Adding a New Storage Provider to Rook

Jared Watts
Rook Senior Maintainer
Upbound Founding Engineer

https://rook.io/
https://github.com/rook/rook

What is Rook?

- Cloud-Native Storage Orchestrator
- Extends Kubernetes with custom types and controllers
- Automates deployment, bootstrapping, configuration, provisioning, scaling, upgrading, migration, disaster recovery, monitoring, and resource management
- Framework for many storage providers and solutions
- Open Source (Apache 2.0)
- Hosted by the Cloud-Native Computing Foundation (CNCF)

Storage Challenges

- Reliance on external storage
  - Requires these services to be accessible
  - Deployment burden
- Reliance on cloud provider managed services
  - Vendor lock-in
- Day 2 operations - who is managing the storage?

Possible Solutions

- Deploy storage systems INTO the cluster
- Harness the power of Kubernetes
- Automated management by smart software
- Portable abstractions for all our storage needs

Power of Portability

- Power of **choice** - cost, features, availability, compliance, etc.
- Take our **data** wherever Kubernetes goes
- **Pod** and **Volume** abstractions enables portability
  - What about databases, buckets, message queues, data pipelines, etc.?
- **Crossplane** - open source multicloud control plane
  - https://crossplane.io/

Custom Resource Definitions (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 object (e.g. Pod)
  - Enabled native `kubectl` experience
- A means for user to describe their desired state

Rook Operators

- Implements the **Operator Pattern** for storage solutions
- User defines *desired state* for the storage cluster
- The Operator runs reconciliation loops
  - Observe - Watches for changes in desired state and cluster
  - Analyze - Determine differences between desired and actual
  - Act - Applies changes to the cluster to drive it towards desired

Operator Frameworks

Current: Register CRDs, watch events and invoke handler functions

- Rook operator-kit: https://github.com/rook/operator-kit

Future: Auto-generate APIs, CRDs, controllers, reconciliation, boilerplate code, unit tests, deployment, etc.

- Operator SDK: https://github.com/operator-framework/operator-sdk
- Kubebuilder: https://github.com/kubernetes-sigs/kubebuilder

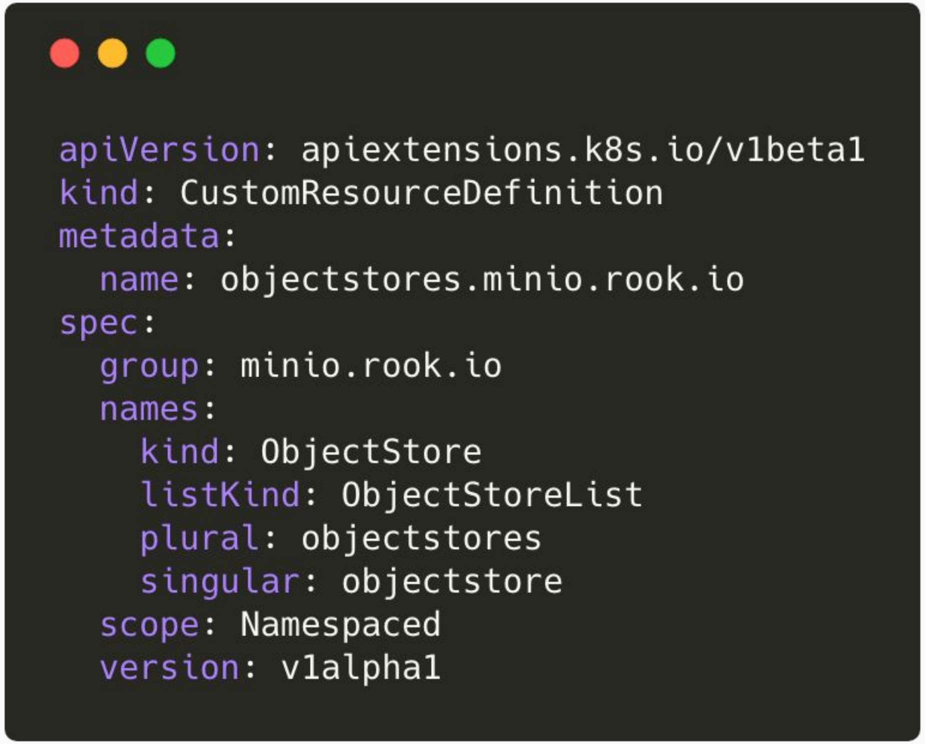
Rook Framework for Storage Solutions

- Rook is more than just a collection of Operators and CRDs
- **Framework** for storage providers to integrate their solutions into cloud-native environments
    - Storage resource normalization
    - Operator patterns/plumbing
    - Common policies, specs, logic
    - Testing effort
- Ceph, CockroachDB, Minio, NFS, Cassandra, EdgeFS, and more...

Community Driven Effort

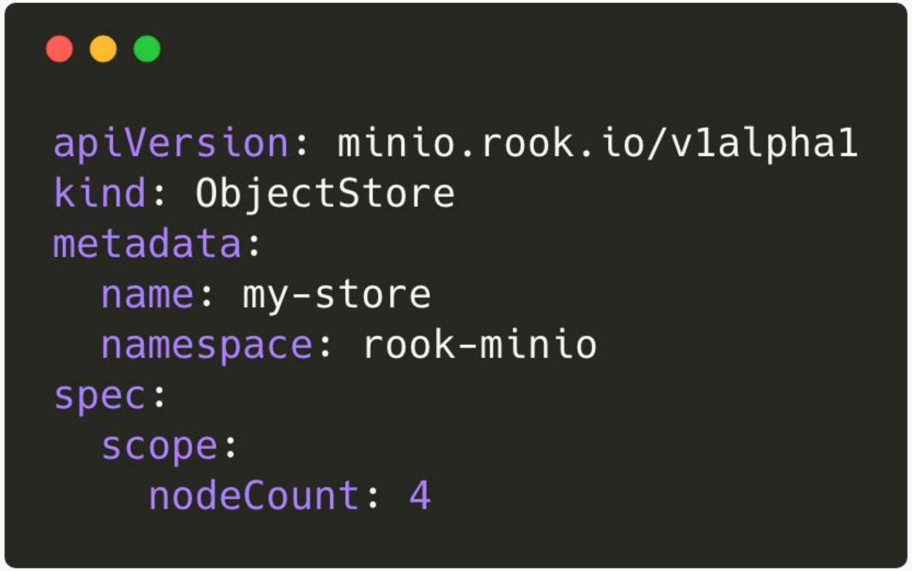
- Rook’s framework has enabled new contributors to add new storage solutions - the community is growing!
- Storage teams themselves are enabled
- **Yannis Zarkadas** at Arrikto - amazing effort on **Cassandra**
- **Rohan Gupta** - Google Summer of Code project for **NFS**
- **Nexenta** team - **EdgeFS**

Minio ObjectStore CRD



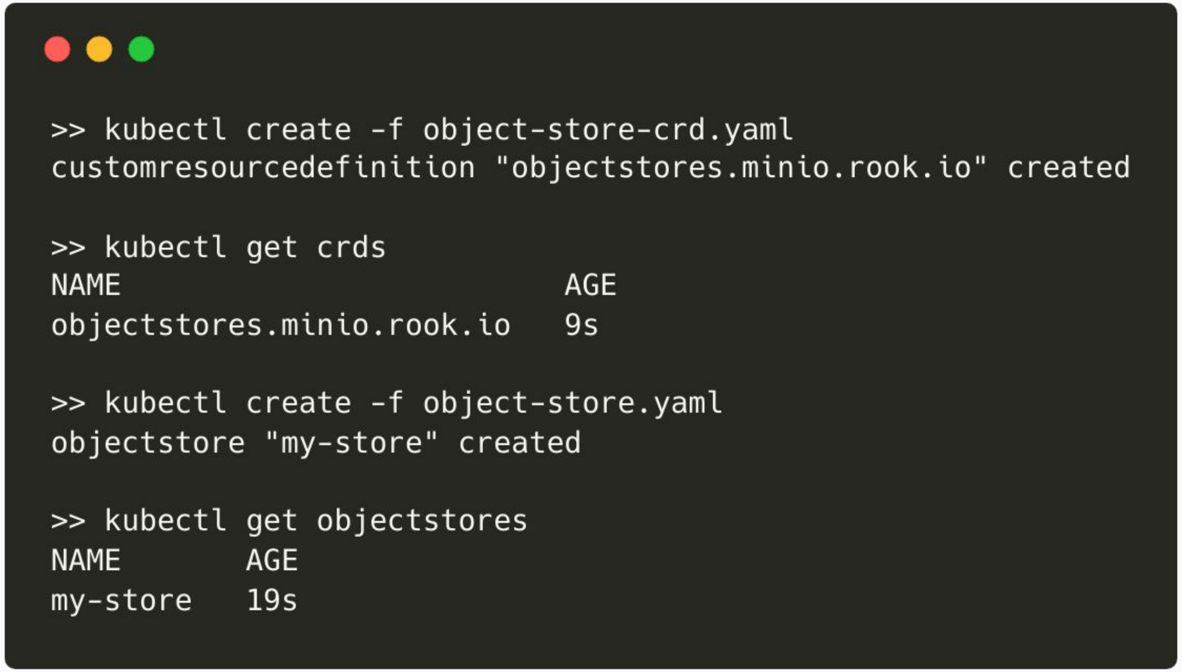
```yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: objectstores.minio.rook.io
spec:
  group: minio.rook.io
  names:
    kind: ObjectStore
    listKind: ObjectStoreList
    plural: objectstores
    singular: objectstore
  scope: Namespaced
  version: v1alpha1
```

Minio ObjectStore Custom Object



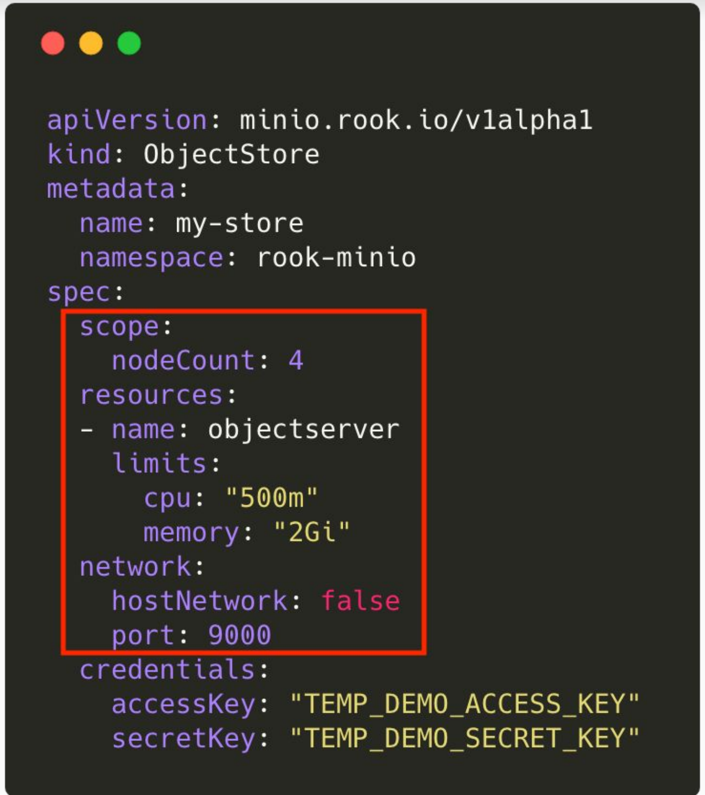
```yaml
apiVersion: minio.rook.io/v1alpha1
kind: ObjectStore
metadata:
  name: my-store
  namespace: rook-minio
spec:
  scope:
    nodeCount: 4
```

Using the Object Store CRD



```
>> kubectl create -f object-store-crd.yaml
customresourcedefinition "objectstores.minio.rook.io" created

>> kubectl get crds
NAME                         AGE
objectstores.minio.rook.io   9s

>> kubectl create -f object-store.yaml
objectstore "my-store" created

>> kubectl get objectstores
NAME       AGE
my-store   19s
```

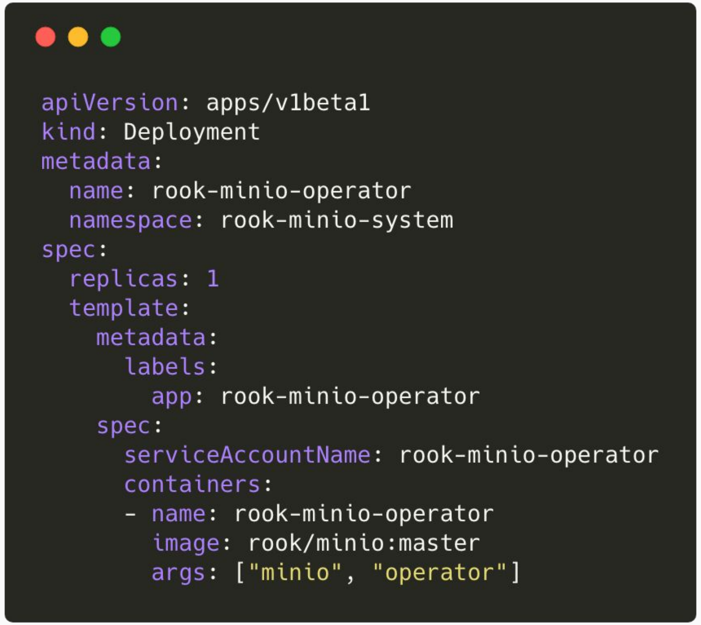
Revisiting the ObjectStore



```yaml
apiVersion: minio.rook.io/v1alpha1
kind: ObjectStore
metadata:
  name: my-store
  namespace: rook-minio
spec:
  scope:
    nodeCount: 4
  resources:
  - name: objectserver
    limits:
      cpu: "500m"
      memory: "2Gi"
  network:
    hostNetwork: false
    port: 9000
  credentials:
    accessKey: "TEMP_DEMO_ACCESS_KEY"
    secretKey: "TEMP_DEMO_SECRET_KEY"
```

- Rook knows how to work with common information in storage object specs (networking, node counts, etc.)
- Only the credentials are Minio-specific
- We can use this information to deploy a Minio cluster

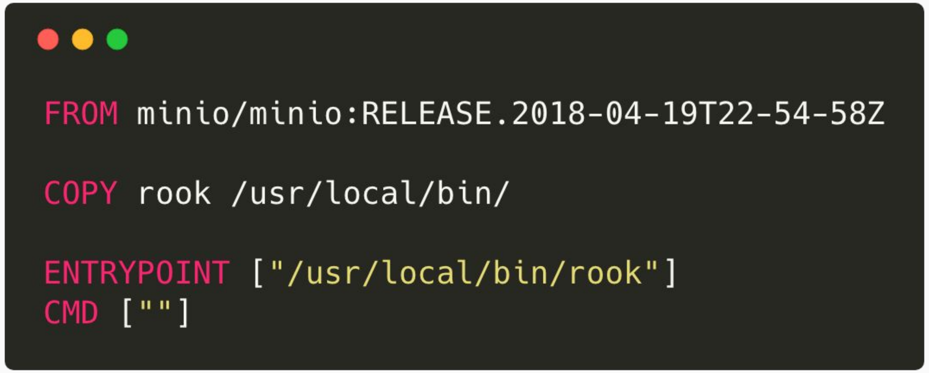
Minio Operator



```yaml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: rook-minio-operator
  namespace: rook-minio-system
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: rook-minio-operator
    spec:
      serviceAccountName: rook-minio-operator
      containers:
      - name: rook-minio-operator
        image: rook/minio:master
        args: ["minio", "operator"]
```

- We specify the container that the Minio operator will reside in
- Args are provided to inform the Rook binary that it needs to operate on Minio
- We include the CRD in the same file as this operator description

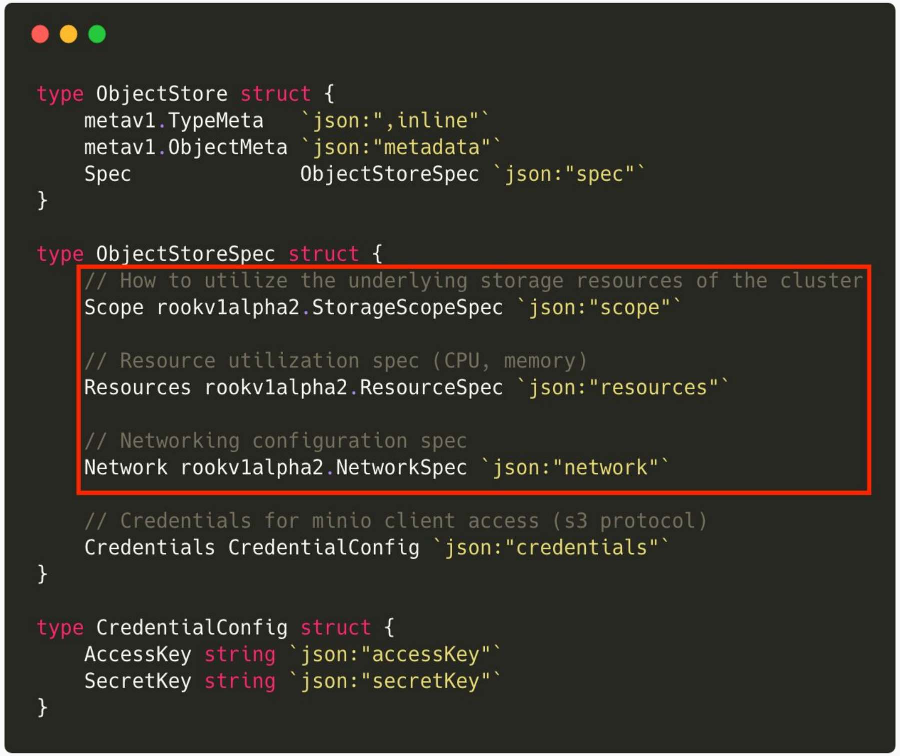
Minio Operator Container Image



```
FROM minio/minio:RELEASE.2018-04-19T22-54-58Z

COPY rook /usr/local/bin/

ENTRYPOINT ["/usr/local/bin/rook"]
CMD [""]
```

- Contains both Minio server/tools and Rook libraries
- Optimized Docker build to collapse layers and minify image
- Base image is Alpine Linux

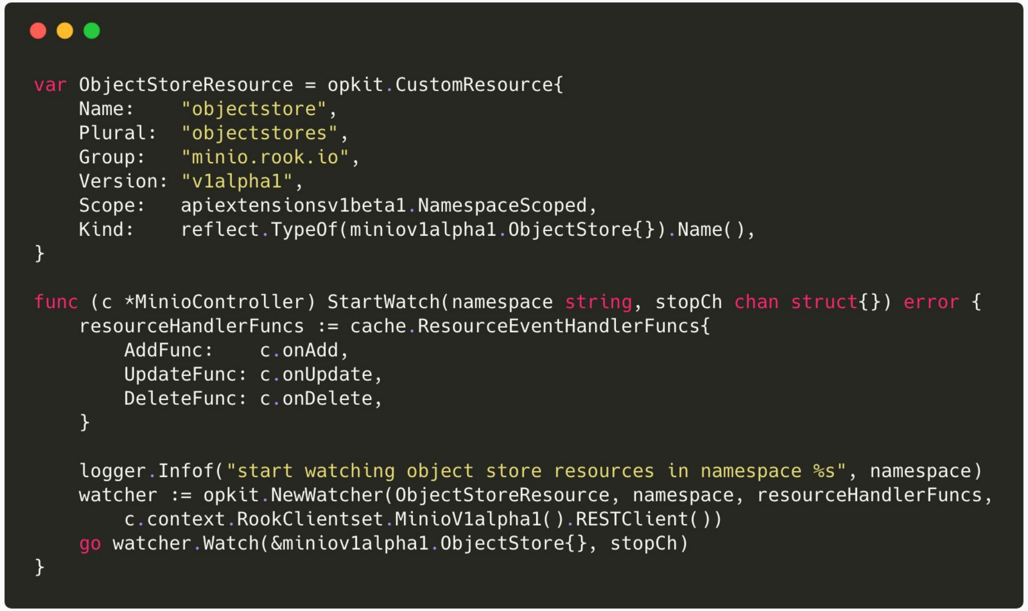
Minio ObjectStore Golang Types



```
type ObjectStore struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata"`
    Spec              ObjectStoreSpec `json:"spec"`
}

type ObjectStoreSpec struct {
    // How to utilize the underlying storage resources of the cluster
    Scope rookv1alpha2.StorageScopeSpec `json:"scope"`

    // Resource utilization spec (CPU, memory)
    Resources rookv1alpha2.ResourceSpec `json:"resources"`

    // Networking configuration spec
    Network rookv1alpha2.NetworkSpec `json:"network"`

    // Credentials for minio client access (s3 protocol)
    Credentials CredentialConfig `json:"credentials"`
}

type CredentialConfig struct {
    AccessKey string `json:"accessKey"`
    SecretKey string `json:"secretKey"`
}
```

- ObjectStoreSpec struct defines the config properties exposed to the user in `object-store.yaml`
- Notice the spec takes advantage of the common types/specs from the Rook framework

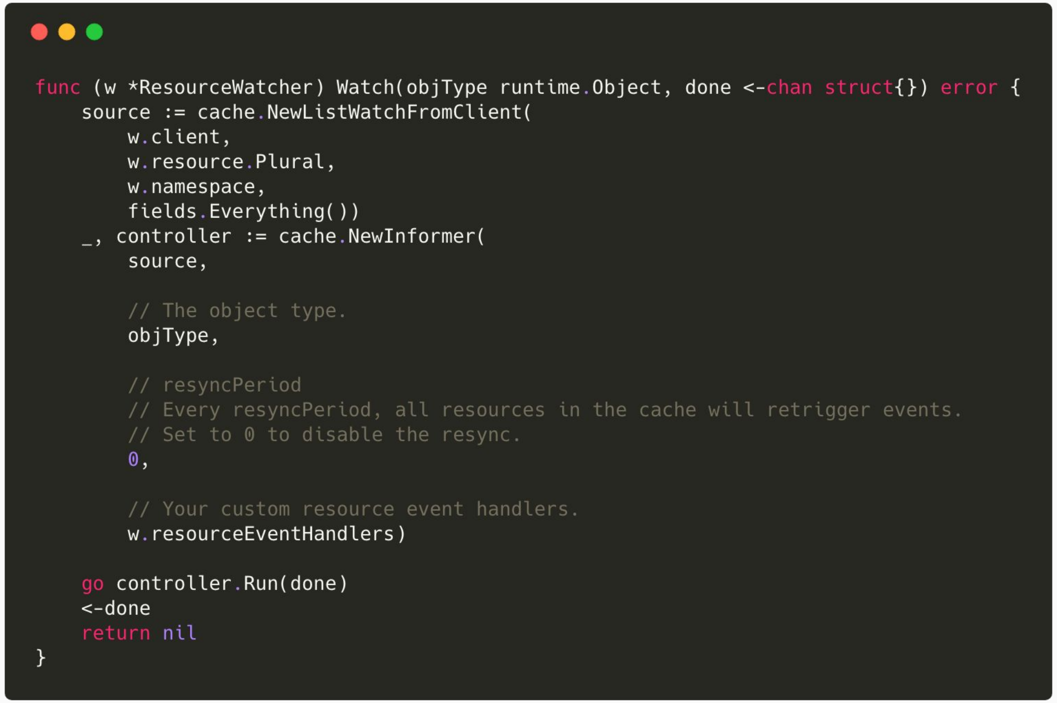
Minio Operator Watching for Events



```go
var ObjectStoreResource = opkit.CustomResource{
    Name:    "objectstore",
    Plural:  "objectstores",
    Group:   "minio.rook.io",
    Version: "v1alpha1",
    Scope:   apiextensionsv1beta1.NamespaceScoped,
    Kind:    reflect.TypeOf(miniov1alpha1.ObjectStore{}).Name(),
}

func (c *MinioController) StartWatch(namespace string, stopCh chan struct{}) error {
    resourceHandlerFuncs := cache.ResourceEventHandlerFuncs{
        AddFunc:    c.onAdd,
        UpdateFunc: c.onUpdate,
        DeleteFunc: c.onDelete,
    }

    logger.Infof("start watching object store resources in namespace %s", namespace)
    watcher := opkit.NewWatcher(ObjectStoreResource, namespace, resourceHandlerFuncs,
        c.context.RookClientset.MinioV1alpha1().RESTClient())
    go watcher.Watch(&miniov1alpha1.ObjectStore{}, stopCh)
}
```

- We create a new watcher to watch for **add**, **update**, or **delete** events
- Event handler functions are passed to the Rook operator-kit

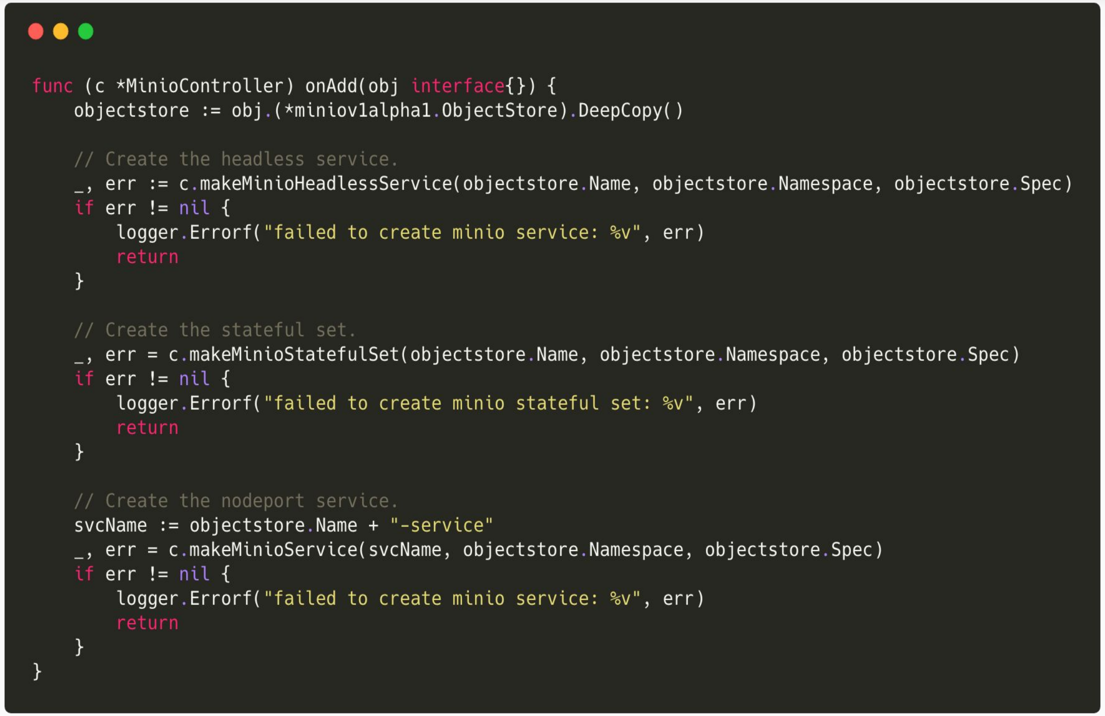
Watching with Informers



```
func (w *ResourceWatcher) Watch(objType runtime.Object, done <-chan struct{}) error {
    source := cache.NewListWatchFromClient(
        w.client,
        w.resource.Plural,
        w.namespace,
        fields.Everything())
    _, controller := cache.NewInformer(
        source,

        // The object type.
        objType,

        // resyncPeriod
        // Every resyncPeriod, all resources in the cache will retrigger events.
        // Set to 0 to disable the resync.
        0,

        // Your custom resource event handlers.
        w.resourceEventHandlers)

    go controller.Run(done)
    <-done
    return nil
}
```

- We use an Informer to watch for k8s events, which prevents excessive polling on the API server
- The informer keeps a cache of objects to limit GETs

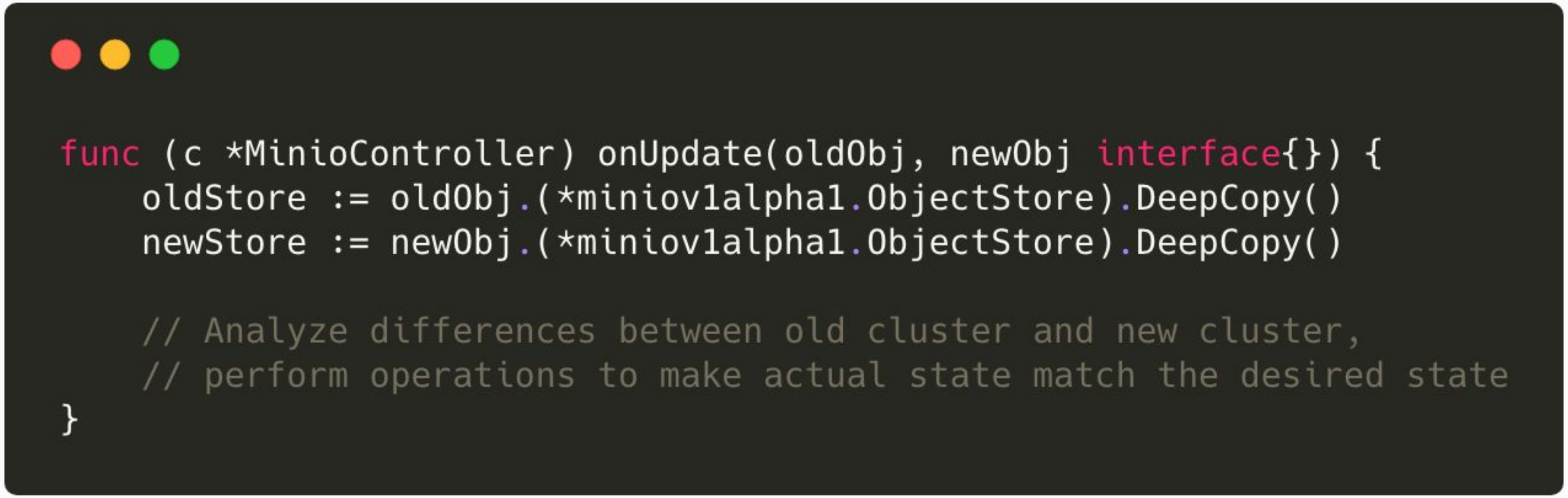
ObjectStore Add Handler



```go
func (c *MinioController) onAdd(obj interface{}) {
    objectstore := obj.(*miniov1alpha1.ObjectStore).DeepCopy()

    // Create the headless service.
    _, err := c.makeMinioHeadlessService(objectstore.Name, objectstore.Namespace, objectstore.Spec)
    if err != nil {
        logger.Errorf("failed to create minio service: %v", err)
        return
    }

    // Create the stateful set.
    _, err = c.makeMinioStatefulSet(objectstore.Name, objectstore.Namespace, objectstore.Spec)
    if err != nil {
        logger.Errorf("failed to create minio stateful set: %v", err)
        return
    }

    // Create the nodeport service.
    svcName := objectstore.Name + "-service"
    _, err = c.makeMinioService(svcName, objectstore.Namespace, objectstore.Spec)
    if err != nil {
        logger.Errorf("failed to create minio service: %v", err)
        return
    }
}
```

- The onAdd handler implementation uses the K8s API to create services, stateful sets, etc.
- We programmatically follow the deployment procedure for the Minio cluster

ObjectStore Update Handler

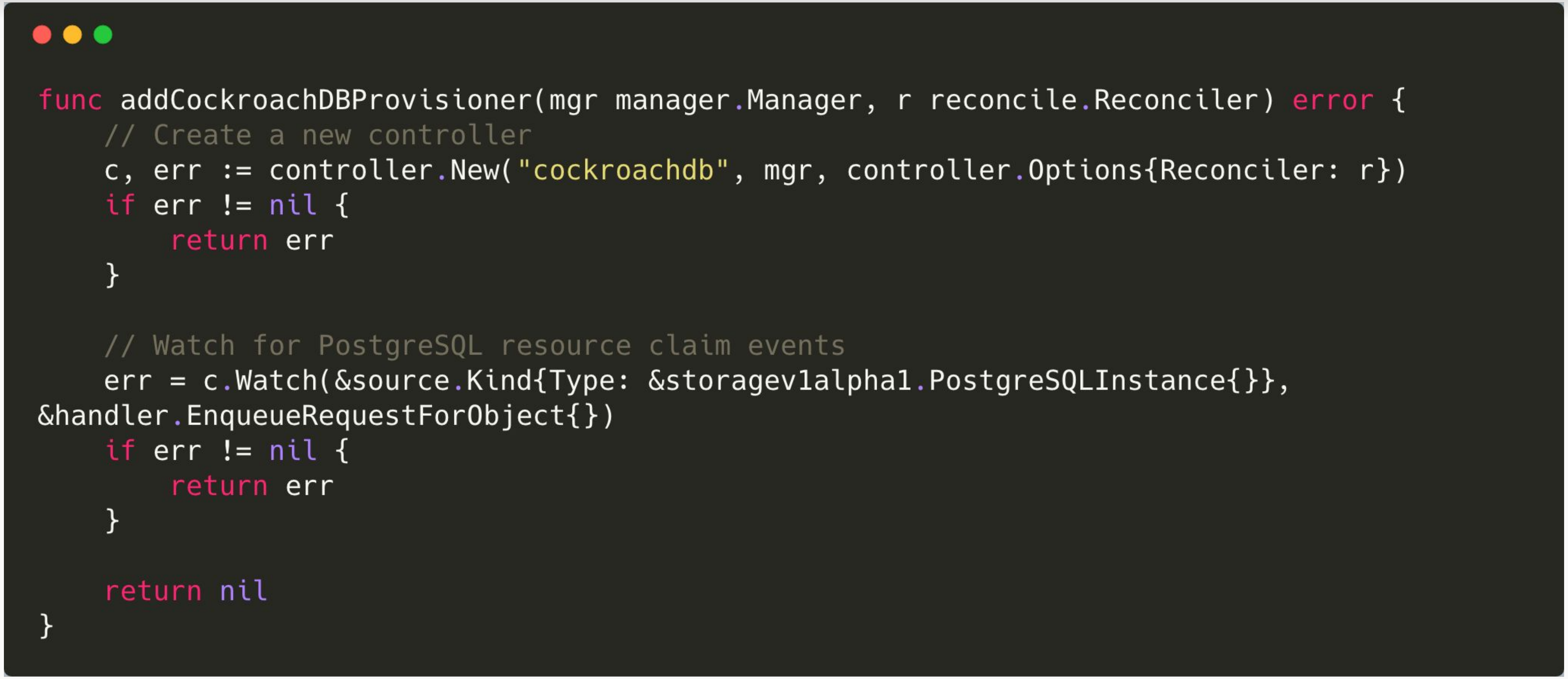


```go
func (c *MinioController) onUpdate(oldObj, newObj interface{}) {
    oldStore := oldObj.(*miniov1alpha1.ObjectStore).DeepCopy()
    newStore := newObj.(*miniov1alpha1.ObjectStore).DeepCopy()

    // Analyze differences between old cluster and new cluster,
    // perform operations to make actual state match the desired state
}
```

Dynamic provisioning for new storage types

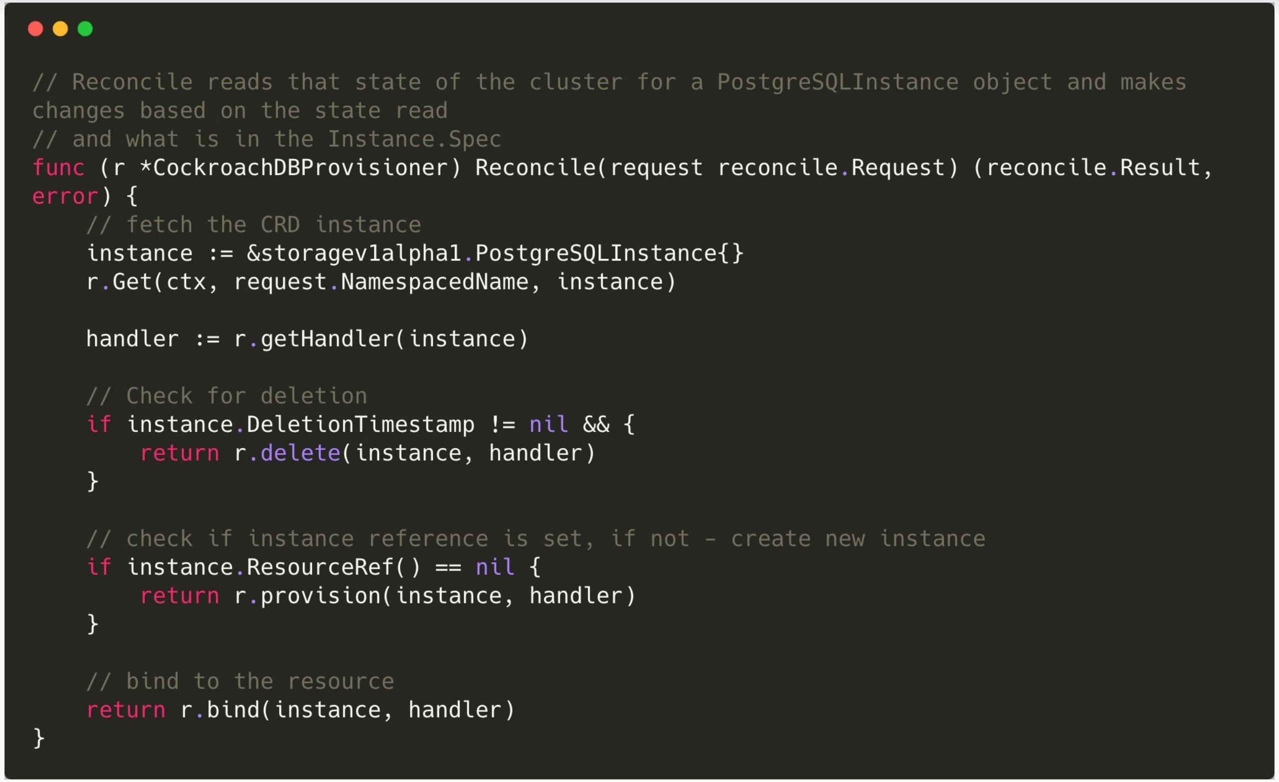
- Similar pattern to **StorageClass** and **PersistentVolumeClaim**
- **ResourceClass** - a “blueprint” created by the administrator
  - contains all environment specifics and details to create a “class” of storage
  - Fast, Standard, Cheap, etc.
- **ResourceClaim** - developer defined, simply expresses their general need for a given storage type
- Separation of concerns - promotes reusability and reduces complexity
- Storage is created on demand as it’s needed, no need to pre-allocate
- Enables **portability** and the power of choice
- ***Write once, run anywhere***

Dynamic Provisioner - Observe



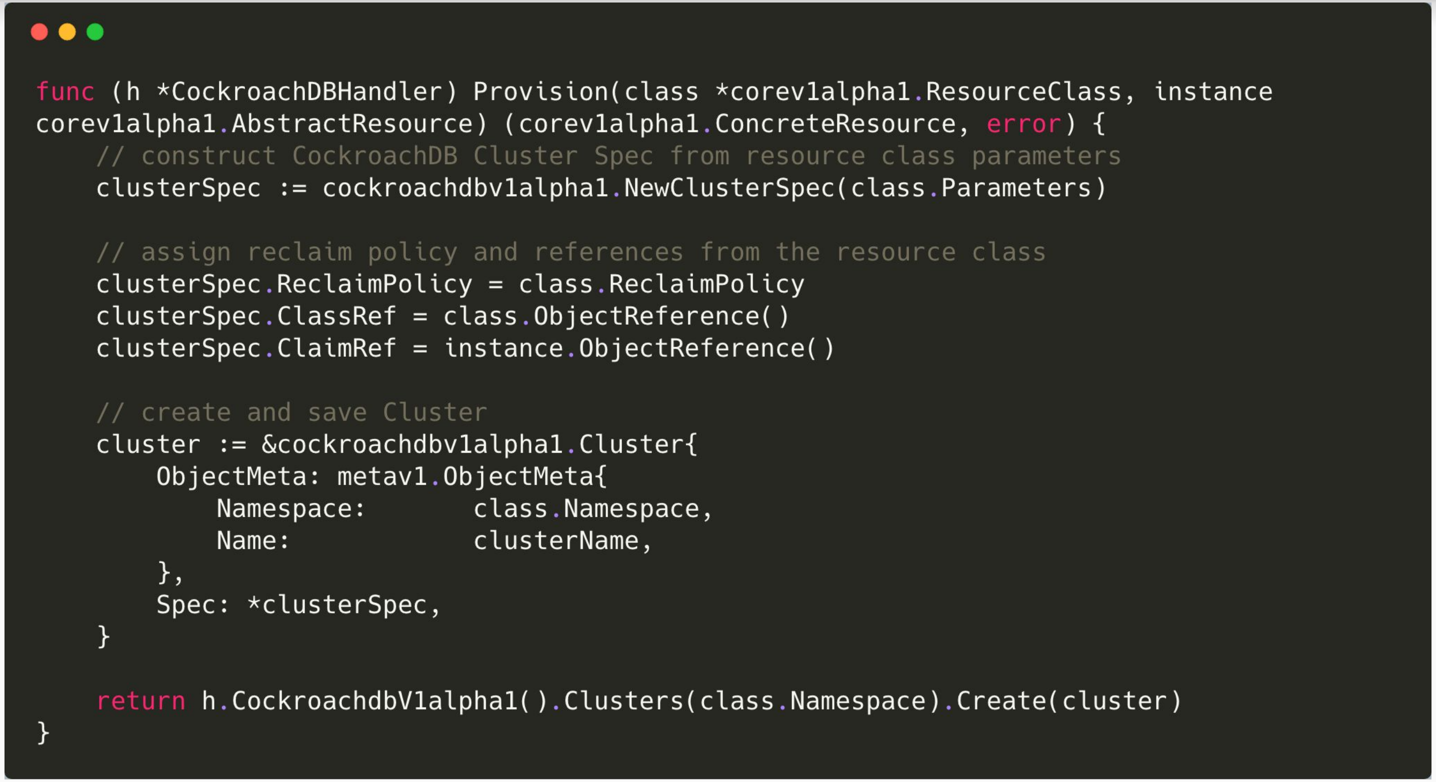
```go
func addCockroachDBProvisioner(mgr manager.Manager, r reconcile.Reconciler) error {
    // Create a new controller
    c, err := controller.New("cockroachdb", mgr, controller.Options{Reconciler: r})
    if err != nil {
        return err
    }

    // Watch for PostgreSQL resource claim events
    err = c.Watch(&source.Kind{Type: &storagev1alpha1.PostgreSQLInstance{}},
&handler.EnqueueRequestForObject{})
    if err != nil {
        return err
    }

    return nil
}
```

Dynamic Provisioner - Analyze



```go
// Reconcile reads that state of the cluster for a PostgreSQLInstance object and makes
changes based on the state read
// and what is in the Instance.Spec
func (r *CockroachDBProvisioner) Reconcile(request reconcile.Request) (reconcile.Result,
error) {
    // fetch the CRD instance
    instance := &storagev1alpha1.PostgreSQLInstance{}
    r.Get(ctx, request.NamespacedName, instance)

    handler := r.getHandler(instance)

    // Check for deletion
    if instance.DeletionTimestamp != nil && {
        return r.delete(instance, handler)
    }

    // check if instance reference is set, if not - create new instance
    if instance.ResourceRef() == nil {
        return r.provision(instance, handler)
    }

    // bind to the resource
    return r.bind(instance, handler)
}
```

Dynamic Provisioner - Act



```go
func (h *CockroachDBHandler) Provision(class *corev1alpha1.ResourceClass, instance
corev1alpha1.AbstractResource) (corev1alpha1.ConcreteResource, error) {
    // construct CockroachDB Cluster Spec from resource class parameters
    clusterSpec := cockroachdbv1alpha1.NewClusterSpec(class.Parameters)

    // assign reclaim policy and references from the resource class
    clusterSpec.ReclaimPolicy = class.ReclaimPolicy
    clusterSpec.ClassRef = class.ObjectReference()
    clusterSpec.ClaimRef = instance.ObjectReference()

    // create and save Cluster
    cluster := &cockroachdbv1alpha1.Cluster{
        ObjectMeta: metav1.ObjectMeta{
            Namespace:        class.Namespace,
            Name:             clusterName,
        },
        Spec: *clusterSpec,
    }

    return h.CockroachdbV1alpha1().Clusters(class.Namespace).Create(cluster)
}
```

Demo

PostgreSQL Dynamic Provisioning

CockroachDB on-premises

Google CloudSQL in the cloud

What did we cover today?

- Rook is a cloud-native storage orchestrator
- Framework to create storage operators that deploy, configure, and manage many storage solutions in Kubernetes
- Dynamically provision all sorts of storage types in the cloud and on-premises with Crossplane & Rook
- Separation of concerns for admins and devs - promote reusability, reduce complexity
- Portability - power of choice

How to get involved?

- Contribute to Rook and Crossplane
  - https://rook.io/
  - https://crossplane.io/
- Slack
  - https://slack.rook.io/
  - https://slack.crossplane.io/
- Twitter - @rook_io & @crossplane_io
- Forums - rook-dev & crossplane-dev on google groups
- Community Meetings

Questions?

https://rook.io/

https://crossplane.io/

Thank you!

https://rook.io/

https://crossplane.io/