Getting started with Go modules

Niraj Fonseka
Dec 6, 2018 · 5 min read

Golang is still a fairly new programming language. And dependency management for Go has been a bit tricky since its inception. At the beginning, there was no dependency management system. `go get` was the only way to download dependencies but this became very tricky because this pulls the code from the `master` branch of a repository and puts the files into your GOPATH. That means if there were any major changes to any dependencies there's high likelihood that your code will break. Then tools like `dep` and concepts such as `vendoring` came along. Personally I liked `dep` because it took away a lot of the headaches that came with using `go get` . But `dep` was still a third party tool that you had to download to get it to work. But now, with `go 1.11` things are finally changing.

So what are Go modules ?

Go modules in a nutshell is a built-in dependency versioning and dependency management feature for Go. Please note that this is still in experimental stage and the the finalized version will come with `go 1.12` .

In this article I will show you how Go modules work. And also how to move to Go modules if you are currently using `dep` as your dependency management tool.

First of all, you will need to make sure that you have Go 1.11 installed. But if you are a fan of docker and don't want to install Go 1.11 right now you can spin up a Go 1.11 container and use that to play around with Go modules. Also don't forget to install a text editor when you get inside the docker container.

```
docker run --rm -it golang:1.11

//after you get inside the container

apt-get update && apt-get install nano
```

Our first application !

Inside the container you can see that the go path is set to `/go`

```
root@275af4a4091c:~# echo $GOPATH
/go
```

But I'm going to try something that would've been very wrong to do in previous Go versions. I'm going to go into my home directory and create a folder for our app called

`TestApp` and set up my app there instead of inside GOPATH.

```
root@275af4a4091c:~# mkdir TestApp
root@275af4a4091c:~# ls
TestApp
```

Now let’s write our application inside the TestApp folder and name it `main.go`

```
package main

import (
 "fmt"
)

func main(){
 fmt.Println("Running the TestApp")
}
```

Now let’s run it.

```
root@275af4a4091c:~/TestApp# go run main.go
Running the TestApp
```

That’s right !! Now we can run Go programs outside of GOPATH !! But this app runs without throwing any errors because it’s only using the `fmt` library. But let’s try adding a third party library and see how that plays out. I’ll be using go-randomdata as our third party library. It lets you generate random data. In this case I’m going to call the function `SillyName()` after the first print statement.

This is how the update code looks like.

```
package main

import (
        "fmt"
        "github.com/Pallinder/go-randomdata"
)

func main(){
        fmt.Println("Running the TestApp")
        fmt.Println(randomdata.SillyName())
}
```

Let’s now run it.

```
root@275af4a4091c:~/TestApp# go run main.go
main.go:5:2: cannot find package "github.com/Pallinder/go-
randomdata" in any of:
 /usr/local/go/src/github.com/Pallinder/go-randomdata (from $GOROOT)
 /go/src/github.com/Pallinder/go-randomdata (from $GOPATH)
```

It complains about not finding the package because #1 we are outside of the GOPATH and #2 we haven’t downloaded that package yet. Let’s see how we can use Go modules to fix this. To initialize Go modules in to your application you can run `go modules init /path/to/directory`

```
root@275af4a4091c:~/TestApp# go mod init /root/TestApp
go: creating new go.mod: module /root/TestApp
```

You will see a new file called `go.mod` gets created in your app directory. Apart from the definition of the module there’s nothing in the go.mod at the moment. To get all the dependencies lets run `go build`

```
root@275af4a4091c:~/TestApp# go build
go: finding github.com/Pallinder/go-randomdata v1.1.0
go: downloading github.com/Pallinder/go-randomdata v1.1.0
```

Now let’s take a look at the go.mod file.

```
module /root/TestApp

require github.com/Pallinder/go-randomdata v1.1.0
```

It shows us that in order for our app to run, v1.1.0 is required of the go-randomdata package. After running `go build` you will also see a file called `go.sum` gets created. The purpose of this file is to keep track of which codebase of the package you are using. It has a cryptographic hash attached to every dependency we have. This is very helpful if you are working on a busy codebase that multiple people are working on at the same time. If someone is using a different version of a package than the version the app is intended to use, it will complain. If your third party library has multiple versions and you would like to use a different version, you can change the version number and run `go build` again. Let's say I would like to use v1.00 of go-randomdata instead. I simply change v1.1.0 to v1.00 and then run `go build` again. You will see how the `go.mod` and `go.sum` files change accordingly.

Now let's try running the app again.

```
root@275af4a4091c:~/TestApp# ls
TestApp  go.mod  go.sum  main.go
root@275af4a4091c:~/TestApp# go run main.go
Running the TestApp
Oxhorn
```

It works !!

Now let's take a look at some other commands that comes with Go modules.

`go mod tidy` : This command allows you to fetch all the dependencies that you need for testing in your module.

```
root@275af4a4091c:~/TestApp# go mod tidy
go: finding golang.org/x/text/language latest
go: finding golang.org/x/text v0.3.0
go: downloading golang.org/x/text v0.3.0
```

`go mod why -m <module>` : This command let's you find out where any of your dependencies are used. To demonstrate this I will use golang.org/x/text found in my `go.mod` file.

```
root@275af4a4091c:~/TestApp# go mod why -m golang.org/x/text
# golang.org/x/text
/root/TestApp
github.com/Pallinder/go-randomdata
github.com/Pallinder/go-randomdata.test
golang.org/x/text/language
```

The output shows that *golang.org/x/text/language* gets consumed by *github.com/Pallinder/go-randomdata.test* , *github.com/Pallinder/go-randomdata* is consumed by *github.com/Pallinder/go-randomdata*.

Like I mentioned before, Go modules are still in an experimental stage. While it works with Go 1.11, it may not work well with older Go versions. So you might run into a situation that you need to have your `vendor` files in your repo just to be safe.

To generate a vendor directory in your app while still maintaining the Go modules you can run `go mod vendor`

```
root@275af4a4091c:~/TestApp# go mod vendor
root@275af4a4091c:~/TestApp# ls
TestApp  go.mod  go.sum  main.go  vendor
root@275af4a4091c:~/TestApp# ls vendor/
github.com  modules.txt
```

Now let's talk about how to transition into Go modules from the most popular dependency management tool called `dep` . I used the same application we used in the previous example and used dep instead of go modules. You can find that code base here https://github.com/Niraj-Fonseka/TestAppDep. I will clone that into my home directory in the container.

```
root@275af4a4091c:~/TestAppDep# ls
Gopkg.lock  Gopkg.toml main.go  vendor
```

Now let's initialize go modules

```
root@275af4a4091c:~/TestAppDep# go mod init
go: creating new go.mod: module github.com/Niraj-Fonseka/TestAppDep
go: copying requirements from Gopkg.lock
```

And then `go mod tidy`

```
root@275af4a4091c:~/TestAppDep# go mod tidy
go: finding golang.org/x/text/language latest
```

and then delete `Gopkg.lock` , `Gopkg.toml` and the `vendor` directory. Then run `go get ./...`

Let’s look at the directory structure now.

```
root@275af4a4091c:~/TestAppDep# ls
go.mod go.sum main.go
```

And finally let’s run the app.

```
root@275af4a4091c:~/TestAppDep# go run main.go
Running the TestApp
Edgeorange
```

That’s it. Hopefully this will give you a basic understanding on how to get started with Go modules. If you would like to learn more about Go modules I would suggest looking at the Go release notes for `go 1.11` and https://github.com/golang/go/wiki/Modules. Also Francesc Campoy, an ex developer advocate for Golang, has an amazing video series called justforfunc where goes a lot more in-depth into Go modules and many other Go related topics.

Golang Dependencies Go Technology Programming