# HTTP

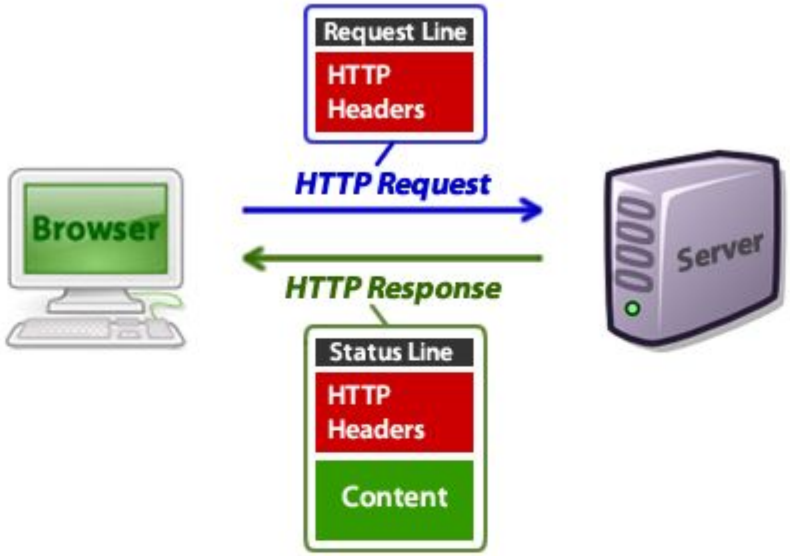
## HTTP

- a protocol on top of TCP
- spec
  - http://www.w3.org/Protocols/rfc2616/rfc2616.txt

servers receive requests
and send back responses

**Remember it's this simple**

servers receive requests
and send back responses

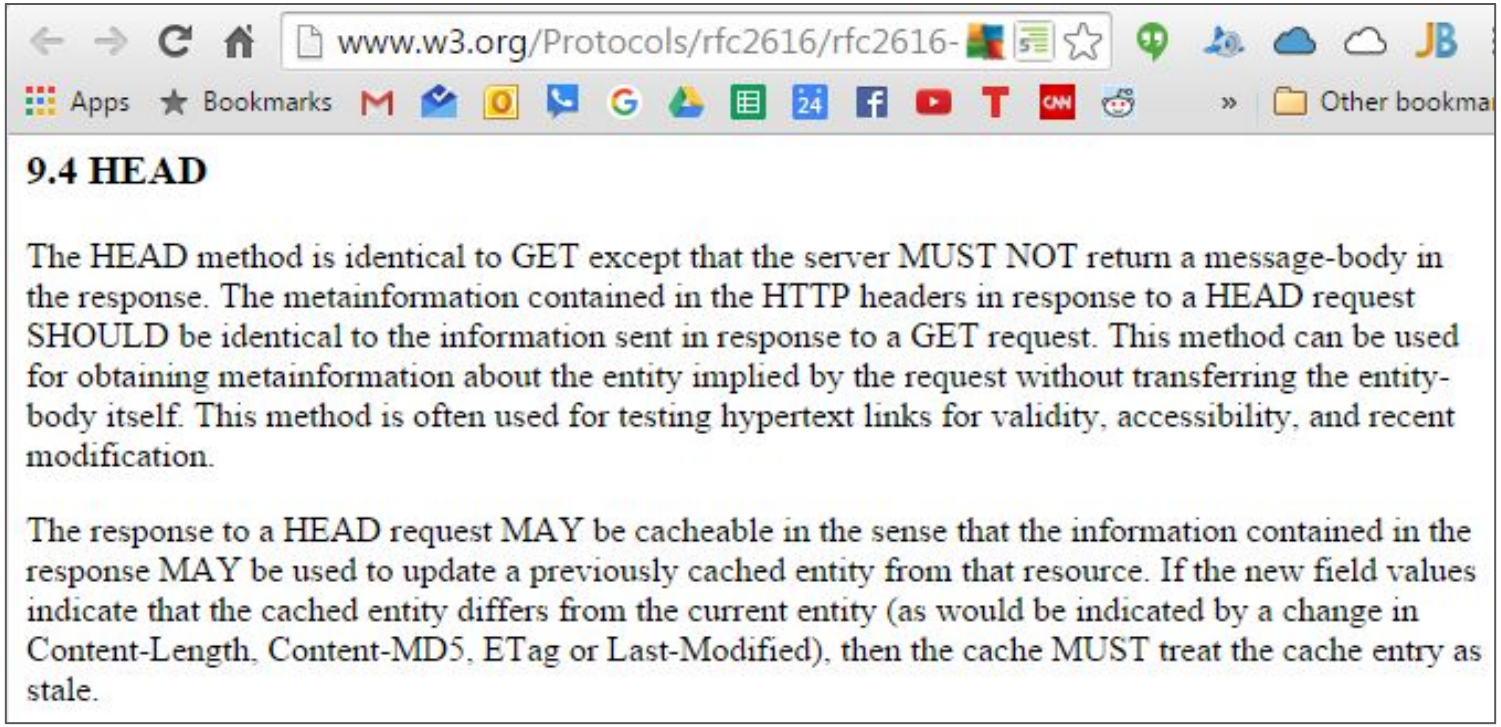

Request Line
HTTP Headers
HTTP Request
Browser
Server
HTTP Response
Status Line
HTTP Headers
Content

## HTTP Verbs / Methods

- GET
- POST
- PUT
- DELETE
- HEAD

read more on wikipedia

- GET
- POST
- PUT
- DELETE
- HEAD

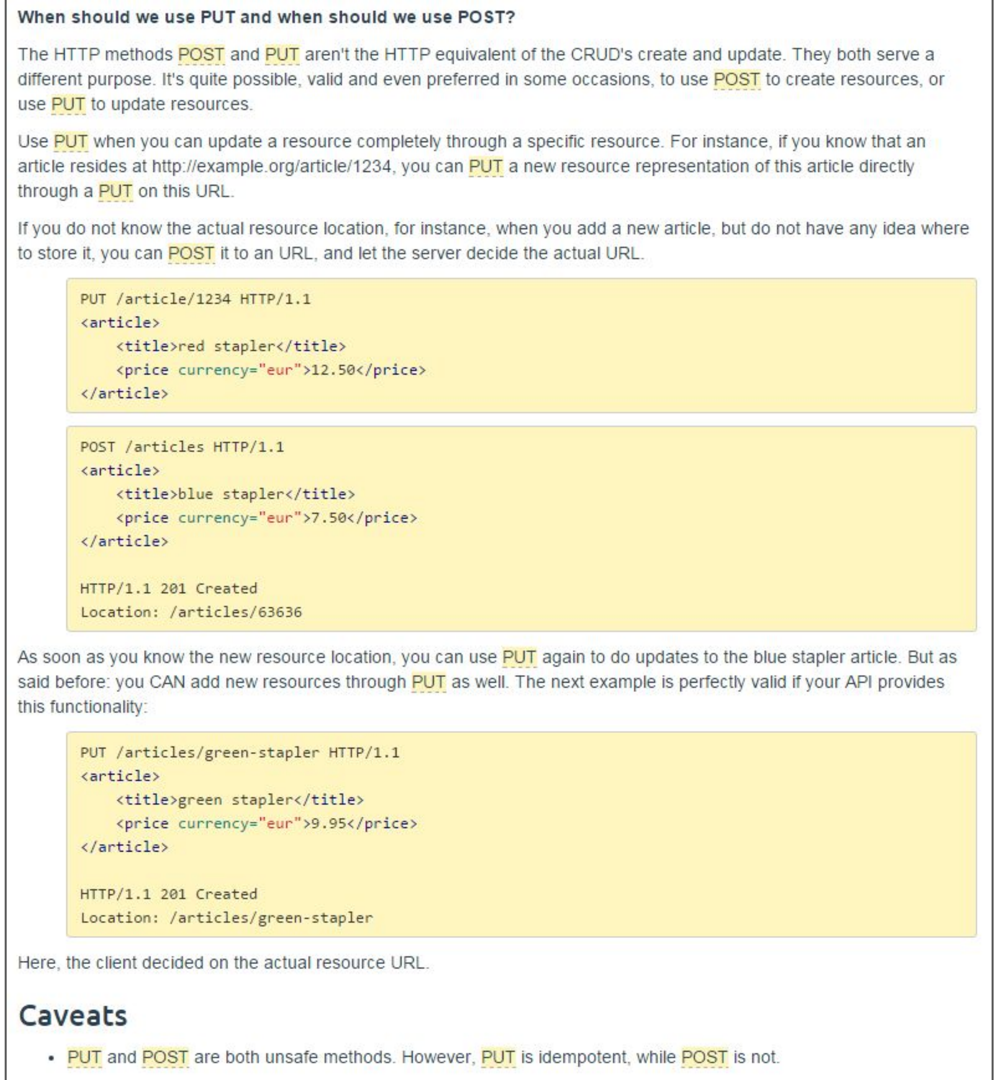


### 9.4 HEAD

The HEAD method is identical to GET except that the server MUST NOT return a message-body in the response. The metainformation contained in the HTTP headers in response to a HEAD request SHOULD be identical to the information sent in response to a GET request. This method can be used for obtaining metainformation about the entity implied by the request without transferring the entity-body itself. This method is often used for testing hypertext links for validity, accessibility, and recent modification.

The response to a HEAD request MAY be cacheable in the sense that the information contained in the response MAY be used to update a previously cached entity from that resource. If the new field values indicate that the cached entity differs from the current entity (as would be indicated by a change in Content-Length, Content-MD5, ETag or Last-Modified), then the cache MUST treat the cache entry as stale.

- GET
- **POST**
- **PUT**
- DELETE
- HEAD



**When should we use PUT and when should we use POST?**

The HTTP methods POST and PUT aren't the HTTP equivalent of the CRUD's create and update. They both serve a different purpose. It's quite possible, valid and even preferred in some occasions, to use POST to create resources, or use PUT to update resources.

Use PUT when you can update a resource completely through a specific resource. For instance, if you know that an article resides at http://example.org/article/1234, you can PUT a new resource representation of this article directly through a PUT on this URL.

If you do not know the actual resource location, for instance, when you add a new article, but do not have any idea where to store it, you can POST it to an URL, and let the server decide the actual URL.

```
PUT /article/1234 HTTP/1.1
<article>
    <title>red stapler</title>
    <price currency="eur">12.50</price>
</article>
```

```
POST /articles HTTP/1.1
<article>
    <title>blue stapler</title>
    <price currency="eur">7.50</price>
</article>

HTTP/1.1 201 Created
Location: /articles/63636
```

As soon as you know the new resource location, you can use PUT again to do updates to the blue stapler article. But as said before: you CAN add new resources through PUT as well. The next example is perfectly valid if your API provides this functionality:

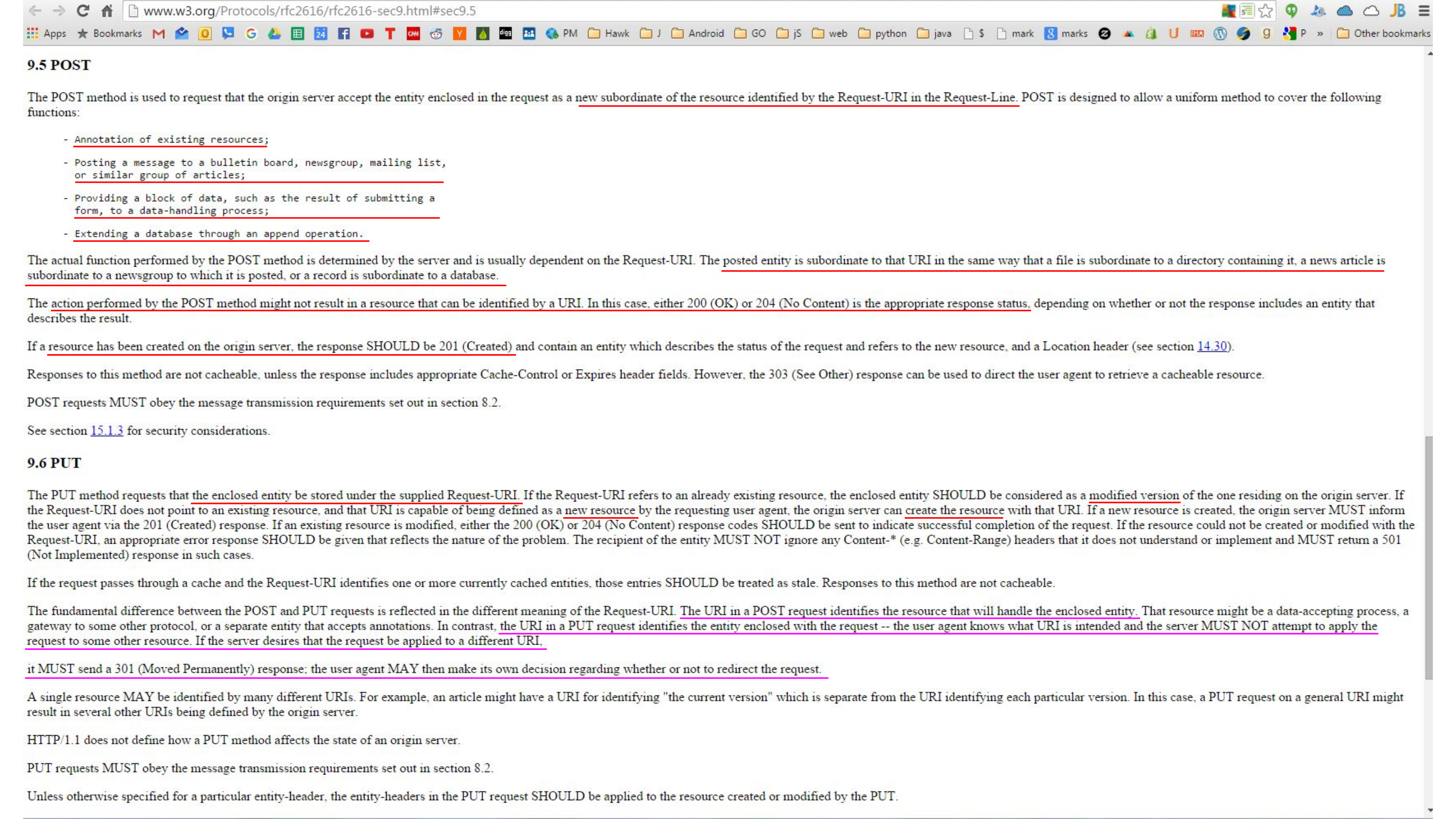
```
PUT /articles/green-stapler HTTP/1.1
<article>
    <title>green stapler</title>
    <price currency="eur">9.95</price>
</article>

HTTP/1.1 201 Created
Location: /articles/green-stapler
```

Here, the client decided on the actual resource URL.

## Caveats

- PUT and POST are both unsafe methods. However, PUT is idempotent, while POST is not.

http://restcookbook.com/HTTP%20Methods/put-vs-post/



### 9.5 POST

The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line. POST is designed to allow a uniform method to cover the following functions:

```
      - Annotation of existing resources;

      - Posting a message to a bulletin board, newsgroup, mailing list,
        or similar group of articles;

      - Providing a block of data, such as the result of submitting a
        form, to a data-handling process;

      - Extending a database through an append operation.
```

The actual function performed by the POST method is determined by the server and is usually dependent on the Request-URI. The posted entity is subordinate to that URI in the same way that a file is subordinate to a directory containing it, a news article is subordinate to a newsgroup to which it is posted, or a record is subordinate to a database.

The action performed by the POST method might not result in a resource that can be identified by a URI. In this case, either 200 (OK) or 204 (No Content) is the appropriate response status, depending on whether or not the response includes an entity that describes the result.

If a resource has been created on the origin server, the response SHOULD be 201 (Created) and contain an entity which describes the status of the request and refers to the new resource, and a Location header (see section 14.30).

Responses to this method are not cacheable, unless the response includes appropriate Cache-Control or Expires header fields. However, the 303 (See Other) response can be used to direct the user agent to retrieve a cacheable resource.

POST requests MUST obey the message transmission requirements set out in section 8.2.

See section 15.1.3 for security considerations.

### 9.6 PUT

The PUT method requests that the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource, the enclosed entity SHOULD be considered as a modified version of the one residing on the origin server. If the Request-URI does not point to an existing resource, and that URI is capable of being defined as a new resource by the requesting user agent, the origin server can create the resource with that URI. If a new resource is created, the origin server MUST inform the user agent via the 201 (Created) response. If an existing resource is modified, either the 200 (OK) or 204 (No Content) response codes SHOULD be sent to indicate successful completion of the request. If the resource could not be created or modified with the Request-URI, an appropriate error response SHOULD be given that reflects the nature of the problem. The recipient of the entity MUST NOT ignore any Content-* (e.g. Content-Range) headers that it does not understand or implement and MUST return a 501 (Not Implemented) response in such cases.

If the request passes through a cache and the Request-URI identifies one or more currently cached entities, those entries SHOULD be treated as stale. Responses to this method are not cacheable.

The fundamental difference between the POST and PUT requests is reflected in the different meaning of the Request-URI. The URI in a POST request identifies the resource that will handle the enclosed entity. That resource might be a data-accepting process, a gateway to some other protocol, or a separate entity that accepts annotations. In contrast, the URI in a PUT request identifies the entity enclosed with the request -- the user agent knows what URI is intended and the server MUST NOT attempt to apply the request to some other resource. If the server desires that the request be applied to a different URI,

it MUST send a 301 (Moved Permanently) response; the user agent MAY then make its own decision regarding whether or not to redirect the request.

A single resource MAY be identified by many different URIs. For example, an article might have a URI for identifying "the current version" which is separate from the URI identifying each particular version. In this case, a PUT request on a general URI might result in several other URIs being defined by the origin server.

HTTP/1.1 does not define how a PUT method affects the state of an origin server.

PUT requests MUST obey the message transmission requirements set out in section 8.2.

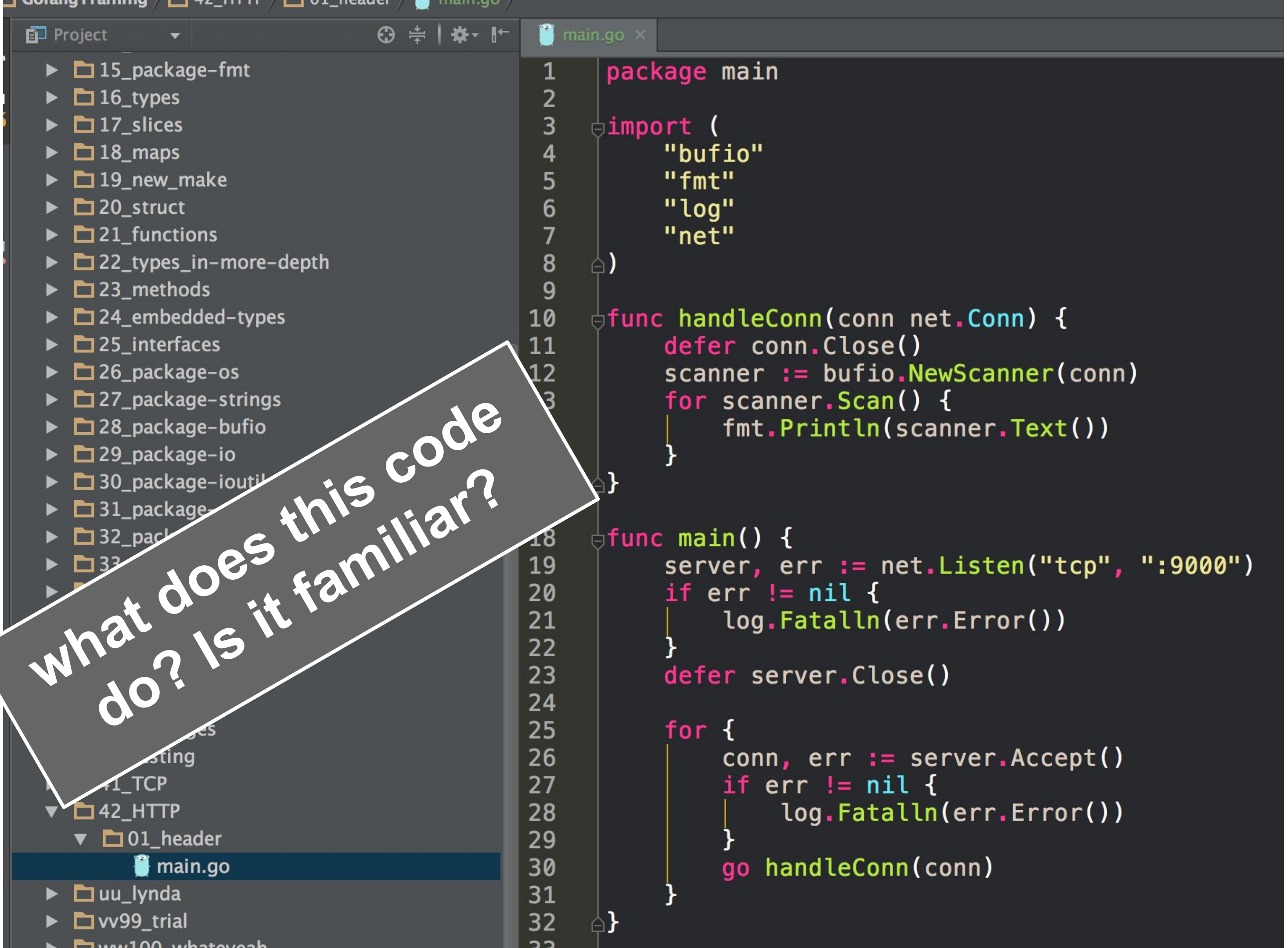
Unless otherwise specified for a particular entity-header, the entity-headers in the PUT request SHOULD be applied to the resource created or modified by the PUT.

## HTTP Headers

- Accept
- Connection
- Content-Type
- Location
- Range
- Referer
- Transfer-Encoding
- WWW-Authenticate

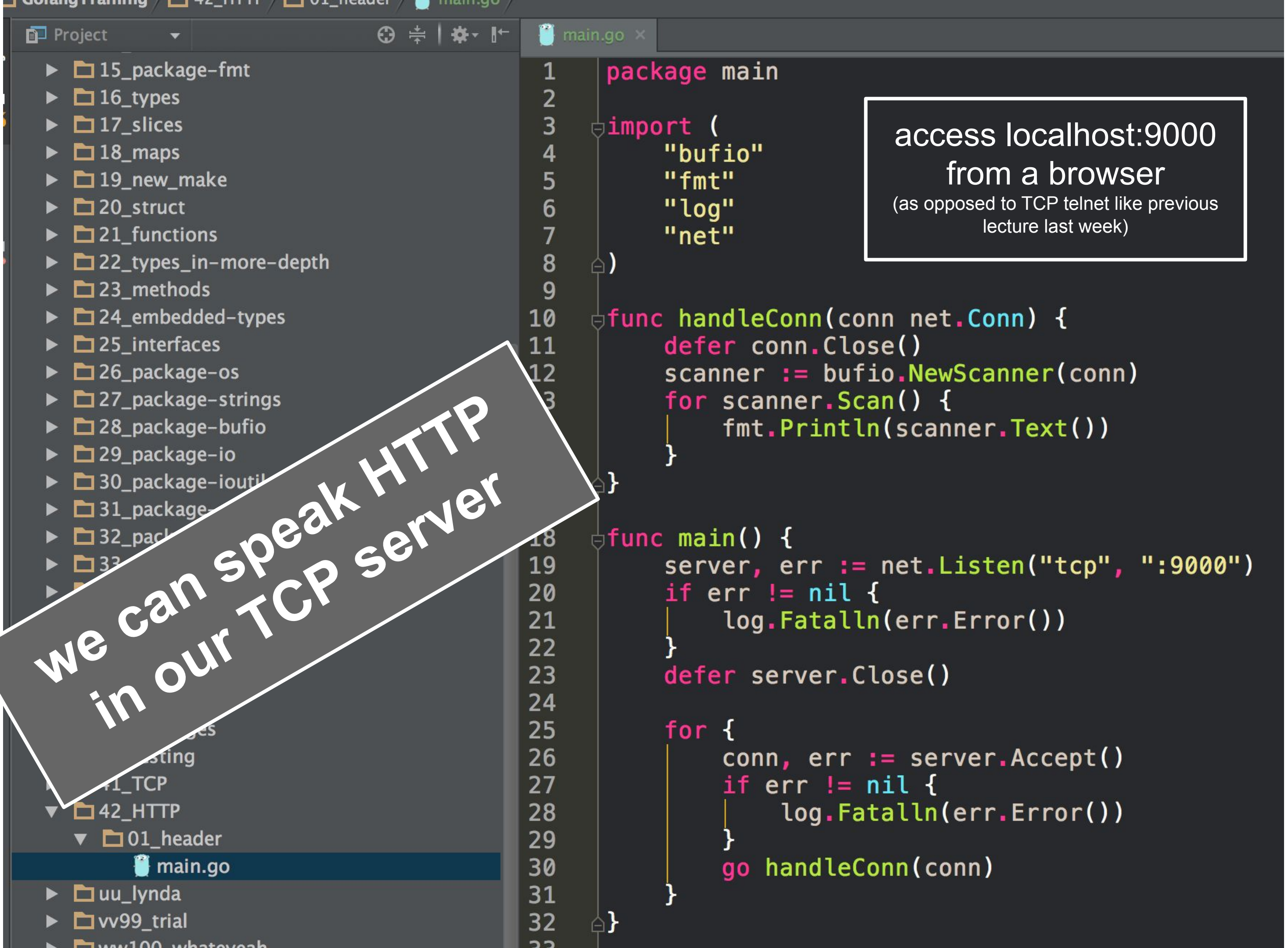
**request and response will both have headers**

read more on wikipedia



```go
package main

import (
	"bufio"
	"fmt"
	"log"
	"net"
)

func handleConn(conn net.Conn) {
	defer conn.Close()
	scanner := bufio.NewScanner(conn)
	for scanner.Scan() {
		fmt.Println(scanner.Text())
	}
}

func main() {
	server, err := net.Listen("tcp", ":9000")
	if err != nil {
		log.Fatalln(err.Error())
	}
	defer server.Close()

	for {
		conn, err := server.Accept()
		if err != nil {
			log.Fatalln(err.Error())
		}
		go handleConn(conn)
	}
}
```

what does this code do? Is it familiar?



```go
package main

import (
	"bufio"
	"fmt"
	"log"
	"net"
)

func handleConn(conn net.Conn) {
	defer conn.Close()
	scanner := bufio.NewScanner(conn)
	for scanner.Scan() {
		fmt.Println(scanner.Text())
	}
}

func main() {
	server, err := net.Listen("tcp", ":9000")
	if err != nil {
		log.Fatalln(err.Error())
	}
	defer server.Close()

	for {
		conn, err := server.Accept()
		if err != nil {
			log.Fatalln(err.Error())
		}
		go handleConn(conn)
	}
}
```

access localhost:9000 from a browser

(as opposed to TCP telnet like previous lecture last week)

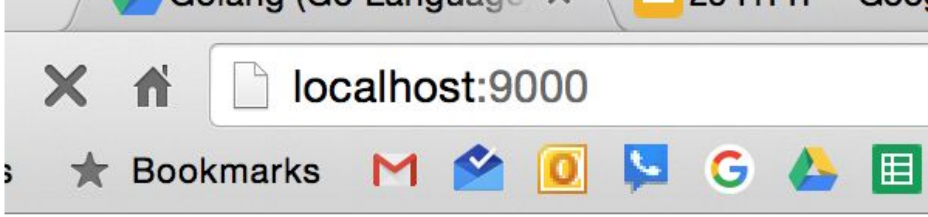
**we can speak HTTP in our TCP server**

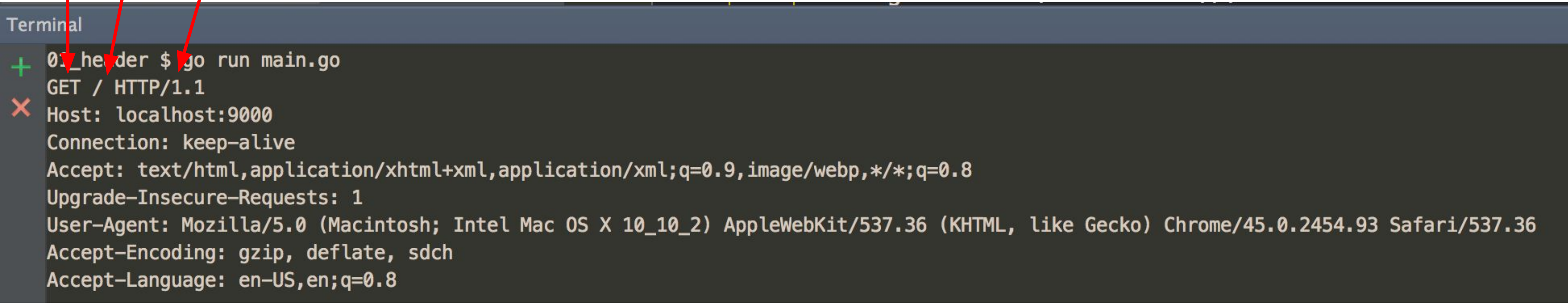
request header

GET request

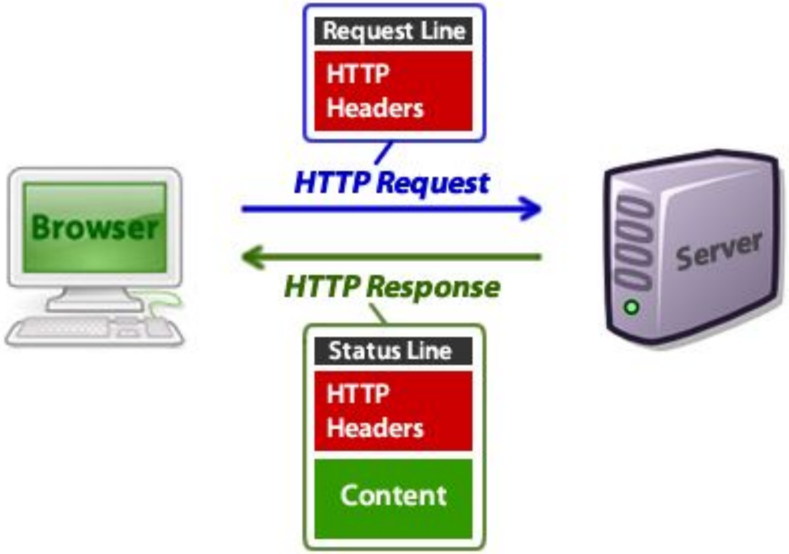
URL

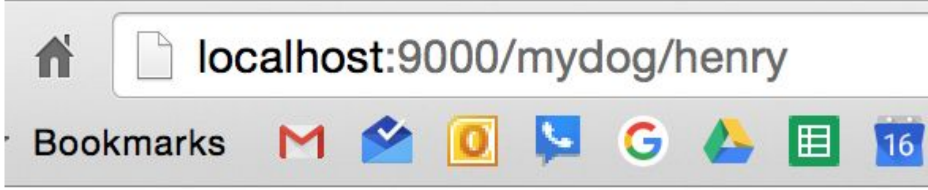
HTTP Protocol Version





```
01_header $ go run main.go
GET / HTTP/1.1
Host: localhost:9000
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.93 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
```

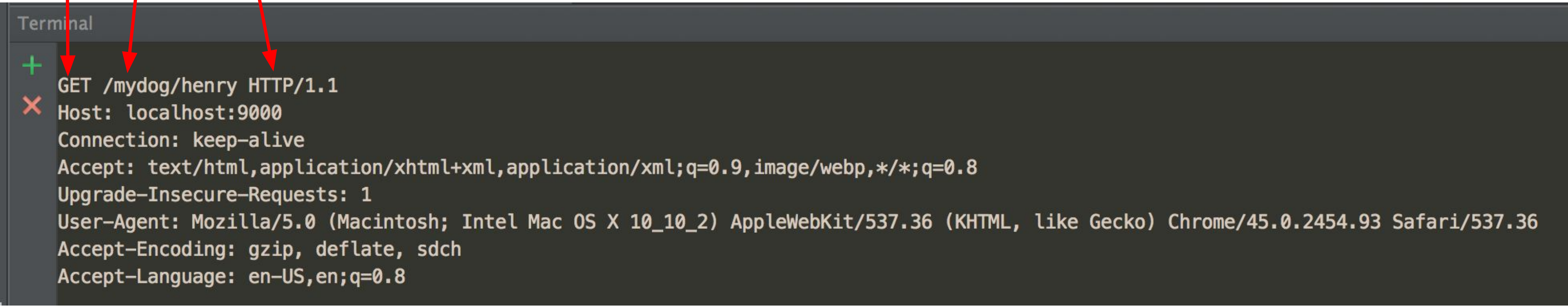

Request Line
HTTP Headers
HTTP Request
Browser
Server
HTTP Response
Status Line
HTTP Headers
Content



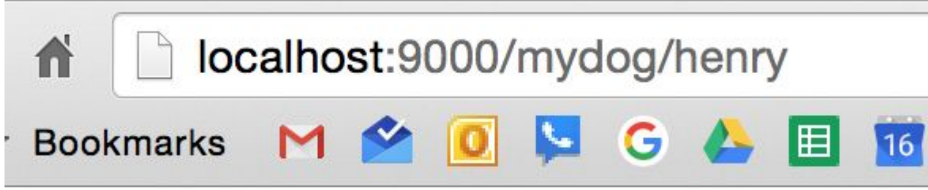
GET request

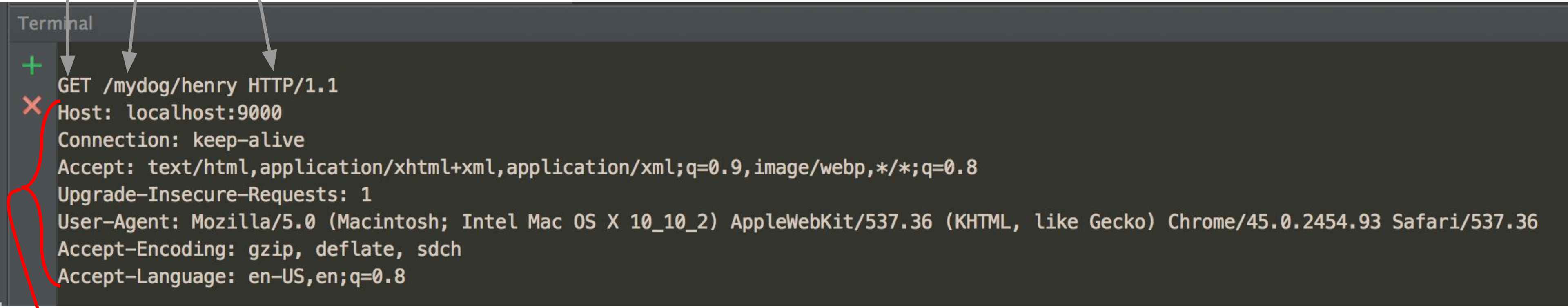
URL

HTTP Protocol Version



```
GET /mydog/henry HTTP/1.1
Host: localhost:9000
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.93 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
```





```
GET /mydog/henry HTTP/1.1
Host: localhost:9000
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.93 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
```

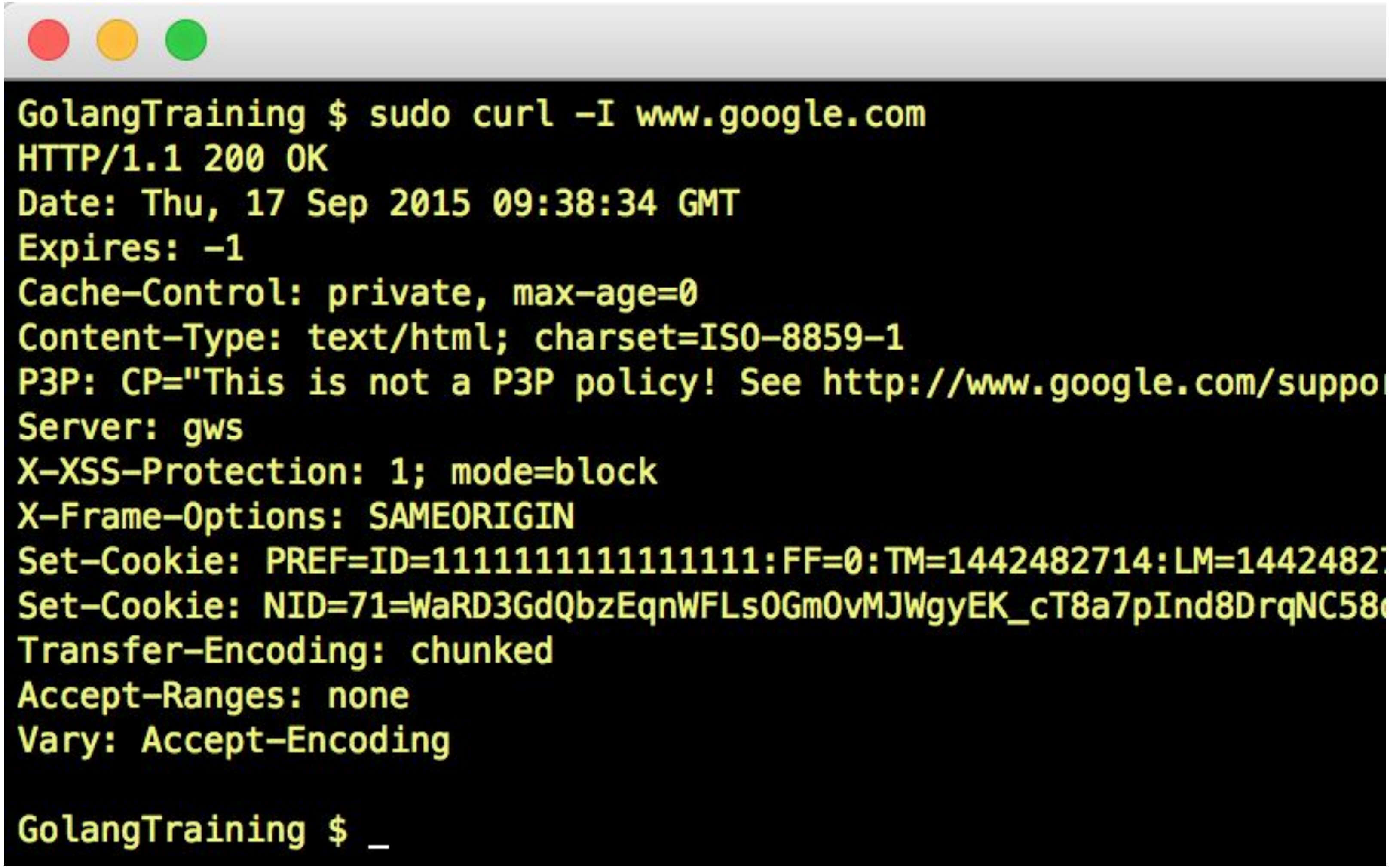
```
Request = Request-Line
          *(header CRLF)
          CRLF
          [ message-body ]
Request-Line = Method Request-URI HTTP-Version CRLF
Header = Name: Value
```

GET - no message-body
POST - message-body

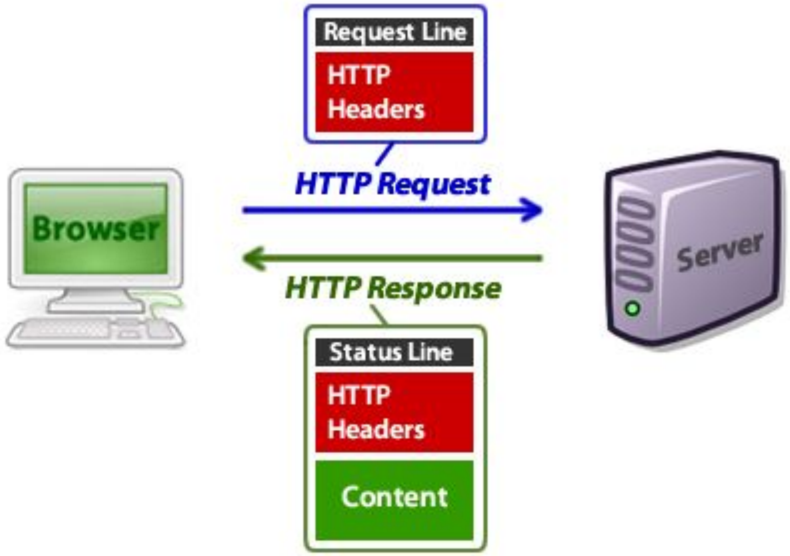
```
Request = Request-Line
          *(header CRLF)
          CRLF
          [ message-body ]
Request-Line = Method Request-URI HTTP-Version CRLF
Header = Name: Value
```

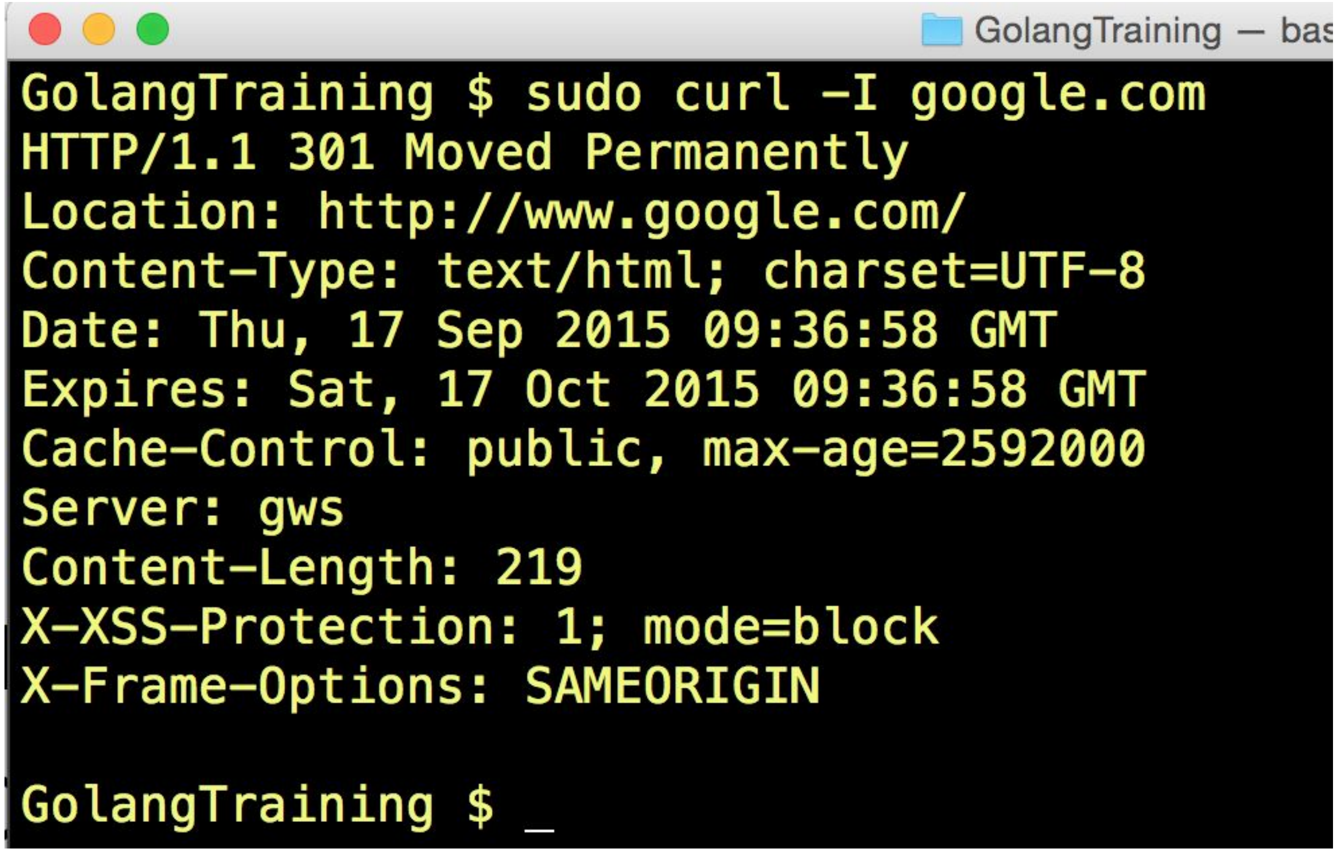
## response header

```
Response = Status-Line
           *(header CRLF)
           CRLF
           [ message-body ]
  Status-Line = HTTP-Version Status-Code Reason-Phrase CRLF
```



```
GolangTraining $ sudo curl -I www.google.com
HTTP/1.1 200 OK
Date: Thu, 17 Sep 2015 09:38:34 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
P3P: CP="This is not a P3P policy! See http://www.google.com/suppo
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Set-Cookie: PREF=ID=1111111111111111:FF=0:TM=1442482714:LM=144248
Set-Cookie: NID=71=WaRD3GdQbzEqnWFLsOGmOvMJWgyEK_cT8a7pInd8DrqNC58
Transfer-Encoding: chunked
Accept-Ranges: none
Vary: Accept-Encoding

GolangTraining $ _
```


Request Line
HTTP Headers
HTTP Request
Browser
Server
HTTP Response
Status Line
HTTP Headers
Content

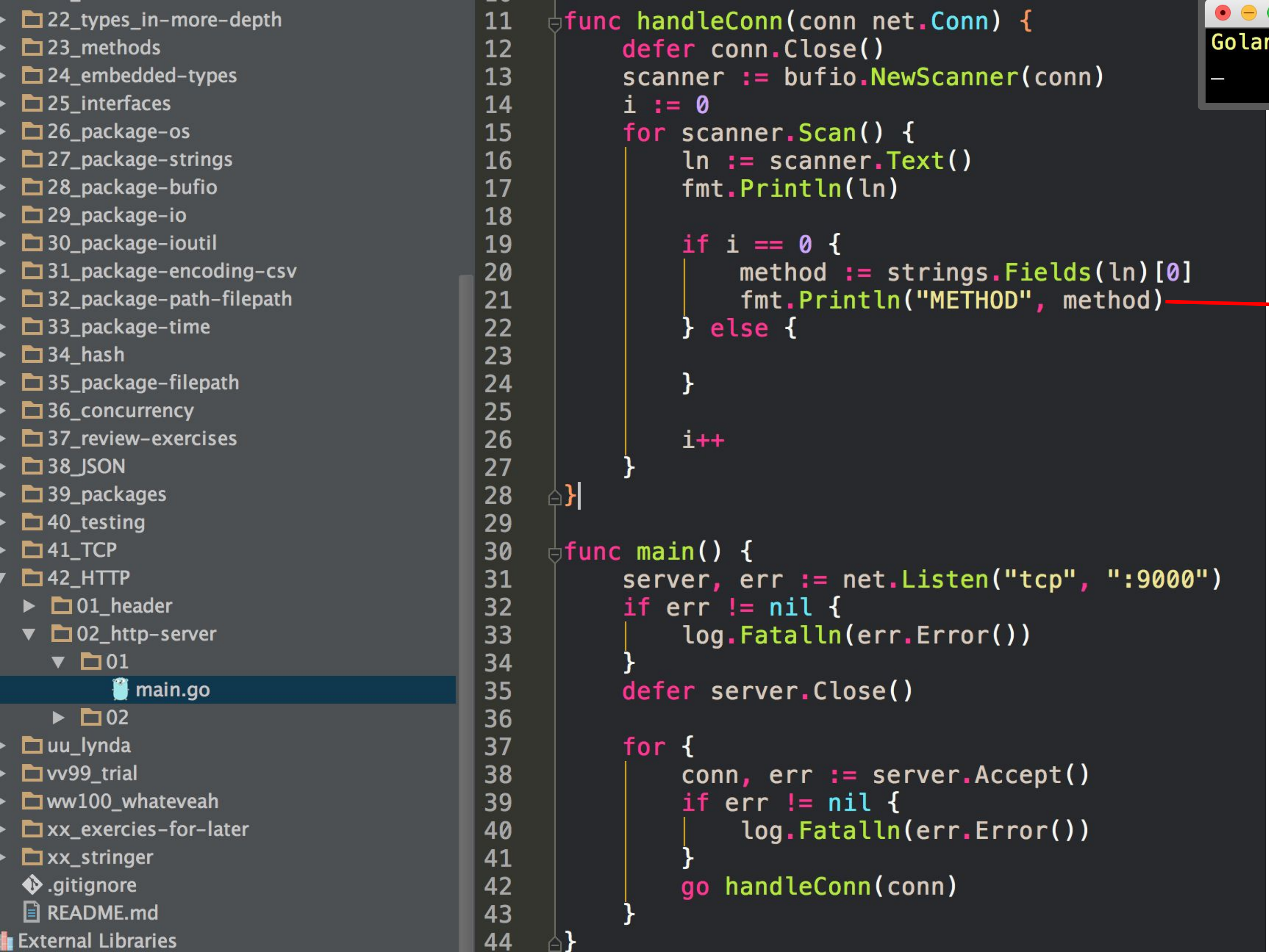


```
GolangTraining $ sudo curl -I google.com
HTTP/1.1 301 Moved Permanently
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Date: Thu, 17 Sep 2015 09:36:58 GMT
Expires: Sat, 17 Oct 2015 09:36:58 GMT
Cache-Control: public, max-age=2592000
Server: gws
Content-Length: 219
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

GolangTraining $ _
```

## To make an HTTP server

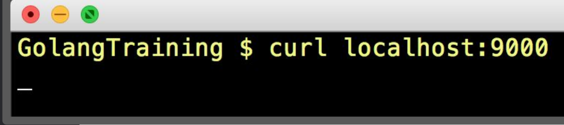
- parse the request
- serve the response

## HTTP server from scratch

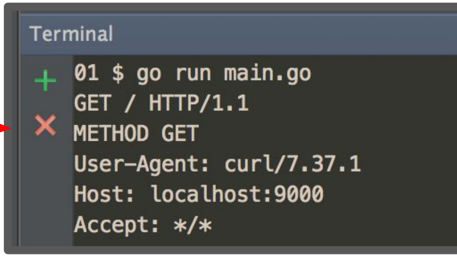


```
22_types_in-more-depth
23_methods
24_embedded-types
25_interfaces
26_package-os
27_package-strings
28_package-bufio
29_package-io
30_package-ioutil
31_package-encoding-csv
32_package-path-filepath
33_package-time
34_hash
35_package-filepath
36_concurrency
37_review-exercises
38_JSON
39_packages
40_testing
41_TCP
42_HTTP
    01_header
    02_http-server
        01
            main.go
        02
uu_lynda
vv99_trial
ww100_whateveah
xx_exercies-for-later
xx_stringer
.gitignore
README.md
External Libraries
```

```go
func handleConn(conn net.Conn) {
	defer conn.Close()
	scanner := bufio.NewScanner(conn)
	i := 0
	for scanner.Scan() {
		ln := scanner.Text()
		fmt.Println(ln)

		if i == 0 {
			method := strings.Fields(ln)[0]
			fmt.Println("METHOD", method)
		} else {

		}

		i++
	}
}

func main() {
	server, err := net.Listen("tcp", ":9000")
	if err != nil {
		log.Fatalln(err.Error())
	}
	defer server.Close()

	for {
		conn, err := server.Accept()
		if err != nil {
			log.Fatalln(err.Error())
		}
		go handleConn(conn)
	}
}
```

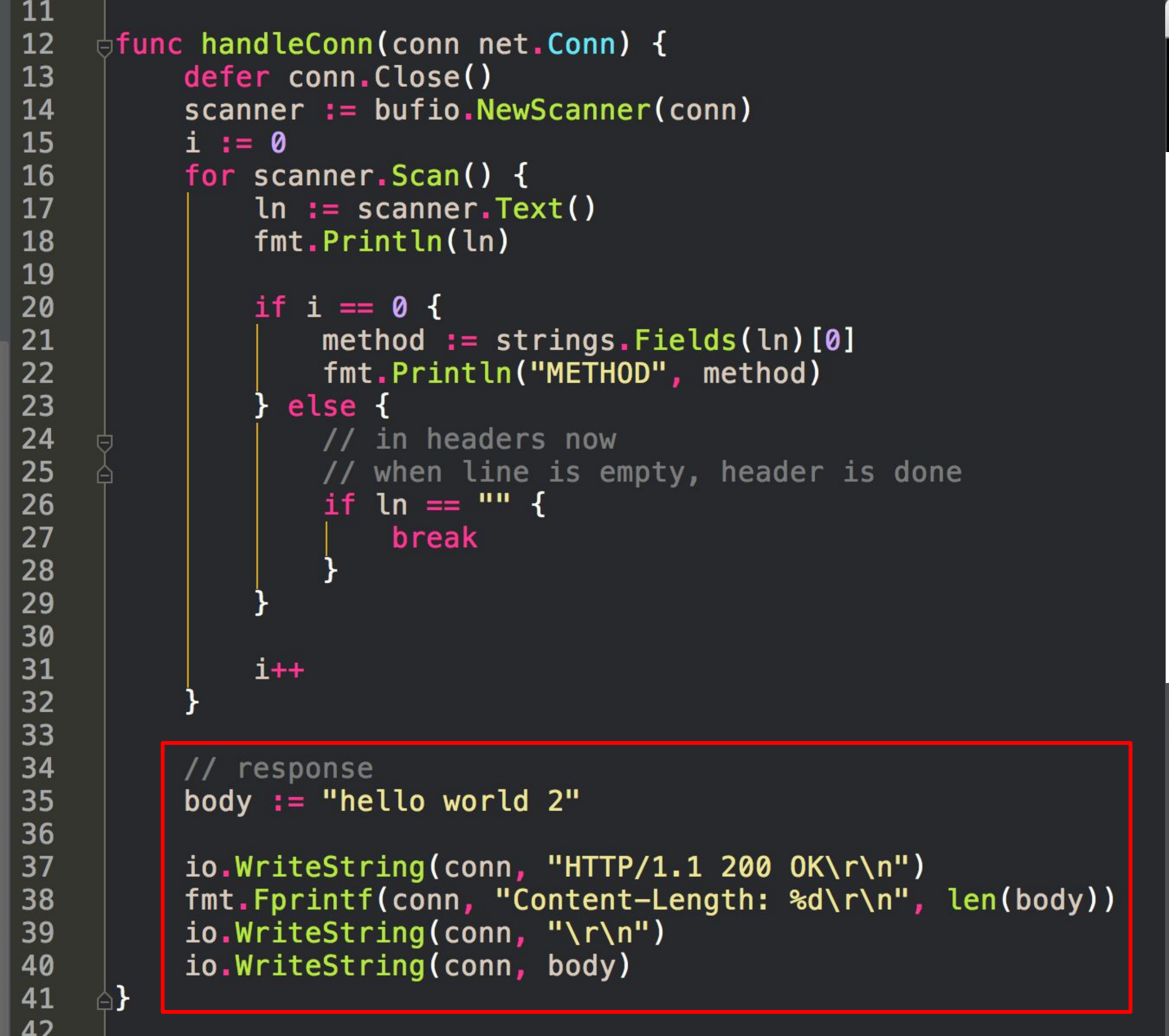


```
GolangTraining $ curl localhost:9000
_
```

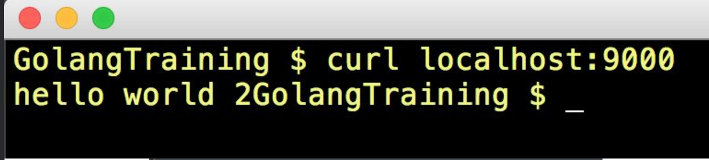


Terminal

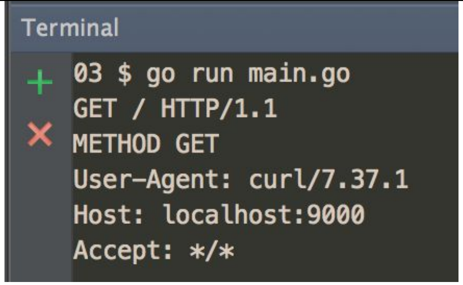
```
01 $ go run main.go
GET / HTTP/1.1
METHOD GET
User-Agent: curl/7.37.1
Host: localhost:9000
Accept: */*
```



```go
func handleConn(conn net.Conn) {
	defer conn.Close()
	scanner := bufio.NewScanner(conn)
	i := 0
	for scanner.Scan() {
		ln := scanner.Text()
		fmt.Println(ln)

		if i == 0 {
			method := strings.Fields(ln)[0]
			fmt.Println("METHOD", method)
		} else {
			// in headers now
			// when line is empty, header is done
			if ln == "" {
				break
			}
		}

		i++
	}

	// response
	body := "hello world 2"

	io.WriteString(conn, "HTTP/1.1 200 OK\r\n")
	fmt.Fprintf(conn, "Content-Length: %d\r\n", len(body))
	io.WriteString(conn, "\r\n")
	io.WriteString(conn, body)
}
```

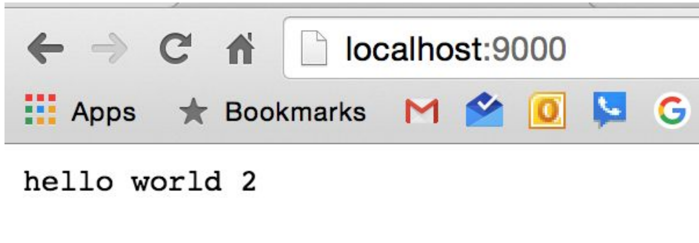


```
GolangTraining $ curl localhost:9000
hello world 2GolangTraining $ _
```

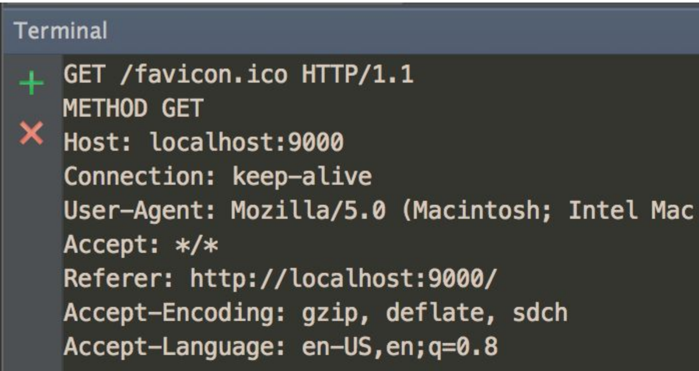


Terminal

```
03 $ go run main.go
GET / HTTP/1.1
METHOD GET
User-Agent: curl/7.37.1
Host: localhost:9000
Accept: */*
```

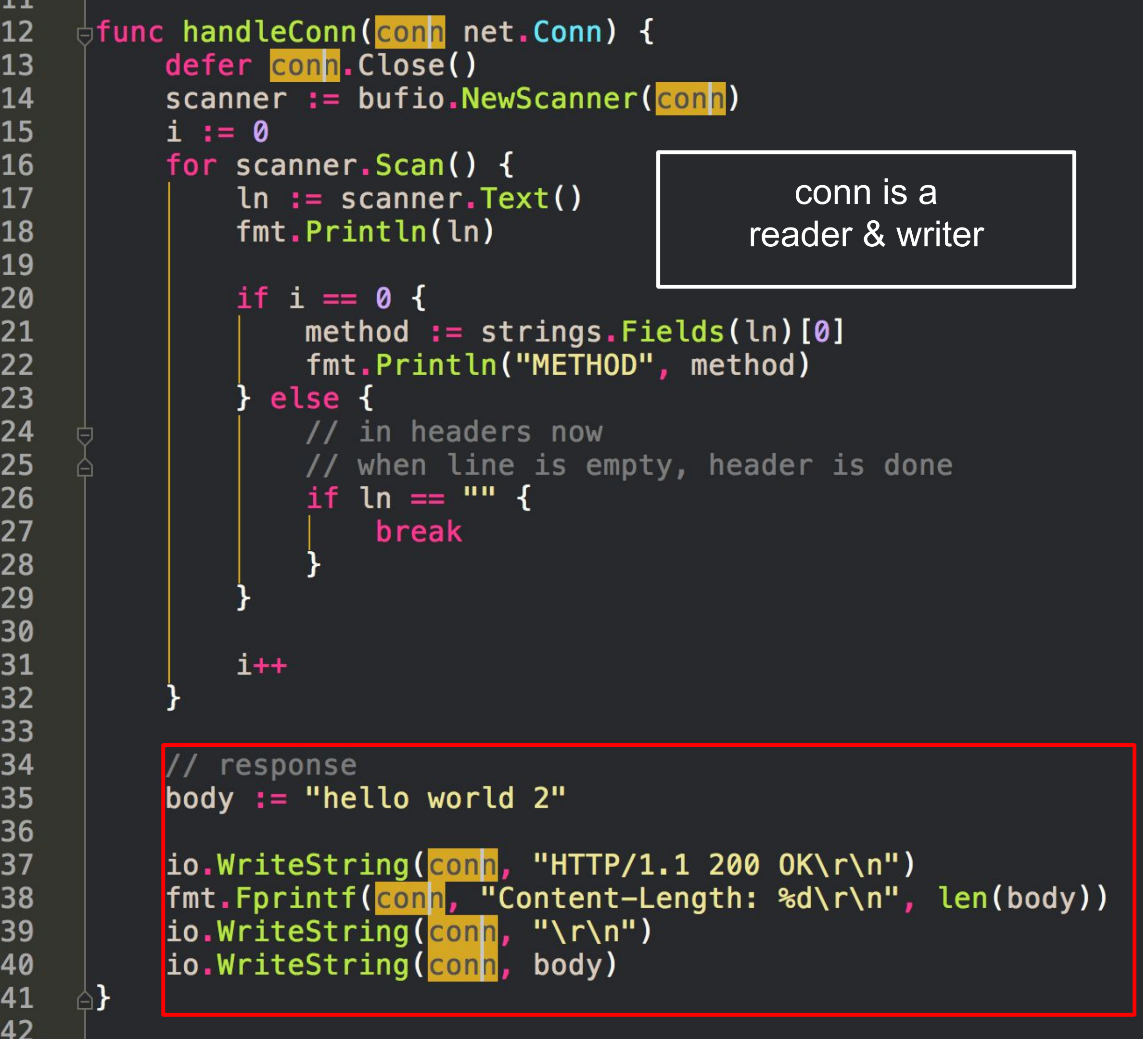


```
hello world 2
```



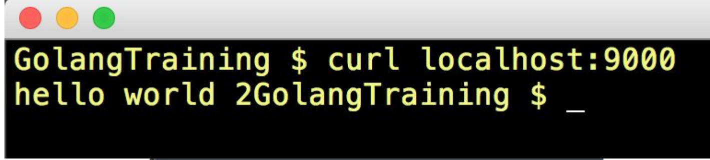
Terminal

```
GET /favicon.ico HTTP/1.1
METHOD GET
Host: localhost:9000
Connection: keep-alive
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac
Accept: */*
Referer: http://localhost:9000/
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
```

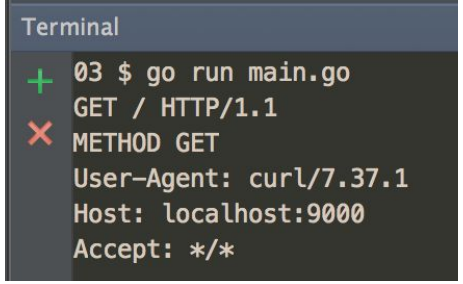


```go
func handleConn(conn net.Conn) {
	defer conn.Close()
	scanner := bufio.NewScanner(conn)
	i := 0
	for scanner.Scan() {
		ln := scanner.Text()
		fmt.Println(ln)

		if i == 0 {
			method := strings.Fields(ln)[0]
			fmt.Println("METHOD", method)
		} else {
			// in headers now
			// when line is empty, header is done
			if ln == "" {
				break
			}
		}

		i++
	}

	// response
	body := "hello world 2"

	io.WriteString(conn, "HTTP/1.1 200 OK\r\n")
	fmt.Fprintf(conn, "Content-Length: %d\r\n", len(body))
	io.WriteString(conn, "\r\n")
	io.WriteString(conn, body)
}
```

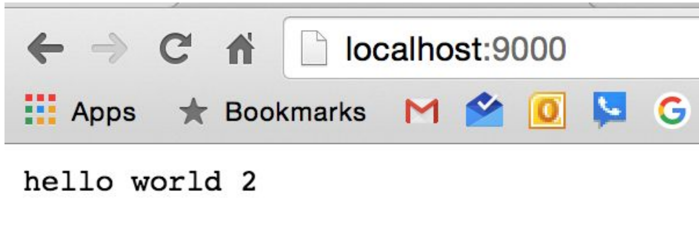
conn is a
reader & writer



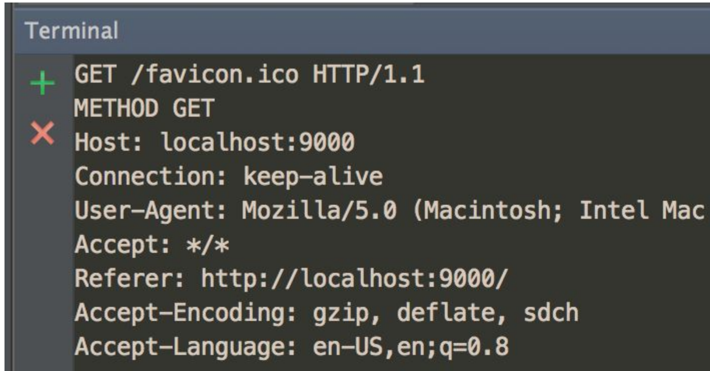
```
GolangTraining $ curl localhost:9000
hello world 2GolangTraining $ _
```



```
Terminal
03 $ go run main.go
GET / HTTP/1.1
METHOD GET
User-Agent: curl/7.37.1
Host: localhost:9000
Accept: */*
```

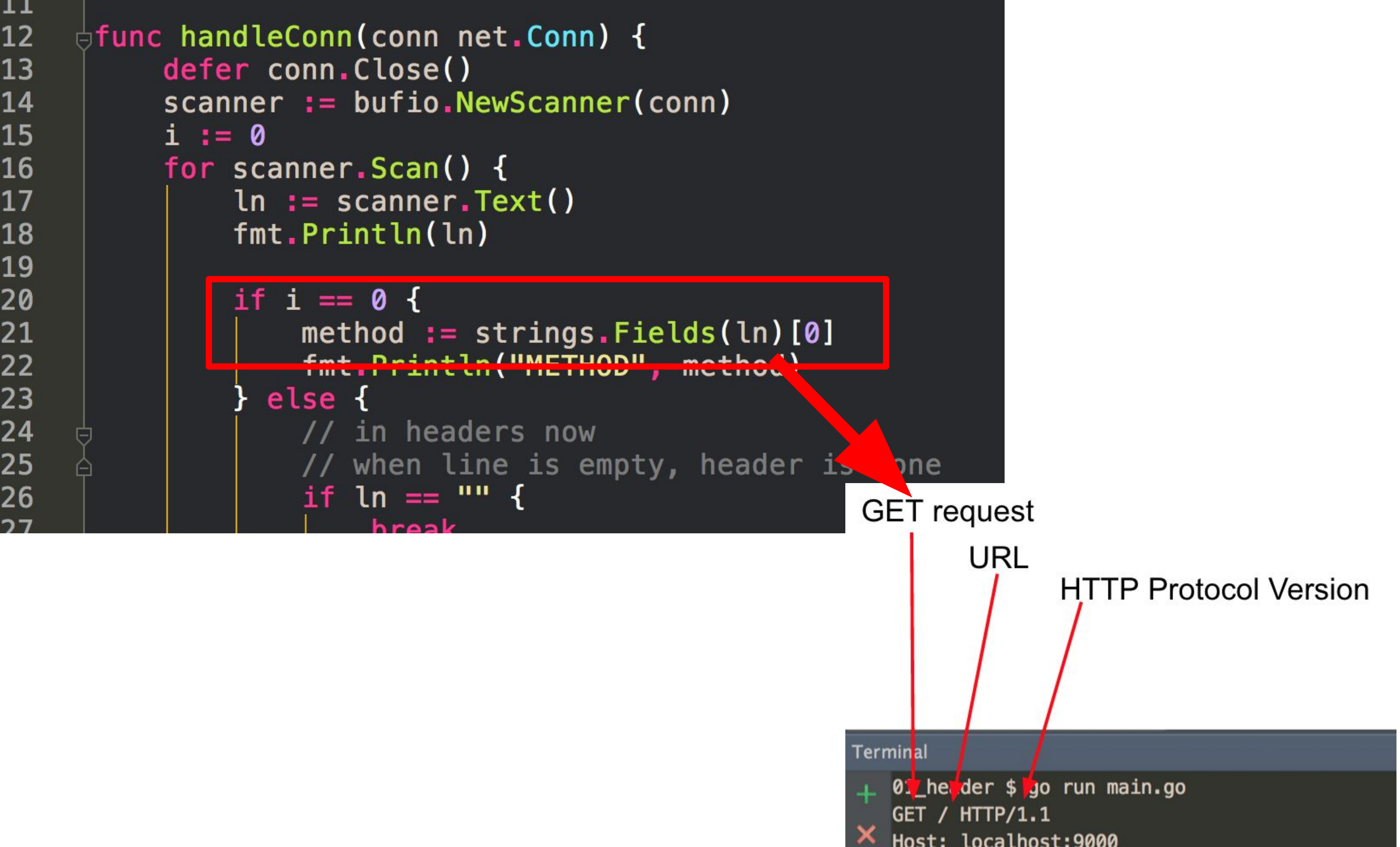


```
hello world 2
```



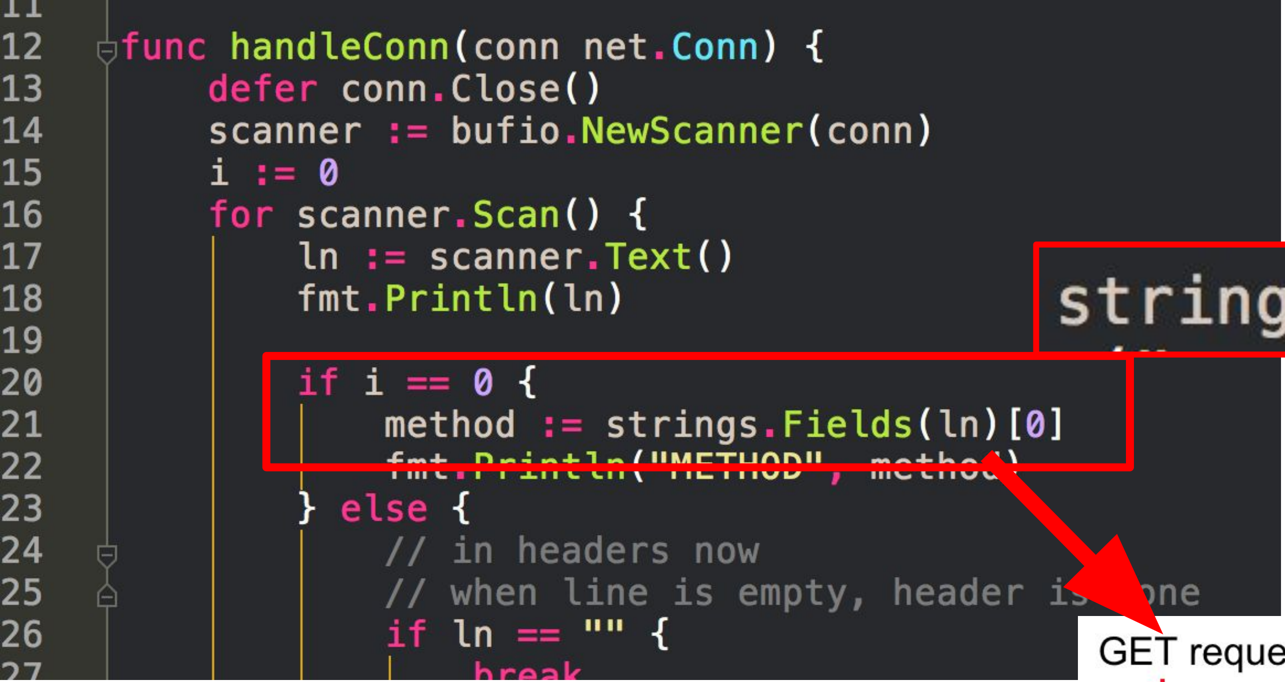
```
Terminal
GET /favicon.ico HTTP/1.1
METHOD GET
Host: localhost:9000
Connection: keep-alive
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac
Accept: */*
Referer: http://localhost:9000/
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
```

you can keep building from scratch

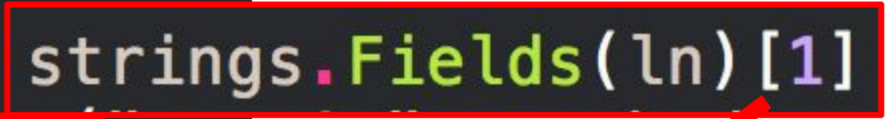


```go
func handleConn(conn net.Conn) {
	defer conn.Close()
	scanner := bufio.NewScanner(conn)
	i := 0
	for scanner.Scan() {
		ln := scanner.Text()
		fmt.Println(ln)

		if i == 0 {
			method := strings.Fields(ln)[0]
			fmt.Println("METHOD", method)
		} else {
			// in headers now
			// when line is empty, header is done
			if ln == "" {
				break
```

```
01_header $ go run main.go
GET / HTTP/1.1
Host: localhost:9000
```

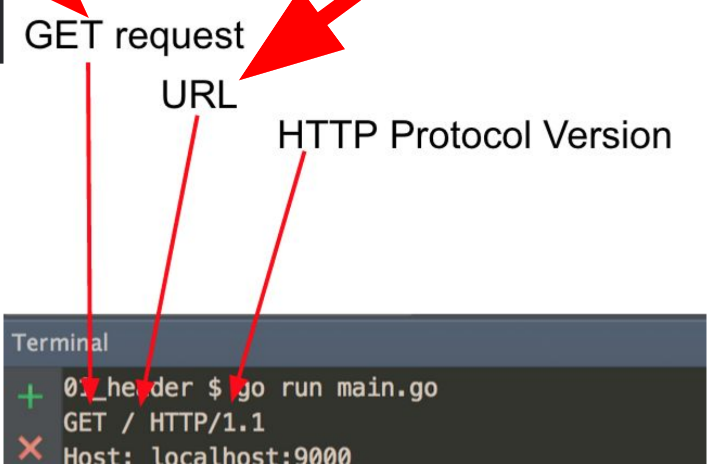


```go
func handleConn(conn net.Conn) {
	defer conn.Close()
	scanner := bufio.NewScanner(conn)
	i := 0
	for scanner.Scan() {
		ln := scanner.Text()
		fmt.Println(ln)

		if i == 0 {
			method := strings.Fields(ln)[0]
			fmt.Println("METHOD", method)
		} else {
			// in headers now
			// when line is empty, header is done
			if ln == "" {
				break
```



```go
strings.Fields(ln)[1]
```

We could then start doing request routing based on the URL



```
01_header $ go run main.go
GET / HTTP/1.1
Host: localhost:9000
```

## The main point here - to make an HTTP server

- parse the request
- serve the response

## exercise

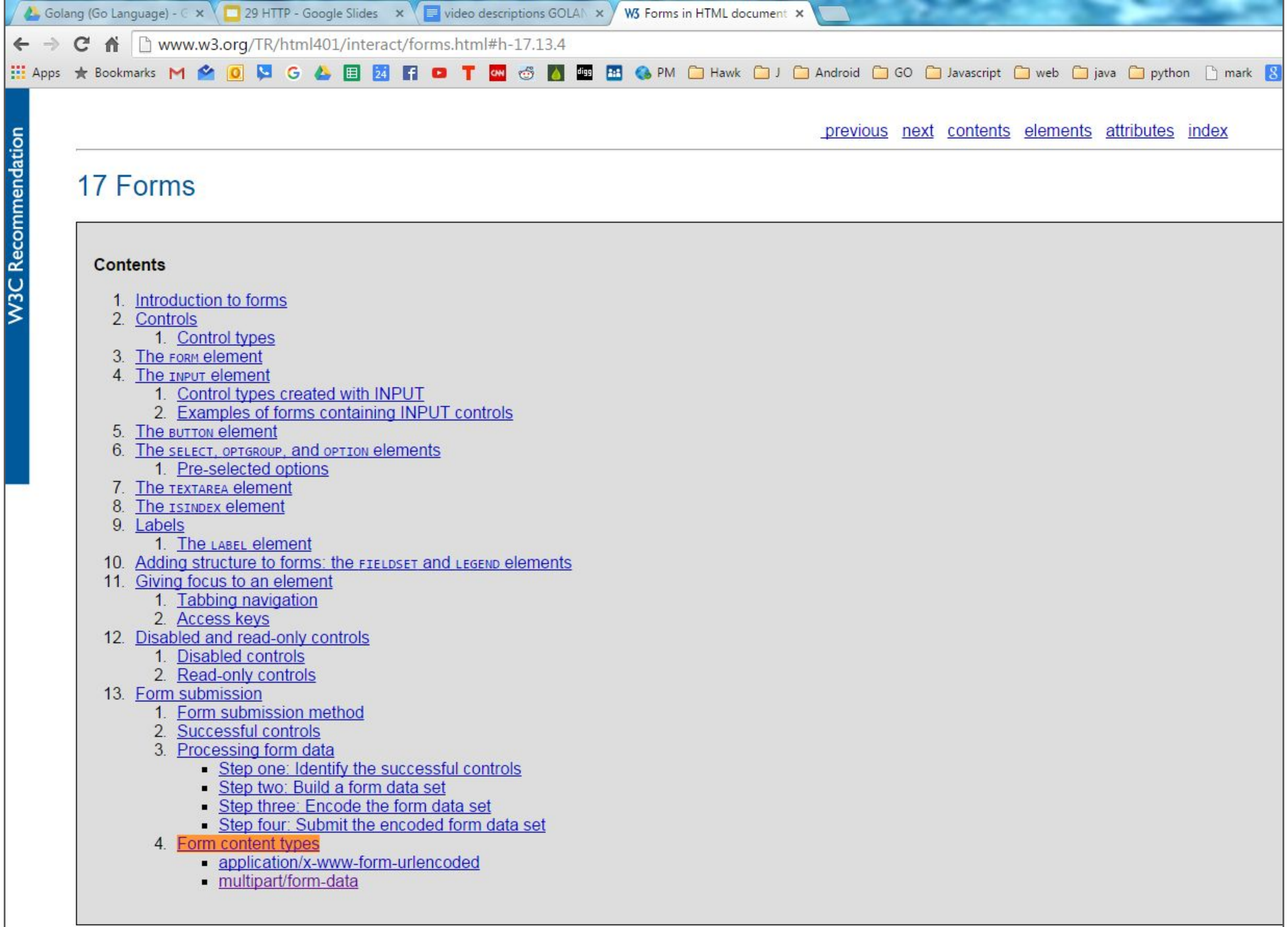
Create a TCP server which can handle a simple HTTP request and return the URL that was passed into it

## HTTP Headers

## HTTP Headers

- Accept
- Connection
- Content-Type
    - application/x-www-form-urlencoded
    - multipart/form-data
    - text/plain
    - text/html
- Location
- Range
- Referer
- Transfer-Encoding
- WWW-Authenticate

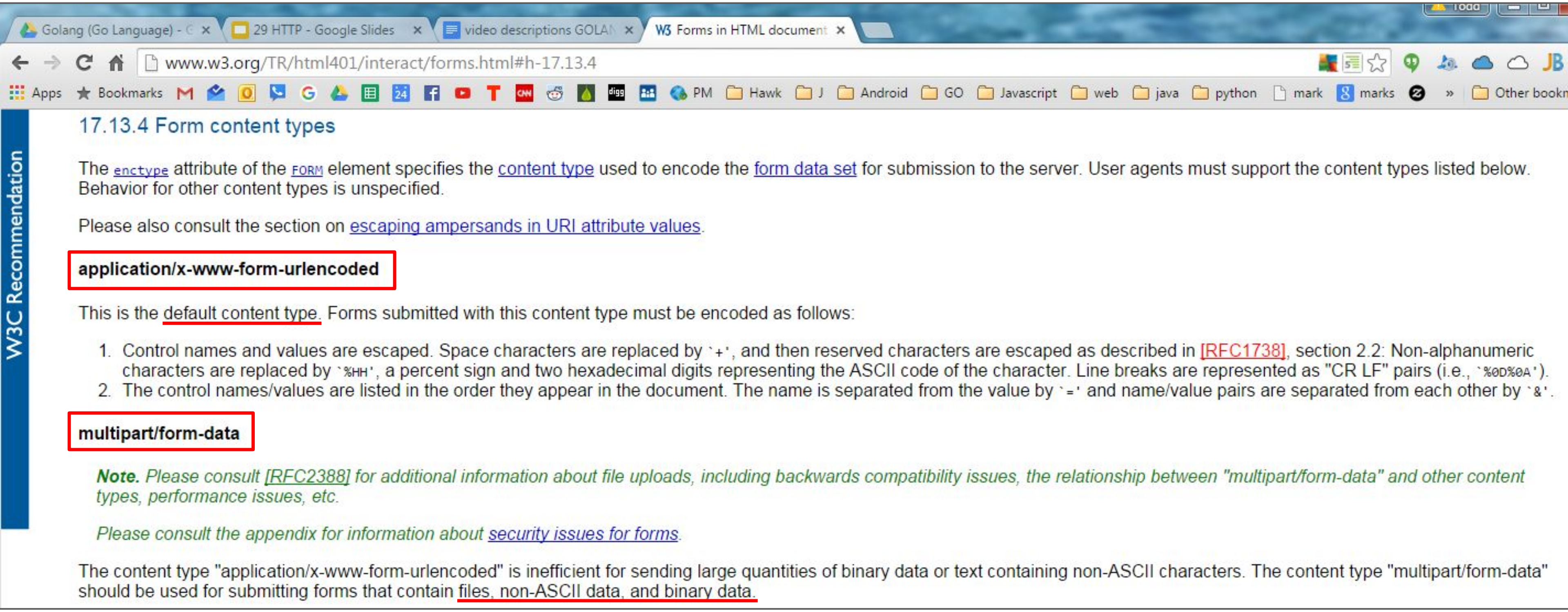
read more on wikipedia



previous next contents elements attributes index

# 17 Forms

**Contents**



## 17.13.4 Form content types

The enctype attribute of the FORM element specifies the content type used to encode the form data set for submission to the server. User agents must support the content types listed below. Behavior for other content types is unspecified.

Please also consult the section on escaping ampersands in URI attribute values.

**application/x-www-form-urlencoded**

This is the default content type. Forms submitted with this content type must be encoded as follows:

1. Control names and values are escaped. Space characters are replaced by `+`, and then reserved characters are escaped as described in [RFC1738], section 2.2: Non-alphanumeric characters are replaced by `%HH`, a percent sign and two hexadecimal digits representing the ASCII code of the character. Line breaks are represented as "CR LF" pairs (i.e., `%0D%0A`).
2. The control names/values are listed in the order they appear in the document. The name is separated from the value by `=` and name/value pairs are separated from each other by `&`.

**multipart/form-data**

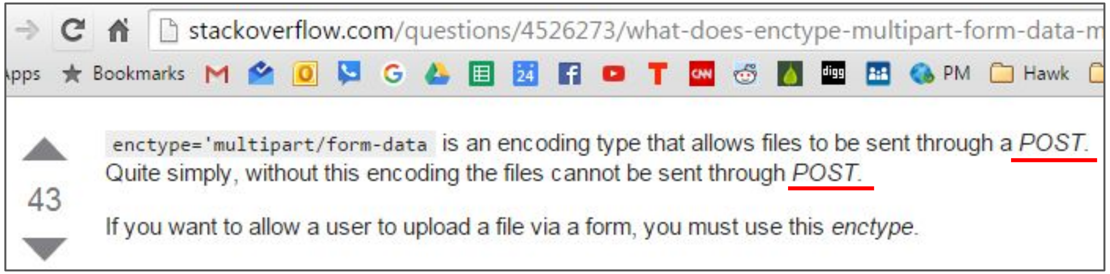
***Note.*** *Please consult [RFC2388] for additional information about file uploads, including backwards compatibility issues, the relationship between "multipart/form-data" and other content types, performance issues, etc.*

*Please consult the appendix for information about security issues for forms.*

The content type "application/x-www-form-urlencoded" is inefficient for sending large quantities of binary data or text containing non-ASCII characters. The content type "multipart/form-data" should be used for submitting forms that contain files, non-ASCII data, and binary data.

## HTTP Verbs

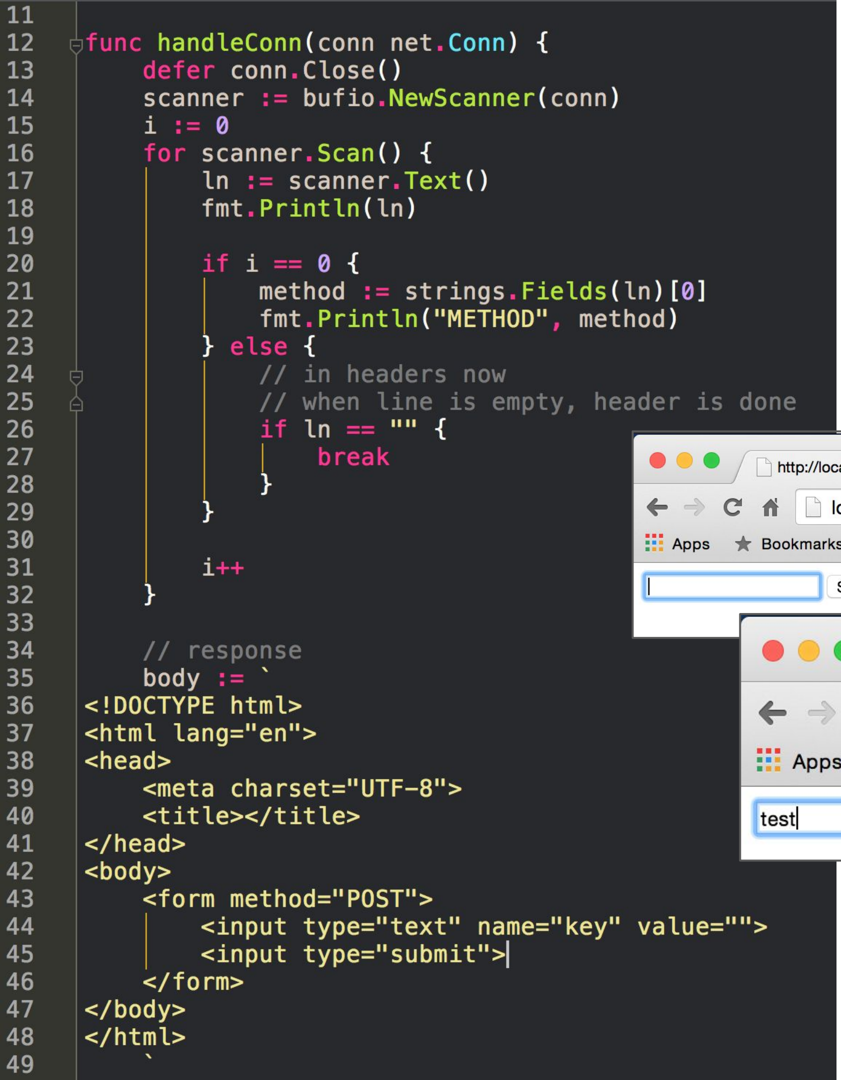
- GET
- POST
- PUT
- DELETE
- HEAD


stackoverflow.com/questions/4526273/what-does-enctype-multipart-form-data-m
Apps
Bookmarks
PM
Hawk
43

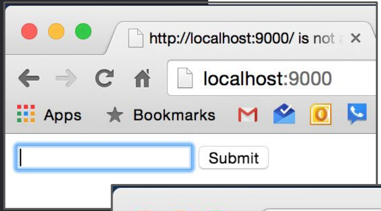
`enctype='multipart/form-data` is an encoding type that allows files to be sent through a *POST*. Quite simply, without this encoding the files cannot be sent through *POST*.

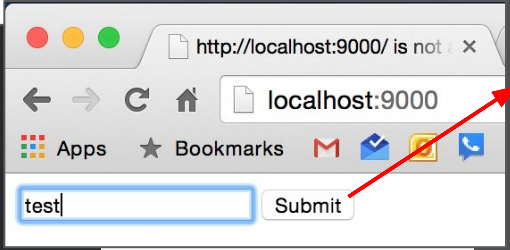
If you want to allow a user to upload a file via a form, you must use this *enctype*.

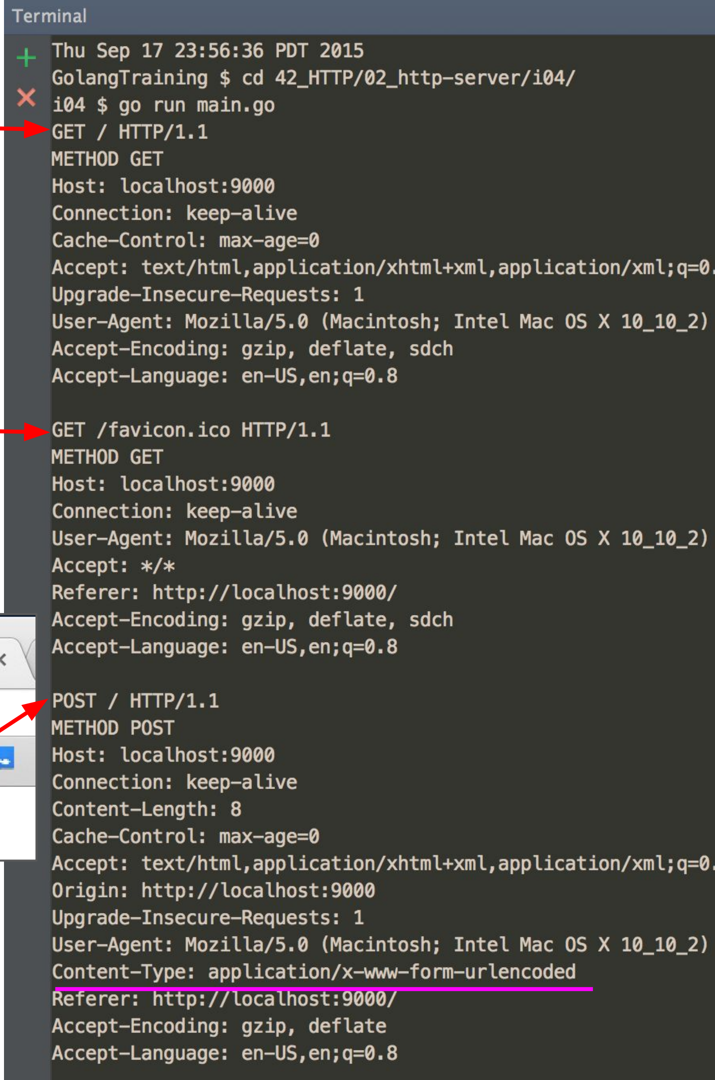
read more on wikipedia



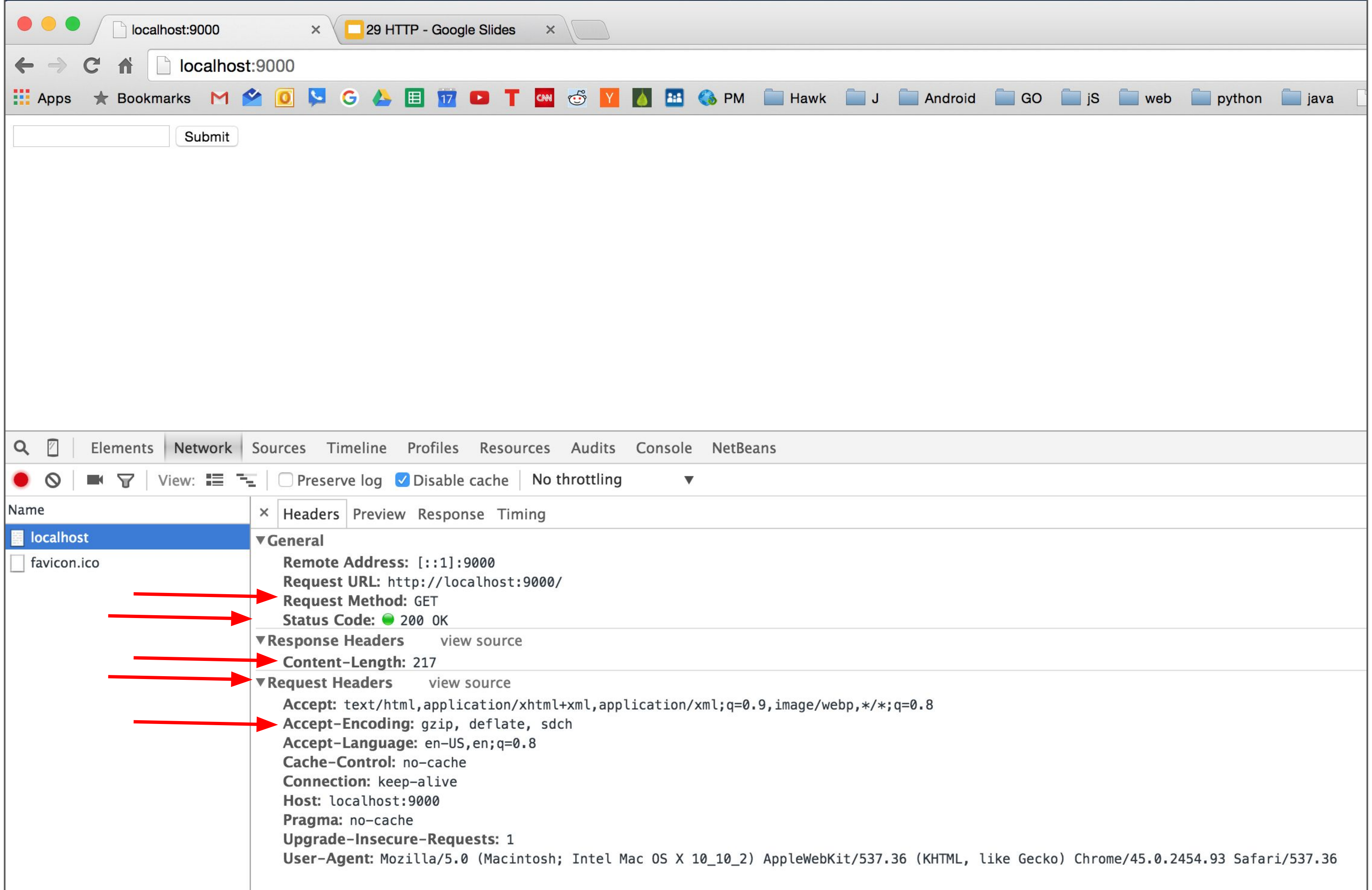
```go
func handleConn(conn net.Conn) {
    defer conn.Close()
    scanner := bufio.NewScanner(conn)
    i := 0
    for scanner.Scan() {
        ln := scanner.Text()
        fmt.Println(ln)

        if i == 0 {
            method := strings.Fields(ln)[0]
            fmt.Println("METHOD", method)
        } else {
            // in headers now
            // when line is empty, header is done
            if ln == "" {
                break
            }
        }

        i++
    }

    // response
    body := `
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
</head>
<body>
    <form method="POST">
        <input type="text" name="key" value="">
        <input type="submit">
    </form>
</body>
</html>
`
```







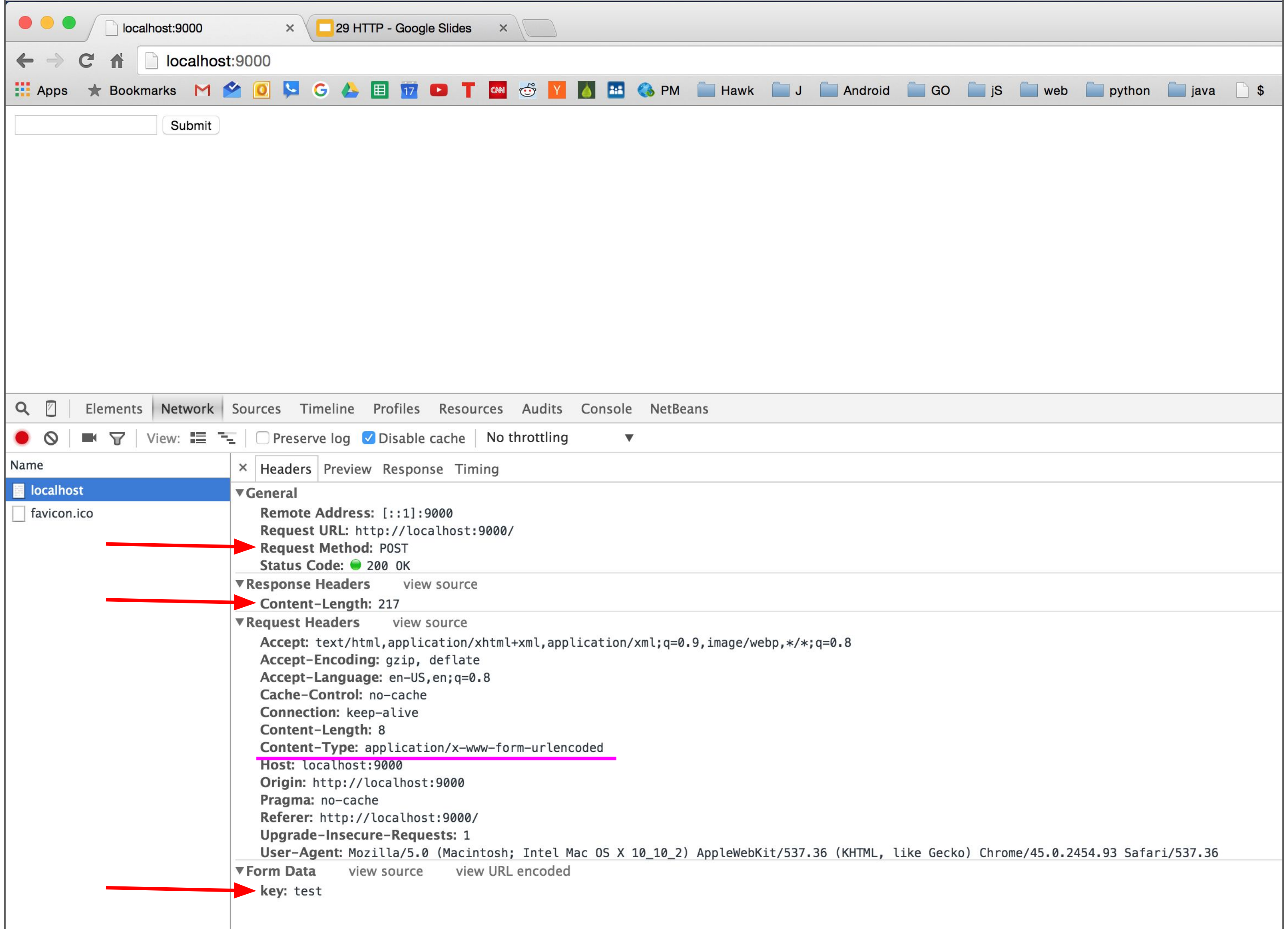
```
Terminal
Thu Sep 17 23:56:36 PDT 2015
GolangTraining $ cd 42_HTTP/02_http-server/i04/
i04 $ go run main.go
GET / HTTP/1.1
METHOD GET
Host: localhost:9000
Connection: keep-alive
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2)
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8

GET /favicon.ico HTTP/1.1
METHOD GET
Host: localhost:9000
Connection: keep-alive
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2)
Accept: */*
Referer: http://localhost:9000/
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8

POST / HTTP/1.1
METHOD POST
Host: localhost:9000
Connection: keep-alive
Content-Length: 8
Cache-Control: max-age=0
Accept: text/html,application/xhtml+xml,application/xml;q=0.
Origin: http://localhost:9000
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2)
Content-Type: application/x-www-form-urlencoded
Referer: http://localhost:9000/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8
```



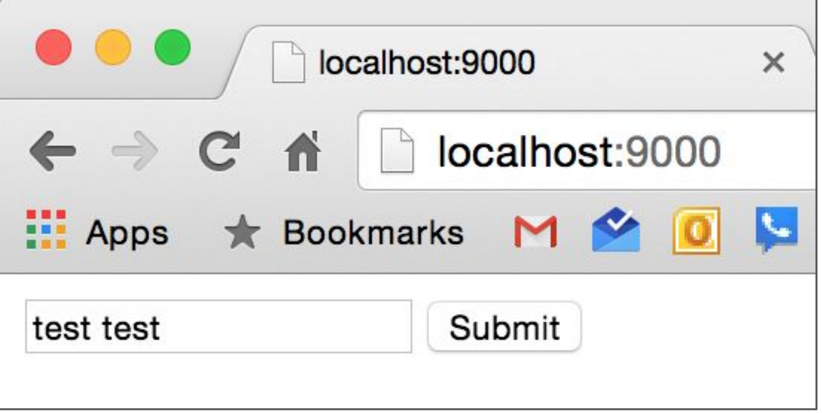
**General**
- Remote Address: [::1]:9000
- Request URL: http://localhost:9000/
- Request Method: GET
- Status Code: ● 200 OK

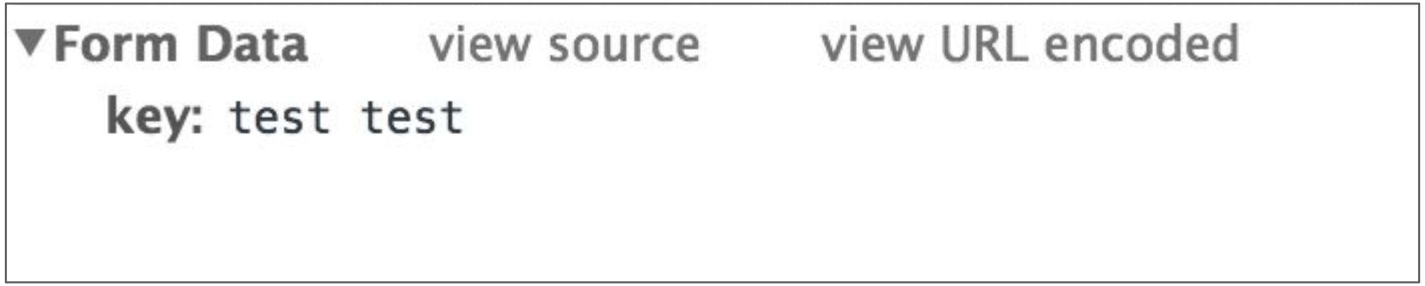
**Response Headers** view source
- Content-Length: 217

**Request Headers** view source
- Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
- Accept-Encoding: gzip, deflate, sdch
- Accept-Language: en-US,en;q=0.8
- Cache-Control: no-cache
- Connection: keep-alive
- Host: localhost:9000
- Pragma: no-cache
- Upgrade-Insecure-Requests: 1
- User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.93 Safari/537.36

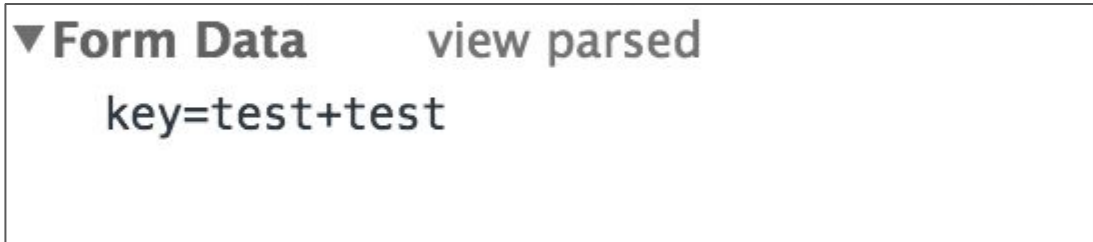


```
▼General
    Remote Address: [::1]:9000
    Request URL: http://localhost:9000/
    Request Method: POST
    Status Code: ● 200 OK
▼Response Headers     view source
    Content-Length: 217
▼Request Headers     view source
    Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
    Accept-Encoding: gzip, deflate
    Accept-Language: en-US,en;q=0.8
    Cache-Control: no-cache
    Connection: keep-alive
    Content-Length: 8
    Content-Type: application/x-www-form-urlencoded
    Host: localhost:9000
    Origin: http://localhost:9000
    Pragma: no-cache
    Referer: http://localhost:9000/
    Upgrade-Insecure-Requests: 1
    User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/45.0.2454.93 Safari/537.36
▼Form Data     view source     view URL encoded
    key: test
```

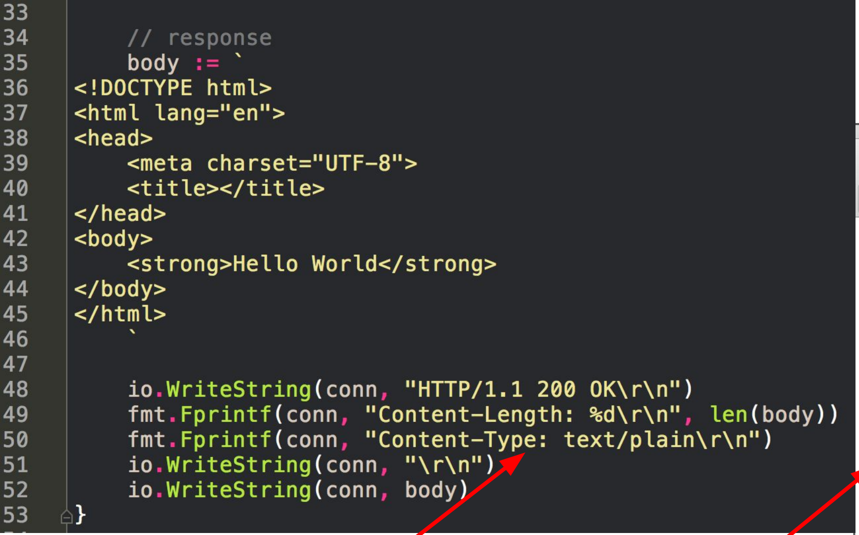




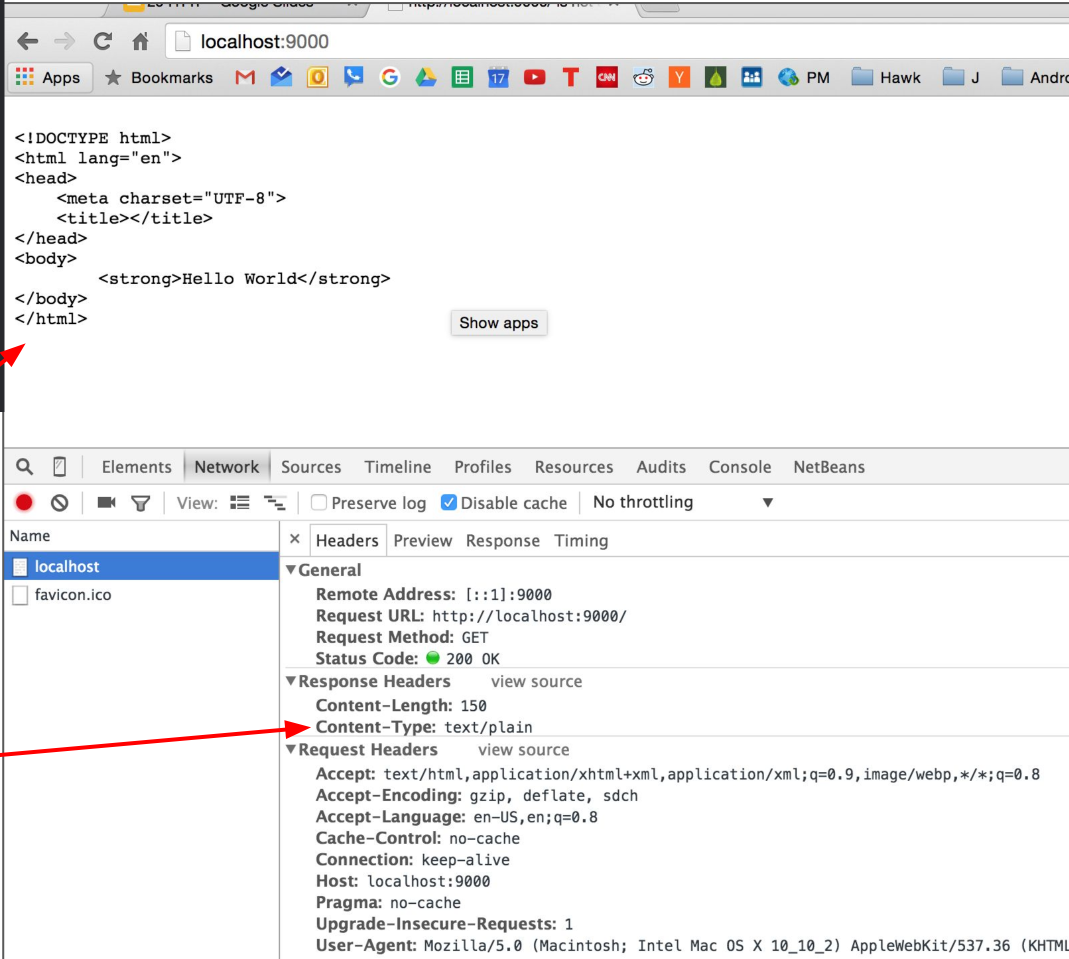
```
key: test test
```



```
key=test+test
```



```
    // response
    body := `
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
</head>
<body>
    <strong>Hello World</strong>
</body>
</html>
    `

    io.WriteString(conn, "HTTP/1.1 200 OK\r\n")
    fmt.Fprintf(conn, "Content-Length: %d\r\n", len(body))
    fmt.Fprintf(conn, "Content-Type: text/plain\r\n")
    io.WriteString(conn, "\r\n")
    io.WriteString(conn, body)
}
```



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title></title>
</head>
<body>
        <strong>Hello World</strong>
</body>
</html>
```

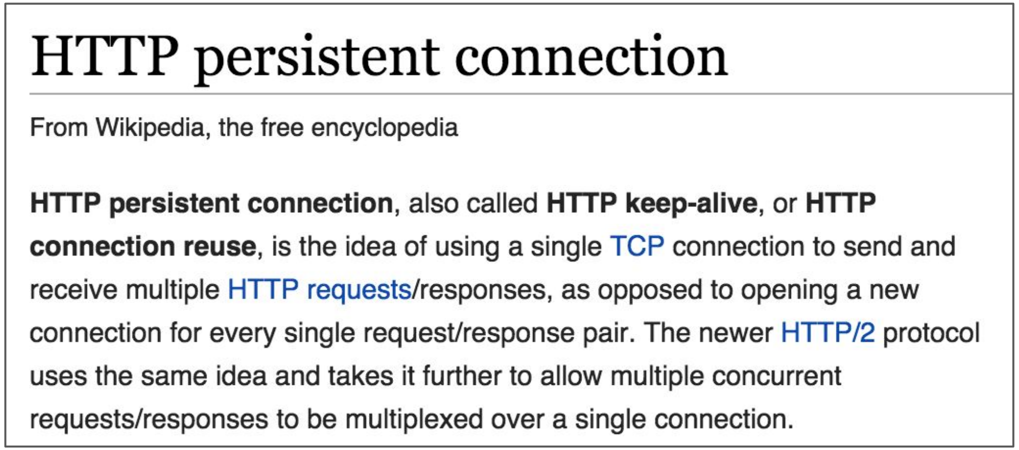
▼General
- **Remote Address:** [::1]:9000
- **Request URL:** http://localhost:9000/
- **Request Method:** GET
- **Status Code:** ● 200 OK

▼Response Headers view source
- **Content-Length:** 150
- **Content-Type:** text/plain

▼Request Headers view source
- **Accept:** text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
- **Accept-Encoding:** gzip, deflate, sdch
- **Accept-Language:** en-US,en;q=0.8
- **Cache-Control:** no-cache
- **Connection:** keep-alive
- **Host:** localhost:9000
- **Pragma:** no-cache
- **Upgrade-Insecure-Requests:** 1
- **User-Agent:** Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) AppleWebKit/537.36 (KHTML

## HTTP Headers

- Accept
- Connection
  - keep alive
    - needs content length
      - otherwise doesn't know when request ends
  - closed
- Content-Type
  - application/x-www-form-urlencoded
  - multipart/form-data
  - text/plain
- Location
- Range
- Referer
- Transfer-Encoding
- WWW-Authenticate



HTTP persistent connection

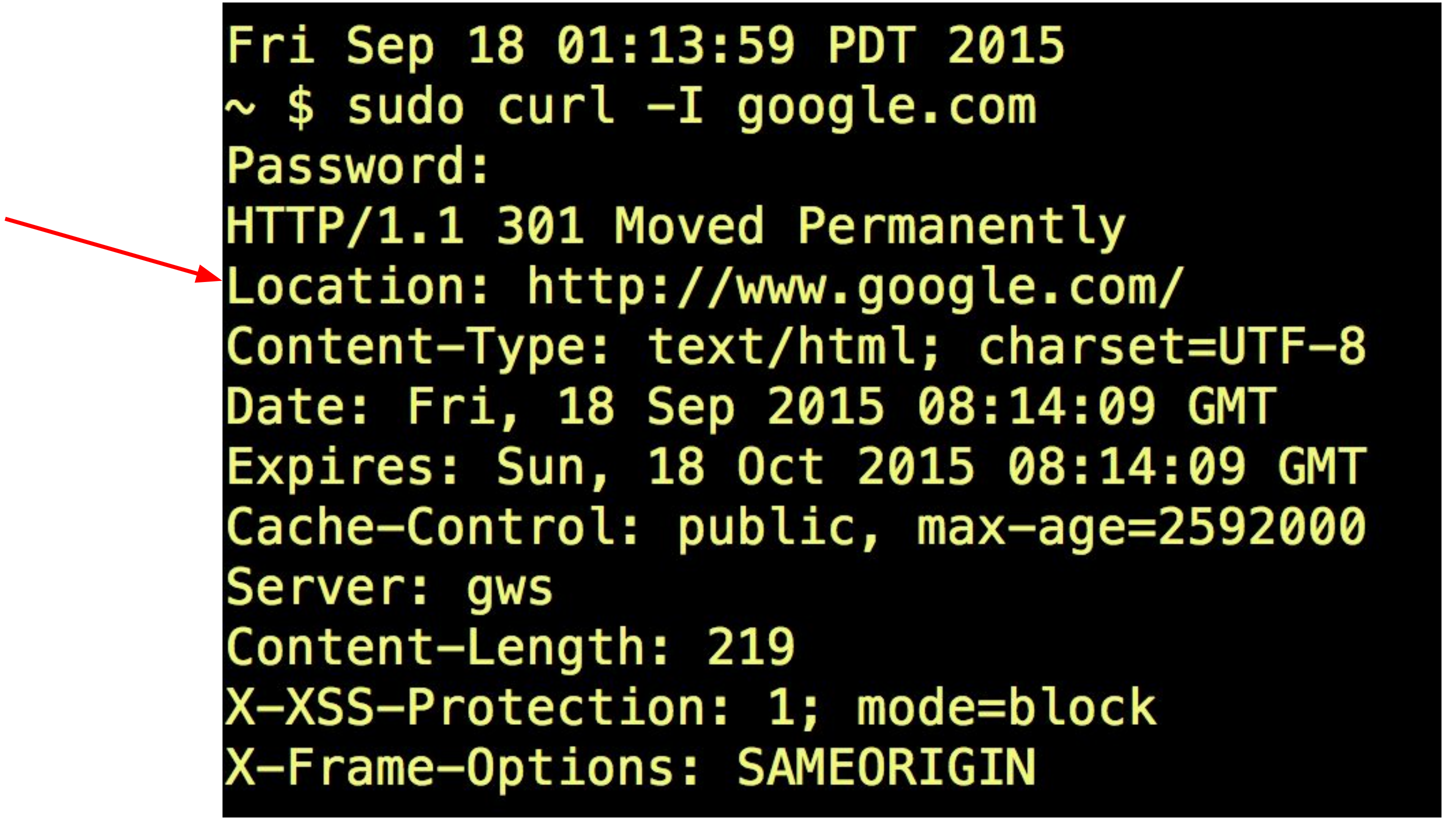
From Wikipedia, the free encyclopedia

**HTTP persistent connection**, also called **HTTP keep-alive**, or **HTTP connection reuse**, is the idea of using a single TCP connection to send and receive multiple HTTP requests/responses, as opposed to opening a new connection for every single request/response pair. The newer HTTP/2 protocol uses the same idea and takes it further to allow multiple concurrent requests/responses to be multiplexed over a single connection.

# HTTP Headers

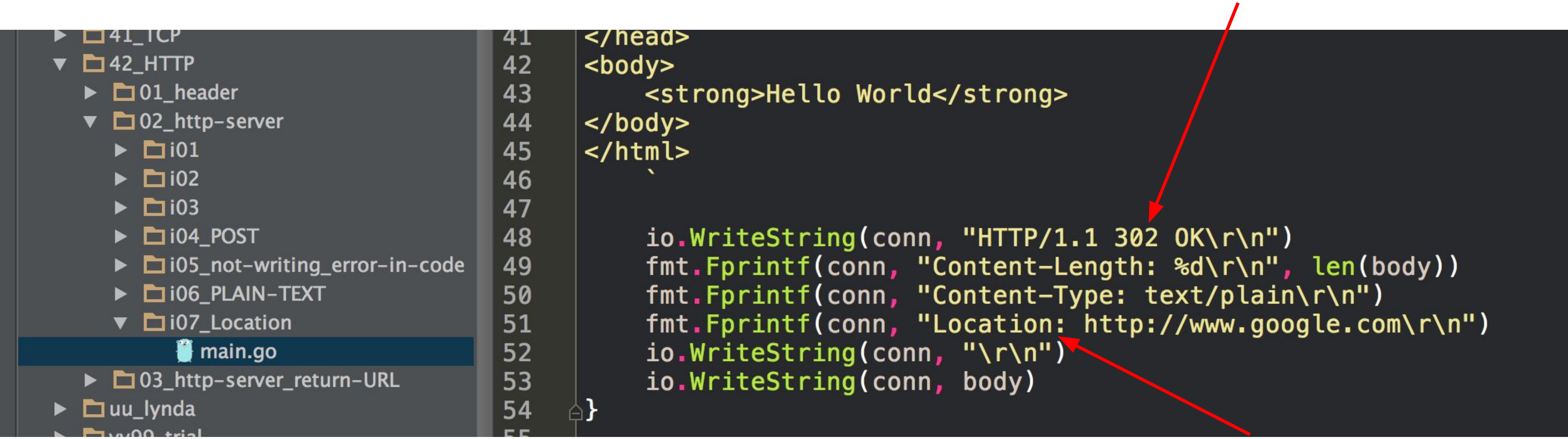
- Accept
- Connection
  - keep alive
    - needs content length
      - otherwise doesn’t know when request ends
  - closed
- Content-Type
  - application/x-www-form-urlencoded
  - multipart/form-data
  - text/plain
- Location
  - used for redirects

- Range
- Referer
- Transfer-Encoding
- WWW-Authenticate



```
Fri Sep 18 01:13:59 PDT 2015
~ $ sudo curl -I google.com
Password:
HTTP/1.1 301 Moved Permanently
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Date: Fri, 18 Sep 2015 08:14:09 GMT
Expires: Sun, 18 Oct 2015 08:14:09 GMT
Cache-Control: public, max-age=2592000
Server: gws
Content-Length: 219
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
```

HTTP status codes



```
</head>
<body>
    <strong>Hello World</strong>
</body>
</html>
    `


    io.WriteString(conn, "HTTP/1.1 302 OK\r\n")
    fmt.Fprintf(conn, "Content-Length: %d\r\n", len(body))
    fmt.Fprintf(conn, "Content-Type: text/plain\r\n")
    fmt.Fprintf(conn, "Location: http://www.google.com\r\n")
    io.WriteString(conn, "\r\n")
    io.WriteString(conn, body)
}
```

# HTTP Headers

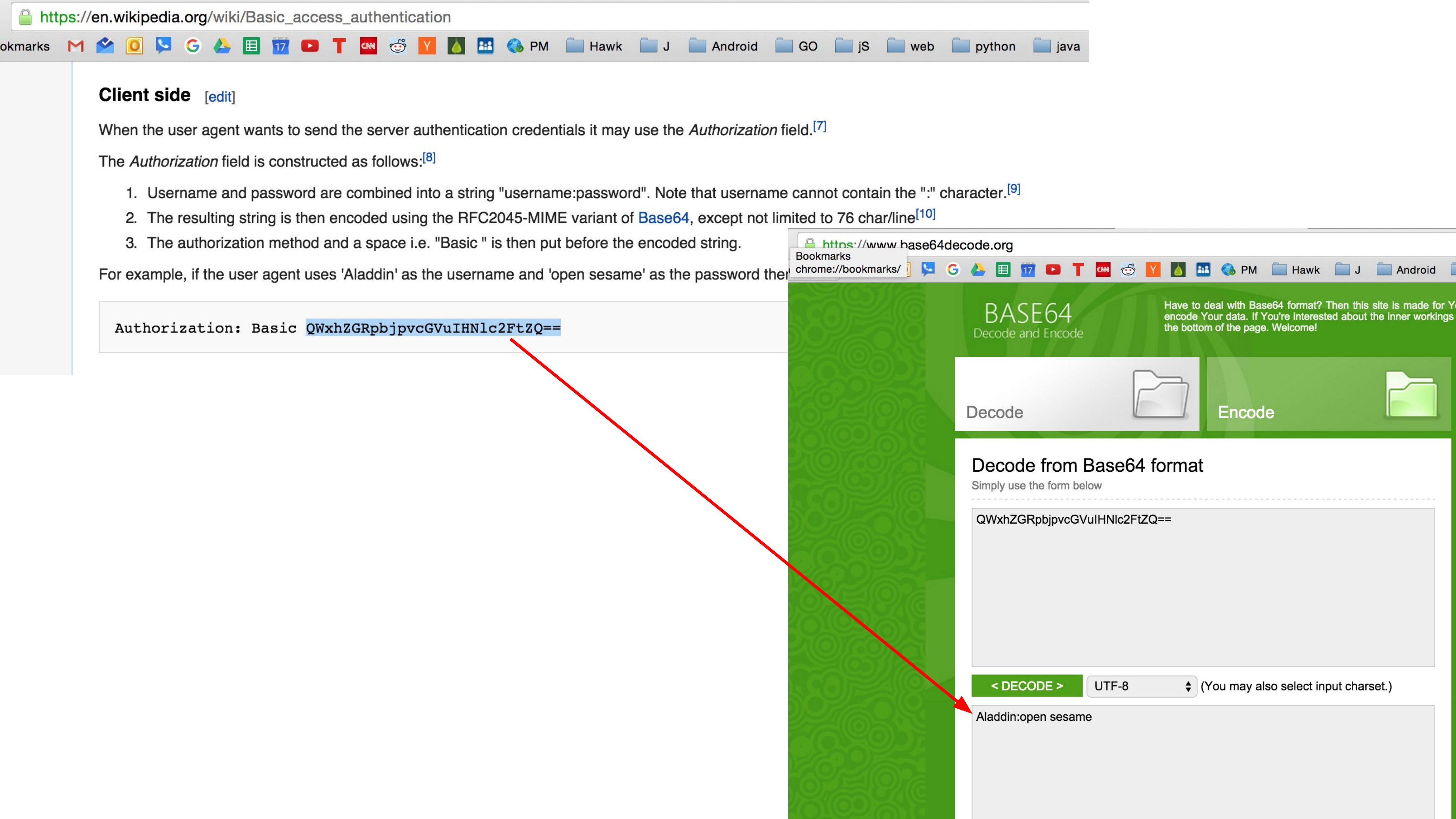
- Accept
- Connection
    - keep alive
        - needs content length
            - otherwise doesn’t know when request ends
    - closed
- Content-Type
    - application/x-www-form-urlencoded
    - multipart/form-data
    - text/plain
- Location
    - used for redirects
- Range
    - for resuming download of files
- Referer
- Transfer-Encoding
- WWW-Authenticate

# HTTP Headers

- Accept
- Connection
    - keep alive
        - needs content length
            - otherwise doesn't know when request ends
    - closed
- Content-Type
    - application/x-www-form-urlencoded
    - multipart/form-data
    - text/plain
- Location
    - used for redirects
- Range
    - for resuming download of files
- Referer
    - misspelled in the spec
        - so you have to use it that way
    - use it to see where a client came from
        - not trustworthy b/c anyone can change it
- Transfer-Encoding
- WWW-Authenticate

# HTTP Headers

- Accept
- Connection
  - keep alive
    - needs content length
      - otherwise doesn't know when request ends
  - closed
- Content-Type
  - application/x-www-form-urlencoded
  - multipart/form-data
  - text/plain
- Location
  - used for redirects
- Range
  - for resuming download of files
- Referer
  - misspelled in the spec
    - so you have to use it that way
  - use it to see where a client came from
    - not trustworthy b/c anyone can change it
- Transfer-Encoding
- WWW-Authenticate
  - basic access authentication
    - wikipedia
  - not secure; shouldn't use



https://en.wikipedia.org/wiki/Basic_access_authentication

okmarks PM Hawk J Android GO jS web python java

## Client side [edit]

When the user agent wants to send the server authentication credentials it may use the *Authorization* field.[7]

The *Authorization* field is constructed as follows:[8]

1. Username and password are combined into a string "username:password". Note that username cannot contain the ":" character.[9]
2. The resulting string is then encoded using the RFC2045-MIME variant of Base64, except not limited to 76 char/line[10]
3. The authorization method and a space i.e. "Basic " is then put before the encoded string.

For example, if the user agent uses 'Aladdin' as the username and 'open sesame' as the password the

```
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

https://www.base64decode.org

Bookmarks
chrome://bookmarks/

PM Hawk J Android

BASE64
Decode and Encode

Have to deal with Base64 format? Then this site is made for Y
encode Your data. If You're interested about the inner workings
the bottom of the page. Welcome!

Decode Encode

## Decode from Base64 format

Simply use the form below

QWxhZGRpbjpvcGVuIHNlc2FtZQ==

< DECODE > UTF-8 (You may also select input charset.)

Aladdin:open sesame

caching