows, and standardized interfaces for runtime, network, and storage providers. This lesson covers API extension and reconciliation. Post 27 moves to the Kubernetes networking model, so packet paths and node-to-node connectivity are intentionally not developed here. Hands-on repair of CRDs and Operators belongs to Practice post 47.

Official documentation

Custom Resources explains custom resources, declarative APIs, and custom controllers.

Extend the Kubernetes API with CustomResourceDefinitions documents structural schemas, versions, printer columns, and status and scale subresources.

Operator pattern describes how custom resources and controllers encode operational knowledge.

kubectl api-resources is the command reference for discovering server-supported resource types.

kubectl explain documents schema-backed field discovery for supported API resources.

Using RBAC Authorization includes permission patterns for custom resources and subresources.