

How (not) to build Kubernetes Operators

Philipp Schmitt, Simon Kienzler

Mannheim, 20.11.2025



inovex

These two definitely know how not to do it:

Philipp Schmitt

Cloud Platform Engineer
Kubernetes Trainer
String Templating Officer



Simon Kienzler

Cloud Software Developer
Helped build a KaaS offering
Guilty Pleasure: Writing Docs



Who here has already
used a Kubernetes
Operator?



inovex

Who here has already
built a Kubernetes
Operator?



inovex

→ AGENDA

- Short Recap: What are Operators?
- How to build an Operator? (Basics)
- How not to build an Operator?
- Conclusion



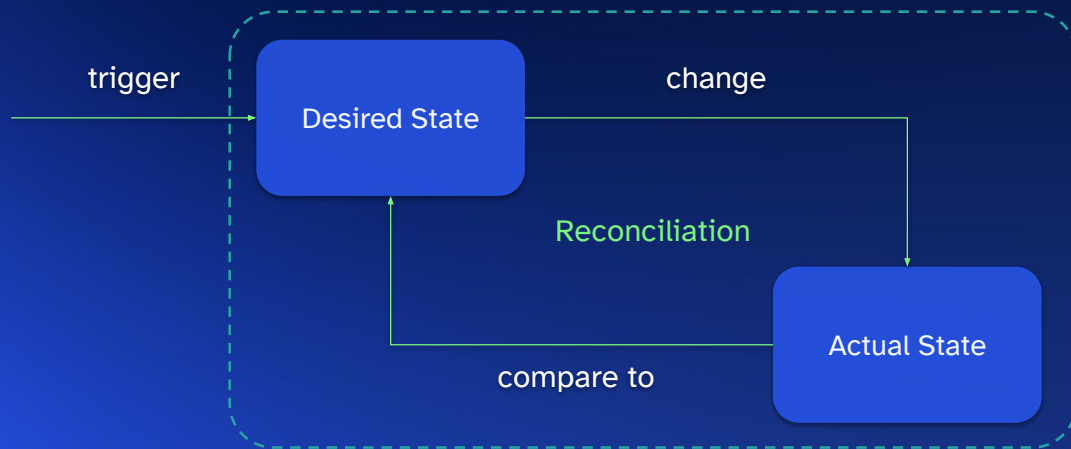
What are Operators?

A short recap.



Operator Pattern in Kubernetes

- “extend the cluster’s behaviour without modifying the code of Kubernetes itself”^[1]
- Operators are clients of the Kubernetes API that act as controllers for a Custom Resource.”^[1]
- Comparison to IaC tools
 - Both: Declarative approaches
 - Kubernetes Operators reconcile permanently
- Examples:
 - CertManager
 - Prometheus Operator
 - ...



Expectations for the Talk

- We don't expect you to have already programmed a Kubernetes Operator on your own!
- Knowledge that is helpful to have:
 - General Kubernetes Know-How
 - Experience in using CRDs with existing Kubernetes Operators (e.g. Prometheus Operator)
 - Experience with at least one programming language - preferably Go

How to build an Operator?

The basics.



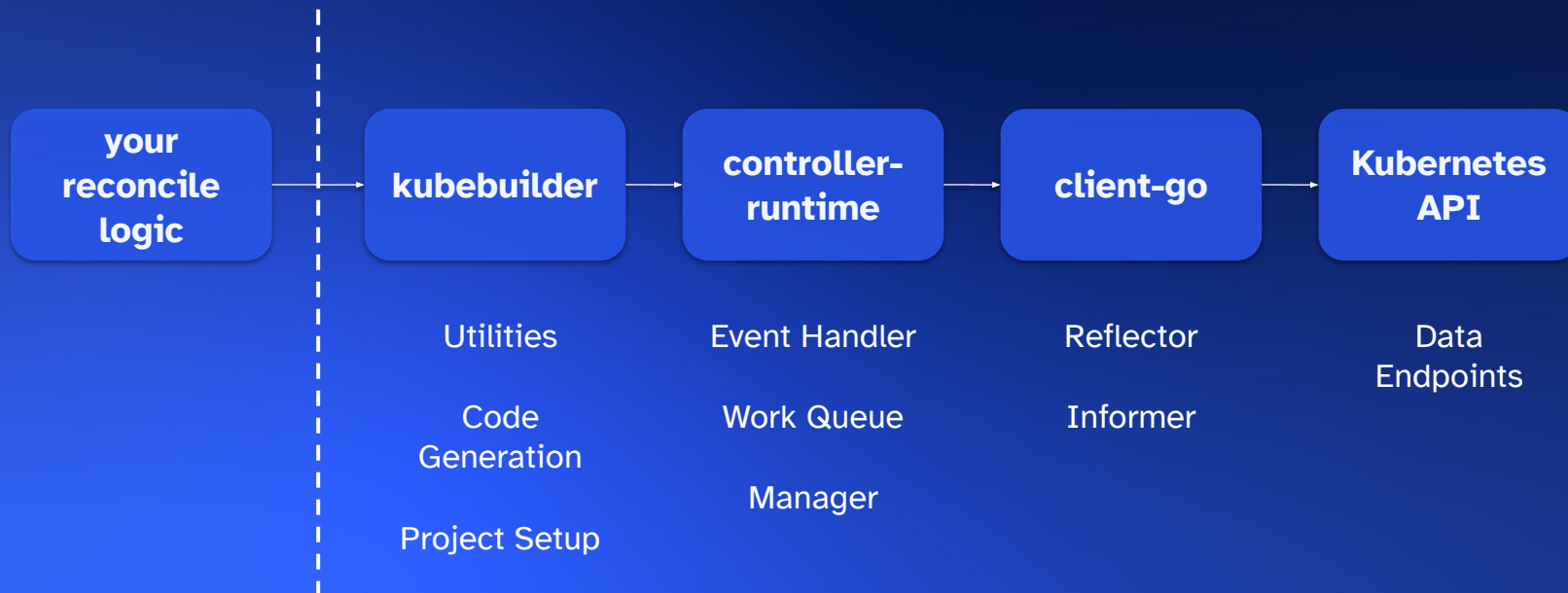
Choose wisely...

- **From scratch in any language you like**
 - great for learning
 - not so great if you want to get stuff out the door
- **<https://github.com/nolar/kopf>** - Kubernetes Operator Pythonic Framework
- **kubebuilder**
 - de-facto standard framework
 - relies on [sigs.k8s.io/controller-runtime](https://github.com/kubernetes-sigs/controller-runtime)
- **Operator SDK**
 - based on kubebuilder
 - backed by RedHat
 - additional capabilities



focus of this talk

The Operator Onion has Many Layers



How not to build an Operator. (What you actually are here for.)




Defensive Programming

```
// Bad example (panic inside reconcile func)
func (r *CLCReconciler) Reconcile(
    ctx context.Context, req ctrl.Request,
) (ctrl.Result, error) {
    err := r.doUpdate()

    if err != nil {
        log.Fatalf("Failed to update: %v", err)
    }
}
```

```
342
343 // Run runs RunOnce in a loop with a delay until context is canceled
344 ✓ func (c *Controller) Run(ctx context.Context) {
345     ticker := time.NewTicker(time.Second)
346     defer ticker.Stop()
347     var softErrorCount int
348     for {
349         if c.ShouldRunOnce(time.Now()) {
350             if err := c.RunOnce(ctx); err != nil {
351                 if errors.Is(err, provider.SoftError) {
352                     softErrorCount++
353                     consecutiveSoftErrors.Gauge.Set(float64(softErrorCount))
354                     log.Errorf("Failed to do run once: %v (consecutive soft errors: %d)", err, softErrorCount)
355                 } else {
356                     log.Fatalf("Failed to do run once: %v", err) // nolint: gocritic // exitAfterDefer
357                 }
358             } else {
359                 if softErrorCount > 0 {
360                     log.Infof("Reconciliation succeeded after %d consecutive soft errors", softErrorCount)

```

☰  kubernetes-sigs / external-dns

<> Code Issues 153 Pull requests 38 Discussions Actions Security

📁 Files

external-dns / controller / controller.go 

🔑 master

Code

Blame

373 lines (331 loc) · 11 KB

373

}

```
// Good example (error handling, retry, ...)
func (r *CLCReconciler) Reconcile(
    ctx context.Context, req ctrl.Request,
) (ctrl.Result, error) {
    // Retry on retryable errors
    err := retry.RetryOnConflict(retry.DefaultRetry, func() error {
        return r.doUpdate()
    })

    // Handle properly (logging, returning empty result and error)
    if err != nil {
        log.Errorf("Failed to do foo: %v", err)
        return ctrl.Result{}, err
    }
}
```

Object Validation

```
// Bad example (basic validation within reconcile func)
func (r *CLCReconciler) Reconcile(
    ctx context.Context, req ctrl.Request,
) (ctrl.Result, error) {
    var obj &inovexdev1alpha1.CLC{}

    // get the object from the NamespacedName,
    // lets assume this just works
    _ = r.Get(ctx, req.NamespacedName, &obj)

    if obj.Spec.Year <= 2013 || obj.Spec.year >= 2100 {
        log.Fatalf("not a valid year: %d", obj.Spec.year)
    }
}
```

Shift as far left as possible

- **kubebuilder supports generation markers that influence the resulting CRDs**
 - will be blocked by the Kubernetes API server during resource creation automatically
- **you can also drop down to CEL directly, e.g. to make sure a field is not mutated after creation**
- **For more complex validations, use Validating Webhooks**
 - allows you to hook into the creation via code and check things, e.g. across resources

```
type CLCSpec struct {  
    // Year is the year of the conference.  
    // +kubebuilder:validation:Minimum=2013  
    // +kubebuilder:validation:Maximum=2100  
    // +required  
    Year int `json:"year"`  
}
```

```
type CLCSpec struct {  
    // TalkTitle is the official title.  
    // +required  
    // +kubebuilder:validation:XValidation:rule=  
    // "self == oldSelf",message="TalkTitle is  
    // immutable"  
    TalkTitle string `json:"talkTitle"`  
}
```

When to Reconcile?

```
// Bad example (always reconcile)

func (r *CLCReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        // Watch the primary resource (CLC)
        For(&inovexdev1alpha1.CLC{}).
        Complete(r)
}
```

```

// Definition of custom predicate
updatePred := predicate.Funcs{
    UpdateFunc: func(e event.UpdateEvent) bool {
        oldObj := e.ObjectOld.(*inov1alpha1.CLC)
        newObj := e.ObjectNew.(*inov1alpha1.CLC)
        // Trigger reconciliation only
        // if the spec has changed!
        return !equality.Semantic.DeepEqual(
            oldObj.Spec,
            newObj.Spec,
        )
    },
    CreateFunc: func(e event.CreateEvent) bool {
        return true // Allow create events
    },
    // ...
}

```

```

// Good example (use predicates)
func (r *CLCReconciler) SetupWithManager(
    mgr ctrl.Manager,
) error {
    return ctrl.NewControllerManagedBy(mgr).
        // Watch the primary resource (CLC)
        For(&inov1alpha1.CLC{}).

        // Alternative 1
        // (Apply custom Predicate)
        builder.WithPredicates(updatePred),

        // Alternative 2
        // (Apply preconfigured Predicate)
        builder.WithPredicates(
            predicates.Or(
                predicate.GenerationChangedPredicate{},
                predicate.AnnotationChangedPredicate{},
            )
        ),

        Complete(r)
}

```

Reconciling Cascading Resources

How not to reconcile cascading resources

- Have a “God Controller” that does everything
- Implement ownership between resources on your own

How to reconcile cascading resources

- Unix Philosophy: Do one thing, and do it well
- One Controller = One Resource Type
- Work with **OwnerReferences** to express cascades
- Check if your requirements can be express as a cascade, like e.g. Deployment, ReplicaSet, Pod

Constructing Changes

How not to construct changes

```
// Bad example (doing everything, then updating)
func (r *CLCReconciler) Reconcile(
    ctx context.Context, req ctrl.Request,
) (ctrl.Result, error) {
    var obj CLCObj

    r.doBlockingCallFor10s()

    r.setFinalizer(obj)
    r.createOrUpdateChildResources(obj)
    r.setStatus(obj)

    if err := r.updateObj(obj); err != nil {
        return ctrl.Result{}, err
    }
}
```

How to construct changes

```
// Good example (reconcile step by step)
func (r *CLCReconciler) Reconcile(
    ctx context.Context, req ctrl.Request,
) (ctrl.Result, error) {
    var obj CLCObj
    defer func() {
        r.updateStatus(obj)
    }()

    if !hasFinalizer(obj) {
        return r.setFinalizer(obj)
    }
    if !childResourcesUpToDate(obj) {
        return r.createOrUpdateChildResources(obj)
    }

    return r.updateObj(obj)
}
```

Development Setup & Debugging

How not to set up development

- Immediately try-out changes in a shared (development) cluster
 - might break CRDs for others
- printf Statements (we all know this 😏)

How to set up development

- Local Development
 - Local cluster (e.g. using [kind](#))
 - Isolated environment for CRDs to mess up with
 - Using debugger (IDE, delve)
 - connect to local cluster
- envtest for Integration tests
 - Test API + etcd
- Production
 - Use Annotations to exclude specific CR instances

Conclusion

**What this means,
in a nutshell.**



Conclusion

- Building an operator is fun!
- You get a lot of helpful tooling out of the box with kubebuilder
- Suboptimally written controllers can work well in the beginning, then rapidly disintegrate when challenged
- Adhering to good practices can give you confidence
- We hope we could show you easy-to-implement advice to make your controllers more robust



Interested in our training?
Find out more:



New in our Portfolio

inovex Academy

Kubernetes Operator Development
Training



Kubernetes Operator Development Training

- 3 days, on-site or remote
- Trainers with real world experience
- Topics we cover in the training:
 - Designing custom Kubernetes operators
 - Best practices for developing Kubernetes operators
 - Best practices for administering Kubernetes operators



Using these Exercises

Exercise 1: Writing Your First Operator

Exercise 2: Managing Dependent Objects

Exercise 3: CRD Versioning

Exercise 4: Testing Operators

Exercise 5: Defaulting and Validation

Exercise 6: Debugging

Exercise 7: Controlling External APIs

Preparation

Implementing a Bucket Reconciler

Reacting to MinIO Events

Summary

Exercise 8: API Design and Architecture

Appendix

Exercise 7: Controlling External APIs > Implementing a Bucket Reconciler

Implementing a Bucket Reconciler

With the previously described code base, you can now start implementing the bucket controller's `Reconcile` method.

Your Task:

Implement the `Reconcile` method in

`internal/controller/bucket_controller.go` to handle the following cases:

1. **Create:** If the bucket does not exist, create it using the MinIO API.
2. **Delete:** If the bucket exists but the resource is marked for deletion, delete it using the MinIO API.
3. **Update the status:** After successfully creating the bucket, update the `status.ready` field of the `Bucket` resource.

Your implementation should pass the existing tests in

`internal/controller/bucket_controller_test.go` and

`internal/controller/bucket_controller_unit_test.go` (except for two *pending* tests, you can ignore those for now).

You can run the tests using `make test` in the root of the repository.

▼ Hint: MinIO Client

Use the `getMinioClient` function to get a MinIO client for the MinIO instance referenced in the `Bucket` resource. The client provides `BucketExists`, `MakeBucket`, and `RemoveBucket` methods.

> Hint: Handling Deletion


> Hint: Identifying Deleted Resources

> Hint: Updating the Status

> Solution


Thank you!

Philipp Schmitt

 philipp.schmitt@inovex.de



Simon Kienzler

 simon.kienzler@inovex.de



Interested in our training?
Find out more:



inovex