Testing CRDs

KubeCon Shanghai 2018
Hi!

I’m Christie Wilson

knative/serving conformance
knative/build-pipeline
Testing CRD based platforms
You will get the most out of this talk if:

- You have implemented or worked on a CRD controller
- You are programming in Go
- You are using client-go (kubernetes/sample-controller) or kubebuilder
CRDs

Kubernetes Custom Resource Definitions

- Extend Kubernetes functionality by adding your own types
- Usually for building platforms
CRDs + Controllers = Declarative API
Controller

Continually works to achieve your desired state

- When a CRD changes state:
  - Create
  - Update
  - Delete
Platform

Software to build software on

“...an operating environment which teams can build upon to deliver product features to customers more quickly, supported by reusable capabilities.”

- Evan Bottcher
Platforms are the foundation
Foundations better be done well
Platforms should be our best tested software
Test pyramid

- **UNIT** (fast)
- **SERVICE** (medium)
- **UI** (slow)

Cost:
- **$**
- **$$**
- **$$$$**

Source: github.com/bobcatfish/testing-crd
CRD testing pyramid

- **Unit**: Fast, low cost
- **Integration**: Faster, medium cost
- **End-to-End**: Slow, high cost

github.com/bobcatfish/testing-crds
End to end tests for CRDs
End to end tests

- Use a real Kubernetes cluster
Very few end to end tests!

- Don’t try to cover all the gaps
- Use a real Kubernetes cluster
End to End Test Prerequisites

- We’ll be assuming:
  - You have a kubernetes cluster (or a pool of clusters!)
  - Your CRDs and controllers have been deployed
    - e.g. using a tool like `ko`, or kubebuilder’s `make deploy`
How to write end to end tests

1. Build these tests separate from your other go code
2. Don’t use YAML for test cases
3. Use a kubeconfig to configure cluster access
   a. With client-go
   b. With kubebuilder
4. Poll resources using apimachinery wait
5. Output (unbuffered) logs regularly
6. One namespace per test
7. Cleanup on interrupt
   a. Don’t create a framework
1. Running/building only end to end tests

Use golang build tags [https://golang.org/pkg/go/build/](https://golang.org/pkg/go/build/)

```go
// +build e2e
```

To run tests:

```
# Unit tests
go test ./...

# Integration tests (against your current kube cluster)
go test -v -count=1 -tags=e2e ./test
```
2. YAML vs Go for test input

- Go is more flexible
- YAML is better for examples
What do we do with these objects?

Two ways of working with them:

- client-go
- kubebuilder
client-go

kubernetes/sample-controller

k8s.io/code-generator/generate-groups.sh

- Statically generated libs specific to your CRDs
- Must be updated when your spec is updated
- Lots of boilerplate
kubebuilder

SDK for kubernetes APIs

- Dynamic clients
- Supporting libs:
  - controller-runtime
  - Controller-tools
- Instead of copy-pasting an example, it sets up an environment
  - Way less boilerplate
3. Initialize clients with kubeconfig

- Use the **kubeconfig**
  - Endpoint config
  - Auth config
- Either the current-context or use flags to use a different context

```go
import "k8s.io/client-go/tools/clientcmd"
...
cfg, err := clientcmd.NewNonInteractiveDeferredLoadingClientConfig(
    &clientcmd.ConfigOverrides{}`).ClientConfig()
```
```go
import "github.com/bobcatfish/testing-crds/client-go/pkg/client/clientset/versioned"
...

nonce, err := versioned.NewForConfig(cfg)
...

c := nonce.CatVlalphal().Cats(namespace)
```

```go
import {
    "sigs.k8s.io/controller-runtime/pkg/manager"
    "k8s.io/client-go/kubernetes/scheme"
    "github.com/bobcatfish/testing-crds/kubebuilder/pkg/apis"
"
...}

apis.AddToScheme(scheme.Scheme)
mgr, err := manager.New(cfg, manager.Options{})
...
c := mgr.GetClient()
```
4. Polling resources

- Use k8s.io/apimachinery/pkg/util/wait
- Many polling functions such as PollImmediate

```go
wait.PollImmediate(
    interval, timeout, func() (bool, error) {
        return true, nil
    })
```
4. Polling resources

```go
func WaitForPodState(c *kubernetes.Clientset, name string, namespace string, inState func(r *corev1.Pod) (bool, error)) error {
    return wait.PollImmediate(interval, timeout, func() (bool, error) {
        r, err := c.CoreV1().Pods(namespace).Get(name, metav1.GetOptions{})
        if err != nil {
            return true, err
        }
        return inState(r)
    })
}

func Test(t *testing.T) {
    // ...
    if err := WaitForPodState(c, podName, namespace, func(p *corev1.Pod) (bool, error) {
        if p.Status.Phase == corev1.PodRunning {
            return true, nil
        } else {
            return false, nil
        }
    }); err != nil {
        t.Errorf("error waiting for Pod %s to finish: %s", podName, err)
    }
    //...
}
```
5. Regular log output

- Run tests with verbose logging
- Output a log before any long running action

```go
logger.Infos("Creating Task %s", kanikoTaskName)
// ... Call create

logger.Infos("Creating TaskRun %s", kanikoTaskRunName)
// ... Call create

logger.Infos("Waiting for TaskRun %s to be created", kanikoTaskRunName)
// ... polling function
```
5. Don’t buffer your logs

- The testing log functions buffer output until the test has completed
- The log library does not
- Use named loggers for tests running in parallel
6. Create a namespace for every test

- At the beginning of the test, create a namespace
  - With a unique/random name
  - 7 random chars is good enough (like git commit short-ids)
- Makes your tests parallelizable
- Doesn’t work if your resource is cluster scoped

```go
namespace := appendRandomString("cattopia")
if _, err := c.CoreV1().Namespaces().Create(&corev1.Namespace{
    ObjectMeta: metav1.ObjectMeta{
        Name: namespace,
    },
}); err != nil {
    t.Fatal("Failed to create namespace %s for tests: %s", namespace, err)
}
```
7. Cleanup your tests properly

Delete the namespace:

- At the end of the test (defer)
- Catch the interrupt signal

```go
func Test(t *testing.T) {
    //...
    cleanupOnInterrupt(func() { tearDown(t, c, namespace) })
    defer tearDown(t, c, namespace)
    //...
}
```
Integration tests for CRDs
Integration tests for CRDs

- Combines (integrates) pieces of your software
- Some pieces may be faked out or missing
How to write integration tests for CRDs

1. Integration tests with kubebuilder
2. Unit tests with client-go test doubles
Integration tests with kubebuilder
Integration tests with kubebuilder

mytype_controller_suite_test.go
mytype_controller_test.go

- Skeleton generated for your controller
- Actual running processes
- Start controller + API server processes locally
Integration tests with kubebuilder

- Use Ginkgo
  - A BDD Go library
- No k8s cluster required!

```go
// Create the Feline object and expect the Reconcile and Deployment to be created
err = c.Create(context.TODO(), instance)
g.Expect(err).NotTo(gomega.HaveOccurred())
defer c.Delete(context.TODO(), instance)
g.Eventually(requests, timeout).Should(gomega.Receive(gomega.Equal(expectedRequest)))

deploy := &apps.v1.Deployment{}
g.Eventually(func() error { return c.Get(context.TODO(), depKey, deploy) }, timeout).Should(gomega.Succeed())
```
Integration tests with kubebuilder

What can you do?

- Create resources
- Assert that other resources were created, i.e. assert on the state

What can’t you do?

- Expect anything to be scheduled and run
- Hit any expected endpoints
Integration tests with client-go
Test Double

“...the generic term for any kind of pretend object used in place of a real object for testing purposes”- Martin Fowler

This term covers many different kinds of “pretend objects”, including:

- Mocks
- Fakes
- Stubs
CRD Test Doubles

client-go generated code provides:

- Fake kubernetes client ([k8s.io/client-go/kubernetes/fake](https://k8s.io/client-go/kubernetes/fake))
- Fake client for your CRDs, generated by client-go

You can reach into the indexer and add your own objects
CRD Test Doubles - Fake clients

Creating the client is simple:

```go
import "github.com/myorg/myproj/pkg/client/clientset/versioned/fake"
c := fake.NewSimpleClientset()
```
CRD Test Doubles - Seeding data

Seeding data is a bit more complicated:

c := fake.NewSimpleClientset()
sharedInfomer := informers.NewSharedInformerFactory(c, 0)
i := sharedInfomer.MyProj().V1alpha1().MyType()
// create your object `obj`
i.Informer().GetIndexer().Add(obj)
CRD Test Doubles - Seeding data

Starts to look worse for more complex cases
You can also assert on the function calls you expected:

```go
action := c.Actions()[0]
if action.GetVerb() != "list" {
    t.Errorf("expected list to be called")
}
```

(but your tests are less brittle if they assert on state instead of behaviour!)
Why are these integration tests?

- Inputs and outputs are at the same scope
- Combine pieces of the system together
  - But some are using fake data
- Not testing code in isolation
  - If something breaks, which code is broken?
kubebuilder or client-go integration tests?

● Local testing:
  ○ kubebuilder tests can run on an airplane!

● Data seeding:
  ○ client-go fakes let you seed data
  ○ With kubebuilder you’d have to actually “create” all the data

● Stuck with one or the other based on which approach you’ve taken
Unit tests for CRDs
Unit tests for CRDs

- Ideally most of your coverage is through unit tests
- Test your code in isolation
- Extremely fast
Integrated tests are a scam

Integrate multiple components
Integrated tests are a scam

- Slow
- Expensive
- Hard to debug
- Promote bad design
What is good design?

Testable code has:

- Loose coupling
- High cohesion
- Well defined interfaces
Integrated tests hide design problems
Integrated tests are okay with this reconcile function
Unit testing this would be a nightmare!
Using lots of unit tests encourages good design
Functional Core + Imperative Shell

- Implement the logic of your controller as well factored functions
- Glue the functions together in the imperative shell
Functional core + imperative shell

Functional =

- Doesn’t mutate state
- No I/O
- No side effects

Imperative =

- Mutates state
- I/O operations
  - Reading files, writing to sockets
- Non-determinism
  - Time, random
Sep. functionality from consumer

#golang top tip: use packages like a knife to separate code that consumes functionality from code that provides functionality. Use unit tests along the cut line to lock in the behaviour of your packages.

3:42 AM - 24 Oct 2018

83 Retweets 289 Likes
Isolate logic from reconcile loop
Reconcile loop

client-go

```go
func (c *Controller) syncHandler(key string)
```

kubebuilder

```go
func (r *ReconcileMyType) Reconcile(request reconcile.Request)
```
CRD: core + shell

- Shell =
  - The reconcile loop, with all of its:
    - Informers
    - Listers
    - Work queue
    - Event broadcasters
    - Workers

- Core =
  - The logic in the sync handlers
  - i.e. the business value
Isolating the functional core

Don’t pass around listers or informers, instead:

- Write functions that take functions as params (e.g. List)
  - Or interfaces that provide those functions
- Better yet write functions take objects directly
We can make well tested CRD based platforms!
Apply the CRD test pyramid
What can you do?

1. Measure your coverage
2. Refactor your reconcile loop
Measure your coverage

- 80%+ coverage is good
- How well covered is your platform?

go test -cover fmt
Refactor your reconcile loop

(Then measure coverage again!)

- How much logic is in your Reconcile / syncHandler?
- How many functions are members of your Controller/Reconcile struct?
Stable, reliable, maintainable platforms
Sources

- Integrated tests are a scam - JB Rainsberger
- Boundaries - Gary Bernhardt
- Functional core imperative shell - Gary Bernhardt
- Mocks aren’t stubs - Martin Fowler
- What I talk about when I talk about platforms - Evan Bottcher
- Test Pyramid - Martin Fowler
- Dave Cheney on twitter re. packages - Dave Cheney