Zero To Operator

In ~~2.2 seconds~~ 90 minutes (give-or-take)

Who am I?

Solly Ross (@directxman12 / metamagical.dev)

Software Engineer on GKE and KubeBuilder Maintainer

My mission is to make writing Kubernetes extensions less arcane

First of all, what’s an Operator?

Okay, please pre-emptively retrive your ~~pitchforks~~ bikeshedding keyboards.

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.

How’s this going to work?

We’ll **learn** about the concepts…

…then **code** the implementation…

…and **try it out** against an actual cluster

How’s this going to work?

Scaffold & Design

Types

Behavior

Launching

Without further ado…

Without further ado…

Wait a minute, I feel like I’ve seen this somewhere before!

Yeah, but we’re more declarative now!

Alright, all set with the ado

Let’s talk about KubeBuilder, scaffolding, and CRD design!

What’s KubeBuilder?

and how do I captialize it?

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, etc for your operator

Enough talk, let’s build something!

We’ll be building an operator for a simple bespoke application: the **Kubernetes Guestbook example**.

The guestbook has two components: a **frontend** PHP app and a **Redis** instance (the backend).

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

Enough talk, let's build something!

You can follow along with the tutorial at pres.metamagical.dev/kubecon-us-2019/code.

Check out the `goals/` directory to see what we're aiming to produce.

What do we need?

KubeBuilder, plus ~~an unlimitted supply of Xena tapes and hot pockets~~ Go 1.12+ (and probably git):

```
~ $ wget https://go.kubebuilder.io/dl/2.1.0/<linux-or-darwin> # and extract
~ $ git clone https://github.com/directxman12/kubebuilder-workshops /tmp/workshop --branch kubecon-us-2019
```

* See also https://kubernetes.io/docs/tutorials/stateless-application/guestbook/

Go-go-gadget KubeBuilder!

Initialize a new **Go module** to hold the project

Initialize a new KubeBuilder **project**

Generate a **Deployment** for running the controller manager in Kubernetes

Configure the **API Group** suffix
(`webapp --> webapp.metamagical.dev`[1])

```
$ alias k=kubectl # I'm lazy ;-)
$ cd ~/kubecon-project
$ go mod init mykubecon
$ kubebuilder init --domain <your-domain-here>
```

1. metamagical.dev is my own domain; please use yours here 🙂

Groups and Versions and Kinds, oh my!

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**.

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

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

Spec holds desired state

Status holds observed states

Metadata holds name/namespace/etc

List holds many objects

METADATA

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-app
  namespace: default
  ...
```

SPEC

```yaml
spec:
  containers:
  - args: [sh]
    image: gcr.io/bowei-gke/udptest
    imagePullPolicy: Always
    name: client
    ...
  dnsPolicy: ClusterFirst
  ...
```

STATUS

```yaml
status:
  podIP: 10.8.3.11
  ...
```

Practically speaking…

```
$ kubebuilder create api \
  --group webapp \
  --kind GuestBook \
  --version v1
```

`api/v1/guestbook_types.go` (before modification)

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

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

// +kubebuilder:object:root=true

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

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

Practically speaking…

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

The **list** holds multiple root objects.

We use **marker comments** [2] like `// +marker` to indicate additional metadata about the types

On the root object, we can use markers to specify data about how the CRD behaves in general. Here, we specify that:

- we're using the **status subresource** (`// +kubebuilder:subresource:status`)
- we want custom **print columns** to show up in `kubectl get` output (`// +kubebuider:printcolumn`)

`api/v1/guestbook_types.go` (root object & list)

```go
// +kubebuilder:object:root=true
// +kubebuilder:subresource:status
// +kubebuilder:printcolumn:JSONPath=".status.url",name=URL,typ
// +kubebuilder:printcolumn:JSONPath=".spec.frontend.replicas",

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

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

2. https://book.kubebuilder.io/reference/markers.html

Practically speaking…

The **spec** holds some desired state.

Each field has a **json tag** specifying the field name in the JSON/YAML[3].

On spec (and status), **markers** specify metadata about types *and* fields, such as:

validation (`// +kubebulder:validation:xyz`)

default values for the server to apply, without needing a webhook (`// +kubebuilder:default`)

whether a field is **optional or required** (`// +optional`)

3. generally, it should be the same as the field name, but in `camelCase` instead of `PascalCase`.

`api/v1/guestbook_types.go` (spec)

```go
type GuestBookSpec struct {
    Frontend FrontendSpec `json:"frontend"`
    RedisName string `json:"redisName,omitempty"`
}

type FrontendSpec struct {
    // +optional
    Resources corev1.ResourceRequirements `json:"resources"`

    // +optional
    // +kubebuilder:default=8080
    // +kubebuilder:validation:Minimum=0
    ServingPort int32 `json:"servingPort,omitempty"`

    // +optional
    // +kubebuilder:default=1
    // +kubebuilder:validation:Minimum=0
    Replicas *int32 `json:"replicas,omitempty"`
}
```

Practically speaking…

The **status** holds observed state. **Status should always be recreatable from the state of the world**. Don’t store information here that you don’t care about losing.

We use the same types, structures, and markers from the spec here.

`api/v1/guestbook_types.go` (status)

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

Practically speaking…

We also need similar types for Redis.
Printcolumns left as an excercise to the reader :-)

Notice that:

- we can use **godoc to set API documentation** for our types
- we can separate markers from fields by whitespace to help organize

putz around with `api/v1/redis_types.go`

```go
type RedisSpec struct {
    // +optional
    // +kubebuilder:default=1
    // +kubebuilder:validation:Minimum=0

    // The number of follower instances to run.
    FollowerReplicas *int32 `json:"followerReplicas,omitempty"`
}

type RedisStatus struct {
    // The name of the service created for the Redis leader.
    LeaderService string `json:"leaderService"`
    // The name of the service created for the Redis followers.
    FollowerService string `json:"followerService"`
}
```

A bit more detail on those points…

When we implement Kubernetes APIs, there's a couple things to keep in mind:

We allow generally allow **most Go types**, with a couple exceptions:

floats aren't allowed – use `resource.Quantity` instead [4]

We use **tagged unions** instead of interfaces

When we create optional fields, it's important to think about whether or not we want the zero value to be usuable. When in doubt **use a pointer for optional values**

```go
type DifferentDefaulting struct {
    // (unset)       --> (default): 1
    // 0 == (unset) --> (default): 1

    // +optional
    // +kubebuilder:default=1
    UnstoppableReplicas int32 `json:"unstoppable"`

    // 0               --> (set) 0
    // nil == (unset) --> (default): 1

    // +optional
    // +kubebuilder:default=1
    StoppableReplicas *int32 `json:"stoppable"`
}

type MyUnion struct {
    Type MyUnionType `json:"type"`

    // +optional
    VariantOne string `json:"variantOne,omitempty"`
    // +optional
    VariantTwo bool `json:"variantTwo,omitempty"`
}

// +kubebuilder:validation:Enum=VariantOne;VariantTwo
// We can put validation markers on type aliases too,
// to reuse validation.
type MyUnionType string
```

4. floats don't round trip through different systems without changing, whereas `resource.Quantity` is consistent. You've probably seen Quantities in the resource requirements section of the Pod spec, like `500m`.

Let’s try it out

First, we’ll make sure our sample is all set…

edit `config/samples/webapp_v1_guestbook.yaml`

```yaml
apiVersion: webapp.metamagical.dev/v1
kind: GuestBook
metadata:
  name: guestbook-sample
spec:
  redisName: redis-sample
  frontend:
    resources:
      requests:
        cpu: 80m
    # check that this doesn't work,
    # then delete it
    servingPort: -1
```

Let’s try it out

…then, we’ll actually test it against the cluster!

```
$ make manifests
$ k create -f config/crd/bases
$ k create -f config/samples/webapp_v1_guestbook.yaml
$ k get guestbooks
NAME               URL   DESIRED
guestbook-sample         1
```

Yeah, but how do I make it go?

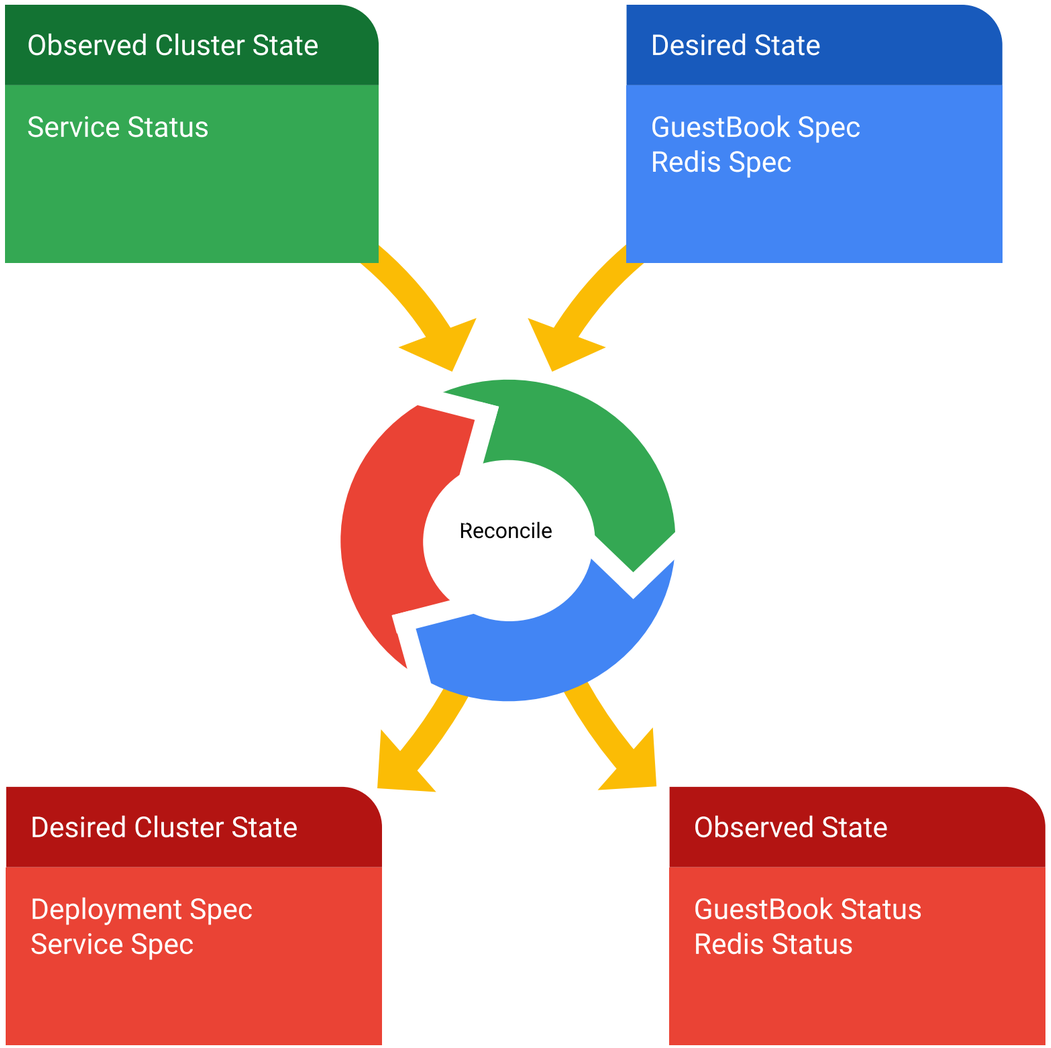
Read, reconcile, repeat

Read our root object

Fetch other objects we care about

Ensure those objects are in the right state

Write our root object's status



Soooo…. how do I do that?

We'll get a **Request** to reconcile, and fetch the corresponding objects with a **Client**.

We'll set up our desired state[5], marking our child objects as **owned** by our root object.

We'll ask the server to **apply** that state correctly with everyone else's changes.

We'll return a **Result** saying we're all done processing for now, or an error saying to try again in a bit[6].

our controller, in essence

```
var book webappv1.GuestBook
if err := r.Get(ctx, req.NamespacedName, &book); err != nil {
    return ctrl.Result{}, client.IgnoreNotFound(err)
}

var redis webappv1.Redis
redisName := client.ObjectKey{Name: book.Spec.RedisName, Namespace: req.Namespace}
if err := r.Get(ctx, redisName, &redis); err != nil {
    return ctrl.Result{}, client.IgnoreNotFound(err)
}

deployment, err := r.desiredDeployment(book, redis)
if err != nil {
    return ctrl.Result{}, err
}
svc, err := r.desiredService(book)
if err != nil {
    return ctrl.Result{}, err
}

applyOpts := []client.PatchOption{client.ForceOwnership, client.FieldOwner("guestbook")}
err := r.Patch(ctx, &deployment, client.Apply, applyOpts...)
if err != nil {
    return ctrl.Result{}, err
}
err = r.Patch(ctx, &svc, client.Apply, applyOpts...)
if err != nil {
    return ctrl.Result{}, err
}

book.Status.URL = urlForService(svc, *book.Spec.Frontend.ServingPort)

err = r.Status().Update(ctx, &book)
if err != nil {
    return ctrl.Result{}, err
}

return ctrl.Result{}, nil
```

5. ***Only*** what we care about, though

6. We want to ignore some errors, like not found, because trying again won't help until the object we're trying to get actually exists.

but what was in those helper functions?

Basically just the Go form of Kubernetes objects, but **only what we care about**!

When writing controllers, we want to be **declarative** and **tolerate changes by other components**.

Server-Side Apply lets us declare the structure that we care about, and let the server take care of merging those changes into the object [7].

We also **set the owner reference**, so that we keep track of which Guestbook **owns** these objects.

`controllers/helpers.go` (`desiredService` and `urlForService`)

```go
func (r *GuestBookReconciler) desiredService(book webappv1.GuestBook) (corev1.Service, error) {
    svc := corev1.Service{
        TypeMeta: metav1.TypeMeta{
            APIVersion: corev1.SchemeGroupVersion.String(), Kind: "Service"
        },
        ObjectMeta: metav1.ObjectMeta{Name: book.Name, Namespace: book.Namespace},
        Spec: corev1.ServiceSpec{
            Ports: []corev1.ServicePort{
                {Name: "http", Port: 8080,
                 Protocol: "TCP", TargetPort: intstr.FromString("http")},
            },
            Selector: map[string]string{"guestbook": book.Name},
            Type:     corev1.ServiceTypeLoadBalancer,
        },
    }

    // always set the controller reference so that we know which object owns this.
    if err := ctrl.SetControllerReference(&book, &svc, r.Scheme); err != nil {
        return svc, err
    }
    return svc, nil
}

func urlForService(svc corev1.Service, port int32) string {
    // unset this if it's not present -- we always want the
    // state to reflect what we observe.
    if len(svc.Status.LoadBalancer.Ingress) == 0 {
        return ""
    }

    host := svc.Status.LoadBalancer.Ingress[0].Hostname
    if host == "" {
        host = svc.Status.LoadBalancer.Ingress[0].IP
    }
    return fmt.Sprintf("http://%s", net.JoinHostPort(host, fmt.Sprintf("%v", port)))
}
```

7. In the words of Picard: "make it so!"

Now, we just need some wiring!

Controller Wiring 101

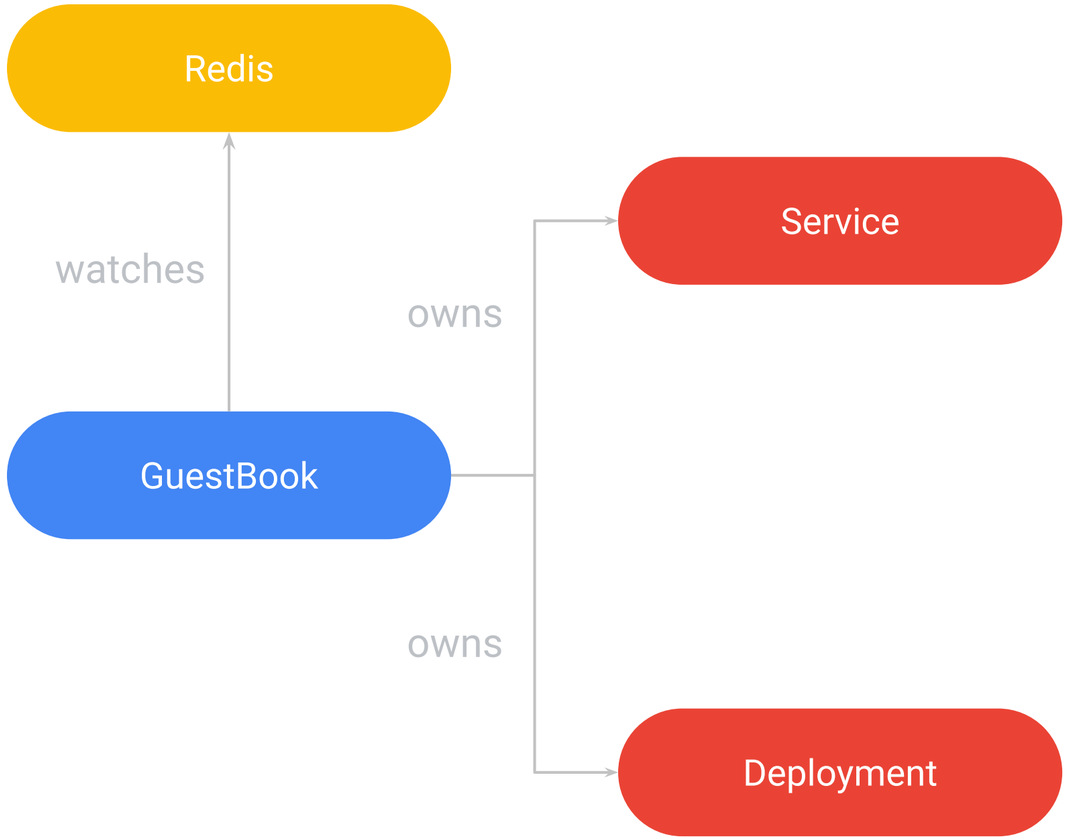
A **controller** is responsible for executing our logic. We call that logic a **reconciler**.

Each controller functions on (is **for**) a *single* Kind.

This kind may **own** other Kinds that it creates, or **watch** Kinds that are otherwise related.

For instance, our GuestBook controller:

> is **for** *GuestBooks*
>
> will **own** *Services* and *Deployments* to run and expose the frontend PHP app.
>
> **watches** *Redis*-es [8] to see the leader and follower service names.



8. Redi? What's the plural of Redis anyway? 10 points to anyone who can come for with a reasonable yet completely false linguistic explanation for the plural of their choice.

Wire it up…

We'll wrap the code from above in the `Reconcile` function (which is part of the `Reconcile` interface).

We'll provide a helper to set up the controller to run as part of a **manager**, which is responsible for coordinating all the controllers.

We'll need to add an **index** on the `RedisName` field so that we can tell our controller how a given Redis **relates back to** one or more GuestBooks in `booksUsingRedis` [9].

`controllers/guestbook_controller.go`

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

    // all that jazz above goes here!

    log.Info("reconciled guestbook")
    return ctrl.Result{}
}

func (r *GuestBookReconciler) SetupWithManager(mgr ctrl.Manager) error {
    mgr.GetFieldIndexer().IndexField(
        &webappv1.GuestBook{}, ".spec.redisName",
        func(obj runtime.Object) []string {
            redisName := obj.(*webappv1.GuestBook).Spec.RedisName
            if redisName == "" {
                return nil
            }
            return []string{redisName}
        })

    return ctrl.NewControllerManagedBy(mgr).
        For(&webappv1.GuestBook{}).
        Owns(&corev1.Service{}).
        Owns(&appsv1.Deployment{}).
        Watches(
            &source.Kind{Type: &webappv1.Redis{}},
            &handler.EnqueueRequestsFromMapFunc{
                ToRequests: handler.ToRequestsFunc(r.booksUsingRedis),
            }).
        Complete(r)
}
```

9. To do this, we'll use `r.List()` with the `client.MatchingField` option, as we'll see a bit later.

…add the watch helper…

We just use our `List()` method to list all *GuestBook* items that **match the index on `redisName`**.

Then, we **map** those GuestBook instances to reconcile **Requests**.

`controllers/helpers.go` (`booksUsingRedis`)

```
func (r *GuestBookReconciler) booksUsingRedis(
    obj handler.MapObject) []ctrl.Request {

    ctx := context.Background()

    listOptions := client.ListOption{
        // matching our index
        client.MatchingField(".spec.redisName",
            obj.Meta.GetName()),
        // in the right namespace
        client.InNamespace(obj.Meta.GetNamespace()),
    }

    var list webappv1.GuestBookList
    if err := r.List(ctx, &list, listOptions...); err != nil {
        return nil
    }

    res := make([]ctrl.Request, len(list.Items))
    for i, book := range list.Items {
        res[i].Name = book.Name
        res[i].Namespace = book.Namespace
    }
    return res
}
```

…and try it out!

Let’s give it a try:

```
(terminal 1) $ make run
(terminal 2) $ k get guestbooks
NAME               URL                     DESIRED
guestbook-sample   http://somehost:8080    1
```

Run as cluster admin – who needs permissions anyway?

Well, actually we do 🤦‍♀️

We'll use the `// +kubebuilder:rbac` marker to add **additional RBAC permissions** to our controller.

We've already got all thre permissions for our types, and we'll need to add **GET, LIST, and PATCH** [10] for **Deployments** and **Services**.

`controllers/guestbook_controller.go` (above the reconciler)

```
// notice that these aren't in the reconciler godoc --
// they're global to the project, not specific to a particular reconciler,
// so they can't go in the godoc.

// +kubebuilder:rbac:groups=webapp.metamagical.dev,resources=guestbooks,verbs=get;list;watch;crea
// +kubebuilder:rbac:groups=webapp.metamagical.dev,resources=guestbooks/status,verbs=get;update;p

// +kubebuilder:rbac:groups=apps,resources=deployments,verbs=list;get;patch
// +kubebuilder:rbac:groups=core,resources=services,verbs=list;get;patch

func (r *GuestBookReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
    // a bunch o' code
}
```

10. as you might've noticed, PATCH is the verb we use for Server-Side Apply

My work laptop is a viable deployment platform, right?

We'll need to **build** an image containing our controller manager and **push** it somewhere we can use it.

Then, we'll **deploy** it to our cluster.

Finally, we'll wait a bit, then open the browser with the URL from the **status** of our object, and we should get a working guestbook!

```
$ export IMG=your-docker-repo/webapp-manager
$ make docker-build docker-push deploy
$ brew-tea "earl grey" --sweetener=sugar && sleep 1m
♨

  ♨
   ♨
   ☕
  ┏━┓

$ $BROWSER $(k get guestbooks guestbook-sample \
    -o jsonpath='.status.url')
```

That’s all folks!

KubeBuilder: book.kubebuilder.io

controller-runtime godocs: godoc.org/sigs.k8s.io/controller-runtime

This (and other) workshops: pres.metamagical.dev/kubecon-us-2019/code

These slides: pres.metamagical.dev/kubecon-us-2019