

[Backend from the Beginning, part 3: Databases, Dependency Injection, Middleware, and Routing](#)

A software article by Efron Amber Licht.

September 2023

[ALL ARTICLES](#)

[LICENSE](#)

[Feeds](#)

- [RSS](#)
- [ATOM](#)
- [JSON](#)

This is the third article in a series on backend development in Go that aims to teach **understanding** of how backend is put together, rather than just lego-like assembly of pre-built components. This article will try to be accessible on it's own, but it will be much easier to follow if you've read the [first](#) and [second](#) articles in the series.

A brief note before we begin: it's wild how well the first two articles went over - 60,000 views and counting from reddit alone! An encouraging sign, to say the least. Anyways, let's get to it.

- [Backend from the Beginning, part 3: Databases, Dependency Injection, Middleware, and Routing](#)
 - [1. Review](#)
 - [2. Databases](#)
 - [2.1. Connecting to a Database](#)
 - [2.2. database/sql](#)
 - [2.3. sql.Open](#)
 - [2.4. Sidenote: configuration](#)
 - [2.4.1. DB Example 1: dbping](#)
 - [2.5. Using *sql.DB](#)

- [3. Dependency Injection, or, “how do I put a database in my server?”](#)
- [4. client middleware](#)
 - [4.0.1. Building Client Middleware](#)
 - [4.1. Client Middleware without Dependencies](#)
 - [4.2. Client Middleware with Injected Dependencies](#)
 - [4.3. Context and Request Metadata](#)
 - [4.4. Tracing & Logging Requests \(client-side\)](#)
 - [4.5. Using Client Middleware](#)
 - [5.1. Tracing and Logging Requests \(server-side\)](#)
 - [5.2. Intercepting Writes to the Response](#)
 - [5.3. Recovering from Panics](#)
 - [5.4. Using Server Middleware](#)
- [6. Routing](#)
 - [6.1. simple routing](#)
 - [Note to the reader](#)
 - [6.2. Building a router with gorilla/mux-like syntax](#)
 - [6.2.1. Summary of API](#)
 - [6.2.2. Building the Router](#)
- [7. Graduation: Putting it all together](#)
 - [7.1. building our router](#)
 - [7.2. building & starting the server](#)
 - [7.3. demo](#)
- [8. Testing HTTP servers](#)
- [9. Conclusion](#)

[1. Review](#)

- In the [first article](#), we went through the components of the internet up to HTTP - TCP, IP, & DNS, and built our own HTTP library and basic servers.
- In the [second article](#), we went on a tour through the standard library’s networking packages and demonstrated how to build HTTP servers and clients in practice, rather than theory.
- In this third article, we’ll put the pieces together by covering our final missing pieces.
 - the `database/sql` package (how to connect to a database)
 - dependency injection for clients & servers (how to put a database in our server)
 - middleware (adding sophisticated behavior to our servers and clients, like authentication, logging, and tracing)
 - **routing** requests to the correct handler.

- The fourth article will be an opinion piece on software design and frameworks in the large. It may not come out; only if I think I can do it justice rather than make Yet Another Software Rant.

2. Databases

Sooner or later your backend project will need to store data that falls into one or more of the following categories:

- **Persistent** - data that needs to be stored for longer than the lifetime of a single process; i.e, across program or system restarts.
- **Shared** - data that needs to be accessed by multiple processes or systems.
- **Large** - data that's too large to keep in memory.

There are a variety of ways to solve these problems, including but not limited to:

- writing to files (either locally, using a networked filesystem, or something like aws s3)
- using a key-value store like redis or memcached
- using a traditional relational database like postgres or mysql

But traditional relational databases are by far the most common, so we'll focus on that for now. Specifically, we will use [PostgreSQL](#), a popular open-source relational database. SQL is too large of a topic to cover in this article, so I'll assume you have some knowledge and stick to the parts about integrating it with Go and backend development specifically. I hope to eventually do an article series covering SQL and databases more in-depth.

2.1. Connecting to a Database

Like most databases, **PostgreSQL** is a client-server application that uses the TCP/IP stack to communicate with clients. We covered the TCP/IP stack in the [first article](#), so we won't go into detail here. Unlike our servers so far, which used HTTP on top of that, postgres uses its own binary application-level protocol. If we wanted to communicate with postgres directly, the process would look broadly similar to the HTTP-by-hand we did in the [first article](#). Starting with a URL like "postgres://user:password@host:port/database", we'd do something like this:

- Use `net.ResolveTCPAddr` to resolve the hostname and port to an IP address.
- Use `net.DialTCP` to connect to the server.
- Spin up a goroutine to listen for responses from the server, and another to send requests to the server, communicating between them using channels or another synchronization primitive.

Unlike HTTP, however, the postgres protocol is not human-readable, and parsing the requests and responses would be a lot of work, so this time we'll skip straight to using libraries. Specifically, the `database/sql` package from the standard library.

[2.2. database/sql](#)

The `database/sql` package provides a unified interface for interacting with SQL databases. Each different database needs its own driver, which is registered at import time by convention. A couple of notes before we get started:

- Even though the overall interface is generic, your queries must be written in the SQL dialect of the database you're using; there's no abstraction layer for that. In particular, watch out for differences in parameter placeholders: postgres uses `$1`, `$2`, etc, while mysql uses `?`.
- Similarly, the error messages are specific to the database you're using and not particularly helpful. You'll need to read the documentation for your database to understand what they mean.

While Go provides a standard *interface* for interacting with databases, it does not provide *drivers* to make the actual connections and translate the requests and responses to and from the database's protocol. That's the domain of third-party drivers. The github.com/jackc/pgx/v5 provides such a driver for postgres.

If we want to use it, we'll need to import that package, like so:

```
1 import (  
2     "database/sql"  
3     _ "github.com/jackc/pgx/v5" // register the driver  
4 )
```

Note the `'_'`: this is a “blank import”, which means that we're importing the package for its side effects (that is, registering the driver).

[2.3. sql.Open](#)

The `sql.Open` function connects to a database and returns a `*sql.DB` object that we can use to interact with it. It takes two arguments: the name of the driver, and a connection string. The connection string is driver-specific, but for postgres it looks like this:

```
"postgres://user:password@host:port/database?sslmode=mode"
```

That is, it's a URL with the following required parameters:

- `user` - the username to connect with
- `password` - the password to connect with
- `host` - the hostname of the database server

- port - the port to connect to
- database - the name of the database to connect to

And a query parameter:

- mode - whether to use SSL to connect to the database. This is required for most hosted databases, but we'll disable it for now.

We covered URLs in detail in the [first article](#), so this should be pretty familiar.

By convention, postgres uses port 5432. Usually, you'll want to store either the entire connection string or the individual components in environment variables, so that you

- can change them without recompiling your program
- don't leak secrets like passwords into your source code or compiled binaries.

The following table shows the environment variables we'll use, and their corresponding components of the connection string:

env var	connection string component	note
PG_USER	user	
PG_PASSWORD	password	be careful not to leak this!
PG_HOST	host	
PG_PORT	port	integer in range 0-65535; 5432 for postgres by convention
PG_DATABASE	database	
PG_SSLMODE	mode	disable or require; optional

[2.4. Sidenote: configuration](#)

A brief note on configuration: as backend programs grow, they tend to accumulate a lot of configuration, which if not carefully managed can make programs fail in mysterious ways.

It's always good to let the user know which configuration knobs they're missing, rather than just failing with a cryptic error message. Configuration struggles are a common source of frustration for developers: spending a little bit of time early on error messages will save you a lot of time in the long run.

I've written a handy library for environment variables, [enve](#) for this purpose, but for now, we'll do it by hand: the concept is easy.

Installing and configuring databases can be a bit tricky, so for the purpose of this article, we'll use the wonderful [fergusstrange/embedded-postgres](#) to stick the database directly in our binary. This obviously isn't suitable for production, since you'll

have no persistent storage, but it's great for testing and development, and means that my examples will work right out of the box for you on a variety of platforms.

The following complete program, `dbping`, sets up an embedded postgres database and connects to it, pinging it to make sure we can connect.

2.4.1. DB Example 1: dbping

```
1 // dbping.go
2 package main
3
4 import (
5     "context"
6     "database/sql"
7     "flag"
8     "fmt"
9     "io"
10    "log"
11    "os"
12    "sort"
13    "strconv"
14    "time"
15
16    embeddedpostgres "github.com/fergusstrange/embedded-
17    postgres" // embedded postgres server.
18    _ "github.com/jackc/pgx/
19    v5" // register the db
20    driver
21
22    func main() {
23        timeout := flag.Duration("timeout", 5*time.Second,
24        "timeout for connecting to postgres")
25        flag.Parse()
26
27        cfg, err := pgConfigFromEnv() // defined below
28        if err != nil {
29            log.Fatalf("postgres configuration error: %v",
30            err)
31        }
32        // ---- setup embedded postgres server ----
33        portN, err := strconv.Atoi(cfg.port)
34        if err != nil {
35            panic(err)
36        }
37
38        // we'll mirror the postgres config in the environment
39        // so that you can't actually get it 'wrong' when running
40        // this example; you do need to set the environment
41        // variables, though.
42        embeddedCfg := embeddedpostgres.DefaultConfig().
```

```

38     Username(cfg.user).
39     Password(cfg.password).
40     Database(cfg.database).
41     Port(uint32(portN)).
42     Logger(io.Discard) // discard embedded postgres'
logs; they're not helpful for this example
43
44     embeddedDB :=
embeddedpostgres.NewDatabase(embeddedCfg)
45     if err := embeddedDB.Start(); err != nil {
46         panic(err)
47     }
48     log.Printf("postgres is running on: %s\n",
embeddedCfg.GetConnectionString())
49     defer embeddedDB.Stop() // if we don't stop the
database, it will continue running after our program exits
and block the port.
50
51     // ---- connect to postgres ----
52
53     db, err := sql.Open(
54         "postgres",
55         cfg.String(), // defined below
56     )
57     if err != nil {
58         panic(err)
59     }
60     defer db.Close() // always close the database when
you're done with it.
61
62     // always ping the database to ensure a connection is
made.
63     // any time you talk to a DB, use a context with a
timeout, since DB connections could be lost or delayed
indefinitely.
64     ctx, cancel :=
context.WithTimeout(context.Background(), *timeout)
65     defer cancel()
66     if err := db.PingContext(ctx); err != nil {
67         panic(err)
68     }
69     log.Println("ping successful")
70
71 }
72
73 // pgconfig is a struct that holds the configuration for
connecting to a postgres database.
74 // each field corresponds to a component of the connection
string.
75 // the following required environment variables are used
to populate the struct:
76 //
77 //     PG_USER
78 //     PG_PASSWORD

```

```

79 // PG_HOST
80 // PG_PORT
81 // PG_DATABASE
82 //
83 // additionally, the following optional environment
variable is used to populate the sslmode:
84 //
85 // PG_SSLMODE: must be one of "", "disable", "allow",
"require", "verify-ca", or "verify-full"
86 type pgconfig struct {
87     user, database, host, password, port string //
required
88     sslMode string //
optional
89 }
90
91 func pgConfigFromEnv() (pgconfig, error) {
92     var missing []string
93     // small closures like this can help reduce code
duplication and make intent clearer.
94     // you generally pay a small performance penalty for
this, but for configuration, it's not a big deal;
95     // you can spare the nanoseconds.
96     // i prefer little helper functions like this to a
complicated configuration framework like viper, cobra,
envconfig, etc.
97     get := func(key string) string {
98         val := os.Getenv(key)
99         if val == "" {
100             missing = append(missing, key)
101         }
102         return val
103     }
104     cfg := pgconfig{
105         user:     get("PG_USER"),
106         database: get("PG_DATABASE"),
107         host:      get("PG_HOST"),
108         password: get("PG_PASSWORD"),
109         port:     get("PG_PORT"),
110         sslMode:
os.Getenv("PG_SSLMODE"), // optional, so we don't add it
to missing
111     }
112     switch cfg.sslMode {
113     case "", "disable", "allow", "require", "verify-ca",
"verify-full":
114         // valid sslmode
115     default:
116         return cfg, fmt.Errorf(`invalid sslmode "%s":
expected one of "", "disable", "allow", "require",
"verify-ca", or "verify-full`, cfg.sslMode)
117     }
118
119     if len(missing) > 0 {

```

```

120         sort.Strings(missing) // sort for consistency in
        error message
121         return cfg, fmt.Errorf("missing required
        environment variables: %v", missing)
122     }
123     return cfg, nil
124 }
125
126 // String returns the connection string for the given
        pgconfig.
127 func (pg pgconfig) String() string {
128     s := fmt.Sprintf("postgres://%s:%s@%s:%s/%s",
        pg.user, pg.password, pg.host, pg.port, pg.database)
129     if pg.sslMode != "" {
130         s += "?sslmode=" + pg.sslMode
131     }
132     return s
133 }
134

```

Let's build and run it:

```

1 go build -o dbping ./dbping.go
2 ./dbping

```

OUT:

```

1 2023/09/17 09:52:45 missing 5 required environment
        variable(s): [PG_DATABASE PG_HOST PG_PASSWORD PG_PORT
        PG_USER]

```

Whoops; we forgot to set the environment variables. Good thing we added those error messages.

Let's try again:

```

1 PG_USER=postgres PG_PASSWORD=admin PG_HOST=localhost
        PG_PORT=5432 PG_DATABASE=postgres ./dbping

```

OUT:

```

1 panic: pq: SSL is not enabled on the server

```

SSL, or Secure Sockets Layer, is a protocol for encrypting network traffic; it's the **S** in HTTPS. (This article series won't get into SSL and HTTPS, but you should, on your own time.) Let's set the PG_SSLMODE environment variable to "disabled":

```

1 PG_USER=postgres PG_PASSWORD=admin PG_HOST=localhost
        PG_PORT=5432 PG_DATABASE=postgres PG_SSLMODE=disable ./
        dbping

```

OUT:

```


```

```
1 2023/09/17 10:14:14 postgres configuration error: invalid
sslmode "disabled": expected one of "", "disable", "allow",
"require", "verify-ca", or "verify-full"
```

... looks like it's *disable*, not *disabled*. One last time:

```
1 PG_USER=postgres PG_PASSWORD=admin PG_HOST=localhost
PG_PORT=5432 PG_DATABASE=postgres PG_SSLMODE=disable ./
dbping
```

OUT:

```
1 2023/09/17 10:15:19 postgres is running on: postgresql://
postgres:admin@localhost:5432/postgres
2 2023/09/17 10:15:19 ping successful
```

OK, looks good. Little configuration errors like this can easily stall a project for hours or days, so it's worth taking the time to make sure your error messages are clear and helpful. **If you run into a configuration error, take the time to add a message that guides your next user to the solution; after all, that next user might be you.**

[2.5. Using *sql.DB](#)

The following table summarizes the basic API of *sql.DB. Note that all methods take a context.Context as their first argument. **Never use the non-Context versions of these methods; they are a deprecated API. If you're not sure what context to use, use context.TODO().**

Method	Returns	Description	Use Cases
PingContext	error	Ping the database to ensure a connection is made.	Health check
ExecContext	Result, error	Execute a query that does not return rows.	Create, Update, Delete
QueryRowContext	Row	Execute a query that returns a single row.	Single item lookup
QueryContext	Rows, error	Execute a query that returns rows.	All other queries

I had a big section here where I demonstrated the APIs, but it quickly grew so big it completely overwhelmed the rest of the article, which is already over twice as long as part 2. Instead, I'll just point you to the [official docs for the database/sql package](#)

3. Dependency Injection, or, “how do I put a database in my server?”

So far when we’ve created http handlers, they’ve been self-contained: they don’t depend on anything outside of themselves; that is, they’re just `func(http.ResponseWriter, *http.Request)s`. But in the real world, we’ll want to access a database, cache, message queue, or other outside dependency from within our handlers.

The simplest and best way to handle this is to **pass the dependencies in as arguments** to a function that creates the handler.

That is, instead of:

```
1 // example: 'global' database connection
2 var db *sql.DB
3 func init() {
4     db, err := sql.Open("postgres", "...")
5     if err != nil {
6         panic(err)
7     }
8 }
9 func getUser(w http.ResponseWriter, r *http.Request) {
10     // ... parse & validate request...
11
12
13     if err := db.QueryRowContext(r.Context(),
14     "SELECT * FROM users WHERE id = $1", id).Scan(&user.ID,
15     &user.Name, &user.Email); err != nil {
16
17 }
```

inject the dependency:

```
1 // this function RETURNS a handler, rather than being a
2 // handler itself.
3 func getUser(db *sql.DB) http.HandlerFunc {
4     // db is now a local variable, rather than a global
5     // variable.
6
7     // this is the actual handler function, sometimes
8     // called a 'closure' since it "closes over" the db variable.
9     return func(w http.ResponseWriter, r *http.Request) {
10         // ... parse & validate request...
11         if err := db.QueryRowContext(r.Context(), "SELECT *
12         FROM users WHERE id = $1", id).Scan(&user.ID, &user.Name,
13         &user.Email); err != nil {
14
15             // ...
16         }
17     }
```

```
11     }
12 }
```

Alternatively, you can declare a struct containing the dependencies and let that struct implement the `http.Handler` interface:

```
1  type userHandler struct { db *sql.DB }
2  func (u userHandler) ServeHTTP(w http.ResponseWriter, r
   *http.Request) {
3      // ... parse & validate request...
4      if err :=u.db.QueryRowContext(r.Context(), "SELECT *
   FROM users WHERE id = $1", id).Scan(&user.ID, &user.Name,
   &user.Email); err != nil {
5          // ...
6      }
7 }
```

I recommend you stick with closures, since they are lighter-weight: the code is only in one place rather than two, and you don't need to create a new struct type for each handler.

We'll use dependency injection repeatedly throughout the rest of this article, especially while writing middleware.

[4. client middleware](#)

There's a lot of shared behavior we might want to add to many of our outgoing HTTP requests.

A few things that come to mind off the top of my head:

- Timing how long it takes for a request to complete
- Adding an authorization header
- Retrying failed requests with exponential backoff
- Collecting metrics on the number of requests made to each endpoint
- Logging failed requests
- Set `Accept-Encoding: gzip` header on requests to let the server know we can handle gzipped responses
- Transparently unzip responses sent with the `'Content-Encoding: gzip'` header (actually, go's http client does this for us already, but sshhh)
- etc, etc, etc

The total weight of this code is probably substantial. If we added all of these behaviors to simple routes that just make a GET request and unmarshal the response to JSON, our 'wrapper' code would quickly dwarf the actual business logic.

Another approach might be to make our own `DoRequest` function that would encapsulate all this behavior. This is certainly possible, though it gets rather complex. Here's what that might look like for a subset of the above behaviors:

```
1
2 // DoRequest is a helper function that sends the given
  request using the given client. It adds the following
  functionality:
3 //   - adds a context to the request
4 //   - adds an authorization header to the request
5 //   - retries the request up to 3 times if the server is
  unavailable or returns a 5xx status code
6 //   - returns an error if the server returns a 4xx status
  code
7 //   - logs the request duration
8 //
9 func DoRequest(ctx context.Context, c *http.Client, r
  *http.Request) (*http.Response, error) {
10     r = r.WithContext(ctx) // add context to request
11     // track execution time
12     start := time.Now()
13     defer func() { log.Printf("request took %s",
  time.Since(start)) }()
14
15     r = addAuthHeader(r) // add auth header to request
16
17     // retry logic
18     var retryErrs error
19     for retry := uint(0); retry < 3; retry++ {
20         if retry > 0 {
21             time.Sleep(10 * time.Millisecond << retry)
22         }
23         resp, err := c.Do(r)
24         if errors.Is(retryErrs, syscall.ECONNREFUSED) ||
  errors.Is(retryErrs, syscall.ECONNRESET) {
25             retryErrs = errors.Join(retryErrs, err)
26             continue
27         }
28         if retryErrs != nil {
29             return nil, fmt.Errorf("failed after %d
  retries: %w", retry, retryErrs)
30         }
31         switch sc := resp.StatusCode; {
32         case sc <= 200 && sc < 400:
33             return resp, nil // success! we're done here.
34         case sc <= 400 && sc < 500: // 4xx status code
35             return nil, fmt.Errorf("failed after %d
  retries: %s", retry, resp.Status)
36         default: // 5xx, 1xx, or unknown status code
37             retryErrs = errors.Join(retryErrs,
  fmt.Errorf("try %d: %s", retry, resp.Status))
38         }
39     }
```

```
40     }
41     return nil, fmt.Errorf("failed after 3 retries: %w",
    retryErrs)
42
43 }
```

Then we could simply replace `client.Do` with `DoRequest(client, r)`.

This has some advantages:

- Only one place to look
- Simple control flow
- Easy to add new functionality

But things get difficult very quickly if we want to be able to add *some* but not all of this functionality to a request. For example:

- Different routes might need different authorization headers (what if we're hitting two different services?)
- One route might need a longer timeout b/c it's known to be slow.
- Some routes must be rate limited to avoid overloading the server, but others are O.K. to hit as hard as we can.

What we really need is some kind of composability, where we can quickly apply *some* of the options to a client on an as-needed basis. We can build such a system by using *middleware*. Let's talk about how `http.Client` works first:

when we call `Client.Do`, the client sends a request to the server by calling the `RoundTrip` method on its `http.RoundTripper`, which is usually `http.DefaultTransport`. That `RoundTrip` method does all the low-level work of sending the request and receiving the response that we covered in the [first article](#) (though, admittedly, in a much more sophisticated way).

If we substituted out that `RoundTripper` for our own, we could intercept the request and modify it before it's sent to the server. We could also intercept the response and modify it before it's returned to the caller. We'd just have to make sure to eventually call the original `RoundTrip` method, so that the request actually gets sent to the server.

That's exactly what **middleware** does. Essentially, middleware "wraps" a client, sitting between it and the outside world. It modifies requests and responses as they pass through it, and can short-circuit the request/response cycle entirely.

Our desired API will look something like this:

```
1  var rt http.RoundTripper = http.DefaultTransport
2  rt = TimeRequest(rt)
3  rt = RetryOn5xx(rt, 10*time.Millisecond, 3)
4  rt = ...
5  client := &http.Client{
6      Timeout: 1 * time.Second,
7      Transport: rt,
```

```
8 }
```

[4.0.1. Building Client Middleware](#)

The RoundTripper interface looks like this:

```
1 type RoundTripper interface {
2     RoundTrip(*http.Request) (*http.Response, error)
3 }
```

We'll follow the example of the `http.HandlerFunc` and build a `RoundTripFunc` type that implements this interface:

```
1
2 // RoundTripFunc is an adapter to allow the use of ordinary
3 // functions as RoundTrippers, a-la http.HandlerFunc
4 type RoundTripFunc func(*http.Request) (*http.Response,
5 error)
6
7 // RoundTrip implements the RoundTripper interface by
8 // calling f(r)
9 func (f RoundTripFunc) RoundTrip(r *http.Request)
10 (*http.Response, error) {return f(r)}
11
12 var _ http.RoundTripper = RoundTripFunc(nil) // assert that
13 RoundTripFunc implements http.RoundTripper at compile time
```

[4.1. Client Middleware without Dependencies](#)

Let's build a package, `clientmw`, and implement a few simple middlewares. Each middleware will be a function that takes a `http.RoundTripper` and returns a `RoundTripFunc` that wraps it.

The most basic kind of middleware has no other arguments, and simply wraps the `RoundTripper` in a closure. Here's an example:

```
1 package clientmw
2
3 // we'll use this helper function to log the beginning and
4 // end of each middleware. no need for this in the real world,
5 // but it should help you understand what's going on.
6 func logExec(name string) func() {
7     log.Printf("middleware: begin %s", name)
8     return func() { defer log.Printf("middleware: end %s",
9 name) }
10 }
11 // TimeRequest returns a RoundTripFunc that logs the
12 // duration of the request.
13 func TimeRequest(rt http.RoundTripper) RoundTripFunc {
14     return func(r *http.Request) (*http.Response, error) {
```

```

12         // for demonstration purposes, we'll add these logs
13         to each middleware; don't do this in production!
14         defer logExec("TimeRequest")()
15
16         start := time.Now()
17         resp, err := rt.RoundTrip(r) // call next
18         middleware, or http.DefaultTransport.RoundTrip if this is
19         the last middleware
20         if err != nil {
21             log.Printf("%s %s: errored after %s",
22             r.Method, r.URL, time.Since(start))
23             return nil, err
24         }
25         log.Printf("%s %s: %d %s in %s", r.Method, r.URL,
26         resp.StatusCode, http.StatusText(resp.StatusCode),
27         time.Since(start))
28         return resp, nil
29     }
30 }

```

[4.2. Client Middleware with Injected Dependencies](#)

But generally speaking you'll want to pass some arguments to your middleware, *injecting* the dependencies.

For example, we might want to have configurable retries:

```

1  package clientmw
2
3  // RetryOn5xx returns a RoundTripFunc that retries the
4  // request up to n times if the server returns a 5xx status
5  // code.
6  // It will use exponential backoff: first retry will be
7  // after wait, second after 2*wait, third after 4*wait, etc.
8  func RetryOn5xx(rt http.RoundTripper, wait time.Duration,
9  tries int) RoundTripFunc {
10     // validate arguments OUTSIDE of the closure, so that
11     // it only happens once
12     if n <= 1 {
13         panic("n must be > 1")
14     }
15     if wait <= 0 {
16         panic("wait must be > 0")
17     }
18     return func(r *http.Request) (*http.Response, error) {
19         defer logExec("RetryOn5xx")()
20         // retry logic
21         var retryErrs error
22         for retry := uint(0); retry < tries; retry++ {
23             if retry > 0 {
24                 time.Sleep(wait << retry)
25             }

```

```

21         resp, err := rt.RoundTrip(r) // call next
middleware, or http.DefaultTransport.RoundTrip if this is
the last middleware
22         if errors.Is(retryErrs, syscall.ECONNREFUSED)
|| errors.Is(retryErrs, syscall.ECONNRESET) {
23             retryErrs = errors.Join(retryErrs, err)
24             continue
25         }
26         if retryErrs != nil {
27             return nil, fmt.Errorf("failed after %d
retries: %w", retry, retryErrs)
28         }
29         switch sc := resp.StatusCode; {
30         case sc <= 200 && sc < 400:
31             return resp, nil // success! we're done
here.
32         case sc <= 400 && sc < 500: // 4xx status code
33             return nil, fmt.Errorf("failed after %d
retries: %s", retry, resp.Status)
34         default: // 5xx, 1xx, or unknown status code
35             retryErrs = errors.Join(retryErrs,
fmt.Errorf("try %d: %s", retry, resp.Status))
36         }
37     }
38     }
39     return nil, fmt.Errorf("failed after 3 retries:
%w", retryErrs)
40 }
41 }

```

4.3. Context and Request Metadata

Most middleware modifies the context of the request; this allows later middlewares to access the values set by earlier ones. For example, we may wish to **trace** our requests, adding a unique ID to each request that will be associated with every log and carried from service to service via the headers. We can do this with a middleware that keeps track of a `Trace` struct in the context.

Additionally, we'll use the github.com/google/uuid package to generate unique IDs. I talk about uuids in some detail in my article on [simple byte hacking](#); don't worry about it for now.

```

1 package trace
2 import "github.com/google/uuid"
3 type Trace struct {
4     TraceID uuid.UUID // TraceID is unique across the
lifecycle of a single 'event', regardless of how many
requests it takes to complete. Carried in the `X-Trace-ID`
header.
5     RequestID uuid.UUID // RequestID is unique to each
request. Carried in the `X-Request-ID` header.

```

```
6 }
```

We'll use the following generic methods to add and retrieve values of a type from a context. See [my article on Go quirks and tricks, pt 1](#) for more details on how this works.

```
1 package ctxutil
2 type key[T any] struct{} // key is a unique type that we
  can use as a key in a context
3
4 // WithValue returns a new context with the given value
  set. Only one value of each type can be set in a context;
  setting a value of the same type will overwrite the
  previous value.
5 func WithValue[T any](ctx context.Context, value T)
  context.Context {
6     return context.WithValue(ctx, key[T] {}, value)
7 }
8 // Value returns the value of type T in the given context,
  or false if the context does not contain a value of type T.
9 func Value[T any](ctx context.Context) (T, bool) {
10    value, ok := ctx.Value(key[T] {}).(*T)
11    return value, ok
12 }
```

[4.4. Tracing & Logging Requests \(client-side\)](#)

Let's put together a Trace middleware that adds a Trace struct to the context and adds the X-Trace-ID and X-Request-ID headers to the request.

```
1 package clientmw
2 // Trace returns a RoundTripFunc that
3 // - adds a trace to the request context
4 // - generating a new one if necessary
5 // - adds the X-Trace-ID and X-Request-ID headers to the
  request
6 // - then calls the next RoundTripper
7 func Trace(rt http.RoundTripper) RoundTripFunc {
8     return func(r *http.Request) (*http.Response, error) {
9         defer logExec("Trace")()
10        // does the request already have a trace? if so,
  use it. otherwise, generate a new one.
11        traceID, err := uuid.Parse(r.Header.Get("X-Trace-
  ID"))
12        if err != nil {
13            traceID = uuid.New()
14        }
15
16        // build the trace. it's a small struct, so we put
  it directly in the context and don't bother with a pointer.
17        trace := trace.Trace{ TraceID: traceID, RequestID:
  uuid.New()}
18    }
```

```

19
20     ctx := ctxutil.WithValue(r.Context(),
    trace) // add trace to context; retrieve with
    ctxutil.Value[Trace](ctx)
21     r = r.WithContext(ctx) // add context to request
22
23     // add trace id & request id to headers
24     r.Header.Set("X-Trace-ID", trace.TraceID.String())
25     r.Header.Set("X-Request-ID",
    trace.RequestID.String())
26     return rt.RoundTrip(r) // call next middleware, or
    http.DefaultTransport.RoundTrip if this is the last
    middleware
27 }
28 }

```

Let's pick up this trace in the next middleware, one that adds a logger to our requests. We'll just use the standard library's unstructured [log](#) package for now.

Note: In practice you should probably use a structured logger. Both [rs/zerolog](#) and [uber-go/zap](#) are popular choices, and the standard library has recently introduced it's own structured logging package, [log/slog](#). I can happily recommend any of these. But for now, we'll dodge the question entirely and leave logging and metrics for a future article.

This will supersede our original `TimeRequest` middleware, so we'll add the timing logic here as well.

```

1  package clientmw
2  // Log returns a RoundTripFunc that logs the request
    duration and status code. It uses the trace from the
    context as a prefix, if it exists. See Trace in this
    package and servermw.Log for the server-side
    implementation.
3  func Log(rt http.RoundTripper, log *log.Logger)
    RoundTripFunc {
4      return func(r *http.Request) (*http.Response, error) {
5          defer logExec("Log")()
6          trace, ok := ctxutil.Value[Trace](r.Context())
7          if ok {
8              prefix := fmt.Sprintf("%s %s: [%s %s]: ",
    r.Method, r.URL, trace.TraceID, trace.RequestID)
9          } else {
10             prefix := fmt.Sprintf("%s %s: ", r.Method,
    r.URL)
11         }
12
13         logger := log.New(os.Stderr, prefix, log.LstdFlags
    | log.Lshortfile)
14         ctx := ctxutil.WithValue(r.Context(), logger) //
    add logger to context; retrieve with
    ctxutil.Value[log.Logger](ctx)
15         r = r.WithContext(ctx) // add context to request

```

```

16
17     start := time.Now()
18     resp, err := rt.RoundTrip(r) // call next
middleware, or http.DefaultTransport.RoundTrip if this is
the last middleware
19     if err != nil {
20         logger.Printf("errored after %s: %s",
time.Since(start), err)
21         return nil, err
22     }
23     logger.Printf("%d %s in %s", resp.StatusCode,
http.StatusText(resp.StatusCode), time.Since(start))
24     return resp, nil
25 }
26 }

```

4.5. Using Client Middleware

Using our middleware is simple. We just wrap the `http.DefaultTransport` with our middleware, and use it to build a new `http.Client`. It's important to note that middleware is applied "first-in, last-out"; that is, the first middleware we apply will be the last one to run, and the last middleware we apply will be the first one to run!

```

1 func clientMiddleware() http.RoundTripper {
2     var rt RoundTripFunc // specify the type as a
RoundTripFunc, not a http.RoundTripper, so that we don't
have to repeatedly wrap it in RoundTripFunc(rt)
3     const wait, tries = 10 * time.Millisecond, 3
4     // first middleware applied will be the last one to run.
5     rt = clientmw.RetryOn5xx(http.DefaultTransport, wait,
tries) // retry on 5xx status codes
6     rt =
clientmw.Log(rt) // log request duration and status code;
uses trace from next middleware
7     rt = clientmw.Trace(rt) // add trace id to request
header
8     return rt
9 }

```

Let's test this out. The following full program, `clientmiddlewareex` makes a GET request to the specified URL, and prints the response body to `stdout`, using our middleware.

```

1 // clientmiddlewareex makes a GET request to the specified
URL, and prints the response body to stdout, using our
middleware.
2 package main
3
4 import (
5     "context"
6     "io"

```

```

7     "log"
8     "net/http"
9     "os"
10    "time"
11
12    "gitlab.com/efronlicht/blog/articles/backendbasics/cmd/
clientmiddlewareex/clientmw"
13    )
14
15
16    func main() {
17        if len(os.Args) < 2 {
18            log.Fatal("target url required")
19        }
20        target := os.Args[1]
21        client := &http.Client{Transport: clientMiddleware(),
Timeout: 5 * time.Second}
22        req, err := http.NewRequestWithContext(context.TODO(),
"GET", target, nil)
23        resp, err := client.Do(req)
24        if err != nil {
25            log.Fatal(err)
26        }
27        defer resp.Body.Close()
28        io.Copy(os.Stdout, resp.Body)
29    }

```

IN:

```
1 go run clientmiddlewareex.go https://eblog.fly.dev
```

OUT:

```

1 2023/09/12 06:43:53 middleware: begin trace
2 2023/09/12 06:43:53 middleware: begin log
3 2023/09/12 06:43:53 middleware: begin retryOn5xx
4 2023/09/12 06:43:53 middleware: end retryOn5xx
5 GET https://eblog.fly.dev/index.html: [8c63dfffb-2901-4ebc-
bd7c-73ea843f89e2 9a56e7e8-062f-42db-b087-7018cd6a3610]:
2023/09/12 06:43:53 clientmw.go:103: 200 OK in 88.321876ms
6 2023/09/12 06:43:53 middleware: end log
7 2023/09/12 06:43:53 middleware: end trace

```

Checking my server logs, I note that the request was received with the following headers:

```

1 X-Request-Id: 9a56e7e8-062f-42db-b087-7018cd6a3610
2 X-Trace-Id: 8c63dfffb-2901-4ebc-bd7c-73ea843f89e2

```

Looks like everything is working as expected. Let's move on to **server middleware**.

A brief final note on client middleware: the [documentation for RoundTripper](#) says that it shouldn't modify the request or response. I disagree with this;

it's simpler and easier to intercept the RoundTripper than to build another layer *on top* of **http.Client**. Over the years, this seems to be the consensus for backend development in Go. If you disagree, you can always build your own layer **on top** of http.Client that wraps it's Do () method instead, like so:

```
1 > type HTTPDoer interface { Do(*http.Request)
  (*http.Response, error) }
2 > type HTTPDoerFunc func(*http.Request)
  (*http.Response, error)
3 > func (f HTTPDoerFunc) Do(r *http.Request)
  (*http.Response, error) { return f(r) }
4 > var _ HTTPDoer =
  HTTPDoerFunc(nil) // assert that HTTPDoerFunc
  implements HTTPDoer at compile time
5 > var _ HTTPDoer = (*http.Client)(nil) // assert
  that http.Client implements HTTPDoer at compile
  time
6 > ```
7
8 ## 5. Server Middleware
9
10 Server middleware is very similar to Client
  middleware. Rather than wrapping a `RoundTripper`,
  we wrap a `http.Handler`. [We covered this in the
  last article](./backendbasics2.html), but let's
  briefly review:
11
12 The [ `http.Handler` ](https://pkg.go.dev/net/
  http#Handler) interface looks like this:
13
14 ```go
15 type Handler interface {
16     ServeHTTP(http.ResponseWriter, *http.Request)
17 }
```

We don't need to define our own HandlerFunc type, because [the standard library already provides one](#).

```
1 // HandlerFunc adapts a function to work as a http.Handler.
2 type HandlerFunc func(http.ResponseWriter, *http.Request)
3 // ServeHTTP calls f(w, r)
4 func (f HandlerFunc) ServeHTTP(w http.ResponseWriter, r
  *http.Request) { f(w, r) }
```

We can use this to build our own middleware by wrapping handlers in a closure and returning a HandlerFunc, the same way wrapped RoundTrippers in a closure and returned a RoundTripFunc.

Let's add traces and logs to our server. The implementation is broadly symmetrical to the client middleware:

5.1. Tracing and Logging Requests (server-side)

```
1
2  package servermw
3
4  import (
5      "fmt"
6      "log"
7      "net/http"
8      "os"
9
10     "github.com/google/uuid"
11     "gitlab.com/efronlicht/blog/articles/backendbasics/cmd/
ctxutil"
12     "gitlab.com/efronlicht/blog/articles/backendbasics/cmd/
trace"
13 )
14
15 // Trace returns a middleware that injects a trace into the
request context,
16 // picking up the trace id from the request header if it
exists, or generating a new one if it doesn't.
17 // See clientmw.Trace for the client-side implementation.
18 func Trace(h http.Handler) http.HandlerFunc {
19     return func(w http.ResponseWriter, r *http.Request) {
20         ctx := r.Context()
21         // get trace/req id from request header, or
generate new ones if they don't exist
22         traceID, err := uuid.Parse(r.Header.Get("X-Trace-
Id"))
23         if err != nil {
24             traceID = uuid.New()
25         }
26         reqID, err := uuid.Parse(r.Header.Get("X-Request-
Id"))
27         if err != nil {
28             reqID = uuid.New()
29         }
30
31         // pop trace into context, and pop context into
request
32         trace := trace.Trace{TraceID: traceID, RequestID:
reqID}
33         ctx = ctxutil.WithValue(ctx, trace)
34         r = r.WithContext(ctx)
35
36         // serve the request using the populated context
37         h.ServeHTTP(w, r)
38     }
39 }
40 }
41
```

```

42 // Log returns a middleware that injects a logger into the
    request context. See clientmw.Log for the client-side
    implementation.
43 // It uses the trace from the context as a prefix, if it
    exists. For most servers, use a structured logger instead;
    that API is outside the scope of this article.
44 func Log(h http.Handler) http.HandlerFunc {
45     return func(w http.ResponseWriter, r *http.Request) {
46         trace, ok := ctxutil.Value[trace.Trace]
            (r.Context())
47         var prefix string
48         if ok {
49             // like GET /articles: [trace-id request-id]:
50             prefix = fmt.Sprintf("%s %s: [%s %s]: ",
r.Method, r.URL, trace.TraceID, trace.RequestID)
51         } else {
52             // like GET /articles:
53             prefix = fmt.Sprintf("%s %s: ", r.Method,
r.URL)
54         }
55         logger := log.New(os.Stderr, prefix, log.LstdFlags)
56         ctx := ctxutil.WithValue(r.Context(), logger)
57         r = r.Clone(ctx)
58         h.ServeHTTP(w, r)
59     }
60 }

```

[5.2. Intercepting Writes to the Response](#)

Some server middleware may want to track or intercept writes to the response headers or body. Let's list a few examples:

- Automatically gzip-encode writes to the body if the client sent an `Accept-Encoding: gzip` header.
- Track the status code of the response so we can add it to our logs or metrics.
- Track the total number of bytes written to the response body so we can add it to our metrics.
- Rewrite certain headers so as not to leak internal information to the client.

We can do this by wrapping the `ResponseWriter` in a custom struct that implements the `http.ResponseWriter` interface. This is a bit more complex than wrapping a `RoundTripper` or `Handler`. It's easiest to demonstrate with an example.

The following middleware, `RecordResponse` and its associated `RecordingResponseWriter` struct will track the status code and bytes written to the response body, and log them when the request is complete.

```

1 package servermw
2

```

```

3 // RecordResponse returns a middleware that records the
  response status code and total bytes written to the
  response.
4 func RecordResponse(h http.Handler) http.HandlerFunc {
5     return func(w http.ResponseWriter, r *http.Request) {
6         rrw := &RecordingResponseWriter{RW: w}
7         start := time.Now()
8         h.ServeHTTP(rrw, r)
9         elapsed := time.Since(start)
10        // use the logger from the context if it exists
11        logger, ok := ctxutil.Value[*log.Logger]
  (r.Context())
12        if !ok {
13            // fall back to the default logger
14            log.Printf("%s %s: %d %s: %d bytes in %s",
  r.Method, r.URL, rrw.StatusCode,
  http.StatusText(rrw.StatusCode), rrw.Bytes, elapsed)
15            return
16        }
17        logger.Printf("%d %s: %d bytes in %s",
  rrw.StatusCode, http.StatusText(rrw.StatusCode),
  rrw.Bytes, elapsed)
18    }
19 }
20
21 // RecordingResponseWriter is an http.ResponseWriter that
  keeps track of the status code and total body bytes written
  to it.
22 type RecordingResponseWriter struct {
23     // underlying response writer
24     RW http.ResponseWriter
25     StatusCode int // first status code written to the
  response writer
26     Bytes int // total bytes written
27 }
28
29 // WriteHeader sets the status code, if it hasn't been set
  already.
30 func (w *RecordingResponseWriter) WriteHeader(statusCode
  int) {
31     if w.StatusCode == 0 { // first status code written;
  track it
32         w.StatusCode = statusCode
33     }
34     w.RW.WriteHeader(statusCode) // write to underlying
  response writer
35 }
36
37 // Header just returns the underlying response writer's
  header.
38 func (w *RecordingResponseWriter) Header() http.Header {
  return w.RW.Header() }
39

```

```

40 // Write writes the given bytes to the underlying response
    writer, setting the status code to 200 if it hasn't been
    set already.
41 func (w *RecordingResponseWriter) Write(b []byte) (int,
    error) {
42     if w.StatusCode == 0 {
43         w.WriteHeader(http.StatusOK)
44     }
45     n, err :=
w.RW.Write(b) // write to underlying response writer
46     w.Bytes += n // update total bytes written
47     return n, err
48 }

```

This `RecordingResponseWriter` is broadly similar to the one implemented by the standard library's [httptest.ResponseRecorder](#). As usual, Go's standard library uses a small set of simple interfaces to cover a wide range of use cases.

5.3. Recovering from Panics

Let's add one last server middleware, `Recovery`, to protect our server from unexpected panics. While ideally we would write perfect code without panics, everyone makes mistakes, and it would be good to be able to continue *some* service even if one of our endpoints panics under certain conditions.

As before, our `Recovery` handle takes advantage of the log injected into the context (if it exists). **It's good to have a 'fallback' for any context value you use, since context values are not visible in the type signature or guaranteed to exist.**

```

1 // Recovery returns a middleware that recovers from panics,
    writing a 500 status code and "internal server error"
    message to the response,
2 // and logging the panic and associated stack trace.
3 func Recovery(h http.Handler) http.HandlerFunc {
4     return func(w http.ResponseWriter, r *http.Request) {
5         defer func() { // recover from panic
6             if err := recover(); err != nil { // recover
from panic
7                 stack := debug.Stack()
8                 logger, ok := ctxutil.Value[*log.Logger]
(r.Context())
9                 if !ok { // use the default logger
10                    log.Printf("%s %s: panic: %v\n%s",
r.Method, r.URL, err, stack)
11                } else { // use the logger from the context
12                    logger.Printf("panic: %v\n%s", err,
stack)
13                }
14                // write 500 status code and "internal
server error" message to response so it doesn't hang

```

```

15 w.WriteHeader(http.StatusInternalServerError)
16           _, _ = w.Write([]byte("internal server
17 error"))
18           }
19           }()
20           h.ServeHTTP(w, r)
21           }
22           }

```

5.4. Using Server Middleware

The following complete program, `servermiddlewareex`, implements a simple server that serves two endpoints. GET `/time` returns the current time in RFC3339 format, and GET `/panic` panics. Any other endpoint returns a 404.

```

1 package main
2
3 import (
4     "errors"
5     "flag"
6     "fmt"
7     "log"
8     "net/http"
9     "time"
10
11     "gitlab.com/efronlicht/blog/articles/backendbasics/cmd/
12 servermw"
13 )
14 func main() {
15     port := flag.Int("port", 8080, "port to listen on")
16     flag.Parse()
17     // our base handler.
18     var h http.HandlerFunc = func(w http.ResponseWriter, r
19 *http.Request) {
20         // route the request. note that there's no need for
21 ANY router, even the stdlib's http.ServeMux
22         // if you have a simple enough routing scheme.
23         // a switch statement is perfectly fine.
24         switch r.URL.Path {
25             case "/time":
26                 fmt.Fprintln(w,
27 time.Now().Format(time.RFC3339))
28             case "/panic":
29                 panic("oh my god JC, a bomb!")
30             default:
31                 http.NotFound(w, r)
32         }
33     }
34     // remember, middleware is applied in First In, Last
35 Out order.

```

```

32
33     h = servermw.RecordResponse(h)
34     h = servermw.Recovery(h)
35     h = servermw.Log(h)
36     h = servermw.Trace(h)
37
38     // always apply timeouts to your server, even if you've
put cancellations in the context using a middleware.
39     server := http.Server{
40         Addr:             fmt.Sprintf(":%d", *port),
41         Handler:           h,
42         ReadTimeout:       1 * time.Second,
43         WriteTimeout:      1 * time.Second,
44         ReadHeaderTimeout: 200 * time.Millisecond,
45     }
46     log.Printf("listening on %s", server.Addr)
47     if err := server.ListenAndServe(); !errors.Is(err,
http.ErrServerClosed) {
48         log.Fatal(err)
49     }
50 }

```

Let's fire it up and visit `http://localhost:8080/time`, `http://localhost:8080/panic`, and `http://localhost:8080/foobar` to see what happens.

IN:

```
1 go run servermiddlewareex.go
```

OUT (client):

```
1 localhost:8080/time: 2023-09-12T07:55:16-07:00
2 localhost:8080/panic: internal server error
3 localhost:8080/foobar: 404 page not found
```

OUT (server)

```

1 2023/09/12 07:50:39 listening on :8080
2 GET /time: [4c5ef2f9-cd58-4dec-a28f-770b5786fcba fcc3b01b-
d18d-49a8-96b2-0514c7ac24c6]: 2023/09/12 07:55:16 200 OK: 26
bytes in 3.636µs
3 GET /panic: [98f8f8cc-18ce-4b18-9044-3258c24e57e1
88b97381-1866-4689-a23a-fac50bea0da0]: 2023/09/12 07:55:31
panic: oh my god JC, a bomb!
4 goroutine 6 [running]:
5 runtime/debug.Stack()
6     /usr/local/go/src/runtime/debug/stack.go:24 +0x5e
7 <snip... stack trace ...>

```

Looks like everything is working as expected. So far, we've covered (nearly) everything you might have used a framework for:

- Requests

- Responses
- Middleware
- Serialization
- Dependency Injection

But we haven't covered **routing** yet. Let's fix that.

6. Routing

Routing is the process of matching a request to a handler via its METHOD and PATH. In Go, there's nothing particularly special about routing: it's just something that the Handler inside your Server does.

The most basic kind of routing is just a `switch` statement, like we saw above. That only dealt with paths, but routing based off METHOD is just as easy: the following code is the 'router' that serves **the website you're reading this on**.

6.1. simple routing

```

1  var router http.HandlerFunc = func(w http.ResponseWriter,
   r *http.Request) {
2      p := strings.TrimSuffix(r.URL.Path, "/")
3      switch {
4      case r.Method != "GET": // only GET is allowed
5          w.WriteHeader(http.StatusMethodNotAllowed)
6          return
7      case p == "/debug/uptime": // return uptime
8          d := (time.Since(start).Seconds())
9          const MIN = 60
10         const HOUR = 60 * MIN
11         const DAY = 24 * HOUR
12         _, _ = fmt.Fprintf(w, "%2dd %02dh %02dm
   %02ds", int(d/DAY), int(d/HOUR)%24, int(d/MIN)%60, int(d)
   %60)
13         return
14         case p == "/debug/meta": // return metadata
   about the server
15             _, _ = w.Write(metaJSON)
16             return
17         case p == "": // redirect to index.html
18             http.Redirect(w, r, "./index.html",
   http.StatusPermanentRedirect)
19             return
20         default: // serve webpages
21             // fonts are immutable and large, so we can
   cache them for a long time.~
22             // everything else is tiny and might
   change, so we don't cache it.

```

```

23         Cache-Control: max-age=604800,
        stale-while-revalidate=86400
24         static.ServeFile(w,
        r) // serve pre-zipped, embedded files
25             return
26     }
27 }

```

This is all you need for small programs. For convenience, Go's stdlib comes with a built-in Router, [http.ServeMux](#), which uses a simple prefix-based matching scheme: the longest prefix that matches the request path wins. It's implemented like this:

```

1 // Find a handler on a handler map given a path string.
2 // Most-specific (longest) pattern wins.
3 func (mux *ServeMux) match(path string) (h Handler,
    pattern string) {
4     // Check for exact match first.
5     v, ok := mux.m[path]
6     if ok {
7         return v.h, v.pattern
8     }
9
10    // Check for longest valid match. mux.es contains all
    patterns
11    // that end in / sorted from longest to shortest.
12    for _, e := range mux.es {
13        if strings.HasPrefix(path, e.pattern) {
14            return e.h, e.pattern
15        }
16    }
17    return nil, ""
18 }

```

ServeMux is perfectly fine for the vast majority of backend programs, but is not very flexible: it doesn't even support routing by METHOD. (There is an [accepted proposal](#) to add both this and wildcards to http.ServeMux, but it's not implemented yet.)

More complicated routers, like the popular [gorilla/mux](#), allow for routing by method, by patterns matching regular expressions, and for extracting variables from the URL path.

[Note to the reader](#)

As of [go 1.22](#), [http.ServeMux](#) has much more functionality. I'd stick with that rather than using [gorilla/mux](#). This article will still duplicate [gorilla/mux](#)'s API rather than [http.ServeMux](#), though - I don't want to rewrite it.

I'll quote their documentation here to give you an idea of what this looks like:

- `### gorilla/mux: quoted documentation`

— GORILLA DOCUMENTATION STARTS HERE —

Let's start registering a couple of URL paths and handlers:

```
1 func main() {
2     r := mux.NewRouter()
3     r.HandleFunc("/", HomeHandler)
4     r.HandleFunc("/products", ProductsHandler)
5     r.HandleFunc("/articles", ArticlesHandler)
6     http.Handle("/", r)
7 }
```

Here we register three routes mapping URL paths to handlers. This is equivalent to how `http.HandleFunc()` works: if an incoming request URL matches one of the paths, the corresponding handler is called passing (`http.ResponseWriter`, `*http.Request`) as parameters.

Paths can have variables. They are defined using the format `{name}` or `{name: pattern}`. If a regular expression pattern is not defined, the matched variable will be anything until the next slash. For example:

```
1 r := mux.NewRouter()
2 r.HandleFunc("/products/{key}", ProductHandler)
3 r.HandleFunc("/articles/{category}/",
4   ArticlesCategoryHandler)
5 r.HandleFunc("/articles/{category}/{id:[0-9]+}",
6   ArticleHandler)
```

The names are used to create a map of route variables which can be retrieved calling `mux.Vars()`:

```
1 func ArticlesCategoryHandler(w http.ResponseWriter, r
2   *http.Request) {
3     vars := mux.Vars(r)
4     w.WriteHeader(http.StatusOK)
5     fmt.Fprintf(w, "Category: %v\n", vars["category"])
6 }
```

Paths can have variables. They are defined using the format `{name}` or `{name: pattern}`. If a regular expression pattern is not defined, the matched variable will be anything until the next slash. For example:

```
1 r := mux.NewRouter()
2 r.HandleFunc("/products/{key}", ProductHandler)
3 r.HandleFunc("/articles/{category}/",
4   ArticlesCategoryHandler)
5 r.HandleFunc("/articles/{category}/{id:[0-9]+}",
6   ArticleHandler)
```

The names are used to create a map of route variables which can be retrieved calling `mux.Vars()`:

```

1 func ArticlesCategoryHandler(w http.ResponseWriter, r
  *http.Request) {
2     vars := mux.Vars(r)
3     w.WriteHeader(http.StatusOK)
4     fmt.Fprintf(w, "Category: %v\n", vars["category"])
5 }

```

— GORILLA DOCUMENTATION ENDS HERE —

[6.2. Building a router with gorilla/mux-like syntax](#)

Let's build our own router that can handle expressions of this kind.

[See the full code, including 100% test coverage, at https://go.dev/play/p/BBGLxqepogO.](https://go.dev/play/p/BBGLxqepogO)

[6.2.1. Summary of API](#)

The following table summarizes our external API:

Type/Func	Description
Router	A router that matches HTTP requests to handlers based on the request path.
Router.AddRoute(pattern, method, handler)	Add a route to the router.
Router.ServeHTTP(w, r)	Serves the request by matching it against the routes in the router.
PathVars	A map[string]string of path params to their values
Vars(r)	Returns the path parameters for the current request, or an empty map if there are none.

Each route will contain a method (e.g, GET), a pattern (e.g, /articles/{category}/{id:[0-9]+}), and a http.Handler. Patterns that match a regular expression corresponding to the pattern will be dispatched to the handler.

That is, we'll need to convert a pattern like /articles/{category}/{id:[0-9]+} into a regular expression like ^/articles/([a-zA-Z]+)/([0-9]+)\$, and extract the path parameters from the request path, so that the *server* can find what we meant by “category” and “id”.

Each route will look like this:

```

1 type route struct {
2     pattern *regexp.Regexp // the compiled regexp
3     names   []string // the names of the path parameters;
  one per capture group in the regexp
4     raw     string // the raw pattern string

```

```

5     method string // the HTTP method to match; if empty,
    all methods match.
6     handler http.Handler // underlying handler
7 }
8

```

6.2.2. Building the Router

We'll define a Router type that contains a slice of routes, matching against each in turn:

```

1 // Router allows you to match HTTP requests to handlers
  based on the request path.
2 // It use a syntax similar to gorilla/mux:
3 // /path/{regexp}/{name:captured-regexp}
4 // AddRoute adds a route to the router.
5 // Vars returns the path parameters for the current request,
  or nil if there are none.
6 type Router struct {routes []route}

```

We'll need to compile the pattern into a regular expression, and extract the path parameters from the pattern.

This entails keeping track of which names correspond to which capture groups in the regexp.

While we could use named capture groups (`(?P<name> . . .)`), this is slow and bulky and requires extra iteration. Instead, we'll just keep track of the names in a slice, and use the slice index to determine which capture group corresponds to which name.

That is, suppose we have pattern

```

1 /chess/replay/{white:[a-zA-Z]+}/{black:[a-zA-Z]+}/{id:
  [0-9]+}

```

This should compile into the regexp

```

1 ^/chess/replay/([a-zA-Z]+)/([a-zA-Z]+)/([0-9]+)$

```

With the names slice

```

1 []string{"white", "black", "id"}

```

We define a function, `buildRoute`, to do this:

```

1 func buildRoute(pattern string) (re *regexp.Regexp, names
  []string, err error) {
2     if pattern == "" || pattern[0] != '/' {
3         return nil, nil, fmt.Errorf("invalid pattern %s:
  must begin with '/'", pattern)
4     }
5     var buf strings.Builder

```

```

6     buf.WriteByte('^') // match the beginning of the string
7
8     // we gradually build up the regexp, and keep track of
the path parameters we encounter.
9     // e.g, on successive iterations, we'll have:
10    // FOR {
11    // 0: /chess, nil
12    // 1: /chess/replay, nil
13    // 2: /chess/replay/([a-zA-Z]+), [white]
14    // 3: /chess/replay/([a-zA-Z]+)/([a-zA-Z]+), [white,
black]
15    // 4: /chess/replay/([a-zA-Z]+)/([a-zA-Z]+)/([0-9]+),
[white, black, id]
16    // }
17    for _, f := range strings.Split(pattern, "/")[1:] {
18
19    buf.WriteByte('/') //
add the '/' back
20    if len(f) >= 2 && f[0] == '{' && f[len(f)-1] ==
'}' { // path parameter
21        trimmed := f[1 :
len(f)-1] // strip off the '{' and '}'
22        // - {white:[a-zA-Z]+} -> [a-zA-Z]+
23        if before, after, ok := strings.Cut(trimmed,
":"); ok { // its a regexp-capture group
24            names = append(names, before)
25
26            // replace with a capture group: i.e, if we
have {id:[0-9]+}, we want to replace it with ([0-9]+)
27            buf.WriteByte('(') //
28            buf.WriteString(after)
29            buf.WriteByte(')')
30            // white:[a-zA-Z]+ -> ([a-zA-Z]+)
31        } else {
32            buf.WriteString(trimmed) // a regular
expression, but not a captured one
33        }
34    } else {
35        buf.WriteString(regexp.QuoteMeta(f)) // escape
any special characters
36    }
37    }
38    // check for duplicate path parameters
39    for i := range names {
40        for j := i + 1; j < len(names); j++ {
41            if names[i] == names[j] {
42                return nil, nil,
fmt.Errorf("duplicate path parameter %s in %q", names[i],
pattern)
43            }
44        }
45    }
46    buf.WriteByte('$') // match the end of the string

```

```

47     re, err = regexp.Compile(buf.String())
48     if err != nil {
49         return nil, nil, fmt.Errorf("invalid regexp %s:
    %w", buf.String(), err)
50     }
51     return re, names, nil
52 }

```

We'll obtain path parameters on the server by creating a map of path parameters to their values, and adding it to the request context, reusing the `ctxutil` package we wrote earlier.

Then, we'll store those path parameters in a `map[string]string` and put the map in the context. We'll use a unique type for the map so that we can use `ctxutil.Value` to retrieve it without stepping on any other context values. (A collision is unlikely, but it's good practice to avoid it anyway; types are free.)

```

1 // Vars is a map of path parameters to their values. It is a
  unique type so that ctxutil.Value can be used to retrieve
  it.
2 type PathVars map[string]string
3
4 var empty = make(PathVars)
5 // Vars returns the path parameters for the current request.
  It will be nil if you didn't use a router to serve the
  request.
6 func Vars(ctx context.Context) PathVars { v, _ :=
  ctxutil.Value[PathVars](ctx); return v }
7 // AddRoute adds a route to the router. Method is the HTTP
  method to match; if empty, all methods match.
8 // Method will be converted to uppercase and trimmed of
  whitespace, so
9 // "get", "gEt", " geT", and "GET" are all equivalent.

```

Adding routes is straightforward:

```

1 // AddRoute adds a route to the router. Method is the HTTP
  method to match; if empty, all methods match.
2 func (r *Router) AddRoute(pattern string, h http.Handler,
  method string) error {
3     re, names, err := buildRoute(pattern)
4     if err != nil {
5         return err
6     }
7     r.routes = append(r.routes, route{
8         raw:      pattern,
9         pattern: re,
10        names:   names,
11        method:
  strings.ToUpper(strings.TrimSpace(method)),
12        handler: h,
13    })
14 }

```

```

15     // sort the routes by length, so that the longest
16     routes are matched first.
17     sort.Slice(r.routes, func(i, j int) bool {
18         return len(r.routes[i].raw) > len(r.routes[j].raw)
19     || (len(r.routes[i].raw) == len(r.routes[j].raw) &&
20         r.routes[i].raw < r.routes[j].raw) // sort by length, then
lexicographically
18     })
19     return nil
20 }

```

Building the routes was the hard part; actual dispatch is easy. We pick the first route that matches the request path, add the relevant path segments to a map in the context, and serve the request using the associated handler.

```

1
2 // pathVars extracts the path parameters from the path and
3 into a map. leave it as-is.
4 func pathVars(re *regexp.Regexp, names []string, path
5 string) PathVars {
6     matches := re.FindStringSubmatch(path)
7     if len(matches) != len(names)+1 { // +1 because the
8     first match is the entire string
9
10    panic(fmt.Errorf("programmer error: expected regexp %q to
11    match %q", path, re.String()))
12    }
13    vars := make(PathVars, len(names))
14    for i, match := range matches[1:] { // again, skip the
15    first match, which is the entire string
16        vars[names[i]] = match
17    }
18    return vars
19 }
20
21 // ServeHTTP implements http.Handler, dispatching requests
22 to the appropriate handler.
23 func (rt *Router) ServeHTTP(w http.ResponseWriter, r
24 *http.Request) {
25     for _, route := range rt.routes {
26         if route.pattern.MatchString(r.URL.Path) &&
27         (route.method == "" || route.method == r.Method) {
28             vars := pathVars(route.pattern, route.names,
29             r.URL.Path)
30             ctx := CtxWithVal(r.Context(), vars)
31             route.handler.ServeHTTP(w, r.WithContext(ctx))
32             return
33         }
34     }
35     http.NotFound(w, r) // no route matched; serve a 404
36 }
37

```

This router is missing some important features: among other things, it doesn't do any kind of path normalization, it has no performance guarantees, and it doesn't properly handle URL and Regexp escaping and normalization. As such, unlike my usual advice, if you need more sophisticated routing than the stdlib can provide, **I suggest you use an external routing library rather than build it yourself. But just use a router** - don't use anything that forces you in to an entire ecosystem of libraries and frameworks.

That being said, this router is perfectly fine for small programs, and more importantly, *now you know how they work*, so you can figure out how to fix them if they go wrong.

7. Graduation: Putting it all together

Let's write ourselves a client and server that puts together all the pieces we've written so far. Our server will route to multiple endpoints. Some of our routes will take a POST json body, some will take query parameters, and still others will take path parameters. We'll use all the middleware we've written so far in order to log, trace, and recover from panics.

The following program, `graduation`, implements a complete web server that demonstrates all of the concepts we've covered so far.

7.1. building our router

First, we'll use our router to register all the endpoints we want to serve.

```
1  package main
2  // register routes.
3  func buildBaseRouter() (*Router, error) {
4      var r = new(Router) // we'll add routes to this
5                          router.
6                          /* --- design note: ---
7                          you could just add the routes on a separate line for
8                          each,
9                          but I like building the slice of routes and iterating
10                         over it;
11                         it makes the essential similarity of each route more
12                         obvious.
13                         */
14     for _, route := range []struct {
15         pattern, method string
16         handler          http.HandlerFunc
17     }{
18         // GET / returns "Hello, world!"
19         {
20             pattern: "/",
21             method:  "GET",
22             /* ----- design note: -----
```

```

20             this route demonstrates the simplest
possible handler.
21             note the _ for the request parameter; we
don't need it, so we don't bind it.
22             -----*/
23             handler: func(w http.ResponseWriter, _
*http.Request) { w.Write([]byte("Hello, world!\r\n")) },
24             },
25
26             // GET /panic always panics.
27
28             {
29                 pattern: "/panic",
30                 method: "GET",
31                 /* ----- design note: -----
32                 this route demonstrates how middleware can
handle error conditions:
33                 the panic will be caught by Recovery,
which will write a 500 status code and "internal server
error" message to the response.
34                 rather than leaving the connection
hanging.
35                 note the _ and _ for the request and
response parameters; we don't need them, so we don't bind
them.
36                 -----*/
37                 handler: func(_ http.ResponseWriter, _
*http.Request) { panic("oh my god JC, a bomb!") },
38                 },
39
40                 // POST /greet/json returns a JSON object with a
greeting and a category based on the age.
41                 // it must be called with a JSON object in the
form {"first": "efron", "last": "licht", "age": 32}
42                 {
43                     "/greet/json",
44                     "POST",
45                     /* ----- design note: -----
46                     // this route is a sophisticated example that
has both path parameters (using our custom router) and
query parameters.
47                     // it 'puts everything together' and
demonstrates how to use the router and middleware
together.
48                     // ----- */
49                     func(w http.ResponseWriter, r *http.Request) {
50
51                         var req struct {
52                             First, Last string
53                             Age          int
54                         }
55                         if err :=
json.NewDecoder(r.Body).Decode(&req); err != nil {

```

```

56         WriteError(w, err,
http.StatusBadRequest) // remember to return after writing
an error!
57         return
58     }
59     if req.Age < 0 {
60         WriteError(w, errors.New("age must be
>= 0"), http.StatusBadRequest)
61         return
62     }
63     var category string
64     switch {
65     case req.Age < 13:
66         WriteError(w, errors.New("forbidden:
come back when you're older"), http.StatusForbidden)
67         return
68     case req.Age < 21:
69         category = "teenager"
70     case req.Age > 65:
71         category = "senior"
72     default:
73         category = "adult"
74     }
75     - = WriteJSON(w, struct {
76         Greeting string `json:"greeting"`
77         Category string `json:"category"`
78     }) {
79         fmt.Sprintf("Hello, %s %s!",
req.First, req.Last),
80         category,
81     })
82     }},
83
84     // GET /time returns the current time in the given
format.
85     // it demonstrates how to use query parameters.
86     {
87         pattern: "/time",
88         method: "GET",
89         handler: func(w http.ResponseWriter, r
*http.Request) {
90             format := r.URL.Query().Get("format")
91             if format == "" {
92                 format = time.RFC3339
93             }
94             tz := r.URL.Query().Get("tz")
95             var loc *time.Location = time.Local
96             if tz != "" {
97                 var err error
98                 loc, err = time.LoadLocation(tz)
99                 if err != nil {
100                     WriteError(w, fmt.Errorf("invalid
timezone %q: %w", tz, err), http.StatusBadRequest)
101                     return

```

```

102         }
103     }
104     _ = WriteJSON(w, struct {
105         Time string `json:"time"`
106     }{time.Now().In(loc).Format(format)})
107     },
108 },
109 // GET /echo/{a}/{b}/{c} returns the path
parameters as a JSON object in the form {"a": "value of
a", "b": "value of b", "c": "value of c"}
110 // the query parameter "case" can be "upper" or
"lower" to convert the values to uppercase or lowercase.
111 {
112     pattern: "/echo/{a:.+}/{b:.+}/{c:.+}",
113     method: "GET",
114     /* ----- design note: -----
115     this route is a sophisticated example that has
both path parameters (using our custom router)
116     and query parameters.
117     it 'puts everything together' and demonstrates
how to use the router and middleware together.
118     ---- */
119     handler: func(w http.ResponseWriter, r
*http.Request) {
120         vars, _ := ctxutil.Value[PathVars]
(r.Context())
121         switch
strings.ToLower(r.URL.Query().Get("case")) {
122         case "upper":
123             for k, v := range vars {
124                 vars[k] = strings.ToUpper(v)
125             }
126         case "lower":
127             for k, v := range vars {
128                 vars[k] = strings.ToLower(v)
129             }
130         }
131         _ = WriteJSON(w, vars)
132     },
133 },
134 } {
135     if err := r.AddRoute(route.pattern,
route.handler, route.method); err != nil {
136         return nil, fmt.Errorf("AddRoute(%q, %v, %q)
returned error: %v", route.pattern, route.handler,
route.method, err)
137     }
138     log.Printf("registered route: %s %s",
route.method, route.pattern)
139 }
140 }
141 return r, nil
142 }

```

7.2. building & starting the server

In main, we'll take the router we just built and add some of the server middleware we previously wrote:

```
1
2 func main() {
3     port := flag.Int("port", 8080, "port to listen on")
4     flag.Parse()
5
6     h, err := buildBaseRouter()
7     if err != nil {
8         log.Fatal(err)
9     }
10    h = applyMiddleware(h)
```

Then we'll start up the server.

```
1 // build and start the server.
2 // remember, you should always apply at least the Read
  and Write timeouts to your server.
3 server := http.Server{
4     Addr:         fmt.Sprintf(":%d", *port),
5     Handler:      h,
6     ReadTimeout:  1 * time.Second,
7     WriteTimeout: 1 * time.Second,
8 }
9 log.Printf("listening on %s", server.Addr)
10 go server.ListenAndServe()
11 time.Sleep(20 * time.Millisecond)
12 }
```

7.3. demo

The following program, `graduationdemo`, hits the server we just wrote with a variety of requests, demonstrating all the endpoints we just wrote.

```
1 package main
2 func main() {
3     var port int
4     flag.IntVar(&port, "port", 8080, "port for outgoing
  requests")
5     flag.Parse()
6     rt = clientmw.Trace(rt)
7     rt = clientmw.Log(rt)
8     client := &http.Client{
9         Transport: rt,
10        Timeout:    1 * time.Second,
11    }
12    req, err := http.NewRequest("GET", fmt.Sprintf("http://
  localhost:%d/panic", port), nil)
13    if err != nil {
```

```

14         log.Fatal(err)
15     }
16
17     resp, err := client.Do(req)
18     if err != nil {
19         log.Fatal(err)
20     }
21     fmt.Println("GET /panic:")
22     resp.Write(os.Stdout)
23
24     var paths = []string{
25         "/",
26         "/echo/first/second/third",
27         "/echo/first/second/third",
28         "/echo/first/second/third",
29         "/time",
30         "/time",
31         "/time",
32         "/time",
33     }
34
35     var queries = []map[string]string{
36         nil,
37         nil,
38         {"case": "upper"},
39         {"case": "lower"},
40         nil,
41         {"format": time.RFC1123},
42         {"format": time.RFC1123, "tz": "America/New_York"},
43         {"format": time.RFC1123, "tz": "America/
Los_Angeles"},
44         {"format": time.RFC1123, "tz": "America/Chicago"},
45     }
46     for i := range paths {
47         q := make(url.Values)
48         for k, v := range queries[i] {
49             q.Set(k, v)
50         }
51         url := fmt.Sprintf("http://localhost:%d%s", port,
paths[i])
52         if len(q) > 0 {
53             url += "?" + q.Encode()
54         }
55         ctx, cancel :=
context.WithTimeout(context.Background(), 1*time.Second)
56         req, err := http.NewRequestWithContext(ctx, "GET",
url, nil)
57         if err != nil {
58             log.Fatal(err)
59         }
60         resp, err := client.Do(req)
61         cancel()
62         if err != nil {
63             log.Fatal(err)

```

```

64     }
65     fmt.Printf("GET %s: \n", url)
66     resp.Write(os.Stdout)
67     fmt.Println("\n-----")
68 }
69
70 }

```

If you run both programs, you should see something like this (lightly edited and heavily compressed for space):

```

1  server: GET /echo/first/second/third:
   [d4df5e13-fadf-4f4a-8eb9-7ca1dca0428b
   c16dc184-90ae-4a23-85cd-4898456ac25e]: 2023/09/18 10:59:04
   200 OK: 39 bytes in 28.093µs
2  client: GET http://localhost:8080/echo/first/second/third:
   2023/09/18 10:59:04 clientmw.go:112: 200 OK in 348.138µs
3  GET http://localhost:8080/echo/first/second/third:
4  HTTP/1.1 200 OK
5  Content-Length: 39
6  Content-Type: application/json
7  Date: Mon, 18 Sep 2023 17:59:04 GMT
8
9  {"a":"first","b":"second","c":"third"}
10
11 server: GET /echo/first/second/third?case=upper:
   [2253f79f-e49d-455e-b32a-3d842c7f1ea5
   bb748b7a-2a07-4d86-8992-0ba3f2e7423c]: 2023/09/18 10:59:04
   200 OK: 39 bytes in 21.069µs
12 client: GET http://localhost:8080/echo/first/second/third?
   case=upper: 2023/09/18 10:59:04 clientmw.go:112: 200 OK in
   342.51µs
13 GET http://localhost:8080/echo/first/second/third?
   case=upper:
14 HTTP/1.1 200 OK
15 Content-Length: 39
16 Content-Type: application/json
17 Date: Mon, 18 Sep 2023 17:59:04 GMT
18
19 {"a":"FIRST","b":"SECOND","c":"THIRD"}
20
21 server: GET /time?
   format=Mon%2C+02+Jan+2006+15%3A04%3A05+MST&tz=America%2FLos_Angeles:
   [de41ddc1-fd01-4aa7-b666-a9a5d81a1c08
   e7605ac9-aa5d-43b1-9fcc-5a90af39d737]: 2023/09/18 10:59:04 200 OK:
   41 bytes in 21.901µs
22 client: GET http://localhost:8080/time?
   format=Mon%2C+02+Jan+2006+15%3A04%3A05+MST&tz=America%2FLos_Angeles:
   2023/09/18 10:59:04 clientmw.go:112: 200 OK in 244.977µs
23 GET http://localhost:8080/time?
   format=Mon%2C+02+Jan+2006+15%3A04%3A05+MST&tz=America%2FLos_Angeles:
24 HTTP/1.1 200 OK
25 Content-Length: 41

```

```
26 Content-Type: application/json
27 Date: Mon, 18 Sep 2023 17:59:04 GMT
28
29 {"time":"Mon, 18 Sep 2023 10:59:04 PDT"}
```

8. Testing HTTP servers

We'd like to have a better guarantee that our server is working correctly than just "it seems to work when I run it", though.

The [httptest.Server](#) listens on a random port and serves http using a provided handler.

The following heavily annotated test suite demonstrates how to use `http.Server` and table-driven tests to test our server.

```
1  package main
2
3  import (
4      "bytes"
5      "context"
6      "encoding/json"
7      "fmt"
8      "io"
9      "log"
10     "net/http"
11     "net/http/httptest"
12     "net/url"
13     "os"
14     "reflect"
15     "strings"
16     "testing"
17
18     "gitlab.com/efronlicht/blog/articles/backendbasics/
19     cmd/clientmw"
20     "gitlab.com/efronlicht/blog/articles/backendbasics/
21     cmd/ctxutil"
22 )
23
24 // initialized during TestMain.
25 var client *http.Client
26 var server *httptest.Server
27
28 // The TestMain function is a special function that runs
29 // before any tests are run; think of it as init()
30 // that only runs when you run tests.
31 func TestMain(m *testing.M) {
32     router, err := buildBaseRouter()
33     if err != nil {
34         log.Fatal(err)
35     }
36 }
```

```

33     router = applyMiddleware(router) // apply middleware
34     // httptest.NewServer starts an http server that
    listens on a random port.
35     // you can use the URL field of the returned
    httptest.Server to make requests to the server.
36     server = httptest.NewServer(router) // create a test
    server with the router
37
38     // httptest.Server.Client() returns an http.Client
    that uses the test server and always accepts it's auth
    certificate.
39     client = server.Client()
40
41     if client.Transport == nil {
42         client.Transport = http.DefaultTransport // use
    the default transport if the client doesn't have one.
43     }
44     // apply our client middleware to the client.
45     client.Transport =
    clientmw.Log(clientmw.Trace(client.Transport)) // add
    logging and tracing to the client
46
47     code := m.Run() // run the tests
48
49     server.Close() // close the test server
50
51
    os.Exit(code) // exit with the same code as the tests; 0
    if all tests passed, non-zero otherwise.
52 }
53
54 // TestNotFound tests that the router returns a 404 status
    code for requests that don't match any routes.
55 func TestNotFound(t *testing.T) {
56     for _, tt := range []struct {
57         method, path string
58     }{
59         {"DELETE", "/"},
60         {"GET", "/notfound"},
61         {"GET", "/chess/replay/efronlicht/bobross/1234"},
62     } {
63         req, _ := http.NewRequest(tt.method,
    server.URL+tt.path, nil)
64
65         if resp, err := client.Do(req); err != nil {
66             t.Errorf("client.Do(%q, %q) returned error:
    %v", tt.method, tt.path, err)
67         } else if resp.StatusCode != http.StatusNotFound {
68             t.Errorf("client.Do(%q, %q) returned status
    %d, want %d", tt.method, tt.path, resp.StatusCode,
    http.StatusNotFound)
69         }
70
71     }

```

```

72 }
73
74 // TestGraduation tests that the server works as expected.
75 // This is meant to demonstrate how to write tests for a
76 // server in a way that doesn't have too many dependencies
77 // or use any external libraries.
78 func TestGraduation(t *testing.T) {
79     defer server.Close()
80
81     // table-based testing is a common pattern in Go.
82     for _, tt := range []struct {
83         method, path string // where is the
84         request going?
85         body          map[string]any // what is the
86         request body, if any? nil if no body.
87         queries       map[string]string // what are the
88         query parameters, if any? nil if no query parameters.
89         wantStatus    int // what status code do we expect?
90         // what substrings do we expect to find in the
91         response body?
92         /* ----- design note: ----
93         we're not testing the entire response body,
94         because it's too brittle; small changes in the response
95         body
96         like whitespace or punctuation will cause the test
97         to fail.
98         instead, we test for substrings that we expect to
99         find in the response body.
100        this is a good compromise between testing the
101        entire response body and testing nothing.
102        if you have structured content, like JSON, you can
103        unmarshal the response body into a struct and test that.
104        */
105        wantBodyContains []string
106    }{
107        { // GET / returns "Hello, world!"
108            method: "GET",
109            path: "/",
110            wantStatus: http.StatusOK,
111            wantBodyContains: []string{"Hello, world!"},
112        },
113        { // GET /panic always panics.
114            method: "GET",
115            path: "/panic",
116            wantStatus:
117            http.StatusInternalServerError,
118            wantBodyContains: []string{"Internal Server
119            Error"},
120        },
121        // POST /greet/json returns a JSON object with a
122        greeting and a category based on the age,
123        // and it's forbidden to children under 13.

```

```

111     {
112         method:      "POST",
113         path:         "/greet/json",
114         body:         map[string]any{"first":
"Raphael", "last": "Frasca", "age": 10}, // my nephew
turned 10 today! ;P
115         wantStatus:  http.StatusForbidden,
116         wantBodyContains: []string{"forbidden"},
117     },
118     // adults should get a greeting based on their
age: i'm an adult.
119     {
120         method:      "POST",
121         path:         "/greet/json",
122         body:         map[string]any{"first":
"Efron", "last": "Licht", "age": 32},
123         wantStatus:  http.StatusOK,
124         wantBodyContains: []string{"Efron", "Licht",
"adult"},
125     },
126     // GET /time returns the current time in UTC, or
in the timezone specified by the tz query parameter.
127     {
128         method:      "GET",
129         path:         "/time",
130         queries:     map[string]string{"tz": "America/
New_York"},
131         wantStatus: http.StatusOK,
132         /* wantBodyContains: []string{"-4"}
133         this test is bad, because it assumes that the
test is running in a timezone that is 4 hours behind UTC;
134         America/New_York is only 4 hours behind UTC
during daylight savings time; otherwise it's 5 hours
behind.
135         be careful when writing tests that depend on
timezones! */
136     },
137     // GET /time returns a 400 status code if the tz
query parameter is invalid.
138     {
139         method:      "GET",
140         path:         "/time",
141         queries:     map[string]string{"tz":
"invalid"},
142         wantStatus: http.StatusBadRequest,
143     },
144 } {
145     tt := tt // capture the range variable
146     if len(tt.queries) != 0 {
147         query := make(url.Values, len(tt.queries))
148         for k, v := range tt.queries {
149             query.Set(k, v)
150         }
151         tt.path += "?" + query.Encode()

```

```

152     }
153     // give the test a name that describes the request
so we can easily see what passed and failed in the output:
154     // i.e, TestGraduation/GET/ -> 200-OK
155     // TestGraduation/POST/greet/json->403-Forbidden
156     name := fmt.Sprintf("%s%->%d-%s", tt.method,
tt.path, tt.wantStatus, strings.ReplaceAll(
157         http.StatusText(tt.wantStatus), " ", "-"))
158     t.Run(name, func(t *testing.T) {
159         path := server.URL + tt.path
160
161         var body io.Reader
162         if tt.body != nil {
163             b, _ := json.Marshal(tt.body)
164             body = bytes.NewReader(b)
165         }
166
167         req, err :=
http.NewRequestWithContext(context.TODO(), tt.method,
path, body)
168         if err != nil {
169             t.Errorf("http.NewRequestWithContext(%q,
%q, %v) returned error: %v", tt.method, tt.path, tt.body,
err)
170         }
171
172         resp, err := client.Do(req)
173         if err != nil {
174             t.Errorf("client.Do(%q, %q) returned
error: %v", tt.method, tt.path, err)
175         }
176
177         if resp.StatusCode != tt.wantStatus {
178             t.Errorf("router.ServeHTTP(%q, %q)
returned status %d, want %d", tt.method, tt.path,
resp.StatusCode, tt.wantStatus)
179         }
180         bodyBytes, _ := io.ReadAll(resp.Body)
181
182         resp.Body.Close()
183
184         for _, want := range tt.wantBodyContains {
185             if !strings.Contains(string(bodyBytes),
want) {
186                 t.Errorf("router.ServeHTTP(%q, %q)
returned body %s, want body to contain %s", tt.method,
tt.path, bodyBytes, want)
187             }
188         }
189     })
190 }
191 }
192 }

```

IN:

```
1 go test ./...
```

OUT:

```
1 ok      gitlab.com/efronlicht/blog/articles/backendbasics/
  cmd/graduation      0.005s
```

Or to give a little more detail:

IN:

```
1 go test -v ./... | RG "PASS"
```

OUT

```
1 --- PASS: TestNotFound (0.00s)
2 --- PASS: TestGraduation (0.00s)
3     --- PASS: TestGraduation/GET/->200-OK (0.00s)
4     --- PASS: TestGraduation/GET/
  panic->500-Internal-Server-Error (0.00s)
5     --- PASS: TestGraduation/POST/greet/json->403-Forbidden
  (0.00s)
6     --- PASS: TestGraduation/POST/greet/json->200-OK (0.00s)
7     --- PASS: TestGraduation/GET/time?
  tz=America%2FNew_York->200-OK (0.00s)
8     --- PASS: TestGraduation/GET/time?
  tz=invalid->400-Bad-Request (0.00s)
```

9. Conclusion

That's it! Congratulations on reading all, uh

IN

```
1 wc -w backendbasics.md backendbasics2.md backendbasics3.md
```

OUT

```
1 8152 backendbasics.md
2 5470 backendbasics2.md
3 10801 backendbasics3.md
4 24423 total
```

24,423 words of this series so far.

With luck, you should feel significantly more comfortable with the basics of building backend infrastructure 'from scratch' (well, from the stdlib) in Go. This article is not even close to comprehensive. Beyond the fact we only barely scratched the surface of

databases, there are many other topics we didn't cover, which include but are not limited to:

- authentication and authorization
- cryptography
- non-tcp transport-layer protocols (udp, unix sockets, etc)
- non-http application-layer protocols (grpc, thrift, etc)
- websockets
- foreign function interfaces (FFI)
- graph databases
- distributed systems
- concurrency
- etc etc etc.

A brief personal note before we finish up: I've been blown away by the positive response to this article so far. Last time I checked, my reddit post about this article had nearly **60,000** views, well more than double the response to my [second-most-popular-article](#). Not bad, considering this website has no link-trading, no click-bait, no SEO, no ads, ~~ and most importantly ~~ , no javascript - I've deliberately avoided doing any of the things you're "supposed" to do to get views on the internet, but an audience has found me anyway. Nice to know web 1.0 has a little life left in it yet.

A note on libraries and frameworks: there's nothing wrong with using someone else's code to solve a problem. I don't expect you to walk away from this article, climb on to the highest peak with nothing but a thin robe, a laptop, and a go compiler, and become completely self-sufficient. Libraries are just fine, and we've used a few in this article. My issue is with the attitude of reaching for a solution *without knowing the problem the solution is trying to solve*. Taking the effort to try and build something yourself is the only way to evaluate whether a library or framework is actually good or bad for your use case.

[MORE ARTICLES](#)