KubeCon | CloudNativeCon
Europe 2018

Kubernetes Runs Anywhere, but Does your Data?

Jared Watts, Maintainer for *Rook*, Founding Engineer at *Upbound*

Kubernetes Runs Everywhere

- Kubernetes is the defacto container orchestrator
- Supported by everyone
  - Cloud providers (Google, Amazon, Microsoft, etc.)
  - Bare-metal, on-premise, hybrid environments
  - Local machine and developer laptops
- Most importantly, it allows your apps to run everywhere too

The Power of Portability

- A portable solution can run in diverse environments
- Gives us the “power of choice”
- Able to take advantage of the best environment for the job
    - Cost
    - Service quality
    - Features
    - Resiliency (multi-cloud)
    - Compliance

How does Kubernetes do it?

- Common **abstractions** and primitives for application deployment concepts
- Pods, Deployments, Services, etc.
- “Write once, run many”
- Allows the same exact application deployment to run in all the environments Kubernetes runs in
- Developers can focus on building their application, not the environment details

Storage Abstractions

- Real applications are stateful - they need to persist data somewhere
- There are useful abstractions defined for storage too
  - Persistent Volumes (PVs)
  - Persistent Volume Claims (PVCs)
  - Storage Classes
  - Plugins (CSI)
- Helps support portability of stateful applications

Persistent Volumes and Claims

- Persistent storage for applications is normalized on the concept of a **volume**
- Independent of the backing storage provider or solution
- This abstraction is a powerful concept that allows applications to not know details about where the storage is coming from

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv001
spec:
  capacity:
    storage: 250Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  storageClassName: fast
```

Storage Classes

- “Blueprints” that abstract away details of how to fulfill storage requests
    - Admin fills in details so apps don’t have to
- Provider, quality-of-service levels, backup policies, etc.
- Enables dynamic (automatic) provisioning of storage for applications on-demand
    - Enormous leap beyond static provisioning

```
kind: StorageClass
apiVersion: storage/v1
metadata:
  name: durable
provisioner: azure-disk
parameters:
  location: eastus
```

Volume Plugins

- Allows storage solutions that are external to the cluster to be integrated into Kubernetes
- Managing and deploying the storage solution is an exercise left to the reader (or admin)
- Often relies on provider specific managed services
  - Google Persistent Disk
  - Amazon EBS
  - Azure Disk

Where Storage Falls Short

- Focus is on consumption side, not provider side
- External storage solutions have to be accessible
- This dependency causes deployment and management burden

Where Storage Falls Short

- Not all data behaves like a volume
- Data intensive workloads have needs for higher level storage types
  - Databases, object stores, key-value stores, caches, message queues, etc.
- No portable abstractions exist for these types of data/storage services
- Applications often have to depend on proprietary managed services external to the cluster and specific to the environment
  - Vendor lock-in

A Portable Storage Solution

- We need something more in order to make stateful apps as portable and environment agnostic as Kubernetes itself
- What about storage that runs *inside* the Kubernetes cluster?
  - take advantage of its powerful abstractions
- What about a broader set of storage abstractions?
  - utilize rich managed services in a portable way

A Portable Stateful Application

- Wordpress is stateful (needs a database)
- All cloud providers have a managed database service
- Wordpress and its data needs could all be described in one YAML that works across environments
- At deployment time, the right service gets set up

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
 name: wordpress
 labels:
   app: wordpress
...
---
apiVersion: databases/v1alpha1
kind: MySql
metadata:
 name: mydb
spec:
 size: 200Gi
 high-availability: auto
 backup-schedule: default
```

Extending Kubernetes - CRDs

- Teaches Kubernetes about new first-class objects
- Custom Resource Definition (CRDs) are arbitrary types that extend the Kubernetes API
    - look just like any other built-in Kubernetes object (e.g. Pod)
- Persisted in etcd

Extending Kubernetes - CRDs

- Allows for native management experience with `kubectl`
    - `kubectl create -f my-storage.yaml`
- Doesn’t have any functionality or logic by itself
- Captures user’s desired state of the system
- We can define powerful new **abstractions** with CRDs

CRD Declaration

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
 name: database.rook.io
spec:
 group: rook.io
 version: v1alpha1
 scope: Namespaced
 names:
   kind: Database
   listKind: DatabaseList
   plural: databases
   singular: database
```

Extending Kubernetes - Operators

- Special type of Controller we’re already familiar with
- Codifies domain expertise to deploy and manage an application
  - Automates actions a human would normally do
- Allows users to consume a software system without any operational experience

Operator’s Control Loop

- Control loop that watches for events (add/update/delete) on your CRDs and changes to cluster
- Reconciles the user’s desired state with the cluster’s actual state
  - Observe - discover current actual state of cluster
  - Analyze - determine differences from desired state
  - Act - perform operations to drive actual towards desired
- Continuously drives towards the desired state

Let the Operator Manage it

- After a software system is deployed, it needs to be managed
  - Day 2 operations
- Ensure critical health and functionality
- Handling failures of key components, self-healing
- Scaling the system in accordance with load
  - Shuffling data without loss or downtime
- Backups and restorations
- Rolling upgrades and rollbacks
  - Migrations across versions with breaking changes
- Regularly recurring maintenance tasks

Kubernetes API and Clientsets

- Kubernetes API is the way to get anything done in the cluster
- Clientset: A collection of clients that can each talk to an API group
  - Often used from a controller's Go code
- Hugely important in both discovering system state and performing operations to influence that state

```
pods, err := clientset.CoreV1().Pods(“default”).List(opts)

sc, err := clientset.StorageV1().StorageClasses().Create(sc)
```

Informers

- We want to be know when an event happens
- We could poll the API server, but that’s expensive
- **Informer**: Intelligently and efficiently watch for object events on the Kubernetes API server
- Maintains an local cache of objects so we don’t always have to query the API server
- **SharedInformers** should be used when you have multiple controllers

Informer Usage

```
_, podInformer := cache.NewInformer(
  source,
  &v1.Pod{},
  time.Second * 30,
  cache.ResourceEventHandlerFuncs{
    AddFunc: handlePodAdd,
    UpdateFunc: handlePodUpdate,
    DeleteFunc: handlePodDelete,
  })
go podInformer.Run()
```

Event Triggers

- Level driven, not edge driven
- Can’t assume your controller is always running, it may miss the event “transition”
- Perform reconciliation of desired and actual state *any* time a difference is observed

So...Running Your Data Anywhere

- By extending Kubernetes, we can define **new** useful portable abstractions
- Users and apps can describe their need for storage in a general way
- **All** of an app’s requirements could be described in a single manifest now, including storage
- Operators can make that desire for data “just happen” in any environment
  - Bring the storage INTO the cluster
  - Provision cloud provider managed services

Demo

Let’s make some storage happen!

Questions

Questions?

@jbw976 (github/twitter)

@rook_io

https://rook.io/

https://upbound.io/

(we’re hiring!)