Custom Resource Definitions (CRDs) and Controllers

1. What is a CRD (Custom Resource Definition)?

- Think of CRD as **a way to teach Kubernetes a new language**.
- By default, Kubernetes understands “pods”, “services”, “deployments”, etc.
- CRD lets you create your own resource like:
  → BackupJob, CustomConfigMap, AlertRule, etc.

Example:
You define a new type of object called CustomConfigMap.
Once defined, you can create instances of it just like kubectl apply -f.

2. What is a CR (Custom Resource)?

- After defining a CRD, you can create actual **objects of that new type**.
- These objects are called **Custom Resources (CRs)**.
- It’s like:

> You create a new class (CRD), and now you're creating objects from it (CRs).

Example:

You created a CRD called CustomConfigMap.
Now you create a CR like:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomConfigMap
metadata:
  name: my-config
spec:
  data:
    key1: value1
    key2: value2
```

3. What is a Custom Controller?

- Controllers are like **watchdogs**.
- They keep an eye on certain resources (like your CR) and **take actions** based on changes.

A Custom Controller:

- Watches your custom resources (CustomConfigMap)
- Creates/updates/deletes a real Kubernetes object (like a ConfigMap) based on your CR

It's like:

"Hey, if a new CustomConfigMap is created, let me also create a real ConfigMap automatically."

4. Why Do We Use CRDs + Controllers?

Automate custom logic
Make Kubernetes behave how you want
Integrate external systems or business logic

Hands-On Steps

Step 1: Create a CRD

customconfigmap-crd.yaml

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: customconfigmaps.kiran.com
spec:
  group: kiran.com
```

```
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                data:
                  type: object
                  additionalProperties:
                    type: string
  scope: Namespaced
  names:
    plural: customconfigmaps
    singular: customconfigmap
    kind: CustomConfigMap
```

apply:

```
kubectl apply -f customconfigmap-crd.yaml
```

Step 2: Create a Custom Resource

```
# customconfigmap.yaml

  apiVersion: kiran.com/v1
  kind: CustomConfigMap
  metadata:
    name: my-custom-cm
  spec:
    data:
      key1: hello
      key2: world
```

kubectl apply -f customconfigmap.yaml

Step 3: Create a Controller (Python example)

Save this as controller.py:

```
from kubernetes import client, config, watch

def create_configmap(name, data, namespace="default"):
    core = client.CoreV1Api()
    body = client.V1ConfigMap(
        metadata=client.V1ObjectMeta(name=name),
        data=data
    )
    core.create_namespaced_config_map(namespace, body)

def delete_configmap(name, namespace="default"):
    core = client.CoreV1Api()
    core.delete_namespaced_config_map(name, namespace)

def main():
    config.load_incluster_config()
    api = client.CustomObjectsApi()
    w = watch.Watch()
    for event in w.stream(api.list_namespaced_custom_object,
                          group="kiran.com", version="v1",
                          namespace="default", plural="customconfigmaps"):
        cr = event['object']
        name = cr['metadata']['name']
        spec_data = cr['spec'].get('data', {})
        if event['type'] == 'ADDED':
            create_configmap(name, spec_data)
        elif event['type'] == 'DELETED':
            delete_configmap(name)

main()
```

Step 4: Dockerize the Controller

dockerfile

```
# Dockerfile
FROM python:3.9
RUN pip install kubernetes
COPY controller.py .
CMD ["python", "controller.py"]
```

Apply:

docker build -t kiran/custom-controller:v1 .

Push it to Docker Hub or your private registry.

Step 5: Deploy the Controller to Cluster

Create controller-deployment.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: custom-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      app: custom-controller
  template:
    metadata:
      labels:
        app: custom-controller
    spec:
      serviceAccountName: custom-controller-sa
      containers:
        - name: custom-controller
          image: kiran/custom-controller:v1
```

Step 6: Set Up RBAC

serviceaccount.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: custom-controller-sa
```

clusterrole.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: custom-controller-role
rules:
  - apiGroups: ["kiran.com"]
    resources: ["customconfigmaps"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["configmaps"]
    verbs: ["create", "delete"]
```

clusterrolebinding.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: custom-controller-binding
subjects:
  - kind: ServiceAccount
    name: custom-controller-sa
    namespace: default
roleRef:
  kind: ClusterRole
  name: custom-controller-role
  apiGroup: rbac.authorization.k8s.io
```

kubectl apply -f serviceaccount.yaml
kubectl apply -f clusterrole.yaml
kubectl apply -f clusterrolebinding.yaml
kubectl apply -f controller-deployment.yaml

Step 7: Test It

1. Apply your custom resource:

   kubectl apply -f customconfigmap.yaml

2. Verify that the ConfigMap is created:

   kubectl get configmap

3. Delete the custom resource:

   kubectl delete -f customconfigmap.yaml

4. Verify that the ConfigMap is also deleted:

   kubectl get configmap

🏁 **Conclusion**

You just extended Kubernetes!
You made it handle a brand new resource: CustomConfigMap.
You wrote automation logic via a controller.