Mastering Go: A Comprehensive Guide to the Go Programming Language

This book provides an extensive guide through the Go Programming Language documentation, offering clear tutorials, in-depth modules, and efficient coding practices to enhance your programming skills in Go. Whether you're starting fresh or improving existing knowledge, this resource will guide you through the essentials and advanced topics.

Table of Contents

Getting Started with Go

- Installing Go
- Tutorial: Getting started
- Tutorial: Create a module
- Tutorial: Getting started with multi-module workspaces
- Tutorial: Developing a RESTful API with Go and Gin
- Tutorial: Getting started with generics
- Tutorial: Getting started with fuzzing

Developing Modules in Go

- Developing and publishing modules
- Module release and versioning workflow
- Managing module source
- Organizing a Go module
- Developing a major version update
- Publishing a module

Download and install

Download and install Go quickly with the steps described here.

For other content on installing, you might be interested in:

- Managing Go installations — How to install multiple versions and uninstall.
- Installing Go from source — How to check out the sources, build them on your own machine, and run them.

Download

Go installation

Select the tab for your computer's operating system below, then follow its installation instructions.

Linux

1. **Remove any previous Go installation** by deleting the `/usr/local/go` folder (if it exists), then extract the archive you just downloaded into `/usr/local`, creating a fresh Go tree in `/usr/local/go`:

```
$ rm -rf /usr/local/go && tar -C /usr/local -xzf go1.14.3.linux-amd64.tar.gz
```

You may need to run each command separately with the necessary permissions, as root or through `sudo`.

Do not untar the archive into an existing `/usr/local/go` tree. This is known to produce broken Go installations.

2. Add `/usr/local/go/bin` to the `PATH` environment variable:

- Add the following line to your `$HOME/.profile` or `/etc/profile` (for a system-wide installation):

```
export PATH=$PATH:/usr/local/go/bin
```

Changes made to a profile file may not apply until the next time you log into your computer. To apply the changes immediately, just run the shell commands directly or execute them from the profile using a command such as `source $HOME/.profile`.

3. Verify that you've installed Go by opening a command prompt and typing:

```
$ go version
```

Confirm that the command prints the installed version of Go.

Mac

Follow the instructions similar to Linux, with the appropriate package for macOS.

Windows

1. Open the MSI file you downloaded and follow the prompts to install Go.

By default, the installer will install Go to `Program Files` or `Program Files (x86)`. You can change the location as needed. After installing, close and reopen any open command prompts for environment changes to take effect.

2. Verify the installation:

- Click the **Start** menu.
- Type `cmd` in the search box and press **Enter**.
- In the Command Prompt window, type:

```
$ go version
```

And confirm that it displays the installed version of Go.

You're all set!

Visit the Getting Started tutorial to write some simple Go code. It takes about 10 minutes to complete.

Report Issues

If you spot bugs, mistakes, or inconsistencies in the Go project's code or documentation, please let us know by filing a ticket on our issue tracker. Of course, you should check it's not an existing issue before creating a new one.

Filing a ticket

Footer:

- Why Go
- Use Cases
- Case Studies
- Get Started
- Playground
- Tour
- Stack Overflow
- Help
- Packages
- Standard Library
- About Go Packages
- About
- Download
- Blog
- Issue Tracker
- Release Notes
- Brand Guidelines

- Code of Conduct
- Connect
- Twitter
- GitHub
- Slack
- Reddit
- Meetup
- Golang Weekly

Opens in new window.

Legal:

- Copyright
- Terms of Service
- Privacy Policy
- Report an Issue

Cookie Notice:

go.dev uses cookies from Google to deliver and enhance the quality of its services and to analyze traffic. Learn more.

Okay

Hello

Hello

Hello

Hello

Hello

Hello

```
console.log("Hello")
```

Hello

- Hello
- World

1. Hello
2. World

Hello World

```
console.log("Hello")
```

Tutorial: Create a Go module

This is the first part of a tutorial that introduces a few fundamental features of the Go language. If you're just getting started with Go, be sure to take a look at Tutorial: Get started with Go, which introduces the `go` command, Go modules, and very simple Go code.

In this tutorial you'll create two modules. The first is a library which is intended to be imported by other libraries or applications. The second is a caller application which will use the first.

This tutorial's sequence includes seven brief topics that each illustrate a different part of the language.

1. Create a module -- Write a small module with functions you can call from another module.
2. Call your code from another module -- Import and use your new module.
3. Return and handle an error -- Add simple error handling.
4. Return a random greeting -- Handle data in slices (Go's dynamically-sized arrays).
5. Return greetings for multiple people -- Store key/value pairs in a map.
6. Add a test -- Use Go's built-in unit testing features to test your code.
7. Compile and install the application -- Compile and install your code locally.

   Note: For other tutorials, see Tutorials.

Prerequisites

- **Some programming experience.** The code here is pretty simple, but it helps to know something about functions, loops, and arrays.
- **A tool to edit your code.** Any text editor you have will work fine. Most text editors have good support for Go. The most popular are VSCode (free), GoLand (paid), and Vim (free).
- **A command terminal.** Go works well using any terminal on Linux and Mac, and on PowerShell or cmd in Windows.

Start a module that others can use

Start by creating a Go module. In a module, you collect one or more related packages for a discrete and useful set of functions. For example, you might create a module with packages that have functions for doing financial analysis so that others writing financial applications can use your work. For more about developing modules, see Developing and publishing modules.

Go code is grouped into packages, and packages are grouped into modules. Your module specifies dependencies needed to run your code, including the Go version and the set of other modules it requires.

As you add or improve functionality in your module, you publish new versions of the module. Developers writing code that calls functions in your module can import the module's updated packages and test with the new version before putting it into production.

1. Open a command prompt and `cd` to your home directory.

   On Linux or Mac:

   ```
   cd
   ```

   On Windows:

   ```
   cd %HOMEPATH%
   ```

2. Create a `greetings` directory for your Go module source code.

From your home directory:

```
mkdir greetings
cd greetings
```

3. Start your module using the go mod init command.

   Run:

   ```
   $ go mod init example.com/greetings
   ```

   This creates a `go.mod` file and initializes your module with the path `example.com/greetings`. This path should be a URL where your module can be downloaded.

4. Create a file called `greetings.go` in your editor.

5. Paste the following code into `greetings.go`:

   ```
   package greetings

   import "fmt"

   // Hello returns a greeting for the named person.
   func Hello(name string) string {
       // Return a greeting that embeds the name in a message.
       message := fmt.Sprintf("Hi, %v. Welcome!", name)
       return message
   }
   ```

   This code:

   - Declares a `greetings` package.
   - Implements a `Hello` function that returns a greeting with the provided name.
   - Notes that `Hello` is an exported function (starts with a capital letter) and can be called from other packages.

6. In your terminal, you can now write code that calls this `Hello` function in other modules.

   Call your code from another module >

Tutorial: Getting started with multi-module workspaces

This tutorial introduces the basics of multi-module workspaces in Go.
With multi-module workspaces, you can tell the Go command that you're writing code in multiple modules at the same time and easily build and run code in those modules.

In this tutorial, you'll create two modules in a shared multi-module workspace, make changes across those modules, and see the results of those changes in a build.

Note: For other tutorials, see Tutorials.

Prerequisites

- **An installation of Go 1.18 or later.**
- **A tool to edit your code.** Any text editor you have will work fine.
- **A command terminal.** Go works well using any terminal on Linux and Mac, and on PowerShell or cmd in Windows.

This tutorial requires go1.18 or later. Make sure you've installed Go at Go 1.18 or later using the links at go.dev/dl.

Create a module for your code

To begin, create a module for the code you'll write.

1. Open a command prompt and change to your home directory.

   On Linux or Mac:

   ```
   $ cd
   ```

   On Windows:

   ```
   C:\> cd %HOMEPATH%
   ```

   The rest of the tutorial will show a `$` as the prompt. The commands you use will work on Windows too.

2. From the command prompt, create a directory for your code called `workspace`.

   ```
   $ mkdir workspace
   $ cd workspace
   ```

3. Initialize the module

   Our example will create a new module `hello` that will depend on the `golang.org/x/example` module.

   Create the hello module:

   ```
   $ mkdir hello
   $ cd hello
   $ go mod init example.com/hello
   go: creating new go.mod: module example.com/hello
   ```

Add a dependency on the `golang.org/x/example/hello/reverse` package by using `go get`.

```
$ go get golang.org/x/example/hello/reverse
```

Create `hello.go` in the `hello` directory with the following contents:

```
package main

import (
    "fmt"
    "golang.org/x/example/hello/reverse"
)

func main() {
    fmt.Println(reverse.String("Hello"))
}
```

Now, run the hello program:

```
$ go run .
olleH
```

Create the workspace

In this step, we'll create a `go.work` file to specify a workspace with the module.

Initialize the workspace

In the `workspace` directory, run:

```
$ go work init ./hello
```

The `go work init` command tells `go` to create a `go.work` file for a workspace containing the modules in the `./hello` directory.

The `go` command produces a `go.work` file that looks like this:

```
go 1.18

use ./hello
```

The `go.work` file has similar syntax to `go.mod`.

- The `go` directive specifies the version of Go the file is for.
- The `use` directive tells Go that the module in the `hello` directory should be a main module when doing a build.

So in any subdirectory of `workspace`, the module will be active.

Run the program in the workspace directory

In the `workspace` directory, run:

```
$ go run ./hello
olleH
```

The Go command includes all the modules in the workspace as main modules.
This allows us to refer to a package in the module, even outside the module.

Running `go run` outside the module or workspace would result in an error because `go` wouldn't know which modules to use.

Next, we'll add a local copy of the `golang.org/x/example/hello` module to the workspace.
That module is stored in a subdirectory of the `go.googlesource.com/example` Git repository.
We'll add a new function to the `reverse` package for use instead of `String`.

Download and modify the `golang.org/x/example/hello` module

In this step, we'll download a copy of the Git repo containing the `golang.org/x/example/hello` module, add it to the workspace, and then add a new function to it that we will use from the hello program.

1. **Clone the repository**

From the workspace directory, run:

```
$ git clone https://go.googlesource.com/example
Cloning into 'example'...
remote: Total 165 (delta 27), reused 165 (delta 27)
Receiving objects: 100% (165/165), 434.18 KiB | 1022.00 KiB/s, done.
Resolving deltas: 100% (27/27), done.
```

2. **Add the module to the workspace**

The Git repo is checked out into `./example`.
The source code for the `golang.org/x/example/hello` module is in `./example/hello`.
Add it to the workspace:

```
$ go work use ./example/hello
```

The `go work use` command adds a new module to the `go.work` file, which will now look like:

```
go 1.18

use (
    ./hello
    ./example/hello
)
```

The workspace now includes both the `example.com/hello` module and the `golang.org/x/example/hello` module, which provides the `golang.org/x/example/hello/reverse` package.

This will allow us to use the new code we will write in our copy of the `reverse` package instead of the version in the module cache downloaded with `go get`.

3. **Add the new function**

Create a new file named `int.go` in the `workspace/example/hello/reverse` directory with:

```
package reverse

import "strconv"

// Int returns the decimal reversal of the integer i.
func Int(i int) int {
    i, _ = strconv.Atoi(String(strconv.Itoa(i)))
```

```
    return i
}
```

4. **Modify the hello program to use the function**

Update `workspace/hello/hello.go` to:

```
package main

import (
    "fmt"
    "golang.org/x/example/hello/reverse"
)

func main() {
    fmt.Println(reverse.String("Hello"), reverse.Int(24601))
}
```

Run the code in the workspace

From the workspace directory, run:

```
$ go run ./hello
olleH 10642
```

The Go command finds the `example.com/hello` module specified in the command line in the `hello` directory, as well as resolving the import `golang.org/x/example/hello/reverse` using the `go.work` file.

`go.work` can be used instead of adding replace directives to work across multiple modules.

Since the two modules are in the same workspace, it's easy to make a change in one module and use it in another.

Future steps

To properly release these modules, you'd need to make a release of `golang.org/x/example/hello`, e.g., at `v0.1.0`.
This is usually done by tagging a commit in the module's version control repository.
See module release workflow documentation for details.
Once the release is done, you can update the requirement in `hello/go.mod`:

```
cd hello
go get golang.org/x/example/hello@v0.1.0
```

This allows `go` to resolve the modules outside the workspace correctly.

Learn more about workspaces

The `go` command has additional subcommands for working with workspaces beyond `go work init`:

- `go work use [-r] [dir]` adds or removes a directory from the workspace.
- `go work edit` edits the `go.work` file.
- `go work sync` synchronizes dependencies from the workspace's build list into each module.

See Workspaces in the Go Modules Reference for more details.

Tutorial: Developing a RESTful API with Go and Gin

This tutorial introduces the basics of writing a RESTful web service API with Go and the Gin Web Framework (Gin).
You'll get the most out of this tutorial if you have a basic familiarity with Go and its tooling. If this is your first exposure to Go, please see Tutorial: Get started with Go for a quick introduction.

Gin simplifies many coding tasks associated with building web applications, including web services. In this tutorial, you'll use Gin to route requests, retrieve request details, and marshal JSON for responses.

In this tutorial, you will build a RESTful API server with two endpoints. Your example project will be a repository of data about vintage jazz records.

The tutorial includes the following sections:

1. Design API endpoints.
2. Create a folder for your code.
3. Create the data.
4. Write a handler to return all items.
5. Write a handler to add a new item.
6. Write a handler to return a specific item.

Note: For other tutorials, see Tutorials.

To try this as an interactive tutorial you complete in Google Cloud Shell, click the button below.

Start Tutorial

Prerequisites

- **An installation of Go 1.16 or later.** For installation instructions, see Installing Go.
- **A tool to edit your code.** Any text editor will work fine.
- **A command terminal.** Go works well using any terminal on Linux, Mac, and on PowerShell or cmd in Windows.
- **The curl tool.** On Linux and Mac, this should already be installed. On Windows, it's included on Windows 10 Insider build 17063 and later. For earlier Windows versions, you might need to install it. See Tar and Curl Come to Windows.

Design API endpoints

You'll build an API that provides access to a store selling vintage recordings on vinyl. So you'll need to provide endpoints through which a client can get and add albums for users.

When developing an API, you typically begin by designing the endpoints. Your API's users will have more success if the endpoints are easy to understand.

Endpoints:

- `/albums`
  - `GET` – Get a list of all albums, returned as JSON.
  - `POST` – Add a new album from request data sent as JSON.

- `/albums/:id`
  - `GET` – Get an album by its ID, returning the album data as JSON.

Next, create a folder for your code.

Create a folder for your code

1. Open a command prompt and change to your home directory.

   On Linux or Mac:

   ```
   $ cd
   ```

   On Windows:

   ```
   C:\> cd %HOMEPATH%
   ```

2. Create a directory for your project:

   ```
   $ mkdir web-service-gin
   $ cd web-service-gin
   ```

3. Create a module to manage dependencies:

   ```
   $ go mod init example/web-service-gin
   ```

   This creates a `go.mod` file for tracking dependencies.

Create the data

For simplicity, store data in memory. A real API would interact with a database.

Note: storing data in memory means the data is lost when the server stops and recreated on startup.

Write the code

- Create a file called `main.go` in your project directory.
- Paste the following code:

```
package main

import (
    "net/http"

    "github.com/gin-gonic/gin"
)

// album represents data about a record album.
type album struct {
    ID     string  `json:"id"`
    Title  string  `json:"title"`
    Artist string  `json:"artist"`
    Price  float64 `json:"price"`
}

// albums slice to seed record album data.
var albums = []album{
```

```
    {ID: "1", Title: "Blue Train", Artist: "John Coltrane", Price: 56.99},
    {ID: "2", Title: "Jeru", Artist: "Gerry Mulligan", Price: 17.99},
    {ID: "3", Title: "Sarah Vaughan and Clifford Brown", Artist: "Sarah Vaughan", Pric
}
```

Write a handler to return all items

When a client makes a `GET /albums` request, return all albums as JSON.

Write the code

Paste the following below the `albums` slice:

```
// getAlbums responds with the list of all albums as JSON.
func getAlbums(c *gin.Context) {
    c.IndentedJSON(http.StatusOK, albums)
}
```

Set up the route

Add this in your `main()` function:

```
func main() {
    router := gin.Default()
    router.GET("/albums", getAlbums)

    router.Run("localhost:8080")
}
```

Import necessary packages

At the top of `main.go`, after `package main`, add:

```
import (
    "net/http"

    "github.com/gin-gonic/gin"
)
```

Run the server

1. Add Gin as a dependency:

```
$ go get github.com/gin-gonic/gin
```

2. Run the server:

```
$ go run .
```

3. Use curl to test:

```
$ curl http://localhost:8080/albums
```

Expected output:

```
[
    {
```

```
        "id": "1",
        "title": "Blue Train",
        "artist": "John Coltrane",
        "price": 56.99
    },
    {
        "id": "2",
        "title": "Jeru",
        "artist": "Gerry Mulligan",
        "price": 17.99
    },
    {
        "id": "3",
        "title": "Sarah Vaughan and Clifford Brown",
        "artist": "Sarah Vaughan",
        "price": 39.99
    }
]
```

Write a handler to add a new item

When the client makes a `POST /albums` request, add the album described in the request body.

Write the code

Add this function after `getAlbums`:

```
// postAlbums adds an album from JSON received in the request body.
func postAlbums(c *gin.Context) {
    var newAlbum album

    // Call BindJSON to bind the received JSON to newAlbum.
    if err := c.BindJSON(&newAlbum); err != nil {
        return
    }

    // Add the new album to the slice.
    albums = append(albums, newAlbum)
    c.IndentedJSON(http.StatusCreated, newAlbum)
}
```

Update main() to route POST requests

```
func main() {
    router := gin.Default()
    router.GET("/albums", getAlbums)
    router.POST("/albums", postAlbums)

    router.Run("localhost:8080")
}
```

Run and test

1. Restart server:

```
$ go run .
```

2. Make a POST request:

```
$ curl http://localhost:8080/albums \
    --include \
    --header "Content-Type: application/json" \
    --request "POST" \
    --data '{"id": "4","title": "The Modern Sound of Betty Carter","artist": "Betty Ca
```

3. Confirm addition with GET:

```
$ curl http://localhost:8080/albums \
    --header "Content-Type: application/json" \
    --request "GET"
```

Expected output includes the new album.

Write a handler to return a specific item

When a client makes a `GET /albums/:id` request, return the album with matching ID.

Write the code

Add this below the existing handlers:

```
// getAlbumByID locates the album whose ID value matches the id
// parameter sent by the client, then returns that album as a response.
func getAlbumByID(c *gin.Context) {
    id := c.Param("id")

    // Loop over the list of albums, looking for
    // an album whose ID value matches the parameter.
    for _, a := range albums {
        if a.ID == id {
            c.IndentedJSON(http.StatusOK, a)
            return
        }
    }
    c.IndentedJSON(http.StatusNotFound, gin.H{"message": "album not found"})
}
```

Update main() route

```
func main() {
    router := gin.Default()
    router.GET("/albums", getAlbums)
    router.GET("/albums/:id", getAlbumByID)
    router.POST("/albums", postAlbums)

    router.Run("localhost:8080")
}
```

Test

Start the server:

```
$ go run .
```

Make a request:

```
$ curl http://localhost:8080/albums/2
```

Expected response:

```
{
    "id": "2",
    "title": "Jeru",
    "artist": "Gerry Mulligan",
    "price": 17.99
}
```

Conclusion

Congratulations! You've just used Go and Gin to write a simple RESTful web service.

Next topics:

- See Effective Go and How to write Go code.
- Try Go Tour.
- Explore Gin Web Framework documentation and Gin docs.

Complete code

```
package main

import (
    "net/http"

    "github.com/gin-gonic/gin"
)

// album represents data about a record album.
type album struct {
    ID     string  `json:"id"`
    Title  string  `json:"title"`
    Artist string  `json:"artist"`
    Price  float64 `json:"price"`
}

// albums slice to seed record album data.
var albums = []album{
    {ID: "1", Title: "Blue Train", Artist: "John Coltrane", Price: 56.99},
    {ID: "2", Title: "Jeru", Artist: "Gerry Mulligan", Price: 17.99},
    {ID: "3", Title: "Sarah Vaughan and Clifford Brown", Artist: "Sarah Vaughan", Pric
}

func main() {
    router := gin.Default()
    router.GET("/albums", getAlbums)
    router.GET("/albums/:id", getAlbumByID)
    router.POST("/albums", postAlbums)
```

```
    router.Run("localhost:8080")
}

// getAlbums responds with the list of all albums as JSON.
func getAlbums(c *gin.Context) {
    c.IndentedJSON(http.StatusOK, albums)
}

// postAlbums adds an album from JSON received in the request body.
func postAlbums(c *gin.Context) {
    var newAlbum album

    // Call BindJSON to bind the received JSON to newAlbum.
    if err := c.BindJSON(&newAlbum); err != nil {
        return
    }

    // Add the new album to the slice.
    albums = append(albums, newAlbum)
    c.IndentedJSON(http.StatusCreated, newAlbum)
}

// getAlbumByID locates the album whose ID value matches the id
// parameter sent by the client, then returns that album as a response.
func getAlbumByID(c *gin.Context) {
    id := c.Param("id")

    // Loop through the list of albums
    for _, a := range albums {
        if a.ID == id {
            c.IndentedJSON(http.StatusOK, a)
            return
        }
    }
    c.IndentedJSON(http.StatusNotFound, gin.H{"message": "album not found"})
}
```

Tutorial: Getting started with generics - The Go Programming Language

---

Introduction

This tutorial introduces the basics of generics in Go. With generics, you can declare and use functions or types that are written to work with any of a set of types provided by calling code.

In this tutorial, you'll declare two simple non-generic functions, then capture the same logic in a single generic function.

You'll progress through the following sections:

1. Create a folder for your code.
2. Add non-generic functions.
3. Add a generic function to handle multiple types.
4. Remove type arguments when calling the generic function.
5. Declare a type constraint.

Note: For other tutorials, see Tutorials.

Note: If you prefer, you can use the Go playground in "Go dev branch" mode to edit and run your program instead.

Prerequisites

- **An installation of Go 1.18 or later.** For installation instructions, see Installing Go.
- **A tool to edit your code.** Any text editor will work fine.
- **A command terminal.** Go works well using any terminal on Linux and Mac, and on PowerShell or cmd in Windows.

Create a folder for your code

To begin, create a folder for the code you'll write.

1. Open a command prompt and change to your home directory.
    - On Linux or Mac:

      ```
      $ cd
      ```

    - On Windows:

      ```
      C:\> cd %HOMEPATH%
      ```

    The rest of the tutorial will show a `$` as the prompt. The commands you use will work on Windows too.

2. From the command prompt, create a directory called `generics` and navigate into it:

```
$ mkdir generics
$ cd generics
```

3. Create a module to hold your code:

```
$ go mod init example/generics
```

Output:

```
go: creating new go.mod: module example/generics
```

Note: For production code, specify a module path that's more specific to your needs. For more, see Managing dependencies.

Add non-generic functions

In this step, you'll add two functions that each add together the values of a map and return the total.

You're declaring two functions instead of one because you're working with two different types of maps: one that stores `int64` values, and one that stores `float64` values.

Write the code

1. Using your text editor, create a file called `main.go` in the `generics` directory and write your Go code here.

2. Paste the following package declaration at the top:

```
package main
```

A standalone program (as opposed to a library) is always in package `main`.

3. Beneath the package declaration, paste the following two function declarations:

```
// SumInts adds together the values of m.
func SumInts(m map[string]int64) int64 {
    var s int64
    for _, v := range m {
        s += v
    }
    return s
}

// SumFloats adds together the values of m.
func SumFloats(m map[string]float64) float64 {
    var s float64
    for _, v := range m {
        s += v
    }
    return s
}
```

In this code:

- Declare two functions to add together the values of a map and return the sum.

- `SumFloats` takes a map with `string` keys and `float64` values.
- `SumInts` takes a map with `string` keys and `int64` values.

4. Add a `main` function to initialize the maps and call these functions:

```
func main() {
    // Initialize a map for the integer values
    ints := map[string]int64{
        "first":  34,
        "second": 12,
    }

    // Initialize a map for the float values
    floats := map[string]float64{
        "first":  35.98,
        "second": 26.99,
    }

    fmt.Printf("Non-Generic Sums: %v and %v\n",
        SumInts(ints),
        SumFloats(floats))
}
```

5. Import the `fmt` package at the top:

```
package main

import "fmt"
```

6. Save `main.go`.

Run the code

In the directory containing `main.go`, execute:

```
$ go run .
```

Output:

```
Non-Generic Sums: 46 and 62.97
```

Next: You'll write a single generic function to replace these two.

Add a generic function to handle multiple types

In this section, you'll add a single generic function that can receive a map containing either integer or float values, replacing the two functions you just wrote.

The concept

- Support for values of either type will be achieved using type parameters and constraints.
- The function will declare `<em>type parameters</em>`, making it generic.

- You'll specify `<em>type constraints</em>` to limit permissible types.

Write the code

1. Paste the following generic function beneath the existing functions:

   ```
   // SumIntsOrFloats sums the values of map m. It supports both int64 and float64
   // as types for map values.
   func SumIntsOrFloats[K comparable, V int64 | float64](m map[K]V) V {
       var s V
       for _, v := range m {
           s += v
       }
       return s
   }
   ```

   Details:

   - It has type parameters `K` (key type) constrained by `comparable` and `V` (value type) constrained to `int64 | float64`.
   - It sums the map values, supporting both types.

2. Update your `main.go` to call the generic function:

   ```
   fmt.Printf("Generic Sums: %v and %v\n",
       SumIntsOrFloats[string, int64](ints),
       SumIntsOrFloats[string, float64](floats))
   ```

   Note: Often the type arguments can be omitted as the compiler infers them.

3. Run:

   ```
   $ go run .
   ```

   Output:

   ```
   Non-Generic Sums: 46 and 62.97
   Generic Sums: 46 and 62.97
   ```

Remove type arguments when calling the generic function

In many cases, you can omit specifying type arguments and rely on inference:

```
fmt.Printf("Generic Sums, type parameters inferred: %v and %v\n",
    SumIntsOrFloats(ints),
    SumIntsOrFloats(floats))
```

Run:

```
$ go run .
```

Output:

```
Non-Generic Sums: 46 and 62.97
Generic Sums: 46 and 62.97
```

```
Generic Sums, type parameters inferred: 46 and 62.97
```

Declare a type constraint

To make constraints reusable, define an interface:

```
type Number interface {
    int64 | float64
}
```

Implementing the constraint and a new function

Paste above `main`:

```
type Number interface {
    int64 | float64
}
```

Paste the new generic function below existing ones:

```
// SumNumbers sums the values of map m. Supports both integers and floats.
func SumNumbers[K comparable, V Number](m map[K]V) V {
    var s V
    for _, v := range m {
        s += v
    }
    return s
}
```

Use the constraint in main

```
fmt.Printf("Generic Sums with Constraint: %v and %v\n",
    SumNumbers[int, int64](ints),
    SumNumbers[string, float64](floats))
```

Type arguments can often be omitted because the compiler infers them.

Run:

```
$ go run .
```

Output:

```
Non-Generic Sums: 46 and 62.97
Generic Sums: 46 and 62.97
Generic Sums, type parameters inferred: 46 and 62.97
Generic Sums with Constraint: 46 and 62.97
```

Conclusion

Congratulations! You've just introduced yourself to generics in Go.

Next topics:

- The Go Tour offers a step-by-step introduction to Go fundamentals.
- See Effective Go and How to write Go code for best practices.

Completed code

You can run this program in the Go playground. Simply click the **Run** button.

```
package main

import "fmt"

type Number interface {
    int64 | float64
}

func main() {
    // Initialize a map for the integer values
    ints := map[string]int64{
        "first": 34,
        "second": 12,
    }

    // Initialize a map for the float values
    floats := map[string]float64{
        "first": 35.98,
        "second": 26.99,
    }

    fmt.Printf("Non-Generic Sums: %v and %v\n",
        SumInts(ints),
        SumFloats(floats))

    fmt.Printf("Generic Sums: %v and %v\n",
        SumIntsOrFloats[string, int64](ints),
        SumIntsOrFloats[string, float64](floats))

    fmt.Printf("Generic Sums, type parameters inferred: %v and %v\n",
        SumIntsOrFloats(ints),
        SumIntsOrFloats(floats))

    fmt.Printf("Generic Sums with Constraint: %v and %v\n",
        SumNumbers(ints),
        SumNumbers(floats))
}

// SumInts adds together the values of m.
func SumInts(m map[string]int64) int64 {
    var s int64
    for _, v := range m {
        s += v
    }
    return s
}
```

```
// SumFloats adds together the values of m.
func SumFloats(m map[string]float64) float64 {
    var s float64
    for _, v := range m {
        s += v
    }
    return s
}

// SumIntsOrFloats sums the values of map m. It supports both floats and integers as m
func SumIntsOrFloats[K comparable, V int64 | float64](m map[K]V) V {
    var s V
    for _, v := range m {
        s += v
    }
    return s
}

// SumNumbers sums the values of map m. It supports both integers and floats as map va
func SumNumbers[K comparable, V Number](m map[K]V) V {
    var s V
    for _, v := range m {
        s += v
    }
    return s
}
```

Tutorial: Getting started with fuzzing

Overview

This tutorial introduces the basics of fuzzing in Go. With fuzzing, random data is run against your test in an attempt to find vulnerabilities or crash-causing inputs. Some examples of vulnerabilities that can be found by fuzzing are SQL injection, buffer overflow, denial of service, and cross-site scripting attacks.

In this tutorial, you'll write a fuzz test for a simple function, run the `go` command, and debug and fix issues in the code.

For help with terminology throughout this tutorial, see the Go Fuzzing glossary.

You'll progress through the following sections:

1. Create a folder for your code
2. Add code to test
3. Add a unit test
4. Add a fuzz test
5. Fix two bugs
6. Explore additional resources

Note: For other tutorials, see Tutorials.

Note: Go fuzzing currently supports a subset of built-in types, listed in the Go Fuzzing docs, with support for more built-in types to be added in the future.

Prerequisites

- **Go 1.18 or later.** For installation instructions, see Installing Go.
- **A tool to edit your code.** Any text editor will work.
- **A command terminal.** Go works well using any terminal on Linux or Mac, and PowerShell or cmd in Windows.
- **An environment that supports fuzzing.** Go fuzzing with coverage instrumentation is only available on AMD64 and ARM64 architectures.

Create a folder for your code

To begin, create a folder for the code you'll write:

1. Open a command prompt and change to your home directory.
    - On Linux or Mac:

      ```
      $ cd
      ```

    - On Windows:

      ```
      C:\> cd %HOMEPATH%
      ```

The rest of the tutorial will show a `$` as the prompt. The commands you use will work on Windows too.

2. From the command prompt, create a directory called `fuzz`:

```
$ mkdir fuzz
$ cd fuzz
```

3. Create a module for your code:

```
$ go mod init example/fuzz
```

Output:

```
go: creating new go.mod: module example/fuzz
```

Note: For production code, specify a more specific module path. See Managing dependencies.

Next, add some simple code to reverse a string, which we'll fuzz later.

Add code to test

In this step, you'll add a function to reverse a string.

Write the code

1. Create a file called `main.go` in the `fuzz` directory.

2. Paste the following package declaration at the top:

```
package main
```

3. Paste the following `Reverse` function:

```
func Reverse(s string) string {
    b := []byte(s)
    for i, j := 0, len(b)-1; i < len(b)/2; i, j = i+1, j-1 {
        b[i], b[j] = b[j], b[i]
    }
    return string(b)
}
```

This function accepts a `string`, loops over it byte-by-byte, and returns the reversed string.

4. Add a `main` function to demonstrate usage:

```
func main() {
    input := "The quick brown fox jumped over the lazy dog"
    rev := Reverse(input)
    doubleRev := Reverse(rev)
    fmt.Printf("original: %q\n", input)
    fmt.Printf("reversed: %q\n", rev)
    fmt.Printf("reversed again: %q\n", doubleRev)
}
```

5. Import the `fmt` package at the top:

```
import "fmt"
```

Run the code

In the directory containing `main.go`, execute:

```
$ go run .
```

Output:

```
original: "The quick brown fox jumped over the lazy dog"
reversed: "god yzal eht revo depmuj xof nworb kciuq ehT"
reversed again: "The quick brown fox jumped over the lazy dog"
```

Add a unit test

Now, write a basic unit test for the `Reverse` function.

Write the code

1. Create a file called `reverse_test.go`.
2. Paste the following code:

```
package main

import (
    "testing"
)

func TestReverse(t *testing.T) {
    testcases := []struct {
        in, want string
    }{
        {"Hello, world", "dlrow ,olleH"},
        {" ", " "},
        {"!12345", "54321!"},
    }

    for _, tc := range testcases {
        rev := Reverse(tc.in)
        if rev != tc.want {
            t.Errorf("Reverse(%q): got %q, want %q", tc.in, rev, tc.want)
        }
    }
}
```

Run the test

```
$ go test
PASS
ok      example/fuzz  0.013s
```

Add a fuzz test

The unit test has limitations: each input must be added manually. Fuzzing can generate inputs automatically and find edge cases.

Convert the unit test to a fuzz test

1. In `reverse_test.go`, replace the contents with:

```
package main

import (
    "testing"
    "unicode/utf8"
)

func FuzzReverse(f *testing.F) {
    testcases := []string{"Hello, world", " ", "!12345"}
    for _, tc := range testcases {
        f.Add(tc)
    }
    f.Fuzz(func(t *testing.T, orig string) {
        rev := Reverse(orig)
        doubleRev := Reverse(rev)
        if orig != doubleRev {
            t.Errorf("Before: %q, after: %q", orig, doubleRev)
        }
        if utf8.ValidString(orig) && !utf8.ValidString(rev) {
            t.Errorf("Reverse produced invalid UTF-8 string %q", rev)
        }
    })
}
```

Run the fuzz test

1. Run without fuzzing to verify seed inputs:

```
$ go test
PASS
ok      example/fuzz  0.013s
```

2. Run with fuzzing:

```
$ go test -fuzz=FuzzReverse
```

This starts fuzzing, potentially discovering bugs. To limit fuzzing time:

```
$ go test -fuzz=FuzzReverse -fuzztime=10s
```

Fix the invalid string error

Fuzzing may uncover inputs that cause invalid UTF-8 strings.

Diagnose the error

Reversing a string byte-by-byte can corrupt multi-byte characters such as `汚`. To fix this, reverse rune-by-rune.

Update the code

Replace `Reverse` with:

```
func Reverse(s string) string {
    r := []rune(s)
    for i, j := 0, len(r)-1; i < len(r)/2; i, j = i+1, j-1 {
        r[i], r[j] = r[j], r[i]
    }
    return string(r)
}
```

Now, test again:

```
$ go test
PASS
ok      example/fuzz  0.016s
```

Fix the double reverse error

Fuzzing with non-UTF-8 characters can cause issues. To handle this gracefully, modify `Reverse` to return errors:

```
func Reverse(s string) (string, error) {
    if !utf8.ValidString(s) {
        return s, errors.New("input is not valid UTF-8")
    }
    r := []rune(s)
    for i, j := 0, len(r)-1; i < len(r)/2; i, j = i+1, j-1 {
        r[i], r[j] = r[j], r[i]
    }
    return string(r), nil
}
```

Update the code accordingly, and modify calls to `Reverse` to handle errors.

Summary

You have learned how to:

- Write initial code and tests
- Convert unit tests to fuzz tests
- Run fuzzing with `go test -fuzz`
- Debug and fix issues related to string encoding and reversal

Fuzzing is a powerful tool to discover edge cases and vulnerabilities in your code.

Additional Resources

- Go dev documentation on fuzzing
- Strings, bytes, runes, and characters in Go
- Managing dependencies

Complete code

main.go

```
package main

import (
    "errors"
    "fmt"
    "unicode/utf8"
)

func main() {
    input := "The quick brown fox jumped over the lazy dog"
    rev, revErr := Reverse(input)
    doubleRev, doubleRevErr := Reverse(rev)
    fmt.Printf("original: %q\n", input)
    fmt.Printf("reversed: %q, err: %v\n", rev, revErr)
    fmt.Printf("reversed again: %q, err: %v\n", doubleRev, doubleRevErr)
}

func Reverse(s string) (string, error) {
    if !utf8.ValidString(s) {
        return s, errors.New("input is not valid UTF-8")
    }
    r := []rune(s)
    for i, j := 0, len(r)-1; i < len(r)/2; i, j = i+1, j-1 {
        r[i], r[j] = r[j], r[i]
    }
    return string(r), nil
}
```

reverse_test.go

```
package main

import (
    "testing"
    "unicode/utf8"
)

func FuzzReverse(f *testing.F) {
    testcases := []string{"Hello, world", " ", "!12345"}
    for _, tc := range testcases {
        f.Add(tc)
    }
    f.Fuzz(func(t *testing.T, orig string) {
        rev, err1 := Reverse(orig)
        if err1 != nil {
            return
        }
        doubleRev, err2 := Reverse(rev)
        if err2 != nil {
            return
```

```
        }
        if orig != doubleRev {
            t.Errorf("Before: %q, after: %q", orig, doubleRev)
        }
        if utf8.ValidString(orig) && !utf8.ValidString(rev) {
            t.Errorf("Reverse produced invalid UTF-8 string %q", rev)
        }
    })
}
```

Developing and publishing modules

You can collect related packages into modules, then publish the modules for other developers to use. This topic gives an overview of developing and publishing modules.

To support developing, publishing, and using modules, you use:

- A **workflow** through which you develop and publish modules, revising them with new versions over time. See Workflow for developing and publishing modules.
- **Design practices** that help a module's users understand it and upgrade to new versions in a stable way. See Design and development.
- A **decentralized system for publishing** modules and retrieving their code. You make your module available for other developers to use from your own repository and publish with a version number. See Decentralized publishing.
- A **package search engine** and documentation browser (pkg.go.dev) at which developers can find your module. See Package discovery.
- A module **version numbering convention** to communicate expectations of stability and backward compatibility to developers using your module. See Versioning.
- **Go tools** that make it easier for other developers to manage dependencies, including getting your module's source, upgrading, and so on. See Managing dependencies.

See also

- If you're interested simply in using packages developed by others, this isn't the topic for you. Instead, see Managing dependencies.
- For a tutorial that includes a few module development basics, see Tutorial: Create a Go module.

Workflow for developing and publishing modules

When you want to publish your modules for others, you adopt a few conventions to make using those modules easier.

The following high-level steps are described in more detail in Module release and versioning workflow:

1. Design and code the packages that the module will include.
2. Commit code to your repository using conventions that ensure it's available to others via Go tools.
3. Publish the module to make it discoverable by developers.
4. Over time, revise the module with versions that use a version numbering convention that signals each version's stability and backward compatibility.

Design and development

Your module will be easier for developers to find and use if the functions and packages in it form a coherent whole. When you're designing a module's public API, try to keep its functionality focused and discrete.

Also, designing and developing your module with backward compatibility in mind helps its users upgrade while minimizing churn to their own code. You can use certain techniques in code to avoid releasing a version that breaks backward compatibility. For more about those techniques, see Keeping your modules compatible on the Go blog.

Before you publish a module, you can reference it on the local file system using the replace directive. This makes it easier to write client code that calls functions in the module while the module is still in development.

For more information, see “Coding against an unpublished module” in Module release and versioning workflow.

Decentralized publishing

In Go, you publish your module by tagging its code in your repository to make it available for other developers to use. You don’t need to push your module to a centralized service because Go tools can download your module directly from your repository (located using the module’s path, which is a URL with the scheme omitted) or from a proxy server.

After importing your package in their code, developers use Go tools (including the `go get` command) to download your module’s code to compile with. To support this model, you follow conventions and best practices that make it possible for Go tools (on behalf of another developer) to retrieve your module’s source from your repository. For example, Go tools use the module’s module path you specify, along with the module version number you use to tag the module for release, to locate and download the module for its users.

For more about source and publishing conventions and best practices, see Managing module source.

For step-by-step instructions on publishing a module, see Publishing a module.

Package discovery

After you’ve published your module and someone has fetched it with Go tools, it will become visible on the Go package discovery site at pkg.go.dev. There, developers can search the site to find it and read its documentation.

To begin using the module, a developer imports packages from the module, then runs the `go get` command to download its source code to compile with.

For more about how developers find and use modules, see Managing dependencies.

Versioning

As you revise and improve your module over time, you assign version numbers (based on the semantic versioning model) designed to signal each version’s stability and backward compatibility. This helps developers using your module determine when the module is stable and whether an upgrade may include significant changes in behavior. You indicate a module’s version number by tagging the module’s source in the repository with the number.

For more on developing major version updates, see Developing a major version update.

For more about how you use the semantic versioning model for Go modules, see Module version numbering.

Module release and versioning workflow

When you develop modules for use by other developers, you can follow a workflow that helps ensure a reliable, consistent experience for developers using the module. This topic describes the high-level steps in that workflow.

For an overview of module development, see Developing and publishing modules.

See also

- If you're merely wanting to use external packages in your code, be sure to see Managing dependencies.
- With each new version, you signal the changes to your module with its version number. For more, see Module version numbering.

Common workflow steps

The following sequence illustrates release and versioning workflow steps for an example new module. For more about each step, see the sections in this topic.

1. **Begin a module** and organize its sources to make it easier for developers to use and for you to maintain.

   If you're brand new to developing modules, check out Tutorial: Create a Go module.

   In Go's decentralized module publishing system, how you organize your code matters. For more, see Managing module source.

2. Set up to **write local client code** that calls functions in the unpublished module.

   Before you publish a module, it's unavailable for the typical dependency management workflow using commands such as `go get`. A good way to test your module code at this stage is to try it while it is in a directory local to your calling code.

   See Coding against an unpublished module for more about local development.

3. When the module's code is ready for other developers to try it out, **begin publishing v0 pre-releases** such as alphas and betas. See Publishing pre-release versions for more.

4. **Release a v0** that's not guaranteed to be stable, but which users can try out. For more, see Publishing the first (unstable) version.

5. After your v0 version is published, you can (and should!) continue to **release new versions** of it.

   These new versions might include bug fixes (patch releases), additions to the module's public API (minor releases), and even breaking changes. Because a v0 release makes no guarantees of stability or backward compatibility, you can make breaking changes in its versions.

   For more, see Publishing bug fixes and Publishing non-breaking API changes.

6. When you're getting a stable version ready for release, you **publish pre-releases as alphas and betas**. For more, see Publishing pre-release versions.

7. Release a v1 as the **first stable release**.

This is the first release that makes commitments about the module's stability. For more, see Publishing the first stable version.

8. In the v1 version, **continue to fix bugs** and, where necessary, make additions to the module's public API.

   For more, see Publishing bug fixes and Publishing non-breaking API changes.

9. When it can't be avoided, publish breaking changes in a **new major version**.

   A major version update – such as from v1.x.x to v2.x.x – can be a very disruptive upgrade for your module's users. It should be a last resort. For more, see Publishing breaking API changes.

Coding against an unpublished module

When you begin developing a module or a new version of a module, you won't yet have published it. Before you publish a module, you won't be able to use Go commands to add the module as a dependency. Instead, at first, when writing client code in a different module that calls functions in the unpublished module, you'll need to reference a copy of the module on the local file system.

You can reference a module locally from the client module's `go.mod` file by using the `replace` directive. For more information, see Requiring module code in a local directory.

Publishing pre-release versions

You can publish pre-release versions to make a module available for others to try it out and give you feedback. A pre-release version includes no guarantee of stability.

Pre-release version numbers are appended with a pre-release identifier. For more on version numbers, see Module version numbering.

Examples:

```
v0.2.1-beta.1
v1.2.3-alpha
```

When making a pre-release available, keep in mind that developers using the pre-release will need to explicitly specify it by version with the `go get` command. That's because, by default, the `go` command prefers release versions over pre-release versions when locating the module you're asking for. So developers must get the pre-release by specifying it explicitly, as in the following example:

```
go get example.com/theirmodule@v1.2.3-alpha
```

You publish a pre-release by tagging the module code in your repository, specifying the pre-release identifier in the tag. For more, see Publishing a module.

Publishing the first (unstable) version

As with other published versions, you can publish release versions that don't guarantee stability or backward compatibility, but give your users an opportunity to try out the module and give feedback.

Unstable releases are those whose version numbers are in the `v0.x.x` range. A `v0` version makes no stability or backward compatibility guarantees. But it allows you to get feedback and refine your API before making stability commitments with `v1` and later. For more, see Module version number.

You can increment the minor and patch parts of the `v0` version number as you make changes towards releasing a stable `v1` version. For example, after releasing `v0.0.0`, you might release `v0.0.1` with bug fixes.

Example version number:

```
v0.1.3
```

You publish an unstable release by tagging the module code in your repository, specifying a `v0` version number in the tag. For more, see Publishing a module.

Publishing the first stable version

Your first stable release will have a `v1.x.x` version number. The first stable release follows pre-release and `v0` releases through which you got feedback, fixed bugs, and stabilized the module.

With a `v1` release, you're making the following commitments:

- Developers can upgrade to future minor and patch releases without breaking their code.
- You won't make further changes to the module's public API that break backward compatibility.
- You won't remove exported types that would break backward compatibility.
- Future changes to your API (such as adding new struct fields) will be backward compatible.
- Bug fixes (like security fixes) will be included in patches or minor releases.

Note: While your first major version might be a `v0` release, a `v0` version does not signal stability or backward compatibility guarantees. When you increment from `v0` to `v1`, you needn't be concerned about breaking backward compatibility because `v0` was not considered stable.

For more, see Module version numbering.

Example of a stable version:

```
v1.0.0
```

You publish a first stable release by tagging the module code in your repository, specifying a `v1` version number. For more, see Publishing a module.

Publishing bug fixes

You can publish a release limited to bug fixes, called a patch release.

A *patch release* includes only minor changes with no modifications to the public API. Developers can upgrade safely without changing their code.

Note: Your patch release should avoid upgrading transitive dependencies by more than a patch, to prevent unintended major changes.

A patch release increments the patch part of the version number. For example:

- Old version: `v1.0.0`
- New version: `v1.0.1`

Publish by tagging the repository with the new patch version.

Publishing non-breaking API changes

Make non-breaking changes and publish as a *minor* version.

This changes the API but doesn't break existing code, including dependencies or adding new functions, methods, struct fields, or types. Backward compatibility is preserved.

Increment the minor part of the version number. Example:

- Old version: `v1.0.1`
- New version: `v1.1.0`

Publish by tagging the module with the new minor version.

Publishing breaking API changes

Publish a *major* version that breaks backward compatibility.

This generally involves changes that break code relying on the previous version. It should be avoided unless necessary.

Steps:

1. Create a new branch for the major version in your repository.
2. Update the module path to include the new major version, e.g.:

   ```
   example.com/mymodule/v2
   ```

3. Change package import paths accordingly.
4. Publish pre-releases for feedback before official release.
5. Tag the code with the new major version, e.g., from `v1.5.2` to `v2.0.0`, and push.

For more, see Developing a major version update.

This completes the summarized module release and versioning workflow.

Managing module source

When you're developing modules to publish for others to use, you can help ensure that your modules are easier for other developers to use by following the repository conventions described in this topic.

This topic describes actions you might take when managing your module repository. For information about the sequence of workflow steps you'd take when revising from version to version, see Module release and versioning workflow.

Some of the conventions described here are required in modules, while others are best practices. This content assumes you're familiar with the basic module use practices described in Managing dependencies.

Go supports the following repositories for publishing modules: Git, Subversion, Mercurial, Bazaar, and Fossil.

For an overview of module development, see Developing and publishing modules.

How Go tools find your published module

In Go's decentralized system for publishing modules and retrieving their code, you can publish your module while leaving the code in your repository. Go tools rely on naming rules that have repository paths and repository tags indicating a module's name and version number. When your repository follows these requirements, your module code is downloadable from your repository by Go tools such as the `[go get] (https://golang.org/ref/mod#go-get)` command.

When a developer uses the `go get` command to get source code for packages their code imports, the command does the following:

1. From `import` statements in Go source code, `go get` identifies the module path within the package path.
2. Using a URL derived from the module path, the command locates the module source on a module proxy server or at its repository directly.
3. Locates source for the module version to download by matching the module's version number to a repository tag to discover the code in the repository. When a version number to use is not yet known, `go get` locates the latest release version.
4. Retrieves module source and downloads it to the developer's local module cache.

Organizing code in the repository

You can keep maintenance simple and improve developers' experience with your module by following the conventions described here. Getting your module code into a repository is generally as simple as with other code.

The following diagram illustrates a source hierarchy for a simple module with two packages.

Your initial commit should include files listed in the following table:

File	Description
LICENSE	The module's license.
go.mod	Describes the module, including its module path (in effect, its name) and its dependencies. For more, see the go.mod reference. The module path will be given in a module directive, such as: `module example.com/mymodule`

File	Description
	For more about choosing a module path, see Managing dependencies. Though you can edit the go.mod file, you'll find it more reliable to make changes through `go` commands.
go.sum	Contains cryptographic hashes that represent the module's dependencies. Go tools use these hashes to authenticate downloaded modules, attempting to confirm that the downloaded module is authentic. Where this confirmation fails, Go will display a security error. The file will be empty or not present when there are no dependencies. You shouldn't edit this file except by using the `go mod tidy` command, which removes unneeded entries.
Package directories and `.go` sources.	Directories and `.go` files that comprise the Go packages and sources in the module.

From the command-line, you can create an empty repository, add the files that will be part of your initial commit, and commit with a message. Here's an example using git:

```
$ git init
$ git add --all
$ git commit -m "mycode: initial commit"
$ git push
```

Choosing repository scope

You publish code in a module when the code should be versioned independently from code in other modules.

Designing your repository so that it hosts a single module at its root directory will help keep maintenance simpler, particularly over time as you publish new minor and patch versions, branch into new major versions, and so on. However, if your needs require it, you can instead maintain a collection of modules in a single repository.

Sourcing one module per repository

You can maintain a repository that has a single module's source in it. In this model, you place your `go.mod` file at the repository root, with package subdirectories containing Go source beneath.

This is the simplest approach, making your module likely easier to manage over time. It helps you avoid the need to prefix a module version number with a directory path.

Sourcing multiple modules in a single repository

You can publish multiple modules from a single repository. For example, you might have code in a single repository that constitutes multiple modules, but want to version those modules separately.

Each subdirectory that is a module root directory must have its own `go.mod` file.

Sourcing module code in subdirectories changes the form of the version tag you must use when publishing a module. You must prefix the version number part of the tag with the name of the subdirectory that is the module root. For more about version numbers, see Module version numbering.

For example, for module `example.com/mymodules/module1` below, you would have the following for version v1.2.3:

- Module path: `example.com/mymodules/module1`
- Version tag: `module1/v1.2.3`

- Package path imported by a user: `example.com/mymodules/module1/package1`
- Module path and version as specified in a user's require directive: `example.com/mymodules/module1 v1.2.3`

Organizing a Go module

A common question developers new to Go have is “How do I organize my Go project?”, in terms of the layout of files and folders. The goal of this document is to provide some guidelines that will help answer this question. To make the most of this document, make sure you’re familiar with the basics of Go modules by reading the tutorial and managing module source.

Go projects can include packages, command-line programs or a combination of the two. This guide is organized by project type.

Basic package

A basic Go package has all its code in the project’s root directory. The project consists of a single module, which consists of a single package. The package name matches the last path component of the module name. For a very simple package requiring a single Go file, the project structure is:

```
project-root-directory/
  go.mod
  modname.go
  modname_test.go
```

[throughout this document, file/package names are entirely arbitrary]

Assuming this directory is uploaded to a GitHub repository at `github.com/someuser/modname`, the `module` line in the `go.mod` file should say:

```
module github.com/someuser/modname
```

The code in `modname.go` declares the package with:

```
package modname

// ... package code here
```

Users can then rely on this package by *import*-ing it in their Go code with:

```
import "github.com/someuser/modname"
```

A Go package can be split into multiple files, all residing within the same directory, e.g.:

```
project-root-directory/
  go.mod
  modname.go
  modname_test.go
  auth.go
  auth_test.go
  hash.go
  hash_test.go
```

All the files in the directory declare `package modname`.

Basic command

A basic executable program (or command-line tool) is structured according to its complexity and code size. The simplest program can consist of a single Go file where `func main` is defined. Larger programs can

have their code split across multiple files, all declaring `package main`:

```
project-root-directory/
  go.mod
  auth.go
  auth_test.go
  client.go
  main.go
```

Here the `main.go` file contains `func main`, but this is just a convention. The "main" file can also be called `modname.go` (for an appropriate value of `modname`) or anything else.

Assuming this directory is uploaded to a GitHub repository at `github.com/someuser/modname`, the `module` line in the `go.mod` file should say:

```
module github.com/someuser/modname
```

And a user should be able to install it on their machine with:

```
$ go install github.com/someuser/modname@latest
```

Package or command with supporting packages

Larger packages or commands may benefit from splitting off some functionality into supporting packages. Initially, it's recommended placing such packages into a directory named `internal`; this prevents other modules from depending on packages we don't necessarily want to expose and support for external uses. Since other projects cannot import code from our `internal` directory, we're free to refactor its API and generally move things around without breaking external users. The project structure for a package is thus:

```
project-root-directory/
  internal/
    auth/
      auth.go
      auth_test.go
    hash/
      hash.go
      hash_test.go
  go.mod
  modname.go
  modname_test.go
```

The `modname.go` file declares `package modname`, `auth.go` declares `package auth`, and so on. `modname.go` can import the `auth` package as follows:

```
import "github.com/someuser/modname/internal/auth"
```

The layout for a command with supporting packages in an `internal` directory is very similar, except that the file(s) in the root directory declare `package main`.

Multiple packages

A module can consist of multiple importable packages; each package has its own directory, and can be structured hierarchically. Here's a sample project structure:

```
project-root-directory/
  go.mod
  modname.go
  modname_test.go
```

```
  auth/
    auth.go
    auth_test.go
    token/
      token.go
      token_test.go
  hash/
    hash.go
  internal/
    trace/
      trace.go
```

As a reminder, we assume that the `module` line in `go.mod` says:

```
module github.com/someuser/modname
```

The `modname` package resides in the root directory, declares `package modname`, and can be imported by users with:

```
import "github.com/someuser/modname"
```

Sub-packages can be imported by users as follows:

```
import "github.com/someuser/modname/auth"
import "github.com/someuser/modname/auth/token"
import "github.com/someuser/modname/hash"
```

Package `trace` that resides in `internal/trace` cannot be imported outside this module. It's recommended to keep packages in `internal` as much as possible.

Multiple commands

Multiple programs in the same repository will typically have separate directories:

```
project-root-directory/
  go.mod
  internal/
    ... shared internal packages
  prog1/
    main.go
  prog2/
    main.go
```

In each directory, the program's Go files declare `package main`. A top-level `internal` directory can contain shared packages used by all commands in the repository.

Users can install these programs as follows:

```
$ go install github.com/someuser/modname/prog1@latest
$ go install github.com/someuser/modname/prog2@latest
```

A common convention is placing all commands in a repository into a `cmd` directory; while this isn't strictly necessary in a repository that consists only of commands, it's very useful in a mixed repository that has both commands and importable packages, as we will discuss next.

Packages and commands in the same repository

Sometimes a repository will provide both importable packages and installable commands with related functionality. Here's a sample project structure for such a repository:

```
project-root-directory/
  go.mod
  modname.go
  modname_test.go
  auth/
    auth.go
    auth_test.go
  internal/
    ... internal packages
  cmd/
    prog1/
      main.go
    prog2/
      main.go
```

Assuming this module is called `github.com/someuser/modname`, users can now both import packages from it:

```
import "github.com/someuser/modname"
import "github.com/someuser/modname/auth"
```

And install programs from it:

```
$ go install github.com/someuser/modname/cmd/prog1@latest
$ go install github.com/someuser/modname/cmd/prog2@latest
```

Server project

Go is a common language choice for implementing *servers*. There is a very large variance in the structure of such projects, given the many aspects of server development: protocols (REST? gRPC?), deployments, front-end files, containerization, scripts, and so on. We will focus our guidance here on the parts of the project written in Go.

Server projects typically won't have packages for export, since a server is usually a self-contained binary (or a group of binaries). Therefore, it's recommended to keep the Go packages implementing the server's logic in the `internal` directory. Moreover, since the project is likely to have many other directories with non-Go files, it's a good idea to keep all Go commands together in a `cmd` directory:

```
project-root-directory/
  go.mod
  internal/
    auth/
      ...
    metrics/
      ...
    model/
      ...
  cmd/
    api-server/
      main.go
    metrics-analyzer/
      main.go
    ...
  ... the project's other directories with non-Go code
```

In case the server repository grows packages that become useful for sharing with other projects, it's best to split these off to separate modules.

Developing a major version update

You must update to a major version when changes you're making in a potential new version can't guarantee backward compatibility for the module's users. For example, you'll make this change if you change your module's public API such that it breaks client code using previous versions of the module.

> **Note:** Each release type – major, minor, patch, or pre-release – has a different meaning for a module's users. Those users rely on these differences to understand the level of risk a release represents to their own code. In other words, when preparing a release, be sure that its version number accurately reflects the nature of the changes since the preceding release. For more on version numbers, see Module version numbering.

See also

- For an overview of module development, see Developing and publishing modules.
- For an end-to-end view, see Module release and versioning workflow.

Considerations for a major version update

You should only update to a new major version when it's absolutely necessary. A major version update represents significant churn for both you and your module's users. When you're considering a major version update, think about the following:

- Be clear with your users about what releasing the new major version means for your support of previous major versions.

  Are previous versions deprecated? Supported as they were before? Will you be maintaining previous versions, including with bug fixes?

- Be ready to take on the maintenance of two versions: the old and the new. For example, if you fix bugs in one, you'll often be porting those fixes into the other.

- Remember that a new major version is a new module from a dependency management perspective. Your users will need to update to use a new module after you release, rather than simply upgrading.

  That's because a new major version has a different module path from the preceding major version. For example, for a module whose module path is `example.com/mymodule`, a v2 version would have the module path `example.com/mymodule/v2`.

- When you're developing a new major version, you must also update import paths wherever code imports packages from the new module. Your module's users must also update their import paths if they want to upgrade to the new major version.

Branching for a major release

The most straightforward approach to handling source when preparing to develop a new major version is to branch the repository at the latest version of the previous major version.

For example, in a command prompt you might change to your module's root directory, then create a new v2 branch there:

```
$ cd mymodule
$ git checkout -b v2
Switched to a new branch "v2"
```

Once you have the source branched, you'll need to make the following changes to the source for your new version:

- In the new version's `go.mod` file, append the new major version number to the module path, as in the following example:
    - Existing version: `example.com/mymodule`
    - New version: `example.com/mymodule/v2`
- In your Go code, update every imported package path where you import a package from the module, appending the major version number to the module path portion.
    - Old import statement: `import "example.com/mymodule/package1"`
    - New import statement: `import "example.com/mymodule/v2/package1"`

For publishing steps, see Publishing a module.

Publishing a module

When you want to make a module available for other developers, you publish it so that it's visible to Go tools. Once you've published the module, developers importing its packages will be able to resolve a dependency on the module by running commands such as `go get`.

> **Note:** Don't change a tagged version of a module after publishing it. For developers using the module, Go tools authenticate a downloaded module against the first downloaded copy. If the two differ, Go tools will return a security error. Instead of changing the code for a previously published version, publish a new version.

See also

- For an overview of module development, see Developing and publishing modules
- For a high-level module development workflow – which includes publishing – see Module release and versioning workflow.

Publishing steps

Use the following steps to publish a module.

1. Open a command prompt and change to your module's root directory in the local repository.
2. Run `go mod tidy`, which removes any dependencies the module might have accumulated that are no longer necessary.

```
$ go mod tidy
```

3. Run `go test ./...` a final time to make sure everything is working.
   This runs the unit tests you've written to use the Go testing framework.

```
$ go test ./...
ok      example.com/mymodule       0.015s
```

4. Tag the project with a new version number using the `git tag` command.
   For the version number, use a number that signals to users the nature of changes in this release. For more, see Module version number.

```
$ git commit -m "mymodule: changes for v0.1.0"
$ git tag v0.1.0
```

5. Push the new tag to the origin repository.

```
$ git push origin v0.1.0
```

6. Make the module available by running the `go list` command to prompt Go to update its index of modules with information about the module you're publishing.
   Precede the command with a statement to set the `GOPROXY` environment variable to a Go proxy. This will ensure that your request reaches the proxy.

```
$ GOPROXY=proxy.golang.org go list -m example.com/mymodule@v0.1.0
```

Developers interested in your module import a package from it and run the `go get` command just as they would with any other module. They can run the `go get` command for latest versions or they can specify a particular version, as in the following example:

```
$ go get example.com/mymodule@v0.1.0
```