Backend from the Beginning, Part 2: Practical Backend with `net/http`, `context`, and `encoding/JSON`

A software article by Efron Amber Licht

September 2023

ALL ARTICLES

LICENSE

Feeds

- RSS
- ATOM
- JSON

This is the second part of a series of articles about backend development in Go.

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 this one, we'll start diving into Go's standard library and the show how it provides everything you need for basic client/server HTTP communication; using `net/http` and `net/url` to send and receive HTTP requests and responses, `encoding/json` to manage our API payloads, and `context` to manage timeouts and cancellation.

In the next article, we'll talk about middleware, routing, and the basics of databases.

As before, **all source code** for each article is available on gitlab; I also have links to runnable examples for many of these programs on the go playground.

1. The `net/http` package

`net/http` provides a complete HTTP client and server implementation. It's a great place to start when building a web application in Go. Let's start by looking at the client side.

1.1. Requests and Responses

While we built our own `Request` and `Response` types in the previous article, `net/http` provides its own; as Go developers, these are the types you should use 99.9% of the time. Before we move forward, take a peek at the struct definitions of `http.Request` and `http.Response` structs to see how they differ from ours. The most important difference is that they use `io.Readers` for the bodies rather than strings, since `HTTP` is a streaming protocol.

1.1.1. `http.Request`

Build a HTTP request with `http.NewRequestWithContext(ctx, method, url, body)`. The library will automatically parse the URL and set the `Host` header. The `body` argument is an `io.Reader` that provides the request body. If you don't have a body, pass `nil`.

Never use `http.NewRequest` without a context. If you don't know what context to use, use `context.TODO()`. This will save you from a lot of headaches later on.

The most basic example is a `GET` request with no body:

```
ctx := context.TODO() // use context.TODO() if you don't know what context to use.
var body io.Reader = nil // nil readers are OK; it means there's no body.
const method = "GET"
const url = "https://eblog.fly.dev/index.html"
req, err := http.NewRequestWithContext(ctx, method, url, body) // the function will parse the URL and set the Host header; invalid URLs will return an error.
```

For a `POST` request, you'll need to provide a body. The simplest way to do this is to use `strings.NewReader` to create a reader from a string:

```
const method = "POST"
const url = "https://eblog.fly.dev/index.html"
var body io.Reader = strings.NewReader("hello, world")
req, err := http.NewRequestWithContext(ctx, method, url, body)
```

1.1.2. Headers

Requests automatically set the `Host` header (and a few others, like `User-Agent` and `Accept-Encoding`), but you'll need to set the rest yourself. Go provides a `Header` type to represent a request or response's HTTP headers. That is, the type represents the complete set of headers for a request or response, not an individual key-value pair.

`http.Header` is a `map[string][]string` with some special methods to make it easier to work with. Why `[]string`? Because HTTP allows multiple headers with the same key. For example, the following is a valid HTTP request:

```
GET / HTTP/1.1
Host: eblog.fly.dev
User-Agent: eblog/1.0
Accept-Encoding: gzip
Accept-Encoding: deflate
Some-Key: somevalue
```

All of `http.Header`'s methods *canonicalize* keys, turning them into `Title-Case`. For example, `Header.Add("accept-encoding", "gzip")` will add a header with the key `Accept-Encoding`. See the previous article for details on that.

To briefly summarize using `http.Header`:

- use `Header.Add(key, value)` to add a header: it will automatically canonicalize the key to `Title-Case` and append the value to the list of values for that key. Read as `k := AsTitle(key); Header[k] = append(Header[k], value)`.
- use `Header.Set(key, value)` to set a header: it will automatically canonicalize the key to `Title-Case` and set the value for that key to a single-element list containing the value. Read as `Header[AsTitle(key)] =`

`[]string{value}`.
- use `Header.Get` to get the *first* header matching a key, or the empty string if none are found.
- use `Header.Values(key)` to get the list of header values matching the canonical key.

Both `http.Request` and `http.Response` use the same header type.

To demonstrate, let's build the above request.

```
1
2   package main
3
4   func main() {
5       // https://go.dev/play/p/eE32qPmuDeS
6       const method = "GET"
7       const url = "https://eblog.fly.dev/index.html"
8       var body io.Reader = nil
9       req, err := http.NewRequestWithContext(context.TODO(),
    method, url, body)
10      if err != nil {
11          log.Fatal(err)
12      }
13      req.Header.Add("Accept-Encoding", "gzip")
14      req.Header.Add("Accept-Encoding", "deflate")
15      req.Header.Set("User-Agent", "eblog/1.0")
16      req.Header.Set("some-key", "a value")   // will be
    canonicalized to Some-Key
17      req.Header.Set("SOMe-KEY", "somevalue") // will
    overwrite the above since we used Set rather than Add
18      req.Write(os.Stdout)
19  }
20
```

`http.Request.Write` serializes the request as HTTP to the provided `io.Writer`. In this case, we're using `os.Stdout`, so it will print the request to the terminal.

We run the program:
IN:

```
1   go run ./main.go
```

OUT:

```
1   GET /index.html HTTP/1.1
2   Host: eblog.fly.dev
3   User-Agent: eblog/1.0
4   Accept-Encoding: gzip
5   Accept-Encoding: deflate
6   Some-Key: somevalue
```

1.2. Building URLs with `net/url.Values`

While you can build a URL by hand, query parameters can occasionally be tricky to get right, since they must be properly escaped. The `url.Values` type provides a convenient way to build query parameters using an API extremely similar to `http.Header`. In the previous article, we searched scryfall for magic cards with the word "ice" in their name, sorted by release date, in ascending order. The URL looked like this:

```
GET /search?q=ice&order=released&dir=asc HTTP/1.1
Host: scryfall.com
```

This time, let's search for cards with the phrase "of Emrakul" in their name instead. The API documentation mentions we'll need to use double quotes to search for a phrase containing a space; additionally, since it's a URL, we'll need to escape the space between "of" and "Emrakul". This could be tricky to do by hand, so let's use `url.Values`:

Let's build this request using `url.Values`:

IN:

```
// https://go.dev/play/p/OzX3Ule7Q3r
func main() {
    const method = "GET"
    v := make(url.Values)
    v.Add("q", `"of Emrakul"`) // note we use go's raw string syntax (`) to avoid having to escape the double quotes.
    v.Add("order", "released")
    v.Add("dir", "asc")
    const path = "https://scryfall.com/search"
    dst := path + "?" + v.Encode() // Encode() will escape the values for us. Remember the '?' separator!
    req, err := http.NewRequestWithContext(context.TODO(), method, dst, nil)
    if err != nil {
        log.Fatal(err)
    }
    req.Write(os.Stdout)
}
```

OUT:

```
GET /search?dir=asc&order=released&q=%22of+Emrakul%22 HTTP/1.1
Host: scryfall.com
User-Agent: Go-http-client/1.1
```

Note that Go automatically added a `User-Agent` header for us alongside the Host. `net/url` covers much more than just query parameters; it's a complete URL parser and

builder. It's not a large package; just take twenty minutes to read the documentation sometime.

1.3. Responses

Responses are broadly similar to requests; I'll cover them in more detail as we go along, but to briefly summarize:

- Access the response body using `Response.Body`, which is an `io.ReadCloser`.
- Headers are available in `Response.Header`, and the status code is in `Response.StatusCode`.
- See the full HTTP response by calling `Response.Write` with an `io.Writer`.

Only **clients** should see responses; **servers** use the `http.ResponseWriter`` API instead.

2. `http.Client`

`Client` allows you to make a `Request` to a server using `Do` and receive a `Response`.

```
1   func (c *Client) Do(req *Request) (*Response, error)
```

`Do` is the core method of `http.Client`, and all the other methods are wrappers around it. For the purpose of this article, we will *only* use `Do`; I suggest you 'do' the same, since it provides a single consistent API.

- `http.Get`, `http.Post`, `http.PostForm`, and `http.Do` are wrappers around `http.DefaultClient.Do`.
- `http.Client.Get`, `http.Client.Post` and `http.Client.PostForm` are also wrappers around `http.Client.Do`. (`PostForm` is occasionally useful, but the others conceal more than they simplify, IMO).

The following complete program, `download`, uses `http.Client` to download a file from the internet and save it to the local filesystem.

```
1   // download is a command-line tool to download a file from
    a URL.
2   // usage: download [-timeout duration] url filename
3   package main
4
5   import (
6       "context"
7       "flag"
8       "io"
9       "log"
10      "net/http"
11      "os"
12      "path/filepath"
```

```
13      "time"
14  )
15
16  func main() {
17      dir := flag.String("dir", ".", "directory to save
    file")
18      timeout := flag.Duration("timeout", 30*time.Second,
    "timeout for download")
19      flag.Parse()
20      args := flag.Args()
21      if len(args) != 2 {
22          log.Fatal("usage: download [-timeout duration] url
    filename")
23      }
24      url, filename := args[0], args[1]
25      // always set a timeout when you make an HTTP request.
26      c := http.Client{Timeout: *timeout}
27
28      // don't worry about the details of context for now;
    we'll talk about it later in this article.
29      // if you don't know what context to use, use
    context.TODO().
30      if err := downloadAndSave(context.TODO(), &c, url,
    filename); err != nil {
31          log.Fatal(err)
32      }
33  }
34  func downloadAndSave(ctx context.Context, c *http.Client,
    url, dst string) error {
35      req, err := http.NewRequestWithContext(ctx, "GET",
    url, nil)
36      if err != nil {
37          return fmt.Errorf("creating request: GET %q: %v",
    url, err)
38      }
39      resp, err :=
    c.Do(req) // Do serializes a http.Request, sends it to the
    server, and then deserializes the response to a
    http.Response.
40
41      // always check for errors after calling Do. errors
    from 'Do' usually mean something went wrong on the network.
42      if err != nil {
43          return fmt.Errorf("request: %v", err)
44      }
45      defer
    resp.Body.Close() // always close response bodies when
    you're done with them.
46
47      // immediately after checking for errors, check the
    response status code; this is how the server tells us if
    the request succeeded.
48      if resp.StatusCode != http.StatusOK {
```

```
49          return fmt.Errorf("response status: %s",
    resp.Status)
50      }
51
52      // ok, we have a successful response. let's save it to
    a file.
53
54      dstPath := filepath.Join(*dir, filename)
55      dstFile, err := os.Create(dstPath)
56      if err != nil {
57          return fmt.Errorf("creating file: %v", err)
58      }
59      defer
    dstFile.Close() // always close files when you're done with
    them.
60      if _, err := io.Copy(dstFile, resp.Body); err != nil {
61          return fmt.Errorf("copying response to file: %v",
    err)
62      }
63  }
```

Let's try it out by downloading the index page of this blog:
IN:

```
1  go build -o download ./download.go # build the program
2  ./download https://eblog.fly.dev/index.html index.html #
   save the index page to index.html
3  cat index.html # print the contents of the file
```

OUTPUT:

```
 1  <!DOCTYPE html>
 2  <html>
 3    <head>
 4      <title>index.html</title>
 5      <meta charset="utf-8" />
 6      <link rel="stylesheet" type="text/css" href="/
    dark.css" />
 7    </head>
 8    <body>
 9      <h1>articles</h1>
10      <h4><a href="/backendbasics.html">backendbasics.html</
    a></h4>
11      <h4><a href="/console.html">console.html</a></h4>
12      <h4><a href="/mermaid_test.html">mermaid_test.html</
    a></h4>
13      <h4><a href="/
    benchmark_results.html">benchmark_results.html</a></h4>
14      <h4><a href="/quirks2.html">quirks2.html</a></h4>
15      <h4><a href="/fastdocker.html">fastdocker.html</a></h4>
16      <h4><a href="/faststack.html">faststack.html</a></h4>
17      <h4><a href="/console-autocomplete.html">console-
    autocomplete.html</a></h4>
18      <h4><a href="/quirks3.html">quirks3.html</a></h4>
```

```
      <h4><a href="/cheatsheet.html">cheatsheet.html</a></h4>
      <h4><a href="/bytehacking.html">bytehacking.html</a></h4>
      <h4><a href="/quirks.html">quirks.html</a></h4>
      <h4><a href="/reflect.html">reflect.html</a></h4>
      <h4><a href="/startfast.html">startfast.html</a></h4>
      <h4><a href="/performanceanxiety.html">performanceanxiety.html</a></h4>
      <h4><a href="/article_list.html">article_list.html</a></h4>
      <h4><a href="/noframework.html">noframework.html</a></h4>
      <h4><a href="/onoff.html">onoff.html</a></h4>
      <h4><a href="/README.html">README.html</a></h4>
      <h4><a href="/testfast.html">testfast.html</a></h4>
      <h4><a href="/index.html">index.html</a></h4>
    </body>
</html>
```

If we set the timeout to `1ms`, we'll get an error:

IN:

```
./download -timeout 1ms https://eblog.fly.dev/index.html
index.html
```

OUT:

```
2023/09/09 09:33:15 request: Get "https://eblog.fly.dev/
index.html": context deadline exceeded (Client.Timeout
exceeded while awaiting headers)
```

There's that word `context` again: wherever you see timeouts and cancellation, you'll see `context`.

Clients are safe for concurrent use across multiple goroutines. The following complete program, `pardownload` downloads a list of URLs in parallel, saving them to the specified directory; it's a straightforward extension of the previous program.

In general, you should reuse a `http.Client` as much as possible; avoid creating a new one for each request.

```
// pardownload downloads a list of URLs in parallel, saving
them to the specified directory.
// It exits with a nonzero status code if any of the
downloads fail, where the status code is the number of
failed downloads.
package main

import (
    "context"
    "flag"
    "fmt"
```

```
 9      "io"
10      "log"
11      "net/http"
12      "os"
13      "path/filepath"
14      "sync"
15      "time"
16  )
17
18  func main() {
19      var dstDir string
20      var client
    http.Client // the zero value of http.Client is a usable
    client.
21      flag.StringVar(&dstDir, "dst", "", "destination
    directory; defaults to current directory")
22      // set the timeout for the client using a command-line
    flag.
23      flag.DurationVar(&client.Timeout, "timeout",
    1*time.Minute, "timeout for the request")
24      flag.Parse()
25
26      src := flag.Args()
27      if len(src) == 0 {
28          log.Fatalf("can't copy")
29      }
30      dstDir, err := filepath.Abs(dstDir) // make the
    destination directory absolute, so our error messages are
    easier to read.
31      if err != nil {
32          log.Fatalf("invalid destination directory: %v",
    err)
33      }
34      dst := make([]string, len(src)) // make a slice of the
    same length as src, so we can access it in parallel,
    without worrying about synchronization.
35      for i := range src {
36          dst[i] = filepath.Join(dstDir,
    filepath.Base(src[i]))
37      }
38
39      errs := make([]error, len(src)) // similarly, make a
    slice of errors.
40
41      wg := new(sync.WaitGroup) // a WaitGroup waits for a
    collection of goroutines to finish.
42      wg.Add(len(src))          // add the number of
    goroutines we're going to wait for.
43
44      now := time.Now()
45      for i := range src {
46          i := i // see https://golang.org/doc/
    faq#closures_and_goroutines
47          go func() {
```

```
48              defer wg.Done() // tell the WaitGroup that
    we're done.
49              // this is a simple function, so we don't
    'really' need to defer, but it's a good habit to get into.
50              errs[i] = downloadAndSave(context.TODO(),
    &client, src[i], dst[i])
51          }()
52      }
53      wg.Wait() // wait for all the goroutines to finish.
54
55      log.Printf("downloaded %d files in %v", len(src),
    time.Since(now))
56      var errCount int // number of errors
57      for i := range errs {
58          if errs[i] != nil {
59              log.Printf("err: %s -> %s: %v", src[i],
    dst[i], errs[i])
60              errCount++
61          } else {
62              log.Printf("ok: %s -> %s", src[i], dst[i])
63          }
64      }
65      os.Exit(errCount) // nonzero exit codes indicate
    failure.
66  }
```

That covers an outbound `*http.Request`: let's talk about serving incoming ones.

3. Handlers & Servers

The `http.Handler` interface is the core of Go's HTTP server. We might expect the interface to mirror the Client's `Do` method:

```
1  func (c *Client) Do(req *Request) (*Response, error)
```

Maybe something like this:

```
1  type NotQuiteHandler interface { ServeHTTP(req
    *http.Request) (*http.Response, error) }
```

But this is not the case. Among other things, HTTP is a *streaming* response protocol; we need some way to write the response body as it's generated without buffering the entire response in memory; something that might be inadviseable or even impossible for large responses (like a file download).

So instead of returning a response, `http.Handler` has the following signature:

```
1  type Handler interface { ServeHTTP(http.ResponseWriter,
    *http.Request) }
```

We've already seen plenty of `*http.Request`, so let's talk about `http.ResponseWriter`. It's an interface with three methods: `Header`, `Write`, and `WriteHeader`. It's the `http.Handler`'s job to call these methods to construct the response.

```
1  // ResponseWriter interface is used by an HTTP handler to
   construct a HTTP response.
2  type ResponseWriter interface {
3      // Get access to the Response headers. Headers must be
   written before the first call to Write.
4      Header() Header // same underlying type as
   http.Request.Header
5
6      // Write data to the response body.
7      Write([]byte) (int, error)
8
9      // WriteHeader sends an HTTP response header with the
   provided
10     // status code. (i.e, 200, 404, etc.) This must be
   called before the first call to Write; otherwise,
11     // an implicit WriteHeader(http.StatusOK) will be sent.
12     WriteHeader(statusCode int)
13 }
```

Build a `Server` by passing it a `Handler` and address to listen on, then calling `Server.ListenAndServe`.

The following complete program demonstrates a minimal HTTP server that returns a 200 OK response with the text "hello, world".

IN:

```
1  // https://go.dev/play/p/AjoS1drDEpn
2  func main() {
3      server := http.Server{Addr: ":8080", Handler:
   TextHandler("hello, world!\r\n")}
4      go server.ListenAndServe()
5      req, _ := http.NewRequestWithContext(context.TODO(),
   "GET", "http://localhost:8080", nil)
6      resp, err := new(http.Client).Do(req)
7      _ = err
8      defer resp.Body.Close()
9      resp.Write(os.Stdout) // print the response to stdout.
10
11 }
12
13 // TextHandler is a simple http.Handler that returns a 200
   OK response with the provided text.
14 type TextHandler string
15 var _ http.Handler = TextHandler("") // ensure TextHandler
   implements http.Handler
16 func (t TextHandler) ServeHTTP(w http.ResponseWriter, _
   *http.Request) { w.Write([]byte(t)) } // implicit 200 OK
```

OUT:

```
HTTP/1.1 200 OK
Content-Length: 12
Content-Type: text/plain; charset=utf-8
Date: Tue, 10 Nov 2009 23:00:00 GMT
```

Note that the `Content-Length`, `Date`, and `Content-Type` headers were automatically added by the server. For small payloads, it is OK to rely on the `Content-Length`, but you should always set the `Content-Type` header; go's builtin sniffer often confuses `text/plain` and `encoding/json`.

The server has four different configurable timeouts:

Field	Description	Inherits From?
`ReadTimeout`	The maximum amount of time to wait for the client to send a request.	N/A
`WriteTimeout`	The maximum amount of time to wait for the server to send a response.	N/A
`IdleTimeout`	The maximum amount of time to wait for the client to send a new request on a persistent connection.	`ReadTimeout`
`ReadHeaderTimeout`	The maximum amount of time to wait for the client to send the request headers.	`ReadTimeout`

I **strongly recommend you set** `ReadTimeout` **and** `WriteTimeout` **for all servers**; the default of `0` means 'no timeout', which opens you up to denial of service attacks. `IdleTimeout` is also a good idea, but it's less important.

3.1. `http.HandlerFunc`

The `http.Handler` interface only has a single method, `ServeHTTP`, so it seems like overkill to have to define a new type for every handler, when you could just use a function. Use the `http.HandlerFunc` type to turn a function of type `func(http.ResponseWriter, *http.Request)` into an `http.Handler`.

The following complete program is identical to the previous one, but uses `http.HandlerFunc` instead of a custom type.

```
// https://go.dev/play/p/Cc8AMjR-_sc
func helloWorld(w http.ResponseWriter, r *http.Request) {
w.Write([]byte("hello, world!\r\n")) }
func main() {
    server := http.Server{Addr: ":8080", Handler:
http.HandlerFunc(helloWorld)}
    go server.ListenAndServe()
```

```
6       req, _ := http.NewRequestWithContext(context.TODO(),
    "GET", "http://localhost:8080", nil)
7       resp, err := new(http.Client).Do(req)
8       _ = err
9       defer resp.Body.Close()
10      resp.Write(os.Stdout) // print the response to stdout.
11  }
```

In practice, `HandlerFunc` **is significantly more common than custom** `http.Handler`**s**, especially when middleware is involved. We'll talk more about that in the next article.

4. Sending and Receiving JSON

`JSON` is by far the most popular serialization format for web APIs. The combination of HTTP and JSON, often (if inaccurately) called "REST", is the most common way to build web APIs today. The `encoding/json` package provides a complete JSON encoder and decoder.

The following cheatsheet summarizes the `encoding/json` API:

Function	Description	Note
`json.Marshal`	Encode a value as JSON.	
`json.Unmarshal`	Decode a value from JSON.	Always pass a non-nil pointer to `Unmarshal`.
`json.Marshaler`	Implement this interface to customize JSON encoding.	Usually not necessary.
`json.Unmarshaler`	Implement this interface to customize JSON decoding.	Usually not necessary.
`json.NewEncoder`	Create a new JSON encoder around an `io.Writer`.	Then call `Encode` to encode a value.
`json.NewDecoder`	Create a new JSON decoder around an `io.Reader`.	Then pass a non-nil pointer to `Decode` to decode a value.
`json.RawMessage`	As `[]byte`, but implements `json.Marshaler` and `json.Unmarshaler`.	Useful for 'pass-through' JSON APIs.

The following complete program demonstrates how to use `encoding/json` to send and receive JSON.

The REQUEST body will be a JSON object containing two optional fields: "Format" and "TZ"

```
1   {
2     "format": "RFC3339",
```

```
  "tz": "America/New_York"
}
```

The RESPONSE body will be a JSON object containing exactly one of the following fields: `"time"` or `"error"`.

```
{
  "time": "2021-09-09T09:33:15Z"
}
```

```
{
  "error": "unknown time zone faketz"
}
```

We will use [json struct tags`](#) to customize the JSON encoding and decoding of our structs.

IN:

```
// https://go.dev/play/p/A8QVJwFEeq3
type Request struct {
    Format string `json:"format"`  // Format, as in time.Format. If empty, use time.RFC3339.
    TZ     string `json:"tz"`      // TZ, as in time.LoadLocation. If empty, use time.Local.
}
 // The time, formatted according to the request's Format and TZ.
type Resp struct {Time time.Time `json:"time"`} // no need for omitempty here; we'll never send a zero time.
type Error struct {Error string `json:"error"`} // no need for omitempty here; we'll never send an empty error.
```

Our handler will use these structs and `json.NewDecoder` to decode the request body, and `json.NewEncoder` to encode the response body.

```
// https://go.dev/play/p/A8QVJwFEeq3
// http handler: writes current time as JSON object (`{"Time": <time>}`)
func getTime(w http.ResponseWriter, r *http.Request) {
    var req Request
    w.Header().Set("Content-Type", "encoding/json")
    if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
        w.WriteHeader(400) // bad request
        json.NewEncoder(w).Encode(Error{err.Error()})
        return
    }
    r.Body.Close() // always close request bodies when you're done with them.
    var tz *time.Location = time.Local
    if req.TZ != "" {
        var err error
```

```
            tz, err = time.LoadLocation(req.TZ)
            if err != nil || tz == nil {
                w.WriteHeader(400) // bad request
                json.NewEncoder(w).Encode(Error{err.Error()})
                return
            }
        }
        format := time.RFC3339
        if req.Format != "" {
            format = req.Format
        }

        resp := Response{time.Now().In(tz).Format(format)}
        json.NewEncoder(w).Encode(resp)

    }

```

IN:

```
    // https://go.dev/play/p/A8QVJwFEeq3
    var client = &http.Client{Timeout: 2 * time.Second}

    func sendRequest(tz, format string) {
        body := new(bytes.Buffer)
        json.NewEncoder(body).Encode(Request{TZ: tz, Format: format})
        log.Printf("request body: %v", body)
        req, err := http.NewRequestWithContext(context.TODO(), "GET", "http://localhost:8080", body)
        if err != nil {
            panic(err)
        }
        resp, err := client.Do(req)
        if err != nil {
            panic(err)
        }
        resp.Write(os.Stdout)
        resp.Body.Close() // always close response bodies when you're done with them.
    }
    func main() {
        server := http.Server{Addr: ":8080", Handler: http.HandlerFunc(getTime)}
        go server.ListenAndServe()

        sendRequest("", "") // rely on defaults
        sendRequest("America/Los_Angeles", time.RFC3339)
        sendRequest("America/New_York", time.RFC822Z) // "02 Jan 06 15:04 -0700" // RFC822 with numeric zone
        sendRequest("faketz", "") // should get 400 Bad Request

    }
```

Let's try it out:

OUT:

```
2009/11/10 23:00:00 request body: {"format":"","tz":""}
HTTP/1.1 200 OK
Content-Length: 32
Content-Type: encoding/json
Date: Tue, 10 Nov 2009 23:00:00 GMT

{"Time":"2009-11-10T23:00:00Z"}
2009/11/10 23:00:00 request body:
{"format":"2006-01-02T15:04:05Z07:00","tz":"America/
Los_Angeles"}
HTTP/1.1 200 OK
Content-Length: 37
Content-Type: encoding/json
Date: Tue, 10 Nov 2009 23:00:00 GMT

{"Time":"2009-11-10T15:00:00-08:00"}
2009/11/10 23:00:00 request body: {"format":"02 Jan 06
15:04 -0700","tz":"America/New_York"}
HTTP/1.1 200 OK
Content-Length: 33
Content-Type: encoding/json
Date: Tue, 10 Nov 2009 23:00:00 GMT

{"Time":"10 Nov 09 18:00 -0500"}
2009/11/10 23:00:00 request body:
{"format":"","tz":"faketz"}
HTTP/1.1 400 Bad Request
Content-Length: 37
Content-Type: encoding/json
Date: Tue, 10 Nov 2009 23:00:00 GMT

{"error":"unknown time zone faketz"}
```

A few hints on producing good JSON APIs:

- Always set the `Content-Type` header to `application/json`.
- Your top-level response should almost always be a JSON object, not an array or string. That is, return `{"data": <data>}` instead of `<data>`.
- Avoid `map[string]any`; this is tempting for programmers used to javascript, lua, or python, but it's a bad idea in Go. Instead, define a new type for each request or response.
- Avoid lists of heterogeneous objects. That is, wherever possible, try to have `[]int` or `[]string` instead of `[]any`.
- This allows you to add additional fields in the future without breaking clients. Some APIs like an additional level of nesting, but I find this to be overkill.
- You can use anonymous structs to avoid having to define a new type for every response while maintaining type safety.

4.1. JSON Helper functions w/ generics

Reading and writing JSON can seem tedious. The following generic functions can help reduce boilerplate and help you avoid common 'gotchas', like forgetting to close the response body.

```
1  // ReadJSON reads a JSON object from an io.ReadCloser,
   closing the reader when it's done. It's primarily useful
   for reading JSON from *http.Request.Body.
2  func ReadJSON[T any](r io.ReadCloser) (T, error) {
3      var v T // declare a variable of type T
4      err :=
   json.NewDecoder(r).Decode(&v) // decode the JSON into v
5      return v, errors.Join(err, r.Close()) // close the
   reader and return any errors.
6  }
7
8  // WriteJSON writes a JSON object to a http.ResponseWriter,
   setting the Content-Type header to application/json.
9  func WriteJSON(w http.ResponseWriter, v any) error {
10     w.Header().Set("Content-Type", "application/json")
11     return json.NewEncoder(w).Encode(v)
12 }
```

Similarly, you may wish to define some helper functions for your own JSON APIs.

```
1  // WriteError logs an error, then writes it as a JSON object
   in the form {"error": <error>}, setting the Content-Type
   header to application/json.
2  func WriteError(w http.ResponseWriter, err error, code int)
   {
3      og.Printf("%d %v: %v", code, http.StatusText(code),
   err) // log the error; http.StatusText gets "Not Found" from
   404, etc.
4      w.Header().Set("Content-Type", "encoding/json")
5      w.WriteHeader(code)
6      json.NewEncoder(w).Encode(Error{err.Error()})
7  }
```

The combination of anonymous structs and generics allows us to write much more compact handlers without dragging in a full-blown web framework.

Let's rewrite the logic of `getTime` to use this technique.

```
1
2  // http handler: writes current time as JSON object
   (`{"Time": <time>}`)
3  func getTime(w http.ResponseWriter, r *http.Request) {
4      req, err := ReadJSON[struct {TZ, Format string }]
   (r.Body)
5      if err != nil {
6          WriteError(w, err, 400)
7          return
```

```
 8       }
 9       var tz *time.Location = time.Local
10       if req.TZ != "" {
11           var err error
12           tz, err = time.LoadLocation(req.TZ)
13           if err != nil {
14               WriteError(w, err, 400)
15               return
16           }
17       }
18       format := time.RFC3339
19       if req.Format != "" {
20           format = req.Format
21       }
22       WriteJSON(w,
     Response{time.Now().In(tz).Format(format)})
23   }
```

5. Other Serialization Formats

While JSON is by far the most popular serialization format, it's not appropriate for everything. JSON is rather slow and inefficient, especially for numeric data or deeply nested structs or arrays, where repeated field names and braces can take up a significant amount of space. Usually, **compressing JSON with gzip is a better idea than using a different serialization format**. This is not because `JSON` or `gzip` are particularly good,but because both are so ubiquitous that they're almost always available, regardless of language or platform, and the combination has reasonable performance under most circumstances.

That being said, there are a few other serialization formats that are worth knowing about. The following table summarizes the most common ones.

Format/ Location	Description	Portable?	Note
encoding/ json	JSON, the JavaScript Object Notation.	Everywhere.	Slow and verbose, but ubiquitous. **Just use JSON.**
encoding/gob	Go's builtin serialization format.	Go only.	Reasonably fast, but not blazingly so.
encoding/xml	XML, the eXtensible Markup Language.	Usually.	Slow and verbose, but ubiquitous. XML 1.0 only.
encoding/ base64	Base64, a binary-to-text encoding.	Yes	Useful for embedding binary data in JSON or URLs.
encoding/csv	Comma-separated values.	Not really.	Slow & awkward, but spreadsheets are universal.
encoding/ binary	Binary serialization.	Be careful with endianness	

Format/ Location	Description	Portable?	Note
			Usually requires code generation or careful manual work.
`go-yaml/yaml`	YAML, a superset of JSON common in SAAS configuration.	Everywhere.	Avoid; YAML is wildly complicated and has many subtle bugs.
`golang/protobuf`	Protocol Buffers, a binary serialization format.	Everywhere.	Fast, but requires code generation. v2 is very painful to use in Go; v3 is much better.
`google/flatbuffers`	Binary serialization format that allows for zero-copy deserialization.	Everywhere.	Fast, but requires code generation, and the API is a little awkward.

We've now covered the basics of HTTP servers and clients, but there's one big piece of missing Context (pun intended): timeouts and cancellation.

6. Contexts and Cancellation

Any internet communication can fail. The network can go down, the server can crash, or the server can just be slow. When I make a network call, I'm implicitly expecting it to finish *soon*, not just 'eventually'; it's no good for me if my request to buy a plane ticket finishes after the flight has already left.

Go's `context.Context` type is for managing state 'about' a function, rather than 'in' a function. These break down into two large groups: function metadata (start time, request IDs) and deadlines/cancellation. This package is too complex for me to cover in detail here, so I strongly recommend you read both the package documentation and the blog post that introduced it.

In short, context is used for two related-but-distinct purposes:

6.1. Context as a key-value storage to carry metadata about a request across function boundaries

```
1  func DoSomething(ctx context.Context) {
2      reqID := ctx.Value(trace.Key).(string) // get the
   request ID from the context.
3      log.Printf("request %s: starting", reqID)
4      defer log.Printf("request %s: done", reqID)
5      // ...
6  }
```

Don’t worry about this too much; we’ll cover this in more detail in the next article, when we talk about middleware.

6.2. Context as a ceilling on execution time

```
func MakeRequest(ctx context.Context, someArg string) error {
    if err := ctx.Err(); err != nil {
        return err // out of time; don't even try.
    }
}
```

Use a context to set a ceiling on execution time for any function that does I/O. I/O includes, but is not limited to, network requests, database queries, and file operations. Always set a timeout for any I/O operation, even if it’s expected to take only a few milliseconds. Failure to set timeouts can lead to indefinite resource leaks, denial of service attacks, or hard-to-track-down bugs.

The **context should always be the first argument** to any function that that does I/O or could possibly time out. This makes it easy to propagate cancellation signals. For example, the following function uses `context.WithTimeout` to set a timeout for a network request.

```
func GetGoogle(ctx context.Context) error {
    // deadline of the context is either 1 second from now, or the deadline of the parent context, whichever is sooner.
    ctx, cancel := context.WithTimeout(ctx, 1*time.Second)
    defer cancel() // always call cancel when you're done with the context to free associated resources.
    req, err := http.NewRequestWithContext(ctx, "GET", "https://google.com", nil)
    if err != nil {
        return err
    }
    resp, err := new(http.Client).Do(req)
    if err != nil {
        return err
    }
    resp.Write(os.Stdout) // print the response to stdout.
    return nil
}
```

The following table summarizes the parts of the `context` API that are relevant to timeouts and cancellation.

Function	Description	Usage
`context.Background()`	returns a context with no metadata or cancellation signal; effective ‘zero value’	To start a new context chain.

Function	Description	Usage
`context.TODO()`	as `Background()`	During prototyping, when you don't know what context to use.
`context.WithCancel(parent)`	returns a context with a cancellation signal that is triggered when `parent` is cancelled	To propagate cancellation signals; usually `WithTimeout` or `WithDeadline` are more appropriate.
`context.WithDeadline(parent, deadline)`	returns a context with a cancellation signal that is triggered when `deadline` elapses	To preserve a deadline set by another service.
`context.WithTimeout(parent, timeout)`	returns a context with a deadline equal to `time.Now().Add(timeout)`	For any I/O operation.

Many of Go's core packages contain both `context` and `context`-free versions of functions; **always use the `context` version under all circumstances**. If you don't know what context to use, use `context.TODO()`.

The following table summarizes common functions that take a context and their replacements.

Function	Replacement	Description	Note
`http.NewRequest`	`http.NewRequestWithContext`	Build an outgoing HTTP request.	
`sql.DB.Query`	`sql.DB.QueryContext`	Query a database.	
`sql.DB.Exec`	`sql.DB.ExecContext`	Execute a database query.	
File.Read / File.Write	N/A	File I/O.	Close the file in a separate goroutine after a timeout instead.
`net.Dial`			

Function	Replacement	Description	Note
	`DialTimeout` / `net.Dialer.DialContext`	Dialoll a network connection.	

6.3. Contexts and HTTP: Clients

Add a context to an outgoing HTTP request using `http.NewRequestWithContext(ctx, method, url, body)`. That's pretty much it.

6.4. Contexts and HTTP: Servers

You have three options, which overlap in functionality:

- The `BaseContext` field of `http.Server` is a context for the *listener*, which is then passed to each request. Note that cancelling this context will cancel *all* requests based on that listener, so it's generally only appropriate for 'universal' shutdowns, like handling `SIGINT` from the OS. See my article on turning off software for more details on signal handling and graceful shutdowns.
- The `ConnContext` field of `http.Server` is the default context for each TCP connection. This is useful for setting a timeout for *all* requests on a connection, even if they're properly sending packets back and forth.
- You can set the request context manually in a `http.Handler` by wrapping the `http.ResponseWriter` and `*http.Request` in a new `http.Request` with a new context. This is useful for setting a timeout for *individual* requests, and adding metadata to the context. We'll talk about this one more in the next article when we get into middleware.

It's your job to check the cancellation signal and return an error if it's set. You can check the cancellation signal in two ways: `ctx.Err()` returns the cancellation error; `ctx.Done()` returns a channel that is closed when the context is cancelled.

Either way, you should return an error if the context is cancelled.

7. Conclusion

In our first article, we covered the basics of the HTTP protocol. Now we've done a rough-and-ready tour of Go's HTTP client and server APIs. For some web servers, this is all you need; and I'd encourage you to use the techniques we've covered to build a

simple HTTP server and client and get some practice building APIS. Still, though, we're missing a few key pieces of the puzzle:

- **Middleware**: How do we add common functionality to a web server, like logging, authentication, and rate limiting?
- **Routing**: How do we map URLs and methods to handlers?
- **Databases & Dependencies**: How do we store and retrieve data? How do we connect to a database? How should we structure our APIs to deal with dependencies like these?

We'll cover all of these in the next article.

MORE ARTICLES