KubeCon | CloudNativeCon

North America 2019

kubectl development workshop

Sean Sullivan - Google
co-chair SIG CLI
github: seans3
slack: seans

Goals

This workshop is intended to create a more thriving and knowledgeable kubectl development community. We hope to give developers interested in kubectl (and SIG CLI in general) the skills necessary to quickly become productive in this area.

Outline

kubectl development workshop

A) Basic Structure of a kubectl subcommand: Cobra/Options/Flags
Codelab: Connect new kubectl subcommand

B) Communicating with the APIServer and Converting Resources
Codelab: Add a resource.Builder and resource.Helper

C) Printing with Printers and the ResourcePrinter interface
Codelab: Add printing to new kubectl subcommand

D) kubectl Unit Testing
Codelab: Add unit test to new kubectl subcommand

Part A: Structure of kubectl subcommand

Cobra/Options/Flags

```
// Also stores flag values
type FooOptions struct {}

// Returns a pointer to the structure encapsulating the command
func NewCmdFoo(factory, ioStreams) *cobra.Command {
  o := NewFooOptions()
  cmd := &cobra.Command{
    // usage and help fields: usage, short help, long help, examples
    Run : func (cmd *cobra.Command, args []string) error {
      o.Complete() // Fill in the options struct
      o.Validate() // Validate the options struct
      o.RunFoo()   // Run the command using the options values
    }
  }
  // Define flags the command understands; stores values in options
  cmd.Flags.StringVar(&o.flag, name, default, usage)
}
```

Part A: Cobra.Command

Cobra is a library providing a simple interface to create powerful modern CLI interfaces similar to git & go tools.

- Structures for subcommand-based CLIs
- Fully POSIX-compliant flags (including short & long versions)
- Nested subcommands
- Global, local and cascading flags
- Intelligent suggestions (app srver... did you mean app server?)
- Automatic help generation for commands and flags
- Automatic help flag recognition of -h, --help, etc.

https://github.com/spf13/cobra

Part A: Parameters to kubectl subcommand

Factory/IOStreams

Factory

// functions which return clients

```
factory cmdutil.Factory
   // NewBuilder returns an object that assists in loading objects
   // from both disk and the server and which implements
   // patterns for CLI interactions with generic resources.
   NewBuilder() *resource.Builder
```

IOStreams

```
encapsulates stdin/stdout/stderr
var ioStreams genericclioptions.IOStreams
```

Codelab: Preliminaries

```
Go Version: >= 1.13.4
$ go version

Code location:

K8SROOT/staging/src/k8s.io/kubectl/
K8SROOT/pkg/kubectl/
K8SROOT/staging/src/k8s.io/cliruntime/

From K8SROOT:

bazel test //staging/src/k8s.io/kubectl/...
bazel test //pkg/kubectl/...
bazel build //cmd/kubectl

kubectl binary location:

K8SROOT/bazel-bin/cmd/kubectl/linux_amd64_pure_stripped/kubectl

make kubectl
make kubectl WHAT=./pkg/kubectl
```

Codelab: Preliminaries

Verifying the compiled kubectl

```
$ kubectl version
Client Version: version.Info{Major:"1", Minor:"18+",
GitVersion:"v1.18.0-alpha.0.1000+5c54e4b6baf555-dirty",
GitCommit:"5c54e4b6baf5557ddaf98024609282189274978a",
GitTreeState:"dirty", BuildDate:"2019-11-15T05:02:45Z",
GoVersion:"go1.13.4", Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"14+",
GitVersion:"v1.14.7-gke.23",
GitCommit:"81c87c699557fed991e292cd328b2129c2f242a2",
GitTreeState:"clean", BuildDate:"2019-11-07T19:23:23Z",
GoVersion:"go1.12.11b4", Compiler:"gc", Platform:"linux/amd64"}
```

Codelab: Setup

Option 1 From empty dir:

```
$ git clone
https://github.com/seankubecon/kubernetes.git
$ cd kubernetes
```

Option 2 From the K8SROOT:

```
$ git remote add kubecon-workshop
https://github.com/seankubecon/kubernetes.git
$ git fetch kubecon-workshop
```

Next:

```
$ git checkout kubecon-cs-workshop
$ ls staging/src/k8s.io/kubectl/pkg/cmd/foo
BUILD  foo.go  foo_test.go
$ bazel build //cmd/kubectl
```

Codelab: Connect kubectl Foo Command

1) Connect the NewCmdFoo() to other kubectl subcommands
2) `$ kubectl foo -h`
3) `$ kubectl foo options`

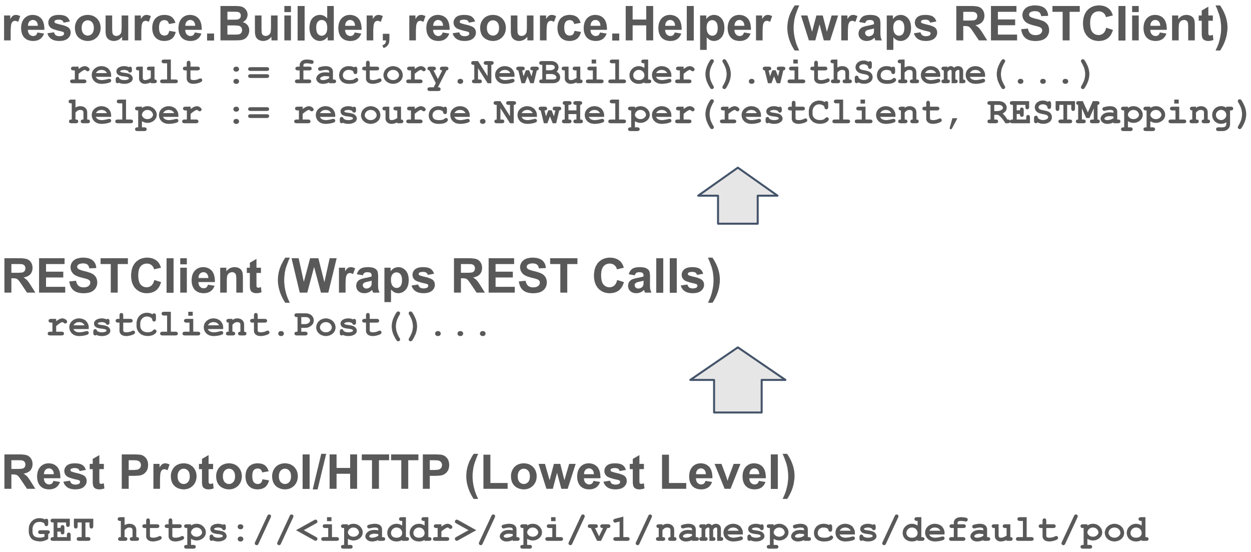
```
K8SROOT/pkg/kubectl/cmd/cmd.go (connect subcommands)
K8SROOT/cmd/kubectl/kubectl.go (main)
```

Part B

Just enough APIMachinery to ~~get confused~~ develop kubectl

Part B: APIMachinery Layers

APIMachinery Layers



resource.Builder, resource.Helper (wraps RESTClient)

```
result := factory.NewBuilder().withScheme(...)
helper := resource.NewHelper(restClient, RESTMapping)
```

RESTClient (Wraps REST Calls)

```
restClient.Post()...
```

Rest Protocol/HTTP (Lowest Level)

```
GET https://<ipaddr>/api/v1/namespaces/default/pod
```

Part B: REST Request over HTTP

Example of what HTTP REST call looks like (Method, URL, Body)

```
GET https://<ipaddr>/api/v1/namespaces/default/pods?limit=500
Request Headers:
  Accept: application/json;as=Table;v=v1beta1;g=meta.k8s.io,
application/json
  User-Agent: kubectl/v1.14.7 (linux/amd64) kubernetes/8fca2ec

Try:
$ kubectl get po -v=8
$ kubectl get po -v=10 -o yaml
```

Part B: REST Response over HTTP

REST HTTP Response

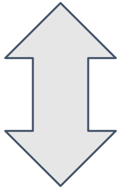
```
Response Headers:
    Date: Fri, 08 Nov 2019 21:45:53 GMT
    Audit-Id: 052a40f5-15ab-438b-b7c5-bd65b356bf5d
    Content-Type: application/json
    Content-Length: 3772
Response Body:
{"kind":"Table","apiVersion":"meta.k8s.io/v1beta1","metadata":{"selfLink
":"/api/v1/namespaces/default/pods","resourceVersion":"6270266"},"column
Definitions":

...
```

Part B: APIMachinery -- Why?

APIMachinery & all the structures and code is basically trying to transform JSON blob (resource YAML) into/out of a golang struct for that type.

JSON bytes
Response Body: {"kind":"Pod","apiVersion":"v1", …



golang struct

```
// Pod is a collection of containers that can run on a host. This resource is created
// by clients and scheduled onto hosts.
type Pod struct {
    metav1.TypeMeta `json:",inline"`
...
```

Part B: APIMachinery

There is a significant amount of complexity here

Glossary of APIMachinery structures:

```
GVK (Group/Version/Kind): just three strings
  Example: apps/v1/Deployment, core/v1/Pod
GVR (Group/Version/Resource): basically the same as a GVK
RESTMapping: basically a GVK and/or a GVR with namespace scope
Scheme: All the GVK's that the app (kubectl) knows how to transform
Codec/Serializer
  Encoder: Transform the go struct into JSON bytes
  Decoder: Transform the JSON bytes into go struct
```

Part B: clientgo.RESTClient

```
// staging/src/k8s.io/client-go/rest/client.go
//
// Interface captures the set of operations for generically interacting
// with Kubernetes REST apis.
type Interface interface {
    GetRateLimiter() flowcontrol.RateLimiter
    Verb(verb string) *Request
    Post() *Request
    Put() *Request
    Patch(pt types.PatchType) *Request
    Get() *Request
    Delete() *Request
    APIVersion() schema.GroupVersion
}
```

Part B: resource.RESTClient

```
// K8SROOT/staging/src/k8s.io/cli-runtime/pkg/resource/interfaces.go
//
// This interface narrows the client-go RESTClient interface slightly.
type RESTClient interface {
    Get() *rest.Request
    Post() *rest.Request
    Patch(types.PatchType) *rest.Request
    Delete() *rest.Request
    Put() *rest.Request
}
```

Part B: resource.RESTClient

Example of using a RESTClient. Most of the code is actually on the rest.Request object. The rest.Request uses the Builder pattern. The following creates the passed *obj* on the APIServer. The call will return a rest.Result:

```
restClient.Post().
        NamespaceIfScoped(namespace, m.NamespaceScoped).
        Resource(resource).
        VersionedParams(options, metav1.ParameterCodec).
        Body(obj).
        Do().
        Get()
```

Part B: resource Builder/Helper

The following two structures are at the level of abstraction of the app (kubectl). You will see these two structures frequently in kubectl. These structures wrap the `resource.RESTClient`:

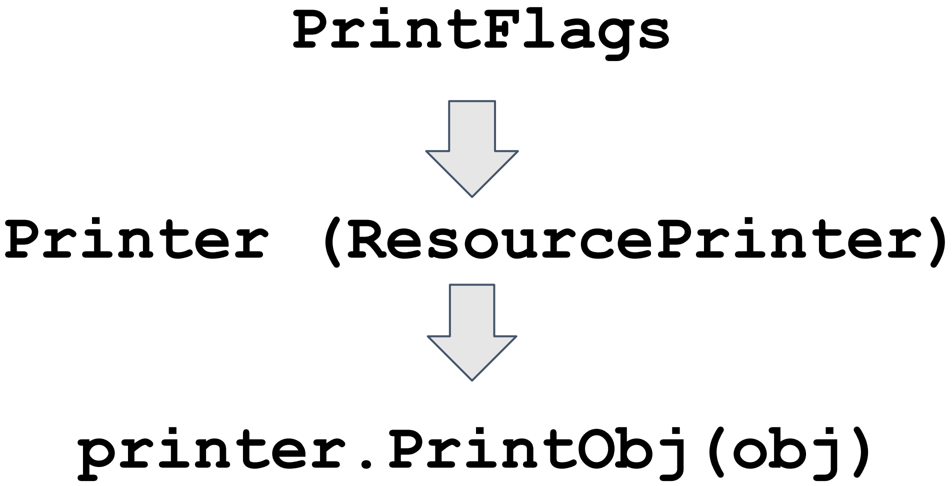
1) `resource.Builder` is used for retrieving/decoding resources, whether from the local file system (YAML files) or from the APIServer.
2) `resource.Helper` is a wrapper around the RESTClient. It is used for communicating with the APIServer.

Codelab: resource Builder & Helper

1) Add a resource Builder to the foo subcommand
   - Builder will allow reading a local YAML file into a runtime.Object
   - Look at other kubectl subcommands to see the necessary Builder params for reading a file off the local filesystem (e.g. FilenameOptions).

2) Add a resource Helper to create the resource (that was read with the Builder) on the API Server (e.g. helper.Create(...))

Part C: kubectl printing



PrintFlags

Printer (ResourcePrinter)

printer.PrintObj(obj)

Part C: kubectl printing

Some kubectl print options:

```
$ kubectl get po -o name
$ kubectl get po -o yaml
$ kubectl get po -o json
$ kubectl get po -o jsonpath=...
```

Each one of these corresponds to a ResourcePrinter. These live at:

```
K8SROOT/staging/src/k8s.io/cli-runtime/pkg/printers
```

We get to use these standard printers by filling in PrintFlags:

```
K8SROOT/staging/src/k8s.io/cli-runtime/pkg/genericclioptions
```

```
PrintFlags: genericclioptions.NewPrintFlags("op").WithDefaultOutput("name")
```

Codelab: Add printing

1) Add `genericclioptions.PrintFlags` to `FooOptions`

2) Create new `PrintFlags` with default output as “name” Printer

3) Create Printer from flags

4) Call `printer.PrintObj(obj)` on object created previously with Builder

Part D: kubectl unit tests

Example Unit Test (delete_test.go):

```
// Create a fake/test factory using the “test” namespace.
testfactory := cmdtesting.NewTestFactory().WithNamespace("test")
defer tf.Cleanup()

// Create a real Codec with the Scheme.
codec := scheme.Codecs.LegacyCodec(scheme.Scheme.PrioritizedVersionsAllGroups()...)

// Create a fake RESTClient, mocking the HTTP Response (based on HTTP Request).
testfactory.UnstructuredClient = &fake.RESTClient{...}

// Create fake IOStreams.
streams, _, buf, _ := genericclioptions.NewTestIOStreams()

// Create the kubectl subcommand, and pass some flags.
cmd := NewCmdDelete(tf, streams)
cmd.Flags().Set("namespace", "test")
cmd.Flags().Set("cascade", "false")
cmd.Flags().Set("output", "name")

// Execute the command, and check what was printed to the buffer.
cmd.Run(cmd, []string{"replicationcontrollers/redis-master-controller"})
if buf.String() != "replicationcontroller/redis-master-controller\n" {
  t.Errorf("unexpected output: %s", buf.String())
}
```

Part D: kubectl unit tests

Example Fake RESTClient:

```
    testfactory.UnstructuredClient = &fake.RESTClient{
        Client: fake.CreateHTTPClient(func(req *http.Request) (*http.Response, error) {
            switch p, m := req.URL.Path, req.Method; {

            // replication controller with cascade off
            case p == "/namespaces/test/replicationcontrollers/redis-master-controller" && m
== "DELETE":
                return &http.Response{StatusCode: http.StatusOK, Header:
cmdtesting.DefaultHeader(), Body: cmdtesting.ObjBody(codec, &rc.Items[0]\
)}, nil

            // secret with cascade on, but no client-side reaper
            case p == "/namespaces/test/secrets/mysecret" && m == "DELETE":
                return &http.Response{StatusCode: http.StatusOK, Header:
cmdtesting.DefaultHeader(), Body: cmdtesting.ObjBody(codec, &rc.Items[0]\
)}, nil

            default:
                // Ensures no GET is performed when deleting by name
           t.Fatalf("unexpected request: %#v\n%#v", req.URL, req)
                return nil, nil
            }
        }),
    }
```

Codelab: kubectl unit tests

1) Write a unit test for the Builder to fake out the creation of the pod

KubeCon | CloudNativeCon

North America 2019