Operators and Helm - It Takes Two to Tango

Devdatta Kulkarni
devdatta@cloudark.io
Founder, CloudARK

CloudARK
https://cloudark.io/

About me

Devdatta Kulkarni
Founder, CloudARK

PhD, Distributed systems,
University of Minnesota Minneapolis

Rackspace (2009-2016)
10 years industry experience in the areas of cloud native technologies, OpenStack, PaaS and distributed systems. Adjunct Professor at UT Austin (CS Dept) teaching courses on cloud computing and modern web applications.
LinkedIn

Outline

CloudARK

- What is - Operator and Helm chart
- Why Helm chart for an Operator deployment?
- Analysis of Helm charts of existing Operators
- Guidelines for creating Operator Helm charts
- Summary

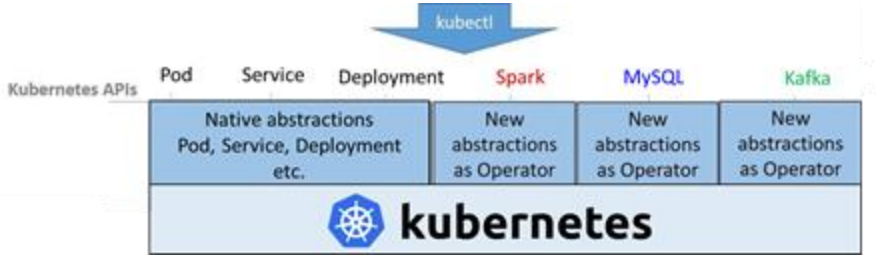
Kubernetes Operator Pattern

CloudARK

- Kubernetes Operator
  - Custom Resource/API + Custom Controller

- Essentially – a Kubernetes API extension that adds Custom APIs / Resources to manage third-party softwares as Kubernetes objects (e.g.: databases)
  - Works with kubectl
  - Works with Kubernetes constructs - Namespace, Service Accounts, RBAC
  - Works with ecosystem tools like Helm

- 400+ GitHub repositories of Operators built for platform softwares like API gateways, relational / non-relational databases, application workloads like Spark, Tensorflow, etc.



High level goal of Helm charts

CloudARK

- Templatization and Parameterization of Kubernetes artifacts via Values.yaml

- Orchestration and ordering via Hooks
  - pre-install, post-install, pre-delete, post-delete, pre-upgrade, post-upgrade, pre-rollback, post-rollback, crd-install

Why Helm Chart for Operator deployment?

CloudARK

- Operator just another Kubernetes artifact
  - Use standard Kubernetes Package Manager for distributing it
- Templatization and Parameterization of Operator Deployment
  - Standard way to customize Operator deployments for different setups
- No new Spec format to learn (like OLM or Operator CRD)
  - Use same Helm format for deploying applications (resources/manifests) as well as control plane (Operators)
  - Leverage Kubernetes constructs like labels, annotations etc. on CRD manifests

Analysis of Helm charts of existing Operators

CloudARK

102 Open source Operators analyzed

https://medium.com/@cloudark/analysis-of-open-source-kubernetes-operators-f6be898f2340

For 75 repositories of Operators written in Go:

- Helm charts not defined: 38
- Helm charts defined: 37

- CRD defined as YAML manifests / in Helm chart: 52
- CRD registered in Code: 19
- CRD defined in YAML and also registered in Code: 4

- CRD validation defined in YAML manifests / in Helm: 26

Guidelines for creating Operator Helm charts

CloudARK

1. Register CRDs in Operator Helm chart
2. Define '*helm.sh/hook: crd-install*' annotation on CRD
3. Support Operator configurables via Values.yaml or ConfigMap
4. Define Custom Resource Validation Rules in CRD YAML
5. Additional annotations to enable Custom Resource discovery & binding
   Platform-as-Code/usage : <<detailed information about using a Custom Resource>>
   Platform-as-Code/composition: <<sub-resources that will be created by a Custom Resource >>

1. Register CRDs in Operator Helm Chart

Register CRDs using Helm chart rather than in Operator Go code

What is CRD?

- Custom Resource Definition - YAML that defines the Custom Resources managed by an Operator

Why? – Varying permissions needed by different steps

- Permission for **CRD installation**
  - Cluster scope permissions needed
- Permissions for **Operator Pod installation**
  - Namespace scope permissions sufficient

If CRDs are registered through Operator Go code, all the steps get cluster scope permissions.

2. Define '*helm.sh/hook : crd-install*' annotation on CRD

Why?

- Ensures CRD is registered prior to installation of any Custom Resource of that CRD

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  labels:
    app: '{{ template "mysql-operator.name" . }}'
    chart: '{{ template "mysql-operator.chart" . }}'
    controller-tools.k8s.io: "1.0"
    heritage: '{{ .Release.Service }}'
    release: '{{ .Release.Name }}'
  name: mysqlbackups.mysql.presslabs.org
  annotations:
    helm.sh/hook: crd-install
spec:
  group: mysql.presslabs.org
  names:
    kind: MysqlBackup
    plural: mysqlbackups
  scope: Namespaced
  version: v1alpha1
```

3. Support Operator configurables via Values.yaml or ConfigMap

Use Helm chart Values.yaml or ConfigMap for Operator configurables

Examples of Operator configurable inputs through Values.yaml:

- Option to deploy the Operator cluster-wide or within a namespace
- For a MySQL Operator, which version of MySQL should be installed

Examples of Operator configurable inputs through ConfigMap:

Provide default plugins to be installed (names & their paths) to a Moodle Operator

(If using a ConfigMap, make sure its name is well-documented)

4. Define Custom Resource Validation Rules in CRD YAML

CloudARK

Define Custom Resource Spec Validation rules as part of Custom Resource Definition YAML

```yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: postgreses.postgrescontroller.kubeplus
  annotations:
    helm.sh/hook: crd-install
    platform-as-code/composition: Deployment, Service
spec:
  group: postgrescontroller.kubeplus
  version: v1
  names:
    kind: Postgres
    plural: postgreses
  scope: Namespaced
validation:
   # openAPIV3Schema is the schema for validating custom objects.
    openAPIV3Schema:
      properties:
        spec:
          properties:
            databases:
              type: array
              items:
                type: string
            users:
              type: array
              items:
                type: string
```

5. Additional annotations to enable Custom Resource discovery & binding

CloudARK

kubectl explain is supported for Custom Resources

But it is not enough - Additional information beyond Spec properties is required by application developers to consume Custom Resources e.g.

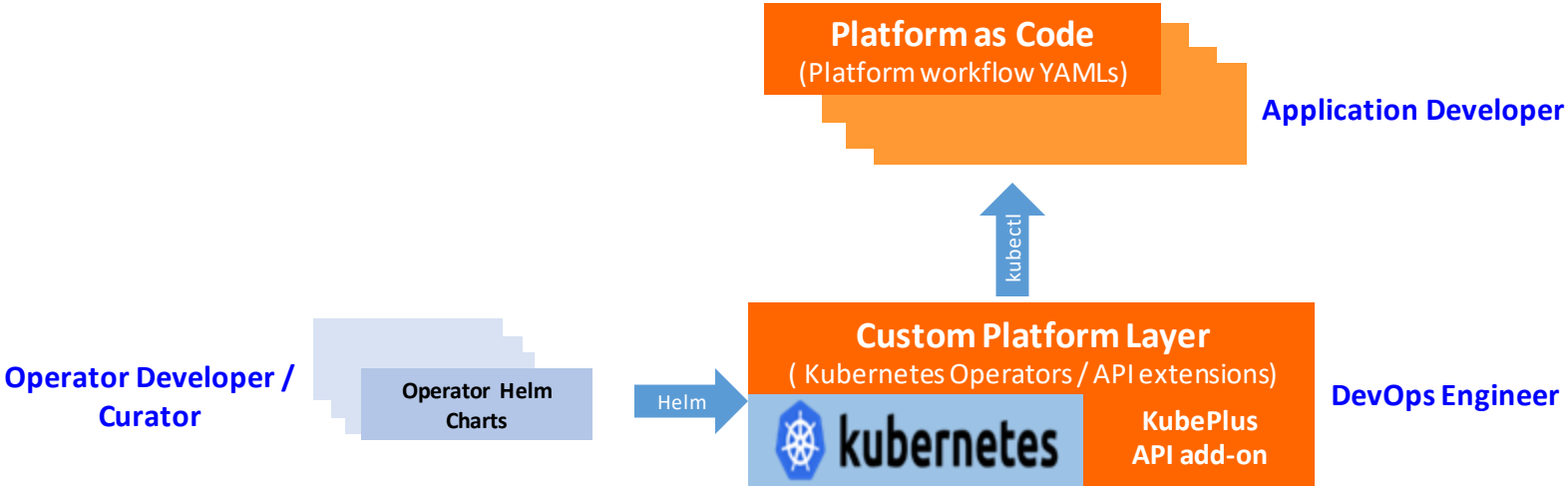
- How to use Custom Resources
- Any code-level assumptions made by an Operator developer
- Composition tree of Kubernetes built-in Resources created by a Custom Resource

Add Platform-as-Code annotations on your CRD YAML

Why? - Enables discovery and use of Custom Resources

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: moodles.moodlecontroller.kubeplus
  annotations:
    helm.sh/hook: crd-install
    platform-as-code/usage: moodle-operator-usage.usage
    platform-as-code/composition: Deployment, Service, PersistentVolume, PersistentVolumeClaim, Secret, Ingress
```

Platform-as-Code & Declarative Application Management (DAM)



Platform-as-Code == Declarative Application Management involving built-in + Custom Resources

- Declarative Application Management in Kubernetes Community Document
- Platform-as-Code

DAM for Custom Resources

CloudARK

Declarative Application Management of Custom Resources

- **Discovery** of Custom Resources
  - What are the available Custom Resources on our cluster?
  - What are the supported actions / workflows by these Custom Resources?
  - What are the underlying Kubernetes Resources created by the Custom Resources?

- **Binding** between Custom Resources to define a workflow
  - Bind CR1 to CR2 based on underlying Resource created by CR2
  - Bind CR1 to CR2 based on Spec-attribute value of CR2
  - Bind CR1 and CR2 based on some label

These requirements are not completely satisfied by existing tools like kubectl or Helm.

Gaps in Custom Resource DAM

CloudARK

DAM Requirements	Base Kubernetes		Helm		Gaps
Discovery of Custom Resources	# kubectl explain		-		Additional guidance to use Custom Resources beyond Spec Properties
Binding and resolution between Custom Resources	Focus on resolution based on statically defined Names and Labels in Kubernetes YAML	+	Focus on resolution based on statically defined values in Values.yaml	+	Binding between Custom Resources with run-time resolution during stack creation

KubePlus API add-on components

CloudARK

New tool to fill the gaps – KubePlus API add-on

Goal: Simplify use of Custom Resources for application developer

Design principles: No new CLI, **Complement other packaging / configuration tools** like Helm / Kustomize

KubePlus API add-on		
Aggregated API server	Mutating webhook	PlatformStack CRD
Discovery of Custom Resources through API endpoints	***Binding*** and ***resolution*** of Custom Resources	Describe ***platform stack dependencies***

https://github.com/cloud-ark/kubeplus

Custom Resource DAM with KubePlus

CloudARK

New tool to fill the gaps – KubePlus API add-on

Goal: Simplify use of Custom Resources for application developer

Design principles: No new CLI, **Complement other packaging / configuration tools** like Helm / Kustomize

DAM Requirements	Base Kubernetes		Helm		KubePlus API add-on
Discovery of Custom Resources	# kubectl explain	+	-	+	New endpoints through 'kubectl' that provide static and dynamic information about Custom Resources
Binding and resolution between Custom Resources	Focus on resolution based on statically defined Names and Labels in Kubernetes YAML		Focus on resolution based on statically defined values in Values.yaml		Focus on run-time resolution based on functionally defined dependencies between Custom Resources in Kubernetes YAML

Example scenario

CloudARK

Operators installed: MySQL & Moodle (Open-source eLearning application)

Step 1: Create MysqlCluster instance

cluster1.yaml

```
apiVersion: mysql.presslabs.org/v1alpha1
kind: MysqlCluster
metadata:
  name: cluster1
  namespace: namespace1
spec:
  replicas: 1
  secretName: cluster1-secret
```

Step 2: Create Moodle instance

moodle1.yaml

```
apiVersion: moodlecontroller.kubeplus/v1
kind: Moodle
metadata:
  name: moodle1
  namespace: namespace1
spec:
  plugins: ["profilecohort"]
  mySQLServiceName: cluster1-mysql-master
  mySQLUserName: root
  mySQLUserPassword: cluster1-secret.ROOT_PASSWORD
  moodleAdminEmail: test@test.com
```

Discovery: How does a developer know name of the Service created by the cluster1 resource?

Binding: Can the binding between moodle1 and cluster1 be resolved automatically?

Custom Resource Discovery

CloudARK

New endpoints to learn static and dynamic information about Custom Resources

KubePlus component - Aggregated API server

For additional information beyond kubectl explain

Static information – man page for supported operations, assumptions etc.

```
kubectl get --raw "/apis/platform-as-code/v1/man"
```

Dynamic information – composition tree

```
kubectl get --raw "/apis/platform-as-code/v1/composition"
```

How to discover name of the Service created by the *cluster1* resource to be given in *moodle1.yaml*?

CloudARK

```
devdatta:moodle-mysql$ kubectl get --raw "/apis/platform-as-code/v1/composition?kind=MysqlCluster&instance=cluster1" | python -mjson.tool
[
    {
        "Children": [
            {
                "Children": [
                    {
                        "Children": [],
                        "Kind": "Pod",
                        "Level": 3,
                        "Name": "cluster1-mysql-0",
                        "Namespace": "default",
                        "Status": "Running"
                    }
                ],
                "Kind": "StatefulSet",
                "Level": 2,
                "Name": "cluster1-mysql",
                "Namespace": "default",
                "Status": ""
            },
            {
                "Children": [],
                "Kind": "Service",
                "Level": 2,
                "Name": "cluster1-mysql",
                "Namespace": "default",
                "Status": ""
            },
            {
                "Children": [],
                "Kind": "Service",
                "Level": 2,
                "Name": "cluster1-mysql-master",
                "Namespace": "default",
                "Status": ""
            },
```

Custom Resource Binding

CloudARK

New functional constructs to be used in YAML definition to bind Custom Resources

KubePlus component – Mutating webhook

Existing technologies offer resolution based on statically defined values.
Need for run-time resolution of dependencies.

Import Value - Imports value of the specified parameter into the Spec where the function is defined

```
Fn::ImportValue(<Parameter>)
```

Add label - Adds the specified label to the specified resource

```
Fn::AddLabel(label, <Resource>)
```

Run-time **resolution** (Post kubectl apply) for Custom Resource Spec Properties & sub-resources

How to create automatic binding between *moodle1* and *cluster1*?

CloudARK

Creating Moodle instance

moodle1.yaml

```
apiVersion: moodlecontroller.kubeplus/v1
kind: Moodle
metadata:
  name: moodle1
  labels:
    stack: moodle1-stack
spec:
  plugins: ["profilecohort"]
  mySQLServiceName: Fn::ImportValue(MysqlCluster:default.cluster1:Service(filter=master))
  mySQLUserName: root
  mySQLUserPassword: cluster1-secret.ROOT_PASSWORD
  moodleAdminEmail: test@test.com
```

Summary

CloudARK

- Create Helm Chart for your Operator following the presented guidelines.
- Declarative Application Management for Custom Resources requires special attention.
- Explore KubePlus API add-on alongside Helm to get the best Platform-as-Code experience.

Thanks!

My co-ordinates

devdatta@cloudark.io
Twitter: @devkulkarni_
Github: devdattakulkarni

CloudARK co-ordinates

https://cloudark.io/
https://github.com/cloud-ark/kubeplus
Twitter: @cloudarknews
Slack

Sign-up to get free copy of your Platform-as-Code e-Book.