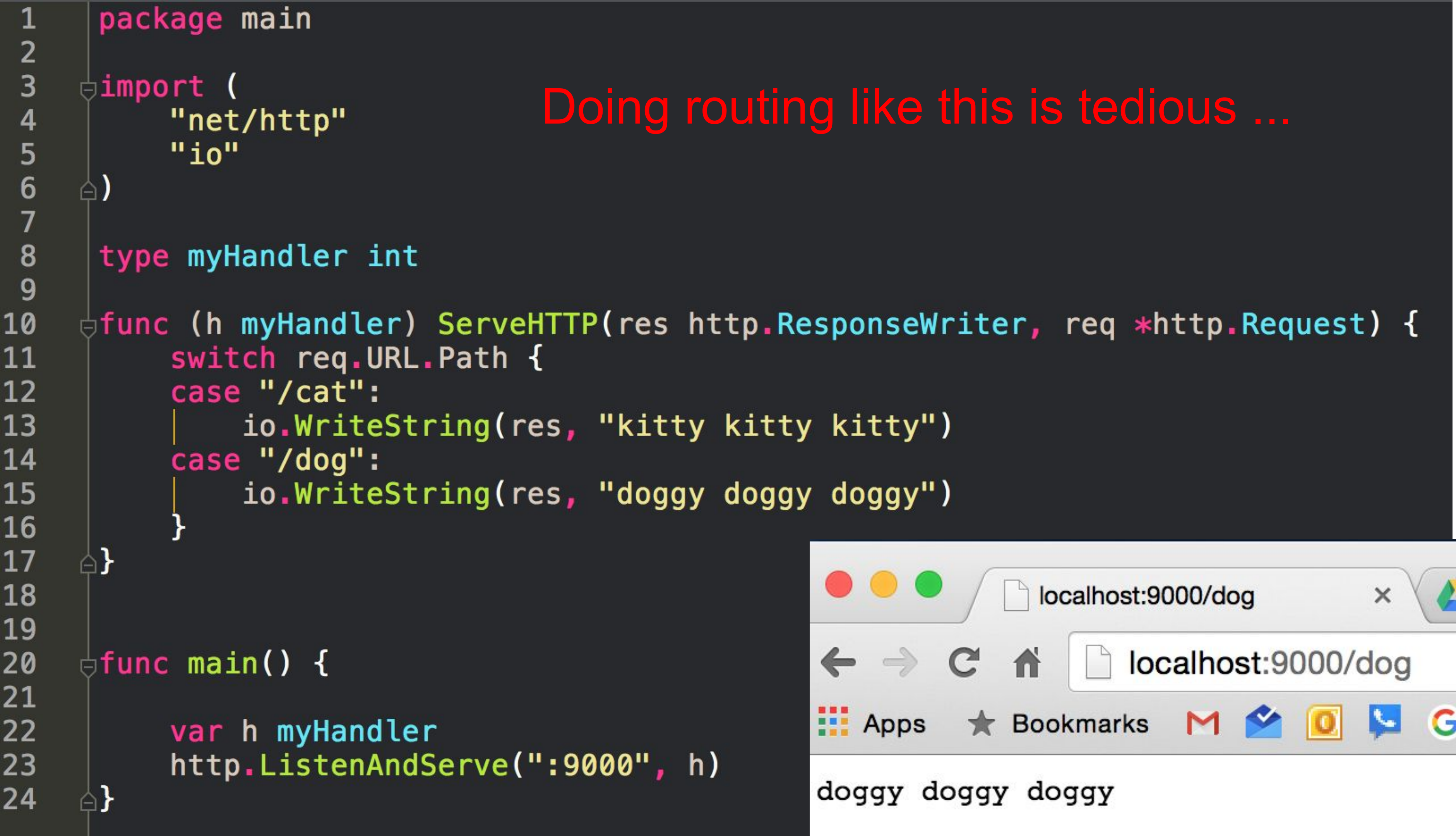
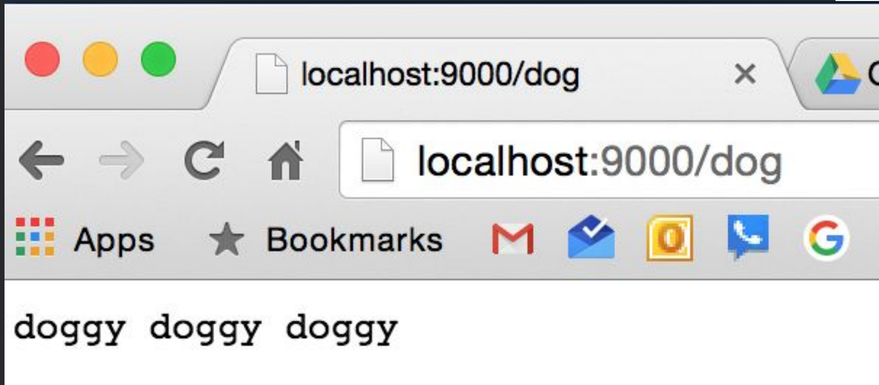
Routing & Restful

Doing routing like this is tedious ...



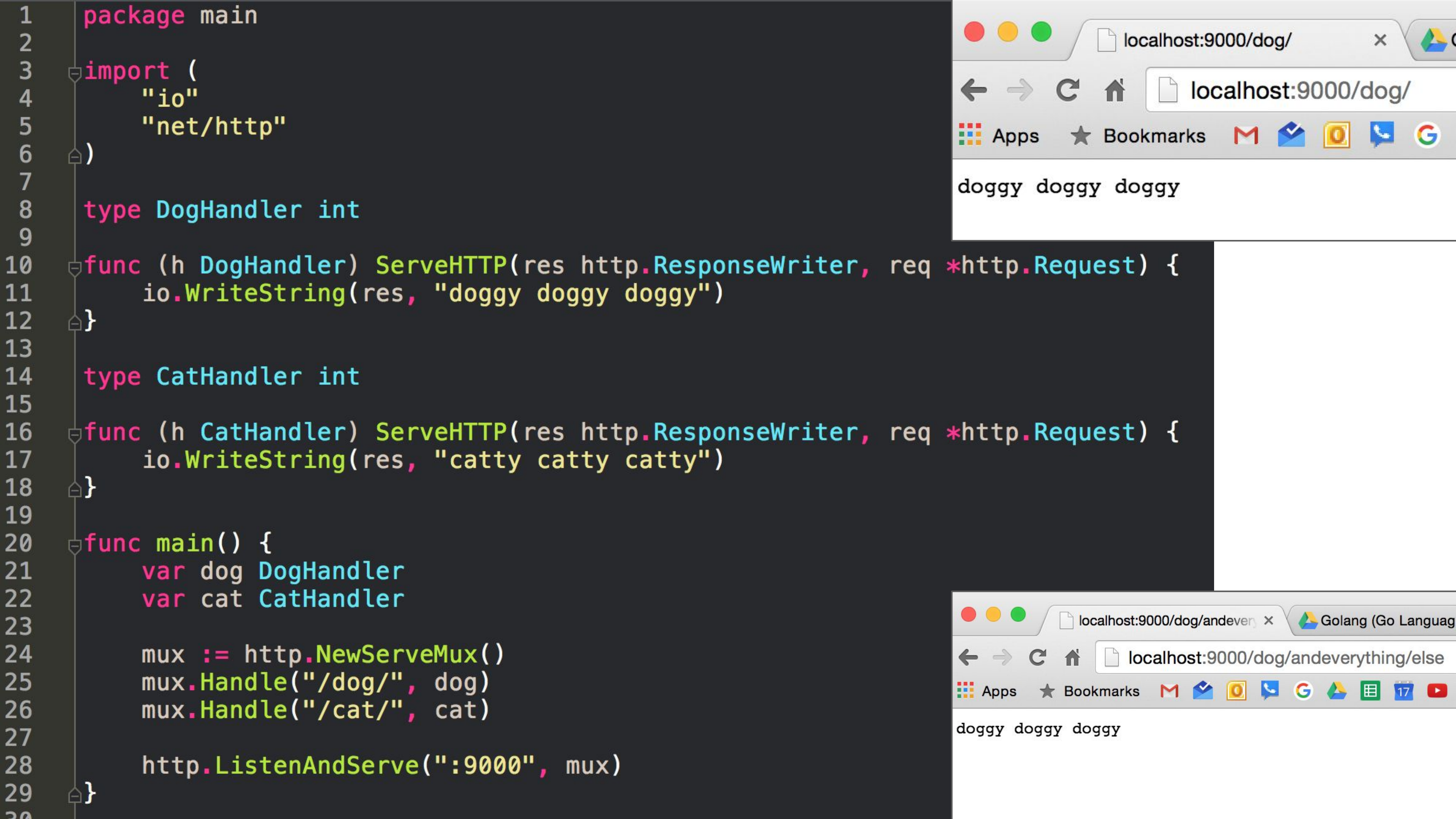
```go
package main

import (
	"net/http"
	"io"
)

type myHandler int

func (h myHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	switch req.URL.Path {
	case "/cat":
		io.WriteString(res, "kitty kitty kitty")
	case "/dog":
		io.WriteString(res, "doggy doggy doggy")
	}
}


func main() {

	var h myHandler
	http.ListenAndServe(":9000", h)
}
```



```
doggy doggy doggy
```

servemux

HTTP request **mu**ltiple**x**er

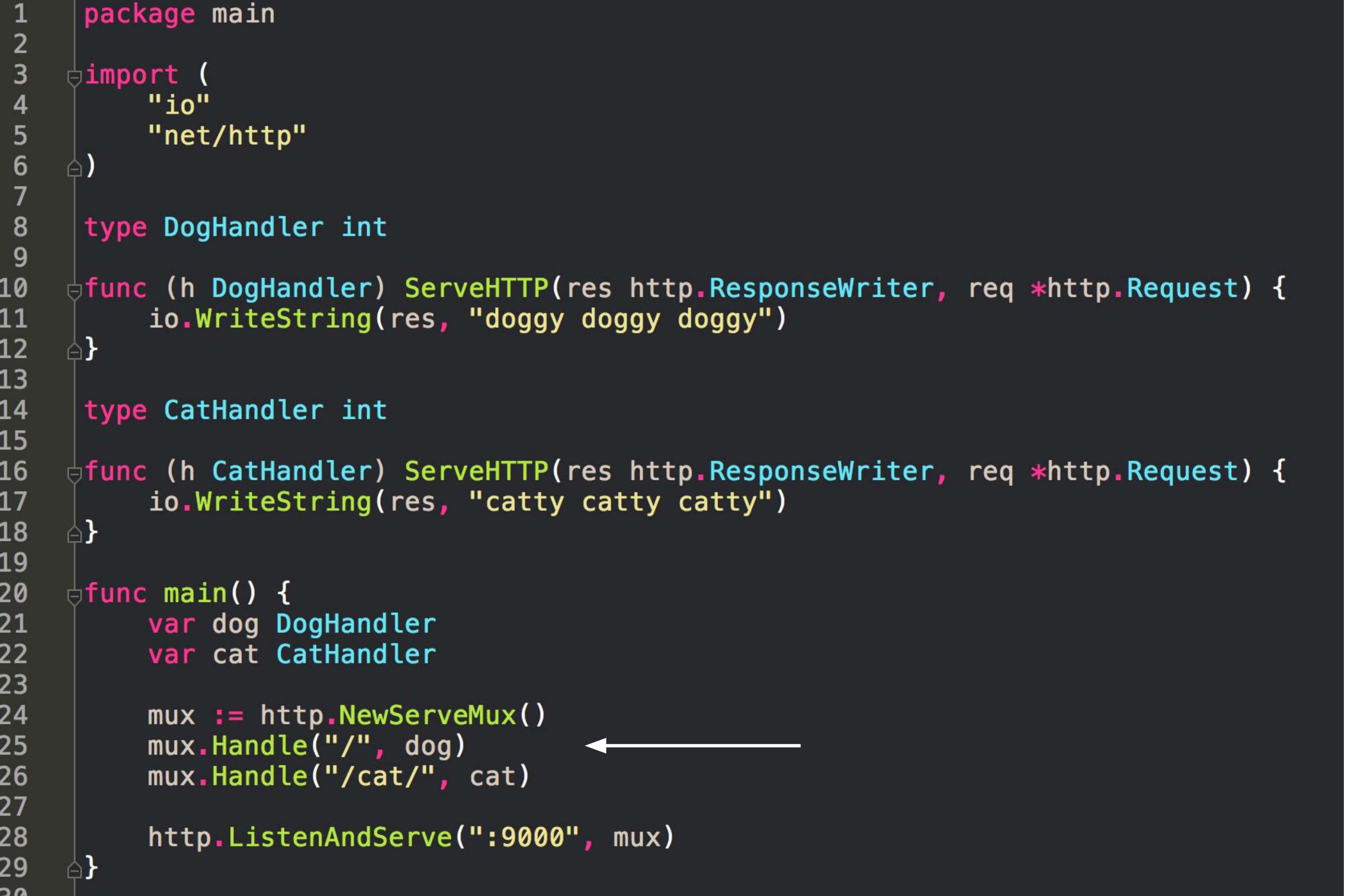


```go
package main

import (
	"io"
	"net/http"
)

type DogHandler int

func (h DogHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "doggy doggy doggy")
}

type CatHandler int

func (h CatHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "catty catty catty")
}

func main() {
	var dog DogHandler
	var cat CatHandler

	mux := http.NewServeMux()
	mux.Handle("/dog/", dog)
	mux.Handle("/cat/", cat)

	http.ListenAndServe(":9000", mux)
}
```

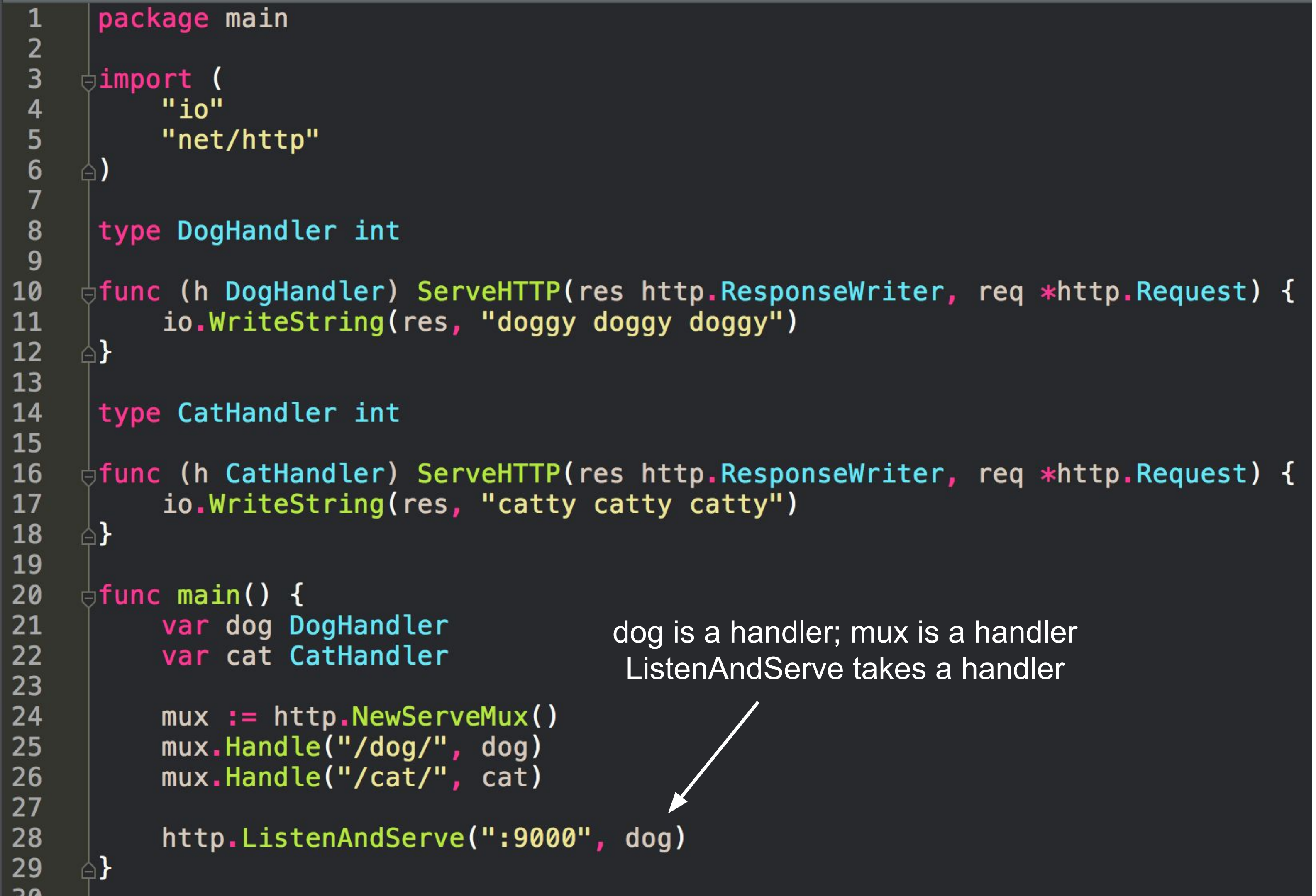
```
doggy doggy doggy
```

localhost:9000/dog/andever
Golang (Go Languag
localhost:9000/dog/andeverything/else
Apps
Bookmarks

```
doggy doggy doggy
```



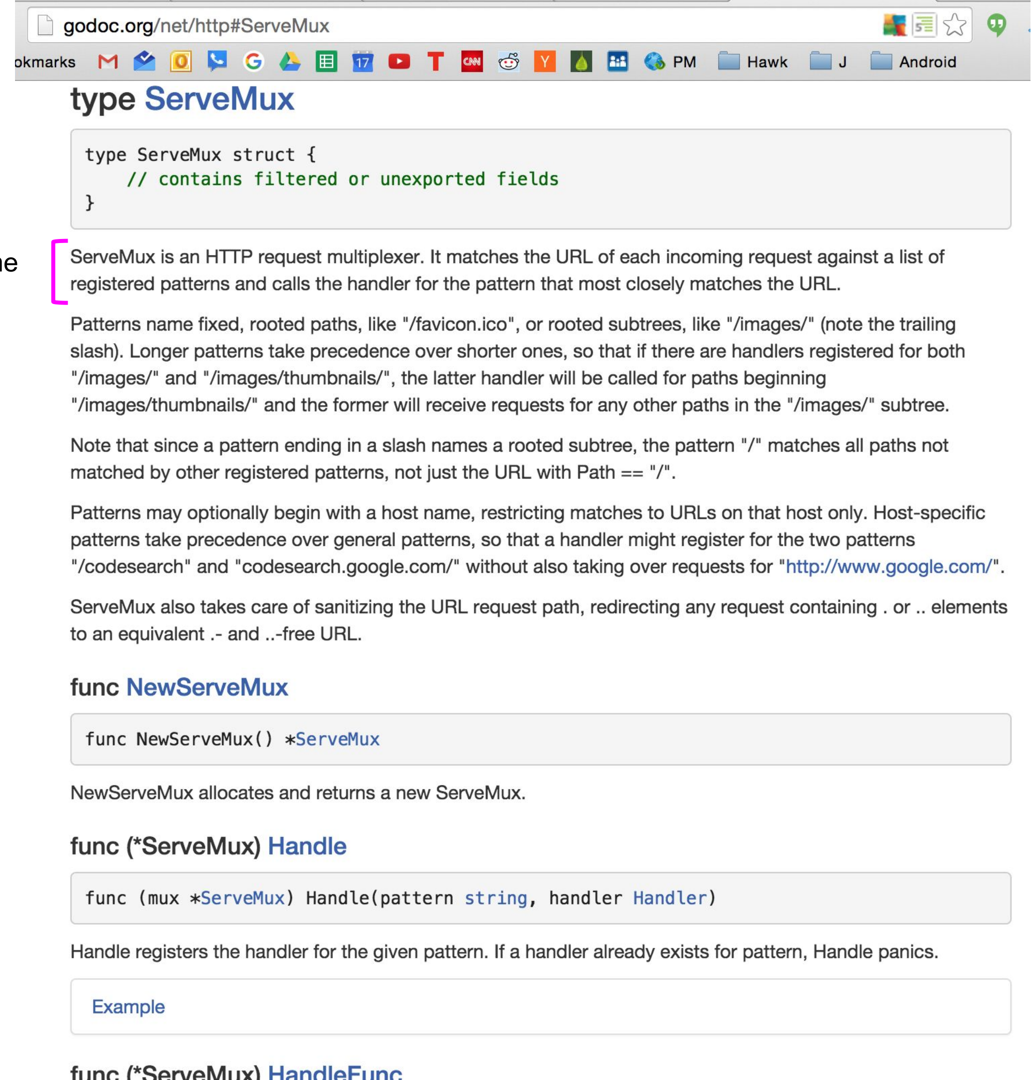
```go
package main

import (
    "io"
    "net/http"
)

type DogHandler int

func (h DogHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
    io.WriteString(res, "doggy doggy doggy")
}

type CatHandler int

func (h CatHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
    io.WriteString(res, "catty catty catty")
}

func main() {
    var dog DogHandler
    var cat CatHandler

    mux := http.NewServeMux()
    mux.Handle("/", dog)
    mux.Handle("/cat/", cat)

    http.ListenAndServe(":9000", mux)
}
```



```go
package main

import (
	"io"
	"net/http"
)

type DogHandler int

func (h DogHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "doggy doggy doggy")
}

type CatHandler int

func (h CatHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "catty catty catty")
}

func main() {
	var dog DogHandler
	var cat CatHandler

	mux := http.NewServeMux()
	mux.Handle("/dog/", dog)
	mux.Handle("/cat/", cat)

	http.ListenAndServe(":9000", dog)
}
```

dog is a handler; mux is a handler
ListenAndServe takes a handler

Routing a URL path to some chunk of code



godoc.org/net/http#ServeMux

type ServeMux

```
type ServeMux struct {
    // contains filtered or unexported fields
}
```

ServeMux is an HTTP request multiplexer. It matches the URL of each incoming request against a list of registered patterns and calls the handler for the pattern that most closely matches the URL.

Patterns name fixed, rooted paths, like "/favicon.ico", or rooted subtrees, like "/images/" (note the trailing slash). Longer patterns take precedence over shorter ones, so that if there are handlers registered for both "/images/" and "/images/thumbnails/", the latter handler will be called for paths beginning "/images/thumbnails/" and the former will receive requests for any other paths in the "/images/" subtree.

Note that since a pattern ending in a slash names a rooted subtree, the pattern "/" matches all paths not matched by other registered patterns, not just the URL with Path == "/".

Patterns may optionally begin with a host name, restricting matches to URLs on that host only. Host-specific patterns take precedence over general patterns, so that a handler might register for the two patterns "/codesearch" and "codesearch.google.com/" without also taking over requests for "http://www.google.com/".

ServeMux also takes care of sanitizing the URL request path, redirecting any request containing . or .. elements to an equivalent .- and ..-free URL.

func NewServeMux

```
func NewServeMux() *ServeMux
```

NewServeMux allocates and returns a new ServeMux.

func (*ServeMux) Handle

```
func (mux *ServeMux) Handle(pattern string, handler Handler)
```

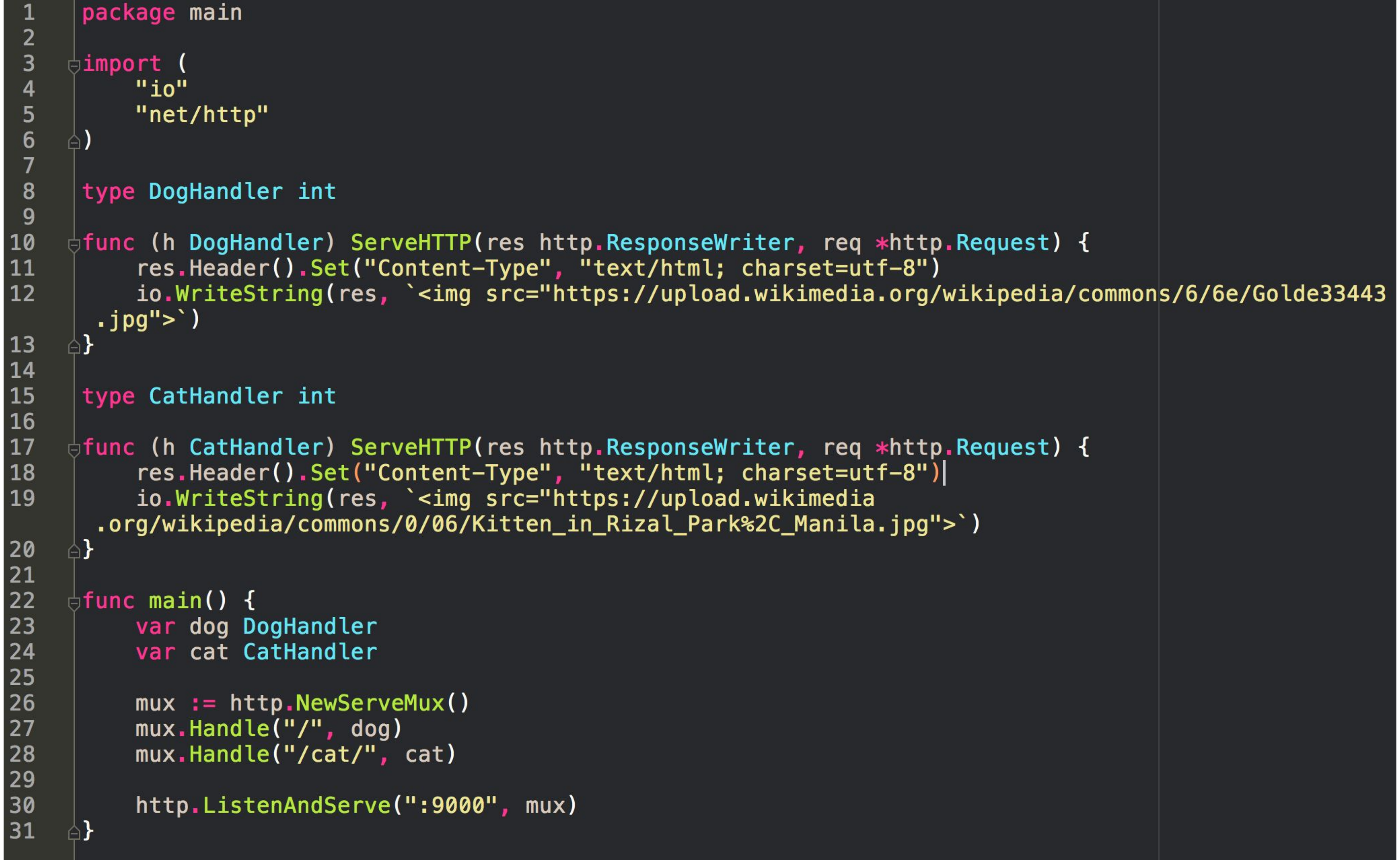
Handle registers the handler for the given pattern. If a handler already exists for pattern, Handle panics.

Example

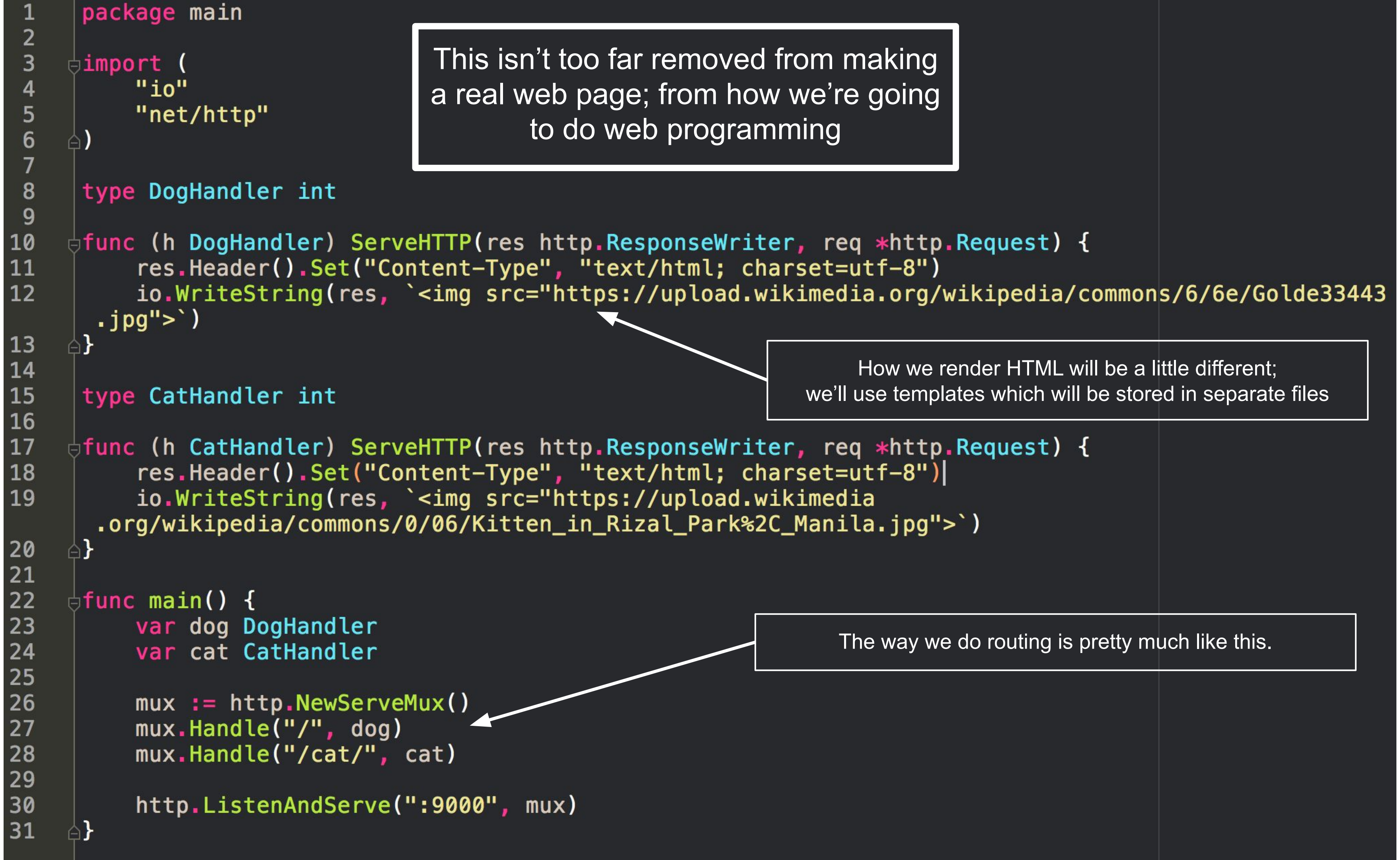
func (*ServeMux) HandleFunc

exercise

Create an http server which returns an html page with a picture of a cat for ‘/cat’ and a picture of a dog for ‘/dog’ using a ServeMux



```go
package main

import (
	"io"
	"net/http"
)

type DogHandler int

func (h DogHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	res.Header().Set("Content-Type", "text/html; charset=utf-8")
	io.WriteString(res, `<img src="https://upload.wikimedia.org/wikipedia/commons/6/6e/Golde33443.jpg">`)
}

type CatHandler int

func (h CatHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	res.Header().Set("Content-Type", "text/html; charset=utf-8")
	io.WriteString(res, `<img src="https://upload.wikimedia.org/wikipedia/commons/0/06/Kitten_in_Rizal_Park%2C_Manila.jpg">`)
}

func main() {
	var dog DogHandler
	var cat CatHandler

	mux := http.NewServeMux()
	mux.Handle("/", dog)
	mux.Handle("/cat/", cat)

	http.ListenAndServe(":9000", mux)
}
```



```go
package main

import (
    "io"
    "net/http"
)

type DogHandler int

func (h DogHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
    res.Header().Set("Content-Type", "text/html; charset=utf-8")
    io.WriteString(res, `<img src="https://upload.wikimedia.org/wikipedia/commons/6/6e/Golde33443.jpg">`)
}

type CatHandler int

func (h CatHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
    res.Header().Set("Content-Type", "text/html; charset=utf-8")
    io.WriteString(res, `<img src="https://upload.wikimedia.org/wikipedia/commons/0/06/Kitten_in_Rizal_Park%2C_Manila.jpg">`)
}

func main() {
    var dog DogHandler
    var cat CatHandler

    mux := http.NewServeMux()
    mux.Handle("/", dog)
    mux.Handle("/cat/", cat)

    http.ListenAndServe(":9000", mux)
}
```

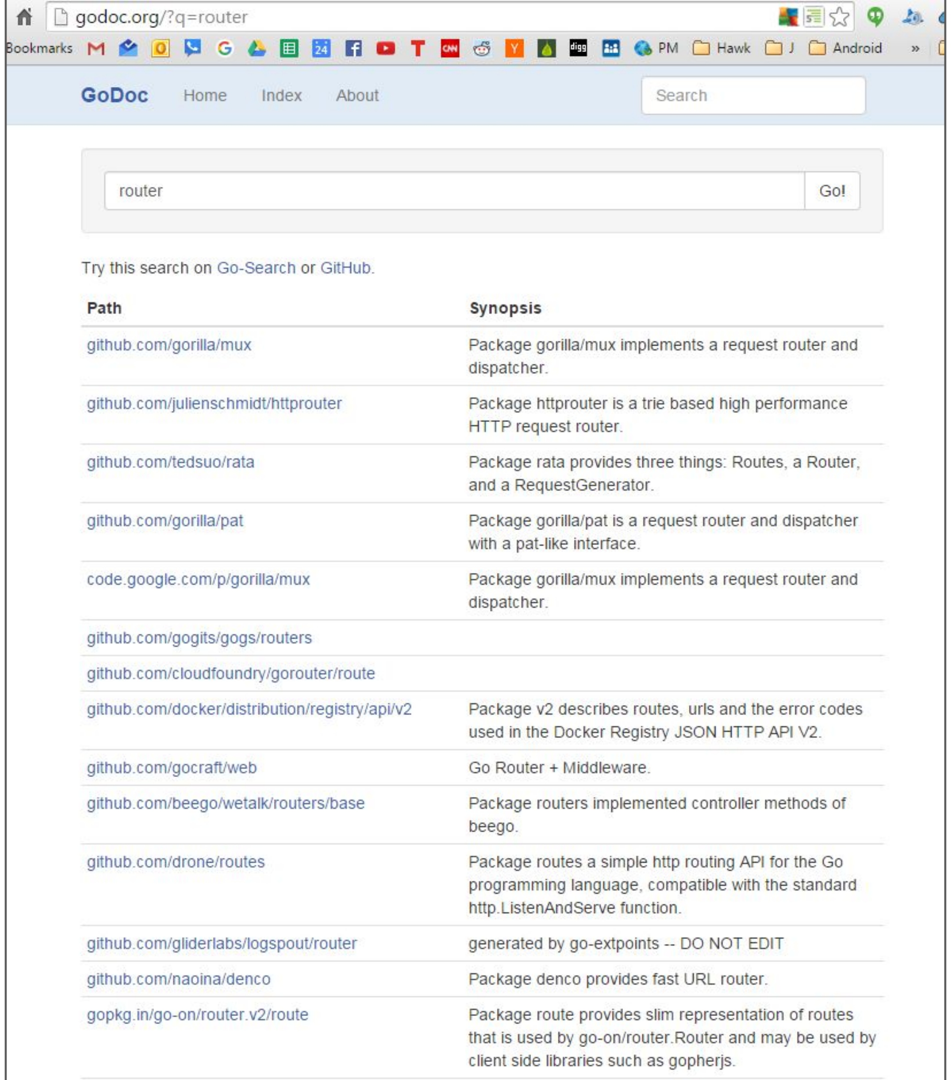
This isn't too far removed from making a real web page; from how we're going to do web programming

How we render HTML will be a little different; we'll use templates which will be stored in separate files

The way we do routing is pretty much like this.

other routers

good to mention
in addition to the standard library ServeMux
there are other third-party routers



godoc.org/?q=router

Bookmarks PM Hawk J Android

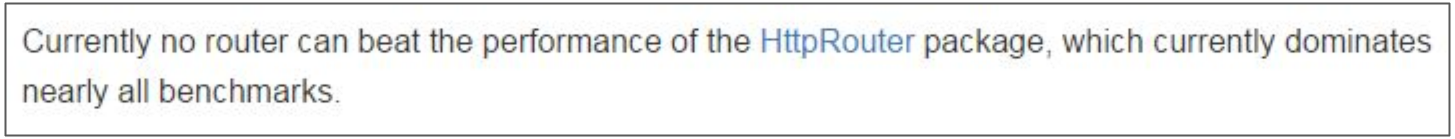
GoDoc Home Index About Search

router Go!

Try this search on Go-Search or GitHub.

Path	Synopsis
github.com/gorilla/mux	Package gorilla/mux implements a request router and dispatcher.
github.com/julienschmidt/httprouter	Package httprouter is a trie based high performance HTTP request router.
github.com/tedsuo/rata	Package rata provides three things: Routes, a Router, and a RequestGenerator.
github.com/gorilla/pat	Package gorilla/pat is a request router and dispatcher with a pat-like interface.
code.google.com/p/gorilla/mux	Package gorilla/mux implements a request router and dispatcher.
github.com/gogits/gogs/routers	
github.com/cloudfoundry/gorouter/route	
github.com/docker/distribution/registry/api/v2	Package v2 describes routes, urls and the error codes used in the Docker Registry JSON HTTP API V2.
github.com/gocraft/web	Go Router + Middleware.
github.com/beego/wetalk/routers/base	Package routers implemented controller methods of beego.
github.com/drone/routes	Package routes a simple http routing API for the Go programming language, compatible with the standard http.ListenAndServe function.
github.com/gliderlabs/logspout/router	generated by go-extpoints -- DO NOT EDIT
github.com/naoina/denco	Package denco provides fast URL router.
gopkg.in/go-on/router.v2/route	Package route provides slim representation of routes that is used by go-on/router.Router and may be used by client side libraries such as gopherjs.

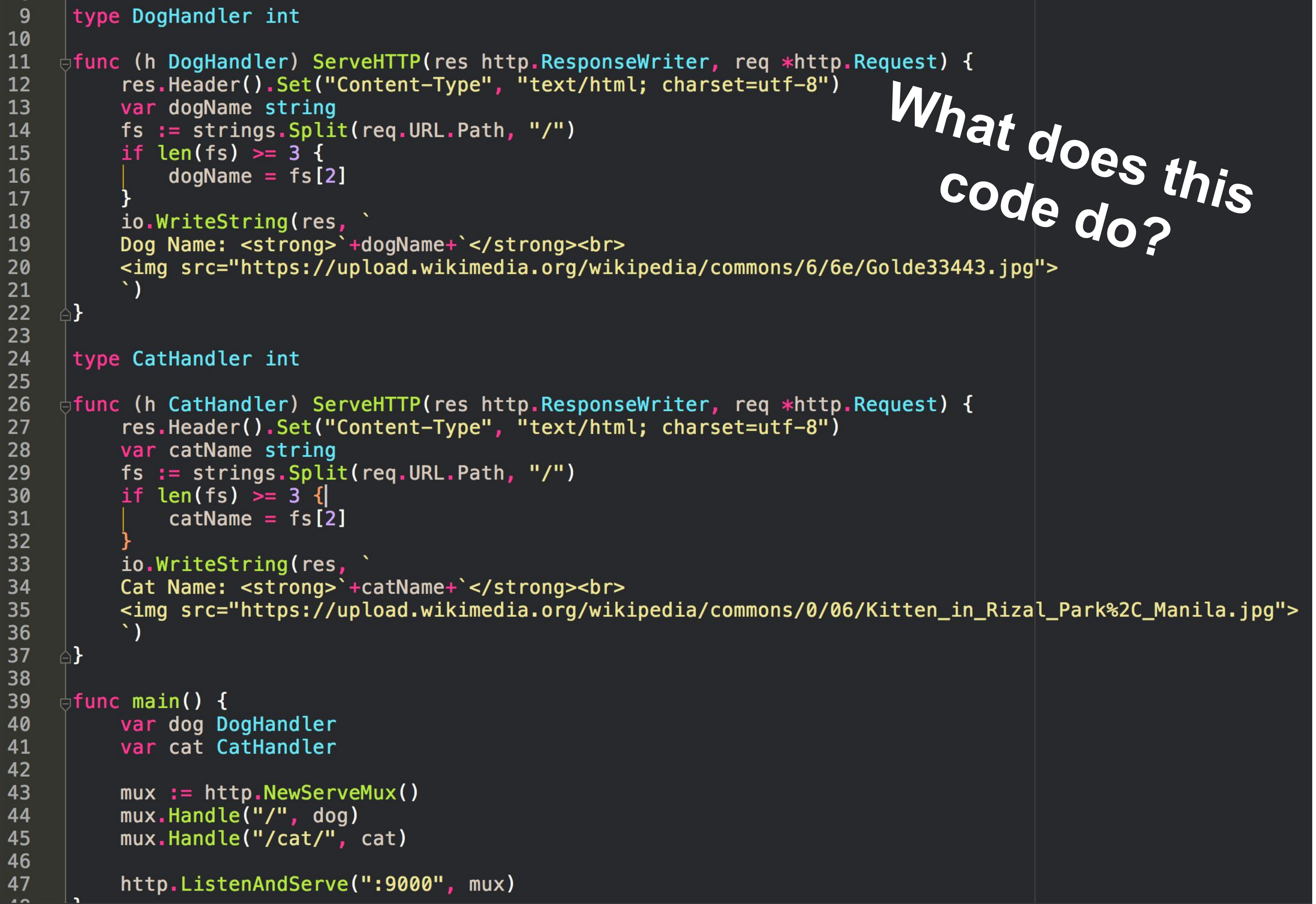
https://github.com/julienschmidt/go-http-routing-benchmark



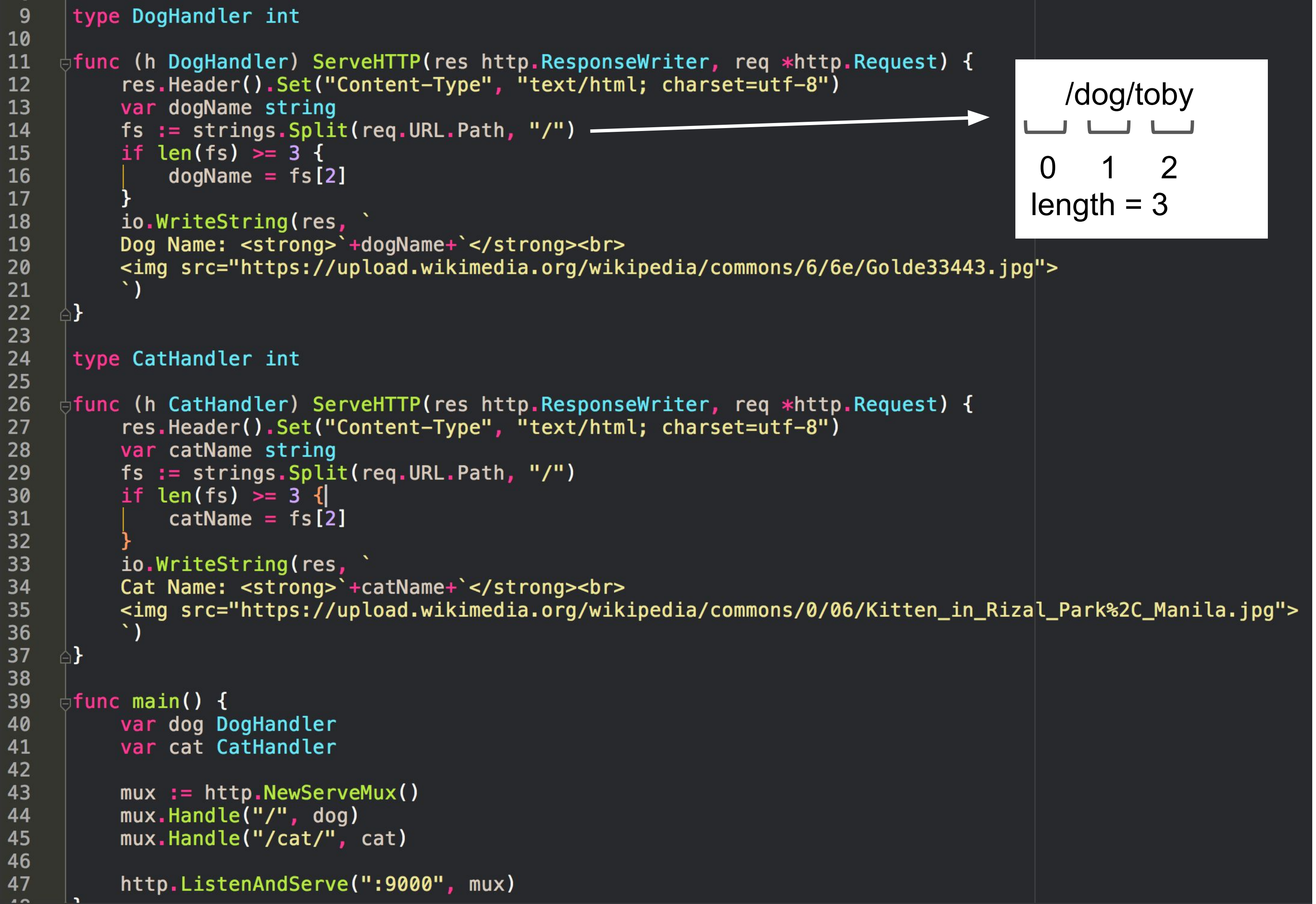
Currently no router can beat the performance of the HttpRouter package, which currently dominates nearly all benchmarks.

restful web services

What does this code do?



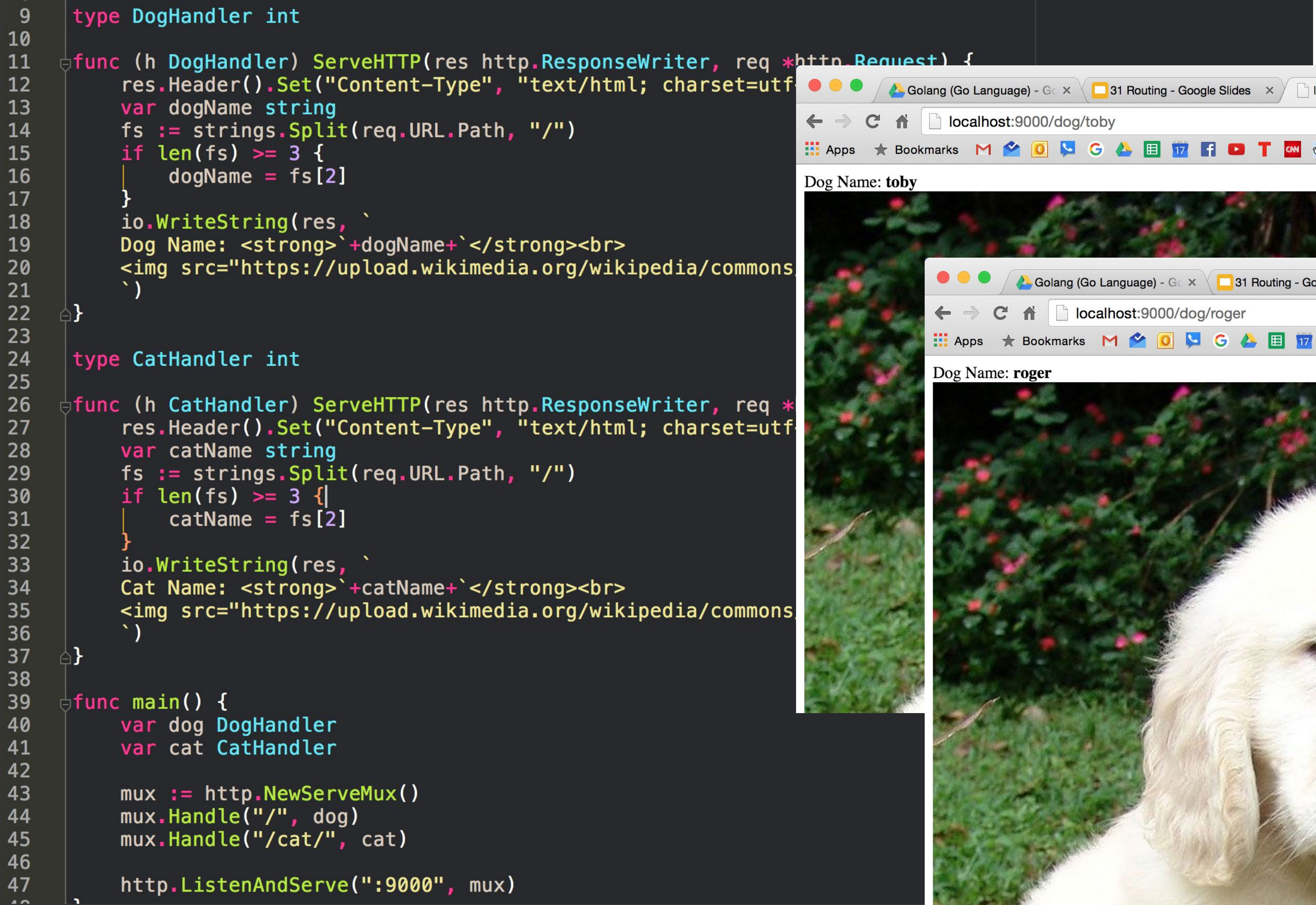
```go
type DogHandler int

func (h DogHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
    res.Header().Set("Content-Type", "text/html; charset=utf-8")
    var dogName string
    fs := strings.Split(req.URL.Path, "/")
    if len(fs) >= 3 {
        dogName = fs[2]
    }
    io.WriteString(res, `
    Dog Name: <strong>`+dogName+`</strong><br>
    <img src="https://upload.wikimedia.org/wikipedia/commons/6/6e/Golde33443.jpg">
    `)
}

type CatHandler int

func (h CatHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
    res.Header().Set("Content-Type", "text/html; charset=utf-8")
    var catName string
    fs := strings.Split(req.URL.Path, "/")
    if len(fs) >= 3 {
        catName = fs[2]
    }
    io.WriteString(res, `
    Cat Name: <strong>`+catName+`</strong><br>
    <img src="https://upload.wikimedia.org/wikipedia/commons/0/06/Kitten_in_Rizal_Park%2C_Manila.jpg">
    `)
}

func main() {
    var dog DogHandler
    var cat CatHandler

    mux := http.NewServeMux()
    mux.Handle("/", dog)
    mux.Handle("/cat/", cat)

    http.ListenAndServe(":9000", mux)
}
```



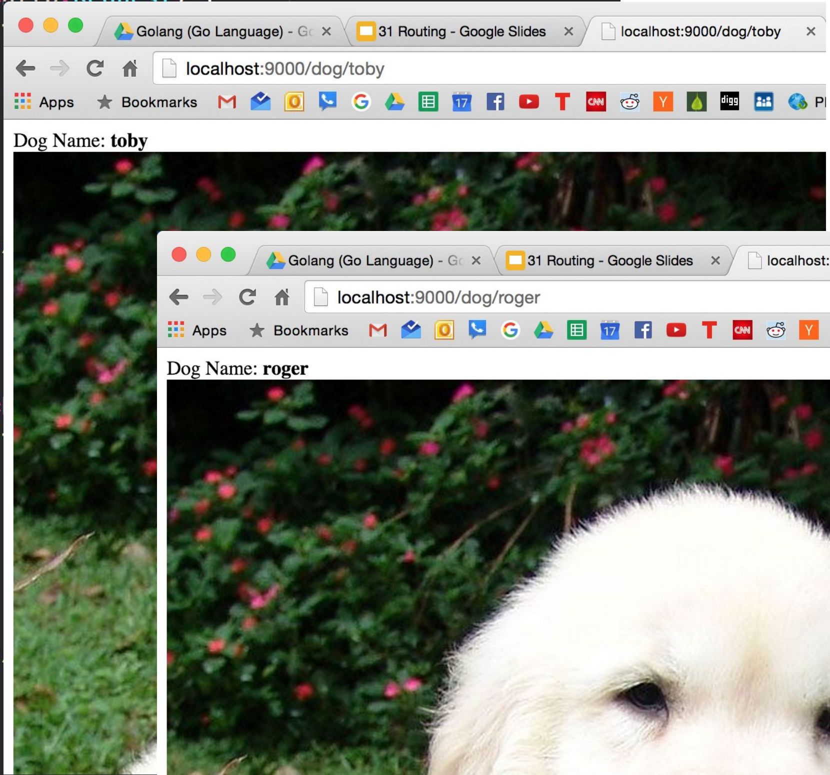
```go
type DogHandler int

func (h DogHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	res.Header().Set("Content-Type", "text/html; charset=utf-8")
	var dogName string
	fs := strings.Split(req.URL.Path, "/")
	if len(fs) >= 3 {
		dogName = fs[2]
	}
	io.WriteString(res, `
	Dog Name: <strong>`+dogName+`</strong><br>
	<img src="https://upload.wikimedia.org/wikipedia/commons/6/6e/Golde33443.jpg">
	`)
}

type CatHandler int

func (h CatHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	res.Header().Set("Content-Type", "text/html; charset=utf-8")
	var catName string
	fs := strings.Split(req.URL.Path, "/")
	if len(fs) >= 3 {
		catName = fs[2]
	}
	io.WriteString(res, `
	Cat Name: <strong>`+catName+`</strong><br>
	<img src="https://upload.wikimedia.org/wikipedia/commons/0/06/Kitten_in_Rizal_Park%2C_Manila.jpg">
	`)
}

func main() {
	var dog DogHandler
	var cat CatHandler

	mux := http.NewServeMux()
	mux.Handle("/", dog)
	mux.Handle("/cat/", cat)

	http.ListenAndServe(":9000", mux)
}
```

$$\underbrace{}_{0}/\underbrace{\text{dog}}_{1}/\underbrace{\text{toby}}_{2}$$

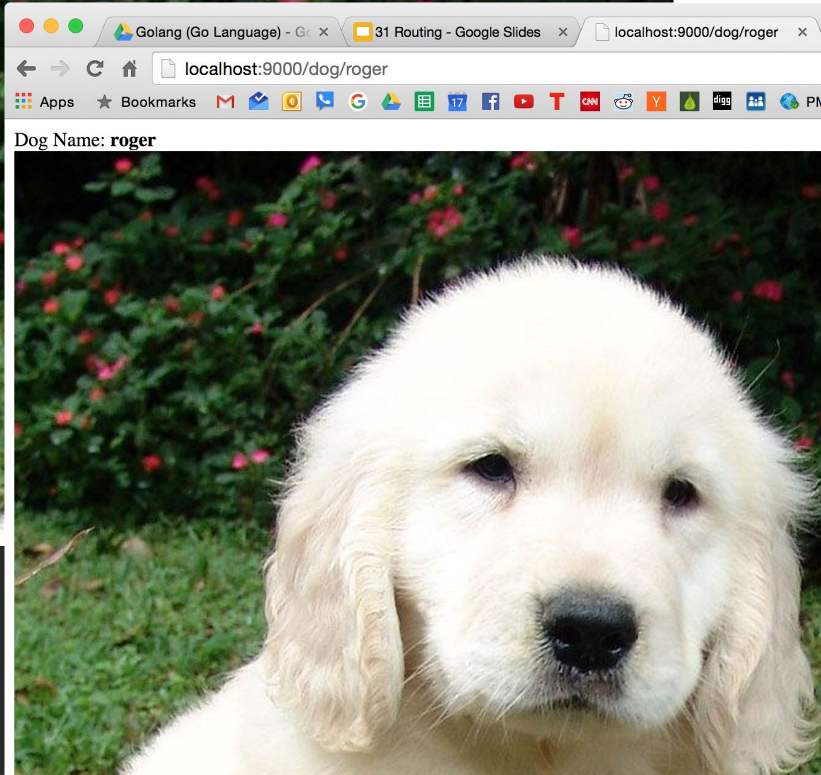
length = 3



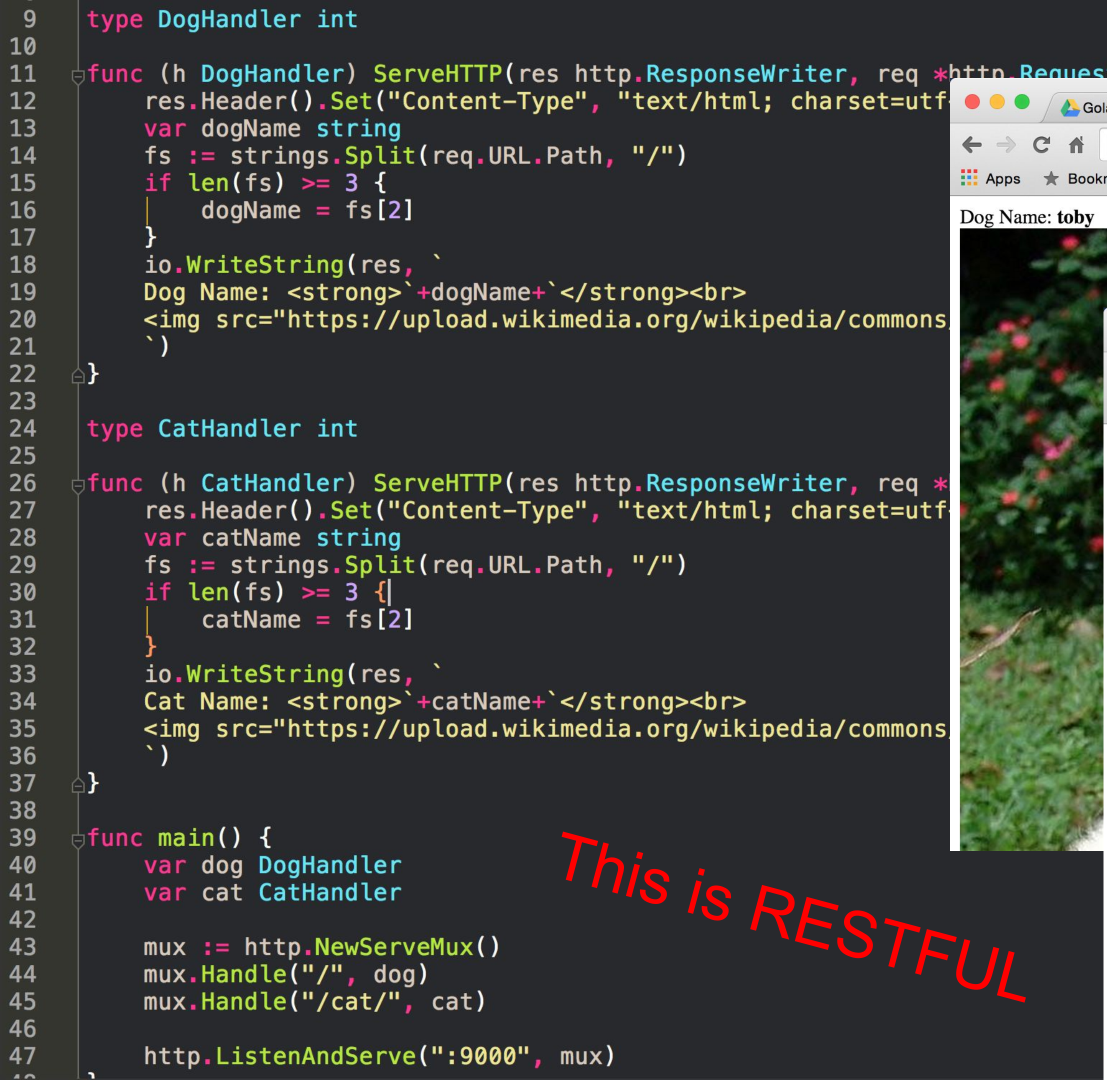
```go
type DogHandler int

func (h DogHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
    res.Header().Set("Content-Type", "text/html; charset=utf
    var dogName string
    fs := strings.Split(req.URL.Path, "/")
    if len(fs) >= 3 {
        dogName = fs[2]
    }
    io.WriteString(res, `
    Dog Name: <strong>`+dogName+`</strong><br>
    <img src="https://upload.wikimedia.org/wikipedia/commons
    `)
}

type CatHandler int

func (h CatHandler) ServeHTTP(res http.ResponseWriter, req *
    res.Header().Set("Content-Type", "text/html; charset=utf
    var catName string
    fs := strings.Split(req.URL.Path, "/")
    if len(fs) >= 3 {
        catName = fs[2]
    }
    io.WriteString(res, `
    Cat Name: <strong>`+catName+`</strong><br>
    <img src="https://upload.wikimedia.org/wikipedia/commons
    `)
}

func main() {
    var dog DogHandler
    var cat CatHandler

    mux := http.NewServeMux()
    mux.Handle("/", dog)
    mux.Handle("/cat/", cat)

    http.ListenAndServe(":9000", mux)
```



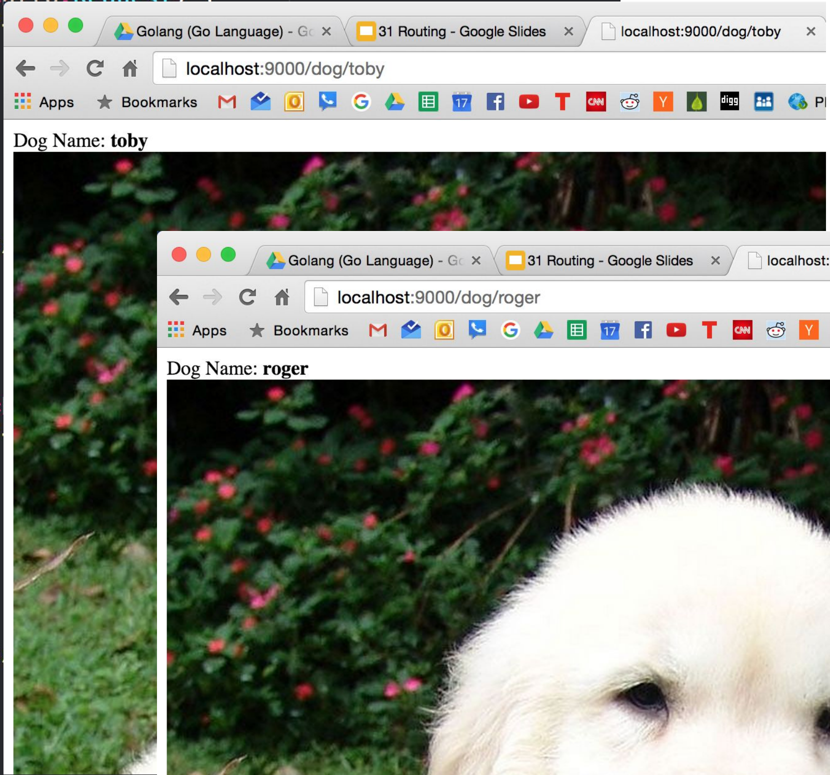
Dog Name: **toby**



Dog Name: **roger**

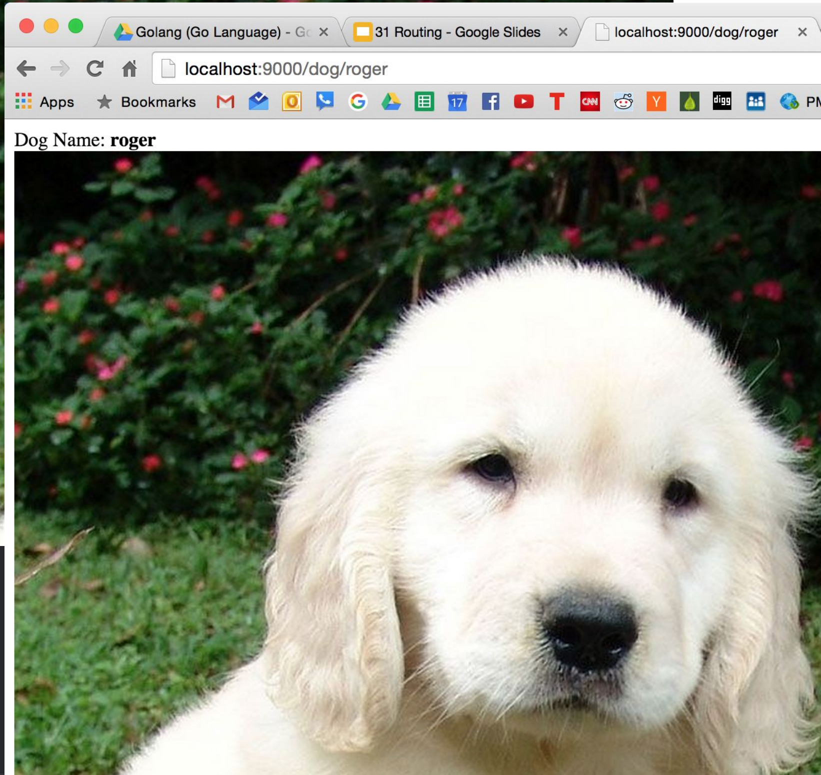


```go
type DogHandler int

func (h DogHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	res.Header().Set("Content-Type", "text/html; charset=utf
	var dogName string
	fs := strings.Split(req.URL.Path, "/")
	if len(fs) >= 3 {
		dogName = fs[2]
	}
	io.WriteString(res, `
	Dog Name: <strong>`+dogName+`</strong><br>
	<img src="https://upload.wikimedia.org/wikipedia/commons
	`)
}

type CatHandler int

func (h CatHandler) ServeHTTP(res http.ResponseWriter, req *
	res.Header().Set("Content-Type", "text/html; charset=utf
	var catName string
	fs := strings.Split(req.URL.Path, "/")
	if len(fs) >= 3 {
		catName = fs[2]
	}
	io.WriteString(res, `
	Cat Name: <strong>`+catName+`</strong><br>
	<img src="https://upload.wikimedia.org/wikipedia/commons
	`)
}

func main() {
	var dog DogHandler
	var cat CatHandler

	mux := http.NewServeMux()
	mux.Handle("/", dog)
	mux.Handle("/cat/", cat)

	http.ListenAndServe(":9000", mux)
```



localhost:9000/dog/toby

Dog Name: **toby**

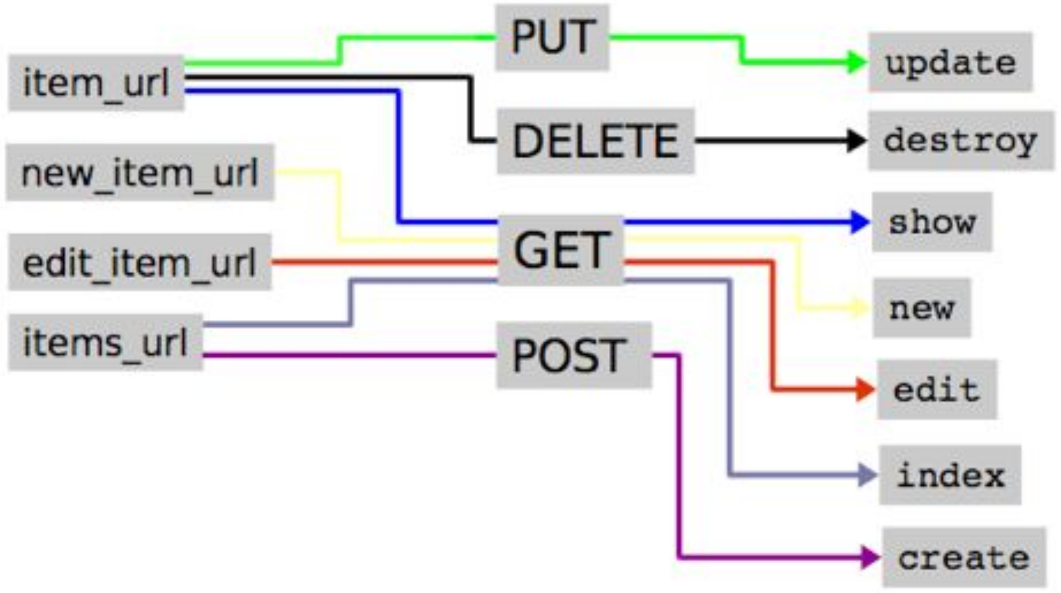


localhost:9000/dog/roger

Dog Name: **roger**

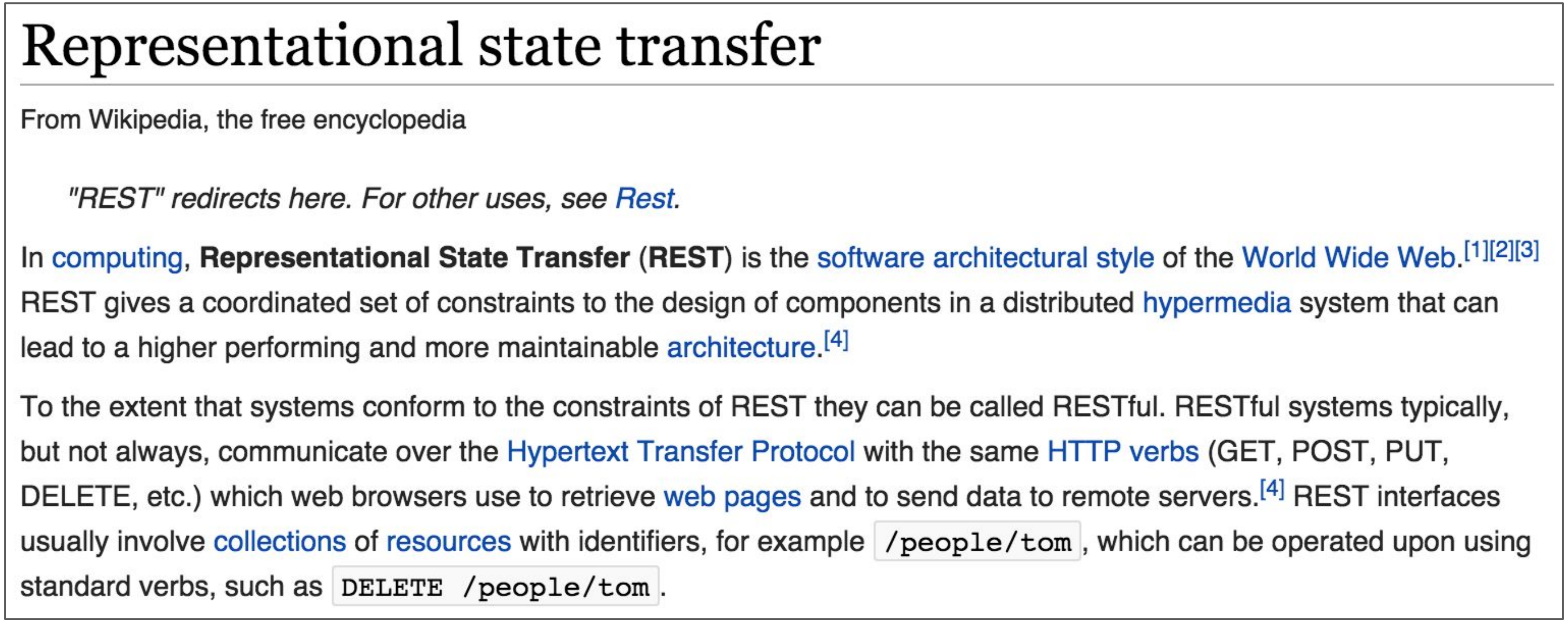
This is RESTFUL

EVERY MORNING
YOU HAVE TWO CHOICES:
CONTINUE TO SLEEP WITH
YOUR DREAMS
OR WAKE UP
AND CHASE THEM


item_url
new_item_url
edit_item_url
items_url
PUT
DELETE
GET
POST
update
destroy
show
new
edit
index
create

RESTful Hint #403

If you have to ship an SDK for your RESTful API, it's not a RESTful API.



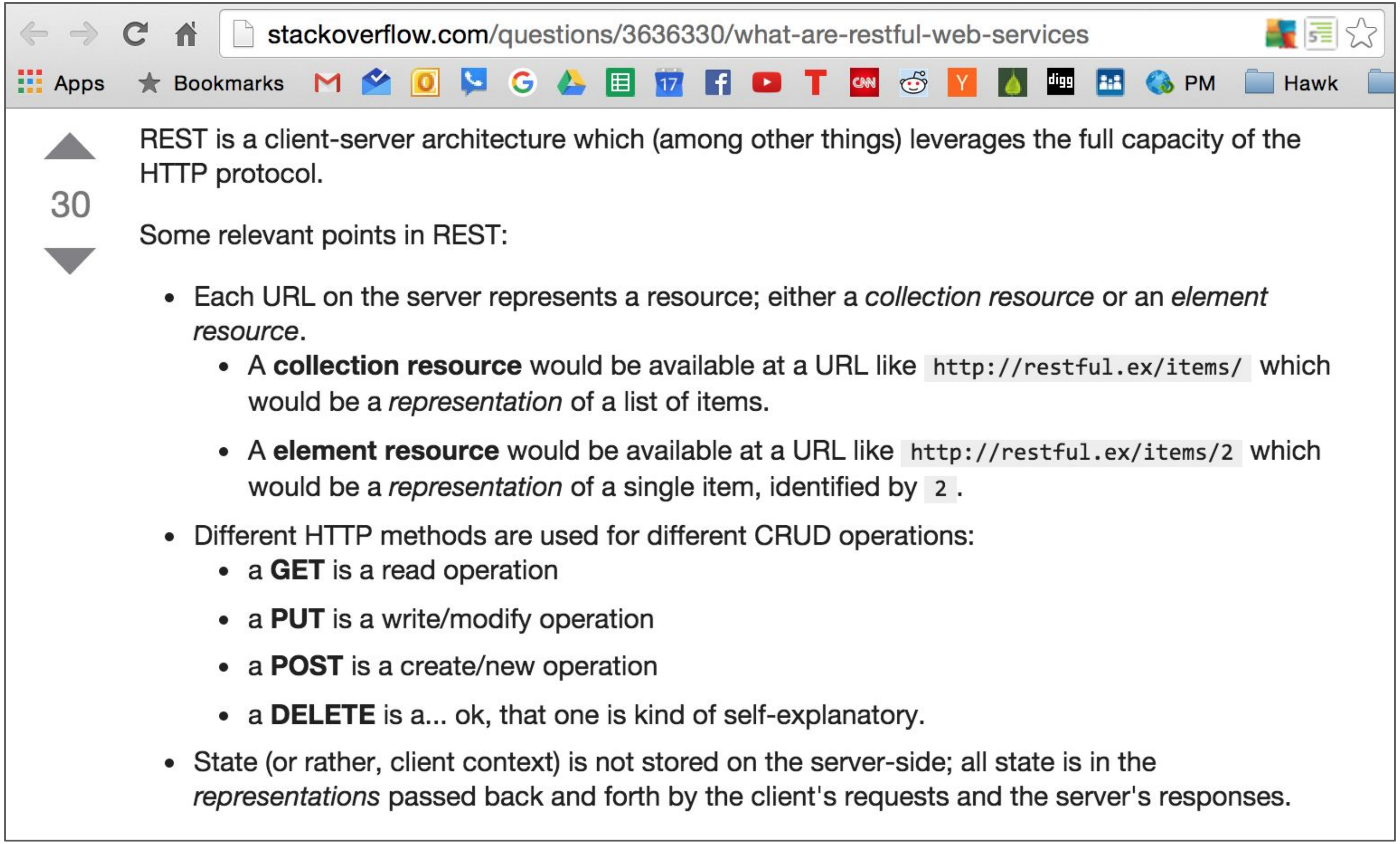
Representational state transfer

From Wikipedia, the free encyclopedia

"REST" redirects here. For other uses, see Rest.

In computing, **Representational State Transfer** (**REST**) is the software architectural style of the World Wide Web.[1][2][3] REST gives a coordinated set of constraints to the design of components in a distributed hypermedia system that can lead to a higher performing and more maintainable architecture.[4]

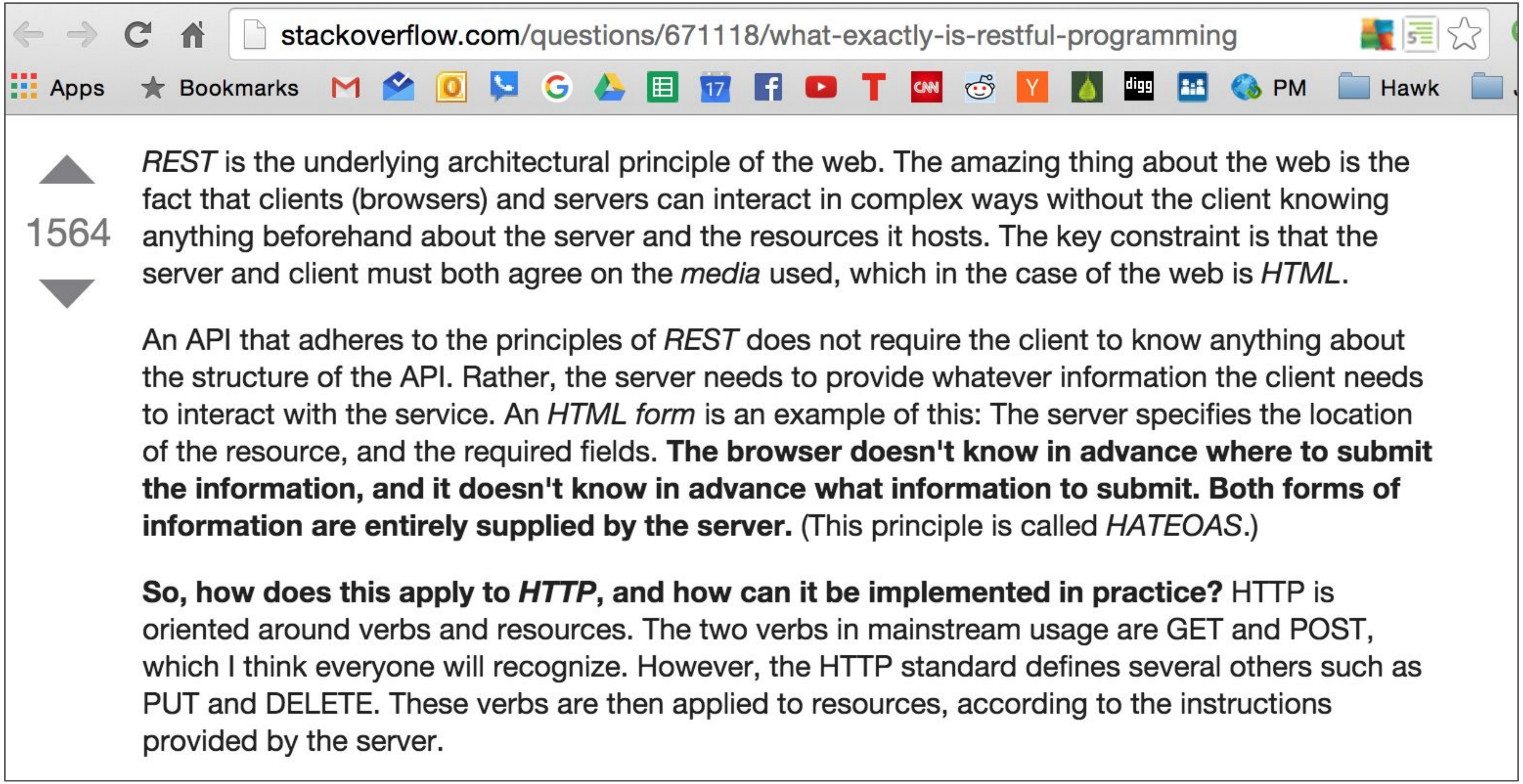
To the extent that systems conform to the constraints of REST they can be called RESTful. RESTful systems typically, but not always, communicate over the Hypertext Transfer Protocol with the same HTTP verbs (GET, POST, PUT, DELETE, etc.) which web browsers use to retrieve web pages and to send data to remote servers.[4] REST interfaces usually involve collections of resources with identifiers, for example `/people/tom`, which can be operated upon using standard verbs, such as `DELETE /people/tom`.


stackoverflow.com/questions/3636330/what-are-restful-web-services
Apps
Bookmarks
PM
Hawk
30

REST is a client-server architecture which (among other things) leverages the full capacity of the HTTP protocol.

Some relevant points in REST:

- Each URL on the server represents a resource; either a *collection resource* or an *element resource*.
  - A **collection resource** would be available at a URL like `http://restful.ex/items/` which would be a *representation* of a list of items.
  - A **element resource** would be available at a URL like `http://restful.ex/items/2` which would be a *representation* of a single item, identified by `2`.
- Different HTTP methods are used for different CRUD operations:
  - a **GET** is a read operation
  - a **PUT** is a write/modify operation
  - a **POST** is a create/new operation
  - a **DELETE** is a... ok, that one is kind of self-explanatory.
- State (or rather, client context) is not stored on the server-side; all state is in the *representations* passed back and forth by the client's requests and the server's responses.


stackoverflow.com/questions/671118/what-exactly-is-restful-programming
Apps
Bookmarks
PM
Hawk
1564

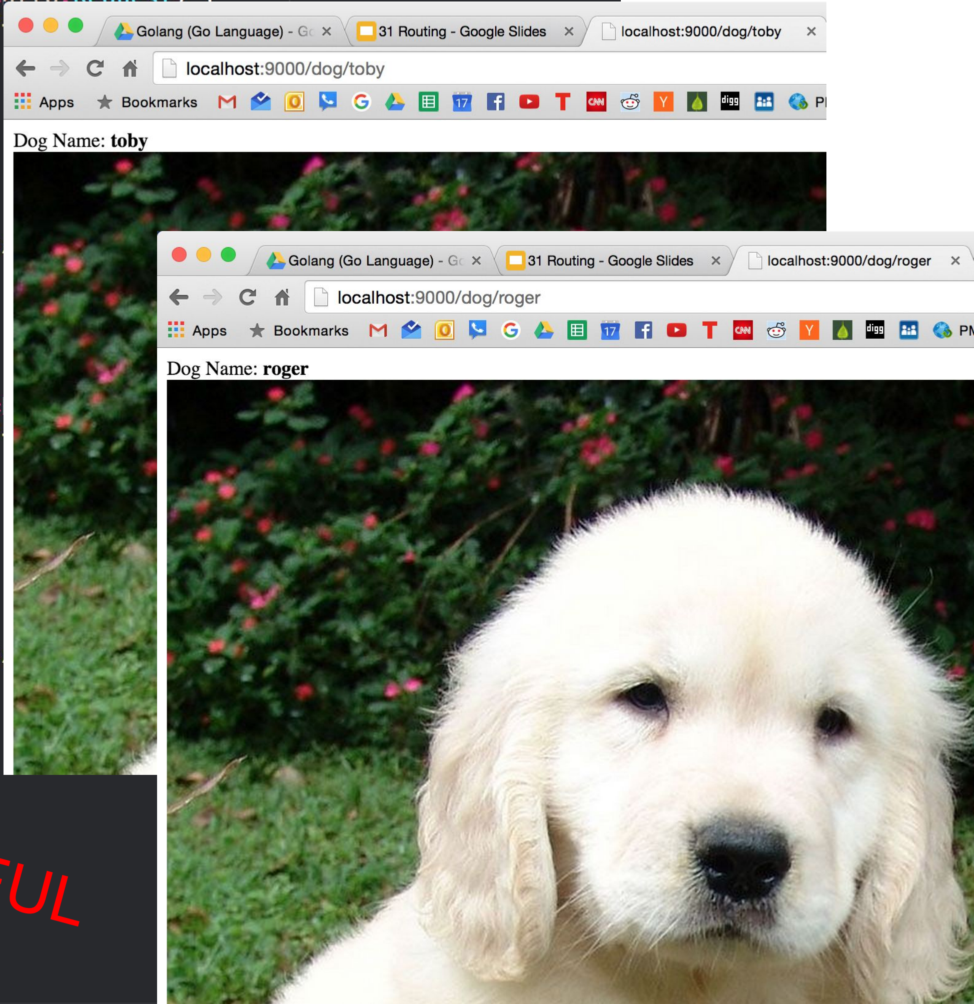
REST is the underlying architectural principle of the web. The amazing thing about the web is the fact that clients (browsers) and servers can interact in complex ways without the client knowing anything beforehand about the server and the resources it hosts. The key constraint is that the server and client must both agree on the *media* used, which in the case of the web is *HTML*.

An API that adheres to the principles of *REST* does not require the client to know anything about the structure of the API. Rather, the server needs to provide whatever information the client needs to interact with the service. An *HTML form* is an example of this: The server specifies the location of the resource, and the required fields. **The browser doesn't know in advance where to submit the information, and it doesn't know in advance what information to submit. Both forms of information are entirely supplied by the server.** (This principle is called *HATEOAS*.)

So, how does this apply to *HTTP*, and how can it be implemented in practice? HTTP is oriented around verbs and resources. The two verbs in mainstream usage are GET and POST, which I think everyone will recognize. However, the HTTP standard defines several others such as PUT and DELETE. These verbs are then applied to resources, according to the instructions provided by the server.

```go
type DogHandler int

func (h DogHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	res.Header().Set("Content-Type", "text/html; charset=utf
	var dogName string
	fs := strings.Split(req.URL.Path, "/")
	if len(fs) >= 3 {
		dogName = fs[2]
	}
	io.WriteString(res, `
	Dog Name: <strong>`+dogName+`</strong><br>
	<img src="https://upload.wikimedia.org/wikipedia/commons
	`)
}

type CatHandler int

func (h CatHandler) ServeHTTP(res http.ResponseWriter, req *
	res.Header().Set("Content-Type", "text/html; charset=utf
	var catName string
	fs := strings.Split(req.URL.Path, "/")
	if len(fs) >= 3 {
		catName = fs[2]
	}
	io.WriteString(res, `
	Cat Name: <strong>`+catName+`</strong><br>
	<img src="https://upload.wikimedia.org/wikipedia/commons
	`)
}

func main() {
	var dog DogHandler
	var cat CatHandler

	mux := http.NewServeMux()
	mux.Handle("/", dog)
	mux.Handle("/cat/", cat)

	http.ListenAndServe(":9000", mux)
```

This is RESTFUL


Golang (Go Language) - Go
31 Routing - Google Slides
localhost:9000/dog/toby
Apps
Bookmarks

localhost:9000/dog/toby

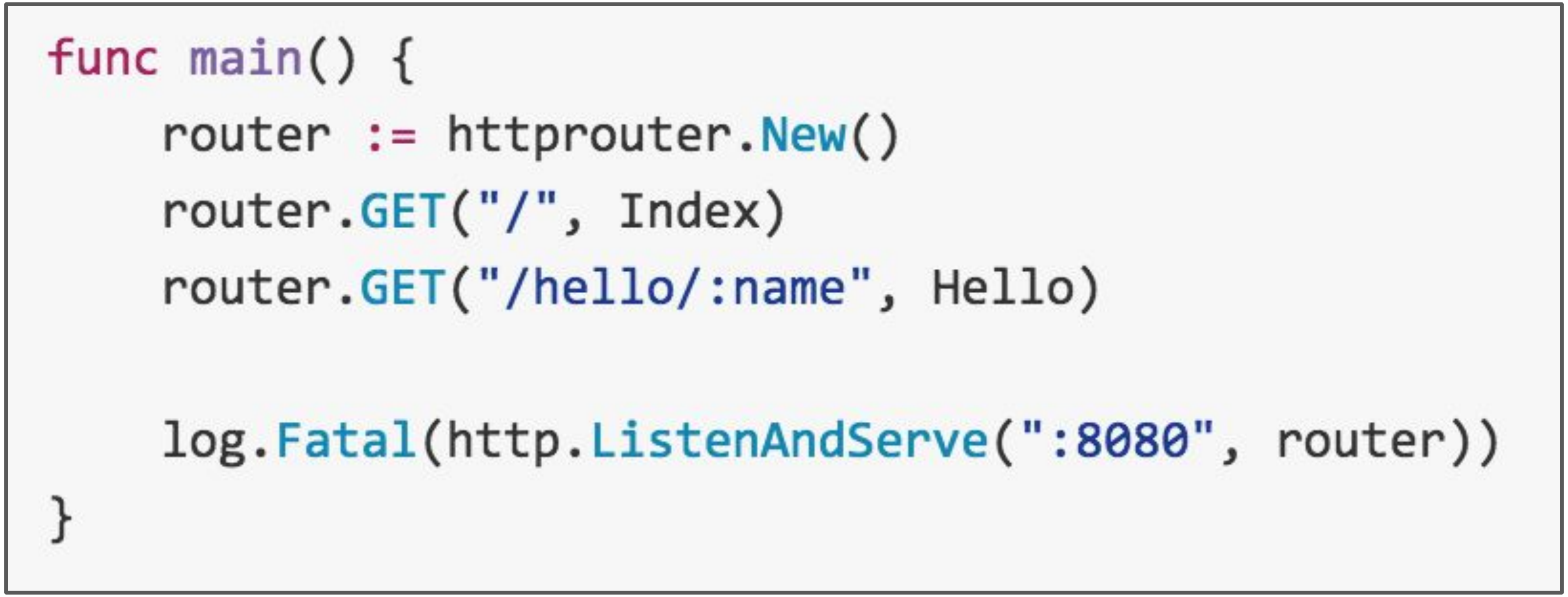
Dog Name: **toby**

Golang (Go Language) - Go
31 Routing - Google Slides
localhost:9000/dog/roger
Apps
Bookmarks

localhost:9000/dog/roger

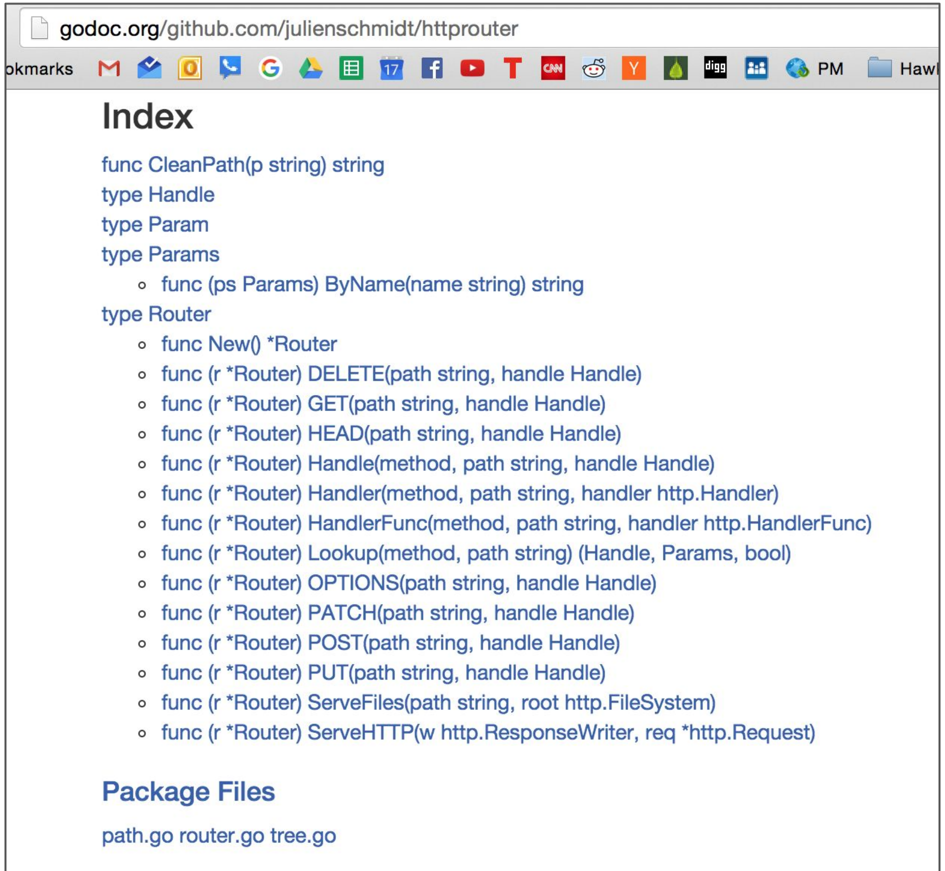
Dog Name: **roger**

github.com/julienschmidt/httprouter



```go
func main() {
    router := httprouter.New()
    router.GET("/", Index)
    router.GET("/hello/:name", Hello)

    log.Fatal(http.ListenAndServe(":8080", router))
}
```

github.com/julienschmidt/httprouter



godoc.org/github.com/julienschmidt/httprouter

okmarks PM Hawl

Index

Package Files

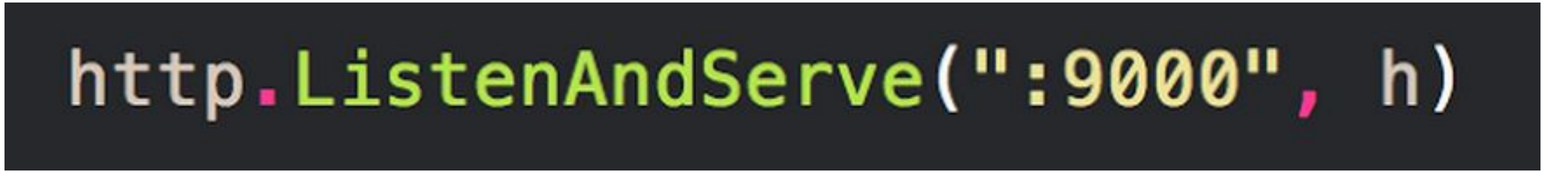
path.go router.go tree.go

HandleFunc

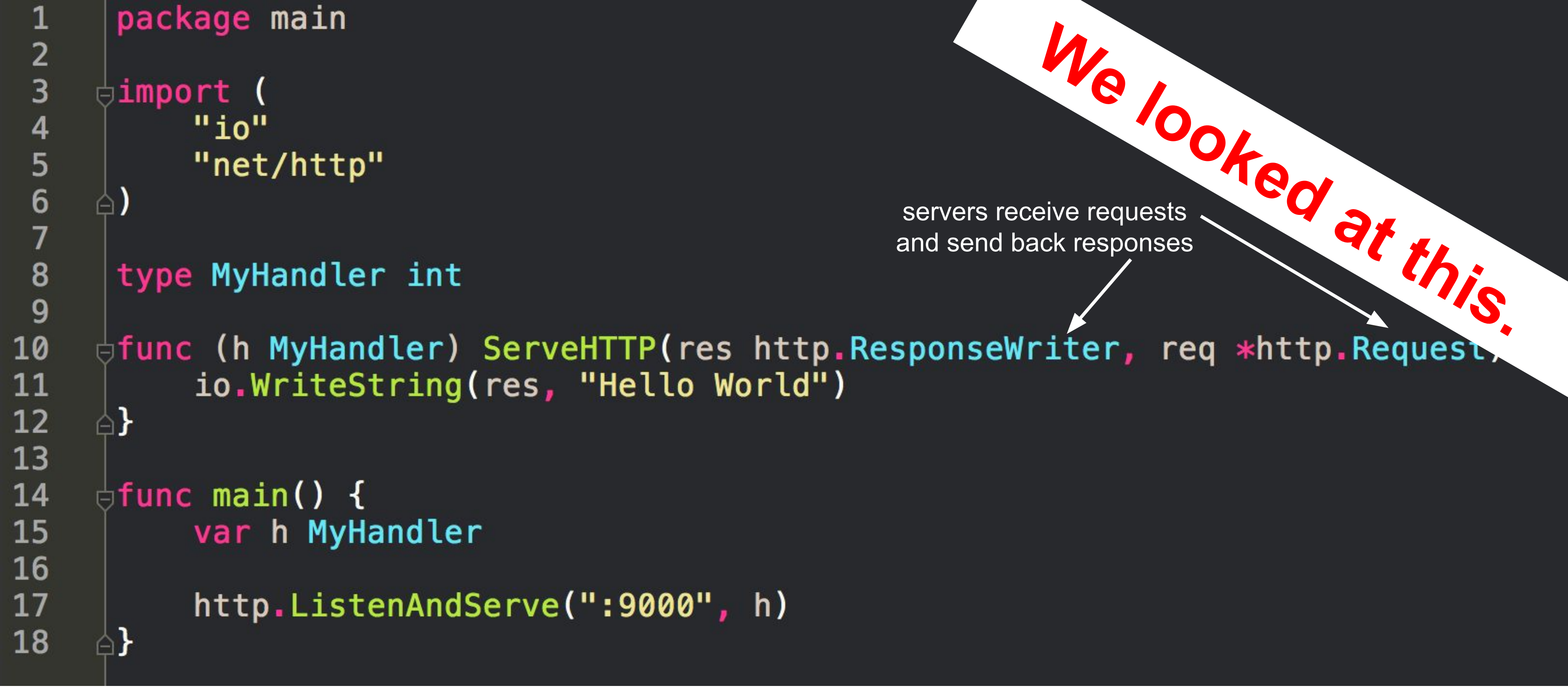
http.ListenAndServe

We looked at this.

- func ListenAndServe(**addr** string, **handler** Handler) error
- a handler implements the handler interface
  - that means the type has this method:
    - ServeHTTP(ResponseWriter, *Request)



```
http.ListenAndServe(":9000", h)
```

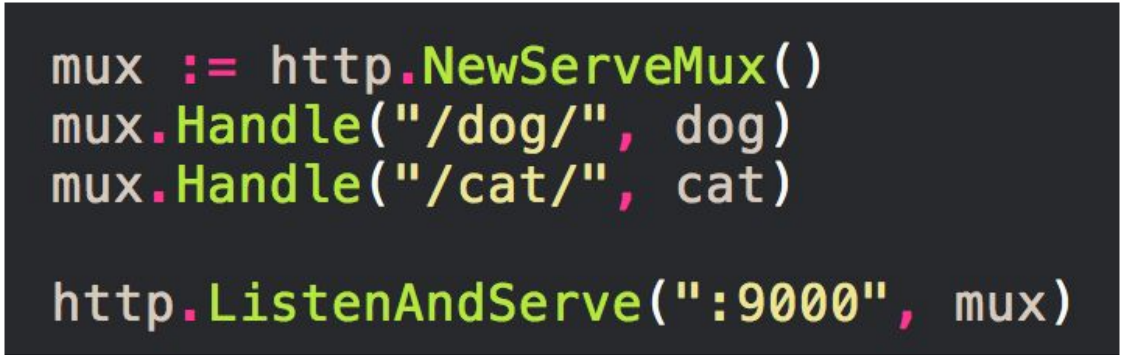


```go
package main

import (
	"io"
	"net/http"
)

type MyHandler int

func (h MyHandler) ServeHTTP(res http.ResponseWriter, req *http.Request)
	io.WriteString(res, "Hello World")
}

func main() {
	var h MyHandler

	http.ListenAndServe(":9000", h)
}
```

http.NewServeMux()

We looked at this.

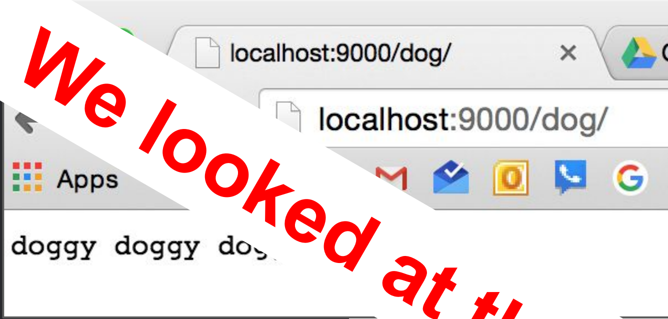
- ServeMux
  - a multiplexer
  - allows us to do routing
- *ServeMux.Handle
  - func (mux *ServeMux) Handle(pattern string, **handler** Handler)



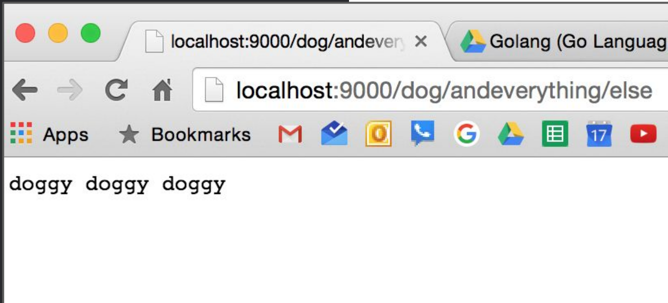
```
mux := http.NewServeMux()
mux.Handle("/dog/", dog)
mux.Handle("/cat/", cat)

http.ListenAndServe(":9000", mux)
```

We looked at this.

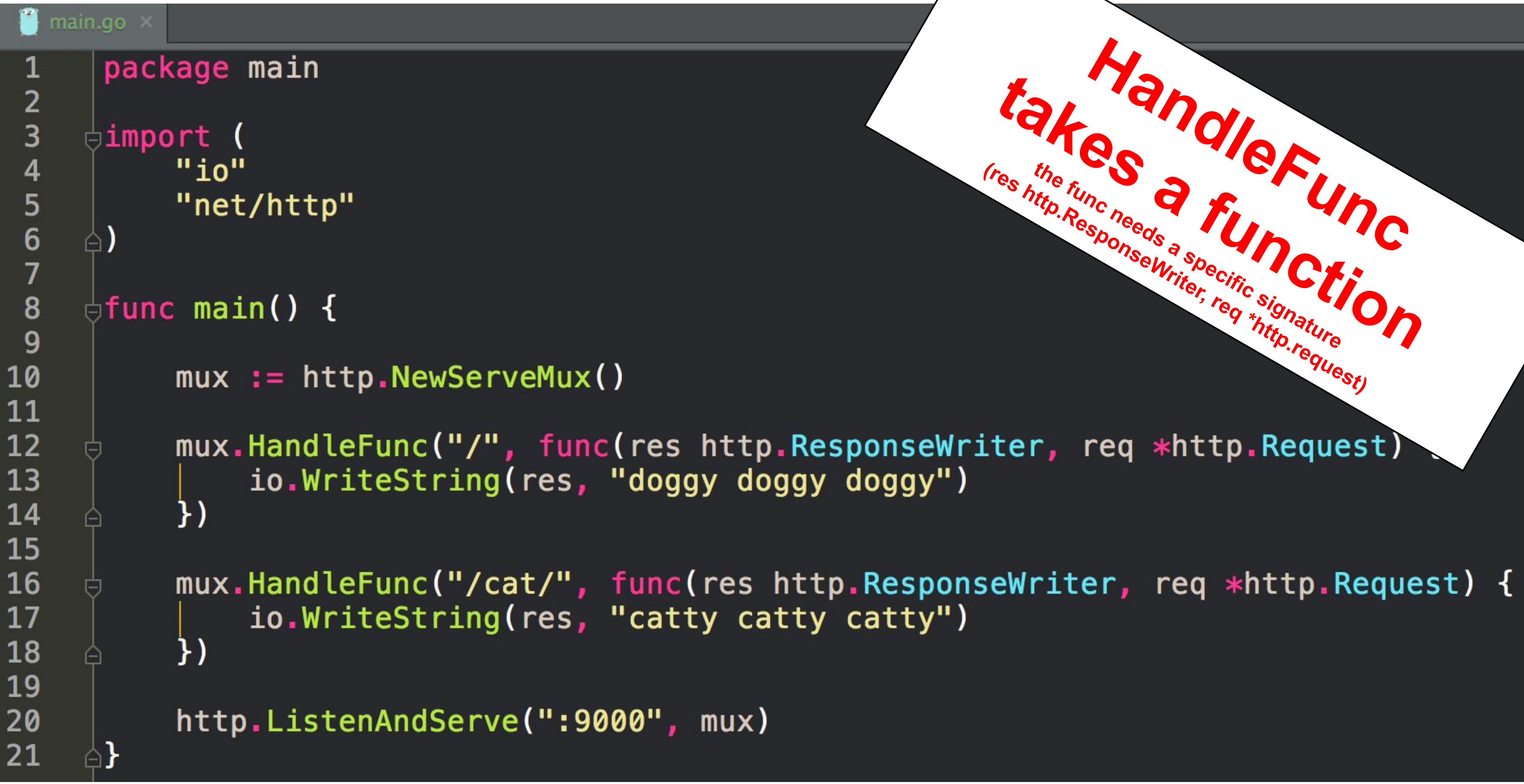
```go
package main

import (
	"io"
	"net/http"
)

type DogHandler int

func (h DogHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "doggy doggy doggy")
}

type CatHandler int

func (h CatHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "catty catty catty")
}

func main() {
	var dog DogHandler
	var cat CatHandler

	mux := http.NewServeMux()
	mux.Handle("/dog/", dog)
	mux.Handle("/cat/", cat)

	http.ListenAndServe(":9000", mux)
}
```



```
doggy doggy do
```


localhost:9000/dog/andever
Golang (Go Languag
localhost:9000/dog/andeverything/else
Apps
Bookmarks

```
doggy doggy doggy
```

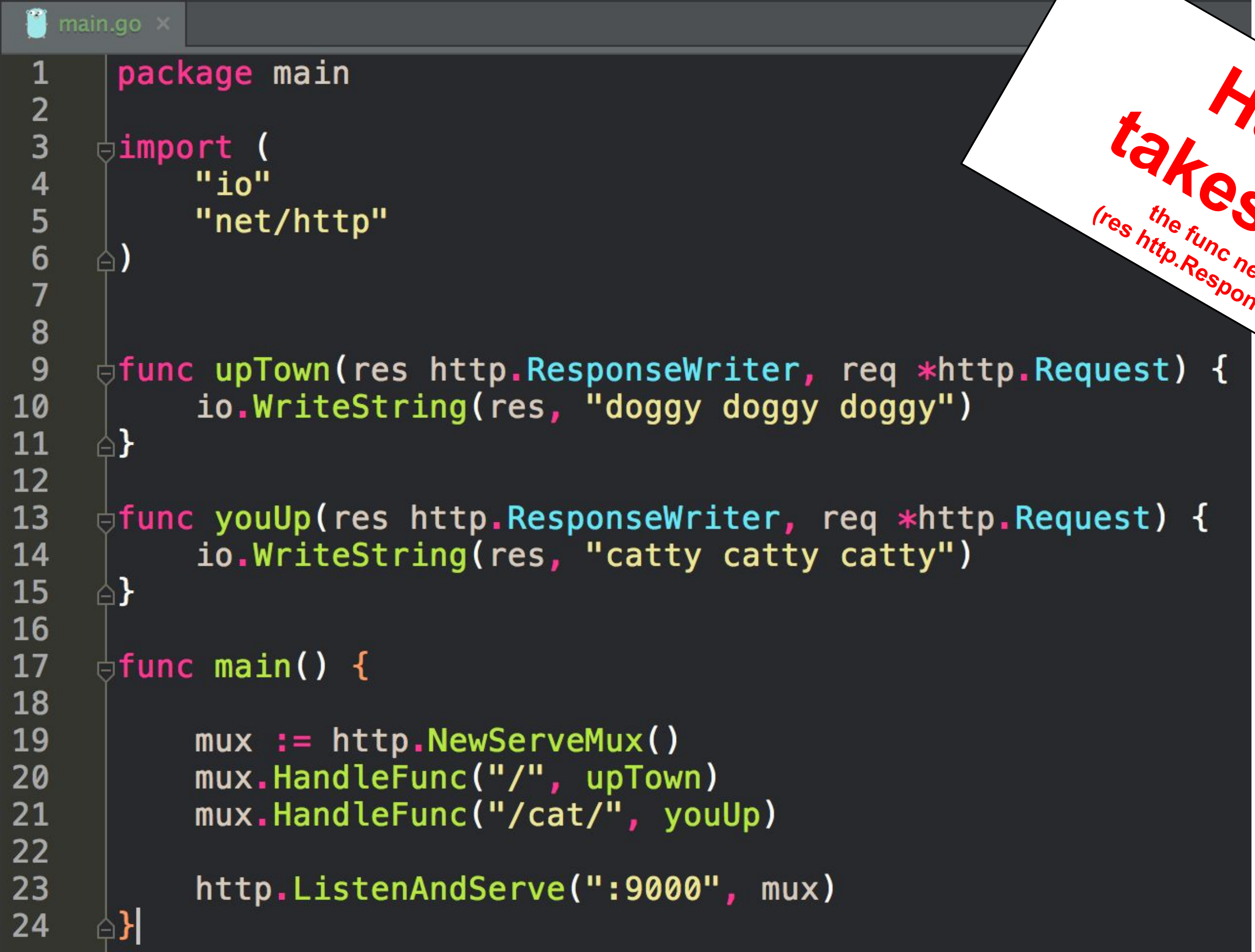


```go
package main

import (
	"io"
	"net/http"
)

func main() {

	mux := http.NewServeMux()

	mux.HandleFunc("/", func(res http.ResponseWriter, req *http.Request)
		io.WriteString(res, "doggy doggy doggy")
	})

	mux.HandleFunc("/cat/", func(res http.ResponseWriter, req *http.Request) {
		io.WriteString(res, "catty catty catty")
	})

	http.ListenAndServe(":9000", mux)
}
```

HandleFunc takes a function

the func needs a specific signature

(res http.ResponseWriter, req *http.request)

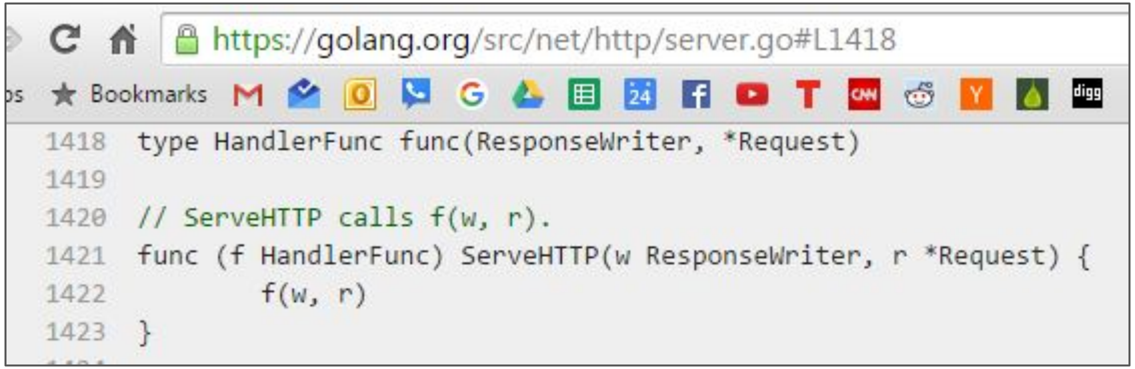


main.go

```go
package main

import (
	"io"
	"net/http"
)


func upTown(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "doggy doggy doggy")
}

func youUp(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "catty catty catty")
}

func main() {

	mux := http.NewServeMux()
	mux.HandleFunc("/", upTown)
	mux.HandleFunc("/cat/", youUp)

	http.ListenAndServe(":9000", mux)
}
```

HandleFunc takes a function

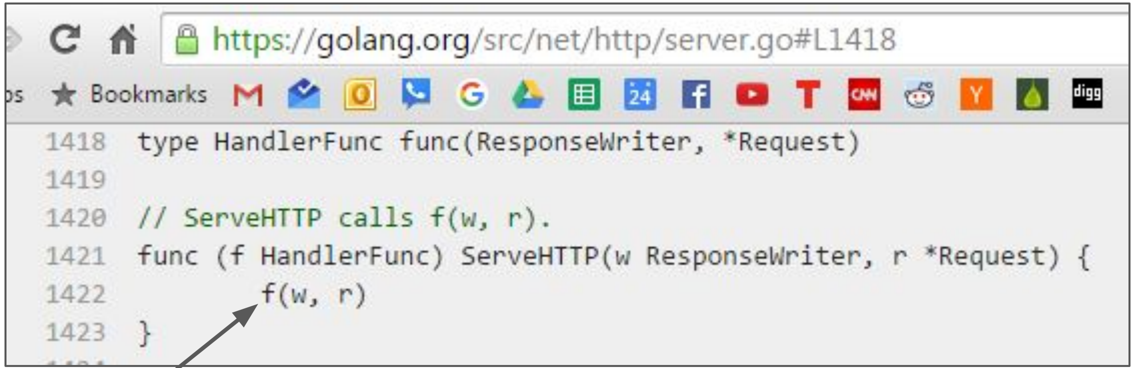
the func needs a specific signature
(res http.ResponseWriter, req *http.request)



```go
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(w, r).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
	f(w, r)
}
```

HandlerFunc is of type func(ReponseWriter, *Request)

The type HandlerFunc implements the handler interface

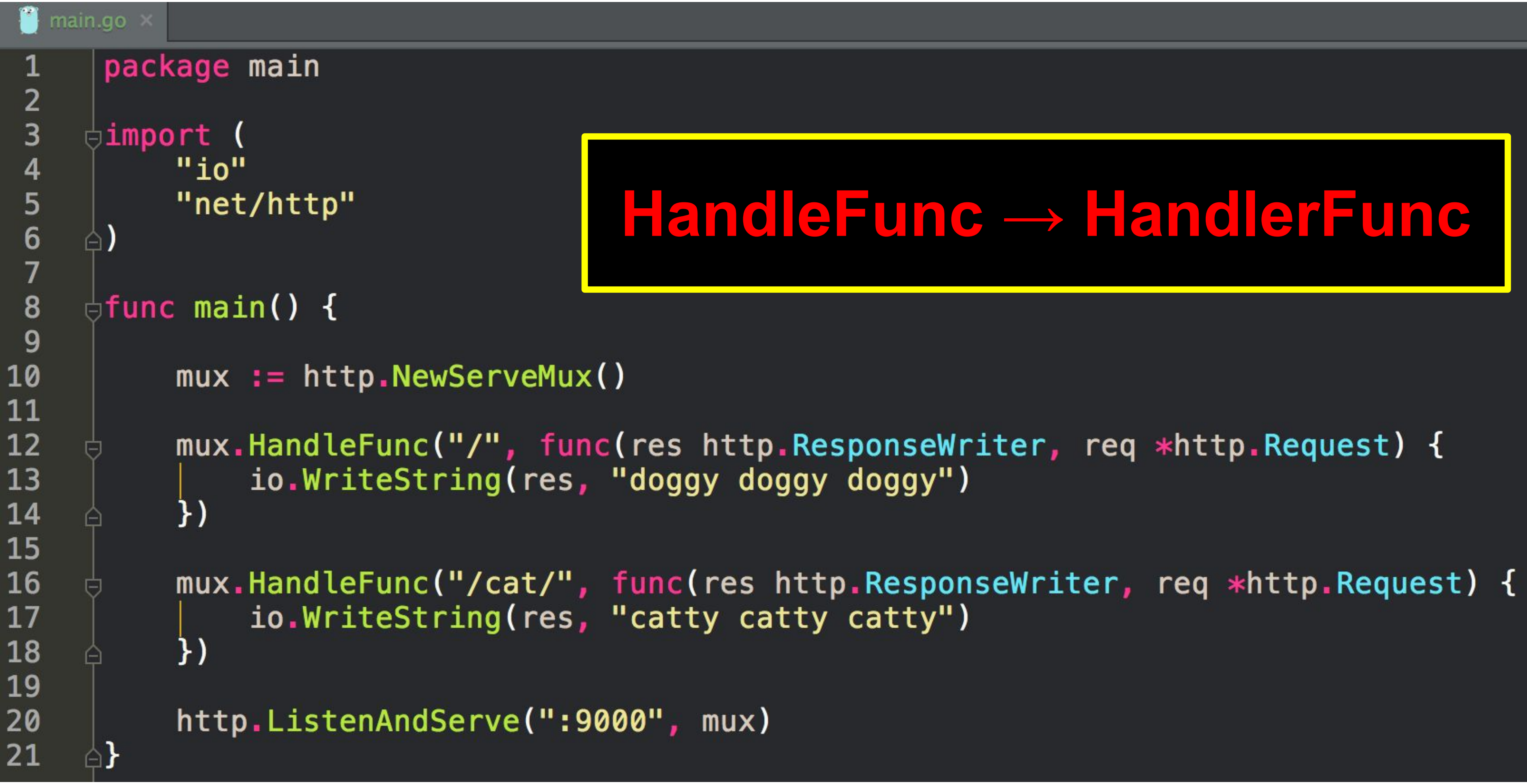

https://golang.org/src/net/http/server.go#L1418

```
type HandlerFunc func(ResponseWriter, *Request)

// ServeHTTP calls f(w, r).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
	f(w, r)
}
```

All it does is invoke itself.
It just calls itself here with two arguments

HandlerFunc is of type func(ReponseWriter, *Request)

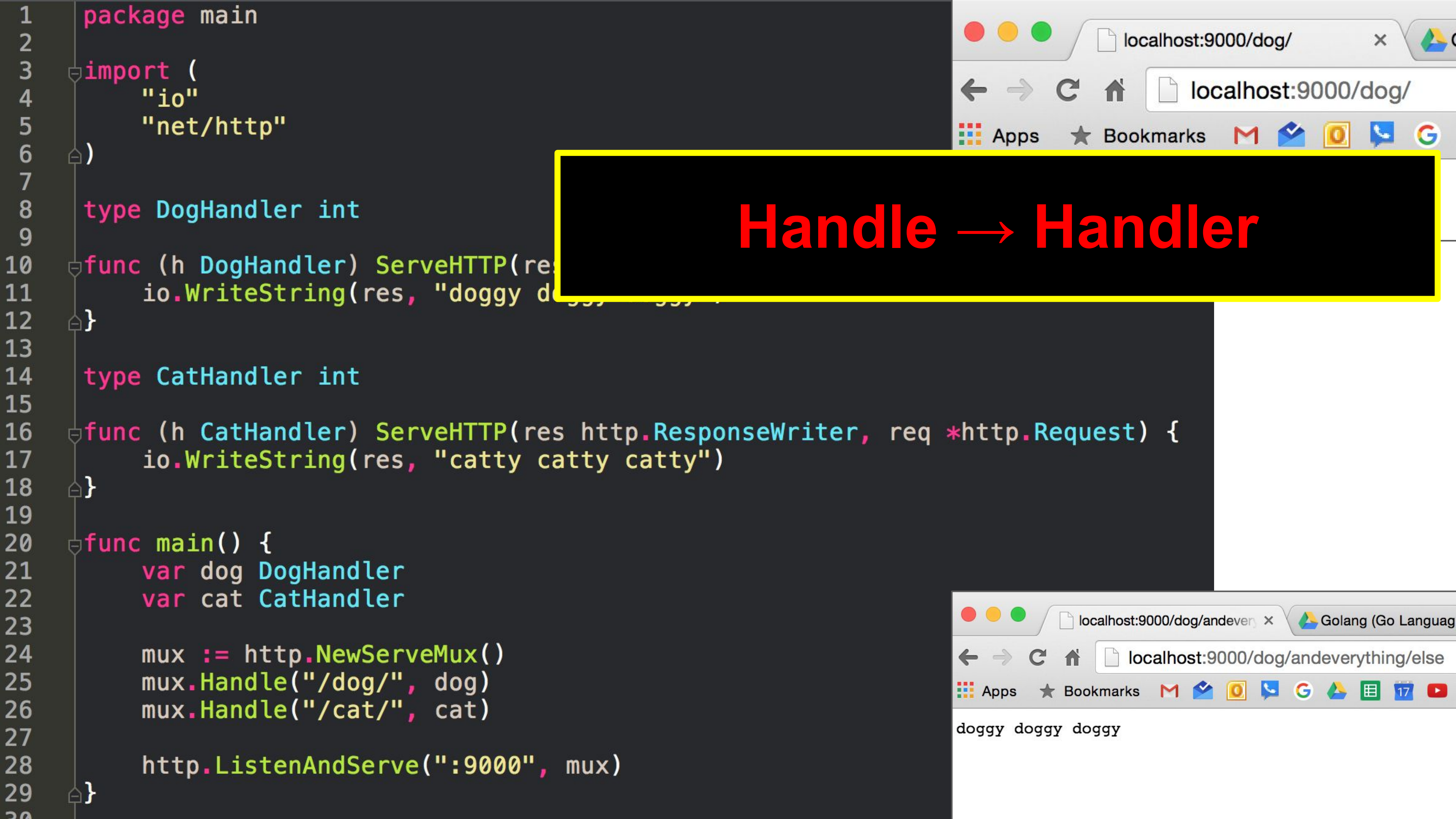
The type HandlerFunc implements the handler interface



```go
package main

import (
	"io"
	"net/http"
)

func main() {

	mux := http.NewServeMux()

	mux.HandleFunc("/", func(res http.ResponseWriter, req *http.Request) {
		io.WriteString(res, "doggy doggy doggy")
	})

	mux.HandleFunc("/cat/", func(res http.ResponseWriter, req *http.Request) {
		io.WriteString(res, "catty catty catty")
	})

	http.ListenAndServe(":9000", mux)
}
```

HandleFunc → HandlerFunc

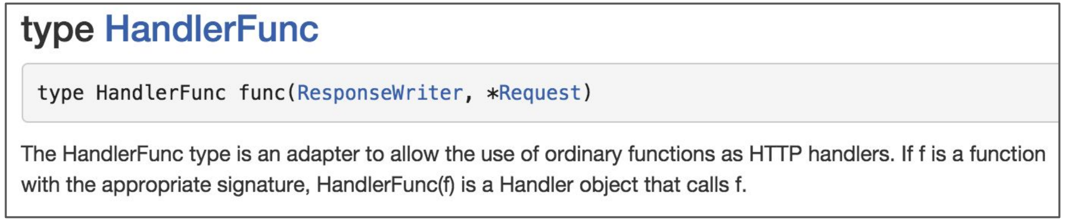
Handle → Handler



```go
package main

import (
	"io"
	"net/http"
)

type DogHandler int

func (h DogHandler) ServeHTTP(re
	io.WriteString(res, "doggy d
}

type CatHandler int

func (h CatHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "catty catty catty")
}

func main() {
	var dog DogHandler
	var cat CatHandler

	mux := http.NewServeMux()
	mux.Handle("/dog/", dog)
	mux.Handle("/cat/", cat)

	http.ListenAndServe(":9000", mux)
}
```

```
doggy doggy doggy
```

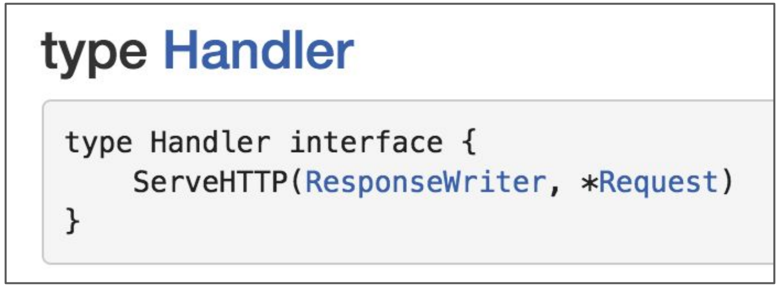
HandleFunc → HandlerFunc


type HandlerFunc

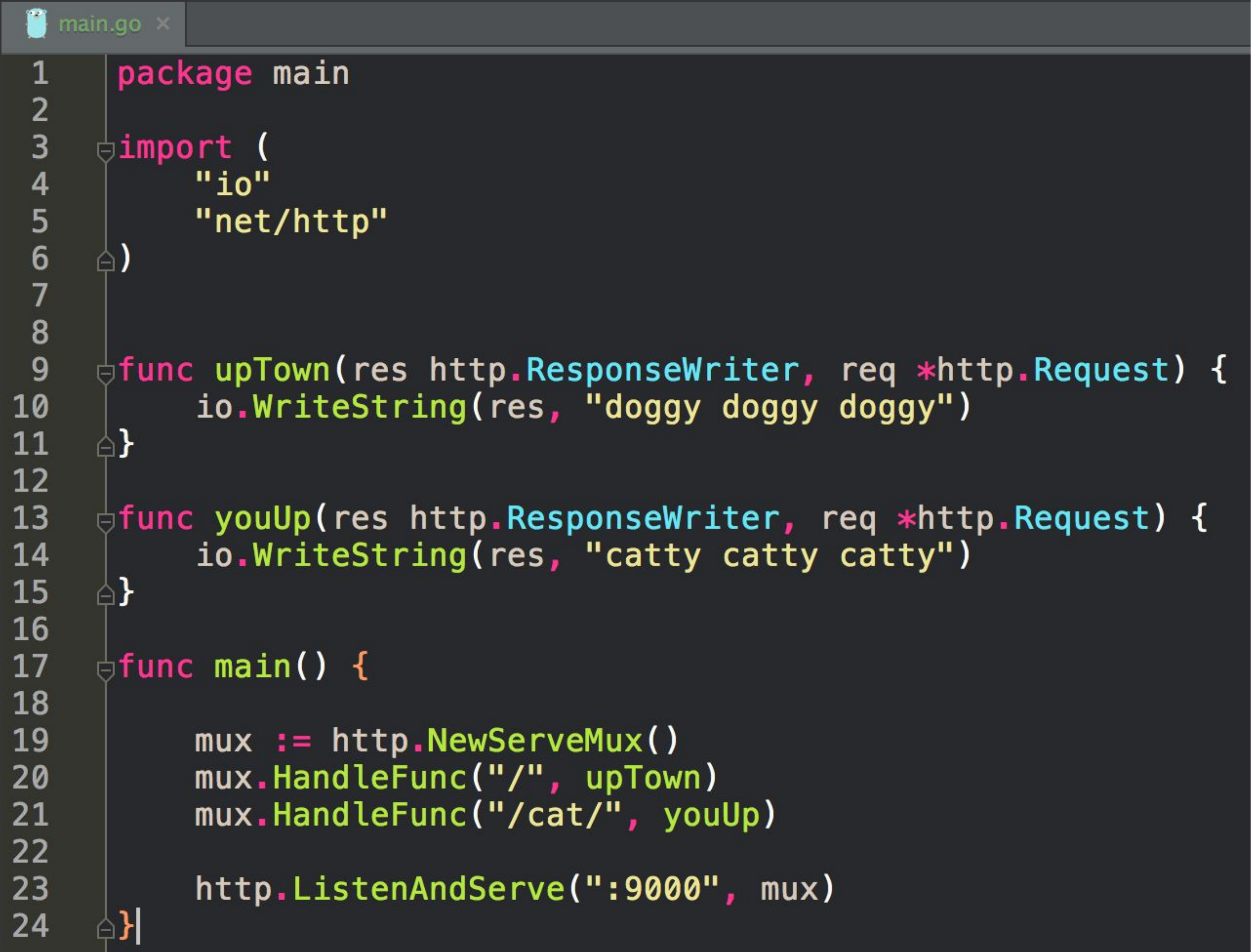
```
type HandlerFunc func(ResponseWriter, *Request)
```

The HandlerFunc type is an adapter to allow the use of ordinary functions as HTTP handlers. If f is a function with the appropriate signature, HandlerFunc(f) is a Handler object that calls f.

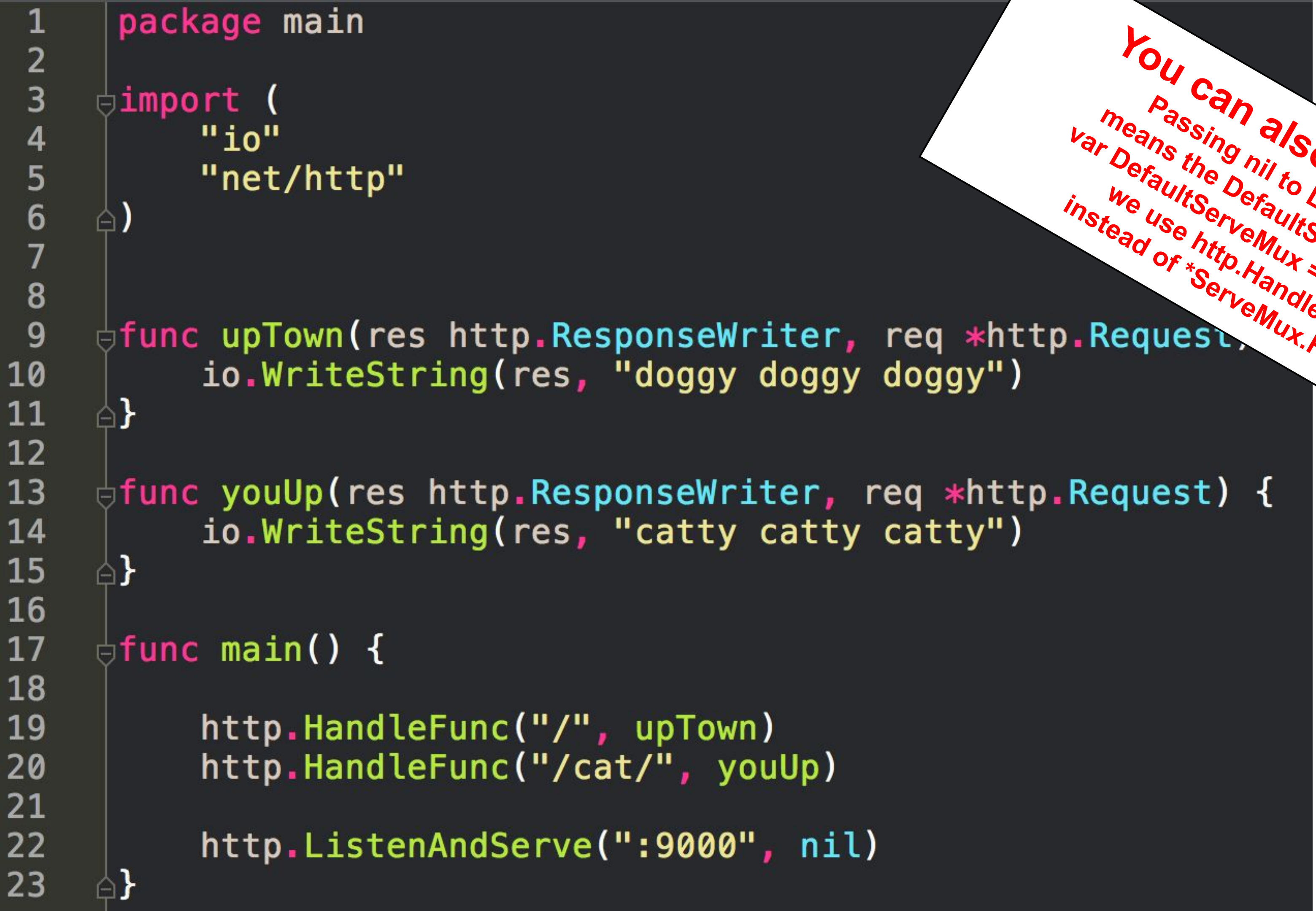
Handle → Handler


type Handler

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


main.go

```go
package main

import (
	"io"
	"net/http"
)


func upTown(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "doggy doggy doggy")
}

func youUp(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "catty catty catty")
}

func main() {

	mux := http.NewServeMux()
	mux.HandleFunc("/", upTown)
	mux.HandleFunc("/cat/", youUp)

	http.ListenAndServe(":9000", mux)
}
```



```go
package main

import (
	"io"
	"net/http"
)


func upTown(res http.ResponseWriter, req *http.Request,
	io.WriteString(res, "doggy doggy doggy")
}

func youUp(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "catty catty catty")
}

func main() {

	http.HandleFunc("/", upTown)
	http.HandleFunc("/cat/", youUp)

	http.ListenAndServe(":9000", nil)
}
```

You can also do it like this

Passing nil to ListenAndServe
means the DefaultServeMux is used
var DefaultServeMux = NewServeMux()
we use http.HandleFunc now
instead of *ServeMux.HandleFunc

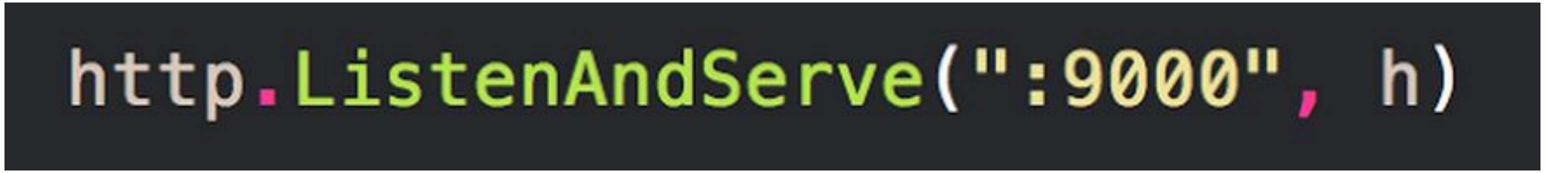
Building Web Apps in Go

- We have a bunch of routes
- For each of those routes
  - we call a HandlerFunc or Handler
    - call code that does stuff

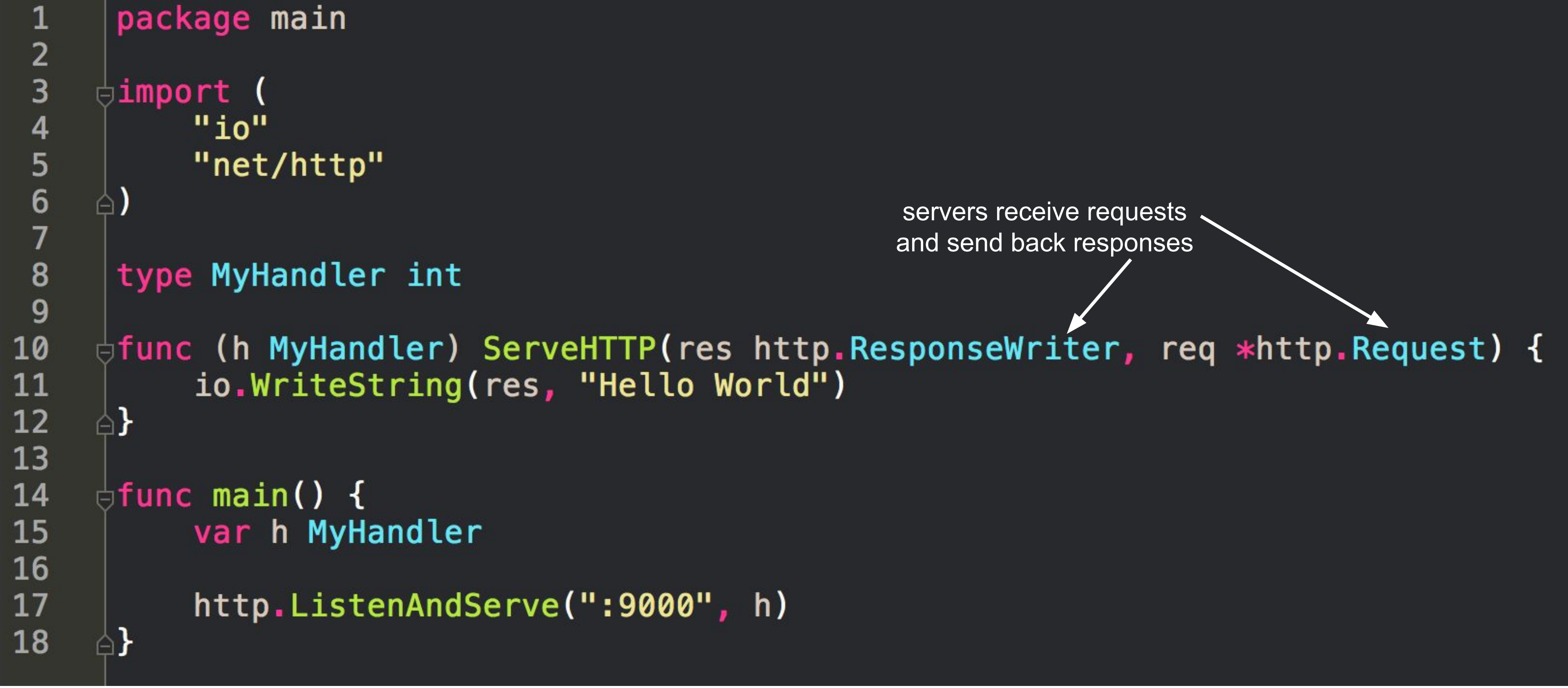
review

http.ListenAndServe

- func ListenAndServe(**addr** string, **handler** Handler) error
- a handler implements the handler interface
    - that means the type has this method:
        - ServeHTTP(ResponseWriter, *Request)



```
http.ListenAndServe(":9000", h)
```

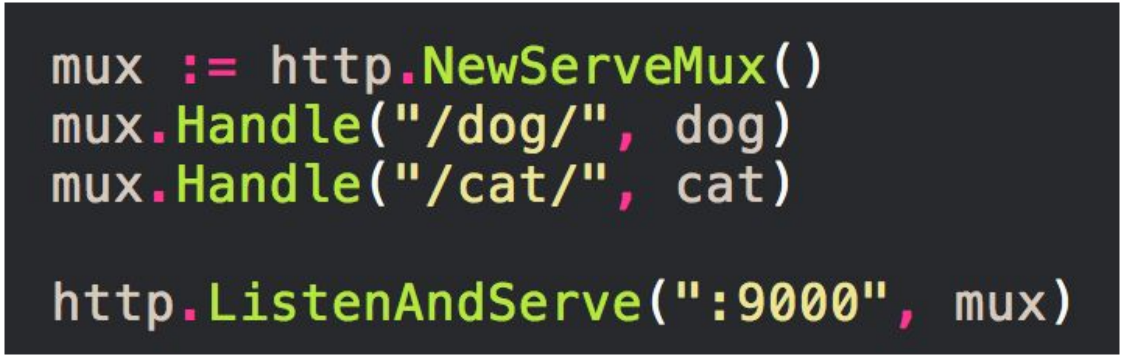


```go
package main

import (
	"io"
	"net/http"
)

type MyHandler int

func (h MyHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "Hello World")
}

func main() {
	var h MyHandler

	http.ListenAndServe(":9000", h)
}
```

servers receive requests and send back responses

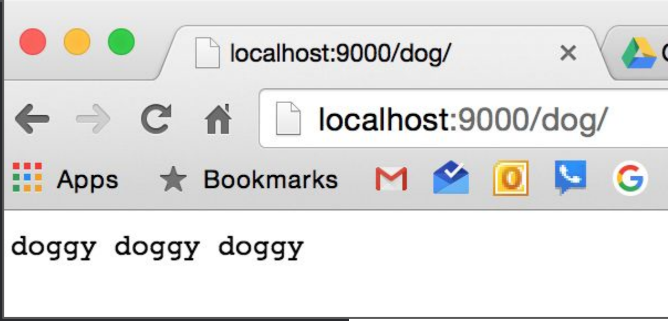
http.NewServeMux()

- ServeMux
  - a multiplexer
  - allows us to do routing
- *ServeMux.Handle
  - func (mux *ServeMux) Handle(pattern string, **handler** Handler)

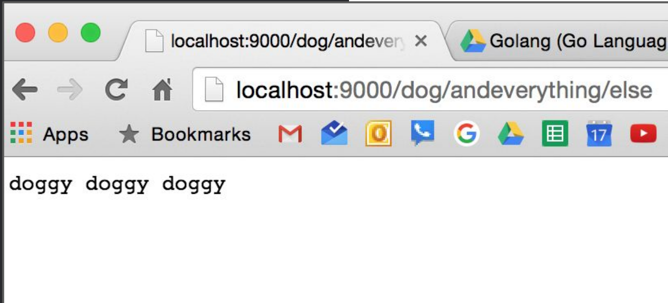


```
mux := http.NewServeMux()
mux.Handle("/dog/", dog)
mux.Handle("/cat/", cat)

http.ListenAndServe(":9000", mux)
```

```go
package main

import (
	"io"
	"net/http"
)

type DogHandler int

func (h DogHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "doggy doggy doggy")
}

type CatHandler int

func (h CatHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "catty catty catty")
}

func main() {
	var dog DogHandler
	var cat CatHandler

	mux := http.NewServeMux()
	mux.Handle("/dog/", dog)
	mux.Handle("/cat/", cat)

	http.ListenAndServe(":9000", mux)
}
```

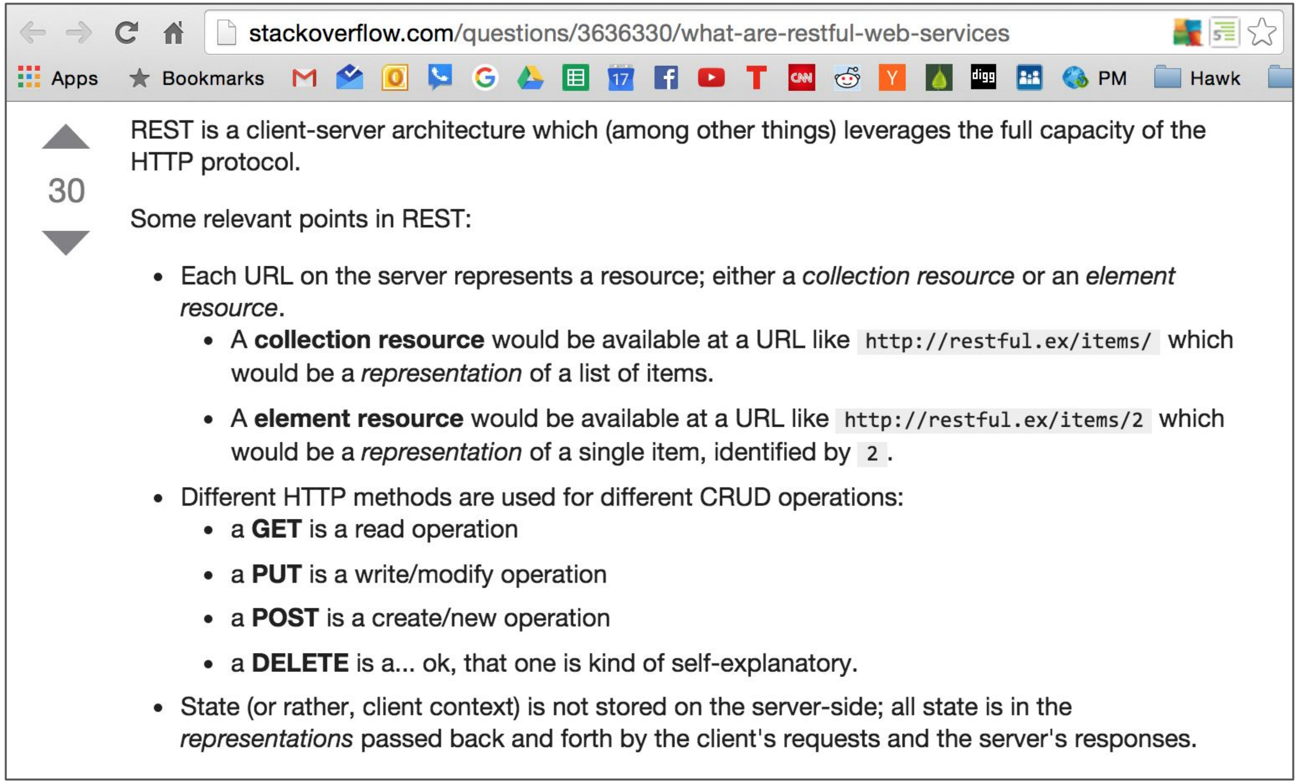


```
doggy doggy doggy
```


localhost:9000/dog/andever
Golang (Go Languag
localhost:9000/dog/andeverything/else
Apps
Bookmarks

```
doggy doggy doggy
```

Restful

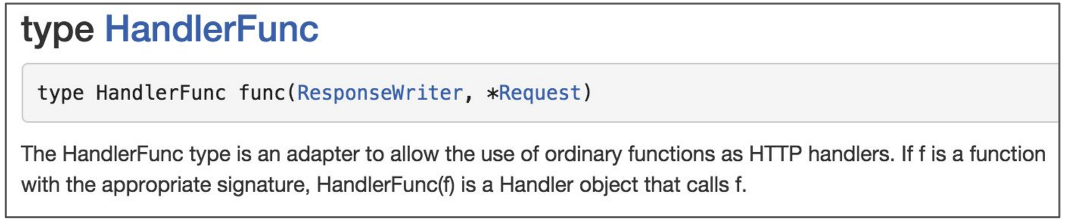

stackoverflow.com/questions/3636330/what-are-restful-web-services
Apps
Bookmarks
PM
Hawk
30

REST is a client-server architecture which (among other things) leverages the full capacity of the HTTP protocol.

Some relevant points in REST:

- Each URL on the server represents a resource; either a *collection resource* or an *element resource*.
  - A **collection resource** would be available at a URL like `http://restful.ex/items/` which would be a *representation* of a list of items.
  - A **element resource** would be available at a URL like `http://restful.ex/items/2` which would be a *representation* of a single item, identified by `2`.
- Different HTTP methods are used for different CRUD operations:
  - a **GET** is a read operation
  - a **PUT** is a write/modify operation
  - a **POST** is a create/new operation
  - a **DELETE** is a... ok, that one is kind of self-explanatory.
- State (or rather, client context) is not stored on the server-side; all state is in the *representations* passed back and forth by the client's requests and the server's responses.

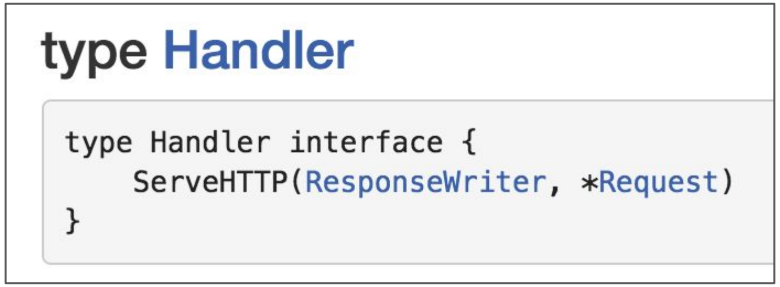
HandleFunc → HandlerFunc


type HandlerFunc

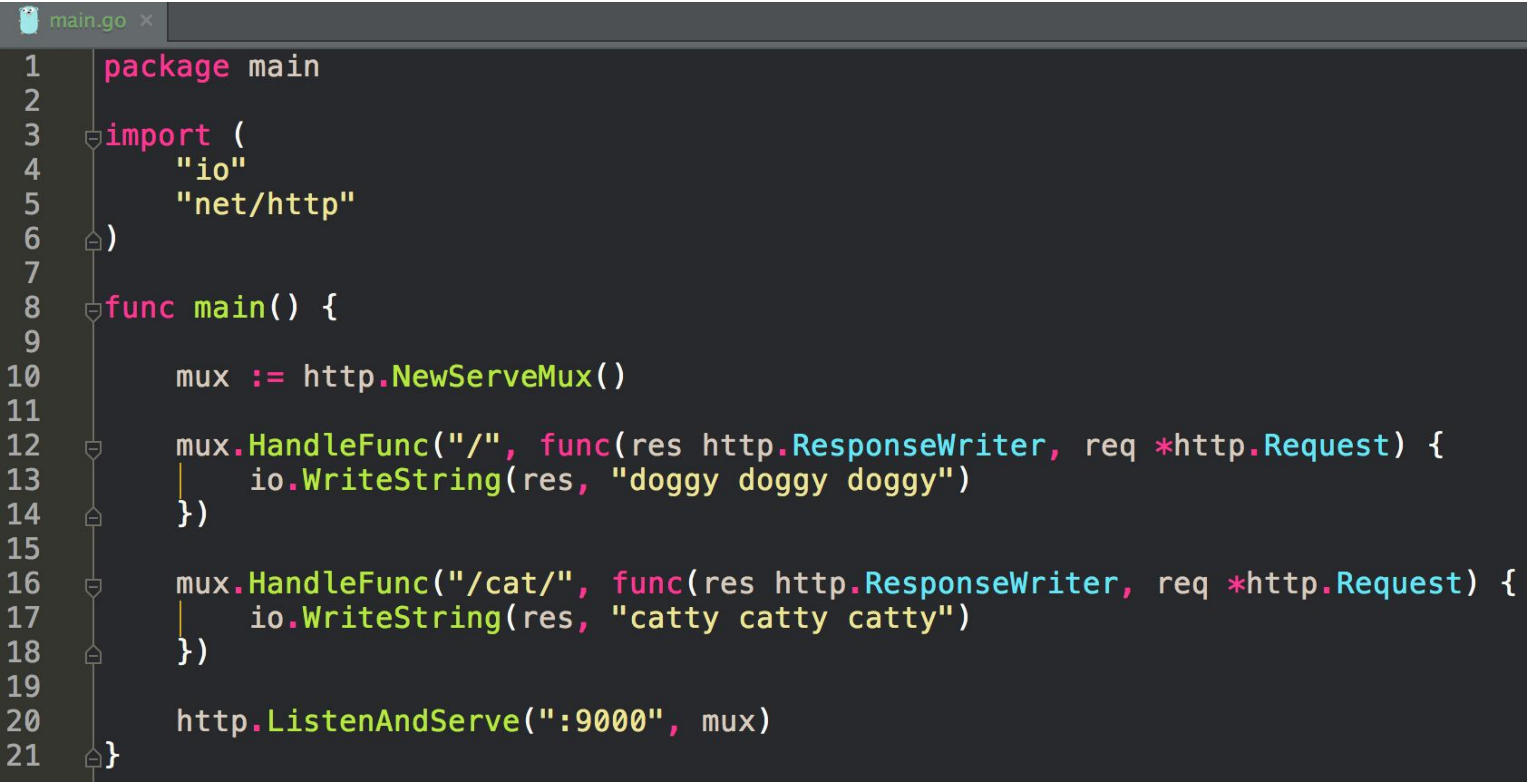
```
type HandlerFunc func(ResponseWriter, *Request)
```

The HandlerFunc type is an adapter to allow the use of ordinary functions as HTTP handlers. If f is a function with the appropriate signature, HandlerFunc(f) is a Handler object that calls f.

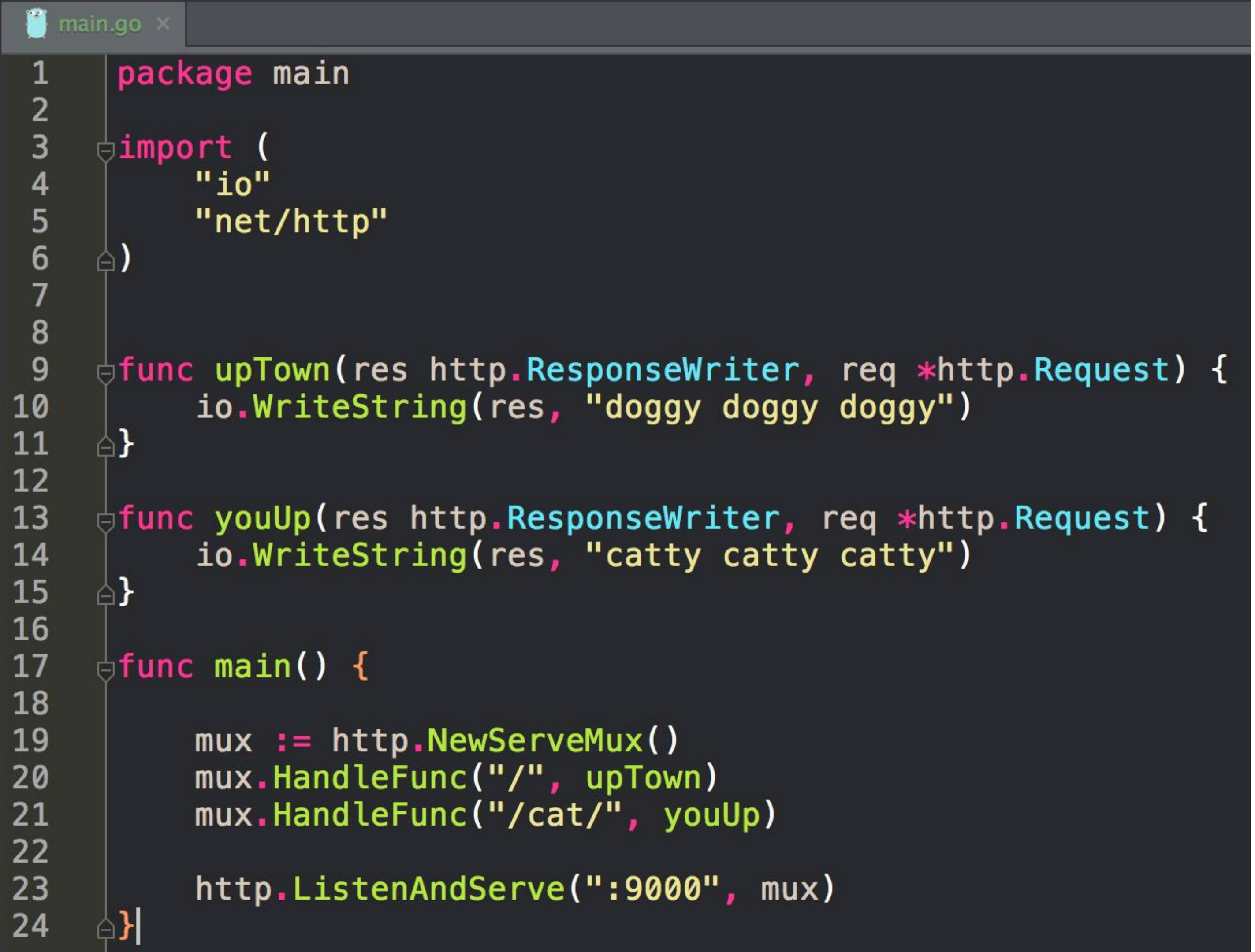
Handle → Handler


type Handler

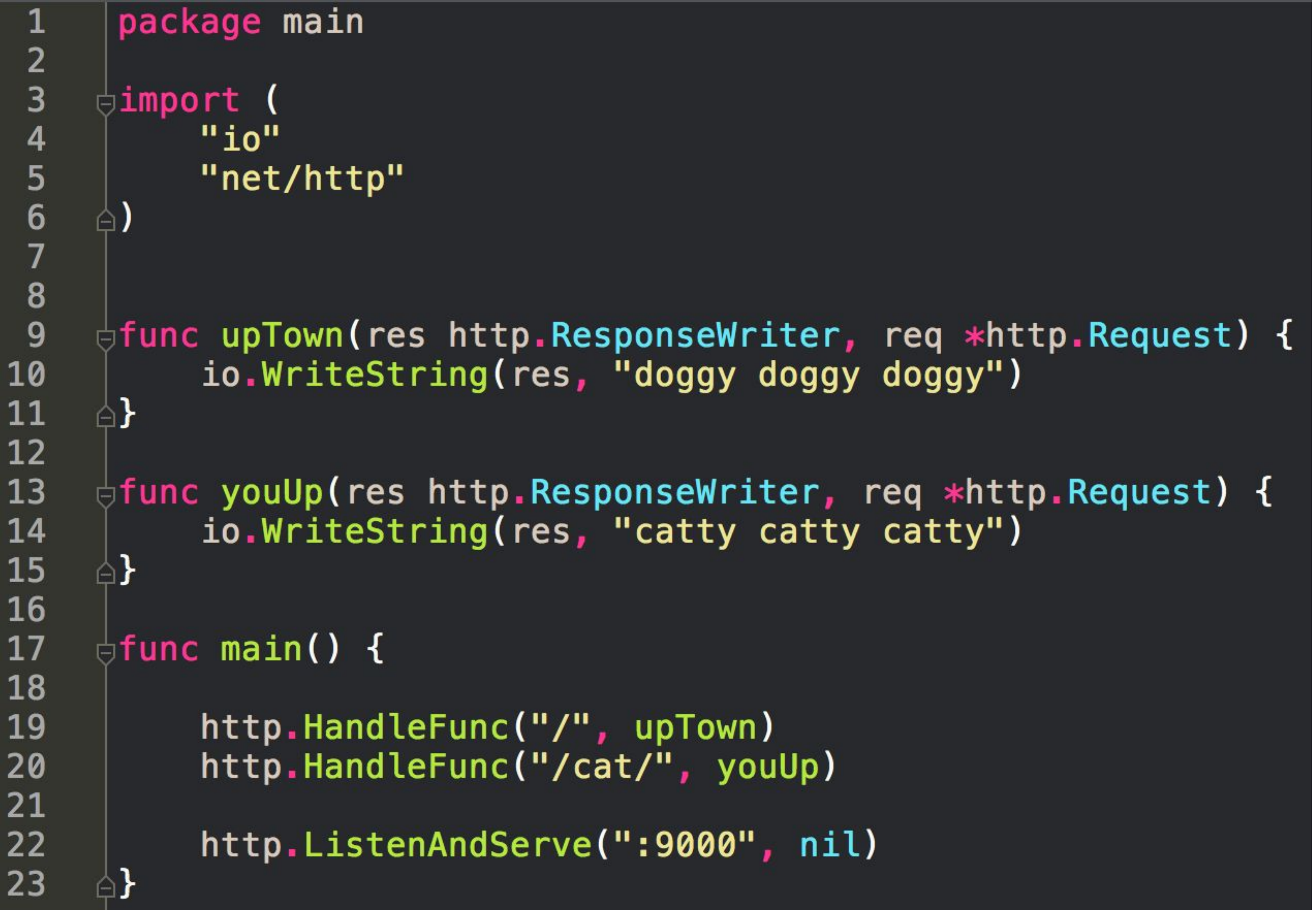
```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```


main.go

```go
package main

import (
	"io"
	"net/http"
)

func main() {

	mux := http.NewServeMux()

	mux.HandleFunc("/", func(res http.ResponseWriter, req *http.Request) {
		io.WriteString(res, "doggy doggy doggy")
	})

	mux.HandleFunc("/cat/", func(res http.ResponseWriter, req *http.Request) {
		io.WriteString(res, "catty catty catty")
	})

	http.ListenAndServe(":9000", mux)
}
```



```go
package main

import (
	"io"
	"net/http"
)


func upTown(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "doggy doggy doggy")
}

func youUp(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "catty catty catty")
}

func main() {

	mux := http.NewServeMux()
	mux.HandleFunc("/", upTown)
	mux.HandleFunc("/cat/", youUp)

	http.ListenAndServe(":9000", mux)
}
```



```go
package main

import (
	"io"
	"net/http"
)


func upTown(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "doggy doggy doggy")
}

func youUp(res http.ResponseWriter, req *http.Request) {
	io.WriteString(res, "catty catty catty")
}

func main() {

	http.HandleFunc("/", upTown)
	http.HandleFunc("/cat/", youUp)

	http.ListenAndServe(":9000", nil)
}
```