Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 2, 2026

kubectl Foundations: Contexts, Namespaces, Resource Discovery, and Output

Build a reliable kubectl workflow for CKA administration: verify the active context, control namespace scope, discover the API resources your cluster serves, and choose output that answers the question at hand.



kubectl is concise, but every command carries several decisions: which cluster receives the request, which identity is used, which namespace limits the request, which API resource the command names, and how the response is displayed. If any one of those

decisions is wrong, a valid command can show the wrong objects—or change the wrong cluster.

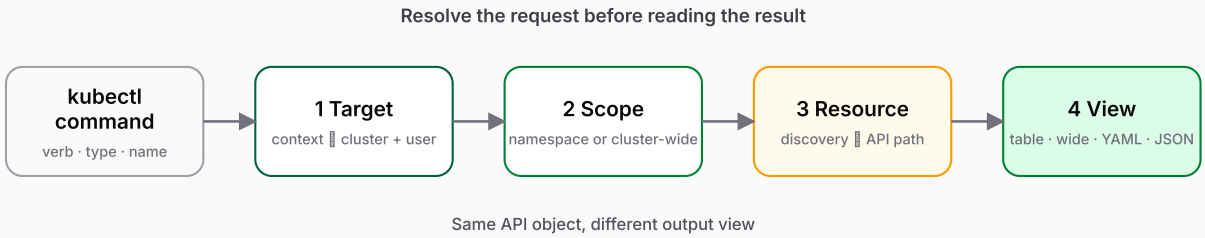
The previous lesson established that kubectl is an API client. This lesson turns that architecture into an administrator workflow: resolve the target and scope, ask the server what it supports, then choose a useful view of the returned API data.

What you'll learn

- How kubeconfig contexts connect a cluster, a user, and an optional default namespace.
- How request flags override context defaults without changing what a Kubernetes resource is.
- How to distinguish namespaced resources from cluster-scoped resources.
- How api-resources and explain turn the live API into a searchable reference.
- When to use the default table, wide output, YAML, JSON, names, custom columns, and JSONPath.

The mental model: target, scope, resource, view

> Before trusting a kubectl result, answer four questions: Which target? Which scope? Which resource? Which view?



Resolve a kubectl request in four decisions

Target, scope, and resource determine the API request. View determines how kubectl presents the response; it does not change the stored object.

The first three decisions determine which object or collection the request addresses. The fourth selects the response and printing representation without changing the stored object. A default table can be produced with server-side printing, while YAML or JSON exposes the structured object representation.

This sequence is useful both before a change and when a result looks surprising. Check the target before asking why an object is missing. Check the scope before assuming it does not exist. Check discovery before guessing a resource name. Check the view before assuming an omitted field has no value.

Read a kubectl command as a request

Most resource commands follow one reusable shape:

```bash
kubectl [command] [TYPE] [NAME] [flags]

# Example
kubectl get pods -n kube-system -o wide
```

- **Command:** get expresses the operation. Here it asks the API to read a collection.
- **Type:** pods names the API resource. kubectl also accepts server-advertised short names such as po for interactive use.
- **Name:** an optional object name narrows a collection request to one object.
- **Flags:** -n kube-system overrides namespace scope for this command, while -o wide selects a more detailed table.

Because kubectl translates this syntax into an API request, command-line aliases are client conveniences, not Kubernetes object fields. The next lesson teaches the object structure used in manifests; keep CLI vocabulary and manifest vocabulary separate.

Contexts choose the target

A kubeconfig context is a named grouping of three access parameters: a cluster, a user, and an optional namespace. The cluster entry supplies the API server connection information. The user entry supplies an authentication method. The namespace is a default scope for namespaced requests. The context ties those entries together; it is not another name for the cluster.

Where kubectl loads configuration

1. If --kubeconfig is set, kubectl loads only that file.
2. Otherwise, if KUBECONFIG is set, kubectl merges the listed files according to kubeconfig merge rules.
3. Otherwise, it uses the default kubeconfig file at $HOME/.kube/config.

> Treat kubeconfig files like executable configuration: use only trusted files. A malicious kubeconfig can reference credential plugins or files and can expose data or execute code.

Inspect before you act

Use these commands whenever you inherit a shell, switch environments, or are about to make a consequential change:

```bash
kubectl config current-context
kubectl config get-contexts
kubectl config view --minify
```

Representative context list · plaintext

```
CURRENT   NAME      CLUSTER     AUTHINFO    NAMESPACE
*         cka-admin   lab-cluster   cka-admin   team-a
          staging     staging       deployer    default
```

The asterisk marks current-context. view --minify removes entries unrelated to that context, making the selected server, user reference, and namespace easier to verify. Avoid adding --raw when sharing output: raw mode can expose certificate data and secrets.

Persistent selection versus one-command override

```bash
# Persistently change current-context in kubeconfig
kubectl config use-context staging

# Use another context for only this command
kubectl --context=cka-admin get namespaces

# Set the default namespace on the current context
kubectl config set-context --current --namespace=team-a

# Verify the current context's namespace
kubectl config view --minify -o jsonpath='{..namespace}{"\n"}'
```

use-context mutates the kubeconfig's current-context, so later commands inherit the change. --context overrides selection for one process and leaves current-context untouched. Likewise, set-context --current --namespace changes a persistent default; -n changes only one request.

Namespaces choose the scope

A namespace partitions the names of namespaced resources inside one cluster. A Pod named web can exist in team-a and another Pod named web can exist in team-b because namespace is part of a namespaced object's identity. Namespaces do not nest.

Namespace scope is not, by itself, a complete security or network-isolation boundary. Role-Based Access Control, resource quotas, and NetworkPolicies add different controls and have dedicated later lessons. Here, focus on naming and request scope.

Namespaced and cluster-scoped resources

- **Namespaced resources** belong to exactly one namespace. Pods, Services, Deployments, and ConfigMaps are common examples.
- **Cluster-scoped resources** belong to the cluster rather than a namespace. Nodes, PersistentVolumes, and Namespace objects are examples. A Namespace is therefore not inside itself or inside another Namespace.

```bash
kubectl api-resources --namespaced=true
kubectl api-resources --namespaced=false
```

Do not memorize scope when the server can tell you. These commands query discovery and separate the resources the current API server exposes by their NAMESPACED property. This also works for extension resources you have never seen before.

Default, explicit, and all-namespace reads

```bash
# Current context's namespace, or default if none is set
kubectl get pods

# One-command namespace override
kubectl get pods --namespace=kube-system
kubectl get pods -n kube-system

# List this resource type across every namespace
kubectl get pods --all-namespaces
kubectl get pods -A

# Cluster-scoped: no namespace selection is needed
kubectl get nodes
```

For a namespaced get, omitting -n uses the context's namespace and falls back to default when the context has no namespace. -n selects one namespace for that request. -A lists across namespaces and ignores the current context's namespace. Cluster-

scoped resources have no namespace segment in their API identity, so changing namespace cannot move or duplicate them.

Resource discovery gives you the cluster's vocabulary

Kubernetes is extensible, so a static list of resource types is never the whole truth. kubectl discovers the API groups, versions, resources, short names, scopes, and supported verbs advertised by the selected API server. That makes the live cluster the authoritative catalog.

api-resources answers “what can I address?”

bash

```
kubectl api-resources
kubectl api-resources -o wide
kubectl api-resources --api-group=apps
kubectl api-resources --verbs=get,list --namespaced=true
kubectl api-versions
```

Representative api-resources rows · plaintext

```
NAME          SHORTNAMES   APIVERSION   NAMESPACED   KIND
configmaps    cm           v1           true         ConfigMap
namespaces    ns           v1           false        Namespace
nodes         no           v1           false        Node
pods          po           v1           true         Pod
deployments   deploy       apps/v1      true         Deployment
```

NAME is the plural resource name accepted as TYPE. SHORTNAMES are kubectl aliases. APIVERSION identifies the served group/version. NAMESPACED tells you whether -n can select its scope. KIND is the singular object kind used in API representations. Exact rows depend on the selected cluster and its installed extensions.

Because discovery comes from the active server, switching context can change the catalog. If one cluster has an extension API and another does not, the same kubectl binary can recognize a resource in the first context and reject it in the second.

explain answers “what fields does it have?”

```bash
kubectl explain pods
kubectl explain pods.spec.containers
kubectl explain deployments --api-version=apps/v1
```

explain retrieves field descriptions and structure from the server's OpenAPI information. Use api-resources to find the resource and its group; use explain to inspect the resource schema. Post 03 will use this workflow while teaching apiVersion, kind, metadata, spec, status, and manifest fields in depth.

Output is a lens over API data

The default get output is a compact, human-oriented table. It contains useful columns, but it is not the complete object. Choose output according to the question you need to answer rather than using one format for everything.

- **Default table:** scan routine state quickly.
- **-o wide:** add resource-specific columns while keeping a readable table.
- **-o yaml or -o json:** inspect the structured API representation and fields omitted from tables.
- **-o name:** return stable resource/name identifiers with no table decoration.
- **custom-columns:** build a small readable report from selected fields.
- **JSONPath:** extract or format exact fields for concise commands and scripts.

One resource, several useful views

A Namespace is cluster-scoped and present on a normal Kubernetes cluster, so it provides a focused output demonstration without introducing workload behavior:

bash

```
# Human-readable summary
kubectl get namespace kube-system

# Structured API representations
kubectl get namespace kube-system -o yaml
kubectl get namespace kube-system -o json

# Stable identifier
kubectl get namespace kube-system -o name

# A two-column report
kubectl get namespace kube-system \
  -o custom-columns='NAME:.metadata.name,PHASE:.status.phase'

# One exact value; add a newline for a clean shell prompt
kubectl get namespace kube-system \
  -o jsonpath='{.metadata.name}{"\n"}'
```

Selected results · plaintext

```
namespace/kube-system

NAME          PHASE
kube-system   Active

kube-system
```

Each command targets the same Namespace; only the requested output representation changes. YAML and JSON expose nested fields, while custom columns and JSONPath navigate those same field paths. The next lesson explains what the important object fields mean.

Single objects and lists have different roots

A named get returns one object, so .metadata.name starts at that object. A collection get returns a List whose objects are under .items. This root difference explains many empty JSONPath results.

```bash
# One object
kubectl get namespace kube-system \
  -o jsonpath='{.metadata.name}{"\n"}'

# A list: iterate over .items
kubectl get namespaces \
  -o jsonpath='{range .items[*]}{.metadata.name}{"\n"}{end}'

# Sort a list by a field before printing
kubectl get namespaces --sort-by=.metadata.name
```

> For automation, do not parse the spacing of the default human-readable table. Request -o name, JSON, custom columns, JSONPath, or another documented machine-oriented format.

Worked demonstration: establish what a command really means

This read-only sequence works on a normal cluster and applies the four-question model without creating or changing resources. Exact object names and rows vary by cluster.

1. Confirm the target

```bash
kubectl config current-context
kubectl config view --minify
```

Notice the context name, server under clusters, user reference, and optional namespace. Those values answer where the next request will go and which configured identity and default scope it will use.

2. Compare namespace scopes

```bash
kubectl get pods
kubectl get pods -n kube-system
kubectl get pods -A
```

The resource type stays Pods, but the collection scope changes. The first command uses the context default, the second targets kube-system, and the third lists across namespaces. In -A output, the NAMESPACE column becomes essential because object names alone are no longer sufficient identifiers.

3. Ask discovery instead of guessing

```bash
kubectl api-resources --namespaced=false --sort-by=name
kubectl explain namespaces
```

Find namespaces in the cluster-scoped catalog, then let explain identify the resource and show its schema description. Discovery tells you that namespace selection does not apply to Namespace objects.

4. Select the smallest useful view

```bash
kubectl get namespaces
kubectl get namespaces -o name
kubectl get namespaces \
  -o custom-columns='NAME:.metadata.name,PHASE:.status.phase'
```

The first view is convenient for a person, the second produces identifiers suitable for composition with other commands, and the third states exactly which two fields matter. None of them changes the Namespace objects.

Official documentation

- Organizing Cluster Access Using kubeconfig Files
- Namespaces
- kubectl api-resources
- kubectl explain
- kubectl command and output reference
- JSONPath Support