

Custom Resources

Custom resources are extensions of the Kubernetes API. This page discusses when to add a custom resource to your Kubernetes cluster and when to use a standalone service. It describes the two methods for adding custom resources and how to choose between them.

Custom resources

A *resource* is an endpoint in the [Kubernetes API](#) that stores a collection of [API objects](#) of a certain kind; for example, the built-in *pods* resource contains a collection of Pod objects.

A *custom resource* is an extension of the Kubernetes API that is not necessarily available in a default Kubernetes installation. It represents a customization of a particular Kubernetes installation. However, many core Kubernetes functions are now built using custom resources, making Kubernetes more modular.

Custom resources can appear and disappear in a running cluster through dynamic registration, and cluster admins can update custom resources independently of the cluster itself. Once a custom resource is installed, users can create and access its objects using `kubectl`, just as they do for built-in resources like *Pods*.

Custom controllers

On their own, custom resources let you store and retrieve structured data. When you combine a custom resource with a *custom controller*, custom resources provide a true *declarative API*.

The Kubernetes [declarative API](#) enforces a separation of responsibilities. You declare the desired state of your resource. The Kubernetes controller keeps the current state of Kubernetes objects in sync with your declared desired state. This is in contrast to an imperative API, where you *instruct* a server what to do.

You can deploy and update a custom controller on a running cluster, independently of the cluster's lifecycle. Custom controllers can work with any kind of resource, but they are especially effective when combined with custom resources. The [Operator pattern](#) combines custom resources and custom controllers. You can use custom controllers to encode domain knowledge for specific applications into an extension of the Kubernetes API.

Should I add a custom resource to my Kubernetes cluster?

When creating a new API, consider whether to [aggregate your API with the Kubernetes cluster APIs](#) or let your API stand alone.

Consider API aggregation if:	Prefer a stand-alone API if:
Your API is Declarative .	Your API does not fit the Declarative model.
You want your new types to be readable and writable using <code>kubectl</code> .	<code>kubectl</code> support is not required

Consider API aggregation if:	Prefer a stand-alone API if:
You want to view your new types in a Kubernetes UI, such as dashboard, alongside built-in types.	Kubernetes UI support is not required.
You are developing a new API.	You already have a program that serves your API and works well.
You are willing to accept the format restriction that Kubernetes puts on REST resource paths, such as API Groups and Namespaces. (See the API Overview .)	You need to have specific REST paths to be compatible with an already defined REST API.
Your resources are naturally scoped to a cluster or namespaces of a cluster.	Cluster or namespace scoped resources are a poor fit; you need control over the specifics of resource paths.
You want to reuse Kubernetes API support features .	You don't need those features.

Declarative APIs

In a Declarative API, typically:

- Your API consists of a relatively small number of relatively small objects (resources).
- The objects define configuration of applications or infrastructure.
- The objects are updated relatively infrequently.
- Humans often need to read and write the objects.
- The main operations on the objects are CRUD-y (creating, reading, updating and deleting).
- Transactions across objects are not required: the API represents a desired state, not an exact state.

Imperative APIs are not declarative. Signs that your API might not be declarative include:

- The client says "do this", and then gets a synchronous response back when it is done.
- The client says "do this", and then gets an operation ID back, and has to check a separate Operation object to determine completion of the request.
- You talk about Remote Procedure Calls (RPCs).
- Directly storing large amounts of data; for example, > a few kB per object, or > 1000s of objects.
- High bandwidth access (10s of requests per second sustained) needed.
- Store end-user data (such as images, PII, etc.) or other large-scale data processed by applications.
- The natural operations on the objects are not CRUD-y.
- The API is not easily modeled as objects.
- You chose to represent pending operations with an operation ID or an operation object.

Should I use a ConfigMap or a custom resource?

Use a ConfigMap if any of the following apply:

- There is an existing, well-documented configuration file format, such as a `mysql.cnf` or `pom.xml`.
- You want to put the entire configuration into one key of a ConfigMap.
- The main use of the configuration file is for a program running in a Pod on your cluster to consume the file to configure itself.

- Consumers of the file prefer to consume via file in a Pod or environment variable in a pod, rather than the Kubernetes API.
- You want to perform rolling updates via Deployment, etc., when the file is updated.

Note: Use a [Secret](#) for sensitive data, which is similar to a ConfigMap but more secure.

Use a custom resource (CRD or Aggregated API) if most of the following apply:

- You want to use Kubernetes client libraries and CLIs to create and update the new resource.
- You want top-level support from `kubectl`; for example, `kubectl get my-object object-name`.
- You want to build new automation that watches for updates on the new object, and then CRUD other objects, or vice versa.
- You want to write automation that handles updates to the object.
- You want to use Kubernetes API conventions like `.spec`, `.status`, and `.metadata`.
- You want the object to be an abstraction over a collection of controlled resources, or a summarization of other resources.

Adding custom resources

Kubernetes provides two ways to add custom resources to your cluster:

- CRDs are simple and can be created without any programming.
- [API Aggregation](#) requires programming, but allows more control over API behaviors like how data is stored and conversion between API versions.

Kubernetes provides these two options to meet the needs of different users, so that neither ease of use nor flexibility is compromised.

Aggregated APIs are subordinate API servers that sit behind the primary API server, which acts as a proxy. This arrangement is called [API Aggregation](#) (AA). To users, the Kubernetes API appears extended.

CRDs allow users to create new types of resources without adding another API server. You do not need to understand API Aggregation to use CRDs.

Regardless of how they are installed, the new resources are referred to as Custom Resources to distinguish them from built-in Kubernetes resources (like pods).

Note:

Avoid using a Custom Resource as data storage for application, end user, or monitoring data: architecture designs that store application data within the Kubernetes API typically represent a design that is too closely coupled.

Architecturally, [cloud native](#) application architectures favor loose coupling between components. If part of your workload requires a backing service for its routine operation, run that backing service as a component or consume it as an external service. This way, your workload does not rely on the Kubernetes API for its normal operation.

CustomResourceDefinitions

The [CustomResourceDefinition](#) API resource allows you to define custom resources. Defining a CRD object creates a new custom resource with a name and schema that you specify. The Kubernetes API serves and handles the storage of your custom resource. The name of a CRD object must be a valid [DNS subdomain name](#).

This frees you from writing your own API server to handle the custom resource, but the generic nature of the implementation means you have less flexibility than with [API server aggregation](#).

Refer to the [custom controller example](#) for an example of how to register a new custom resource, work with instances of your new resource type, and use a controller to handle events.

API server aggregation

Usually, each resource in the Kubernetes API requires code that handles REST requests and manages persistent storage of objects. The main Kubernetes API server handles built-in resources like *Pods* and *Services*, and can also generically handle custom resources through [CRDs](#).

The [aggregation layer](#) allows you to provide specialized implementations for your custom resources by writing and deploying your own API server. The main API server delegates requests to your API server for the custom resources that you handle, making them available to all of its clients.

Choosing a method for adding custom resources

CRDs are easier to use. Aggregated APIs are more flexible. Choose the method that best meets your needs.

Typically, CRDs are a good fit if:

- You have a handful of fields
- You are using the resource within your company, or as part of a small open-source project (as opposed to a commercial product)

Comparing ease of use

CRDs are easier to create than Aggregated APIs.

CRDs	Aggregated API
Do not require programming. Users can choose any language for a CRD controller.	Requires programming and building binary and image.
No additional service to run; CRDs are handled by API server.	An additional service to create and that could fail.
No ongoing support once the CRD is created. Any bug fixes are picked up as part of normal Kubernetes Master upgrades.	May need to periodically pickup bug fixes from upstream and rebuild and update the Aggregated API server.
No need to handle multiple versions of your API; for example, when you control the client for this resource, you can upgrade it in sync with the API.	You need to handle multiple versions of your API; for example, when developing an extension to share with the world.

Advanced features and flexibility

Aggregated APIs offer more advanced API features and customization of other features; for example, the storage layer.

Feature	Description	CRDs	Aggregated API
Validation	Help users prevent errors and allow you to evolve your API independently of your clients. These features are most useful when there are many clients who can't all update at the same time.	Yes. Most validation can be specified in the CRD using OpenAPI v3.0 validation . Any other validations supported by addition of a Validating Webhook .	Yes, arbitrary validation checks
Defaulting	See above	Yes, either via OpenAPI v3.0 validation default keyword (GA in 1.17), or via a Mutating Webhook (though this will not be run when reading from etcd for old objects).	Yes
Multi-versioning	Allows serving the same object through two API versions. Can help ease API changes like renaming fields. Less important if you control your client versions.	Yes	Yes
Custom Storage	If you need storage with a different performance mode (for example, a time-series database instead of key-value store) or isolation for security (for example, encryption of sensitive information, etc.)	No	Yes
Custom Business Logic	Perform arbitrary checks or actions when creating, reading, updating or deleting an object	Yes, using Webhooks .	Yes
Scale Subresource	Allows systems like HorizontalPodAutoscaler and PodDisruptionBudget interact with your new resource	Yes	Yes
Status Subresource	Allows fine-grained access control where user writes the spec section and the controller writes the status section. Allows incrementing object Generation on custom resource data mutation (requires separate spec and status sections in the resource)	Yes	Yes
Other Subresources	Add operations other than CRUD, such as "logs" or "exec".	No	Yes

Feature	Description	CRDs	Aggregated API
strategic-merge-patch	The new endpoints support PATCH with <code>Content-Type: application/strategic-merge-patch+json</code> . Useful for updating objects that may be modified both locally, and by the server. For more information, see "Update API Objects in Place Using kubectl patch"	No	Yes
Protocol Buffers	The new resource supports clients that want to use Protocol Buffers	No	Yes
OpenAPI Schema	Is there an OpenAPI (swagger) schema for the types that can be dynamically fetched from the server? Is the user protected from misspelling field names by ensuring only allowed fields are set? Are types enforced (in other words, don't put an <code>int</code> in a <code>string</code> field?)	Yes, based on the OpenAPI v3.0 validation schema (GA in 1.16).	Yes

Common Features

When you create a custom resource, either via a CRD or an AA, you get many features for your API, compared to implementing it outside the Kubernetes platform:

Feature	What it does
CRUD	The new endpoints support CRUD basic operations via HTTP and <code>kubectl</code>
Watch	The new endpoints support Kubernetes Watch operations via HTTP
Discovery	Clients like <code>kubectl</code> and dashboard automatically offer list, display, and field edit operations on your resources
json-patch	The new endpoints support PATCH with <code>Content-Type: application/json-patch+json</code>
merge-patch	The new endpoints support PATCH with <code>Content-Type: application/merge-patch+json</code>
HTTPS	The new endpoints uses HTTPS
Built-in Authentication	Access to the extension uses the core API server (aggregation layer) for authentication
Built-in Authorization	Access to the extension can reuse the authorization used by the core API server; for example, RBAC.
Finalizers	Block deletion of extension resources until external cleanup happens.

Feature	What it does
Admission Webhooks	Set default values and validate extension resources during any create/update/delete operation.
UI/CLI Display	Kubect1, dashboard can display extension resources.
Unset versus Empty	Clients can distinguish unset fields from zero-valued fields.
Client Libraries Generation	Kubernetes provides generic client libraries, as well as tools to generate type-specific client libraries.
Labels and annotations	Common metadata across objects that tools know how to edit for core and custom resources.

Preparing to install a custom resource

There are several points to be aware of before adding a custom resource to your cluster.

Third party code and new points of failure

While creating a CRD does not automatically add any new points of failure (for example, by causing third party code to run on your API server), packages (for example, Charts) or other installation bundles often include CRDs as well as a Deployment of third-party code that implements the business logic for a new custom resource.

Installing an Aggregated API server always involves running a new Deployment.

Storage

Custom resources consume storage space in the same way that ConfigMaps do. Creating too many custom resources may overload your API server's storage space.

Aggregated API servers may use the same storage as the main API server, in which case the same warning applies.

Authentication, authorization, and auditing

CRDs always use the same authentication, authorization, and audit logging as the built-in resources of your API server.

If you use RBAC for authorization, most RBAC roles will not grant access to the new resources (except the cluster-admin role or any role created with wildcard rules). You'll need to explicitly grant access to the new resources. CRDs and Aggregated APIs often come bundled with new role definitions for the types they add.

Aggregated API servers may or may not use the same authentication, authorization, and auditing as the primary API server.

Accessing a custom resource

Kubernetes [client libraries](#) can be used to access custom resources. Not all client libraries support custom resources. The *Go* and *Python* client libraries do.

When you add a custom resource, you can access it using:

- `kubect1`
- The Kubernetes dynamic client.
- A REST client that you write.

- A client generated using [Kubernetes client generation tools](#) (generating one is an advanced undertaking, but some projects may provide a client along with the CRD or AA).

What's next

- Learn how to [Extend the Kubernetes API with the aggregation layer](#).
- Learn how to [Extend the Kubernetes API with CustomResourceDefinition](#).

Feedback

Was this page helpful?

☐ Yes ☐ No

Last modified January 08, 2022 at 6:09 PM PST: [Reorganize Working with Kubernetes Objects section \(634c17f61c\)](#)