What's New in Go 1.6—Vendoring

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 1.6 was released in Q1 of 2016 and introduced support for Linux on MIPS and Android, HTTP2 support in the standard library, and some amazing improvements to the garbage collector (GC), which reduced latency and improved performance for the most demanding applications written in the language. Improvements to the standard library and runtime normally get the most attention leading up to a new release. However, with the release of Go 1.6 it's all about dependency management, which takes on one of the biggest pain points in the Go community.

Before you can really appreciate the impact of the Go 1.6 release, and the work around improving dependency management, we need to review how we got here.

Manual Dependency Management

Until recently, managing dependencies in Go required pulling external libraries into your GOPATH and attempting to build your application. To make things easy, Go ships with the go tool, which automates fetching dependencies and putting things in the right place. In the early days one could argue this was enough to get by. There was no real pressure to focus on tracking versions of your dependencies, largely because everything was relatively new and had a single version.

Let's take a look at managing dependencies for a simple application called hashpass—which prints a bcrypt hash for a given password.

First we need to create a directory to hold the hashpass source code:

```
$ mkdir -p $GOPATH/src/github.com/kelseyhightower/hashpass
$ cd $GOPATH/src/github.com/kelseyhightower/hashpass
```

Now save the hashpass source code to a file named main.go:

```
package main

import (
    "fmt"
    "log"
    "syscall"

    "golang.org/x/crypto/bcrypt"
    "golang.org/x/crypto/ssh/terminal"
)

func main() {
    fmt.Println("Password:")
    password, err := terminal.ReadPassword(syscall.Stdin)
    if err != nil {
        log.Fatal(err)
```

```
    }
    passwordHash, err := bcrypt.GenerateFromPassword
(password, 12)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println(string(passwordHash))
}
```

With the hashpass source code in place it's time to compile a binary:

```
$ go build .

main.go:8:5: cannot find package "golang.org/x/crypto
/bcrypt" in any of:
    /usr/local/go/src/golang.org/x/crypto/bcrypt (from $GOROOT)
    /Users/khightower/go/src/golang.org/x/crypto/bcrypt
(from $GOPATH)
main.go:9:5: cannot find package "golang.org/x/crypto/ssh
/terminal" in any of:
    /usr/local/go/src/golang.org/x/crypto/ssh/terminal
(from $GOROOT)
    /Users/khightower/go/src/golang.org/x/crypto/ssh/terminal
(from $GOPATH)
```

Fail.

The build did not work, but what happened? The error message is telling us that we are missing a few dependencies required to build hashpass. Recall the import block at the top of the main.go source file.

```
import (
    "fmt"
    "log"
    "syscall"

    "golang.org/x/crypto/bcrypt"
    "golang.org/x/crypto/ssh/terminal"
)
```

The first three imports—fmt, log, and syscall—can be found in the standard library installed as part of the Go distribution. The next two libraries are external dependencies that must be fetched and installed into our GOPATH before we can use them.

Use the go tool to fetch both external dependencies:

```
$ go get golang.org/x/crypto/bcrypt
$ go get golang.org/x/crypto/ssh/terminal
```

Notice we are not fetching a specific version of our external dependencies. Yeah, you can see where this is going; stay with me. With our external dependencies in place, try building hashpass again, but this time use the -v flag to print each of the dependencies as they are being compiled:

```
$ go -v build .
golang.org/x/crypto/blowfish
golang.org/x/crypto/ssh/terminal
golang.org/x/crypto/bcrypt
github.com/kelseyhightower/hashpass
```

Success! We now have the hashpass binary in our current directory. Run the hashpass binary and enter a (fake) password to get a bcrypt hash:

```
$ ./hashpass
Password:
$2a$12$bD51ZjG//
NWrbo5dYWSFeeppvJwZazRBWqLBh4afnPOpUQSg3yAMy...
```

Managing dependencies this way works for simple projects with few external dependencies, but there are many hidden gotchas here. We are not tracking each version of our dependencies, which means other people will have a hard time reproducing our build. The version of our dependencies is based on when we fetched them, not specific versions we declared ahead of time. This is the problem the Go community has been trying to solve for nearly five years.

Third Party Dependency Management Tools

Given the challenge of manually managing dependencies, many third party tools started to appear, Godep being the most popular. Godep helps track which version of a dependency your project is using and optionally includes those dependency source trees in the same repository as your code through a process called vendoring.

Let's see Godep in action. Install Godep using the go get command:

```
$ go get github.com/tools/godep
```

Remove the existing hashpass external dependencies from your GOPATH:

```
$ rm -rf $GOPATH/src/golang.org/x/
```

Now we are ready to use Godep to manage the hashpass external dependencies. Let's start from the hashpass source code directory:

```
$ cd $GOPATH/src/github.com/kelseyhightower/hashpass
```

Use the godep get command to fetch the hashpass external dependencies.

```
$ godep get .
Fetching https://golang.org/x/crypto/bcrypt?go-get=1
Fetching https://golang.org/x/crypto?go-get=1
Fetching https://golang.org/x/crypto/ssh/terminal?go-get=1
...
```

Use the godep save command to record the version of each dependency in use and copy their source code into your repository:

```
$ godep save
```

The godep save command creates a working directory named Godeps. Take a moment to explore it.

```
$ ls Godeps/
Godeps.json   Readme   _workspace
```

The Godeps.json file is used to record the versions of our dependencies:

```
$ cat Godeps/Godeps.json
{
  "ImportPath": "github.com/kelseyhightower/hashpass",
  "GoVersion": "go1.5",
  "GodepVersion": "v62",
  "Deps": [
    {
      "ImportPath": "golang.org/x/crypto/bcrypt",
      "Rev": "b8a0f4bb4040f8d884435cff35b9691e362cf00c"
    },
    {
      "ImportPath": "golang.org/x/crypto/blowfish",
      "Rev": "b8a0f4bb4040f8d884435cff35b9691e362cf00c"
    },
    {
      "ImportPath": "golang.org/x/crypto/ssh/terminal",
      "Rev": "b8a0f4bb4040f8d884435cff35b9691e362cf00c"
    }
  ]
}
```

Not what you are accustomed to seeing, right? The Godeps.json file is not tracking semantic version numbers like 1.0.0 or 2.2.1. This is where the Go community differs from many others. In the Go community we track the entire source tree that we depend on and do not rely on version numbers—version numbers can be reused and can point to later versions of a source tree, a security nightmare waiting to happen.

Another thing that's not so obvious at first glance, Godeps copies all of our dependencies into our source tree under the Godeps /_workspace directory:

```
$ tree -d Godeps/_workspace/
Godeps/_workspace/
└── src
    └── golang.org
        └── x
            └── crypto
                ├── bcrypt
                ├── blowfish
                └── ssh
                    └── terminal

8 directories
```

The Godeps/_workspace directory mirrors part of the GOPATH we depend on for building our application. The idea here is to check in the entire Godeps directory and ensure our dependencies live next to our code. The Godeps/_workspace directory is ignored by the go build tool because it starts with an underscore and will require the godep command as a wrapper around the go build command.

```
$ godep go build -v .
```

The godep command ensures the Godeps/_workspace directory is included at the front of the GOPATH, which causes our vendored dependencies to take priority during the build process.

The main drawback to using Godep to manage dependencies was that Godep was non-standard and incompatible with the go tool—go get will ignore the Godeps/_workspace directory and force users to check out your entire project to build your application.

Vendoring

Vendoring makes it easier to deliver reproducible builds and reduces reliance on remote code repositories hosting your dependencies—it also prevents your development team from scrambling when GitHub goes down in the middle of a release: fun times!

Godep proved that managing dependencies can be done with the right tools, but required help from the core Go project to reach its full potential. That help arrived in Go 1.6; a snippet from the release notes:

- Go 1.6 includes support for using local copies of external dependencies to satisfy imports of those dependencies, often referred to as vendoring.
- Code below a directory named "vendor" is importable only by code in the directory tree rooted at the parent of "vendor," and only using an import path that omits the prefix up to and including the vendor element.

In a nutshell Go 1.6 will search a vendor directory for external dependencies, but you'll still need tools to copy them there.

Newer versions of Godeps will detect that you're using Go 1.6 and copy your dependencies into the vendor directory.

```
$ cd $GOPATH/src/github.com/kelseyhightower/hashpass
$ rm -rf Godep
$ godep save
```

Godep creates a vendor directory and copies the hashpass external dependences into it:

```
$ tree -d vendor/
vendor/
└── golang.org
    └── x
        └── crypto
            ├── bcrypt
            ├── blowfish
            └── ssh
                └── terminal

7 directories
```

As an added bonus, Godeps continues to write the Godeps/Godeps.json file, so you can track exactly where your dependencies came from. At this point we can rebuild hashpass without any wrapper tools thanks to support for the vendor directory.

```
$ go build -v
github.com/kelseyhightower/hashpass/vendor/golang.org/x/crypto/blowfish
github.com/kelseyhightower/hashpass/vendor/golang.org/x/crypto/ssh/terminal
github.com/kelseyhightower/hashpass/vendor/golang.org/x/crypto/bcrypt
github.com/kelseyhightower/hashpass
```

Notice all the hashpass dependencies are being pulled in from the local vendor directory. Feel free to take the rest of the day off, you've earned it!

Summary

Since the release of Go 1.0, the Go community has been on a long journey to get a handle on dependency management. Over the years the community has stepped in to help define what the right dependency management solution looks like for Go and, as a result, gave way to the idea of vendoring and reproducible builds. Now, with the release of Go 1.6, the community has a standard we can rely on for many years to come.