# $ kubectl blame

In Search of a ‘kubectl blame’ Command

Nick Santos

The Problem

# Determining Cause and Effect

# Display Cause -> Effect

```
$ kubectl blame deployment my-busybox


Deployment my-busybox: Edited 15s ago - In progress…
  -> Creating ReplicaSet my-busybox-v10

ReplicaSet my-busybox-v10: Created 10s ago - In progress…
  -> Deleting Pod my-busybox-pod-v9
  -> Creating Pod my-busybox-pod-v10

Pod my-busybox-pod-v10: Created 5s ago - In progress…
  -> ERROR: Container [main] completed with exit code 1
```

# Display Effect -> Cause

```
$ kubectl blame pod my-busybox-v10


ERROR: Container [main] completed with exit code 1
  -> Container [main] from Deployment my-busybox revision 10
  -> Container [istio-init] from IstioController revision 2

Deployment my-busybox revision 10: Edited 15s ago
  -> Image busybox:v3 (updated from busybox:v2)
```

# Outline

1. The Problem (We're Here!)
2. Why This Is Hard
3. Examples: kubectl, helm --wait, kubespy trace
4. A Rant About The Future

# What this talk is NOT

How to build a

‘kubectl blame’ plugin!

(Though you will learn that.)

# What this talk COULD BE

Abstractions!

1. Problem Statement

# 2. Why This Is Hard

3. Examples: kubectl, helm --wait, kubespy trace
4. The Future

# Control Loops

- Declare the desired state
- Diff the desired state from the current state
- What you do next is a function of the diff!

# Control loops are old!

“Origins of Feedback Control” — Otto Mayr, Scientific American, 1970

- Water clocks (3rd Century BC)
- Thermostats (Early 1600s)
- Windmills (Mid 1700s)

Discusses the “cyclic structure of cause and effect”

# What about thermostats?

Two types of people in the world:

1. Ones who set the thermostat to 70F
2. Ones who set the thermostat to 90F so it “heats up faster”

# Declarative Systems are Everywhere

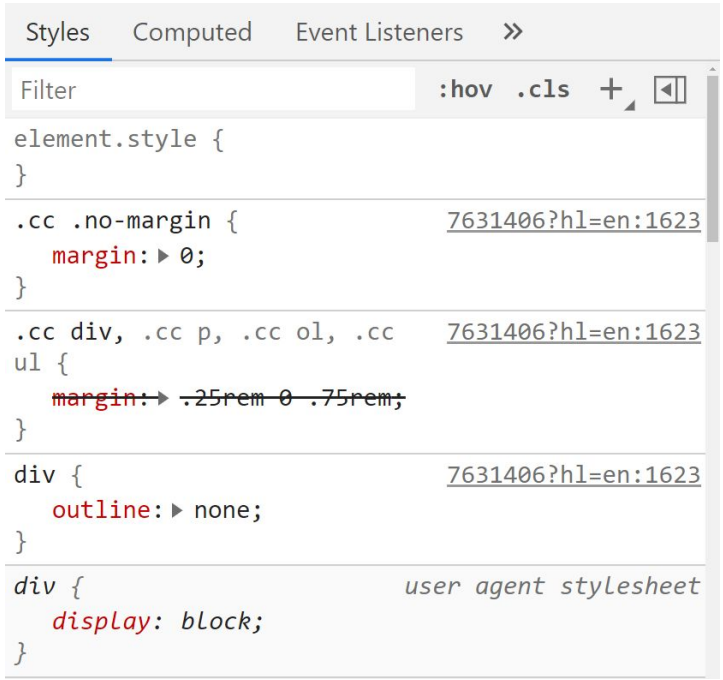
CSS - Cascading Style Sheets

- Used by every web UI dev
- Browsers, Websites, and Users all contribute style rules
- The browser interprets them, and how they overlay

# Declarative Systems are Hard to Debug

- Good CSS debuggers took years to get right
- A bad selector could kill site performance
- Arguments throughout the ‘10s about how to debug CSS problems

# How does CSS do this?




Let's look at
a CSS debugger today:

- An ancestry of rules
- See each rule's effect
- Rules override each other
- The 'Computed' tag shows the result

1. Problem Statement
2. Why This Is Hard

# 3. Examples

4. The Future

# The Sample App

1. Build and push an image
2. Apply a Deployment
3. Track the Deployment's progress
    a. The Deployment begets a ReplicaSet
    b. The ReplicaSet begets a Pod
    c. The Pod begets a Container

# The Sample App

Each one will track the Deployment differently.

Can they tell us:

- What is happening?
- When is it happening?
- Who is doing it?

**https://github.com/tilt-dev/kubectl-blame-examples**

# Naive Approach

Here's what Tilt did first:

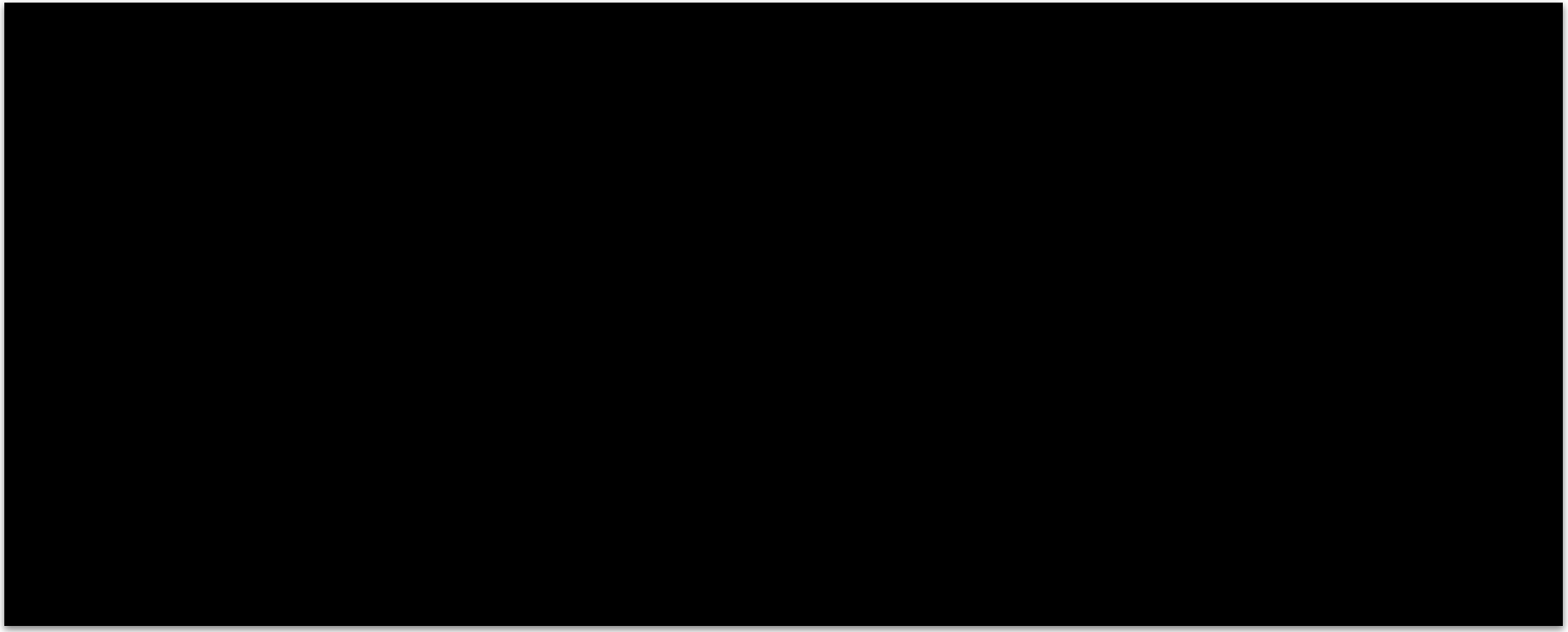
- Each Deployment gets a random label
- Attach the label everywhere
- Watch for pods with that label

# Naive Approach

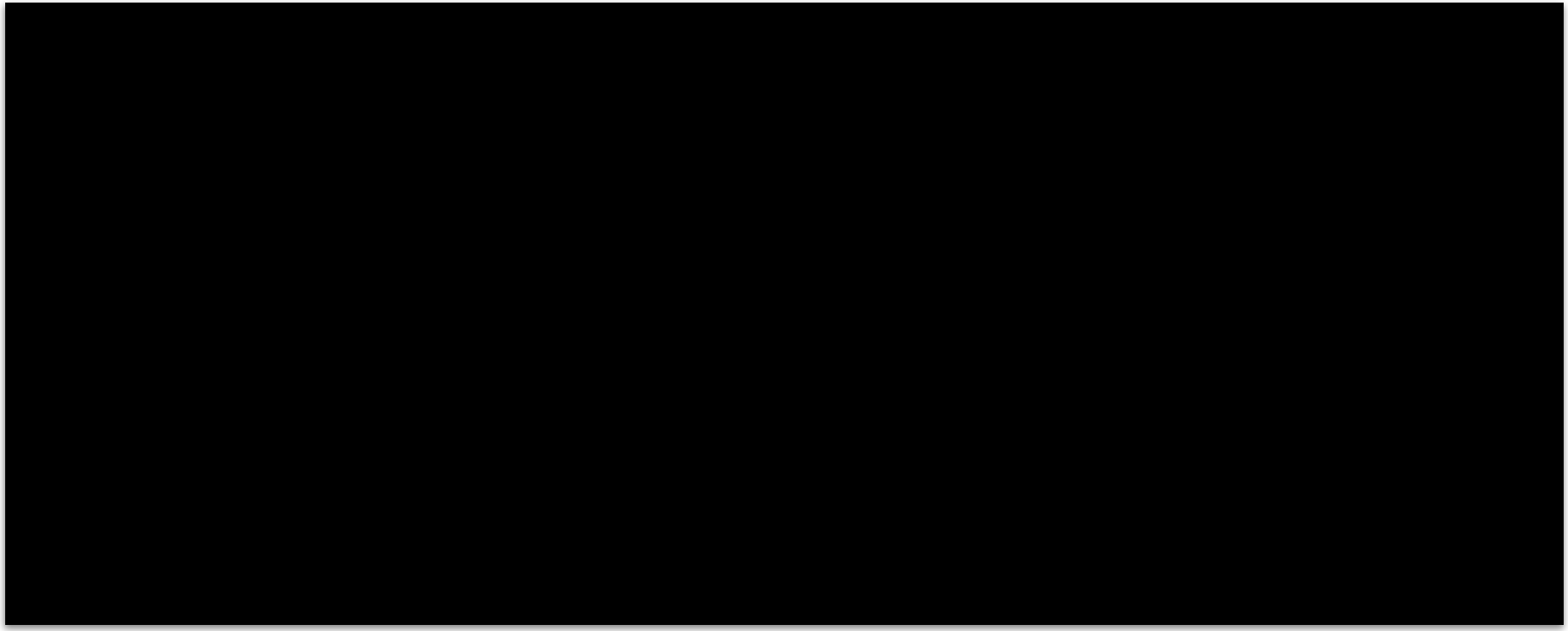
```
informers.NewSharedInformerFactoryWithOptions(
  client,
  15*time.Minute,
  informers.WithTweakListOptions(func(options *metav1.ListOptions) {
    options.LabelSelector = fmt.Sprintf("%s=%s", labelKey, labelValue)
  })).
  ForResource(v1.SchemeGroupVersion.WithResource("pods"))
```

0-naive/main.go

# Naive Approach



# Naive Approach (crash)



# Naive Approach

Pros:

- Efficient, easy to reason about
- See pods crash

Cons:

- Need to inject labels
- **Everybody hated it**

# Naive Approach

Restarted the app every time – even when nothing changed!

Like a thermostat that spends 5 minutes restarting the boiler every time you touch it.

# kubectl rollout

```
kubectl rollout status deployment --watch [name]
```

# kubectl rollout

# kubectl rollout (crash)

# kubectl rollout

How it works:

- Control loop that fetches latest Deployment
- Checks spec field against status field

# kubectl rollout

```
cond := GetDeploymentCondition(deployment.Status, appsv1.DeploymentProgressing)
if cond ≠ nil && cond.Reason == TimedOutReason {
     return "", false, fmt.Errorf("deployment exceeded its progress deadline", …)
}
if deployment.Spec.Replicas ≠ nil &&
   deployment.Status.UpdatedReplicas < *deployment.Spec.Replicas {
     return fmt.Sprintf("Waiting for deployment …\n", …)
}
if deployment.Status.Replicas > deployment.Status.UpdatedReplicas {
     return fmt.Sprintf("Waiting for deployment  ... \n", …)
}
 ...
```

# kubectl rollout

Pros:

- Knows one thing and knows it well

Cons:

- Knows nothing about pods
- Can’t tell you why things are failing

# helm upgrade --wait --debug

The Helm source code is like stackoverflow for client-go patterns!

Helm is great for learning.

Helm is also great for copying and using as a library. 😀

# helm upgrade --wait --debug

```
helmKubeClient := kube.New(nil)
res := &resource.Info{
     Namespace: "default",
     Name:      deployment.Name,
     Object:    &deployment,
}
err := helmKubeClient.Wait([]*resource.Info{res}, 15*time.Second)
```

# helm upgrade --wait --debug

# helm upgrade --wait (crash)

```
nick@dopey:~/src/kubectl-blame-examples/2-helm$ go run main.go --crash
Generated index.html = `Hello world! I'm deployment impacted-viscountesss!`
docker build -t localhost:5000/my-busybox:deploy-dd9b8df7a3d5aa49d3e35d948cf21acd -
docker push localhost:5000/my-busybox:deploy-dd9b8df7a3d5aa49d3e35d948cf21acd
[go] Adding command = ["sh", "-c", "exit 1"] because --crash=true
kubectl apply -o yaml -f -
[go] helm wait my-busybox
beginning wait for 1 resources with timeout of 15s
Deployment is not ready: default/my-busybox. 0 out of 1 expected pods are ready
Deployment is not ready: default/my-busybox. 0 out of 1 expected pods are ready
Deployment is not ready: default/my-busybox. 0 out of 1 expected pods are ready
Deployment is not ready: default/my-busybox. 0 out of 1 expected pods are ready
Deployment is not ready: default/my-busybox. 0 out of 1 expected pods are ready
Deployment is not ready: default/my-busybox. 0 out of 1 expected pods are ready
Deployment is not ready: default/my-busybox. 0 out of 1 expected pods are ready
```

# helm upgrade --wait --debug

How it works:

- Top-down traversal of resources
- Big switch statement of all the resource types
- Reaches down into ReplicaSet
- Compares ReplicaSet creation dates and pod templates to get right one

# helm upgrade --wait --debug

```
currentDeployment, err := w.c.AppsV1().Deployments(v.Namespace).Get(ctx.Name, ...)
if err != nil {
      return false, err
}
// If paused deployment will never be ready
if currentDeployment.Spec.Paused {
      continue
}
// Find RS associated with deployment
newReplicaSet, err := deploymentutil.GetNewReplicaSet(currentDeployment, w.c.AppsV1())
if err != nil || newReplicaSet == nil {
      return false, err
}
if !w.deploymentReady(newReplicaSet, currentDeployment) {
      return false, nil
}
```

# helm upgrade --wait --debug

Pros:

- Next logical enhancement of kubectl rollout
- Easy to add new roles

Cons:

- Can only watch one revision
- Traversing downwards requires hardcoding type-specific knowledge

# kubespy trace

Comprehensive checks about the state of your rollout.

Hard to use as a library.

But you can learn a lot by reading it!

# kubespy trace

# kubespy trace (crash)

# kubespy trace

```
deploymentEvents, err := watch.Forever("apps/v1", "Deployment",
     watch.ThisObject(namespace, name))

replicaSetEvents, err := watch.Forever("apps/v1", "ReplicaSet",
     watch.ObjectsOwnedBy(namespace, name))

podEvents, err := watch.Forever("v1", "Pod", watch.All(namespace))

table := map[string][]k8sWatch.Event{} // apiVersion/Kind → []k8sWatch.Event
repSets := map[string]k8sWatch.Event{} // Deployment name → Pod
pods := map[string]k8sWatch.Event{}    // ReplicaSet name → Pod
```

# kubespy trace

How it works:

- Creates a table of Deployments, ReplicaSets, Pods
- Uses owner references to match Pods with ReplicaSets
- Understands both top-down (Deployment -> ReplicaSet) and bottom-up (Pod -> ReplicaSet) relationships

# kubespy trace

Why both?

- Bottom-up can tell you Who Does What
  (who created pod X)
- Top-down can tell you The Path It Took
  (what this revision did)

# kubespy trace

Pros:

- Can tell you best why something is failing

Cons:

- Lots of indexing up-front
- Lots of type-specific analysis

# Tilt up

So what does Tilt do today?

1) Watches all pods
2) Tracks UIDs of all deployed resources
3) Traverses owner references upwards (like `kubespy`)
4) Uses content-based labels to handle revisions

# Tilt up

On content-based labels:

1. The label changes if and only if content changes.
2. All images in Tilt use content-based labels.
3. Deployments have a label that's a hash of PodTemplateSpec.

   (Same strategy used by DeploymentController to match ReplicaSets.)

# Tilt up

# Tilt up (crash)

# Tilt up

Pros:

- Works for deployments or CRDs
- Also used to track owners of Events!

Cons:

- Difficult to optimize
- Difficult to add new checks

1. Problem Statement
2. Why This Is Hard
3. Examples: kubectl, helm --wait, kubespy trace

# 4. Rant Time

# Rant Time

Please don't

come away taking this as recommending
a particular tool for tracking Deployments.

# Rant Time

kubectl tree:

```
// getAllResources finds all API objects in specified
// API resources in all namespaces (or non-namespaced).

func getAllResources(
  client dynamic.Interface, apis []apiResource, allNs bool)
  ([]unstructured.Unstructured, error) {
```

# Rant Time

All these approaches are WAY too complex.

Please do come away energized
at how much work there is to do!

(And it’s nobody’s fault, we just need to make it better.)

# What Does Kubernetes Give You?

Bottom-up analysis:

- Owner references only
- No attribution of which revision did what

What would it look like for the API to support object graphs efficiently?

# What Does Kubernetes Give You?

Top-down paths:

- Deployments have `deployment.kubernetes.io/revision` (auto-incrementing annotation)
- ReplicaSets have `pod-template-hash` content-based label
- Status is sometimes propagated “up” in a type-specific way

Could Kubernetes generalize these patterns for all types?

# Cascading Kubernetes Specs

Kubernetes, Operators, and Developers all
contribute desired state rules, and Kubernetes interprets it

- CRDs and Operators: New types of rules
- Mutating admission: New ways to interpret old rules

What would a CSS debugger for Kubernetes specs look like?

# Thank you!

# Nick Santos he/him

- nick@tilt.dev
- @nicksantos
- #tilt@slack.k8s.io
- tilt.dev

Other awesome humans
- @ellenkorbes
- @supermombartz

**Thanks to these projects**
- client-go https://github.com/kubernetes/client-go
- kubectl https://github.com/kubernetes/kubectl
- Helm https://github.com/helm/helm
- Kubespy https://github.com/pulumi/kubespy

# Questions?