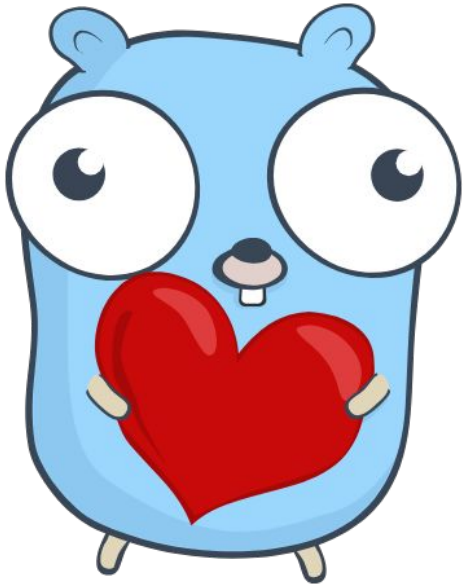
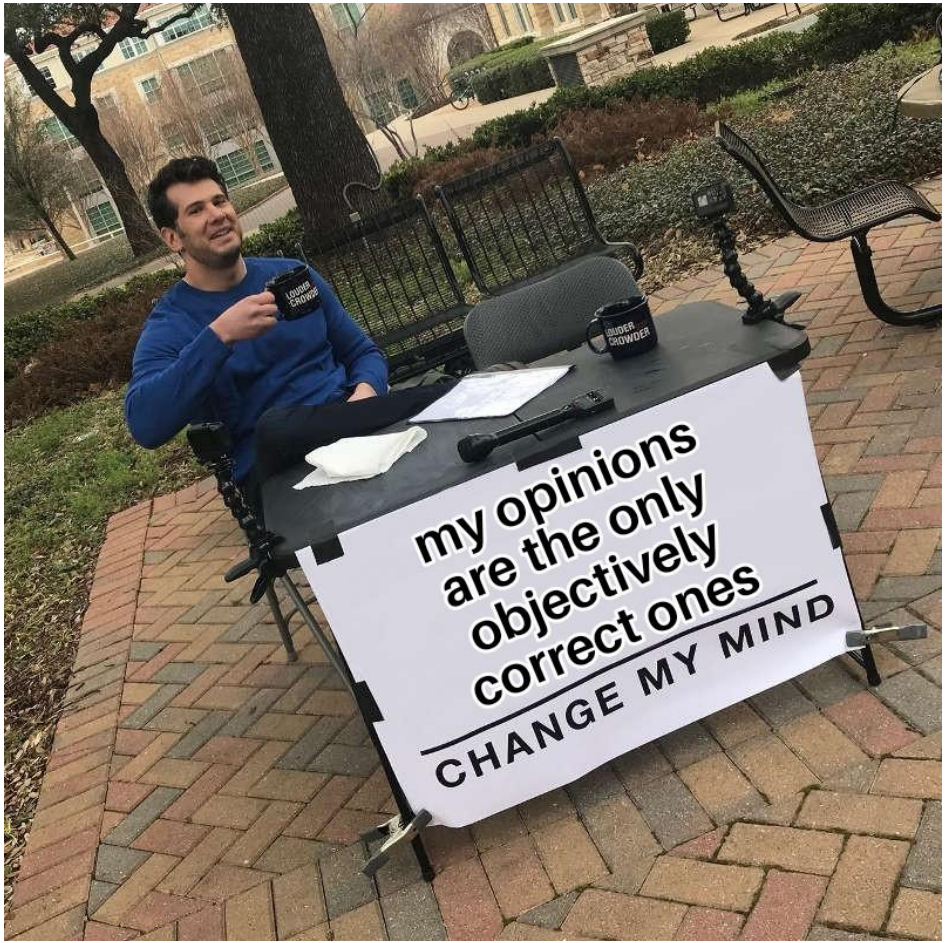


Dr. Golove

Or: How I Learned To Stop Worrying and Love //go:generate

Welcome to my journey!

- Why I disliked code generation
- What changed my mind
- A sprinkling of //go:generate
- What I learned along the way



Code generation is bad and you're bad if you like it.

How did I get there?

- The year is 2000 something-something, SOA rules the world.
- Glorious Architects will write UML diagrams.
- Business Process Engineers will write BPMN.
- All the code will be generated.
- Programmers are no longer needed.

- Automated builds and CI are barely a thing, let alone pipelines.
- External generation tools need to be built, versioned and maintained.
- Generation is a disconnected step that usually needs to be triggered manually.
- Nothing ever actually works better than a good old library or rolling your own.

Go Life

- Mobiliar Versicherung - Platform Engineering Tools and Services (2017 - 2020) (CLIs, Metrics Services, Pipeline Tooling)
- Elblox - Green Energy Trading Platform (2020 - 2022) (Microservices)
- Axpo - Productivity Tools for Business Teams (2022) (Web-Services)
- bespinian - Internal and Open-source Tools (Today) (CLIs, Web-Services)
- Swiss Marketplace Group - Tutti Admin Tooling (Today) (Web-Service, Backend Service)

//go:generate moq -out myinterface_mock_test.go . MyInterface

- Still have to download a tool
- AND make sure it's up to date
- AND always run `go generate`

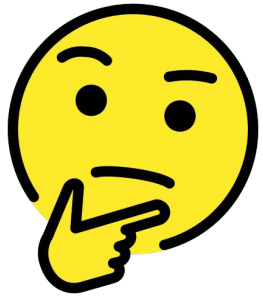
- I'm having a sudden feeling of deja-vu

- Meh, can't I just write mocks by myself?

(Although, after getting a better understanding of how to use interfaces in Go, writing mocks became a bit more painful than using a mock generator.)

//go:generate protoc --go_out=. myproto.proto

- Generating protobuf client/server makes a lot of sense, can't hand code all of that stuff.
- Makes life much easier for creating and updating transport code and interoperation with other languages.



- Tooling is actually not that painful.
- Go is already installed
- `go test` is already used in the pipeline
- Most generation tools can be downloaded using `go get`
- You're already using `go generate` for mocks, right?

- What's wrong, why am I feeling… intrigued?

(Also `bufbuild/buf` is a much more elegant way of generating protobuf)

//go:generate gowrap gen -g -i MyService -t ./zerolog -o myservice_logger.go

//go:generate gowrap gen -g -i MyService -t ./zerolog -o myservice_logger.go

- Microservices require cross cutting functionality and lots of it.
- Service interfaces must be wrapped in middlewares for logging, monitoring, auth, retry, metrics, etc., etc.
- gowrap generates the necessary decorator code based on interface reflection and code templates.
- No magic code or runtime reflection, all the runtime code is there and can be used to debug or understand the flow.

- This whole generation thing is starting to make sense

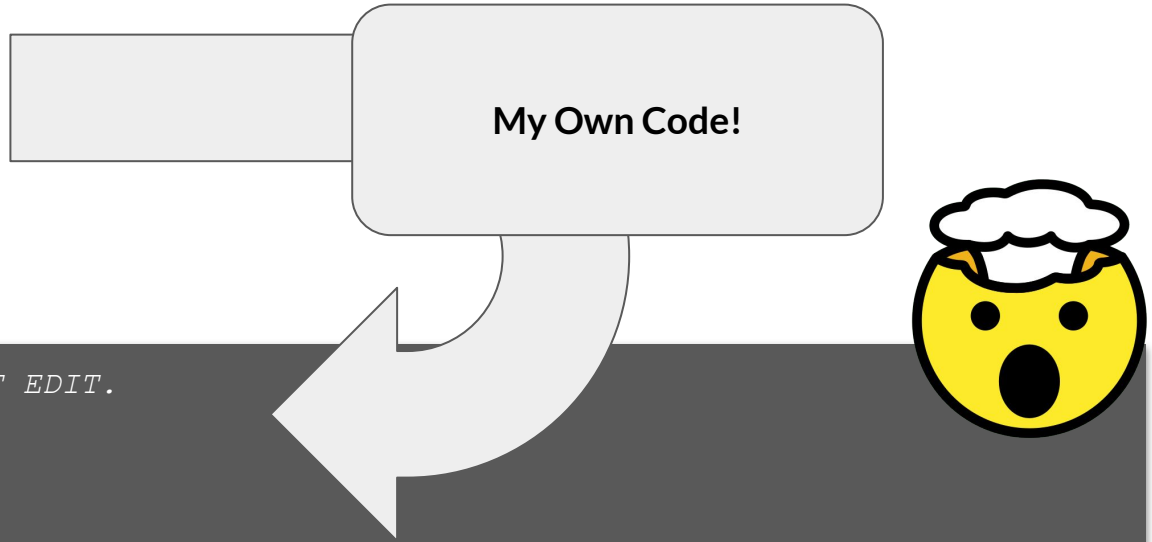
//go:generate go run ../cmd/apierr -s errors.yml -o errors.go

Context

- API interface returns errors
- We want to give each error a unique identifier to match client errors to code efficiently
- If possible, use the IDE to find the exact place an error occurred

//go:generate go run ../cmd/apierr -s errors.yml -o errors.go

```yaml
namespace: app
errors:
  1:
    name: bad request
    category: BadRequest
    message: Bad request.
    params:
    method: string
    path: string
```




```go
// Code generated by apierr; DO NOT EDIT.
package app

import "github.com/myapp/apierr"

// errAppBadRequest returns the apierr with id app000001 BadRequest "Bad request."
func errAppBadRequest(method string, path string, errs ...error) apierr.APIErr {
     return apierr.New("app000001", apierr.CategoryBadRequest, "Bad request.", errs...)
          .With("method", method)
          .With("path", path)
}
```

//go:generate go run ../cmd/apierr -s errors.yml -o errors.go

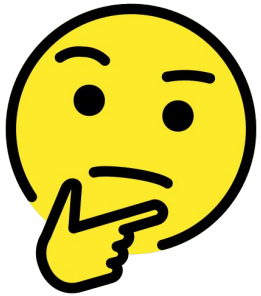
- Go generate allows you to run your own code as a generator.
- It doesn't even know that it can do it.
- This also means you don't need to install binaries as long as the generator is written in Go.

```
//go:generate buf generate
//go:generate go run github.com/bufbuild/buf/cmd/buf generate
```

- Mind blown

Why does it work??!?

- Code generation was never a bad idea.
- Go tooling makes generation a first class citizen.
- Generate is well designed: flexible, powerful and yet simple.
- Consistent with other built in Go tools.
- Strong synergies with other tools like Go templates.
- Virtuous cycle, because it’s easy, the Go ecosystem offers plenty of generators for different use-cases.
- Generated code is there in your project. There is no magic.



Use-cases

- Test-Mocks (moq, gomock)
- Transport Layer (protoc)
- Middleware / Decorators (gomock)
- Error Handling
- Parsers (pigeon, yacc)
- OpenAPI / Swagger Documentation
- Simple JS / CSS Bundling (esbuild)
- Additional Documentation
- Typesafe SQL from Queries (sqlc)
- Translations
- Test Cases
- Constant Strings (Stringer)

Correct Way to Generate*

- Like any tool, it has a purpose and a place. Just because it's powerful, doesn't mean you should use it for everything.
- Use generate before the build step for things you will use in your project. Generate repetitive code you would otherwise need to write by hand.
- Check-in the generated code. (No Magic!)
- Run a check on the pipeline, that runs generate and fails if there are changes.
- Don't test generated code (directly). Test the generator.

- Rob Pike's Blog Post: https://go.dev/blog/generate

*see slide 2

Thanks!

bespinian.io

Gophers by @egonelbre - https://github.com/egonelbre/gophers
All emojis designed by OpenMoji – the open-source emoji and icon project.