User Guide

While you are welcome to provide your own organization, typically a Cobra-based application will follow the following organizational structure:

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

In a Cobra app, typically the main.go file is very bare. It serves one purpose: initializing Cobra.

```
package main

import (
  "{pathToYourApp}/cmd"
)

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

Using the Cobra Generator

Cobra-CLI is its own program that will create your application and add any commands you want. It's the easiest way to incorporate Cobra into your application.

For complete details on using the Cobra generator, please refer to The Cobra-CLI Generator README

Using the Cobra Library

To manually implement Cobra you need to create a bare main.go file and a rootCmd file. You will optionally provide additional commands as you see fit.

Create rootCmd

Cobra doesn't require any special constructors. Simply create your commands.

Ideally you place this in app/cmd/root.go:

```
var rootCmd = &cobra.Command{
  Use:   "hugo",
  Short: "Hugo is a very fast static site generator",
  Long: `A Fast and Flexible Static Site Generator built with
                love by spf13 and friends in Go.
                Complete documentation is available at http://hugo.spf13.com`,
  Run: func(cmd *cobra.Command, args []string) {
    // Do Stuff Here
```

```
  },
}

func Execute() {
  if err := rootCmd.Execute(); err != nil {
    fmt.Fprintln(os.Stderr, err)
    os.Exit(1)
  }
}
```

You will additionally define flags and handle configuration in your init() function.

For example cmd/root.go:

```
package cmd

import (
    "fmt"
    "os"

    "github.com/spf13/cobra"
    "github.com/spf13/viper"
)

var (
    // Used for flags.
    cfgFile     string
    userLicense string

    rootCmd = &cobra.Command{
        Use:   "cobra-cli",
        Short: "A generator for Cobra based Applications",
        Long: `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.`,
    }
)

// Execute executes the root command.
func Execute() error {
    return rootCmd.Execute()
}

func init() {
    cobra.OnInitialize(initConfig)

    rootCmd.PersistentFlags().StringVar(&cfgFile, "config", "", "config file
(default is $HOME/.cobra.yaml)")
    rootCmd.PersistentFlags().StringP("author", "a", "YOUR NAME", "author name for
copyright attribution")
    rootCmd.PersistentFlags().StringVarP(&userLicense, "license", "l", "", "name of
license for the project")
```

```
    rootCmd.PersistentFlags().Bool("viper", true, "use Viper for configuration")
    viper.BindPFlag("author", rootCmd.PersistentFlags().Lookup("author"))
    viper.BindPFlag("useViper", rootCmd.PersistentFlags().Lookup("viper"))
    viper.SetDefault("author", "NAME HERE <EMAIL ADDRESS>")
    viper.SetDefault("license", "apache")

    rootCmd.AddCommand(addCmd)
    rootCmd.AddCommand(initCmd)
}

func initConfig() {
    if cfgFile != "" {
        // Use config file from the flag.
        viper.SetConfigFile(cfgFile)
    } else {
        // Find home directory.
        home, err := os.UserHomeDir()
        cobra.CheckErr(err)

        // Search config in home directory with name ".cobra" (without extension).
        viper.AddConfigPath(home)
        viper.SetConfigType("yaml")
        viper.SetConfigName(".cobra")
    }

    viper.AutomaticEnv()

    if err := viper.ReadInConfig(); err == nil {
        fmt.Println("Using config file:", viper.ConfigFileUsed())
    }
}
```

Create your main.go

With the root command you need to have your main function execute it. Execute should be run on the root for clarity, though it can be called on any command.

In a Cobra app, typically the main.go file is very bare. It serves one purpose: to initialize Cobra.

```
package main

import (
  "{pathToYourApp}/cmd"
)

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

Create additional commands

Additional commands can be defined and typically are each given their own file inside of the cmd/ directory.

If you wanted to create a version command you would create cmd/version.go and populate it with the following:

```
package cmd

import (
  "fmt"

  "github.com/spf13/cobra"
)

func init() {
  rootCmd.AddCommand(versionCmd)
}

var versionCmd = &cobra.Command{
  Use:   "version",
  Short: "Print the version number of Hugo",
  Long:  `All software has versions. This is Hugo's`,
  Run: func(cmd *cobra.Command, args []string) {
    fmt.Println("Hugo Static Site Generator v0.9 -- HEAD")
  },
}
```

Returning and handling errors

If you wish to return an error to the caller of a command, `RunE` can be used.

```
package cmd

import (
  "fmt"

  "github.com/spf13/cobra"
)

func init() {
  rootCmd.AddCommand(tryCmd)
}

var tryCmd = &cobra.Command{
  Use:   "try",
  Short: "Try and possibly fail at something",
  RunE: func(cmd *cobra.Command, args []string) error {
    if err := someFunc(); err != nil {
    return err
    }
    return nil
  },
}
```

The error can then be caught at the execute function call.

Working with Flags

Flags provide modifiers to control how the action command operates.

Assign flags to a command

Since the flags are defined and used in different locations, we need to define a variable outside with the correct scope to assign the flag to work with.

```
var Verbose bool
var Source string
```

There are two different approaches to assign a flag.

Persistent Flags

A flag can be 'persistent', meaning that this flag will be available to the command it's assigned to as well as every command under that command. For global flags, assign a flag as a persistent flag on the root.

```
rootCmd.PersistentFlags().BoolVarP(&Verbose, "verbose", "v", false, "verbose
output")
```

Local Flags

A flag can also be assigned locally, which will only apply to that specific command.

```
localCmd.Flags().StringVarP(&Source, "source", "s", "", "Source directory to read
from")
```

Local Flag on Parent Commands

By default, Cobra only parses local flags on the target command, and any local flags on parent commands are ignored. By enabling `Command.TraverseChildren`, Cobra will parse local flags on each command before executing the target command.

```
command := cobra.Command{
  Use: "print [OPTIONS] [COMMANDS]",
  TraverseChildren: true,
}
```

Bind Flags with Config

You can also bind your flags with viper:

```
var author string

func init() {
  rootCmd.PersistentFlags().StringVar(&author, "author", "YOUR NAME", "Author name
```

```
for copyright attribution")
  viper.BindPFlag("author", rootCmd.PersistentFlags().Lookup("author"))
}
```

In this example, the persistent flag `author` is bound with `viper`. **Note**: the variable `author` will not be set to the value from config, when the `--author` flag is provided by user.

More in viper documentation.

Required flags

Flags are optional by default. If instead you wish your command to report an error when a flag has not been set, mark it as required:

```
rootCmd.Flags().StringVarP(&Region, "region", "r", "", "AWS region (required)")
rootCmd.MarkFlagRequired("region")
```

Or, for persistent flags:

```
rootCmd.PersistentFlags().StringVarP(&Region, "region", "r", "", "AWS region
(required)")
rootCmd.MarkPersistentFlagRequired("region")
```

Positional and Custom Arguments

Validation of positional arguments can be specified using the `Args` field of `Command`.

The following validators are built in:

- `NoArgs` - the command will report an error if there are any positional args.
- `ArbitraryArgs` - the command will accept any args.
- `OnlyValidArgs` - the command will report an error if there are any positional args that are not in the `ValidArgs` field of `Command`.
- `MinimumNArgs(int)` - the command will report an error if there are not at least N positional args.
- `MaximumNArgs(int)` - the command will report an error if there are more than N positional args.
- `ExactArgs(int)` - the command will report an error if there are not exactly N positional args.
- `ExactValidArgs(int)` - the command will report an error if there are not exactly N positional args OR if there are any positional args that are not in the `ValidArgs` field of `Command`
- `RangeArgs(min, max)` - the command will report an error if the number of args is not between the minimum and maximum number of expected args.
- `MatchAll(pargs ...PositionalArgs)` - enables combining existing checks with arbitrary other checks (e.g. you want to check the ExactArgs length along with other qualities).

An example of setting the custom validator:

```
var cmd = &cobra.Command{
  Short: "hello",
  Args: func(cmd *cobra.Command, args []string) error {
    if len(args) < 1 {
      return errors.New("requires a color argument")
```

```
    }
    if myapp.IsValidColor(args[0]) {
      return nil
    }
    return fmt.Errorf("invalid color specified: %s", args[0])
  },
  Run: func(cmd *cobra.Command, args []string) {
    fmt.Println("Hello, World!")
  },
}
```

Example

In the example below, we have defined three commands. Two are at the top level and one (cmdTimes) is a child of one of the top commands. In this case the root is not executable, meaning that a subcommand is required. This is accomplished by not providing a 'Run' for the 'rootCmd'.

We have only defined one flag for a single command.

More documentation about flags is available at https://github.com/spf13/pflag

```
package main

import (
  "fmt"
  "strings"

  "github.com/spf13/cobra"
)

func main() {
  var echoTimes int

  var cmdPrint = &cobra.Command{
    Use:   "print [string to print]",
    Short: "Print anything to the screen",
    Long: `print is for printing anything back to the screen.
For many years people have printed back to the screen.`,
    Args: cobra.MinimumNArgs(1),
    Run: func(cmd *cobra.Command, args []string) {
      fmt.Println("Print: " + strings.Join(args, " "))
    },
  }

  var cmdEcho = &cobra.Command{
    Use:   "echo [string to echo]",
    Short: "Echo anything to the screen",
    Long: `echo is for echoing anything back.
Echo works a lot like print, except it has a child command.`,
    Args: cobra.MinimumNArgs(1),
    Run: func(cmd *cobra.Command, args []string) {
```

```
      fmt.Println("Echo: " + strings.Join(args, " "))
    },
  }

  var cmdTimes = &cobra.Command{
    Use:   "times [string to echo]",
    Short: "Echo anything to the screen more times",
    Long: `echo things multiple times back to the user by providing
a count and a string.`,
    Args: cobra.MinimumNArgs(1),
    Run: func(cmd *cobra.Command, args []string) {
      for i := 0; i < echoTimes; i++ {
        fmt.Println("Echo: " + strings.Join(args, " "))
      }
    },
  }

  cmdTimes.Flags().IntVarP(&echoTimes, "times", "t", 1, "times to echo the input")

  var rootCmd = &cobra.Command{Use: "app"}
  rootCmd.AddCommand(cmdPrint, cmdEcho)
  cmdEcho.AddCommand(cmdTimes)
  rootCmd.Execute()
}
```

For a more complete example of a larger application, please checkout Hugo.

Help Command

Cobra automatically adds a help command to your application when you have subcommands. This will be called when a user runs 'app help'. Additionally, help will also support all other commands as input. Say, for instance, you have a command called 'create' without any additional configuration; Cobra will work when 'app help create' is called. Every command will automatically have the '--help' flag added.

Example

The following output is automatically generated by Cobra. Nothing beyond the command and flag definitions are needed.

```
$ cobra help

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.

Usage:
  cobra [command]

Available Commands:
  add         Add a command to a Cobra Application
  help        Help about any command
  init        Initialize a Cobra Application
```

```
Flags:
  -a, --author string    author name for copyright attribution (default "YOUR NAME")
      --config string    config file (default is $HOME/.cobra.yaml)
  -h, --help             help for cobra
  -l, --license string   name of license for the project
      --viper            use Viper for configuration (default true)

Use "cobra [command] --help" for more information about a command.
```

Help is just a command like any other. There is no special logic or behavior around it. In fact, you can provide your own if you want.

Defining your own help

You can provide your own Help command or your own template for the default command to use with following functions:

```
cmd.SetHelpCommand(cmd *Command)
cmd.SetHelpFunc(f func(*Command, []string))
cmd.SetHelpTemplate(s string)
```

The latter two will also apply to any children commands.

Usage Message

When the user provides an invalid flag or invalid command, Cobra responds by showing the user the 'usage'.

Example

You may recognize this from the help above. That's because the default help embeds the usage as part of its output.

```
$ cobra --invalid
Error: unknown flag: --invalid
Usage:
  cobra [command]

Available Commands:
  add         Add a command to a Cobra Application
  help        Help about any command
  init        Initialize a Cobra Application

Flags:
  -a, --author string    author name for copyright attribution (default "YOUR NAME")
      --config string    config file (default is $HOME/.cobra.yaml)
  -h, --help             help for cobra
  -l, --license string   name of license for the project
      --viper            use Viper for configuration (default true)

Use "cobra [command] --help" for more information about a command.
```

Defining your own usage

You can provide your own usage function or template for Cobra to use. Like help, the function and template are overridable through public methods:

```
cmd.SetUsageFunc(f func(*Command) error)
cmd.SetUsageTemplate(s string)
```

Version Flag

Cobra adds a top-level '--version' flag if the Version field is set on the root command. Running an application with the '--version' flag will print the version to stdout using the version template. The template can be customized using the `cmd.SetVersionTemplate(s string)` function.

PreRun and PostRun Hooks

It is possible to run functions before or after the main `Run` function of your command. The `PersistentPreRun` and `PreRun` functions will be executed before `Run`. `PersistentPostRun` and `PostRun` will be executed after `Run`. The `Persistent*Run` functions will be inherited by children if they do not declare their own. These functions are run in the following order:

- `PersistentPreRun`
- `PreRun`
- `Run`
- `PostRun`
- `PersistentPostRun`

An example of two commands which use all of these features is below. When the subcommand is executed, it will run the root command's `PersistentPreRun` but not the root command's `PersistentPostRun`:

```
package main

import (
  "fmt"

  "github.com/spf13/cobra"
)

func main() {

  var rootCmd = &cobra.Command{
    Use:   "root [sub]",
    Short: "My root command",
    PersistentPreRun: func(cmd *cobra.Command, args []string) {
      fmt.Printf("Inside rootCmd PersistentPreRun with args: %v\n", args)
    },
    PreRun: func(cmd *cobra.Command, args []string) {
      fmt.Printf("Inside rootCmd PreRun with args: %v\n", args)
    },
    Run: func(cmd *cobra.Command, args []string) {
```

```
      fmt.Printf("Inside rootCmd Run with args: %v\n", args)
    },
    PostRun: func(cmd *cobra.Command, args []string) {
      fmt.Printf("Inside rootCmd PostRun with args: %v\n", args)
    },
    PersistentPostRun: func(cmd *cobra.Command, args []string) {
      fmt.Printf("Inside rootCmd PersistentPostRun with args: %v\n", args)
    },
  }

  var subCmd = &cobra.Command{
    Use:   "sub [no options!]",
    Short: "My subcommand",
    PreRun: func(cmd *cobra.Command, args []string) {
      fmt.Printf("Inside subCmd PreRun with args: %v\n", args)
    },
    Run: func(cmd *cobra.Command, args []string) {
      fmt.Printf("Inside subCmd Run with args: %v\n", args)
    },
    PostRun: func(cmd *cobra.Command, args []string) {
      fmt.Printf("Inside subCmd PostRun with args: %v\n", args)
    },
    PersistentPostRun: func(cmd *cobra.Command, args []string) {
      fmt.Printf("Inside subCmd PersistentPostRun with args: %v\n", args)
    },
  }

  rootCmd.AddCommand(subCmd)

  rootCmd.SetArgs([]string{""})
  rootCmd.Execute()
  fmt.Println()
  rootCmd.SetArgs([]string{"sub", "arg1", "arg2"})
  rootCmd.Execute()
}
```

Output:

```
Inside rootCmd PersistentPreRun with args: []
Inside rootCmd PreRun with args: []
Inside rootCmd Run with args: []
Inside rootCmd PostRun with args: []
Inside rootCmd PersistentPostRun with args: []

Inside rootCmd PersistentPreRun with args: [arg1 arg2]
Inside subCmd PreRun with args: [arg1 arg2]
Inside subCmd Run with args: [arg1 arg2]
Inside subCmd PostRun with args: [arg1 arg2]
Inside subCmd PersistentPostRun with args: [arg1 arg2]
```

Suggestions when "unknown command" happens

Cobra will print automatic suggestions when "unknown command" errors happen. This allows Cobra to behave similarly to the `git` command when a typo happens. For example:

```
$ hugo srever
Error: unknown command "srever" for "hugo"

Did you mean this?
        server

Run 'hugo --help' for usage.
```

Suggestions are automatic based on every subcommand registered and use an implementation of Levenshtein distance. Every registered command that matches a minimum distance of 2 (ignoring case) will be displayed as a suggestion.

If you need to disable suggestions or tweak the string distance in your command, use:

```
command.DisableSuggestions = true
```

or

```
command.SuggestionsMinimumDistance = 1
```

You can also explicitly set names for which a given command will be suggested using the `SuggestFor` attribute. This allows suggestions for strings that are not close in terms of string distance, but makes sense in your set of commands and for some which you don't want aliases. Example:

```
$ kubectl remove
Error: unknown command "remove" for "kubectl"

Did you mean this?
        delete

Run 'kubectl help' for usage.
```

Generating documentation for your command

Cobra can generate documentation based on subcommands, flags, etc. Read more about it in the docs generation documentation.

Generating shell completions

Cobra can generate a shell-completion file for the following shells: bash, zsh, fish, PowerShell. If you add more information to your commands, these completions can be amazingly powerful and flexible. Read more about it in Shell Completions.