Version 4
04/18/21

Provided by Tech Skills Transformations LLC

Prerequisites

- Modern system with at least 8G of memory and 20G free storage
- VirtualBox installed and running
  http://www.virtualbox.org

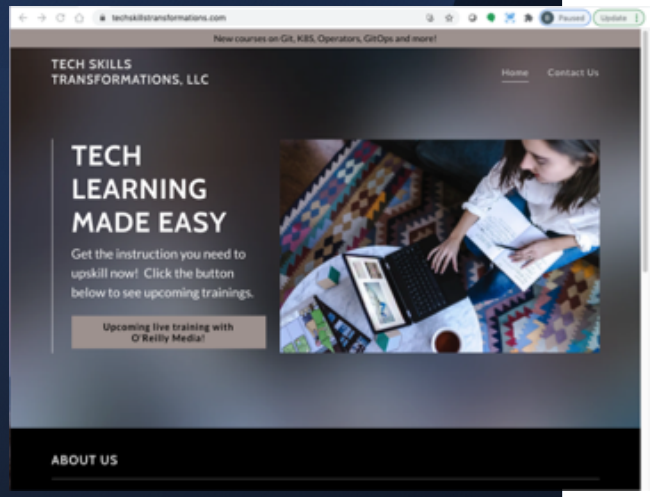
- Virtual machine (.ova file) installed in VirtualBox
  https://www.dropbox.com/s/konq5fctnalu9lz/k8s-op.ova?dl=0
  OR
  https://bclconf.s3-us-west-2.amazonaws.com/k8s-op.ova

- Workshop docs are in https://github.com/skilldocs/k8s-op
- Setup doc is at
  https://github.com/skilldocs/k8s-op/blob/main/k8s-op-setup.pdf
- Labs doc for workshop
  https://github.com/skilldocs/k8s-op/blob/main/k8s-op-labs.pdf

Building a Kubernetes Operator: Extending Kubernetes to Fit your Applications

Brent Laster

About me



- Founder, Tech Skills Transformations LLC
- R&D DevOps Director
- Global trainer – training (Git, Jenkins, Gradle, CI/CD, pipelines, Kubernetes, Helm, ArgoCD, operators)
- Author -
  - OpenSource.com
  - Professional Git book
  - Jenkins 2 – Up and Running book
  - Continuous Integration vs. Continuous Delivery vs. Continuous Deployment mini-book on Safari
- https://www.linkedin.com/in/brentlaster
- @BrentCLaster
- GitHub: brentlaster

Professional Git Book

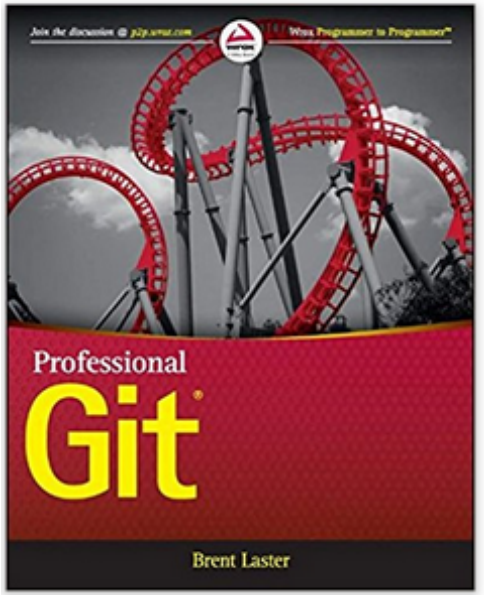
- Extensive Git reference, explanations,
- and examples
- First part for non-technical
- Beginner and advanced reference
- Hands-on labs

Professional Git 1st Edition

by Brent Laster (Author)

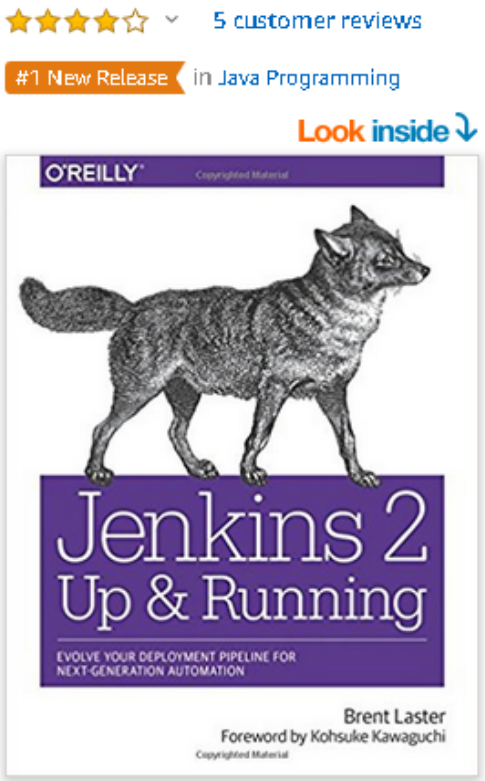
7 customer reviews

Look inside



Jenkins 2 Book

- Jenkins 2 – Up and Running
- "It's an ideal book for those who are new to CI/CD, as well as those who have been using Jenkins for many years. This book will help you discover and rediscover Jenkins." ***By Kohsuke Kawaguchi, Creator of Jenkins***



O'Reilly Training

April 26 & 27, 2021

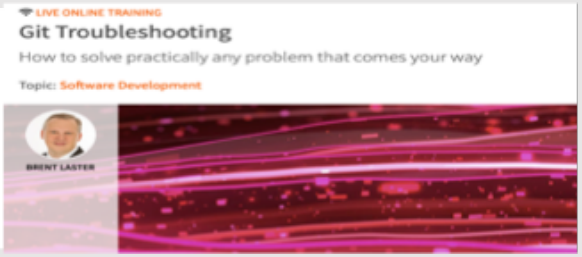
Git Fundamentals

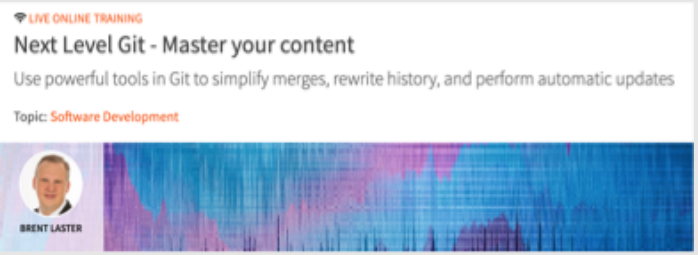
Join Brent Laster to gain the knowledge and skills you need to leverage Git to greatly simplify and speed up managing all of the changes in your source code. Once you ...

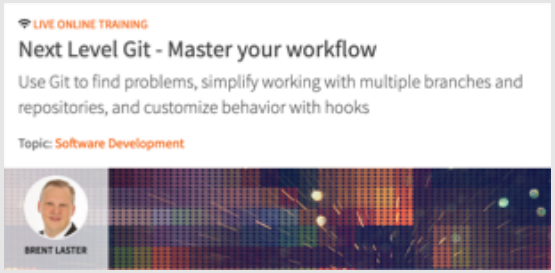
June 3, 10, 17 & 24, 2021

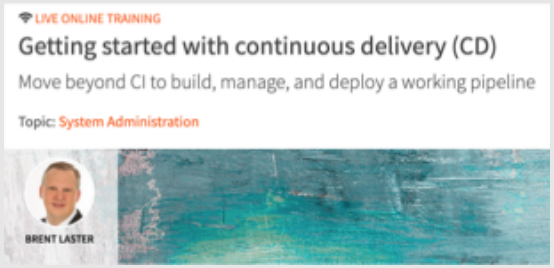
Git in 4 Weeks

Join author, trainer, and DevOps director Brent Laster to learn how Git works—and discover all the things you can do with it. Over four sessions, Brent walks you through everything you ...





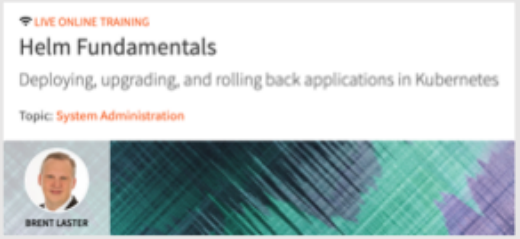


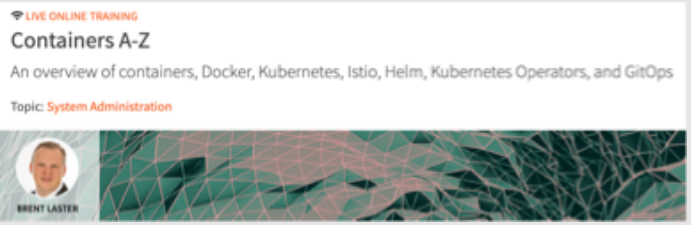


June 28, 2021

Troubleshooting Kubernetes

In this 3-hour course, global trainer, author, and DevOps director Brent Laster will show you how to respond to the most common problem situations you may encounter with Kubernetes. You'll learn ...





May 24, 2021

Continuous Delivery in Kubernetes with ArgoCD

Join expert Brent Laster to explore GitOps and learn how to use Argo CD to implement GitOps in your Kubernetes deployments. APAC time friendly - You're a Kubernetes admin who wants ...

May 17, 2021

Building a Kubernetes Operator

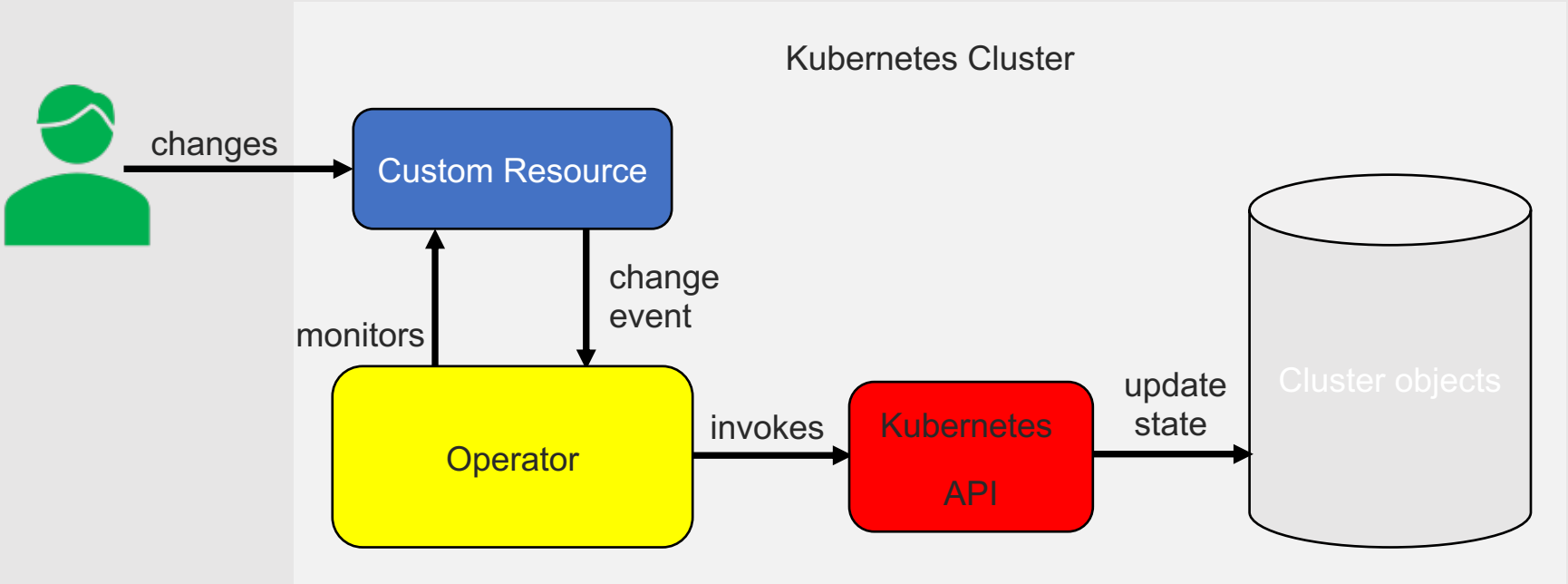
Join expert Brent Laster to learn how the Operator pattern helps address these kinds of situations by allowing you to create custom controllers that extend the functionality of the Kubernetes API ...

Agenda

- Introduction to Operators
- The Operator Framework and SDK
- Custom Resources
- All about Controllers
- Permissions and RBAC
- Initial Testing
- Packaging and Deploying the Operator in a Cluster

What is an Operator?

- A pattern for software extension to Kubernetes
- A way to package, deploy, and manage an application as K8s-native
  - K8s-native application - application deployed on K8s and managed via K8s-api and K8s tooling
- Application that watches a custom resource (new Kind)
- Executes some defined operations when custom resource changes



From a human perspective...

- A design pattern
  - Original blog by CoreOS from 2016 - https://coreos.com/blog/introducing-operators.html
- Encapsulate operational knowledge in code
  - Integrates with Kubernetes and its API
  - “Kubernetes-native”
- Similar to an SRE role
- Automate/implement Day-1 activities
  - Installation, configuration, etc.
- Automate/implement Day-2 activities
  - Update, backup, failover, restore, re-configuration, etc.
- Manages applications that need to persist state (stateful)

Stateful vs Stateless

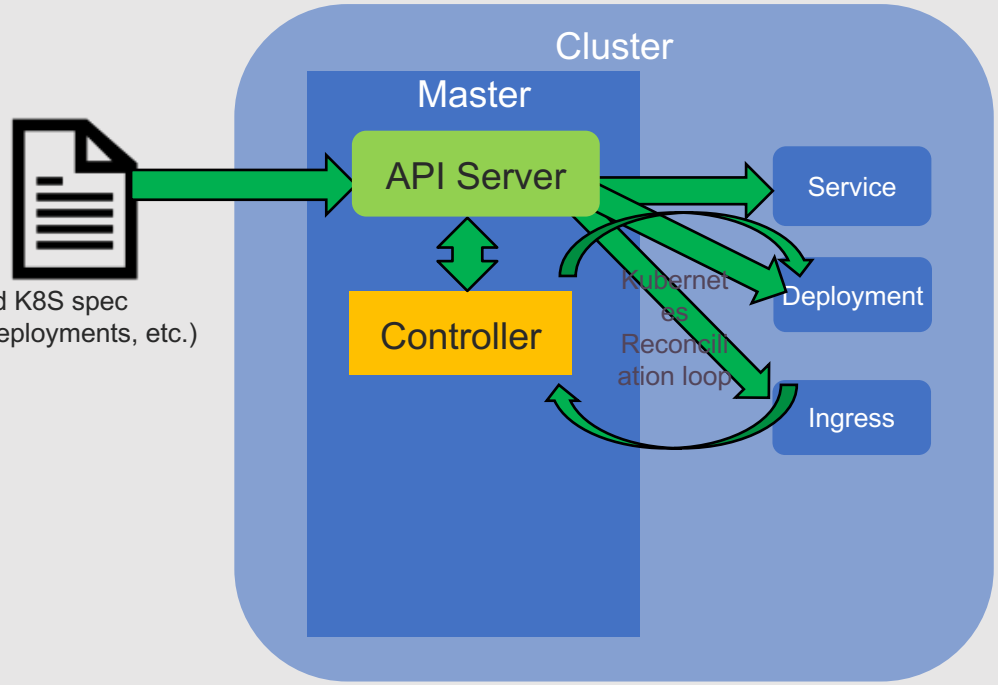
- Native k8s objects (deployments, services, etc.) are designed to work well with the standard k8s controller model
- Considered “stateless” – don’t need to persist/rely on state
- That idea has issues for more complex, stateful applications
  - Example: database running on multiple nodes
  - If nodes go down, need to reload data to get back to previous good state
- Usual cloud friendly operations like scaling, upgrading, disaster recovery for stateful app don’t fit

How are Operators different from other tools?

- Purpose-built to run a Kubernetes application, with operational knowledge baked in
- “Smarter” and more tailored than generic tools
- Cloud-like capabilities encoded into Operator code can provide an advanced user experience
- Can automate features like updates, backups and scaling
- All of this is accomplished using standard Kubernetes tools, CLI and API
- An operator is essentially like a custom controller

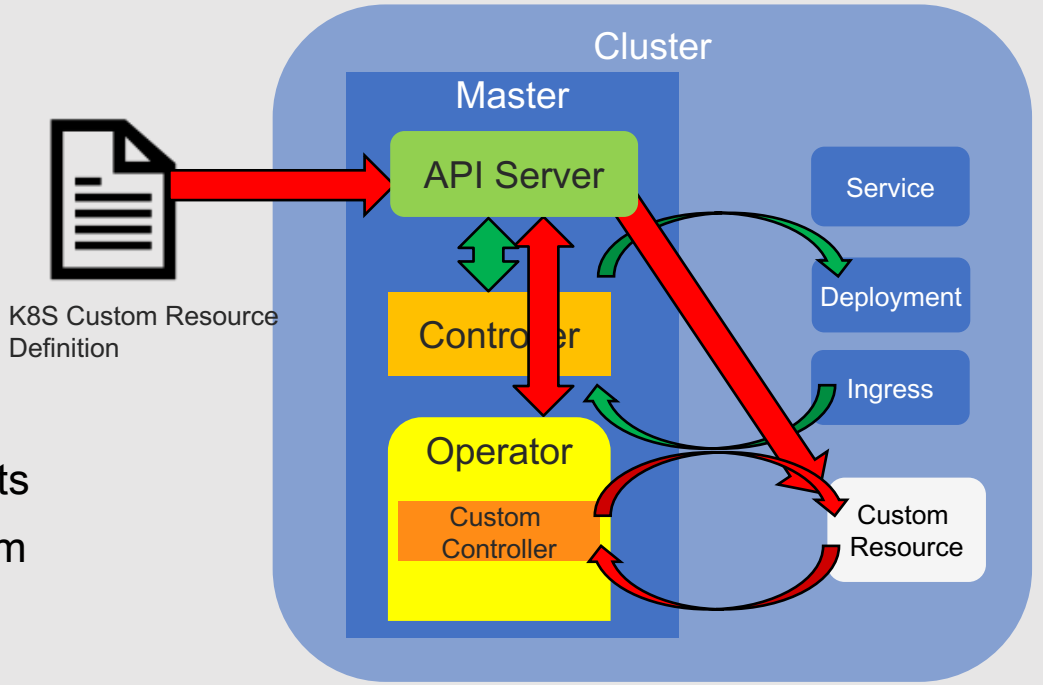
What is a controller?

- Core part of Kubernetes
- Software loop that runs continuously on master node (in “control plane”)
  - Compares desired state and live/current state of an “object”
  - Objects are Kubernetes resources like Pods, ConfigMaps, Services, etc.
  - If differences in desired and live, controller reconciles to get to desired



How Do Operators Work?

- Define a custom controller to watch your app and do custom processing based on its state
- Application that is to be watched by operator is defined as new object type in k8s - custom resource (CR)
  - Has its own yaml spec and object type CustomResource
  - Format understood by the API server
- Operators run and oversee a custom controller loop
- Overall behavior of an operator's controller reconciling state for CR is similar to way native k8s handles reconciling native objects
- Operators give you the chance to do custom processing when reconciling



What can an Operator do?

- Simple to complex operations
- Operations on custom types
- Help manage stateful types
- Backup/restore
- Deploying clusters for a particular type
- Upgrading versions
- Additional logging
- Replication
- Creating additional resources
- Note: Could write operator to manage software that is not your own - if it doesn't already exist

When do you use an operator?

- Operators vs controllers
  - All operators are controllers but not all controllers are operators
  - To be considered as an operator, there must be application-specific domain knowledge built in
  - Able to perform automated tasks on behalf of user

- Operators are appropriate when
  - You need to encapsulate approach(es) for business logic around stateful applications (using Kubernetes API). Incorporates automation in Kubernetes for custom object.
  - You need to have an automated mechanism to watch your application for changes and respond with certain actions when certain events happen.

- Other alternative implementations may be appropriate/better

Alternatives to Operators*

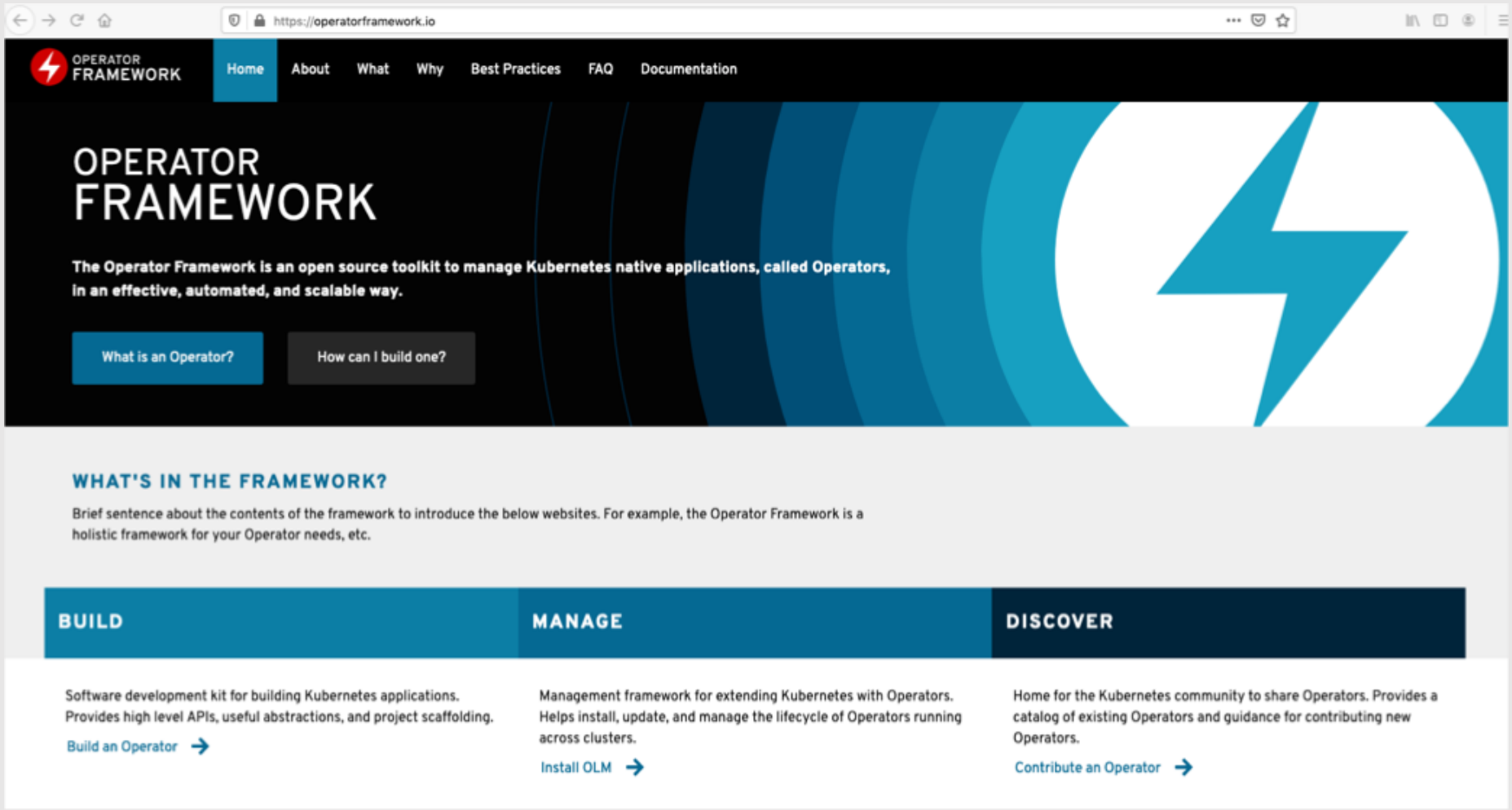
- Outside of cluster
  - Shell scripts
  - Ansible automation
- Inside of cluster
  - Jobs
  - Custom controllers

* Often can be advisable to implement logic first in a simpler way to vet need for an operator

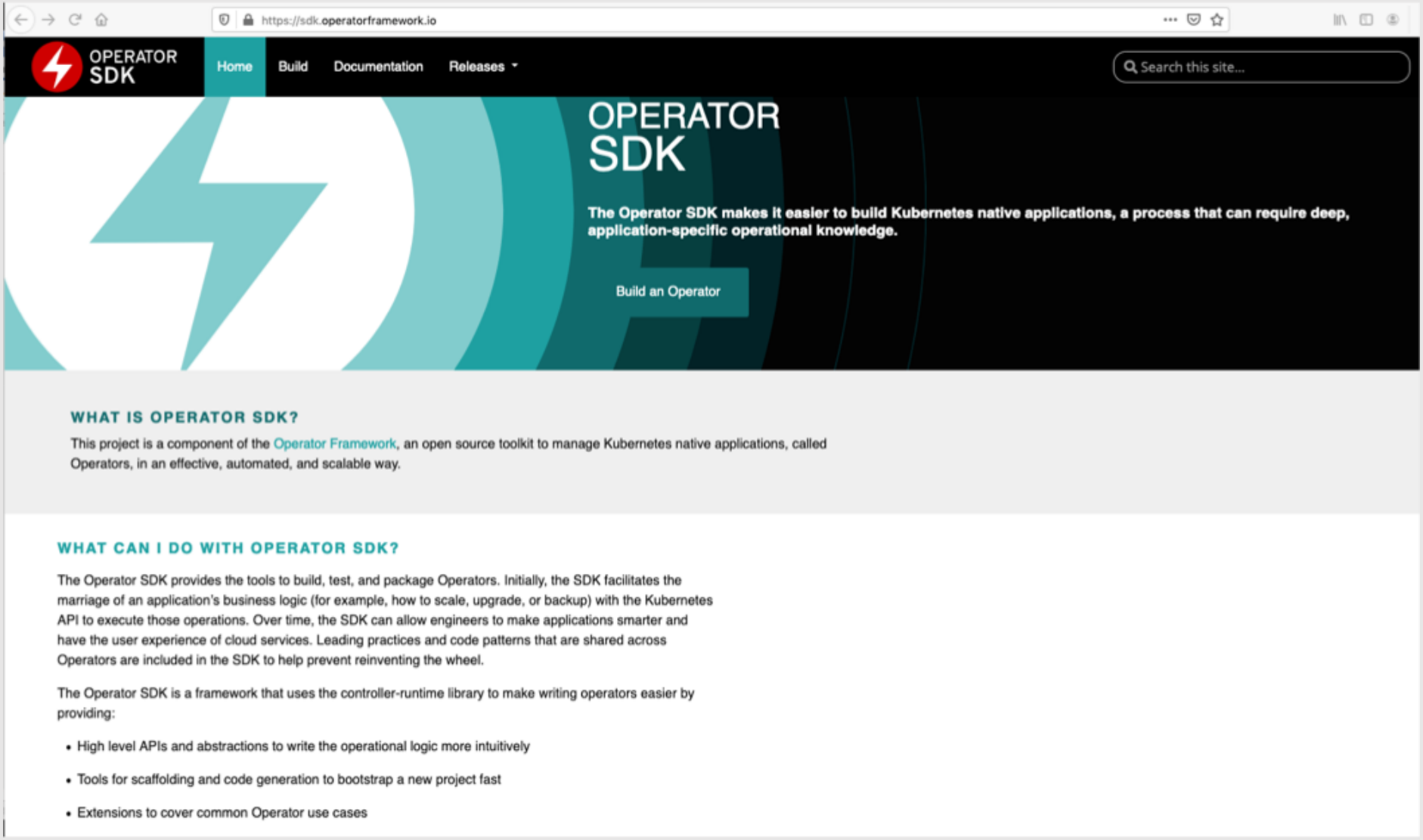
Writing an Operator

- Technically, could write it in any language/framework that can work with K8s API
- General approach
  - Code the logic for the operator – what do you want to do and how
  - Validate that an operator implementation makes sense!
    - » Are you compensating for architecture that needs to be changed?
  - Containerize the operator and make the image available in a registry
  - Create the Custom Resource Definition (CRD) – defines the custom resource
  - Create the RBAC definitions that will allow the operator to connect and work with resources in Kubernetes (via the API)
  - Package and deploy the operator into a cluster
  - Create the Custom Resource for it to monitor and respond to (based on the definition/structure provided by the CRD)
- Don’t do everything from scratch - use available frameworks/tools

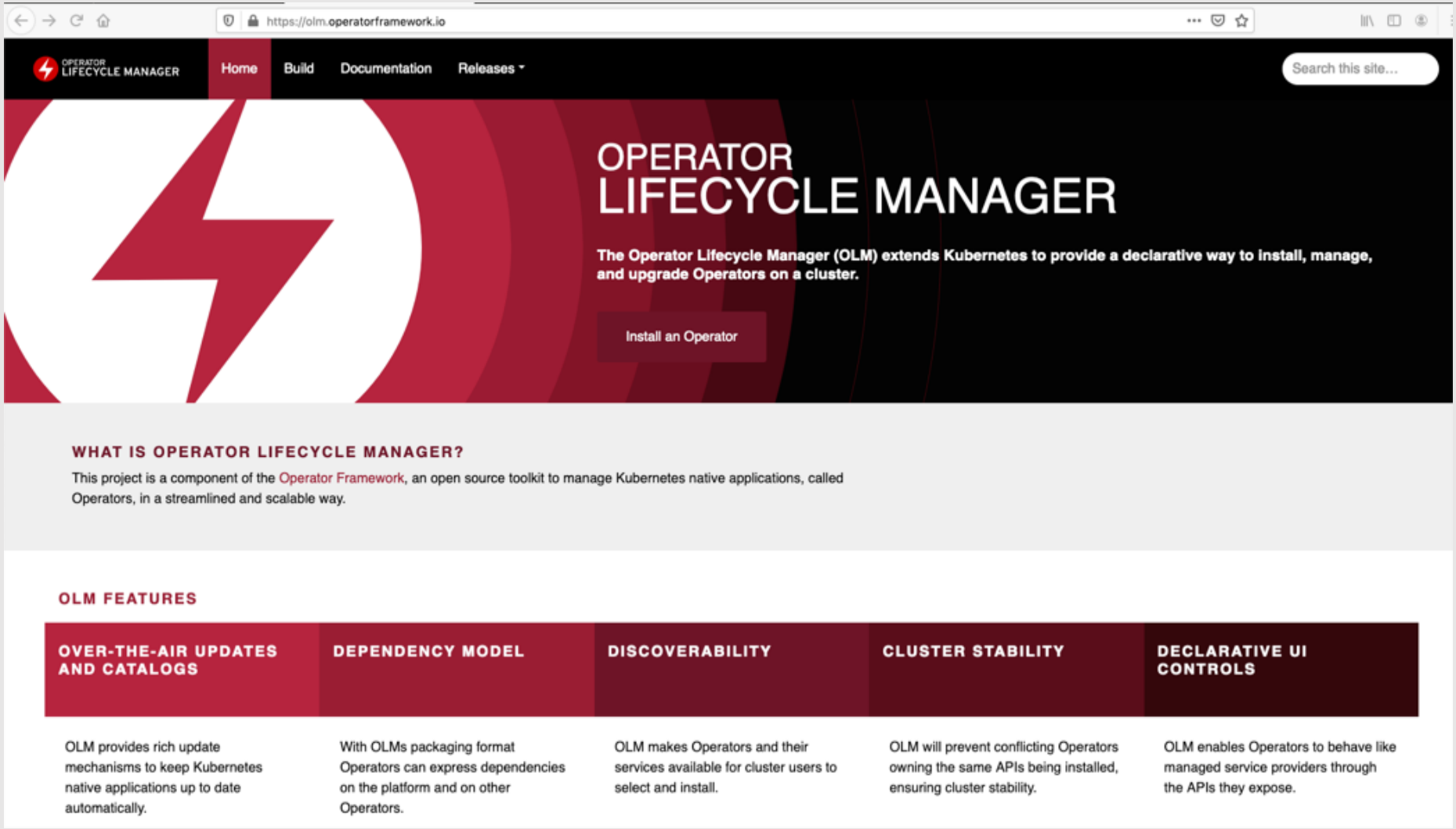
Operator Framework



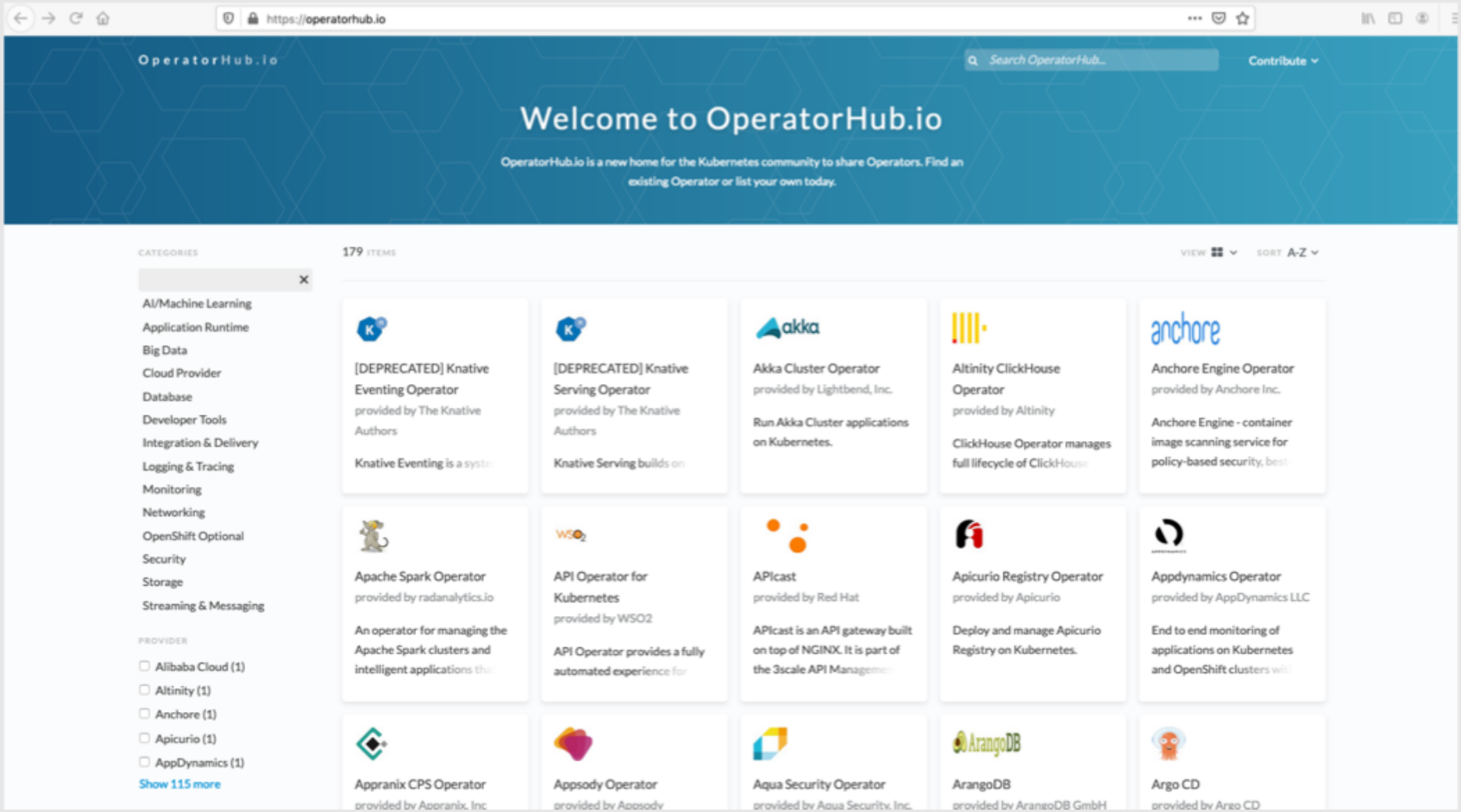
Operator SDK



Operator Lifecycle Manager (OLM)



OperatorHub.io



Alternative Frameworks

- KOPF - Kubernetes Operator Pythonic Framework
  - Implement in Python in operator.py file
  - Must manually create all boilerplate files - crd, role, rolebinding, etc.
- Java Operator SDK
  - Does some scaffolding
  - Implement in Java
  - Still fairly new
- Java or other language
  - Requires writing more code
  - Have to implement all files
  - Single programming language
- Also Kubebuilder, but now incorporated in SDK

Operator SDK vs kubebuilder

- Formerly separate approaches for building operators
- kubebuilder
  - a kubernetes-sig (part of API Machinery SIG)
  - out of Google
- operator-sdk
  - a part of the operator-framework - CNCF (Cloud Native Computing Foundation) project
  - originated by CoreOS and incorporated/managed now by Red Hat
- SDK has adopted kubebuilder approach and mechanisms
- SDK is “consumer” of kubebuilder
- SDK refers to kubebuilder documentation
- SDK maintainers contribute back to kubebuilder code
- Beyond kubebuilder functionality, SDK contains functionality for Operator Lifecycle Management

General Operator-SDK (kubebuilder) flow

- Prereqs - Go, Docker, kubectl, cluster access
- Install operator-sdk
- Create a project
- Create an API
- Test it out
- Install instances of Custom Resources
- Run it on the Cluster

Operator-SDK Languages - Go

- Go - general-purpose programming
  - can code any logic
  - good match for Kubernetes api client library since Kubernetes is written in Go
  - scaffold Go project
    - `$ operator-sdk init --repo=<Git repo>`
    - `$ operator-sdk create api --group <name> --version <version> --kind <kind> --resource=true --controller=true`
  - Add logic for operator into Reconcile function in controllers/<name>_controller.go
  - Add default parameters into api/v1/<name>_types.go
- Notes:
  - Imperative approach - more work required and more complicated
  - Controller code is scaffolded but not sufficient
  - Go is well integrated with K8s
  - Essentially can do any logic without limits
  - Operator configuration files (role, role-binding, crd, etc.) are automatically generated

Workflow

- Use sdk to create scaffold project ($ operator-sdk init)
- Create scaffold API , CRD, and controller ($ operator-sdk create api)
- Update API to have necessary fields for spec and status ($ edit *_types.go)
- After above, automatically update generated code for resource - updates interface/implementation for runtime.Object to be a valid Kubernetes Kind ($ make generate)
  - updates api/<version>/zz_generated.deepcopy.go
- Generate CRD manifests ($ make manifests)
  - creates config/crd/bases/<item>.yaml
  - creates config/rbac/role.yaml
- Update scaffolded controller code to complete controller functionality
  - Add needed packages and core functions from Kubernetes
  - Add code to reconcile desired state of CR against observed state
  - Add code to create any needed K8s objects (Kinds)
  - Add any additional markers to manage items such as permissions/generate manifests
  - updates controllers/<item>_controller.go
- Update CRD manifests ($ make manifests)
  - updates config/crd/bases/<item>.yaml
  - updates config/rbac/role.yaml
- Execute install target to register CRD with K8s apiserver ($ make install)
- Test the operator ($ make run)
- Create the docker package ($ make docker-build)
- Push image to registry ($ make docker-push)
- Deploy operator to cluster ($ make deploy)

Lab 1 – Scaffolding an Operator with the Operator SDK

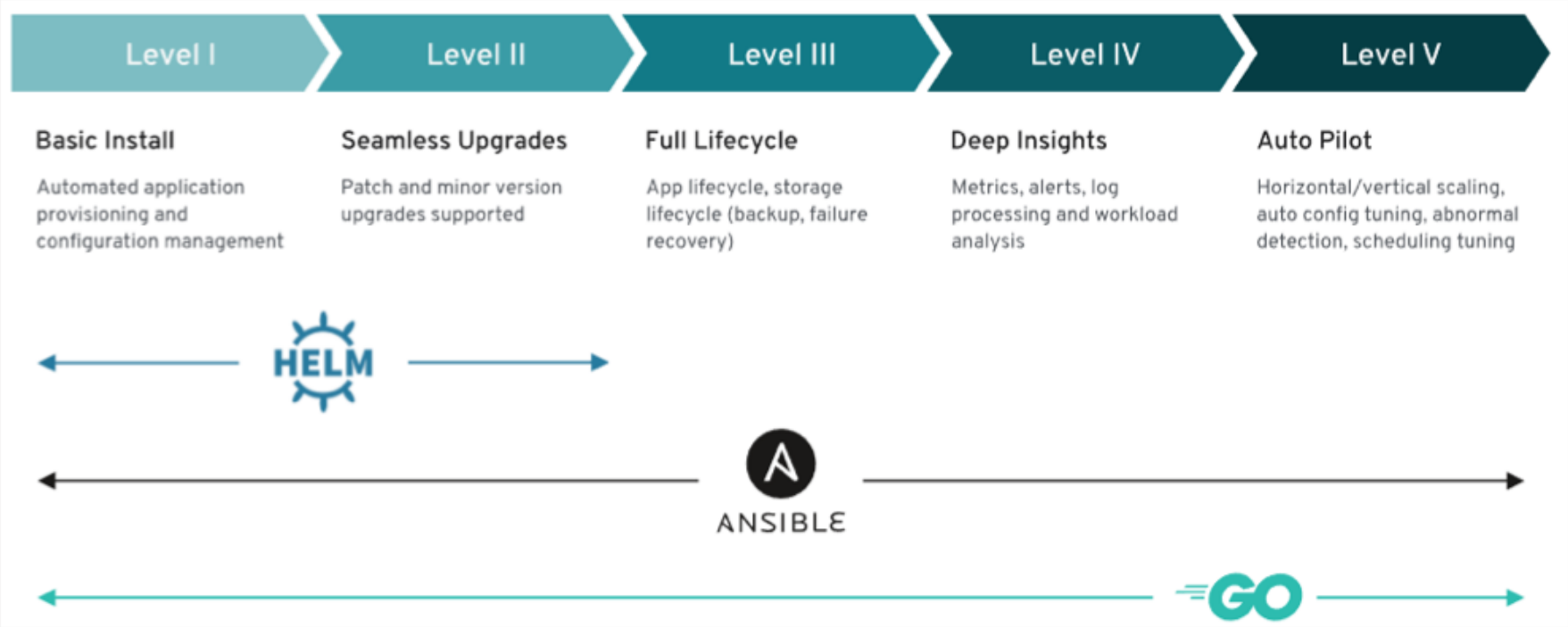
Comparison of Languages for Operators using Operator-SDK

- source: operatorhub.io

Operator Type	What the SDK generates	What you need to define
Go Operator	• General go program structure • Boilerplate code to talk to the Kubernetes API • Boilerplate code to watch for Kubernetes objects of interest • An entry point to the reconciliation loop • An example YAML files based on CRDs	• Custom objects via CRDs • Control loop logic in Go • Potentially artistic stunts only possible by talking directly to the API from Go
Ansible Operator	• A Go program that runs an Ansible playbook or role every time a certain type of object is detected / modified	• Ansible playbook or role • Custom objects via CRD
Helm Operator	• A custom object via CRD containing the same properties as the chart's values.yaml • A Go program that reads a helm chart and deploys all its resources • Watch statements to detect changes in the custom objects specification, re-deploying all resources with updated values	• The location / repository of the helm chart

Operator Maturity Model

- source: operatorhub.io



Parts of the project

- Build infrastructure
  - go.mod - Go module for project with basic dependencies
  - Makefile - make targets for building and deploying the controller
  - PROJECT - metadata for new component scaffolds
- Launch configuration
  - Initially, just has Kustomize YAML to launch controller on cluster
  - After starting to write controller, has
    - » Custom Resource Definitions
    - » RBAC config
    - » Webhook config
  - config/default - Kustomize base for launching controller (standard config)
  - config/manager - Kustomize base to launch controller as pods in cluster
  - config/rbac - Kustomize base for service account to run controllers as
- Entrypoint - main.go

Initializing a project w/ operator-sdk init

- **bin/manager** - executable to wrap 1 or more controllers
- **config/** - launch configurations
  - initially just contains Kustomize YAML definitions needed to launch controller in cluster
  - eventually will also contain CRD, RBAC configs, WebhookConfigs
- **config/certmanager** - files for K8s add-on (cert-manager) that automates management and issuance of TLS certs from various sources
- **config/default** - Kustomize base to launch controller in standard config
- **config/manager** - launch controllers as pods in cluster
- **config/prometheus** - patch file to enable prometheus metrics via ServiceMonitor
- **config/rbac** - permissions req'd to run controllers under their own SA
- **config/webhook** - for deploying admission webhooks
- **Dockerfile** - for building the manager binary
- **go.mod** - project dependencies
- **go.sum** - crytographic checksums of specific module versions
- **hack/boilerplate.go.txt** - Apache 2 license file example
- **main.go** - basic entrypoint for project
- **Makefile** - Make targets to build and deploy controller and more
- **PROJECT** - Kubebuilder metadata for scaffolding new components

```
diyuser3@training1:~/ops$ tree
.
├── bin
│   └── manager
├── config
│   ├── certmanager
│   │   ├── certificate.yaml
│   │   ├── kustomization.yaml
│   │   └── kustomizeconfig.yaml
│   ├── default
│   │   ├── kustomization.yaml
│   │   ├── manager_auth_proxy_patch.yaml
│   │   ├── manager_webhook_patch.yaml
│   │   └── webhookcainjection_patch.yaml
│   ├── manager
│   │   ├── kustomization.yaml
│   │   └── manager.yaml
│   ├── prometheus
│   │   ├── kustomization.yaml
│   │   └── monitor.yaml
│   ├── rbac
│   │   ├── auth_proxy_client_clusterrole.yaml
│   │   ├── auth_proxy_role_binding.yaml
│   │   ├── auth_proxy_role.yaml
│   │   ├── auth_proxy_service.yaml
│   │   ├── kustomization.yaml
│   │   ├── leader_election_role_binding.yaml
│   │   ├── leader_election_role.yaml
│   │   └── role_binding.yaml
│   └── webhook
│       ├── kustomization.yaml
│       ├── kustomizeconfig.yaml
│       └── service.yaml
├── Dockerfile
├── go.mod
├── go.sum
├── hack
│   └── boilerplate.go.txt
├── main.go
├── Makefile
└── PROJECT
```

Groups and Versions and Kinds

- https://book.kubebuilder.io/cronjob-tutorial/gvks.html
- API Group in K8s
  - collection of related functionality
  - one or more versions - denotes version of API over time
- Kinds
  - API types contained in group-version
  - Must be backwards compatible with previous API versions
- Resource
  - Use of a Kind in the API
- GVK
  - GroupVersionKind
  - corresponds to given root Go type in package
- Scheme
  - tracks Go type to GVK

Spec and Status

- Every running k8s object includes two nested object fields
  - spec
    - User-provided
    - Describes desired state for the object
  - status
    - Provided and updated by k8s system
    - Describes the actual state of the object

```
diyuser3@training1:~$ kubectl edit -n roar2 pod roar-web-74bb47bdb8-56l4n
```

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2019-07-21T20:04:34Z"
  generateName: roar-web-74bb47bdb8-
  labels:
    app: roar-web
    pod-template-hash: 74bb47bdb8
  name: roar-web-74bb47bdb8-56l4n
  namespace: roar2
  ownerReferences:
  - apiVersion: apps/v1
    blockOwnerDeletion: true
    controller: true
    kind: ReplicaSet
    name: roar-web-74bb47bdb8
    uid: 89d00855-abf0-11e9-b30c-080027c4188a
  resourceVersion: "346085"
  selfLink: /api/v1/namespaces/roar2/pods/roar-w
  uid: c2b50783-abf2-11e9-b30c-080027c4188a
spec:
  containers:
  - image: localhost:5000/roar-web:v2
    imagePullPolicy: Always
    name: roar-web
    ports:
    - containerPort: 8080
      name: web
      protocol: TCP
```

```
    secret:
      defaultMode: 420
      secretName: default-token-dxx7t
status:
  conditions:
  - lastProbeTime: null
    lastTransitionTime: "2019-07-21T20:04:
    status: "True"
    type: Initialized
  - lastProbeTime: null
    lastTransitionTime: "2019-07-21T20:04:
    status: "True"
    type: Ready
  - lastProbeTime: null
    lastTransitionTime: "2019-07-21T20:04:
    status: "True"
    type: ContainersReady
  - lastProbeTime: null
    lastTransitionTime: "2019-07-21T20:04:
    status: "True"
    type: PodScheduled
  containerStatuses:
  - containerID: docker://41c2d31ef46b8ea1
    image: localhost:5000/roar-web:v2
    imageID: docker-pullable://localhost:5
    lastState:
      terminated:
        containerID: docker://c0ddd9686971
        exitCode: 143
```

operator-sdk create api

- scaffolds new Kind and related controller
- expects group, version, and kind
- prompts to create resource & controller
- in api/v1alpha1
  - new file for Kind - roarapp_types.go
  - groupversion_info.go - common metadata about group-version
  - zz_generated.deepcopy.go
    - » contains automatically generated code of runtime.Object interface
    - » needed for K8s to recognize as a Kind
- in controllers
  - roarapp_controller.go - scaffolded source for controller
  - suite_test.go - does the bare bones of setting up a test environment

```
diyuser3@training1:~/ops$ tree
.
├── api
│   └── v1alpha1
│       ├── groupversion_info.go
│       ├── roarapp_types.go
│       └── zz_generated.deepcopy.go
```

```
├── controllers
│   ├── roarapp_controller.go
│   └── suite_test.go
```

What's in an API? (_types.go)

- License & imports
- spec
  - holds desired state
  - "inputs" to controller
  - for our app
    - » number of replicas
    - » image for webapp
    - » image for backend db
- status
  - holds observed state
  - info that we want users or other controllers to have access to
- schema
  - defines type and list of type
  - TypeMeta - describes API version & Kind
  - ObjectMeta - holds items like name, namespace, labels
- init
  - adds Go types to API group
  - allows adding types in this group to any scheme

```go
// RoarAppSpec defines the desired state of RoarApp
type RoarAppSpec struct {
        // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
        // Important: Run "make" to regenerate code after modifying this file
        Replicas int32  `json:"replicas"`
        WebImage string `json:"webImage,omitempty"`
        DbImage  string `json:"dbImage,omitempty"`
}
```

```go
// RoarAppStatus defines the observed state of RoarApp
type RoarAppStatus struct {
        // INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
        // Important: Run "make" to regenerate code after modifying this file
        PodNames []string `json:"podNames"`
}
```

```go
// RoarApp is the Schema for the roarapps API
type RoarApp struct {
        metav1.TypeMeta   `json:",inline"`
        metav1.ObjectMeta `json:"metadata,omitempty"`

        Spec   RoarAppSpec   `json:"spec,omitempty"`
        Status RoarAppStatus `json:"status,omitempty"`
}
```

```go
// +kubebuilder:object:root=true

// RoarAppList contains a list of RoarApp
type RoarAppList struct {
        metav1.TypeMeta `json:",inline"`
        metav1.ListMeta `json:"metadata,omitempty"`
        Items           []RoarApp `json:"items"`
}
```

// +kubebuilder - marker - tells object generator that this type is a Kind

```go
func init() {
        SchemeBuilder.Register(&RoarApp{}, &RoarAppList{})
}
```

Scheme - methods for serializing, converts to/from Go schemas

Custom Resource

- Background
  - In k8s, a resource is an endpoint in the API that stores groups of API objects of a specific kind
  - Objects can be managed by kubectl
- A new kind of object (other than standard ones such as Pod, Service, etc.) targeted around a resource you want to run and manage in Kubernetes
- Typically requires special handling to respond to Kubernetes operations (may be stateful) such as scaling, deleting, etc.
- Actually 2 methods to add these -
  - Custom Resource Definitions (CRDs) - easier to implement/share
  - Aggregated API - more flexible, requires programming

Makefile

- Convenient way to wrap operations
- Used for SDK Go process
- Leverages tools like controller-gen and kustomize

```
# Current Operator version
VERSION ?= 0.0.1
# Default bundle image tag
BUNDLE_IMG ?= controller-bundle:$(VERSION)
# Options for 'bundle-build'
ifneq ($(origin CHANNELS), undefined)
BUNDLE_CHANNELS := --channels=$(CHANNELS)
endif
ifneq ($(origin DEFAULT_CHANNEL), undefined)
BUNDLE_DEFAULT_CHANNEL := --default-channel=$(DEFAULT_CHANNEL)
endif
BUNDLE_METADATA_OPTS ?= $(BUNDLE_CHANNELS) $(BUNDLE_DEFAULT_CHANNEL)

# Image URL to use all building/pushing image targets
IMG ?= controller:latest
# Produce CRDs that work back to Kubernetes 1.11 (no version conversion)
CRD_OPTIONS ?= "crd:trivialVersions=true"

# Get the currently used golang install path (in GOPATH/bin, unless GOBIN is set)
ifeq (,$(shell go env GOBIN))
GOBIN=$(shell go env GOPATH)/bin
else
GOBIN=$(shell go env GOBIN)
endif

all: manager

# Run tests
test: generate fmt vet manifests
        go test ./... -coverprofile cover.out

# Build manager binary
manager: generate fmt vet
        go build -o bin/manager main.go

# Run against the configured Kubernetes cluster in ~/.kube/config
run: generate fmt vet manifests
        go run ./main.go

# Install CRDs into a cluster
install: manifests kustomize
        $(KUSTOMIZE) build config/crd | kubectl apply -f -

# Uninstall CRDs from a cluster
uninstall: manifests kustomize
        $(KUSTOMIZE) build config/crd | kubectl delete -f -

# Deploy controller in the configured Kubernetes cluster in ~/.kube/config
deploy: manifests kustomize
        cd config/manager && $(KUSTOMIZE) edit set image controller=${IMG}
        $(KUSTOMIZE) build config/default | kubectl apply -f -

# Generate manifests e.g. CRD, RBAC etc.
manifests: controller-gen
        $(CONTROLLER_GEN) $(CRD_OPTIONS) rbac:roleName=manager-role webhook paths="./..."
```

Workflow

- ✓ Use sdk to create scaffold project ($ operator-sdk init)
- ✓ Create scaffold API , CRD, and controller ($ operator-sdk create api)
- ❑ Update API to have necessary fields for spec and status ($ edit *_types.go)
- ❑ After above, automatically update generated code for resource - updates interface/implementation for runtime.Object to be a valid Kubernetes Kind ($ make generate)
  - ❑ updates api/<version>/zz_generated.deepcopy.go
- ❑ Generate CRD manifests ($ make manifests)
  - ❑ creates config/crd/bases/<item>.yaml
  - ❑ creates config/rbac/role.yaml
- ❑ Update scaffolded controller code to complete controller functionality
  - ❑ Add needed packages and core functions from Kubernetes
  - ❑ Add code to reconcile desired state of CR against observed state
  - ❑ Add code to create any needed K8s objects (Kinds)
  - ❑ Add any additional markers to manage items such as permissions/generate manifests
  - ❑ updates controllers/<item>_controller.go
- ❑ Update CRD manifests ($ make manifests)
  - ❑ updates config/crd/bases/<item>.yaml
  - ❑ updates config/rbac/role.yaml
- ❑ Execute install target to register CRD with K8s apiserver ($ make install)
- ❑ Test the operator ($ make run)
- ❑ Create the docker package ($ make docker-build)
- ❑ Push image to registry ($ make docker-push)
- ❑ Deploy operator to cluster ($ make deploy)

Lab 2 - Defining the API

What happens when we create a CRD?

- Create by defining spec and applying it
- CRD can be either scoped for a namespace or cluster
  - » Per “scope” field
- K8S api server creates a new RESTful resource path for each version you supply
- Creates new RESTful API endpoint
  - » Example:

/apis/stable.example.com/v1/namespaces/*/crontabs/…

- Endpoint URL can then be used to create and manage custom objects
- Manipulate as other objects
- Example Kind objects will be CronTab
- Can then make instances of these custom objects
- And use them with operations like kubectl get
- Creating these made easier with SDK tools, such as ”**controller-runtime**” & “**controller-gen**”

```yaml
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: my-new-cron-object
spec:
  cronSpec: "* * * * */5"
  image: my-awesome-cron-image
```

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <p
  name: crontabs.stable.example.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: stable.example.com
  # list of versions supported by this CustomResourceDefinition
  versions:
    - name: v1
      # Each version can be enabled/disabled by Served flag.
      served: true
      # One and only one version must be marked as the storage ve
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                cronSpec:
                  type: string
                image:
                  type: string
                replicas:
                  type: integer
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    # plural name to be used in the URL: /apis/<group>/<version>/
    plural: crontabs
    # singular name to be used as an alias on the CLI and for dis
    singular: crontab
    # kind is normally the CamelCased singular type. Your resourc
    kind: CronTab
    # shortNames allow shorter string to match your resource on t
    shortNames:
    - ct
```

What is the controller-runtime?

- set of Go libraries for building Controllers
- used by operator-sdk and kubebuilder
- referenced via the Controller Runtime Client API
- provides ways to watch and reconcile resources in a K8s cluster
- supports CRUD (Create, [Get, List], Update, Delete) operations
- operators use at least 1 controller to perform task
  - generally through some combination of CRUD operations
- Operator SDK uses controller-runtime's Client interface
  - provides interface for these operations

What is controller-gen?

- Tool (from KubeBuilder) for generating K8s YAML and utility code
- Part of larger set of Go libraries for generating controllers - **controller-tools**
- Code and config generation controlled by “marker comments” in Go code
- Types of items produced via controller-gen
  - operator role
  - CRD
  - add validation to CRD
  - scaffold DeepCopy functions for objects
  - help in producing webhooks
- Controller-gen is composed of:
  - Generators - specify what to generate
  - Output rules - specify where and how to write the results
- Integration with make in sdk
  - make manifests - generates K8s object YAML
    - » CRDs, webhook configurations, RBAC roles
  - make generate - generates code
    - » runtime.Object/DeepCopy implementations

What makes up an operator controller?

- Standard imports
  - core controller-runtime library
  - client package
  - package for API types

```
import (
        "context"

        "github.com/go-logr/logr"
        "k8s.io/apimachinery/pkg/runtime"
        ctrl "sigs.k8s.io/controller-runtime"
        "sigs.k8s.io/controller-runtime/pkg/client"

        roarappv1alpha1 "github.com/brentlaster/roar-op/api/v1alpha1"
)
```

- Reconciler data structure
  - Scaffolded to start
  - Includes logging and fetching objs
  - Schemes - associate Go types with K8s groups, versions, and kinds

```
// RoarAppReconciler reconciles a RoarApp object
type RoarAppReconciler struct {
        client.Client
        Log    logr.Logger
        Scheme *runtime.Scheme
}
```

- RBAC permissions
  - via RBAC markers
  - Starts with bare minimum

```
// +kubebuilder:rbac:groups=roarapp.roarapp.com,resources=roarapps,verbs=get;list;watch;create;update;patch;delete
// +kubebuilder:rbac:groups=roarapp.roarapp.com,resources=roarapps/status,verbs=get;update;patch
```

- Reconciler function
  - Request contains info needed to uniquely identify an object - Name and Namespace
  - Context allows cancellation of requests, tracing, etc.
    - » Background context is basic one
  - logging handle uses structured logging library named “logr”
  - logging works via attaching key-value pairs to static message

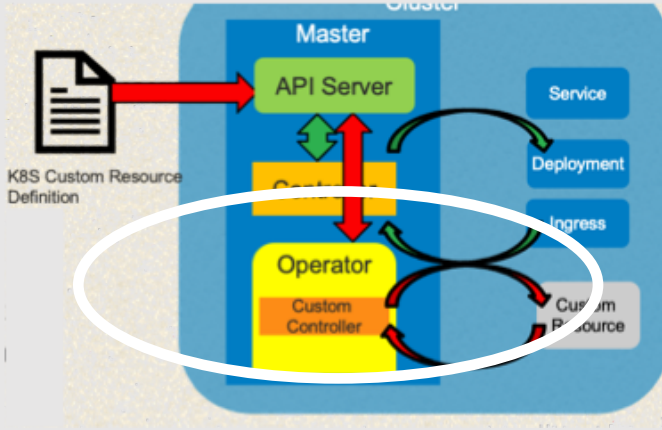
```
func (r *RoarAppReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
        _ = context.Background()
        _ = r.Log.WithValues("roarapp", req.NamespacedName)

        // your logic here

        return ctrl.Result{}, nil
}
```

- SetupWithManager function
  - Adds reconciler to manager
  - Want it started when manager is started

```
func (r *RoarAppReconciler) SetupWithManager(mgr ctrl.Manager) error {
        return ctrl.NewControllerManagedBy(mgr).
                For(&roarappv1alpha1.RoarApp{}).
                Complete(r)
}
```

Reconcile and Client API

- Reconciler implements reconcile.Reconciler interface
  - exposes Reconcile method
  - added to corresponding Controller for a Kind
  - called in response to cluster or external Evens
  - reads and writes cluster state by Controller
  - returns ctrl.Result
  - SDK Reconcilers have Client access to make K8s API calls



https://github.com/brentlaster/ops-ref/blob/main/controllers/roarapp_controller.go

```
58  func (r *RoarAppReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
59
60          log := r.Log.WithValues("roarapp", req.NamespacedName)
61
62          log.Info("Reconciling instance")
63
```

```
119         if numAvailable > instance.Spec.Replicas {
120                 log.Info("Scaling down pods", "Currently available", numAvailable, "Required replicas", instance.Spec.Replicas)
121                 diff := numAvailable - instance.Spec.Replicas
122                 dpods := available[:diff]
```

```
151         if numAvailable < instance.Spec.Replicas {
152                 log.Info("Scaling up pods", "Currently available", numAvailable, "Required replicas", instance.Spec.Replicas)
153                 // Define a new Pod object
154                 pod := newPodForCR(instance)
```

Operators vs Custom Controllers

- Operators are controllers
  - All operators use the controller pattern
  - Not all controllers are operators
- Operators codify operational knowledge for an application
- Operators
  - Encapsulate application-specific knowledge
  - Manage lifecycle for an application
  - Provide a CRD
- Operator = controller pattern + API extension + single-app focus
- Controllers work on existing K8s resources, operators on custom
- Custom controller - program that subscribes and listens to API server
  - When detects event it is interested in, takes action, may use information found in event's data

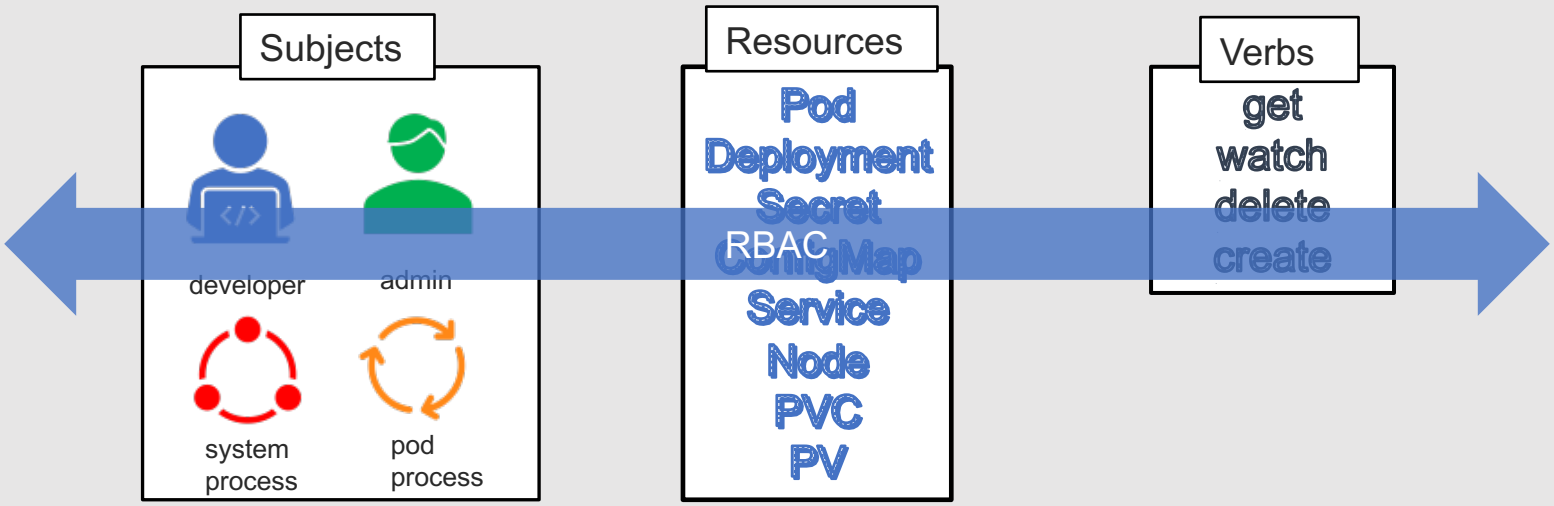
Workflow

- ✓ Use sdk to create scaffold project ($ operator-sdk init)
- ✓ Create scaffold API , CRD, and controller ($ operator-sdk create api)
- ✓ Update API to have necessary fields for spec and status ($ edit *_types.go)
- ✓ After above, automatically update generated code for resource - updates interface/implementation for runtime.Object to be a valid Kubernetes Kind ($ make generate)
  - ✓ updates api/<version>/zz_generated.deepcopy.go
- ✓ Generate CRD manifests ($ make manifests)
  - ✓ creates config/crd/bases/<item>.yaml
  - ✓ creates config/rbac/role.yaml
- ❑ Update scaffolded controller code to complete controller functionality
  - ❑ Add needed packages and core functions from Kubernetes
  - ❑ Add code to reconcile desired state of CR against observed state
  - ❑ Add code to create any needed K8s objects (Kinds)
  - ❑ Add any additional markers to manage items such as permissions/generate manifests
  - ❑ updates controllers/<item>_controller.go
- ❑ Update CRD manifests ($ make manifests)
  - ❑ updates config/crd/bases/<item>.yaml
  - ❑ updates config/rbac/role.yaml
- ❑ Execute install target to register CRD with K8s apiserver ($ make install)
- ❑ Test the operator ($ make run)
- ❑ Create the docker package ($ make docker-build)
- ❑ Push image to registry ($ make docker-push)
- ❑ Deploy operator to cluster ($ make deploy)

Lab 3 - Implementing the Controller

Kubernetes RBAC

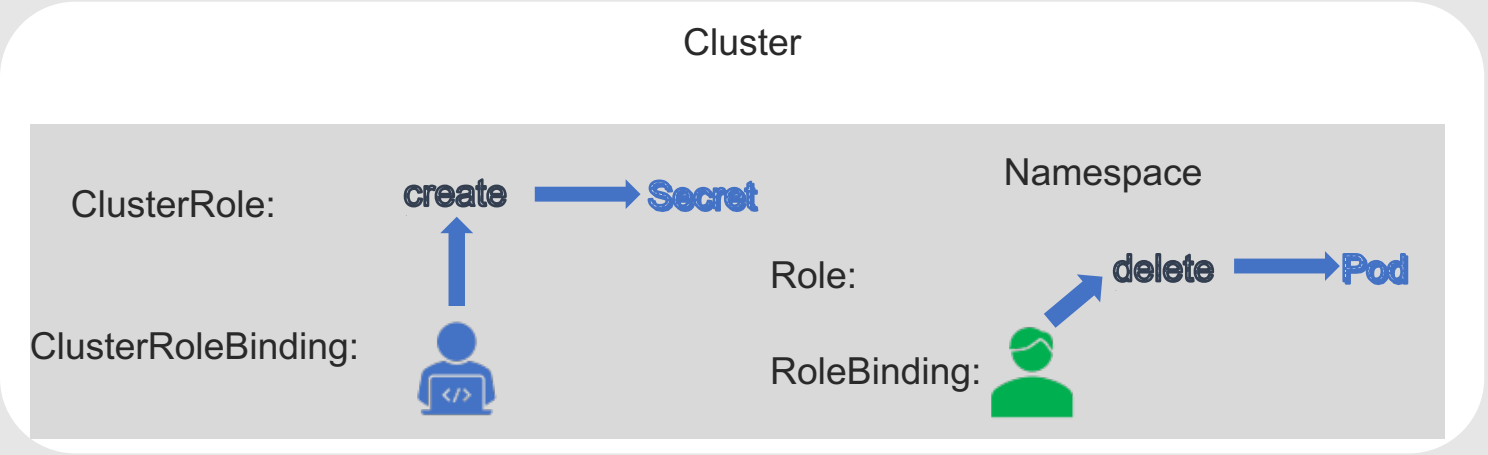
- The pieces
  - 3 types of elements involved
    - Subjects: users or processes that need to access the API
    - Resources: Kubernetes API Objects available in the cluster. (Examples include Pods, Deployments, Services, Nodes, and Persistent Volumes, etc.)
    - Verbs: The Kubernetes operations that can be executed on the resources above. Multiple kinds of verbs available – all are CRUD operations (Examples include get, watch, create, delete, etc.)



Goal: Connect subjects, API resources, and verbs. Specify, given a user, which operations can be executed over which resources.

Kubernetes RBAC

- The mechanisms
  - Object types
    - k8s Roles: Connect API Resources and Verbs; can be reused for different Subjects; bound to a namespace
    - k8s RoleBinding: Connects Subjects to Roles; Given a role that combines API objects and verbs, defines which subjects can use it.
    - k8s ClusterRole: a role to be applied across a cluster
    - k8s ClusterRoleBinding: Connects Subjects to ClusterRole.



Kubernetes RBAC - Service Accounts

- Users: These are global, and meant for humans or processes living outside the cluster.
- ServiceAccounts: These are namespaced and meant for intra-cluster processes running inside pods.
- Both authenticate against the API to get access to resources
- But there is no kubernetes “User” object
- So you can do “kubectl create serviceaccount <name>” but not “kubectl create user <name>”
- Bottom line: cluster does not store any information about users, so must be managed outside of cluster with certificates, tokens, etc.

Create ServiceAccount

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: myapp
  labels:
    app: myapp
```

Create Role

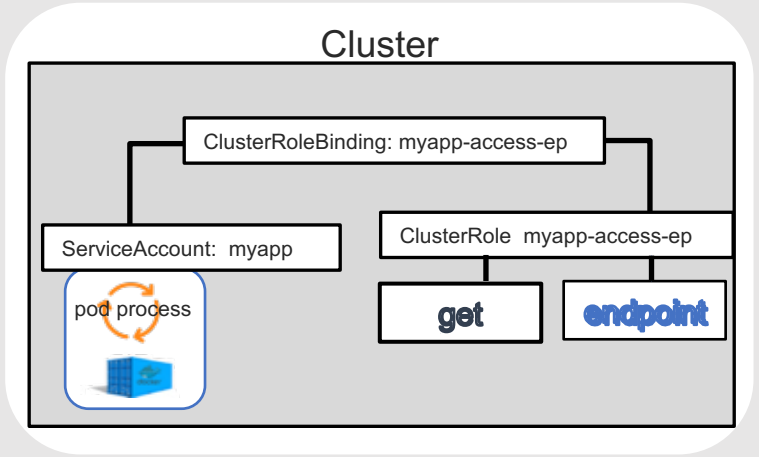
```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: myapp-access-ep
  labels:
    app: myapp
rules:
- apiGroups: [""]
  resources: ["endpoints"]
  verbs: ["get"]
```

Add ServiceAccount to Pod

```yaml
[...]
spec:
  template:
      metadata:
        name: myapp
        labels:
          app: myapp
      spec:
        serviceAccountName: myapp
        containers:
[...]
```

Tie Role to ServiceAccount

```yaml
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: myapp-access-ep
  labels:
    app: myapp
subjects:
- kind: ServiceAccount
  name: myapp
roleRef:
  kind: ClusterRole
  name: myapp-access-ep
  apiGroup: rbac.authorization.k8s.io
```



Markers for Config/Code Generation

- Carried over in sdk from Kubebuilder
- Markers are special comments in Go code
  - start with "// +"
  - indicate additional info around types, fields, and packages
- Signal "controller-gen" tool to generate
  - utility code
    - » done via "make generate"
    - » examples: runtime.Object/DeepCopy implementations
  - object YAML
    - » done via "make manifests"
    - » examples: CRDs, RBAC roles

https://github.com/brentlaster/ops-ref/blob/main/controllers/roarapp_controller.go

```
54  // +kubebuilder:rbac:groups=core,resources=pods,verbs=get;list;watch;create;update;delete
55  // +kubebuilder:rbac:groups=batch,resources=jobs,verbs=get;list;watch;create;update;patch;delete
56  // +kubebuilder:rbac:groups=core,resources=services,verbs=get;list;watch;create;update;delete
57
58  func (r *RoarAppReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
59
```

Marker Syntax

- 3 types
  - Empty (+kubebuilder:validation:Optional)
    - » behave like boolean flags - prescence indicates ”on”
  - Anonymous (+kubebuilder:validation:Maximum=20)
    - » take a single value as argument
  - Multi-option (+kubebuilder:printcolumn:JSONPath=“.status.replicas”,name=Replicas,type=string)
    - » take one or more arguments
    - » first argument separated from name by colon
    - » remaining arguments separated by comma
    - » order doesn’t matter

```
43  // +kubebuilder:subresource:status
44
45  // RoarApp is the Schema for the roarapps API
46  type RoarApp struct {
47          metav1.TypeMeta   `json:",inline"`
48          metav1.ObjectMeta `json:"metadata,omitempty"`
49
50          Spec   RoarAppSpec   `json:"spec,omitempty"`
51          Status RoarAppStatus `json:"status,omitempty"`
52  }
```

```
54  // +kubebuilder:object:root=true
55
56  // RoarAppList contains a list of RoarApp
57  type RoarAppList struct {
58          metav1.TypeMeta `json:",inline"`
59          metav1.ListMeta `json:"metadata,omitempty"`
60          Items           []RoarApp `json:"items"`
61  }
```

https://github.com/brentlaster/ops-ref/blob/main/controllers/roarapp_controller.go

```
54  // +kubebuilder:rbac:groups=core,resources=pods,verbs=get;list;watch;create;update;delete
55  // +kubebuilder:rbac:groups=batch,resources=jobs,verbs=get;list;watch;create;update;patch;delete
56  // +kubebuilder:rbac:groups=core,resources=services,verbs=get;list;watch;create;update;delete
57
58  func (r *RoarAppReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
59
```

Marker Generator Categories

CRD Generation

These markers describe how to construct a custom resource definition from a series of Go types and packages. Generation of the actual validation schema is described by the validation markers.

See Generating CRDs for examples.

▶ Show Detailed Argument Help

// +kubebuilder:printcolumn :JSONPath=‹string›,description=‹string›,format=‹string›,name=‹string›,priority=‹int›,type=‹string› on type

adds a column to "kubectl get" output for this CRD.

// +kubebuilder:resource :categories=‹[]string›,path=‹string›,scope=‹string›,shortName=‹[]string›,singular=‹string› on type

configures naming and scope for a CRD.

// +kubebuilder:skipversion on type

▶ removes the particular version of the CRD from the CRDs spec.

// +kubebuilder:storageversion on type

▶ marks this version as the "storage version" for the CRD for conversion.

// +kubebuilder:subresource:scale :selectorpath=‹string›,specpath=‹string›,statuspath=‹string› on type

enables the "/scale" subresource on a CRD.

// +kubebuilder:subresource:status on type

enables the "/status" subresource on a CRD.

CRD Processing

These markers help control how the Kubernetes API server processes API requests involving your custom resources.

See Generating CRDs for examples.

▶ Show Detailed Argument Help

// +kubebuilder:pruning:PreserveUnknownFields on field

▶ PreserveUnknownFields stops the apiserver from pruning fields which are not specified.

// +kubebuilder:validation:XPreserveUnknownFie on field

▶ PreserveUnknownFields stops the apiserver from pruning fields which are not specified.

// +listMapKey:=‹string› on field

▶ specifies the keys to map listTypes.

// +listType:=‹string› on field

▶ specifies the type of data-structure that the list represents (map, set, atomic).

// +mapType:=‹string› on field

▶ specifies the level of atomicity of the map; i.e. whether each item in the map is independent of the others, or all fields are treated as a single unit.

CRD Validation

These markers modify how the CRD validation schema is produced for the types and fields they modify. Each corresponds roughly to an OpenAPI/JSON schema option.

See Generating CRDs for examples.

▶ Show Detailed Argument Help

// +kubebuilder:default:=‹any› on field

▶ sets the default value for this field.

// +kubebuilder:validation:EmbeddedResource on field

▶ EmbeddedResource marks a fields as an embedded resource with apiVersion, kind and metadata fields.

// +kubebuilder:validation:Enum:=‹[]any› on field

specifies that this (scalar) field is restricted to the *exact* values specified here.

// +kubebuilder:validation:ExclusiveMaximum :=‹bool› on field

indicates that the maximum is "up to" but not including that value.

// +kubebuilder:validation:ExclusiveMinimum :=‹bool› on field

indicates that the minimum is "up to" but not including that value.

Workflow

- ✓ Use sdk to create scaffold project ($ operator-sdk init)
- ✓ Create scaffold API , CRD, and controller ($ operator-sdk create api)
- ✓ Update API to have necessary fields for spec and status ($ edit *_types.go)
- ✓ After above, automatically update generated code for resource - updates interface/implementation for runtime.Object to be a valid Kubernetes Kind ($ make generate)
  - ✓ updates api/<version>/zz_generated.deepcopy.go
- ✓ Generate CRD manifests ($ make manifests)
  - ✓ creates config/crd/bases/<item>.yaml
  - ✓ creates config/rbac/role.yaml
- ✓ Update scaffolded controller code to complete controller functionality
  - ✓ Add needed packages and core functions from Kubernetes
  - ✓ Add code to reconcile desired state of CR against observed state
  - ✓ Add code to create any needed K8s objects (Kinds)
  - ❑ Add any additional markers to manage items such as permissions/generate manifests
  - ❑ updates controllers/<item>_controller.go
- ❑ Update CRD manifests ($ make manifests)
  - ❑ updates config/crd/bases/<item>.yaml
  - ❑ updates config/rbac/role.yaml
- ❑ Execute install target to register CRD with K8s apiserver ($ make install)
- ❑ Test the operator ($ make run)
- ❑ Create the docker package ($ make docker-build)
- ❑ Push image to registry ($ make docker-push)
- ❑ Deploy operator to cluster ($ make deploy)

Lab 4 - Specifying Permissions and Adding RBAC Manifests

main.go

- Basic imports
- Scheme - provides mappings between Kinds and Go types
- Basic flags for metrics
- Leader election - only one instance can be in charge (for example if multiple during upgrade)
- Manager instantiation
  - manager keeps tracks of running controllers, sets up shared caches and API client connections
  - runs controllers and webhooks
  - runs until it gets a graceful shutdown signal
  - argument to manager instantiation defines if single or multiple namespaces

```
var (
        scheme   = runtime.NewScheme()
        setupLog = ctrl.Log.WithName("setup")
)

func init() {
        utilruntime.Must(clientgoscheme.AddToScheme(scheme))

        utilruntime.Must(roarappv1alpha1.AddToScheme(scheme))
        // +kubebuilder:scaffold:scheme
}

func main() {
        var metricsAddr string
        var enableLeaderElection bool
        flag.StringVar(&metricsAddr, "metrics-addr", ":8080", "The addres
binds to.")
        flag.BoolVar(&enableLeaderElection, "enable-leader-election", fal
                "Enable leader election for controller manager. "+
                        "Enabling this will ensure there is only one acti
        flag.Parse()

        ctrl.SetLogger(zap.New(zap.UseDevMode(true)))

        mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
                Scheme:             scheme,
                MetricsBindAddress: metricsAddr,
                Port:               9443,
                LeaderElection:     enableLeaderElection,
                LeaderElectionID:   "53dff33f.roarapp.com",
        })
        if err != nil {
                setupLog.Error(err, "unable to start manager")
                os.Exit(1)
        }

        if err = (&controllers.RoarAppReconciler{
                Client: mgr.GetClient(),
                Log:    ctrl.Log.WithName("controllers").WithName("RoarAp
                Scheme: mgr.GetScheme(),
        }).SetupWithManager(mgr); err != nil {
                setupLog.Error(err, "unable to create controller", "contr
                os.Exit(1)
        }
        // +kubebuilder:scaffold:builder

        setupLog.Info("starting manager")
        if err := mgr.Start(ctrl.SetupSignalHandler()); err != nil {
                setupLog.Error(err, "problem running manager")
                os.Exit(1)
        }
}
```

What is the manager?

```
diyuser3@training1:~/ops$ tree
.
├── bin
│   └── manager
├── config
│   ├── certmanager
│   │   ├── certificate.yaml
│   │   └── kustomization.yaml
```

- Manager (package manager component)
  - required to create, start/stop Controllers
  - provides/initializes shared dependencies
    - » schemes, caches, clients,etc.
    - » Scheme - used to resolve runtime objects to group/version/kind
  - executable that wraps 1 or more Controllers
    - » Controllers have to be started by invoking Manager.start
  - can be built & run locally against a remote cluster OR can be run as container in cluster
  - if in container, needs its own Namespace with ServiceAccount and RBAC perms for resources
    - » configs for that automatically created via running make command
  - example:
    - » assume running cluster referenced in ~/.kube/config
    - » build/run locally against cluster defined in ~/.kube/config

```
$ make run
```

then separately create an instance of your resource

```
$ kubectl apply -f config/<file path>
```

- SDK generates code to create Manager
- manager holds Cache and Client to be used with CRUD operations and communicate with API server

Operators Scope

- Operators can have two scopes - cluster-scoped & namespaced-scoped
  - cluster-scoped
    - » can watch resources across cluster
    - » requires cluster-scoped permissions (generally a concern)
    - » default for operator-sdk (init)
  - namespaced-scoped
    - » watches resources in a Namespace
    - » best for flexible deployment
    - » permits namespace isolation for monitoring, failure cases, etc.
    - » allows decoupled upgrades

Changing Operator Scope

- Two components for changing scope
  - Manager settings
    - » By default, Manager does not have any namespace defined
    - » So watches all namespaces
    - » Add namespace or set of namespaces in call to new Manager
    - » Can also use environment variables
  - Restrict roles and permissions
    - » Change role.yaml and role_binding.yaml in config/rbac directory
    - » From ClusterRole/ClusterRoleBinding to Role/Rolebinding

```
mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
        Scheme:             scheme,
        MetricsBindAddress: metricsAddr,
        Port:               9443,
        LeaderElection:     enableLeaderElection,
        LeaderElectionID:   "53dff33f.roarapp.com",
})
```

```
mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
        Scheme:             scheme,
        MetricsBindAddress: metricsAddr,
        Port:               9443,
        LeaderElection:     enableLeaderElection,
        LeaderElectionID:   "53dff33f.roarapp.com",
        Namespace:          "operator-namespace",
})
```

Kustomize

- Kustomize creates modified Kubernetes manifests by "overlaying" declarative specifications on top of existing Kubernetes manifests
- Changes to make/overlay are stored in a separate file called *kustomization.yaml*
- Kustomize reads the kustomization.yaml file and the Kubernetes spec files (manifests) and dumps out completed resources to stdout with the changes from kustomization.yaml applied "on top of" the standard Kubernetes files
- *kustomize build* command causes files to be processed and output created
- Kustomize leverages the idea of "transformers"
- Transformers are functionality built into Kustomize that "transforms" your Kubernetes manifests given a simple set of declarative rules

Simple Example

Directory

roar-web-deploy.yaml
roar-web-svc.yaml
kustomization.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: roar-web
  name: roar-web
spec:
  replicas: 1
  template:
    metadata:
      labels:
```

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: roar-web
  name: roar-web
spec:
  ports:
  - name: frontend
```

```yaml
# common label to apply
commonLabels:
  level: test

# List of resource files that
# kustomize reads, modifies
# and emits as a YAML string
resources:
- roar-web-deploy.yaml
- roar-web-svc.yaml
```

kustomize build .

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: roar-web
    level: test
  name: roar-web
spec:
  ports:
  - name: frontend
    port: 8089
    targetPort: 8080
  selector:
    app: roar-web
    level: test
  type: NodePort
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    app: roar-web
    level: test
  name: roar-web
spec:
  replicas: 1
  selector:
    matchLabels:
      level: test
  template:
```

Variants

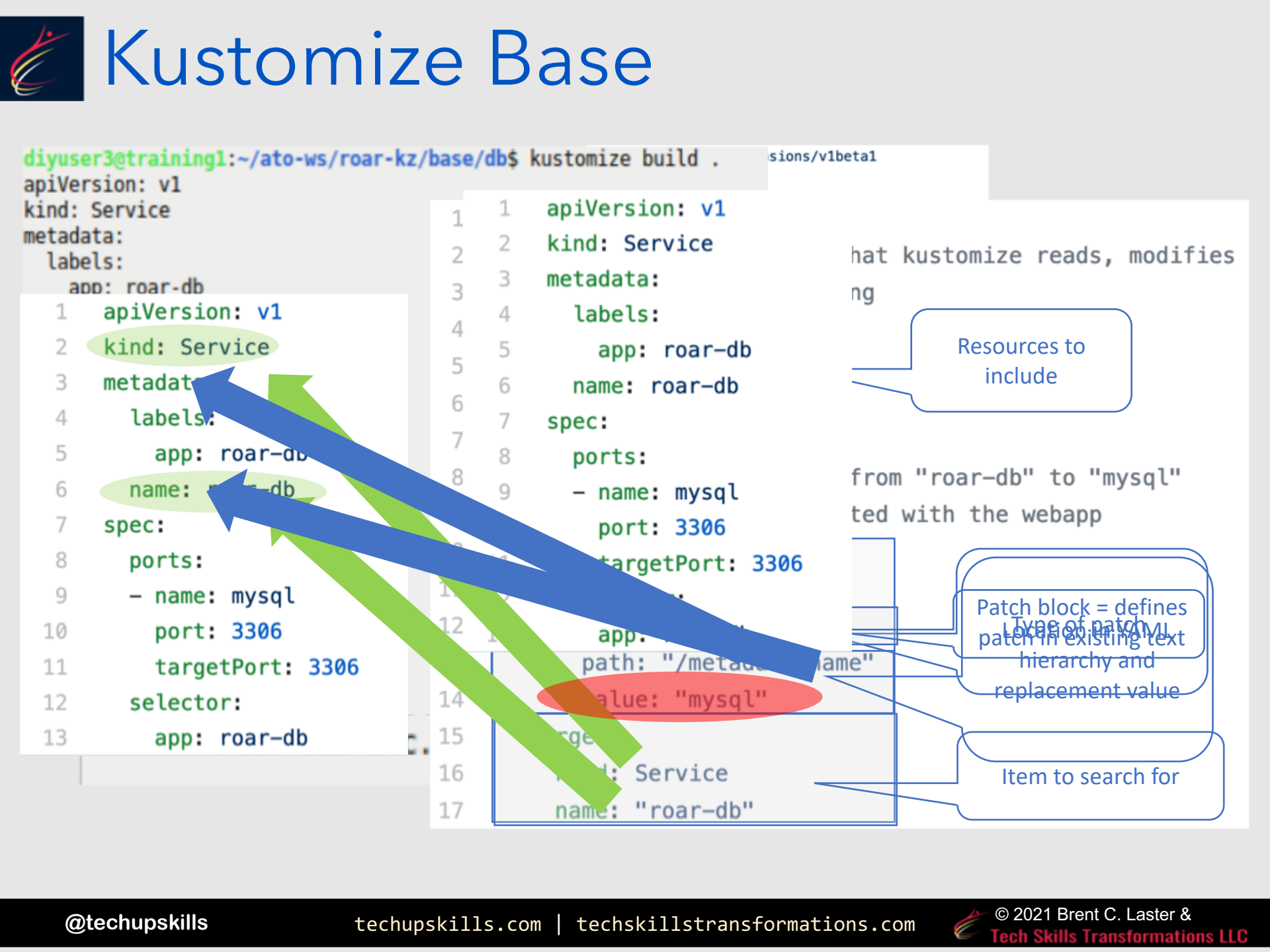
- Common use case is needing multiple variations (variants) of a set of common resources (i.e. dev, stage, prod)
- To do this, Kustomize has concept of “overlay” and “base”
  - Both represented via kustomization.yaml file (includes resources + customizations)
  - Base – declares things variants have in common
  - Overlays – declare the differences
- Variants can also apply “patches”
  - Patch – partial deployment spec – used to update some existing spec as opposed to adding new information

Kustomize Base

```
diyuser3@training1:~/ato-ws/roar-kz/base/db$ kustomize build .
apiVersion: v1
kind: Service
metadata:
  labels:
    app: roar-db
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: roar-db
  name: roar-db
spec:
  ports:
  - name: mysql
    port: 3306
    targetPort: 3306
  selector:
    app: roar-db
```

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: roar-db
  name: roar-db
spec:
  ports:
  - name: mysql
    port: 3306
    targetPort: 3306
    app:
  path: "/metad ame"
  alue: "mysql"
  rge
  : Service
  name: "roar-db"
```



Running the operator for testing outside of the cluster

```
diyuser3@training1:~/ops$ make run ENABLE_WEBHOOKS=false
go: creating new go.mod: module tmp
go: found sigs.k8s.io/controller-tools/cmd/controller-gen in sigs.k8s.io/controller-tools v0.3.0
/home/diyuser3/go/bin/controller-gen object:headerFile="hack/boilerplate.go.txt" paths="./..."
go fmt ./...
controllers/roarapp_controller.go
go vet ./...
/home/diyuser3/go/bin/controller-gen "crd:trivialVersions=true" rbac:roleName=manager-role webhook paths="./..." output:crd:artifacts:config=config/crd/bases
go run ./main.go
2021-02-18T21:12:04.958-0500    INFO    controller-runtime.metrics      metrics server is starting to listen    {"addr": ":8080"}
2021-02-18T21:12:04.958-0500    INFO    setup   starting manager
2021-02-18T21:12:04.959-0500    INFO    controller-runtime.controller   Starting EventSource    {"controller": "roarapp", "source": "kind source: /, Kind="}
2021-02-18T21:12:04.959-0500    INFO    controller-runtime.manager      starting metrics server {"path": "/metrics"}
2021-02-18T21:12:05.064-0500    INFO    controller-runtime.controller   Starting Controller     {"controller": "roarapp"}
2021-02-18T21:12:05.064-0500    INFO    controller-runtime.controller   Starting workers        {"controller": "roarapp", "worker count": 1}
```

```
diyuser3@training1:~$ cat ref/roarapp.yaml
apiVersion: roarapp.roarapp.com/v1alpha1
kind: RoarApp
metadata:
  name: roarapp-sample
spec:
  # Add fields here
  replicas: 3
  dbImage: quay.io/bclaster/roar-db:1.0.2
  webImage: quay.io/bclaster/roar-web:1.0.1
```

```
diyuser3@training1:~$ k apply -n op-test -f ref/roarapp.yaml
roarapp.roarapp.roarapp.com/roarapp-sample created
diyuser3@training1:~$ k get pods -n op-test
NAME                          READY   STATUS    RESTARTS   AGE
roarapp-sample-pod-32000      2/2     Running   0          47s
roarapp-sample-pod-32001      2/2     Running   0          46s
roarapp-sample-pod-32002      2/2     Running   0          46s
```

```
2021-02-18T21:12:05.064-0500    INFO    controller-runtime.controller   Starting workers        {"controller": "roarapp", "worker count": 1}
2021-02-18T21:21:18.825-0500    INFO    controllers.RoarApp     Reconciling instance    {"roarapp": "op-test/roarapp-sample"}
2021-02-18T21:21:18.935-0500    INFO    controllers.RoarApp     Scaling up pods {"roarapp": "op-test/roarapp-sample", "Currently available": 0, "Required repl
icas": 3}
2021-02-18T21:21:18.992-0500    INFO    controllers.RoarApp     Reconciling instance    {"roarapp": "op-test/roarapp-sample"}
2021-02-18T21:21:19.002-0500    INFO    controllers.RoarApp     Scaling up pods {"roarapp": "op-test/roarapp-sample", "Currently available": 1, "Required repl
icas": 3}
2021-02-18T21:21:19.045-0500    INFO    controllers.RoarApp     Reconciling instance    {"roarapp": "op-test/roarapp-sample"}
2021-02-18T21:21:19.053-0500    INFO    controllers.RoarApp     Scaling up pods {"roarapp": "op-test/roarapp-sample", "Currently available": 2, "Required repl
icas": 3}
2021-02-18T21:21:19.098-0500    INFO    controllers.RoarApp     Reconciling instance    {"roarapp": "op-test/roarapp-sample"}
2021-02-18T21:21:19.103-0500    DEBUG   controller-runtime.controller   Successfully Reconciled {"controller": "roarapp", "request": "op-test/roarapp-sample"}
2021-02-18T21:21:19.103-0500    INFO    controllers.RoarApp     Reconciling instance    {"roarapp": "op-test/roarapp-sample"}
2021-02-18T21:21:19.103-0500    DEBUG   controller-runtime.controller   Successfully Reconciled {"controller": "roarapp", "request": "op-test/roarapp-sample"}
2021-02-18T21:21:19.118-0500    INFO    controllers.RoarApp     Reconciling instance    {"roarapp": "op-test/roarapp-sample"}
2021-02-18T21:21:19.119-0500    DEBUG   controller-runtime.controller   Successfully Reconciled {"controller": "roarapp", "request": "op-test/roarapp-sample"}
```

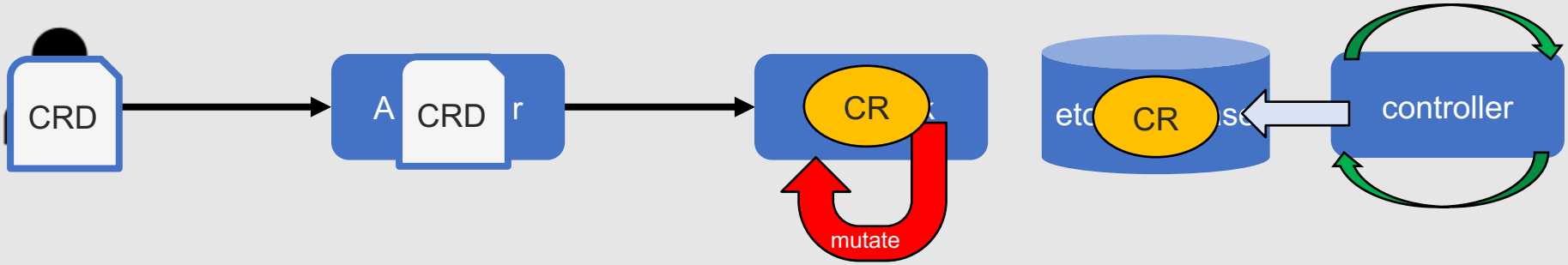
Workflow

- ✓ Use sdk to create scaffold project ($ operator-sdk init)
- ✓ Create scaffold API , CRD, and controller ($ operator-sdk create api)
- ✓ Update API to have necessary fields for spec and status ($ edit *_types.go)
- ✓ After above, automatically update generated code for resource - updates interface/implementation for runtime.Object to be a valid Kubernetes Kind ($ make generate)
  - ✓ updates api/<version>/zz_generated.deepcopy.go
- ✓ Generate CRD manifests ($ make manifests)
  - ✓ creates config/crd/bases/<item>.yaml
  - ✓ creates config/rbac/role.yaml
- ✓ Update scaffolded controller code to complete controller functionality
  - ✓ Add needed packages and core functions from Kubernetes
  - ✓ Add code to reconcile desired state of CR against observed state
  - ✓ Add code to create any needed K8s objects (Kinds)
  - ✓ Add any additional markers to manage items such as permissions/generate manifests
  - ✓ updates controllers/<item>_controller.go
- ✓ Update CRD manifests ($ make manifests)
  - ✓ updates config/crd/bases/<item>.yaml
  - ✓ updates config/rbac/role.yaml
- ❑ Execute install target to register CRD with K8s apiserver ($ make install)
- ❑ Test the operator ($ make run)
- ❑ Create the docker package ($ make docker-build)
- ❑ Push image to registry ($ make docker-push)
- ❑ Deploy operator to cluster ($ make deploy)

Lab 5 - Building and Testing the Controller

Operators and Webhooks

- Kubernetes api request passes through admission controllers that can invoke webhooks to validate or update api requests
- Webhook is an HTTP ”callback” that is registered with the K8s API server
- When a designated event occurs, the K8s server queries registered webhooks and forwards a message
- Can be ”mutating” (allowed to modify input objects)
- Can be “validating” (read and validate input objects)



- User creates a CRD and K8s API forwards on to webhook based on registered list.
- Webhook completes any validations/mutations - such as filling in default parameters and/or validating existing ones.
- CR then gets written to database and is available.
- Controller monitors CR in the background and does any needed processing.
- Any updated status resulting from the processing is recorded by controller as status data in the CRD.

Running the Operator in a Cluster

- Create the docker package ($ make docker-build)
- Push image to registry ($ make docker-push)
- Deploy operator to cluster ($ make deploy)

```
diyuser3@training1:~/ops$ k get all -n ops-system
NAME                                             READY   STATUS    RESTARTS   AGE
pod/ops-controller-manager-776bbdbdc7-sls8g      2/2     Running   0          53s

NAME                                                TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/ops-controller-manager-metrics-service      ClusterIP   10.104.206.229   <none>        8443/TCP   53s

NAME                                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ops-controller-manager      1/1     1            1           53s

NAME                                                   DESIRED   CURRENT   READY   AGE
replicaset.apps/ops-controller-manager-776bbdbdc7      1         1         1       53s
```

```
diyuser3@training1:~/ops$ k logs -n ops-system ops-controller-manager-776bbdbdc7-sls8g -c manager
2021-02-19T03:50:21.806Z        INFO    controller-runtime.metrics      metrics server is starting to listen    {"a
ddr": "127.0.0.1:8080"}
2021-02-19T03:50:21.806Z        INFO    setup   starting manager
I0219 03:50:21.806944       1 leaderelection.go:242] attempting to acquire leader lease  ops-system/53dff33f.roarap
p.com...
2021-02-19T03:50:21.807Z        INFO    controller-runtime.manager      starting metrics server {"path": "/metrics"
}
I0219 03:50:21.811838       1 leaderelection.go:252] successfully acquired lease ops-system/53dff33f.roarapp.com
2021-02-19T03:50:21.811Z        DEBUG   controller-runtime.manager.events       Normal  {"object": {"kind":"ConfigM
ap","namespace":"ops-system","name":"53dff33f.roarapp.com","uid":"3f25431e-3d08-4e68-b2e7-11deffe4ca53","apiVersion
":"v1","resourceVersion":"19114"}, "reason": "LeaderElection", "message": "ops-controller-manager-776bbdbdc7-sls8g_
41d4206b-fbad-4ad8-bd0d-9875dbdca192 became leader"}
2021-02-19T03:50:21.812Z        INFO    controller-runtime.controller   Starting EventSource    {"controller": "roa
rapp", "source": "kind source: /, Kind="}
2021-02-19T03:50:21.913Z        INFO    controller-runtime.controller   Starting Controller     {"controller": "roa
rapp"}
2021-02-19T03:50:21.914Z        INFO    controller-runtime.controller   Starting workers        {"controller": "roa
```

How Manager container/pod map with main and controller

```
diyuser3@training1:~/ops$ k get all -n ops-system
NAME                                          READY   STATUS    RESTARTS   AGE
pod/ops-controller-manager-776bbdbdc7-sls8g   2/2     Running   0          25m

NAME                                                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/ops-controller-manager-metrics-service   ClusterIP   10.104.206.229   <none>        8443/TCP   25m

NAME                                     READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/ops-controller-manager   1/1     1            1           25m

NAME                                                DESIRED   CURRENT   READY   AGE
replicaset.apps/ops-controller-manager-776bbdbdc7   1         1         1       25m
```

main.go

```
10 RUN go mod do
11
12 # Copy the go
13 COPY main.go
14 COPY api/ api
15 COPY controll
16
17 # Build
18 RUN CGO_ENABL
   manager main.
19
20 # Use distrol
21 # Refer to ht
   details
22 FROM gcr.io/d
23 WORKDIR /
24 COPY --from=b
25 USER nonroot:
26
27 ENTRYPOINT ["
```

```
57
58          mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
59                  Scheme:             scheme,
60                  MetricsBindAddress: metricsAddr,
61                  Port:               9443,
62                  LeaderElection:     enableLeaderElection,
63                  LeaderElectionID:   "53dff33f.roarapp.com",
64          })
65          if err != nil {
66                  setupLog.Error(err, "unable to start manager")
67                  os.Exit(1)
68          }
69
70          if err = (&controllers.RoarAppReconciler{
71                  Client: mgr.GetClient(),
72                  Log:    ctrl.Log.WithName("controllers").WithName("RoarApp"),
73                  Scheme: mgr.GetScheme(),
74          }).SetupWithManager(mgr); err != nil {
75                  setupLog.Error(err, "unable to create controller", "controller", "RoarApp") ')
76                  os.Exit(1)
77          }
78          // +kubebuilder:scaffold:builder
79
80          setupLog.Info("starting manager")
81          if err := mgr.Start(ctrl.SetupSignalHandler()); err != nil {
82                  setupLog.Error(err, "problem running manager")
83                  os.Exit(1)
84          }
85 }
```

Custom Resource/Types in Production

From API - _types.go

```
// RoarAppSpec defines the desired state of RoarApp
type RoarAppSpec struct {
	// INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
	// Important: Run "make" to regenerate code after modifying this file
	Replicas int32  `json:"replicas"`
	WebImage string `json:"webImage,omitempty"`
	DbImage  string `json:"dbImage,omitempty"`
}
```

```
// RoarAppStatus defines the observed state of RoarApp
type RoarAppStatus struct {
	// INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
	// Important: Run "make" to regenerate code after modifying this file
	PodNames []string `json:"podNames"`
}
```

```
diyuser3@training1:~$ k get -n ops-system RoarApp
NAME              AGE
roarapp-sample    28m
diyuser3@training1:~$ k get -n ops-system -o yaml RoarAp
apiVersion: roarapp.roarapp.com/v1alpha1
kind: RoarApp
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"roarapp.roarapp.com/v1alpha1","kind
cas":3,"webImage":"quay.io/bclaster/roar-web:1.0.1"}}
  creationTimestamp: "2021-02-19T04:20:31Z"
  generation: 3
  managedFields:
  - apiVersion: roarapp.roarapp.com/v1alpha1
    fieldsType: FieldsV1
    fieldsV1:
          .: {}
        .: {}
    manager: kubectl
    operation: Update
    time: "2021-02-19T04:31:59Z"
  - apiVersion: roarapp.roarapp.com/v1alpha1
    fieldsType: FieldsV1
    fieldsV1:
        .: {}
    manager: manager
    operation: Update
    time: "2021-02-19T04:31:59Z"
  name: roarapp-sample
  namespace: ops-system
  resourceVersion: "21012"
  selfLink: /apis/roarapp.roarapp.com/v1alpha1/namespace
  uid: 
pec:
  dbImage: quay.io/bclaster/roar-db:1.0.2
  replicas: 3
  webImage: quay.io/bclaster/roar-web:1.0.1
tatus:
  podNames:
  - roarapp-sample-pod-32003
  - roarapp-sample-pod-32004
  - roarapp-sample-pod-32002
```

Workflow

- ✓ Use sdk to create scaffold project ($ operator-sdk init)
- ✓ Create scaffold API , CRD, and controller ($ operator-sdk create api)
- ✓ Update API to have necessary fields for spec and status ($ edit *_types.go)
- ✓ After above, automatically update generated code for resource - updates interface/implementation for runtime.Object to be a valid Kubernetes Kind ($ make generate)
  - ✓ updates api/<version>/zz_generated.deepcopy.go
- ✓ Generate CRD manifests ($ make manifests)
  - ✓ creates config/crd/bases/<item>.yaml
  - ✓ creates config/rbac/role.yaml
- ✓ Update scaffolded controller code to complete controller functionality
  - ✓ Add needed packages and core functions from Kubernetes
  - ✓ Add code to reconcile desired state of CR against observed state
  - ✓ Add code to create any needed K8s objects (Kinds)
  - ✓ Add any additional markers to manage items such as permissions/generate manifests
  - ✓ updates controllers/<item>_controller.go
- ✓ Update CRD manifests ($ make manifests)
  - ✓ updates config/crd/bases/<item>.yaml
  - ✓ updates config/rbac/role.yaml
- ✓ Execute install target to register CRD with K8s apiserver ($ make install)
- ✓ Test the operator ($ make run)
- ❑ Create the docker package ($ make docker-build)
- ❑ Push image to registry ($ make docker-push)
- ❑ Deploy operator to cluster ($ make deploy)

Lab 6 - Creating the Operator Image and Deploying it into a Cluster

That's all - thanks!

Professional Git 1st Edition

by Brent Laster (Author)

3 customer reviews

Look inside