Tutorial Series: How To Code in Go

1/52 How To Code in Go eBook

2/52 How To Install Go and Set

// Tutorial //

How To Build and Install Go Programs

Published on November 7, 2019 · Updated on April 26, 2022

Go Development

By Gopher Guides
Developer and author at DigitalOcean.

English

Introduction

So far in our How To Code in Go series, you have used the command `go run` to automatically compile your source code and run the resulting executable. Although this command is useful for testing your code on the command line, distributing or deploying your application requires you to build your code into a shareable *binary executable*, or a single file containing machine byte code that can run your application. To do this, you can use the Go toolchain to *build* and *install* your program.

In Go, the process of translating source code into a binary executable is called *building*. Once this executable is built, it will contain not only your application, but also all the support code needed to execute the binary on the target platform. This means that a Go binary does not need system dependencies such as Go tooling to run on a new system. Putting these executables in an executable filepath on your own system will allow you to run the program from anywhere on your system. This is the same thing as *installing* the program onto your system.

In this tutorial, you will use the Go toolchain to run, build, and install a sample `Hello, World!` program, allowing you to use, distribute, and deploy future applications effectively.

Prerequisites

To follow the example in this article, you will need:

- A Go workspace set up by following How To Install Go and Set Up a Local Programming Environment.

Step 1 – Setting Up and Running the Go Binary

First, create an application to use as an example for demonstrating the Go toolchain. To do this, you will use the classic “Hello, World!” program from the How To Write Your First Program in Go tutorial.

Create a directory called `greeter` in your `src` directory:

```
$ mkdir greeter
```

Next, move into the newly created directory and create the `main.go` file in the text editor of your choice:

```
$ cd greeter
$ nano main.go
```

Once the file is open, add the following contents:

src/greeter/main.go

```go
package main

import "fmt"

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

When run, this program will print the phrase `Hello, World!` to the console, and then the program will exit successfully.

Save and exit the file.

To test the program, use the `go run` command, as you've done in previous tutorials:

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

You'll receive the following output:

```
Output
Hello, World!
```

As mentioned before, the `go run` command built your source file into an executable binary, and then ran the compiled program. However, this tutorial aims to build the binary in such a way that you can share and distribute it at will. To do this, you will use the `go build` command in the next step.

Step 2 – Creating a Go Module to Build a Go Binary

Go programs and libraries are built around the core concept of a module. A module contains information about the libraries that are used by your program and what versions of those libraries to use.

In order to tell Go that this is a Go module, you will need to create a Go module using the `go mod` command:

```
$ go mod init greeter
```

This will create the file `go.mod`, which will contain the name of the module and what version of Go was used to build it.

```
Output
go: creating new go.mod: module greeter
go: to add module requirements and sums:
        go mod tidy
```

Go will prompt you to run `go mod tidy` in order to update this module's requirements if they change in the future. Running it now will have no additional effect.

Step 3 – Building Go Binaries With `go build`

Using `go build`, you can generate an executable binary for our sample Go application, allowing you to distribute and deploy the program where you want.

Try this with `main.go`. In your `greeter` directory, run the following command:

```
$ go build
```

If you do not provide an argument to this command, `go build` will automatically compile the `main.go` program in your current directory. The command will include all your `*.go` files in the directory. It will also build all of the supporting code needed to be able to execute the binary on any computer with the same system architecture, regardless of whether that system has the `.go` source files, or even a Go installation.

In this case, you built your `greeter` application into an executable file that was added to your current directory. Check this by running the `ls` command:

```
$ ls
```

If you are running macOS or Linux, you will find a new executable file that has been named after the directory in which you built your program:

```
Output
greeter  main.go  go.mod
```

Note: On Windows, your executable will be `greeter.exe`.

By default `go build` will generate an executable for the current platform and architecture. For example, if built on a `linux/386` system, the executable will be compatible with any other `linux/386` system, even if Go is not installed. Go supports building for other platforms and architectures, which you can read more about in our Building Go Applications for Different Operating Systems and Architectures article.

Now, that you've created your executable, run it to make sure the binary has been built correctly. On macOS or Linux, run the following command:

```
$ ./greeter
```

On Windows, run:

```
$ greeter.exe
```

The output of the binary will match the output from when you ran the program with `go run`:

```
Output
Hello, World!
```

Now you have created a single executable binary that contains, not only your program, but also all of the system code needed to run that binary. You can now distribute this program to new systems or deploy it to a server, knowing that the file will always run the same program.

In the next section, this tutorial will explain how a binary is named and how you can change it, so that you can have better control over the build process of your program.

Step 4 – Changing the Binary Name

Now that you know how to generate an executable, the next step is to identify how Go chooses a name for the binary and to customize this name for your project.

When you run `go build`, the default is for Go to automatically decide on the name of the generated executable. It does this by using the module you created earlier. When the `go mod init greeter` command was run, it created the module with the name 'greeter', which is why the binary generated is named `greeter` in turn.

Let's take a closer look at the module method. If you had a `go.mod` file in your project with a `module` declaration such as the following:

go.mod

```
module github.com/sammy/shark
```

Then the default name for the generated executable would be `shark`.

In more complex programs that require specific naming conventions, these default values will not always be the best choice for naming your binary. In these cases, it would be best to customize your output with the `-o` flag.

To test this out, change the name of the executable you made in the last section to `hello` and have it placed in a sub-folder called `bin`. You don't have to create this folder; Go will do that on its own during the build process.

Run the following `go build` command with the `-o` flag:

```
$ go build -o bin/hello
```

The `-o` flag makes Go match the output of the command to whatever argument you chose. In this case, the result is a new executable named `hello` in a sub-folder named `bin`.

To test the new executable, change into the new directory and run the binary:

```
$ cd bin
$ ./hello
```

You will receive the following output:

```
Output
Hello, World!
```

You can now customize the name of your executable to fit the needs of your project, completing our survey of how to build binaries in Go. But with `go build`, you are still limited to running your binary from the current directory. In order to use newly built executables from anywhere on your system, you can install them using `go install`.

Step 5 – Installing Go Programs with `go install`

So far in this article, we have discussed how to generate executable binaries from our `.go` source files. These executables are helpful to distribute, deploy, and test, but they cannot yet be executed from outside of their source directories. This would be a problem if you wanted to actively use your program in shell scripts or in other workflows. To make the programs easier to use, you can install them into your system and access them from anywhere.

To understand what is meant by this, you will use the `go install` command to install your sample application.

The `go install` command behaves almost identically to `go build`, but instead of leaving the executable in the current directory, or a directory specified by the `-o` flag, it places the executable into the `$GOPATH/bin` directory.

To find where your `$GOPATH` directory is located, run the following command:

```
$ go env GOPATH
```

The output you receive will vary, but the default is the `go` directory inside of your `$HOME` directory:

```
Output
$HOME/go
```

Since `go install` will place generated executables into a sub-directory of `$GOPATH` named `bin`, this directory must be added to the `$PATH` environment variable. This is covered in the **Creating Your Go Workspace** step of the prerequisite article How To Install Go and Set Up a Local Programming Environment.

With the `$GOPATH/bin` directory set up, move back to your `greeter` directory:

```
$ cd ..
```

Now run the install command:

```
$ go install
```

This will build your binary and place it in `$GOPATH/bin`. To test this, run the following:

```
$ ls $GOPATH/bin
```

This will list the contents of `$GOPATH/bin`:

```
Output
greeter
```

Note: The `go install` command does **not** support the `-o` flag, so it will use the default name described earlier to name the executable.

With the binary installed, test to see if the program will run from outside its source directory. Move back to your home directory:

```
$ cd $HOME
```

Use the following to run the program:

```
$ greeter
```

This will yield the following:

```
Output
Hello, World!
```

Now you can take the programs you write and install them into your system, allowing you to use them from wherever, whenever you need them.

Conclusion

In this tutorial, you demonstrated how the Go toolchain makes it easy to build executable binaries from source code. These binaries can be distributed to run on other systems, even ones that do not have Go tooling and environments. You also used `go install` to automatically build and install our programs as executables in the system's `$PATH`. With `go build` and `go install`, you now have the ability to share and use your application at will.

Now that you know the basics of `go build`, you can explore how to make modular source code with the Customizing Go Binaries with Build Tags tutorial, or how to build for different platforms with Building Go Applications for Different Operating Systems and Architectures. If you'd like to learn more about the Go programming language in general, check out the entire How To Code in Go series.

Next in series: How To Use Struct Tags 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.

Subscribe

Go Development

Browse Series: 52 articles

Expand to view all

About the authors

Gopher Guides Author

Developer and author at DigitalOcean.

Still looking for an answer?

Ask a question

Search for more help

Was this helpful? Yes No

Comments

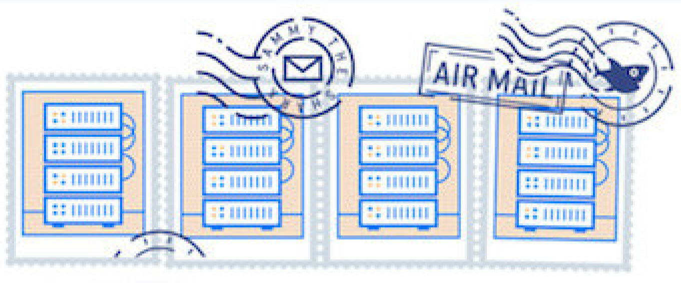
Leave a comment

Leave a comment...

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

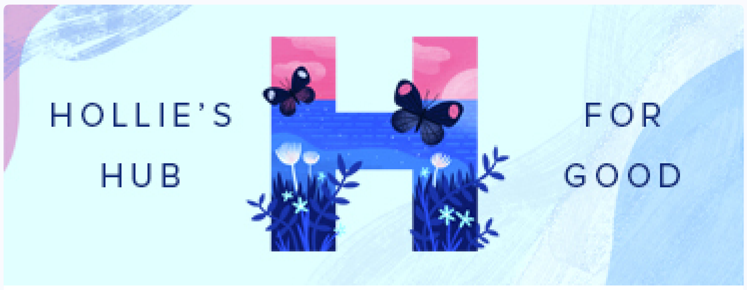
You can type `!ref` in this text area to quickly search our full set of tutorials, documentation & marketplace offerings and insert the link!

Sign In or Sign Up to Comment



GET OUR BIWEEKLY NEWSLETTER

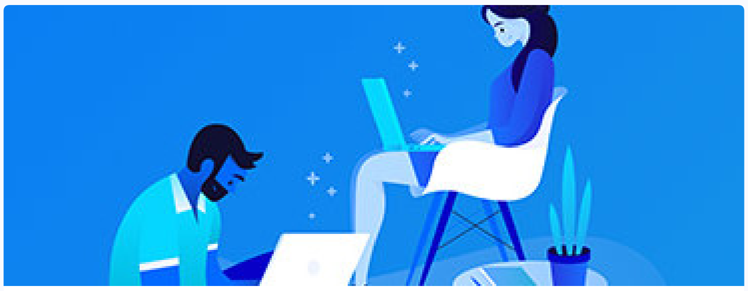
Sign up for Infrastructure as a Newsletter.



HOLLIE'S HUB FOR GOOD

Working on improving health and education, reducing inequality, and spurring economic growth?

We'd like to help.



BECOME A CONTRIBUTOR

You get paid; we donate to tech nonprofits.

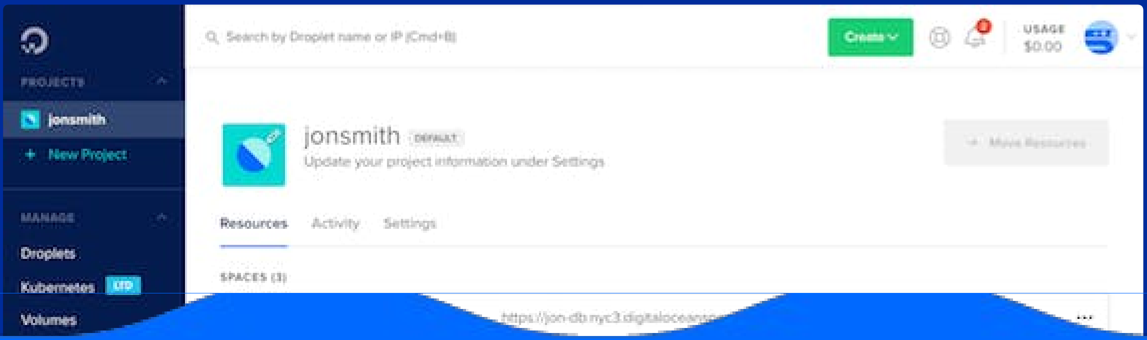
Featured on Community Kubernetes Course Learn Python 3 Machine Learning in Python Getting started with Go Intro to Kubernetes

DigitalOcean Products 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

About
Leadership
Blog
Careers
Customers
Partners
Channel Partners
Referral Program
Affiliate Program
Press
Legal
Security
Investor Relations
DO Impact

Products

Products Overview
Droplets
Kubernetes
App Platform
Functions
Cloudways
Managed Databases
Spaces
Marketplace
Load Balancers
Block Storage
Tools & Integrations
API
Pricing
Documentation
Release Notes
Uptime

Community

Tutorials
Q&A
CSS-Tricks
Write for DOnations
Currents Research
Hatch Startup Program
deploy by DigitalOcean
Shop Swag
Research Program
Open Source
Code of Conduct
Newsletter Signup
Meetups

Solutions

Website Hosting
VPS Hosting
Web & Mobile Apps
Game Development
Streaming
VPN
SaaS Platforms
Cloud Hosting for Blockchain
Startup Resources

Contact

Support
Sales
Report Abuse
System Status
Share your ideas