Extending Go Applications with Exec Plugins

KELSEY HIGHTOWER

Kelsey Hightower has worn every hat possible throughout his career in tech, and enjoys leadership roles focused on making things happen and shipping software. Kelsey is a strong open source advocate focused on building simple tools that make people smile. When he is not slinging Go code, you can catch him giving technical workshops covering everything from programming to system administration and distributed systems.
kelsey.hightower@gmail.com

Go has found a sweet spot among developers for building system tools ranging from Web services and distributed databases to command line tools. Most system tools of this nature tend to support multiple backends for providing application-specific functionality. For example, think about a command line tool that manages DNS records. Given the number of DNS providers available today, it would be nearly impossible to build support for every DNS API a user would want to interact with. This is where a plugin system can help. Plugin systems provide a common interface for extending applications with new functionality without major changes to the primary application.

In the Go ecosystem, the two most common ways of extending an application are by adopting an RPC plugin mechanism or by adding new code to a project that implements a plugin interface. Writing plugins using an RPC interface has the benefit of supporting plugins that live outside the core code base. This way, users can add and remove plugins without recompiling the main application. The major drawback to RPC plugins is the increased complexity that comes with running each plugin as a daemon and interacting with them over a network socket.

The practice of using RPC plugins is a bit of a hack; however, the method has become widespread because there is no other way to extend a Go application without adding new code and recompiling the main application. Go lacks the ability to load and execute external code at runtime, a feature that is common in languages like C or Java.

In the case of source plugins, each plugin lives in the main code base and typically implements a well-defined interface. Each plugin is only responsible for implementing the methods defined in the interface, which can greatly streamline plugin development.

Let's dive deeper into this topic and write some code that implements a source plugin mechanism. The application we are going to build is called translate, so named because it can translate a message from English to another language. It works like this:

```
$ translate -m "hello" -t spanish
hola
```

The translate application supports multiple languages through a simple plugin system. For each language we want to support we must add a new Go package for that language and implement the following interface:

```
type Translator interface {
  Translate(message string) (string, error)
}
```

Each translation plugin must provide a method named Translate that takes a message argument and returns the translated message and an error if the translation fails. The translate application allows users to select the translation plugin to use via the -t flag.

Create the following directories to hold the translate app source code:

```
$ mkdir -p $GOPATH/src/translate/plugins/spanish
```

Change into the translate source directory:

```
$ cd $GOPATH/src/translate
```

Save the following source code to a file named main.go:

```
package main

import (
   "flag"
   "fmt"
   "log"
   "translate/plugins"
   "translate/plugins/spanish"
)

func main() {
   var translator string
   var message string

   flag.StringVar(&translator, "t", "english", "Which translator
plugin to use")
   flag.StringVar(&message, "m", "", "The message to translate")
   flag.Parse()

   var t plugins.Translator

   switch translator {
   case "spanish":
      t = spanish.New()
   default:
      fmt.Printf("Plugin %s not found.", translator)
   }

   response, err := t.Translate(message)
   if err != nil {
      log.Fatal(err)
   }
   fmt.Println(response)
}
```

Save the following source code to a file named plugins/translator.go:

```
package plugins

type Translator interface {
   Translate(message string) (string, error)
}
```

```
Save the following source code to a file named plugins/spanish/
translator.go
```

```
package spanish

import (
   "errors"
)

type Translator struct{}

func New() Translator {
   return Translator{}
}

func (t Translator) Translate(message string) (string, error) {
   if message == "hello" {
      return "hola", nil
   }
   return "", errors.New("Translation error.")
}
```

At this point you can build and execute the translator application:

```
$ go build .
$ ./translate -t spanish -m "hello"
hola
```

As you can see, the -t flag allows us to select the language translator to use, and the -m flag sets the message to translate. The translate program will select the right plugin to process the translation based on the value of the -t flag. Let's review the code that makes this happens:

```
switch translator {
   case "spanish":
      t = spanish.New()
   default:
      fmt.Printf("Plugin %s not found.", translator)
}
```

Notice the problem here? We must know and implement every translator plugin before compiling the application. To extend the translate application, you'll have to modify its source tree, recompile, and reinstall the translate application. For many end users, source plugins set a barrier too high for contribution, which results in more work for project maintainers, who must either build or review every plugin users want to implement or use. A better solution to this problem would be to allow users to extend the translate application without modifying the code base. That's where exec plugins come in.

Exec Plugins

Exec plugins allow you to leverage external binaries as a plugin framework for extending applications. It's easy to think of exec plugins as similar to talking to an RPC endpoint. For each action, an executable can be invoked with a specific set of flags or environment variables to complete a task. Exec plugins can be written in any language (avoiding one drawback of source plugins) and have simple interface requirements.

Let's explore how an exec plugin system can improve the extensibility of our translate application. The goal is to keep the same top-level interface but make it possible to add new translation plugins without recompiling the translate application.

In the next part of this tutorial, you'll rewrite the translate application to provide extensibility through exec plugins. Start by deleting the current translate code base:

```
$ rm -rf $GOPATH/src/translate/*
```

Change to the translate source directory:

```
$ cd $GOPATH/src/translate
```

Save the following source code to a file named main.go:

```
package main

import (
    "bytes"
    "flag"
    "fmt"
    "log"
    "os"
    "os/exec"
    "path"
)

var (
    pluginsDir string
)

func main() {
    var translator string
    var message string

    flag.StringVar(&translator, "t", "english", "Which translator
plugin to use")
    flag.StringVar(&message, "m", "", "The message to translate")
    flag.StringVar(&pluginsDir, "p", "/tmp", "The plugin
directory")
    flag.Parse()

    t := path.Join(pluginsDir, translator)
    if _, err := os.Stat(t); os.IsNotExist(err) {
        fmt.Printf("Plugin %s not found.\n", translator)
        os.Exit(1)
    }

    cmd := exec.Command(t, "-m", message)
    var response bytes.Buffer
    cmd.Stdout = &response
    err := cmd.Run()
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(response.String())
}
```

The main difference from the original translate application is that we no longer need to know and implement each plugin before compile time. Instead, the translator plugin (identified by the -t flag with the path to an executable) will be called to translate the message:

```
t := path.Join(pluginsDir, translator)
if _, err := os.Stat(t); os.IsNotExist(err) {
   fmt.Printf("Plugin %s not found.", translator)
   os.Exit(1)
}
```

If the executable is not found on the file system, the translate application will print an error message and exit non-zero. This is a big improvement over the source plugin model, where the search for plugins is done only at runtime. Now plugins can be added by simply creating a binary and placing it in the translate plugins directory. If our app were a daemon, this would mean no restarts!

At this point you can build and execute the translate application:

```
$ go build .
$ ./translate -t spanish -m "hello"
Plugin spanish not found.
```

The updated translate application throws an error here because we have not written or installed any exec plugins. Let's write our first translator plugin and add support for a new translation; how about Japanese?

But just like the source plugin we need an interface to conform too. In the case of the exec plugin, we define the following interface:

- Translator plugins MUST accept a message to translate via an "-m" flag.
- Translator plugins MUST print the translation to standard out if the translation is successful.
- Translator plugins MUST exit non-zero if the translation fails.

With our interface defined, let's create a plugin.

First, create the the following directory to hold the Japanese translate plugin source code:

```
$ mkdir -p $GOPATH/src/japanese-translate-plugin/
```

Change into the Japanese translate plugin source directory:

```
$ cd $GOPATH/src/japanese-translate-plugin/
```

Save the following source code to a file named main.go:

```
package main

import (
    "flag"
    "fmt"
    "log"
    "os"
)

func main() {
    var message string
    flag.StringVar(&message, "m", "", "The message to translate")
    flag.Parse()

    if message == "hello" {
        fmt.Printf("こんにちは")
        os.Exit(0)
    }
    log.Fatal("Translation error.")
}
```

Compile the Japanese translator plugin:

```
$ go build -o japanese .
```

Before we copy the Japanese translator plugin to the translate application's plugin directory, we should test that it works. This is a clear benefit of exec plugins—we can test them outside of the main application:

```
$ ./japanese -m "hello"
こんにちは
```

Everything seems to be working. Now you need to copy the Japanese translator plugin to the translate plugin directory:

```
$ cp japanese /tmp/japanese
```

With the Japanese translator plugin in place, we can now rerun the translate application and use the Japanese plugin to handle the translation:

```
$ cd $GOPATH/src/translate
$ ./translate -m "hello" -t japanese
こんにちは
```

Following this pattern, adding support for new translations is easy:

```
$ cp norwegian /tmp/norwegian
$ ./translate -m "goodbye" -t norwegian
ha det
```

Exec plugins provide a simple mechanism for extending applications in a way that empowers end users and reduces maintenance overhead for project owners, and is a testament to the longevity of UNIX semantics and philosophy.