How kubectl Plugins Work

Learn how kubectl plugins work in Kubernetes.

We'll cover the following

Overview

The `kubectl` is a powerful command line tool that allows us to manipulate Kubernetes with ease. We could write scripts to invoke a set of `kubectl` commands to do fancy tasks, such as querying objects, maintaining nodes, and more. More complex operations can be implemented by utilizing these `kubectl` commands as essential building blocks. Our work becomes much simpler if we can plumb our implementations as a `kubectl` plugin.

How plugins are supported

Let's first take a look at how these `kubectl` plugins are supported in Kubernetes. This will help us better understand how `kubectl` plugins work and how we can plumb our plugin implementations. Below is the code snippet that shows a entry function `NewDefaultKubectlCommand` (**line 3**):

```
1  // Read more from k8s.io/kubectl/pkg/cmd/cmd.go
2  // NewDefaultKubectlCommand creates the `kubectl` command with default arguments
3  func NewDefaultKubectlCommand() *cobra.Command {
4      return NewDefaultKubectlCommandWithArgs(KubectlOptions{
5          PluginHandler: NewDefaultPluginHandler(plugin.ValidPluginFilenamePrefixes),
6          Arguments:     os.Args,
7          ConfigFlags:   defaultConfigFlags,
8          IOStreams:     genericclioptions.IOStreams{In: os.Stdin, Out: os.Stdout, ErrOut: os.Stderr},
9      })
10 }
11
12 // NewDefaultKubectlCommandWithArgs creates the `kubectl` command with arguments
13 func NewDefaultKubectlCommandWithArgs(o KubectlOptions) *cobra.Command {
14     cmd := NewKubectlCommand(o)
15
16     if o.PluginHandler == nil {
17         return cmd
18     }
19
20     if len(o.Arguments) > 1 {
21         cmdPathPieces := o.Arguments[1:]
22
23         // only look for suitable extension executables if
```

```
24          // the specified command does not already exist
25          if _, _, err := cmd.Find(cmdPathPieces); err != nil {
26              // Also check the commands that will be added by Cobra.
27              // These commands are only added once rootCmd.Execute() is called, so we
28              // need to check them explicitly here.
29              var cmdName string // first "non-flag" arguments
30              for _, arg := range cmdPathPieces {
```

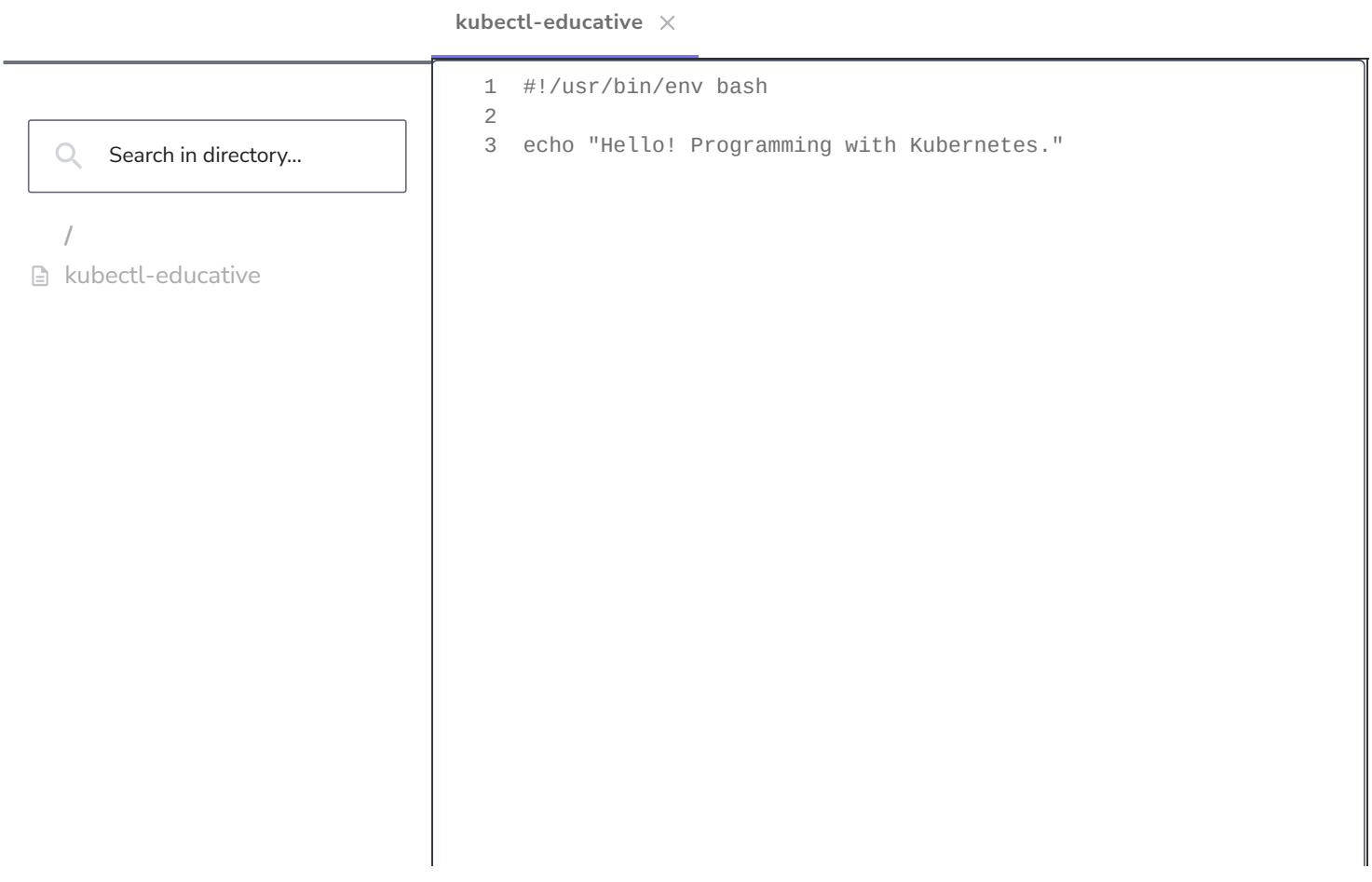
How kubectl plugins are hooked

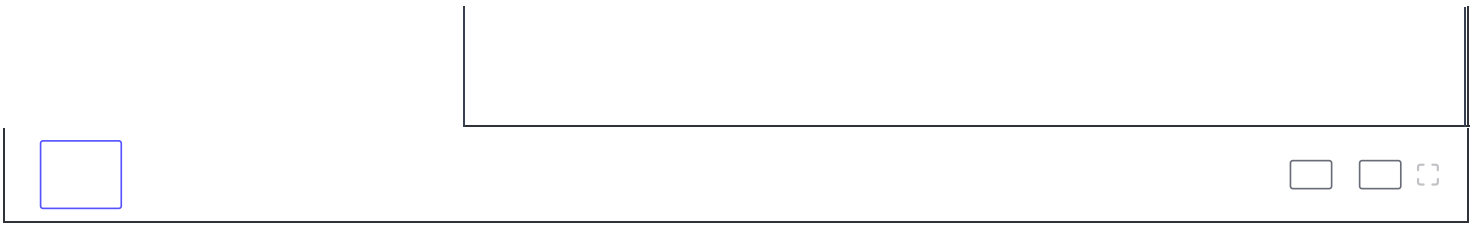
In the code above, we can see that a `PluginHandler` (**line 5**) is specified during initialization. When the command `kubectl` is invoked with a sub-command, the length `o.Arguments` (**line 20**) will be longer than `1`. Then, we need to search a matching command function for that sub-command. If there is no such matching command function, we call the `HandlePluginCommand` (**line 41**) to invoke our plugins.

From the execution flow, we can see that it's not possible to create plugins to overwrite existing sub-commands in `kubectl`. Those built-in sub-commands will always take precedence over third-party plugins. For example, if we try to implement our own plugin to overwrite the default `kubectl version` behavior, our plugin will never be executed. This is also not possible for other child commands. For example, we can't add our plugin implementation of `kubectl create deployment` to the existing `kubectl` commands.

How plugins are discovered

So, how does `kubectl` discover our plugins?





Our Kubernetes environment

Let's run `kubectl plugin list` in the terminal above to see what happens. Since we've not installed any plugins in our environment, the output will be as follows:

```
error: unable to find any kubectl plugins in your PATH
```

The output of listing plugins

From the output, we can see `kubectl` searches for plugins in `PATH`. Actually, any executable files beginning with `kubectl-` in our `PATH` are discoverable as plugins. Let's take a test. We've got a file called `kubectl-educative` in the folder `/usercode`. Let's copy it to a valid directory (such as `/usr/local/bin`) included in the `PATH`. Run the following commands in the terminal above:

```
cp /usercode/kubectl-educative /usr/local/bin/kubectl-educative
chmod +x /usr/local/bin/kubectl-educative
```

Move our kubectl plugin

Now, let's run `kubectl plugin list` in the terminal above to see what happens. The output will be as follows:

```
The following compatible plugins are available:

/usr/local/bin/kubectl-educative
```

Our plugin is discovered

Our plugin is successfully discovered. Awesome. So, how are these plugins executed then?

How plugins are executed

Just as the code snippet below shows, discovered plugins will be executed as binaries by `syscall`. The environment variables and arguments will be passed down as well.

```
// Learn more from k8s.io/kubectl/pkg/cmd/cmd.go
// Execute implements PluginHandler
func (h *DefaultPluginHandler) Execute(executablePath string, cmdArgs, environment []string) error {

    // Windows does not support exec syscall.
    if runtime.GOOS == "windows" {
        cmd := Command(executablePath, cmdArgs...)
        cmd.Stdout = os.Stdout
        cmd.Stderr = os.Stderr
        cmd.Stdin = os.Stdin
```

```
11          cmd.Env = environment
12          err := cmd.Run()
13          if err == nil {
14              os.Exit(0)
15          }
16          return err
17      }
18
19      // invoke cmd binary relaying the environment and args given
20      // append executablePath to cmdArgs, as execve will make first argument the "binary name".
21      return syscall.Exec(executablePath, append([]string{executablePath}, cmdArgs...), environment)
22  }
```

Execute plugins with syscall

Now, let's try to execute our simple `kubectl-educative` plugin.

We invoke our plugin running command `kubectl educative`. After running this command in the terminal above, we'll get the following output:

```
1  Hello! Programming with Kubernetes.
```

The output of our simple plugin

Back

Introduction to kubectl

Next

Implementing a kubectl Plugin