Building Testable CLIs With Cobra

by Simon Bein (@SimonTheLeg)

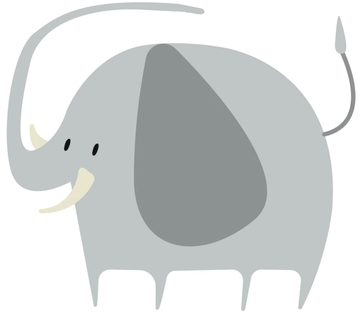
Comprehension Questions Gladly During The Talk

This is an opinionated talk

💡 = my takeaway

Talk is not about Pro/Cons of cobra

Let’s discuss this during the cold 🍻



freepik.com

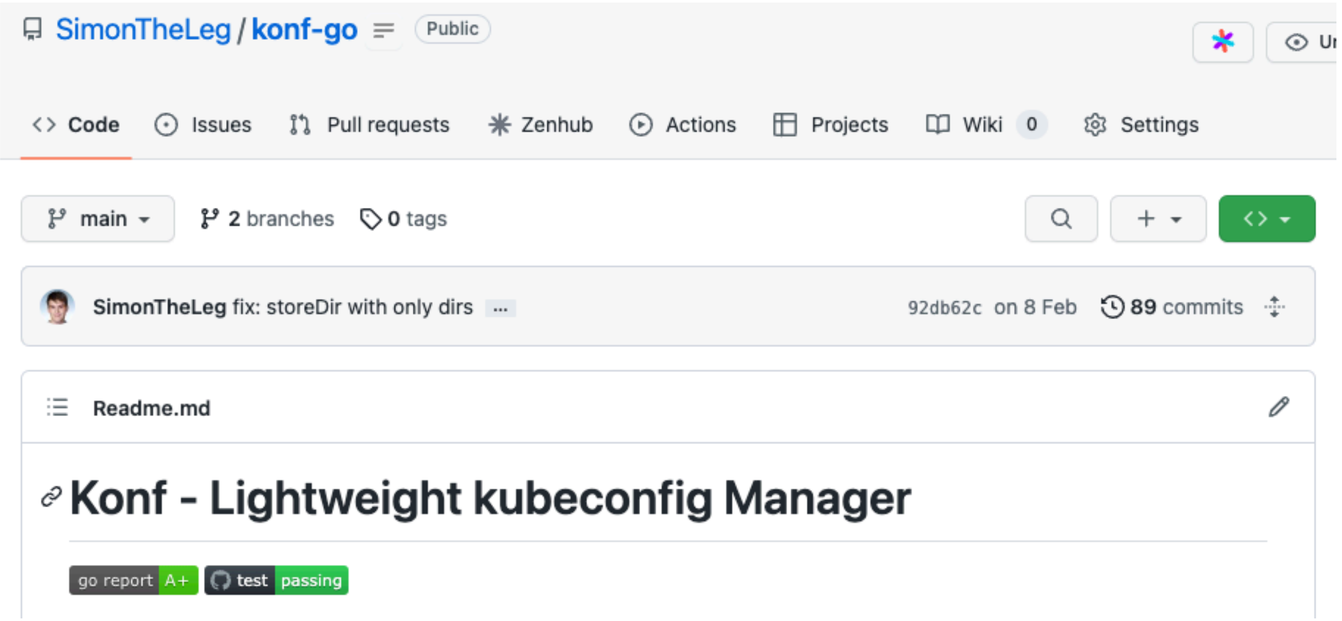
A Quick Search For “go cli library”

acmd
argparse
argv
carapace
carapace-bin
carapace-spec
cli
cli
climax
clir
cmd
cmdr
cobra
command-chain
commandeer
complete
Dnote
elvish
env
flag
flaggy
flagvar
go-andotp
go-arg
go-commander
go-flags
go-getoptions
gocmd
hiboot cli
job
kingpin
liner
mitchellh/cli
mow.cli
ops
pflag
sand
sflags
strumt
subcmd
ts
ukautz/clif
urfave/cli
wlog
wmenu

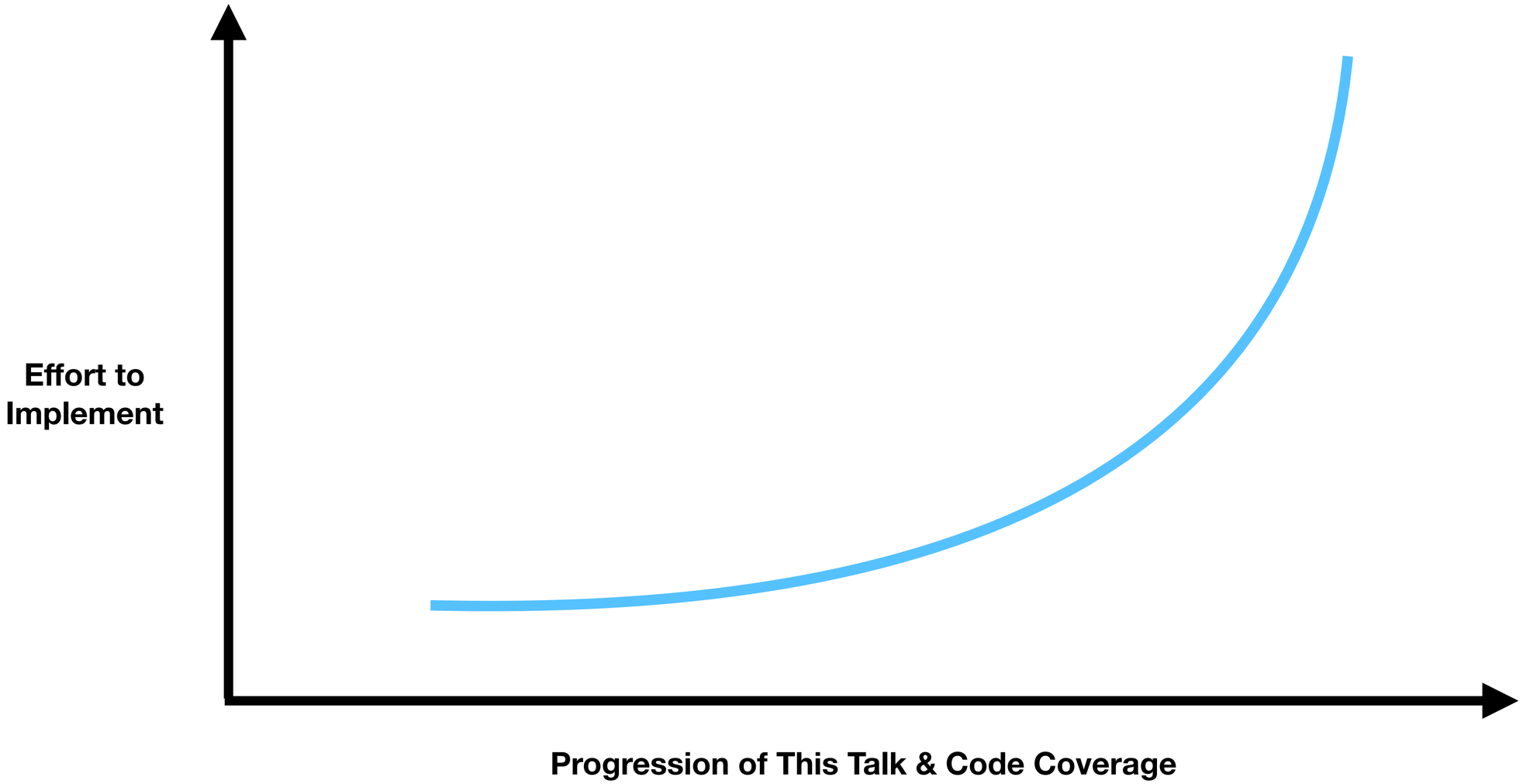
avelino/awesome-go

I recently built a cli with cobra and there were some things I wish I knew before

Shameless Plug: SimonTheLeg/konf-go



Structure of This Talk



What is Cobra?

Cli library to help with:

- Creating nested command-trees (e.g. kubectl create cm, kubectl delete cm)
- Handling flags at different levels (global, local, cascading)
- Generating auto-completion

What is Cobra-CLI?

Helper tool to generate necessary dirs, files and boilerplate for your commands

- setup root-command and main.go

```
❯ cobra-cli init
```

- add a new command

```
❯ cobra-cli add <your-cmd>
```

How Is a Cobra App Structured?

Cobra does not enforce a file structure, each child is just added to its parent…

A

```go
import (
 "github.com/simontheleg/root"
)

...

  var mycmd = &cobra.Command{}
  root.rootCmd.AddCommand(mycmd)
```

child.go

B

```go
import (
 "github.com/simontheleg/child"
)

...

  var rootcmd = &cobra.Command{}
  rootCmd.AddCommand(child.mycmd)
```

root.go

… But there are best practices

Standard* Cobra File Structure

```
❯ mkcd mycli
❯ cobra-cli init
❯ cobra-cli add cmd1
❯ cobra-cli add cmd2
```

→

```
mycli
├── cmd
|   ├── cmd1.go
|   ├── cmd2.go
|   └── root.go
└── main.go
```

→

```
❯ mycli cmd1

❯ mycli cmd2
```

** as in what `cobra-cli init` and `cobra-cli add <your-cmd>` produce*

Standard* Cobra File Structure

```
.
├── cmd
│   ├── cmd1.go
│   ├── cmd2.go
│   └── root.go
└── main.go
```

1

```go
func main() {
  cmd.Execute()
}
```

main.go

2

```go
var cmd1Cmd = &cobra.Command{
   // we'll go into more detail shortly
}

func init() {
  rootCmd.AddCommand(cmd1Cmd)
}
```

cmd1.go

3

```go
// Execute uses the args (os.Args[1:] by default) and run
// through the command tree finding appropriate matches
// for commands and then corresponding flags.
func Execute() {
  err := rootCmd.Execute()
  if err != nil {
    os.Exit(1)
  }
}
```

root.go

⟹ This works, because root.go and cmd1.go are in the same package

And So Far I Have Not Outgrown This

```
.
└── cmd
    ├── cleanup.go
    ├── cleanup_test.go
    ├── completion.go
    ├── completion_test.go
    ├── import.go
    ├── import_test.go
    ├── namespace.go
    ├── namespace_test.go
    ├── root.go
    ├── set.go
    ├── set_test.go
    ├── shellwrapper.go
    └── shellwrapper_test.go
```

Example for konf-go

Nested Commands Benefit From A Different Structure…

```
pkg/cmd
├── create
|   ├── create.go
|   ├── create_clusterrole.go
|   ├── create_clusterrole_test.go
|   ├── create_clusterrolebinding.go
|   ├── create_clusterrolebinding_test.go
|   ├── create_configmap.go
|   ├── create_configmap_test.go
|   ...
├── expose
|   ├── expose.go
|   └── expose_test.go
├── cmd.go
...
```

```
❯ kubectl create clusterrole
❯ kubectl create clusterrolebinding
❯ kubectl create configmap

❯ kubectl expose
```

Example for kubectl

… And an Inversion of Control

```
pkg/cmd
├── create
|   ├── create.go
|   ...
├── expose
|   ├── expose.go
├── cmd.go
...
```

```go
import (
  "k8s.io/kubectl/pkg/cmd/create"
  "k8s.io/kubectl/pkg/cmd/expose"
  ...
)

groups := templates.CommandGroups{
    {
      Message: "Basic Commands (Beginner):",
      Commands: []*cobra.Command{
        create.NewCmdCreate(f, o.IOStreams),
        expose.NewCmdExposeService(f, o.IOStreams),
      },
    },
}

...

func (g CommandGroups) Add(c *cobra.Command) {
  for _, group := range g {
    c.AddCommand(group.Commands...)
  }
}
```

kubectl/cmd.go

💡 Unless building nested commands, the standard structure should be sufficient

So is it also a good idea to use cobra-cli?

In my opinion: Not really..

So What Does `cobra-cli add` Generate?

```go
// cmd1Cmd represents the cmd1 command
var cmd1Cmd = &cobra.Command{
  Use:   "cmd1",
  Short: "A brief description of your command",
  Long:  `long description...`,
  Run: func(cmd *cobra.Command, args []string) {
    fmt.Println("cmd1 called")
  },
}

func init() {
  rootCmd.AddCommand(cmd1Cmd)
}
```

cmd/cmd1.go

Why I am Not A Big Fan

#1 Error Handling

Run cannot return an error, therefore you have to handle them like this:

```go
var cmd1Cmd = &cobra.Command{
  ...
  Run: func(cmd *cobra.Command, args []string) {

     workDir, err := os.Getwd()
     cobra.CheckErr(err)

     …

   },
}
```

cmd1.go

```go
func CheckErr(msg interface{}) {
  if msg != nil {
    fmt.Fprintln(os.Stderr, "Error:", msg)
    os.Exit(1)
  }
}
```

cobra.go

Rescue Is Around The Corner: RunE

```go
var cmd1Cmd = &cobra.Command{
  RunE: func(cmd *cobra.Command, args []string) error {

     workDir, err := os.Getwd()
     if err != nil {
      return err
     }

     ...

  },
}
```

cmd/cmd1.go

#2 Testability

If you choose to declare the func inside the var block, you can test it like this

```go
func TestCmd1(t *testing.T) {
  _, err := cmd1Cmd.ExecuteC()
  if err != nil {
    t.Error(err)
  }
}
```

cmd1_test.go

```go
func (c *Command) ExecuteC() (cmd *Command, err error) {
  ...

  // Regardless of what command execute is called on,
  // run on Root only
  if c.HasParent() {
    return c.Root().ExecuteC()
  }
  ...
```

cobra/command.go

But you have to be aware, that this will call the root command instead of the parent

So I Like To Extract Into Its Own Func And Test It

```go
var cmd1Cmd = &cobra.Command{
  ...
  RunE:  cmd1,
}

func cmd1(cmd *cobra.Command, args []string) error {
 ...
}
```

cmd1.go

```go
func TestCmd1(t *testing.T) {

  err := cmd1(nil, []string{})
  if err != nil {
    t.Error(err)
  }
}
```

cmd1_test.go

Summary

```go
var cmd1Cmd = &cobra.Command{
  Use:   "cmd1",
  Short: "A brief description of your command",
  Long:  `long description...`,
  Run: func(cmd *cobra.Command, args []string) {
     ...
     cobra.CheckErr(err)
  },
}

func init() {
  rootCmd.AddCommand(cmd1Cmd)
}
```

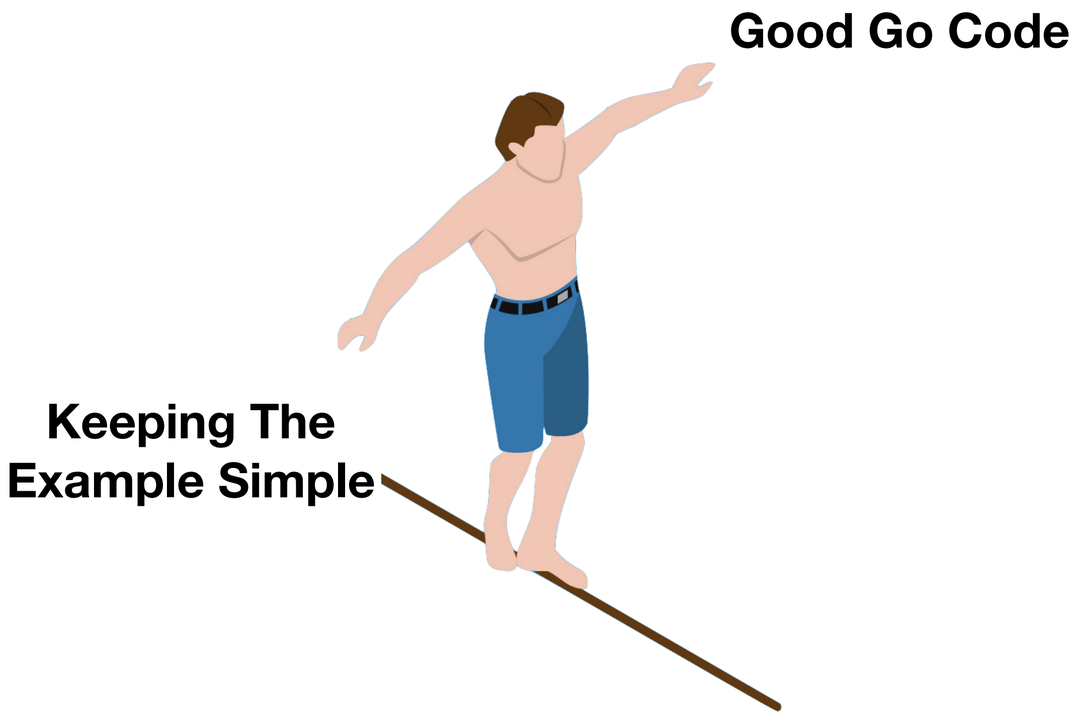
cmd1.go

```go
var cmd1Cmd = &cobra.Command{
  Use:   "cmd1",
  Short: "A brief description of your command",
  Long:  `long description...`,
  RunE:  cmd1,
}

func cmd1(cmd *cobra.Command, args []string) error {
  ...
  return nil
}

func init() {
  rootCmd.AddCommand(cmd1Cmd)
}
```

m cmd1.go*

💡 All the little tweaks add up, so I rather just copy-paste an old command instead of using `cobra-cli add`

Handling External Components



freepik.com

Let’s Say You Want Filesystem-Access

We want to read a file based on the passed arguments

```go
import (
  "io/fs"
  "os"
)

var cmd1Cmd = &cobra.Command{
  ...
  Args: cobra.MinimumArgs(1),
}

func cmd1(cmd *cobra.Command, args []string) error {
  fsys := os.DirFS("")
  f, err := fs.ReadFile(fsys, args[1])
  return nil
}
```

cmd1.go

This is not easy to test, because we cannot switch out os.DirFS and extracting it would not allow us to test for args

In These Cases I Like To Create My Own Type

```go
type cmd1Cmd struct {
  fs  fs.FS
  cmd *cobra.Command
}

func (c *cmd1Cmd) cmd1(cmd *cobra.Command, ... {
  f, err := c.fs.ReadFile(fsys, args[1])
  fmt.Println(f)
  return err
}

func newCmd1Cmd() *cmd1Cmd {
  return &cmd1Cmd{
    fs: os.DirFS(""),
    cmd: &cobra.Command{
      Short: "cmd1",
      Args:  cobra.MinimumNArgs(1),
      RunE: cmd1
    },
  }
}

func init() {
  rootCmd.AddCommand(newCmd1Cmd().cmd)
}
```

cmd1.go

```go
func TestCmd1(t *testing.T) {

  fsys := fstest.MapFS{
    "file1": {},
    "file2": {},
  }

  cmd1 := &cmd1Cmd{
    fs:  fsys,
    cmd: &cobra.Command{
      ...
    },
  }

  // will work
  cmd1.cmd1(nil, []string{“create", "file1"})
  // will fail in cobra assessment
  cmd1.cmd1(nil, []string{})
  // will fail in our code (file not found)
  cmd1.cmd1(nil, []string{“file-not-exist”})

}
```

cmd1_test.go

Sidenote: Sometimes It Can Make Sense To Offload Cobra Logic Into Your Run Func

And directly test on the cmd func

```go
func cmd1(cmd *cobra.Command, args []string) error {
  if len(args) < 1 {
    return fmt.Errorf("Args must be at least 1”)
  }

  fsys := os.DirFS("")
  f, err := fs.ReadFile(fsys, args[1])
  fmt.Println(f)
  return err
}
```

cmd1.go

```go
func TestCmd1(t *testing.T) {

  cmd1(nil, []string{"create", "file1"})
  cmd1(nil, []string{})
  cmd1(nil, []string{“file-not-exist"})

}
```

cmd1_test.go

💡 I like creating my own type for commands and offloading some cobra logic directly into the command

Squeezing Out the Last Percent in Complicated Use-Cases

freepik.com

Injecting Funcs Into Your Custom Type

And then mocking them out in your tests

```go
type namespaceCmd struct {
  fs               afero.Fs
  promptFunc       prompt.RunFunc
  selectNamespace  func(clientSetCreator, ...
  setNamespace     func(afero.Fs, string) error
  clientSetCreator clientSetCreator
  cmd              *cobra.Command
}

func (c *namespaceCmd) namespace(cmd *cobra.Command...
  ...
  if len(args) == 0 {
    ns, err = c.selectNamespace(c.clientSetCreator, ...)
    if err != nil {
      return err
    }
  } else {
    ns = args[0]
  }
  err = c.setNamespace(c.fs, ns)
  if err != nil {
    return err
  ...
```

namespace.go

```go
selectNamespaceCalled := false
setNamespaceCalled := false
var mockSelectNamespace = func(clientSetCreator, ... {
  selectNamespaceCalled = true
  return "", nil
}
var mockSetNamespace = func(afero.Fs, string) error {
  setNamespaceCalled = true; return nil
}

nscmd := newNamespaceCmd()
nscmd.selectNamespace = mockSelectNamespace
nscmd.setNamespace = mockSetNamespace
```

namespace_test.go

💡 While I think it func injection looks cool, I personally am undecided if it justifies the additional effort

This and many more Real-Life Examples: SimonTheLeg/konf-go@v0.1.1

- namespace.go → full func injection
- set.go → custom type, but no func injection
- cleanup.go → standard var pattern

konf-go@v0.1.1

My Main Take-Aways And Q&A

- 💡 Unless building nested commands, the standard structure should be sufficient
- 💡 All the little tweaks add up, so I rather just copy-paste an old command instead of using `cobra-cli add`
- 💡 I like creating my own type for commands and offloading some cobra logic directly into the command
- 💡 While I think it func injection looks cool, I personally am undecided if it justifies the additional effort