Understanding CRDs

Understand Kubernetes CRDs and what's behind the scenes.

We'll cover the following

More about CRDs

With CRDs, we can define our own API schema and plumb them to the `kube-apiserver` at any time. No broken changes in the `kube-apiserver` are made. We don't need to restart or recompile the `kube-apiserver`. They work pretty much like add-on plugins, in the sense that we can apply or remove them whenever we want. Super handy!

What does a CRD look like?

To better understand CRDs, let's see what one looks like.

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: crontabs.stable.example.com
spec:
  group: stable.example.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                image:
                  type: string
                replicas:
                  type: integer
  scope: Namespaced
  names:
    plural: crontabs
    singular: crontab
    kind: CronTab
    shortNames:
```

```
30      - ct
```

CRD crontabs

Now, let's try to break down the definitions above so that we can better understand every field in a CRD. These fields are quite different in other built-in Kubernetes objects, and some of them are quite important and worth discussing.

- `apiVersion` **(line 1)**: This field specifies the `apiVersion` that we'll use for the `CustomResourceDefinition`. The `CustomResourceDefinition` is a Kubernetes built-in API, which has its own API versions. Normally, we use the `apiextensions.k8s.io/v1` API. If you're using a lower version of Kubernetes, it could be `apiextensions.k8s.io/v1beta1` instead.
- `kind` **(line 2)**: This field indicates the resource `kind`. Of course, here, we want to create a `CustomResourceDefinition`.
- `metadata.name` **(line 4)**: This field is quite important, because it specifies the name of the resource. It must match the `spec` fields below and be in `<plural>.<group>` form.
- `spec.group` **(line 6)**: Every resource has a group. Just like `Deployment` in group `apps`, we need to specify a group name for our CRD.
- `spec.versions` **(lines 7–23)**: In this field, we can list all the versions supported by this CRD. Every version can be served. Just like `Deployment`, we have versions `v1` and `v1beta1`. For CRDs, we could have multiple versions as well. The version that serves will be used in the API URL.
  - `spec.versions.served` **(line 9)**: This field indicates whether this version should be enabled or disabled.
  - `spec.verisons.storage` (**line 10**): Only one version must be marked as the storage version.
  - `spec.versions.schema` **(lines 11–23)**: This field specifies a structural schema that we want to validate the `CRD` of using the openAPIV3Schema validation. We can mark some custom object fields as required, specify the value type, set the default values, or apply regex matching rules for each field. In our example, we specify that the field `spec.replicas` must be an integer. Additionally, we could set the default integer value for this field. If we're applying a custom object with non-integer `spec.replicas`, the `kube-apiserver` will reject the request due to the validation failure.
- `spec.scope` **(line 24)**: This field is quite important, which specifies if the custom object is namespaced or cluster-wide. In Kubernetes, resources are either one of them. For example, `Deployment` is namespace-scoped, while `PersistentVolume` is cluster-scoped. For namespace-scoped resources, we need to specify a namespace for them when we perform operations, such as creating, listing, or deleting. If we don't set the `spec.scope`, it will be defaulted to `Cluster`, which stands for cluster-scoped.
- `spec.names` **(line 25–30)**:

- `spec.names.plural` **(line 26)** and `spec.names.singular` **(line 27)** specify the plural and singular names of the CRD. The plural name will be used in the serving URL `/apis/<group>/<version>/<plural>`.
- `spec.names.kind`: This specifies the type of the custom object. It's normally the CamelCased singular type, such as `Deployment`, `StatefulSet`, or `CronTab`.
- `spec.names.shortNames`: With this, we can specify the short string or aliases when we use the CLI. For example, we can use `kubectl get deploy` to list deployments. Of course, we could use the plural form `kubectl get deployments`. With the alias, we can type faster and usage becomes easier, especially when the `kind` has got a long name and is hard to spell.

For the `CRD` in the example above, the serving URL will be `/apis/stable.example.com/v1/namespaces/<some-namespace>/crontabs`.

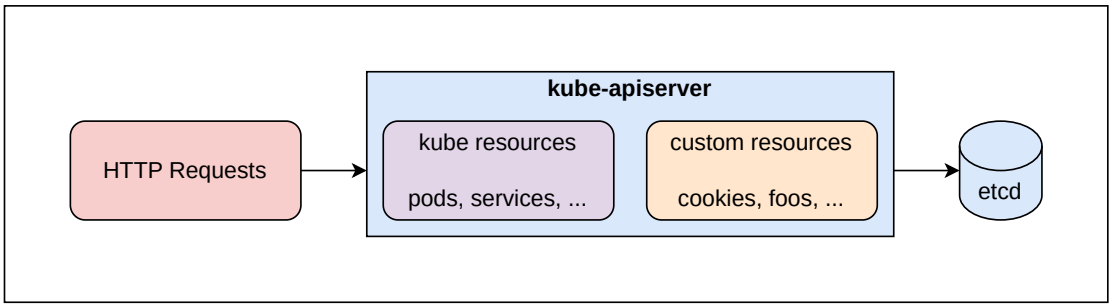
All `CRD` APIs work natively as other built-in APIs. They're both Kubernetes-style with a consistent user experience for the CLI.

Now, let's dive deeper to find out what happens behind the scenes to the `kube-apiserver` when we create this kind of a `CRD`.

Behind the scenes

A CRD defines what our object looks like and lets the `kube-apiserver` help manage the entire lifecycle, from creation to deletion. Through dynamic registration, these custom CRD APIs can appear and disappear in a Kubernetes cluster.

When we apply a CRD to the `kube-apiserver`, it will create a new ad hoc RESTful resource path for each version we specify. The path can be accessed either in a namespace or a cluster. Then, we can create and access the objects using `kubectl` or other RESTful requests, just as what we would do for Pods.



Storing custom resources to etcd

As with other existing built-in objects, such as Pods and ConfigMaps, deleting a namespace deletes all the resources in that namespace, including custom objects. Cluster-scoped custom resources are available to all namespaces.

When we delete such a CRD from the `kube-apiserver`, the ad hoc RESTful resource path of each version gets removed as well. We'll get the error `the server doesn't have a resource type xxxx` when we use `kubectl` to list the custom resources.

We can also grant access to CRD resources, like what we do for Pods.

So far, we've figured out what happens behind the scenes. However, the CRD itself is just a definition; it doesn't contain any logic. By looking at the CRDs, we may see some similarities with ConfigMap. Then, what's the difference between them?

CRD VS ConfigMaps

If we use CRDs without a custom controller to handle and process the objects, both of them can be used to store configurations for us.

However, they do have some noticeable differences between them.

First of all, ConfigMaps are used to provide configurations for various resources, such as Pods. They can be mounted as volume or injected as environment variables into the Pods. Rolling updates will be performed when ConfigMaps are updated. For CRDs, no Kubernetes components are sensitive.

CRDs have different purposes than ConfigMaps. They aren't meant to provide configurations for Pods, but instead extend the Kubernetes API to build our own custom logic. Normally, we would have a custom controller to handle updates to custom objects.

Back

Introduction to Custom Resources

Next

How to Use Kubernetes CRDs