Custom Resources

What are custom resources? Kubernetes API extensions. (Any mention of API is assumed to be Kubernetes API)

Terms to define:
- Resource: endpoint in API that stores a collection of API objects of a certain kind.
    - Example: built-in pods resource contains a collection of pod objects
- Custom Resource: extension of API that is not necessarily available in a default Kubernetes installation, so it represents a customization of a particular Kubernetes installation.

Custom resources can appear and disappear in a running cluster through dynamic registration. Once installed, users can create and access its objects using kubectl, like for built-in resources like Pods.

Custom Controllers

Custom resources only allow you to store and retrieve structured data, but custom resource + custom controller = true declarative API (allows you to declare or specify the desired state of your resource and tries to keep the current state of the Kubernetes objects in sync with the desired state.)
- controllers interprets the structured data as a record of the user's desired state, and maintains this state.

- Controllers can be deployed and updated on a running cluster.
- They can work with any kind of resource, but most effective when combined with custom resources.
- You can use custom controllers to encode domain knowledge for specific applications into an extension of the Kubernetes API.

Considerations

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 `kubectl`.	`kubectl` support is not required
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 (Model)

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.

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.

Two Ways to Add Custom Resources to Your Cluster

1. Without programming (CRDs), simple and easy.
    1. Allow users to create new types of resources without adding another API server
    2. No need to understand API aggregation
2. With programming (aggregated APIs), allows more control over API behaviors like how data is stored and conversion between API versions.
    1. Subordinate API servers that sit behind the primary API server, which acts as a proxy

Custom Resource Definitions

CustomResourceDefintion API resource allows you to define custom resources.
- a new CRD object creates a new custom resource with a name and schema that you specify
- Kub API serves and handles the storage of your custom resource
- The name of a CRD object must be a valid DNS subdomain name.

See here for an example of using a custom controller.

Choosing a Method for Adding Custom Resources

CRDs are a good fit for:
• A handful of fields
• Using a resource within your company, or as part of a small open-source project

Comparing CRDs and Aggregated APIs

CRDs	Aggregated API
Do not require programming. Users can choose any language for a CRD controller.	Requires programming in Go 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.

(See External Images for Advanced Features and Flexibility, Common Features)

Preparing to Install a Custom Resource

Be aware:
- installing an aggregated API server always involves running a new Deployment
- Custom resources require storage space like ConfigMaps do. Too many resources may overwhelm the 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.
- CRDs always use the same authentication, authorization, and audit logging as the built-in resources of your API server.
- Aggregated API servers may or may not use the same authentication, authorization, and auditing as the primary API server.

Accessing a Custom Resource

Not all client libraries support custom resources, Go and Python client libraries do.
Access:
- Kubectl
- The Kubernetes dynamic client
- A REST client that you write
- A client generated using Kubernetes client generation tools

Click Here for the full documentation