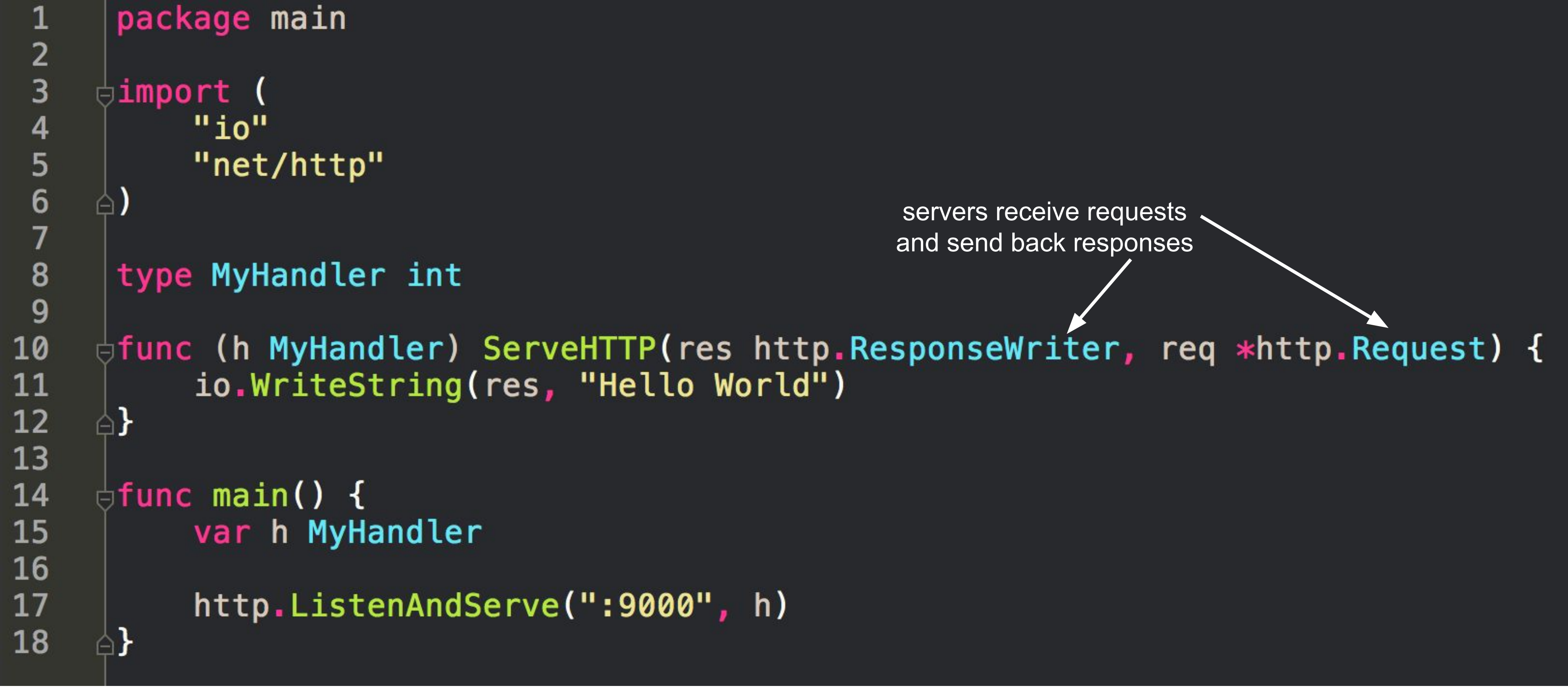
HTTP Server

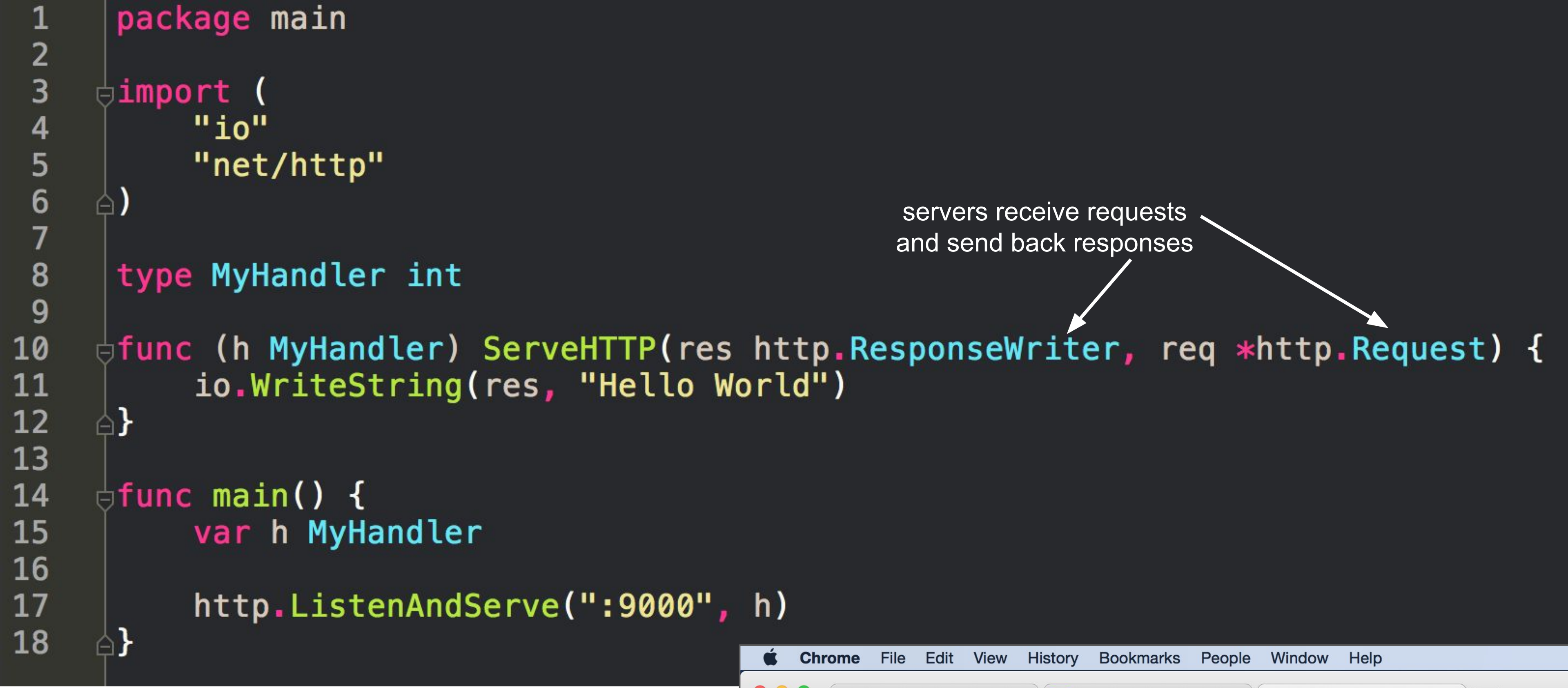
as we saw previously
parsing html manually is brutal
package net/http is here to make life easier

HTTP server

we can use package net/http
to create an HTTP server
(remember, it’s still on top of TCP)

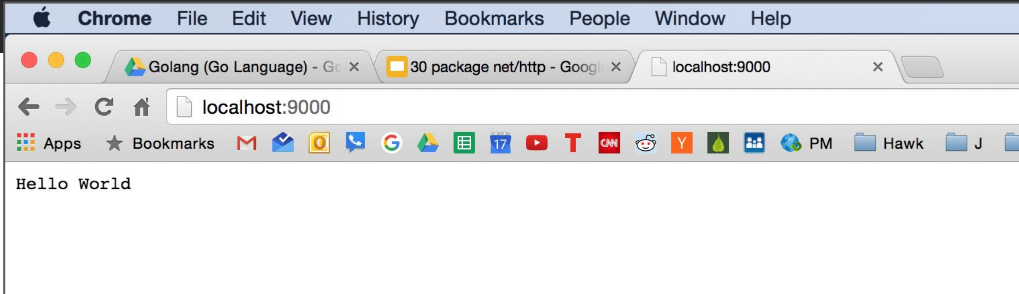


```go
package main

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)
}
```

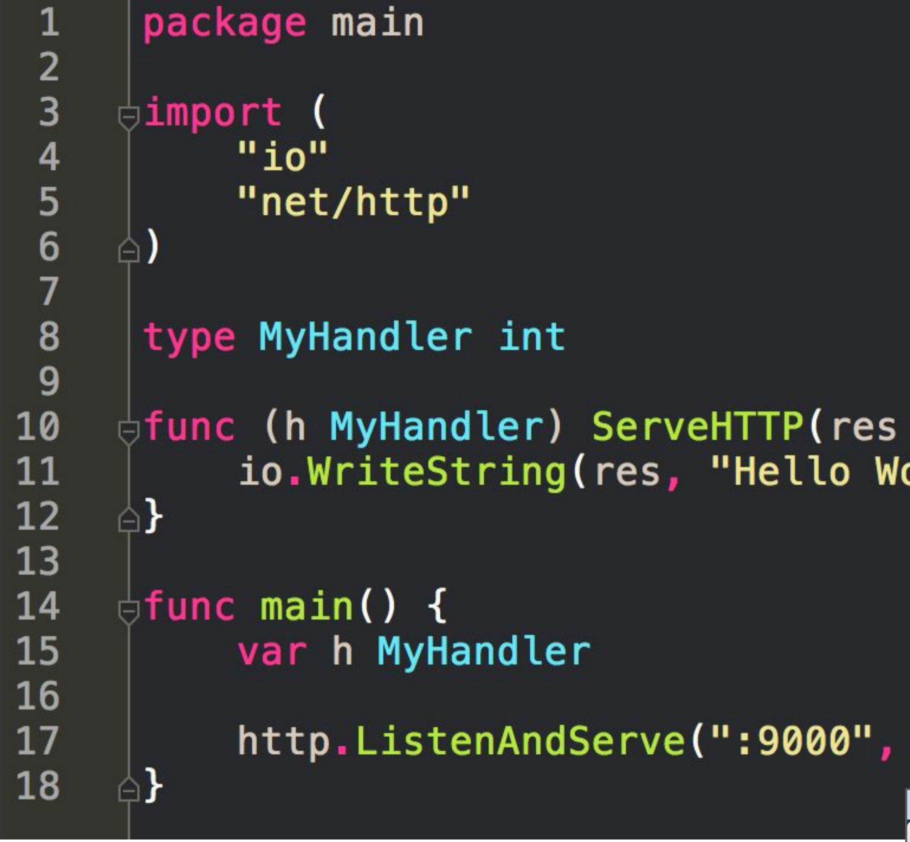


```go
package main

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)
}
```

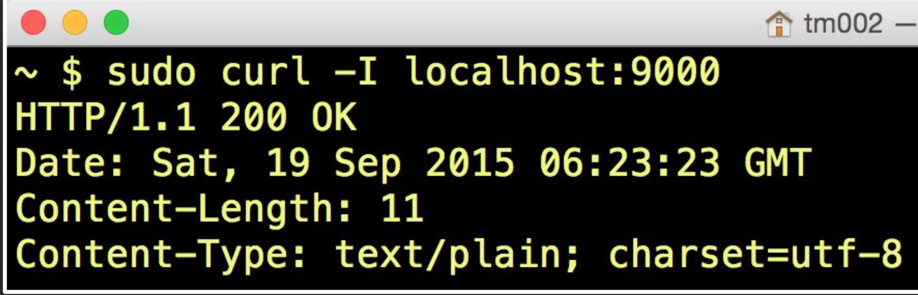
servers receive requests
and send back responses



```
Hello World
```

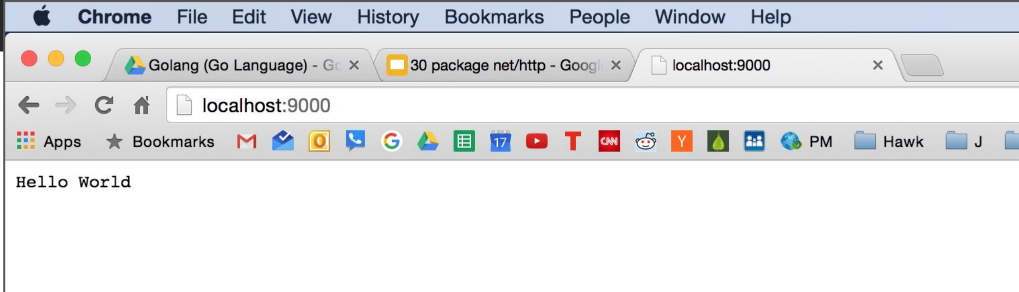


```go
package main

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)
}
```

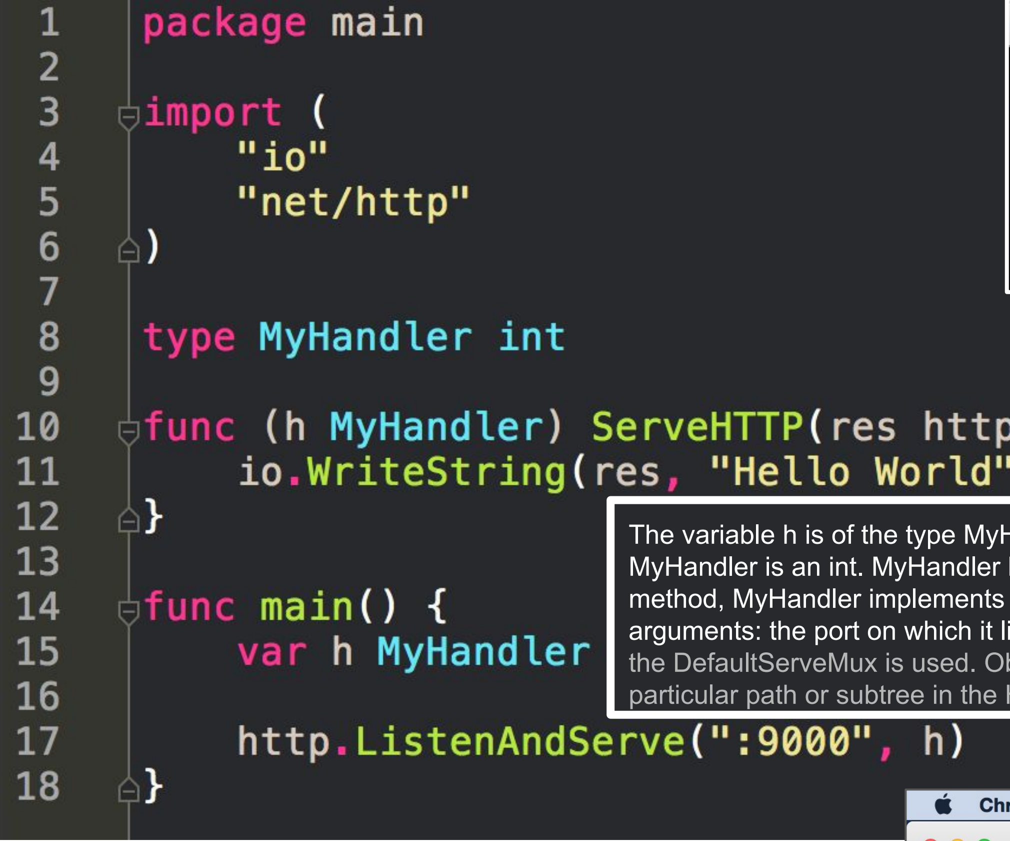


```
~ $ sudo curl -I localhost:9000
HTTP/1.1 200 OK
Date: Sat, 19 Sep 2015 06:23:23 GMT
Content-Length: 11
Content-Type: text/plain; charset=utf-8
```

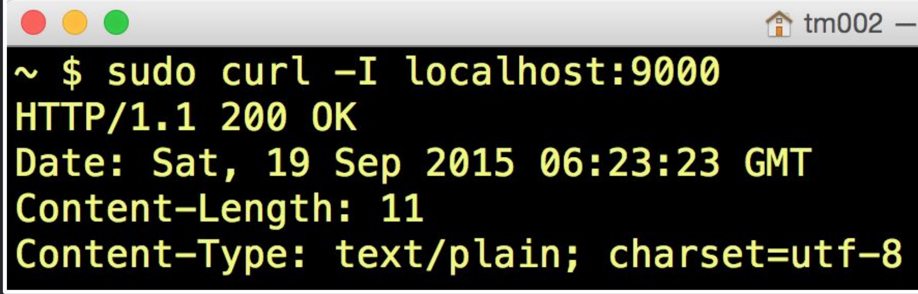
servers receive requests
and send back responses


Chrome File Edit View History Bookmarks People Window Help
Golang (Go Language) - Go
30 package net/http - Googl
localhost:9000
Apps
Bookmarks
PM
Hawk
J

```
Hello World
```



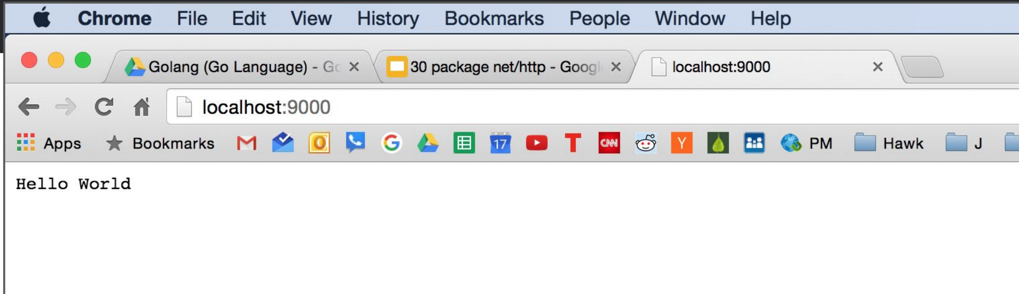
```go
1  package main
2
3  import (
4  	"io"
5  	"net/http"
6  )
7
8  type MyHandler int
9
10 func (h MyHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
11 	io.WriteString(res, "Hello World")
12 }
13
14 func main() {
15 	var h MyHandler
16
17 	http.ListenAndServe(":9000", h)
18 }
```



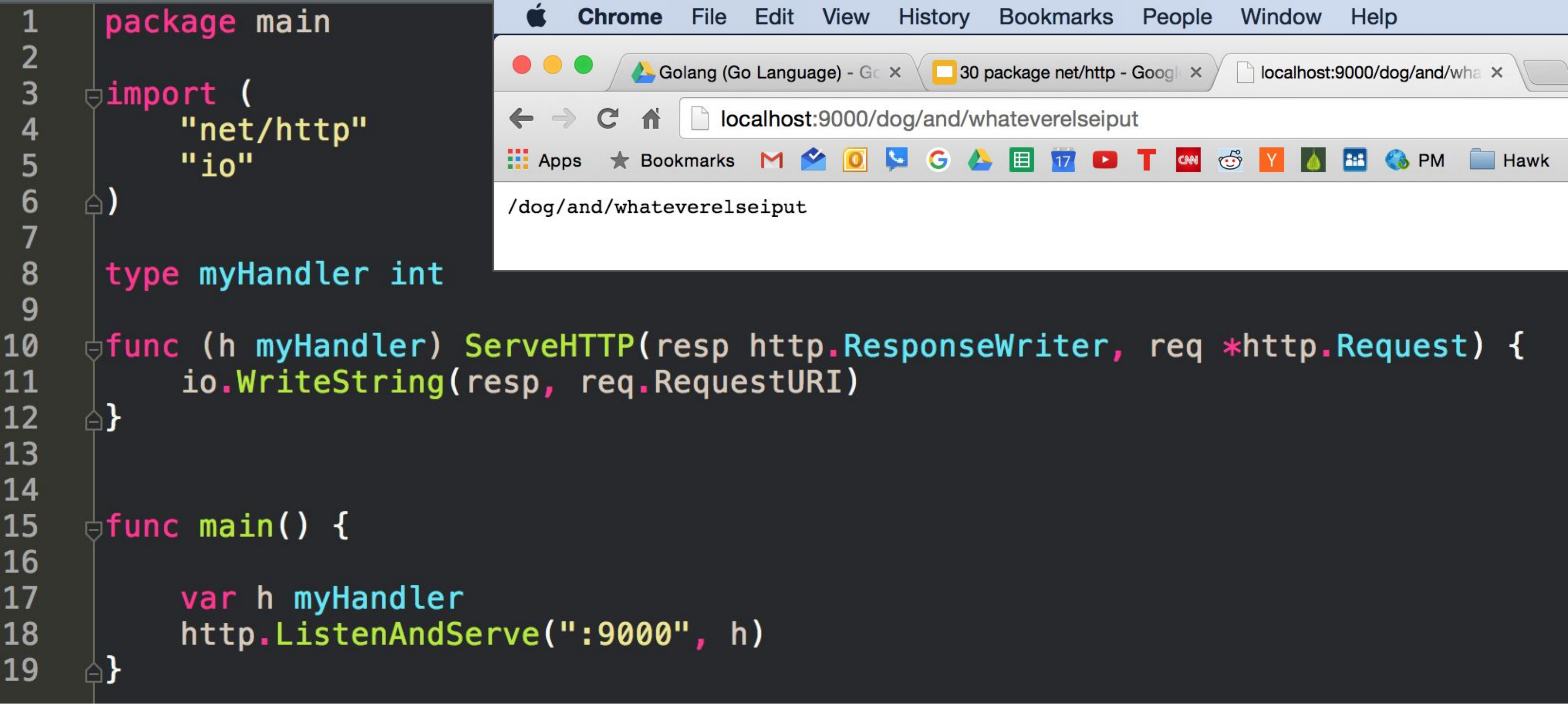
```
~ $ sudo curl -I localhost:9000
HTTP/1.1 200 OK
Date: Sat, 19 Sep 2015 06:23:23 GMT
Content-Length: 11
Content-Type: text/plain; charset=utf-8
```

servers receive requests
and send back responses

The variable h is of the type MyHandler. MyHandler is a user-defined type. The underlying type of MyHandler is an int. MyHandler has a method called ServeHTTP. Because MyHandler has the ServeHTTP method, MyHandler implements the handler interface. On line 17, http.ListenAndServe takes two arguments: the port on which it listens and a handler. If the second argument to ListenAndServe is nil, then the DefaultServeMux is used. Objects implementing the Handler interface can be registered to serve a particular path or subtree in the HTTP server. Handle registers the handler for the given pattern.


Chrome File Edit View History Bookmarks People Window Help
Golang (Go Language) - Go
30 package net/http - Googl
localhost:9000
localhost:9000
Apps
Bookmarks
PM
Hawk
J

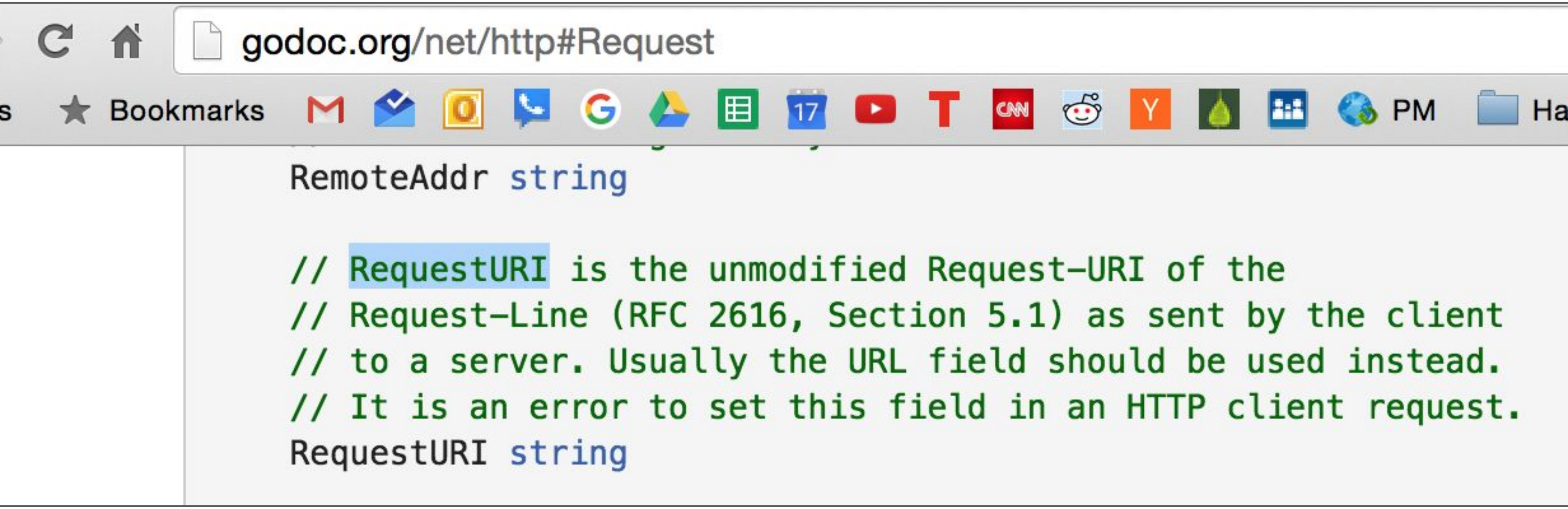
```
Hello World
```


Chrome
File
Edit
View
History
Bookmarks
People
Window
Help
Golang (Go Language) - Gc
30 package net/http - Googl
localhost:9000/dog/and/wha
Apps
Bookmarks
PM
Hawk

```go
package main

import (
	"net/http"
	"io"
)

type myHandler int

func (h myHandler) ServeHTTP(resp http.ResponseWriter, req *http.Request) {
	io.WriteString(resp, req.RequestURI)
}


func main() {

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

localhost:9000/dog/and/whateverelseiput

```
/dog/and/whateverelseiput
```



godoc.org/net/http#Request

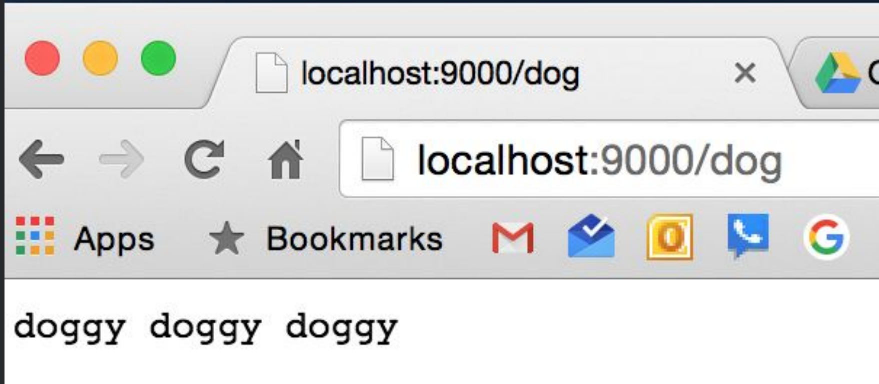
Bookmarks PM

```
RemoteAddr string

// RequestURI is the unmodified Request-URI of the
// Request-Line (RFC 2616, Section 5.1) as sent by the client
// to a server. Usually the URL field should be used instead.
// It is an error to set this field in an HTTP client request.
RequestURI string
```

http://godoc.org/net/http#Request

look at the fields and methods for type Request

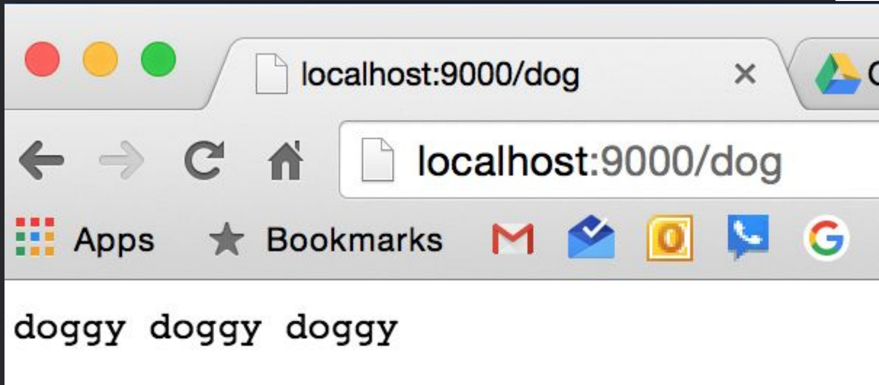
```go
package main

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
```

```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)
}
```

Important thing to keep in mind: there is no correspondence to the URL and anything on the computer -- not a file, not a path -- it's whatever you want it to mean. The definition, the meaning of /cat or /dog is in the code. There is no "cat.php" or "dog.asp" file on my computer. Routing is completely defined by me.

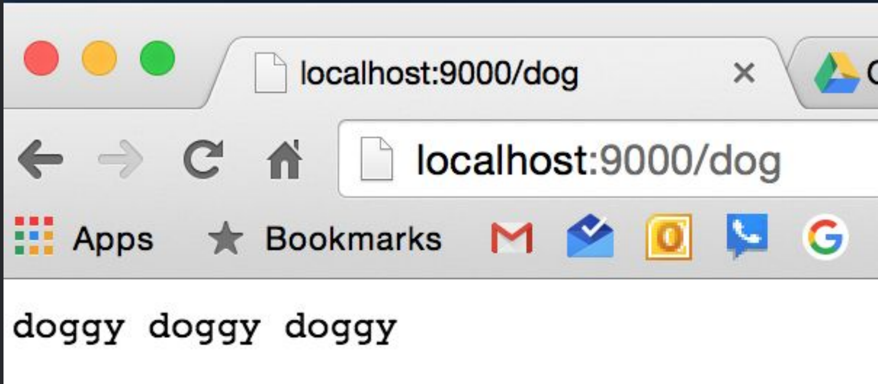


```
doggy doggy doggy
```

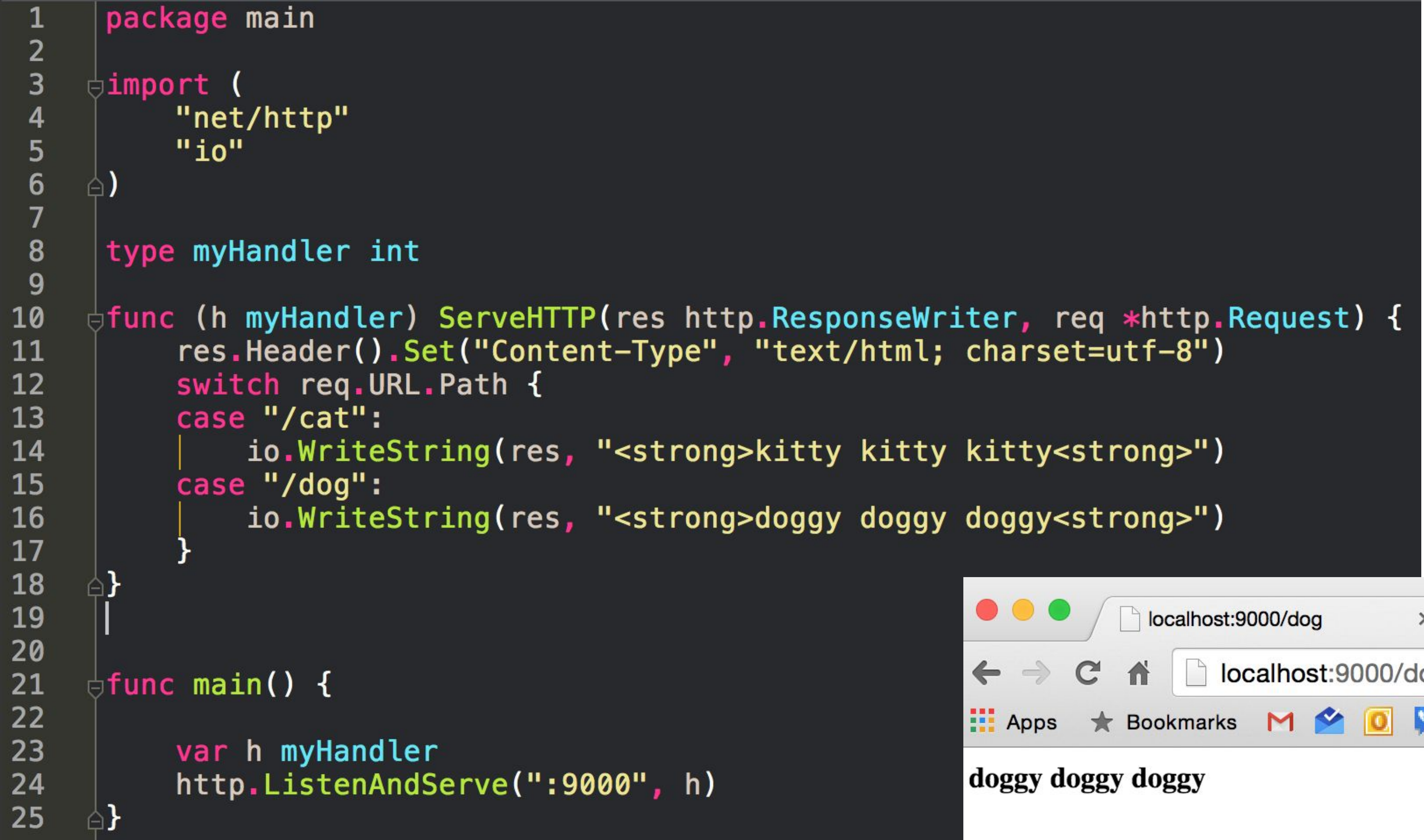
Important thing to keep in mind: there is no correspondence to the URL and anything on the computer -- not a file, not a path -- it’s whatever you want it to mean. The definition, the meaning of /cat or /dog is in the code. There is no “cat.php” or “dog.asp” file on my computer. Routing is completely defined by me.

```go
package main

import (
	"net/http"
	"io"
)

type myHandler int

func
st) {



		io.WriteString(res, "doggy doggy doggy")
	}
}


func main() {

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

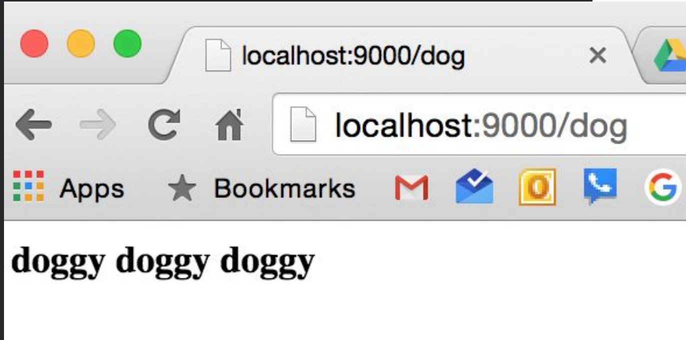
It’s up to you to define the meaning of your URLs



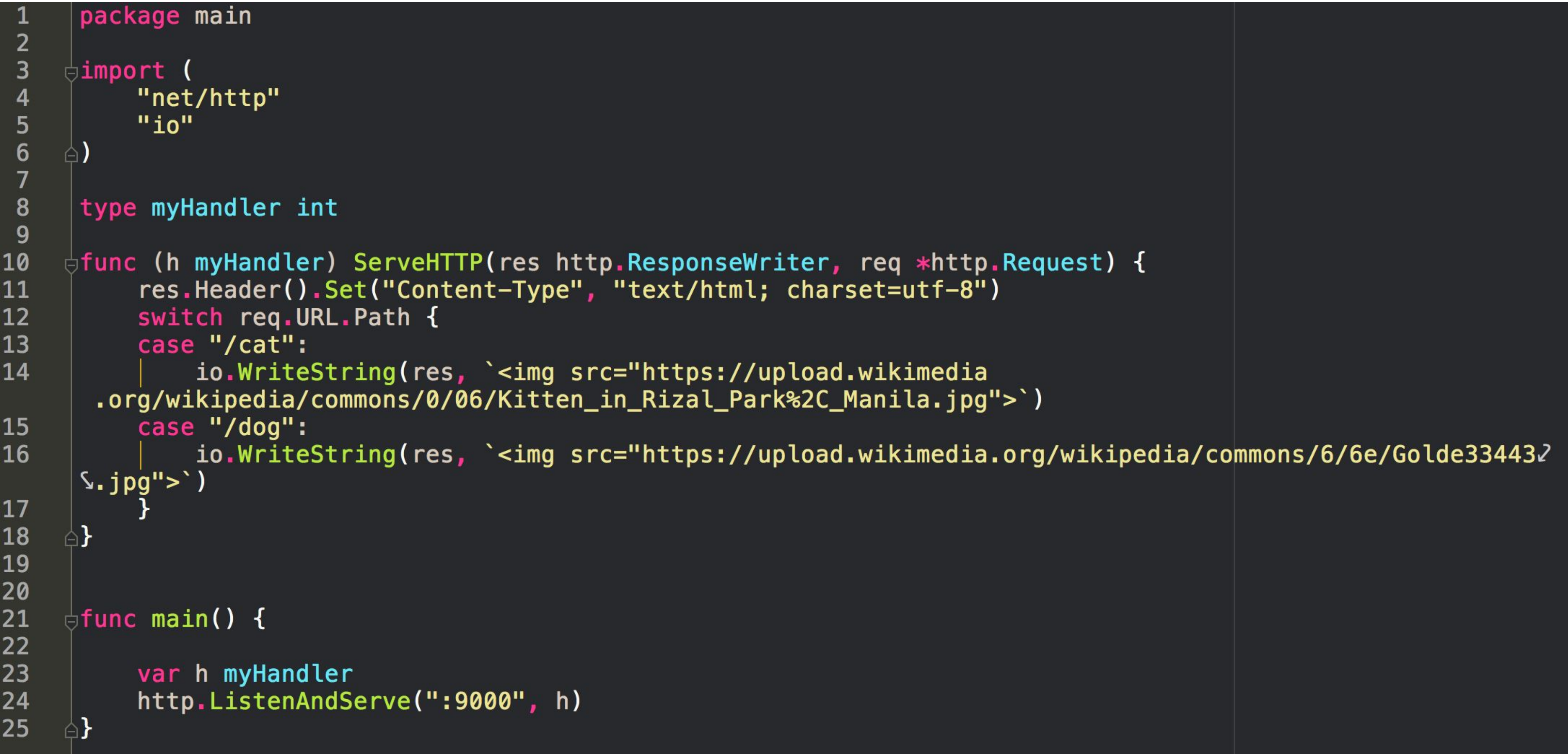
```
doggy doggy doggy
```



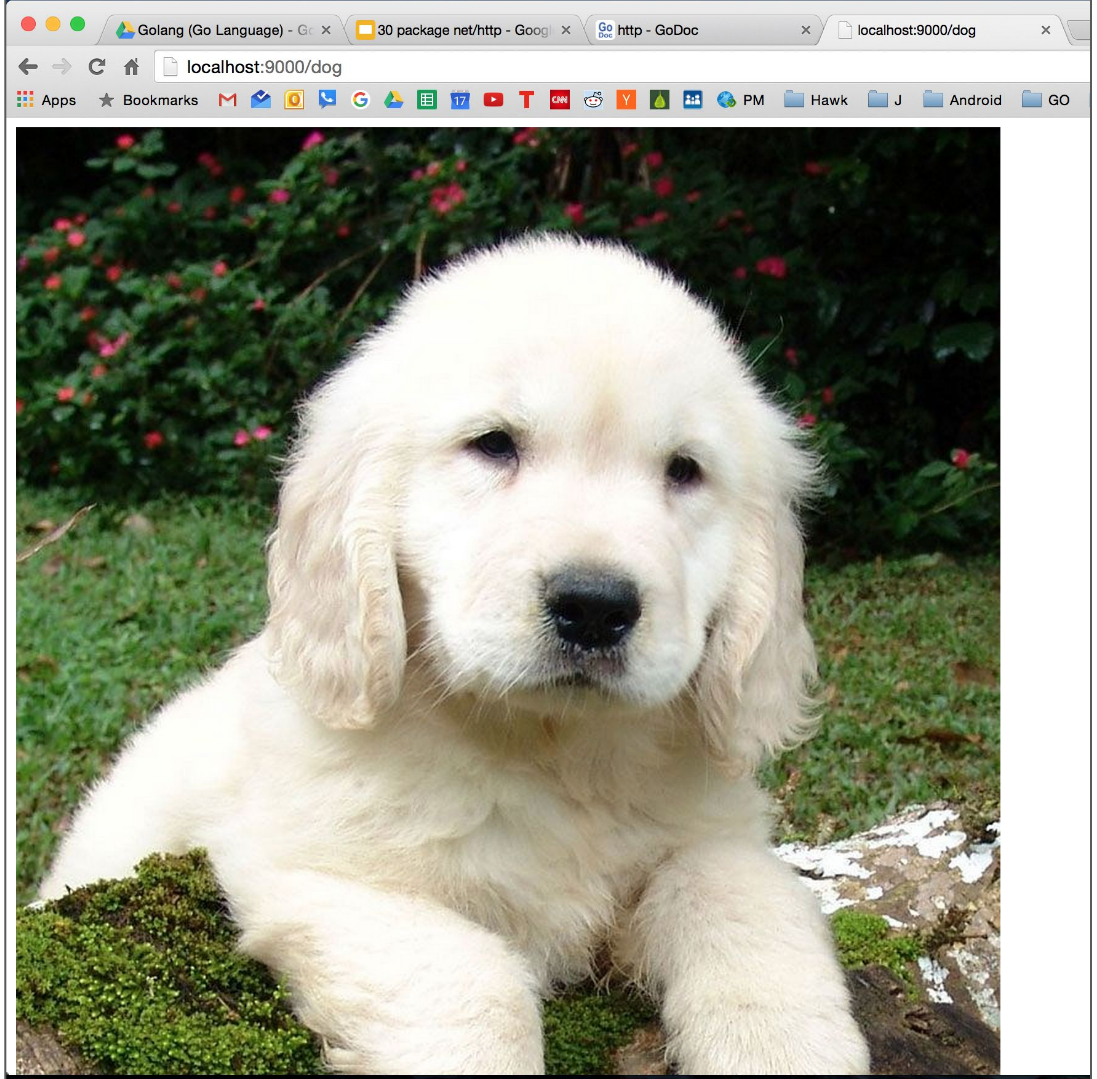
```go
package main

import (
	"net/http"
	"io"
)

type myHandler int

func (h myHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
	res.Header().Set("Content-Type", "text/html; charset=utf-8")
	switch req.URL.Path {
	case "/cat":
		io.WriteString(res, "<strong>kitty kitty kitty<strong>")
	case "/dog":
		io.WriteString(res, "<strong>doggy doggy doggy<strong>")
	}
}


func main() {

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

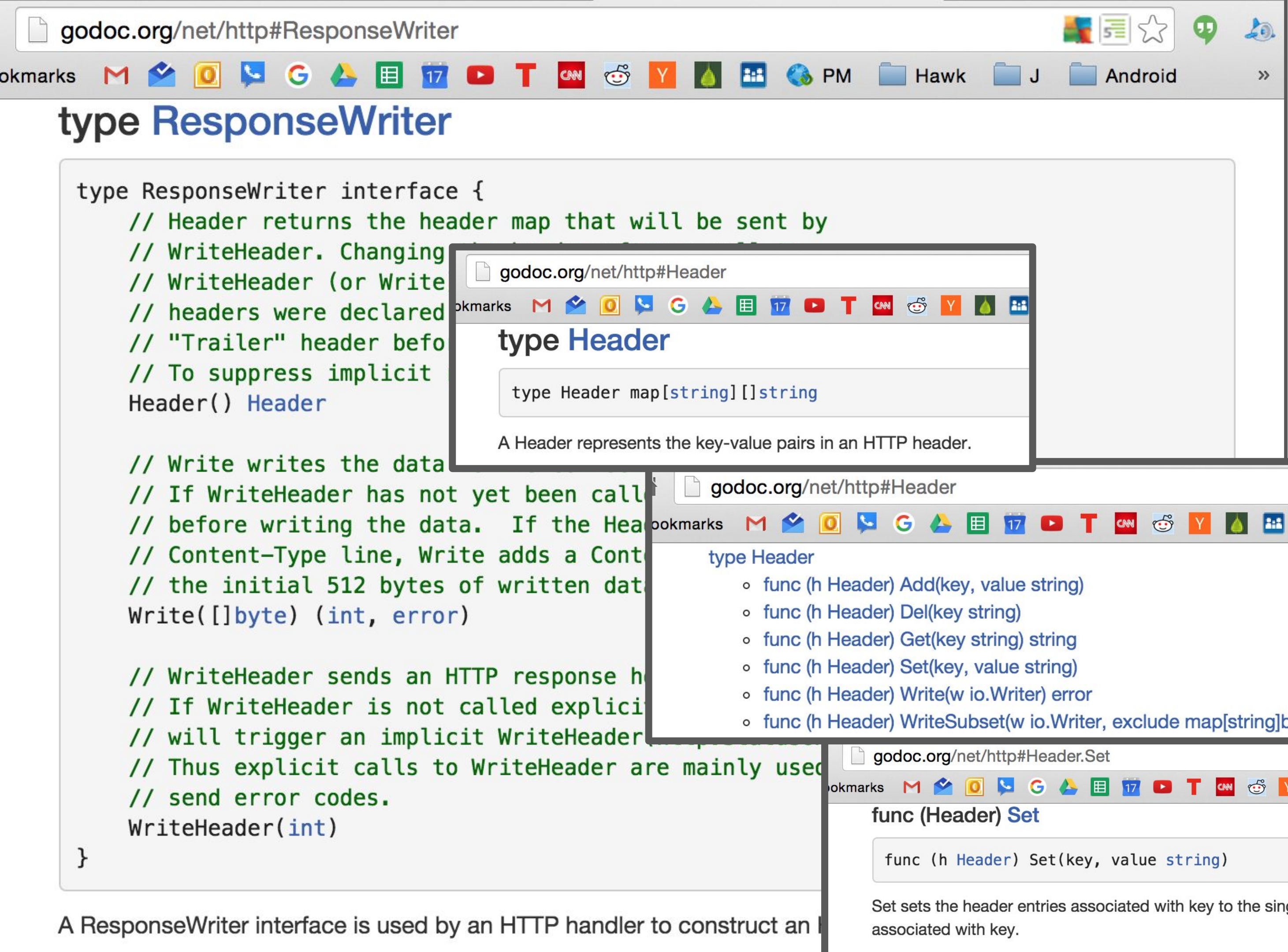


doggy doggy doggy



```go
package main

import (
    "net/http"
    "io"
)

type myHandler int

func (h myHandler) ServeHTTP(res http.ResponseWriter, req *http.Request) {
    res.Header().Set("Content-Type", "text/html; charset=utf-8")
    switch req.URL.Path {
    case "/cat":
        io.WriteString(res, `<img src="https://upload.wikimedia.org/wikipedia/commons/0/06/Kitten_in_Rizal_Park%2C_Manila.jpg">`)
    case "/dog":
        io.WriteString(res, `<img src="https://upload.wikimedia.org/wikipedia/commons/6/6e/Golde33443.jpg">`)
    }
}


func main() {

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

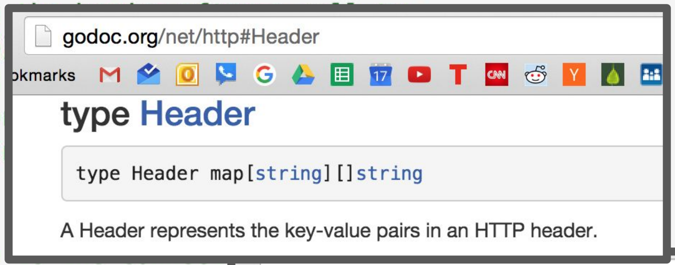

Golang (Go Language) - Go
30 package net/http - Googl
http - GoDoc
localhost:9000/dog
localhost:9000/dog
Apps
Bookmarks
PM
Hawk
J
Android
GO



type ResponseWriter

```
type ResponseWriter interface {
    // Header returns the header map that will be sent by
    // WriteHeader. Changing
    // WriteHeader (or Write
    // headers were declared
    // "Trailer" header befo
    // To suppress implicit
    Header() Header

    // Write writes the data
    // If WriteHeader has not yet been call
    // before writing the data.  If the Hea
    // Content-Type line, Write adds a Cont
    // the initial 512 bytes of written dat
    Write([]byte) (int, error)

    // WriteHeader sends an HTTP response h
    // If WriteHeader is not called explici
    // will trigger an implicit WriteHeader
    // Thus explicit calls to WriteHeader are mainly use
    // send error codes.
    WriteHeader(int)
}
```

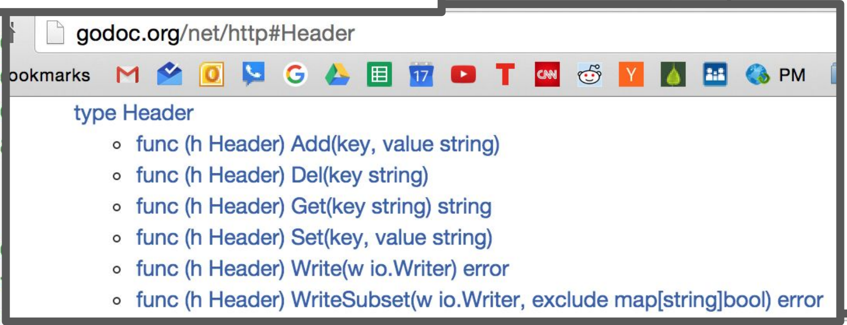
A ResponseWriter interface is used by an HTTP handler to construct an



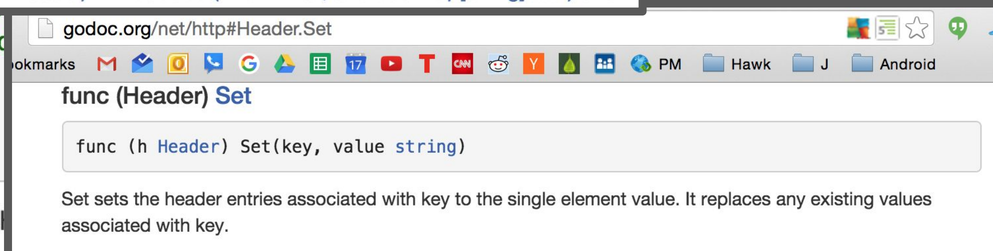
type Header

```
type Header map[string][]string
```

A Header represents the key-value pairs in an HTTP header.



type Header
- func (h Header) Add(key, value string)
- func (h Header) Del(key string)
- func (h Header) Get(key string) string
- func (h Header) Set(key, value string)
- func (h Header) Write(w io.Writer) error
- func (h Header) WriteSubset(w io.Writer, exclude map[string]bool) error



func (Header) Set

```
func (h Header) Set(key, value string)
```

Set sets the header entries associated with key to the single element value. It replaces any existing values associated with key.

http://godoc.org/net/http#ResponseWriter

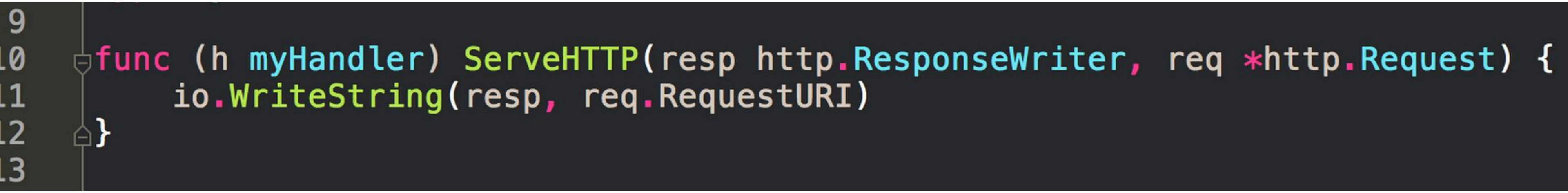
notice that **ResponseWriter** is an **interface**
whereas type **Request** was a **struct**

Remember that it’s this simple

servers receive requests
and send back responses

Remember that it's this simple

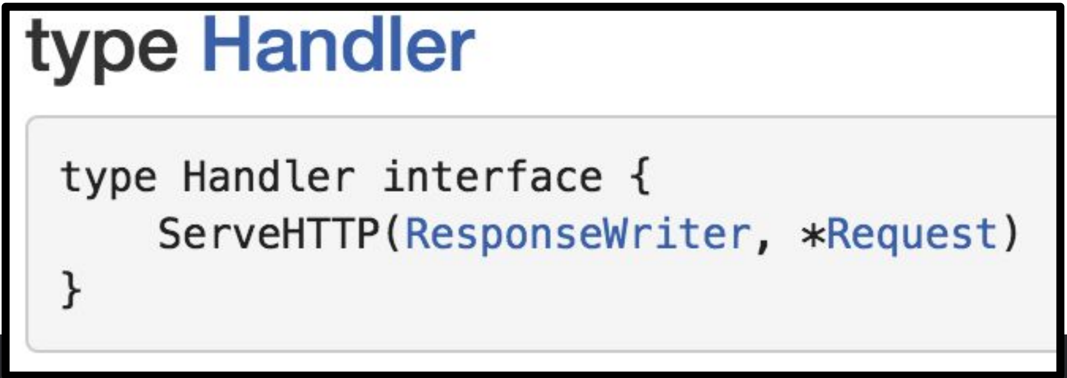
servers receive requests
and send back responses



```go
func (h myHandler) ServeHTTP(resp http.ResponseWriter, req *http.Request) {
	io.WriteString(resp, req.RequestURI)
}
```

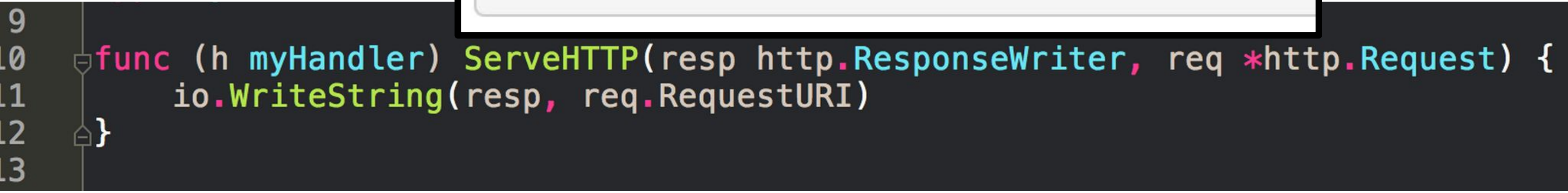
Remember that it's this simple

servers receive requests
and send back responses



type Handler

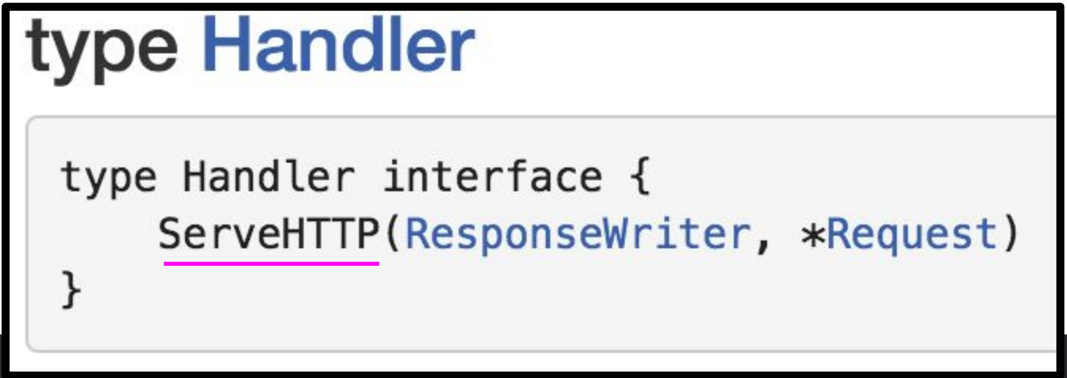
```
type Handler interface {
        ServeHTTP(ResponseWriter, *Request)
}
```



```
func (h myHandler) ServeHTTP(resp http.ResponseWriter, req *http.Request) {
    io.WriteString(resp, req.RequestURI)
}
```

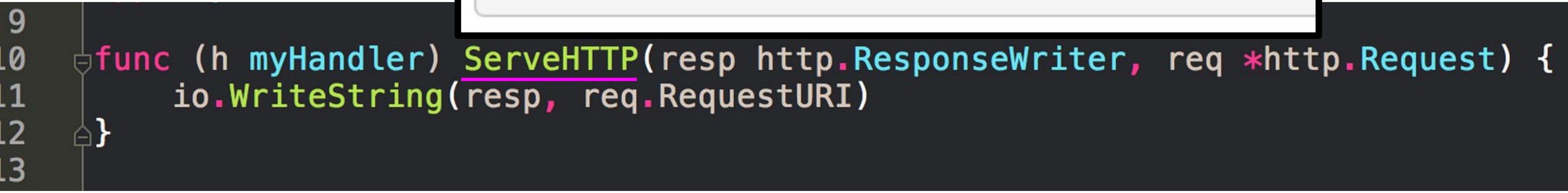
Remember that it’s this simple

servers receive requests
and send back responses



type Handler

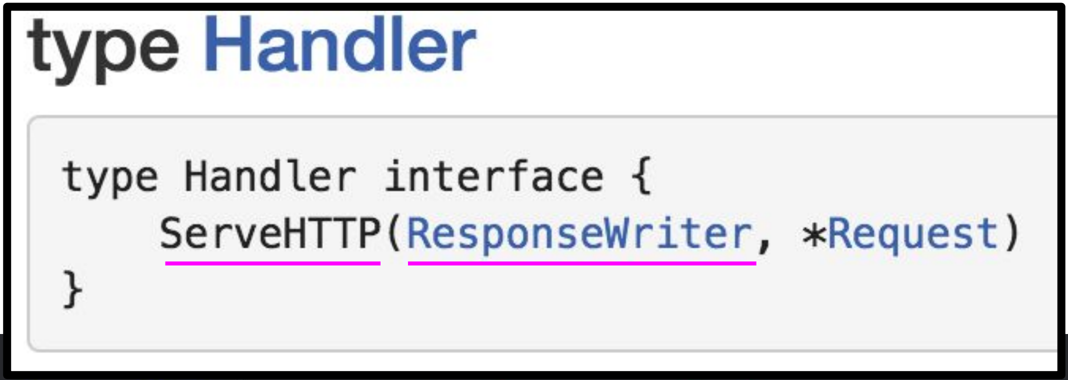
```
type Handler interface {
        ServeHTTP(ResponseWriter, *Request)
}
```



```
func (h myHandler) ServeHTTP(resp http.ResponseWriter, req *http.Request) {
    io.WriteString(resp, req.RequestURI)
}
```

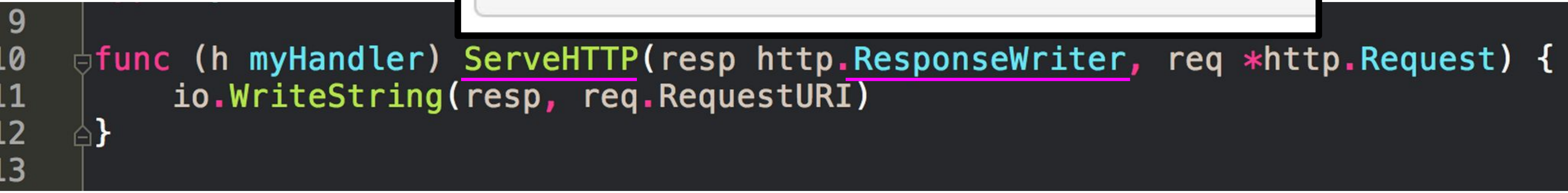
Remember that it's this simple

servers receive requests
and send back responses



type Handler

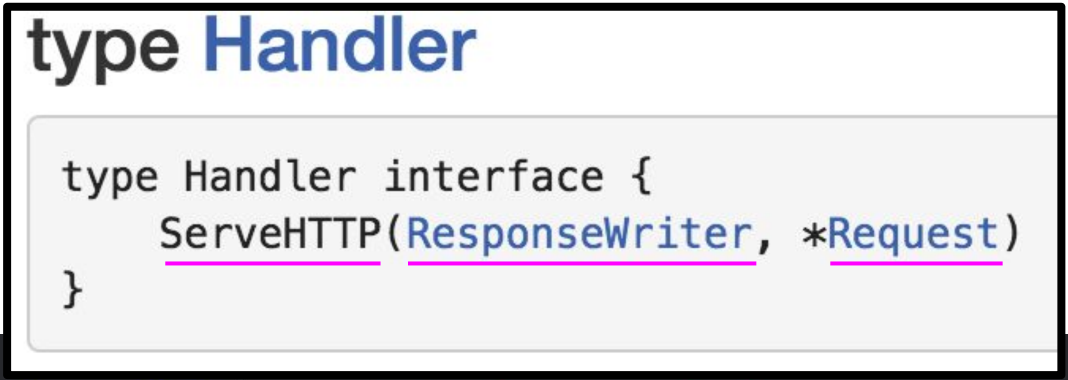
```go
type Handler interface {
        ServeHTTP(ResponseWriter, *Request)
}
```



```go

func (h myHandler) ServeHTTP(resp http.ResponseWriter, req *http.Request) {
    io.WriteString(resp, req.RequestURI)
}

```

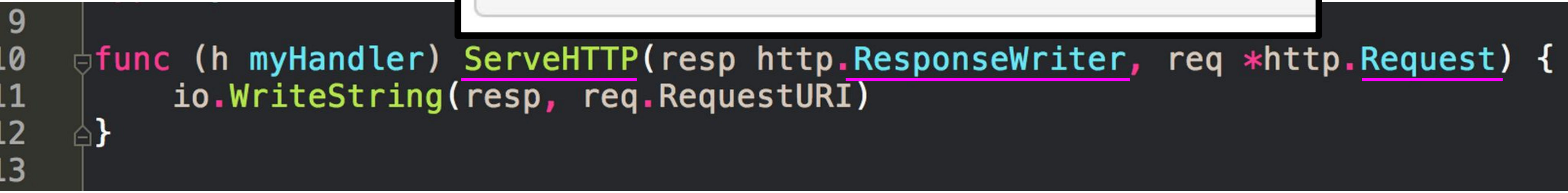
Remember that it's this simple

servers receive requests
and send back responses



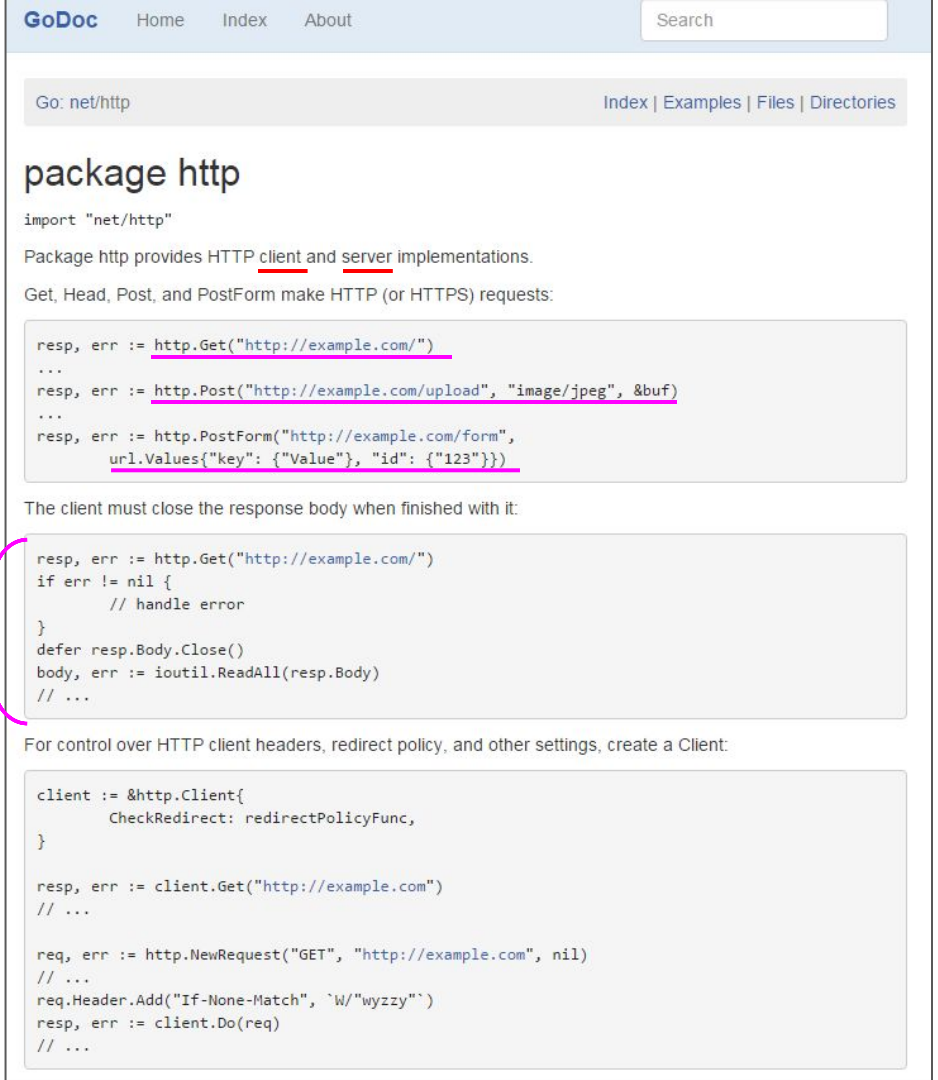
type Handler

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



```
func (h myHandler) ServeHTTP(resp http.ResponseWriter, req *http.Request) {
    io.WriteString(resp, req.RequestURI)
}
```

package net/http



Go: net/http

Index | Examples | Files | Directories

package http

```
import "net/http"
```

Package http provides HTTP client and server implementations.

Get, Head, Post, and PostForm make HTTP (or HTTPS) requests:

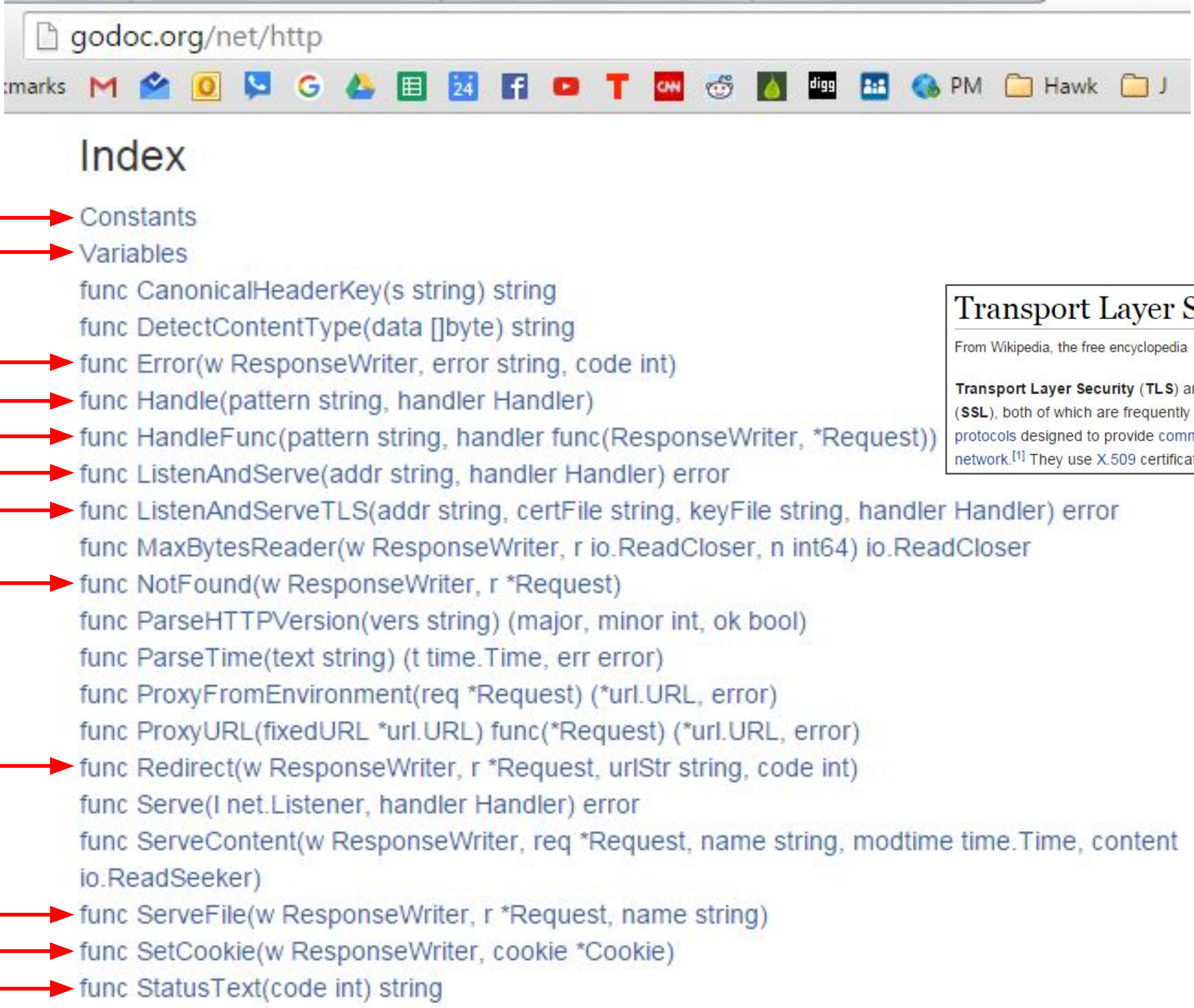
```
resp, err := http.Get("http://example.com/")
...
resp, err := http.Post("http://example.com/upload", "image/jpeg", &buf)
...
resp, err := http.PostForm("http://example.com/form",
        url.Values{"key": {"Value"}, "id": {"123"}})
```

The client must close the response body when finished with it:

```
resp, err := http.Get("http://example.com/")
if err != nil {
        // handle error
}
defer resp.Body.Close()
body, err := ioutil.ReadAll(resp.Body)
// ...
```

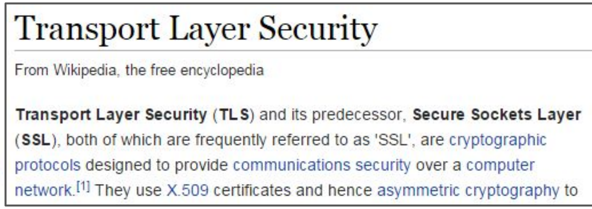
For control over HTTP client headers, redirect policy, and other settings, create a Client:

```
client := &http.Client{
        CheckRedirect: redirectPolicyFunc,
}

resp, err := client.Get("http://example.com")
// ...

req, err := http.NewRequest("GET", "http://example.com", nil)
// ...
req.Header.Add("If-None-Match", `W/"wyzzy"`)
resp, err := client.Do(req)
// ...
```



Index

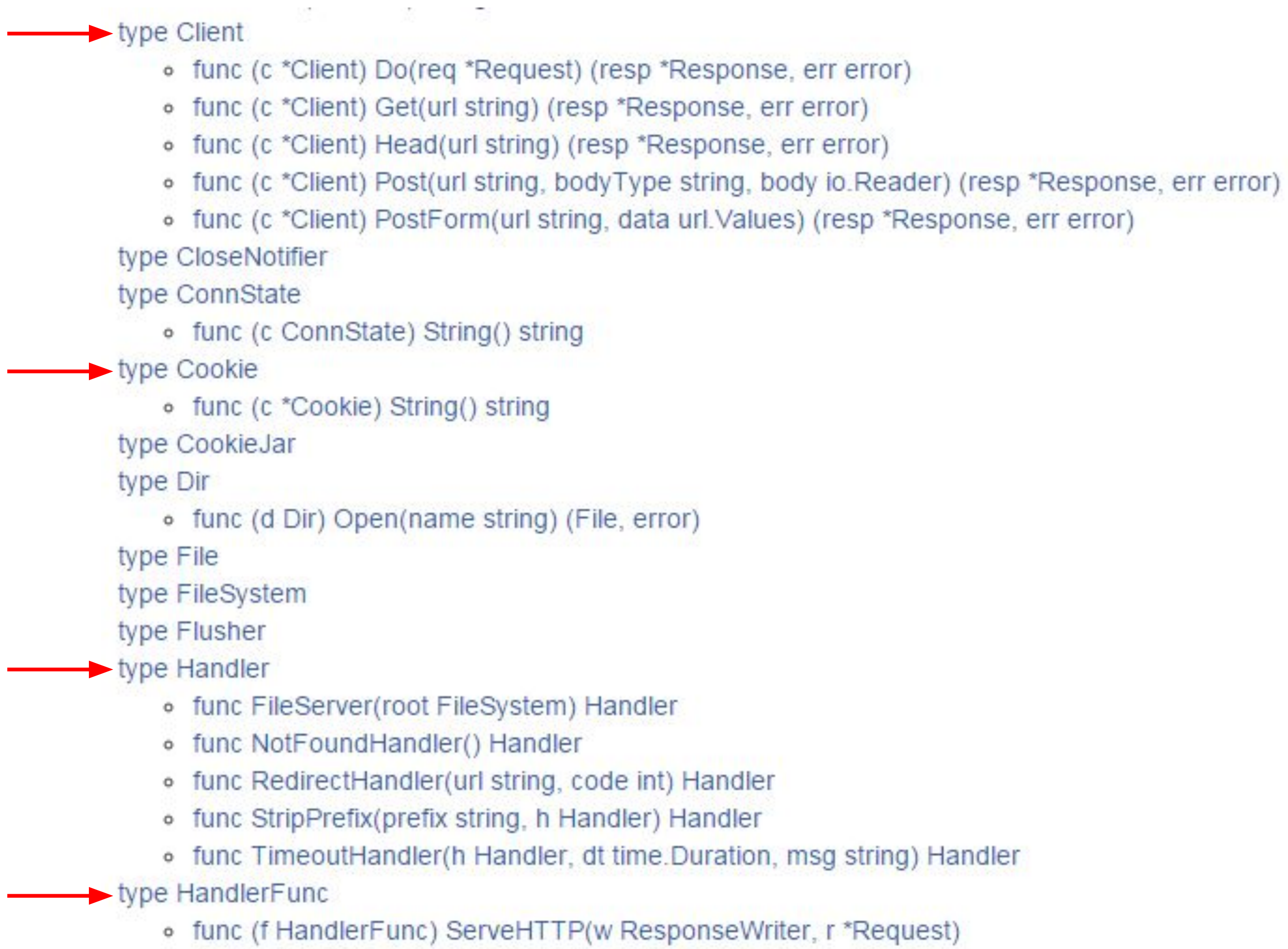
- Constants
- Variables
- func CanonicalHeaderKey(s string) string
- func DetectContentType(data []byte) string
- func Error(w ResponseWriter, error string, code int)
- func Handle(pattern string, handler Handler)
- func HandleFunc(pattern string, handler func(ResponseWriter, *Request))
- func ListenAndServe(addr string, handler Handler) error
- func ListenAndServeTLS(addr string, certFile string, keyFile string, handler Handler) error
- func MaxBytesReader(w ResponseWriter, r io.ReadCloser, n int64) io.ReadCloser
- func NotFound(w ResponseWriter, r *Request)
- func ParseHTTPVersion(vers string) (major, minor int, ok bool)
- func ParseTime(text string) (t time.Time, err error)
- func ProxyFromEnvironment(req *Request) (*url.URL, error)
- func ProxyURL(fixedURL *url.URL) func(*Request) (*url.URL, error)
- func Redirect(w ResponseWriter, r *Request, urlStr string, code int)
- func Serve(l net.Listener, handler Handler) error
- func ServeContent(w ResponseWriter, req *Request, name string, modtime time.Time, content io.ReadSeeker)
- func ServeFile(w ResponseWriter, r *Request, name string)
- func SetCookie(w ResponseWriter, cookie *Cookie)
- func StatusText(code int) string



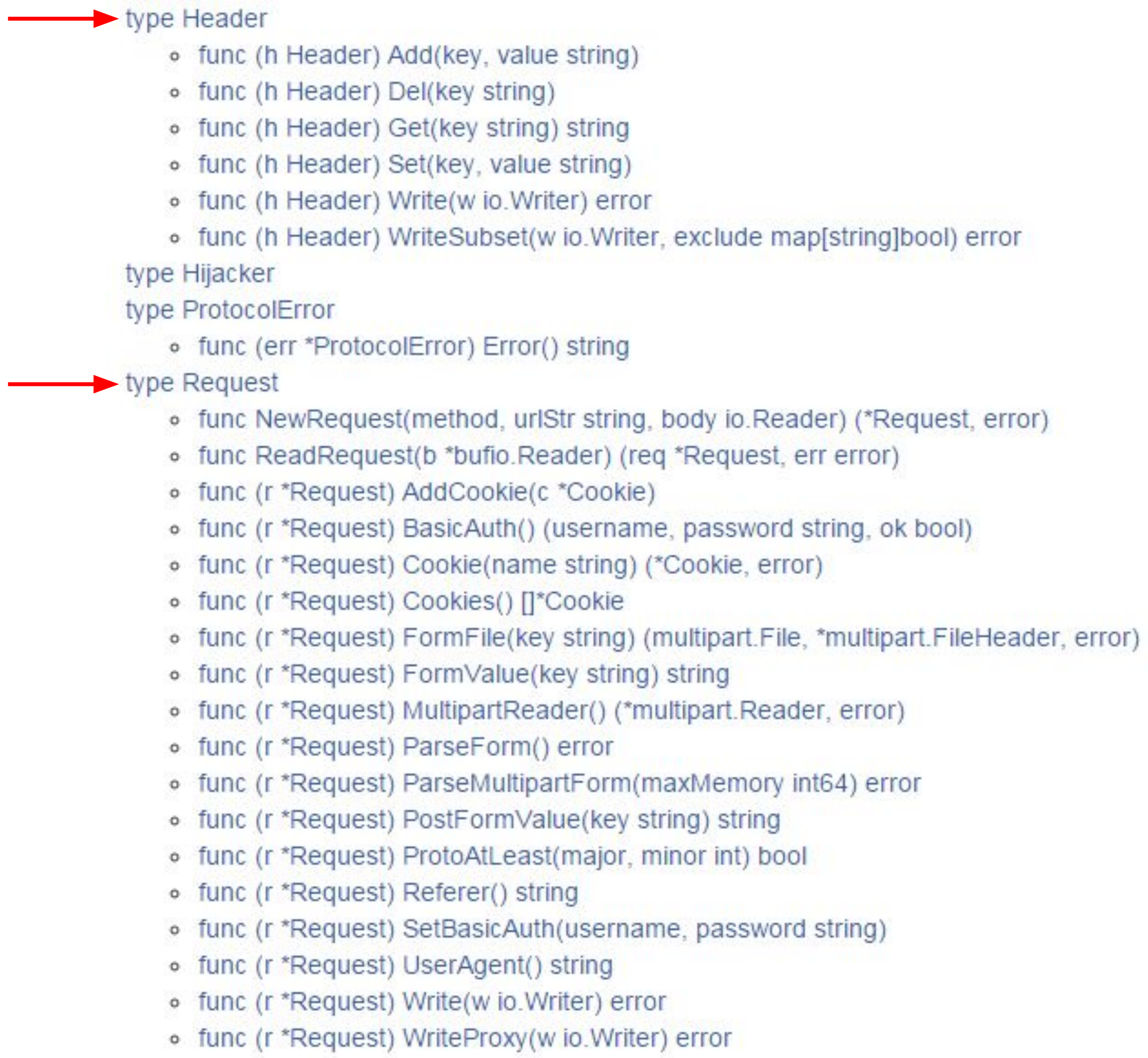
Transport Layer Security

From Wikipedia, the free encyclopedia

Transport Layer Security (**TLS**) and its predecessor, **Secure Sockets Layer** (**SSL**), both of which are frequently referred to as 'SSL', are cryptographic protocols designed to provide communications security over a computer network.[1] They use X.509 certificates and hence asymmetric cryptography to

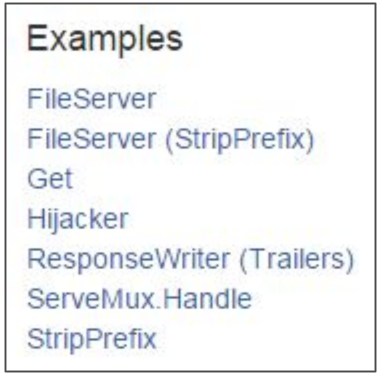


- type Client
  - func (c *Client) Do(req *Request) (resp *Response, err error)
  - func (c *Client) Get(url string) (resp *Response, err error)
  - func (c *Client) Head(url string) (resp *Response, err error)
  - func (c *Client) Post(url string, bodyType string, body io.Reader) (resp *Response, err error)
  - func (c *Client) PostForm(url string, data url.Values) (resp *Response, err error)
- type CloseNotifier
- type ConnState
  - func (c ConnState) String() string
- type Cookie
  - func (c *Cookie) String() string
- type CookieJar
- type Dir
  - func (d Dir) Open(name string) (File, error)
- type File
- type FileSystem
- type Flusher
- type Handler
  - func FileServer(root FileSystem) Handler
  - func NotFoundHandler() Handler
  - func RedirectHandler(url string, code int) Handler
  - func StripPrefix(prefix string, h Handler) Handler
  - func TimeoutHandler(h Handler, dt time.Duration, msg string) Handler
- type HandlerFunc
  - func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request)



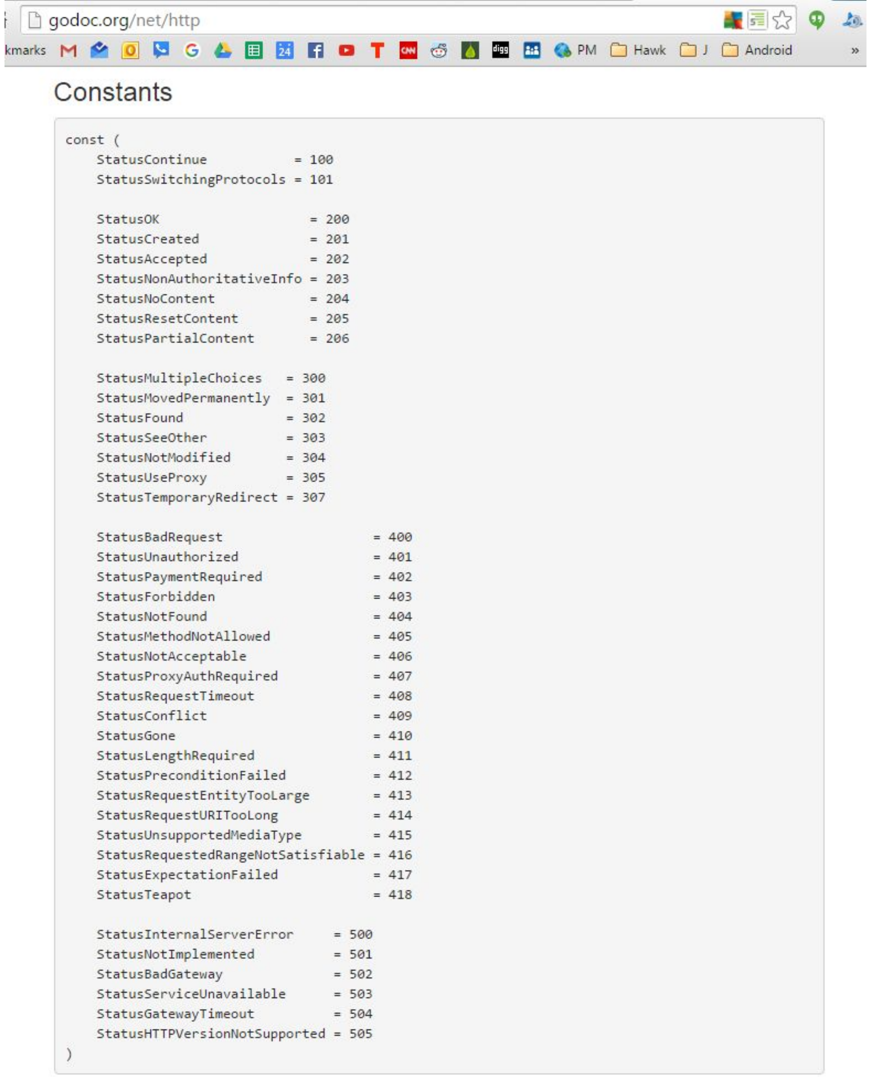
- type Header
  - func (h Header) Add(key, value string)
  - func (h Header) Del(key string)
  - func (h Header) Get(key string) string
  - func (h Header) Set(key, value string)
  - func (h Header) Write(w io.Writer) error
  - func (h Header) WriteSubset(w io.Writer, exclude map[string]bool) error
- type Hijacker
- type ProtocolError
  - func (err *ProtocolError) Error() string
- type Request
  - func NewRequest(method, urlStr string, body io.Reader) (*Request, error)
  - func ReadRequest(b *bufio.Reader) (req *Request, err error)
  - func (r *Request) AddCookie(c *Cookie)
  - func (r *Request) BasicAuth() (username, password string, ok bool)
  - func (r *Request) Cookie(name string) (*Cookie, error)
  - func (r *Request) Cookies() []*Cookie
  - func (r *Request) FormFile(key string) (multipart.File, *multipart.FileHeader, error)
  - func (r *Request) FormValue(key string) string
  - func (r *Request) MultipartReader() (*multipart.Reader, error)
  - func (r *Request) ParseForm() error
  - func (r *Request) ParseMultipartForm(maxMemory int64) error
  - func (r *Request) PostFormValue(key string) string
  - func (r *Request) ProtoAtLeast(major, minor int) bool
  - func (r *Request) Referer() string
  - func (r *Request) SetBasicAuth(username, password string)
  - func (r *Request) UserAgent() string
  - func (r *Request) Write(w io.Writer) error
  - func (r *Request) WriteProxy(w io.Writer) error

- type Response
  - func Get(url string) (resp *Response, err error)
  - func Head(url string) (resp *Response, err error)
  - func Post(url string, bodyType string, body io.Reader) (resp *Response, err error)
  - func PostForm(url string, data url.Values) (resp *Response, err error)
  - func ReadResponse(r *bufio.Reader, req *Request) (*Response, error)
  - func (r *Response) Cookies() []*Cookie
  - func (r *Response) Location() (*url.URL, error)
  - func (r *Response) ProtoAtLeast(major, minor int) bool
  - func (r *Response) Write(w io.Writer) error
- type ResponseWriter
- type RoundTripper
  - func NewFileTransport(fs FileSystem) RoundTripper
- type ServeMux
  - func NewServeMux() *ServeMux
  - func (mux *ServeMux) Handle(pattern string, handler Handler)
  - func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter, *Request))
  - func (mux *ServeMux) Handler(r *Request) (h Handler, pattern string)
  - func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request)
- type Server
  - func (srv *Server) ListenAndServe() error
  - func (srv *Server) ListenAndServeTLS(certFile, keyFile string) error
  - func (srv *Server) Serve(l net.Listener) error
  - func (srv *Server) SetKeepAlivesEnabled(v bool)
- type Transport
  - func (t *Transport) CancelRequest(req *Request)
  - func (t *Transport) CloseIdleConnections()
  - func (t *Transport) RegisterProtocol(scheme string, rt RoundTripper)
  - func (t *Transport) RoundTrip(req *Request) (resp *Response, err error)



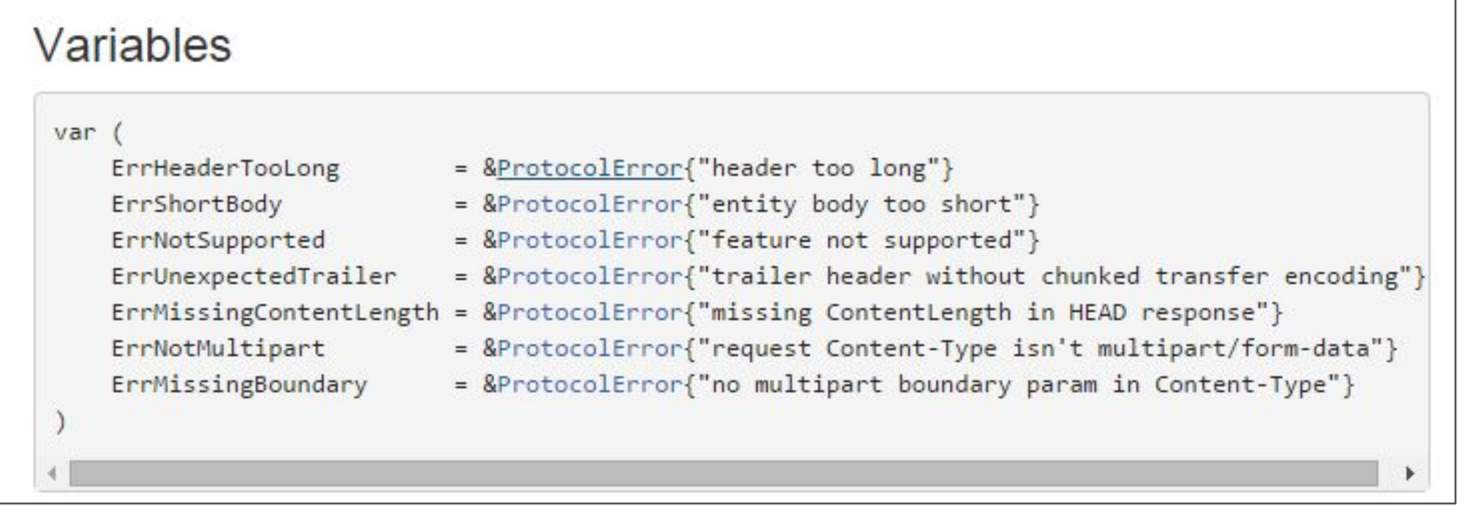
Examples

FileServer
FileServer (StripPrefix)
Get
Hijacker
ResponseWriter (Trailers)
ServeMux.Handle
StripPrefix



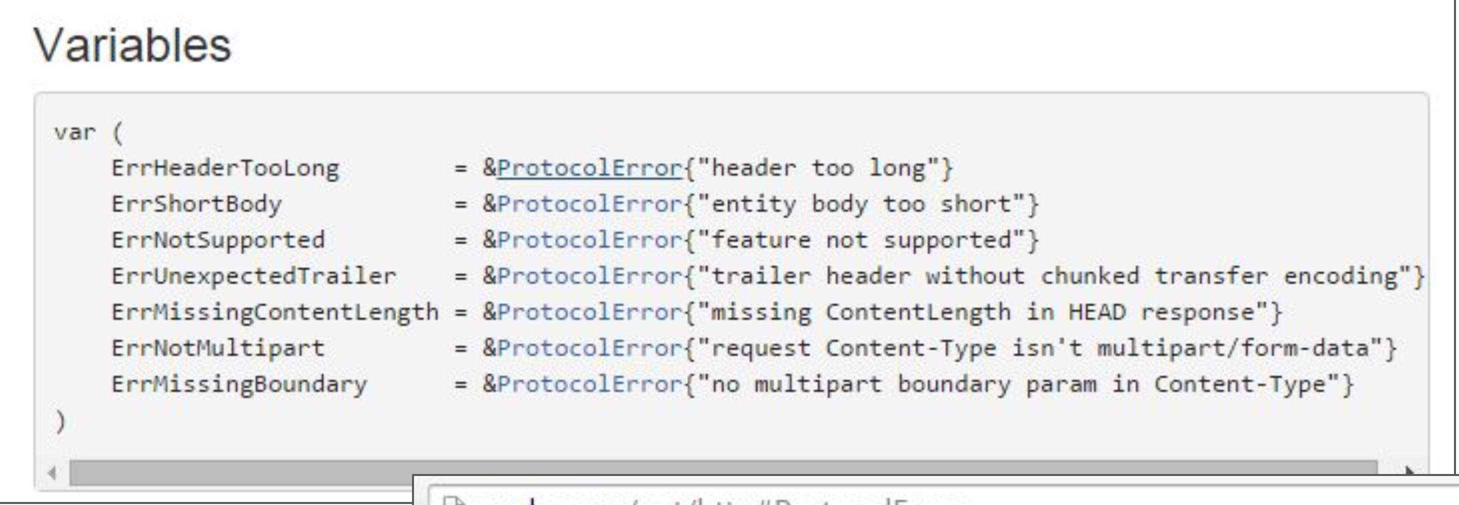
Constants

```
const (
    StatusContinue           = 100
    StatusSwitchingProtocols = 101

    StatusOK                   = 200
    StatusCreated              = 201
    StatusAccepted             = 202
    StatusNonAuthoritativeInfo = 203
    StatusNoContent            = 204
    StatusResetContent         = 205
    StatusPartialContent       = 206

    StatusMultipleChoices   = 300
    StatusMovedPermanently  = 301
    StatusFound             = 302
    StatusSeeOther          = 303
    StatusNotModified       = 304
    StatusUseProxy          = 305
    StatusTemporaryRedirect = 307

    StatusBadRequest                   = 400
    StatusUnauthorized                 = 401
    StatusPaymentRequired              = 402
    StatusForbidden                    = 403
    StatusNotFound                     = 404
    StatusMethodNotAllowed             = 405
    StatusNotAcceptable                = 406
    StatusProxyAuthRequired            = 407
    StatusRequestTimeout               = 408
    StatusConflict                     = 409
    StatusGone                         = 410
    StatusLengthRequired               = 411
    StatusPreconditionFailed           = 412
    StatusRequestEntityTooLarge        = 413
    StatusRequestURITooLong            = 414
    StatusUnsupportedMediaType         = 415
    StatusRequestedRangeNotSatisfiable = 416
    StatusExpectationFailed            = 417
    StatusTeapot                       = 418

    StatusInternalServerError     = 500
    StatusNotImplemented          = 501
    StatusBadGateway              = 502
    StatusServiceUnavailable      = 503
    StatusGatewayTimeout          = 504
    StatusHTTPVersionNotSupported = 505
)
```



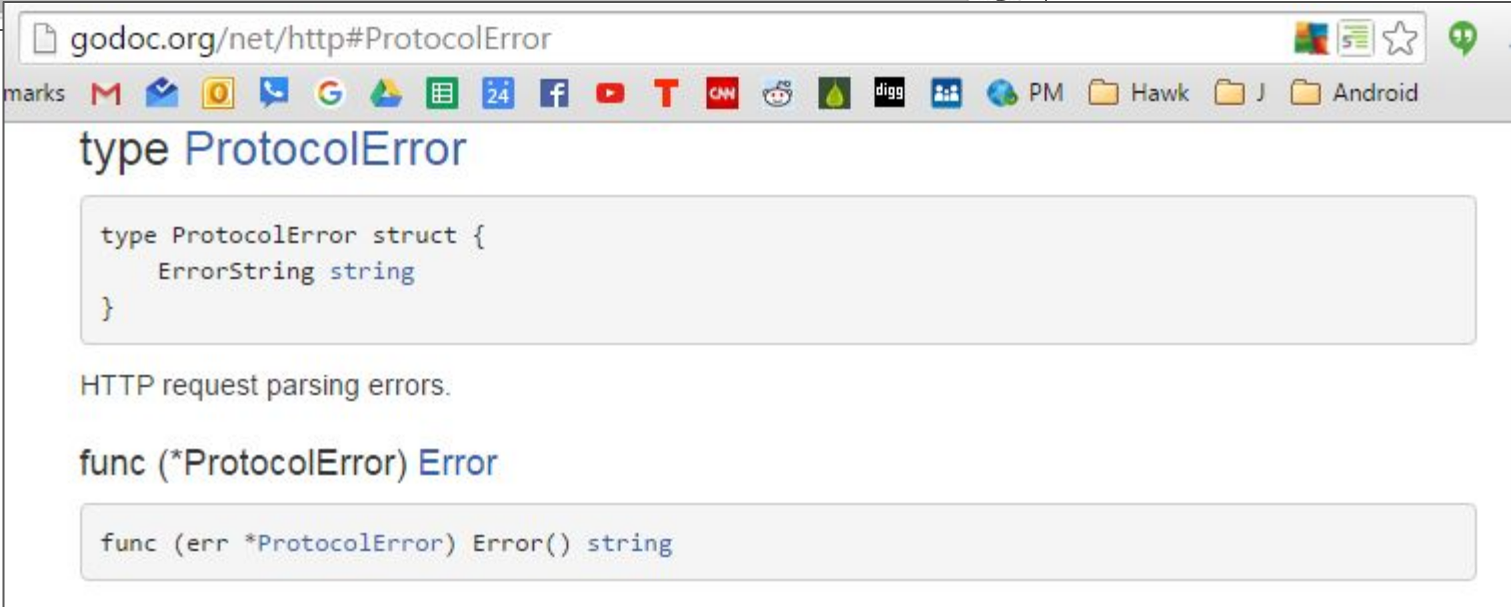
Variables

```
var (
    ErrHeaderTooLong        = &ProtocolError{"header too long"}
    ErrShortBody            = &ProtocolError{"entity body too short"}
    ErrNotSupported         = &ProtocolError{"feature not supported"}
    ErrUnexpectedTrailer    = &ProtocolError{"trailer header without chunked transfer encoding"}
    ErrMissingContentLength = &ProtocolError{"missing ContentLength in HEAD response"}
    ErrNotMultipart         = &ProtocolError{"request Content-Type isn't multipart/form-data"}
    ErrMissingBoundary      = &ProtocolError{"no multipart boundary param in Content-Type"}
)
```



Variables

```
var (
    ErrHeaderTooLong        = &ProtocolError{"header too long"}
    ErrShortBody            = &ProtocolError{"entity body too short"}
    ErrNotSupported         = &ProtocolError{"feature not supported"}
    ErrUnexpectedTrailer    = &ProtocolError{"trailer header without chunked transfer encoding"}
    ErrMissingContentLength = &ProtocolError{"missing ContentLength in HEAD response"}
    ErrNotMultipart         = &ProtocolError{"request Content-Type isn't multipart/form-data"}
    ErrMissingBoundary      = &ProtocolError{"no multipart boundary param in Content-Type"}
)
```



godoc.org/net/http#ProtocolError

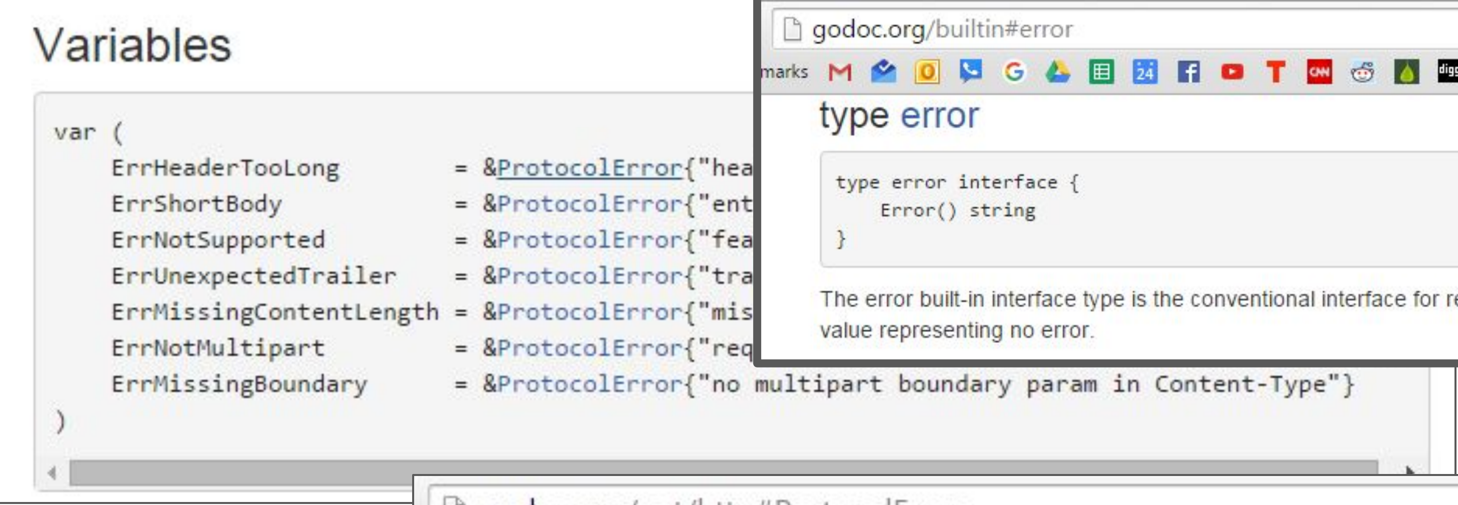
type ProtocolError

```
type ProtocolError struct {
    ErrorString string
}
```

HTTP request parsing errors.

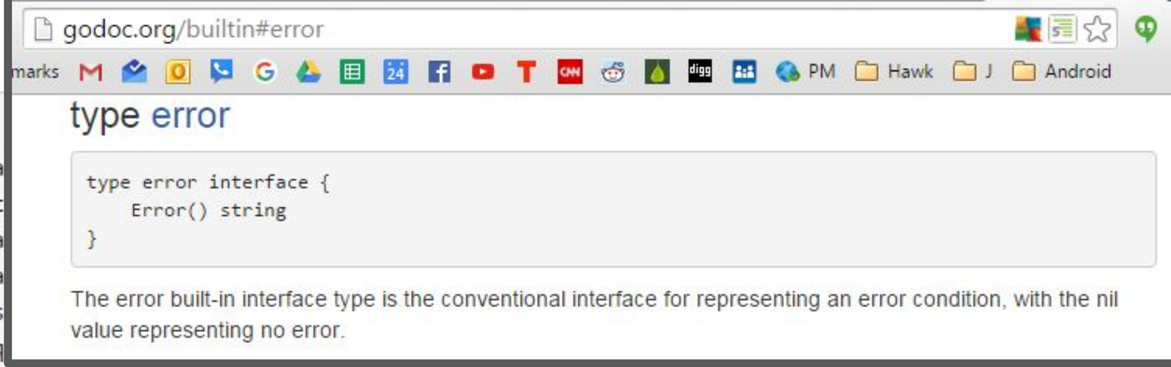
func (*ProtocolError) Error

```
func (err *ProtocolError) Error() string
```



Variables

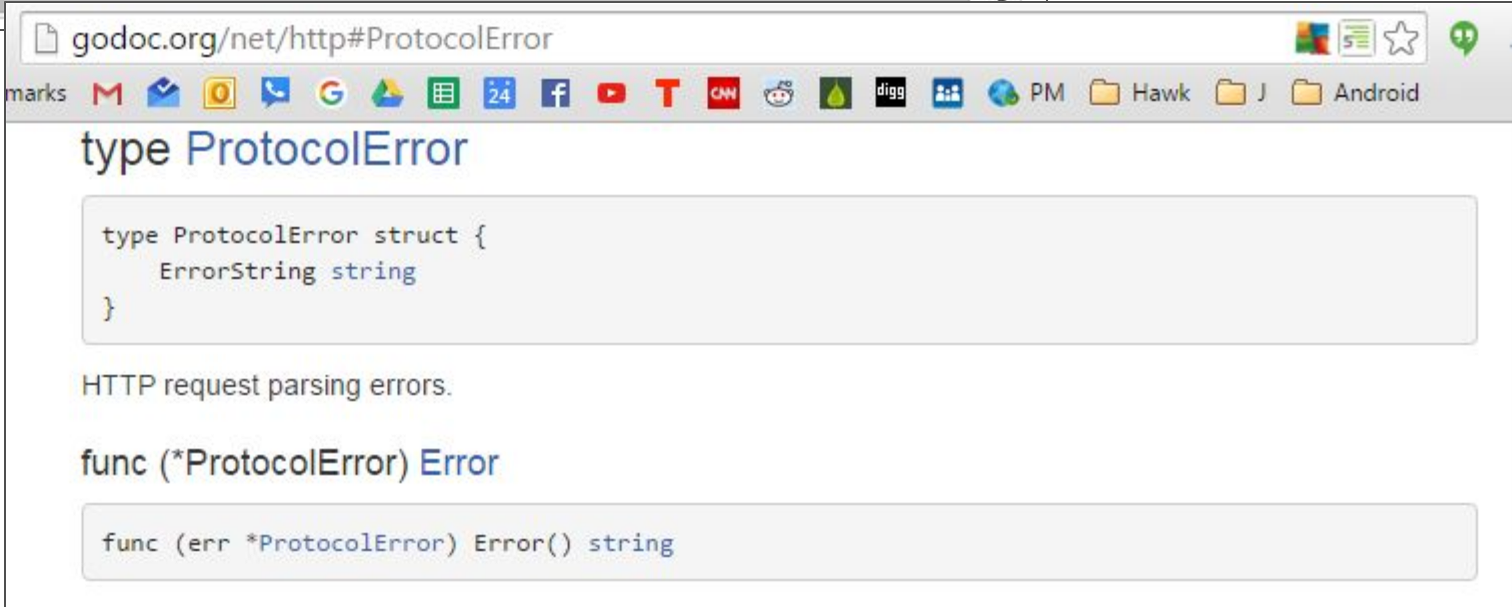
```
var (
    ErrHeaderTooLong        = &ProtocolError{"hea
    ErrShortBody            = &ProtocolError{"ent
    ErrNotSupported         = &ProtocolError{"fea
    ErrUnexpectedTrailer    = &ProtocolError{"tra
    ErrMissingContentLength = &ProtocolError{"mis
    ErrNotMultipart         = &ProtocolError{"req
    ErrMissingBoundary      = &ProtocolError{"no multipart boundary param in Content-Type"}
)
```



type error

```
type error interface {
    Error() string
}
```

The error built-in interface type is the conventional interface for representing an error condition, with the nil value representing no error.



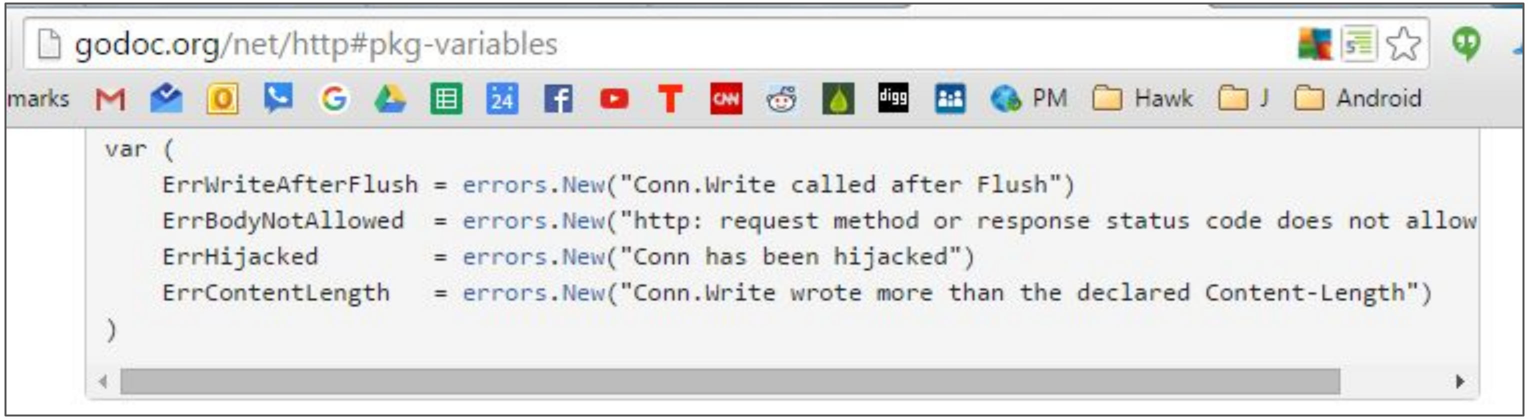
type ProtocolError

```
type ProtocolError struct {
    ErrorString string
}
```

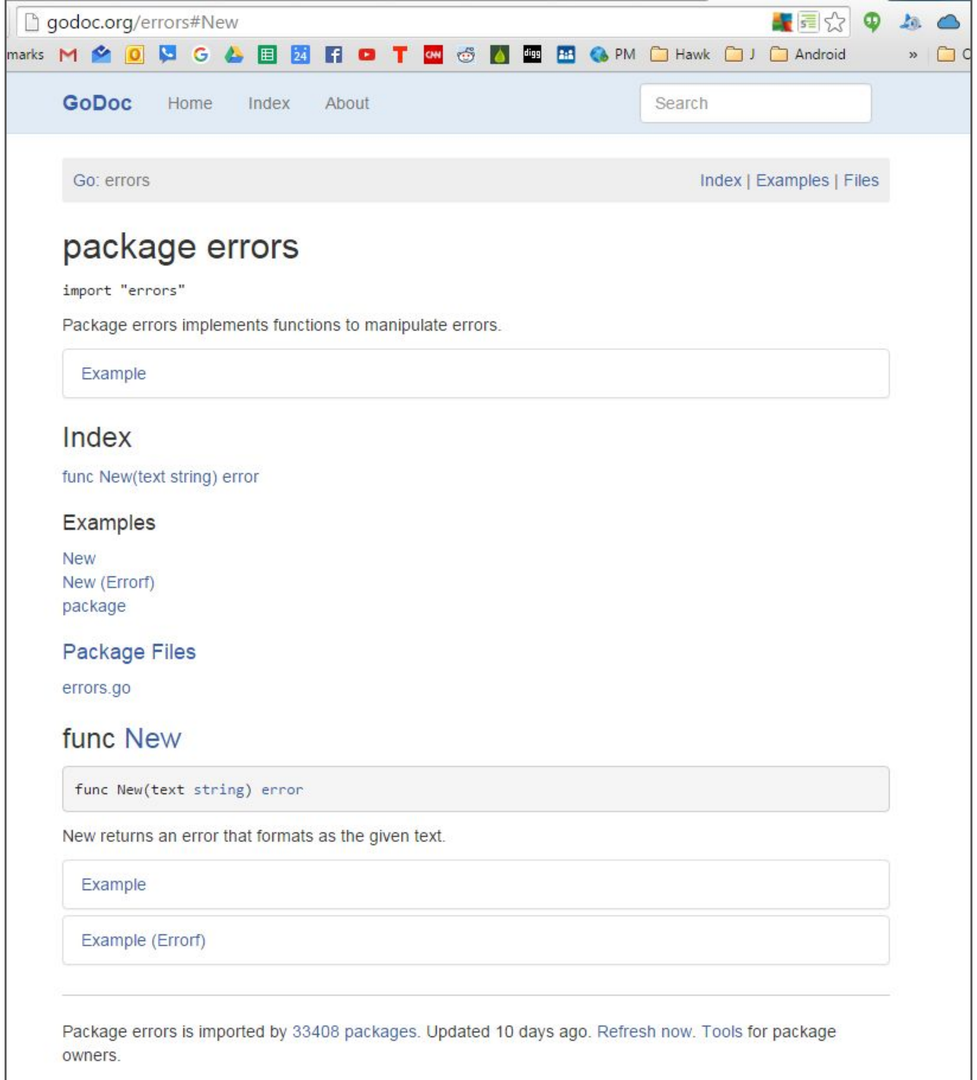
HTTP request parsing errors.

func (*ProtocolError) Error

```
func (err *ProtocolError) Error() string
```


godoc.org/net/http#pkg-variables

```
var (
    ErrWriteAfterFlush = errors.New("Conn.Write called after Flush")
    ErrBodyNotAllowed  = errors.New("http: request method or response status code does not allow
    ErrHijacked        = errors.New("Conn has been hijacked")
    ErrContentLength   = errors.New("Conn.Write wrote more than the declared Content-Length")
)
```



godoc.org/errors#New

marks M O 24 PM Hawk J Android » C

GoDoc Home Index About Search

Go: errors Index | Examples | Files

package errors

```
import "errors"
```

Package errors implements functions to manipulate errors.

Example

Index

func New(text string) error

Examples

New
New (Errorf)
package

Package Files

errors.go

func New

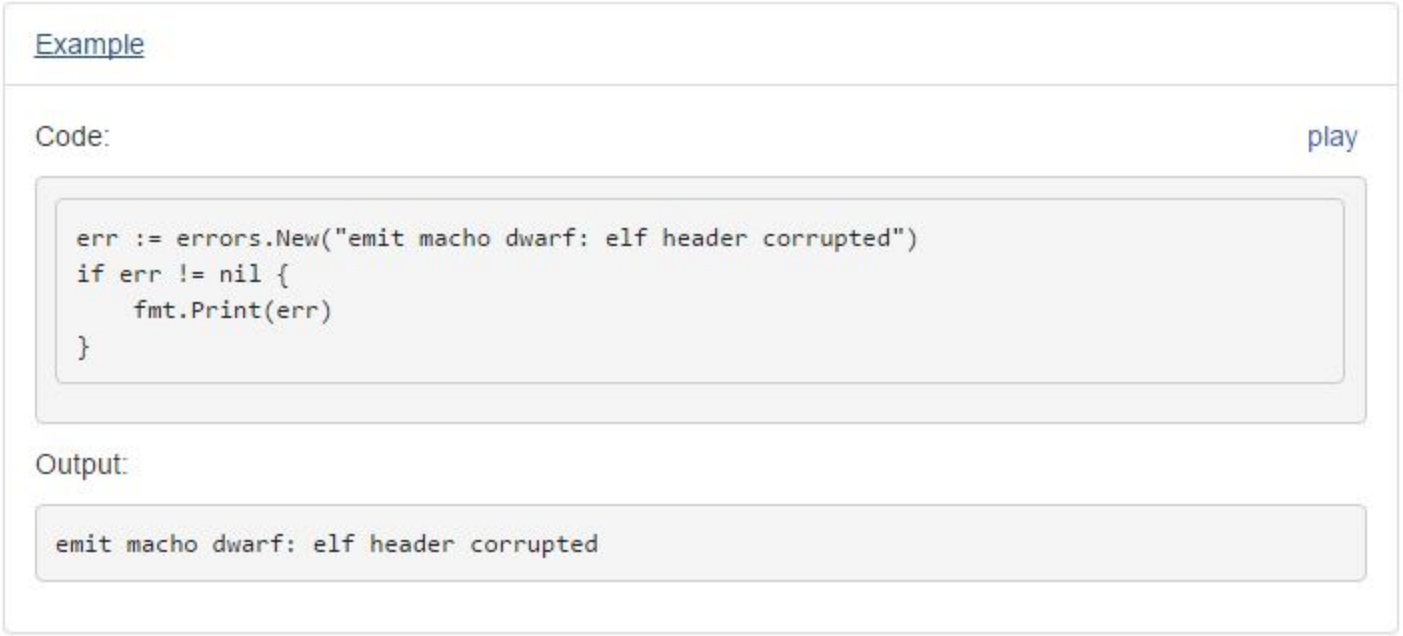
```
func New(text string) error
```

New returns an error that formats as the given text.

Example

Example (Errorf)

Package errors is imported by 33408 packages. Updated 10 days ago. Refresh now. Tools for package owners.



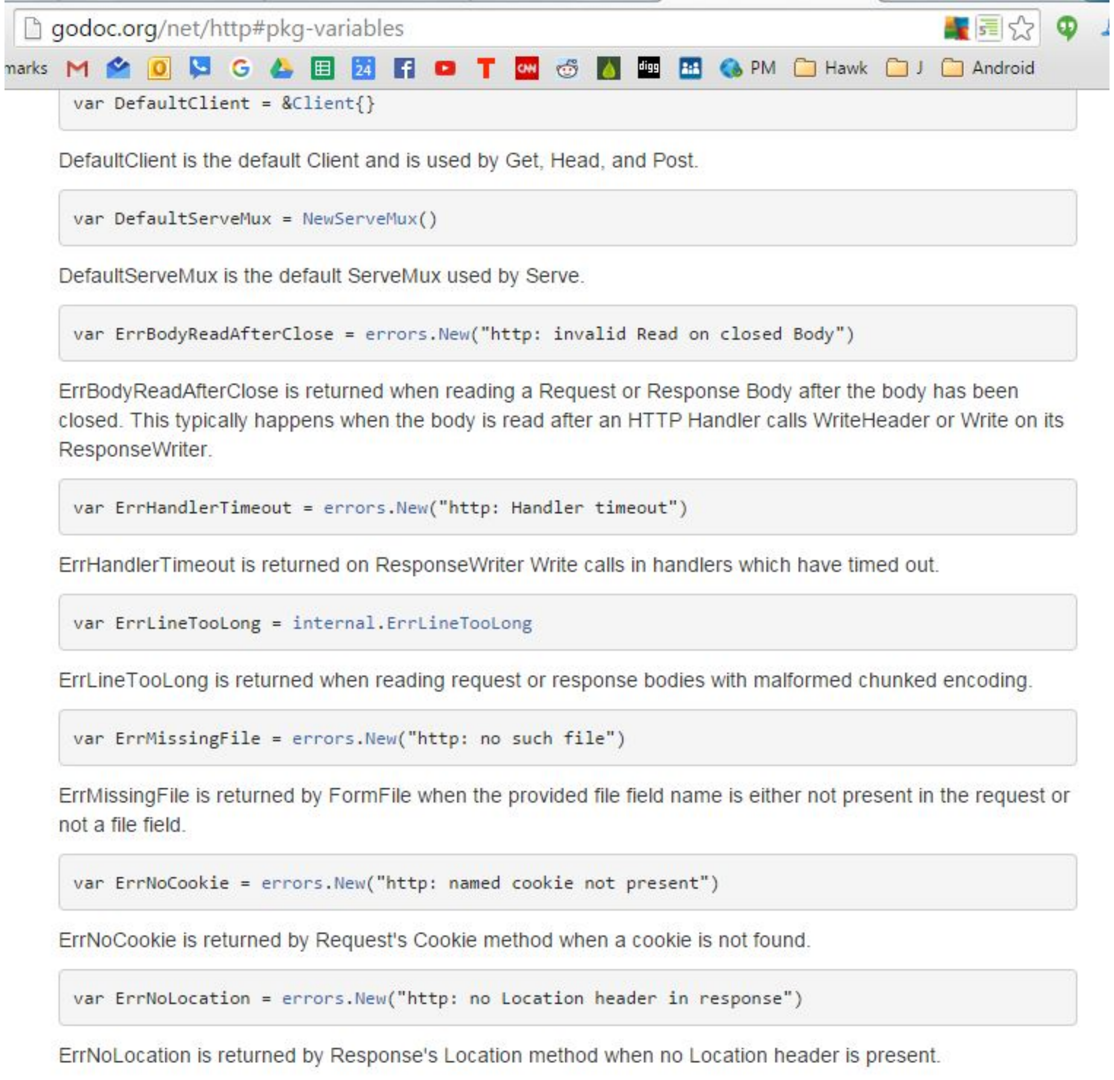
Example

Code: play

```
err := errors.New("emit macho dwarf: elf header corrupted")
if err != nil {
    fmt.Print(err)
}
```

Output:

```
emit macho dwarf: elf header corrupted
```


godoc.org/net/http#pkg-variables

marks PM Hawk J Android

```
var DefaultClient = &Client{}
```

DefaultClient is the default Client and is used by Get, Head, and Post.

```
var DefaultServeMux = NewServeMux()
```

DefaultServeMux is the default ServeMux used by Serve.

```
var ErrBodyReadAfterClose = errors.New("http: invalid Read on closed Body")
```

ErrBodyReadAfterClose is returned when reading a Request or Response Body after the body has been closed. This typically happens when the body is read after an HTTP Handler calls WriteHeader or Write on its ResponseWriter.

```
var ErrHandlerTimeout = errors.New("http: Handler timeout")
```

ErrHandlerTimeout is returned on ResponseWriter Write calls in handlers which have timed out.

```
var ErrLineTooLong = internal.ErrLineTooLong
```

ErrLineTooLong is returned when reading request or response bodies with malformed chunked encoding.

```
var ErrMissingFile = errors.New("http: no such file")
```

ErrMissingFile is returned by FormFile when the provided file field name is either not present in the request or not a file field.

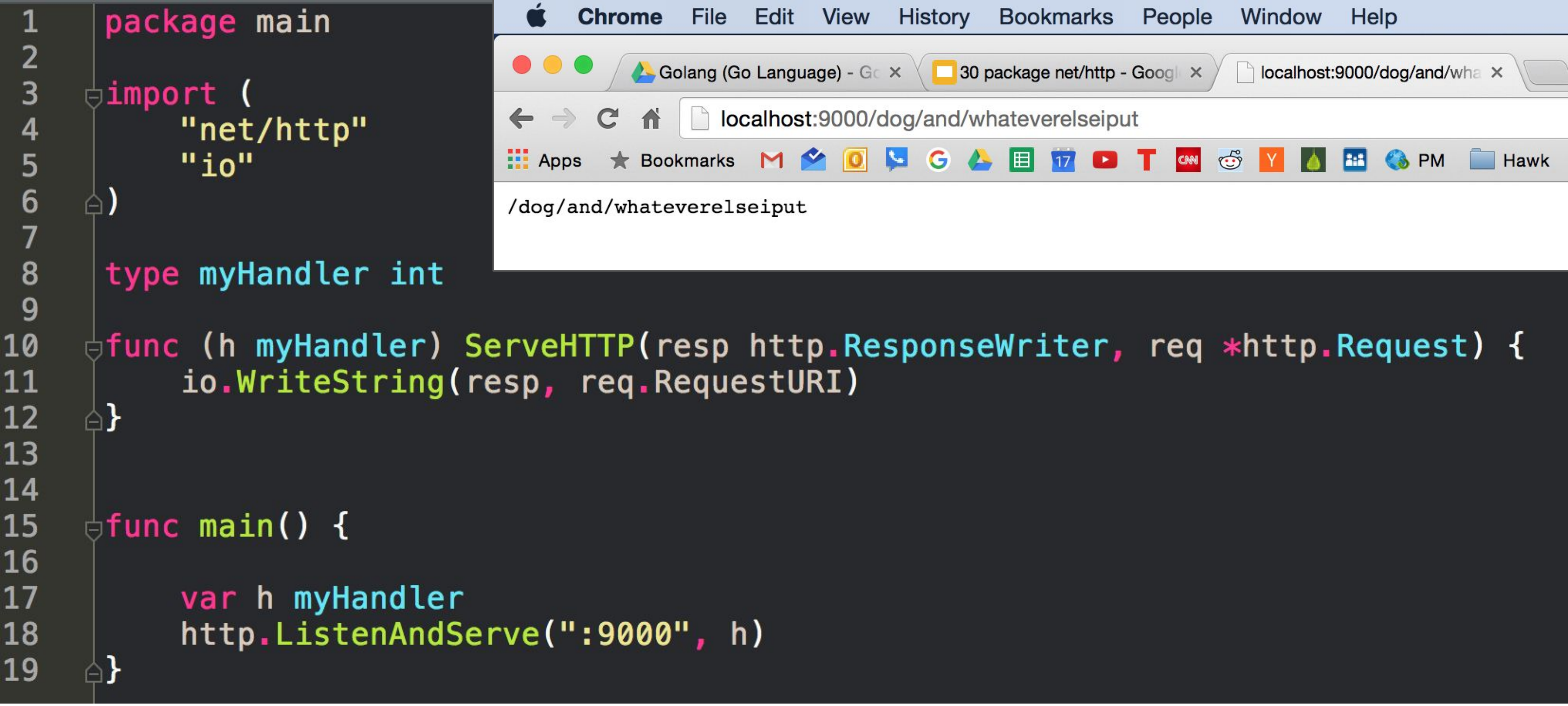
```
var ErrNoCookie = errors.New("http: named cookie not present")
```

ErrNoCookie is returned by Request's Cookie method when a cookie is not found.

```
var ErrNoLocation = errors.New("http: no Location header in response")
```

ErrNoLocation is returned by Response's Location method when no Location header is present.

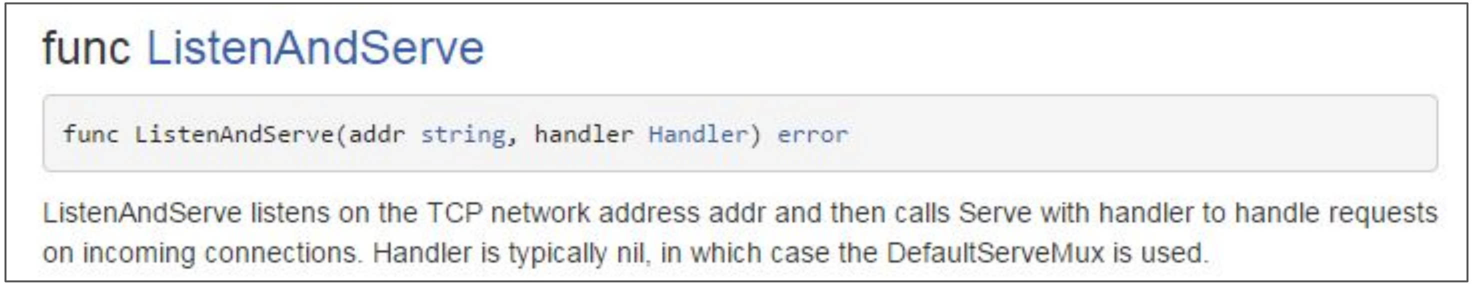
understanding our example


Chrome
File
Edit
View
History
Bookmarks
People
Window
Help
Golang (Go Language) - Go
30 package net/http - Googl
localhost:9000/dog/and/wha
Apps
Bookmarks
PM
Hawk

```go
package main

import (
	"net/http"
	"io"
)

type myHandler int

func (h myHandler) ServeHTTP(resp http.ResponseWriter, req *http.Request) {
	io.WriteString(resp, req.RequestURI)
}


func main() {

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

localhost:9000/dog/and/whateverelseiput

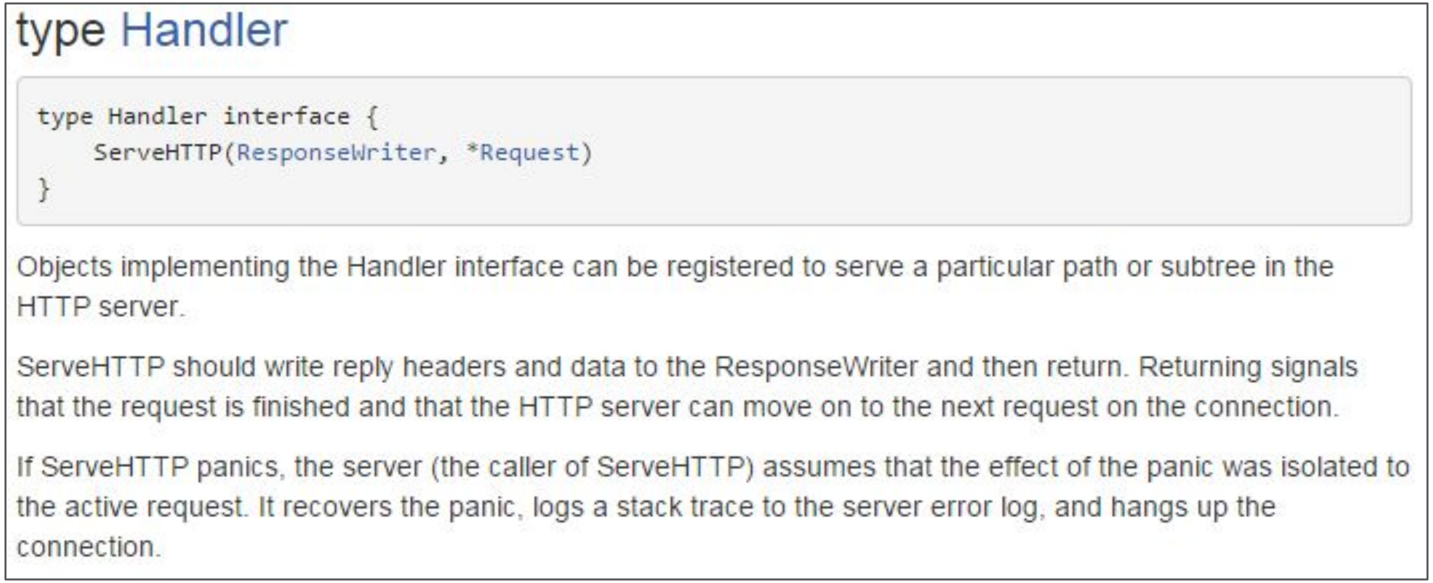
```
/dog/and/whateverelseiput
```



func ListenAndServe

```
func ListenAndServe(addr string, handler Handler) error
```

ListenAndServe listens on the TCP network address addr and then calls Serve with handler to handle requests on incoming connections. Handler is typically nil, in which case the DefaultServeMux is used.



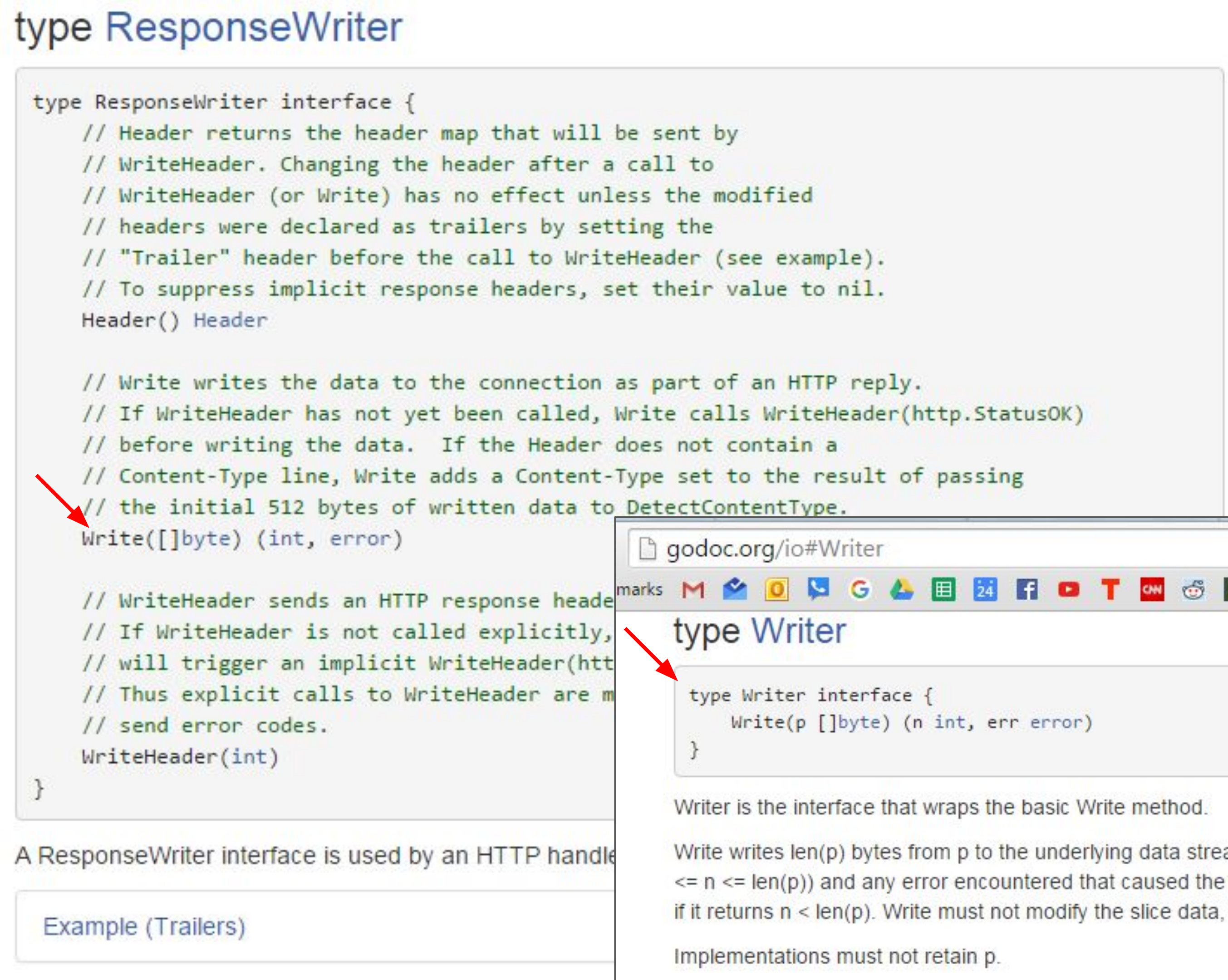
type Handler

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

Objects implementing the Handler interface can be registered to serve a particular path or subtree in the HTTP server.

ServeHTTP should write reply headers and data to the ResponseWriter and then return. Returning signals that the request is finished and that the HTTP server can move on to the next request on the connection.

If ServeHTTP panics, the server (the caller of ServeHTTP) assumes that the effect of the panic was isolated to the active request. It recovers the panic, logs a stack trace to the server error log, and hangs up the connection.

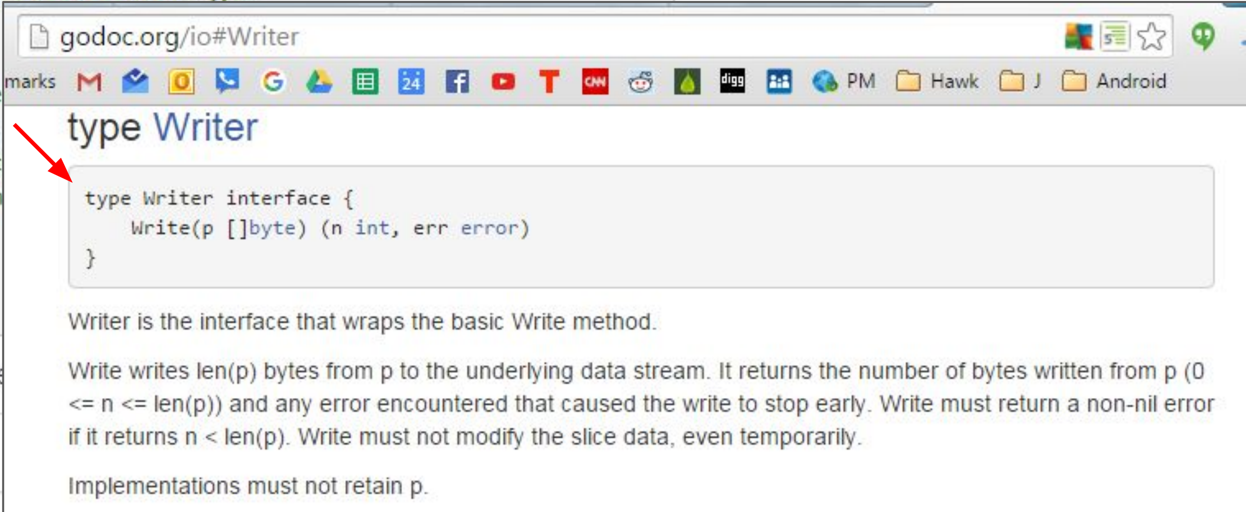


type ResponseWriter

```
type ResponseWriter interface {
    // Header returns the header map that will be sent by
    // WriteHeader. Changing the header after a call to
    // WriteHeader (or Write) has no effect unless the modified
    // headers were declared as trailers by setting the
    // "Trailer" header before the call to WriteHeader (see example).
    // To suppress implicit response headers, set their value to nil.
    Header() Header

    // Write writes the data to the connection as part of an HTTP reply.
    // If WriteHeader has not yet been called, Write calls WriteHeader(http.StatusOK)
    // before writing the data.  If the Header does not contain a
    // Content-Type line, Write adds a Content-Type set to the result of passing
    // the initial 512 bytes of written data to DetectContentType.
    Write([]byte) (int, error)

    // WriteHeader sends an HTTP response heade
    // If WriteHeader is not called explicitly,
    // will trigger an implicit WriteHeader(htt
    // Thus explicit calls to WriteHeader are m
    // send error codes.
    WriteHeader(int)
}
```

A ResponseWriter interface is used by an HTTP handle

Example (Trailers)



type Writer

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Writer is the interface that wraps the basic Write method.

Write writes len(p) bytes from p to the underlying data stream. It returns the number of bytes written from p (0 <= n <= len(p)) and any error encountered that caused the write to stop early. Write must return a non-nil error if it returns n < len(p). Write must not modify the slice data, even temporarily.

Implementations must not retain p.