# Extending K8s using the Operator Pattern

A Gentle Introduction

# Agenda

- Operator and its use cases
- Operator Pattern
- Controller - Reconciliation loop with an example
- Components that are required to implement an operator (Informer, Lister, WorkQueue etc.)
- Code Walkthrough - kubernetes/sample-controller
- Overview on Controller Runtime and Kubebuilder
- Case Study - K8s Cleaner Implementation and Demo
- Discussion - Q&A

# Speaker Bio

Hello everyone! 👋

My name is Roopesh Saravanan. I work as a network consulting engineer at Cisco. My interest lies in cloud-native (specifically container networking and Kubernetes operators) and eBPF. I am a member and contributor to the Internet Health Report (IHR), an open-source observability solution for the Internet.

Portfolio: https://roopeshsn.com
GitHub: https://github.com/roopeshsn
Linkedin: https://linkedin.com/in/roopeshsn

## What's an Operator?

- Yet another pod running in your cluster
- Designed for automation
- Automate repeatable tasks
- A business logic that Kubernetes need to handle

## Use cases:

- Configuration
- Auto scaling
- Backup
- Upgrade
- Monitoring and more

# Operator Pattern

*“Kubernetes' operator pattern concept lets you extend the cluster's behaviour without modifying the code of Kubernetes itself by linking controllers to one or more custom resources. Operators are clients of the Kubernetes API that act as controllers for a Custom Resource.”*

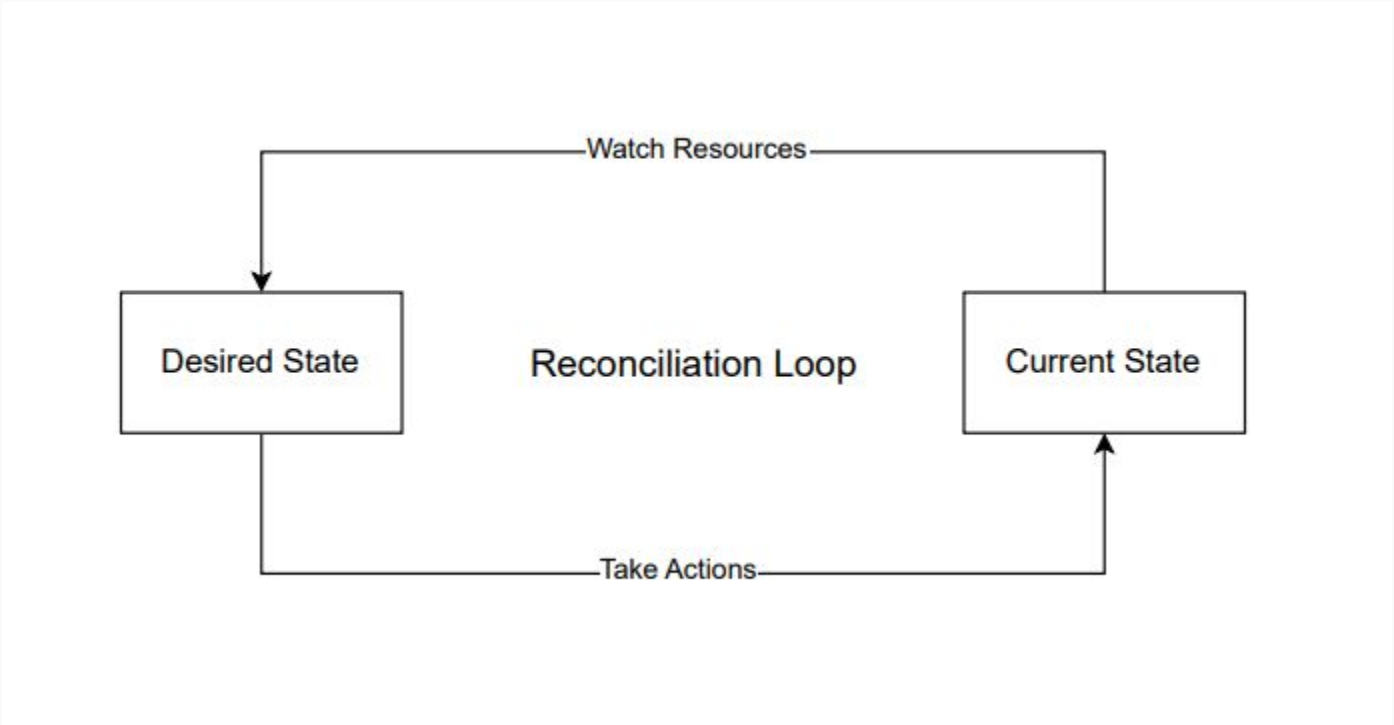
*“Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components. Operators follow Kubernetes principles, notably the control loop.”*

- Using Custom Resource Definitions (CRD) and Custom Resources (CR) to extend Kubernetes based on your business needs.
- As part of an operator you’ll implement a controller that watches the custom resource and make decisions. (For eg. Deployment Controller)

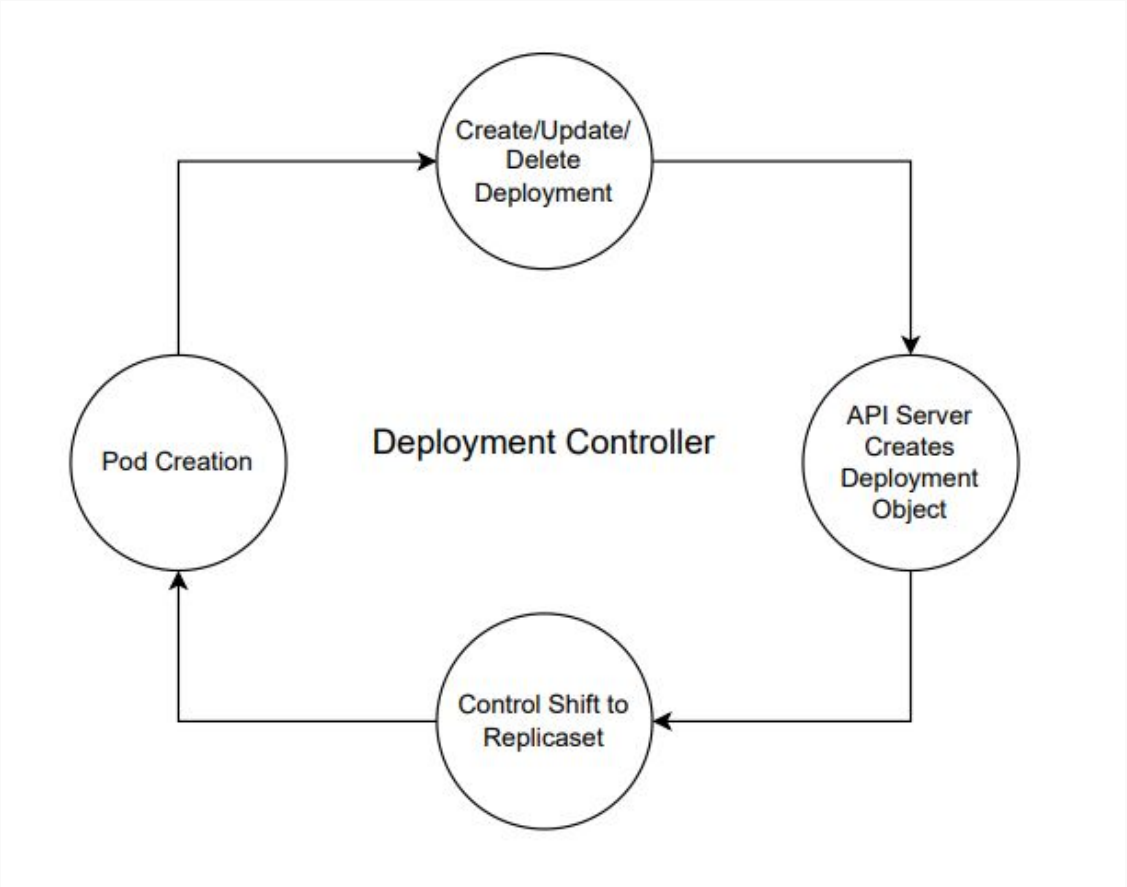
# Kubernetes-native Controllers

- Deployment Controller
- ReplicaSet Controller
- Namespace Controller
- Statefulset Controller and more.

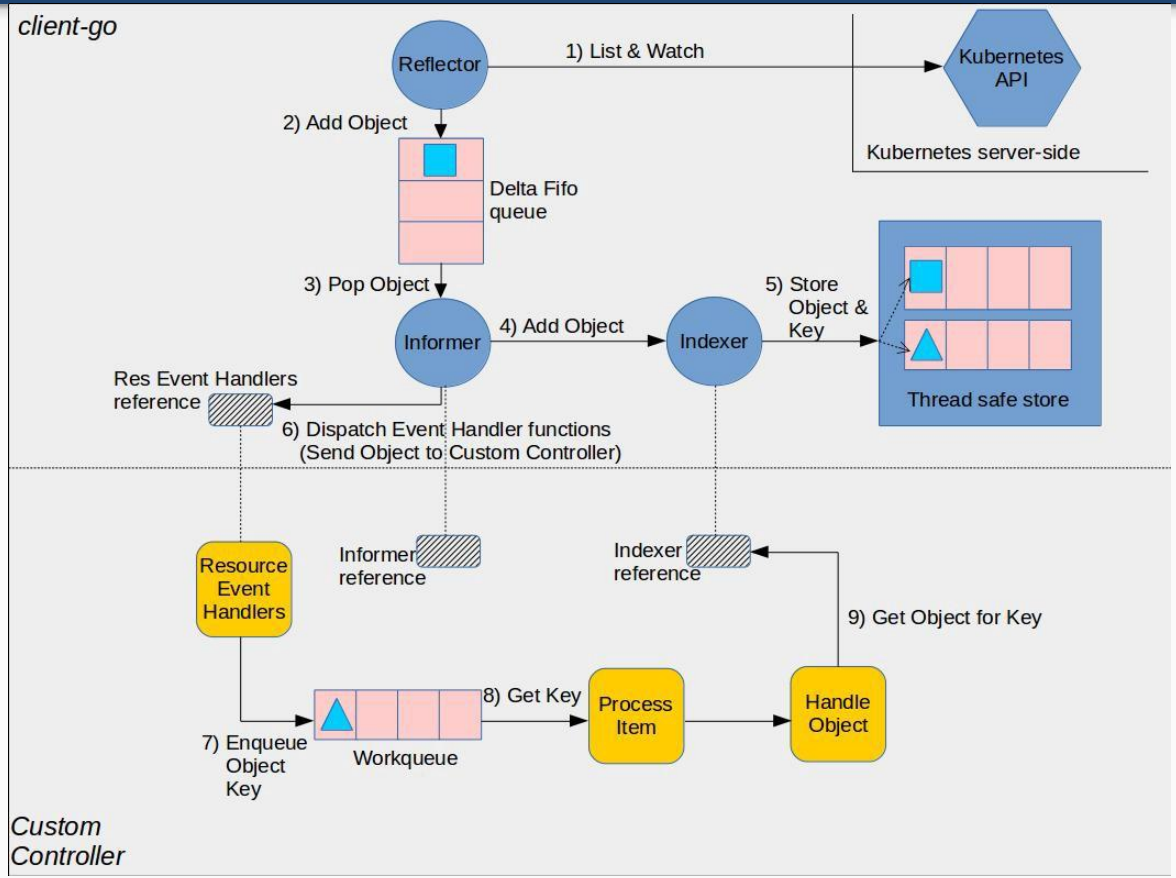
# Controller - Reconciliation Loop



# Deployment Controller



# Informer, Lister, Cache, Work Queue




Ref: [sample-controller documentation]

# Informer, Lister, Cache, Work Queue

1. Reflector - Watches the K8s API and add it in to a queue
2. Informer - Listen for events (create/update/delete) and invoke handler functions and change the cache
3. Shared Informer - Shared here means that it can be used by multiple controllers which uses the same informer instance and cache to communicate with the API server
4. Indexer - Provides indexing functionality for faster retrievals
5. Lister - Retrieves current state of a K8s resource from cache
6. Worker Queue - Resources are processed in the order they were received. Also they can be processed in parallel.

# Overview on Controller Runtime and Kubebuilder

Ways to build an operator/controller

- Using client-go to implement informers, and listers without any code generation or framework
- Using the code-generator shell script to generate informers and listers
- Using Controller Runtime
- Using frameworks like Operator SDK and Kubebuilder

Note: Operator SDK and Kubebuilder uses Controller Runtime under the hood.

# kubernetes/sample-controller

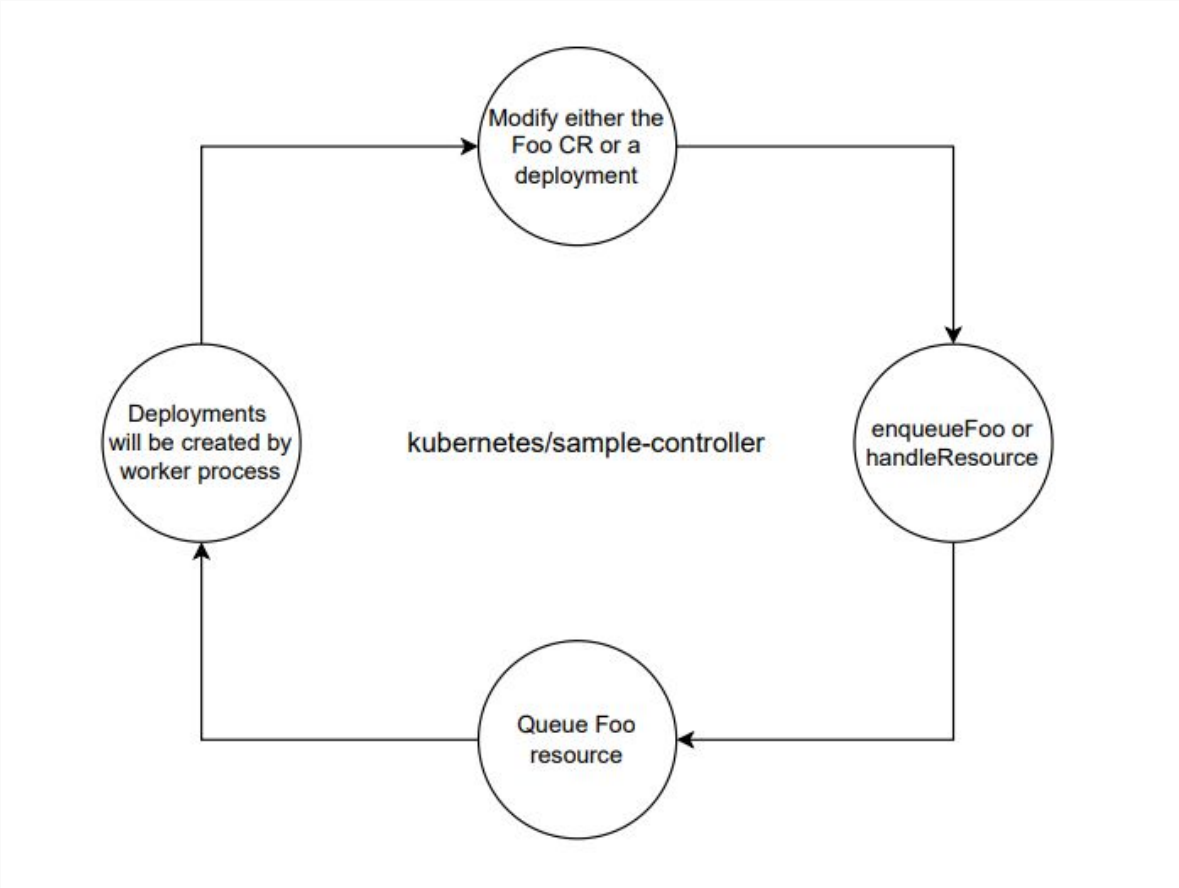
sample-controller will watch for Foo resource and create deployments accordingly.

1. When a Foo resource is modified (created/updated/deleted) it'll invoke the enqueueFoo function.
2. The enqueueFoo() function will add a Foo resource key in the workqueue
3. processNextWorkItem() function will call syncHandler function to create deployment that is mentioned in the Foo custom resource
4. If a deployment is modified by another actor then handleObject() will be called which in turns queue the Foo resource that is linked with that deployment.

```
1    apiVersion: samplecontroller.k8s.io/v1alpha1
2    kind: Foo
3    metadata:
4      name: example-foo
5    spec:
6      deploymentName: example-foo
7      replicas: 1
```

Ref: https://github.com/kubernetes/sample-controller

# kubernetes/sample-controller Reconciliation Loop



# kubernetes/sample-controller

```
66    // Controller is the controller implementation for Foo resources
67    type Controller struct {
68            // kubeclientset is a standard kubernetes clientset
69            kubeclientset kubernetes.Interface
70            // sampleclientset is a clientset for our own API group
71            sampleclientset clientset.Interface
72
73            deploymentsLister appslisters.DeploymentLister
74            deploymentsSynced cache.InformerSynced
75            foosLister        listers.FooLister
76            foosSynced        cache.InformerSynced
77
78            // workqueue is a rate limited work queue. This is used to queue work to be
79            // processed instead of performing it as soon as a change happens. This
80            // means we can ensure we only process a fixed amount of resources at a
81            // time, and makes it easy to ensure we are never processing the same item
82            // simultaneously in two different workers.
83            workqueue workqueue.TypedRateLimitingInterface[string]
84            // recorder is an event recorder for recording Event resources to the
85            // Kubernetes API.
86            recorder record.EventRecorder
87    }
88
```

# kubernetes/sample-controller

```
40  func main() {
54          kubeClient, err := kubernetes.NewForConfig(cfg)
55          if err != nil {
56                  logger.Error(err, "Error building kubernetes clientset")
57                  klog.FlushAndExit(klog.ExitFlushTimeout, 1)
58          }
59
60          exampleClient, err := clientset.NewForConfig(cfg)
61          if err != nil {
62                  logger.Error(err, "Error building kubernetes clientset")
63                  klog.FlushAndExit(klog.ExitFlushTimeout, 1)
64          }
65
66          kubeInformerFactory := kubeinformers.NewSharedInformerFactory(kubeClient, time.Second*30)
67          exampleInformerFactory := informers.NewSharedInformerFactory(exampleClient, time.Second*30)
68
69          controller := NewController(ctx, kubeClient, exampleClient,
70                  kubeInformerFactory.Apps().V1().Deployments(),
71                  exampleInformerFactory.Samplecontroller().V1alpha1().Foos())
72
73          // notice that there is no need to run Start methods in a separate goroutine. (i.e. go kubeInformerFactory.Start(ctx.done())
74          // Start method is non-blocking and runs all registered informers in a dedicated goroutine.
75          kubeInformerFactory.Start(ctx.Done())
76          exampleInformerFactory.Start(ctx.Done())
77
78          if err = controller.Run(ctx, 2); err != nil {
79                  logger.Error(err, "Error running controller")
80                  klog.FlushAndExit(klog.ExitFlushTimeout, 1)
81          }
82  }
```

# kubernetes/sample-controller

```
90  func NewController(
125         // Set up an event handler for when Foo resources change
126         fooInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
127                 AddFunc: controller.enqueueFoo,
128                 UpdateFunc: func(old, new interface{}) {
129                         controller.enqueueFoo(new)
130                 },
131         })
132         // Set up an event handler for when Deployment resources change. This
133         // handler will lookup the owner of the given Deployment, and if it is
134         // owned by a Foo resource then the handler will enqueue that Foo resource for
135         // processing. This way, we don't need to implement custom logic for
136         // handling Deployment resources. More info on this pattern:
137         // https://github.com/kubernetes/community/blob/8cafef897a22026d42f5e5bb3f104febe7e29830/contributors/devel/controllers.md
138         deploymentInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
139                 AddFunc: controller.handleObject,
140                 UpdateFunc: func(old, new interface{}) {
141                         newDepl := new.(*appsv1.Deployment)
142                         oldDepl := old.(*appsv1.Deployment)
143                         if newDepl.ResourceVersion == oldDepl.ResourceVersion {
144                                 // Periodic resync will send update events for all known Deployments.
145                                 // Two different versions of the same Deployment will always have different RVs.
146                                 return
147                         }
148                         controller.handleObject(new)
149                 },
150                 DeleteFunc: controller.handleObject,
151         })
152
153         return controller
154 }
```

# kubernetes/sample-controller

```
337  func (c *Controller) enqueueFoo(obj interface{}) {
338          var key string
339          var err error
340          if key, err = cache.MetaNamespaceKeyFunc(obj); err != nil {
341                  utilruntime.HandleError(err)
342                  return
343          }
344          c.workqueue.Add(key)
345  }
```

```
160  func (c *Controller) Run(ctx context.Context, workers int) error {
```

```
176          // Launch two workers to process Foo resources
177          for i := 0; i < workers; i++ {
178                  go wait.UntilWithContext(ctx, c.runWorker, time.Second)
179          }
180
181          logger.Info("Started workers")
182          <-ctx.Done()
183          logger.Info("Shutting down workers")
184
185          return nil
186  }
187
188  // runWorker is a long-running function that will continually call the
189  // processNextWorkItem function in order to read and process a message on the
190  // workqueue.
191  func (c *Controller) runWorker(ctx context.Context) {
192          for c.processNextWorkItem(ctx) {
193          }
194  }
```

# kubernetes/sample-controller

```
352  func (c *Controller) handleObject(obj interface{}) {
353          var object metav1.Object
354          var ok bool
355          logger := klog.FromContext(context.Background())
356          if object, ok = obj.(metav1.Object); !ok {
357                  tombstone, ok := obj.(cache.DeletedFinalStateUnknown)
358                  if !ok {
359                          utilruntime.HandleError(fmt.Errorf("error decoding object, invalid type"))
360                          return
361                  }
362                  object, ok = tombstone.Obj.(metav1.Object)
363                  if !ok {
364                          utilruntime.HandleError(fmt.Errorf("error decoding object tombstone, invalid type"))
365                          return
366                  }
367                  logger.V(4).Info("Recovered deleted object", "resourceName", object.GetName())
368          }
369          logger.V(4).Info("Processing object", "object", klog.KObj(object))
370          if ownerRef := metav1.GetControllerOf(object); ownerRef != nil {
371                  // If this object is not owned by a Foo, we should not do anything more
372                  // with it.
373                  if ownerRef.Kind != "Foo" {
374                          return
375                  }
376
377                  foo, err := c.foosLister.Foos(object.GetNamespace()).Get(ownerRef.Name)
378                  if err != nil {
379                          logger.V(4).Info("Ignore orphaned object", "object", klog.KObj(object), "foo", ownerRef.Name)
380                          return
381                  }
382
383                  c.enqueueFoo(foo)
384                  return
385          }
```

# K8s Cleaner Operator

K8s Cleaner is an operator/controller that helps you to

- Identify unused or unhealthy resources
- Resource scheduling
- Resource removal or update
- Notifications via Slack, Webex and more.


Ref: https://github.com/gianlucam76/k8s-cleaner

# K8s Cleaner - Implementation Details

1. You'll create a custom resource of kind "Cleaner"
2. If you want to automate to delete any resource, you'll mention the scheduled time in the custom resource
3. If the scheduled time matches the current time the controller add it in the queue to process
4. The process function will take appropriate action. In this case the resource will be deleted

```yaml
apiVersion: apps.projectsveltos.io/v1alpha1
kind: Cleaner
metadata:
  name: cleaner-sample
spec:
  schedule: "* 1 * * *" # Runs every day at 1 AM
  resourcePolicySet:
    resourceSelectors:
    - namespace: test
      kind: Secret
      group: ""
      version: v1
  action: Delete # Deletes matching Secrets
```

# K8s Cleaner - Code Walkthrough

```
128  func (r *CleanerReconciler) reconcileNormal(ctx context.Context, cleanerScope *scope.CleanerScope,
129          logger logr.Logger) (reconcile.Result, error) {
130
131          logger.Info("reconcile Cleaner instance")
132
133          // old finalizer (cleanerfinalizer.projectsveltos.io) caused an warning message.
134          // Since we switched to new one, remove old one if ever set.
135          r.removeOldFinalizer(cleanerScope)
136
137          if err := r.addFinalizer(ctx, cleanerScope.Cleaner, appsv1alpha1.CleanerFinalizer); err != nil {
138                  logger.Info(fmt.Sprintf("failed to add finalizer: %s", err))
139                  return reconcile.Result{}, err
140          }
141
142          executorClient := executor.GetClient()
143          result := executorClient.GetResult(cleanerScope.Cleaner.Name)
144          if result.ResultStatus != executor.Unavailable {
145                  if result.Err != nil {
146                          msg := result.Err.Error()
147                          cleanerScope.SetFailureMessage(&msg)
148                  } else {
149                          cleanerScope.SetFailureMessage(nil)
150                  }
151          }
152
153          now := time.Now()
154          nextRun, err := schedule(ctx, cleanerScope, logger)
155          if err != nil {
156                  logger.Info("failed to get next run. Err: %v", err)
157                  msg := err.Error()
158                  cleanerScope.SetFailureMessage(&msg)
159                  return ctrl.Result{}, err
160          }
161
162          logger.Info("reconcile Cleaner succeeded")
163          scheduledResult := ctrl.Result{RequeueAfter: nextRun.Sub(now)}
164          return scheduledResult, nil
165  }
```

# Return Options - Kubebuilder Documentation

**Return Options**

The following are a few possible return options to restart the Reconcile:

- With the error:

```
return ctrl.Result{}, err
```

- Without an error:

```
return ctrl.Result{Requeue: true}, nil
```

- Therefore, to stop the Reconcile, use:

```
return ctrl.Result{}, nil
```

- Reconcile again after X time:

```
return ctrl.Result{RequeueAfter: nextRun.Sub(r.Now())}, nil
```

Ref: https://book.kubebuilder.io/getting-started#return-options

# K8s Cleaner - Code Walkthrough

```
122  func (m *Manager) startWorkloadWorkers(ctx context.Context, numOfWorker int, logger logr.Logger) {
123          m.mu = &sync.Mutex{}
124          m.dirty = make([]string, 0)
125          m.inProgress = make([]string, 0)
126          m.jobQueue = make([]string, 0)
127          m.results = make(map[string]error)
128          k8sClient = m.Client
129          config = m.config
130          scheme = m.scheme
131
132          for i := 0; i < numOfWorker; i++ {
133                  go processRequests(ctx, i, logger.WithValues("worker", fmt.Sprintf("%d", i)))
134          }
135  }
```

# Demo

# Q&A

# Reference

Operator Pattern: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/
sample-controller: https://github.com/kubernetes/sample-controller
k8s-cleaner: https://github.com/gianlucam76/k8s-cleaner
expose-k8s-operator: https://github.com/roopeshsn/expose-k8s-operator
Demo: https://roopeshsn.notion.site/K8s-Cleaner-Guide-92e803d12b74407e868ea65ec9512e7b?pvs=4