KubeCon | CloudNativeCon

North America 2024

Still Don’t Do What Charlie Don’t Does: Making CRD Changes safer

Nick Young, @youngnick, Isovalent at Cisco

@youngnick.net

Who am I to talk about this?

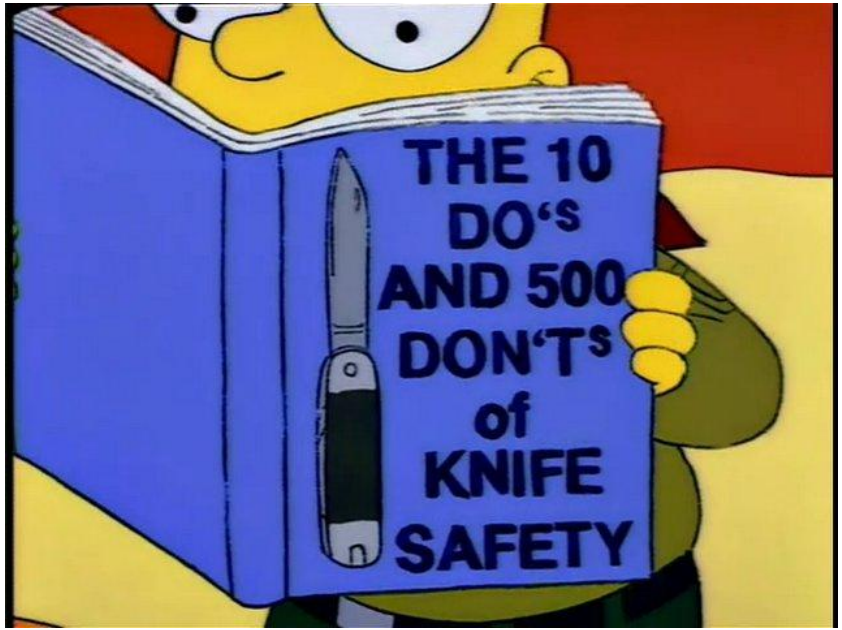
- Started looking into CRDs in early 2017, when they were called Third-Party Resources or TPRs
- Was involved in building out Contour's `HTTPProxy` resource, that replaced `IngressRoute`
- Have been involved in Gateway API since its inception in 2018 at Kubecon San Diego

Today’s agenda

- Explain some important context about how Kubernetes stores objects and versioning
- Walk through some CRD Change 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

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

- 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

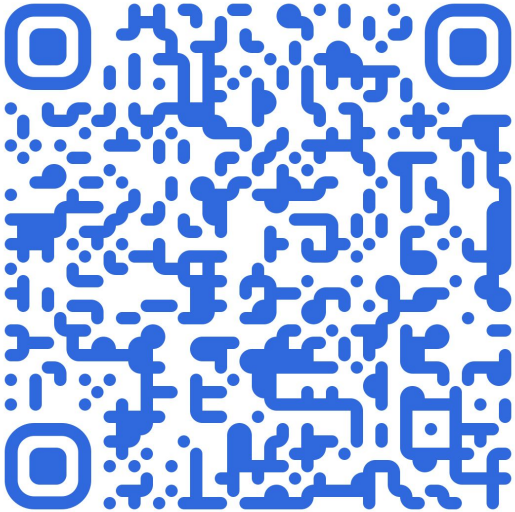
Why are API changes so important?

- No matter what you've built your API for, there are a few invariants:
  - You'll *always* need to make some changes to the API
  - There's no such thing as temporary
  - Except for "no". Saying "yes" to a feature affects you approximately forever, saying "no" to a feature is temporary.

Before we hear more about Charlie though…

The main reference!

The API Changes community documentation is the main reference for this talk.



Let’s talk about Kubernetes object versioning

- Every object has:
  - A **Group**: a domain-like string that identifies the a set of resources - usually an owning group or similar. For example: `gateway.networking.k8s.io` and `cilium.io`
  - A **Kind** and **Resource**: a Kind identifies an object Schema (like `Gateway` or `Node`), and a Resource identifies a resource in a unique way, used via HTTP (like `gateways` or `nodes`)
  - A **Version**: identifies a unique *version* of the resource, looks like `v1`, `v1beta1`, or `v2alpha1`.
- The **Group** and **Version** are often combined into the `apiVersion` field
  - Takes the form `<group>/<version>`, and looks like:
    - `gateway.networking.k8s.io/v1`
    - `cilium.io/v1alpha2`

Let’s talk about Kubernetes object versioning

- Versioning is Really Important.
- Kubernetes has three classes of *object* stability:
  - `alpha` - objects are experimental and subject to large, breaking changes between versions. No guarantees around conversion either.
  - `beta` - objects are experimental, but more stable than `alpha`. Generally will not have breaking changes between versions, and should include conversion if they do.
  - `stable` - indicated by `v1` or similar, this means that *no* breaking changes will *ever* be made to this API version. *Additive*, *safe* changes *may* be made. (More on what these are later)
- Making a breaking API change means incrementing the API version!
  - Going from `v1alpha1` to `v1alpha2`, or `v1beta2` to `v1beta3`, etc
  - Making a breaking change to a `v1` object requires a new `v2alpha1` or similar

Every object also has a **Storage** version

- The **storage** version sets the version of the object's schema that's to be persisted to storage (etcd).
- An object *may* have multiple versions available at any time.
- The apiserver converts between the versions when required.
- When are conversions required?
  - When there are incompatible changes between versions, *and*
  - When a consumer *reads* a version that is not the storage version, *or*
  - When something *writes* to an object that is stored as a version that's *not* the storage version - more on this in a minute
- In-tree resources have all required conversions included in the apiserver code
- CRDs must supply a webhook to handle changes between versions

Conversion Example

The Widget object on the right is version `v1alpha2`.

It has a `bar` field for storing the number of `bar`s that a widget provides.

```yaml
apiVersion: foomake.io/v1alpha2
Kind: Widget
metadata:
  name: testWidget
  namespace: widgethome
spec:
  bar: 20
```

Conversion Example

Now we decide that we want to support multiple numbers of `bars` with a single `Widget`, so we *pluralize* the `bar` field into `bars`.

Note that we updated the Version as well.

```yaml
apiVersion: foomake.io/v1alpha3
Kind: Widget
metadata:
  name: testWidget
  namespace: widgethome
spec:
  bars:
    - 10
    - 20
```

Conversion Example

This is not a *compatible* change, so a conversion would be required, which would take the value from the `bar` field, and insert it into the first entry of the `bars` field.

This change is not *compatible* because you can't take a `v1alpha3` version and turn it back into a `v1alpha2` version without possibly losing information.

```
apiVersion: foomake.io/v1alpha3
Kind: Widget
metadata:
  name: testWidget
  namespace: widgethome
spec:
  bars:
    - 10
    - 20
```

Conversion Example

But, you could write conversion code to safely convert all `v1alpha2` objects to `v1alpha3` objects.

```yaml
apiVersion: foomake.io/v1alpha3
Kind: Widget
metadata:
  name: testWidget
  namespace: widgethome
spec:
  bars:
    - 10
    - 20
```

You can skip conversions though!

- We can skip having to convert values by making sure changes are *backwards compatible*.
- What makes a backwards compatible change?
- Let’s check in with Charlie!

Charlie Don’t makes breaking API changes in the same `version`

- Charlie doesn’t understand the rules for API changes.
- Charlie makes a change that’s not backwards compatible and doesn’t increment his API version
- Users of his CRDs are broken when they upgrade!
- Poor Charlie *and* his users!

So what *can* we do?

It’s complicated!

The Simplified Backwards Compatibility Rule

If you can take a new object, with only *required* fields set, change the version to the old version, and apply it, and *nothing behaves differently*, then the changes between the versions are **backwards compatible**.

Let’s head back to the real world now though

By that previous definition, almost any change is not backwards compatible!

However, we can make some changes safely, *if and only if*:

- They are *additive*
- We use a feature flag mechanism to control processing of those fields

Additive Changes

Additive changes are a special class of changes that may break in *limited*, *understandable* ways, so they *can* be okay to make within the same API version.

Let’s run through some examples

Charlie Don’t adds a new field that is `+required`

- Adding a new field that’s *required* is always a breaking change, because it means that the new version can’t be serialized to the old version without changing behavior.

How do we add new fields?

The best way to do this is to make the new field *optional*, and have the default value for the optional field mean “the same behavior as the old version”.

For default values, you can either use the *zero value*, or you can set a default value in the CRD definition.

In this case, if no value is supplied, then the default value will be used instead.

Adding new fields - scalar fields

- Scalar fields are individual values, like `string` or `int`.
- If you need to tell the difference between “set to a value”, “set to the zero value”, and “unset”, use a pointer!
- Examples
  - Adding a new, optional `newBehavior` string field that defaults to the empty string, which means “don’t do the new behavior”.
  - Adding a new, optional `awesomenessLevel` string field that defaults to `none` (which is how you’re describing the old behavior).
  - Adding a new `timeoutSeconds` field where `0` means “unlimited”, unset means “use the default”, and a value means to use that specific value.

Charlie Don’t uses Enum fields without declaring them open for changes

- An enumerated field is a `string` field that has a set of permitted values - usually supplied in Go types as aliased constants.
- Charlie Don’t doesn’t make it clear in the field definition that he might add fields later.
- Implementations don’t know they need to handle unknown values.

Enum strings

The answer here is straightforward:

- Ensure that your enumerated strings are *always* documented as being open for changes
- You must also document what happens when there's an invalid value!

Charlie Don’t adds values to `bool` fields

- A `bool` field can only ever be `true` or `false`.
- Adding other values to a `bool` field requires changing it to a `string`
- Type changes for fields are breaking changes.
- Don’t use `bool` fields!

What to use instead of `bool` fields?

Use enumerated string fields instead (but make sure you declare them open for changes!)

Don’t do:

```
enableAwesomeness: true|false
```

Do:

```
awesomenessLevel: extreme|some|none
```

Charlie Don’t adds `struct` fields that aren’t pointers

- For struct fields to be optional, they *must* be pointers.
- If you add a struct field that’s not optional, that’s a breaking change.
- A good way to have easily expandable struct fields is to use a *union*.

Union fields

The Widget object on the right is version `v1alpha2`.

It has a `union` setup, where you choose a `type`, and then have a separate config struct per type.

Importantly, we've marked the `type` field as "can be expanded", since it's an enum field.

But, in this case we're only using a `round` type.

```
apiVersion: foomake.io/v1alpha2
Kind: Widget
metadata:
 name: testWidget
 namespace: widgethome
spec:
 type: round
 roundConfig:
   radius: 10
```

Union fields

Here, we've added a new value to the `type` field, and an accompanying config struct.

This *can* be an additive change IF:

- the `type` enum allows expansion
- the `squareConfig` struct is added as a pointer, so that it defaults to unset
- The contract for the Widget object includes that invalid values for `type` are processed but recorded as an error in the `status`.

```
apiVersion: foomake.io/v1alpha2
Kind: Widget
metadata:
 name: testWidget
 namespace: widgethome
spec:
 type: square
 roundConfig:
   radius: 10
 squareConfig:
   sideLength: 10
```

Charlie Don’t makes validation rules more strict between versions

- This is backwards incompatible because values that used to be valid are not valid any longer.
- *Loosening* validation can be okay, if you’re careful.
- Remember that once you loosen, you can never go back!

Summing up

What did we learn from Charlie this time?

- 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

Charlie Don’t and I both say

Thanks for Listening!