KubeCon | CloudNativeCon

Europe 2019

Writing Operators with Kubebuilder v2

Who Am I?

Solly Ross (@directxman12)

Software Engineer on GKE and KubeBuilder Maintainer

My mission is to make writing Kubernetes extensions less arcane

First of all, what’s an Operator?

A **controller** is a loop that reads *desired state* (“spec”), *observed cluster state* (others’ “status”), and *external state*, and the *reconciles* cluster state and external state with the desired state, writing any observations down (to our own “status”).

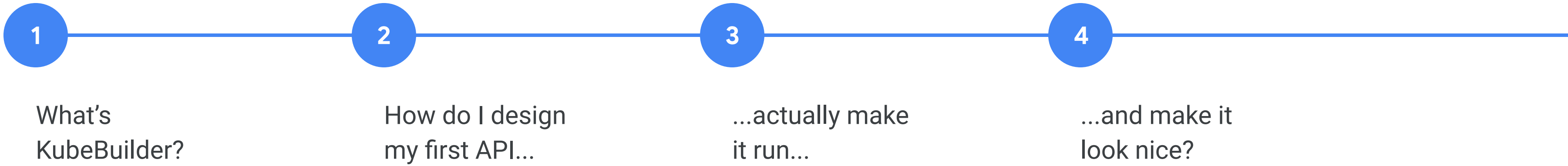
All of Kubernetes functions on this model.

An **operator** is a controller that encodes *human operational knowledge*: how do I run and manage a *specific piece of complex software*.

All operators are controllers, but not all controllers are operators.

So, how’s this going to work?

A 4-part miniseries...



1

What's KubeBuilder?

2

How do I design my first API...

3

...actually make it run...

4

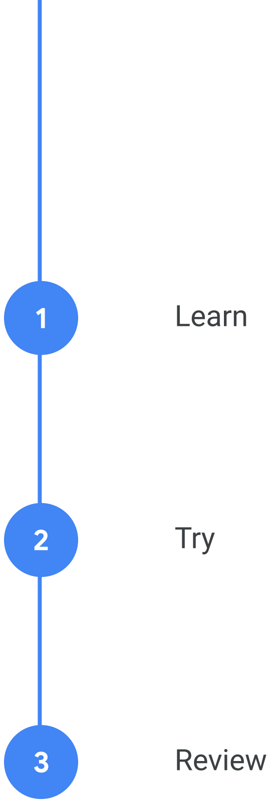
...and make it look nice?

...with 3-act episodes

Learn the general process of things from slides

Try building things yourself based on the goal objects

Review my solution from the Git repo



1 Learn

2 Try

3 Review

What’s KubeBuilder?

Building Blocks + Opinions

KubeBuilder is a set of tooling and opinions how about how to structure custom controllers and operators, built on top of...

Controller-runtime, which contains libraries for building the controller part of your operator, and...

Controller-tools, which contains tools for generating CustomResourceDefinitions for your operator

So, what are we building

We’ll be building a **Guestbook Operator**, along the lines of the guestbook tutorial (https://cloud.google.com/kubernetes-engine/docs/tutorials/guestbook).

The Guestbook contains two components: a **frontend** and a **Redis instance**.

We’ll need to manage and deploy both for the app to work, and we’ll want to **expose** the frontend via a service.

Check out the `goal/` directory if you want to see all the objects we’ll need to create.

How do I get started?

```
~ $ wget https://go.kubebuilder.io/dl/2.0.0-alpha.1/<linux-or-darwin> # and extract
```

```
~ $ git clone https://github.com/directxman12/kubebuilder-workshops /tmp/reference --branch start
```

```
~/$GOPATH/src/proj $ kubebuilder init --project-version 2 --domain <your-domain-here>
```

* See also https://cloud.google.com/kubernetes-engine/docs/tutorials/guestbook

What did we just do?

Initialize a new KubeBuilder **project**

Initialize a new **Go module** for our project

Generate **deployment** config for running in Kubernetes

Configure the **API groups** suffix (`foo` → `foo.metamagical.io`)

How do I design my first API?

What is an API, but a complicated pile of YAML?

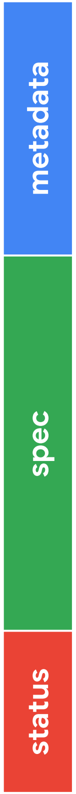
Spec + **Status** + **Metadata** + **List**

Spec holds desired state

Status holds observed state

Metadata holds name/namespace/etc

List holds many objects



```
apiVersion: v1
kind: Pod
metadata:
  name: my-app
  namespace: default
  ...
spec:
  containers:
  - args: [sh]
    image: gcr.io/bowei-gke/udptest
    imagePullPolicy: Always
    name: client
    ...
  dnsPolicy: ClusterFirst
...
status:
  podIP: 10.8.3.11
  ...
```

Cool, but what does that actually mean?

```
// +kubebuilder:object:root=true
// +kubebuilder:subresource:status

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

    Spec   GuestBookSpec   `json:"spec,omitempty"`
    Status GuestBookStatus `json:"status,omitempty"`
}

// +kubebuilder:object:root=true
// +kubebuilder:subresource:status

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

type GuestBookSpec struct { /* MORE STUFF HERE */ }

type GuestBookStatus struct {
    // MORE STUFF HERE

    Conditions []StatusCondition `json:"conditions"`
}
```

Cool, but what does that actually mean?

The **root** object holds the spec, status, and metadata

It’s **list** holds multiple root objects.

```
// +kubebuilder:object:root=true
// +kubebuilder:subresource:status

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

    Spec   GuestBookSpec   `json:"spec,omitempty"`
    Status GuestBookStatus `json:"status,omitempty"`
}

// +kubebuilder:object:root=true
// +kubebuilder:subresource:status

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

type GuestBookSpec struct { /* MORE STUFF HERE */ }

type GuestBookStatus struct {
    // MORE STUFF HERE

    Conditions []StatusCondition `json:"conditions"`
}
```

Cool, but what does that actually mean?

The **spec** holds some desired state.

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

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

    Spec   GuestBookSpec   `json:"spec,omitempty"`
    Status GuestBookStatus `json:"status,omitempty"`
}

// +kubebuilder:object:root=true
// +kubebuilder:subresource:status

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

type GuestBookSpec struct { /* MORE STUFF HERE */ }

type GuestBookStatus struct {
    // MORE STUFF HERE

    Conditions []StatusCondition `json:"conditions"`
}
```

Cool, but what does that actually mean?

The **status** holds some observed state, and status conditions.

Status Conditions let us communicate object health to the user.

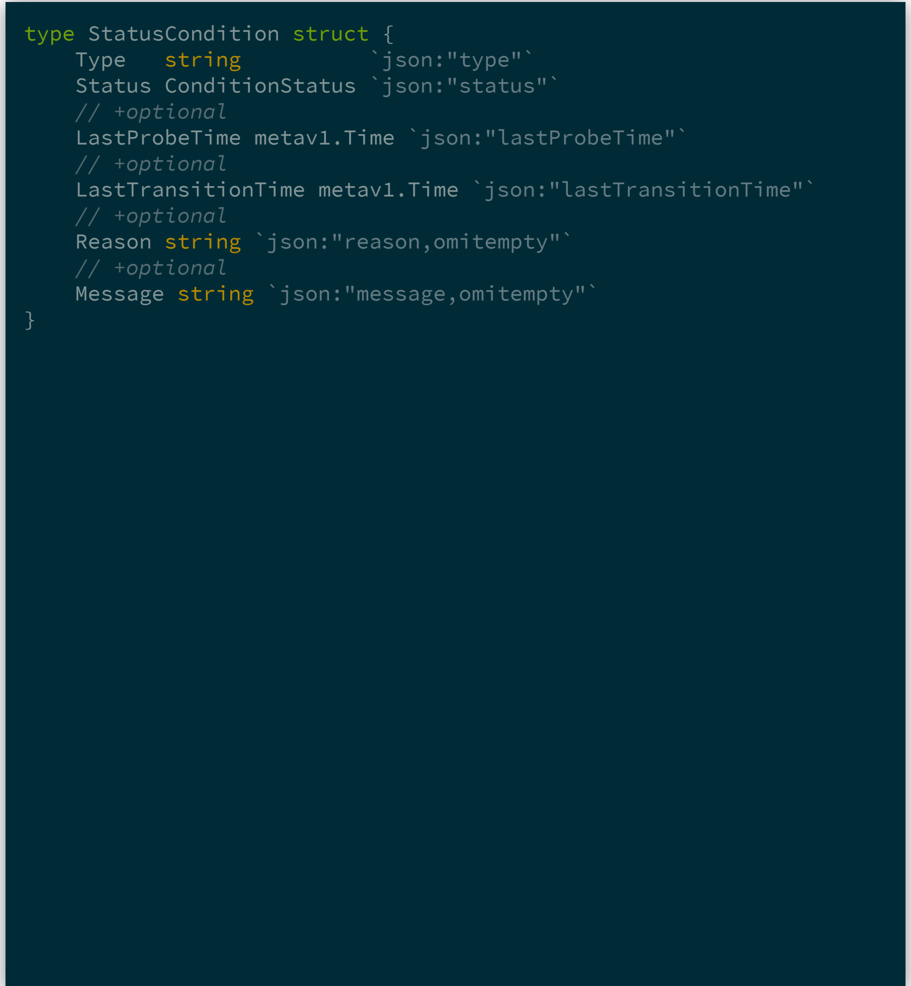
```go
// +kubebuilder:object:root=true
// +kubebuilder:subresource:status

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

    Spec   GuestBookSpec   `json:"spec,omitempty"`
    Status GuestBookStatus `json:"status,omitempty"`
}

// +kubebuilder:object:root=true
// +kubebuilder:subresource:status

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

type GuestBookSpec struct { /* MORE STUFF HERE */ }

type GuestBookStatus struct {
     // MORE STUFF HERE

     Conditions []StatusCondition `json:"conditions"`
}
```

Rules of an API Type

Fields must have JSON tags in **`camelCase`**

Fields may be

string
int32, resource.Quantity (fixed-point)
[]byte
bool
structs
slices
pointers (for optional data)



```go
type StatusCondition struct {
    Type   string          `json:"type"`
    Status ConditionStatus `json:"status"`
    // +optional
    LastProbeTime metav1.Time `json:"lastProbeTime"`
    // +optional
    LastTransitionTime metav1.Time `json:"lastTransitionTime"`
    // +optional
    Reason string `json:"reason,omitempty"`
    // +optional
    Message string `json:"message,omitempty"`
}
```

Try It!

```
~/$GOPATH/src/proj $ kubebuilder create api --group webapp --kind GuestBook --version v1
```

Create an **API group** named `webapp.<your-domain>`

Create an **API version** `webapp.<your-domain>/v1`

Add a new **Kind** `GuestBook` to that group, and a controller for it

```
~/$GOPATH/src/proj $ $EDITOR api/v1/guestbook_types.go
```

```
~/$GOPATH/src/proj $ make generate manifests
```

Generate the `runtime.Object` interface and **CustomResourceDefinition manifests**

Review!

```go
type GuestBookSpec struct {
    Frontend FrontendSpec `json:"frontend"`

    // +optional
    RedisName string `json:"redisName,omitempty"`

    UseDNS bool `json:"useDNS"`
}

type FrontendSpec struct {
    // +optional
    Resources corev1.ResourceRequirements `json:"resources"`
    // +optional
    ServingPort int32 `json:"servingPort,omitempty"`

    // +optional
    Replicas *int32 `json:"replicas,omitempty"`
}

type GuestBookStatus struct {
    URL string `json:"url,omitempty"`

    Conditions []StatusCondition `json:"conditions,omitempty"`
}
```

Review!

```go
type RedisSpec struct {
    FollowerReplicas *int32 `json:"followerReplicas,omitempty"`
}

type RedisStatus struct {
    Conditions []StatusCondition `json:"conditions,omitempty"`

    LeaderService   string `json:"leaderService"`
    FollowerService string `json:"followerService"`
}
```

Interlude: What’s this about Groups, Versions, and Kinds?

An **API group** is a collection of related API types.

We call each API type a **Kind**.

Each API group has one or more **API versions**, which let us change the API over time

Each Kind is used in at least one **Resource**, which is a “use” the Kind in the API (generally, these are one-to-one with Kinds). They’re referred to in lower-case.

Each Go type corresponds to a particular **Group-Version-Kind**.

But how do I actually make it run?

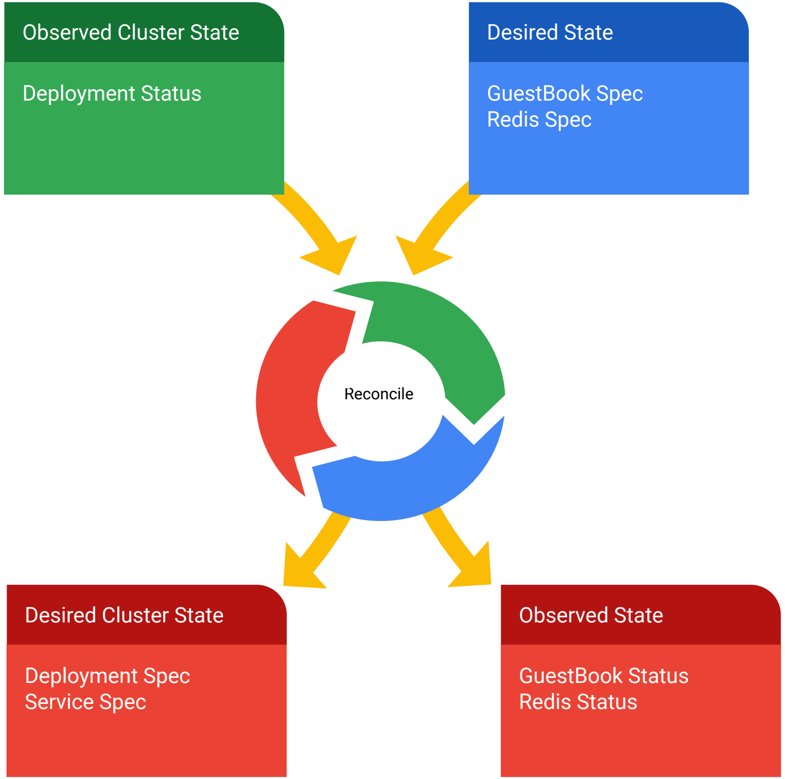
Read, Reconcile, Repeat

Read our root object

Fetch other objects we care about

Ensure those objects are in the right state

Write our root status out



Observed Cluster State

Deployment Status

Desired State

GuestBook Spec
Redis Spec

Reconcile

Desired Cluster State

Deployment Spec
Service Spec

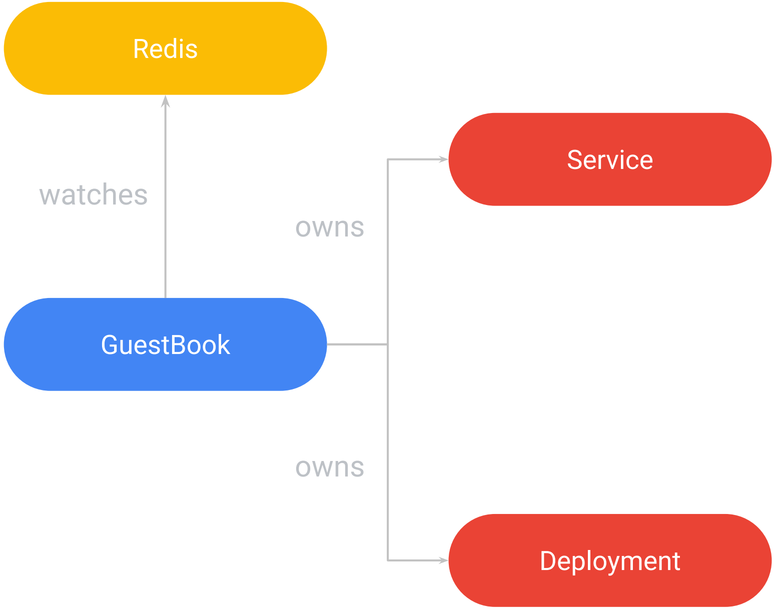
Observed State

GuestBook Status
Redis Status

One Kind to rule them all

Each **reconciler** (control loop) functions on a *single* Kind.

This kind may **own** other Kinds that it creates, and may otherwise **watch** kinds that it has relationships with.



Clients and Schemes and Requests, oh my!

Each reconciler takes a **request**, and returns a **result** and **error**

Requests can use **client.Get** to turn the request into an actual object, and **CreateOrUpdate** to ensure that an object is up-to-date.

Clients use a **Scheme** to associate Go types with Kinds. All types referenced in a reconciler need to be added to the Scheme with **`<api-package>.AddToScheme`** in `main.go`

When creating objects, make sure to mark that your object owns them with **`SetControllerReference`**

Errors and Results can be used to trigger **requeues**. The reconciler will also be called when in the cluster updates.

Cool, but what does that actually mean?

Fetch our GuestBook

Ensure desired state

Update status with observed state

```go
func (r *GuestBookReconciler) Reconcile(req ctrl.Request)
        (ctrl.Result, error) {
    ctx := context.Background()
    log := r.Log.WithValues("guestbook", req.NamespacedName)

    // first, fetch our guestbook
    var app webappv1.GuestBook
    if err := r.Get(ctx, req.NamespacedName, &app); err != nil {
        // it might be not found if this is a delete request
        if ignoreNotFound(err) == nil {
            return ctrl.Result{}, nil
        }
        log.Error(err, "unable to fetch guestbook")
        return ctrl.Result{}, err
    }

    // process the request, make some changes to the cluster,
    // set some status on `app`, etc

    // update status, since we probably changed it above
    if err := r.Status().Update(ctx, &app); err != nil {
        log.Error(err, "unable to update guestbook status")
        return ctrl.Result{}, err
    }

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

Try It (Briefly)!

```
~/$GOPATH/src/proj $ $EDITOR controllers/guestbook_controller.go
```

```
~/$GOPATH/src/proj $ kubectl create -f config/crd/bases
```

```
~/$GOPATH/src/proj (terminal 1) $ make run
```

```
~/$GOPATH/src/proj (terminal 2) $ kubectl create -f config/samples && kubectl describe guestbooks
```

Let’s see if we can make our controller set a status field on our CRD.

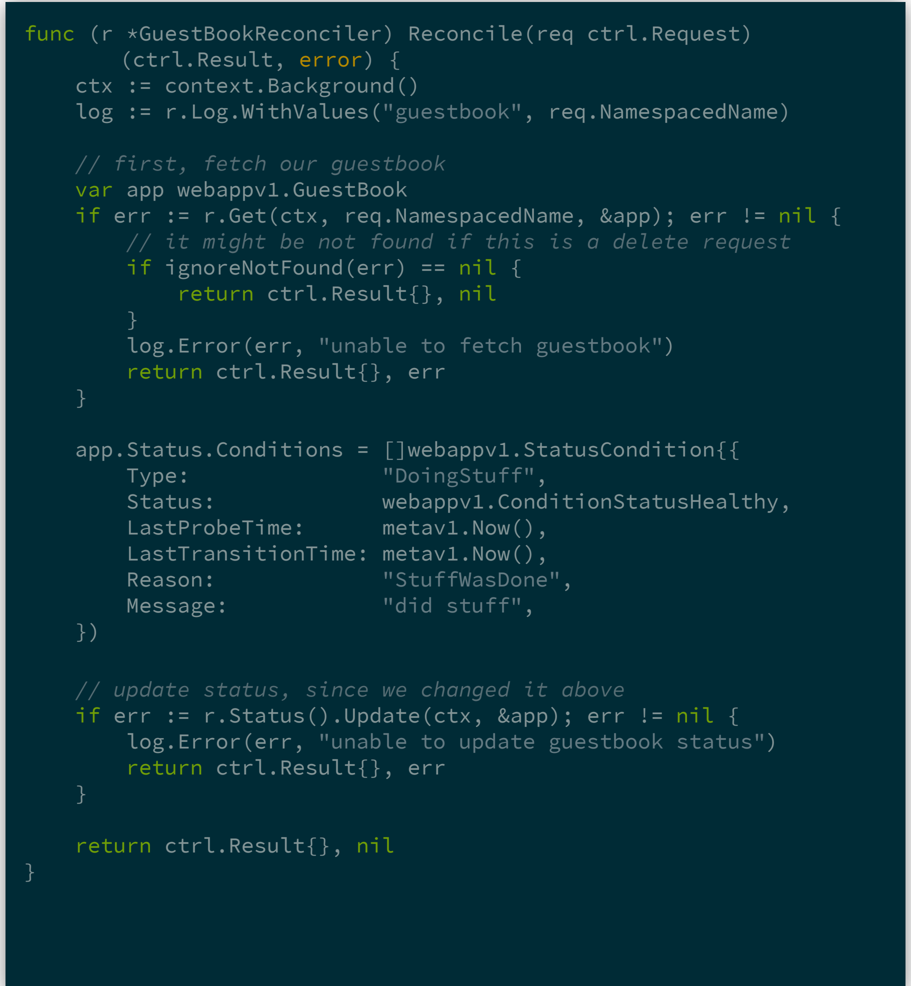
Publish our CRDs to the API server and run our **controller manager** locally against the API server

Create and **fetch** our guestbook

Review (Briefly)!

Fetch our GuestBook

Update status some status condition



```go
func (r *GuestBookReconciler) Reconcile(req ctrl.Request)
        (ctrl.Result, error) {
    ctx := context.Background()
    log := r.Log.WithValues("guestbook", req.NamespacedName)

    // first, fetch our guestbook
    var app webappv1.GuestBook
    if err := r.Get(ctx, req.NamespacedName, &app); err != nil {
        // it might be not found if this is a delete request
        if ignoreNotFound(err) == nil {
            return ctrl.Result{}, nil
        }
        log.Error(err, "unable to fetch guestbook")
        return ctrl.Result{}, err
    }

    app.Status.Conditions = []webappv1.StatusCondition{{
        Type:               "DoingStuff",
        Status:             webappv1.ConditionStatusHealthy,
        LastProbeTime:      metav1.Now(),
        LastTransitionTime: metav1.Now(),
        Reason:             "StuffWasDone",
        Message:            "did stuff",
    }}

    // update status, since we changed it above
    if err := r.Status().Update(ctx, &app); err != nil {
        log.Error(err, "unable to update guestbook status")
        return ctrl.Result{}, err
    }

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

Idempotency

Reconcilers should be **idempotent**: reconciling on an object that needs nothing done should have no side effects

Always take actions based on the observed cluster and external state, *not* the event that triggered a reconciliation.

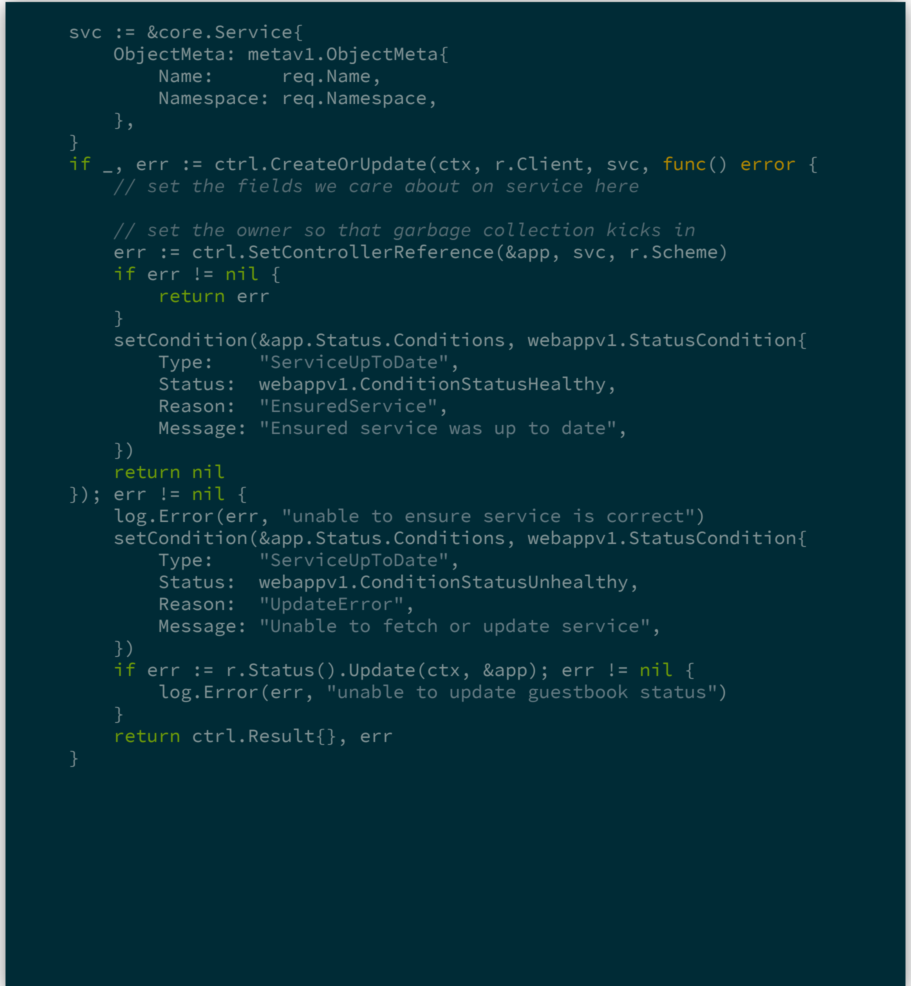
Prefer writing logic in terms of “**ensure this is correct”**, not specifically create or update.

Use **owner references** to take care of delete for you, so that even after uninstallation resources get cleaned up.

Cool, but what does that actually mean?

Ensure desired state with **CreateOrUpdate**

Set **StatusConditions** to indicate health



```go
svc := &core.Service{
    ObjectMeta: metav1.ObjectMeta{
        Name:      req.Name,
        Namespace: req.Namespace,
    },
}
if _, err := ctrl.CreateOrUpdate(ctx, r.Client, svc, func() error {
    // set the fields we care about on service here

    // set the owner so that garbage collection kicks in
    err := ctrl.SetControllerReference(&app, svc, r.Scheme)
    if err != nil {
        return err
    }
    setCondition(&app.Status.Conditions, webappv1.StatusCondition{
        Type:    "ServiceUpToDate",
        Status:  webappv1.ConditionStatusHealthy,
        Reason:  "EnsuredService",
        Message: "Ensured service was up to date",
    })
    return nil
}); err != nil {
    log.Error(err, "unable to ensure service is correct")
    setCondition(&app.Status.Conditions, webappv1.StatusCondition{
        Type:    "ServiceUpToDate",
        Status:  webappv1.ConditionStatusUnhealthy,
        Reason:  "UpdateError",
        Message: "Unable to fetch or update service",
    })
    if err := r.Status().Update(ctx, &app); err != nil {
        log.Error(err, "unable to update guestbook status")
    }
    return ctrl.Result{}, err
}
```

Cool, but what does that actually mean?

Add referenced API groups to our **Scheme**

Pass the Scheme to the reconciler

```
import (
    // ...
    appsv1 "k8s.io/api/apps/v1"
    corev1 "k8s.io/api/core/v1"
    // ...
)

func init() {
    corev1.AddToScheme(scheme)
    appsv1.AddToScheme(scheme)
    webappv1.AddToScheme(scheme)
    // +kubebuilder:scaffold:scheme
}

func main() {
    // ...
    err = (&controllers.GuestBookReconciler{
        Client: mgr.GetClient(),
        Log: ctrl.Log.
            WithName("controllers").
            WithName("GuestBook"),
        Scheme: mgr.GetScheme(),
    }).SetupWithManager(mgr)
    // ...
}
```

In case you need it...

Full GoDoc for controller-runtime:
https://godoc.org/sigs.k8s.io/controller-runtime

Example controller for controller-runtime:
https://godoc.org/sigs.k8s.io/controller-runtime#example-package

Try It!

```
~/$GOPATH/src/proj $ $EDITOR controllers/guestbook_controller.go
```

```
~/$GOPATH/src/proj $ kubectl replace -f config/crd/bases
```

```
~/$GOPATH/src/proj $ make run
```

Publish our CRDs to the API server

Run our **controller manager** locally against the API server

Review!

Ensuring state inside CreateOrUpdate
(aim for idempotency)

```go
replicas := int32(1)
if app.Spec.Frontend.Replicas != nil {
    replicas = *app.Spec.Frontend.Replicas
}
depl.Spec.Replicas = &replicas

templ := &depl.Spec.Template

// set a label for our service and deployment
if templ.ObjectMeta.Labels == nil {
    templ.ObjectMeta.Labels = map[string]string{}
}
templ.ObjectMeta.Labels["guestbook"] = req.Name
sel := map[string]string{"guestbook": req.Name}
depl.Spec.Selector = &metav1.LabelSelector{MatchLabels: sel}

// make sure we actually run what we want, though
cont := findOrAddContainer(&templ.Spec.Containers, "frontend")
cont.Name = "frontend"
cont.Image = "gcr.io/google-samples/gb-frontend:v4"

// and again for env
if app.Spec.UseDNS {
    setEnv(cont, "GET_HOSTS_FROM", "dns")
} else {
    setEnv(cont, "GET_HOSTS_FROM", "env")
    setEnv(cont, "REDIS_SLAVE_SERVICE_HOST", redisSvc)
}

// copy resources
for res, val := range app.Spec.Frontend.Resources.Requests {
    cont.Resources.Requests[res] = val
}
for res, val := range app.Spec.Frontend.Resources.Limits {
    cont.Resources.Limits[res] = val
}

// and again for the port
port := findOrAddPort(&cont.Ports, "http")
port.Name = "http"
port.ContainerPort = 80
```

Interlude:
Server-Side Apply?

Set *all* fields that we care about, server computes the appropriate changes.

Coming soon to a cluster near you (alpha in Kubernetes 1.14)!

```
svc := &core.Service{
    ObjectMeta: metav1.ObjectMeta{
        Name:      req.Name,
        Namespace: req.Namespace,
    },
    Spec: core.ServiceSpec{
        Selector: map[string]string{"guestbook": req.Name},
        Ports: []core.ServicePort{{Name: "http", Port: port}},
        Type: "LoadBalancer",
    },
}
err := ctrl.SetControllerReference(&app, svc, r.Scheme)
if err != nil {
    return err
}
if err := r.Apply(ctx, svc); err != nil {
    log.Error(err, "unable to ensure service is correct")
    setCondition(&app.Status.Conditions, webappv1.StatusCondition{
        Type:    "ServiceUpToDate",
        Status:  webappv1.ConditionStatusUnhealthy,
        Reason:  "UpdateError",
        Message: "Unable to fetch or update service",
    })
    if err := r.Status().Update(ctx, &app); err != nil {
        log.Error(err, "unable to update guestbook status")
    }
}
```

Now how do I make it nice and usable?

Printer Columns

Expose extra information in `kubectl get`, to feel like built-in resources:

```
~/$GOPATH/src/proj $ kubectl get guestbooks
NAME               URL                          DEPLOYMENT   SERVICE
guestbook-sample   http://35.238.150.235:8080   Healthy      Healthy
```

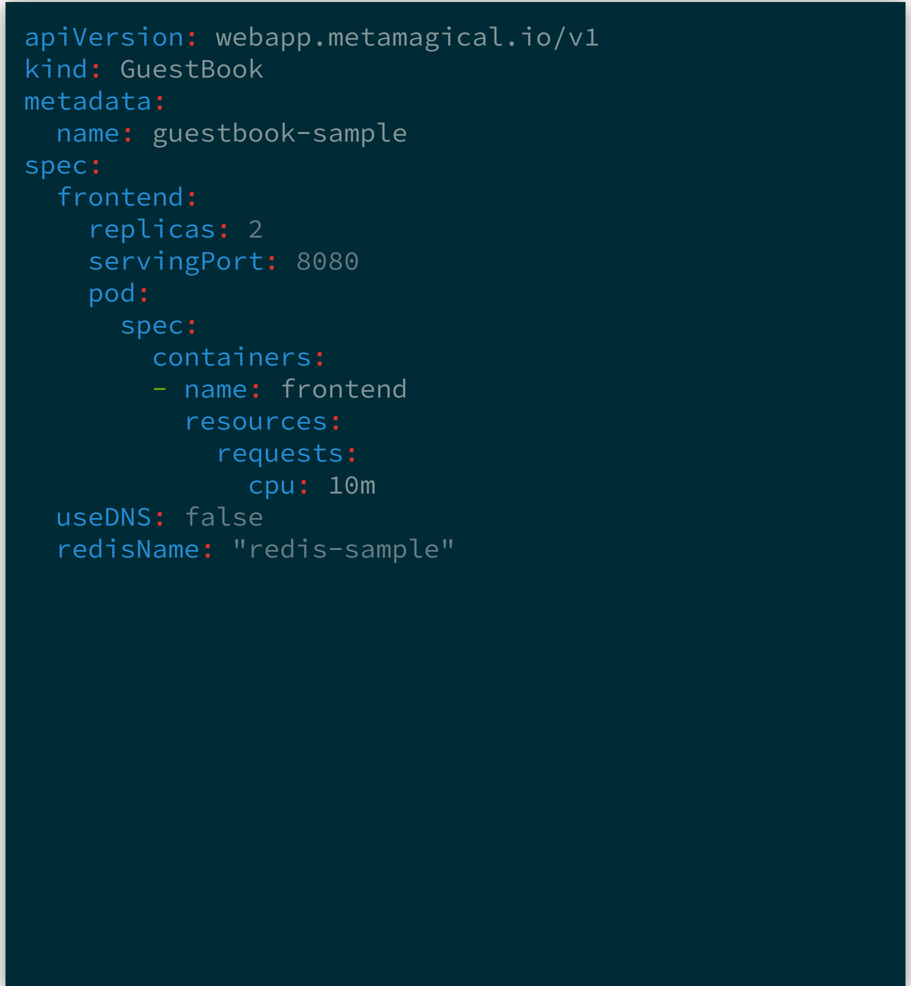
Uses “markers” in the source on the Go type (closest non-godoc comment):

```go
// +kubebuilder:printcolumn:name=URL,type=string,JSONPath=".status.url",description="GuestBook Frontend URL"


// GuestBook is the Schema for the guestbooks API
type GuestBook struct { … }
```

Samples

`config/samples` contains sample objects for all of your CRDs.

Fill these in to provide samples to your users, and to test out your controller:



```yaml
apiVersion: webapp.metamagical.io/v1
kind: GuestBook
metadata:
  name: guestbook-sample
spec:
  frontend:
    replicas: 2
    servingPort: 8080
    pod:
      spec:
        containers:
        - name: frontend
          resources:
            requests:
              cpu: 10m
  useDNS: false
  redisName: "redis-sample"
```

Try It!

```
~/$GOPATH/src/proj $ $EDITOR api/v1/guestbook_types.go
```

```
~/$GOPATH/src/proj $ make manifests && kubectl replace -f config/crd/bases
```

```
~/$GOPATH/src/proj $ $EDITOR config/samples/guestbook/*.yaml && kubectl create -f config/samples
```

```
~/$GOPATH/src/proj $ kubectl get guestbooks
```

Add printer columns and **Replace** our CRDs

Edit and **Create** our sample

List the objects in action

Interlude: Kustomize

Kustomize is a tool for **declarative configuration management**.

Kubebuilder uses it to composite optional patches when running the manager.

We'll have to install it (we'll put in in `/tmp` here, but you can put it elsewhere):

```
~/$GOPATH/src/proj $ curl -sL -o kustomize
https://go.kubebuilder.io/kustomize/<linux-or-darwin>/amd64
```

```
~/$GOPATH/src/proj $ mkdir -p /tmp/kustomize && mv kustomize /tmp/kustomize
```

```
~/$GOPATH/src/proj $ EXPORT PATH=$PATH:/tmp/kustomize
```

All together now!

```
~/$GOPATH/src/proj $ make docker-build IMG=gcr.io/<your-project>/controller
```

```
~/$GOPATH/src/proj $ make docker-push IMG=gcr.io/<your-project>/controller
```

```
~/$GOPATH/src/proj $ $EDITOR config/default/manager_image_patch.yaml && make deploy
```

```
~/$GOPATH/src/proj $ $BROWSER $(kubectl get guestbooks -o jsonpath='.items[].status.url')
```

Build and **push** our controller manager to GCR

Replace `IMAGE_URL` and **Run** out controller manager as a pod on the cluster

View the running guest book in your browser

If you’re feeling precocious...

Check out the reference repository[1] for additional tasks, like defaulting webhooks.

Fill in support for launching Redis, if you haven’t already!

[1] https://github.com/directxman12/kubebuilder-workshop

For Reference...

KubeBuilder Repository + Samples: https://sigs.k8s.io/kubebuilder

Controller-Runtime GoDocs: https://godoc.org/sigs.k8s.io/controller-runtime

KubeBuilder Book: https://book.kubebuilder.io

Workshop Repo: https://github.com/directxman12/kubebuilder-workshops