Introduction to Microservices

****

First, we are going to look at how easy it is to create a simple web server with a single endpoint using the net/http package. Then, we will move on to examine the encoding/json package to see just how easy Go makes it for us to use JSON objects for our requests and our responses. Finally, we will look at how routing and handlers work and how we can manage context between these handlers.

# Building a simple web server with net/http

The net/http package provides all the features we need to write HTTP clients and servers. It gives us the capability to send requests to other servers communicating using the HTTP protocol as well as the ability to run a HTTP server that can route requests to separate Go funcs, serve static files, and much more.

To begin we should ask the question, *what technical book would be complete without a simple hello world example?* I say none and this is exactly where we will begin.

In this example, we are going to create an HTTP server with a single endpoint that returns static text represented by the JSON standard, this will introduce the basic functions of the HTTP server and handlers. We will then modify this endpoint to accept a request that is encoded in JSON and using the encoding/json package return a response to the client. We will also examine how the routing works by adding a second endpoint that returns a simple image.

By the end of this chapter, you will have a fundamental grasp of the basic packages and how you can use them to quickly and efficiently build a simple microservice.

Building a web server in Go is incredibly easy thanks to the HTTP package, which is distributed as part of the standard library.

It has everything you need to manage routing, dealing with **Transport Layer Security** (**TLS**), which we will cover in Chapter 8, *Security*, support for HTTP/2 out of the box, and the capability to run an incredibly efficient server that can deal with a huge number of requests.

The source code for this chapter can be found on GitHub at [http://github.com/building-](http://github.com/building-microservices-with-go/chapter1.git) [microservices-with-go/chapter1.git](http://github.com/building-microservices-with-go/chapter1.git), all the examples in this and subsequent chapters will reference the source extensively so if you have not already done so, go and clone this repo before continuing.

Let's look at the syntax for creating a basic server then we can walk through the packages in more depth:

Example 1.0 basic\_http\_example/basic\_http\_example.go

09 func main() { 10 port := 8080

11

12 http.HandleFunc("/helloworld", helloWorldHandler) 13

1. log.Printf("Server starting on port %v\n", 8080)
2. log.Fatal(http.ListenAndServe(fmt.Sprintf(":%v", port), nil)) 16 }

17

1. func helloWorldHandler(w http.ResponseWriter, r \*http.Request) {
2. fmt.Fprint(w, "Hello World\n") 20 }

The first thing we are doing is calling the HandleFunc method on the http package. The HandleFunc method creates a Handler type on the DefaultServeMux handler, mapping the path passed in the first parameter to the function in the second parameter:

func HandleFunc(pattern string, handler func(ResponseWriter, \*Request))

In line **15** we start the HTTP server, ListenAndServe takes two parameters, the TCP network address to bind the server to and the handler that will be used to route requests:

func ListenAndServe(addr string, handler Handler) error

In our example, we are passing the network address :8080" this means we would like to bind the server to all available IP addresses on port 8080.

The second parameter we are passing is nil, this is because we are using the DefaultServeMux handler, which we are setting up with our call to http.HandleFunc. In Chapter 3, *Introducing Docker*, you will see the use of this second parameter when we introduce more sophisticated routers, but for now we can ignore it.

If the ListenAndServe function fails to start a server it will return an error, the most common reason for this is that you may be trying to bind to a port that is already in use on the server. In our example, we are passing the output of ListenAndServe straight to log.Fatal(error), which is a convenience function equivalent to calling fmt.Print(a

...interface{}) followed by a call to os.Exit(1). Since ListenAndServe blocks if the server starts correctly we will never exit on a successful start.

Let's quickly run and test our new server:

**$ go run ./basic\_http\_example.go**

You should now see the application output:

**2016/07/30 01:08:21 Server starting on port 8080**

What if you do not see the preceding output and instead see something like the following?

**2016/07/19 03:51:11 listen tcp :8080: bind: address already in use exit status 1**

Take another look at the signature of ListenAndServe and the way we are calling it. Remember what we were saying about why we were using log.Fatal?

If you do get this error message it means that you are already running an application on your computer that is using port 8080, this could be another instance of your program or it could be another application. You can check that you do not have another instance running by checking the running processes:

**$ ps -aux | grep 'go run'**

If you do see another go run ./basic\_http\_example.go then you can simply kill it and retry. If you do not have another instance running, then you probably have some other software that is bound to this port. Try changing the port on line **10** and restart your program.

To test the server, open a new browser and type in the URI http://127.0.0.1:8080/helloworld and if things are working correctly you should see the following response from the server:

Hello World

Congratulations, that's the first step into microservice mastery. Now that we have our first program running, let's take a closer look at how we can return and accept JSON.

# Reading and writing JSON

Thanks to the encoding /json package, which is built into the standard library encoding and decoding JSON to and from Go types is both fast and easy. It implements the simplistic Marshal and Unmarshal functions; however, if we need them, the package also provides Encoder and Decoder types that allow us greater control when reading and writing streams of JSON data. In this section, we are going to examine both of these approaches, but first let's take a look at how simple it is to convert a standard Go struct into its corresponding JSON string.

## Marshalling Go structs to JSON

To encode JSON data, the encoding/json package provides the Marshal function, which has the following signature:

func Marshal(v interface{}) ([]byte, error)

This function takes one parameter, which is of type interface, so pretty much any object you can think of since interface represents any type in Go. It returns a tuple of ([]byte, error), you will see this return style quite frequently in Go, some languages implement a try catch approach that encourages an error to be thrown when an operation cannot be performed, Go suggests the pattern (return type, error), where the error is nil when an operation succeeds.

In Go, unhandled errors are a bad thing, and whilst the language does implement Panic and Recover, which resemble exception handling in other languages, the situations where you should use these are quite different (see *The Go Programming Language*, Kernaghan). In Go, the panic function causes normal execution to stop and all deferred function calls in the Go routine are executed, the program will then crash with a log message. It is generally used for unexpected errors that indicate a bug in the code and good robust Go code will attempt to handle these runtime exceptions and return a detailed error object back to the calling function.

This pattern is exactly what is implemented with the Marshal function. In the instance that Marshal cannot create a JSON encoded byte array from the given object, which could be due to a runtime panic, then this is captured and an error object detailing the problem is returned to the caller.

Let's try this out, expanding on our existing example, instead of simply printing a string from our handler, let's create a simple struct for the response and return this instead.

Example 1.1 reading\_writing\_json\_1/reading\_writing\_json\_1.go

1. type helloWorldResponse struct {
2. Message string 12 }

In our handler, we will create an instance of this object, set the message, then use the

Marshal function to encode it to a string before returning.

Let's see what that will look like:

1. func helloWorldHandler(w http.ResponseWriter, r \*http.Request) {
2. response := helloWorldResponse{Message: "HelloWorld"}
3. data, err := json.Marshal(response)
4. if err != nil {
5. panic("Ooops") 28 }

29

30 fmt.Fprint(w, string(data)) 31 }

Now, when we run our program again and refresh our browser, we see the following output rendered in valid JSON:

{"Message":"Hello World"}

This is awesome; however, the default behavior of Marshal is to take the literal name of the field and use this as the field in the JSON output. What if I prefer to use camel case and would rather see "message", could we just rename the field in the helloWorldResponse struct?

Unfortunately we can't, as in Go, lowercase properties are not exported, Marshal will ignore these and will not include them in the output.

All is not lost as the encoding/json package implements struct field attributes that allow us to change the output for the property to anything we choose.

Example 1.2 reading\_writing\_json\_2/reading\_writing\_json\_2.go

1. type helloWorldResponse struct {
2. Message string `json:"message"` 12 }

Using the struct field's tags, we can have greater control of how the output will look. In the preceding example, when we marshal this struct the output from our server would be:

{"message":"Hello World"}

This is exactly what we want, but we can use field tags to control the output even further. We can convert object types and even ignore a field altogether if we need to:

type helloWorldResponse struct {

// change the output field to be "message" Message string `json:"message"`

// do not output this field Author string `json:"-"`

// do not output the field if the value is empty Date string `json:",omitempty"`

// convert output to a string and rename "id" Id int `json:"id, string"`

}

Channel, complex types, and functions cannot be encoded in JSON; attempting to encode these types will result in an UnsupportedTypeError being returned by the Marshal function.

It also can't represent cyclic data structures; if your stuct contains a circular reference then

Marshal will result in an infinite recursion, which is never a good thing for a web request.

If we want to export our JSON prettily formatted with indentation, we can use the MarshallIndent function, this allows you to pass an additional parameter of string to specify what you would like the indent to be. Two spaces right, not a tab?

func MarshalIndent(v interface{}, prefix, indent string) ([]byte, error)

The astute reader might have noticed that we are decoding our struct into a byte array and then writing that to the response stream, this does not seem to be particularly efficient and in fact it is not. Go provides Encoders and Decoders, which can write directly to a stream, since we already have a stream with the ResponseWriter then let's do just that.

Before we do, I think we need to look at the ResponseWriter a little to see what is going on there.

The ResponseWriter is an interface that defines three methods:

// Returns the map of headers which will be sent by the

// WriteHeader method. Header()

// Writes the data to the connection. If WriteHeader has not

// already been called then Write will call

// WriteHeader(http.StatusOK). Write([]byte) (int, error)

// Sends an HTTP response header with the status code. WriteHeader(int)

If we have a ResponseWriter interface, how can we use this with fmt.Fprint(w io.Writer, a ...interface{})? This method requires a Writer interface as a parameter and we have a ResponseWriter interface. If we look at the signature for Writer we can see that it is:

Write(p []byte) (n int, err error)

Because the ResponseWriter interface implements this method, it also satisfies the interface Writer and therefore any object that implements ResponseWriter can be passed to any function that expects Writer.

Amazing, Go rocks, but we have not answered our question, *Is there any better way to send our data to the output stream without marshalling to a temporary object before we return it?*

The encoding/json package has a function called NewEncoder this returns us an Encoder object that can be used to write JSON straight to an open writer and guess what; we have one of those:

func NewEncoder(w io.Writer) \*Encoder

So instead of storing the output of Marshal into a byte array, we can write it straight to the HTTP response.

Example 1.3 reading\_writing\_json\_3/reading\_writing\_json\_3.go:

func helloWorldHandler(w http.ResponseWriter, r \*http.Request) { response := HelloWorldResponse{Message: "HelloWorld"} encoder := json.NewEncoder(w)

encoder.Encode(&response)

}

We will look at benchmarking in a later chapter, but to see why this is important we have created a simple benchmark to check the two methods against each other, have a look at the output.

Example 1.4 reading\_writing\_json\_2/reading\_writing\_json\_2.go:

**$go test -v -run="none" -bench=. -benchtime="5s" -benchmem**

**BenchmarkHelloHandlerVariable-8 20000000 511 ns/op 248 B/op 5 allocs/op**

**BenchmarkHelloHandlerEncoder-8 20000000 328 ns/op 24 B/op 2 allocs/op**

**BenchmarkHelloHandlerEncoderReference-8 20000000 304 ns/op 8 B/op 1 allocs/op**

**PASS**

**ok github.com/building-microservices-with- go/chapter1/reading\_writing\_json\_2 24.109s**

Using Encoder rather than marshalling to a byte array is nearly 50% faster. We are dealing with nanoseconds here so that time may seem irrelevant, but it isn't; this was two lines of code. If you have that level of inefficiency throughout the rest of your code then your application will run slower, you will need more hardware to satisfy the load and that will cost you money. There is nothing clever in the differences between the two methods all we have done is understood how the standard packages work and chosen the correct option for our requirements, that is not performance tuning, that is understanding the framework.

## Unmarshalling JSON to Go structs

Now we have learned how we can send JSON back to the client, what if we need to read input before returning the output? We could use URL parameters and we will see what that is all about in the next chapter, but commonly you will need more complex data structures that involve the service to accept JSON as part of an HTTP POST request.

Applying similar techniques that we learned in the previous section to write JSON, reading JSON is just as easy. To decode JSON into a stuct field the encoding/json package provides us with the Unmarshal function:

func Unmarshal(data []byte, v interface{}) error

The Unmarshal function works in the opposite way to Marshal; it allocates maps, slices, and pointers as required. Incoming object keys are matched using either the struct field name or its tag and will work with a case-insensitive match; however, an exact match is preferred. Like Marshal, Unmarshal will only set exported struct fields, those that start with an upper-case letter.

We start by adding a new struct field to represent the request, whilst Unmarshal can decode the JSON into an interface{}, which would be of map[string]interface{} // for JSON objects type or: []interface{} // for JSON arrays, depending if our JSON is an object or an array.

In my opinion it is much clearer to the readers of our code if we explicitly state what we are expecting as a request. We can also save ourselves work by not having to manually cast the data when we come to use it.

Remember two things:

You do not write code for the compiler, you write code for humans to understand You will spend more time reading code than you do writing it

Taking these two points into account we create a simple struct to represent our request, which will look like this:

Example 1.5 reading\_writing\_json\_4/reading\_writing\_json\_4.go:

1. type helloWorldRequest struct {
2. Name string `json:"name"` 16 }

Again, we are going to use struct field tags as whilst we could let Unmarshal do case- insensitive matching so {"name": "World} would correctly unmarshal into the struct the same as {"Name": "World"}, when we specify a tag we are being explicit about the request form and that is a good thing. In terms of speed and performance it is also about 10% faster, and remember, performance matters.

To access the JSON sent with the request we need to take a look at the http.Request object passed to our handler. The following listing does not show all the methods on the request, just the ones we are going to be immediately dealing with, for full documentation I recommend checking out the documentation at <https://godoc.org/net/http#Request>:

type Requests struct {

...

// Method specifies the HTTP method (GET, POST, PUT, etc.). Method string

// Header contains the request header fields received by the server. The type Header is a link to map[string] []string.

Header Header

// Body is the request's body. Body io.ReadCloser

...

}

The JSON that has been sent with the request is accessible in the Body field. Body implements the interface io.ReadCloser as a stream and does not return a []byte or a string. If we need the data contained in the body, we can simply read it into a byte array, as shown in the following example:

1. body, err := ioutil.ReadAll(r.Body)
2. if err != nil {
3. http.Error(w, "Bad request", http.StatusBadRequest)
4. return 34 }

Here is something we'll need to remember. We are not calling Body.Close(), if we were making a call with a client we would need to do this as it is not automatically closed, however, when used in a ServeHTTP handler, the server automatically closes the request stream.

To see how this all works inside our handler, we can look at the following handler:

28 func helloWorldHandler(w http.ResponseWriter, r \*http.Request) { 29

1. body, err := ioutil.ReadAll(r.Body)
2. if err != nil {
3. http.Error(w, "Bad request", http.StatusBadRequest)
4. return

34 }

35

1. var request helloWorldRequest
2. err = json.Unmarshal(body, &request)
3. if err != nil {
4. http.Error(w, "Bad request", http.StatusBadRequest)
5. return

41 }

42

43 response := helloWorldResponse{Message: "Hello " + request.Name} 44

1. encoder := json.NewEncoder(w)
2. encoder.Encode(response) 47 }

Let's run this example and see how it works. To test, we can simply use the curl command to send a request to the running server. If you feel more comfortable using a GUI tool than Postman (which is available for the Google Chrome browser), they will work just fine or feel free to use your preferred tool:

**$ curl localhost:8080/helloworld -d '{"name":"Nic"}'**

You should see the following response:

{"message":"Hello Nic"}

What do you think will happen if you do not include a body with your request?

**$ curl localhost:8080/helloworld**

If you guessed correctly, that you would get a HTTP status 400 Bad Request, then you win a prize:

func Error(w ResponseWriter, error string, code int)

Errors reply to the request with the given message and status code. Once we have sent this, we need to return stopping further execution of the function as this does not close the ResponseWriter interface and return flow to the calling function automatically.

Just before you think you are done, have a go and see if you can improve the performance of the handler. Think about the things we were talking about when marshaling JSON.

Got it?

Well if not here is the answer, again all we are doing is using the Decoder, which is the opposite of the Encoder that we used in writing JSON. It has an instant 33% performance increase and less code too.

Example 1.6 reading\_writing\_json\_5/reading\_writing\_json\_5.go:

27 func helloWorldHandler(w http.ResponseWriter, r \*http.Request) { 28

1. var request HelloWorldRequest
2. decoder := json.NewDecoder(r.Body) 31
3. err := decoder.Decode(&request)
4. if err != nil {
5. http.Error(w, "Bad request", http.StatusBadRequest)
6. return

36 }

37

38 response := HelloWorldResponse{Message: "Hello " + request.Name} 39

1. encoder := json.NewEncoder(w)
2. encoder.Encode(response) 42 }

Now we can see just how easy it is to encode and decode JSON with Go, I would recommend taking five minutes now to spend some time digging through the documentation for the encoding/json package ([https://golang.org/pkg/encoding/jso](https://golang.org/pkg/encoding/json/) [n/](https://golang.org/pkg/encoding/json/)) as there is a whole lot more that you can do with this.

# Routing in net/http

Even a simple microservice will need the capability to route requests to different handlers dependent on the requested path or method. In Go this is handled by the DefaultServeMux method which is an instance of ServerMux. Earlier in this chapter, we briefly covered that when nil is passed to the handler parameter for the ListenAndServe function then the DefaultServeMux method is used. When we call the http.HandleFunc("/helloworld", helloWorldHandler) package function we are actually just indirectly calling http.DefaultServerMux.HandleFunc(…).

The Go HTTP server does not have a specific router instead any object which implements the http.Handler interface is passed as a top level function to the Listen() function, when a request comes into the server the ServeHTTP method of this handler is called and it is responsible for performing or delegating any work. To facilitate the handling of multiple routes the HTTP package has a special object called ServerMux, which implements the http.Handler interface.

There are two functions to adding handlers to a ServerMux handler:

func HandlerFunc(pattern string, handler func(ResponseWriter, \*Request)) func Handle(pattern string, handler Handler)

The HandleFunc function is a convenience function that creates a handler who's

ServeHTTP method calls an ordinary function with the func(ResponseWriter,

\*Request) signature that you pass as a parameter.

The Handle function requires that you pass two parameters, the pattern that you would like to register the handler and an object that implements the Handler interface:

type Handler interface { ServeHTTP(ResponseWriter, \*Request)

}

## Paths

We already explained how ServeMux is responsible for routing inbound requests to the registered handlers, however the way that the routes are matched can be quite confusing. The ServeMux handler has a very simple routing model it does not support wildcards or regular expressions, with ServeMux you must be explicit about the registered paths.

You can register both fixed rooted paths, such as /images/cat.jpg, or rooted subtrees such as /images/. The trailing slash in the rooted subtree is important as any request that starts with /images/, for example /images/happy\_cat.jpg, would be routed to the handler associated with /images/.

If we register a path /images/ to the handler foo, and the user makes a request to our service at /images (note no trailing slash), then ServerMux will forward the request to the

/images/ handler, appending a trailing slash.

If we also register the path /images (note no trailing slash) to the handler bar and the user requests /images then this request will be directed to bar; however, /images/ or

/images/cat.jpg will be directed to foo:

http.Handle("/images/", newFooHandler()) http.Handle("/images/persian/", newBarHandler()) http.Handle("/images", newBuzzHandler())

/images => Buzz

/images/ => Foo

/images/cat => Foo

/images/cat.jpg => Foo

/images/persian/cat.jpg => Bar

Longer paths will always take precedence over shorter ones so it is possible to have an explicit route that points to a different handler to a catch all route.

We can also specify the hostname, we could register a path such as search.google.com/ and /ServerMux would forward any requests to [http://search.google.com](http://search.google.com/) and [http://www.google.com](http://www.google.com/) to their respective handlers.

If you are used to a framework based application development approach such as using Ruby on Rails or ExpressJS you may find this router incredibly simple and it is, remember that we are not using a framework but the standard packages of Go, the intention is always to provide a basis that can be built upon. In very simple cases the ServeMux approach more than good enough and in fact I personally don't use anything else. Everyone's needs are different however and the beauty and simplicity of the standard packages makes it incredibly simple to build your own route as all is needed is an object which implements the Handler interface. A quick trawl through google will surface some very good third party routers but my recommendation for you is to learn the limitations of ServeMux first before deciding to choose a third-party package it will greatly help with your decision process as you will know the problem you are trying to solve.

## Convenience handlers

The net/http package implements several methods that create different types of convenience handlers, let's examine these.

## FileServer

A FileServer function returns a handler that serves HTTP requests with the contents of the filesystem. This can be used to serve static files such as images or other content that is stored on the file system:

func FileServer(root FileSystem) Handler

Take a look at the following code:

http.Handle("/images", http.FileServer(http.Dir("./images")))

This allows us to map the contents of the file system path ./images to the server route

/images, Dir implements a file system which is restricted to a specific directory tree, the

FileServer method uses this to be able to serve the assets.

## NotFoundHandler

The NotFoundHandler function returns a simple request handler that replies to each request with a 404 page not found reply:

func NotFoundHandler() Handler

## RedirectHandler

The RedirectHandler function returns a request handler that redirects each request it receives to the given URI using the given status code. The provided code should be in the 3xx range and is usually StatusMovedPermanently, StatusFound, or StatusSeeOther:

func RedirectHandler(url string, code int) Handler

## StripPrefix

The StripPrefix function returns a handler that serves HTTP requests by removing the given prefix from the request URL's path and then invoking h handler. If a path does not exist, then StripPrefix will reply with an HTTP 404 not found error:

func StripPrefix(prefix string, h Handler) Handler

## TimeoutHandler

The TimeoutHandler function returns a Handler interface that runs h with the given time limit. When we investigate common patterns in Chapter 6, *Microservice Frameworks*, we will see just how useful this can be for avoiding cascading failures in your service:

func TimeoutHandler(h Handler, dt time.Duration, msg string) Handler

The new handler calls h.ServeHTTP to handle each request, but if a call runs for longer than its time limit, the handler responds with a 503 Service Unavailable response with the given message (msg) in its body.

The last two handlers are especially interesting as they are, in effect, chaining handlers. This is a technique that we will go into more in-depth in a later chapter as it allows you to both practice clean code and also allows you to keep your code DRY.

I may have lifted most of the descriptions for these handlers straight from the Go documentation and you probably have already read these because you have read the documentation right? With Go, the documentation is excellent and writing documentation for your own packages is heavily encouraged, even enforced, if you use the golint command that comes with the standard package then this will report areas of your code which do not conform to the standards. I really recommend spending a little time browsing the standard docs when you are using one of the packages, not only will you learn the correct usage, you may learn that there is a better approach. You will certainly be exposed to good practice and style and you may even be able to keep working on the sad day that Stack Overflow stops working and the entire industry grinds to a halt.

## Static file handler

Whilst we are mostly going to be dealing with APIs in this book, it is a useful illustration to see how the default router and paths work by adding a secondary endpoint.

As a little exercise, try to modify the code in reading\_writing\_json\_5/reading\_writing\_json\_5.go to add an endpoint /cat, which returns the cat picture specified in the URI. To give you a little hint, you are going to need to use the FileServer function on the net/http package and your URI will look something like http://localhost:8080/cat/cat.jpg.

Did it work the first time or did you forget to add the StripPrefix handler?

Example 1.7 reading\_writing\_json\_6/reading\_writing\_json\_6.go:

1. cathandler := http.FileServer(http.Dir("./images"))
2. http.Handle("/cat/", http.StripPrefix("/cat/", cathandler))

In the preceding example, we are registering a StripPrefix handler with our path /cat/. If we did not do this, then the FileServer handler would be looking for our image in the images/cat directory. It is also worth reminding ourselves about the difference with /cat and /cat/ as paths. If we registered our path as /cat then we would not match

/cat/cat.jpg. If we register our path as /cat/, we will match both /cat and

/cat/whatever.

## Creating handlers

We will now finish off our examples here by showing how you can create a Handler rather than just using HandleFunc. We are going to split the code that performs the request validation for our helloworld endpoint and the code that returns the response out into separate handlers to illustrate how it is possible to chain handlers.

Example 1.8 chapter1/reading\_writing\_json\_7.go:

1. type validationHandler struct {
2. next http.Handler 33 }

34

1. func newValidationHandler(next http.Handler) http.Handler {
2. return validationHandler{next: next} 37 }

The first thing we need to do when creating our own Handler is to define a struct field that will implement the methods in the Handlers interface. Since in this example, we are going to be chaining handlers together, the first handler, which is our validation handler, needs to have a reference to the next in the chain as it has the responsibility for calling ServeHTTP or returning a response.

For convenience, we have added a function that returns us a new handler; however, we could have just set the next field. This method, however, is better form as it makes our code a little easier to read and when we need to pass complex dependencies to the handler using a function to create, it keeps things a little neater:

1. func (h validationHandler) ServeHTTP(rw http.ResponseWriter, r

\*http.Request) {

1. var request helloWorldRequest

|  |  |  |
| --- | --- | --- |
| 39 |  | decoder := json.NewDecoder(r.Body) |
| 40 |  |  |
| 41 |  | err := decoder.Decode(&request) |
| 42 |  | if err != nil { |
| 43 |  | http.Error(rw, "Bad request", http.StatusBadRequest) |
| 44 |  | return |
| 45 |  | } |
| 46 |  |  |
| 47 |  | h.next.ServeHTTP(rw, r) |
| 48 | } |  |

The previous code block illustrates how we would implement the ServeHTTP method. The only interesting thing to note here is the statement that begins at line **44**. If an error is returned from decoding the request, we write a 500 error to the response, the handler chain would stop here. Only when no error is returned do we call the next handler in the chain and we do this simply by invoking its ServeHTTP method. To pass the name decoded from the request, we are simply setting a variable:

53 type helloWorldHandler struct{} 54

1. func newHelloWorldHandler() http.Handler {
2. return helloWorldHandler{} 57 }

58

1. func (h helloWorldHandler) ServeHTTP(rw http.ResponseWriter, r

\*http.Request) {

1. response := helloWorldResponse{Message: "Hello " + name} 61
2. encoder := json.NewEncoder(rw)
3. encoder.Encode(response) 64 }

The helloWorldHandler type that writes the response does not look too different from when we were using a simple function. If you compare this to *example 1.6*, you will see that all we really have done is remove the request decoding.

Now the first thing I want to mention about this code is that it is purely to illustrate how you can do something, not that you should do something. In this simple case, splitting the request validation and response sending into two handlers adds a lot of needless complexity and it is not really making our code DRYer. The technique, however, is useful. When we examine authentication in a later chapter, you will see this pattern as it allows us to centralize our authentication logic and share it among handlers.

# Context

The problem with the previous pattern is that there is no way that you can pass the validated request from one handler to the next without breaking the http.Handler interface, but guess what Go has us covered. The context package was listed as experimental for several years before finally making it in to the standard package with Go

1.7. The Context type implements a safe method for accessing request-scoped data that is

safe to use simultaneously by multiple Go routines. Let’s take a quick look at this package and then update our example to see it in use.

## Background

The Background method returns an empty context that has no values; it is typically used by the main function and as the top-level Context:

func Background() Context

## WithCancel

The WithCancel method returns a copy of the parent context with a cancel function, calling the cancel function releases resources associated with the context and should be called as soon as operations running in the Context type are complete:

func WithCancel(parent Context) (ctx Context, cancel CancelFunc)

## WithDeadline

The WithDeadline method returns a copy of the parent context that expires after the current time is greater than deadline. At this point, the context's Done channel is closed and the resources associated are released. It also passes back a CancelFunc method that allows manual cancellation of the context:

func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)

## WithTimeout

The WithTimeout method is similar to WithDeadline except you pass it a duration for which the Context type should exist. Once this duration has elapsed, the Done channel is closed and the resources associated with the context are released:

func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)

## WithValue

The WithValue method returns a copy of the parent Context in which the val value is associated with the key. The Context values are perfect to be used for request-scoped data:

func WithValue(parent Context, key interface{}, val interface{}) Context

Why not attempt to modify *example 1.7* to implement a request scoped context. The key could be in the previous sentence; every request needs its own context.

## Using contexts

You probably found that rather painful, especially if you come from a background in a framework such as Rails or Spring. Writing this kind of code is not really something you want to be spending your time on, building application features is far more important. One thing to note however is that neither Ruby or Java have anything more advanced in their base packages. Thankfully for us, over the seven years that Go has been in existence, many excellent people have done just that, and when looking at frameworks in Chapter 3, *Introducing Docker*, we will find that all of this complexity has been taken care of by some awesome open source authors.

In addition to the adoption of context into the main Go release version 1.7 implements an important update on the http.Request structure, we have the following additions:

func (r \*Request) Context() context.Context

The Context() method gives us access to a context.Context structure which is always non nil as it is populated when the request is originally created. For inbound requests the http.Server manages the lifecycle of the context automatically cancelling it when the client connection closes. For outbound requests, Context controls cancellation, by this we mean that if we cancel the Context() method we can cancel the outgoing request. This concept is illustrated in the following example:

1. func fetchGoogle(t \*testing.T) {
2. r, \_ := http.NewRequest("GET", "https://google.com", nil) 72

73 timeoutRequest, cancelFunc := context.WithTimeout(r.Context(), 1\*time.Millisecond)

|  |  |  |
| --- | --- | --- |
| 74 |  | defer cancelFunc() |
| 75 |  |  |
| 76 |  | r = r.WithContext(timeoutRequest) |
| 77 |  |  |
| 78 |  | \_, err := http.DefaultClient.Do(r) |
| 79 |  | if err != nil { |
| 80 |  | fmt.Println("Error:", err) |
| 81 |  | } |
| 82 | } |  |

In line **74**, we are creating a timeout context from the original in the request, and unlike an inbound request where the context is automatically cancelled for you we must manually perform this step in an outbound request.

Line **77** implements the second of the two new context methods which have been added to the http.Request object:

func (r \*Request) WithContext(ctx context.Context) \*Request

The WithContext object returns a shallow copy of the original request which has the context changed to the given ctx context.

When we execute this function we will find that after 1 millisecond the request will complete with an error:

Error: Get https://google.com: context deadline exceeded

The context is timing out before the request has a change to complete and the do method immediately returns. This is an excellent technique to use for outbound connections and thanks to the changes in Go 1.7 is now incredibly easy to implement.

What about our inbound connection Let’s see how we can update our previous example. Example 1.9 updates our example to show how we can leverage the context package to implement Go routine safe access to objects. The full example can be found in reading\_writing\_json\_8/reading\_writing\_json\_8.go but all of the modification we need to make are in the two ServeHTTP methods for our handlers:

1. func (h validationHandler) ServeHTTP(rw http.ResponseWriter, r

\*http.Request) {

1. var request helloWorldRequest
2. decoder := json.NewDecoder(r.Body) 44
3. err := decoder.Decode(&request)
4. if err != nil {
5. http.Error(rw, "Bad request", http.StatusBadRequest)
6. return 49 }

50

1. c := context.WithValue(r.Context(), validationContextKey("name"), request.Name)
2. r = r.WithContext(c) 53

54 h.next.ServeHTTP(rw, r) 55 }

If we take a quick look at our validationHandler you will see that when we have a valid request, we are creating a new context for this request and then setting the value of the Name field in the request into the context. You might also wonder what is going on with line

**51**. When you add an item to a context such as with the WithValue call, the method returns a copy of the previous context, to save a little time and add a little confusion, we are holding a pointer to the context, so in order to pass this as a copy to WithValue, we must dereference it. To update our pointer, we must also set the returned value to the value referenced by the pointer hence again we need to dereference it. The other think we need to look at with this method call is the key, we are using validationContextKey this is an explicitly declared type of string:

13 type validationContextKey string

The reason we are not just using a simple string is that context often flows across packages and if we just used string then we could end up with a key clash where one package within your control is writing a name key and another package which is outside of your control is also using the context and writing a key called name, in this instance the second package would inadvertently overwrite your context value. By declaring a package level type validationContextKey and using this we can ensure that we avoid these collisions:

1. func (h helloWorldHandler) ServeHTTP(rw http.ResponseWriter, r

\*http.Request) {

1. name := r.Context().Value(validationContextKey("name")).(string)
2. response := helloWorldResponse{Message: "Hello " + name} 67
3. encoder := json.NewEncoder(rw)
4. encoder.Encode(response) 70 }

To retrieve the value, all we have to do is obtain the context and then call the Value method casting it into a string.

# RPC in the Go standard library

As expected, the Go standard library has fantastic support for RPC right out-of-the-box. Let's look at a few examples of how we can use this.

## Simple RPC example

In this simple example, we will see how we can use the standard RPC package to create a client and server that use a shared interface to communicate over RPC. We will follow the typical Hello World example that we ran through when learning the net/http package and see just how easy it is to build an RPC-based API in go:

rpc/server/server.go:

34 type HelloWorldHandler struct{} 35

1. func (h \*HelloWorldHandler) HelloWorld(args \*contract.HelloWorldRequest, reply \*contract.HelloWorldResponse) error {
2. reply.Message = ""Hello "" + args.Name
3. return nil 39 }

Like our example on creating REST APIs using the standard library for RPC, we will also define a handler. The difference between this handler and http.Handler is that it does not need to conform to an interface; as long as we have a struct field with methods on it we can register this with the RPC server:

func Register(rcvr interface{}) error

The Register function, which is in the rpc package, publishes the methods that are part of the given interface to the default server and allows them to be called by clients connecting to the service. The name of the method uses the name of the concrete type, so in our instance if my client wanted to call the HelloWorld method, we would access it using HelloWorldHandler.HelloWorld. If we do not wish to use the concrete types name, we can register it with a different name using the RegisterName function, which uses the provided name instead:

func RegisterName(name string, rcvr interface{}) error

This would enable me to keep the name of the struct field to whatever is meaningful to my code; however, for my client contract I might decide to use something different such as Greet:

1. func StartServer() {
2. helloWorld := &HelloWorldHandler{}
3. rpc.Register(helloWorld) 22
4. l, err := net.Listen("("tcp",", fmt.Sprintf(":%(":%v",", port))
5. if err != nil {
6. log.Fatal(fmt.Sprintf("("Unable to listen on given port: %s",", err))

|  |  |  |
| --- | --- | --- |
| 26 |  | } |
| 27 |  |  |
| 28 |  | for { |
| 29 |  | conn, \_ := l.Accept() |
| 30 |  | go rpc.ServeConn(conn) |
| 31 |  | } |
| 32 | } |  |

In the StartServer function, we first create a new instance of our handler and then we register this with the default RPC server.

Unlike the convenience of net/http where we can just create a server with ListenAndServe, when we are using RPC we need to do a little more manual work. In line **23**, we are creating a socket using the given protocol and binding it to the IP address and port. This gives us the capability to specifically select the protocol we would like to use for the server, tcp, tcp4, tcp6, unix, or unixpacket:

func Listen(net, laddr string) (Listener, error)

The Listen() function returns an instance that implements the Listener interface:

type Listener interface {

// Accept waits for and returns the next connection to the listener. Accept() (Conn, error)

// Close closes the listener.

// Any blocked Accept operations will be unblocked and return errors. Close() error

// Addr returns the listener's network address. Addr() Addr

}

To receive connections, we must call the Accept method on the listener. If you look at line **29**, you will see that we have an endless for loop, this is because unlike ListenAndServe which blocks for all connections, with an RPC server we handle each connection individually and as soon as we deal with the first connection we need to continue to again call Accept to handle subsequent connections or the application would exit. Accept is a blocking method so if there are no clients currently attempting to connect to the service then Accept will block until one does. Once we receive a connection then we need to call the Accept method again to process the next connection. If you look at line **30** in our example code, you will see we are calling the ServeConn method:

func ServeConn(conn io.ReadWriteCloser)

The ServeConn method runs the DefaultServer method on the given connection, and will block until the client completes. In our example, we are using the go statement before running the server so that we can immediately process the next waiting connection without blocking for the first client to close its connection.

In terms of communication protocol, ServeConn uses the gob wire format <https://golang.org/pkg/encoding/gob/>, we will see when we look at JSON-RPC how we can use a different encoding.

The gob format was specifically designed to facilitate Go to Go-based communication and was designed around the idea of something easier to use and possibly more efficient than the likes of protocol buffers, this comes at a cost of cross language communication.

With gobs, the source and destination values and types do not need to correspond exactly, when you send struct, if a field is in the source but not in the receiving struct, then the decoder will ignore this field and the processing will continue without error. If a field is present in the destination that is not in the source, then again the decoder will ignore this field and will successfully process the rest of the message. Whilst this seems like a minor benefit, it is a huge advancement over the RPC messages of old such as JMI where the exact same interface must be present on both the client and server. This level of inflexibility with JMI introduced tight coupling between the two code bases and caused no end of complexity when it was required to deploy an update to our application.

To make a request to our client we can no longer simply use curl as we are no longer are using the HTTP protocol and the message format is no longer JSON. If we look at the example in rpc/client/client.go we can see how to implement a connecting client:

1. func CreateClient() \*rpc.Client {
2. client, err := rpc.Dial("tcp", fmt.Sprintf("localhost:%v", port))
3. if err != nil {
4. log.Fatal("dialing:", err) 17 }

18

19 return client 20 }

The previous block shows how we need to setup rpc.Client, the first thing we need to do on line **14** is to create the client itself using the Dial() function in the rpc package:

func Dial(network, address string) (\*Client, error)

We then use this returned connection to make a request to the server:

1. func PerformRequest(client \*rpc.Client) contract.HelloWorldResponse {
2. args := &contract.HelloWorldRequest{Name: "World"}
3. var reply contract.HelloWorldResponse 25
4. err := client.Call("HelloWorldHandler.HelloWorld", args, &reply)
5. if err != nil {

|  |  |  |
| --- | --- | --- |
| 28 |  | log.Fatal("error:", err) |
| 29 |  | } |
| 30 |  |  |
| 31 |  | return reply |
| 32 | } |  |

In line **26**, we are using the Call() method on the client to invoke the named function on the server:

func (client \*Client) Call(serviceMethod string, args interface{}, reply interface{}) error

Call is a blocking function which waits until the server sends a reply writing the response assuming there is no error to the reference of our HelloWorldResponse passed to the method and if an error occurs when processing the request this is returned and can be handled accordingly.

## RPC over HTTP

In the instance that you need to use HTTP as your transport protocol then the rpc package can facilitate this by calling the HandleHTTP method.

The HandleHTTP method sets up two endpoints in your application:

const (

// Defaults used by HandleHTTP DefaultRPCPath = "/\_"/\_goRPC\_"\_" DefaultDebugPath = "/"/debug/rpc""

)

If you point your browser at the DefaultDebugPath you can see details for the registered endpoints, there are two things to note:

This does not mean you can communicate easily with your API from a web browser. The messages are still gob encoded so you would need to write a gob encoder and decoder in JavaScript, which I am not actually sure is possible. It was certainly never the intent of the package to support this capability and therefore I would not advise this action, a JSON or JSON-RPC based message is much better suited to this use case.

The debug endpoint is not going to provide you with auto-generated documentation for your API. The output is fairly basic and the intention seems to be so you can track the number of calls made to an endpoint.

All that said there may be a reason why you need to use HTTP, possibly your network does not allow any other protocol or potentially you have a load balancer that is not capable of dealing with pure TCP connections. We can also take advantage of HTTP headers and other metadata which is not available using a pure TCP request.

rpc\_http/server/server.go

1. func StartServer() {
2. helloWorld := &HelloWorldHandler{}
3. rpc.Register(helloWorld)
4. rpc.HandleHTTP() 26
5. l, err := net.Listen("("tcp",", fmt.Sprintf(":%(":%v",", port))
6. if err != nil {
7. log.Fatal(fmt.Sprintf("("Unable to listen on given port: %s",", err))

|  |  |  |
| --- | --- | --- |
| 30 |  | } |
| 31 |  |  |
| 32 |  | log.Printf("("Server starting on port %v\n",", port) |
| 33 |  |  |
| 34 |  | http.Serve(l, nil) |
| 35 | } |  |

If we look at line **25**, in the preceding example, we can see we are calling the rpc.HandleHTTP method, this is a requirement using HTTP with RPC as it will register the HTTP handlers we mentioned earlier with the DefaultServer method. We then call the http.Serve method and pass it the listener we are creating in line **27**, we are setting the second parameter to be nil as we wish to use the DefaultServer method. This is exactly the same method that we looked at in the previous examples when we were looking at RESTful endpoints.

## JSON-RPC over HTTP

In this last example, we will look at the net/rpc/jsonrpc package that provides a built-in codec for serializing and deserializing to the JSON-RPC standard. We will also look at how we can send these responses over HTTP, whilst you may ask why not just use REST, and to some extent I will agree with you, it is an interesting example to be able to see how we can extend the standard framework.

The StartServer method contains nothing we have not seen before it is the standard rpc server setup, the main difference is line **42** where instead of starting the RPC server we are starting an http server and passing the listener to it along with a handler:

rpc\_http\_json/server/server.go

1. func StartServer() {
2. helloWorld := new(HelloWorldHandler)
3. rpc.Register(helloWorld) 36
4. l, err := net.Listen("("tcp",", fmt.Sprintf(":%(":%v",", port))
5. if err != nil {
6. log.Fatal(fmt.Sprintf("("Unable to listen on given port: %s",", err)) 40 }

41

42 http.Serve(l, http.HandlerFunc(httpHandler)) 43 }

The handler we are passing to the server is where the magic happens:

1. func httpHandler(w http.ResponseWriter, r \*http.Request) {
2. serverCodec := jsonrpc.NewServerCodec(&HttpConn{in: r.Body, out: w})
3. err := rpc.ServeRequest(serverCodec)
4. if err != nil {
5. log.Printf("("Error while serving JSON request: %v",", err)
6. http.Error(w, ""Error while serving JSON request, details have been logged.",.", 500)
7. return

52 }

53 }

In line **46**, we are calling the jsonrpc.NewServerCodec function and passing to it a type that implements io.ReadWriteCloser. The NewServerCodec method returns a type that implements rpc.ClientCodec, which has the following methods:

type ClientCodec interface {

// WriteRequest must be safe for concurrent use by multiple goroutines. WriteRequest(\*Request, interface{}) error

ReadResponseHeader(\*Response) error ReadResponseBody(interface{}) error

Close() error

}

A ClientCodec type implements the writing of RPC request and reading RPC responses. To write a request to the connection a client calls the WriteRequest method. To read the response, the client must call ReadResponseHeader and ReadResponseBody as a pair. Once the body has been read, it is the client's responsibility to call the Close method to close the connection. If a nil interface is passed to ReadResponseBody then the body of the response should be read and then discarded:

1. type HttpConn struct {
2. in io.Reader
3. out io.Writer 20 }

21

1. func (c \*HttpConn) Read(p []byte) (n int, err error) { return c.in.Read(p) }
2. func (c \*HttpConn) Write(d []byte) (n int, err error) { return c.out.Write(d) }
3. func (c \*HttpConn) Close() error { return nil }

The NewServerCodec method requires that we pass it a type that implements the ReadWriteCloser interface. As we do not have such a type passed to us as parameters in the httpHandler method we have defined our own type, HttpConn, which encapsulates the http.Request body, which implements io.Reader, and the ResponseWriter method, that implements io.Writer. We can then write our own methods that proxy the calls to the reader and writer creating a type that has the correct interface.

And that is it for our short intro to RPC with the standard libraries; we will see when we look at some frameworks more in depth in Chapter 3, *Introducing Docker*, how these can be used to build a production microservice.

# Summary

That's it for this chapter, we have just written our first microservice in Go and only using the standard library, you should now have an appreciation of just how powerful the standard library is providing us with many of the features we need to write RESTful and RPC-based microservices. We have also looked at encoding and decoding data using the encoding/json package and how we can create light weight messaging by using gobs.

As you progress through this book, you will see how the many, wonderful open source packages build on these foundations to make Go such a fantastic language for microservice development, and by the end of the book you will have all the knowledge required for successfully building microservices in Go.