Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 6, 2026

Kubernetes Objects and YAML: Metadata, Spec, Status, Labels, Selectors, and Annotations

Read Kubernetes objects with confidence: separate metadata, spec, and status; use YAML, labels, selectors, and annotations in a clear administrator workflow.

This is Learn post 03 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. You already know how a request reaches the API server and how kubectl chooses its target, scope, and output. Now we will read the object inside that request.

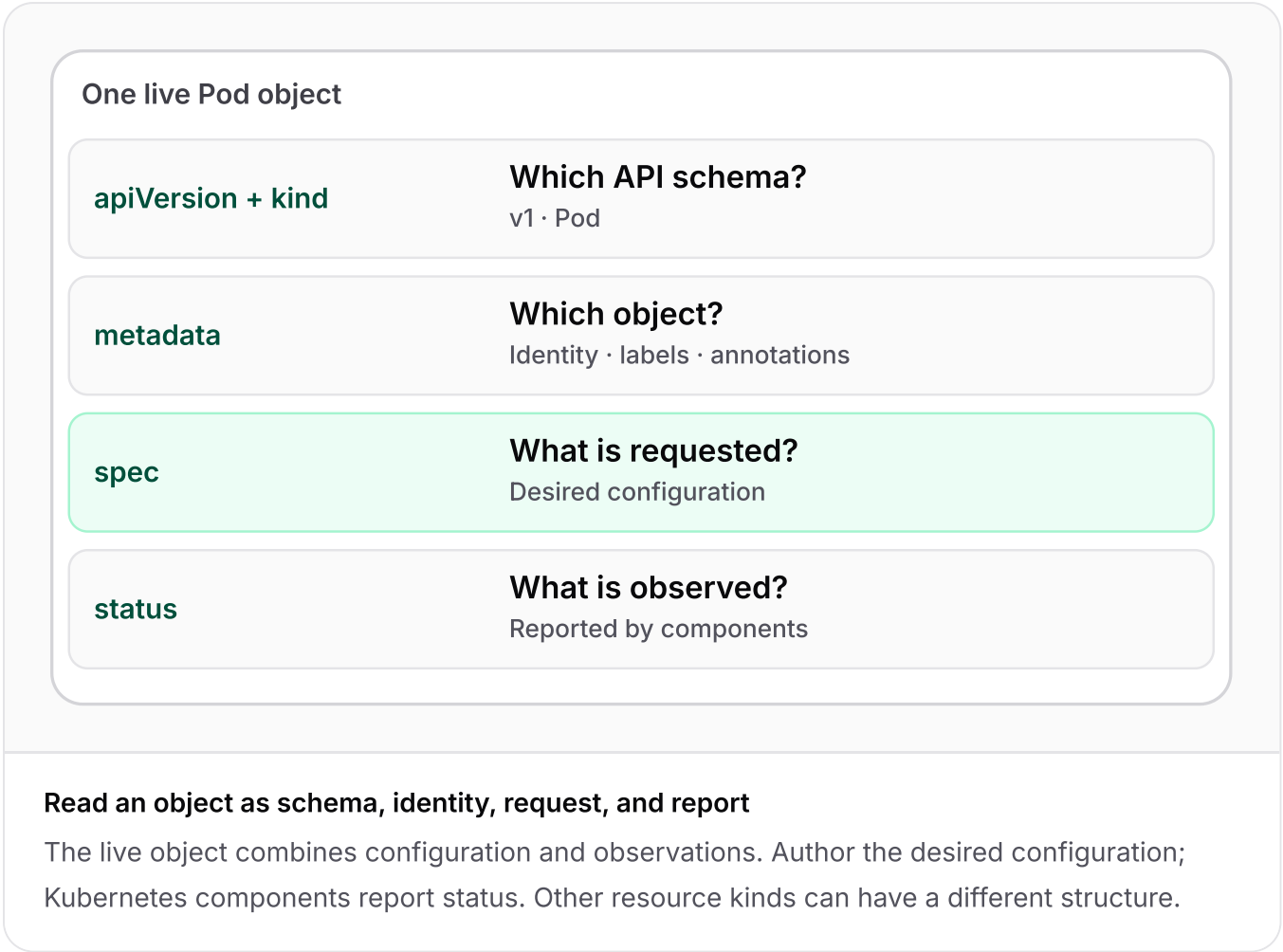
A manifest becomes much easier to understand when you can answer three questions: Which object is this? What are we asking Kubernetes to do? What has Kubernetes observed? Labels, selectors, and annotations then explain how administrators and components find and describe those objects.

What you'll learn

- Read a manifest as structured API data, including YAML maps, lists, and value types.
- Separate object identity and metadata from desired configuration and observed status.
- Generate, validate, apply, and inspect a small manifest with a clear purpose for each command.
- Use labels and selectors to identify a changing group of objects.
- Choose annotations for descriptive information and keep live edits consistent with your source file.

The mental model: identity, request, report

Think of a Kubernetes object as a shared record. Its identity tells clients which record to address. Its specification records the requested configuration. Its status records observations made by Kubernetes components.



Read an object as schema, identity, request, and report

The live object combines configuration and observations. Author the desired configuration; Kubernetes components report status. Other resource kinds can have a different structure.

The API server stores the accepted record. Components act on that stored configuration and report observations back through the API. Because this work happens asynchronously, accepting a request and achieving its desired result are separate events. The official object overview describes this spec/status distinction.

YAML is a way to express the record, not a sequence of instructions for Kubernetes to execute. A local file has no effect until a client submits its contents. Kubernetes continues working from the stored object after kubectl exits; it does not keep watching that local file.

Read one manifest from the outside in

We will use one standalone Pod named `web-demo` in a dedicated namespace, `objects-demo`. For this lesson, a Pod is the object that asks Kubernetes to run a container. Its lifecycle and multi-container behavior belong to post 04.

Here is the complete manifest used throughout the demonstration. Save it as `web-demo.yaml` when following the walkthrough.

web-demo.yaml · yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-demo
  namespace: objects-demo
  labels:
    app: catalog
    environment: demo
  annotations:
    example.com/purpose: "Observe object metadata and state"
spec:
  containers:
    - name: web
      image: nginx:stable-alpine
```

apiVersion and kind choose the schema

`kind: Pod` identifies the object kind, while `apiVersion: v1` selects its API version in the core API group. Other groups use a group/version spelling, such as `apps/v1` for Deployments. This is the object's API version, not the version of kubectl or the whole cluster.

The schema determines which fields are valid and what their values mean. A field valid for a Deployment is not automatically valid for a Pod. Use the discovery skills from post 02 when reading an unfamiliar field:

```bash
kubectl explain pod.spec.containers --api-version=v1
```

An administrator reaches for this command to confirm the field's structure and meaning against the connected cluster. Notice that `containers` is a list of container definitions, rather than a string containing an image name.

metadata identifies and describes the object

`metadata.name` is the Pod's API name. `metadata.namespace` scopes that name. For this resource type, `objects-demo/web-demo` and `another-namespace/web-demo` can identify different Pods. The container name `web` identifies an entry inside this Pod; it is not the Pod's name.

The server also assigns metadata. `uid` uniquely identifies this particular object instance: deleting and recreating a Pod with the same name produces a new UID (unique identifier). `resourceVersion` is an opaque version token used to coordinate reads and updates; do not interpret it as an application release number. `creationTimestamp` records creation time. Keep these server-managed values out of a fresh authoring manifest. See object names and IDs for the naming model.

Names select individual records. Labels let multiple records share attributes. Annotations hold additional information. Both label and annotation maps belong under `metadata`.

spec requests configuration; status reports observations

Here, `spec.containers` requests a container named `web` using the given image. The image tag is convenient for this demonstration, but it can point to a newer image later; the lesson does not depend on an exact NGINX release.

We omit `status` from the manifest. Kubernetes components populate it as work progresses. Writing a hoped-for status into YAML does not make a container run. The spec/status model is common, but the exact fields vary by kind; some resources, such as ConfigMaps, use other top-level fields and do not have this pair.

The useful administrator habit is to compare the request with the report, rather than assuming the report must already match the request.

YAML structure is part of the configuration

Indentation expresses nesting. In our manifest, `labels` is inside `metadata`, and `image` belongs to an item inside `spec.containers`. Moving a field changes its path and therefore its meaning or validity. Use spaces, not tabs, for indentation.

- A map associates keys with values: `app: catalog` is one entry in the labels map.
- A list uses `-` for each item. The `name` and `image` aligned beneath one item describe the same container.
- Scalars have types. Label and annotation values must be strings: quote values such as `"3"` or `"true"` when using them as metadata. Fields that require numbers or booleans should retain those types.
- `#` begins a comment outside a quoted string. Comments help readers but are not stored as object fields.
- `---` separates YAML documents when one file contains several objects. It does not make those objects one atomic operation.

Keep one occurrence of each key within a map. A duplicate `labels` key is not a way to add a second label; put both entries inside the same map. Valid YAML alone is insufficient: the parsed data must also satisfy the Kubernetes schema and the cluster's admission rules.

Worked demonstration: file, accepted object, observed state

This walkthrough assumes a learning cluster with a schedulable worker, image registry access, and permission to create a namespace and Pods. Cluster policy must permit this minimal Pod. The expected results below explain normal behavior; timings, generated fields, and runtime status depend on your cluster.

Establish the scope and generate a starting point

First, confirm the active context as introduced in post 02, then create the dedicated namespace. If you already created `objects-demo` for this walkthrough, reuse it.

```bash
kubectl config current-context
kubectl create namespace objects-demo
```

The first command identifies the request target. The second creates the namespace the manifest names. A `namespace` field on a Pod does not create that namespace automatically.

When starting a manifest from a familiar imperative command, an administrator can generate YAML without creating the Pod:

```bash
kubectl run web-demo --image=nginx:stable-alpine \
  --namespace=objects-demo \
  --labels=app=catalog,environment=demo \
  --dry-run=client -o yaml > web-demo.generated.yaml
```

`--dry-run=client` generates the object locally; `-o yaml` chooses the serialization and `>` saves it. This command makes no Pod creation request. Inspect the generated file to connect flags to fields. It may contain defaults or empty fields, and its container name comes from the command. Use the complete `web-demo.yaml` above for the remaining steps, including its annotation and container name. The kubectl run reference documents these flags.

Ask the server to validate, then apply

Before persisting the manifest, submit a server dry run:

```bash
kubectl apply --dry-run=server --validate=strict -f web-demo.yaml
```

This contacts the API server and runs the request through applicable validation and admission without persisting the requested change. A successful result means this request is acceptable now. It does not prove that a worker can pull the image or run the container, because no Pod is actually started by the dry run. See the kubectl apply options.

Now submit the same configuration for persistence, then read the resulting object:

```bash
kubectl apply -f web-demo.yaml
kubectl get pod web-demo -n objects-demo -o yaml
```

For a new object, apply reports `pod/web-demo created`. Reapplying the same file can report `unchanged`; changing a managed field can report `configured`. These messages describe the API update, not application health.

Compared with the authoring file, the readback contains server-assigned metadata, defaulted fields, and runtime observations. Default client-side apply also stores a `kubectl.kubernetes.io/last-applied-configuration` annotation so later applies can calculate changes. It uses the file, that saved configuration, and the live object; it does not blindly replace the entire live object. See declarative object management.

Read the request beside the report

A compact view helps distinguish what was requested from what has been observed:

```bash
kubectl get pod web-demo -n objects-demo \
  -o custom-columns='NAME:.metadata.name,IMAGE:.spec.containers[0].image,PHASE:.status.phase'
```

`NAME` comes from identity, `IMAGE` from desired configuration, and `PHASE` from observed status. In a normal startup, the phase may initially be `Pending` and later become `Running`. The image field remains the requested image while startup progresses. A phase is a summary, not proof of application readiness; posts 04 and 05 explain lifecycle and health checks.

A live YAML readback is therefore useful evidence, but not a clean template to copy wholesale. Start reusable configuration from deliberate fields you understand, rather than preserving a previous object's UID, resource version, timestamps, or status.

Labels are attributes; selectors ask which objects match

The Pod has two labels: `app=catalog` and `environment=demo`. These keys express our convention. Naming a Pod `catalog` would not automatically attach an `app=catalog` label, and using the same label on several Pods is allowed.

Administrators use labels to work with groups whose individual names can change. To see the attributes on this namespace's Pods:

```bash
kubectl get pods -n objects-demo --show-labels
```

Look for both configured key/value pairs on `web-demo`; their display order is not significant. To select just the catalog Pods in the demo environment:

```bash
kubectl get pods -n objects-demo \
  -l 'app=catalog,environment=demo' -o name
```

With only our example present, the result is `pod/web-demo`. The comma means AND: both requirements must hold. The namespace still limits the search. A selector is a condition evaluated against current labels, not a saved list of Pod names. The labels and selectors guide defines the matching rules.

Use sets and existence when equality is too narrow

For a view that includes several environment values, keep the application constraint and widen the allowed set:

```bash
kubectl get pods -n objects-demo \
  -l 'app=catalog,environment in (demo,staging)' -o name
```

This still matches `web-demo`: its app is catalog AND its environment is either demo OR staging. Quote the expression so shell punctuation reaches kubectl intact. Multiple comma-separated requirements remain AND; this syntax does not provide arbitrary OR between different keys.

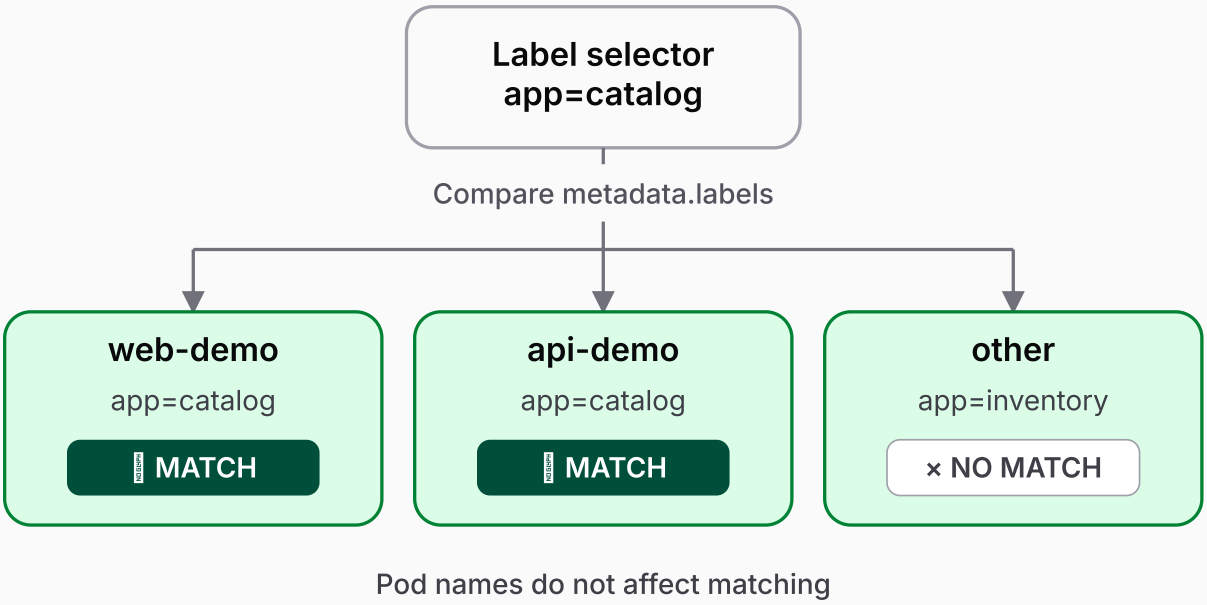
A bare key such as `environment` requires that label to exist; `!environment` requires it to be absent. Negative conditions such as `environment!=production` and `environment notin (production)` also match objects missing that key. To exclude unlabeled objects as well, include existence:

```bash
kubectl get pods -n objects-demo \
  -l 'app=catalog,environment,environment notin (production)' -o name
```

Our Pod matches all three requirements. This pattern is useful when an administrator means “explicitly classified outside production,” rather than including unclassified objects.

Resource selectors connect objects through these same attributes

A query selector filters a read. A selector stored in another object's configuration can determine which Pods a component manages or targets. The selecting object's own labels do not substitute for that selector.



A selector matches labels, not Pod names

Each arrow means “compare labels.” Both catalog Pods match regardless of name; the inventory Pod does not. Only web-demo is created in this walkthrough.

This diagram is conceptual; only `web-demo` is created in our walkthrough. Sharing a label does not by itself establish ownership or start a controller.

Selector YAML depends on the resource. A Deployment uses `spec.selector.matchLabels` and can use `matchExpressions`; its Pod labels are under `spec.template.metadata.labels`. The following is a reading fragment, not a complete manifest to apply:

Deployment selector and Pod-template fragment · yaml

```yaml
spec:
  selector:
    matchLabels:
      app: catalog
    matchExpressions:
      - key: environment
        operator: In
        values:
          - demo
          - staging
  template:
    metadata:
      labels:
        app: catalog
        environment: demo
```

The template labels satisfy both selector requirements. `matchLabels` expresses equality; the expression permits a set of values. Labels on the Deployment's outer `metadata` would describe the Deployment itself. They are not automatically copied into the Pod template.

A Service instead uses a direct equality map under `spec.selector`. Do not copy one resource's selector shape into another. Deployment selectors must match their template labels and are immutable after creation in `apps/v1`. Post 06 teaches those controllers; post 28 teaches how Services use selectors. For now, remember where the labels live and which object reads them.

A label selector also differs from a field selector. `-l` reads labels; `--field-selector` filters supported object fields, such as `metadata.name`. It cannot query every arbitrary YAML path. See field selectors.

Annotations carry notes and tool information

Our `example.com/purpose` annotation explains why the Pod exists. Use annotations for descriptions, build information, or tool-specific metadata that does not belong in a label selector. They can contain text that label values cannot, including spaces. Both keys and

values are still strings; an embedded structured value must be serialized as a string.

Labels and annotations can use a domain prefix to distinguish who defines a key. `example.com/` is illustrative here; use your organization's domain for shared tooling. Reserve Kubernetes-owned prefixes for their documented uses. Short unprefixed labels such as `app` are adequate for this demonstration.

Annotations are not necessarily inert comments. A tool can assign behavior to a documented annotation, as kubectl does with its last-applied configuration. An arbitrary annotation has no automatic operational effect without a consumer. `kubectl get -l` does not select annotation values. See the annotations guide.

Update metadata and observe the consequence

An administrator can change a live label without recreating the Pod. Here we reclassify the same object from demo to staging:

```bash
kubectl label pod web-demo -n objects-demo \
  environment=staging --overwrite
kubectl get pods -n objects-demo -l 'app=catalog,environment=demo' -o name
kubectl get pods -n objects-demo -l 'app=catalog,environment=staging' -o name
```

`--overwrite` permits changing an existing label value. The demo query now returns no matching names; the staging query returns `pod/web-demo`. The Pod did not move namespaces or acquire a new identity. Because its label changed, the matching set changed. In a workload selected by other components, that same metadata edit can affect which components target it.

To update the explanation recorded on that object:

```bash
kubectl annotate pod web-demo -n objects-demo \
  example.com/purpose='Observe selector membership in staging' --overwrite
kubectl get pod web-demo -n objects-demo -o yaml
```

Inspect `metadata.annotations` for the new value and `metadata.labels` for staging. The annotation edit does not change the label query's result. For later reference, removing a label or annotation uses its key followed by `-`; the label and annotate references document that syntax.

These commands changed the live object, not `web-demo.yaml`. To keep staging as the intended configuration, edit that file's `metadata.labels.environment` to `staging` and its purpose annotation to `"Observe selector membership in staging"`, then apply and inspect:

```bash
kubectl apply -f web-demo.yaml
kubectl get pods -n objects-demo -L app,environment
```

The selected label columns should show `catalog` and `staging`. If you instead reapplied the original file, its explicitly configured demo label and original annotation would restore those values. Keep the file and intentional live edits aligned so the next apply remains predictable.

Remove the demonstration object

Once the walkthrough is complete, an administrator can remove precisely the object described by the file:

```bash
kubectl delete -f web-demo.yaml
```

This deletes `web-demo` in `objects-demo`; it does not remove the namespace. If you created that namespace only for this walkthrough and it contains nothing you need, remove it too:

```bash
kubectl delete namespace objects-demo
```

Namespace deletion also deletes resources inside it. The explicit cleanup is why the walkthrough uses a dedicated namespace.

Common mistakes to recognize

- Treating `status` as an instruction: configure desired fields and read status as evidence from the system.
- Assuming YAML validity implies API validity: check nesting, types, the kind's schema, and server acceptance separately.
- Confusing names with labels: `web-demo`, container `web`, and label `app=catalog` have different roles.
- Assuming selectors are permanent membership: changing labels changes matches, even when the object's name stays the same.
- Copying a selector between resource kinds: inspect the supported selector shape and the actual target labels.
- Changing a live object and forgetting its manifest: a later apply can restore the file's configured values. Also, apply does not bypass immutable-field restrictions.

Where this fits in the CKA path

Post 01: Kubernetes architecture explained the request path. Post 02: kubectl foundations established how to target and inspect API resources. This lesson connects those foundations to the fields inside an object.

Next, post 04 examines Pod lifecycle, init containers, and multi-container Pods. Post 05 develops health checks, post 06 develops ReplicaSets and Deployments, and post 28 develops Services and EndpointSlices. Those lessons will reuse this distinction between a configuration, the labels attached to its objects, the selectors that find them, and the status that reports progress.

What to remember

- Read identity, request, and report separately: `metadata`, `spec`, and `status` answer different questions.
- YAML describes structured data. Field paths and value types matter as much as spelling.
- Generate a starting point, review its fields, validate with the server, apply, then observe the live object.
- Labels describe attributes; selectors evaluate them. Annotations store additional information for people and tools.
- A manifest and a live object are separate records. Keep intended edits in the manifest and leave generated observations to Kubernetes.