**Introduction to Custom Resources**

Learn why we need custom resources (CRDs) and what we can use them for.

**We'll cover the following**

* [The flexibility of Kubernetes](https://www.educative.io/courses/programming-with-kubernetes/introduction-to-custom-resources#The-flexibility-of-Kubernetes)
  + [Kubernetes APIs](https://www.educative.io/courses/programming-with-kubernetes/introduction-to-custom-resources#Kubernetes-APIs)
  + [Why CRDs?](https://www.educative.io/courses/programming-with-kubernetes/introduction-to-custom-resources#Why-CRDs)
* [Conclusion](https://www.educative.io/courses/programming-with-kubernetes/introduction-to-custom-resources#Conclusion)

**The flexibility of Kubernetes**

Kubernetes is well designed to orchestrate, deploy, and manage containers. In fact, this is the main job of Kubernetes, but it can do much more than that. What makes Kubernetes more popular and fundamental is its flexibility. We can build more things on top of Kubernetes and make it a customized platform to serve our own business needs.

**Kubernetes APIs**

Taking a deep look at the inner workings of Kubernetes, we can see that its core part is an API hosting server and the etcd data store. The API server receives and processes HTTP requests and then stores data back to the etcd cluster. All the other important components that we know, like the kube-scheduler, kube-controller-manager, and cloud-controller-manager, need to communicate with an API server to get the data and do the rest of work, like Pod scheduling, object population, state-driven, etc.

What makes these components work coherently is the **API schema**, which is the metadata of Kubernetes resources (like Pods, Namespaces, etc.), that defines the specifications and maps the relationships between them.

Every resource in Kubernetes has its own kind to indicate itself and is grouped by the group name. They can have multiple versions as well. Below is a YAML file of a Deployment.

Press+to interact

apiVersion: apps/v1

kind: Deployment

metadata:

  name:  nginx-demo

  namespace: default

spec:

  selector:

    matchLabels:

      app: nginx

  template:

    metadata:

      labels:

        app: nginx

    spec:

      containers:

      - name: nginx-demo

        image: nginx

        ports:

        - containerPort: 80

The YAML for a Deployment nginx-demo

The kind is declared as Deployment with version v1 in group apps. Above, the whole YAML file is actually an interpretation of the struct below:

Press+to interact

*// +genclient*

*// +genclient:method=GetScale,verb=get,subresource=scale,result=k8s.io/api/autoscaling/v1.Scale*

*// +genclient:method=UpdateScale,verb=update,subresource=scale,input=k8s.io/api/autoscaling/v1.Scale,result=k8s.io/api/autoscaling/v1.Scale*

*// +genclient:method=ApplyScale,verb=apply,subresource=scale,input=k8s.io/api/autoscaling/v1.Scale,result=k8s.io/api/autoscaling/v1.Scale*

*// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object*

*// Deployment enables declarative updates for Pods and ReplicaSets.*

type Deployment struct {

    metav1.TypeMeta `json:",inline"`

*// Standard object's metadata.*

*// More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata*

*// +optional*

    metav1.ObjectMeta `json:"metadata,omitempty" protobuf:"bytes,1,opt,name=metadata"`

*// Specification of the desired behavior of the Deployment.*

*// +optional*

    Spec DeploymentSpec `json:"spec,omitempty" protobuf:"bytes,2,opt,name=spec"`

*// Most recently observed status of the Deployment.*

*// +optional*

    Status DeploymentStatus `json:"status,omitempty" protobuf:"bytes,3,opt,name=status"`

}

Deployment struct definition

For every group, like apps, the resources defined in that group will be registered into the schema. The code below shows exactly how this is done

Press+to interact

var (

*// TODO: move SchemeBuilder with zz\_generated.deepcopy.go to k8s.io/api.*

*// localSchemeBuilder and AddToScheme will stay in k8s.io/kubernetes.   SchemeBuilder      = runtime.NewSchemeBuilder(addKnownTypes)*

   localSchemeBuilder = &SchemeBuilder

   AddToScheme        = localSchemeBuilder.AddToScheme

)

*// Adds the list of known types to the given scheme.*

func addKnownTypes(scheme \*runtime.Scheme) error {

   scheme.AddKnownTypes(SchemeGroupVersion,

      &Deployment{},

      &DeploymentList{},

      &StatefulSet{},

      &StatefulSetList{},

      &DaemonSet{},

      &DaemonSetList{},

      &ReplicaSet{},

      &ReplicaSetList{},

      &ControllerRevision{},

      &ControllerRevisionList{},

   )

   metav1.AddToGroupVersion(scheme, SchemeGroupVersion)

   return nil

}

API registration in group apps

Then, the kube-apiserver can know the exact semantics for these objects and expose them via the RESTful HTTP API for CREATE, GET, DELETE, etc. That’s why we get a list of deployments whenever we execute kubectl get deploy.

Press+to interact

1

kubectl get deploy -n kube-system

Listing deployments in kube-system namespace

The output will be as follows:

Press+to interact

1

2

NAME      READY   UP-TO-DATE   AVAILABLE   AGE

coredns   2/2     2            2           61d

The output

However, if we try to get a list of objects that don’t exist in Kubernetes, we would get a response similar to the one below.

Press+to interact

1

kubectl get cookie

Listing our customized resources with kubectl

The output will be as follows:

Press+to interact

1

error: the server doesn't have a resource type "cookie"

The output for unknown resources

**Why CRDs?**

So, what’s the point of adding such a cookie definition to our Kubernetes cluster?

We’ve seen the magic of Kubernetes. For example, with Deployment we could run our services with multiple replicas. Kubernetes helps us find the most suitable nodes to run the containers and keep all replicas at high availability. Kubernetes serves as our SRE (Site Reliability Engineering) and handles the grunt work. Kubernetes natively provides lots of APIs and objects to help us publish and manage our workloads and services.

Although no API can meet all of our requirements, there are instances where customization becomes necessary to fulfill our specific use cases. Let’s suppose we would like to define a workload to do specific operational tasks inside of the Kubernetes cluster, like creating a backup, restoring from a backup, cleaning up unwanted resources, etc. As a result, we do need to have our own Kubernetes-style, declarative APIs to declare these kinds of tasks, just like Pods do in Kubernetes.

Starting from v1.7, Kubernetes introduces powerful custom resource definitions, which are also known as CRDs. This helps us extend the Kubernetes APIs and add our own objects to the Kubernetes cluster. They can use all the features as other built-in Kubernetes resources, such as CLI, role-based access control, resource quota, etc. This means if we can add Cookie objects to Kubernetes, we can run kubectl get cookie to list all the cookies, exactly the same way listed Pods run in the cluster. That’s cool, isn’t it?

CRDs can be used in many different scenarios and help extend the capabilities of Kubernetes. We use them like any other native Kubernetes objects.

**Conclusion**

It’s worth noting that CRDs themselves don’t contain any logic. What a CRD does is provide a way to create, store, and expose Kubernetes-style APIs for custom objects, and that’s enough for us to make up our own logic by consuming these RESTful APIs.

**Understanding CRDs**

Understand Kubernetes CRDs and what's behind the scenes.

**We'll cover the following**

* [More about CRDs](https://www.educative.io/courses/programming-with-kubernetes/understanding-crds#More-about-CRDs)
  + [What does a CRD look like?](https://www.educative.io/courses/programming-with-kubernetes/understanding-crds#What-does-a-CRD-look-like)
* [Behind the scenes](https://www.educative.io/courses/programming-with-kubernetes/understanding-crds#Behind-the-scenes)
* [CRD VS ConfigMaps](https://www.educative.io/courses/programming-with-kubernetes/understanding-crds#CRD-VS-ConfigMaps)

**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.

Press+to interact

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:

    - 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](https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/#validation) 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.

Press+to interact

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.