Gin is a very bad software library

A software article by Efron Amber Licht.

December 2025

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

1. Prelude - The software equivalent of Athlete's Foot

In my experience, Go is the best general-purpose programming language for backend development, and a huge part of this comes from the thoughtful design of it's standard libraries. If you are willing to be a little bit patient, read the documentation, and spend some time getting familiar with their idioms, you have everything you need without needing to go far afield.

Most programmers are not willing to be a little bit patient. They google 'go web framework' and then they pick the first web result. More than likely, this is Gin, a kind of insidous fungus masquerading as a software library.

Like many fungi,

- It is easy to pick up Gin and almost impossible to remove it
- Unless you're extremely careful you'll pass it on to your friends
- The features that make it inimical to human life are directly related to it's success, and it will likely outlive you and me despite everything we do to eradicate it.
- You can learn to live with it, but you shouldn't have to.

Gin is not the only bad library - in fact, it's not nearly the worst library in common usage - but it is the library that *pisses me off* the most day to day, and I think it's emblematic of many of the biggest flaws of software library design writ large.

- Gin is a very bad software library
  - 1. Prelude - The software equivalent of Athlete's Foot
    - 1.1. Tablesetting & Caveats
  - 2. Comparison of Basic Servers in `net/http` and `gin`
    - 2.0.1. basic server: `net/http`
    - 2.1. Basic Server: Gin
  - 3. HTTP is not that complicated: a brief review
    - 3.1. Diagram: Basic HTTP Flow
    - 3.2. HTTP Requests
      - 3.2.1. Chalkboard Diagram: HTTP Request Example
      - 3.2.2. Chalkboard Diagram: HTTP Request Structure
    - 3.3. HTTP Responses
      - 3.3.1. Chalkboard Diagram: HTTP Response Structure
      - 3.3.2. Chalkboard Diagram: HTTP Response Status Line
    - 3.4. A couple of structural notes about HTTP
    - 3.5. Gin is significantly larger than the problem domain
  - 4. "minimal" interfaces
    - 4.0.1. Diagram: minimal interface for `net/http` server
    - 4.0.2. Diagram: "minimal" interface for `gin` sever and kafkaesque nightmare:
  - 5. Choosing Gin: Human beings are bad at making judgements (especially about things they consider boring)
  - 6. Gin is too big: Code & Binary Bloat
    - 6.1. Table: Comparison of Gin and other Frameworks, Programs, Libraries etc.
    - 6.2. Gin includes too much code even if you don't use the features
    - 6.3. Sidenote: `go build -tags nomsgpack`
  - 7. Gin's API has the surface area of an industrial heat sink and sucks nearly as much
    - 7.1. `net/http` is a beautiful API
      - 7.1.1. net/http: interface summary and graph
    - 7.2. Investigating Gin's Core API
      - 7.2.1. `*gin.Engine`
      - 7.2.2. `*gin.RouterGroup`
      - 7.2.3. `gin.HandlerFunc` and `gin.Context`
      - 7.2.4. `*gin.Context` has more methods than Jerry Seinfeld has cars
      - 7.2.5. A list of methods that must be seen to be believed
      - 7.2.6. What if we just want to write JSON?
      - 7.2.7. A chart of just indirections in Gin's JSON handling
      - 7.2.8. The worst of both worlds
      - 7.2.9. Small nitpicks about `render`

- 8. Gin's documentation is very bad
  - 8.0.1. gin.RouterGroup: example (bad) documentation
  - 8.0.2. `net/http.ServeMux`: example (good) documentation
- 9. The Spider's Web
- 10. Conclusion and Advice on Software Dependencies
  - 10.1. What if I already use Gin?

1.1. Tablesetting & Caveats

Before we begin:

- This article is mostly rant, for which I apologize in advance.
- Gin is a *very old* library as far as Go goes and is based off an even older one, go-martini. Many of it's worst mistakes are artifacts of that time, and some things that *appear* to be mistakes are an artifact of predating that functionality in the standard library.
- I shouldn't have to say this, but you are not a bad person if you use, have used, or work on Gin: please do not use this article as a way to brigade people for using "unfashionable" code. People making software decisions based off 'clout' is part of how we got into this mess.
- Update: I am **not** trying to imply that you should never use libraries for backend development or that using something besides `net/http` is somehow "wrong". I have happily used libraries like `gorilla/mux` and it's associated friends and I see nothing wrong with, for example, `chi`. I am using the standard library as a comparison because it's
  - What I use
  - A very good library (not perfect, but *very good*).
  - A dependency every Go programmer has by definition
- I *am* saying that Gin is a bad library, and that you should be suspicious of libraries that share it's flaws.

2. Comparison of Basic Servers in `net/http` and `gin`

OK, let's get to it. On a surface level, basic HTTP work doesn't look too different in `net/http` and Gin:

2.0.1. basic server: `net/http`

```
1   func main() {
2       // route METHOD / PATH to handlers: here, GET /ping.
3       mux := http.NewServeMux()
4         mux.HandleFunc("GET /ping", func(w
    http.ResponseWriter, r *http.Request) {
5           w.WriteHeader(200)
6           json.NewEncoder(w).Encode(map[string]string {
```

```
            "message": "pong",
        })
    })
    // Create a HTTP server...
    srv := http.Server {
        Handler: mux, // that uses our router...
        Addr: ":8080", // on port 8080
    }
    srv.ListenAndServe()
}
```

2.1. Basic Server: Gin

```
func main() {
  // create a default gin router / server / engine
  r := gin.Default()
  // route METHOD / PATH to handlers: here, GET /ping.
  r.GET("/ping", func(c *gin.Context) {
    c.JSON(http.StatusOK, gin.H{
      "message": "pong",
    })
  })
  r.Run()
}
```

On a surface impression, `gin` **might seem easier**- it's slightly fewer lines of code, and there seems to be less configuration.

But this is all surface.

The proper way to judge a map is by the terrain it covers. In other words, before you choose any software, first you should know the problem you're trying to solve with it. So before we pick on Gin, let's review that terrain - HTTP.

3. HTTP is not that complicated: a brief review

Happily, HTTP is not that complicated and we can go over the basics in about ninety seconds and a handful of chalk drawings.

The **H**yper**T**ext **T**ransport **P**rotocol has a **client** send HTTP **Requests**, and a **server** responds with HTTP **Responses**.

A **client** sends a HTTP **Request** to a **server**. The server parses the request, figures out what the client wants, and sends back a HTTP **Response**.

> This is *very* quick and dirty. If you want more details on the structure of HTTP, my article series 'Backend from the Beginning' builds an entire HTTP library from scratch and goes over all these parts in detail.

3.1. Diagram: Basic HTTP Flow

diagram source
mermaid diagram showing basic HTTP request flow.

3.2. HTTP Requests

HTTP **Requests** have four main parts, separated by newlines:

1. A **Request Line** that specifies the HTTP method (GET, POST, etc), the path being requested, and the HTTP version.
2. One or more **Headers** that provide metadata about the request.
3. A blank line.
4. An optional **Body** that contains data being sent to the server (usually JSON).

I.E, they look like this:

3.2.1. Chalkboard Diagram: HTTP Request Example

literal chalkboard diagram showing an example HTTP request and it's parts.

3.2.2. Chalkboard Diagram: HTTP Request Structure

literal chalkboard diagram showing the structure of an HTTP request.

3.3. HTTP Responses

HTTP **Responses** have a similar structure, with four main parts, separated by newlines

3.3.1. Chalkboard Diagram: HTTP Response Structure

literal chalkboard diagram showing the structure of an HTTP response.

3.3.2. Chalkboard Diagram: HTTP Response Status Line

3.4. A couple of structural notes about HTTP

- These parts are **ordered** - you can't change your mind about the request or status line once you've sent them.
- Once you've sent the body, you (usually) can't send any more headers.

- You don't have to send the whole body at once on either side - you can stream it.
- You now know more about HTTP than many senior web developers. I wish this were not true.

Fundamentally, the structure of our solution - the HTTP library - should mirror the structure of the problem If the **solution** is significantly larger than the problem, one or more of the following is true:

3.5. Gin is significantly larger than the problem domain

The go stdlib's `net/http` covers all of HTTP in 35 files of pure go and 25,597 lines of code, including the server, client, TLS, proxies, etc.

Gin and it's dependency chain covers *only* server-side handling and requires 2,148 files and 1,032,635 lines of code, including 80084 lines of platform-specific GNU style assembly.

This is *nuts*. You *can* crack an egg with a 155mm artillery shell. This does not make it a good idea, even if you add rocket boosters and laser guidance.

4. "minimal" interfaces

Some people would argue that the *code weight* doesn't matter - what we should be worried about is the **API**: i.e, the interface we have to keep in our heads. They'd probably sneak in a quote about premature optimization or something. No problem.

The following diagrams illustrate the 'minimal' APIs to understand `net/http` and gin.

4.0.1. Diagram: minimal interface for `net/http` server

diagram source
mermaid diagram showing minimal interface for net/http server.

4.0.2. Diagram: "minimal" interface for `gin` sever and kafkaesque nightmare:

diagram source
mermaid diagram showing minimal interface for gin server and kafkaesque nightmare.

> As hard as it is to believe, this graph omits a ton of details.

---

5. Choosing Gin: Human beings are bad at making judgements (especially about things they consider boring)

If you're reading this, you're probably a programmer. Take a moment to think about how the dependencies were chosen for your current project(s). Ask yourself - or better yet, a team-mate - the following questions re: your major dependencies:

- What *are* your major dependencies?
- Who chose to add them to your project? Why? When?
    - Did they write those decisions down anywhere?
    - If so, did they ever go back to re-evaluate those decisions?
- How many options did they evaluate?
    - What was the evaluation process?
- Was one of those options 'write it ourselves'?
    - If so, why didn't you do it?
    - If not, why didn't you consider it?
- What are the perceived strengths and weaknesses in the following categories?
    - Familiarity
    - Performance (what kind?)
    - API surface
    - Documentation
    - Test coverage
    - Code bloat
        - Within the package
        - Within it's dependency tree
    - Security (did someone vet it? Did you read the code? *Can* you read the code? Does it rely on, say, opaque binaries or platform-specific assembly)?
- Are more features better or worse? Why? Is this always the same?
- **How hard will it be to switch if this decision is wrong?**
  > This final point is Gin's curse - it is *incredibly* difficult to remove - and, I think, the root of it's success. We'll come back to it in our final section.

For the vast majority of projects, there *is* no answer to these questions, because no one ever thought about it. They went into google search or chatgpt, typed "best golang web framework reddit" and called it a day. I know this because I have seen it happen at least twenty times at half a dozen software houses. While *understandable* - software is a busy, stressful, job - this is not *acceptable*. This is the kind of reasoning you apply to choosing lunch, not critical software dependencies for million or billion-dollar projects.

6. Gin is too big: Code & Binary Bloat

> In anything at all, perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away.
> ~Antoine De Saint Expry.

Gin is too big. Gin is enormously, staggeringly big. It's dependency tree is over 55MiB. If we *just* taking the lines of code in Gin and it's dependencies - ignoring comments and documentation - we have **877615** lines. This is huge, enormous, *elephantine* cost must be paid by every single project on every single `git clone` or `go build`, and some of that cost leaks into the compiled binary too.

Gin contains, I kid you not, ~~four~~ ~~five~~ at least *six* different JSON libraries, not counting the one built in to the standard library. (more about this later.)

These include

- `goccy/go-json` (1204K)
- `bytedance/sonic` (13 MiB!!!!)
- `quic-go/quic-go/qlogwriter/jsontext` (12 KiB - you pass)
- `ugorji/go/codec`(3MiB!!!,)
- `./github.com/quic-go/quic-go/qlogwriter/jsontext`
- `gabriel-vasile/mimetype/internal/json`
- `json-iterator/go` (348K)

> I thought there were only four, but I kept finding more.

6.1. Table: Comparison of Gin and other Frameworks, Programs, Libraries etc.

The following table compares the code bloat of Gin to other popular and/or historically important programs or written material.

Program or Library	Description	Files	Code Lines	%target	Size	%size
github.com/gin-gonic/gin	A popular go web framework and OSHA violation	2189	877615	100.000%	55.461 MiB	100.00%
lua	General-purpose scripting language a-la Python or Javascript	105	36685	4.180%	14.926 MiB	26.91%
chi	Minimalistic go HTTP framework	85	7781	0.887%	4.746 MiB	8.56%
	A best-selling real-time strategy game	1893	368288	41.965%		72.05%

Program or Library	Description	Files	Code Lines	%target	Size	%size
Command and Conquer: Red Alert	(1996) with it's own GUI, networking code, custom game engine, etc etc etc.				39.957 MiB	
DOOM	ID software's revolutionary first-person shooter, including networked play	152	39250	4.472%	2.375 MiB	4.28%
gorilla/mux	Popular go HTTP router.	19	6214	0.708%	1.059 MiB	1.91%
labstack/echo	Popular go web framework	600	326000	37.146%	23.855 MiB	43.01%
golang/go/src	The go programming language, it's runtime, tooling, and compiler	9591	2244096	255.704%	143.129 MiB	258.07%
MechCommander2-Source/	2001 real-time strategy game	1875	858811	97.857%	1.771 GiB	3269.17%
MS-DOS/v1.25/	Complete operating system, predecessor of microsoft windows	20	12001	1.367%	504.000 KiB	0.89%
MS-DOS/v2.0/	"	116	41417	4.719%	2.527 MiB	4.56%
MS-DOS/v4.0	Final release of MS-DOS with true multitasking support	1065	332117	37.843%	23.203 MiB	41.84%
original-bsd/	The original berkley systems distribution operating system and hundreds of programs, libraries, and games	9562	1526953	173.989%	185.387 MiB	334.27%
Quake	ID software's third-person shooter, including 3d graphical engine, GUI, networking code, etc etc	516	170211	19.395%	15.266 MiB	27.53%
Research-Unix-v10/v10	Original 'research' unix before split into BSD and other	8755	1671269	190.433%	137.430 MiB	247.80%

Program or Library	Description	Files	Code Lines	%target	Size	%size
	distributions, including networking, productivity software, and games					
zig/src/	Systems programming language and tooling, including an entire C compiler for dozens of targets	175	473612	53.966%	24.094 MiB	43.44%
musl	implementation of core C library used by Linux and other operating systems	1922	64837	7.388%	9.199 MiB	0.16586
Bible (King James Version)	Popular Translation of the Jewish & Christian Core Religious Text	31104	—	—	4.436 Mib	—
War and Peace	Tolstoy's extremely long novel about the Napoleonic wars	23637	—	—	3.212 MiB	—

This is, to be blunt, completely unacceptable. If you picked a sane framework like chi (you don't need a framework, but for the sake of argument), you could bundle in DOOM, a C compiler to build it with (let's pick Zig), and an operating system to run it on like MS-DOS 4.0, and throw in War and Peace and the entire Kings James Bible for good measure and you'd *still have less bloat than Gin* and it's source tree.

This bloat carries over to the compiled binary, too.

6.2. Gin includes too much code even if you don't use the features

While Go's compiler is pretty good about eliminating unused code, Gin does it's best to touch as many different libraries as it can at import time so the compiler can't do that.

To demonstrate, let's strip down our examples even further and build the simplest possible Gin programs and an equivalent HTTP servers and see how big the resulting binaries are.

```
1  // simplegin.go
2  func main() {
3    e := gin.Default()
4    e.ANY("/", func(c *gin.Context) {
```

```
        c.Writer.WriteHeader(200)
      })
    e.Run()
}
```

```
// simplehttp/main.go
func main() {
    http.ListenAndServe(":8080", http.HandlerFunc(func(w
http.ResponseWriter, r *http.Request) {
        w.WriteHeader(200)
    }))
}
```

Let's examine the compiled output:

```
#!/usr/bin/env bash
du -H simplehttp simplegin

19640K  simplegin
7864K   simplehttp
```

Maybe it's just debug symbols? Let's strip the binaries and try again:

```
#!/usr/bin/env bash
strip simplehttp
strip simplegin
du -H simplehttp simplegin
13572K  simplegin
5444K   simplehttp
```

Where's all this bloat coming from? After all, we don't use most of Gin's features... Let's use `GODEBUG=inittrace=1` to see what packages are being initialized to see if we can figure out where all this bloat is coming from.

```
GODEBUG=inittrace=1 ./simplegin
```

```
init internal/bytealg @0.005 ms, 0 ms clock, 0 bytes, 0
allocs
init runtime @0.078 ms, 0.10 ms clock, 0 bytes, 0 allocs
init crypto/internal/fips140deps/cpu @0.63 ms, 0.003 ms
clock, 0 bytes, 0 allocs
init math @0.67 ms, 0 ms clock, 0 bytes, 0 allocs
... many, many lines omitted
```

There's a lot of noise here, so I'll summarize a couple highlights:

- You pay for `toml`, `gob`, `yaml`, `protobuf`, `xml`, and at least two JSON libraries, regardless of whether you use them:
    - init encoding/gob @1.6 ms, 0.087 ms clock, 26496 bytes, 395 allocs
    - init github.com/gin-gonic/gin/codec/json @1.8 ms, 0 ms clock, 0 bytes, 0 allocs

    - init github.com/goccy/go-yaml/token @1.8 ms, 0.006 ms clock, 3784 bytes, 18 allocs
    - init github.com/goccy/go-yaml/printer @1.8 ms, 0 ms clock, 0 bytes, 0 allocs
    - init github.com/goccy/go-yaml/parser @1.8 ms, 0 ms clock, 336 bytes, 2 allocs
    - init github.com/pelletier/go-toml/v2 @1.8 ms, 0 ms clock, 0 bytes, 0 allocs
    - init google.golang.org/protobuf/reflect/protoreflect @2.0 ms, 0 ms clock, 0 bytes, 0 allocs
    - init google.golang.org/protobuf/reflect/protoregistry @2.0 ms, 0 ms clock, 88 bytes, 3 allocs
    - init google.golang.org/protobuf/proto @2.0 ms, 0 ms clock, 80 bytes, 2 allocs
    - init encoding/json @1.7 ms, 0.005 ms clock, 32 bytes, 2 allocs
    - init encoding/xml @1.7 ms, 0.016 ms clock, 18776 bytes, 6 allocs
    - init github.com/pelletier/go-toml/v2 @1.8 ms, 0 ms clock, 0 bytes, 0 allocs

- you pay for `http/3 (QUIC)` even if you aren't using it

    - init github.com/gabriel-vasile/mimetype @2.1 ms, 0.022 ms clock, 20024 bytes, 243 allocs
    - init github.com/quic-go/quic-go/internal/protocol @2.6 ms, 0.005 ms clock, 144 bytes, 3 allocs
    - init github.com/quic-go/quic-go/internal/utils @2.6 ms, 0 ms clock, 48 bytes, 1 allocs
    - init github.com/quic-go/quic-go/internal/wire @2.6 ms, 0 ms clock, 0 bytes, 0 allocs
    - init github.com/quic-go/quic-go/internal/handshake @2.6 ms, 0 ms clock, 32 bytes, 1 allocs

This cost is a direct result of Gin's horrific 'everything and the kitchen sink' API - more about that in a bit.

6.3. Sidenote: `go build -tags nomsgpack`

> As it turns out, the `gin` team *has* been trying to deal with this enormous binary bloat.
> You can eliminate the dependency on msgpack by adding the built tag `nomsgpack`, which shaves ten megabytes off the binary. This should be the *default*, but still, good job.

7. Gin's API has the surface area of an industrial heat sink and sucks nearly as much

> *Increasingly, people seem to misinterpret complexity as sophistication, which is baffling – the incomprehensible should cause suspicion rather than admiration.*

Niklaus Wirth, inventor of PASCAL

A quick note on UNIX before we dive into Gin's API.

UNIX is one of the oldest traditions in software still standing. In this tradition, good APIs have a small surface that exposes deep functionality. The classic example is UNIX's filesystem API, which made it a long way with only six verbs: OPEN, CLOSE, READ, WRITE, SEEK, and FCTNL - this is enough to handle disk drives, shared network filesystems, terminals, printers, etc.

> There's a good argument to be made that this is not the *correct* filesystem API anymore - FCTNL is clearly cheating, and it doesn't handle nonblocking or concurrent IO that well. See the excellent talk **What Unix Cost Us** by Benno Rice for a discussion of this topic.
>
> > For more on UNIX programming, see my Starting Systems Programming series of article.

Go is firmly part of this tradition, and as such, it's standard library tries to minimize API surface where possible. The vast, vast majority of interfaces in Go's standard library have three or fewer methods, usually just one. Even the largest interface in Go, `net.Conn` tops out at 8 methods. Gin... does not do this.

> `reflect.Type` doesn't count: it's never meant to be implemented by external libraries: all it's implementors are internal codegen, and reflection is *always* kind of an exception to every rule. Please don't @ me.

Let's take a look at how `net/http` is designed to see this philosophy in action.

7.1. `net/http` is a beautiful API

Server-side HTTP in go can be summarized in four types and one sentence: The `http.Server` parses packets into `http.Request` structs, hands them to a `http.Handler`, which writes a response via `http.ResponseWriter`.

> Usually, that handler is some kind of router like `http.ServeMux` that dispatches to different sub-handlers - but it doesn't have to be.

To give a quick example, here's a minimal HTTP server using only the Go standard library that responds to `POST /greet`.

While we use a number of types here, there's only a handful of interfaces we need to understand this code - `http.Handler`, `http.ResponseWriter`, and the omnipresent `io.Reader` and `io.Writer` interfaces used by the JSON encoder and decoder.

7.1.1. net/http: interface summary and graph

```
// 43 words of interface surface area, not counting comments
type Handler interface {
  ServeHTTP(w ResponseWriter, r *Request)
}
type ResponseWriter interface {
  WriteHeader(statusCode int)
  Header() Header
  Write([]byte) (int, error)
}
type Reader interface {
  Read(p []byte) (n int, err error)
}
type Writer interface {
  Write(p []byte) (n int, err error)
}
```

diagram source

7.2. Investigating Gin's Core API

To summarize Gin's API in a similar way, the `gin.Engine` gets http requests, routes them using it's embedded `gin.RouterGroup`, and turns them into a `*gin.Context`, which contains a `*http.Request` and a `gin.ResponseWriter`, and hands them to one or more `gin.HandlerFuncs`, which modify the `*gin.Context`.

This doesn't sound too bad - in fact, it sounds almost the same. Let's take a look at the method summaries of these types to see what we're dealing with here, starting with `gin.Engine`

7.2.1. `*gin.Engine`

```

    func (engine *Engine) Delims(left, right string) *Engine
    func (engine *Engine) HandleContext(c *Context)
    func (engine *Engine) Handler() http.Handler
    func (engine *Engine) LoadHTMLFS(fs http.FileSystem, patterns ...string)
    func (engine *Engine) LoadHTMLFiles(files ...string)
    func (engine *Engine) LoadHTMLGlob(pattern string)
    func (engine *Engine) NoMethod(handlers ...HandlerFunc)
    func (engine *Engine) NoRoute(handlers ...HandlerFunc)
    func (engine *Engine) Routes() (routes RoutesInfo)
    func (engine *Engine) Run(addr ...string) (err error)
    func (engine *Engine) RunFd(fd int) (err error)
    func (engine *Engine) RunListener(listener net.Listener) (err error)
    func (engine *Engine) RunQUIC(addr, certFile, keyFile string) (err error)
```

```
15      func (engine *Engine) RunTLS(addr, certFile, keyFile
    string) (err error)
16      func (engine *Engine) RunUnix(file string) (err error)
17      func (engine *Engine) SecureJsonPrefix(prefix string)
    *Engine
18      func (engine *Engine) ServeHTTP(w http.ResponseWriter,
    req *http.Request)
19      func (engine *Engine) SetFuncMap(funcMap
    template.FuncMap)
20      func (engine *Engine) SetHTMLTemplate(templ
    *template.Template)
21      func (engine *Engine) SetTrustedProxies(trustedProxies
    []string) error
22      func (engine *Engine) Use(middleware ...HandlerFunc)
    IRoutes
23      func (engine *Engine) With(opts ...OptionFunc) *Engine
```

This is a *mess*. This seems to cover

- routing and middleware, like we'd expect (`delims`, `NoMethod`, `NoRoute`, `Use`, `Routes()`)
- server configuration ('SetTrustedProxies', `RunTLS`, 'RunQUIC', `With`)
- and HTML templating ('SetHTMLTemplate', 'SetFuncMap', 'LoadHTMLGlob', `LoadHTMLGlob`, `LoadHTMLFS`, `LoadHTMLFiles`), and HTML templating. That is, it combines the concerns of `http.Server`, `http.ServeMux`, `template/html`(https://pkg.go.dev/html/template), not to mention entirely separate HTTP protocols like QUIC.

  BTW, for the ten thousand configuration options here, **none of them let you select the `http.Server` to use, so good luck if you want to do things like set timeouts or do connection or packet-level configuration. Gin is hardcoded to use the default HTTP server. I *think* you can do that by calling .Handler() and passing that to a `*http.Server`, but I'm not sure and it's not covered by the documentation. Maybe it's in `With`?

But that's not all - like I mentioned earlier, the gin.Engine **embeds a** `RouterGroup`. That means in addition to the previous, it *also* exposes the following methods:

7.2.2. `*gin.RouterGroup`

```
1       func (group *RouterGroup) Any(relativePath string,
    handlers ...HandlerFunc) IRoutes
2       func (group *RouterGroup) BasePath() string
3       func (group *RouterGroup) DELETE(relativePath string,
    handlers ...HandlerFunc) IRoutes
4       func (group *RouterGroup) GET(relativePath string,
    handlers ...HandlerFunc) IRoutes
5       func (group *RouterGroup) Group(relativePath string,
    handlers ...HandlerFunc) *RouterGroup
```

```
 6       func (group *RouterGroup) HEAD(relativePath string,
    handlers ...HandlerFunc) IRoutes
 7       func (group *RouterGroup) Handle(httpMethod,
    relativePath string, handlers ...HandlerFunc) IRoutes
 8       func (group *RouterGroup) Match(methods []string,
    relativePath string, handlers ...HandlerFunc) IRoutes
 9       func (group *RouterGroup) OPTIONS(relativePath string,
    handlers ...HandlerFunc) IRoutes
10       func (group *RouterGroup) PATCH(relativePath string,
    handlers ...HandlerFunc) IRoutes
11       func (group *RouterGroup) POST(relativePath string,
    handlers ...HandlerFunc) IRoutes
12       func (group *RouterGroup) PUT(relativePath string,
    handlers ...HandlerFunc) IRoutes
13       func (group *RouterGroup) Static(relativePath, root
    string) IRoutes
14       func (group *RouterGroup) StaticFS(relativePath
    string, fs http.FileSystem) IRoutes
15       func (group *RouterGroup) StaticFile(relativePath,
    filepath string) IRoutes
16       func (group *RouterGroup) StaticFileFS(relativePath,
    filepath string, fs http.FileSystem) IRoutes
17       func (group *RouterGroup)
    Use(middleware ...HandlerFunc) IRoutes
```

These methods cover routing - for some reason they present every HTTP verb as a separate method - and static file serving in in four different ways. All of these - except Group - return an `IRoutes` interface, and nearly all take a `HandlerFunc`.

7.2.3. `gin.HandlerFunc` and `gin.Context`

Ok, what's a `HandlerFunc`?

```
1   type HandlerFunc func(c *gin.Context)
```

Finally, a small interface. Maybe this is the equivalent of a `http.ResponseWriter`? Let's look at the exported interface (fields and methods) of `*gin.Context`.

```
1   type Context struct {
2       Request *http.Request
3       Writer  ResponseWriter // a gin.ResponseWriter, not a
    http.ResponseWRiter
4       Params Params
5       Keys map[any]any
6       Errors errorMsgs
7       Accepted []string
8       // contains filtered or unexported fields
9   }
```

So it contains the `http.Request` and a `gin.ResponseWriter`. What's a `gin.ResponseWriter`?

```
type ResponseWriter interface {
    http.ResponseWriter
    http.Hijacker
    http.Flusher
    http.CloseNotifier
    Status() int
    Size() int
    WriteString(string) (int, error)
    Written() bool
    WriteHeaderNow()
    Pusher() http.Pusher
}
```

7.2.4. `*gin.Context` has more methods than Jerry Seinfeld has cars

> *The bigger the interface, the weaker the abstraction*.
>
> **Rob Pike, "Go Proverbs".**

Ok, so stopping here, this already contains the entire API surface area of the `net/HTTP` interface - gin only adds complexity - as well as an extra five public fields and ten methods ON those fields.

That's not good, but the real horrors are yet to come:`gin.Context`'s list of methods.

> You might want to take a deep breath. You're not ready for this.

7.2.5. A list of methods that must be seen to be believed

```
// 133 functions. Do you have them all memorized? I sure hope so. https://pkg.go.dev/github.com/gin-gonic/gin#Context
    func (c *Context) Abort()
    func (c *Context) AbortWithError(code int, err error) *Error
    func (c *Context) AbortWithStatus(code int)
    func (c *Context) AbortWithStatusJSON(code int, jsonObj any)
    func (c *Context) AbortWithStatusPureJSON(code int, jsonObj any)
    func (c *Context) AddParam(key, value string)
    func (c *Context) AsciiJSON(code int, obj any)
    func (c *Context) Bind(obj any) error
    func (c *Context) BindHeader(obj any) error
    func (c *Context) BindJSON(obj any) error
    func (c *Context) BindPlain(obj any) error
    func (c *Context) BindQuery(obj any) error
    func (c *Context) BindTOML(obj any) error
    func (c *Context) BindUri(obj any) error
```

```
16      func (c *Context) BindWith(obj any, b
    binding.Binding) errordeprecated
17      func (c *Context) BindXML(obj any) error
18      func (c *Context) BindYAML(obj any) error
19      func (c *Context) ClientIP() string
20      func (c *Context) ContentType() string
21      func (c *Context) Cookie(name string) (string, error)
22      func (c *Context) Copy() *Context
23      func (c *Context) Data(code int, contentType string,
    data []byte)
24      func (c *Context) DataFromReader(code int,
    contentLength int64, contentType string, reader
    io.Reader, ...)
25      func (c *Context) Deadline() (deadline time.Time, ok
    bool)
26      func (c *Context) DefaultPostForm(key, defaultValue
    string) string
27      func (c *Context) DefaultQuery(key, defaultValue
    string) string
28      func (c *Context) Done() <-chan struct{}
29      func (c *Context) Err() error
30      func (c *Context) Error(err error) *Error
31      func (c *Context) File(filepath string)
32      func (c *Context) FileAttachment(filepath, filename
    string)
33      func (c *Context) FileFromFS(filepath string, fs
    http.FileSystem)
34      func (c *Context) FormFile(name string)
    (*multipart.FileHeader, error)
35      func (c *Context) FullPath() string
36      func (c *Context) Get(key any) (value any, exists
    bool)
37      func (c *Context) GetBool(key any) (b bool)
38      func (c *Context) GetDuration(key any) (d
    time.Duration)
39      func (c *Context) GetFloat32(key any) (f32 float32)
40      func (c *Context) GetFloat32Slice(key any) (f32s
    []float32)
41      func (c *Context) GetFloat64(key any) (f64 float64)
42      func (c *Context) GetFloat64Slice(key any) (f64s
    []float64)
43      func (c *Context) GetHeader(key string) string
44      func (c *Context) GetInt(key any) (i int)
45      func (c *Context) GetInt16(key any) (i16 int16)
46      func (c *Context) GetInt16Slice(key any) (i16s
    []int16)
47      func (c *Context) GetInt32(key any) (i32 int32)
48      func (c *Context) GetInt32Slice(key any) (i32s
    []int32)
49      func (c *Context) GetInt64(key any) (i64 int64)
50      func (c *Context) GetInt64Slice(key any) (i64s
    []int64)
51      func (c *Context) GetInt8(key any) (i8 int8)
52      func (c *Context) GetInt8Slice(key any) (i8s []int8)
```

```
53      func (c *Context) GetIntSlice(key any) (is []int)
54      func (c *Context) GetPostForm(key string) (string,
    bool)
55      func (c *Context) GetPostFormArray(key string)
    (values []string, ok bool)
56      func (c *Context) GetPostFormMap(key string)
    (map[string]string, bool)
57      func (c *Context) GetQuery(key string) (string, bool)
58      func (c *Context) GetQueryArray(key string) (values
    []string, ok bool)
59      func (c *Context) GetQueryMap(key string)
    (map[string]string, bool)
60      func (c *Context) GetRawData() ([]byte, error)
61      func (c *Context) GetString(key any) (s string)
62      func (c *Context) GetStringMap(key any) (sm
    map[string]any)
63      func (c *Context) GetStringMapString(key any) (sms
    map[string]string)
64      func (c *Context) GetStringMapStringSlice(key any)
    (smss map[string][]string)
65      func (c *Context) GetStringSlice(key any) (ss
    []string)
66      func (c *Context) GetTime(key any) (t time.Time)
67      func (c *Context) GetUint(key any) (ui uint)
68      func (c *Context) GetUint16(key any) (ui16 uint16)
69      func (c *Context) GetUint16Slice(key any) (ui16s
    []uint16)
70      func (c *Context) GetUint32(key any) (ui32 uint32)
71      func (c *Context) GetUint32Slice(key any) (ui32s
    []uint32)
72      func (c *Context) GetUint64(key any) (ui64 uint64)
73      func (c *Context) GetUint64Slice(key any) (ui64s
    []uint64)
74      func (c *Context) GetUint8(key any) (ui8 uint8)
75      func (c *Context) GetUint8Slice(key any) (ui8s
    []uint8)
76      func (c *Context) GetUintSlice(key any) (uis []uint)
77      func (c *Context) HTML(code int, name string, obj any)
78      func (c *Context) Handler() HandlerFunc
79      func (c *Context) HandlerName() string
80      func (c *Context) HandlerNames() []string
81      func (c *Context) Header(key, value string)
82      func (c *Context) IndentedJSON(code int, obj any)
83      func (c *Context) IsAborted() bool
84      func (c *Context) IsWebsocket() bool
85      func (c *Context) JSON(code int, obj any)
86      func (c *Context) JSONP(code int, obj any)
87      func (c *Context) MultipartForm() (*multipart.Form,
    error)
88      func (c *Context) MustBindWith(obj any, b
    binding.Binding) error
89      func (c *Context) MustGet(key any) any
90      func (c *Context) Negotiate(code int, config
    Negotiate)
```

```
 91      func (c *Context) NegotiateFormat(offered ...string)
     string
 92      func (c *Context) Next()
 93      func (c *Context) Param(key string) string
 94      func (c *Context) PostForm(key string) (value string)
 95      func (c *Context) PostFormArray(key string) (values
     []string)
 96      func (c *Context) PostFormMap(key string) (dicts
     map[string]string)
 97      func (c *Context) ProtoBuf(code int, obj any)
 98      func (c *Context) PureJSON(code int, obj any)
 99      func (c *Context) Query(key string) (value string)
100      func (c *Context) QueryArray(key string) (values
     []string)
101      func (c *Context) QueryMap(key string) (dicts
     map[string]string)
102      func (c *Context) Redirect(code int, location string)
103      func (c *Context) RemoteIP() string
104      func (c *Context) Render(code int, r render.Render)
105      func (c *Context) SSEvent(name string, message any)
106      func (c *Context) SaveUploadedFile(file
     *multipart.FileHeader, dst string, perm ...fs.FileMode)
     error
107      func (c *Context) SecureJSON(code int, obj any)
108      func (c *Context) Set(key any, value any)
109      func (c *Context) SetAccepted(formats ...string)
110      func (c *Context) SetCookie(name, value string,
     maxAge int, path, domain string, secure, httpOnly bool)
111      func (c *Context) SetCookieData(cookie *http.Cookie)
112      func (c *Context) SetSameSite(samesite http.SameSite)
113      func (c *Context) ShouldBind(obj any) error
114      func (c *Context) ShouldBindBodyWith(obj any, bb
     binding.BindingBody) (err error)
115      func (c *Context) ShouldBindBodyWithJSON(obj any)
     error
116      func (c *Context) ShouldBindBodyWithPlain(obj any)
     error
117      func (c *Context) ShouldBindBodyWithTOML(obj any)
     error
118      func (c *Context) ShouldBindBodyWithXML(obj any) error
119      func (c *Context) ShouldBindBodyWithYAML(obj any)
     error
120      func (c *Context) ShouldBindHeader(obj any) error
121      func (c *Context) ShouldBindJSON(obj any) error
122      func (c *Context) ShouldBindPlain(obj any) error
123      func (c *Context) ShouldBindQuery(obj any) error
124      func (c *Context) ShouldBindTOML(obj any) error
125      func (c *Context) ShouldBindUri(obj any) error
126      func (c *Context) ShouldBindWith(obj any, b
     binding.Binding) error
127      func (c *Context) ShouldBindXML(obj any) error
128      func (c *Context) ShouldBindYAML(obj any) error
129      func (c *Context) Status(code int)
```

```
130       func (c *Context) Stream(step func(w io.Writer) bool)
      bool
131       func (c *Context) String(code int, format string,
      values ...any)
132       func (c *Context) TOML(code int, obj any)
133       func (c *Context) Value(key any) any
134       func (c *Context) XML(code int, obj any)
135       func (c *Context) YAML(code int, obj any)
```

This is a nightmare. Even a 'simple' Gin server that only receives and sends HTTP with JSON bodies over `net/HTTP` is unavoidably linked to this enormous complexity.

7.2.6. What if we just want to write JSON?

Even if you "just" want to send and receive JSON, there are *eleven different ways to do this as methods on* `gin.Context` all of which behave differently depending on build tags and magically invoke multiple layers of struct validation, and some of which depend on the configuration of your `gin.Engine`, not to mention `.Writer.WriteString()` and `.Writer.Write()`.

```
 1       func (c *Context) AbortWithStatusJSON(code int,
     jsonObj any)
 2       func (c *Context) AbortWithStatusPureJSON(code int,
     jsonObj any)
 3       func (c *Context) AsciiJSON(code int, obj any)
 4       func (c *Context) BindJSON(obj any) error
 5       func (c *Context) IndentedJSON(code int, obj any)
 6       func (c *Context) JSON(code int, obj any)
 7       func (c *Context) JSONP(code int, obj any)
 8       func (c *Context) PureJSON(code int, obj any)
 9       func (c *Context) SecureJSON(code int, obj any)
10       func (c *Context) ShouldBindBodyWithJSON(obj any) error
11       func (c *Context) ShouldBindJSON(obj any) error
```

To pick a single example, to know the behavior of `SecureJSON` at runtime, I need to know, among other things

- Which of the six JSON libraries does was this built with? Am I sure my test environment has the same build tags as the deploy?
- Did the `gin.Engine` that's running this function - one that is not visible in the function signature of a `HandlerFunc` - set a `gin.SecureJSONPrefix`?

Status Headers are even more complex: there are 24 different ways to set a response header via methods of `.Context` or it's fields, including:

- `Context.Status()` (writes a status header)
- `Context.Writer.Status()` (READS a previously written status header - sometimes)
- `Context.Writer.WriteHeader()` (WRITES a status header, but not in a way where you can always retrieve the status header with `.Writer.Status()`, yes I

have run into this and I am salty)

As intimidating as these giant lists of methods are, it turns out, the vast majority of these methods are wrappers around the same core functionality. In fact, they're wrappers around the exact same functionality as `net/http.ResponseWriter`. Let's follow the ordinary `JSON` down the chain and figure out what's happening.

The .JSON() method calls the exported function WriteJSON, which calls `c.Render()`. This writes the status - by calling .Status() - which just wraps `http.ResponseWriter.WriteHeader`,
takes the interface `render.Render`, which calls the magic method `WriteContentType`,`render.Render()` on the magic exported global variable `codec/json.API` of type `json.Core`, which happens to be the conditionally-compiled empty struct `codec/json.jsonapi` then writes the marshaled bytes to the `http.ResponseWriter`.

> The magic exported global variable depends on your build tags. Usually, this is the stdlib's encoding/JSON.

That is, it's

```
1   b, _ := json.Marshal(obj)
2   w.Write(b)
```

With a lot of extra steps in between.

Writing the content type header is similarly convoluted.

JSON() calls render.Render.WriteContentType(), which does a vtable lookup to find `render.JSON.WriteContentType()`, which calls the ordinary function `writeContentType()`, which does a vtable lookup to find `.Header()` on the response writer, then sets the header in an ordinary way.

In case that all sounds a bit abstract - and it is - I've provided a handy chart for you.

7.2.7. A chart of just indirections in Gin's JSON handling

diagram source

Nothing inside the box labeled 'gin' does anything at all useful.

And again, this is just ONE of the ELEVEN different ways to send JSON responses in Gin. Most of them go through similar contortions. All of them have their own structs for some ungodly reason. We haven't even covered *requests!* (I meant to, but this article has taken me multiple full workdays already).

7.2.8. The worst of both worlds

This approach is godawful, somehow combining the *worst* of both runtime lookup (extra indirection and function calls) and conditional compilation. Both you and the compiler have to jump through multiple layers of indirection to figure out what is actually happening at runtime, for no benefit whatsoever. These extra layers serve merely to bloat the binary and confuse the programmer.

In the default case - the case for 99.5% of Gin's consumers, _you are doing the exact same thing as the standard library, but splitting the responsibility over a half dozen extra interfaces and types and hundreds of lines of code!

If you wanted to use a different JSON library, you could just... use that library!

All gin does is obscure the control flow, inculcating a sense of helplessness in the programmer and causing cache misses at runtime for no benefit whatsoever.

7.2.9. Small nitpicks about `render`

- Why the hell does `render` take a `http.ResponseWriter` and not just an io.Writer? Is it supposed to do something different from writing the body (e.g, modifying the headers?)
- On a similar note, why does `WriteContentType` take a whole `http.ResponseWriter`? Is it supposed to modify the body? It should take a `*http.Header`! Or maybe be the slightly more sane `interface { ContentType() string }` - or better yet, not exist at all!

8. Gin's documentation is very bad

Let's keep this section short. Gin's documentation is sparse at best. An illustrative example is `gin.RouterGroup`: despite it's enormous API, it's documentation is limited to a handful of sentences split between `gin.RouterGroup` itself and `gin.RouterGroup.Handle`.

8.0.1. gin.RouterGroup: example (bad) documentation

> RouterGroup is used internally to configure router, a RouterGroup is associated with a prefix and an array of handlers (middleware).
> ...
> Handle registers a new request handle and middleware with the given path and method. The last handler should be the real handler, the other ones should be middleware that can and should be shared among different routes. See the example code in GitHub. *(Note: no link is provided!)*

> For GET, POST, PUT, PATCH and DELETE requests the respective shortcut functions can be used.
> This function is intended for bulk loading and to allow the usage of less frequently used, non-standardized or custom methods (e.g. for internal communication with a proxy).

8.0.2. `net/http.ServeMux`: example (good) documentation

On the other hand, `http.ServeMux`'s documentation is nearly a thousand words, **not counting in-documentation examples**, split into five sections: Patterns, Precedence, Trailing-slash redirection, Request sanitizing, and Compatibility. I encourage you to click on the two above links and take a look for yourself.

9. The Spider's Web

None of this is the worst part of Gin. The worst part is this: going from a `http.Handler` to a gin handler is trivial. You can write an adapter to go FROM the standard library TO gin in a single line.

```
1   func adaptHandler(h http.Handler) func(c *gin.Context) {
    return func(c *gin.Context) {return
    h.ServeHTTP(c.ResponseWriter, c.Request)}}
```

Going from a Gin handler to an ordinary http.Handler is functionally impossible - the only practical to do it is to dig into the code, figure out what it's actually trying to do, and rip out all of the indirection.

If you're still early enough in your software project, this is practical - if you're months or years deep into a legacy codebase, you don't have a chance in hell.

If a single person on your team gets the bright idea to use Gin, you're more or less stuck. You can work _around _ it, but it will be lurking at the bottom of your server, a giant chain of dependencies that you can never really get rid of.

This, I think, is the secret to Gin's success. It's attractive enough and popular enough to attract the trendhoppers and the naive, and tolerable enough for them to stick with it long enough to get stuck, and, like restaurants, most people use software because **other people are already using it**. Worse yet, *because* it's so difficult and painful to move away, users of Gin make the wrong conclusion that this is because **other libraries are hard**, and they sing the praises of their jailers. Maybe flies on the web do the same thing.

10. Conclusion and Advice on Software Dependencies

Gin is a bad software library and we as developers should stop using things like it. The purpose of this essay is not *really* to talk about Gin - it's to use it as an illustrated example of what is **bad** in software libraries rather than good.

The choice of what library, if any, to use is an engineering decision, not just a matter of opinion. It has concrete effects on the process of writing code and the resulting programs. While taste is *part* of the decision, it should not be the primary or only one. Gin and libraries like it will make your software worse. Stop using them.

I'll finish off with some advice on picking dependencies

- Figure out what the problem is before you reach for a solution.
- The size of a solution should be proportional to the size of the problem.
- **READ THE CODE AND DOCUMENTATION FOR YOURSELF**.
- The cost of a library is the cost of a library *and it's dependencies*, not just the parts you can see.
- All things being equal, choose the library with fewer features.

10.1. What if I already use Gin?

If you're not in deep, try and rip it out. If it's already spread deep into your codebase, the best you can do is probably containment.

- Make a policy to allow *no new Gin*.
- Use ordinary `net/http` handlers instead where possible for any future work, even if there's still Gin there.
- If and when you split off services, force the point of split to leave Gin behind.

<<byline placeholder