Mat Ryer Follow
Founder at MachineBox.io — Gopher, developer, speaker, author — BitBar app https://getbitbar.com — Author of Go Programming Blueprints
May 9 · 6 min read

How I write Go HTTP services after seven years

I've been writing Go (Golang when not spoken) since r59—a pre 1.0 release—and have been building HTTP APIs and services in Go for the past seven years.

At Machine Box, most of my technical work involves building various APIs. Machine Learning is complicated and inaccessible to most developers, so my job is to tell a simple story through the API endpoints, and we've had great feedback so far.

> *If you haven't witnessed the Machine Box developer experience yet, please give it a go and let me know what you think.*

The way I have written services has changed over the years, so I wanted to share how I write the services today—in case the patterns are useful to you and your work.

A server struct

All of my components have a single `server` structure that usually ends up looking something like this:

```
type server struct {
    db     *someDatabase
    router *someRouter
    email  EmailSender
}
```

- Shared dependencies are fields of the structure

routes.go

I have a single file inside every component called `routes.go` where all the routing can live:

```
package app
```

```
func (s *server) routes() {
    s.router.HandleFunc("/api/", s.handleAPI())
    s.router.HandleFunc("/about", s.handleAbout())
    s.router.HandleFunc("/", s.handleIndex())
}
```

This is handy because most code maintenance starts with a URL and an error report—so one glance at `routes.go` will direct us where to look.

Handlers hang off the server

My HTTP handlers hang off the server:

```
func (s *server) handleSomething() http.HandlerFunc { ... }
```

Handlers can access the dependencies via the `s` server variable.

Return the handler

My handler functions don't actually handle the requests, they return a function that does.

This gives us a closure environment in which our handler can operate:

```
func (s *server) handleSomething() http.HandlerFunc {
    thing := prepareThing()
    return func(w http.ResponseWriter, r *http.Request) {
        // use thing
    }
}
```

The `prepareThing` is called only once, so you can use it to do one-time per-handler initialisation, and then use the `thing` in the handler.

Be sure to only **read** the shared data, if handlers are modifying anything, remember you'll need a mutex or something to protect it.

Take arguments for handler-specific dependencies

If a particular handler has a dependency, take it as an argument.

```
func (s *server) handleGreeting(format string)
http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        fmt.Fprintf(w, format, "World")
    }
}
```

The `format` variable is accessible to the handlers.

HandlerFunc over Handler

I use `http.HandlerFunc` in almost every case now, rather than `http.Handler`.

```
func (s *server) handleSomething() http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        ...
    }
}
```

They are more or less interchangeable, so just pick whichever is simpler to read. For me, that's `http.HandlerFunc`.

Middleware are just Go functions

Middleware functions take an `http.HandlerFunc` and return a new one that can run code before and/or after calling the original handler—or it can decide not to call the original handler at all.

```
func (s *server) adminOnly(h http.HandlerFunc)
http.HandlerFunc {
    return func(w http.ResponseWriter, r *http.Request) {
        if !currentUser(r).IsAdmin {
            http.NotFound(w, r)
            return
        }
        h(w, r)
    }
}
```

The logic inside the handler can optionally decide whether to call the original handler or not—in the example above, if `IsAdmin` is `false`, the handler will return an HTTP `404 Not Found` and return (abort); notice that the `h` handler is **not** called.

If `IsAdmin` is `true`, execution is passed to the `h` handler that was passed in.

Usually I have middleware listed in the `routes.go` file:

```
package app
```

```
func (s *server) routes() {
    s.router.HandleFunc("/api/", s.handleAPI())
    s.router.HandleFunc("/about", s.handleAbout())
```

```
    s.router.HandleFunc("/", s.handleIndex())
    s.router.HandleFunc("/admin",
s.adminOnly(s.handleAdminIndex))
}
```

Request and response types can go in there too

If an endpoint has its own request and response types, usually they're only useful for that particular handler.

If that's the case, you can define them inside the function.

```
func (s *server) handleSomething() http.HandlerFunc {
    type request struct {
        Name string
    }
    type response struct {
        Greeting string `json:"greeting"`
    }
    return func(w http.ResponseWriter, r *http.Request) {
        ...
    }
}
```

This declutters your package space and allows you to name these kinds of types the same, instead of having to think up handler-specific versions.

In test code, you can just copy the type into your test function and do the same thing. Or…

Test types can help frame the test

If your request/response types are hidden inside the handler, you can just declare new types in your test code.

This is an opportunity to do a bit of storytelling to future generations who will need to understand your code.

For example, let's say we have a `Person` type in our code, and we reuse it on many endpoints. If we had a `/greet` endpoint, we might only care about their name, so we can express this in test code:

```
func TestGreet(t *testing.T) {
    is := is.New(t)
    p := struct {
        Name string `json:"name"`
    }{
        Name: "Mat Ryer",
    }
    var buf bytes.Buffer
    err := json.NewEncoder(&buf).Encode(p)
    is.NoErr(err) // json.NewEncoder
    req, err := http.NewRequest(http.MethodPost, "/greet",
&buf)
    is.NoErr(err)

    //... more test code here
```

It's clear from this test, that the only field we care about is the `Name` of the person.

sync.Once to setup dependencies

If I have to do anything expensive when preparing the handler, I defer it until when that handler is first called.

This improves application startup time.

```
func (s *server) handleTemplate(files string...)
http.HandlerFunc {
    var (
        init sync.Once
        tpl  *template.Template
        err  error
    )
    return func(w http.ResponseWriter, r *http.Request) {
        init.Do(func(){
            tpl, err = template.ParseFiles(files...)
        })
        if err != nil {
            http.Error(w, err.Error(),
http.StatusInternalServerError)
            return
        }
        // use tpl
```

```
    }
}
```

`sync.Once` ensures the code is only executed one time, and other calls (other people making the same request) will block until it's finished.

- The error check is outside of the `init` function, so if something does go wrong we still surface the error and won't lose it in the logs
- If the handler is not called, the expensive work is never done—this can have big benefits depending on how your code is deployed

> *Remember that doing this, you are moving the initialisation time from startup, to runtime (when the endpoint is first accessed). I use Google App Engine a lot, so this makes sense for me, but your case might be different so it's worth thinking about where and when to use sync.Once in this way.*

The server is testable

Our server type is very testable.

```
func TestHandleAbout(t *testing.T) {
    is := is.New(t)
    srv := server{
        db:    mockDatabase,
        email: mockEmailSender,
    }
    srv.routes()
    req, err := http.NewRequest("GET", "/about", nil)
    is.NoErr(err)
    w := httptest.NewRecorder()
    srv.ServeHTTP(w, r)
    is.Equal(w.StatusCode, http.StatusOK)
}
```

- Create a server instance inside each test—if expensive things lazy load, this won't take much time at all, even for big components
- By calling ServeHTTP on the server, we are testing the entire stack including routing and middleware, etc. You can of course call the handler methods directly if you want to avoid this

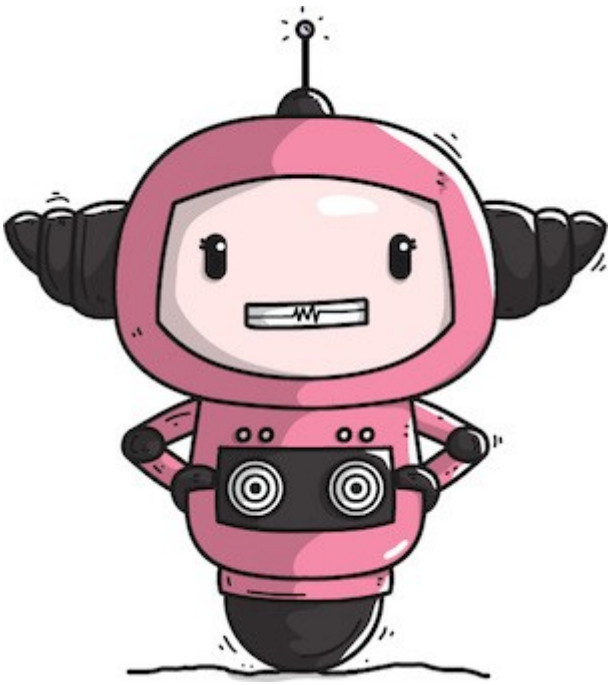
- Use `httptest.NewRecorder` to record what the handlers are doing
- This code sample uses my is testing mini-framework (a mini alternative to Testify)

Conclusion

I hope the items I covered in this post make sense, and help you in your work. If you disagree or have other ideas, please tweet me.

. . .

What is this Machine Box that I keep hearing all this amazing stuff about?



Ashley McNamara's creation—Machina; the Machine Box mascot

Machine Learning in Docker containers for Kubernetes—implement some ML today, without having to learn all that Tensorflow stuff.

If you'd like to learn more about Machine Box, check out our blog and website.

. . .