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

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`.

```yaml
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:

```
// +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,res
// +genclient:method=ApplyScale,verb=apply,subresource=scale,input=k8s.io/api/autoscaling/v1.Scale,resu
// +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#me
    // +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

```
var (
   // TODO: move SchemeBuilder with zz_generated.deepcopy.go to k8s.io/api.
   // localSchemeBuilder and AddToScheme will stay in k8s.io/kubernetes.   SchemeBuilder      = runtime.
   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`.

```
kubectl get deploy -n kube-system
```

Listing deployments in kube-system namespace

The output will be as follows:

```
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.

```
kubectl get cookie
```

Listing our customized resources with kubectl

The output will be as follows:

```
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.

Back

Quiz on Schedulers

Next

Understanding CRDs