



Published in The Startup

You have **2** free member-only stories left this month. [Upgrade for unlimited access.](#)



Stefanie Lai

Follow

Dec 6, 2020 · 11 min read · ✨ · 🎧 Listen



Save

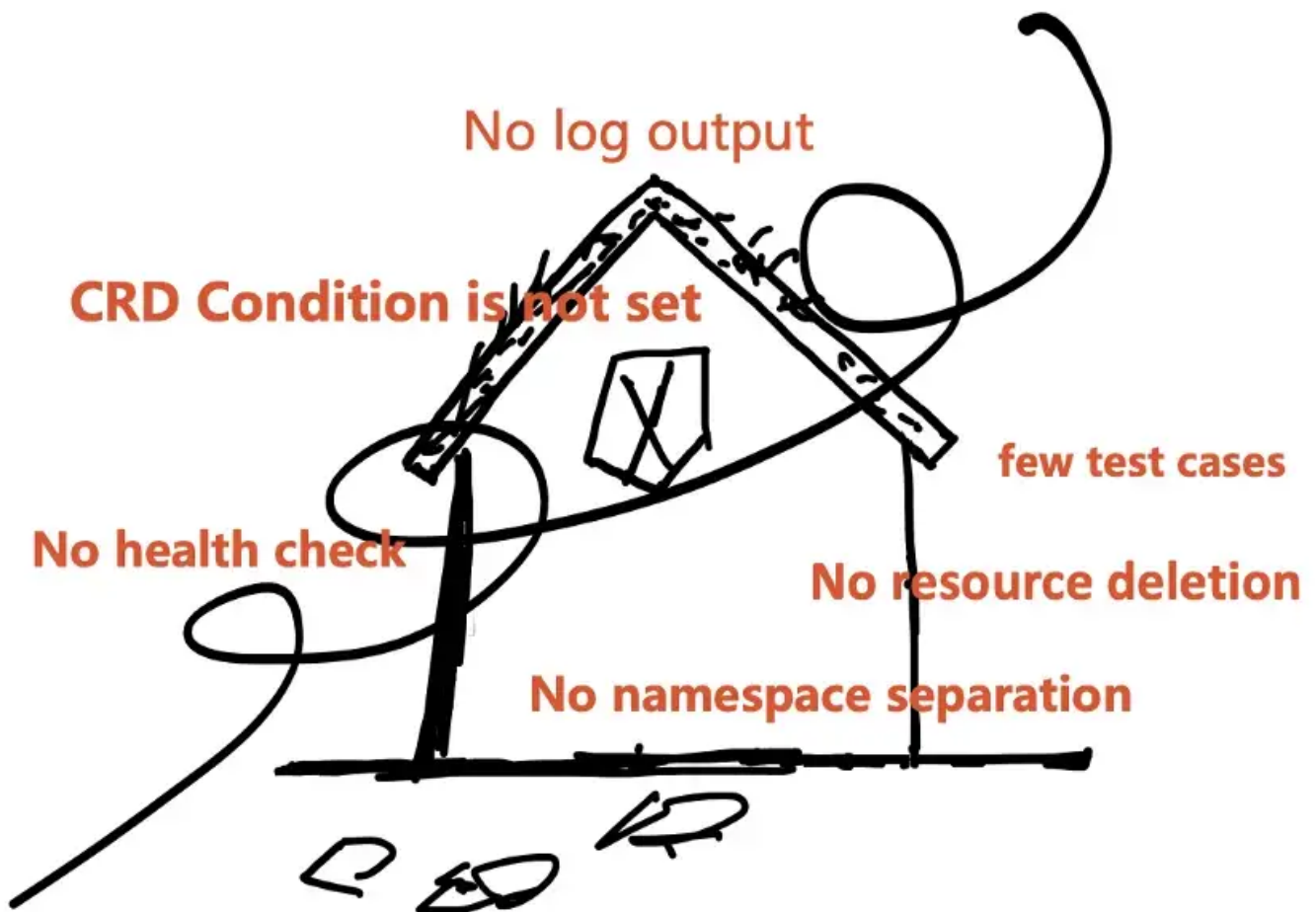


Advanced Kubernetes Operators Development

How to build a production-standard Operator based on Kubebuilder.
Tips and pitfalls

In the previous article [🔗 \(Kubernetes Operator for Beginners — What, Why, How\)](#), I have described Kubernetes Operators' concepts and how to implement one with a simple example of auto-generating `ServiceAccount` and `ClusterRoleBinding` through Kubebuilder.

But that example is rough and does not meet the production standards, just for illustration. 😞



poor operator, by author

- **CRD Condition is not set.** The `Condition` status is generally a monitoring field used by various `kubectl` tools to observe the resources' status.
- Without a health check, we can not add a **liveness probe** and **readiness probe**.

Operators that run in production obviously need more.

This article lists multiple aspects of building a stable and functional Operator by improving the previous one. And we will discuss some pitfalls encountered during the development.

Before starting, let's review what Operator Controller is and what functions we want to achieve with it in Kubebuilder.

- **Controller implements and manages the reconciliation loop**
- **Controller reads the desired state from the resources' YAML and makes sure they reach the expected state**

I recommend everyone to read Operator Best Practice before implementing an operator. I briefly conclude the points that are meaningful to me.

- **Do one thing and do it well**, which is consistent with SRP. It is necessary to follow this principle for better stability, performance, and less difficulty in development and expansion.
- **One controller controls or owns one CRD**, which aligns with the above one.
- **Namespace is configurable**, which is also important and considered to be following Kubernetes Namespace Best practice. In our code case, it is a good strategy to put different `ServiceAccounts` in different namespaces.
- **Expose metrics**. It should go without saying that we need Prometheus to monitor our Operators.
- **An Operator should not relate to another Operator**. Always keep in mind to keep it simple since a too complicated Operator is of no help.
- **Use webhooks to validate CRD input**. When your CRD has different versions, webhooks are very important.

There are more points, and I won't expand here. Our need for a “fancy” Operator is like the pursuit of a nice house.



Fancy Operator, by author

Improve Our Operator

My goal is to make the Operator more stable and reliable. I will keep the `UserIdentity` code for comparison, and develop it on a new Kind and add new content.

So the first step is using the Kubebuilder tool to create the next version of the CRD type and generate the controller.

```
kubebuilder create api --group identity --version v2 --kind
UserIdentityv2
```

If I don't use the new Kind, Kubebuilder will not generate a new controller but requires to write the reconcile logic of two versions in the same controller.

The `Create` command will generate Go classes such as `useridentityv2_types.go` in the `api/v2` directory.

First, I will straightly copy the fields of `v1.UserIdentity`.

The command will also create a new `useridentityv2_controller.go` file in the controller directory and copy the similar logic from the `v1` controller here simultaneously.

Add Logs

The first step of optimization is more logs.

Kubebuilder embeds `logr` in its own framework to record logs, which we can use as well.

Add a variable name to the ignored default log object.

```
log := r.Log.WithValues("useridentity", req.NamespacedName)
```

Generally, we will add logs in err processing.

```
if err != nil {
    log.Error(err, fmt.Sprintf("Error create ServiceAccount for user:
%s, project: %s", user, project))
}
```

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

We also add business logs at key points.

```
log.V(10).Info(fmt.Sprintf("Create Resources for User:%s,
Project:%s", user, project))
log.V(10).Info(fmt.Sprintf("Create ServiceAccount for User:%s,
Project:%s finished", user, project))
```

Tips on verbosity logs:


Verbosity-levels on info logs. This gives developers a chance to indicate arbitrary grades of importance for info logs, without assigning names with semantic meaning such as “warning”, “trace”, and “debug”.

from — <https://github.com/go-logr/logr>

Because of its unique flexibility, Verbosity log has appeared in more and more Go open-source frameworks and has gradually become a prevalent standard. Better isolation greatly improves the convenience of debugging.

When viewing the log with `kubectl`, we use the following command to view verbosity logs. ({{{}} is the variable value that needs to be replaced and the same for the following content in the article.)

```
kubectl get po -n {{ns}} -L {{label}}={{value}} --sort-
by='{{field}}' -v10
```

Be prudent about logs, don't log too much. No one wants to be buried under the millions of logs when debugging. If you are curious about Kubernetes logging mechanism.  [here](#)

Set Conditions

In Kubernetes's Resources management, conditions are very crucial concepts and associate with the Pod lifecycle. Setting the resource condition reasonably in the sync loop is necessary when using probes functions such as readiness probes. This

blog [What the heck are Conditions in Kubernetes controllers?](#) gives a more detailed explanation.

To set Conditions for CRD, we need to add the conditions field to the status definition of `UserIdentity`.

```
type UserIdentityV2Status struct {  
    // Conditions is the list of error conditions for this resource  
    Conditions status.Conditions `json:"conditions,omitempty"`  
}
```

Then you can modify the CRD conditions in the key position of the controller.

- Add the condition of `UpdateFailed` where `err` appears.

```
1      if err != nil {  
2          log.Error(err, fmt.Sprintf("Error create ServiceAccount for user: %s, p",  
3              _ = r.SetConditionFail(err, userIdentity, log)  
4              return ctrl.Result{}, nil  
5      }  
6  
7      // create a func to wrap the logic  
8      func (r *UserIdentityV2Reconciler) SetConditionFail(err error, userIdentity id  
9      conditions := userIdentity.GetConditions()  
10     condition := status.Condition{  
11         Type:    Ready,  
12         Status:  v1.ConditionFalse,  
13         Reason:  UpdateFailed,  
14         Message: err.Error(),  
15     }  
16     if conditions.SetCondition(condition) {  
17         if err := r.Status().Update(context.Background(), &userIdentity); err != nil {  
18             log.Error(err, "Set conditions failed")  
19             //r.Recorder.Event(userIdentity, corev1.EventTypeWarning, string(err))  
20             return err  
21         }  
22     }  
23     return nil  
24 }
```

- Set the condition into UpToDate after it succeeds.

```
condition := status.Condition{
    Type:    Ready,
    Status:   v1.ConditionTrue,
    Reason:   UpToDate,
    Message:  err.Error(),
}
```

After our controller successfully sets the condition, we can simply find the problematic CRD via `kubectl`.

```
kubectl get po -n {{ns}} -L {{label}}={{value}}
```

Add Health Checks

One of the most important functions of Kubernetes as an orchestration tool is to close unhealthy Pods and restart Pods automatically.

Simply put, this feature requires support from the readiness probe and liveness probe. For CRD, you need to add code supportive of readiness probe and liveness probe in reconciling.

Here is an example of a liveness probe and the health check it needs.

```

1  package health
2
3  import (
4      "fmt"
5      "net/http"
6      "sync/atomic"
7      "time"
8
9      "github.com/go-logr/logr"
10     k8sclock "k8s.io/apimachinery/pkg/util/clock"
11 )
12
13 type HealthCheck interface {
14     // Check implements healthz.Checker
15     Check(*http.Request) error
16     // Trigger update last run time
17     Trigger()
18 }
19
20 type healthCheck struct {
21     clock      k8sclock.PassiveClock
22     tolerance  time.Duration
23     log        logr.Logger
24     lastTime   atomic.Value
25 }
26
27 var _ HealthCheck = (*healthCheck)(nil)
28
29 // NewHealthCheck creates a new HealthCheck that will calculate the time inactive
30 // based on the provided clock and configuration.
31 func NewHealthCheck(clock k8sclock.PassiveClock,
32     tolerance time.Duration,
33     log logr.Logger) HealthCheck {
34
35     answer := &healthCheck{
36         clock:      clock,
37         tolerance:  tolerance,
38         log:        log.WithName("HealthCheck"),
39     }
40     answer.Trigger()
41     return answer
42 }
43
44 func (a *healthCheck) Trigger() {
45     a.lastTime.Store(a.clock.Now())
46 }
47
48 func (a *healthCheck) Check(*http.Request) error {

```



```

48 func (a healthChecker) check(_ http.Request) error {
49     lastActivity := a.lastTime.Load().(time.Time)
50     if a.clock.Now().After(lastActivity.Add(a.tolerance)) {
51         err := fmt.Errorf("last activity more than %s ago (%s)",
52             a.tolerance, lastActivity.Format(time.RFC3339))
53         a.log.Error(err, "Failing activity health check")
54         return err
55     }
56     return nil
57 }

```

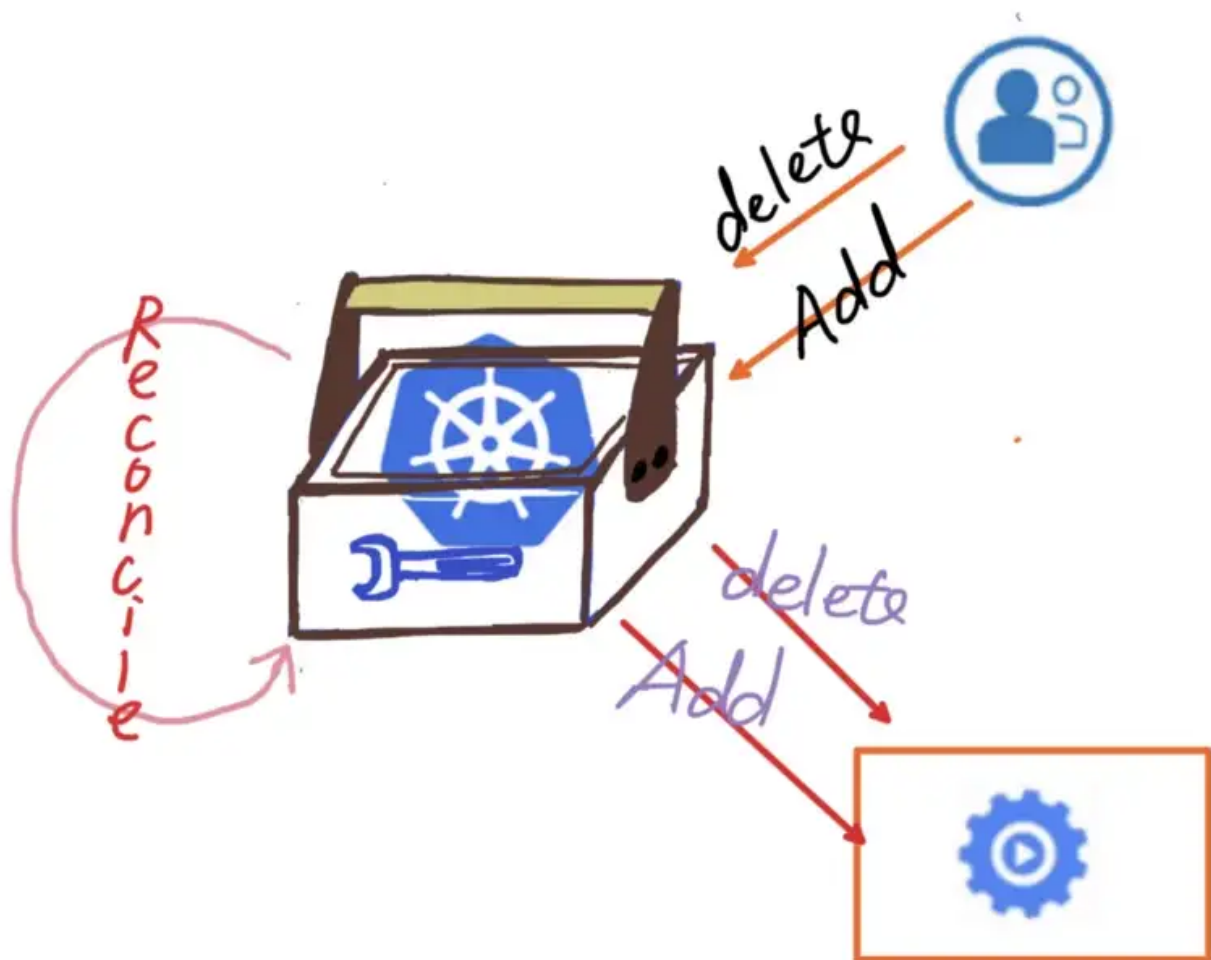
It's not the CRD YAML but the Deployment manager YAML file.

Add resource deletion logic

Our sync loop only has the code of increasing users and configuring related resources. However, when a user is deleted, the related resources also need to be deleted.

Its way of implementation depends on how we receive user update information.

- `FindAll` interface. Get all users, then add and delete related resources by comparison.
- Event notification. Either adding or deleting users can be achieved by subscribing to the upstream events, which is currently the most prevailing event-driven pattern.



Event-driven, by author

Kubernetes APIServer naturally supports event-driven with the watch function.

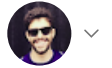
So there are only two things that need to be done in the Controller.

- Create events that need to notice. Here we watch the user update event from Pubsub Topic.

```
1 package controllers
2
3 import (
4     "context"
5     "encoding/json"
6     "reflect"
```

Open in app ↗

Get unlimited access



```
11     ctrl "sigs.k8s.io/controller-runtime"
12     "sigs.k8s.io/controller-runtime/pkg/client"
13     "sigs.k8s.io/controller-runtime/pkg/event"
14 )
15
16 type UserEvents struct {
17     ctx      context.Context
18     log      logr.Logger
19     client    client.Client
20     subscription *pubsub.Subscription
21     lock      sync.RWMutex
22     users     chan<- event.GenericEvent
23 }
24
25 type Event struct {
26     UserName string `json:"userName,omitempty"`
27     Project  string `json:"project,omitempty"`
28 }
29
30 func CreateUserEvents(client client.Client, subscription *pubsub.Subscription, users ch
31     log := ctrl.Log.
32         WithName("source").
33         WithName(reflect.TypeOf(UserEvents{}).Name())
34     return UserEvents{
35         ctx:      context.Background(),
36         log:      log,
37         client:    client,
38         subscription: subscription,
39         lock:      sync.RWMutex{},
40         users:     users,
41     }
42 }
43
44 func (t *UserEvents) Run() {
45     for {
46         select {
47             case <-t.ctx.Done():
48                 return
```

```

48         return
49     default:
50     }
51
52     err := t.subscribe()
53     if err != nil {
54         t.Errorf("failed to subscribe event")
55     }
56 }
57 }
58
59 func (t *UserEvents) subscribe() error {
60     return t.subscription.Receive(t.ctx, func(ctx context.Context, msg *pubsub.Message) {
61         log := t.log.WithValues("messageId", msg.ID)
62         userEvent := Event{}
63         if err := json.Unmarshal(msg.Data, &userEvent); err != nil {
64             log.Error(err, "unable to unmarshal event")
65             msg.Nack()
66             return
67         }
68
69         list := v2.UserIdentityV2List{}
70         if err := t.client.List(context.Background(), &list); err != nil {
71             log.Error(err, "unable to get UserIdentity")
72             msg.Nack()
73             return
74         }

```



reliable operator, by author

Let's take a look at how to implement some of these functions.

Use Kubebuilder features

Kubebuilder's scaffold provides many special functions, helpful to our development and practice. Among them, the comment function is one of the most convenient and useful.

Take an example to see the advantages when adding comments to the types field.

- Add more detailed info to CRD. For example, if you add the following comment to the field, the field information can be displayed in the output of `kubectl get`.

```
// +kubebuilder:printcolumn
```

- Add a default value or verification to the CRD field. For example, adding the following comment to the field enables us to limit the field value to 1, 2, 3; otherwise, an error will be reported.

```
// +kubebuilder:validation:Enum=1,2,3
```

- Stop the ApiServer from pruning fields that are not specified.

```
// +kubebuilder:pruning:PreserveUnknownFields
```

For more comments on Kubebuilder, please refer to the [Kubebuilder book](#).

With our code, we can print the `RoleRef` column used by the `UserIdentityV2`.

```
// +kubebuilder:printcolumn
RoleRef rbacv1.RoleRef `json:"roleRef,omitempty"`
```

Support unstructured data

In CRD design, sometimes we have to jump out of the box. We can't limit ourselves to using the current Kubernetes API or resource types and try to complete some special operations with unstructured data.

Take `UserIdentity` as an example. Here we hardcode the creation of `ServiceAccount` and `ClusterRoleBinding`, which results in the following problems.

- We need to use the `core/v1` and `rbac/v1` libraries corresponding to `ServiceAccount` and `ClusterRoleBinding`. The more resources we create, the more associated APIs, but not all required types are supported by go types.
- We cannot create the required resources by modifying our CRD dynamically. Once the modification is required, you must change the controller logic.

So it is of great importance to support unstructured structure here. We can switch our CRD design to the following YAML:

The code is required to support the template parsing, and here let's modify it in `UserIdentityV3_types.go`. You may have noticed that I have created a `UserIdentityV3` to implement the unstructured function to compare the code of v1 and v2 versions better.

```
// Template is a list of resources to instantiate per repository in
Governator
Template []unstructured.Unstructured `json:"template,omitempty"`
```

The template here essentially uses the [Go template](#) function, allowing us to parse the template, inject parameters in the controller, and create objects by [unstructured API](#). Look at the code.

Support Events

The v2 version already supports conditions, while there is still a missing feature, **events**.

In Kubernetes, resources indicate the resource status changes and other noteworthy information by emitting various events so that users can obtain information through the `kubectl get events` command, avoiding submerging in massive logs.

Here we need Kubernetes `client-go/tools/record` package.

```
import "k8s.io/client-go/tools/record"
```


And we need to define a Recorder in our reconcile.

```
Recorder          record.EventRecorder
```

Then we can start to emit events.

```
r.Recorder.Event(&userIdentity, corev1.EventTypeNormal,  
string(condition.Reason), condition.Message)  
// or  
r.Recorder.Event(&userIdentity, corev1.EventTypeWarning,  
string(UpdateFailed), "Failed to update resource status")
```

Support Webhooks

A WebHook is an HTTP callback: an HTTP POST that occurs when something happens; a simple event-notification via HTTP POST — from kubernetes.io

Kubebuilder naturally supports webhooks, but only admission webhooks.

You can add a verification webhook to `UserIdentityV3` by executing the following command.

```
kubebuilder create webhook --group identity --version v3 --kind  
UserIdentityV3 --defaulting --programmatic-validation
```

Then a `useridentityv3_webhook.go` will be generated in the `api/v3/` directory. And the webhook is set into the manager in `main.go` by default.

```
if err =  
(&identityv3.UserIdentityV3{}).SetupWebhookWithManager(mgr); err !=  
nil {  
    setupLog.Error(err, "unable to create webhook", "webhook",  
"UserIdentityV3")  
    os.Exit(1)  
}
```

As mentioned above, adding comments(`// +kubebuilder:validation`) to the types' field also plays a validation role, but the webhook can do more.

But for the `UserIdentity` , I haven't thought of a good way yet. If you are interested, you can add the logic you want based on my code.

Reconcile Periodically

Adding the code below at the end of the reconcile function triggers Operator reconcile every 10 minutes.

```
return ctrl.Result{RequeueAfter: 10 * time.Minute}, nil
```

Set OwnerReference

Resource ownership is basic knowledge in Kubernetes, helps to delete the sub-resources when deleting the owner resource.

If we run `kubectl delete useridentityv3` , it deletes the sub-resources by default. Of course, we can set `cascade = false` to disable it.

The goal here is to facilitate the Kubernetes Garbage Collection, especially when there are multiple resources deeply coupled. For example, consider the tight bound between `Deployment->Service->Pod`.

Back to our code, we can set the ownership of all the unstructured resources to our CRD. Then we can delete them by deleting the operator itself.

```
return ctrl.SetControllerReference(userIdentity, &existing,  
r.Scheme)
```

Avoid Pitfalls



pitfalls, by author

Although my overall experience with Kubebuilder is good, problems inevitably arise, such as problems with Kubebuilder itself, issues with Kubernetes `controller-runtime` library, and even some with Golang, including the use of Google's Pubsub.

The three most representative ones are

Crazy Ginkgo Tests

I don't like Kubebuilder's `suite_test` and `ginkgo` framework, or maybe I'm not using it correctly. 🙄

- We cannot test different test cases separately in the IDE. Unlike JUnit that you can run or debug a test case separately, `suite_test` is not supported, at least in Goland. Think about how crashed when you have 20 or 100 more test cases. 🤖
- The race condition is annoying. When we design and write unit tests, we generally follow F.I.R.S.T principles. However, the isolated principle has been repeatedly violated in the `ginkgo` test, for which I still don't understand the root

cause. Even if I create different CRDs with different names and use different reconcilers in the same `suite_test`, race will often occur (enable race detector) and cause super flaky unit tests.

CreateOrUpdate

It is a real pitfall. If you only read the method comments, you will have a simple thought that I pass in one object, and if its field value has not changed, then Update will naturally be triggered.

```
// CreateOrUpdate creates or updates the given object obj in the Kubernetes  
// cluster. The object's desired state should be reconciled with the existing  
// state using the passed in ReconcileFn. obj must be a struct pointer so that  
// obj can be updated with the content returned by the Server.
```

In practice, it is not the case. We discover the resources we created by Operator are all deleted and recreated during CRD's upgrade process, even if there is no change in these resources at all.

When we take a closer look at the `CreateOrUpdate` code, we find that this object executes the copy relying on the `DeepCopy` function in the `zz_generated.deepcopy.go` that automatically generated by the Kubebuilder tool after we define the CRD.

If your object is unstructured or dynamically generated and without implementing the `DeepCopy` function, then you need to recreate it every time!

To solve this problem, you need the `mutate` func defined by this method.

```
// The MutateFn is called regardless of creating or updating an  
object.  
//  
// It returns the executed operation and an error.  
func CreateOrUpdate(ctx context.Context, c client.Client, obj  
runtime.Object, f MutateFn) (OperationResult, error) {}
```

Initializing object with `mutataFn` can avoid rebuilding every time.

Set Context Timeout

We met an error when we launched our operator. The operator stuck at some point, and there were no logs, no events and nothing to help the debug.

Then we reviewed the code again and again, and we literally commented on each part of the code to test.

Finally, we found out that the upstream gRPC interface we connected to was already relocated, but somehow the connection didn't timeout or break, causing our operator stuck forever. We changed the gRPC interface to SRV connection to solve the issue.

We still want our operator reliable enough to exit instead of block. It just needs a small change in the `context` we use everywhere.

```
ctx, cancel := context.WithTimeout(context.Background(),
20*time.Minute)
defer cancel()
```

Always add time out to the context!

Besides the aforementioned three issues, other issues will also arise. But as a developer, locating the problems and working them out is part of our daily life. 😊
Don't forget about creating bugs.

To Sum up

After adding all the new code, `UserIdentityV3` is now a more decent Operator and ready for production 100.

Putting aside the shortcomings mentioned above, Kubebuilder is indeed a handy Operator scaffolding tool, which spares developers more time so that we can focus on developing controller logic.

And by implementing Operator, we better understand the internal operating principles of webhook, controller plane, etc., understand the open-source code in the Kubernetes SIG, and even contribute code to some SIGs if interested.

Kubebuilder 3.0 is on its way. Join Kubernetes slack, Google group, or follow this document to learn more.

All the code related to this article is on Github. I will keep writing when I have more to share.

Thanks for reading!

Sign up for Top 5 Stories

By The Startup

Get smarter at building your thing. Join 176,621+ others who receive The Startup's top 5 stories, tools, ideas, books — delivered straight into your inbox, once a week. [Take a look.](#)

Emails will be sent to migueleliasweb@gmail.com. [Not you?](#)



Get this newsletter