Generating shell completions

Cobra can generate shell completions for multiple shells. The currently supported shells are: - Bash - Zsh - fish - PowerShell

Cobra will automatically provide your program with a fully functional `completion` command, similarly to how it provides the `help` command.

Creating your own completion command

If you do not wish to use the default `completion` command, you can choose to provide your own, which will take precedence over the default one. (This also provides backwards-compatibility with programs that already have their own `completion` command.)

If you are using the `cobra-cli` generator, which can be found at spf13/cobra-cli, you can create a completion command by running

```
cobra-cli add completion
```

and then modifying the generated `cmd/completion.go` file to look something like this (writing the shell script to stdout allows the most flexible use):

```
var completionCmd = &cobra.Command{
    Use:   "completion [bash|zsh|fish|powershell]",
    Short: "Generate completion script",
    Long: fmt.Sprintf(`To load completions:

Bash:

  $ source <(%[1]s completion bash)

  # To load completions for each session, execute once:
  # Linux:
  $ %[1]s completion bash > /etc/bash_completion.d/%[1]s
  # macOS:
  $ %[1]s completion bash > /usr/local/etc/bash_completion.d/%[1]s

Zsh:

  # If shell completion is not already enabled in your environment,
  # you will need to enable it.  You can execute the following once:

  $ echo "autoload -U compinit; compinit" >> ~/.zshrc

  # To load completions for each session, execute once:
  $ %[1]s completion zsh > "${fpath[1]}/_%[1]s"
```

```
  # You will need to start a new shell for this setup to take effect.

fish:

  $ %[1]s completion fish | source

  # To load completions for each session, execute once:
  $ %[1]s completion fish > ~/.config/fish/completions/%[1]s.fish

PowerShell:

  PS> %[1]s completion powershell | Out-String | Invoke-Expression

  # To load completions for every new session, run:
  PS> %[1]s completion powershell > %[1]s.ps1
  # and source this file from your PowerShell profile.
`,cmd.Root().Name()),
    DisableFlagsInUseLine: true,
    ValidArgs:             []string{"bash", "zsh", "fish", "powershell"},
    Args:                  cobra.ExactValidArgs(1),
    Run: func(cmd *cobra.Command, args []string) {
        switch args[0] {
        case "bash":
            cmd.Root().GenBashCompletion(os.Stdout)
        case "zsh":
            cmd.Root().GenZshCompletion(os.Stdout)
        case "fish":
            cmd.Root().GenFishCompletion(os.Stdout, true)
        case "powershell":
            cmd.Root().GenPowerShellCompletionWithDesc(os.Stdout)
        }
    },
}
```

Note: The cobra generator may include messages printed to stdout, for example, if the config file is loaded; this will break the auto-completion script so must be removed.

Adapting the default completion command

Cobra provides a few options for the default `completion` command. To configure such options you must set the `CompletionOptions` field on the *root* command.

To tell Cobra *not* to provide the default `completion` command:

```
rootCmd.CompletionOptions.DisableDefaultCmd = true
```

To tell Cobra *not* to provide the user with the `--no-descriptions` flag to the

completion sub-commands:

`rootCmd.CompletionOptions.DisableNoDescFlag = true`

To tell Cobra to completely disable descriptions for completions:

`rootCmd.CompletionOptions.DisableDescriptions = true`

Customizing completions

The generated completion scripts will automatically handle completing commands and flags. However, you can make your completions much more powerful by providing information to complete your program's nouns and flag values.

Completion of nouns

Static completion of nouns

Cobra allows you to provide a pre-defined list of completion choices for your nouns using the `ValidArgs` field. For example, if you want `kubectl get [tab][tab]` to show a list of valid "nouns" you have to set them. Some simplified code from `kubectl get` looks like:

```
validArgs []string = { "pod", "node", "service", "replicationcontroller" }

cmd := &cobra.Command{
    Use:     "get [(-o|--output=)json|yaml|template|...] (RESOURCE [NAME] | RESOURCE/NAME ..
    Short:   "Display one or many resources",
    Long:    get_long,
    Example: get_example,
    Run: func(cmd *cobra.Command, args []string) {
        cobra.CheckErr(RunGet(f, out, cmd, args))
    },
    ValidArgs: validArgs,
}
```

Notice we put the `ValidArgs` field on the `get` sub-command. Doing so will give results like:

```
$ kubectl get [tab][tab]
node   pod   replicationcontroller   service
```

Aliases for nouns If your nouns have aliases, you can define them alongside `ValidArgs` using `ArgAliases`:

```
argAliases []string = { "pods", "nodes", "services", "svc", "replicationcontrollers", "rc" }

cmd := &cobra.Command{
    ...
```

```
    ValidArgs:  validArgs,
    ArgAliases: argAliases
}
```

The aliases are not shown to the user on tab completion, but they are accepted as valid nouns by the completion algorithm if entered manually, e.g. in:

```
$ kubectl get rc [tab][tab]
backend        frontend       database
```

Note that without declaring `rc` as an alias, the completion algorithm would not know to show the list of replication controllers following `rc`.

Dynamic completion of nouns

In some cases it is not possible to provide a list of completions in advance. Instead, the list of completions must be determined at execution-time. In a similar fashion as for static completions, you can use the `ValidArgsFunction` field to provide a Go function that Cobra will execute when it needs the list of completion choices for the nouns of a command. Note that either `ValidArgs` or `ValidArgsFunction` can be used for a single cobra command, but not both. Simplified code from `helm status` looks like:

```
cmd := &cobra.Command{
    Use:   "status RELEASE_NAME",
    Short: "Display the status of the named release",
    Long:  status_long,
    RunE: func(cmd *cobra.Command, args []string) {
        RunGet(args[0])
    },
    ValidArgsFunction: func(cmd *cobra.Command, args []string, toComplete string) ([]string,
        if len(args) != 0 {
            return nil, cobra.ShellCompDirectiveNoFileComp
        }
        return getReleasesFromCluster(toComplete), cobra.ShellCompDirectiveNoFileComp
    },
}
```

Where `getReleasesFromCluster()` is a Go function that obtains the list of current Helm releases running on the Kubernetes cluster. Notice we put the `ValidArgsFunction` on the `status` sub-command. Let's assume the Helm releases on the cluster are: `harbor`, `notary`, `rook` and `thanos` then this dynamic completion will give results like:

```
$ helm status [tab][tab]
harbor notary rook thanos
```

You may have noticed the use of `cobra.ShellCompDirective`. These directives are bit fields allowing to control some shell completion behaviors for your

particular completion. You can combine them with the bit-or operator such as `cobra.ShellCompDirectiveNoSpace | cobra.ShellCompDirectiveNoFileComp`

```
// Indicates that the shell will perform its default behavior after completions
// have been provided (this implies none of the other directives).
ShellCompDirectiveDefault

// Indicates an error occurred and completions should be ignored.
ShellCompDirectiveError

// Indicates that the shell should not add a space after the completion,
// even if there is a single completion provided.
ShellCompDirectiveNoSpace

// Indicates that the shell should not provide file completion even when
// no completion is provided.
ShellCompDirectiveNoFileComp

// Indicates that the returned completions should be used as file extension filters.
// For example, to complete only files of the form *.json or *.yaml:
//    return []string{"yaml", "json"}, ShellCompDirectiveFilterFileExt
// For flags, using MarkFlagFilename() and MarkPersistentFlagFilename()
// is a shortcut to using this directive explicitly.
//
ShellCompDirectiveFilterFileExt

// Indicates that only directory names should be provided in file completion.
// For example:
//    return nil, ShellCompDirectiveFilterDirs
// For flags, using MarkFlagDirname() is a shortcut to using this directive explicitly.
//
// To request directory names within another directory, the returned completions
// should specify a single directory name within which to search. For example,
// to complete directories within "themes/":
//    return []string{"themes"}, ShellCompDirectiveFilterDirs
//
ShellCompDirectiveFilterDirs
```

Note: When using the `ValidArgsFunction`, Cobra will call your registered function after having parsed all flags and arguments provided in the command-line. You therefore don't need to do this parsing yourself. For example, when a user calls `helm status --namespace my-rook-ns [tab][tab]`, Cobra will call your registered `ValidArgsFunction` after having parsed the `--namespace` flag, as it would have done when calling the `RunE` function.

Debugging Cobra achieves dynamic completion through the use of a hidden command called by the completion script. To debug your Go completion code, you can call this hidden command directly:

```
$ helm __complete status har<ENTER>
harbor
:4
Completion ended with directive: ShellCompDirectiveNoFileComp # This is on stderr
```

Important: If the noun to complete is empty (when the user has not yet typed any letters of that noun), you must pass an empty parameter to the `__complete` command:

```
$ helm __complete status ""<ENTER>
harbor
notary
rook
thanos
:4
Completion ended with directive: ShellCompDirectiveNoFileComp # This is on stderr
```

Calling the `__complete` command directly allows you to run the Go debugger to troubleshoot your code. You can also add printouts to your code; Cobra provides the following functions to use for printouts in Go completion code:

```
// Prints to the completion script debug file (if BASH_COMP_DEBUG_FILE
// is set to a file path) and optionally prints to stderr.
cobra.CompDebug(msg string, printToStdErr bool) {
cobra.CompDebugln(msg string, printToStdErr bool)

// Prints to the completion script debug file (if BASH_COMP_DEBUG_FILE
// is set to a file path) and to stderr.
cobra.CompError(msg string)
cobra.CompErrorln(msg string)
```

Important: You should **not** leave traces that print directly to stdout in your completion code as they will be interpreted as completion choices by the completion script. Instead, use the cobra-provided debugging traces functions mentioned above.

Completions for flags

Mark flags as required

Most of the time completions will only show sub-commands. But if a flag is required to make a sub-command work, you probably want it to show up when the user types [tab][tab]. You can mark a flag as 'Required' like so:

```
cmd.MarkFlagRequired("pod")
cmd.MarkFlagRequired("container")
```

and you'll get something like

```
$ kubectl exec [tab][tab]
-c            --container=  -p            --pod=
```

Specify dynamic flag completion

As for nouns, Cobra provides a way of defining dynamic completion of flags. To provide a Go function that Cobra will execute when it needs the list of completion choices for a flag, you must register the function using the `command.RegisterFlagCompletionFunc()` function.

```
flagName := "output"
cmd.RegisterFlagCompletionFunc(flagName, func(cmd *cobra.Command, args []string, toComplete
    return []string{"json", "table", "yaml"}, cobra.ShellCompDirectiveDefault
})
```

Notice that calling `RegisterFlagCompletionFunc()` is done through the `command` with which the flag is associated. In our example this dynamic completion will give results like so:

```
$ helm status --output [tab][tab]
json table yaml
```

Debugging You can also easily debug your Go completion code for flags:

```
$ helm __complete status --output ""
json
table
yaml
:4
Completion ended with directive: ShellCompDirectiveNoFileComp # This is on stderr
```

Important: You should **not** leave traces that print to stdout in your completion code as they will be interpreted as completion choices by the completion script. Instead, use the cobra-provided debugging traces functions mentioned further above.

Specify valid filename extensions for flags that take a filename

To limit completions of flag values to file names with certain extensions you can either use the different `MarkFlagFilename()` functions or a combination of `RegisterFlagCompletionFunc()` and `ShellCompDirectiveFilterFileExt`, like so:

```
flagName := "output"
cmd.MarkFlagFilename(flagName, "yaml", "json")
```

or

```
flagName := "output"
cmd.RegisterFlagCompletionFunc(flagName, func(cmd *cobra.Command, args []string, toComplete
    return []string{"yaml", "json"}, ShellCompDirectiveFilterFileExt})
```

Limit flag completions to directory names

To limit completions of flag values to directory names you can either use the `MarkFlagDirname()` functions or a combination of `RegisterFlagCompletionFunc()` and `ShellCompDirectiveFilterDirs`, like so:

```
flagName := "output"
cmd.MarkFlagDirname(flagName)
```

or

```
flagName := "output"
cmd.RegisterFlagCompletionFunc(flagName, func(cmd *cobra.Command, args []string, toComplete
    return nil, cobra.ShellCompDirectiveFilterDirs
})
```

To limit completions of flag values to directory names *within another directory* you can use a combination of `RegisterFlagCompletionFunc()` and `ShellCompDirectiveFilterDirs` like so:

```
flagName := "output"
cmd.RegisterFlagCompletionFunc(flagName, func(cmd *cobra.Command, args []string, toComplete
    return []string{"themes"}, cobra.ShellCompDirectiveFilterDirs
})
```

Descriptions for completions

Cobra provides support for completion descriptions. Such descriptions are supported for each shell (however, for bash, it is only available in the completion V2 version). For commands and flags, Cobra will provide the descriptions automatically, based on usage information. For example, using zsh:

```
$ helm s[tab]
search  -- search for a keyword in charts
show    -- show information of a chart
status  -- displays the status of the named release
```

while using fish:

```
$ helm s[tab]
search  (search for a keyword in charts)  show  (show information of a chart)  status  (disp
```

Cobra allows you to add descriptions to your own completions. Simply add the description text after each completion, following a `\t` separator. This technique applies to completions returned by `ValidArgs`, `ValidArgsFunction` and `RegisterFlagCompletionFunc()`. For example:

```
ValidArgsFunction: func(cmd *cobra.Command, args []string, toComplete string) ([]string, cob
    return []string{"harbor\tAn image registry", "thanos\tLong-term metrics"}, cobra.ShellCo
}}
```

or

```
ValidArgs: []string{"bash\tCompletions for bash", "zsh\tCompletions for zsh"}
```

Bash completions

Dependencies

The bash completion script generated by Cobra requires the `bash_completion` package. You should update the help text of your completion command to show how to install the `bash_completion` package (Kubectl docs)

Aliases

You can also configure `bash` aliases for your program and they will also support completions.

```
alias aliasname=origcommand
complete -o default -F __start_origcommand aliasname

# and now when you run `aliasname` completion will make
# suggestions as it did for `origcommand`.

$ aliasname <tab><tab>
completion     firstcommand   secondcommand
```

Bash legacy dynamic completions

For backward compatibility, Cobra still supports its bash legacy dynamic completion solution. Please refer to Bash Completions for details.

Bash completion V2

Cobra provides two versions for bash completion. The original bash completion (which started it all!) can be used by calling `GenBashCompletion()` or `GenBashCompletionFile()`.

A new V2 bash completion version is also available. This version can be used by calling `GenBashCompletionV2()` or `GenBashCompletionFileV2()`. The V2 version does **not** support the legacy dynamic completion (see Bash Completions) but instead works only with the Go dynamic completion solution described in this document. Unless your program already uses the legacy dynamic completion solution, it is recommended that you use the bash completion V2 solution which provides the following extra features: - Supports completion descriptions (like the other shells) - Small completion script of less than 300 lines (v1 generates

scripts of thousands of lines; `kubectl` for example has a bash v1 completion script of over 13K lines) - Streamlined user experience thanks to a completion behavior aligned with the other shells

`Bash` completion V2 supports descriptions for completions. When calling `GenBashCompletionV2()` or `GenBashCompletionFileV2()` you must provide these functions with a parameter indicating if the completions should be annotated with a description; Cobra will provide the description automatically based on usage information. You can choose to make this option configurable by your users.

```
# With descriptions
$ helm s[tab][tab]
search  (search for a keyword in charts)          status  (display the status of the named
show    (show information of a chart)

# Without descriptions
$ helm s[tab][tab]
search  show  status
```

Note: Cobra's default `completion` command uses bash completion V2. If for some reason you need to use bash completion V1, you will need to implement your own `completion` command. ## Zsh completions

Cobra supports native zsh completion generated from the root `cobra.Command`. The generated completion script should be put somewhere in your `$fpath` and be named `_<yourProgram>`. You will need to start a new shell for the completions to become available.

Zsh supports descriptions for completions. Cobra will provide the description automatically, based on usage information. Cobra provides a way to completely disable such descriptions by using `GenZshCompletionNoDesc()` or `GenZshCompletionFileNoDesc()`. You can choose to make this a configurable option to your users.

```
# With descriptions
$ helm s[tab]
search  -- search for a keyword in charts
show    -- show information of a chart
status  -- displays the status of the named release

# Without descriptions
$ helm s[tab]
search  show  status
```

Note: Because of backward-compatibility requirements, we were forced to have a different API to disable completion descriptions between `zsh` and `fish`.

Limitations

- Custom completions implemented in Bash scripting (legacy) are not supported and will be ignored for `zsh` (including the use of the `BashCompCustom` flag annotation).
  - You should instead use `ValidArgsFunction` and `RegisterFlagCompletionFunc()` which are portable to the different shells (`bash`, `zsh`, `fish`, `powershell`).
- The function `MarkFlagCustom()` is not supported and will be ignored for `zsh`.
  - You should instead use `RegisterFlagCompletionFunc()`.

Zsh completions standardization

Cobra 1.1 standardized its zsh completion support to align it with its other shell completions. Although the API was kept backward-compatible, some small changes in behavior were introduced. Please refer to Zsh Completions for details.

fish completions

Cobra supports native fish completions generated from the root `cobra.Command`. You can use the `command.GenFishCompletion()` or `command.GenFishCompletionFile()` functions. You must provide these functions with a parameter indicating if the completions should be annotated with a description; Cobra will provide the description automatically based on usage information. You can choose to make this option configurable by your users.

```
# With descriptions
$ helm s[tab]
search  (search for a keyword in charts)  show  (show information of a chart)  status  (disp

# Without descriptions
$ helm s[tab]
search  show  status
```

Note: Because of backward-compatibility requirements, we were forced to have a different API to disable completion descriptions between `zsh` and `fish`.

Limitations

- Custom completions implemented in bash scripting (legacy) are not supported and will be ignored for `fish` (including the use of the `BashCompCustom` flag annotation).
  - You should instead use `ValidArgsFunction` and `RegisterFlagCompletionFunc()` which are portable to the different shells (`bash`, `zsh`, `fish`, `powershell`).
- The function `MarkFlagCustom()` is not supported and will be ignored for `fish`.

  - You should instead use `RegisterFlagCompletionFunc()`.
- The following flag completion annotations are not supported and will be ignored for `fish`:
  - `BashCompFilenameExt` (filtering by file extension)
  - `BashCompSubdirsInDir` (filtering by directory)
- The functions corresponding to the above annotations are consequently not supported and will be ignored for `fish`:
  - `MarkFlagFilename()` and `MarkPersistentFlagFilename()` (filtering by file extension)
  - `MarkFlagDirname()` and `MarkPersistentFlagDirname()` (filtering by directory)
- Similarly, the following completion directives are not supported and will be ignored for `fish`:
  - `ShellCompDirectiveFilterFileExt` (filtering by file extension)
  - `ShellCompDirectiveFilterDirs` (filtering by directory)

PowerShell completions

Cobra supports native PowerShell completions generated from the root `cobra.Command`. You can use the `command.GenPowerShellCompletion()` or `command.GenPowerShellCompletionFile()` functions. To include descriptions use `command.GenPowerShellCompletionWithDesc()` and `command.GenPowerShellCompletionFileWithDesc()`. Cobra will provide the description automatically based on usage information. You can choose to make this option configurable by your users.

The script is designed to support all three PowerShell completion modes:

- TabCompleteNext (default windows style - on each key press the next option is displayed)
- Complete (works like bash)
- MenuComplete (works like zsh)

You set the mode with `Set-PSReadLineKeyHandler -Key Tab -Function <mode>`. Descriptions are only displayed when using the `Complete` or `MenuComplete` mode.

Users need PowerShell version 5.0 or above, which comes with Windows 10 and can be downloaded separately for Windows 7 or 8.1. They can then write the completions to a file and source this file from their PowerShell profile, which is referenced by the `$Profile` environment variable. See `Get-Help about_Profiles` for more info about PowerShell profiles.

```
# With descriptions and Mode 'Complete'
$ helm s[tab]
search  (search for a keyword in charts)  show  (show information of a chart)  status  (disp

# With descriptions and Mode 'MenuComplete' The description of the current selected value wi
```

```
$ helm s[tab]
search    show     status

search for a keyword in charts

# Without descriptions
$ helm s[tab]
search  show  status
```

Limitations

- Custom completions implemented in bash scripting (legacy) are not supported and will be ignored for `powershell` (including the use of the `BashCompCustom` flag annotation).
  - You should instead use `ValidArgsFunction` and `RegisterFlagCompletionFunc()` which are portable to the different shells (`bash`, `zsh`, `fish`, `powershell`).
- The function `MarkFlagCustom()` is not supported and will be ignored for `powershell`.
  - You should instead use `RegisterFlagCompletionFunc()`.
- The following flag completion annotations are not supported and will be ignored for `powershell`:
  - `BashCompFilenameExt` (filtering by file extension)
  - `BashCompSubdirsInDir` (filtering by directory)
- The functions corresponding to the above annotations are consequently not supported and will be ignored for `powershell`:
  - `MarkFlagFilename()` and `MarkPersistentFlagFilename()` (filtering by file extension)
  - `MarkFlagDirname()` and `MarkPersistentFlagDirname()` (filtering by directory)
- Similarly, the following completion directives are not supported and will be ignored for `powershell`:
  - `ShellCompDirectiveFilterFileExt` (filtering by file extension)
  - `ShellCompDirectiveFilterDirs` (filtering by directory)