How I write Go

after 8 years as a Gopher...

by @kinbiko
on 2025-06-21

» About this talk

* Opinionated, but justified.
  * Expect to be asked to analyse some code for smells.
  * We'll discuss together, but I reserve the right to avoid arguments over opinion.
* Writing maintainable Go for large systems.

» Who this is for

* 🤓 Experienced engineers. 1+ years of writing Go recommended.
* 🙋 But I don't mean to exclude, feel free to follow along.
* If it's hard to follow along, feel free to continue the examples from the workshop.
    * Slides shared in discord (ping @kinbiko if they're not)
* Experienced engineers without experience in Go might still find some ideas useful, should they apply to other languages.

» Overview

* Global state and `init()`.
* Build tags.
* Dependency injection.
* Context, Error handling, and logging.
* Module structure.
    * Flat vs nested.
    * `internal`.
* Dependency management.
* Testing
    * Mocking.
    * Testing libraries.
    * Table driven tests.
    * Test templates.
    * Integration tests.
    * Race checker.
* Linting.
* Makefile.

» Global state

* What's wrong here?
* Why is it bad?
* What's the fix?

```
type MyFileUploader struct {
  ImportantConfig []string
  gcsClient       storage.StorageClient
}

var defaultInstance *MyFileUploader

func UploadFile(r io.ReadCloser) {
  // (...) implement file upload using defaultInstance
}

func init() {
  defaultInstance = &MyFileUploader{
    ImportantConfig: []string{"foo", "bar"},
    gcsClient:       storage.NewStorageClient(makeConn()),
  }
}
```

» Build tags

* You probably don't want build tags.
* Extremely strong encapsulation feature.
* Common, but using build tags to bypass normal encapsulation smells.

```
//go:build test
// +build test
```

* What's so bad?
* You're probably better off using env vars.
* So should I never use it?

» Dependency injection

* Small apps: hook it up directly in main. It's fine.
* Medium-big apps: learn how dependency injection works from first principles.
    * Frameworks like wire exist, but aren't necessary, and can generate ugly code & not behave well during large refactoring.

» Module structure

* There is no standard project structure in Go.
  * ⚠️ WARNING: one project claims to be, but this is misleading.
* Be intentional in your structure.
* Critical of new packages, especially with libs.
* Package names are used a lot in code, so be critical of the names.
  * E.g. “repos” is neater than “repositories” and “infra” is neater than “infrastructure”, without really stealing any variable names.
* You can have multiple modules in the same repo. E.g. for example usage of a library.

» Module structure

Here's what I personally like:
Libs: Single package, as much as possible.
CLIs: Try to design as a lib with a `cmd/cliname/main.go` file.
Services: 👉

```
module/
  .github/
  internal/
    cmd/server/main.go
    infra/
    usecase/
    model/
    services/
  Makefile
  go.mod
  go.sum
```

» Dependency management

* Go has a fantastic standard library.
* Be critical of a large amount of transitive dependencies.
* Gophers tend to prefer as few dependencies as possible.
* If a public library has multiple usages, consider multiple modules in the same repo, to avoid polluting the apps that depend on the lib.
* You'll need to find a solution to managing dev-tools:
    * Good news: latest version of go makes this simple.
    * Tools like aqua are also great at a more generic level.

» Observability (o11y) with context.Context

* o11y: logging, tracing, error handling, and metrics.
* "Canonical Logging" based approach:
    * Each request logs exactly once, in a middleware; the application logic never logs.
    * Wrap errors from 3rd party libs, otherwise return as-is.
    * Attach fields (metadata) for structured logging etc. to a `context.Context` with `context.WithValue`.
    * Use an error handling library that lets you attach context when creating/wrapping errors.
        * E.g. `github.com/kinbiko/rogerr`
    * In the middleware, extract the metadata and log as fields.

» Testing

* 🚫 Avoid gomock.
  * It breaks on version upgrades.
  * Its output is so verbose.
  * It's horrible in concurrent tests.
  * Debugging is really hard.
  * Code generation fails on compilation errors which can trigger a compilation deadlock if you're not careful.
  * The code it forces you to write is unidiomatic (EXPECT).
  * You have to make more design decisions around where to put your mocks.

» Testing

* 🚫 Avoid gomock.
* ✔️ Recommend writing your own mocks: this forces you to feel some pain if your interfaces are too large.
    * I use `github.com/kinbiko/mokku` to quickly write my own mocks.

» Testing

* If you treat your tests with respect, then the stdlib testing package is perfect.
    * If you wish Go was more like Java then go ahead and use testify.
* Build your own test harness for integration tests: the ROI is positive after about 10 integration tests for a HTTP server.
    * defer demo if time.
* Sometimes you do want some sugar, e.g. when verifying JSON or running golden tests.
    * `github.com/kinbiko/jsonassert`

» Testing

* Table driven tests are great, but not a goal.
    * There's a test-template generation tool that's built into many IDEs, and it's awful.
    * Use your knowledge of the domain and the problem to form test cases.
    * Identify which test cases are similar in their input and output, and which are very different.
    * Trying to squeeze every possible test case into a single test table structure makes no sense.
        * Use sub tests.
    * For example, if you're testing a service that could return a result or an err, then you often want one subtest for the err and one for the success case.
        * The sub tests can then have a table within them, if it makes sense.
* Don't be clever and write functions in your test tables.
* I've seen 1000+ LOC test tables that have 100+ LOC test bodies.
    * Rewritten this is more like 500 LOC of subtests.

» Testing

* Race checker is great: Always run it on CI.
    * `go test -race`
* Some tests, e.g. that modify state in a DB, cannot be run with the race checker.
    * Use env vars and `t.SkipNow()` to mark these tests, or put them in their own package.
* Make sure you write tests that *can* fail the race checker.
    * You might have to spin up multiple goroutines in a test.
* The race checker might not spot a race condition.
    * But it's always telling the truth when it does spot something.

» Linting

* Golangci-lint is all you need.
* However, invest time in configuring it to your needs.
* Some lints are more annoying than helpful: only keep these if they can be auto-fixed by the linter as well.
    * I recommend doing the fix automatically as part of CI.
* Some lints are annoying to fix, but you should still do it.
    * E.g. "I only added a single nil check, and now the cyclomatic complexity linter fails"
        * Don't nolint these. Fix. It'll only get harder to fix over time.
* You should have a good reason for disabling a linter, and you should document it in the linter config so future you/coworkers can tell if it still applies.

» Makefile

* Makefiles aren't necessary. But they can be very useful in a large org, if standardized.
* In Go, you can think of makefiles as providing project-specific shorthands for what might be longer shell commands. E.g.:
  * Build all binaries in this project.
  * Run tests, but not integration tests.
  * Run only integration tests.
  * Build a production binary (e.g. with ldflags and trimpath).
  * Start the app in docker env.
  * Run all the checks that will run on CI.