ReplicaSets & Deployments
(The Underrated, OG Operators)
Why do we care about ReplicaSets (formerly ReplicaControllers)?
Redundancy

Multiple running instances means failure can be tolerated.
Scale

Multiple running instances mean more requests can be handled.
Sharding

Multiple running instances can handle different parts of a computation in parallel.
ReplicaSets in Action!

```
kubectl create -f myfirstreplicaset.yaml
```

```
kubectl scale replicaset myfirstreplicaset --replicas=3
```

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: myfirstreplicaset
spec:
  selector:
    matchLabels:
      app: myfirstapp
  replicas: 3
  template:
    metadata:
      labels:
        app: myfirstapp
    spec:
      containers:
        - name: nodejs
          image: myimage
```
The ReplicaSet Control Loop

1. Start
2. Find pods matching the label selector
3. Compare matched vs. desired pod count
   - Too few: Create additional pod(s) from the current template
   - Just enough
   - Too many: Delete the excess pod(s)
How do we accomplish this?
The reconciler pattern is a software pattern that can be used or expanded upon for managing cloud native infrastructure. The pattern enforces the idea of having two representations of the infrastructure—the first being the actual state of the infrastructure, and the second being the expected state of the infrastructure.
The **reconciler pattern** will force the engineer to have two independent avenues for getting either of these representations, as well as to implement a solution to reconcile the actual state into the expected state.
ReplicaSets in Action!

```yaml
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: myfirstreplicaset
spec:
spec:
  selector:
    matchLabels:
      app: myfirstapp
  replicas: 3
template:
  metadata:
    labels:
      app: myfirstapp
  spec:
    containers:
    - name: nodejs
      image: myimage
```

```bash
kubectl create -f myfirstreplicaset.yaml
```

0 < spec.replicas?

1 < spec.replicas?

2 < spec.replicas?

3 < spec.replicas?
ReplicaSets in Action!

```yaml
apiVersion: extensions/v1beta1
custom:
kind: ReplicaSet
metadata:
  name: myfirstreplicaset
spec:
  selector:
    matchLabels:
      app: myfirstapp
  replicas: 3
  template:
    metadata:
      labels:
        app: myfirstapp
    spec:
      containers:
        - name: nodejs
          image: myimage
```

kubectl create -f myfirstreplicaset.yaml

Delete Event

Pod 1 Delete Event

Add Event

Pod 4 Add Event

2 < spec.replicas?

3 < spec.replicas?
Garbage Collection (GC)
Garbage Collection assists in deleting objects that have an owner that no longer exists.
OwnerReferences

```
kubectl create -f myfirstreplicaset.yaml
```

```yaml
apiVersion: extensions/v1beta1
g kind: ReplicaSet
metadata:
  name: myfirstreplicaset
spec:
  selector:
    matchLabels:
      app: myfirstapp
  replicas: 3
  template:
    metadata:
      labels:
        app: myfirstapp
    spec:
      containers:
      - name: nodejs
        image: myimage
```
OwnerReferences

- apiVersion: apps/v1
- blockOwnerDeletion: true
- controller: true
- kind: ReplicaSet
- name: myfirstreplicaset
- uid: 30c68160-d992-11e8-84d9-e6f5b7702569

*querying API for UID not currently supported.

Only applicable when doing "foreground" delete (optional).

Strictly informational: shows that a Controller set the ownerReferences (optional).
Finalizers

Allows controllers to implement conditions that must be completed before the object can be deleted.

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  finalizers:
    - finalizer.stable.example.com

metadata:
  deletionTimestamp: 2018-10-20T01:16:04Z
```