Lightweight HTTP Framework for Go

Independent project · 2020–2021 · Solo · Public repository · Open source (MIT)

A small HTTP framework for building REST APIs in Go: a radix-tree router with path parameters and groups, request binding, several response formats and per-route middleware.

Links: Repository (dipra) · Go package docs

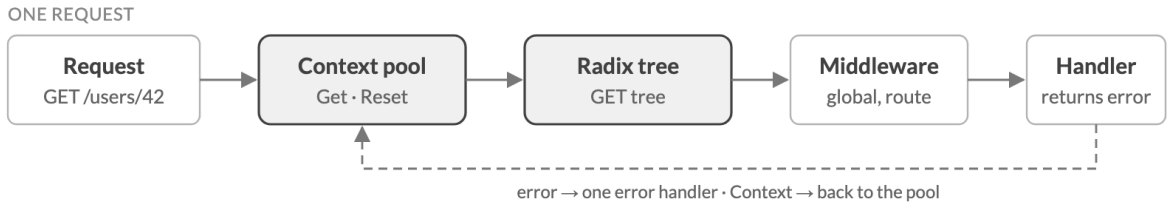
Why

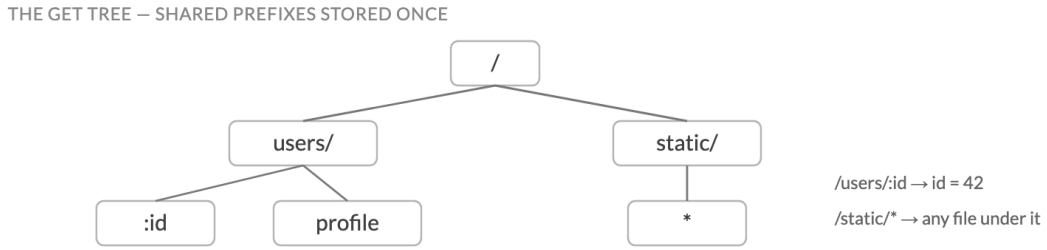
I used Go web frameworks every day without knowing what happens between a request arriving and my handler running. I built one from the standard library up to learn how routing, request contexts and middleware chains work, and how to keep them fast.

What it does

- Routes GET, POST, PUT, PATCH and DELETE, with path parameters (:id), wildcards (*) and route groups.
- Binds JSON, XML and YAML bodies and query strings into structs.
- Responds with text, JSON, JSONP, XML, files and redirects, and serves static folders.
- Runs middleware globally, per group or per route; ships logger, CORS and panic recovery.

Key flow





1. Routes are inserted into a radix tree per method; shared prefixes are stored once.
2. A request takes a Context from a pool and resets it with the new request and response writer.
3. The router walks the tree and collects path parameters. Route middleware was wrapped once at registration; global middleware wraps the handler per request.
4. The handler returns an error or nil; the Context goes back to the pool.

Key decisions

Decision	Why / trade-off
Radix tree router	Lookup walks shared prefixes instead of checking every route one by one. The insert and split code is harder to read than a list of patterns.
Context reused through sync.Pool	No new Context per request; in the benchmark, a request through route middleware allocates nothing. The trade-off: a handler must not keep the Context after it returns.
Handlers return an error	Errors flow back to one place instead of every handler writing its own error response.
Middleware as func(HandlerFunc) HandlerFunc	One shape for global, group and route middleware, so they compose by plain wrapping.
Built on net/http	The engine is an http.Handler, so it runs on the standard server and works with existing net/http tooling.

Non-goals

- A full-stack framework: no templates, ORM or dependency injection.
- Replacing mature frameworks in production; this was built to learn how they work.

Outcome

- Eight versions (v1.0.0 to v1.0.7) published to the Go module proxy, with nine runnable examples.
- Benchmarks: about 180 ns and one allocation for a request through a route, and about 42 ns with no allocation through route middleware.
- Tests pass, with 88% coverage on the middleware package and 37% on the core.
- Lesson: the cost a framework adds per request is mostly allocation; pooling the Context keeps the middleware path allocation-free.

Code pointers

- tree.go: the radix tree: insert, split and lookup with parameters.
- dipra.go: the engine, the Context pool and ServeHTTP.
- benchmark_test.go: the route and middleware benchmarks.

Stack

Go · net/http · sync.Pool · encoding/json · encoding/xml · Travis CI · Codecov