JANUARY 2026 SERIES

FROM GO BUILD TO GO RUN

GOLANG 2026 - NIV RAVE

#28

CREATING MODULES: REPLACE & VERSIONING

WHEN GO GET ISN'T ENOUGH

Semantic Import Versioning

The "Major" Rule

In Go, Semantic Versioning is enforced at the import level. If you break backwards compatibility (bumping from *v1.x.x* to *v2.x.x*), you aren't just changing a tag – you are changing the package's identity.

Go treats *v0* and *v1* as the same module path. However, for *v2* and above, the major version must be appended to the module path in your *go.mod* and all *import* statements.

This is Go's solution to the "Dependency Hell" (Diamond Dependency). Because the paths are different, your project can actually import both *v1* and *v2* of the same library at the same time. This allows you to migrate large codebases incrementally rather than doing a "big bang" refactor when a dependency releases a breaking change.

```go
// --- THE GO.MOD TRANSITION ---

// Version 0.x/1.x
module github.com/user/project

// Version 2.x
module github.com/user/project/v2

// --- THE CONSUMER'S IMPORT ---

import (
    "github.com/user/project"    // This is v1
    "github.com/user/project/v2" // This is v2 (can coexist!)
)
```

Developing Locally

The *replace* Directive

What if you are modifying a library (a shared utility/logger/infra, etc.) and the app that uses it simultaneously? You can't wait to push to GitHub for every test.
The Solution: Use *replace* to point the compiler to a local file path instead of the remote repository.
The Structure:

```
# Force the compiler to use your local folder
go mod edit -replace github.com/deps/lib=../local-lib
```

Warning: Never commit a replace directive that points to a local path to your main branch. It will break the build for everyone else and in CI. Use it strictly for local debugging.

Handling Forked Dependencies

Fixing Upstream Bugs

If a third-party library has a critical bug and the maintainer is unresponsive, you don't have to wait. Fork the repo, fix the bug in your fork, and use *replace* to redirect the official module path to your fork.

```
// In go.mod file:
module my-app

require github.com/original/lib v1.2.3

replace github.com/original/lib => github.com/my-fork/lib v1.2.4-patch
```

Warning: Don’t forget the replace you’ve committed, remember to verify when an updated version of the 3rd party library arrives.

Retracting Versions

The "Oops" Button: *retract*

If you accidentally release a version with a security flaw or a breaking bug, don't delete the tag (which breaks the checksum database). Use *retract*.

```
// In go.mod file:
module github.com/user/project

retract v1.0.1 // Published with critical race condition
retract [v1.0.0, v1.0.5] // Retract a range of versions
```

Retracting a version informs the *go* command to avoid that version. Users running *go list -m -u all* will see a warning that they are on a retracted version.

Multi-Module Workspaces

go.work: The Modern Way

Introduced in Go 1.18, *go.work* files allow you to manage multiple modules in a single workspace without littering your *go.mod* files with replace directives.

```
# Initialize a workspace
go work init ./module1 ./module2
```

This is the preferred way to handle "Monorepo" development. It keeps your *go.mod* files "clean" for production while giving you "replace-like" behavior locally.

Pseudo-versions

Beyond Tags: Decoding Pseudo-versions

When you need a specific commit that hasn't been officially tagged (e.g., a hotfix sitting on main), Go generates a Pseudo-version. It looks like a random string, but it is actually a carefully structured version that satisfies SemVer ordering.

The Anatomy: *v0.0.0-20230101123456-abcdefabcdef*

1. **Base Version**: The most recent tag before this commit (or v0.0.0).
2. **Timestamp**: The UTC commit time (YYYYMMDDHHMMSS).
3. **Revision**: The 12-character prefix of the commit hash.

```
# How to generate one: Tell Go to fetch a specific commit hash
go get github.com/user/project@abcdef123456

# Your go.mod will automatically update to:
# require github.com/user/project v0.0.0-20260114123456-abcdef123456
```

Rule of Thumb:

Use them only as a temporary bridge. If you find yourself relying on a pseudo-version in production, you should pressure the upstream maintainer to cut a proper tag or consider forking the dependency to stabilize your supply chain.

Dependency Vendoring

The */vendor* Strategy

go mod vendor copies all your dependencies into a local directory in your project.

Why do this?

- Zero-network builds (crucial for some CI/CD).
- Guaranteed availability (if GitHub goes down, you still have the code).
- Easier security auditing of third-party source code.

Recommendation:
Only use vendoring if your compliance or CI environment requires it. Most modern teams rely on the Go Proxy for these benefits.

Summary:

- v2+ requires /vN in the module path.
- replace is for local dev and forking; keep it out of main.
- go.work is the modern choice for multi-module projects.
- Use retract instead of deleting git tags.

Tomorrow, we cross into Week 3: Data & Concurrency.
We start with the ‘senior’ way to handle JSON: Encoding, Decoding, and Custom Marshaling.