kubectl apply

...and the dark art of declarative object management

Aaron Levy

CoreOS

github/slack: @aaronlevy

Why "the dark art"?

- Because Harry Potter references get your talk accepted

And...

- Because kubectl apply may not behave how you expect

Why "the dark art"?

My original understanding of the `kubectl apply` behavior:

It... "applies" configuration, right?

```
$ kubectl apply --help
Apply a configuration to a resource by filename or
stdin. This resource will be created if it doesn't
exist yet.
```

- Perfect. Talk over.

Why "the dark art"?

When I started more heavily using 'apply', I started to see:

- Inconsistent behavior across various object types
- Inconsistent behavior across various fields
- Unexpected (and somewhat vague) errors

*most of these were my fault

Why "the dark art"?

I didn't really understand how 'apply' worked.
So I began digging into the behavior:

- How are field values calculated?
- How are patches generated?
- How is the final object generated?
- Is the functionality client or server side (or both)?

What does kubectl apply do?

kubectl apply --help

- When invoked, does a three-way diff between the previous configuration, the provided input and the current configuration of the resource, in order to determine how to modify the resource.

- Applies the changes you've made, without overwriting changes to properties you haven't specified.

How `apply` changes are calculated

Calculating the changes to the object are done by evaluating three sources:

- Object configuration file
  - A file that defines the configuration for a Kubernetes object.
- Live object configuration
  - The object as it exists in the Kubernetes cluster
- Last Applied Configuration
  - View of the object the last time `apply` was invoked

Create object

```
$ kubectl apply --filename my-app.yaml
```

Creates object(s), but also sets the annotation:

kubectl.kubernetes.io/last-applied-configuration

- Set to match the object configuration file.
- Used to compute field add/update/delete

last-applied-configuration

Base Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: my-app:v1
```

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
           {"apiVersion":"apps/v1beta1","kind":"Deployment","metadata":
           {"annotations":{},"name":"my-app","namespace":"default"},"sp
           ec":{"template":{"metadata":{"labels":{"app":"my-app"}},"spe
           c":{"containers":[{"image":"my-app:v1","name":"my-app"}]}}}}
spec:
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Example of adding a field

Local Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  minReadySeconds: 5
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Last Applied

**part of the live object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Live Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

```
$ kubectl apply --filename my-app.yaml
Exists in local, but not on last-applied/live. Action: Add
```

Local Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  minReadySeconds: 5
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Last Applied

**part of the live object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  minReadySeconds: 5
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Live Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  minReadySeconds: 5
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Field has been added to live object, and the last-applied annotation (to be used in future calculations).

Example of deleting a field

Local Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Last Applied

**part of the live object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  minReadySeconds: 5
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Live Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  minReadySeconds: 5
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

```
$ kubectl apply --filename my-app.yaml
Exists in last/live, but not in local object. Action: Delete
```

Local Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Last Applied

**part of the live object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Live Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Field removed from last-applied annotation, and live object.

Okay, it can add and remove fields.

What else?

Preserving & Enforcing Fields

Allow some fields to be "enforced" by being specified as part of your object configuration .

If a field is left unspecified, it will be ignored during the patch calculations.

Leaving some fields able to be controlled by other components. For example, an autoscaler managing replicas.

Local Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Last Applied

**part of the live object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Live Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Replica count only exists in the live object. It is not defined in our local config (do not change during apply).

Local Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Last Applied

**part of the live object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Live Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 5
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Increase replica count external to the object configuration.

```
$ kubectl scale deployment/my-app --replicas=5
```

Local Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v2
```

Last Applied

**part of the live object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

Live Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 5
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v1
```

We now want to update the container image:

```
$ kubectl apply --filename my-app.yaml
```

Local Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v2
```

Last Applied

**part of the live object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v2
```

Live Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 5
  template:
    spec:
      containers:
      - name: my-app
        image: myapp:v2
```

Container image is changed, while replica count in live object is ignored / preserved during the update.

Let's talk about merge calculations

We've seen how `kubectl apply` can add, update, remove, and preserve object fields.

But how are these field values being calculated?

Merge Calculations

There are several ways that different field types can be merged:

- Primitives / string, int, boolean (examples: image, replicas)
  - Action: Replace
- Maps / objects (examples: labels, metadata, spec)
  - Action: Merge elements, or subfields
- Lists (examples: containers, ports, args)
  - Action: Depends...

Merge Calculations - Lists

Several strategies, which depends on the field:

- Replace entire list in-place
- Merge elements in a list of objects

Local Object

```
containers:
- name: my-app
  image: myapp:v2
  args: ["a", "c"]
```

Last Applied

**part of the live object

```
containers:
- name: my-app
  image: myapp:v2
  args: ["a", "b"]
```

Live Object

```
containers:
- name: my-app
  image: myapp:v2
  args: ["a", "b", "d"]
```

$ kubectl apply --filename my-app.yaml

```
containers:
- name: my-app
  image: myapp:v2
  args: ["a", "c"]
```

```
containers:
- name: my-app
  image: myapp:v2
  args: ["a", "c"]
```

```
containers:
- name: my-app
  image: myapp:v2
  args: ["a", "c"]
```

Merge Calculations - Lists (Objects)

Local Object

```
containers:
- name: app
  image: app:v1
- name: sidecar
  image: sidecar:v0.1.0
```

Last Applied

**part of the live object

```
containers:
- name: app
  image: app:v1
```

Live Object

```
containers:
- name: app
  image: app:v1
  args: ["prod"]
```

$ kubectl apply --filename app.yaml

```
containers:
- name: app
  image: app:v1
- name: sidecar
  image: sidecar:v0.1.0
```

```
containers:
- name: app
  image: app:v1
- name: sidecar
  image: sidecar:v0.1.0
```

```
containers:
- name: app
  image: app:v1
  args: ["prod"]
- name: sidecar
  image: sidecar:v0.1.0
```

Local Object

```
tolerations:
- key: "baz"
  operator: "Exists"
  effect: "NoSchedule"
```

Last Applied

**part of the live object

```
tolerations:
- key: "foo"
  operator: "Exists"
  effect: "NoSchedule"
```

Live Object

```
tolerations:
- key: "foo"
  operator: "Exists"
  effect: "NoSchedule"
- key: "bar"
  operator: "Exists"
  effect: "NoSchedule"
```

Expected actions:

- Add "baz"
- Delete "foo"
- Ignore/preserve "bar"

```
tolerations:
- key: "bar"
  operator: "Exists"
  effect: "NoSchedule"
- key: "baz"
  operator: "Exists"
  effect: "NoSchedule"
```

Local Object

```
tolerations:
- key: "baz"
  operator: "Exists"
  effect: "NoSchedule"
```

Last Applied

**part of the live object

```
tolerations:
- key: "foo"
  operator: "Exists"
  effect: "NoSchedule"
```

Live Object

```
tolerations:
- key: "foo"
  operator: "Exists"
  effect: "NoSchedule"
- key: "bar"
  operator: "Exists"
  effect: "NoSchedule"
```

$ kubectl apply --filename app.yaml

```
tolerations:
- key: "baz"
  operator: "Exists"
  effect: "NoSchedule"
```

```
tolerations:
- key: "baz"
  operator: "Exists"
  effect: "NoSchedule"
```

```
tolerations:
- key: "baz"
  operator: "Exists"
  effect: "NoSchedule"
```

Merge Calculations - Lists

- Expected to see list of tolerations merged, but instead they were replaced.
- Why did this happen?

A (very) brief patch explainer

Strategies:

- JSON Merge Patch
  - https://tools.ietf.org/html/rfc7386
- Strategic Merge patch
  - Custom to Kubernetes

Strategic Merge Patch

With a Strategic Merge Patch, you can:

- Treat a list much like a map, and merge elements of the list based on predefined patchMergeKey.
- Individual elements are then added/updated/removed

Strategic Merge - patchMergeKey

- Defined on a per-field basis
- Exists in the Kubernetes source code

Lookup directly:
https://github.com/kubernetes/api/blob/master/core/v1/types.go

Or via api-reference
https://kubernetes.io/docs/api-reference/v1.8

Strategic Merge - patchMergeKey

```
type PodSpec struct {
    // +patchMergeKey=name
    // +patchStrategy=merge
    Containers []Container `patchStrategy:"merge" patchMergeKey:"name"`

    // +optional
    Tolerations []Toleration

    // +optional
    // +patchMergeKey=name
    // +patchStrategy=merge
    Volumes []Volume `patchStrategy:"merge" patchMergeKey:"name"`
}
```

But wait! There's More!

Merge Calculations - Defaulted Fields

"Defaulted" fields may be added to the object.

On a deployment object, for example:

- Replicas defaults to: 1
- Update strategy defaults to: RollingUpdate

In some cases, updating a defaulted field can be problematic.

Local Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    # ...
```

Last-Applied

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    # ...
```

Live Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 1
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge : 1
      maxUnavailable: 1
  template:
    # ...
```

Local Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  strategy:
    type: Recreate
  template:
    # ...
```

Last-Applied

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    # ...
```

Live Object

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 1
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge : 1
      maxUnavailable: 1
  template:
```

spec.strategy.type "recreate" incompatible with
spec.strategy.rollingUpdate

More "gotchas"

We now have a pretty good understanding of apply behavior.

However, there are considerations when using kubectl apply with other object management techniques.

```
user1@foo:~ $ cat my-app.yaml

apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: myapp:v0.1.0
```

* Note that `replicas` is not present

User 1 creates initial "my-app" deployment:

```
user1@foo:~ $ kubectl apply -f my-app.yaml
Deployment "my-app" created
```

Over time the app is scaled up:

```
user1@foo:~ $ kubectl scale deployment my-app --replicas=3
```

Later, user 1 bumps the application version:

```
user1@foo:~ $ sed -i 's/v0.1.0/v0.2.0' my-app.yaml
user1@foo:~ $ kubectl apply -f my-app.yaml
```

User 2 is adding a volume, but doesn't have local copy of app

```
user2@bar:~ $ kubectl get deployment/my-app \
-o yaml > app-copy.yaml

user2@bar:~ $ vim app-copy.yaml
# ...
metadata:
  name: my-app
spec:
  replicas: 3
  template:
    spec: # ...
      volumes:
      - name: data
        hostPath:
          path: /data
```

User 2 thinks "Aaron said to use `apply`, so…"

```
user2@bar:~ $ kubectl apply -f app-copy.yaml
deployment "my-app" configured
```

Later, user 1 wants to bump app version again

```
user1@foo:~ $ sed -i 's/v0.2.0/v0.3.0/' my-app.yaml
user1@foo:~ $ git commit -am 'Bump v0.3.0'
user1@foo:~ $ kubectl apply -f my-app.yaml
```

User 1 inadvertently reset replicas and removed volume!

Local Object

```
apiVersion:
apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  containers:
    [...]
```

Last-Applied

```
apiVersion:
apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  containers:
     [...]
  volumes:
  - name: data
```

Live Object

```
apiVersion:
apps/v1beta1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  containers:
    [...]
  volumes:
  - name: data
```

- User 2 inadvertantly added fields to "last-applied". Changes user 1's actions into "deletion" events.

What happened?

User 1's workflow did not change:

1) Modify source config
2) Use kubectl apply

But, user 2's edit changed behavior of user 1's workflow.

- Wanted: updated image field (v0.3.0)
- Got:
  - replicas reset to 1,
  - new volume removed (and other fields too)

Aaron's brief list of recommendations

- Don't define replicas in the local object configuration file
  - And/or other fields that might be "externally managed"
- Explicitly define defaulted fields (e.g. update strategy)
  - If you need to change in the future, they are "managed"
- Use apply consistently (from same source config object)
  - Mixing imperative commands (create/edit/set) can lead to unintended outcomes (unless you're sure of what you're doing)

Things we didn't get to cover

- `kubectl apply --prune` & declarative object deletion
- Field conflicts when using `kubectl apply --overwrite=false`
- Interacting with last-applied-configuration annotation with
  - `kubectl apply {view,set,edit}-last-applied`
- `kubectl patch` command

Homework...

Object management documentation

- https://goo.gl/GcUqHv

Using kubectl patch

- https://goo.gl/Kyb6RX

Apply "v2" refactor proposal

- https://goo.gl/MRUCX6

Declarative Application Management

- https://goo.gl/T66ZcD

Issues related to declarative application management

- https://goo.gl/UGHLJk

Thank You!

Questions?

aaron.levy@coreos.com
github/slack: @aaronlevy