Building Command Line Tools

Insert a tag line here <-

Some ChatGPT taglines for this chapter

- ... letting the tools command you?
- ... just Ctrl-Z and exit the conversation?
- ... you're really just a GUI in a CLI world?
- ... risking a syntax error in your social life?
- ... just pressing 'Enter' on your aspirations?
- ... choosing the path of least resistance—like using a mouse?
- ... mistaking the space bar for a real bar?
- ... admitting you still double-click on hyperlinks?
- ... getting lost in the directories of life?
- ... you're simply not ready for that terminal commitment?

... watching your aspirations get "piped" away?

... maybe you'd prefer a toolbar with a side of icons?

... realizing not all prompts are in a shell?

... sticking to the desktop shortcuts of life?

... you're in search mode, but can't find your purpose?

... echoing your regrets to the console?

... or perhaps you're just one flag away from a complete meltdown?

... wondering why life doesn't come with a man page?

... you're just a user in a sudo world?

... acknowledging that tab completion can't help with life's choices?

A few categories of CLI tools

- Simple app with no flags and just arguments
- Apps with a few flags and some arguments
- Apps with just flags
- Apps lots of flags and arguments
- Apps that act as their own shell
- Apps take streaming I/O or stream I/O
- Apps that want to draw things

Packages that you should know

- “os”
- “flag”
- github.com/spf13/cobra
- github.com/fatih/color
- Advanced Drawing Packages:
  - github.com/gizak/termui
  - github.com/mum4k/termdash

Some simple App examples using “os” and “flag”

ElephantInTheRoom:~/bin$ qotd qotd.askdoak.com none

Keep your face to the sunshine and you cannot see a shadow.

ElephantInTheRoom:~/bin$ qotd —endpoint=qotd.askdoak.com

It is better to keep your mouth closed and let people think you are a fool than to open it and remove all doubt.

And then: sometimes you want some input versatility

```
jdoak@antarctica:~/bin$ reformat file1.json file2.json

jdoak@antarctica:~/bin$ wget "http://some.server.com" | reformat
```

Using os

For simple tooling

What is os.Args

- A slice of every space separated word in the command line
- os.Args[0] is always set to the name of the program
- Quoted arguments are a single entry

When to consider os.Args

- Number of input parameters are small (2 or less)

Or

- The parameters are all the same, like file names to read or write to
- Order of arguments is fixed

Quote of the Day Client

- Connects to an IP:Port with TCP
- If you want a quote from a specific author, send the author's name and a "\n" within 5 milliseconds
- Receive the quote and print to stdout

Using os.Args

```
const usage = `
Usage: qotd <addr> [author]

addr:   The address of the QOTD server. If no port is specified, port 17
        will be used. This must be a valid IP address, not a hostname.
author: The author of the quote to request. If the author is "none", a random
        quote will be returned. If no author is specified, the client will
        read from stdin and write to the connection. You have 5 milliseconds
        to type a quote before the connection is closed, so be quick!
`
```

Some basic client checks

```go
func main() {
    if len(os.Args) == 1 {
        log.Fatal(usage)
    }

    for _, arg := range os.Args[1:] {
        if arg == "-h" || arg == "-help" {
            fmt.Printf(usage)
            os.Exit(0)
        }
    }

    // Parse the address from the command line.
    // Note that with os.Args, the first argument is the program name as
    // is standard with Unix. Note that this is not the case with Cobra.
    addr, err := getAddr(os.Args[1])
    if err != nil {
        log.Fatalf("invalid remote address(%s): %s", os.Args[1], err)
    }
```

Connect and complete our client send

```go
conn, err := net.Dial("tcp", addr.String())
if err != nil {
    log.Fatal(err)
}

switch len(os.Args) {
case 2:
    handleStdin(conn)
default:
    author := strings.Join(os.Args[2:], " ")
    handleAuthor(conn, author)
}
```

Reading from Stdin

```go
func handleStdin(conn net.Conn) {
    defer conn.Close()

    // Read from stdin and write to the connection.
    _, err := io.Copy(conn, os.Stdin)
    if err != nil {
        log.Fatal(err)
    }

    // Read from the connection and write to stdout.
    _, err = io.Copy(os.Stdout, conn)
    if err != nil {
        log.Fatal(err)
    }
    os.Stdout.WriteString("\n")
}
```

The flag package

The most common way to accept arguments

What is the flag package

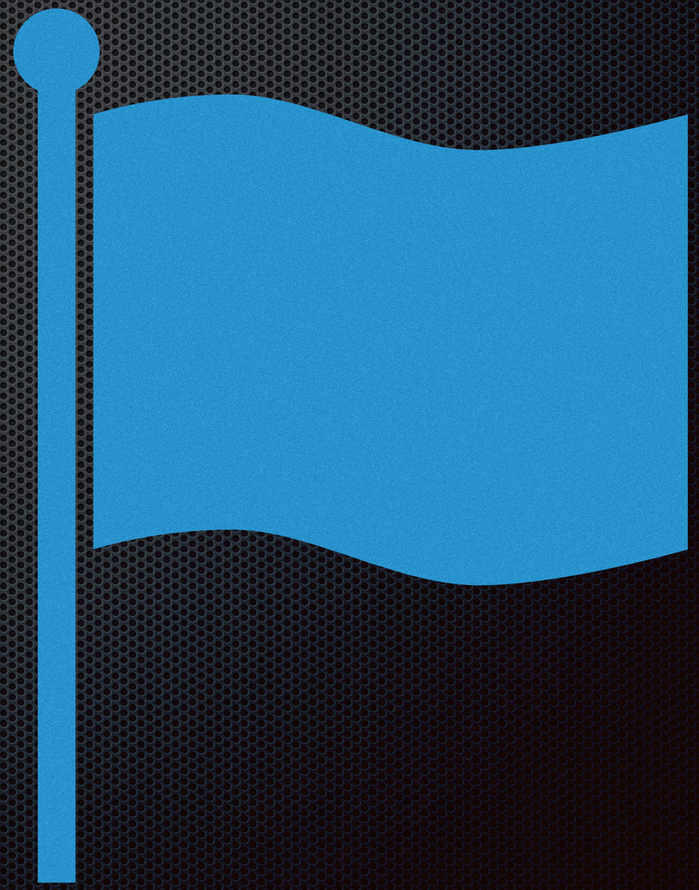
- Reads os.Args
- Looks for defined flags that have a “-“ or “—“ in them
- Assigns the values to variables you define
- Can supply default values
- Handles “-h” and “—help” for you
- Deals with unknown flags for you

Why use the flag package

- The program has optional values
- You want to be explicit in what the user is setting

Flag types

- Bool()
- Int()
- Int64()
- Uint()
- Uint64()
- Float64()
- Duration()
- User Defined Custom Flags



Let’s make another QOTD client, with the flag package

- —endpoint will be the ip:port
  - The default will be the 127.0.0.1:17 address
- —author will specify a specific author
  - If —author is not set, we don’t send one
  - If —author == “stdin”, we read from stdin

Define flags and usage message

```go
var (
    addr   = flag.String("addr", "127.0.0.1:17", "address of the QOTD server, must be ip:port")
    author = flag.String("author", "none", "author of the quote to request")
)

const usage = `
Usage: qotd [flags]
`
```

Define our usage func() and parse our flags

```go
func main() {
    // This provides a custom usage message when a user runs the program
    // with the -h or -help flags. It will also display if the user
    // provides an invalid flag.
    flag.Usage = func() {
        io.WriteString(os.Stderr, usage)
        flag.PrintDefaults()
    }
    flag.Parse()
```

Use the flags

```go
// Parse the address given with our --address flag.
addr, err := getAddr(*address)
if err != nil {
    log.Fatalf("invalid remote address(%s): %s", *address, err)
}

conn, err := net.Dial("tcp", addr.String())
if err != nil {
    log.Fatal(err)
}

switch *author {
case "stdin":
    handleStdin(conn)
default:
    handleAuthor(conn, *author)
}
```

Note: Define custom flag types

flag.Value

```go
type Value interface {
    String() string
    Set(string) error
}
```

```go
// URLValue is a custom flag.Value that parses a URL. It implements the flag.Value interface.
type URLValue struct {
    URL *url.URL
}

// String returns the string representation of the URL.
func (v URLValue) String() string {
    if v.URL != nil {
        return v.URL.String()
    }
    return ""
}

// Set parses the flag text and stores it in the URLValue.
func (v URLValue) Set(s string) error {
    if u, err := url.Parse(s); err != nil {
        return err
    } else {
        *v.URL = *u
    }
    return nil
}

var u = &url.URL{}

func init() {
    flag.Var(&URLValue{u}, "url", "URL to parse")
}
```

Credit: URLValue example from stdlib docs

Running the custom flag

```go
func main() {
    flag.Parse()
    if reflect.ValueOf(*u).IsZero() {
        fmt.Printf("did not pass an URL\n")
        os.Exit(1)
    }
    fmt.Printf("{scheme: %q, host: %q, path: %q}\n", u.Scheme, u.Host, u.Path)
}
```

Runnable example at: https://github.com/johnsiilver/gofordevopsclass/tree/main/custom_flag

Providing ShortHand Flags

```go
var (
    help = new(bool)
)

func init() {
    flag.BoolVar(help, "help", false, "Display help text")
    flag.BoolVar(help, "h", false, "Display help text (shorthand)")
}
```

Both —help and -h are tied to the same boolean value

Be colorful

```go
b := make([]byte, 1024)
n, err := io.ReadFull(conn, b)
if err != nil && err != io.EOF && err != io.ErrUnexpectedEOF {
    log.Fatal(err)
}
color.Blue(string(b[:n]))
```

```
ElephantInTheRoom:both jdoak$ go run client.go helen keller
The true test of a character is to face hard conditions with the determination to make them better
```

Using “os” and “flag”

Best of both worlds

A few notes

- Flags must be used before all os.Args on the command line or it will be set to the default
- When using both, os.Args will contain your flag values as well, you should no longer user os.Args, instead….
- You need to use flag.Args() to retrieve your non-flag values
- flag.Args() DOES NOT have the program path in the first argument!

Exercise: Combine both os.Args and flags

- —endpoint to switch IP:Port of destination
  - Default value of: “127.0.0.1:17”
- —random indicates that we want a random author’s quote
- os.Args to specify a specific author
- If no author specified, used stdin
- If —random is provided, os.Args must not have any arguments
- Use flag to support -h and —help
- Use skeleton Go code to bootstrap this exercise
  - https://github.com/johnsiilver/gofordevopsclass/tree/main/qotd/client/both/skeleton
- Answer at:
  - https://github.com/johnsiilver/gofordevopsclass/tree/main/qotd/client/both

Let’s talk about cobra

For complicated interfaces

What Cobra is good for

- Nested subcommands
- Command suggestions
- Aliases for commands so that you can make changes without breaking users
- Generation of help text from flags and commands
- Generation of autocompletion for various shells
- Man page creation

What’s bad about Cobra

- It’s more complicated, so make sure it is worth it
- You should be using code generation and code generation always introduces more complexity in a process

Code Organization

Suggested Directory Structure

```
appName/
    cmd/
        add.go
        your.go
        commands.go
        here.go
    main.go
```

Package main structure

```go
package main

import (
    "{pathToYourApp}/cmd"
)

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

Using the Cobra generator

- In the root directory, you will need to create a cobra.yaml file:

```
author: [Your Name] youremail@somedomain.com
year: 2023
license: MIT
```

- Download and install the Cobra tooling:
  - go get github.com/spf13/cobra/cobra
  - go install github.com/spf13/cobra/cobra
- Run the Cobra init app in your app's root directory
  - cobra init --pkg-name [repo path]
  - cobra add get

Notes:

- Use camelCase, not snake_case
  - Otherwise you will run into problems
- “Cmd” is appended to every command, so “get” will generate “getCmd”
- If you want to make a sub-command, you would do:
  - cobra add author -p getCmd
    - This adds an “author” command onto the “get” command

Example

cobra.yaml

```yaml
author: John Doak doak@askdoak.com
year: 2023
license: MIT
```

Commands

```
ElephantInTheRoom:cobra jdoak$ cobra init --pkg-name github.com/johnsiilver/gofordevopsclass/qotd/client/cobra
Using config file: /Users/jdoak/.cobra.yaml
Your Cobra application is ready at
/Users/jdoak/trees/gofordevopsclass/qotd/client/cobra
```

```
ElephantInTheRoom:cobra jdoak$ cobra add get
Using config file: /Users/jdoak/.cobra.yaml
get created at /Users/jdoak/trees/gofordevopsclass/qotd/client/cobra
```

Result

```
.
├── LICENSE
├── cmd
│   ├── get.go
│   └── root.go
├── cobra.yaml
└── main.go
```

Setup our “get” command

Boilerplate

```go
package cmd

import (
	"fmt"

	"github.com/spf13/cobra"
)

// getCmd represents the get command
var getCmd = &cobra.Command{
	Use:   "get",
	Short: "A brief description of your command",
	Long: `A longer description that spans multiple lines and likely contains examples
and usage of using your command. For example:

Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.`,
	Run: func(cmd *cobra.Command, args []string) {
		fmt.Println("get called")
	},
}

func init() {
	rootCmd.AddCommand(getCmd)

	// Here you will define your flags and configuration settings.

	// Cobra supports Persistent Flags which will work for this command
	// and all subcommands, e.g.:
	// getCmd.PersistentFlags().String("foo", "", "A help for foo")

	// Cobra supports local flags which will only run when this command
	// is called directly, e.g.:
	// getCmd.Flags().BoolP("toggle", "t", false, "Help message for toggle")
}
```

Handle “get” support

```
var getCmd = &cobra.Command{
    Use:   "get",
    Short: "Get a quote from the QOTD server",
    Long: `Get requests a quote from the QOTD server. If no author is specified,
the client will get a random quote from a random author. For example:

qotd get -a "hellen keller"
or
qotd get
`,
```

Handle “get” support (continued)

```go
Run: func(cmd *cobra.Command, args []string) {
    const devAddr = "127.0.0.1:17"
    fs := cmd.Flags()

    addr := mustString(fs, "addr")
    if mustBool(fs, "dev") {
        addr = devAddr
    }

    addrNetIP, err := getAddr(addr)
    if err != nil {
        log.Fatalf("invalid remote address(%s): %s", addr, err)
    }

    author := mustString(fs, "author")
    if len(author) == 0 {
        author = "none"
    }

    conn, err := net.Dial("tcp", addrNetIP.String())
    if err != nil {
        log.Fatal(err)
    }

    handleAuthor(conn, author)
},
```

Handle “get” support (continued)

```go
func init() {
    rootCmd.AddCommand(getCmd)

    getCmd.Flags().BoolP("dev", "d", false,
        "Uses the dev server instead of prod")
    getCmd.Flags().String("addr", "10.0.0.1:17",
        "Set the QOTD server to use, defaults to production")
    getCmd.Flags().StringP("author", "a", "",
        "Specify the author to get a quote for")
}
```

Handle “get” support (continued)

```go
func mustString(fs *pflag.FlagSet, name string) string {
    v, err := fs.GetString(name)
    if err != nil {
        panic(err)
    }
    return v
}

func mustBool(fs *pflag.FlagSet, name string) bool {
    v, err := fs.GetBool(name)
    if err != nil {
        panic(err)
    }
    return v
}
```

Setup the rootCmd

- With rootCmd, you can make the command run without a sub-command like “get”
- You need to fill out the “short”, “long” description of the app itself, along with any examples
- You can decided if you want to use the “viper” package for config files
  - Viper is a helper for configuration files for services and command line applications. Support for Viper is built into Cobra

Persistent Flags for all sub-commands

- Setup is similar to a regular flag:
  - rootCmd.PersistentFlags().String("addr", "10.0.0.1:17", "Set the QOTD server to use, defaults to production”)
  - This flag is available to all sub-commands from the rootCmd
  - Try removing the “addr” and “dev” flag from the getCmd and moving it to the rootCmd
    - This makes the —add and —dev flag available to all sub-commands, which is what you would want for these

Make flags that must be defined together

```
rootCmd.Flags().StringP("username", "u", "", "Username (required if password is set)")
rootCmd.Flags().StringP("password", "p", "", "Password (required if username is set)")
rootCmd.MarkFlagsRequiredTogether("username", "password")
```

Or that can’t be used together

```
rootCmd.Flags().Bool("json", false, "Output in JSON")
rootCmd.Flags().Bool("yaml", false, "Output in YAML")
rootCmd.MarkFlagsMutuallyExclusive("json", "yaml")
```

And so much more…

- Shell completions:
  - github.com/spf13/cobra/blob/main/site/content/completions/_index.md
- Generate man pages:
  - github.com/spf13/cobra/blob/main/site/content/docgen/man.md

Handling Signals

To prevent problems

Types of Signals in Go

- Synchronous
  - Usually due to program errors
  - Go treats these as runtime panics
  - Can be dealt with via defer
- Asynchronous
  - SIGHUP: The connected terminal disconnected
  - SIGTERM: Please quit and do cleanup (generated from a program)
  - SIGINT: The same as SIGTERM (sent from the terminal)
  - SIGQUIT: The same as SIGTERM plus a core dump (sent from the terminal)
  - SIGKILL: The program must quit; this signal cannot be captured

We care about these

Why would you want to catch a signal?

- Maybe the program should complete, regardless if it looses the terminal
- Some type of operation should happen or finish before the program exits
- You want to display some information about what was done, before the exit

Using os/signals

- Catch signals in main()
- Use context.Context to signal cancellation and cleanup

Let’s register some signals to catch

```go
signals := make(chan os.Signal, 1)
signal.Notify(
    signals,
    syscall.SIGHUP,
    syscall.SIGINT,
    syscall.SIGTERM,
    syscall.SIGQUIT,
)
```

Define some synchronization primitives

```
ctx, cancel := context.WithCancel(context.Background())
wg := &sync.WaitGroup{}
exitLock := &sync.Mutex{}
```

Catch some signals

```go
go func() {
    for {
        switch sig := <-signals; sig {
        case syscall.SIGHUP: // Terminal closed
        // Do nothing, let it keep going
        case syscall.SIGINT, syscall.SIGTERM: // User hit Ctrl+C or told to quit via another program
            go func() {
                log.Printf("%s called", sig.String())
                exitLock.Lock()

                cancel()
                wg.Wait()
                os.Exit(1)
            }()
        case syscall.SIGQUIT: // User hit Ctrl+\
            go func() {
                log.Println("SIGQUIT called")
                exitLock.Lock()

                cancel()
                wg.Wait()
                panic("SIGQUIT called") // Gives us a core dump
            }()
        default:
            log.Println(sig)
        }
    }
}()
```

The rest of main()

```go
wg.Add(1)
func() {
    defer wg.Done()
    for i := 0; i < *n; i++ {
        fmt.Println(i)
        select {
        case <-ctx.Done():
            return
        case <-time.After(time.Second):
        }
    }
}()
exitLock.Lock() // If there is a signal pending, wait for it to finish

if ctx.Err() != nil {
    log.Println(ctx.Err())
    os.Exit(1)
}
```

Danger Will Robinson, DANGER!

Case study – Google satellite disk erase

In the early days, Google did not own all the data center space it does today—we were in a lot of rented space with a large number of machines. In some places, however, this was prohibitively expensive. To speed up connectivity in these places, we would rent small spaces that could have cache machines, terminate HTTP connections and backhaul the traffic to a data center. We called these **satellites**.

Google has an automated process for the decommissioning of machines. One part of this is called disk erase, whereby the machines have their disks wiped. The software was written to grab a list of machines for a satellite and filter out other machines. Unfortunately, if you run it twice on a satellite, the filter is not applied, and your list of machines is all machines in every satellite. Disk eraee was very efficient, putting all machines in all satellites in disk erase at once before anything could be done.

For a more detailed breakdown, you can read https://sre.google/workbook/ postmortem-culture/, where several **Site Reliability Engineers (SREs)** have provided more detail in the context of postmortems.

Recommendations to avoid disasters

- Validate arguments before sending
- Don’t have defaults that specify broad parameters
- If the tool performs operations, consider
  - That the client app should talk to a service that does the work
  - The client should provides some checks on what should be valid
  - The service has validation checks for the scope of work (defense in depth)