Unless otherwise specified...


```
/*
Copyright 2017 The Kubernetes Authors.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
*/
```

Why Are We Copying and Pasting So Much?

Philip Wittrock (@pwittrock) & Solly Ross (@directxman12), Google

Kubernetes APIs are implemented by Controllers

- Respond to changes to Resources made by users
- Respond to changes to Cluster State

Controllers Are Simple

- ✓ watch resources,
- ✓ do some business logic,
- ✓ reconcile differences,
- ✓ … and a bit of plumbing

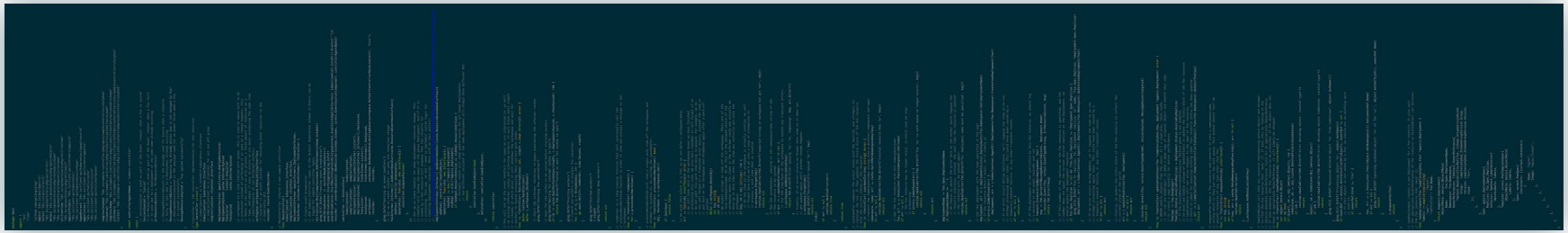
How complicated could it be?

Sample Controller at a Glance

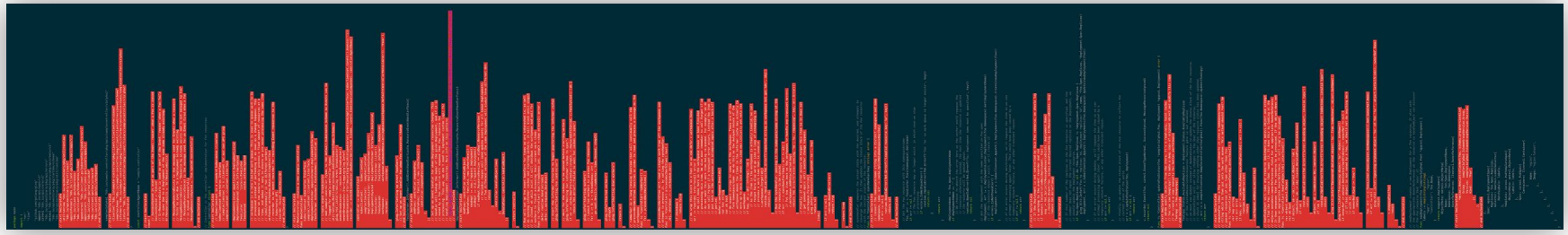

```
~/go/src/k8s.io/sample-controller $ wc -l < controller.go
429
~/go/src/k8s.io/sample-controller $ tree -I vendor
<snip>
28 directories, 52 files
```

Sample C

Sample Controller



This is not a flame chart

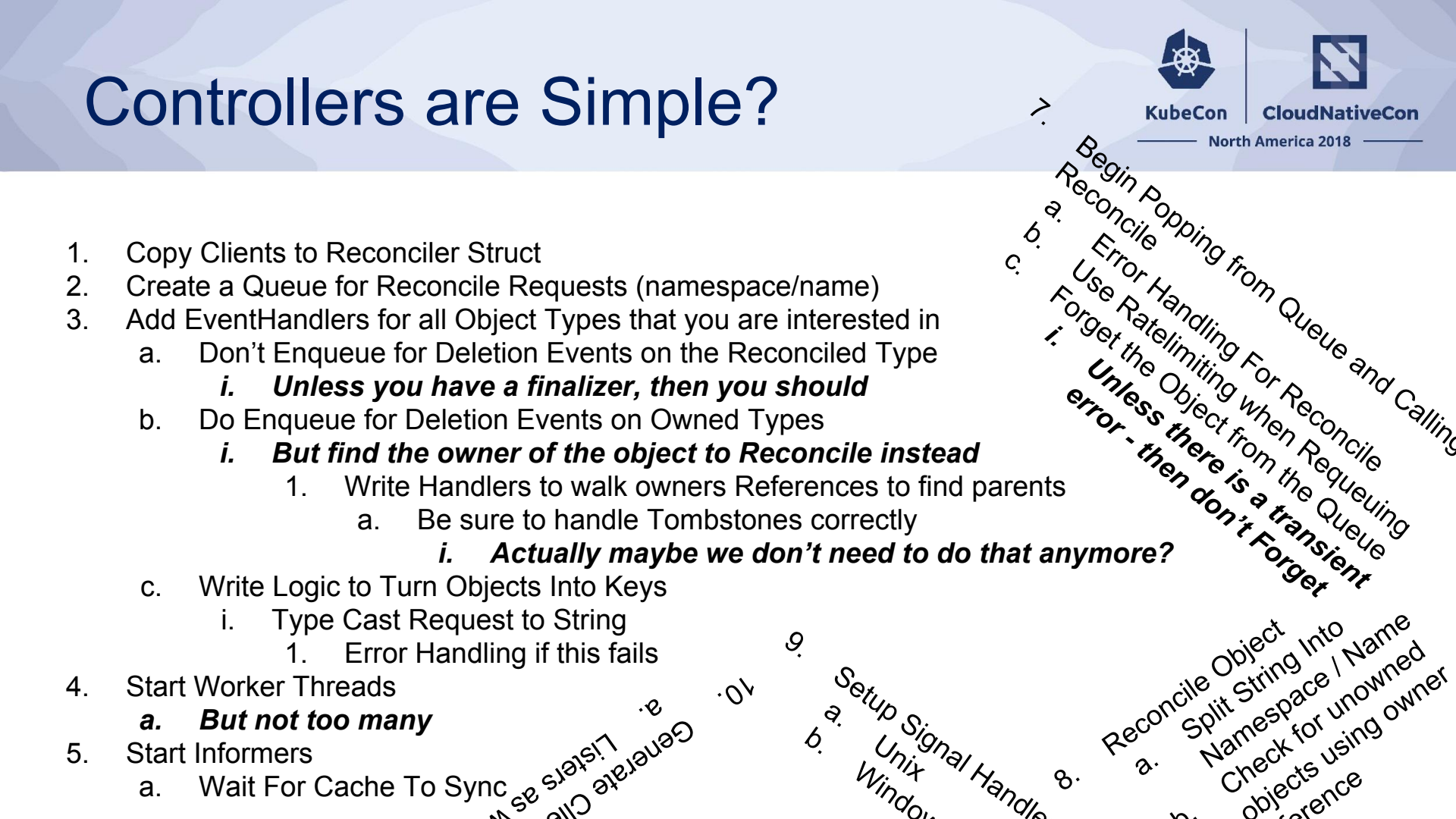


Note: Boilerplate code to copy-paste highlighted in Red

At least the comments are thorough...


```
// We call Done here so the workqueue knows we have finished
// processing this item. We also must remember to call Forget if we
// do not want this work item being re-queued. For example, we do
// not call Forget if a transient error occurs, instead the item is
// put back on the workqueue and attempted again after a back-off
// period.
...
// We expect strings to come off the workqueue. These are of the
// form namespace/name. We do this as the delayed nature of the
// workqueue means the items in the informer cache may actually be
// more up to date that when the item was initially put onto the
// workqueue.
...
// As the item in the workqueue is actually invalid, we call
// Forget here else we'd go into a loop of attempting to
// process a work item that is invalid.
```

Controllers are Simple?



1. Copy Clients to Reconciler Struct
2. Create a Queue for Reconcile Requests (namespace/name)
3. Add EventHandlers for all Object Types that you are interested in
   a. Don't Enqueue for Deletion Events on the Reconciled Type
      *i. **Unless you have a finalizer, then you should***
   b. Do Enqueue for Deletion Events on Owned Types
      *i. **But find the owner of the object to Reconcile instead***
         1. Write Handlers to walk owners References to find parents
            a. Be sure to handle Tombstones correctly
               *i. **Actually maybe we don't need to do that anymore?***
   c. Write Logic to Turn Objects Into Keys
      i. Type Cast Request to String
         1. Error Handling if this fails
4. Start Worker Threads
   a. But not too many
5. Start Informers
   a. Wait For Cache To Sync

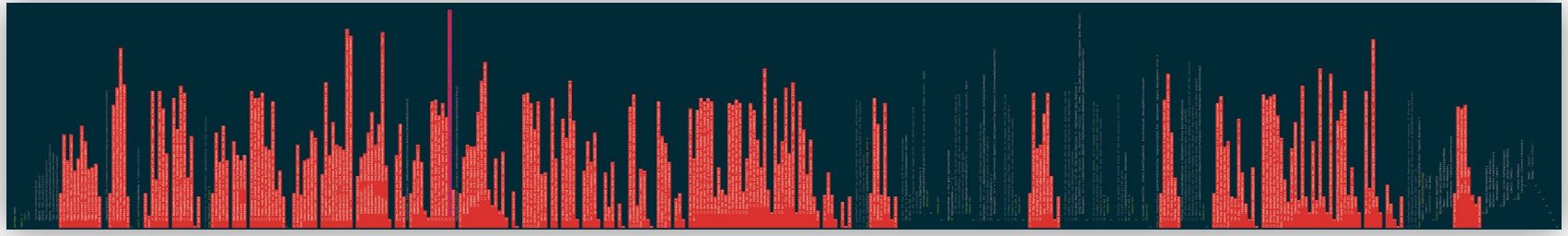
7. Begin Popping from Queue and Calling Reconcile
   a. Error Handling For Reconcile
   b. Use Ratelimiting when Requeuing
   c. Forget the Object from the Queue
      *i. **Unless there is a transient error - then don't Forget***

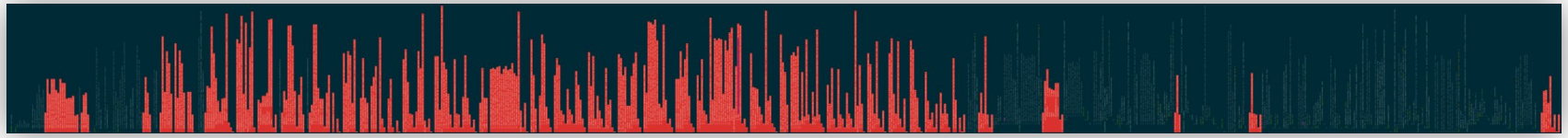
8. Reconcile Object
   a. Split String Into Namespace / Name
   Check for unowned objects using owner reference

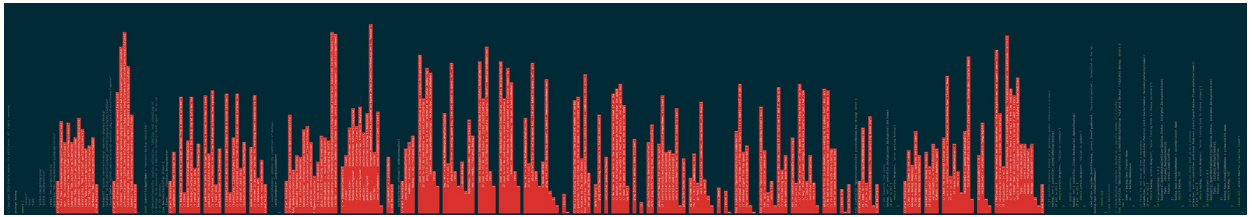
9. Setup Signal Handle
   a. Unix
   b. Windo

10. a. Generate Cli… Listers as …

Which is the real Sample Controller?







Sample Controler + Core Kubernetes Controller + Third Party Operator

Controllers *Should Be* Simple

65% of the sample controller code is…

- Unmodified Copy-Pasted Functions
- Initialization and Instantiation of Deps
- Plumbing and Wiring

Native Controller Support in Go 1.12!

controller-runtime *(proper noun)*

a Go standard library for controllers

```
package main

import (
      "controllers"
)
```

Just Kidding

Seriously though, use this instead

controller-runtime *(proper noun)*

a Go standard library for controllers

```
package main

import (
	"sigs.k8s.io/controller-runtime/pkg/builder"
)
```

The magic of abstractions...

Patterns

- Watch Objects to Reconcile
- Watch Generated Objects
- Watch Objects that Map to Reconciled Objects

Utilities

- Pre-Split Namespace/Name
- Universal Client (no more wiring generated clients)
- Helper for setting Owners References
- Helper for Upserting Objects

Create Controller


```
type Controller struct {
    workqueue workqueue.RateLimitingInterface
}

func Start() {
    stopCh := signals.SetupSignalHandler()

    cfg, err := clientcmd.BuildConfigFromFlags(masterURL, kubeconfig)
    if err != nil {
        klog.Fatalf("Error building kubeconfig: %s", err.Error())
    }

    controller := &Controller{
        workqueue: workqueue.NewNamedRateLimitingQueue(workqueue.DefaultControllerRateLimiter(), "Foos"),
    }

    controller.Run(2, stopCh)
}

func (c *Controller) Run(threadiness int, stopCh <-chan struct{}) error {
    defer runtime.HandleCrash()
    defer c.workqueue.ShutDown()

    klog.Info("Starting Foo controller")

    klog.Info("Waiting for informer caches to sync")
    if ok := cache.WaitForCacheSync(stopCh, c.deploymentsSynced, c.foosSynced); !ok {
        return fmt.Errorf("failed to wait for caches to sync")
    }

    klog.Info("Starting workers")
    for i := 0; i < threadiness; i++ {
        go wait.Until(c.runWorker, time.Second, stopCh)
    }

    klog.Info("Started workers")
    <-stopCh
    klog.Info("Shutting down workers")

    return nil
}

func (c *Controller) runWorker() {
    for c.processNextWorkItem() {
    }
}

func (c *Controller) processNextWorkItem() bool {
    obj, shutdown := c.workqueue.Get()

    if shutdown {
        return false
    }

    err := func(obj interface{}) error {
        defer c.workqueue.Done(obj)
        var key string
        var ok bool
        if key, ok = obj.(string); !ok {
            c.workqueue.Forget(obj)
            runtime.HandleError(fmt.Errorf("expected string in workqueue but got %#v", obj))
            return nil
        }
        if err := c.syncHandler(key); err != nil {
            c.workqueue.AddRateLimited(key)
            return fmt.Errorf("error syncing '%s': %s, requeuing", key, err.Error())
        }
        c.workqueue.Forget(obj)
        klog.Infof("Successfully synced '%s'", key)
        return nil
    }(obj)

    if err != nil {
        runtime.HandleError(err)
        return true
    }

    return true
}
```

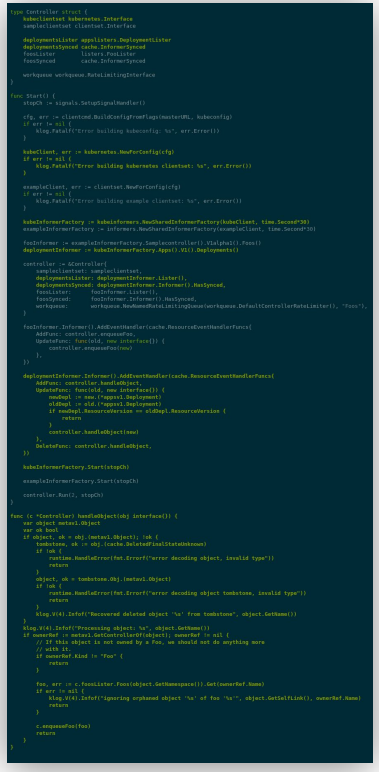
```
type Controller struct {}

func Start() {
    ctrl, _ := builder.SimpleController().Build(&Controller{})
    ctrl.Start(signals.SetupSignalHandler())
}
```

Watch Object


```
type Controller struct {
    sampleclientset clientset.Interface
    foosLister        listers.FooLister
    foosSynced        cache.InformerSynced
    workqueue workqueue.RateLimitingInterface
}

func Start() {
    stopCh := signals.SetupSignalHandler()

    cfg, err := clientcmd.BuildConfigFromFlags(masterURL, kubeconfig)
    if err != nil {
        klog.Fatalf("Error building kubeconfig: %s", err.Error())
    }

    exampleClient, err := clientset.NewForConfig(cfg)
    if err != nil {
        klog.Fatalf("Error building example clientset: %s", err.Error())
    }

    exampleInformerFactory := informers.NewSharedInformerFactory(exampleClient, time.Second*30)
    fooInformer := exampleInformerFactory.Samplecontroller().V1alpha1().Foos()

    controller := &Controller{
        sampleclientset: sampleclientset,
        foosLister:      fooInformer.Lister(),
        foosSynced:      fooInformer.Informer().HasSynced,

        workqueue:       workqueue.NewNamedRateLimitingQueue(workqueue.DefaultControllerRateLimiter(), "Foos"),
    }

    fooInformer.Informer().AddEventHandler(cache.ResourceEventHandlerFuncs{
        AddFunc: controller.enqueueFoo,
        UpdateFunc: func(old, new interface{}) {
            controller.enqueueFoo(new)
        },
    })

    exampleInformerFactory.Start(stopCh)

    controller.Run(2, stopCh)
}

func (c *Controller) enqueueFoo(obj interface{}) {
    var key string
    var err error
    if key, err = cache.MetaNamespaceKeyFunc(obj); err != nil {
        runtime.HandleError(err)
        return
    }
    c.workqueue.AddRateLimited(key)
}
```

```
type Controller struct {
    Client client.Client
}

func (ct *Controller) InjectClient(c client.Client) error {
    ct.Client = c
    return nil
}

func Start() {
    ctrl, _ := builder.SimpleController().
        ForType(&v1alpha1.Foo{}).
        Build(&Controller{})
    ctrl.Start(signals.SetupSignalHandler())
}
```

Watch Owned Objects



```go
type Controller struct {
    Client client.Client
}

func (ct *Controller) InjectClient(c client.Client) error {
    ct.Client = c
    return nil
}

func Start() {
    ctrl, _ := builder.SimpleController().
        ForType(&v1alpha1.Foo{}).
        Owns(&appsv1.Deployment{}).
        Build(&Controller{})
    ctrl.Start(signals.SetupSignalHandler())
}
```

Sophisticated Watch APIs


```
// .ForType(&corev1.Pod{})
c.Watch(&source.Kind{Type: &corev1.Pod{}},
  &handler.EnqueueRequestForObject{})
```

The *.ForType* **builder function is really just a call to** *.Watch* **with** *EnqueueRequestForObject*

Sophisticated Watch APIs


```
// .Owns(&appsv1.ReplicaSet{})
c.Watch(&source.Kind{Type: &corev1.Pod{}},
  &handler.EnqueueRequestForOwner{
    IsController: true, OwnerType: &appsv1.ReplicaSet{}})
```

The *.Owns* builder function is really just a call to *.Watch* with *EnqueueRequestForOwner*

Sophisticated Watch APIs


```
// Not Present In Builder - Map One Object to other Objects
c.Watch(&source.Kind{Type: &appsv1.Deployment{}},
  &handler.EnqueueRequestsFromMapFunc{ToRequests: mapFn})
```

Users can implement their own patterns by providing a function to map events to objects

Extending Watch APIs


```
Watch(src source.Source, eventhandler handler.EventHandler, predicates ...predicate.Predicate) error

type EventHandler interface {
    // Create is called in response to an create event - e.g. Pod Creation.
    Create(event.CreateEvent, workqueue.RateLimitingInterface)

    // Update is called in response to an update event -  e.g. Pod Updated.
    Update(event.UpdateEvent, workqueue.RateLimitingInterface)

    // Delete is called in response to a delete event - e.g. Pod Deleted.
    Delete(event.DeleteEvent, workqueue.RateLimitingInterface)

    // Generic is called in response to an event of an unknown type or a synthetic event triggered as a cron or
    // external trigger request - e.g. reconcile Autoscaling, or a Webhook.
    Generic(event.GenericEvent, workqueue.RateLimitingInterface)
}
type Source interface {
    // Start is internal and should be called only by the Controller to register an EventHandler with the Informer
    // to enqueue reconcile.Requests.
    Start(handler.EventHandler, workqueue.RateLimitingInterface, ...predicate.Predicate) error
}
```

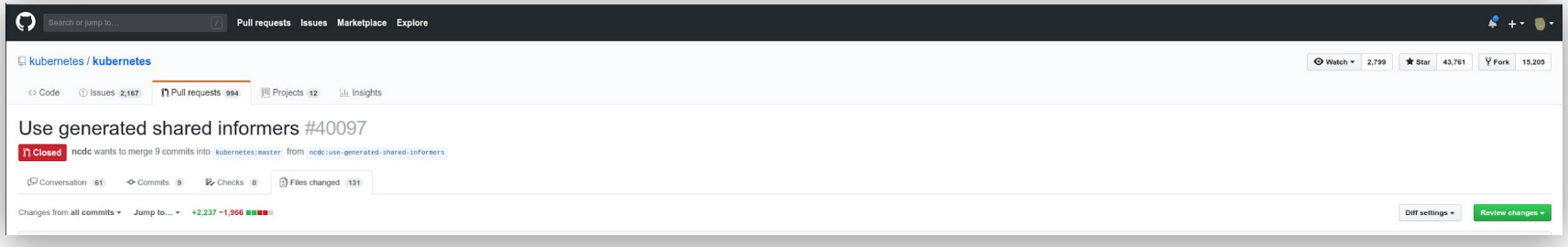
Shared Dependencies

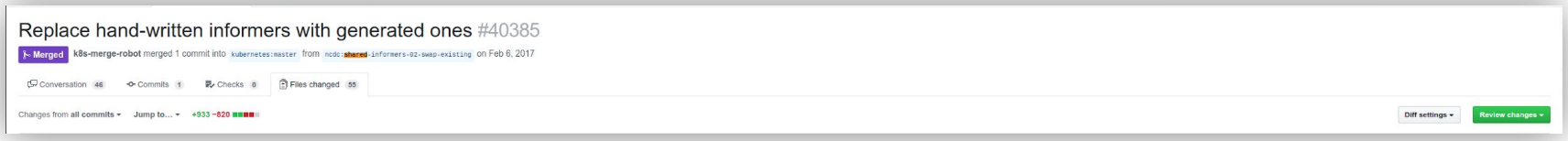

```go
// Implement these to have dependencies Injected
type Client interface {
    InjectClient(client.Client) error
}
type Config interface {
    InjectConfig(*rest.Config) error
}
type Scheme interface {
    InjectScheme(scheme *runtime.Scheme) error
}
type Stoppable interface {
    InjectStopChannel(<-chan struct{}) error
}
// Implement this to get a function to Inject dependencies into other objects
type Injector interface {
    InjectFunc(f Func) error
}
```

Reconcile Utilities


```
// Sample Controller
OwnerReferences: []metav1.OwnerReference{
    *metav1.NewControllerRef(foo, schema.GroupVersionKind{
    Group:   samplev1alpha1.SchemeGroupVersion.Group,
    Version: samplev1alpha1.SchemeGroupVersion.Version,
    Kind:    "Foo"})}

// Controller-Runtime
controllerutil.SetControllerReference(foo, deployment, c.scheme)
```

For Developers...





For Ops...

Standard…

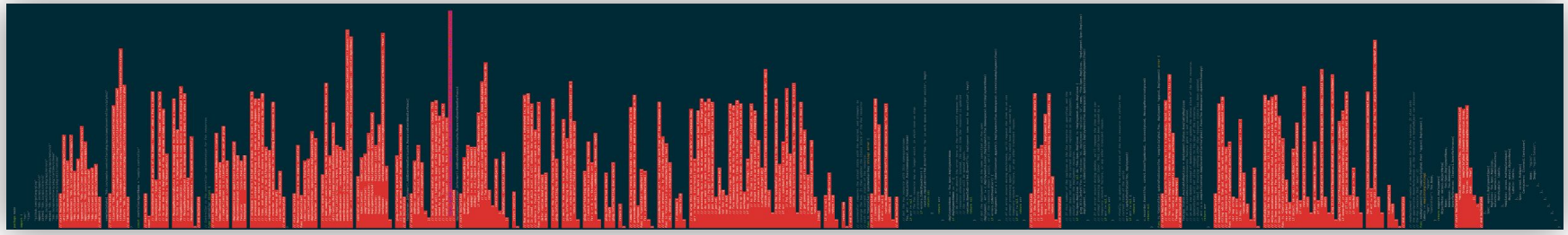
- Behavior
- Logging
- Metrics
- Error handling and degradation
- Maintenance

For Platform Developers...

Build high-level abstractions with common building blocks:

- Kubebuilder
- Operator SDK
- Maestro Declarative Operator (via kubebuilder)
- AddOnOperator (via kubebuilder)

Remember this...



It fits!


```go
func main() {
    mgr, err := builder.SimpleController().
        ForType(&samplev1alpha1.Foo{}).
        Owns(&appsv1.Deployment{}).
        Build(&Controller{})

    if err != nil {
        logf.Log.Error(err, "unable to construct foo controller")
        os.Exit(1)
    }

    if err := mgr.Start(signals.SetupSignalHandler()); err != nil {
        logf.Log.Error(err, "unable to start controllers")
        os.Exit(1)
    }
}
```

```go
type Controller struct {
    client.Client
}

func (c *Controller) InjectClient(cl client.Client) error {
    c.Client = cl
    return nil
}

func (c *Controller) Reconcile(req reconcile.Request) (reconcile.Result, error) {
    log := logf.Log.WithName("foo-controller").WithValues("foo", req.NamespacedName)
    var foo samplev1alpha1.Foo
    if err := c.Get(context.Background(), req.NamespacedName, &foo); err != nil {
        if errors.IsNotFound(err) {
            log.Error(err, "foo no longer exists")
            return reconcile.Result{}, nil
        }
        return reconcile.Result{}, err
    }

    deploymentName := foo.Spec.DeploymentName
    if deploymentName == "" {
        log.Error(nil, "deployment name must be specified")
        return reconcile.Result{}, nil
    }

    var deployment appsv1.Deployment
    if _, err := controllerutil.CreateOrUpdate(context.Background(), c, &deployment, func(existing runtime.Object) {
        return updateDeployment(deploymentName, &foo, existing)
    }); err != nil {
        return reconcile.Result{}, err
    }

    foo.Status.AvailableReplicas = deployment.Status.AvailableReplicas
    if err := c.Update(context.Background(), &foo); err != nil {
        return reconcile.Result{}, err
    }

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

```go
func updateDeployment(deploymentName string, foo *samplev1alpha1.Foo, existing runtime.Object) error {
    depl := existing.(*appsv1.Deployment)
    depl.Name = deploymentName
    depl.Namespace = foo.Namespace
    if err := controllerutil.SetControllerReference(foo, depl, samplescheme.Scheme); err != nil {
        return err
    }
    labels := map[string]string{
        "app":        "nginx",
        "controller": foo.Name,
    }
    depl.Spec = appsv1.DeploymentSpec{
        Replicas: foo.Spec.Replicas,
        Selector: &metav1.LabelSelector{
            MatchLabels: labels,
        },
        Template: corev1.PodTemplateSpec{
            ObjectMeta: metav1.ObjectMeta{
                Labels: labels,
            },
            Spec: corev1.PodSpec{
                Containers: []corev1.Container{
                    {
                        Name:  "nginx",
                        Image: "nginx:latest",
                    },
                },
            },
        },
    }
    return nil
}
```

In conclusion...

Controllers Are Simple

```
package main

import (
      "sigs.k8s.io/controller-runtime/pkg/builder"
)
```

Applause...