

# Operator-SDK

## v0.1.0+

# Writing an Operator from Scratch is Challenging.

- Research client-library.
- Repo organization.
- Write boiler-plate code.
- Use code-generators.
- Knowledge of informers/shared informers and work queues for object cache and event handling.

# The Operator-SDK provides...

- Use of the controller-runtime library to write operational logic more intuitively.
- Choice of writing a controller with Ansible, Go or the Helm App Operator Kit.
- Tools for scaffolding and code generation to bootstrap a new project fast.
- Extensions to cover common operator use cases.

# Operator-SDK Workflow

(In 13 easy steps)

# 1) Create a New Operator Project.

```
$ operator-sdk new app-operator
```

```
$ ls app-operator
```

```
.git  
.gitignore  
Gopkg.lock  
Gopkg.toml  
build  
cmd  
deploy  
pkg  
vendor  
version
```

## 2) Add Custom Resource Definition (CRD) API.

```
$ operator-sdk add api --api-version=app.example.com/v1alpha1 --kind=App
```

```
├─ pkg/apis
│   ├── addtoscheme_app_v1alpha1.go
│   ├── apis.go
│   └── app
│       └── v1alpha1
│           ├── doc.go
│           ├── app_types.go
│           ├── register.go
│           └── zz_generated.deepcopy.go
├─ deploy/crds
│   ├── app_v1alpha1_mug_cr.yaml
│   └── app_v1alpha1_mug_crd.yaml
├─ operator.yaml
├─ role.yaml
├─ role_binding.yaml
└─ service_account.yaml
```

### 3) Define your CRD Spec/Status

```
$ cat pkg/apis/app/v1alpha1/app_types.go
```

```
type AppSpec struct {
    Size int32    `json:"size"`
    Version string `json:"version"`
    ConfigMapName string `json:"configMapName"`
    TLS *TLSPolicy `json:"TLS,omitempty"`
    SecretName string `json:"secretName"`
}

type AppStatus struct {
    Pods string `json:"pods"`
    Standby []string `json:"standby"`
    ExternalAddresses map[string]string `json:"externalAddresses"`
}
```

## 4) Generate DeepCopy Functions

```
$ operator-sdk generate k8s
```

Running code-generation for custom resource group versions: [app:v1alpha1, ]  
Generating deepcopy funcs

```
├── pkg/apis
│   ├── addtoscheme_app_v1alpha1.go
│   ├── apis.go
│   └── app
│       └── v1alpha1
│           ├── doc.go
│           ├── app_types.go
│           ├── register.go
│           └── zz_generated.deepcopy.go
```

## 5) Add a New Controller

```
$ operator-sdk add controller --api-version=app.example.com/v1alpha1 --kind=App
```

```
Create pkg/controller/app/app_controller.go
```

```
Create pkg/controller/add_app.go
```

```
└─ pkg/controller
   └─ add_app.go
   └─ app
      └─ app_controller.go
   └─ controller.go
```

## 6) Define the Operator Reconciling Logic

```
$ vim pkg/controller/app/app_controller.go
```

```
// Watch for changes to primary resource App
err = c.Watch(&source.Kind{Type: &cachev1alpha1.App{}},
&handler.EnqueueRequestForObject{})
if err != nil {
    return err
}

... // Watch for changes to secondary resource Pods and requeue the owner
Memcached
err = c.Watch(&source.Kind{Type: &corev1.Pod{}},
&handler.EnqueueRequestForOwner{
    IsController: true,
    OwnerType:    &cachev1alpha1.App{}},
})
if err != nil {
    return err
}
```

## 7) Create the CRD

```
$ oc create -f deploy/crds/app_v1alpha1_appservice_crd.yaml
```

## 8) Run your Operator locally

```
$ export OPERATOR_NAME=app-operator
```

```
$ operator-sdk up local --namespace myproject
```

## 9) Create your CR and confirm your logic works as expected.

```
$ oc create -f deploy/crds/app_v1alpha1_appservice_cr.yaml
```

# 10) Once satisfied, build your Operator

```
operator-sdk build quay.io/example/app-operator:v.0.0.1
```

# 11) Push the image to a container registry

```
$ docker push quay.io/example/app-operator:v0.0.1
```

## 12) Create the Service Account, Role, and RoleBindings for the Operator

```
$ oc create -f deploy/service_account.yaml  
$ oc create -f deploy/role.yaml  
$ oc create -f deploy/role_binding.yaml
```

# 13) Apply the Operator Deployment Manifest

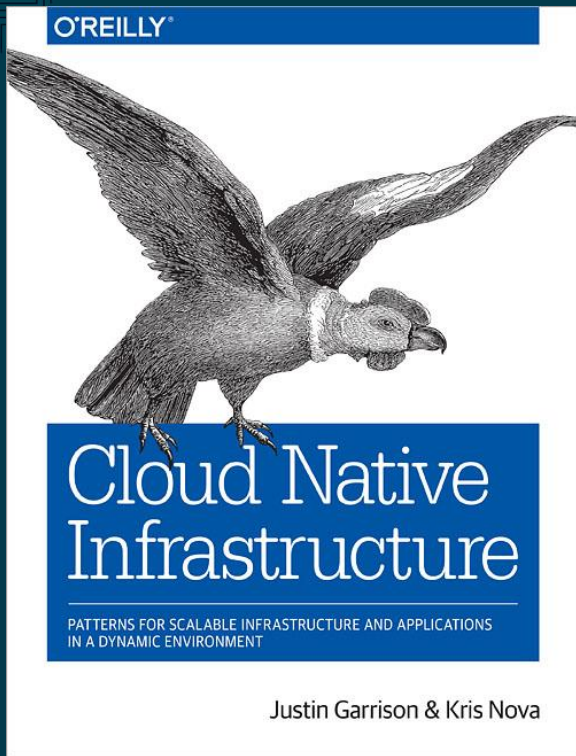
```
$ cat deploy/operator.yaml
```

```
...
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-operator
spec:
  replicas: 1
  ...
  spec:
    containers:
      - name: plex-operator
        image: quay.io/example/app-operator:v0.0.1
  ...
```

```
$ oc create -f deploy/operator.yaml
```

Your Operator is **Alive!**

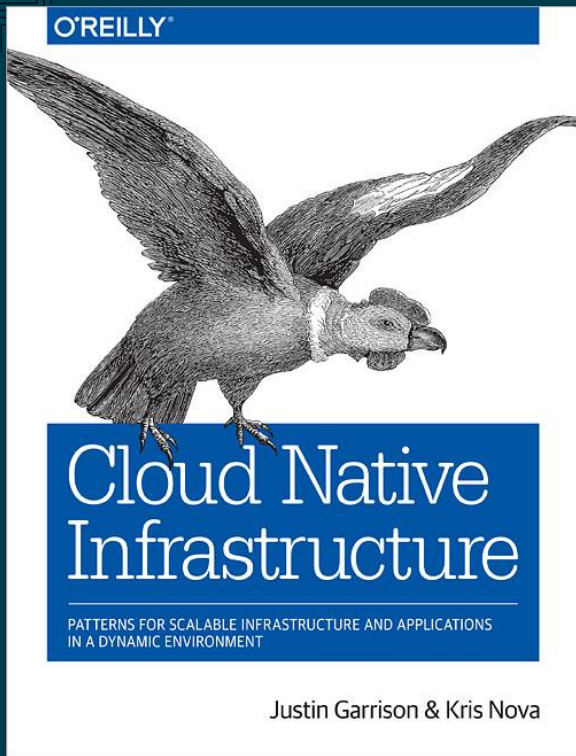
# Controller-Runtime



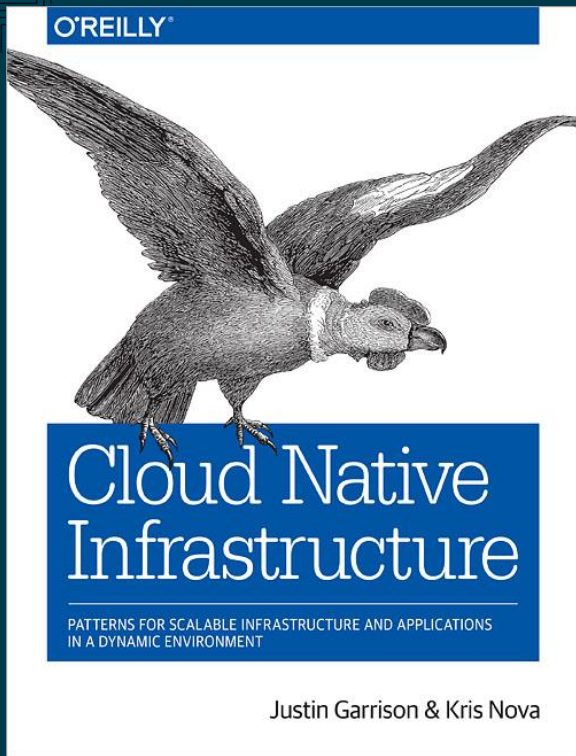
## Chapter 4

### Designing Infrastructure Applications

*The **reconciler pattern** is a software pattern that can be used or expanded upon for managing cloud native infrastructure. The pattern enforces the idea of having two representations of the infrastructure—the first being the actual state of the infrastructure, and the second being the expected state of the infrastructure.*



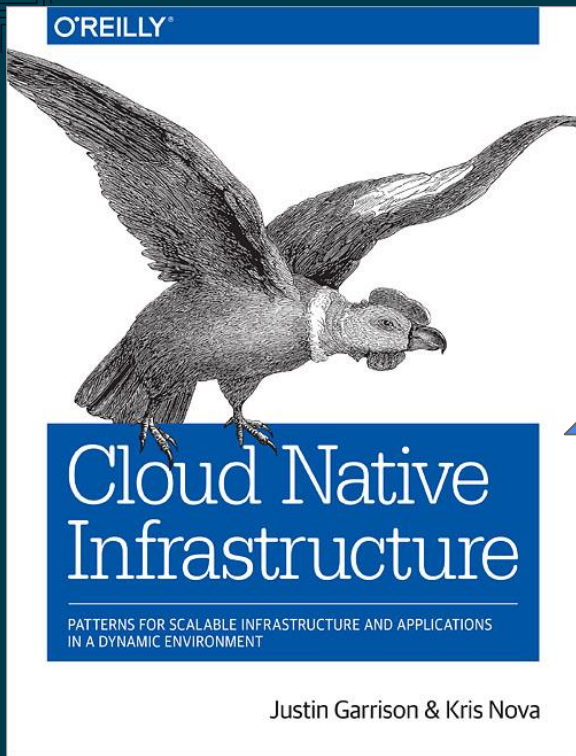
## Four Philosophical Rules of the Reconciler Pattern



## Four Philosophical Rules of the Reconciler Pattern

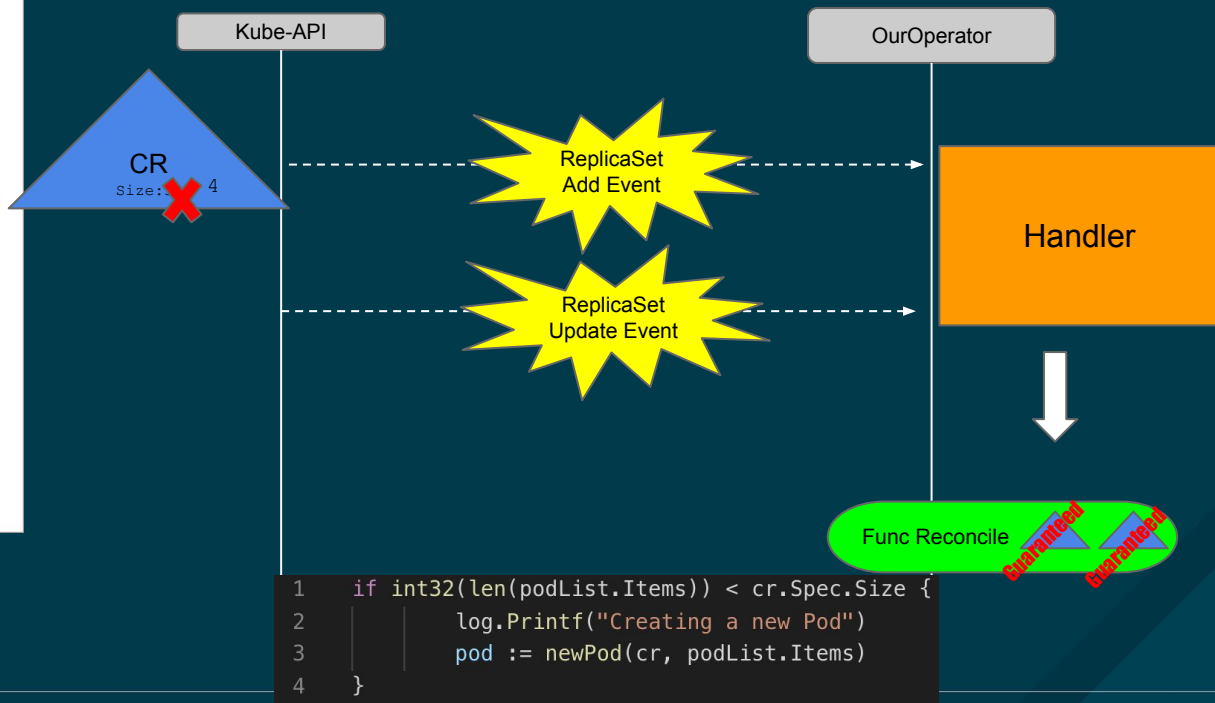
1. Use a data structure for all inputs and outputs.

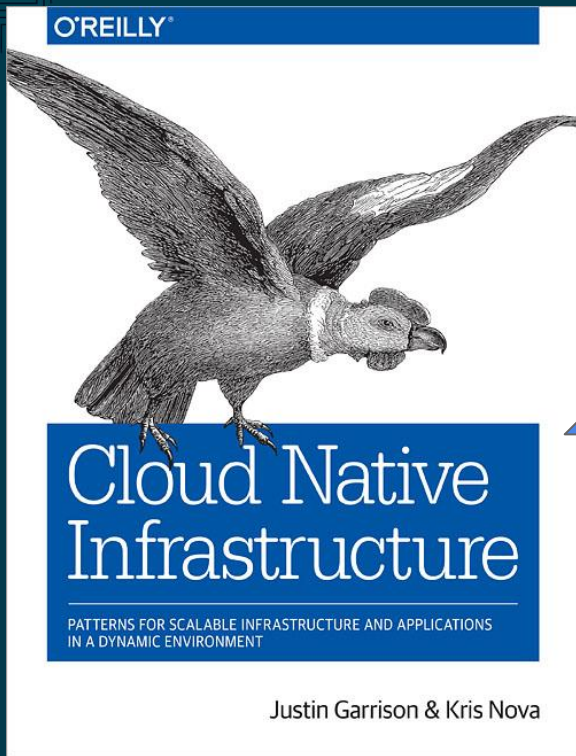
```
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "creationTimestamp": null,
    "name": "test-pd",
    "selfLink": "/api/v1/namespaces/default/pods/test-pd"
  },
  "spec": {
    "containers": [
      {
        "image": "nginx",
        "imagePullPolicy": "Always",
        "name": "test-container",
        "resources": {},
        "terminationMessagePath": "/dev/termination-log",
        "terminationMessagePolicy": "File",
        "volumeMounts": [
          {
            "mountPath": "/var/run/secrets/kubernetes.io/serviceaccount",
```



# Four Philosophical Rules of the Reconciler Pattern

## 2. Ensure That the Data Structure Is Immutable.

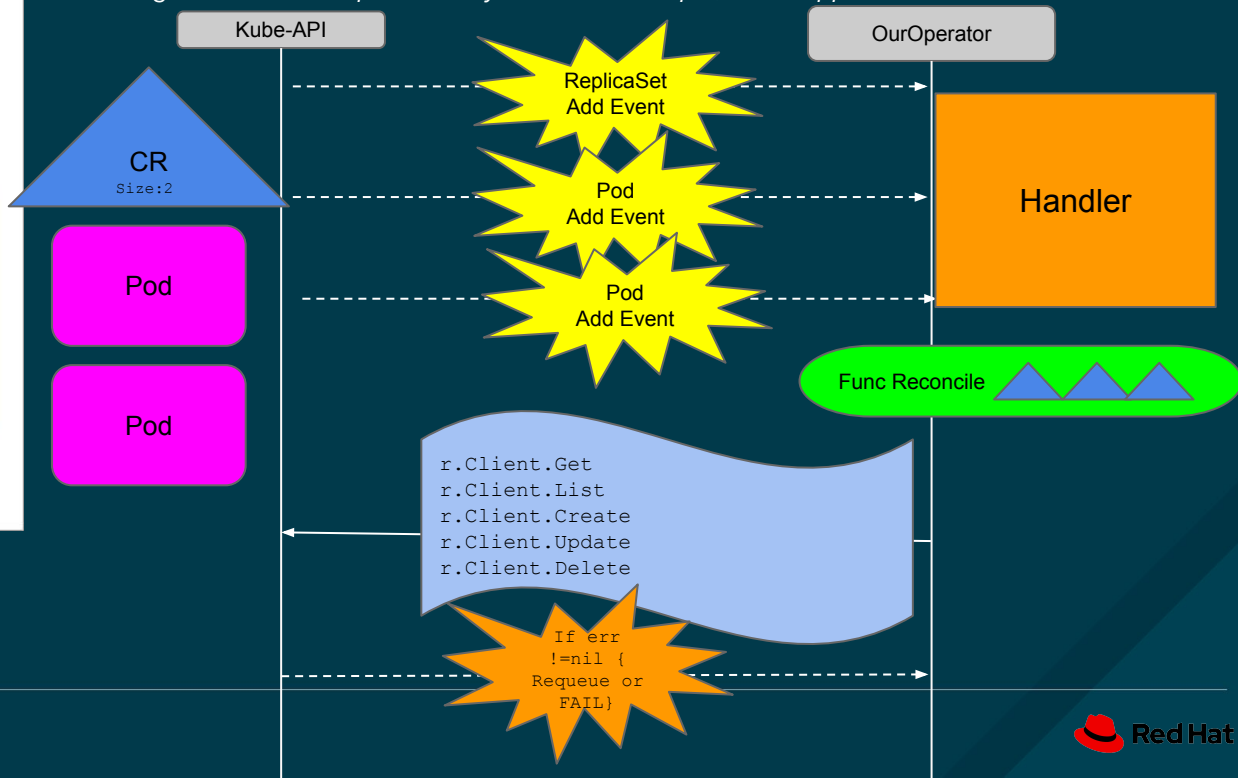


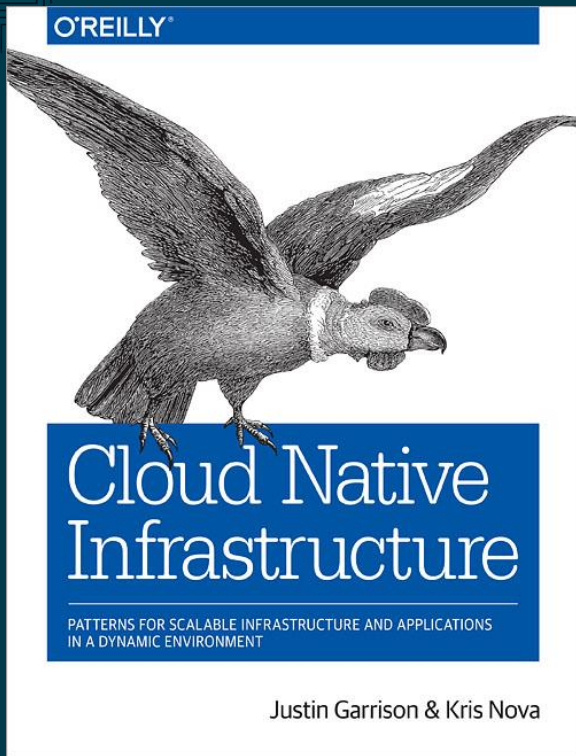


# Four Philosophical Rules of the Reconciler Pattern

## 3. Keep the resource map simple.

*The reconciler pattern should be mapped with a set of resources - reconciler should iterate through the resources procedurally and offer a simple, linear approach.*

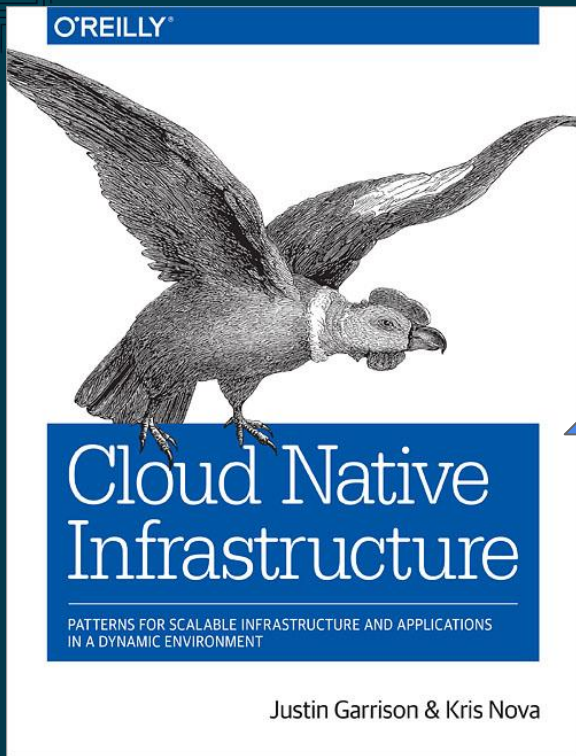




## Four Philosophical Rules of the Reconciler Pattern

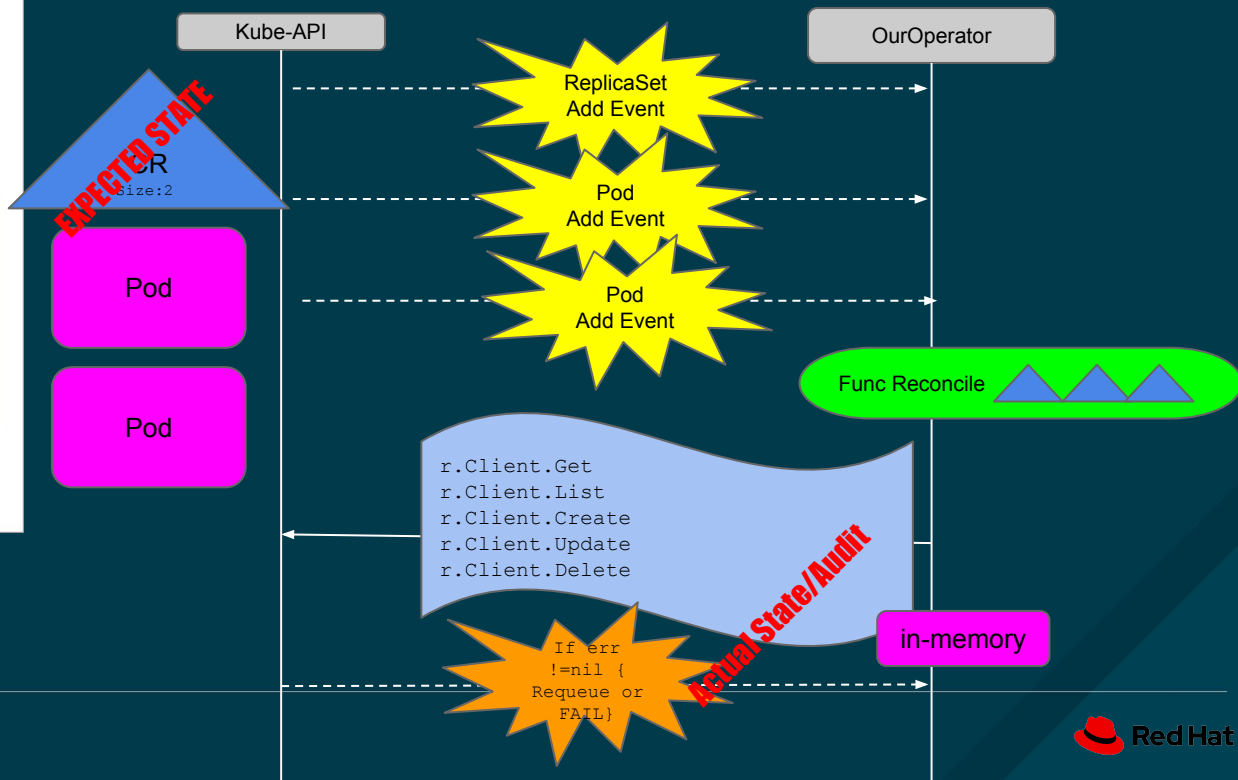
### 4. Make the Actual State Match the Expected State

```
1 while true {  
2   receiveInfoAboutAPIObjects()  
3   synchronizeRealStateToMatchFetchedInfo()  
4 }
```



# Four Philosophical Rules of the Reconciler Pattern

## 4. Make the Actual State Match the Expected State



# Operator-SDK Project Layout

# What is Controller-Runtime?

A set of Go libraries for building Controllers.

It is a subproject of the kubebuilder project by sig-apimachinery.

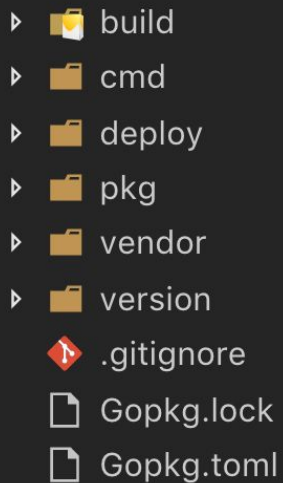
Created to simplify the development of controllers.










- ▲ sigs.k8s.io
  - ▲ controller-runtime
    - example
    - ▲ pkg
      - builder
      - cache
      - client
      - controller
      - envtest
      - event
      - handler
      - internal
      - leaderelection
      - manager
      - patch
      - patterns
      - predicate
      - reconcile
      - recorder
      - runtime
      - source
      - webhook
    - doc.go



# *operator-sdk new* Command

```
$ operator-sdk new app-operator
```



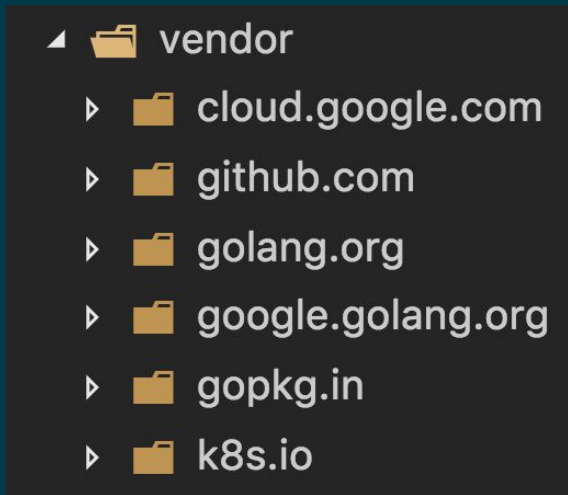
- ▶  build
- ▶  cmd
- ▶  deploy
- ▶  pkg
- ▶  vendor
- ▶  version
- ▶  .gitignore
- ▶  Gopkg.lock
- ▶  Gopkg.toml

# Go Dep Manifests

 `Gopkg.lock` `Gopkg.toml`

- These files are generated when one runs *dep init*.
- *gopkg.toml*: typically handwritten; where one specifies direct dependency based on branch, version, etc.
- *gopkg.lock*: computed outputs (automatically generated).
- **operator-sdk takes care of these for you.**

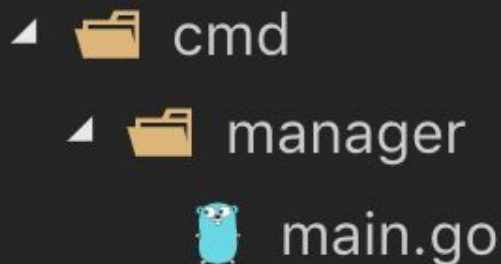
# vendor



Contains the local copies of the external dependencies that satisfy the imports of the project.

- Managed by dep.

# main.go aka the Manager



An executable that wraps one or more Controllers.

- Performs setup steps:
  - Getting a `kubeconfig` to talk to the API
  - Add APIs to the Manager's scheme.
  - Add Controllers to the Manager
  - Initializes Shared Dependencies like Caches and Clients

# main.go aka the Manager

```
62 namespace, err := k8sutil.GetWatchNamespace()
63 if err != nil {
64     log.Error(err, "Failed to get watch namespace")
65     os.Exit(1)
66 }
67
68 // Get a config to talk to the apiserver
69 cfg, err := config.GetConfig()
70 if err != nil {
71     log.Error(err, "")
72     os.Exit(1)
73 }
74
75 ctx := context.TODO()
76
77 // Become the leader before proceeding
78 err = leader.Become(ctx, "your-operator-lock")
79 if err != nil {
80     log.Error(err, "")
81     os.Exit(1)
82 }
83
84 // Create a new Cmd to provide shared dependencies and start components
85 mgr, err := manager.New(cfg, manager.Options{
86     Namespace: namespace,
87     MetricsBindAddress: fmt.Sprintf(":%d", metricsHost, metricsPort),
88 })
89 if err != nil {
90     log.Error(err, "")
91     os.Exit(1)
92 }
```


← Namespace passed via  
flag or variable.

← Kubeconfig passed via  
flag or pod mount  
(in-cluster)

← Creates  
New Manager

# main.go aka the Manager

```
50 // Setup Scheme for all resources
51 if err := apis.AddToScheme(mgr.GetScheme()); err != nil {
52     log.Fatal(err)
53 }
54
55 // Setup all Controllers
56 if err := controller.AddToManager(mgr); err != nil {
57     log.Fatal(err)
58 }
59
60 log.Print("Starting the Cmd.")
61
62 // Start the Cmd
63 log.Fatal(mgr.Start(signals.SetupSignalHandler()))
64 }
```



Add Core Resource  
Type Schemes



Setup All Controllers



Start the Manager

# Dockerfile



build



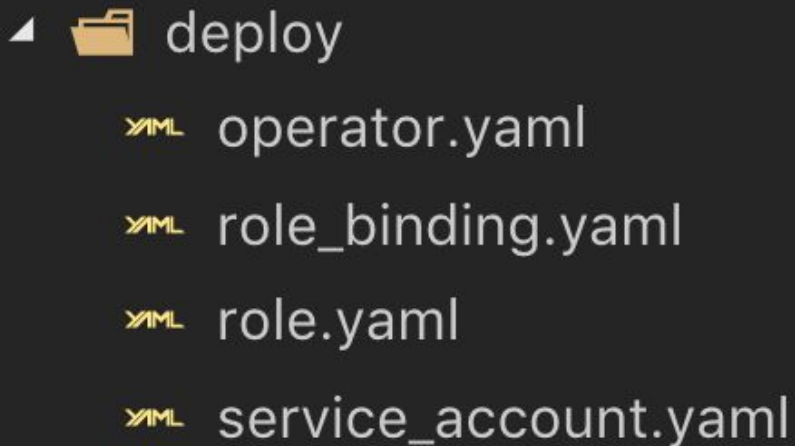
Dockerfile

Used during `operator-sdk build`.

Copies the Operator binary to an alpine-based container

```
1 FROM registry.access.redhat.com/ubi7-dev-preview/ubi-minimal:7.6
2
3 ENV OPERATOR=/usr/local/bin/your-operator \
4     USER_UID=1001 \
5     USER_NAME=your-operator
6
7 # install operator binary
8 COPY build/_output/bin/your-operator ${OPERATOR}
9
10 COPY build/bin /usr/local/bin
11 RUN /usr/local/bin/user_setup
12
13 ENTRYPOINT ["/usr/local/bin/entrypoint"]
14
15 USER ${USER_UID}
```

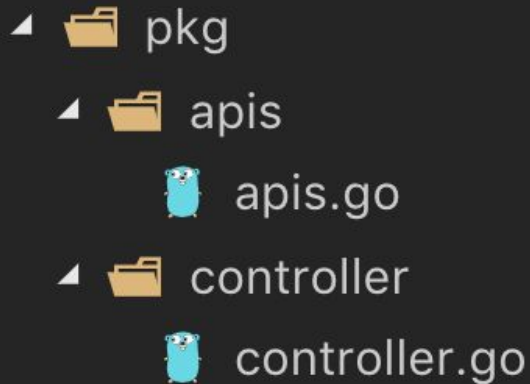
# deploy



```
└─ deploy
  └─ operator.yaml
  └─ role_binding.yaml
  └─ role.yaml
  └─ service_account.yaml
```

A generic set of kubernetes manifests for deploying the operator on a Kubernetes cluster.

# pkg/apis & pkg/controller



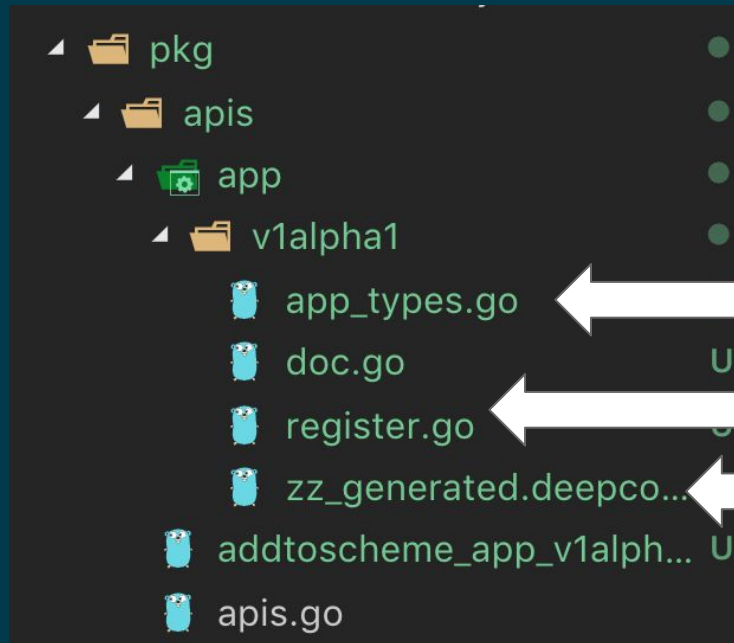
```
└─ pkg
   └─ apis
       └─ apis.go
   └─ controller
       └─ controller.go
```

Defines Custom API type(s) and Controller(s).

- Populated when running `operator-sdk add api` or `operator-sdk add controller`.

# *operator-sdk add* Command

```
$ operator-sdk add api --api-version=app.example.com/v1alpha1 --kind=App
```

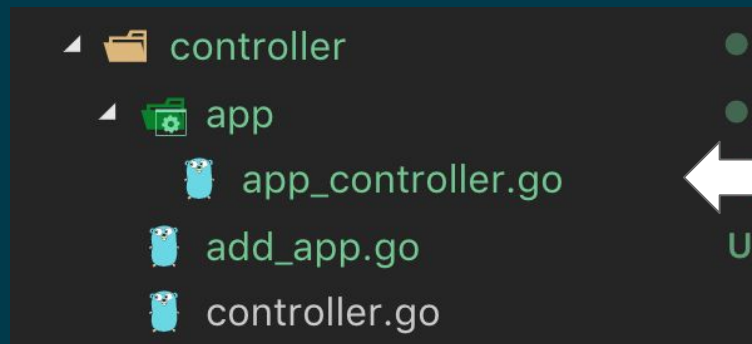


# \*\_types.go

```
10 // AppSpec defines the desired state of App
11 type AppSpec struct {
12     // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
13     // Important: Run "operator-sdk generate k8s" to regenerate code after modifying this file
14 }
15
16 // AppStatus defines the observed state of App
17 type AppStatus struct {
18     // INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
19     // Important: Run "operator-sdk generate k8s" to regenerate code after modifying this file
20 }
```

# *operator-sdk add* Command

```
$ operator-sdk add controller --api-version=app.example.com/v1alpha1 --kind=App
```



Reconciler/Operator Logic.

# *app\_controller.go*

```
75 // Reconcile reads that state of the cluster for a App object and makes changes based on the state read
76 // and what is in the App.Spec
77 // TODO(user): Modify this Reconcile function to implement your Controller logic. This example creates
78 // a Pod as an example
79 // Note:
80 // The Controller will requeue the Request to be processed again if the returned error is non-nil or
81 // Result.Requeue is true, otherwise upon completion it will remove the work from the queue.
82 func (r *ReconcileApp) Reconcile(request reconcile.Request) (reconcile.Result, error) {
83 |     log.Printf("Reconciling App %s/%s\n", request.Namespace, request.Name)
```