What is Custom Controller?

Controller will run in a loop that continuously monitor some of the resources (for which it is intended to) and perform operations.

In the k8s we have kube-control manager which include multiple in-built controllers that monitor and perform operations on different and specific types of resources.

Try always try to match the current state to desired state.

Some of the examples of types of controllers present in k8s internal kube-control manager are:

1. Replication controller
2. Endpoints controller
3. Namespace controller
4. Service Account controller

Controller **reconciles** the **current** State to **expected** state

***Custom controller*** are the extension to the concept of controller in k8s where we define our own logic of control loop or pattern to watch the state of that particular resource in the cluster, then make or request changes wherever needed.

**Possible Usecases:**

1. If namespace is not existing in the cluster, create a namespace.
2. Creating a ConfigMap in a namespace as soon as a Namespace is created.
3. Creating a ConfigMap in all namespace if created in one namespace.

**Writing a Controller for above document:**

1. **Prerequisites**

* [go](https://golang.org/dl/) version v1.20.0+
* [docker](https://docs.docker.com/install/) version 17.03+.
* [kubectl](https://kubernetes.io/docs/tasks/tools/install-kubectl/) version v1.11.3+.
* Access to a Kubernetes v1.11.3+ cluster.

1. Install kubebuilder:

mkdir customcontroller

cd customcontroller

curl -L -o kubebuilder "https://go.kubebuilder.io/dl/latest/$(go env GOOS)/$(go env GOARCH)"

chmod +x kubebuilder && sudo mv kubebuilder /usr/local/bin/

1. **Create a Project**

kubebuilder init --domain dss.io --repo github.com/sandeep/greeting-operator

1. **Create a New API for ConfigMaps**:

Even though ConfigMap is a built-in Kubernetes resource, Kubebuilder generates the boilerplate code to watch and reconcile it.

kubebuilder create api --group core --version v1 **--kind ConfigMap** --resource=false --controller=true

1. **Implement the Controller Logic**

Now, open the generated controller file **internal/controllers/configmap\_controller.go** and implement the logic to replicate the ConfigMap across all namespaces.

import (

"context"

"fmt"

**corev1** "k8s.io/api/core/v1"

"k8s.io/apimachinery/pkg/api/errors"

**metav1** "k8s.io/apimachinery/pkg/apis/meta/v1"

"k8s.io/apimachinery/pkg/runtime"

**ctrl** "sigs.k8s.io/controller-runtime"

"sigs.k8s.io/controller-runtime/pkg/client"

"sigs.k8s.io/controller-runtime/pkg/log"

)

func (r \*ConfigMapReconciler) **Reconcile**(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {

\_ = log.FromContext(ctx)

logger := log.FromContext(ctx)

// TODO(user): your logic here

// Fetch the ConfigMap instance

var configMap corev1.ConfigMap

if err := r.Get(ctx, req.NamespacedName, &configMap); err != nil {

if errors.IsNotFound(err) {

// Object not found, return

return ctrl.Result{}, nil

}

logger.Error(err, "Failed to get ConfigMap")

return ctrl.Result{}, err

}

**// Replicate ConfigMap to all namespaces**

err := r.replicateConfigMapToAllNamespaces(ctx, &configMap)

if err != nil {

logger.Error(err, "Failed to replicate ConfigMap")

return ctrl.Result{}, err

}

return ctrl.Result{}, nil

}

// replicateConfigMapToAllNamespaces replicates the given ConfigMap to all namespaces

func (r \*ConfigMapReconciler) **replicateConfigMapToAllNamespaces**(ctx context.Context, configMap \*corev1.ConfigMap) error {

// List all namespaces

var namespaceList corev1.NamespaceList

if err := r.List(ctx, &namespaceList); err != nil {

return fmt.Errorf("error listing namespaces: %w", err)

}

for \_, ns := range namespaceList.Items {

if ns.Name == configMap.Namespace {

// Skip the original namespace

continue

}

// Check if the ConfigMap already exists in the namespace

cm := &corev1.ConfigMap{}

err := r.Get(ctx, client.ObjectKey{Namespace: ns.Name, Name: configMap.Name}, cm)

if err != nil && errors.IsNotFound(err) {

// Create a new ConfigMap

newConfigMap := &corev1.ConfigMap{

ObjectMeta: metav1.ObjectMeta{

Name: configMap.Name,

Namespace: ns.Name,

},

Data: configMap.Data,

}

if err := r.Create(ctx, newConfigMap); err != nil {

return fmt.Errorf("error creating ConfigMap in namespace %s: %w", ns.Name, err)

}

fmt.Printf("Confimap created in Namespace %s\n", ns.Name)

} else if err == nil {

// Update existing ConfigMap

cm.Data = configMap.Data

if err := r.Update(ctx, cm); err != nil {

return fmt.Errorf("error updating ConfigMap in namespace %s: %w", ns.Name, err)

}

fmt.Printf("Confimap updated in Namespace %s\n", ns.Name)

} else {

return fmt.Errorf("error fetching ConfigMap in namespace %s: %w", ns.Name, err)

}

}

return nil

}

**Explanation of Changes:**

**Controller Setup**:

* 1. The controller is generated by Kubebuilder and then modified to include the logic for replicating ConfigMaps.
  2. SetupWithManager is used to register the controller with the manager, and it watches ConfigMap resources.

**Reconcile Function**:

* 1. This function is triggered when a ConfigMap event occurs (create, update, delete).
  2. The replicateConfigMapToAllNamespaces function replicates the ConfigMap to all namespaces, similar to the earlier example but using the client.Client provided by Kubebuilder.

**Client and Context**:

* 1. The client.Client provided by the controller-runtime package is used to interact with the Kubernetes API.
  2. The context.Context object is passed around to maintain context and manage request lifetimes.

1. **Build and Deploy the controller in your Kubernetes Cluster**

make manager

make deploy

1. **To Debug the Controller**

make run

OR

Alternatively, you can build a Docker image and deploy it in-cluster using:

**Edit make file in project – put the prefix as per your login name in docker hub.**

.PHONY: docker-build

docker-build: ## Build docker image with the manager.

$(CONTAINER\_TOOL) build -t **sandeepsoni**/${IMG} .

.PHONY: docker-push

docker-push: ## Push docker image with the manager.

$(CONTAINER\_TOOL) push **sandeepsoni**/${IMG}

**Build and Deploy**

docker login --username=sandeepsoni –password=\*\*\*\*\*\*\*\*

make docker-build docker-push IMG=<your image name>

make deploy IMG=<your image name>

To clean up the resources (This will remove the CRD, controller, and any associated resources.)

make **undeploy**

1. **Create a ConfigMap: cm.yaml**

apiVersion: v1

kind: ConfigMap

metadata:

name: mysettings-config

data:

name: SANDEEP

location: INDIA

1. Apply the ConfigMap

kubectl -f cm.yaml

1. Verify in all Namespaces
2. Note that the resource is created in all Namespaces

kubectl get cm -A | grep mysettings-config

Custom Resource Definition

A Kubernetes resource is a collection of similar objects accessible via the Kubernetes API. Kubernetes [comes with several resources by default](https://www.techtarget.com/searchitoperations/tutorial/A-step-by-step-tutorial-for-Kubernetes-implementation), such as pods, deployments and ReplicaSets.

To get the list of Resources: **kubectl api-resources**

**What are custom resources?**

* In a nutshell, *custom resources* are **extensions** of the Kubernetes API.
* Custom resources are registered dynamically to a cluster. Once the custom resource is registered, end users can **create, update and delete** its object using **kubectl**, similar to how users interact with built-in resources, like pods, deployments and services.
* **Custom Resource Definitions (CRD)** enable IT admins to introduce **new types** into the Kubernetes cluster to meet their custom requirements.
* CRD is template for creating Custom Resource (CR). It provides the schema for **validating** CR.
* It uses all the features of the Kubernetes ecosystem -- for example, its command-line interface (CLI), security, API services and role-based access control. The custom resource is also stored in the etcd cluster with proper replication and lifecycle management.
* Examples:
  + ISTIO – Service Mesh
  + Argo CD – GitOps
  + KeyCloak – Identity and Access Management
  + Approved List of Projects: <https://www.cncf.io/>
* Note that CRDs **do not have any logic** attached, nor any special behavior; once they are created, modified or removed, they take no actions on their own.
* To bring more advanced functionality for these custom resources, **implement controllers or operators**. These enable IT admins to extend the behavior of Kubernetes without modifications to the underlying code. This functionality interacts well with CRDs, and using these two together, IT teams can implement some relatively advanced features and functionality.

**Example:**

**crd.yaml**

apiVersion: apiextensions.k8s.io/v1

kind: **CustomResourceDefinition**

metadata:

name: **greetings.sandeep.dss.io**

spec:

group: **sandeep.dss.io**

scope: Namespaced

names:

kind: **Greeting**

listKind: **GreetingList**

plural: **greetings**

singular: greeting

shortNames:

- grt

- grts

versions:

- name: **v1**

served: true

storage: true

schema: # schema used for validation

openAPIV3Schema:

type: object

**properties**:

**spec**:

type: object

**properties**:

**emailToGreet**:

type: string

**messageToGreet**:

type: string

**replicas**:

type: integer

minimum: 1

maximum: 10

**status**:

type: object

properties:

**emailToGreet**:

type: string

**availableReplicas**:

type: integer

kubectl apply -f ./crd.yaml

kubectl get grt

kubectl api-resources | grep grt

kubectl proxy --port=8080

curl localhost:8080/apis | grep sandeep.io

curl localhost:8080/apis

curl localhost:8080/apis/sandeep.io/v1/namespaces/default/greetings

**Creating the custom resource**

With the CRD registered in our cluster, we can create records for the custom resource. The manifest below creates a new instance of our new CRD.

**cr.yaml**

apiVersion: sandeep.io/v1

kind: Greeting

metadata:

name: sandeep-greeting

spec:

**emailToGreet**: sandeep.soni@gmail.com

**messageToGreet**: Hello

replicas: 1

kubectl apply -f ./cr.yaml

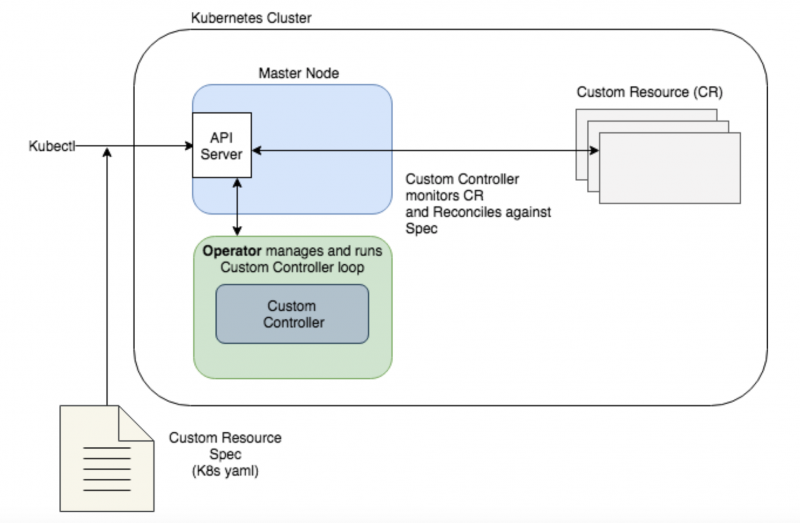
kubectl get grt  #

kubectl proxy --port=8080

curl localhost:8080/apis/sandeep.io/v1/namespaces/default/greetings/sandeep-greeting

**Operator In Kubernetes**

* An operator is a Kubernetes extension that automates the deployment of an application.
* **Operators actually allow for automatic implementation of typical Day-1 tasks** (installation, configuration, etc.) **and Day-2 tasks** (reconfiguration, upgrade, backup, failover, recovery, etc.), for a software running within the Kubernetes cluster, integrating natively with Kubernetes concepts and APIs.
* Operators are controllers for **packaging, managing, and deployin**g applications on Kubernetes. In order to do these things, the Operator uses **Custom Resources (CR)** that define the desired configuration and state of a specific application through **Custom Resource Definitions (CRD)**.
* **Operators are nothing but controllers that can act upon custom resources instead of k8s default resources.**



An almost infinite number of things can be automated through an Operator, however **some typical operations are more common** than others.

* The ability to **deploy**an application on demand.
* Making a **backup**of an application state or restarting an application from a given backup.
* Managing**the update of an application**with all its dependencies including new configuration settings and necessary database changes.
* Exposing a service to applications that do not support **Kubernetes APIs**.

**MySQL Example**:

Assume we need a stateful application such as a MySQL instance. Now, a mysql instance cannot be rolled up/down, upgraded just like that in case of stateless pods. Also being stateful taking regular **backups**, and ability to **restore** them on time also plays a key role in their lifecycle management. So, a mysql operator takes care of all these operations and all we need to do is declaratively tell what needs to be done via yaml files. So, even without much knowledge about mysql administration, one can get started with complex setups like a multi-master database setup with regular **backups, scale up/down , version upgrade easily**.

**MySQL Server Operator:** <https://github.com/mysql/mysql-operator/tree/trunk>

**operatorhub.io has all approved Kubernetes Operators.**

**Examples of Operators**: Prometheus, Elastic Search, ISTIO, ArgoCD, Minio

All these are written using **Operator SDK.**

**Operator vs Helm Charts:**

* HELM doesn’t support reconciliation (Auto Healing).
* Operator has Bundle, Bundle has resources. Operator overrides any wrong changes as it has reconciliation logic.
* It creates and auto heals the resources.
* If Operator sync is Automatic, whenever a new version is released, it automatically install the upgrades.

**Writing an Operator**

1. Create a Project

mkdir ~/operator

cd ~/operator

kubebuilder init --domain dss.io --repo github.com/sandeep/greeting-operator

1. Create an API

kubebuilder create api --group sandeep --version v1 --kind Greeting

1. Edit api/vi/greeting\_types.go

type **GreetingSpec** struct {

EmailToGreet string `json:"emailToGreet,omitempty"`

MessageToGreet string `json:"messageToGreet,omitempty"`

Replicas int32 `json:"replicas,omitempty"`

}

// GreetingStatus defines the observed state of Greeting

type **GreetingStatus** struct {

EmailToGreet string `json:"emailToGreet,omitempty"`

AvailableReplicas int32 `json:"availableReplicas,omitempty"`

}

1. **Edit internal/controller/greeting\_controller.go**

**import** (

"context"

"fmt"

"k8s.io/apimachinery/pkg/api/errors"

"k8s.io/apimachinery/pkg/runtime"

ctrl "sigs.k8s.io/controller-runtime"

"sigs.k8s.io/controller-runtime/pkg/client"

"sigs.k8s.io/controller-runtime/pkg/log"

sandeepv1 "github.com/sandeep/greeting-operator/api/v1"

)

const **greetingFinalizer** = "greeting.finalizers.sandeep.dss.io"

func (r \*GreetingReconciler) **Reconcile**(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {

\_ = log.FromContext(ctx)

var greeting sandeepv1.Greeting

if err := r.Get(ctx, req.NamespacedName, &greeting); err != nil {

if errors.IsNotFound(err) {

// Resource not found. Ignore the request as it's been deleted

return ctrl.Result{}, nil

}

// Error reading the object - requeue the request

return ctrl.Result{}, err

}

**//This function manages the complete lifecycle of the finalizer, ensuring that cleanup is performed before the resource is deleted.**

**// Check if the object is being deleted**

if greeting.ObjectMeta.DeletionTimestamp.IsZero() {

**// Resource is Added or Modified**

if greeting.Spec.EmailToGreet != greeting.Status.EmailToGreet {

fmt.Printf("emailToGreet has changed from %s to %s\n", greeting.Status.EmailToGreet, greeting.Spec.EmailToGreet)

}

**// Update the status subresource with the new observed state**

greeting.Status.EmailToGreet = greeting.Spec.EmailToGreet

if err := r.Status().Update(ctx, &greeting); err != nil {

return ctrl.Result{}, err

}

// The object is not being deleted, so ensure it has a finalizer

if len(greeting.GetFinalizers()) == 0 {

greeting.SetFinalizers(append(greeting.GetFinalizers(), greetingFinalizer))

if err := r.Update(ctx, &greeting); err != nil {

return ctrl.Result{}, err

}

}

} else {

arr := greeting.GetFinalizers()

if len(arr) != 0 {

v := arr[0]

if err := r.cleanupGreetingResources(&greeting); err != nil {

return ctrl.Result{}, err

}

if containsString(greeting.GetFinalizers(), v) {

greeting.SetFinalizers(removeString(greeting.GetFinalizers(), v))

if err := r.Update(ctx, &greeting); err != nil {

return ctrl.Result{}, err

}

}

return ctrl.Result{}, nil

}

}

return ctrl.Result{}, nil

}

func (r \*GreetingReconciler) **cleanupGreetingResources**(greeting \*sandeepv1.Greeting) error {

// Add additional cleanup logic as needed...

fmt.Printf("Cleaned up resources for Greeting: %s\n", greeting.Name)

return nil

}

func **containsString**(slice []string, s string) bool {

for \_, item := range slice {

if item == s {

return true

}

}

return false

}

func **removeString**(slice []string, s string) []string {

result := []string{}

for \_, item := range slice {

if item != s {

result = append(result, item)

}

}

return result

}

1. **make install**
2. **make run**
3. **Write the Custom Resource Greeting**

apiVersion: sandeep.dss.io/v1

kind: Greeting

metadata:

name: sandeep-greeting

finalizers:

- myfinal

spec:

emailToGreet: sandeepsoni@deccansoft.com

messageToGreet: Hello

replicas: 1

**To install an Operator in your Kubernetes cluster using an image you already have, follow these steps:**

1. Create RBAC (Role-Based Access Control) Resources: Operators usually need specific permissions to manage resources in the cluster. Create necessary ServiceAccount, Role, and RoleBinding (or ClusterRole and ClusterRoleBinding if needed).

apiVersion: v1

kind: Namespace

metadata:

name: my-operator-ns

---

apiVersion: v1

kind: **ServiceAccount**

metadata:

name: my-operator-sa

namespace: my-operator

---

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: my-operator-clusterrole

rules:

- apiGroups: [""]

resources: ["pods"]

verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]

- apiGroups: ["apps"] # Allow access to resources in the apps API group

resources: ["deployments"]

verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]

- apiGroups: [""]

resources: ["services"]

verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]

- apiGroups: ["apps"]

resources: ["statefulsets"]

verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]

- apiGroups: ["apiextensions.k8s.io"]

resources: ["customresourcedefinitions"]

verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]

Note: The specific permissions required by an Operator depend on what the Operator is designed to do. If your Operator needs to manage Deployments, Pods, Services, or other resources, you'll need to ensure that the Role (or ClusterRole) includes the appropriate rules.

**Cluster Role Binding**

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: my-operator-clusterrolebinding

subjects:

- kind: ServiceAccount

name: my-operator-sa

namespace: my-operator-ns # Link the ServiceAccount in the specific namespace

roleRef:

kind: ClusterRole

name: my-operator-clusterrole

apiGroup: rbac.authorization.k8s.io

1. **Prepare the Deployment YAML: Create a YAML file to define the Kubernetes deployment. This will specify the Operator’s container image, resources, and necessary RBAC (Role-Based Access Control) permissions.**

apiVersion: apps/v1

kind: **Deployment**

metadata:

name: my-operator-name

namespace: my-operator-ns

spec:

replicas: 1

selector:

matchLabels:

name: my-operator-name

template:

metadata:

labels:

name: my-operator-name

spec:

**serviceAccountName: my-operator-sa**

containers:

- name: my-operator-container

image: **my-operator-image:tag**

imagePullPolicy: IfNotPresent

1. **Apply all the YAML files.**

**Tutorial:** [**https://book.kubebuilder.io/cronjob-tutorial/cronjob-tutorial**](https://book.kubebuilder.io/cronjob-tutorial/cronjob-tutorial)

**Finalizer**: <https://kubernetes.io/blog/2021/05/14/using-finalizers-to-control-deletion/>

**Comparison Between Controller and Operator**

<https://konghq.com/blog/learning-center/kubernetes-controllers-vs-operators>