How to Use Kubernetes CRDs

Learn how to use CRDs in Kubernetes.

We'll cover the following

Kubernetes CRDs

Custom resources (CRDs) are an efficient way of extending Kubernetes APIs, allowing us to make customized and declarative APIs. When we create a new CRD API, the `kube-apiserver` will create a new RESTful handler for each specified version. We'll demonstrate this shortly. The CRD can define either namespaced or cluster-scoped APIs, as specified in the field `spec.scope`.

Now, let's get started on how to use CRDs in Kubernetes.

Create a custom resource

The development environment in which we can add and modify our programs is given below. We can click the "Run" button to initialize it.

In the `crd.yml` file, we define the CRD for the kind `Foo` **(line 31)** with the API group `pwk.educative.io` **(line 6)**. This `Foo` kind will be namespaced, because we specify `spec.scope` **(line 33)** as `Namespaced`. It could be `Cluster` if we want it cluster-scoped. Here, we start off with the version `v1alpha1` **(line 8)**. In this `Foo` kind, we've declared two fields, `deploymentName` **(line 19)** and `replicas` **(line 21)** in `spec`. We also set an integer range from `1` to `10` **(lines 23–24)** for `spec.replicas`. With this kind of structural schema, the `kube-apiserver` will help us validate all the `Foo` kind resources and reject those whose `spec.replicas` aren't in the range. In `status` **(lines 25–29)**, we have a field `availableReplicas` **(line 28–29)**, which is an integer as well.

crd.yml

Search in directory...

/

```
1  apiVersion: apiextensions.k8s.io/v1
2  kind: CustomResourceDefinition
3  metadata:
4    name: foos.pwk.educative.io
5  spec:
6    group: pwk.educative.io
7    versions:
```

crd.vml
example-foo.yml

```yaml
  - name: v1alpha1
    served: true
    storage: true
    schema:
      # schema used for validation
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              deploymentName:
                type: string
              replicas:
                type: integer
                minimum: 1
                maximum: 10
          status:
            type: object
            properties:
              availableReplicas:
                type: integer
names:
  kind: Foo
```

CRD for Foo kind

Now, let's use the `kubectl apply` command to register our CRD in the terminal above:

```
kubectl apply -f ./crd.yml
```

Create a CRD

Once applied, we should see the following output:

```
customresourcedefinition.apiextensions.k8s.io/foos.pwk.educative.io created
```

We can verify this by running the `kubectl get crds | grep foo` command. We could also use `kubectl api-resources | grep foo` instead.

Everything seems to be working fine. Let's list this kind `foo`, just like we list Pods in the namespace `default`.

```
kubectl get foo -n default
```

Command to list resource foo in the namespace called default

The output will be as follows:

```
No resources found in default namespace.
```

The output of listing resource foo in namespace default

So far, we've not created any objects with the kind `foo` and the output proves this.

Let's take a look at the RESTful APIs that the `kube-apiserver` creates for the kind `foo`. This can be easily discovered if we increase the log level verbosity with the command below:

```
kubectl get foo -n default -v 7
```

Increase log level for listing resource foos

The output will be as follows:

```
I1012 09:10:55.254232    4271 loader.go:372] Config loaded from file:  /root/.kube/config
I1012 09:10:55.265509    4271 round_trippers.go:463] GET https://172.17.0.2:6443/apis/pwk.educative.io/v
I1012 09:10:55.265726    4271 round_trippers.go:469] Request Headers:
I1012 09:10:55.265915    4271 round_trippers.go:473]     Accept: application/json;as=Table;v=v1;g=meta.k
I1012 09:10:55.266007    4271 round_trippers.go:473]     User-Agent: kubectl/v1.23.8 (linux/amd64) kuber
I1012 09:10:55.282981    4271 round_trippers.go:574] Response Status: 200 OK in 16 milliseconds
No resources found in default namespace.
```

The output

From the output, we can see that the ad hoc RESTful API is `/apis/pwk.educative.io/v1alpha1/namespaces/<namespace>/foos`. This matches exactly with what we define for the CRD `Foo`. Looks perfect!

So far so good. But, you're probably wondering how this is useful. Now, let's see how to create an object with the kind `Foo`.

Create custom objects

The manifest below creates an object of our new kind `Foo`. We normally call this the custom resource. Here, we're using the kind `Foo` we defined in our CRD:

```
kubectl apply -f example-foo.yml
```

Create a custom resource

Now, when we issue the command in the terminal above to create this kind of a custom resource, we get the error below:

```
The Foo "example-foo" is invalid: spec.replicas: Invalid value: 11: spec.replicas in body should be less
```

The output of creating a resource with invalid settings

Yes, the validation works. Now, let's directly modify this `example-foo.yml` in the widget and change the `replicas` from `11` to `1`. After that, we can successfully run `kubectl apply -f example-foo.yml`.

How to delete a CRD and custom resources

To delete the CRD and custom resources we created, simply run `kubectl delete`, which is exactly how we delete other built-in Kubernetes objects.

As with existing built-in Kubernetes objects, when we delete a namespace, all custom objects within it will be deleted as well.

All custom resources of a kind will be pruned when we delete that CRD for that kind. For example, if we delete CRD `foos.pwk.educative.io`, all the `Foo` objects will be deleted, no matter what namespaces they're in. This is also true for cluster-scoped CRDs.

Back
Understanding CRDs

Next
How to Generate Scaffold CRDs