Drying Up the Architecture of a Go Application

Incremental refactoring you can ship to production every week

---

Henrique Vicente de Oliveira Pinto
bol. × ING Tech Talks — Amsterdam, 2026

The Problem

Context

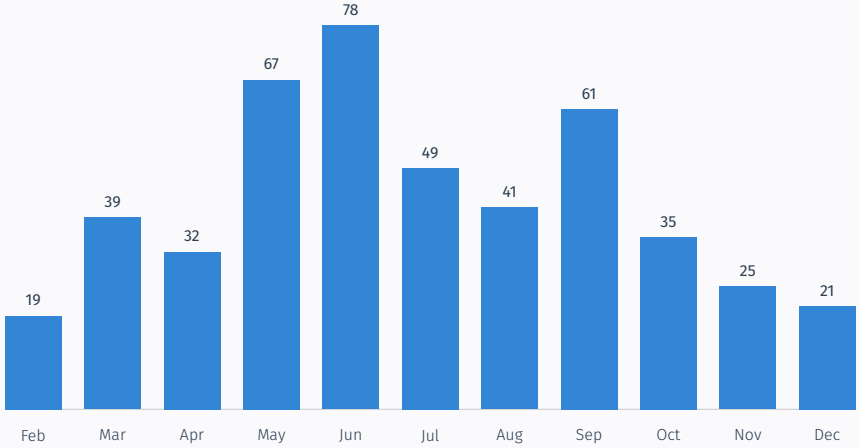
- A back office service (maybe the first Go application at bol.)
- Multiple REST APIs using two OpenAPI .spec files
- Pub/Sub consumers, cronjobs, BigQuery pipelines
- Originally built ~2018 — multiple contributors over time
- Baggage: globals, swallowed errors, limited testing, data race risk
- Modernization driven by the team — 6 contributors, 468 commits

Goal: modernize *incrementally* — no greenfield rewrite, ship every week.

The numbers (Feb–Dec 2025)

468 modernization commits	226 JIRA tickets	+93k / –196k lines (excl. vendor)	–103k net reduction
7,399 file changes	11 months	0 big-bang rewrites	✓ prod every week

Commits per month



Embracing the Standard Library

Why go back to the basics?

- Go's `net/http` gained method-based routing in **Go 1.22**
- Path parameters: `r.PathValue("id")` — no external router needed
- Less API surface, fewer CVEs, smaller binary
- The standard library is maintained by the Go team — forever

Principle: *"A little copying is better than a little dependency."* — Go Proverbs

Before: `go-restful` route registration

```
func NewBookWebService(h Handler) *restful.WebService {
        s := &restful.WebService{}
        s.Path("/books").
          Consumes(restful.MIME_JSON).Produces(restful.MIME_JSON)

        bookID := s.PathParameter("bookID", "book ID (ISBN)").DataType("string")

        s.Route(s.POST("").Reads(entities.Book{}).To(h.Create))
        s.Route(s.PUT("/{bookID}").Param(bookID).To(h.Update))
        s.Route(s.GET("/").Param(PageSize).Param(Page).To(h.Search))
        return s
}
```

After: net/http.ServeMux (Go 1.22+)

```
func (h BookHandler) Mux() http.Handler {
        mux := http.NewServeMux()
        mux.HandleFunc("POST /books", h.Create)
        mux.HandleFunc("PUT /books/{bookID}", h.Update)
        mux.HandleFunc("GET /books", h.Search)
        mux.HandleFunc("POST /books/reviews", h.Reviews)

        sub := http.NewServeMux()
        sub.HandleFunc("GET /books/{bookID}/reviews", h.Reviews)
        mux.Handle("/books/", sub)
        return mux
}
```

Exact match with {}

Patterns without a trailing / already match exactly.
But GET /books/ matches all paths that have it as a prefix.

```
// matches only /books/ --- not /books/fiction, /books/42, etc.
mux.HandleFunc("GET /books/{$}", h.ListBooks)

// matches all paths with /books/ as prefix
mux.HandleFunc("GET /books/", h.CatchAll)
```

Use {} when you need a trailing-slash route that doesn't act as a prefix.

Working around `http.ServeMux` ambiguous pattern conflicts

`ServeMux` panics when patterns conflict — e.g., `GET /books/{bookID}` and `GET /books/reviews` are ambiguous (is `reviews` a book ID?).

The fix: isolate conflicting patterns in separate muxes.

```
mux := http.NewServeMux()
mux.HandleFunc("GET /books", h.Search)
mux.HandleFunc("POST /books", h.Create)

// sub-mux for paths under /books/ avoids the conflict
sub := http.NewServeMux()
sub.HandleFunc("GET /books/{bookID}", h.GetBook)
sub.HandleFunc("GET /books/{bookID}/reviews", h.Reviews)
mux.Handle("/books/", sub)
```

Strict matching catches real ambiguities, but sub-muxes let you structure routes cleanly.

Handler signatures: before

```go
// go-restful: library-specific types everywhere
func (h Handler) Create(req *restful.Request, resp *restful.Response) {
        var s entities.Book
        req.ReadEntity(&s)               // library-specific deserialization
        id := req.PathParameter("id")    // library-specific path params
        user := req.QueryParameter("u")  // library-specific query params
        h.ServeJSON(resp.ResponseWriter, http.StatusCreated, result)
}
```

Handler signatures: after

```
// net/http: only stdlib types
func (h Handler) Create(w http.ResponseWriter, r *http.Request) {
        var s entities.Book
        json.NewDecoder(r.Body).Decode(&s)  // stdlib json
        id := r.PathValue("id")             // Go 1.22+ path params
        user := r.URL.Query().Get("u")      // stdlib query params
        h.ServeJSON(w, http.StatusCreated, result)
}
```

Aspect	go-restful	net/http
Path params	`req.PathParameter()`	`r.PathValue()`
Query params	`req.QueryParameter()`	`r.URL.Query().Get()`
Body	`req.ReadEntity(&v)`	`json.NewDecoder().Decode(&v)`
Response	`resp.ResponseWriter`	`w`

Managing Third-Party Dependencies

45 → 25 direct modules on `go.mod`

```
+ github.com/henvic/httpretty
+ github.com/henvic/pgtools
+ github.com/jackc/pgerrcode
+ github.com/jackc/tern/v2
+ github.com/kelseyhightower/envconfig
+ go.opentelemetry.io/otel
- github.com/dgrijalva/jwt-go
- github.com/emicklei/go-restful
- github.com/go-openapi/…
- github.com/golang-migrate/migrate/v4
- github.com/idebruijn/bddl
- github.com/jessevdk/go-flags
- github.com/lib/pq
- github.com/pascaldekloe/goe
- github.com/robfig/cron/v3
- github.com/syllabix/swagserver
- github.com/uber-go/tally/v4
…
```

PostgreSQL

- Previously using the **textual protocol**: values serialized as strings, parsed back on every query — slow and error-prone
- Switched to `pgx` which uses PostgreSQL's **binary wire format** by default
- Native encoding of data types — `arrays`, `jsonb`, `uuid`, `numeric` — no text conversion overhead
- Pure SQL queries over ORMs — we've always done this; full control over what runs against the database

Especially valuable for financial data where parsing precision matters.

Leveraging new Go release features

- Go 1.22: method-based routing in `http.ServeMux`
- Go 1.24: `go tool` directives — tools in `go.mod`, no `go install`
- Go 1.25: `t.Context()`
- Go 1.26: better `new(expr)`, `errors.AsType`, `slog.NewMultiHandler`, `signal.NotifyContext` now indicates which signal was received, `encoding/json/v2` (experimental), etc.

Eliminating Package-Level Globals

The globals problem

- Package-level `var` for loggers, stores, metrics, validators
- Implicit coupling, anything can read/write them
- Tests must save/restore globals (`t.Cleanup` and `defer` gymnastics)
- Data races when tests run with `-race`
- Impossible to test in parallel with `-parallel` and issues with `-count`

Result: correctness + made testing much easier/faster.

Before: global store

```
var DefaultBookStore BookStore

func copiesSold(ctx context.Context, bookID int64, since time.Time) (int, error) {
        // ...
        return DefaultBookStore.CountSales(ctx, bookID, since)
}

// app.go --- wiring via global assignment
func (app *Application) setupStore() {
        book.DefaultBookStore = app.store.book
}
```

After: struct field + constructor injection

```
type Service struct {
        store BookStore
        log   *slog.Logger
}

func NewService(store BookStore, log *slog.Logger) *Service {
        return &Service{store: store, log: log}
}

func (s *Service) copiesSold(ctx context.Context, bookID int64, since time.Time) (int, error) {
        // ...
        return s.store.CountSales(ctx, bookID, since)
}
```

No `Init()`, no global assignment, no `t.Cleanup` restore in tests.

Before: global logger with `Init()`

```
// bookmarks/package.go
var (
        log     = slog.Default()
        Metrics = metrics.NewDummy()
)
func Init(cfg config.Config) {
        log = slog.With("package", "bookmarks")  // mutation at runtime
}

func (b *Bookmarks) Save(ctx context.Context, remarks []*Remarks) {
        for _, r := range remarks {
                if err := b.save(ctx, r.ID, r.Note); err != nil {
                        Metrics.Seen("bookmarks.remarks.fail", 1)  // global
                        log.Error("...")                           // global
                }
        }
}
```

After: injected logger and metrics

```
type Bookmarks struct {
        booksRepo       booksRepo
        metrics         types.Metrics   // injected
        log             *slog.Logger    // injected
}

func NewBookmarks(repo booksRepo, m types.Metrics, log *slog.Logger) *Bookmarks {
        return &Bookmarks{
                booksRepo:      repo,
                metrics:        m,
                log:            log.With(slog.String("package", "bookmarks")),
        }
}

func (b *Bookmarks) Save(ctx context.Context, remarks []*Remarks) {
        for _, r := range remarks {
                if err := b.save(ctx, r.ID, r.Note); err != nil {
                        b.metrics.Seen("bookmarks.remarks.fail", 1)    // explicit
                        b.log.Error("...")                             // explicit
                }
        }
}
```

The recipe (× n)

1. Identify the global: `var DefaultX = …`
2. Add a field to the struct that uses it
3. No constructor? Add a new constructor
4. Add a new constructor parameter
5. Replace all `GlobalX.Method()` with `s.x.Method()`
6. Delete the global and the `Init()` function
7. Update the `app` wiring to pass the dependency
8. `go vet ./...`, `go test -race ./...`, and `staticcheck ./...`

Each step is a *single, reviewable commit* that ships to production.

Fixing Data Races & Parallel Testing

Strategy: the race detector as a refactoring guide

1. Add `t.Parallel()` to tests ”one-by-one”
2. Run `go test -race` — watch it light up
3. Fix each race
4. Revert `t.Parallel()` only where truly unsafe
5. Eventually, add `-race` and then `-shuffle` to CI

Race #1: mocked time closure

Risk: closure (that doesn't need it) captures mutex-protected state.

```
// Before (race: f() called under RLock, but f accesses shared state)
func (r *Runner) async(f func(timeout time.Duration)) {
        r.m.RLock()
        timeout := r.getTimeout()
        defer r.m.RUnlock()
        if r.f != nil {
                return f(r.timeout)     // called while holding read lock
        }
        return nil
}
```

```
// After
func (r *Runner) async() {
        r.m.RLock()
        timeout := r.getTimeout()           // capture reference
        r.m.RUnlock()
        if f != nil {
                return f(timeout)               // call without lock
        }
        return nil
}
```

Race #2: slice append reallocation

Problem: `append()` can reallocate the backing array.

```
// Before (race: append may reallocate backing array)
func TimeWhenFieldChanged(details *BookDetails, fn fieldCheck) *time.Time {
        history := append(details.History, details)
        for _, v := range history {
                if fn(v) { return &v.LastModified }
        }
        return nil
}
```

```
// After (no allocation, no race)
func TimeWhenFieldChanged(details *BookDetails, fn fieldCheck) *time.Time {
        for _, v := range details.History {
                if fn(v) { return &v.LastModified }
        }
        if fn(details) { return &details.LastModified }
        return nil
}
```

Use `go vet ./...` and `staticcheck ./...` to help you spot such problems.

Race #3: concurrent callbacks

```
// Before (race: multiple callbacks append concurrently)
messages := make([]Message, 0)
notifications.Handler(func(ctx context.Context, msg Message) {
        // ...
        messages = append(messages, parsed)   // RACE
})
```

```
// After (protected with mutex)
var (
        messages []Message
        mu       sync.Mutex
)
notifications.Handler(func(ctx context.Context, msg Message) {
        // ...
        mu.Lock()
        messages = append(messages, parsed)
        mu.Unlock()
})
```

Continuous Integration enforcement

Add to your CI workflows:

- `go test -race ./internal/...` on unit tests
- `go test -shuffle on -race ./integrationtests`
- `go vet ./...` and `staticcheck ./...`
- `govulncheck ./...` for known vulnerabilities

Any new race condition now fails the pipeline.

Integration tests made refactoring possible

Mocks wouldn't catch the bugs we introduced while moving code around. Integration tests against a real PostgreSQL instance did.

```
//go:embed migrations/*.sql
var migrations embed.FS

func TestBookmarkStore(t *testing.T) {
        t.Parallel()
        conn := sqltest.Quick(t, migrations) // temp DB + migrations
        store := postgres.NewBookmarkStore(conn)

        _, err := store.Add(t.Context(), "book-1", Bookmark{
                Page: 42, Note: "important passage",
        })
        if err != nil {
                t.Fatalf("unexpected error: %v", err)
        }
}
```

`sqltest.Quick` creates a temporary database, runs migrations, and cleans up — one line of setup.

Mocks and spies still have their place

- Integration tests verify the *happy path* against real infrastructure
- Mocks/fakes are still valuable to simulate **failure scenarios**: database timeouts, connection errors, constraint violations
- **Spies** (see `/gen:spy`) wrap the real implementation — delegate to the database by default, override individual methods when needed
- This gives you the best of both worlds: real behavior in most tests, controlled failures where it matters

End-to-end integration tests

The `integrationtests/` package tests the full application over HTTP.

- **Dual mode**: probes a Unix socket to detect a running instance; if none is found, starts the entire app *in-process* (ephemeral mode)
- **Strictly local**: multiple checks in `TestMain` enforce that tests only run against local development databases — never production or staging
- **Self-contained**: each test gets a clean state automatically; helper functions let tests read like high-level scenarios
- **Same binary**: tests exercise the real application code, HTTP routing, middleware, and database queries — not a simplified mock version

Comparing structs with google/go-cmp

```
var dog = Animal{Name: "Dog", Class: "Mammal", Sound: "Bark"}

func TestAnimals(t *testing.T) {
        got := fetchAnimal(t.Context(), "dog")
        if diff := cmp.Diff(dog, got); diff != "" {
                t.Errorf("animal mismatch (-want +got):\n%s", diff)
        }
}
```

When a field doesn't match, you get a readable diff:

```
  Animal{
      Name:  "Dog",
-     Class: "Mammal",
+     Class: "Reptile",
      Sound: "Bark",
  }
```

When tests fail, you immediately see *which* field is wrong and *how*.

Test output

```
$ go test
--- FAIL: TestAnimals (0.00s)
    code_test.go:29:
            Error Trace:    animalia_test.go:29
            Error:          Not equal:
                            expected: animalia.Animal{Name:"Dog",
                              Class:"Mammal", Sound:[]string{"Bark"}}
                            actual  : animalia.Animal{Name:"Gecko",
                              Class:"Reptile", Sound:[]string{"Click"}}

                            Diff:
                            --- Expected
                            +++ Actual
                            @@ -1,6 +1,6 @@
                             (animalia.Animal) {
                            - Name: (string) (len=3) "Dog",
                            - Class: (string) (len=6) "Mammal",
                            + Name: (string) (len=5) "Gecko",
                            + Class: (string) (len=7) "Reptile",
                              Sound: ([]string) (len=1) {
                            -  (string) (len=4) "Bark"
                            +  (string) (len=5) "Click"
                              }
            Test:           TestAnimals
FAIL
exit status 1
```

Improving Error Handling

Common problems with error handling

1. **Swallowed errors**: DB functions logging errors but returning `err = nil`
2. **Lost error chains**: `fmt.Errorf("cannot read: %s", err)`
   Fix: `fmt.Errorf("cannot read: %w", err)`
3. **Generic 500s**: upstream HTTP errors all become “Internal Server Error” or worse.
4. **Leaking errors**: internal details exposed to clients — security risk.
5. **Not tracking errors**: no trace IDs, no way for users to report specific failures.

Fix #1: errors that carry HTTP status codes

```go
// types/error.go --- errors carry their intended HTTP status
type errorCode struct {
        statusCode int
        err        error
}
func (e errorCode) StatusCode() int { return e.statusCode }
func (e errorCode) Unwrap() error   { return e.err }

func Error(code int, err error) error {
        return errorCode{statusCode: code, err: err}
}

// Usage: errors declare their HTTP intent at definition
var ErrNotFound = types.Error(http.StatusNotFound, errors.New("query returned no results"))
var ErrNoPermission = types.Error(http.StatusForbidden, errors.New("insufficient permissions"))
```

Unrecognized error? Use a generic Internal Server Error.

Fix #2: RFC 7807 Problem Details

```
func (h Handler) Error(w http.ResponseWriter, r *http.Request, err error) {
        code := getStatusCode(err)       // walks Unwrap() chain
        if code == http.StatusInternalServerError {
                h.InternalServerError(w, r, "", err)
                return
        }
        problem := &api.ProblemDetails{
                Type:   "/problems/" + slugify(http.StatusText(code)),
                Title:  http.StatusText(code),
                Status: code,
                Detail: err.Error(),
        }
        Problem(w, problem)
}
```

Clients get structured JSON errors with a type URI, not a plain string.

RFC 7807 in practice

```go
type ProblemDetails struct {
        Type       string         // URI identifying the problem type
        Title      string         // short summary (stable across occurrences)
        Status     int            // HTTP status code
        Detail     string         // human-readable, specific to this occurrence
        Instance   string         // URI for this specific occurrence
        Violations []Violation    // optional: which fields failed validation
}
```

Example response (`application/problem+json`):

```json
{
    "type":   "https://example.com/problems/not-found",
    "title":  "Not Found",
    "status": 404,
    "detail": "query returned no results"
}
```

Fix #3: stop swallowing database errors

```go
// Before --- error logged and silently discarded
func (s BookStore) LookupByIDs(ctx context.Context, ids []string) map[ProductDir]*Book {
        rows, err := s.db.Query(ctx, query, ids)
        if err != nil {
                slog.Error("error during LookupByIDs")   // swallowed!
        }
        return byProdCat   // may be nil or partial
}
```

```go
// After --- error returned to caller
func (s BookStore) LookupByIDs(ctx context.Context, ids []string) (map[ProductDir]*Book, error) {
        rows, err := s.db.Query(ctx, query, ids)
        if err != nil {
                return nil, fmt.Errorf("BookStore.LookupByIDs: %w", err)
        }
        return byProdCat, nil
}
```

Streamlining AI in the Workflow

AI as a code-quality amplifier

- **Not**: “write my code for me”
- **Instead**: “a force multiplier to take my architecture further”

Create custom Claude skills or Gemini CLI commands. Like:

- `/gen:api` — update Swagger/OpenAPI annotations across handlers
- `/gen:spy` — generate test spy implementations for interfaces

AGENTS.md — project conventions

The AI reads project rules before generating code:

- No new Go modules without explicit approval
- Use `go vet`, `staticcheck`, don't bother with linters
- No testify — use standard `testing` + `google/go-cmp`
- Integration tests preferred over unit tests + mocks
- Use `http.ServeMux`, not routing libraries
- Metrics naming convention enforced
- *"Clear is better than clever"* — Go proverb

The AI should follow the same code review standards as the team.

/gen:spy — generated test spy pattern

```go
// Package spy is maintained using Gemini CLI with /gen:spy
// DO NOT EDIT MANUALLY UNLESS YOU MUST.

type BookmarkStore struct {
        DB                    postgres.BookmarkStore
        FindByAuthorFunc    func(ctx context.Context, author string) (Remark, error)
        findByAuthorCounter atomic.Uint32
}

func (s *BookmarkStore) FindByAuthor(ctx context.Context, author string) (Remark, error) {
        s.findByAuthorCounter.Add(1)
        if s.FindByAuthorFunc != nil {
                return s.FindByAuthorFunc(ctx, author)  // custom mock
        }
        return s.DB.FindByAuthor(ctx, author)        // real impl fallback
}
```

/gen:api — unified OpenAPI from two specs

Before: two separate `.spec` files, divergent styles, manual upkeep. **After**:

- Merged two OpenAPI specs into one using annotations with (`swaggo/swag`)
- Changed a handler? Ask the AI to update the annotations to match
- Consistent response schemas, error formats, and naming across all endpoints

One source of truth: the code *is* the API documentation.

/gen:api — the command definition

`.gemini/commands/gen/api.toml`:

```
description = "Update API annotations and generate API docs"
prompt = """
You are an expert Go developer specialized in API documentation.
Your task is to update the Swagger/OpenAPI documentation for our REST API.
The documentation is generated from annotations in the Go source code using 'swag'.

When you identify changes in REST API endpoint handlers,
you must update the corresponding 'swag' annotations.
Ensure your annotations follow the 'swag' specification precisely.
Pay attention to things like:
- `@Summary`, `@Description`, `@Param`, `@Success`, `@Failure`, `@Router`.
- Correctly defining request and response structures.
- Data types and formats.

The user will provide the files that have changed.
Your focus should be on the files provided in `{{args}}`.
If no arguments are provided, focus on the package `internal/entrypoints/rest`.

After updating the annotations, run `make api` to regenerate the OpenAPI specification and documentation.

Do not perform any other code changes.
"""
```

Claude Code equivalent: a skill

`.claude/commands/gen-api.md`:

```
Update the Swagger/OpenAPI annotations for our REST API.
The documentation is generated from annotations in Go source
code using 'swag'.

When you identify changes in REST API endpoint handlers,
update the corresponding 'swag' annotations.
Pay attention to @Summary, @Description, @Param, @Success,
@Failure, @Router.

Focus on the files provided in $ARGUMENTS.
If no arguments are provided, focus on
`internal/entrypoints/rest`.

After updating the annotations, run `make api`.

Do not perform any other code changes.
```

Invoke with `/gen-api` or `/gen-api internal/entrypoints/rest`

/gen:api — annotated handler example

```
// AddBookmark godoc
// @Summary      Add a bookmark to a book
// @Tags         bookmarks
// @Accept       json
// @Produce      json
// @Param        bookID path string true "Book ID"
// @Param        body body AddBookmarkRequest true "Bookmark"
// @Success      201 {object} Bookmark
// @Failure      400 {object} api.ProblemDetails
// @Failure      404 {object} api.ProblemDetails
// @Router       /bookmarks/{bookID} [post]
func (h Handler) AddBookmark(w http.ResponseWriter, r *http.Request) {
        bookID := r.PathValue("bookID")
        var req AddBookmarkRequest
        if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
                h.Error(w, r, types.Error(http.StatusBadRequest, err))
                return
        }
        bookmark, err := h.service.Add(r.Context(), bookID, req)
        if err != nil {
                h.Error(w, r, err)
                return
        }
        h.ServeJSON(w, http.StatusCreated, bookmark)
}
```

Step 1: you change the code

Add two fields to the request struct:

```
type AddBookmarkRequest struct {
        Page    int      `json:"page"`
        Note    string   `json:"note"`
        Section string   `json:"section"`  // new
        Tags    []string `json:"tags"`     // new
}
```

Then run `/gen:api` — the AI reads the code and updates the annotations.

Step 2: AI updates the annotations

```
  // AddBookmark godoc
  // @Summary      Add a bookmark to a book
```

```
+ // @Description  Creates a bookmark with optional section and tags
```

```
  // @Tags         bookmarks
  // @Accept       json
  // @Produce      json
  // @Param        bookID path string true "Book ID"
  // @Param        body body AddBookmarkRequest true "Bookmark"
  // @Success      201 {object} Bookmark
  // @Failure      400 {object} api.ProblemDetails
  // @Failure      404 {object} api.ProblemDetails
  // @Router       /bookmarks/{bookID} [post]
```

Barely no manual doc maintenance. Change the code → let the AI catch up.

Project Structure & Other Improvements

Restructuring project layout

Before

- `cmd/bookstore/main.go`
- `cmd/bookstore/config.go`
- `pkg/bookstore/bookmarks/`
- REST endpoints in multiple packages
- Cronjobs scheduled on services + database

After

- `main.go` (root level)
- `internal/app/`
- `internal/config/`
- `internal/bookcollection/`
- `internal/entrypoints/rest/`
- `internal/entrypoints/cronjobs/`
- `internal/entrypoints/subscription/`
- Cronjobs using Kubernetes

Clear separation: entrypoints → business logic → entities

Domain types extracted into a shared `internal/entities/` layer to mitigate circular dependency hacks

Improvements shipped incrementally

- **Metrics**: replaced `uber-go/tally` with Prometheus library
- **Cronjobs**: in-app scheduling → Kubernetes CronJobs
- **Tracing**: OpenTelemetry on HTTP clients and servers
- **Context**: fixed cancellation in reading flow
- **Shutdown**: Unix socket-based graceful shutdown
- **Integration tests**: in-process infra, auto-discover address
- **Profiling**: `pprof` locally and in production via Cloud Profiler
- **Observability**: Kubernetes alerts, health check endpoints
- **CI pipeline**: consolidated stages, faster feedback loop
- **PostgreSQL**: `database/sql` → `pgx` with powerful native binary protocol

Results & Lessons Learned

Results

- **103,000 lines deleted** (net) — the codebase *shrank*
- **Zero data races** in CI (enforced by `-race` flag)
- **Zero greenfield rewrites** — every commit shippable
- **5 major dependencies removed** — replaced by stdlib or smaller libs
- **23+ package-level globals eliminated**
- **Structured errors** with RFC 7807 across all API endpoints
- **AI-maintained** API docs and test spies reduce toil
- **Tests run faster** and are more reliable (in parallel, no shared state)
- **6 contributors** participated in the modernization effort

Lessons learned

1. **Small commits win.** Each change is reviewable and revertible.
2. **The race detector is a refactoring tool.** Add `-race` early.
3. **Globals are the root of all evil.** Dependency Injection in Go is just a struct field and a constructor parameter.
4. **AI works best with clear rules.** Write your conventions down.
5. **Good practices are contagious.** Once the pattern is visible in the codebase, others follow.
6. **You don't need a rewrite.** You need a direction and discipline.

Key takeaways

1. Embrace the standard library
2. Remove globals, inject dependencies
3. Let `-race` guide your refactoring
5. Ship incremental improvements, every week
6. Let AI amplify good practices, not replace them

Resources

- **henvic.dev/posts/testing-go**
  On testing Go code using the standard library
- **github.com/henvic/pgxtutorial**
  Tutorial: building a Go service with PostgreSQL and gRPC/HTTP (post)
- **github.com/henvic/httpretty**
  HTTP client/transport inspector for debugging REST requests
- **github.com/henvic/pgtools**
  Helpers for PostgreSQL and Go using pgx, tern, and scany; including pgtools/sqltest
- **github.com/henvic/pgq**
  Query builder for PostgreSQL (if you insist on using one)

Thank you!

Henrique Vicente de Oliveira Pinto

`henvic.dev`

bol × ING Tech Talks — Amsterdam