Introduction

Now that you've spent many hours writing your package and debugging it and testing it (you did [[test it|TableDrivenTests]], didn't you?), you want to publish it so other people can go get your package.

First, you will need to host it online somewhere. Three major code hosting sites are bitbucket (hg/git), GitHub (git) and launchpad (bzr). I recommend choosing whichever version control system you are familiar with or which your code is versioned locally on your machine. Git (git) is the version control system used by the central Go repository, so it is the closest to a guarantee as you can get that a developer wanting to use your project will have the right software. If you have never used version control before, these websites have some nice HOWTOs and you can find many great tutorials by searching Google for "{name} tutorial" where {name} is the name of the version control system you would like to learn.

Package Setup

Choosing the Import

The full import of your package often has something identifying its author (particularly on hosting sites like GitHub, where "github.com/kylelemons/..." is the full import), should always have the project name, and should end with the name of the package you've developed if it is different from the project name. For instance, the go-gypsy project provides a yaml package, and is written by Kyle Lemons, and thus has the following import path:

```
import "github.com/kylelemons/go-gypsy/yaml"
              ^          ^           ^     ^
              |          |           |     `-- Package name
              |          |           `-------- Project name
              |          `-------------------- Author's handle
              `------------------------------ Hosting site
```

Go >= version 1 supports subdirectories of package repositories.

Subdirectories

Frequently, the name that you use for your package will include the name "Go" as a prefix, suffix, or part of its acronym, and you may or may not want this to be a part of the actual command or package name in a go source file. Often, you may have both libraries and commands as a part of your package, and these cannot coexist in the same directory. When these things happen, you will want to structure your repository with subdirectories.

For example, consider a project "Go-PublishingExample" that provides an "epub" package and a "publish" command. The directory structure could be:

```
./epub/      # Package source, all files package "epub"
./publish/   # Command source
./doc/       # Documentation which won't be downloaded
./examples/  # Example code which won't be downloaded
```

The import statement for the package would look like:

```
import "codesite.tld/authorName/Go-PublishingExample/epub"
```

It is often a good idea to make sure the last directory path (in this case, "epub") matches the name of the package used by the source files in the directory. In this case, no go get-able files were included in the base directory because neither the binary nor the package were to be named "Go-PublishingExample".

Branches and Tags

Please note that this section is out of date. The pseudo version numbers below were applicable to Go < version 1; also the Go repository itself uses Git instead of Mercurial now. **Maybe we should remove this section.**

You can get up to date information on go get using "go help get" and "go help importpath".

In general, the Go source tree can exist in three basic states. It can be checked out at a Go Release branch (r60 (on Google Code) at the time of this writing -- this is where most users should be), or it can be checked out at a Go Weekly (a new tag for which is made roughly once per week), or at tip (the Mercurial term for the latest change). The last two are primarily for developers of the Go language itself or developers who need features or fixes which have not been introduced into the latest Release.

Due to the lielihood that you might continue collaborating on your project with your team on code that is not ready for general consumption, it is recommended that you utilize the tagging or branching functionality of your version control system. The go get tool understands some special tags and branches, which you may want to use to ensure users get a compatible version of your package:

```
  go.r60               -- A "go.r##" tag will be checked out if the user has that Go
release installed
  go.weekly.2011-07-19 -- A "go.weekly.YYYY-MM-DD" tag will be checked out if the user
has that weekly installed
```

Go get will attempt to fall back to the previous tag if the installed one has no matches, and if none are found, will default to installing tip.

To create and maintain your release tag in mercurial:

```
# Create or update a release tag
hg tag myProj-v0.0 # tag an easy-to-remember version number if you wish
hg tag go.r60 # tag this as being go release.r60 compatible
```

To create and maintain a release branch in git:

```
# Create a release branch
git tag myProj-v0.0     # Tag an easy-to-remember version number if you wish
git checkout -b go.r60  # create a release branch
git checkout master     # to switch back to your master branch

# Update the release branch
git checkout go.r60  # switch to the release branch
git merge master     # merge in changes from the master branch since last release
git checkout master  # switch back to master branch
```

If you are using other branch names, substitute those names where necessary.

It will typically not be necessary to maintain weekly tags or branches, but it can be very useful to maintain the release branch or tag, as this will ensure the widest audience for your project.

Commands vs Packages

Since go get does not use your project's Makefiles, it is important to understand how it will actually build your project.

All files in the same directory should always share the same package name. Any files named with a `_test` or an `_os` and/or `_arch` suffix will be ignored (unless the os/arch match). If the package name is "main", go get will build an executable from the source files and name it according to the directory name (using the last path segment only). If the package name is anything else, go get will build it as a package and the import path will be the web-accessible URL for your project's root followed by the subdirectory. See the go get documentation for how to make import paths for code hosting sites other than the main four.

Dependencies between packages in the same project are common. In the case where one package or command in your project depends upon another, you must use the full import path in order for go get to recognize the dependency and make sure it is built. Third-party packages which are imported from source files in your project will also be automatically downloaded and installed by go get if it is not already present.

To reuse the example above, the file `./publish/main.go` may look something like this:

```
package main

import (
    "flag"
)

import "codesite.tld/authorName/Go-PublishingExample/epub"

var dir = flag.String("dir", ".", "Directory to publish")

func main() {
    flag.Parse()
    epub.Publish(*dir)
}
```

A user wishing to install this executable would execute:

```
go get codesite.tld/authorName/Go-PublishingExample/publish
```

which would also install the `"../epub"` package because of the dependency. A developer simply wishing to install the library could execute:

```
go get codesite.tld/authorName/Go-PublishingExample/epub
```

and (if they had not already installed `publish`) would only download and install the package. Note that in none of these cases are the examples or documentation downloaded; in most cases these would be available to browse via the code site.

Documentation

godoc

When you are preparing to publish a package, you should make sure that the documentation looks correct by running a local copy of godoc. If your package is installed to the go package tree, you can use the following command:

```
godoc -http=:6060 &
```

Then browse to http://localhost:6060/pkg/ and find your package.

Dashboard

The Go Dashboard will use the first line of your package-level comment (also using the normal godoc format) as the "info" text, so make sure this is set. For instance:

```
// Package epub is an example publishing library.
package epub
```

For more information on godoc, see the Documenting Go Code blog post.