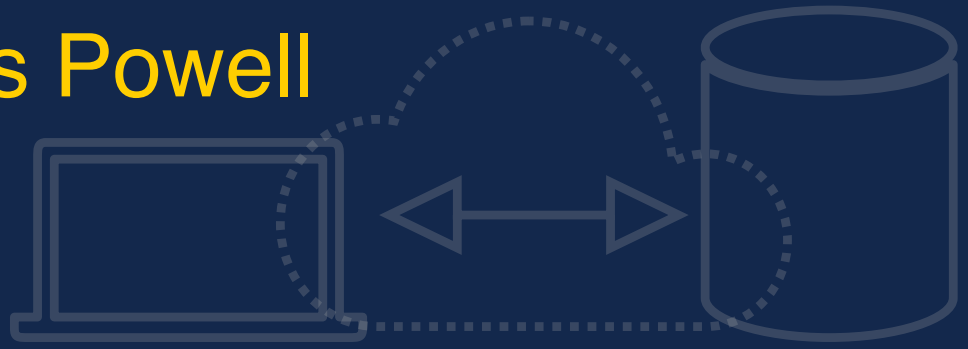
HTTP Overview

with Thomas Powell



Pro Tip: Web Dev - it's all about the network

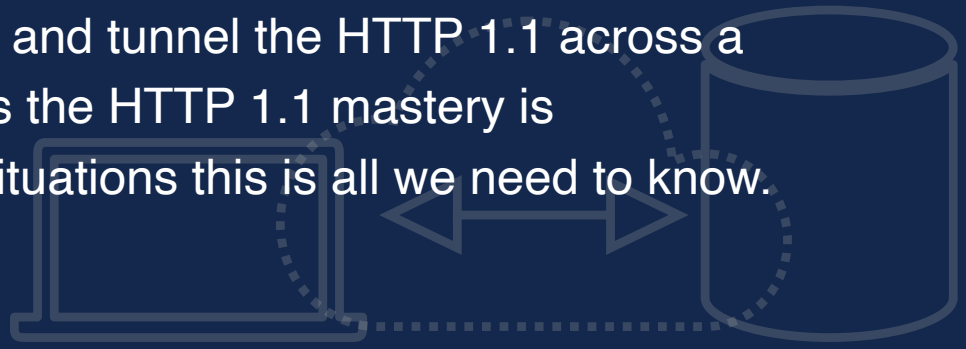
- If you want to really do webdev right you will need to know the ins and outs of HTTP
  - If the network has problems you/users have problems much more than you are probably aware
- Sadly most don't know as much as they think they do
  - This is easily shown by massive perf and security problems
  - A few tests
    - URLs – case sensitive? Length?
    - GET vs POST?
    - Cookies
  - Our massive reliance on "Layer 8" Error Correction aka the "meat layer"

HTTP Intro

- HTTP (Hyper Text Transfer Protocol)
  - It's an *application layer protocols* similar to SMTP, IMAP, NNTP, FTP, etc.
  - Simple protocol that defines the standard way that clients request data from Web servers and how these server respond
  - Typically it is running on top of TCP/IP

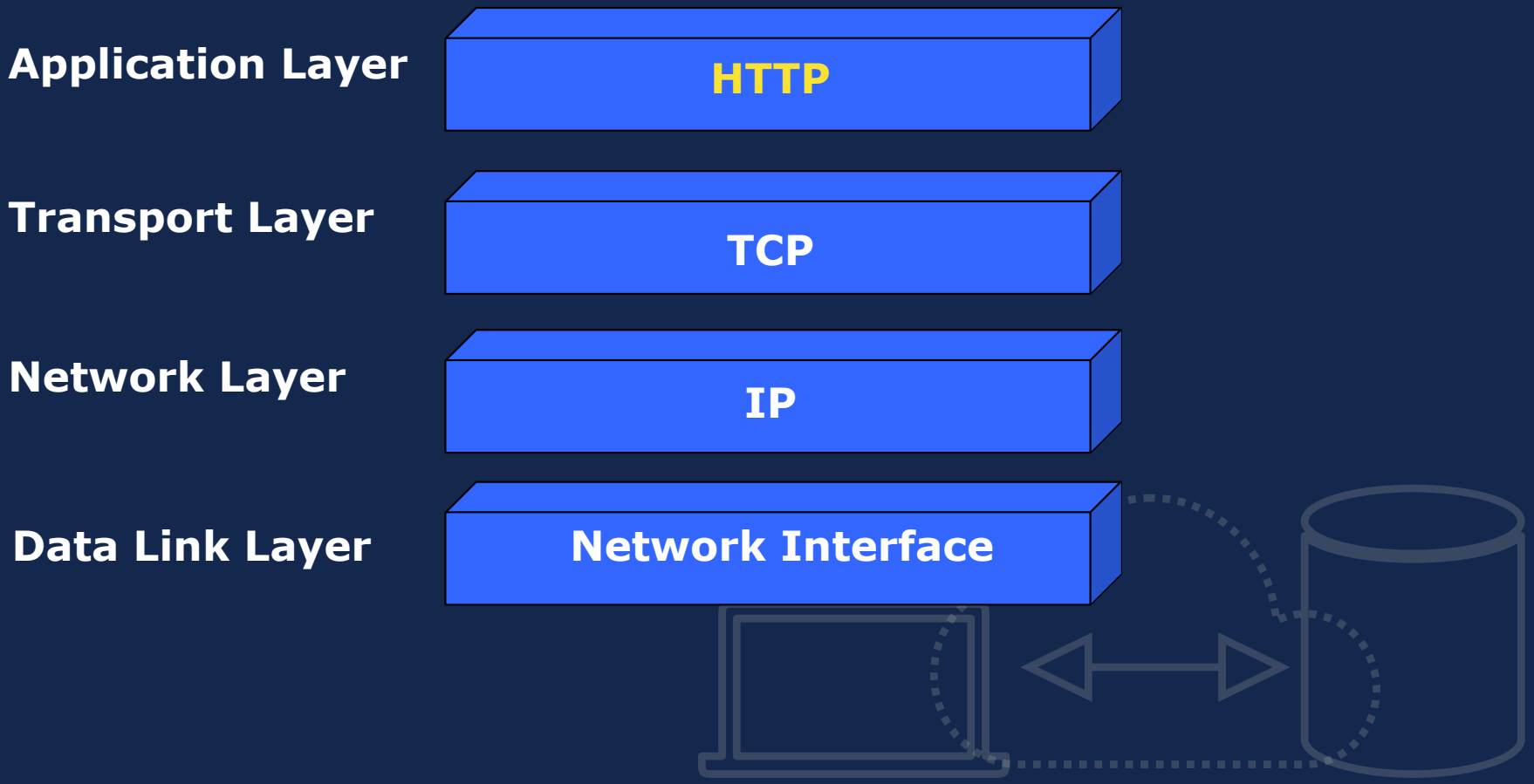
- Three versions initially were used (0.9,1.0,1.1) with 1.1 commonly used today
  - RFC 1945 HTTP 1.0 (1996)
  - RFC 2616 HTTP 1.1 (1999)

- Modern HTTP/2, HTTP/3, and variants exist and tunnel the HTTP 1.1 across a form within and SSL stream. For developers the HTTP 1.1 mastery is paramount before moving on and for most situations this is all we need to know.



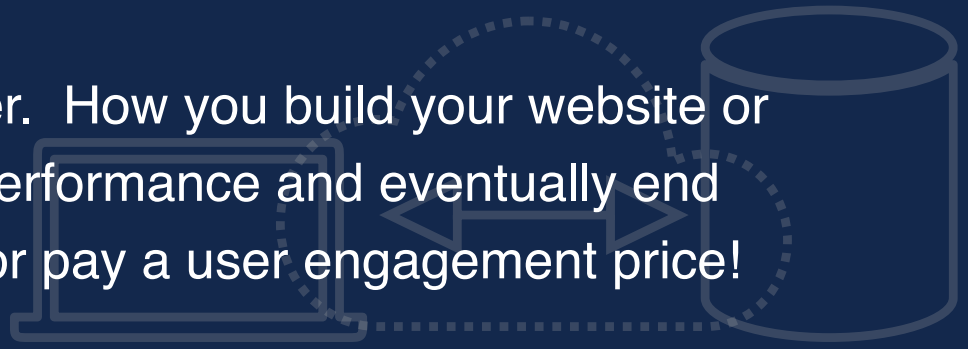
HTTP and TCP/IP

HTTP sits atop the TCP/IP Protocol Stack



HTTP: Lower Layers Effect Upper Layers

- Example: web pages often are composed of many small individual objects. Given HTTPs individual request combined with TCP effects performance can suffer.
- HTTP 1.1 and various dev best practices like bundling, domain sharding, etc. employed to address poor performance.
- Yet today with HTTP/2 & 3 we may find these best practices may actually become anti-patterns!
- Takeaway: The network and HTTP matter. How you build your website or app has a significant material effect on performance and eventually end user happiness. Get performance right or pay a user engagement price!



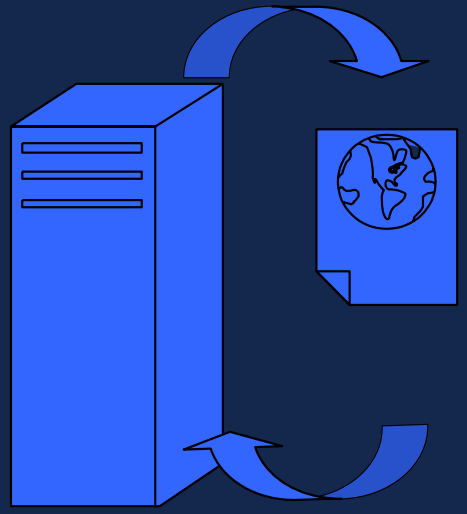
Basic HTTP Request/Response Cycle

Asks for resource by URL:

http://www.foo.com/page.html

HTTP Request

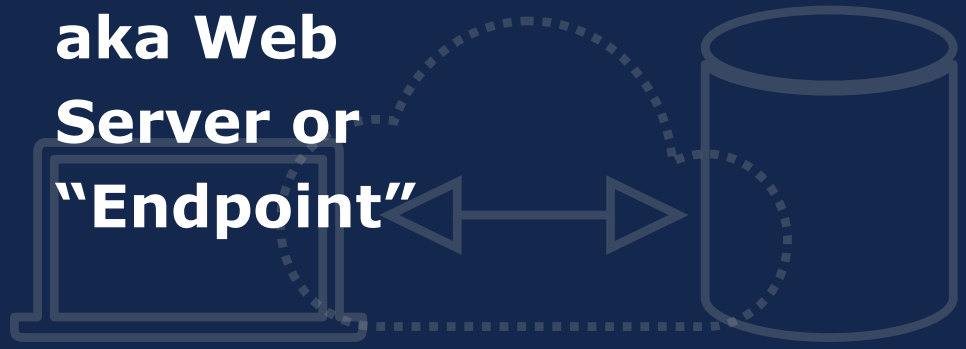
HTTP Response



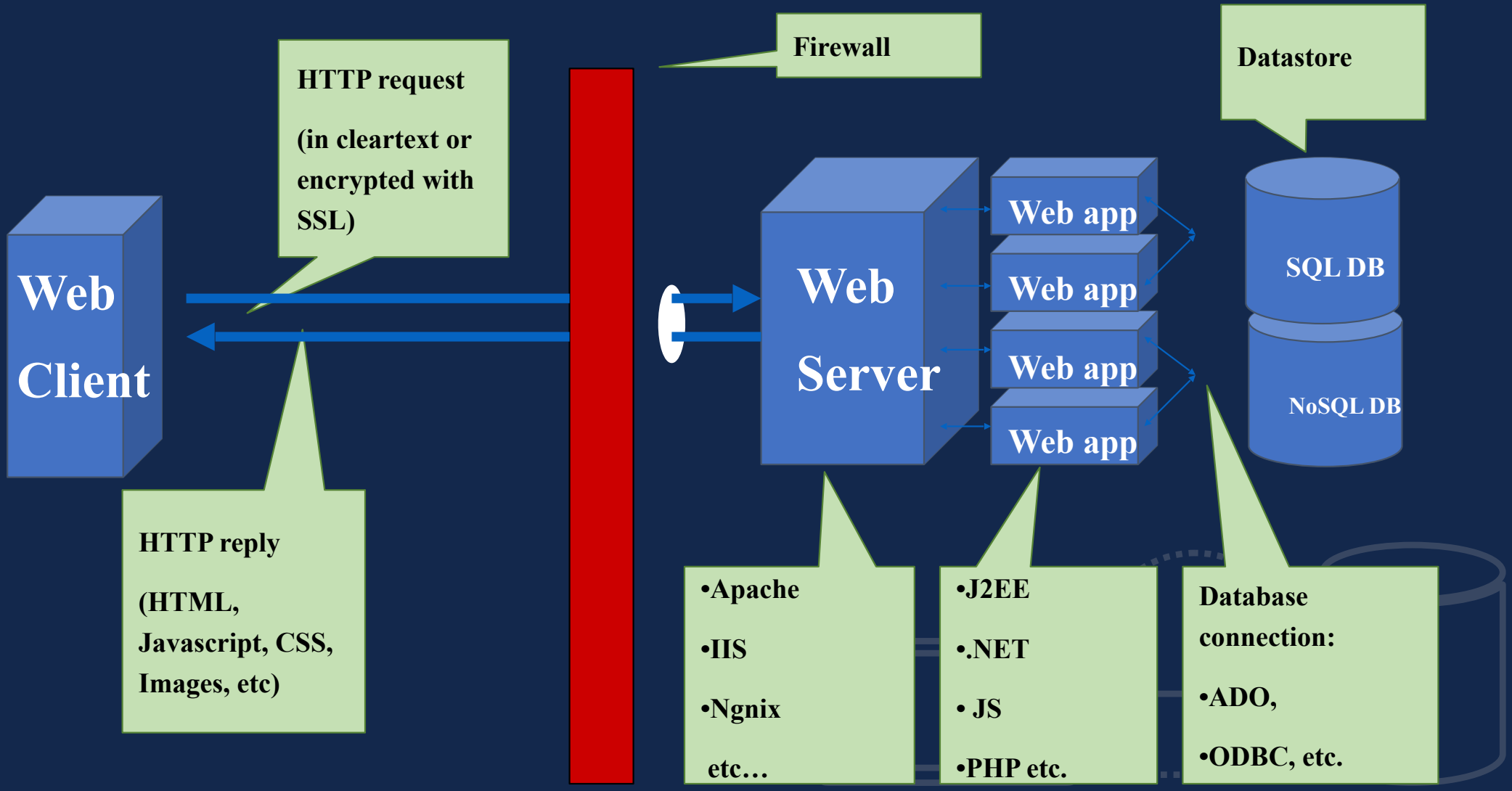
Find and return or run requested resource

HTTP Client aka "User Agent" typically a browser

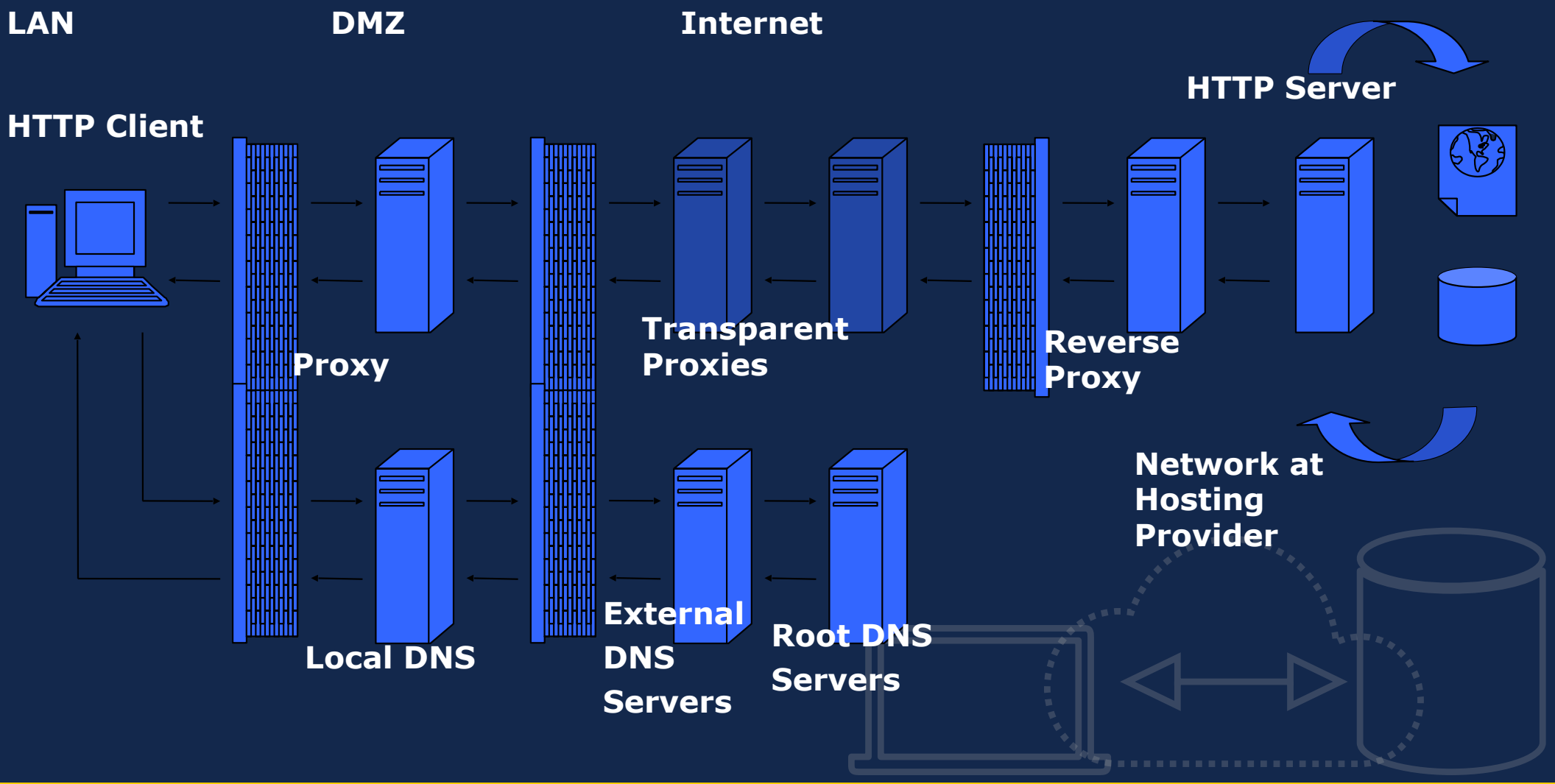
HTTP Server aka Web Server or "Endpoint"



Take 2 - Nothing going on here … it's simple

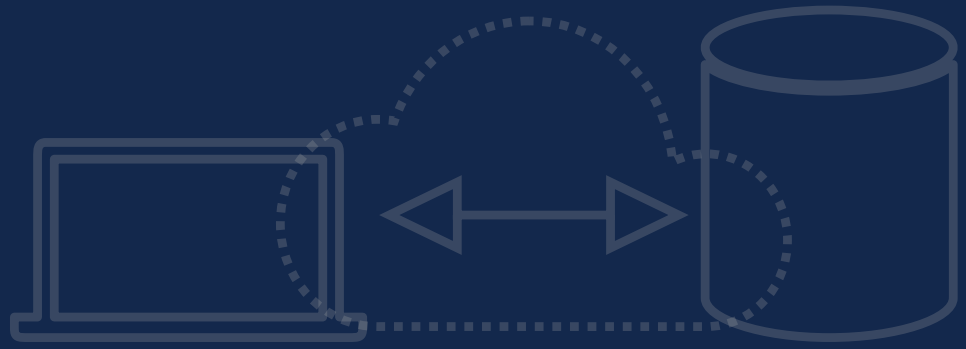


Take 3 – Add to that transit issues



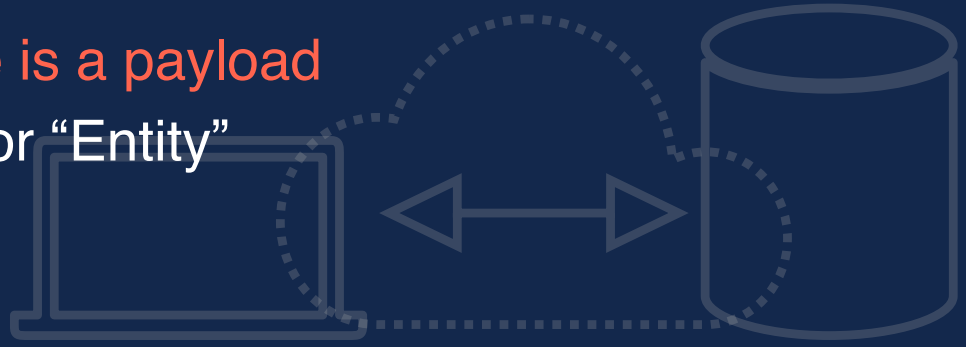
Mind your Proxies

- Proxies are HTTP Intermediaries
  - All act as both clients and servers
  - Friend and foe to Web devs and you should acknowledge them
- Major types of proxies can be distinguished by where they live and how they get traffic
  - Explicit
  - Transparent/Intercepting
  - Reverse/Surrogate
- Three primary uses for proxies
  1. Security
  2. Performance
  3. Content Filtering



HTTP Requests

- HTTP requests and responses are both types of Internet Messages (RFC 822) , and share a general format:
  - A Start Line, followed by a CRLF
    - Request Line for requests
    - Status Line for responses
  - Zero or more Headers
    - field-name ":" [field-value] CRLF
  - An empty line
    - Two CRLFs mark the end of the Headers
  - An optional Message Body if there is a payload
    - All or part of the "Entity Body" or "Entity"



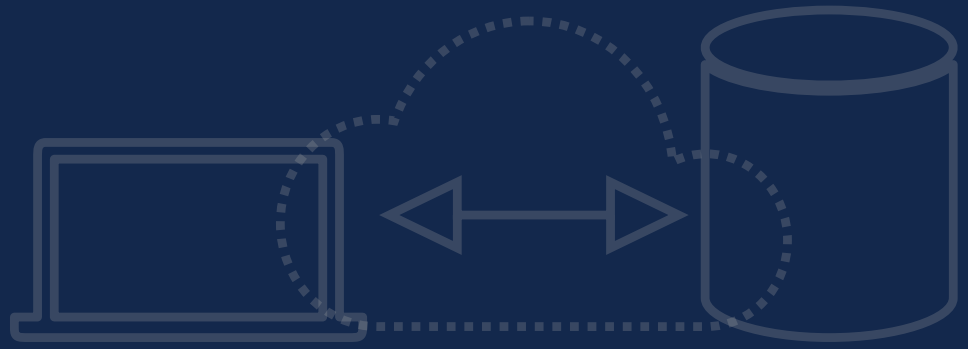
The Law of Three Redux

- Consider the form of HTTP requests and responses there are three pieces to each
- As a server-side program you understand you can only get data from the three pieces of the request (so consider those your input and watch them carefully!)
- Furthermore you can only output via the three pieces of the response (mostly the message body)
- **This is it**, there is **nothing** more to the bedrock of the web transmit wise (save addressing statefulness)

1. Request / Response Line
2. Headers

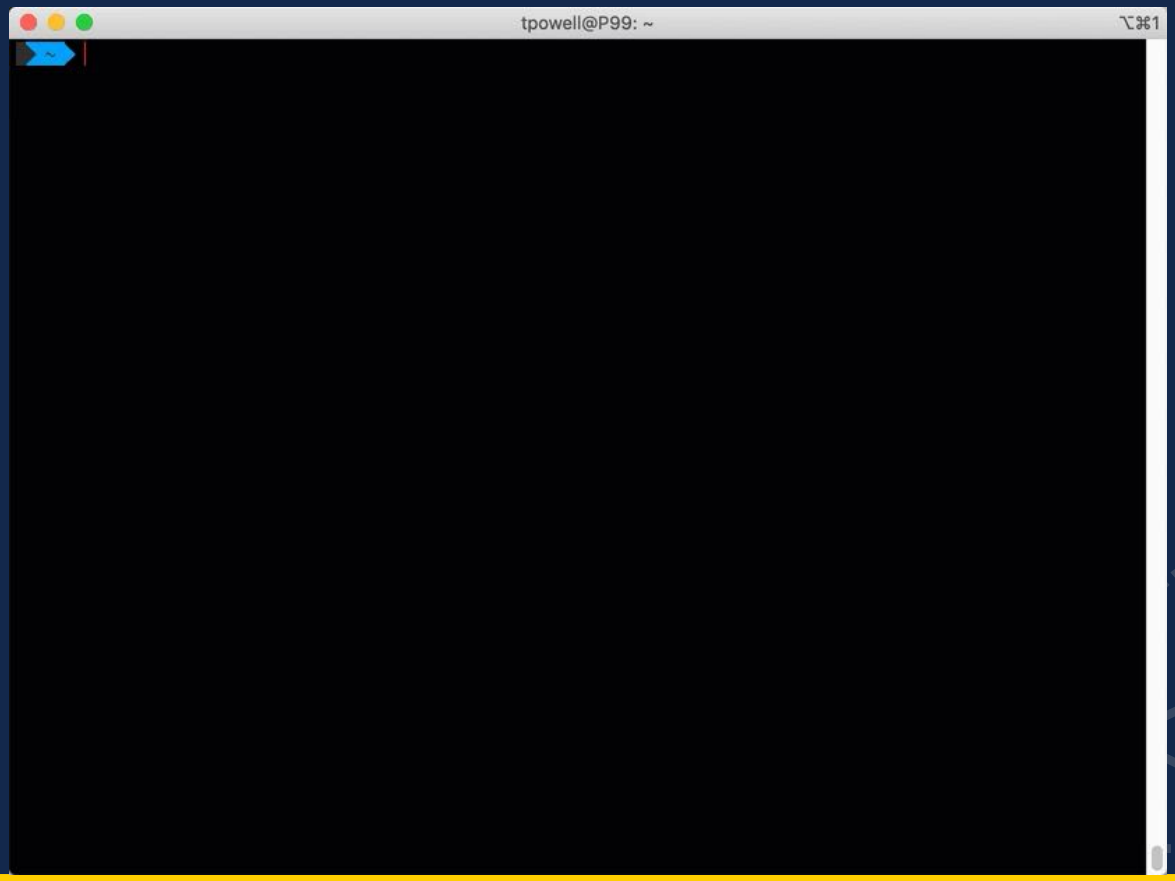
/n/n

3. Message Body (Payload)



Making a simple HTTP request

- Use a command line tool like “curl” to make a request
  - $ curl https://ucsd.edu | more



HTTP Request Example #1

```
07/01/04 09:07:02 Browsing http://www.ucsd.edu
Fetching http://www.ucsd.edu/ ...
GET / HTTP/1.1
Host: www.ucsd.edu
Connection: close
User-Agent: Sam Spade 1.14

HTTP/1.1 200 OK
Date: Thu, 01 Jul 2004 16:07:00 GMT
Server: Apache/1.3.27 (Unix)
Last-Modified: Thu, 01 Jul 2004 16:01:00 GMT
ETag: "c992b-77df-40e4353c"
Accept-Ranges: bytes
Content-Length: 30687
Connection: close
Content-Type: text/html


<!doctype html public "-//W3C//DTD HTML 4.0 Transitional//EN">
<html lang="en">
<head>
<base href="http://www.ucsd.edu/">
<title>University of California, San Diego</title>
<meta name="generator" content="">
<meta name="author" content="UCSD Libraries, Information Technology Depar
<meta name="keywords" content="">
```

Request Headers

Response Headers

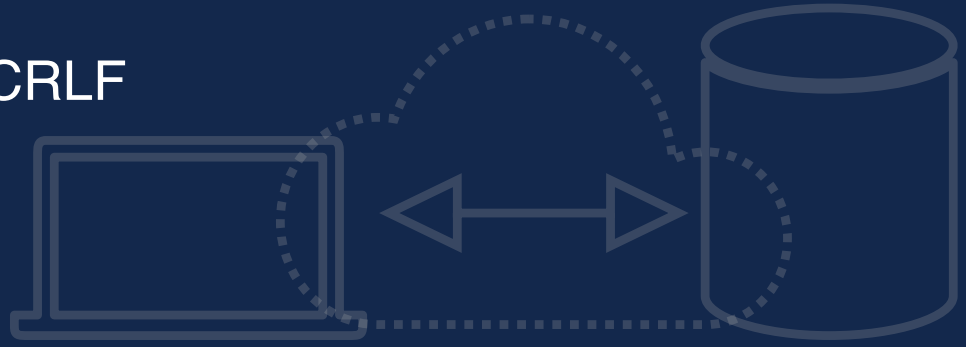
Response data

HTTP Request Example #2

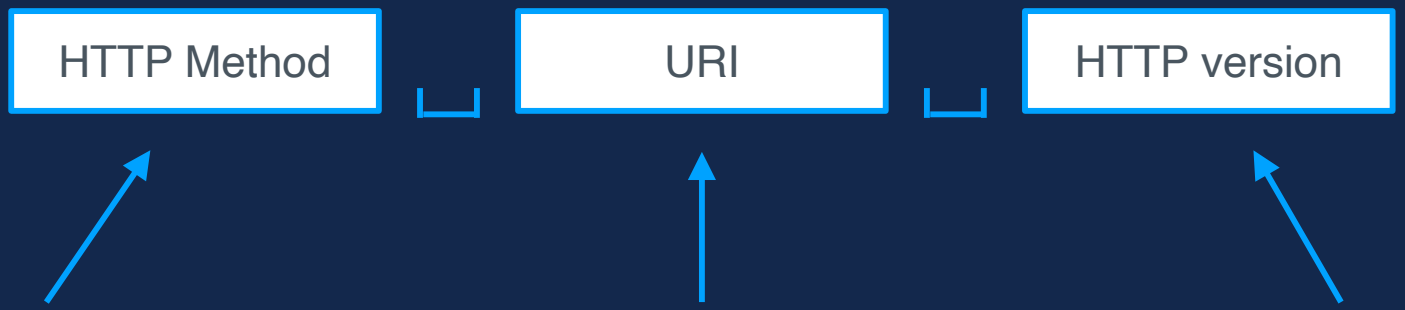
```
Fetching http://www.pint.com/badurl ...
GET /badurl HTTP/1.1
Host: www.pint.com
Connection: close
User-Agent: Sam Spade 1.14

HTTP/1.1 404 Not Found
Content-Length: 16592
Content-Type: text/html
Server: Microsoft-IIS/6.0
Date: Thu, 01 Jul 2004 16:57:38 GMT
Connection: close

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

A Closer Look at the Request Line

- Consists of three major parts
  - The Request Method followed by a space
    - Methods in HTTP 1.1 include: **GET**, **POST**, **HEAD**, **TRACE**, **OPTIONS**, **PUT**, **DELETE** and **CONNECT**
    - GET, POST, and HEAD are the most common
    - Extension methods such as those specified by WebDav (RFC 2518)
  - The Request URI followed by a space
    - The URL associated with the resource to be fetched or acted upon
  - The HTTP Version followed by the CRLF
    - 0.9, 1.0, 1.1



A Closer Look at the Request Line



HTTP method (aka verb) to invoke

URI (aka resource) to act upon

HTTP version used

Examples

```
GET / HTTP/1.1
```

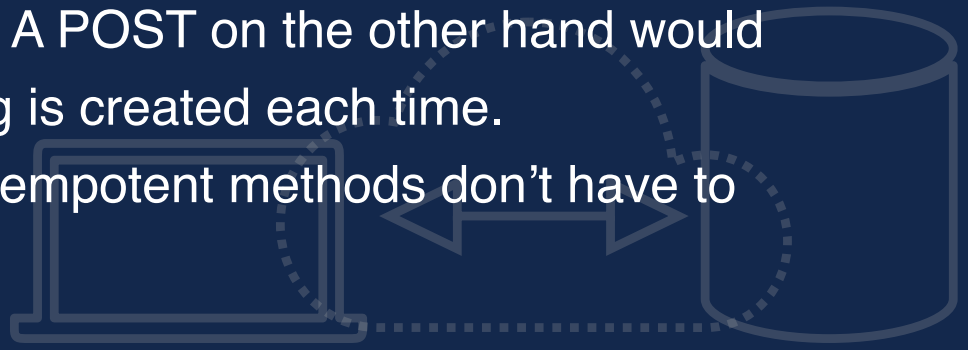
```
POST /collection/add-member HTTP/1.1
```

```
PUT /new-resource HTTP/1.1
```

Jargon Time!

- An HTTP method is considered **SAFE** it doesn't change the state of the endpoint (server, app, etc.)
  - A read style method (GET, HEAD, OPTIONS, etc.) would be considered SAFE.

- An HTTP method is considered **idempotent** if it can be repeated with the same effect and the server stays in the same state.
  - GET would be idempotent but other less obvious methods like DELETE are idempotent because once deleted something can be repeat DELETE and nothing really changes. A POST on the other hand would not be idempotent because something is created each time.
  - SAFE methods are idempotent, but idempotent methods don't have to be SAFE.



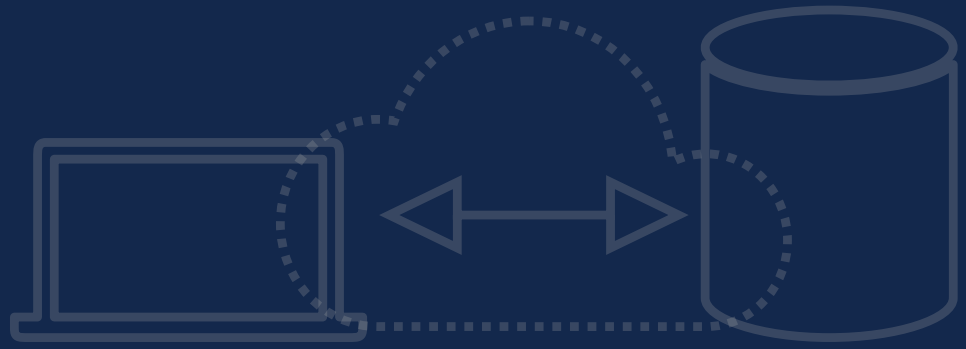
HTTP Methods

Method	Safe	Idempotent	Description
GET	Y*	Y	Fetch data from the specified URI. Data may be passed via a query string. GET requests should only retrieve data, not change state. May be used in an HTML form.
HEAD	Y	Y	Fetch the headers of the content at the specified URI. Useful for diagnostic or monitoring purposes.
POST	N	N	Submits data in the message payload to the specified URI usually used to create a resource. Under REST is used to create a record. This method is not idempotent in that it can be repeated. May be used in an HTML form.
PUT	N	Y	Stores any submitted data at the URI indicated. Used in REST for updates
PATCH	N	Maybe?	Used to partially modify the resource at the specified URI. May be used in REST for a partial edit as opposed to a full edit.
DELETE	N	Y	Delete the resource at the specified URI. Used in REST for deleting an object
OPTIONS	Y	Y	Returns the HTTP methods supported by the server for the specified URI. Often used to negotiate methods before requests are made.
TRACE	Y	Y	Used for debug tracing to echo
CONNECT	Y	Y	Used when attempting to switch protocols such as to SSL.

Why do I care

- HTML forms define the HTTP method directly in an attribute either GET (default) or POST for how the data of the form will be sent to the endpoint defined by the action attribute
- The enctype even can be used to indicate the content encoding!

```
<form action="/createaccount" method="POST">
```

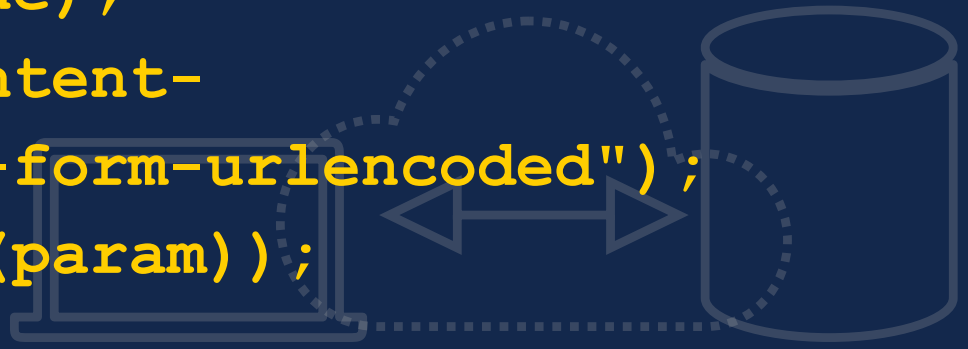


Why do I care?

- In web dev you may have to form raw requests ourselves.

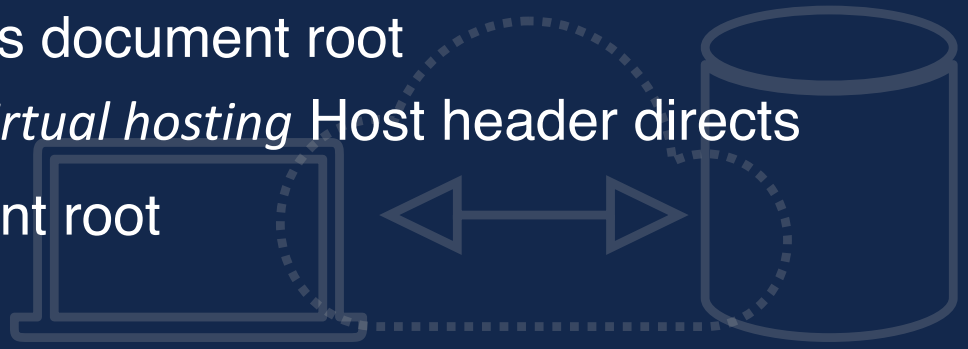
- Example in JavaScript using Ajax you form raw HTTP requests using GET and POST (or even HEAD if you like) to transmit your data

```
let xhr = new XMLHTTPRequest();
xhr.open("POST", url, true);
xhr.setRequestHeader("Content-
Type", "application/x-www-form-urlencoded");
xhr.send("ret=" + escape(param));
```



A Closer Look at the Request URI

- Absolute URI vs. Absolute Path
  - Explicit Proxies Require Absolute URIs
    - Client is connected directly to the proxy
    - Protocol and host name needed to resolve request
    - You might need full URLs too esp. for Web services
      - Watch out for www vs. no www issues
  - Grammar of the Absolute Path
    - Like Absolute URI minus the “http://hostname”
    - Initial “/” equivalent of the host’s document root
    - In HTTP 1.1 with *name-based virtual hosting* Host header directs request to appropriate document root



The HTTP Response

- Response status line consists of three parts:
  1. The HTTP Version followed by a space
  2. Status Code followed by a SP
  3. The “Reason Phrase” followed by the CRLF

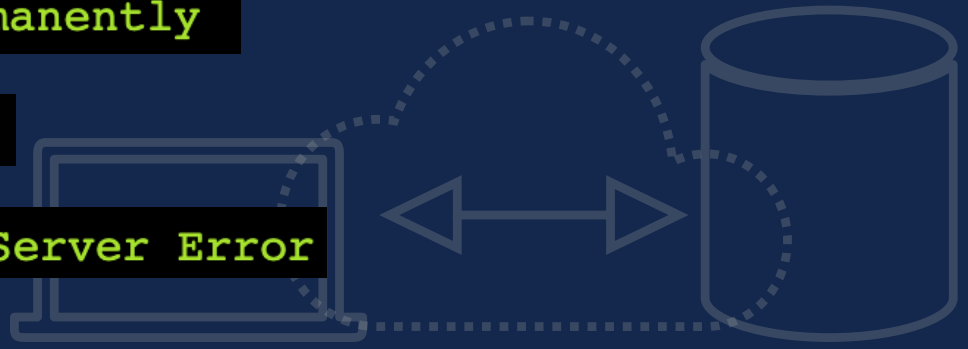
```
HTTP/1.1 100 Continue
```

```
HTTP/1.1 200 OK
```

```
HTTP/1.1 301 Moved Permanently
```

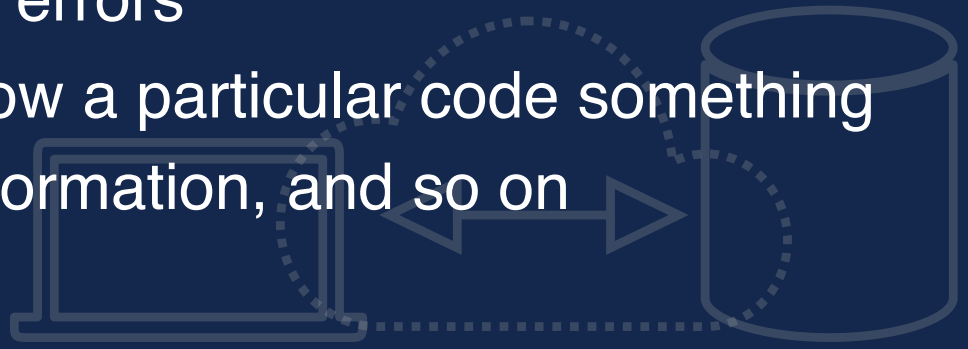
```
HTTP/1.1 404 Not Found
```

```
HTTP/1.1 500 Internal Server Error
```

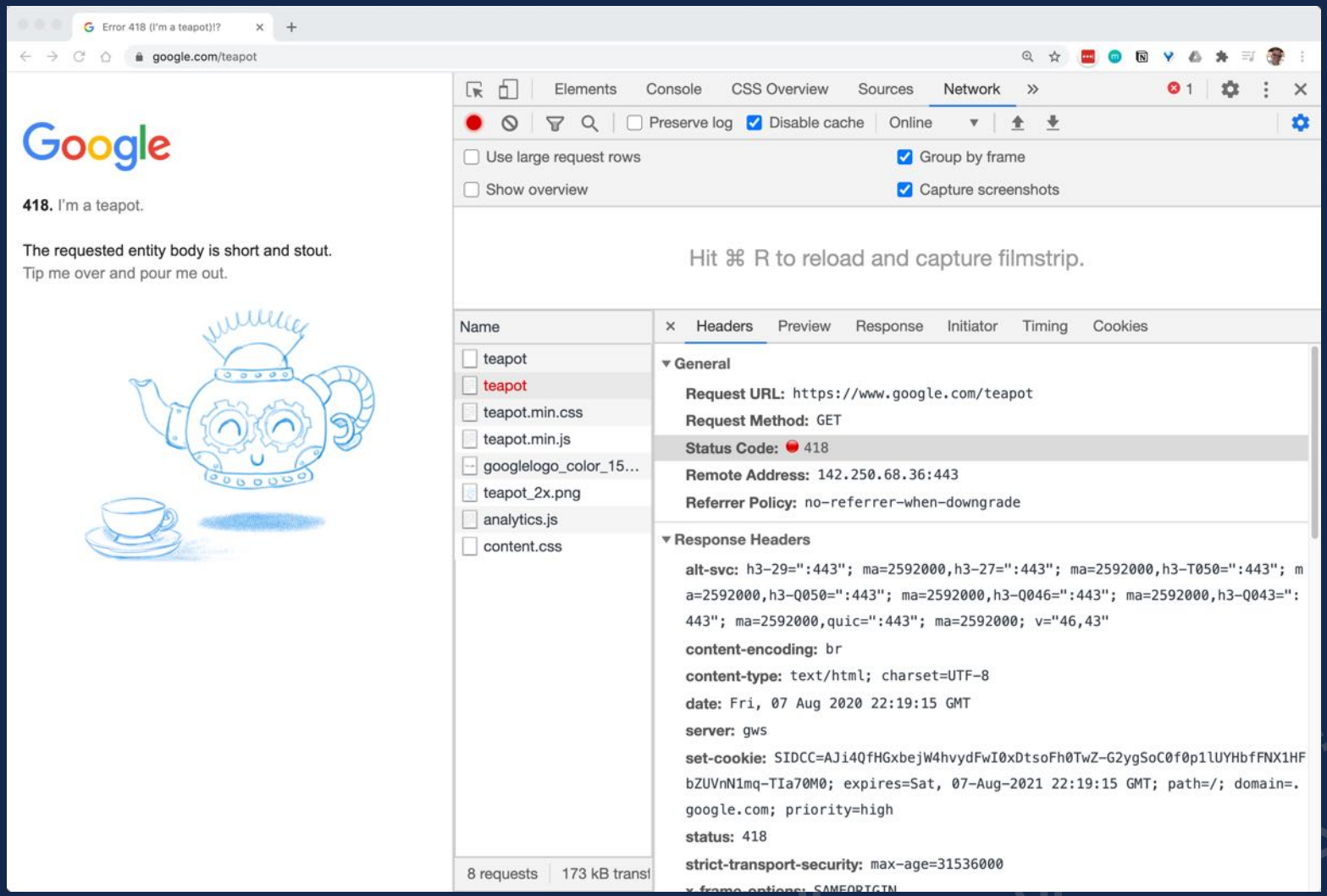


Status Codes

- Status codes come in categories.
- There are 5 groups of 3 digit integers indicating the result of the attempt to satisfy the request
    1. `1xx` are informational
    2. `2xx` are success codes
    3. `3xx` are for alternate resource locations (redirects)
    4. `4xx` indicate client side errors
    5. `5xx` indicate server side errors
- Even if a user agent does not know a particular code something in a 5xx or 4xx is an error, 1xx information, and so on



HTTP Response Fun

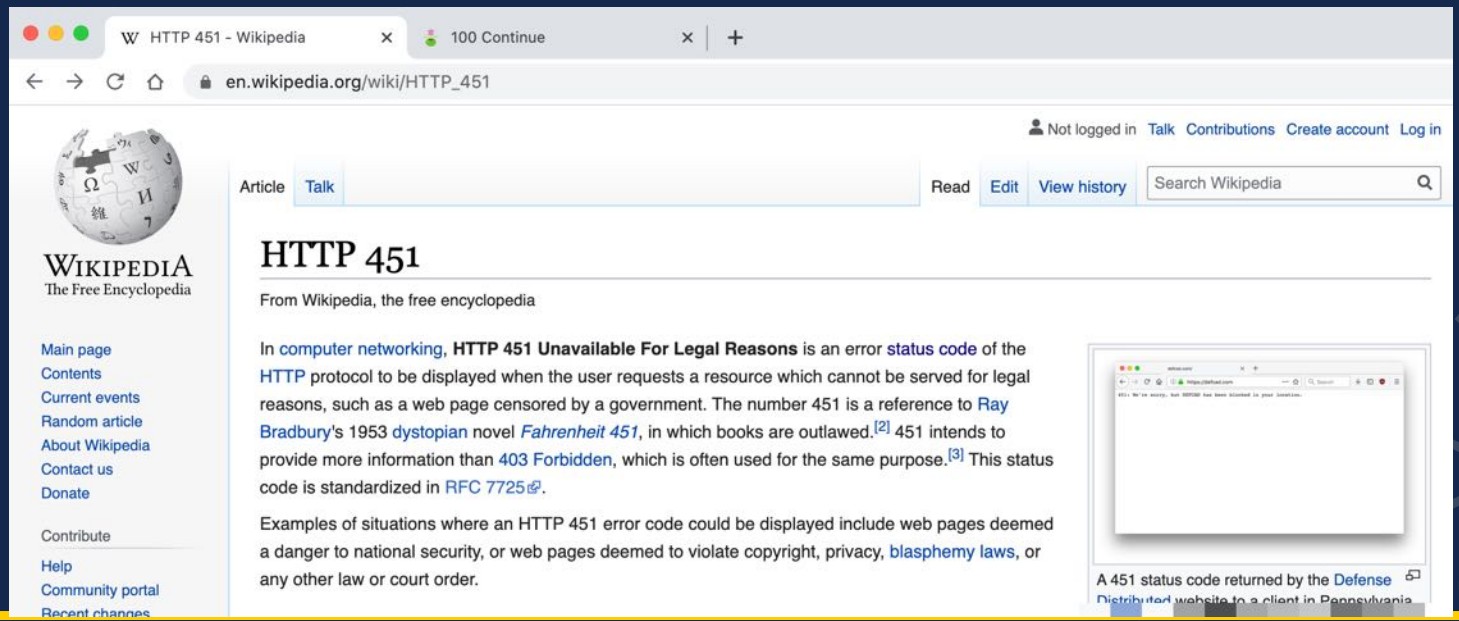


https://www.google.com/teapot

HTTP Response Not Fun

```
HTTP/1.1 451 Unavailable For Legal Reasons
Link: <https://proxy.example.org/legal>; rel="blocked-by"
Content-Type text/html

<h1>Government policy prohibits you from reading this information.</h1>
```

https://evertpot.com/http/451-unavailable-for-legal-reasons



Application – One Way Requests and 204s

- There are many details to HTTP that people don't consider, but are highly useful one example is 204 responses which send back no data.

- You can observe Google using this in its search results page to send a one way request to see what is being clicked on. Such a request is often used in apps to see what is being clicked on or what even happens just before page unload. The `sendBeacon()` API in JS even codifies it!

HTTP Headers

- Headers come in four major types, some for requests, some for responses, some for both and a special extension category.

1. General Headers
   - Provide info about messages of both kinds
2. Request Headers
   - Provide request-specific info
3. Response Headers
   - Provide response-specific info
4. Entity Headers
   - Provide info about request and response entities
5. Extension headers
   - X- or user defined headers are also possible on either request or response

A Closer Look at General Headers

- `Connection` – lets clients and servers manage connection state

```
Connection: close
```

- `Date` – when the message was created

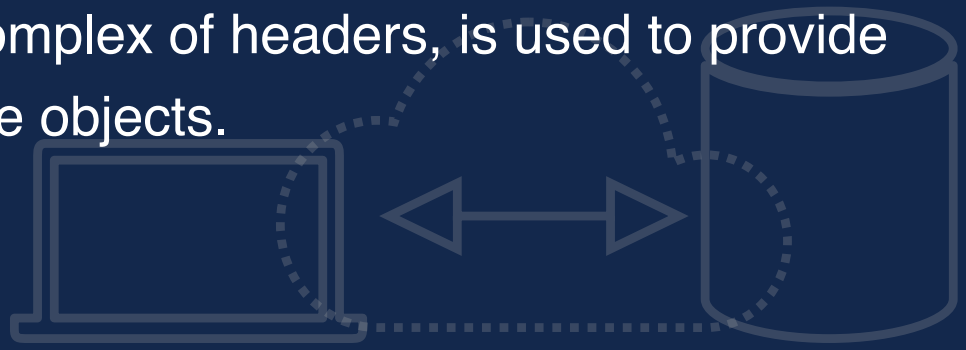
```
Date: Sat, 31-May-03 15:00:00 GMT
```

- `Via` – shows proxies that handled message

```
Via: 1.1 www.myproxy.com (Squid/1.4)
```

- `Cache-Control` – Among the most complex of headers, is used to provide caching directives to cache or not cache objects.

```
Cache-Control: no-cache
```



Why do I care? – Unfriendly Caches

- If you are issuing a GET request and you do it again the browser will not bother to talk to the server (depending on browser settings including defaults), but instead will pull the data from cache. This can cause lots of screw-ups
  - Example – someone looking at stale content
  - Example – Problems with Ajax style apps never waking up because browser using cached data
- Obvious solution to "stale caches" is to add cache control headers (or to change resource names) but then again that does defeat the value
  - Better to know about caching and do it properly
  - Consider typical Web pages what would you want to cache?

A Closer Look at Request Headers

- **`Host`** – The hostname (and optionally port) of server to which request is being sent. Required for name-based virtual hosting

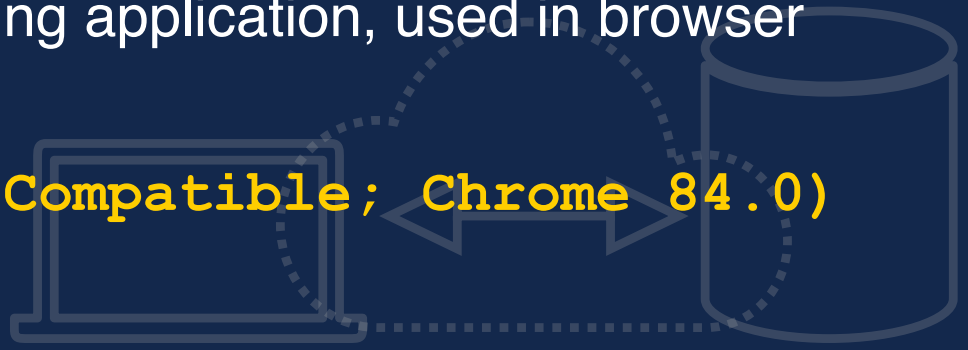
```
Host: stage.example.com
```

- **`Referer`** – The URL of the resource from which the current request URI came. Misspelled in the specification.

```
Referer: http://www.example.com/login.php
```

- **`User-Agent`** – Name of the requesting application, used in browser sensing

```
User-Agent: Mozilla/5.0 (Compatible; Chrome 84.0)
```



Some More Request Headers

- `Accept` and its variants – Inform servers of client's capabilities and preferences and enables *content negotiation* generally or compression, language detection, etc. specifically.

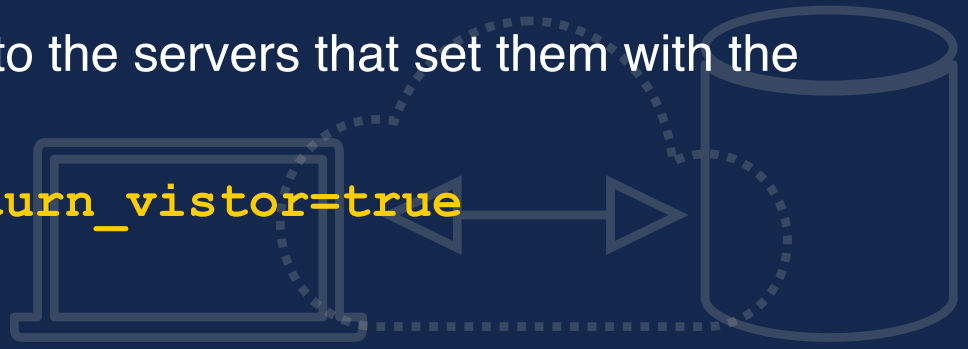
```
Accept: image/gif, image/jpeg;q=0.5
Accept-Encoding: gzip
```

- `If-Modified-Since` and other *conditionals*. Such headers are frequently used by browsers to manage caches or monitor assets.

```
If-Modified-Since: Sat, 31-May-03 15:00:00 GMT
```

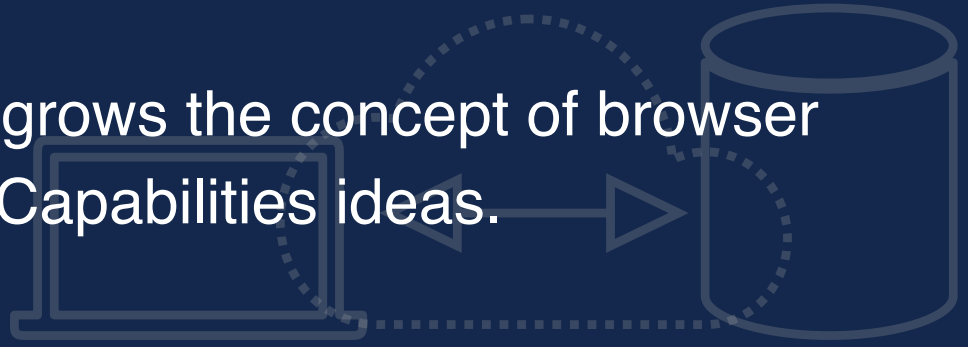
- `Cookie` – How clients pass cookies back to the servers that set them with the related response header Set-Cookie

```
Cookie: id=234;path=/shop;return_vistor=true
```



Using Request Headers: Browser Sniffing

- The `User-agent` is often used in browser detection to serve different type of page to different type of accessing agent
  - Similarity problem
    - Everything looks like old “Mozilla”
  - Spoofing or removing problem
- Better approach is to take this and add in an injected script or program that profiles the device.
- In the long run as device diversity grows the concept of browser will evolve significantly see Client Capabilities ideas.



Using Request Headers: Anti-Leeching

- Often times people may leech your bandwidth with direct hotlinking to your object (GIF, Movie, etc.) without fetching the other related objects
  - This certainly would be bad if your biz model was about people seeing the related ads around the 'stolen' object!
- Since the `Referer` header is sent from the base page a simple form of anti-leeching is to check for it before sending a dependent object
  - Of course the bad guy now moves to forge the header
- Class Question: can you think of other countermeasures?

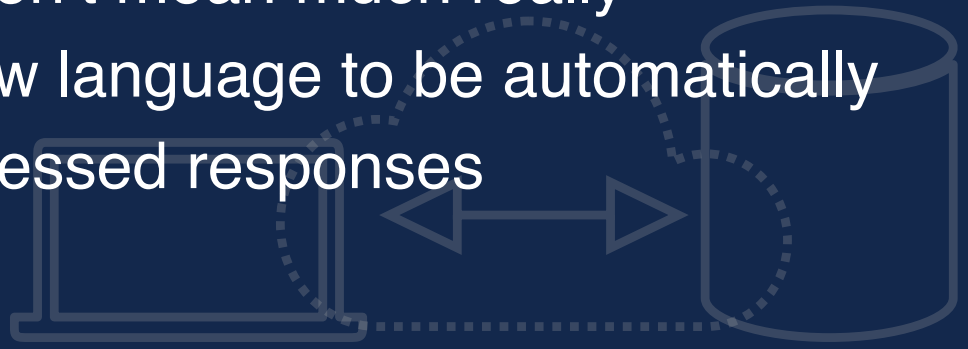
Using Request Headers: Content Negotiation

- User-agent generally sends a variety of “accept” headers indicating type of content it can handle

```
GET /images/HF_servermask.HTTP/1.1
Host: www.port80software.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.6) Gecko/20040206 Firefox/0.8
Accept: image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
```

Using Request Headers: Content Negotiation

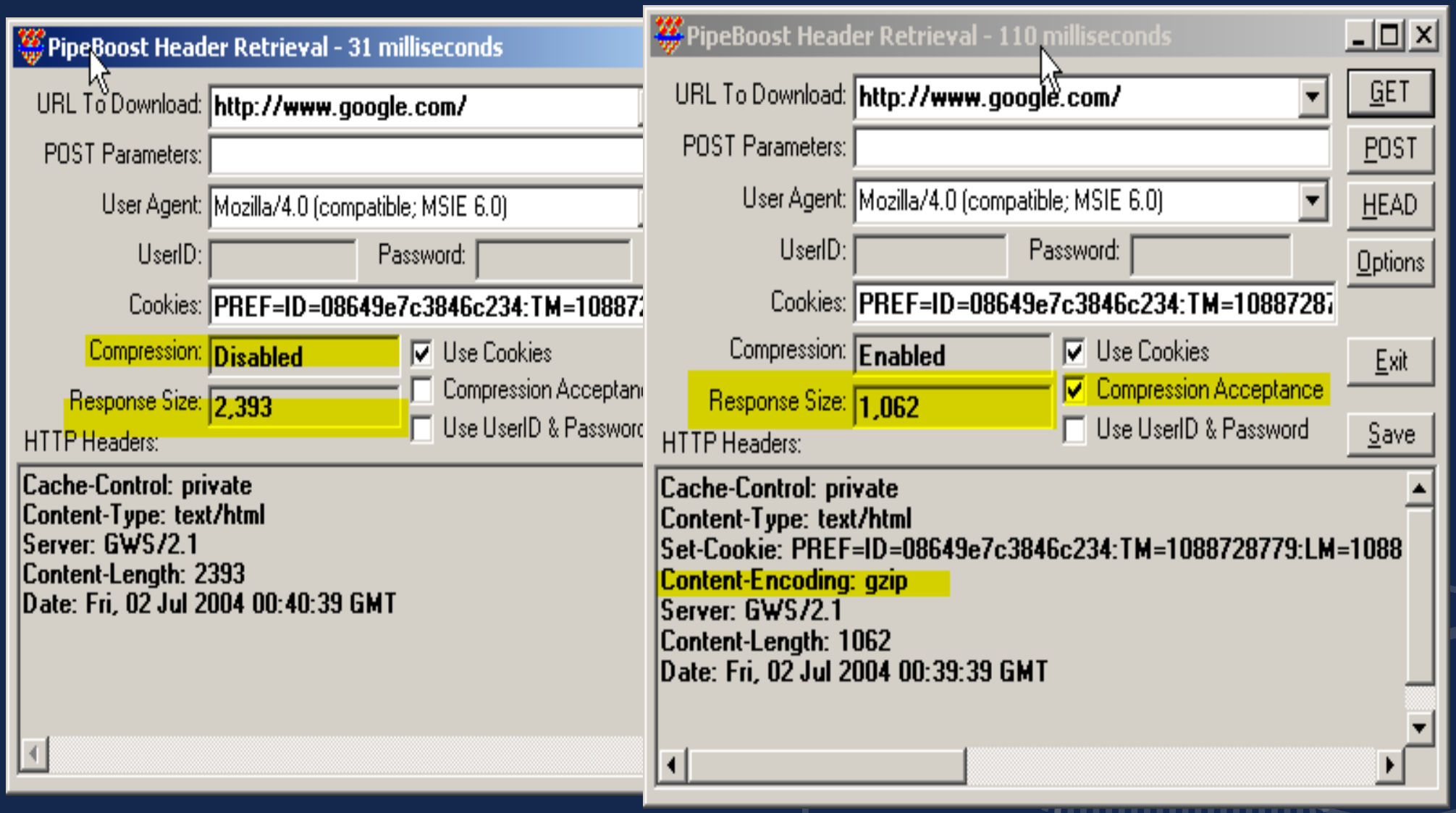
- A “q-rating” can indicate the preference the user agent has for the data requested
- Content negotiation allows us to ask for something like “logo” and then get the appropriate image (PNG, JPG, etc.) based upon what the device can handle.
  - This leads to extensionless URLs which aids in long term maintainability
  - We’ll see the file extensions don’t mean much really
- Content negotiation can also allow language to be automatically negotiated and empowers compressed responses



Using Request Headers: HTTP Compression

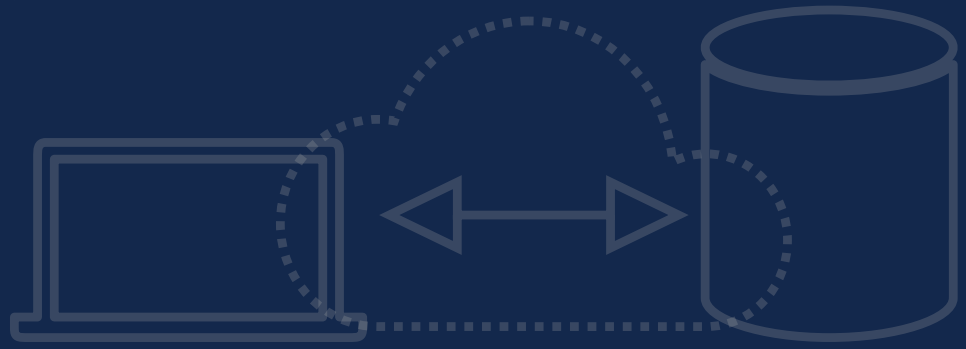
- Compressed HTTP is enabled via `Accept` headers
- User agent sends header indicating compression acceptance (gzip or deflate). For example, consider Apache Server using mod_gzip to send compressed content or not.
- Works only on text (HTML, CSS, JS), but with compression up to 70% or more
- Time to first byte increased so high speed connections may not see as much benefit, though bandwidth is saved. Lower speed clearly sees benefits!

HTTP Compression Example



Compression Considerations

- Increased origin server CPU – or wasted cycles?
- TTFB vs TTLB consideration
- Consider Compress/Decompress times
- Really no reason except at very small or very large sizes you shouldn't be doing HTTP compression
  - Do as the big sites do!



Response Headers

- **`Server`** – The server's name and version which can be problematic if divulges useful details for intrusion. Some complain that obfuscation of the server header is security by obscurity, but generally it is considered a best practice not to show implementation details where possible.

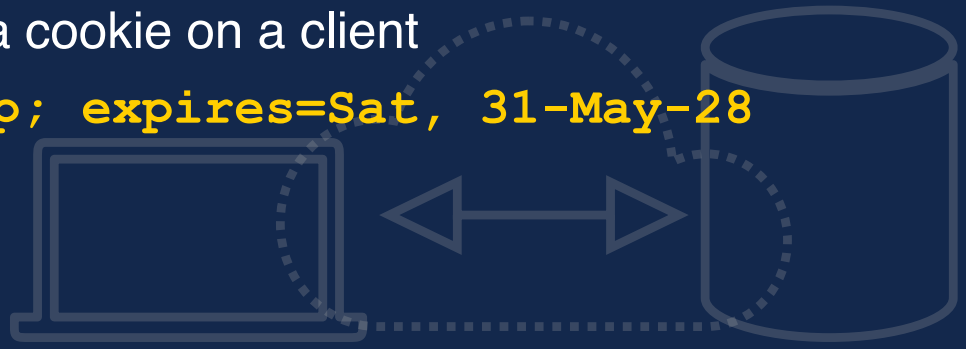
```
Server: Apache/2.2.15 (CentOS)
```

- **`Vary`** – Tells client & proxy caches which headers were used for content negotiation to address cache disambiguation

```
Vary: User-Agent, Accept
```

- **`Set-Cookie`** – This is how a server sets a cookie on a client

```
Set-Cookie: id=234; path=/shop; expires=Sat, 31-May-28 15:00:00 GMT; secure
```



A Closer Look at Entity Headers

- **`Allow`** – Lists the request methods that can be used on the entity. Sent via OPTIONS requests mostly.

```
Allow: GET, HEAD, POST
```

- **`Location`** – Gives the alternate or new location of the entity and is commonly used with 3xx response codes (redirects)

```
Location: http://www.ibm.com/us/
```

- **`Content-Encoding`** – specifies encoding performed on the body of the response and corresponds to Accept-Encoding request header

```
Content-Encoding: gzip
```

More Entity Headers

- `Content-Length` – The size of the entity body in bytes. The header may be absent in streamed value and can be different than the actual size due to compression.

```
Content-Length: 9001
```

- `Content-Location` – The actual URL of the resource if different than its request URL.

```
Content-Location: http://www.foo.com/home.html
```

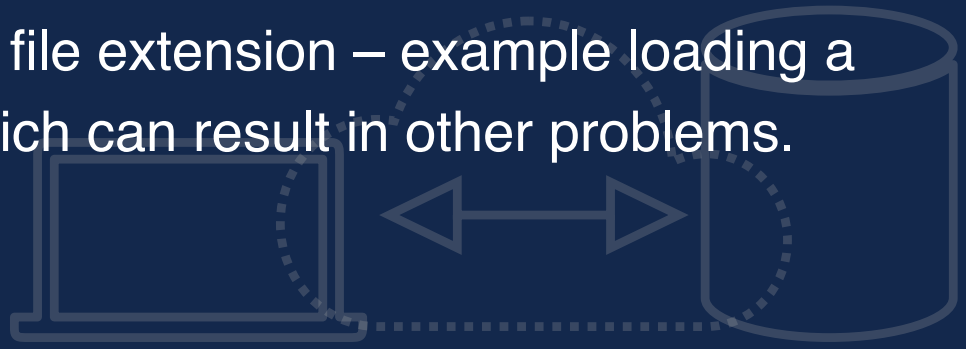
More Entity Headers

- `Content-Type` – specifies Media (MIME) type of the entity body

  ```
  Content-Type: image/png
  ```

- This is the most important header to the browser. The data in this header tells the browser what it is receiving. Now it should make sense why file extensions don't really matter and are arbitrary.
  - Server: file extension -> Mime type.
  - Browser: Mime type -> Action (display, download, etc.)

- Note: Without HTTP browser relies on file extension – example loading a file off a local disk or MIME sniffing which can result in other problems.



Why do I care?

- Because sometimes you need to stamp outgoing data on the server-side with the appropriate MIME type

```
4  header("Content-Type: text/xml");
5
6  $datemsg =  "Hello World to " . rawurlencode($name) ." at ". date("h:i:s A");
7
8  echo <<< END_OF_FILE
9      <hello>
10         <message id="date">$datemsg</message>
11     </hello>
12 END OF FILE
```

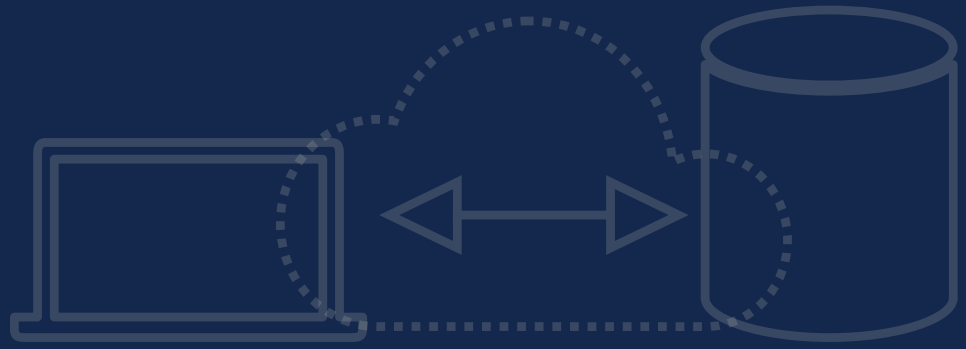
More Entity Headers : Caching Related

- **`Expires`** – Gives expiration for the instance of the resource for use in caching

  ```
  Expires: Sat, 31-May-23 19:00:00 GMT
  ```

- **`Last-Modified`** – Date/time the entity was last changed (or created)

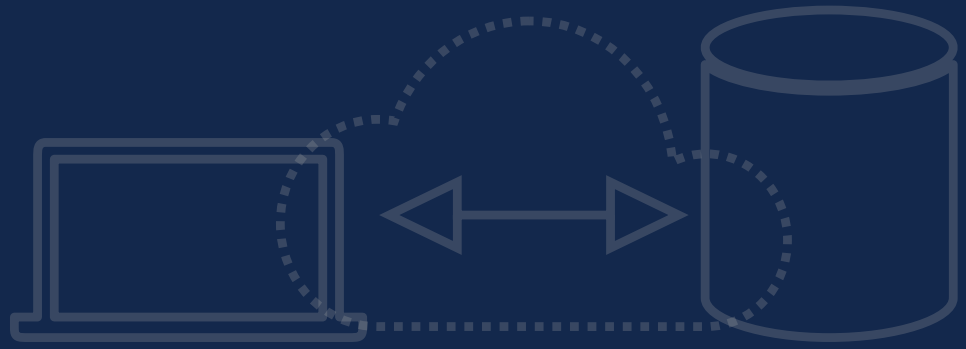
  ```
  Last-Modified: Fri 30-May-03 09:00:00 GMT
  ```



More Entity Headers : Caching Related

- **`Etag`** – Uniquely identifies a particular instance of a given resource. The header is used with conditional request headers to validate cached instances of the resource

  `Etag: adkskdashjgk07563AF`



Why do I care?

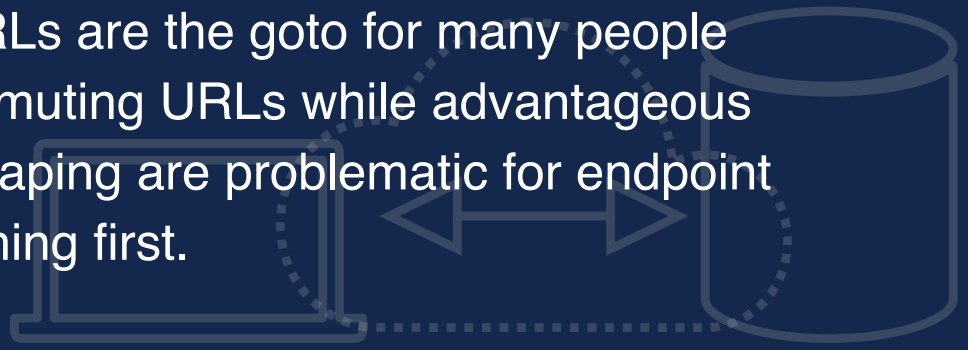
- Well you could go beyond basic cache-control and pragma headers and set Expires and set e-tags ultimately you can run into huge problems if things do not want to purge cache.

- Ultimately since the URL of an object is the cache key you may be forced to use a query string or alternate file name to force misconfigured caches to stop causing you problems

  /logo.gif?ts=324242342

  /logo324cads3c.gif

- Unfortunately, observationally modified URLs are the goto for many people before setting cache control headers. Permuting URLs while advantageous for the forcefulness and helpful for anti-scraping are problematic for endpoint stability. Do not start here, figure out caching first.



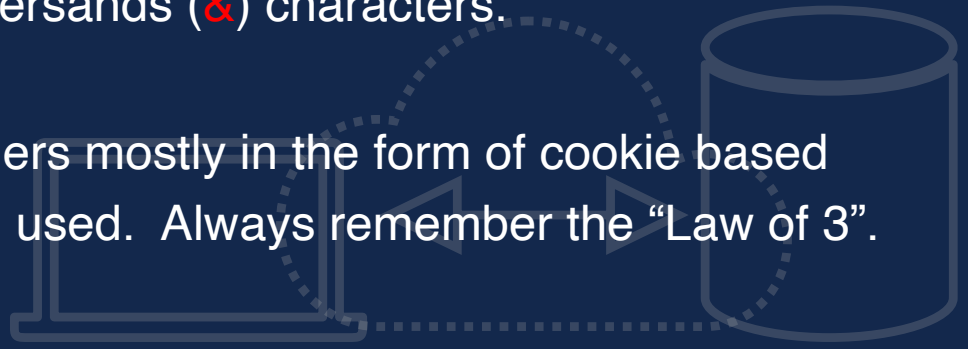
Sending data via HTTP

- Traditionally data can be sent to a server-application in two primary ways:
    1. Query String sent via a GET request
    2. Data body sent via a POST request

- Note: Using JS we can use all HTTP methods via JavaScript, but for now we focus on the two primary methods.

- In both cases the data is encoded in a special manner called x-www-form-urlencoded which replaces spaces with + symbols, special characters with %hex values equivalent to the particular special character being “escaped” and separates individual arguments to be passed with ampersands (&) characters.

- Note: Data may also be sent via HTTP headers mostly in the form of cookie based data. Though other HTTP headers could be used. Always remember the “Law of 3”.



Sending Data with GET

- In the case of GET we see submitted data (often from a fill-in form though may be hard coded in links) with the actual request URL
- The passed data is called the query string and follows a ? Character in the URL
  - Example: http://www.example.com/query.php?name=Thomas
  - Example: http://www.example.com/track.php?x=5&y=7
- These “dirty URLs” have potential downsides including:
  - Technology exposure – “visual reconnaissance”
  - Easy fiddling of parameters
  - Poor usability (may be a good thing as well)
  - Lack of long term maintainability
  - Size limit is dependent on URL size limits
- However, GET string based URLs are portable – you can bookmark them, send to friends, etc.

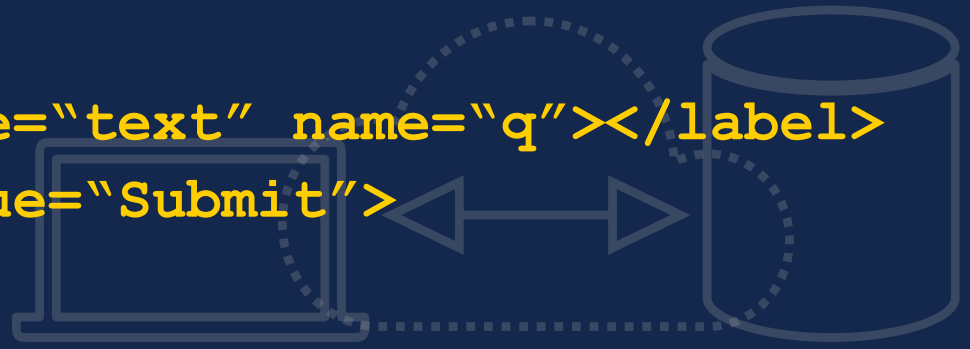
Sending Data with GET Contd.

- The GET based data can be submitted in one of two ways
  - Hard-coded into a link

```
<a href="http://www.google.com/search?
q=Web+server+software">Run query</a>
```

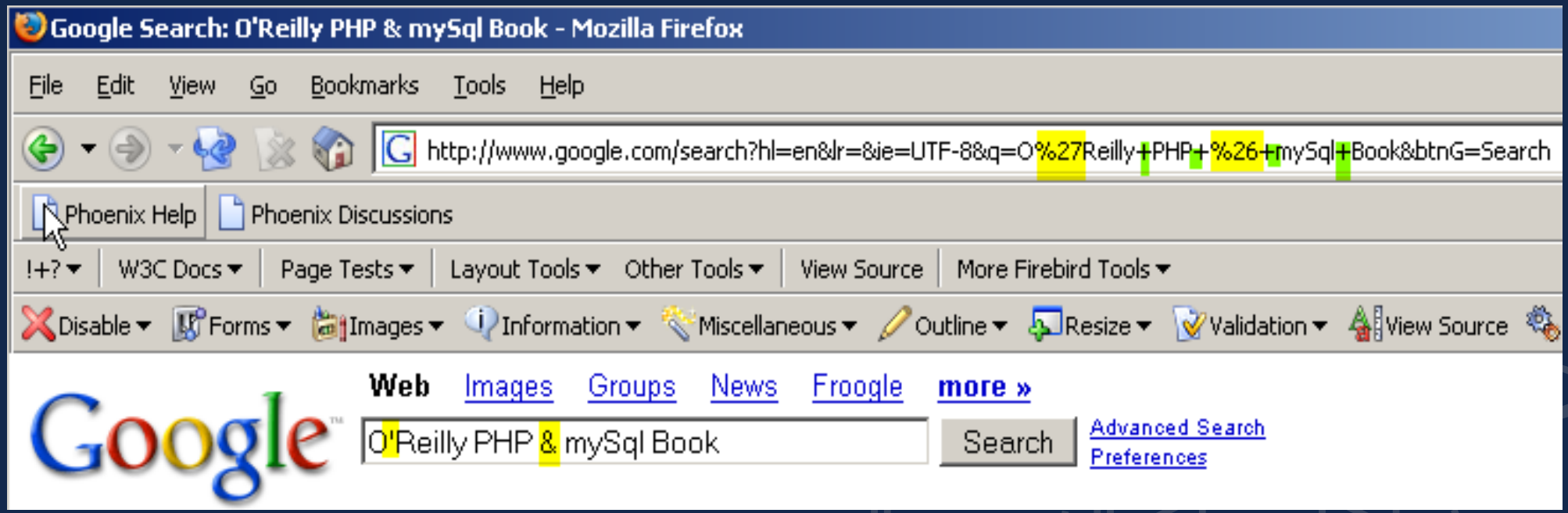
  - As a result of a form submission

```
<form action="http://www.google.com/search"
method="get">
<label>Query: <input type="text" name="q"></label>
<input type="submit" value="Submit">
</form>
```



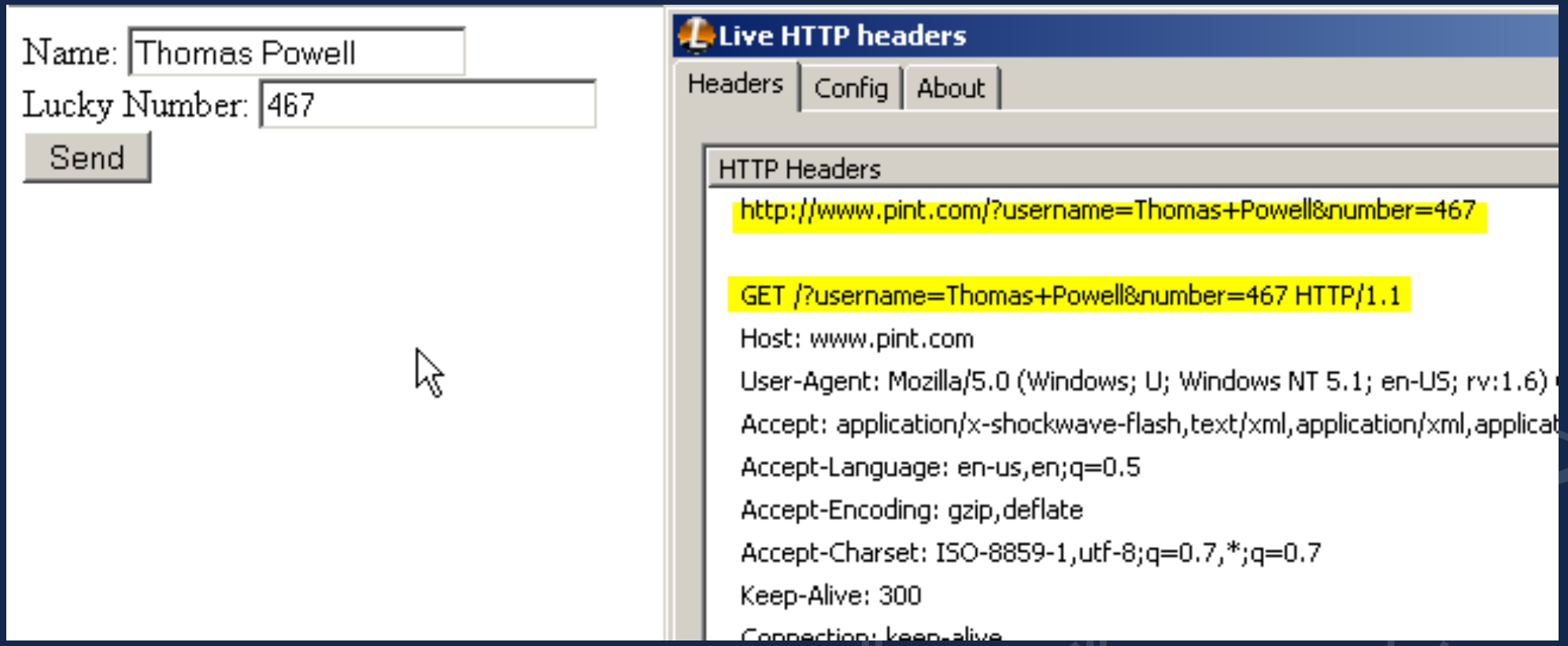
Sending Data with GET Contd.

- Now it should start to make sense what query strings mean and how they are formed



Sending Data with GET Contd.

- Behind the scenes you see that indeed the data is transmitted in the request method itself



Sending Data with Post

- In the case of POST you always generate the request either programmatically or more likely with a form

```
<form action="http://www.example.com/programs/submit-query"
method="post">
<label for="query">Query: </label><input type="text" name="query">
<input type="submit" value="Submit">
</form>
```

- The POST request sends the data in the message body but does so in x-www-form-urlencoded as well so we might have a message body like

```
Name=Al+Smith&Age=30&Sex=male
```

- No size limit, but issues with browsers have to address lack of redos "Repost form data?"

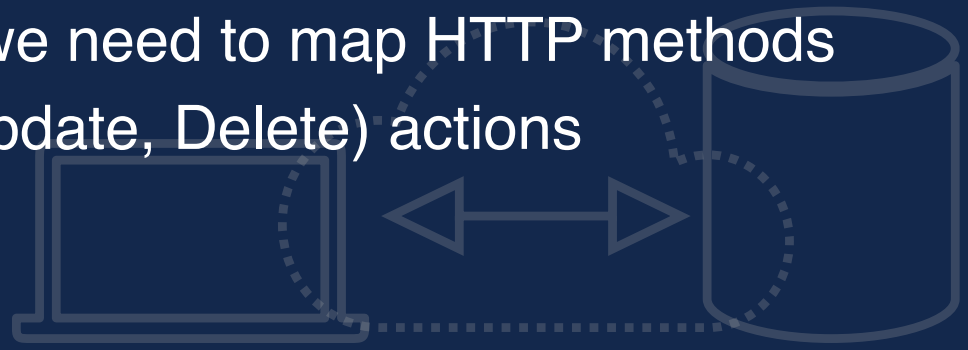
Sending Data with Post Contd.

- The network trace shows the difference between POST and GET

```
http://www.pint.com/

POST / HTTP/1.1
Host: www.pint.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.6) Gecko/20040206 Firefox/0.8
Accept: application/x-shockwave-flash,text/xml,application/xml,application/xhtml+xml,text/html;q=0
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: cookies=true; CFID=3441521; CFTOKEN=37652414
Content-Type: application/x-www-form-urlencoded
Content-Length: 33
    username=Thomas+Powell&number=345
```

Why do I care?

- GET and POST have different uses
- GET used when request is idempotent - meaning multiple requests return same result. POST should be used when you change the state of the server
- Lots of folks will often use GET for state changes because of ease of coding
  - Downsides – inadvertent state changes by spiders, browsers, etc.
- Under REST patterns we will see we need to map HTTP methods to related CRUD (Create, Read, Update, Delete) actions

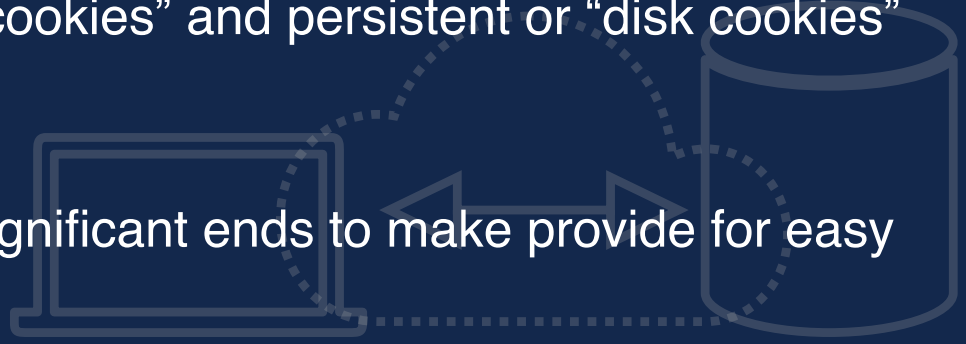


HTTP Considerations

- HTTP is a stateless protocol
  - No “memory” from one request to the next

- Question: How can you keep track of information from one page to the next?
- Answer:
  - Hidden Form fields that are posted backed to the server
    - E.g. Microsoft’s VIEWSTATE value in .NET
  - Data posted in dirty URL strings
  - Cookies
    - Two types – memory or “session cookies” and persistent or “disk cookies”
  - Client Side Storage (yikes!)

- Many programming environments go to significant ends to make provide for easy state management – more on this later!

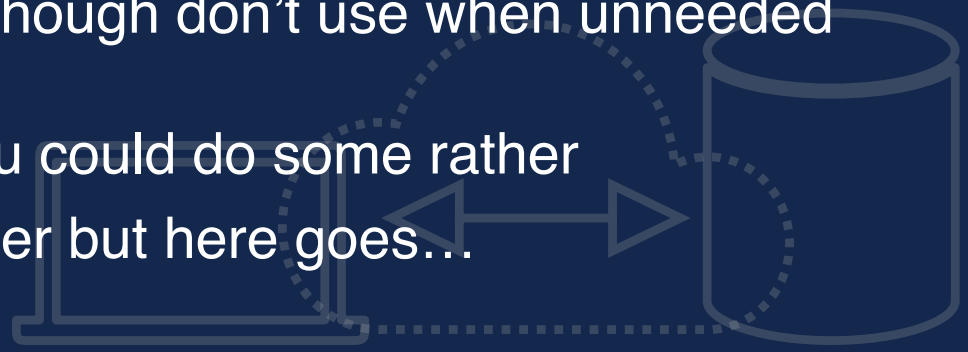


HTTP Futures

- It is clear (particularly when we look later at performance and security) that HTTP is a bit too simple for our modern Web needs.

- What should we do?
  - HTTP 2 or SPDY – yep that's working right now you use it with many Google properties every day!
    - Tunnel over SSL, not optimal but only way and it works today!
  - Parallel protocol for app part
    - See Web Sockets, be careful though don't use when unneeded

- Finally if you could own both ends you could do some rather interesting things … I'm no fortuneteller but here goes…



Summary

- HTTP is a simple stateless textual application protocol at the heart of the web
- Respect the “Law of Three”
- While it is simple do not skip the details of methods, response codes, and especially headers
- Everything is there for a reason…sweat those details!
  - Mastery of HTTP increases understanding of web development
  - Poor use of HTTP can result in performance and security problems