Tutorial: Getting started with multi-module workspaces - The Go Programming Language

Read time: 4 minutes

---

1. Documentation
2. Tutorials
3. 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 tells Go which version of Go the file should be interpreted with. It's similar to the `go` directive in the `go.mod` file.

The `use` directive tells Go that the module in the `hello` directory should be main modules 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 the `go run` command outside the module or the workspace would result in an error because the `go` command 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 then add a new function to the `reverse` package that we can 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 the `git` command to clone the repository:

```
$ 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 was just 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. It will now look like this:

```
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 of the package in the module cache that we downloaded with the `go get` command.

3. Add the new function.

We'll add a new function to reverse a number to the `golang.org/x/example/hello/reverse` package.

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

```
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.

Modify the contents of `workspace/hello/hello.go` to contain the following contents:

```
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 specified by the `go.work` file, and similarly resolves the `golang.org/x/example/hello/reverse` import 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 step¶

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

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

That way, the `go` command can properly resolve the modules outside the workspace.

Learn more about workspaces¶

The `go` command has a couple of subcommands for working with workspaces in addition to `go work init` which we saw earlier in the tutorial:

- `go work use [-r] [dir]` adds a `use` directive to the `go.work` file for `dir`, if it exists, and removes the `use` directory if the argument directory doesn't exist. The `-r` flag examines subdirectories of `dir` recursively.
- `go work edit` edits the `go.work` file similarly to `go mod edit`
- `go work sync` syncs dependencies from the workspace's build list into each of the workspace modules.

See Workspaces in the Go Modules Reference for more detail on workspaces and `go.work` files.