Best Practices for Building Kubernetes Operators

Learn some best practices for writing production-ready Kubernetes operators.

We'll cover the following

Overview

Kubernetes operators provide a way to extend Kubernetes functionality to handle automating tasks, application-specific logic, custom resources, and so on. With operators, we can borrow many good designs and built-in supports from Kubernetes, such as declarative APIs, state-driven reconciling logic, and event informers. We can have easy-to-use APIs in the form of CRDs, where we can still use `kubectl` to interact with Kubernetes.

In this lesson, we'll go over some best practices for creating and maintaining operators. Kubernetes exposes REST HTTP APIs. This means that operators can actually be implemented in any programming language, such as Go, Java, etc. However, normally, we implement using Go, so we can use various Go libraries around the Kubernetes ecosystem. This is why most developers and SRE operators use Go to implement their operators.

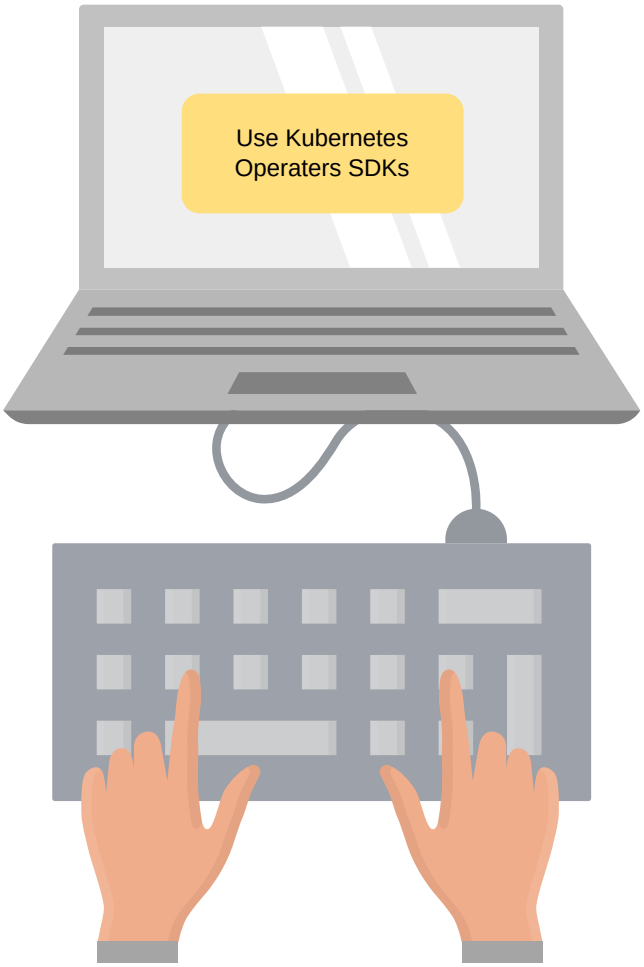
Best practices for writing operators

Here are some best practices for writing a good, production-available operator. Some of the points we discuss in this part describe common rules, which can be applied independently from implementation details.

1. Use the Operator SDK like the `kubebuilder`

Writing an operator from scratch is never an easy task, even for a very experienced developer who knows Kubernetes well. There are lots of low-level details that have steep learning curves, such as how operators work, how an event gets queued and reconciled, how the libraries are implemented, and so on. Moreover, we're not building and maintaining a single operator—therefore, it's not recommended to have heterogeneous operator frameworks, which would definitely bring us trouble when trying to understand each framework or debug operator issues. That's why we suggest using an SDK to help scaffold our operator projects in the first place.

There are also some other SDKs or frameworks to scaffold an operator project, and `kubebuilder` is one of the most popular ones. The `kubebuilder` is a comprehensive development kit that can help us scaffold a completed operator project (such as APIs, controller logic, hack scripts, `Dockerfile`, `Makefile`, etc.) within a minute. It provides all the necessary dependencies and controller logic, which helps reduce the amount of work we need to do and lets us focus solely on implementing the core business logic.



Use the Operator SDK

2. Use declarative APIs

As we know, Kubernetes uses declarative APIs. This holds true for operators as well. With declarative APIs, users only need to describe their desired state, and leave it to the operator to take all the necessary actions to get there.

3. One operator per application

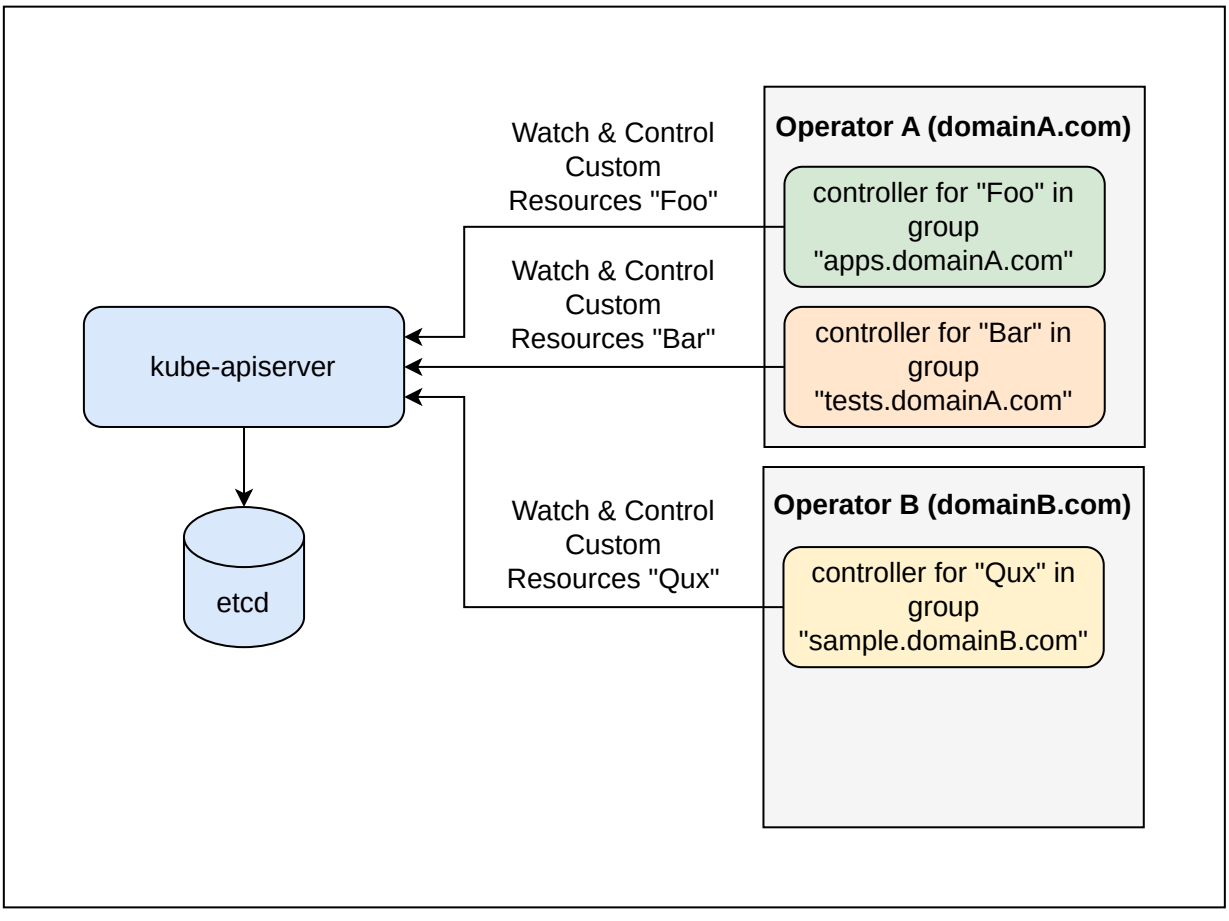
We may have multiple operators. It's recommended to let one operator handle only one application. For example, a MySQL operator is only used to handle MySQL-related operations, and a Redis operator focuses only on the Redis part. We don't mix these application-specific operators into one. The `kubebuilder` provides the capability to generate such a scaffold operator, which helps us with this.

4. Compartmentalize controllers in an operator

In the `kube-controller-manager`, we can see 30+ controllers running. Each controller keeps track of one specific resource and does its own part. Taking the `Deployment` controller as an example, it's the only controller that takes charge of the changes on `Deployment`. This essentially follows the UNIX principle—do one thing and do it well.

We should also follow this principle when writing operators. Operators should own CRDs, and only one operator should be allowed to control a CRD. Two operators managing the same CRD/API may lead to chaos. Inside an operator, we can use multiple controllers that specifically handle each of them, if we've got multiple CRDs to be managed. This not only provides good separations, but also improves code readability. However, this doesn't necessarily mean we need to build one container image per controller. The operator can have a main loop to spawn and manage multiple application controllers, exactly like what the `kube-controller-manager` does. Each controller should be responsible for only one specific CRD.

With the `kubebuilder`, we can plumb multiple controllers into an operator, as shown below:

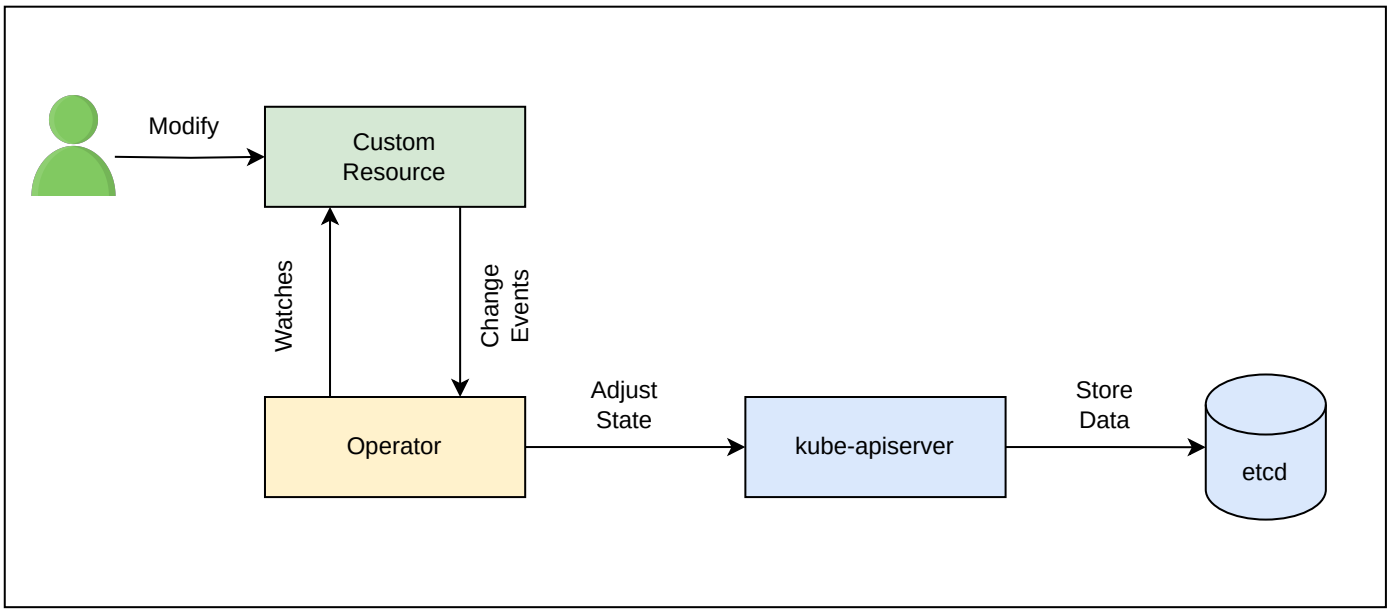


5. Fail fast in the reconciling loop

It is common for errors to occur when reconciling an object to a desired state. In this case, we should immediately terminate the current processing logic (usually, we cancel the Go `context` to do this) and return the error. The work queue will schedule a next-round reconciling. In operators, we shouldn't block the controller by continually polling and driving the state to desired until the error is resolved. The sync loops should be asynchronous.

6. One modification at a time

Our operator keeps watch for resources being changed. The reconcile loop will be triggered for every changed event, such as `CREATE`, `UPDATE`, and `DELETE`. These changes aren't solely made by users, but sometimes also by our reconcile function or its subroutines. It is quite normal to make modifications on resources, such as updating the status, patching labels, garbage collections, etc. These updates let the reconcile loop use updated caches of resources and start another round of the reconciling loop.



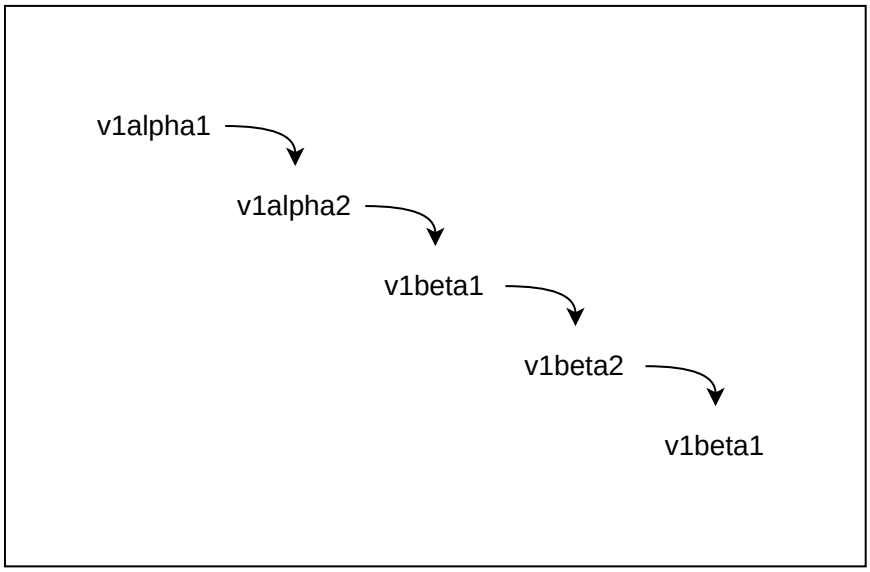
Operator reconciling loop

When writing the reconciling loop of our operators, we should be careful to not introduce operations that could result in race conditions. There shouldn't be a second loop working on the same resource at the same time. Even if the triggered requests are not processed in parallel, we should not pile them up. We need to identify whether or not we should enqueue the requests over and over again, which may keep our operator unnecessarily busy.

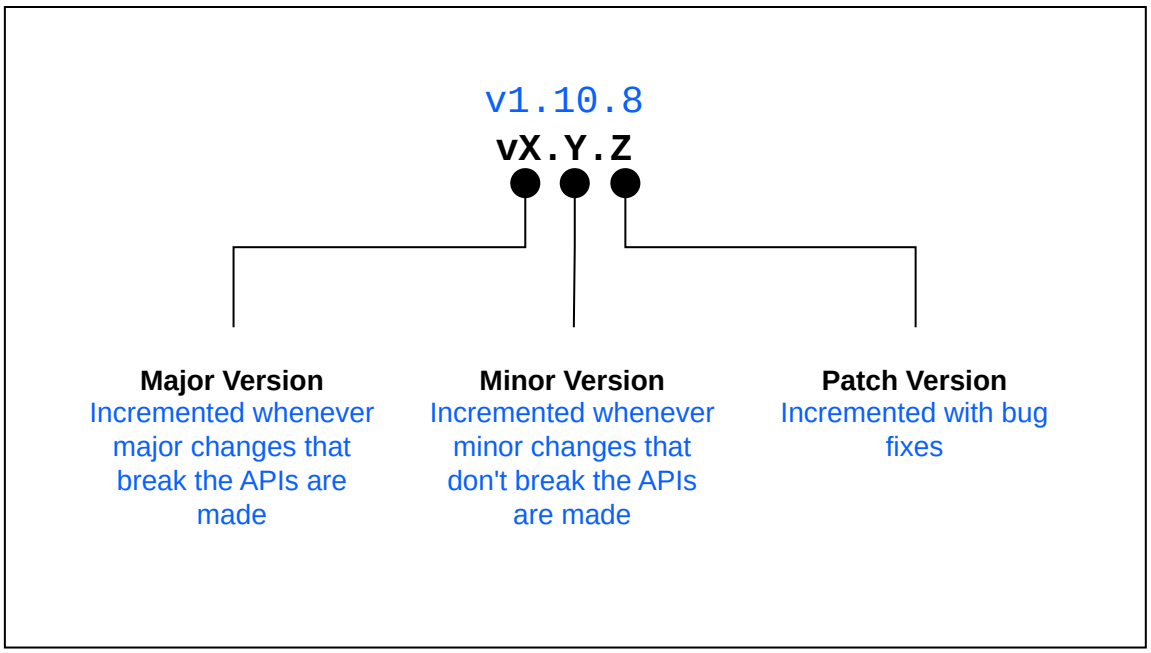
Thus, to lower the risk of race conditions and unnecessary piling up, we need to make sure that we don't perform multiple changes in a single reconciling run loop. Whenever we make changes to the resources we're handling, just exit the current reconciling loop and let the next reconciling loop continue the future actions.

7. Semantic versioning operators and APIs

Kubernetes ships its own APIs with versioning, such as `apps/v1` and `apps/v1beta1`. Using semantic versioning (also know as `semver`) brings us lots of benefits. It's easy for users to know exactly whether there are any breaking changes in the APIs. The versioning guidelines should be applied to CRDs as well. Moreover, when we're defining CRDs, the OpenAPI spec is used to create a structural schema for our CRDs. This part of `work` has been automatically addressed by the Operator SDK `kubebuilder`. After we modify the file `types.go`, running `make manifests` will automatically generate these specs for us. The `kube-apiserver` will use the validation rules set by the OpenAPI spec to help reject invalid custom resources.



Apart from CRDs, `semver` should be used to version our operators as well. Operators are long-running applications that process versioning APIs and provide various capabilities. Therefore, we should version our operator to help determine breaking or non-breaking changes, new features, backwards compatible APIs, and so on.



Basic syntax of semantic versioning

Note that operators need to support handling legacy APIs/resources that were created by an older version of the operator. When our CRDs get bumped to a new version, which may or may not be backwards compatible, operators are updated accordingly to handle a range of supported versions. There are multiple approaches to help handle multiple versioned APIs.

Fanout operators

Inside of our operator, we handle these multi-versioned resources respectively. Each controller is tied to a specific version and only watches changes of that version. With this approach, we can bundle them into a single operator or run them in multiple operators but with different versions.

However, this approach is not recommended, because it is counter to the best practice of having one operator per application.

Internal version

We can converge the multi-versioned APIs to an internal version. In the `kube-apiserver`, all those versioned APIs have a corresponding internal version. Before processing the versioned object, we can convert the objects to the matching internal version.

The side effect of this approach is that those legacy objects persist in the cluster, and we're not getting rid of them. This will become a burden for us if we've got a dozen or more versions.

Version converting (recommended approach)

CRD can define multiple versions for custom resources. We can mark a version as served or not, while only one version can be marked as the storage version. This storage version will be used by the `kube-apiserver` for ultimately storing to `etcd`.

In CRD, there is a field `spec.conversion.webhook` where we can specify the versions and webhook service. If there are schema differences between these versions, especially backwards incompatible versions, conversion webhooks are needed. If there are no schema differences, the `kube-apiserver` will handle the version conversions for us.

This is the recommended approach for bumping CRD APIs.

8. Monitoring and logging our operators

We can't forget to enable logging and monitor for our operators. Our operators are applications as well. We need to know exactly what happens inside of an operator.

Conclusion

The good news? The best practices above have already been adopted by the `kubebuilder` framework to help us rapidly build and publish Kubernetes APIs and operators in Go. The `kubebuilder` framework reduces the complexity of writing and managing operators.

Back