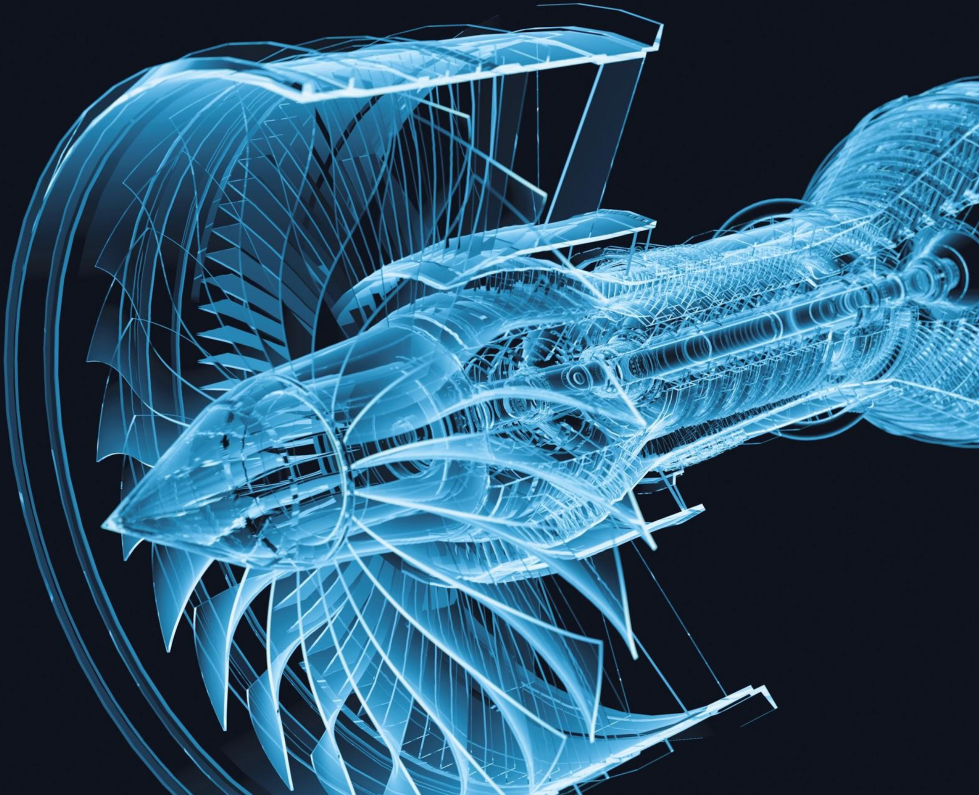
JETSTACK

Presented by James Munnelly

@JamesMunnelly

Extending Kubernetes: what the docs don’t tell you



jetstack.io

Who are Jetstack?

We are a UK-based company that help enterprises in their path to modern cloud-native infrastructure. We also develop tooling & integrations for Kubernetes to improve the user experience for customers and end-users alike.

Who am I?

I've worked as a software engineer (of sorts!) for the best part of 10 years. I started using Kubernetes in production about 2½ yrs ago (0.19.2) & have been immersed ever since! I joined Jetstack about a year ago as a Solutions Engineer.

@munnerz

@JamesMunnelly

jetstack.io

So first, what is & why do this?

Why do this?

Kubernetes has:

- Declarative API types
- Versioning of types
- Tooling for building reconciliation loops (desired vs actual)
- Pre-made CLI (kubectl)
- A well-tested pattern for designing extensions

All of this is hard to design & manage

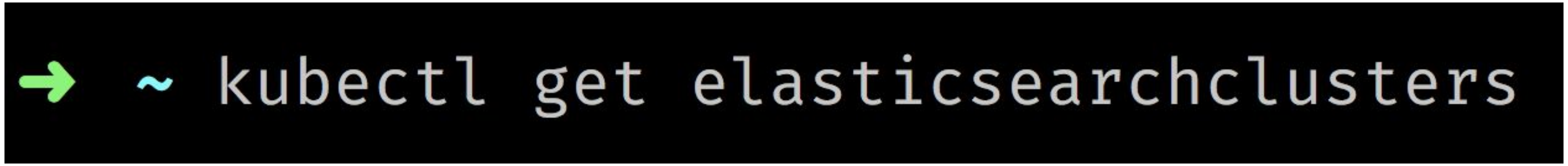
Why do this?

We can ‘extend’ this by adding our own types, some examples:

- cert-manager (github.com/jetstack/cert-manager)
- navigator (github.com/jetstack/navigator)
- etcd-operator (github.com/coreos/etcd-operator)
- service-catalog (github.com/kubernetes-incubator/service-catalog)
- metrics (github.com/kubernetes-incubator/custom-metrics-apiserver)

Why do this?

This lets us do things like…



```
➜  ~ kubectl get elasticsearchclusters
```

Why do this?

Follow the ‘control loop’ pattern found all over Kubernetes

- Observe change to desired or actual state
- Take action to converge ‘desired’ and ‘actual’
- Repeat

Why do this?

Integrate with Kubernetes native auth{n,z}

- RBAC, ABAC, etc. all work out the box
- Don't need to invent your own authorization
- Seamlessly works together

It just feels natural

How is this done?

How is this done?

CustomResourceDefinitions

- Quick & easy. No extra apiserver code.
- Great for simple extensions
- No versioning, admission control or defaulting

https://kubernetes.io/docs/concepts/api-extension/custom-resources

How is this done?

Custom API servers (via kube-aggregator)

- Full power and flexibility of Kubernetes
- Similar to how many existing APIs are created
- Versioning, admission control, validation, defaulting
- Requires etcd to store data

https://kubernetes.io/docs/concepts/api-extension/custom-resources

Tip #1: API aggregation provides far greater flexibility

Both require some supporting code

Generators

- client-go contains a ***clientset***, ***informers*** & ***listers*** for core types
- Useful for building extensions to core Kubernetes types

We can generate our own ***clientset***, ***informers*** & ***listers*** for our types

Generators

- client-gen
- conversion-gen
- deepcopy-gen
- defaulter-gen
- go-to-protobuf
- informer-gen
- lister-gen
- openapi-gen
- codec-gen

Generators

- Code generators assist in development of Kubernetes APIs
- Annotate types.go with some metadata and off you go

- github.com/kubernetes/gengo
- github.com/kubernetes/code-generator

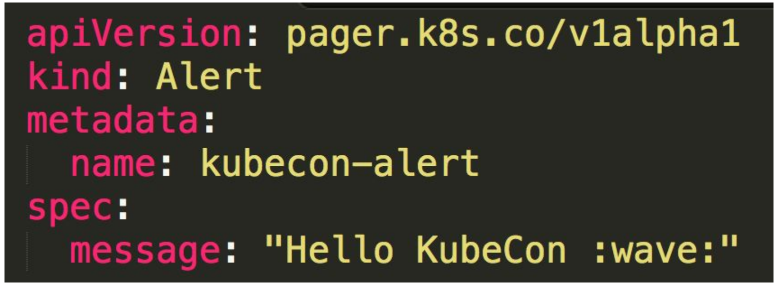
Tip #2: Generate supporting code, don’t write it yourself

k8s-api-pager-demo

Our sample app for today

k8s-api-pager-demo

- A small sample project that implements a Pager for Kubernetes
- Consists of an apiserver and a controller
- Sends 'alerts' via Pushbullet.com
- github.com/munnerz/k8s-api-pager-demo



```yaml
apiVersion: pager.k8s.co/v1alpha1
kind: Alert
metadata:
  name: kubecon-alert
spec:
  message: "Hello KubeCon :wave:"
```

jetstack.io

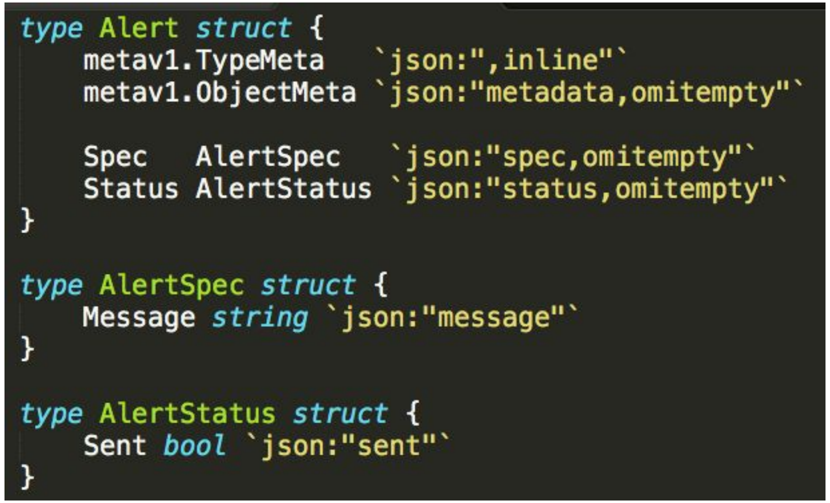
Quick demo

Generators

- client-gen
- conversion-gen
- deepcopy-gen
- defaulter-gen
- go-to-protobuf
- informer-gen
- lister-gen
- openapi-gen
- codec-gen

k8s-api-pager-demo

We define our types in types.go:



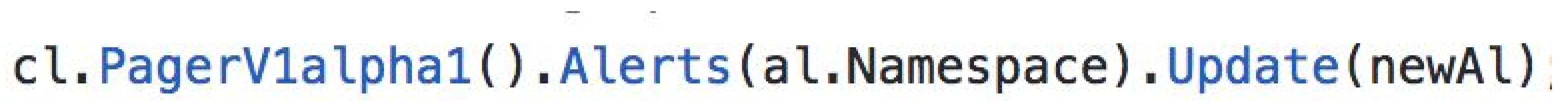
```go
type Alert struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec   AlertSpec   `json:"spec,omitempty"`
    Status AlertStatus `json:"status,omitempty"`
}

type AlertSpec struct {
    Message string `json:"message"`
}

type AlertStatus struct {
    Sent bool `json:"sent"`
}
```

… then run the generators!

Tip #3: Follow the types.go rules. Generators are picky.

client-gen

Generate typed Kubernetes API clients for your types

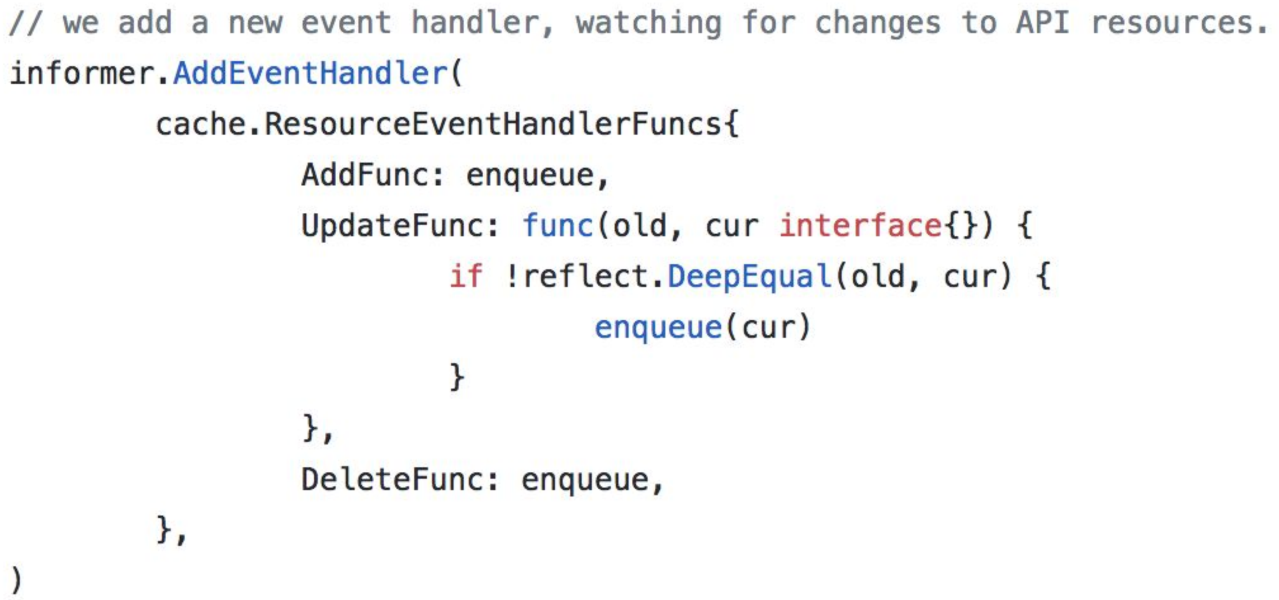


```
cl.PagerV1alpha1().Alerts(al.Namespace).Update(newAl)
```

jetstack.io

informer-gen

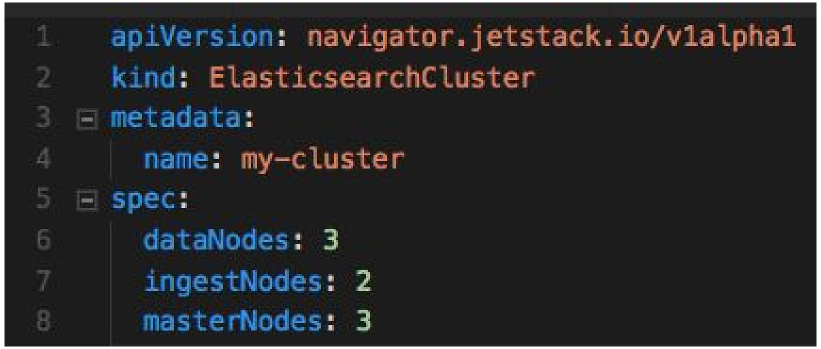
Generate informers that can be used to watch for updates to your types



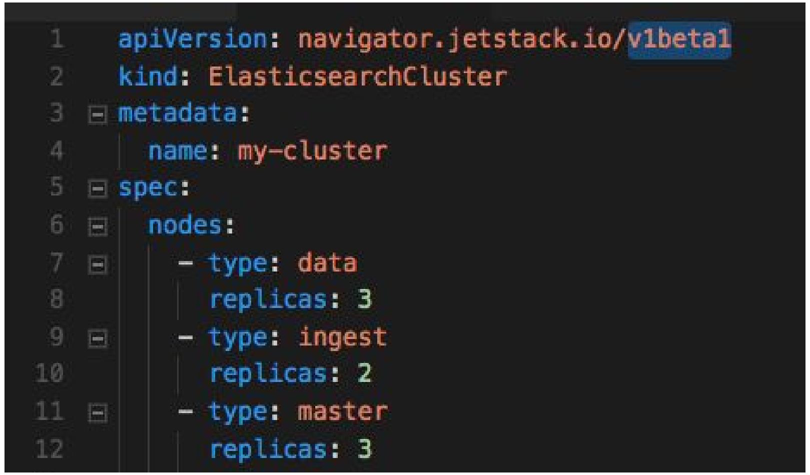
```go
// we add a new event handler, watching for changes to API resources.
informer.AddEventHandler(
        cache.ResourceEventHandlerFuncs{
                AddFunc: enqueue,
                UpdateFunc: func(old, cur interface{}) {
                        if !reflect.DeepEqual(old, cur) {
                                enqueue(cur)
                        }
                },
                DeleteFunc: enqueue,
        },
)
```

conversion-gen

Allow seamless upgrades between API versions



```yaml
apiVersion: navigator.jetstack.io/v1alpha1
kind: ElasticsearchCluster
metadata:
  name: my-cluster
spec:
  dataNodes: 3
  ingestNodes: 2
  masterNodes: 3
```



```yaml
apiVersion: navigator.jetstack.io/v1beta1
kind: ElasticsearchCluster
metadata:
  name: my-cluster
spec:
  nodes:
    - type: data
      replicas: 3
    - type: ingest
      replicas: 2
    - type: master
      replicas: 3
```

Let’s take a look at the controller

Tip #4: *never* modify resources in the cache

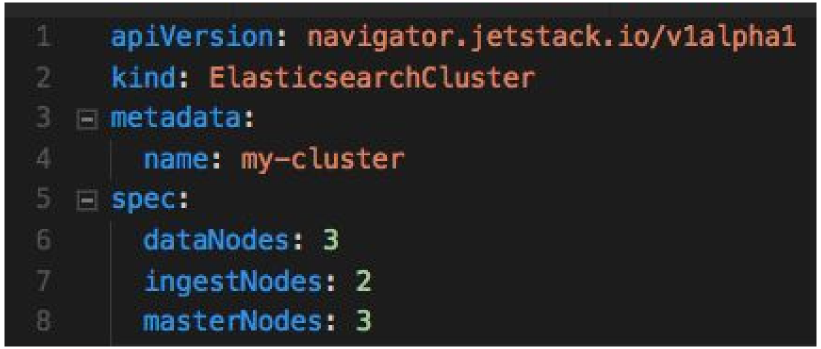
Creating v1beta1

We want to make some breaking changes to our API:

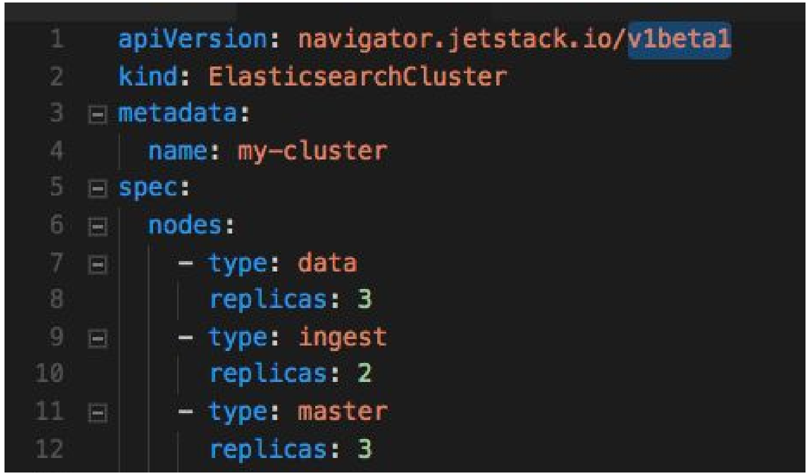
- ‘message’ -> ‘content’
- Add title field

conversion-gen

Allow seamless upgrades between API versions



```yaml
apiVersion: navigator.jetstack.io/v1alpha1
kind: ElasticsearchCluster
metadata:
  name: my-cluster
spec:
  dataNodes: 3
  ingestNodes: 2
  masterNodes: 3
```



```yaml
apiVersion: navigator.jetstack.io/v1beta1
kind: ElasticsearchCluster
metadata:
  name: my-cluster
spec:
  nodes:
    - type: data
      replicas: 3
    - type: ingest
      replicas: 2
    - type: master
      replicas: 3
```

Creating a custom API server

- Define our new API version (v1beta1)
- Implement custom API server using k8s.io/apiserver
- Deploy etcd and pager-apiserver
- Configure an APIService resource
- Run the controller
- … Profit

‘external’ api versions

v1alpha1, v1beta1, v1, etc…

‘internal’ api version

Used to convert between external versions

Let’s take a look

Tip #5: define conversions between external versions

Wrapping up

- Tip #1: API aggregation provides far greater flexibility
- Tip #2: Generate supporting code, don’t write it yourself
- Tip #3: Follow the types.go rules. Generators are picky.
- Tip #4: never modify resources in the cache
- Tip #5: define conversions between external versions

Useful resources

- github.com/jetstack/navigator
- github.com/jetstack/cert-manager
- github.com/kubernetes/sample-controller
- github.com/kubernetes/sample-apiserver
- github.com/munnerz/k8s-api-pager-demo
- github.com/kubernetes-incubator/apiserver-builder/
- github.com/kubernetes/apiserver
- github.com/kubernetes/code-generator
- blog.openshift.com/kubernetes-deep-dive-code-generation-customresources/ by @sttts

JETSTACK

Thanks for watching!

@JamesMunnelly

@JetstackHQ

jetstack.io