Kubernetes as a Database

Chris Kim - Field Engineer
@RancherLabs
Who am I?

Chris Kim
• Kubernetes Aficionado
• Rancher Field Engineer
• Responsible for
  • Submariner
  • HobbyFarm
Disclaimer: I am not a database expert

- Which is probably why I keep trying to use Kubernetes as a Database
- Most of my applications that use K8s are K8s-native applications, or run on K8s.
- When running on-prem, I already have to manage etcd, so why manage another datastore like MySQL?
- It’s even easier when running in managed Kubernetes (GKE/EKS/AKS) because I don’t manage etcd
Popular Database Options

- MySQL/MariaDB
- Oracle DB
- MSSQL
- PostgreSQL
- MongoDB
- CockroachDB
- Kubernetes
Why use Kubernetes as a Database?

• It’s easy (relatively speaking)
• It was a good way to really understand the Kubernetes API
• Rancher (and most of our other open source projects) do this, so it was along with the norm
• High availability of the datastore is handled by the libraries
Why not just use a “normal” database?

• I’m lazy
• All of my apps run in (or are built around) Kubernetes
• I don’t have to figure out where to run my database
• Nothing I have is particularly dependent on database performance
Running Databases in Kubernetes

• Use the MariaDB Helm Chart

• But first, I have to figure out storage
  • Portworx
  • Longhorn
  • StorageOS
  • OpenEBS
  • NFS
  • Local Disk

• How do I make my database highly available?
The Kubernetes API has some cool features

- Custom Resource Definitions (CRD)
- Labels
- Built-in High Availability
- Namespacing
Custom Resource Definitions

• OKD defines a Custom Resource Definition as: “an object that extends the Kubernetes API or allows you to introduce your own API into a project or a cluster.”

• In short, a CRD is just a declaration or notification to the Kubernetes API to let Kubernetes dynamically register a new resource.
Cluster scope vs. Namespace scope

• CRD’s can be scoped at either the Cluster or Namespace level.
• Choose wisely, it can be a pain to convert from Cluster to Namespace scoping or vice-versa down the line.
• Generally, you should use namespace scoping and figure out a way to pass a configurable namespace to your controllers. This allows you to run multiple instances of your app in the same cluster.
CRD Example

```yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: users.hobbyfarm.io
spec:
  group: hobbyfarm.io
  version: v1
  names:
    kind: User
    plural: users
  scope: Cluster
```
Kubernetes Labels

• Kubernetes Labels are key/value pairs that are attached to objects
• You use them to tag and filter your objects
• They allow for efficient queries and watches, and can be used with the CLI
Owner References and Finalizers

• Owner References allow easy garbage collection and relationship definition for your objects
  • In HobbyFarm, a VirtualMachineSet is an “owner” object that has dependents which are VirtualMachine objects. The VirtualMachine object has an OwnerReference that points towards a VirtualMachineSet
  • When I delete the VirtualMachineSet, my VirtualMachine objects are also deleted

• Finalizers block the deletion of your object, and can be used by your controllers to delete other, out of band dependents
  • In HobbyFarm, a VirtualMachine object has a finalizer which is not removed until the actual VirtualMachine represented by the object is deleted
Conventional Database Schemas

**Conventional SQL**

- `SELECT * FROM users`
- `SELECT * FROM users WHERE firstname=chris`

**Kubernetes**

- `kubectl get users -o json`
- `kubectl get users --l firstname=chris --o json`
Frameworks to help make life easier

• Norman - https://github.com/rancher/norman
  • An API framework for Building Rancher Style APIs backed by K8s CustomResources and their controllers.

• Wrangler - https://github.com/rancher/wrangler
  • Framework for wrapping clients, informers, listers into a simple usable controller pattern that promotes some good practices.


• I don’t use these, but instead just generate my code using code-generator
Kubernetes Tool Repositories

• Code Generator - https://github.com/kubernetes/code-generator
  • Used to generate client based on your CRD definition

• API Machinery - https://github.com/kubernetes/apimachinery
  • Contains utilities/definitions for core Kubernetes components

• client-go - https://github.com/kubernetes/client-go
  • client-go is the Golang client
client-go Caches and Indexers

• Kubernetes client-go generated code has a built in cache
• You can also set up indexers that will index based on a custom function definition, which effectively allows you to index by value
Stale cache

• When using the Kubernetes cache-backed listers it is important to realize you may receive stale data

• This is especially important when dealing with “multi-threaded” applications
  • You don’t want to perform an operation twice, or worse, race
Security

• Restricting access to your CRD’s is surprisingly easy
• You use Kubernetes RBAC for this
• ClusterRole + ClusterRoleBinding or Role + RoleBinding
So how do I actually use this?

Overall, a pretty simple set of steps

1. Create your types.go
2. Run the code-generator
3. Import the generated code and interact with your CRD’s
4. Look at the error message you got while trying to compile or run and google
types.go

• The overall format of types.go is pretty self-explanatory
code-generator
Importing generated code

```go
import {
    "encoding/json"
    "fmt"
    "github.com/golang/glog"
    "github.com/gorilla/mux"
    hfv1 "github.com/hobbyfarm/gargantua/pkg/apis/hobbyfarm.io/v1"
    "github.com/hobbyfarm/gargantua/pkg/authclient"
    hfClientset "github.com/hobbyfarm/gargantua/pkg/client/clientset/versioned"
    "github.com/hobbyfarm/gargantua/pkg/util"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "net/http"
}
```
Controllers

• Most of the applications that don’t simply serve/store data based on CRD’s utilize “controllers”

• You set up your controller using the SharedInformerFactory

• The SharedInformerFactory has built in constructs to allow you to process based on
Example Controller Setup

```go
stopCh := signals.SetupSignalHandler()
flag.Parse()
glog.V(level: 2).Infof(format: "Starting Gargantua")
r := mux.NewRouter()

cfg, err := rest.InClusterConfig()
if err != nil {
    cfg, err = clientcmd.BuildConfigFromFlags(localMasterUrl, localKubeconfig)
    if err != nil {
        glog.Fatal(f "Error building kubeconfig: %s", err.Error())
    }
}

hfClient, err := hfClientset.NewForConfig(cfg)
if err != nil {
    glog.Fatal(err)
}

kubeClient, err := kubernetes.NewForConfig(cfg)
if err != nil {
    glog.Fatal(f "Error building kubernetes clientset: %s", err.Error())
}

hfInformerFactory := hfInformers.NewSharedInformerFactory(hfClient, time.Second*30)

scenarioSessionController, err := scenariosession.NewScenarioSessionController(hfClient, hfInformerFactory)
if err != nil {
    glog.Fatal(err)
}
```
Running this Controller

```go
func() {
    defer wg.Done()
    scenarioSessionController.Run(stopCh)
}()
```
Setting up the Informer Event Handler

```go
ssInformer := hfInformerFactory.Hobbyfarm().V1().ScenarioSessions().Informer()
ssInformer.AddEventHandlerWithResyncPeriod(cache.ResourceEventControllerFuncs{
    AddFunc: ssController.enqueueSS,
    UpdateFunc: func(old, new interface{}) {
        ssController.enqueueSS(new)
    },
    DeleteFunc: ssController.enqueueSS,
}, time.Second*30)
```
Adding to the Workqueue

```go
func (s *ScenarioSessionController) enqueueSS(obj interface{}) {
    var key string
    var err error
    if key, err = cache.MetaNamespaceKeyFunc(obj); err != nil {
        //utilruntime.HandleError(err)
        return
    }
    glog.V(level: 8).Infof(format: "Enqueueing ss %s", key)
    s.ssWorkqueue.AddRateLimited(key)
}
```
How the Worker Runs

```go
func (s *ScenarioSessionController) Run(stopCh <-chan struct{}) error {
    defer s.ssWorkqueue.Shutdown()

    log.V(4).Infof("Starting Scenario Session controller")
    log.Info(args: "Waiting for informer caches to sync")
    if ok := cache.WaitForCacheSync(stopCh, s.vmSynced, s.vmcSynced, s.ssSynced); !ok {
        return fmt.Errorf("failed to wait for vm, vmc, and ss caches to sync")
    }
    log.Info(args: "Starting ss controller workers")
    go wait.Until(s.runSSWorker, time.Second, stopCh)
    log.Info(args: "Started ss controller workers")
    //if ok := cache.WaitForCacheSync(stopCh, )
    <-stopCh
    return nil
}

func (s *ScenarioSessionController) runSSWorker() {
    log.V(6).Infof("Starting scenario session worker")
    for s.processNextScenarioSession() {
    }
}
```
Processing Scenario Sessions

```go
func (s *ScenarioSessionController) processNextScenarioSession() bool {
    obj, shutdown := s.ssWorkqueue.Get()

    if shutdown {
        return false
    }
```
Important things to keep in mind when working with the Informers

• Your controller may not always immediately operate on a changed object

• Cache invalidation

• The workqueue will only dole out an object exclusively to a single thread at a time – this means you can run many threads and not worry about contention or racing
  • This only works if you don’t have external dependencies
Thank you! Questions?