

# Kubernetes Operators 101

<http://bit.ly/operators-101>

---

Ali Ok  
Senior Software Engineer  
Red Hat

# About me

Senior Software Engineer @ Red Hat

Knative Eventing Team

Remotee in Istanbul

# WORK REQUEST

Work Order Request No:

# 779

Date of Registration:

24-11-2018

Reason For Requesting:

Postgres DB with 2 instances, version  
11, volume 16iB, ...

Requested Person Name:

Mike Brown

*Mike*

Signature of Authority

# Agenda

First show some stuff

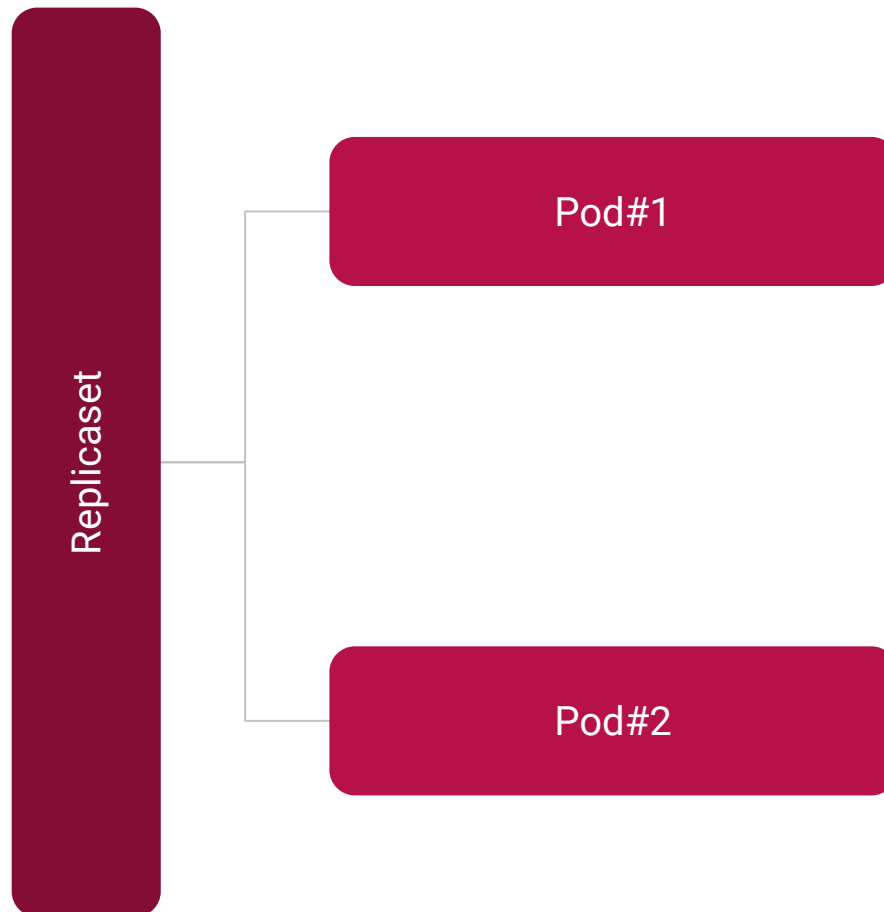
Then explain why we need all of these with use cases

Compare to other approaches

# Step back

Kubernetes resources

# Kubernetes resources



```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
spec:
  replicas: 3
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
        - name: php-redis
          image: gcr.io/google_samples/gb-frontend:v3
  ...
```

# Kubernetes resources

```
$ kubectl get replicaset
```

NAME	DESIRED	CURRENT	READY	AGE
frontend	3	3	3	5s

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
frontend-8rxnz	1/1	Running	0	7s
frontend-t2ffx	1/1	Running	0	7s
frontend-z7dg7	1/1	Running	0	7s



# Custom resources

What if Kubernetes would understand custom resources like a definition of a Postgres cluster?

---

```
apiVersion: example.com/v1
kind: postgresql
metadata:
  name: my-postgres-db
spec:
  instances: 2
  version: '11'
  users:
    jack:
      - superuser
      - createdb
  volume:
    size: 1Gi
```

```
$ kubectl apply -f postgres-cr.yaml  
postgresql.example.com/my-postgres-db created
```

```
$ kubectl get postgresqls  
NAME                VERSION  INSTANCES  VOLUME  AGE    STATUS  
my-postgres-db      2        2          1Gi     12s   
```

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: postgresqls.example.com
spec:
  group: example.com
  names:
    kind: postgresql
    shortNames:
      - pg
  validation:
    ...
    spec:
      type: object
      properties:
        instances:
          type: integer
        version:
          type: string
      ...
  additionalPrinterColumns:
    - name: Instances
      type: integer
      JSONPath: .spec.instances
    ...
  scope: Namespaced
  subresources:
    status: {}
```

...

Ok, we have the custom resource there, but now what?

It is not going to create actual Postgres instances itself!

# Reconciliation

---

```
for {  
    desired := getDesiredState()  
    current := getCurrentState()  
    makeChanges(desired, current)  
}
```

---

# Basic Operator

Simple application that uses Kubernetes REST api to create/update/delete Postgres instances

# Basic Operator

```
resource := schema.GroupVersionResource{Group: "example.com", Version: "v1", Resource: "postgresqls"}

for {
    list, _ := client.Resource(resource).Namespace(namespace).List(...)
    for _, item := range list.Items {

        if handled(item){
            continue
        }

        instances, _, _ := unstructured.NestedString(item.Object, "metadata", "instances")
        // ...
        createPostgres(instances, ...)
    }
}
```



Ok, but ...

... it is awful to use unstructured data, especially in a strongly typed language

# Types for the CRDs

Let's create types for the CRDs and make the REST client use them

# Types for the CRDs

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

    Spec PostgreSQLSpec `json:"spec"`
    ...
}

type PostgreSQLSpec struct {
    Instances int32 `json:"instances"`
    Version string `json:"version"`
    ...
}

type PostgreSQLList struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ListMeta `json:"metadata,omitempty"`

    Items []PostgreSQL `json:"items"`
}
```

# Typed REST client usage

```
var config *rest.Config
...
crdConfig := *config
crdConfig.ContentConfig.GroupVersion = &schema.GroupVersion{Group: "example.com", Version: "v1"}
...

restClient, _ := rest.RESTClientFor(&crdConfig)

for {
    result := PostgresqlList{}
    restClient.Get().Resource("postgresqls").Do().Into(&result)

    for _, item := range result.Items {
        if handled(item) {
            continue
        }

        instances, _, _ := item.Spec.Instances
        // ...
        createPostgres(instances, ...)
    }
}
```

Ok, but ...

... it is awful to define and register these types manually and also do the REST client configuration

# Code generation for the CRDs

Let's generate code for the CRDs and also register them

# Code generation for the CRDs

---

```
// +k8s:deepcopy-gen=package,register
// +groupName=example.com
package v1

// +genclient
// +genclient:noStatus
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object

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

    Spec PostgresqlSpec `json:"spec"`
    ...
}

type PostgresqlSpec struct {
    Instances int32 `json:"instances"`
    Version string `json:"version"`
    ...
}
```

---

# Code generation for the CRDs

will generate CRD yaml file to feed to Kubernetes, as well as a typed REST clientset



## Code generation for the CRDs

```
client, _ := examplecomclientset.NewForConfig(cfg)

for {
    items, _ := exampleClient.ExampleV1().Postgresqls(namespace).List(...)
    ...
}
```

## Watch instead of poll

---

```
watcher, _ := exampleClient.ExampleV1().Postgresqls(namespace).Watch(...)

ch := watcher.ResultChan()

for event := range ch {
    postgres, _ := event.Object.(*Postgresql)

    switch event.Type {
        case watch.Added:
            // create new Postgres DB
        case watch.Deleted:
            // delete the Postgres DB created by this CR
        case watch.Modified:
            // update the Postgres DB created by this CR
    }
}
```

# Informers for better eventing model

Let the informer filter the events for you

...

# Shared informers

You can have multiple controllers where you want to be informed about the same set of resources

Especially secondary resources

# Work queues

Retries

Rate limiting

# Leader election

Ability to horizontally scale the operator

# More and more stuff

...

# Enter Operator SDK

All of the steps are provided for you

CLI to do code generation and scaffolding

High level API and abstraction



# Enter Operator SDK

A particular opinionated way to implement operators

Best practices are included

No need to be a Kubernetes / Golang expert

# Demo

DEMO TIME!

<https://github.com/aliok/postgres-operator-complete>

# Operator Framework

SDK + CLI

Operator Lifecycle Manager (OLM)

OperatorHub

Operator Metering

# OperatorHub



Marketplace for Operators

# Welcome to OperatorHub.io

OperatorHub.io is a new home for the Kubernetes community to share Operators. Find an existing Operator or list your own today.

CATEGORIES

77 ITEMS

VIEW  SORT A-Z 
















- AI/Machine Learning
- Application Runtime
- Big Data
- Cloud Provider
- Database
- Developer Tools
- Integration & Delivery
- Logging & Tracing
- Monitoring
- Networking
- OpenShift Optional
- Security
- Storage
- Streaming & Messaging

PROVIDER

- Altinity (1)
- Amazon Web Services (1)
- Appsoy (1)
- Aqua Security (1)
- AtlasMap (1)

[Show 50 more](#)

CAPABILITY LEVEL

 <b>Akka Cluster Operator</b> provided by Lightbend, Inc.  Run Akka Cluster applications on OpenShift.	 <b>Altinity ClickHouse Operator</b> provided by Altinity  ClickHouse Operator manages full lifecycle of ClickHouse	 <b>Apache CouchDB</b> provided by IBM  Apache CouchDB™ is a highly available NOSQL database for web and mobile application	 <b>Apache Spark Operator</b> provided by radanalytics.io  An operator for managing the Apache Spark clusters and intelligent applications that	 <b>Appsoy Operator</b> provided by Appsoy  Deploys Appsoy based applications
 <b>Aqua Security Operator</b> provided by Aqua Security, Inc.  The Aqua Security Operator runs within Kubernetes cluster and provides a means to	 <b>AtlasMap Operator</b> provided by AtlasMap  AtlasMap is a data mapping solution with an interactive web based user interface, t	 <b>AWS S3 Operator</b> provided by Red Hat  Manage the full lifecycle of installing, configuring and managing AWS S3 Provisio	 <b>AWS Service Operator</b> provided by Amazon Web Services, Inc.  The AWS Service Operator allows you to manage AWS	 <b>Banzai Cloud Kafka Operator</b> provided by Banzai Cloud  Installs and maintains Kafka
 <b>Camel K Operator</b> provided by The Apache	 <b>CockroachDB</b> provided by Helm Community	 <b>Community Jaeger Operator</b> provided by CNCF	 <b>Crunchy PostgreSQL Enterprise</b>	 <b>Dynatrace OneAgent</b> provided by Dynatrace LLC



# Postgres-Operator

Postgres operator creates and manages PostgreSQL clusters running in Kubernetes.

[Home](#) > Postgres-Operator

## Postgres-Operator

The Postgres operator manages PostgreSQL clusters on Kubernetes.

### Key principles

- **Hands free:** Configuration happens only via manifests and its own config
- **Cloud native:** Easy integration in automated deploy pipelines with no access to Kubernetes directly
- **Scalable:** Run highly available PostgreSQL clusters powered by Patroni

### How it works

The operator watches additions, updates, and deletions of PostgreSQL cluster manifests and changes the running clusters accordingly. For each PostgreSQL custom resource it creates StatefulSets, Services, and also Postgres roles. For some configuration changes, e.g. updating a pod's Docker image, the operator carries out the rolling update.

### Creating a Postgres cluster

After installing the Postgres Operator via OLM you can use the provided YAML examples to create a minimal cluster setup with two instances.

```
# First, make sure the operator is running
kubectl get pod -l name=postgres-operator -n operators

# Then create a new Postgres cluster with a manifest file
kubectl create -n <namespace> -f manifests/minimal-postgres-manifest.yaml
```

Install

#### CHANNEL

stable

#### VERSION

1.2.0 (Current)

#### CAPABILITY LEVEL

- Basic Install
- Seamless Upgrades
- Full Lifecycle
- Deep Insights
- Auto Pilot

#### PROVIDER

Zalando SE

#### LINKS

[Documentation](#)

# Operator maturity model



# OLM

Operators that manage your operators

Some level of dependency resolution

Upgrades

Enabling users to use services



# Agenda

First show some stuff

**Then explain why we need all of these with use cases**

Compare to other approaches

# Human operators vs software operators

Target is automating the software operating!

# Use cases

Installing stuff

State preservation

Managed software

Stateful apps

# Real world operator - Postgres DB creation

---

```
apiVersion: "acid.zalan.do/v1"
kind: postgresql
metadata:
  name: acid-minimal-cluster
spec:
  teamId: "ACID"
  volume:
    size: 1Gi
  numberOfInstances: 2
  users:
    zalando:
      - superuser
      - createdb
  databases:
    foo: zalando
  postgresql:
    version: "10"
```

---

# Real world operator - Keycloak realm creation in existing server

---

```
apiVersion: aerogear.org/v1alpha1
kind: KeycloakRealm
spec:
  clients:
    - baseUrl: https://keycloak.com
      bearerOnly: true
    ...
  identityProviders:
    - alias: github
      config:
        clientId: test
        clientSecret: test
        disableUserInfo: ""
        hideOnLoginPage: ""
        useJwksUrl: "true"
      enabled: true
      providerId: github
    ...
```

---

## Real world operator - Grafana dashboard creation in existing service

---

```
apiVersion: integreatly.org/v1alpha1
kind: GrafanaDashboard
metadata:
  name: simple-dashboard
  labels:
    app: grafana
spec:
  name: simple-dashboard.json
  json: >
    {
      "id": null,
      "title": "Simple Dashboard",
      "tags": [],
      "style": "dark",
      "timezone": "browser",
      "editable": true,
      ...
    }
```

# Real world operators - OpenShift

Operator that handles network config

Operator that handles web console

Operator that handles authorization

...

# Compare: kubebuilder

Kubebuilder and Operator SDK converge

Now, both use controller-runtime and controller-tools

Conversion from one to another is easy



## Compare: kubernetes

Operator SDK has support for Operator Framework features

More:

[github.com/operator-framework/operator-sdk/issues/1758](https://github.com/operator-framework/operator-sdk/issues/1758)

## Compare: Metacontroller

Complete abstraction of machinery

Webhook based and you can use any language you like

Not flexible enough, but can be sufficient in some use cases

# Ansible, Helm, Java

Operator Framework also supports Ansible and Helm operators

If you have existing Ansible playbooks or Helm charts, you can wrap them in an operator

There are some Java Operator SDKs which can be sufficient in some cases

# Summary

Make use of operators to use other people's expertise in operating the dependencies

Write your own operator to operate your software

# Final words

Not specific to OpenShift!

No need to be an expert in Golang

kube-builder and Operator SDK converging

Awesome operators: [github.com/operator-framework/awesome-operators](https://github.com/operator-framework/awesome-operators)

[learn.openshift.com/operatorframework](https://learn.openshift.com/operatorframework)

# Thank you

Twitter - @aliok\_tr

Github - aliok