CONTENTS

Tutorial Series: How To Code in Go

21/53 Importing Packages in Go

22/53 How To Write Packages i...

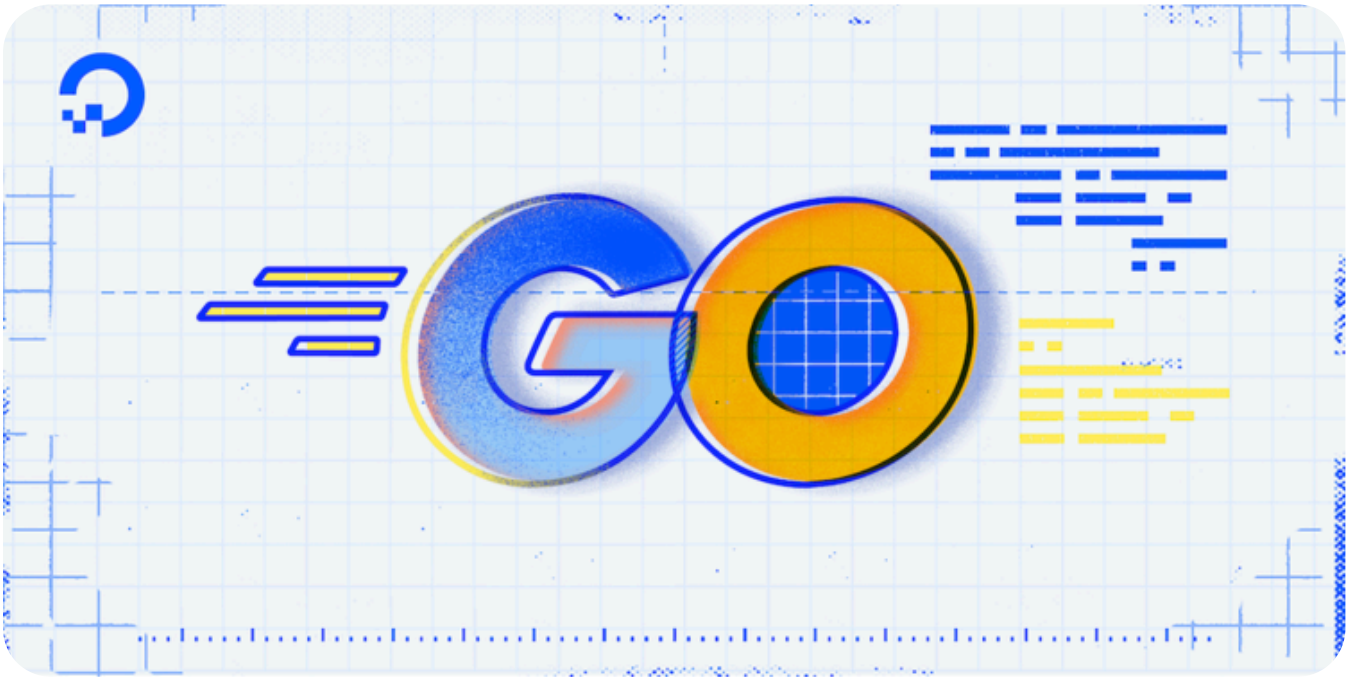
// TUTORIAL //

How to Use Go Modules

Published on October 27, 2021

Go Development

 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

In version 1.13, the authors of Go added a new way of managing the libraries a Go project depends on, called Go modules. Go modules were added in response to a growing need to make it easier for developers to maintain various versions of their dependencies, as well as add more flexibility in the way developers organize their projects on their computer. Go modules commonly consist of one project or library and contain a collection of Go packages that are then released together. Go modules solve many problems with `GOPATH`, the original system, by allowing users to put their project code in their chosen directory and specify versions of dependencies for each module.

In this tutorial, you will create your own public Go module and add a package to your new module. In addition, you will also add someone else's public module to your own project as well as add a specific version of that module to your project.

Prerequisites

To follow this tutorial, you will need:

- Go version 1.16 or greater installed. To set this up, follow the How To Install Go tutorial for your operating system.
- Familiarity with writing packages in Go. To learn more, follow the How To Write Packages in Go tutorial.

Creating a New Module

At first glance, a Go module looks similar to a Go package. A module has a number of Go code files implementing the functionality of a package, but it also has two additional and important files in the root: the `go.mod` file and the `go.sum` file. These files contain information the `go` tool uses to keep track of your module's configuration, and are commonly maintained by the tool so you don't need to.

The first thing to do is decide the directory the module will live in. With the introduction of Go modules, it became possible for Go projects to be located anywhere on the filesystem, not just a specific directory defined by Go. You may already have a directory for your projects, but in this tutorial, you'll create a directory called `projects` and the new module will be called `mymodule`. You can create the `projects` directory either through an IDE or via the command line.

If you're using the command line, begin by making the `projects` directory and navigating to it:

Next, you'll create the module directory itself. Usually, the module's top-level directory name is the same as the module name, which makes things easier to keep track of. In your `projects` directory, run the following command to create the `mymodule` directory:

```
$ mkdir mymodule
```

Once you've created the module directory, the directory structure will look like this:

```
└── projects
    └── mymodule
```

The next step is to create a `go.mod` file within the `mymodule` directory to define the Go module itself. To do this, you'll use the `go` tool's `mod init` command and provide it with the module's name, which in this case is `mymodule`. Now create the module by running `go mod init` from the `mymodule` directory and provide it with the module's name, `mymodule`:

```
$ go mod init mymodule
```

This command will return the following output when creating the module:

```
Output
go: creating new go.mod: module mymodule
```

With the module created, your directory structure will now look like this:

```
└── projects
    └── mymodule
        └── go.mod
```

Now that you have created a module, let's take a look inside the `go.mod` file to see what the `go mod init` command did.

Understanding the `go.mod` File

When you run commands with the `go` tool, the `go.mod` file is a very important part of the process. It's the file that contains the name of the module and versions of other modules your own module depends on. It can also contain other directives, such as `replace`, which can be helpful for doing development on multiple modules at once.

In the `mymodule` directory, open the `go.mod` file using `nano`, or your favorite text editor:

```
$ nano go.mod
```

The contents will look similar to this, which isn't much:

projects/mymodule/go.mod

```
module mymodule

go 1.16
```

The first line, the `module` directive, tells Go the name of your module so that when it's looking at `import` paths in a package, it knows not to look elsewhere for `mymodule`. The `mymodule` value comes from the parameter you passed to `go mod init`:

```
module mymodule
```

The only other line in the file at this point, the `go` directive, tells Go which version of the language the module is targeting. In this case, since the module was created using Go 1.16, the `go` directive says `1.16`:

```
go 1.16
```

As more information is added to the module, this file will expand, but it's a good idea to look at it now to see how it changes as dependencies are added further on.

You've now created a Go module with `go mod init` and looked at what an initial `go.mod` file contains, but your module doesn't do anything yet. It's time to take your module further and add some code.

Adding Go Code to Your Module

To ensure the module is created correctly and to add code so you can run your first Go module, you'll create a `main.go` file within the `mymodule` directory. The `main.go` file is commonly used in Go programs to signal the starting point of a program. The file's name isn't as important as the `main` function inside, but matching the two makes it easier to find. In this tutorial, the `main` function will print out `Hello, Modules!` when run.

To create the file, open the `main.go` file using `nano`, or your favorite text editor:

Copy

```
$ nano main.go
```

In the `main.go` file, add the following code to define your `main` package, import the `fmt` package, then print out the `Hello, Modules!` message in the `main` function:

projects/mymodule/main.go Copy

```
package main

import "fmt"

func main() {
        fmt.Println("Hello, Modules!")
}
```

In Go, each directory is considered its own package, and each file has its own `package` declaration line. In the `main.go` file you just created, the `package` is named `main`. Typically, you can name the package any way you'd like, but the `main` package is special in Go. When Go sees that a package is named `main` it knows the package should be considered a binary, and should be compiled into an executable file, instead of a library designed to be used in another program.

After the `package` is defined, the `import` declaration says to import the `fmt` package so you can use its `Println` function to print the `"Hello, Modules!` message to the screen.

Finally, the `main` function is defined. The `main` function is another special case in Go, related to the `main` package. When Go sees a function named `main` inside a package

named `main`, it knows the `main` function is the first function it should run. This is known as a program's entry point.

Once you have created the `main.go` file, the module's directory structure will look similar to this:

```
└── projects
    └── mymodule
        └── go.mod
        └── main.go
```

If you are familiar with using Go and the `GOPATH`, running code in a module is similar to how you would do it from a directory in the `GOPATH`. (Don't worry if you are not familiar with the `GOPATH`, because using modules replaces its usage.)

There are two common ways to run an executable program in Go: building a binary with `go build` or running a file with `go run`. In this tutorial, you'll use `go run` to run the module directly instead of building a binary, which would have to be run separately.

Run the `main.go` file you've created with `go run`:

Copy

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

Running the command will print the `Hello, Modules!` text as defined in the code:

```
Output
Hello, Modules!
```

In this section, you added a `main.go` file to your module with an initial `main` function that prints `Hello, Modules!`. At this point, your program doesn't yet benefit from being a Go module — it could be a file anywhere on your computer being run with `go run`. The first real benefit of Go modules is being able to add dependencies to your project in any directory and not just the `GOPATH` directory structure. You can also add packages to your module. In the next section, you will expand your module by creating an additional package within it.

Adding a Package to Your Module

Similar to a standard Go package, a module may contain any number of packages and sub-packages, or it may contain none at all. For this example, you'll create a package named `mypackage` inside the `mymodule` directory.

Create this new package by running the `mkdir` command inside the `mymodule` directory with the `mypackage` argument:

```
$ mkdir mypackage
```

This will create the new directory `mypackage` as a sub-package of the `mymodule` directory:

```
└── projects
    └── mymodule
        └── mypackage
        └── main.go
        └── go.mod
```

Use the `cd` command to change the directory to your new `mypackage` directory, and then use `nano`, or your favorite text editor, to create a `mypackage.go` file. This file could have any name, but using the same name as the package makes it easier to find the primary file for the package:

```
$ cd mypackage
$ nano mypackage.go
```

In the `mypackage.go` file, add a function called `PrintHello` that will print the message `Hello, Modules! This is mypackage speaking!` when called:

projects/mymodule/mypackage/mypackage.go

```
package mypackage

import "fmt"

fu  ntHello() {
```

```
        fmt.Println("Hello, Modules! This is mypackage speaking!")
}
```

Since you want the `PrintHello` function to be available from another package, the capital `P` in the function name is important. The capital letter means the function is exported and available to any outside program. For more information about how package visibility works in Go, Understanding Package Visibility in Go includes more detail.

Now that you've created the `mypackage` package with an exported function, you will need to `import` it from the `mymodule` package to use it. This is similar to how you would import other packages, such as the `fmt` package previously, except this time you'll include your module's name at the beginning of the import path. Open your `main.go` file from the `mymodule` directory and add a call to `PrintHello` by adding the highlighted lines below:

projects/mymodule/main.go Copy

```
package main

import (
        "fmt"

        "mymodule/mypackage"
)

func main() {
        fmt.Println("Hello, Modules!")

        mypackage.PrintHello()
}
```

If you take a closer look at the `import` statement, you'll see the new import begins with `mymodule`, which is the same module name you set in the `go.mod` file. This is followed by the path separator and the package you want to import, `mypackage` in this case:

Copy

```
"mymodule/mypackage"
```

In the future, if you add packages inside `mypackage`, you would also add them to the end of the import path in a similar way. For example, If you had another package called

`extrapackage` inside `mypackage`, your import path for that package would be `mymodule/mypackage/extrapackage`.

Run your updated module with `go run` and `main.go` from the `mymodule` directory as before:

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

Copy

When you run the module again you'll see both the `Hello, Modules!` message from earlier as well as the new message printed from your new `mypackage`'s `PrintHello` function:

```
Output
Hello, Modules!
Hello, Modules! This is mypackage speaking!
```

You've now added a new package to your initial module by creating a directory called `mypackage` with a `PrintHello` function. As your module's functionality expands, though, it can be useful to start using other peoples' modules in your own. In the next section, you'll add a remote module as a dependency to yours.

Adding a Remote Module as a Dependency

Go modules are distributed from version control repositories, commonly Git repositories. When you want to add a new module as a dependency to your own, you use the repository's path as a way to reference the module you'd like to use. When Go sees the import path for these modules, it can infer where to find it remotely based on this repository path.

For this example, you'll add a dependency on the `github.com/spf13/cobra` library to your module. Cobra is a popular library for creating console applications, but we won't address that in this tutorial.

Similar to when you created the `mymodule` module, you'll again use the `go` tool. However, this time, you'll run the `go get` command from the `mymodule` directory. Run `go get` and provide the module you'd like to add. In this case, you'll get `github.com/spf13/cobra`:

Copy

```
$ go get github.com/spf13/cobra
```

When you run this command, the `go` tool will look up the Cobra repository from the path you specified and determine which version of Cobra is the latest by looking at the repository's branches and tags. It will then download that version and keep track of the one it chose by adding the module name and the version to the `go.mod` file for future reference.

Now, open the `go.mod` file in the `mymodule` directory to see how the `go` tool updated the `go.mod` file when you added the new dependency. The example below could change depending on the current version of Cobra that's been released or the version of the Go tooling you're using, but the overall structure of the changes should be similar:

projects/mymodule/go.mod Copy

```
module mymodule

go 1.16

require (
        github.com/inconshreveable/mousetrap v1.0.0 // indirect
        github.com/spf13/cobra v1.2.1 // indirect
        github.com/spf13/pflag v1.0.5 // indirect
)
```

A new section using the `require` directive has been added. This directive tells Go which module you want, such as `github.com/spf13/cobra`, and the version of the module you added. Sometimes `require` directives will also include an `// indirect` comment. This comment says that, at the time the `require` directive was added, the module is not referenced directly in any of the module's source files. A few additional `require` lines were also added to the file. These lines are other modules Cobra depends on that the Go tool determined should be referenced as well.

You may have also noticed a new file, `go.sum`, was created in the `mymodule` directory after running the `go run` command. This is another important file for Go modules and contains information used by Go to record specific hashes and versions of dependencies. This ensures consistency of the dependencies, even if they are installed on a different machine.

Once you have the dependency downloaded you'll want to update your `main.go` file with some minimal Cobra code to use the new dependency. Update your `main.go` file in the `mymodule` directory with the Cobra code below to use the new dependency:

projects/mymodule/main.go Copy

```
package main

import (
        "fmt"

        "github.com/spf13/cobra"

        "mymodule/mypackage"
)

func main() {
        cmd := &cobra.Command{
                Run: func(cmd *cobra.Command, args []string) {
                        fmt.Println("Hello, Modules!")

                        mypackage.PrintHello()
                },
        }

        fmt.Println("Calling cmd.Execute()!")
        cmd.Execute()
}
```

This code creates a `cobra.Command` structure with a `Run` function containing your existing "Hello" statements, which will then be executed with a call to `cmd.Execute()`. Now, run the updated code:

Copy

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

You'll see the following output, which looks similar to what you saw before. This time, though, it's using your new dependency as shown by the `Calling cmd.Execute()!` line:

```
Output
Calling cmd.Execute()!
Hello, Modules!
Hello, Modules! This is mypackage speaking!
```

Using `go get` to add the latest version of a remote dependency, such as `github.com/sp13/cobra` here, makes it easier to keep your dependencies updated with the latest bug fixes. However, sometimes there may be times where you'd rather use a

specific version of a module, a repository tag, or a repository branch. In the next section, you'll use `go get` to reference these versions when you'd like that option.

Using a Specific Version of a Module

Since Go modules are distributed from a version control repository, they can use version control features such as tags, branches, and even commits. You can reference these in your dependencies using the `@` symbol at the end of the module path along with the version you'd like to use. Earlier, when you installed the latest version of Cobra, you were taking advantage of this capability, but you didn't need to add it explicitly to your command. The `go` tool knows that if a specific version isn't provided using `@`, it should use the special version `latest`. The `latest` version isn't actually in the repository, like `my-tag` or `my-branch` may be. It's built into the `go` tool as a helper so you don't need to search for the latest version yourself.

For example, when you added your dependency initially, you could have also used the following command for the same result:

```
$ go get github.com/spf13/cobra@latest
```

Now, imagine there's a module you use that's currently in development. For this example, call it `your_domain/sammy/awesome`. There's a new feature being added to this `awesome` module and work is being done in a branch called `new-feature`. To add this branch as a dependency of your own module you would provide `go get` with the module path, followed by the `@` symbol, followed by the name of the branch:

```
$ go get your_domain/sammy/awesome@new-feature
```

Running this command would cause `go` to connect to the `your_domain/sammy/awesome` repository, download the `new-feature` branch at the current latest commit for the branch, and add that information to the `go.mod` file.

Branches aren't the only way you can use the `@` option, though. This syntax can be used for tags and even specific commits to the repository. For example, sometimes the latest version of the library you're using may have a broken commit. In these cases, it can be useful to reference the commit just before the broken commit.

Using your module's Cobra dependency as an example, suppose you need to reference commit `07445ea` of `github.com/spf13/cobra` because it has some changes you need and you can't use another version for some reason. In this case, you can provide the commit hash after the `@` symbol the same as you would for a branch or a tag. Run the `go get` command in your `mymodule` directory with the module and version to download the new version:

```
$ go get github.com/spf13/cobra@07445ea
```

If you open your module's `go.mod` file again you'll see that `go get` has updated the `require` line for `github.com/spf13/cobra` to reference the commit you specified:

projects/mymodule/go.mod

```
module mymodule

go 1.16

require (
        github.com/inconshreveable/mousetrap v1.0.0 // indirect
        github.com/spf13/cobra v1.1.2-0.20210209210842-07445ea179fc // indir
        github.com/spf13/pflag v1.0.5 // indirect
)
```

Since a commit is a particular point in time, unlike a tag or a branch, Go includes additional information in the `require` directive to ensure it's using the correct version in the future. If you look closely at the version, you'll see it does include the commit hash you provided: `v1.1.2-0.20210209210842-07445ea179fc`.

Go modules also use this functionality to support releasing different versions of the module. When a Go module releases a new version, a new tag is added to the repository with the version number as the tag. If you want to use a specific version, you can look at a list of tags in the repository to find the version you're looking for. If you already know the version, you may not need to search through the tags because version tags are named consistently.

Returning to Cobra as an example, suppose you want to use Cobra version 1.1.1. You could look at the Cobra repository and see it has a tag named `v1.1.1`, among others. To use the tagged version, you would use the `@` symbol in a `go get` command, just as you would a non-version tag or branch. Now, update your module to use Cobra 1.1.1 by running the `go get` command with `v1.1.1` as the version:

Copy

```
$ go get github.com/spf13/cobra@v1.1.1
```

Now if you open your module's `go.mod` file, you'll see `go get` has updated the `require` line for `github.com/spf13/cobra` to reference the version you provided:

projects/mymodule/go.mod Copy

```
module mymodule

go 1.16

require (
        github.com/inconshreveable/mousetrap v1.0.0 // indirect
        github.com/spf13/cobra v1.1.1 // indirect
        github.com/spf13/pflag v1.0.5 // indirect
)
```

Finally, if you're using a specific version of a library, such as the `07445ea` commit or `v1.1.1` from earlier, but you determine you'd rather start using the latest version, it's possible to do this by using the special `latest` version. To update your module to the latest version of Cobra, run `go get` again with the module path and the `latest` version:

Copy

```
$ go get github.com/spf13/cobra@latest
```

Once this command finishes, the `go.mod` file will update to look like it did before you referenced a specific version of Cobra. Depending on your version of Go and the current latest version of Cobra your output may look slightly different, but you should still see that the `github.com/spf13/cobra` line in the `require` section is updated to the latest version again:

Copy

```
module mymodule

go 1.16
```

```
require (
        github.com/inconshreveable/mousetrap v1.0.0 // indirect
        github.com/spf13/cobra v1.2.1 // indirect
        github.com/spf13/pflag v1.0.5 // indirect
)
```

The `go get` command is a powerful tool you can use to manage dependencies in your `go.mod` file without needing to edit it manually. As you saw in this section, using the `@` character with a module name allows you to use particular versions for a module, from release versions to specific repository commits. It can even be used to go back to the `latest` version of your dependencies. Using a combination of these options will allow you to ensure the stability of your programs in the future.

Conclusion

In this tutorial, you created a Go module with a sub-package and used that package within your module. You also added another module to yours as a dependency and explored how to reference module versions in various ways.

For more information on Go modules, the Go project has a series of blog posts about 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.

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 Distribute Go Modules →

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 i[illegible]ive handling of concurrency, as well as for the tools it provides for building native binaries on foreign platforms.

Subscribe

Go Development

Browse Series: 53 articles

Collapse list

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

1 Comments

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

864c86e3312a4036b950751c0ced4c • December 22, 2022

Gr article

Reply

Popular Topics

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.

Become a contributor

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

Learn more →

Featured Tutorials

Kubernetes Course Learn Python 3 Machine Learning in Python

Getting started with Go Intro to Kubernetes

DigitalOcean Products

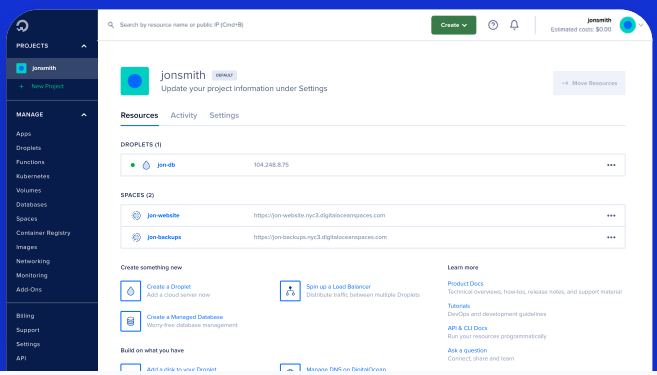
App Platform Virtual Machines Managed Databases Managed Kubernetes

Block Storage Object Storage Marketplace VPC Load Balancers

Welcome to the developer cloud

DigitalOcean makes it simple to launch in the cloud and scale up as you grow — whether you're running one virtual machine or ten thousand.

Learn more →



Company

Products

Community

Solutions

Contact