CONTENTS

Tutorial Series: How To Code in Go

44/53 How to Use a Private Go ... 45/53 How To Run Multiple Fun... 46/5

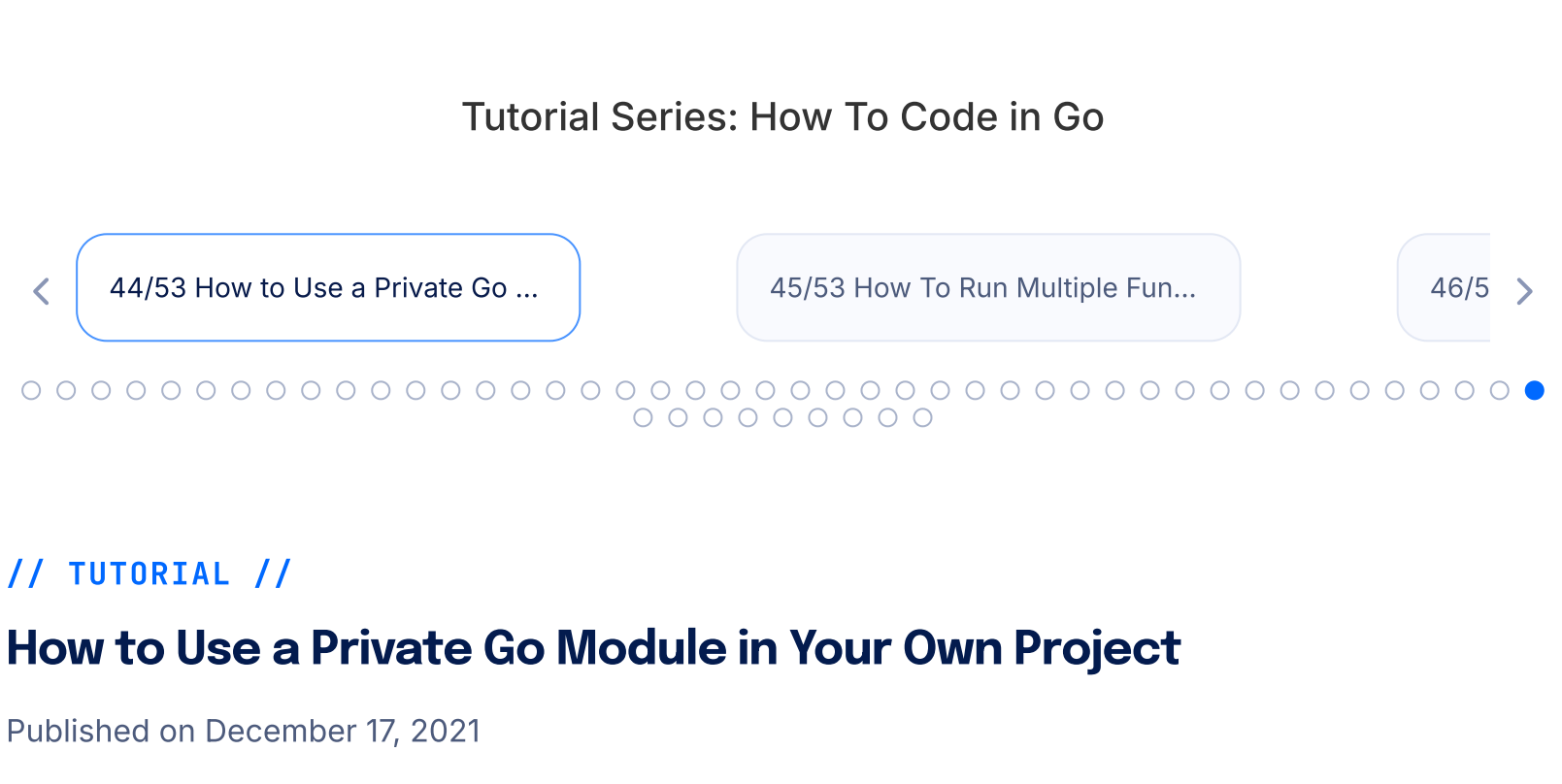
// TUTORIAL //

How to Use a Private Go Module in Your Own Project

Published on December 17, 2021

Development Go

Kristin Davidson and Rachel Lee



The author selected the Diversity in Tech Fund to receive a donation as part of the Write for DOnations program.

Introduction

One beneficial aspect of Go's ecosystem is that a large number of modules are open source. Since they're open source they can be freely accessed, examined, used, and learned from. However, sometimes it's necessary to make a private Go module for various reasons, such as keeping proprietary business logic internal to your company.

In this tutorial, you will publish a private Go module, set up authentication to access a private module, and use a private Go module in a project.

Prerequisites

- Go version 1.16 or greater installed. To set this up, follow the How To Install Go tutorial for your operating system.
- An understanding of distributing Go modules, which you can find in the How to Distribute Go Modules tutorial.
- Familiarity with Git, which you can get by following How To Use Git: A Reference Guide.
- An empty private GitHub repository named `mysecret` for your published private module. To get started, follow the GitHub documentation for creating a repository.
- A GitHub Personal Access Token with access to read from your repositories. You will use this to allow Go to access your private repository.

Distributing a Private Module

Unlike many programming languages, Go distributes modules from repositories instead of a central package server. One benefit of this approach is that publishing a private module is very similar to publishing a public one. Instead of requiring a completely separate private package server, a Go private module is distributed via a private source code repository. Since most source code hosting options support this out of the box, there's no need to set up an additional private server.

In order to use a private module, you'll need to have access to a private Go module. In this section, you'll create and publish a private module you can use later in the tutorial to access a private module from another Go program.

To create your new private Go module, start by cloning the private GitHub repository where it will live. As part of the prerequisites you created a private, empty repository named `mysecret` in your GitHub account and this is the one you will use for your private module. This repository can be cloned anywhere you'd like on your computer, but many developers tend to have a directory for their projects. In this tutorial, you'll use a directory named `projects`.

Make the `projects` directory and navigate to it:

```
$ mkdir projects
$ cd projects
```

From the `projects` directory, run `git clone` to clone your private `mysecret` repository to your computer:

```
$ git clone git@github.com:your_github_username/mysecret.git
```

Git will confirm it has cloned your module and may warn you that you have cloned an empty repository. If so, this is not something you need to worry about:

```
Output
```

```
Cloning into 'mysecret'...
warning: You appear to have cloned an empty repository.
```

Next, use `cd` to go into the new `mysecret` directory you cloned and use `go mod init`, along with the name of your private repository, to create a new Go module:

Copy

```
$ cd mysecret
$ go mod init github.com/your_github_username/mysecret
```

Now that your module is created, it's time to add a function you can use from another project. Use `nano`, or your favorite text editor, to open a file with the same name as your repository, such as `mysecret.go`. The name isn't significant, and could be anything, but using the same name as the repository makes it easier to determine which file to look in first when working with a new module:

Copy

```
$ nano mysecret.go
```

In the `mysecret.go` file, name the package with the same name as your repository, then add a `SecretProcess` function to print the line `Running the secret process!` when called:

projects/mysecret/mysecret.go Copy

```
package mysecret

import "fmt"

func SecretProcess() {
        fmt.Println("Running the secret process!")
}
```

Now that you have your private module created, you will publish it to your private repository for others to use. Since your private repository only allows you to access it initially, you're able to control who has access to your private module. You might restrict access to yourself, but you could also give access to friends or coworkers as well.

Since both private and public Go modules are source repositories, publishing a private Go module follows the same process as publishing a public one. To publish your new module, stage your changes in the current directory using the `git add` command, then commit those changes to your local repository with the `git commit` command:

Copy

```
$ git add .
$ git commit -m "Initial private module implementation"
```

You will see a confirmation from Git that your initial commit has succeeded as well as a summary of the files included in the commit:

```
Output
[main (root-commit) bda059d] Initial private module implementation
 2 files changed, 10 insertions(+)
 create mode 100644 go.mod
 create mode 100644 mysecret.go
```

Now the only part left is to move your changes to your GitHub repository. Similar to a public module, use the `git push` command to publish your code:

```
$ git push
```

Git will then push your changes and make them available to anyone with access to your private repository:

```
git push origin main
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 404 bytes | 404.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:your_github_username/mysecret.git
 * [new branch]      main -> main
```

As with a public Go module, you can also add versions to your private Go module. The Publishing a New Module Version section of the How to Distribute Go Modules tutorial includes information on how to do this.

In this section, you created a new module with a `SecretProcess` function and published it to your private `mysecret` GitHub repository, making it a private Go module. In order to access this module from another Go program, though, you'll need to configure Go so it knows how to access the module.

Configuring Go to Access Private Modules

While Go modules are commonly distributed from their source code repositories, the Go team also runs a few central Go module services to aid ensure modules continue to exist if something happens to the original repository. By default, Go is configured to use these services, but they can cause problems when you try to download a private module because they don't have access to those modules. To tell Go that some import paths are private and that it shouldn't try to use the central Go services, you can use the `GOPRIVATE` environment variable. The `GOPRIVATE` environment variable is a comma-separated list of import path prefixes where, when encountered, the Go tools will try to access them directly instead of going through the central services. One such example would be the private module you just created.

In order to use the private module, you will tell Go which path to consider private by setting it in the `GOPRIVATE` variable. There are a few choices you can make when setting your `GOPRIVATE` variable values. One option is to set `GOPRIVATE` to `github.com`. This might not be what you're looking for, though, because this would tell Go not to use the central services for any module hosted on `github.com`, including the ones that aren't yours.

The next option would be to set `GOPRIVATE` to only your own user path, such as `github.com/your_github_username`. This solves the problem of considering all of GitHub private, but at some point you may have public modules you've created that you'd like to download through the Go module mirror. Doing this wouldn't cause any problems and would be a perfectly reasonable option, but you also have the option of getting even more specific.

The most specific option would be setting `GOPRIVATE` to match the path of your module exactly, such as: `github.com/your_github_username/mysecret`. This solves both of the issues from the previous options, but also introduces the issue that you need to add each of your private repositories to `GOPRIVATE` individually, such as shown here:

```
GOPRIVATE=github.com/your_github_username/mysecret,github.com/your_github_username/othersecret
```

Choosing the best option for youself is a matter of weighing the pros and cons in your situation.

Since you only have a single private module now, we'll use the full repository name for the value. To set the `GOPRIVATE=github.com/your_github_username/mysecret` environment variable in your current terminal, use the export command:

```
$ export GOPRIVATE=github.com/your_github_username/mysecret
```

If you'd like to double-check that it's set, you can use the `env` command along with `grep` to check for the `GOPRIVATE` name:

```
$ env | grep GOPRIVATE
```

```
Output
GOPRIVATE=github.com/your_github_username/mysecret
```

Even though Go now knows your module is private, it's still not quite enough to use the module yet. If you try to `go get` your private module into another module, you'll likely see an error similar to:

```
$ go get github.com/your_github_username/mysecret
```

```
Output
go get: module github.com/your_github_username/mysecret: git ls-remote -q origin in /Users/your_g
        fatal: could not read Username for 'https://github.com': terminal prompts disabled
Confirm the import path was entered correctly.
If this is a private repository, see https://golang.org/doc/faq#git_https for additional information.
```

This error message says Go tried to download your module, but it encountered something it still doesn't have access to. Since Git is being used to download the module, it would usually ask you to enter your credentials. However, in this case, Go is calling Git for you and can't prompt for them. At this point, to access your module you'll need to provide a way for Git to retrieve your credentials without your immediate input.

Providing Private Module Credentials for HTTPS

One way to tell Git how to log in on your behalf is the `.netrc` file. Located in a user's home directory, the `.netrc` file contains various host names as well as log in credentials for those hosts. It's widely used by a number of tools, including Git.

By default, when `go get` tries to download a module, it will try to use HTTPS first. However, as shown in the previous example, it's not able to prompt you for your username and password. To give Git your credentials, you'll need to have a `.netrc` that includes `github.com` in your home directory.

To create a `.netrc` file on Linux, MacOS, or Windows Subsystem for Linux (WSL), open the `.netrc` file in your home directory (`~/`) so you can edit it:

```
$ nano ~/.netrc
```

Next, create a new entry in the file. The `machine` value should be the hostname you're setting the credentials for, which is `github.com` in this case. The `login` value should then be your GitHub username. Finally, the `password` value should be the GitHub personal access token you created.

```
~/.netrc
```

```
machine github.com
login your_github_username
password your_github_access_token
```

If you'd prefer, you can also put the entire entry on one line in the file as well:

~/.netrc

```
machine github.com login your_github_username password your_github_access_token
```

Note: If you are using Bitbucket for your source code hosting you may also need to add a second entry for `api.bitbucket.org` in addition to `bitbucket.org`. In the past, Bitbucket provided hosting for multiple types of version control, so Go would use the API to check the type of repository before trying to download it. While this is no longer the case, the API check still exists. If you encounter this issue, an example error message may look like this:

```
go get bitbucket.org/your_github_username/mysecret: reading https:// api.bitbucket.org /2.0/repositorie
        server response: Access denied. You must have write or admin access.
```

If you see the `403 Forbidden` error when trying to download a private module, double check the hostname Go is trying to connect to. It could indicate another hostname, such as `api.bitbucket.org`, that you need to add to your `.netrc` file.

Now your environment is set up to use HTTPS authentication for downloading your private module. Even though HTTPS is the default way Go and Git will try to download a module, it's also possible to tell Git to use SSH instead. Using SSH instead of HTTPS can be useful so you can use the same SSH key you used to push your private module. It also allows you to use deploy keys when setting up a CI/CD environment if you'd rather not create a personal access token.

Providing Private Module Credentials for SSH

To use your SSH key as the authentication method for your private Go module instead of HTTPS, Git provides a configuration option called `insteadOf`. The `insteadOf` option allows you to say that "instead of" using `https://github.com/` as the request URL for all Git requests, you'd prefer to use `ssh://git@github.com/`.

On Linux, MacOS, and WSL this configuration lives in the `.gitconfig` file. You may already be familiar with this file as it's also where your commit email address and name are configured as well. To edit the file, use `nano`, or your favorite text editor, and open the `~/.gitconfig` file in your home directory:

Copy

```
$ nano ~/.gitconfig
```

Once you have the file open, edit it to include a `url` section for `ssh://git@github.com/` as in the example below:

~/.gitconfig

```
[user]
        email = your_github_username@example.com
        name = Sammy the Shark

[url "ssh://git@github.com/"]
        insteadOf = https://github.com/
```

The order of the `url` section relative to the `user` section doesn't matter, and you also don't need to worry if there's nothing else in the file except for the `url` section you just added. The order of the `email` and `name` fields inside the `user` section also does not matter.

This new section tells Git that any URL you use that starts with `https://github.com/` should have that prefixed replaced with `ssh://git@github.com/` instead. Since Go uses HTTPS by default, this also affects your `go get` commands. Using your private module as an example, this means Go turns the `github.com/your_github_username/mysecret` import path into the URL `https://github.com/your_github_username/mysecret`. When Git encounters this URL it will see the URL matches the `https://github.com/` prefix referenced by `insteadOf` and will turn the resulting URL into `ssh://git@github.com/your_github_username/mysecret`.

This same pattern can be used for domains other than GitHub as long as the `ssh://git@` URL works for that host as well.

In this section, you configured Git to use SSH to download Go modules by updating your `.gitconfig` file and adding a `url` section. Now that authentication for your private module is set up, you can access it for use in your Go programs.

Using a Private Module

In the previous sections, you configured Go to access your private Go module via HTTPS, SSH, or possibly both. Now that Go can access your private module, it can be used similar to any public module you may have used in the past. In this section, you'll create a new Go module that uses your private module.

In the directory you use for your projects, such as `projects`, create a directory named `myproject` for the new project using the `mkdir` command:

```
$ mkdir myproject
```

Once the directory is created, go to the directory using `cd` and initialize a new Go module using `go mod init` for your project based on the repository URL your project would live at, such as `github.com/your_github_username/myproject`. If you don't plan for your project to be pushed to any other repository the module name could be just `myproject`, or any other name, but it's good practice to use full URLs since most modules being shared will need them.

```
$ cd myproject
$ go mod init github.com/your_github_username/myproject
```

```
Output
go: creating new go.mod: module github.com/your_github_username/myproject
```

Now, create your project's first code file by opening `main.go` with `nano`, or your favorite text editor:

```
$ nano main.go
```

Inside the file, set up the initial `main` function you will call your private module from:

projects/myproject/main.go

```
package main
```

```
import "fmt"

func main() {
        fmt.Println("My new project!")
}
```

To run your project now and make sure everything is set up correctly, you can use the `go run` command and provide it the `main.go` file:

Copy

```
$ go run main.go
```

```
Output
My new project!
```

Next, add your private module as a dependency of your new project using `go get`, similar to how you would for a public module:

Copy

```
$ go get github.com/your_github_username/mysecret
```

The `go` tool will then download your private module's code and add it as a dependency using a version string matching your latest commit hash and the time of that commit:

```
Output
go: downloading github.com/your_github_username/mysecret v0.0.0-20210920195630-bda059d63fa2
go get: added github.com/your_github_username/mysecret v0.0.0-20210920195630-bda059d63fa2
```

Finally, open the `main.go` file again and update it to add a call to your private module's `SecretProcess` function in the `main` function. You'll also need to update the `import` statement to add your `github.com/your_github_username/mysecret` private module as an import as well:

projects/myproject/main.go Copy

```
package main

import (
        "fmt"

        "github.com/your_github_username/mysecret"
)

func main() {
        fmt.Println("My new project!")
        mysecret.SecretProcess()
}
```

To see the final project running with your private module, use the `go run` command again while providing the `main.go` file as the parameter:

Copy

```
$ go run main.go
```

You will see the `My new project!` line from the original code, but now you'll also see a `Running the secret process!` line from your imported `mysecret` module as well:

```
Output
My new project!
Running the secret process!
```

In this section, you used `go init` to create a new Go module to access the private module you published earlier. Once you had the module created, you then used `go get` to download your private module as you would with a public Go module. Finally, you used `go run` to compile and run your Go program using the private module.

Conclusion

In this tutorial, you created and published a private Go module. You also set up both HTTPS and SSH authentication to access your private Go module. Finally, you used your private module in a new project.

For more information on Go modules, the Go project has a series of blog posts detailing how the Go tools interact with and understand modules. The Go project also has a very detailed and technical reference for Go modules in the Go Modules Reference.

In addition to the `GOPRIVATE` environment variable, more variables are available to use when working with private Go modules. They can be seen in detail in the Private Modules section of the Go Modules Reference.

If you're interested in exploring the `.netrc` file in more detail, the GNU website on `.netrc` includes a list of all the available keywords. The `git-config` documentation also includes more information about how the `insteadOf` configuration option you used works in addition to other options that are available.

This tutorial is also part of the DigitalOcean How to Code in Go series. The series covers a number of Go topics, from installing Go for the first time to how to use the language itself.

Next in series: How To Run Multiple Functions Concurrently in Go →

Tutorial Series: How To Code in Go

Go (or GoLang) is a modern programming language originally developed by Google that uses high-level syntax similar to scripting languages. It is popular for its minimal syntax and innovative handling of concurrency, as well as for the tools it provides for building native binaries on foreign platforms.

Development Go

Browse Series: 53 articles

About the authors

Kristin Davidson Author

Bit Transducer

Kristin is a life-long geek and enjoys digging into the lowest levels of computing. She also enjoys learning and tinkering with new technologies.

Rachel Lee Editor

Technical Editor

Still looking for an answer? Ask a question Search for more help

Was this helpful? Yes No

Comments

2 Comments

B I U S H1 H2 H3

Leave a comment...

This textbox defaults to using **Markdown** to format your answer.

You can type **!ref** in this text area to quickly search our full set of tutorials, documentation & marketplace offerings and insert the link!

Sign In or Sign Up to Comment

armandoOlamendiColin • January 5, 2022

Hi. Thanks. Do you have a tutorial about building a Golang Web API with the Digital Ocean App Platform and a private github repository?

From here (https://docs.digitalocean.com/products/app-platform/languages-frameworks/go/)I have seen that Go apps whose dependencies reside in private git repos may fail to build. I saw the link to Heroku’s documentation to overcome that, but it is not a detailed guide. Thanks.

Show replies Reply

Scott Stensland • October 14, 2022

Thanks ... gottcha for me was I was incorrectly using a : delimiter for my env var GOPRIVATE it started working after using comma delimiter as shown above ... cheers !!!

Reply

Popular Topics

AI/ML

Ubuntu

Linux Basics

JavaScript

Python

MySQL

Docker

Kubernetes

All tutorials →

Talk to an expert →

Congratulations on unlocking the whale ambience easter egg!

Click the whale button in the bottom left of your screen to toggle some ambient whale noises while you read.

Reset easter egg to be discovered again

Permanently dismiss and hide easter egg

Thank you to the Glacier Bay National Park & Preserve and Merrick079 for the sounds behind this easter egg.

Interested in whales, protecting them, and their connection to helping prevent climate change? We recommend checking out the Whale and Dolphin Conservation.

Become a contributor for community

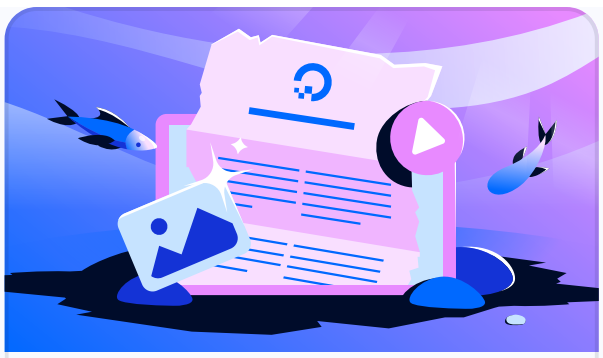
Get paid to write technical tutorials and select a tech-focused charity to receive a matching donation.

Sign Up →

DigitalOcean Documentation

Full documentation for every DigitalOcean product.

Learn more →



Resources for startups and SMBs

The Wave has everything you need to know about building a business, from raising funding to marketing your product.

Learn more →

Get our newsletter

Stay up to date by signing up for DigitalOcean's Infrastructure as a Newsletter.

Email address Submit

New accounts only. By submitting your email you agree to our Privacy Policy

The developer cloud

Scale up as you grow — whether you're running one virtual machine or ten thousand.

View all products

Get started for free

Sign up and get $200 in credit for your first 60 days with DigitalOcean.*

Get started

*This promotional offer applies to new accounts only.