
KubeCon
CloudNativeCon
Europe 2025
LONDON BUS

Don't write controllers like Charlie Don't does:

Avoiding common Kubernetes controller mistakes

Nick Young, Isovalent at Cisco

KubeCon | CloudNativeCon Europe 2025

Who am I to talk about this?

- Started looking into CRDs in early 2017, when they were called Third-Party Resources or TPRs
- Was involved in building out Contour's `HTTPProxy` CRD, that replaced its `IngressRoute` CRD
- Have been involved in Gateway API since its inception in 2018 at Kubecon San Diego

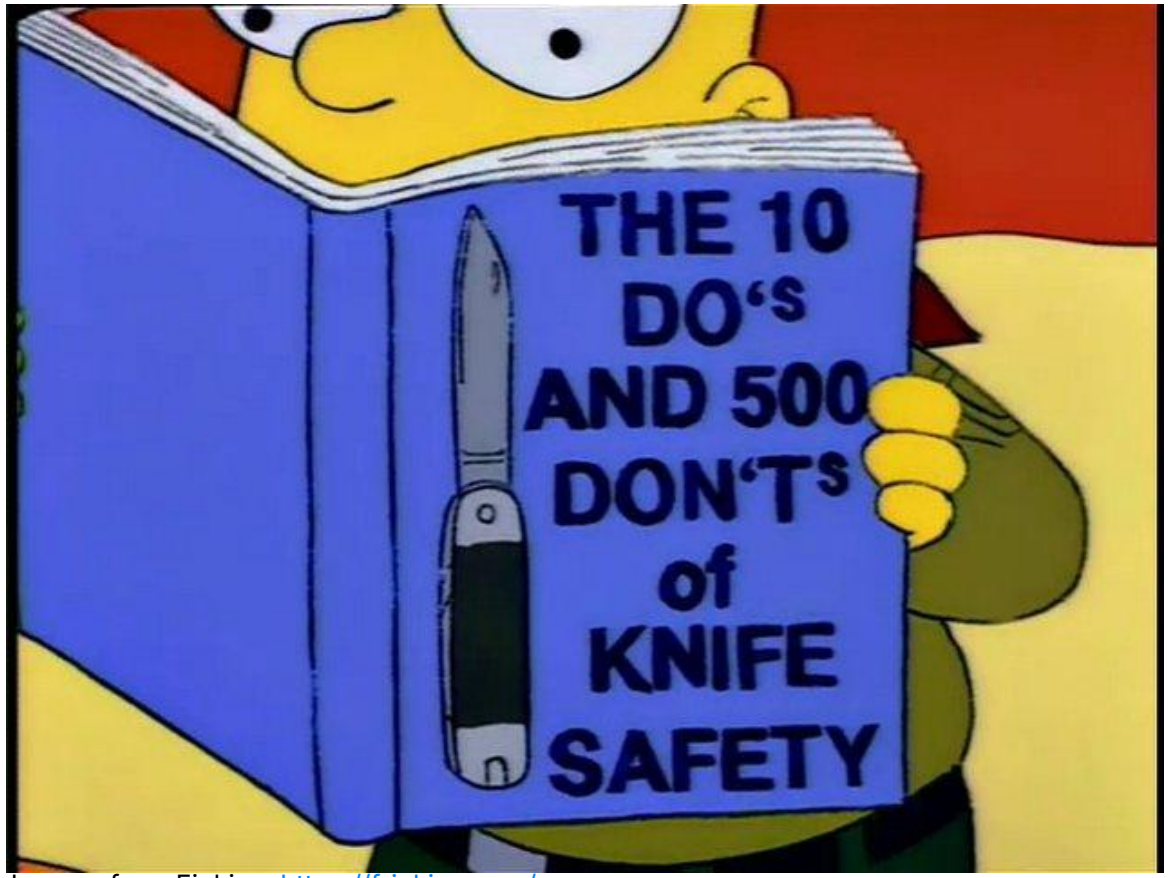
@youngnick.net

Today's Agenda

- Walk through some basic CRD controller Antipatterns, using "**C**ha**R**lie **D**on't" as our straw-man
- Give you some tips for each on what to do to avoid them
- Have a look at some of the frameworks available to help with writing controllers
- Give you some tips on what not to do with them

Why Charlie Don’t?

With thanks to The Simpsons



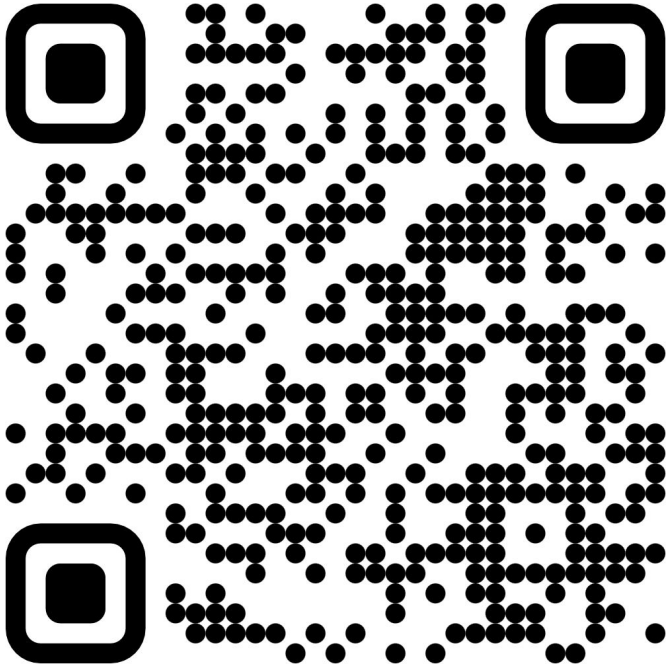
Images from Finkiac: https://frinkiac.com/
Simpsons Episode 1F06, Season 5, Episode 8, Boy Scoutz 'n the Hood

Meet Charlie Don’t

- Charlie works on a custom controller for Kubernetes at BigCo
- He has the worst luck and always manages to choose the wrong design option
- Poor Charlie!

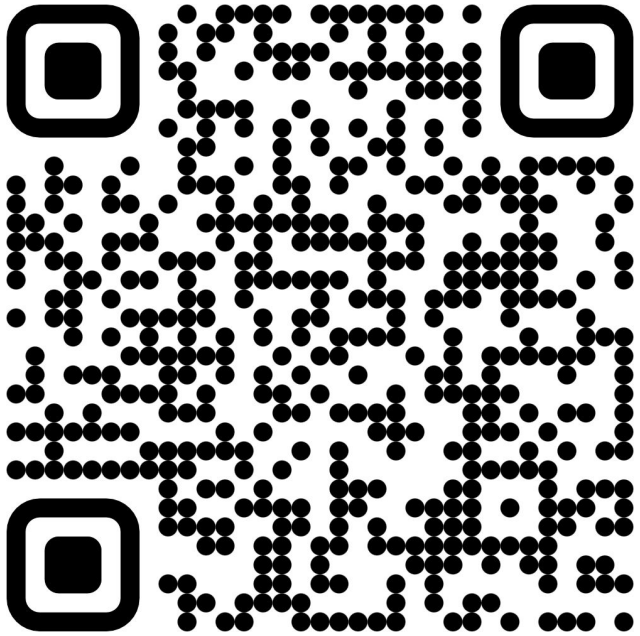
Previously on Charlie Don’t - CRD Design

- Read the API bibles (API Conventions and API changes docs)
- Think about how your users will use the CRD
- Use `status` and `status.Conditions`!
- Make as many fields as possible optional, with defaults if it makes sense
- Avoid maps except for labels and annotations. Use `listType=map` instead.
- Avoid `bool` types and bounded enums
- Avoid cross-namespace references, make them need a handshake if you do use them
- Don’t make breaking API changes without an API version bump



Previously on Charlie Don’t - API changes

- Versioning is important.
- If it can, the apiserver will convert between versions for you. Otherwise, you need a conversion webhook.
- Changes don’t need conversion if they are compatible.
- Some ways to make changes compatible:
  - Add new fields as `+optional`
  - When adding an enumerated string field, ensure that you document that values may be added
  - Don’t use `bool` fields! Use enumerated strings instead.
  - Make struct fields optional properly by making the field be `*struct`
  - Only loosen field validation between versions



Today we're talking about writing controllers

(mostly in Go)

Charlie Don’t uses a simple client

- Every time he performs a Get, List, or Watch, he calls the apiserver directly for that object.
- His controller pushes the whole object in every update to the apiserver, even if there are no changes.
- His clusters have lots of problems with apiserver load.

Use a cached client

- Ideally, you should use a client that maintains a cache of the last-seen state of each object.
- There are a lot of ways to do this!
- It also means that you now also have a cache management and invalidation problem, which is hard!
- More on this in a little bit

Limit the number of apiserver updates

- Every time you send a change to the apiserver, it has to:
  - Check the object and see what's changed
  - Write the changed object to storage
  - Notify everyone else of the change
- If you send a lot of no-op updates, then the apiserver will be busy checking all of them rather than doing useful work
- Check your own updates instead!

Limit the number of apiserver updates

- More specifically:
  - Use Patch instead of Update
    - Only send the changes rather than the whole object
    - Also helps avoid problems with racing updates (if the updates don't overlap)
  - This goes double for `status` updates!

Charlie Don’t makes a custom caching client using Informers

- He hand-rolls a caching client that uses the client-go Informer primitives
- He has to handle all the concurrency and ordering updates that eventual consistency brings
- He finds *lots* of edge cases that cause big problems

Using Informers directly

- This is harder than you would think to get right!
- Controller must maintain a per-Kind cache, and check the state on each event
  - Being sure that you have the current state is hard
  - Eventual consistency plus undefined ordering makes state tracking difficult
  - Multi-object reconciliation is even harder
    - If you see a HTTPRoute update, and you can't find the Service, does it not exist, or have you not seen it yet?

Use a framework instead!

- Frameworks are designed to manage consistency and ordering for you as much as they can
- You're looking for:
  - Something that handles watching resources and maintaining the current state for you
  - Something that allows you to do things when objects that have certain properties change
  - Something that helps with coalescing writes back to the apiserver (nice to have)

Some frameworks I know of

- krt
- StateDB
- controller-runtime

- Written by John Howard as an experimental refactoring of Istio.
  - In use in some Istio experiments and kgateway
- Perform operations on `Collections`, which can be sourced from Kubernetes objects via managed-for-you Informers, or on any object. When the relevant `Fetch` call from the collection would change, transformation functions are called to do things.
  - `Collection` s are the state tracking mechanism
  - The `Fetch` functions help you run transformers when the relevant objects change
- I think this is an interesting approach that's still under active development.

StateDB

- An in-memory, radix-tree database for Go; supports cross-table write transactions, and, most importantly, watch channels that close when that part of the tree is updated.
- This allows a StateDB table storing Kubernetes objects to be updated when an object is changed in a relevant way, allowing a reconciliation to be triggered.
- `Table` s are the relevant state storage mechanism
- `Table` s have configurable Update or Delete operations that are executed on change.
- This approach allows for treating your collection of objects like a database, and is also still relatively new. Some operations in Cilium have been migrated over to use StateDB already, with more on the way.

controller-runtime

- Included as part of the kubebuilder controller-tools set, part of upstream Kubernetes.
- Uses a `Reconcile` pattern with a key-based lookup, on top of a caching Kubernetes client that has the same API as the client-go version.
- The Reconcile pattern works like this:
  - implementors write a Controller that runs goroutines to watch relevant resources, and update the local caches as events come in
  - each Controller has one main resource that it `Reconcile` s, and can be configured to trigger the main `Reconcile` when dependent resources change as well
  - Both the main object and dependent object watches can use map functions to only choose *relevant* objects to trigger reconciliation

controller-runtime

- The availability of a main `Reconcile` function, but also watches on other objects that can trigger that `Reconcile` when they are updated makes this framework really good at writing systems of related CRDs.
- This means that:
  - The local cache stores the state for you,
  - The `Reconcile` mechanism allows functions to be executed on relevant changes

controller-runtime example


```
gatewayBuilder := ctrl.NewControllerManagedBy(mgr).
        // Watch its own resource
        For(&gatewayv1.Gateway{},
            builder.WithPredicates(predicate.NewPredicateFuncs(hasMatchingControllerFn))).
        // Watch GatewayClass resources, which are linked to Gateway
        Watches(&gatewayv1.GatewayClass{},
            r.enqueueRequestForOwningGatewayClass(),

builder.WithPredicates(predicate.NewPredicateFuncs(matchesControllerName(controllerName)))).
        // Watch related LB service for status
        Watches(&corev1.Service{},
            r.enqueueRequestForOwningResource(),
            builder.WithPredicates(predicate.NewPredicateFuncs(func(object client.Object) bool {
                _, found := object.GetLabels()[owningGatewayLabel]
                return found
            }))).
        // Watch HTTPRoute linked to Gateway
        Watches(&gatewayv1.HTTPRoute{}, r.enqueueRequestForOwningHTTPRoute(r.logger)).
        ... (more Watches snipped)
```

Charlie Don’t makes Reconcile mistakes

- He doesn’t realize that any predicates applied to `For` don’t also get applied to predicates applied to other `Watch` calls
- He uses the wrong resource for his main `Reconcile` loop

It’s me. I’m Charlie.

- Cilium’s Gateway API and GAMMA reconcilers (they’re separate) were not each checking `parentRef`s correctly, so they were triggering extra reconciliations that were then thrown away.
- The GAMMA reconciler was reconciling HTTPRoutes, not Service, which meant that if there are multiple HTTPRoutes referencing the same Service, then only the last one processed will take effect.

Takeaways

Takeaways

- It's *really* easy to make mistakes here, this is complex!
- Use a framework for coding controllers.
- Use Patch and/or check your changes to make sure they are relevant before sending to the apiserver.
- If you're using controller-runtime, remember that all your predicate functions need to check that the `Reconciled` resource is relevant as well as whatever other checks they run.

Charlie Don’t and I both say

Thanks for Listening!

KubeCon | CloudNativeCon
Europe 2025

Questions?

@youngnick.net