Package Management With Dep

Package Management With Dep

Package management in Go has been a hotly contested topic since Go went public. There are a lot of tools out there to help you manage package dependencies for your app.

The `dep` tool has been dubbed the "official experiment" for package management in Go.

The `vendor` Directory

The `vendor` directory was added in Go `1.5`. This directory, which leaves in your project is where you can store other Go packages your application depends on.

The Go build tools are aware of this directory and will look at this directory first, before the `$GOPATH` to find dependencies when building your application.

The `vendor` Directory

- Go `1.5`: `vendor/` added, off by default
- Go `1.6`: `vendor/` on by default
- Go `1.7`: `vendor/` always on

A Lot Of Tools

There have been a lot of tools to manage the vendor directory, `glide`, `godeps`, etc... In 2017 the Go team launched `dep` as an "official experiment" to try and solve dependency management more "formally".

Installation

Dep is, primarily, a command line tool.

```
$ go get -u github.com/golang/dep/cmd/dep
```

According to the maintainers it should be used as a CLI tool, and not a Go package.

Dep Fundamentals

- Borrows from others, but is tailored to Go
- Imports rule
- Two-file system: `Gopkg.toml`, `Gopkg.lock`
- Project-oriented
- Semver tagging
- `vendor/`-centric - (almost) no `$GOPATH`

Dep Fundamentals

There are three commands at the heart of the `dep` tool.

- `dep init`
- `dep ensure`
- `dep status`

`dep Init`

1. Identifies your dependencies.
2. Checks if your dependencies use Dep.
3. Picks the highest compatible version for each dependency.

```
$ dep init
```

Migrating

If you are using Glide or Godeps, the `dep init` command nows how to migrate those files to using Dep.

1. Imports your existing configuration.
2. Checks if your dependencies use Dep.
3. Falls back to the simple case to fill in gaps.

Initialize With The `$GOPATH`

```
$ dep init -gopath
```

1. Uses the branch/version/revision found in `$GOPATH`.
2. Checks if your dependencies use Dep.
3. Falls back to the simple case to fill in gaps.

Example

Running `dep init` on the `github.com/gobuffalo/plush` package will generate a `Gopkg.toml` file that looks something like this.

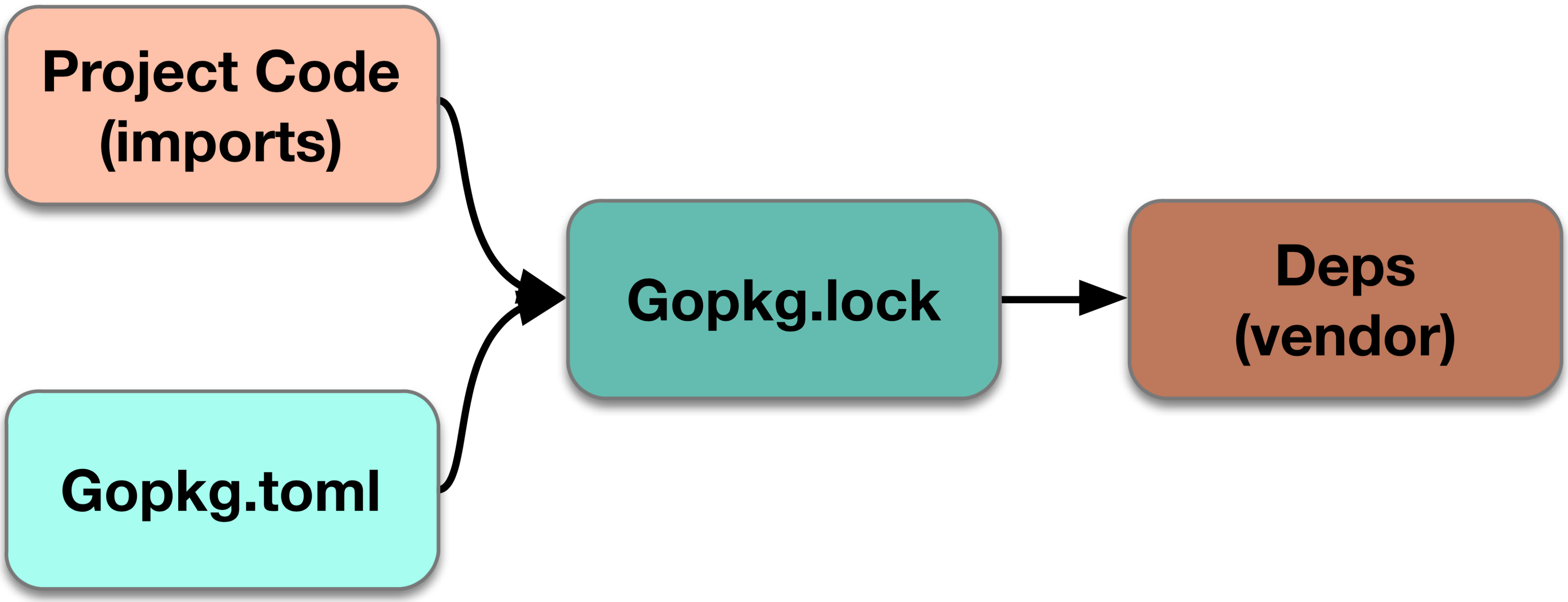
```
  branch = "master"
  name = "github.com/gobuffalo/tags"

[[constraint]]
  branch = "master"
  name = "github.com/markbates/inflect"

[[constraint]]
  name = "github.com/pkg/errors"
  version = "0.8.0"

[[constraint]]
  branch = "master"
  name = "github.com/shurcooL/github_flavored_markdown"

[[constraint]]
  name = "github.com/stretchr/testify"
  version = "1.1.4"

[[constraint]]
  branch = "master"
  name = "golang.org/x/sync"
```

Example

It will also generate a `Gopkg.lock` file that similar to this.

```
# This file is autogenerated, do not edit; changes may be undone by the next 'dep ensure'.

[[projects]]
  name = "github.com/davecgh/go-spew"
  packages = 
  revision = "346938d642f2ec3594ed81d874461961cd0faa76"
  version = "v1.1.0"

[[projects]]
  name = "github.com/fatih/structs"
  packages = 
  revision = "a720dfa8df582c51dee1b36feabb906bde1588bd"
  version = "v1.0"

[[projects]]
  branch = "master"
  name = "github.com/gobuffalo/tags"
  packages = 
  revision = "fb0fc064d3c07b9ce77d8669bbf64f9e8c721869"

[[projects]]
  branch = "master"
  name = "github.com/markbates/inflect"
  packages = 
  revision = "6cacb66d100482ef7cc366289ccb156020e57e76"

# ...
```

How `dep Init` Works



Gopkg.toml

The `Gopkg.toml` file is where you tell Dep which dependencies you want to manage, and how to manage those dependencies.

With a Version

```toml
name = "github.com/sdboyer/deptest"
version = "0.8.0"
```

With a Branch

```toml
branch = "master"
name = "golang.org/x/sync"
```

With a Fork

```toml
name = "github.com/sdboyer/deptest"
source = "http://github.com/carolynvs/deptest"
version = "0.8.0"
```

Version Ranges

Dep prefers ranges over specific versions.

```
1.2.3 becomes >=1.2.3, <2.0.0
0.2.3 becomes >=0.2.3, <0.3.0
0.0.3 becomes >=0.0.3, <0.1.0
```

Use `=0.8.1` to pin to a version

Already Vendored Dependencies

If you already have a `vendor` folder, and `dep init` does not recognize the tool used to create it, it will do the following.

1. Makes a backup of vendor and takes over.
2. Generates an initial set of configuration files.
    - **These, most likely, will not match your original vendor!**
3. Leaves it up to you to fix them by hand.

Who Owns What?

What You Own

- `Gopkg.toml` - Hand edit and update as necessary.

What Dep Owns

- `Gopkg.lock` - Do **not** edit this file. All changes **WILL** be lost!
- `vendor/` - Do **not** modify this directory. All changes **WILL** be lost!

Maintaining Your Application

Check Your Status

While trying to understand what dependencies your application is using, and what versions they are, you can use the `Gopkg.toml` and the `Gopkg.lock` files.

Alternatively, you can use the `dep status` command to print out a user friendly table of this information.

```
$ dep status

PROJECT                     CONSTRAINT  VERSION        REVISION  LATEST   PKGS
github.com/pkg/errors       ^0.8.0      v0.8.0         645ef00   645ef00  1
github.com/sirupsen/logrus  ^1.0.2      1.0.2          a3f95b5   a3f95b5  1
golang.org/x/sys            *           branch master  0f826bd   0f826bd  1
```

Adding A New Package

If we were to add a new `import` statement to an application and run `dep status` we would see something similar to the following.

```
$ dep status

Lock inputs-digest mismatch due to the following packages missing from the lock:

PROJECT                    MISSING PACKAGES
github.com/gobuffalo/envy  [github.com/gobuffalo/envy]

This happens when a new import is added.
Run `dep ensure` to install the missing packages.
```

Adding A New Package

After adding a new `import` to a project, you must run `dep ensure` to let Dep manage that new package.

```
$ dep ensure
```

```
$ dep status

PROJECT                          CONSTRAINT  VERSION        REVISION  LATEST   PKGS USED
github.com/gobuffalo/envy        *           branch master  a901aeb   a901aeb  1
github.com/mitchellh/go-homedir  *           branch master  b8bc1bf   b8bc1bf  1
github.com/pkg/errors            ^0.8.0      v0.8.0         645ef00   645ef00  1
github.com/sirupsen/logrus       ^1.0.2      1.0.2          a3f95b5   a3f95b5  1
golang.org/x/sys                 *           branch master  0f826bd   0f826bd  1
```

Updating A Package

There are two ways to update a dependency in your application.

The first is to update your `Gopkg.toml` file specifying the new version and then run `dep ensure`.

Alternatively, you can update the dependency with the `dep ensure` command directly:

```
$ dep ensure -add github.com/pkg/foo@^1.0.1
```

Updating All Dependencies

If you want to update all of the dependencies of your application you can use the `-update` flag.

```
$ dep ensure -update
```

Pro-tip: Use the `-dry-run` flag to do a dry run first.

```
$ dep ensure -update -dry-run
```

Removing A Dependency

1. Stop using it in your code.
2. Run `dep ensure`.
3. If it was in `Gopkg.toml`, you may remove it.

Vendor And SCM

The decision to commit the `vendor/` directory is up to you and your organization.

Pros

- Repeatable builds.
- No network calls necessary.
- The "owner" can not take the code away. (think "leftpad")

Cons

- Bloated repos.
- (Potentially) large and confusing PRs.

Vendor Folder Gotchas

Prior to Go `1.9`, tooling in Go did **not** ignore the `vendor/` directory.

```
$ go test ./...

ok      github.com/gobuffalo/plush  0.020s
ok      github.com/gobuffalo/plush/ast  0.012s
ok      github.com/gobuffalo/plush/lexer  0.012s
ok      github.com/gobuffalo/plush/parser 0.014s
?       github.com/gobuffalo/plush/token  [no test files]
ok      github.com/gobuffalo/plush/vendor/github.com/davecgh/go-spew/spew 0.016s
ok      github.com/gobuffalo/plush/vendor/github.com/fatih/structs  0.011s
ok      github.com/gobuffalo/plush/vendor/github.com/pkg/errors 0.296s
ok      github.com/gobuffalo/plush/vendor/github.com/pmezard/go-difflib/difflib 0.016s
ok      github.com/gobuffalo/plush/vendor/github.com/russross/blackfriday 11.633s
ok      github.com/gobuffalo/plush/vendor/github.com/serenize/snaker  0.022s
ok      github.com/gobuffalo/plush/vendor/github.com/sergi/go-diff/diffmatchpatch 0.437s
ok      github.com/gobuffalo/plush/vendor/github.com/shurcooL/github_flavored_markdown  0.026s
ok      github.com/gobuffalo/plush/vendor/github.com/shurcooL/go/analysis 0.011s
ok      github.com/gobuffalo/plush/vendor/github.com/shurcooL/go/ctxhttp  0.117s
ok      github.com/gobuffalo/plush/vendor/github.com/shurcooL/go/generated  0.011s
ok      github.com/gobuffalo/plush/vendor/github.com/sourcegraph/annotate 0.012s
ok      github.com/gobuffalo/plush/vendor/github.com/sourcegraph/syntaxhighlight  0.013s
?       github.com/gobuffalo/plush/vendor/github.com/sourcegraph/syntaxhighlight/cmd/syntaxhighlight
...
```

*Run using Go `1.8`

Vendor Folder Gotchas

In Go `1.9` the tooling knows to ignore the `vendor/` directory.

```
$ go test ./...

ok      github.com/gobuffalo/plush  0.012s
ok      github.com/gobuffalo/plush/ast  0.009s
ok      github.com/gobuffalo/plush/lexer  0.011s
ok      github.com/gobuffalo/plush/parser 0.009s
?       github.com/gobuffalo/plush/token  [no test files]
```

*Run using Go `1.9`

The Future Of Vendoring

The Go team has said that Dep is an "official experiment". They, eventually, would like to remove the need for a `vendor/` directory and still have the same functionality using the two file system that Dep currently offers.

The road map for this is currently unknown.

Exercise

```
        main

       (
    "github.com/pkg/errors"
    "github.com/sirupsen/logrus"
)

         () {
    logrus.Error(errors.New("boom!"))
}
```

- Run `dep init` on this application.
- Run `go run main.go` - verify it works.
- Examine the `Gopkg.lock` file.

- Set the version of `github.com/pkg/errors` to `v0.7.0`.
- Run `dep ensure`.
- Run `go run main.go` - verify it works.
- Examine the `Gopkg.lock` file.

- Import a new package.
- Run `dep status`
- Run `dep ensure`.
- Run `go run main.go` - verify it works.
- Examine the `Gopkg.lock` file.

References

Links

- https://github.com/golang/dep - For more information on Dep itself.
- https://github.com/Masterminds/semver - For more information on supported operators.

Thank You

A lot of this module was the result of the following two talks given at GopherCon 2017:

- http://carolynvs.com/dep-in-10
- https://github.com/gophercon/2017-talks/tree/master/samboyer-TheNewEraOfGoPackageManagement