managing-dependencies

- Workflow for using and managing dependencies {#workflow}
- Managing dependencies as modules {#modules}
- Locating and importing useful packages {#locating_packages}
- Enabling dependency tracking in your code {#enable_tracking}
- Naming a module {#naming_module}
- Adding a dependency {#adding_dependency}
- Getting a specific dependency version {#getting_version}
- Discovering available updates {#discovering_updates}
- Upgrading or downgrading a dependency {#upgrading}
- Synchronizing your code's dependencies {#synchronizing}
- Developing and testing against unpublished module code {#unpublished}
  - Requiring module code in a local directory {#local_directory}
  - Requiring external module code from your own repository fork {#external_fork}
- Getting a specific commit using a repository identifier {#repo_identifier}
- Removing a dependency {#removing_dependency}
- Tool dependencies {#tools}
- Specifying a module proxy server {#proxy_server}

When your code uses external packages, those packages (distributed as modules) become dependencies. Over time, you may need to upgrade them or replace them. Go provides dependency management tools that help you keep your Go applications secure as you incorporate external dependencies.

This topic describes how to perform tasks to manage dependencies you take on in your code. You can perform most of these with Go tools. This topic also describes how to perform a few other dependency-related tasks you might find useful.

See also

- If you're new to working with dependencies as modules, take a look at the Getting started tutorial for a brief introduction.

- Using the `go` command to manage dependencies helps ensure that your requirements remain consistent and the content of your go.mod file is valid. For reference on the commands, see Command go. You can also get help from the command line by typing `go help` *command-name*, as with `go help mod tidy`.
- Go commands you use to make dependency changes edit your go.mod file. For more about the contents of the file, see go.mod file reference.
- Making your editor or IDE aware of Go modules can make the work of managing them easier. For more on editors that support Go, see Editor plugins and IDEs.
- This topic doesn't describe how to develop, publish, and version modules for others to use. For more on that, see Developing and publishing modules.

Workflow for using and managing dependencies {#workflow}

You can get and use useful packages with Go tools. On pkg.go.dev, you can search for packages you might find useful, then use the `go` command to import those packages into your own code to call their functions.

The following lists the most common dependency management steps. For more about each, see the sections in this topic.

1. Locate useful packages on pkg.go.dev.
2. Import the packages you want in your code.
3. Add your code to a module for dependency tracking (if it isn't in a module already). See Enabling dependency tracking
4. Add external packages as dependencies so you can manage them.
5. Upgrade or downgrade dependency versions as needed over time.

Managing dependencies as modules {#modules}

In Go, you manage dependencies as modules that contain the packages you import. This process is supported by:

- A **decentralized system for publishing** modules and retrieving their code. Developers make their modules available for other developers to use from their own repository and publish with a version number.
- A **package search engine** and documentation browser (pkg.go.dev) at which you can find modules. See Locating and importing useful packages.
- A module **version numbering convention** to help you understand a module's stability and backward compatibility guarantees. See Module version numbering.
- **Go tools** that make it easier for you to manage dependencies, including getting a module's source, upgrading, and so on. See sections of this topic for more.

Locating and importing useful packages {#locating_packages}

You can search pkg.go.dev to find packages with functions you might find useful.

When you've found a package you want to use in your code, locate the package path at the top of the page and click the Copy path button to copy the path to your clipboard. In your own code, paste the path into an import statement, as in the following example:

```
import "rsc.io/quote"
```

After your code imports the package, enable dependency tracking and get the package's code to compile with. For more, see Enabling dependency tracking in your code and Adding a dependency.

Enabling dependency tracking in your code {#enable_tracking}

To track and manage the dependencies you add, you begin by putting your code in its own module. This creates a go.mod file at the root of your source tree. Dependencies you add will be listed in that file.

To add your code to its own module, use the `go mod init` command. For example, from the command line, change to your code's root directory, then run the command as in the following example:

```
$ go mod init example/mymodule
```

The `go mod init` command's argument is your module's module path. If possible, the module path should be the repository location of your source code.

If at first you don't know the module's eventual repository location, use a safe substitute. This might be the name of a domain you own or another name you control (such as your company name), along with a path following from the module's name or source directory. For more, see Naming a module.

As you use Go tools to manage dependencies, the tools update the go.mod file so that it maintains a current list of your dependencies.

When you add dependencies, Go tools also create a go.sum file that contains checksums of modules you depend on. Go uses this to verify the integrity of downloaded module files, especially for other developers working on your project.

Include the go.mod and go.sum files in your repository with your code.

See the go.mod reference for more.

Naming a module {#naming_module}

When you run `go mod init` to create a module for tracking dependencies, you specify a module path that serves as the module's name. The module path becomes the import path prefix for packages in the module. Be sure to specify a module path that won't conflict with the module path of other modules.

At a minimum, a module path need only indicate something about its origin, such as a company or author or owner name. But the path might also be more descriptive about what the module is or does.

The module path is typically of the following form:

```
<prefix>/<descriptive-text>
```

- The *prefix* is typically a string that partially describes the module, such as a string that describes its origin. This might be:
  - The location of the repository where Go tools can find the module's source code (required if you're publishing the module).
    For example, it might be `github.com/<project-name>/`.

Use this best practice if you think you might publish the module for others to use.

For more about publishing, see Developing and publishing modules.

- A name you control.
  If you're not using a repository name, be sure to choose a prefix that you're confident won't be used by others. A good choice is your company's name. Avoid common terms such as `widgets`, `utilities`, or `app`.

- For the *descriptive text*, a good choice would be a project name. Remember that package names carry most of the weight of describing functionality. The module path creates a namespace for those package names.

Reserved module path prefixes

Go guarantees that the following strings won't be used in package names.

- `test` -- You can use `test` as a module path prefix for a module whose code is designed to locally test functions in another module.
  Use the `test` path prefix for modules that are created as part of a test. For example, your test itself might run `go mod init test` and then set up that module in some particular way in order to test with a Go source code analysis tool.
- `example` -- Used as a module path prefix in some Go documentation, such as in tutorials where you're creating a module just to track dependencies.
  Note that Go documentation also uses `example.com` to illustrate when the example might be a published module.

Adding a dependency {#adding_dependency}

Once you're importing packages from a published module, you can add that module to manage as a dependency by using the `go get` command.

The command does the following:

- If needed, it adds `require` directives to your go.mod file for modules needed to build packages named on the command line. A `require` directive tracks the minimum version of a module that your module depends on. See the go.mod reference for more.
- If needed, it downloads module source code so you can compile packages that depend on them. It can download modules from a module proxy like proxy.golang.org or directly from version control repositories. The source is cached locally.

  You can set the location from which Go tools download modules. For more, see Specifying a module proxy server.

The following describes a few examples.

- To add all dependencies for a package in your module, run a command like the one below ("." refers to the package in the current directory):

```
$ go get .
```

- To add a specific dependency, specify its module path as an argument to the command.

```
$ go get example.com/theirmodule
```

The command also authenticates each module it downloads. This ensures that it's unchanged from when the module was published. If the module has changed since it was published -- for example, the developer changed the contents of the commit -- Go tools will present a security error. This authentication check protects you from modules that might have been tampered with.

Getting a specific dependency version {#getting_version}

You can get a specific version of a dependency module by specifying its version in the `go get` command. The command updates the `require` directive in your go.mod file (though you can also update that manually).

You might want to do this if:

- You want to get a specific pre-release version of a module to try out.
- You've discovered that the version you're currently requiring isn't working for you, so you want to get a version you know you can rely on.
- You want to upgrade or downgrade a module you're already requiring.

Here are examples for using the `go get` command:

- To get a specific numbered version, append the module path with an @ sign followed by the version you want:

```
$ go get example.com/theirmodule@v1.3.4
```

- To get the latest version, append the module path with `@latest`:

```
$ go get example.com/theirmodule@latest
```

The following go.mod file `require` directive example (see the go.mod reference for more) illustrates how to require a specific version number:

```
require example.com/theirmodule v1.3.4
```

Discovering available updates {#discovering_updates}

You can check to see if there are newer versions of dependencies you're already using in your current module. Use the `go list` command to display a list of your module's dependencies, along with the latest version available for that module. Once you've discovered available upgrades, you can try them out with your code to decide whether or not to upgrade to new versions.

For more about the `go list` command, see `go list -m`.

Here are a couple of examples.

- List all of the modules that are dependencies of your current module, along with the latest version available for each:

  ```
  $ go list -m -u all
  ```

- Display the latest version available for a specific module:

  ```
  $ go list -m -u example.com/theirmodule
  ```

Upgrading or downgrading a dependency {#upgrading}

You can upgrade or downgrade a dependency module by using Go tools to discover available versions, then add a different version as a dependency.

1. To discover new versions use the `go list` command as described in Discovering available updates.
2. To add a particular version as a dependency, use the `go get` command as described in Getting a specific dependency version.

Synchronizing your code's dependencies {#synchronizing}

You can ensure that you're managing dependencies for all of your code's imported packages while also removing dependencies for packages you're no longer importing.

This can be useful when you've been making changes to your code and dependencies, possibly creating a collection of managed dependencies and downloaded modules that no longer match the collection specifically required by the packages imported in your code.

To keep your managed dependency set tidy, use the `go mod tidy` command. Using the set of packages imported in your code, this command edits your go.mod file to add modules that are necessary but missing. It also removes unused modules that don't provide any relevant packages.

The command has no arguments except for one flag, -v, that prints information about removed modules.

```
$ go mod tidy
```

Developing and testing against unpublished module code {#unpublished}

You can specify that your code should use dependency modules that may not be published. The code for these modules might be in their respective repositories, in a fork of those repositories, or on a drive with the current module that consumes them.

You might want to do this when:

- You want to make your own changes to an external module's code, such as after forking and/or cloning it. For example, you might want to prepare a fix to the module, then send it as a pull request to the module's developer.
- You're building a new module and haven't yet published it, so it's unavailable on a repository where the `go get` command can reach it.

Requiring module code in a local directory {#local_directory}

You can specify that the code for a required module is on the same local drive as the code that requires it. You might find this useful when you are:

- Developing your own separate module and want to test from the current module.
- Fixing issues in or adding features to an external module and want to test from the current module. (Note that you can also require the external module from your own fork of its repository. For more, see Requiring external module code from your own repository fork.)

To tell Go commands to use the local copy of the module's code, use the `replace` directive in your go.mod file to replace the module path given in a `require` directive. See the go.mod reference for more about directives.

In the following go.mod file example, the current module requires the external module `example.com/theirmodule`, with a nonexistent version number (`v0.0.0-unpublished`) used to ensure the replacement works correctly. The `replace` directive then replaces the original module path with `../theirmodule`, a directory that is at the same level as the current module's directory.

```
module example.com/mymodule

go 1.23.0

require example.com/theirmodule v0.0.0-unpublished

replace example.com/theirmodule v0.0.0-unpublished => ../theirmodule
```

When setting up a `require`/`replace` pair, use the `go mod edit` and `go get` commands to ensure that requirements described by the file remain consistent:

```
$ go mod edit -replace=example.com/theirmodule@v0.0.0-unpublished=../theirmodule
$ go get example.com/theirmodule@v0.0.0-unpublished
```

Note: When you use the replace directive, Go tools don't authenticate external modules as described in Adding a dependency.

For more about version numbers, see Module version numbering.

Go 1.18 adds workspace mode to Go, which lets you work on multiple modules simultaneously. See Tutorial: Getting started with multi-module workspaces.

Requiring external module code from your own repository fork {#external_fork}

When you have forked an external module's repository (such as to fix an issue in the module's code or to add a feature), you can have Go tools use your fork for the module's source. This can be useful for testing changes from your own code. (Note that you can also require the module code in a directory that's on the local drive with the module that requires it. For more, see Requiring module code in a local directory.)

You do this by using a `replace` directive in your go.mod file to replace the external module's original module path with a path to the fork in your repository. This directs Go tools to use the replacement path (the fork's location) when compiling, for example, while allowing you to leave `import` statements unchanged from the original module path.

For more about the `replace` directive, see the go.mod file reference.

In the following go.mod file example, the current module requires the external module `example.com/theirmodule`. The `replace` directive then replaces the original module path with `example.com/myfork/theirmodule`, a fork of the module's own repository.

```
module example.com/mymodule

go 1.23.0

require example.com/theirmodule v1.2.3

replace example.com/theirmodule v1.2.3 => example.com/myfork/theirmodule v1.2.3-fixed
```

When setting up a `require`/`replace` pair, use Go tool commands to ensure that requirements described by the file remain consistent. Use the `go list` command to get the version in use by the current module. Then use the `go mod edit` command to replace the required module with the fork:

```
$ go list -m example.com/theirmodule
example.com/theirmodule v1.2.3
$ go mod edit -replace=example.com/theirmodule@v1.2.3=example.com/myfork/theirmodule@v1.2.3-fixed
```

Note: When you use the `replace` directive, Go tools don't authenticate external modules as described in Adding a dependency.

For more about version numbers, see Module version numbering.

Getting a specific commit using a repository identifier {#repo_identifier}

You can use the `go get` command to add unpublished code for a module from a specific commit in its repository.

To do this, you use the `go get` command, specifying the code you want with an `@` sign. When you use `go get`, the command will add to your go.mod file a `require` directive that requires the external module, using a pseudo-version number based on details about the commit.

The following examples provide a few illustrations. These are based on a module whose source is in a git repository.

- To get the module at a specific commit, append the form @*commithash*:

  ```
  $ go get example.com/theirmodule@4cf76c2
  ```

- To get the module at a specific branch, append the form @*branchname*:

  ```
  $ go get example.com/theirmodule@bugfixes
  ```

Removing a dependency {#removing_dependency}

When your code no longer uses any packages in a module, you can stop tracking the module as a dependency.

To stop tracking all unused modules, run the `go mod tidy` command. This command may also add missing dependencies needed to build packages in your module.

```
$ go mod tidy
```

To remove a specific dependency, use the `go get` command, specifying the module's module path and appending `@none`, as in the following example:

```
$ go get example.com/theirmodule@none
```

The `go get` command will also downgrade or remove other dependencies that depend on the removed module.

Tool dependencies {#tools}

Tool dependencies let you manage developer tools that are written in Go and used when working on your module. For example, you might use `stringer` with `go generate`, or a specific linter or formatter as part of preparing your change for submission.

In Go 1.24 and above, you can add a tool dependency with:

```
$ go get -tool golang.org/x/tools/cmd/stringer
```

This will add a `tool` directive to your `go.mod` file, and ensure the necessary require directives are present. Once this directive is added you can run the tool by passing the last non-major-version component of the tool's import path to `go tool`:

```
$ go tool stringer
```

In the case that multiple tools share the last path fragment, or the path fragment matches one of the tools shipped with the Go distribution, you must pass the full package path instead:

```
$ go tool golang.org/x/tools/cmd/stringer
```

To see a list of all tools currently available, run `go tool` with no arguments:

```
$ go tool
```

You can manually add a `tool` directive to your `go.mod`, but you must ensure that there is a `require` directive for the module that defines the tool. The easiest way to add any missing `require` directives is to run:

```
$ go mod tidy
```

Requirements needed to satisfy tool dependencies behave like any other requirements in your module graph. They participate in minimal version selection and respect `require`, `replace` and `exclude` directives. Due to module pruning, when you depend on a module that itself has a tool dependency, requirements that exist just to satisfy that tool dependency do not usually become requirements of your module.

The `tool` meta-pattern provides a way to perform operations on all tools simultaneously. For example you can upgrade all tools with `go get tool`, which is equivalent to `go get tool@upgrade`, or install them all to $GOBIN with `go install tool`.

In Go versions before 1.24, you can achieve something similar to a `tool` directive by adding a blank import to a go file within the module that is excluded from the build using build constraints. If you do this, you can then use `go run` with the full package path to run the tool.

Specifying a module proxy server {#proxy_server}

When you use Go tools to work with modules, the tools by default download modules from proxy.golang.org (a public Google-run module mirror) or directly from the module's repository. You can specify that Go tools should instead use another proxy server for downloading and authenticating modules.

You might want to do this if you (or your team) have set up or chosen a different module proxy server that you want to use. For example, some set up a module proxy server in order to have greater control over how dependencies are used.

To specify another module proxy server for Go tools use, set the `GOPROXY` environment variable to the URL of one or more servers. Go tools will try each URL in the order you specify. By default, `GOPROXY` specifies a public Google-run module proxy first, then direct download from the module's repository (as specified in its module path):

```
GOPROXY="https://proxy.golang.org,direct"
```

For more about the `GOPROXY` environment variable, including values to support other behavior, see the `go` command reference.

You can set the variable to URLs for other module proxy servers, separating URLs with either a comma or a pipe.

- When you use a comma, Go tools will try the next URL in the list only if the current URL returns an HTTP 404 or 410.

  ```
  GOPROXY="https://proxy.example.com,https://proxy2.example.com"
  ```

- When you use a pipe, Go tools will try the next URL in the list regardless of the HTTP error code.

  ```
  GOPROXY="https://proxy.example.com|https://proxy2.example.com"
  ```

Go modules are frequently developed and distributed on version control servers and module proxies that aren’t available on the public internet. You can set the `GOPRIVATE` environment variable to configure the `go` command to download and build modules from private sources. Then the go command can download and build modules from private sources.

The `GOPRIVATE` or `GONOPROXY` environment variables may be set to lists of glob patterns matching module prefixes that are private and should not be requested from any proxy. For example:

```
GOPRIVATE=*.corp.example.com,*.research.example.com
```

□□□ #2
□ □□□ □□□ 2026-10-26 20:26:29 UTC
□ □□□ □□□ 2026-10-26 20:26:29 UTC