



An Overview of the HTTP Protocol as covered in RFCs

Contents

1. Introduction and History	3
2. HTTP Versions	3
3. HTTP Protocol Request/Response	3
4. HTTP/1.0 and earlier	4
5. HTTP Keep-Alive	4
6. HTTP Pipelining	4
7. HTTP	5
8. HTTP Intermediaries: Proxy	5
9. HTTP Intermediaries: Tunnel	6
10. HTTP Methods	6
11. HTTP/1.1 Request Header Examples	6
12. HTTP/1.1 Response Header Examples	6
13. Cookies	7
13.1 Types of Cookies	7
14. Authentication	8
15. User Agent	9
16. Content Negotiation	9
17. Transfer Encoding	9
17.1 Chunked Transfer Encoding	10
18. MIME Encoding	10
18.1 What is it used for? (MIME)	10
18.2 MIME Format	10
18.3 How is it used? (MIME)	11
18.4 Support for different languages (MIME)	11
18.5 Encoding (MIME)	11
18.6 Base 64 encoding Example (MIME)	11
18.7 Sending large Messages (MIME)	12
19. HTTP Caching	12
19.1 Preventing Caching	12
19.2 Allowing Caching	12
19.3 Cache Validation and the 304 response	13
20. SPDY	13
21. What is HTTP/2?	13
21.1 HTTP/2 Specification	13
21.2 HTTP/2 Over TLS (h2)	14

21.3 HTTP/2 Over TCP (h2c)	14
21.4 TCP Connections - HTTP 1.1 versus HTTP/2.....	14
21.5 HTTP/2.0 Prioritized Requests	14
21.6 HTTP/2.0 Compressed Headers	14
21.7 HTTP/2.0 Push	14
21.8 HTTP/2 Multiplexing	15
22. How to Troubleshoot - Dev Tools	15
23. How to Troubleshoot – Wireshark.....	16
24. REST (Representational State Transfer).....	18
25. REST HTTP Verbs.....	18
25.1 GET	19
25.2 POST	19
25.3 PUT	19
25.4 DELETE.....	20
26. WebSocket.....	20
27. Protocol handshake.....	20
28. HTTP Long Polling	21
29. Web DAV	21
30. JWT (JSON Web Token) – SSO	22
31. JWT (JSON Web Token)	22

1. Introduction and History

Request for Comments (RFC) denotes a type of publication from the Internet Engineering Task Force (IETF) and Internet Society (ISOC). IETF and ISOC are the technical development and standard-setting bodies for the Internet and official documents of Internet specifications, communications protocols, procedures, and events. RFC covers the memorandum, behaviors, research, and innovations associated with the operations of the Internet and systems connected to the internet. RFC is submitted for peer review to convey new concepts and information. Steve Crocker created RFC documents in 1969 to record informal notes about the development of ARPANET (Advanced Research Projects Agency Network).

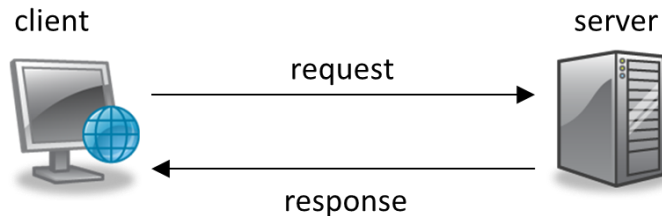
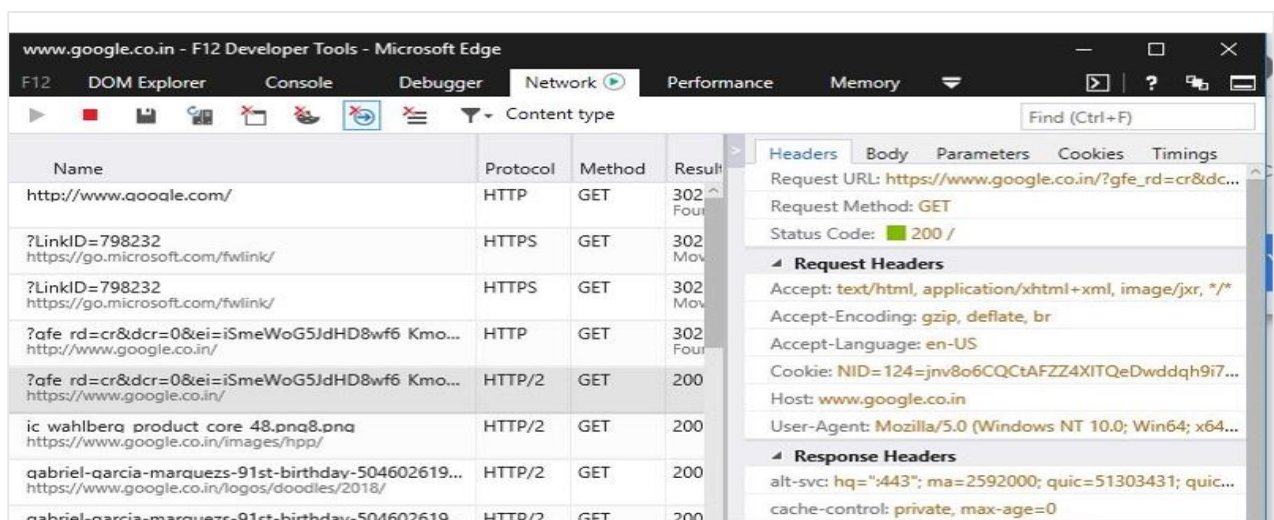
2. HTTP Versions

Following lists specify Application protocol for distributed hypermedia

- First documented in 1991 (HTTP/0.9)
- HTTP/1.0 introduced in 1996 (RFC 1945)
- HTTP/1.1 updated in 1999 (RFC 2616)
- HTTP/2.0 updated in 2015 (RFC 7540)

3. HTTP Protocol Request/Response

Hyper Text Transfer (HTTP) Protocol Request/Response includes Client and server exchange request/response messages, which uses the TCP protocol. For Client and server exchange request/response, the typical port number is 80.

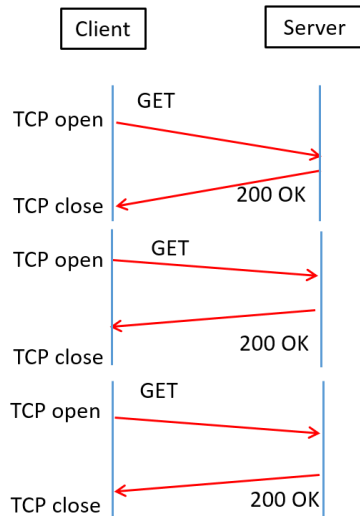



Name	Protocol	Method	Result
http://www.google.co.in/	HTTP	GET	302 Found
?LinkID=798232 https://go.microsoft.com/fwlink/	HTTPS	GET	302 Moved
?LinkID=798232 https://go.microsoft.com/fwlink/	HTTPS	GET	302 Moved
?afe_rd=cr&dc=0&ei=iSmeWoG5JdHD8wf6 Kmo... http://www.google.co.in/	HTTP	GET	302 Found
?afe_rd=cr&dc=0&ei=iSmeWoG5JdHD8wf6 Kmo... https://www.google.co.in/	HTTP/2	GET	200
ic_wahlberq_product_core_48.png8.png https://www.google.co.in/images/hpp/	HTTP/2	GET	200
gabriel-garcia-marquezs-91st-birthday-504602619... https://www.google.co.in/logos/doodles/2018/	HTTP/2	GET	200
gabriel-garcia-marquezs-91st-birthday-504602619...	HTTP/2	GET	200

Headers	Body	Parameters	Cookies	Timings
Request URL: https://www.google.co.in/?gfe_rd=cr&dc...				
Request Method: GET				
Status Code: 200 /				
Request Headers				
Accept: text/html, application/xhtml+xml, image/jxr, */*				
Accept-Encoding: gzip, deflate, br				
Accept-Language: en-US				
Cookie: NID=124=jnv8o6CQCtAFZZ4XITQeDwddqh9i7...				
Host: www.google.co.in				
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64...				
Response Headers				
alt-svc: hq="443"; ma=2592000; quic=51303431; quic...				
cache-control: private, max-age=0				

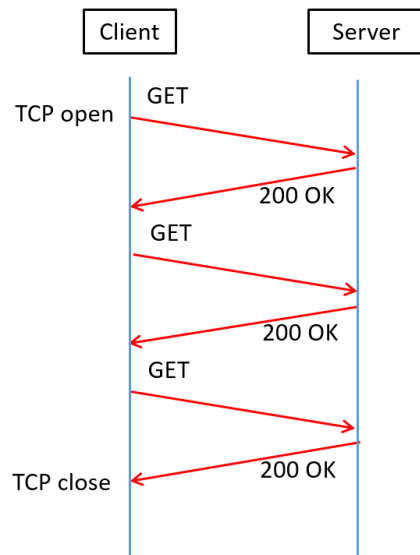
4. HTTP/1.0 and earlier

Before HTTP/1.1, each HTTP request used a separate TCP connection as shown in the figure below.



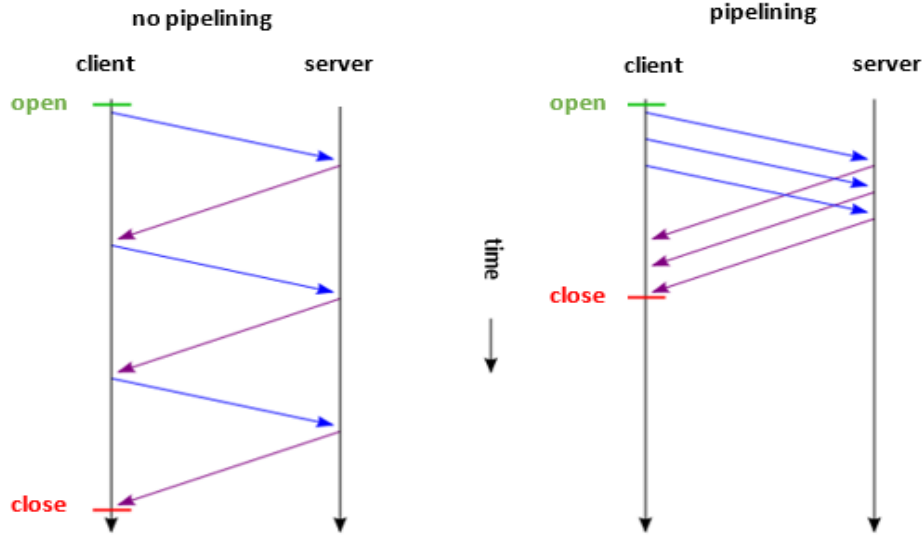
5. HTTP Keep-Alive

HTTP/1.1 introduced the keyword “Keep-Alive.” TCP connections reused for multiple HTTP requests. HTTP uses the keyword “Keep-Alive” in the “Connection” header to denote that the connection is open for additional messages. “Keep-Alive” exists default in HTTP/1.1 while in HTTP/1.0, the default is to use a new connection for each request or reply pair.



6. HTTP Pipelining

HTTP/1.1 pipelining is a technique in which multiple HTTP requests sent on a single TCP connection without waiting for the corresponding responses.

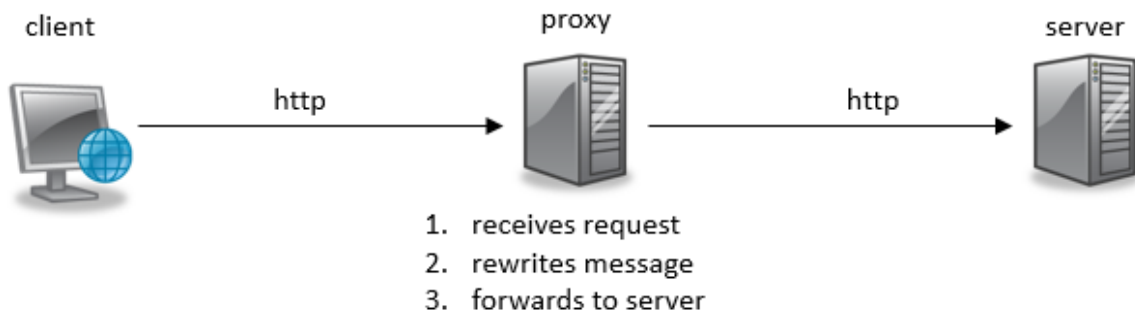


7. HTTP

Hyper Text Transfer Protocol, a direct connection between client and server. It serves as intermediaries in the request/response chain that include the following elements:

- Proxy
- Gateway
- Tunnel

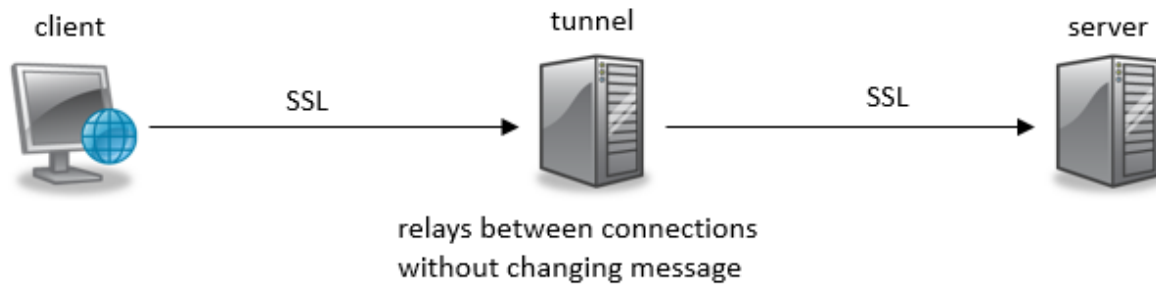
8. HTTP Intermediaries: Proxy



```
GET http://cryptome.org/HTTP/1.1
Host: cryptome.org
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en; rv:1.9.0.3) Gecko/20080528 Epiphany/2.22 Firefox/3.0
Accept: text/html, application/xhtml+xml, application/xml, q=0.9, */*, q=0.8
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1, utf-8, q=0.7, *; q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
If-Modified-Since: Tue, 14 Oct 2008 13:59:19 GMT
If-None-Match: "e01922-62e9-45937059ec2de"
Cache-Control: max-age=0
```

9. HTTP Intermediaries: Tunnel

HTTP tunneling is the process in which communications encapsulated by using the HTTP protocol. An HTTP tunnel often used for network locations restricted connectivity or are behind firewalls or proxy servers. It includes end-to-end encryption and SSL port 8100.



10. HTTP Methods

- GET - retrieves whatever information identified by the Request-URL
- POST - sends data to the server for updates.
- PUT - requests that the enclosed entity stored under the supplied Request-URL.
- DELETE - requests the origin server to delete the resource identified by the Request-URL.
- HEAD - Similar to the GET method, except that the server must not return a message-body in the response.
- TRACE - Allows the client to see what received at the other end of the request chain and use that data for testing

11. HTTP/1.1 Request Header Examples

- Accept: specify desired media type of response
- Accept-Language: specify the desired language of response
- Date: date/time at which the message originated
- Host: host and port number of requested resource
- If-Match: conditional request
- Referrer: URL of the previously visited resource
- User-Agent: identifier string for a Web browser or user agent

12. HTTP/1.1 Response Header Examples

- Allow: lists methods supported by request URI
- Content-Language: the language of representation
- Content-Type: media type of representation
- Content-Length: length in bytes of representation
- Date: date/time at which the message originated
- Expires: date/time after which response is considered stale
- ETag: an identifier for a version of the resource (message digest)
- Last-Modified: date/time at which representation was last change

13. Cookies

A cookie is a text file stored on the hard drive (more precisely in the browser folder) when visiting a website.

13.1 Types of Cookies

- **Session:** They expire when the browser is closed or remained idle. For example, they are used in e-commerce websites to continue browsing without losing chosen items in the cart.
- **Permanent:** They persist even after the browser exits. They have an expiration date though, as per law that limits lasting more than six months. They assist in remembering passwords and login information to reduce the need of re-entering them every time.
- **Third party:** Cookies attributes usually correspond to the website domain they are on and not for third-party cookies. Third-Party websites such as advertisers install these cookies. Third-party cookies collect data about browsing habits and track them across different websites. Websites such as Facebook, Flickr, Google Analytics, Google Maps, Google Plus, Sound Cloud, Tumblr, Twitter, and YouTube use third-party cookies.

The Set-Cookie HTTP response header sends cookies from the server to the user agent. This header from the server tells the client to store a cookie.

```
Set-Cookie: <cookie-name>=<cookie-value>
```

This header from the server tells the client to store a cookie.

```
HTTP/1.0 200 OK
Content-type: text/html
Set-Cookie: yummy_cookie=newcookie
Set-Cookie: tasty_cookie=session
```

Now, with every new request to the server, the browser will send back all previously stored cookies to the server using the Cookie header.

```
GET /sample_page.html HTTP/1.1
Host: www.example.org
Cookie: yummy_cookie=newcookie; tasty_cookie=session
```

Request Headers
Accept: text/html, application/xhtml+xml, image/jxr, */*
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US
Cookie: NID=124=jnv8o6CQCtAFZZ4XITQeDwddqh9i7B0-1x39srgCUc...
Host: www.google.co.in
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/...
Response Headers
alt-svc: hq=":443"; ma=2592000; quic=51303431; quic=51303339; quic=...
cache-control: private, max-age=0
content-type: text/html; charset=UTF-8
date: Tue, 06 Mar 2018 05:39:22 GMT
expires: -1
p3p: CP="This is not a P3P policy! See g.co/p3phelp for more info."
server: gws
set-cookie: 1P_JAR=2018-03-06-05; expires=Thu, 05-Apr-2018 05:39:2...
set-cookie: NID=125=CRoaOa5KU2HgEDrhHlfy1r6SEf25-0R82YRdhesd...
strict-transport-security: max-age=3600
x-frame-options: SAMEORIGIN
x-xss-protection: 1; mode=block
3 s taken (DOMContentLoaded: 5.06 s, load: 8.97 s)

14. Authentication

HTTP supports the use of several authentication mechanisms to control access to pages and other resources. These mechanisms are all based around the use of the 401-status code and the WWW-Authenticate response header.

The most widely used HTTP authentication mechanisms are as follows:

- **Basic:** The client sends the user name and password as unencrypted base64 encoded text. It should only be used with HTTPS, as the password can be easily captured and reused over HTTP.
- **Digest:** The client sends a hashed form of the password to the server. Though the password cannot catch over HTTP, it may be possible to replay requests using the hashed password.
- **NTLM:** This uses a secure challenge/response mechanism that prevents password capture or replay attacks over HTTP. However, the authentication is per connection and will only work with HTTP/1.1 persistent connections. For this reason, it may not work through all HTTP proxies and can introduce large numbers of network roundtrips if connections regularly closed by the web server.

* Force Authentication

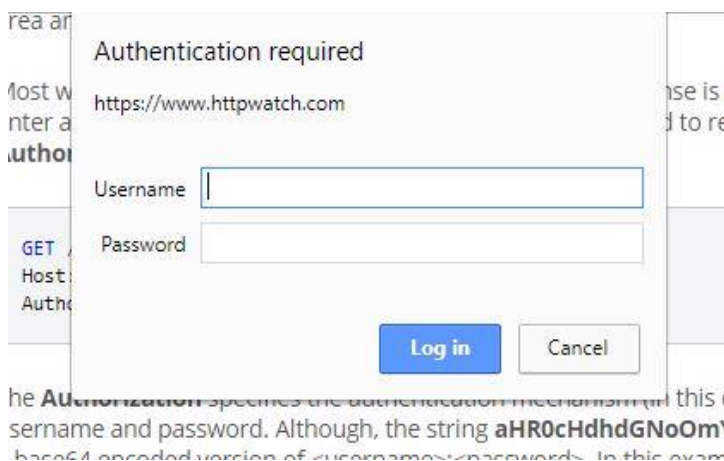
If an HTTP receives an anonymous request for a protected resource it can force the use of Basic authentication by rejecting the request with a **401** (Access Denied) status code and setting the **WWW-Authenticate** response header as shown below:

```
HTTP/1.1 401 Access Denied
WWW - Authenticate: Basic realm= "My Server"
Content-Length: 0
```

The word **Basic** in the **WWW-Authenticate** selects the authentication mechanism that the HTTP client must use to access the resource. The **realm** string can be set to any value to identify the secure area and may be used by HTTP clients to manage passwords.

Most web browsers will display a **login** dialog when this response is received, allowing the user to enter a **Username** and **Password**. This information is then used to retry the request with an Authorization request header:

```
GET /securefiles/ HTTP/1.1
Host: www.httpwatch.com
Authorization: Basic aHR0cHdhbGNoCmY=
```



15. User Agent

In computing, a user agent is a software (a software agent) that is acting on behalf of a user. One common use of the term refers to a web browser telling website information about the browser and operating system. This allows the website to customize content for the capabilities of a particular device, but also raises privacy issues. In HTTP, the User-Agent string often used for content negotiation, where the origin server selects suitable content or operating parameters for the response.

16. Content Negotiation

Content negotiation refers to mechanisms defined as a part of HTTP that make it possible to serve different versions of a document at the same URL, so that user agents can specify which version fits their capabilities the best.

HTTP provides for several different content negotiation mechanisms including:

- Server-driven
- Agent-driven
- Transparent
- hybrid

Client Browser Request:

```
Accept-Language: de; q=1.0, en; q=0.5  
Accept: text/html; q=1.0, text/*; q=0.8, image/gif; q=0.6, image/jpeg; q=0.6, image/*; q=0.5, */*; q=0.1
```

Server-driven or proactive content negotiation is performed by algorithms on the server, which choose among the possible variant representations. This is commonly performed based on a user-agent provided acceptance criteria. To summarize how this works, when a user agent submits a request to a server, the user agent informs the server what media types it understands with ratings of how well it understands them. More precisely, the user agent provides an Accept HTTP header that lists acceptable media types and associated quality factors. The server is then able to supply the version of the resource that best fits the user agent's needs

Example Negotiation Headers

- Accept: Which media types are acceptable for the response, such as "application/json," "application/xml" or a custom media type such as "application/vnd.example+xml."
- Accept-Charset: Which character sets are acceptable, such as UTF-8 or ISO 8859-1.
- Accept-Encoding: Which content encodings are acceptable, such as gzip.
- Accept-Language: The preferred natural language, such as "en-us."

17. Transfer Encoding

The Transfer-Encoding header specifies the form of encoding used to transfer the entity to the user. Transfer-Encoding is a hop-by-hop header applicable to a message between two nodes, not to a resource itself. Each segment of a multi-node connection can use different Transfer-Encoding values. If you want to compress data over the whole connection, use the end-to-end header Content-Encoding header instead.

```
Transfer-Encoding: chunked  
Transfer-Encoding: compress  
Transfer-Encoding: deflate  
Transfer-Encoding: gzip  
Transfer-Encoding: identity  
  
// Several Values can be Listed, separated by a comma  
Transfer-Encoding: gzip, chunked
```

17.1 Chunked Transfer Encoding

Chunked transfer encoding is a streaming data transfer mechanism available in version 1.1 of the Hypertext Transfer Protocol (HTTP). In chunked transfer encoding, the data stream is divided into a series of non-overlapping "chunks." The chunks are sent out and received independently of one another. No knowledge of the data stream outside the processing chunk is necessary for both the sender and the receiver at any given time.

The size in bytes precedes each chunk. The transmission ends when a zero-length chunk is received. The chunked keyword in the Transfer-Encoding header indicates a chunked transfer. This encoding is beneficial if knowledge of the response size is unclear and data size is large.

Encoded data

```
HTTP/1.1 200 OK
Content-Type: text/plain
Transfer-Encoding: chunked

7\r\n
Mozilla\r\n
9\r\n
Developer\r\n
7\r\n
Network\r\n
0\r\n
\r\n
```

18. MIME Encoding

HTTP is largely a text-based protocol. Binary content needs some way of transmission. MIME is an acronym for Multipurpose Internet Mail Extension. It is used to describe message content types. MIME messages can contain the text, images, audio, video and other application-specific data (For examples: PDF files, Microsoft Word Documents, and so on).

18.1 What is it used for? (MIME)

It assists to make internet messages richer. It allows applications (and users) to exchange rich content other than text. It is an extension to the original email specification (RFC-822). RFC documents such as RFC-2045 through RFC-2049 define about MIME. A Request for Comments (RFC) is a document published by the Internet Engineering Task Force (IETF) describing an internet standard.

18.2 MIME Format

MIME types are defined using a **<type>/<subtype> [optional parameters]** format. Some typical examples are as follows:

MIME Type	Extension(s)
text/plain	txt
application/vnd.ms-excel	xls
application/pdf	pdf
text/html	htm; html
text/css	css

18.3 How is it used? (MIME)

MIME passes as a part of the content type of the message header.

- **Content-type: text/plain; charset="us-ascii"**
- The following example is a typical HTTP Response header (MIME highlighted)

```
HTTP/1.x 200 OK
Transfer-Encoding: chunked
Date: Sat, 28 Nov 2009 04:36:25 GMT
Server: LiteSpeed Connection: close
X-Powered-By: W3 Total Cache/0.8 Pragma: public
Expires: Sat, 28 Nov 2009 05:36:25 GMT
Cache-Control: max-age=3600, public
Content-Type: text/html; charset=UTF-8
Last-Modified: Sat, 28 Nov 2009 03:50:37 GMT
Vary: Accept-Encoding, Cookie, User-Agent
```

18.4 Support for different languages (MIME)

Message header

- Content-type field
- Put in the header by the client program creating the e-mail for use by the client program used to display the received message
- Charset= optional parameter; If absent, ASCII is assumed

```
Content-Type: text/plain; charset= "ISO-8859-1"
```

- **ISO-8859-1** character standard extends the basic character set of ASCII to include many of the accented characters used in languages such as German.

18.5 Encoding (MIME)

Binary files need to be "packaged" as text in order to be sent over the internet. MIME uses a BASE-64 binary encoding scheme to package the data for transfer. Because of this encoding, standard SMTP (Simple Mail Transfer Protocol), servers did not require any changes. Encoding transforms binary data into a string. Decoding changes the data back into its original form.

18.6 Base 64 encoding Example (MIME)

- **Normal Text:**

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nulla mattis pulvinar ligula. Ut quis neque ut
lorem mollis hendrerit. Curabitur rhoncus, neque vitae sodales condimentum.
```

- **Encoded Text:**

```
TG9yZW0gaXBzdW0gZG9sb3Igc2l0IGFtZXQsIGNvbml1Y
3RldHVyIGFkaXBpc2NpbmcgZWxpdC4gTnVsbgEgbWFO
dG1zIHBlbHZpbmFyIGxpZ3VsYS4gVXQgcXVpcyBuZXF1Z
SB1dCBsb3JlbSBtb2xsaXMgaGVuZHI1cm10LiBDdXJhYm10d
XIGcmhvbml1cywgbmVxdWUgdm10YWUgc29kYWxlcYBj b25kaW11bnR1bS4=
```

- An online tool, one can use to experiment can be found at [http://www.motobit.com/util/base64-decoder- encoder.asp](http://www.motobit.com/util/base64-decoder-encoder.asp)

18.7 Sending large Messages (MIME)

When sending large messages, the message client splits them into smaller parts. This type of message is called a multi-part message. Multi-part messages have one of the MIME content types such as **content-type = multipart/related** and **content-type = multipart/mixed**.

19. HTTP Caching

Web pages often contain content that remains unchanged for long periods. For example, an image containing a company logo may be used without modification for many years. It is wasteful in terms of bandwidth and round trips to repeatedly download images or other content that is not regularly updated.

HTTP supports caching so that content can be stored locally by the browser and reused when required. Of course, some types of data such as stock prices and weather forecasts are frequently changed and it is important that the browser does not display stale versions of these resources. By carefully controlling caching, it is possible to reuse static content and prevent the storage of dynamic data.

Browser caching is controlled by the use of the Cache-Control, Last-Modified and Expires response headers.

19.1 Preventing Caching

- Servers set the Cache-Control response header to no-cache to indicate that content should not be cached by the browser:

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

- Also, the Pragma header is also often used to stop caching by HTTP 1.0 proxies as they do not support the Cache-Control header:

```
Pragma: no-cache
```

19.2 Allowing Caching

The Cache-Control header can be set to one of the following values to allow caching:

- **<absent>**: If the Cache-Control header is not set, then any cache may store the content.
- **Private**: The content is intended for use by a single user and should only be cached locally in the browser.
- **Public**: The content may be cached in public caches (e.g. shared proxies) and private browser caches.

If the browser is to make effective use of cached content, two extra pieces of information should be supplied. The first is the modification date/time of the content. The server supplies this in the **Last-Modified** response header:

```
Last-Modified: Wed, 25 Feb 2015 12:00:00 GMT
```

The second piece of information is the expiration date that is specified with the **Expires** header:

```
Expires: Thu, 25 Feb 2016 12:00:00 GMT
```

This header will set the cache expiration to be **31536000** seconds or one year in the future:

```
Cache-Control: max-age=31536000
```

19.3 Cache Validation and the 304 response

There are a number of situations in which Internet Explorer needs to check whether a cached entry is valid:

- The cached entry has no expiration date and the content is being accessed for the first time in a browser session
- The cached entry has an expiration date but it has expired
- The user has requested a page update by clicking the Refresh button or pressing F5
- If the cached entry has the last modification date, IE sends it in the **If-Modified-Since** header of a GET request message:

```
GET /images/logo.gif HTTP/1.1
Accept: */*
Referer: http://www.google.com/
Accept-Encoding: gzip, deflate
If-Modified-Since: Wed, 25 Feb 2015 17:42:04 GMT
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64; Trident/7.0; rv:11.0) like Gecko
Host: www.google.com
```

- The server checks the **If-Modified-Since** header and responds accordingly. If the content has not been changed since the date/time specified, it replies with a status code of 304 and a response message that just contains headers:

```
HTTP/1.1 304 Not Modified
Content-Type: text/html
Date: Thu, 26 Feb 2015 10:00:04 GMT
```

20. SPDY

Not an acronym - pronounced 'speedy'

- Development between Google and Microsoft
- Preserves existing HTTP semantics – SPDY is purely a framing layer
- Basis for HTTP/2.0

Offers four improvements over HTTP/1.1:

- Multiplexed requests
- Prioritized requests
- Compressed headers
- Server push

21. What is HTTP/2?

HTTP/2 uses a single, multiplexed connection. Maximum connection limit per domain can be ignored. HTTP/2 compresses header data and sends it in a concise, binary format. Better than the plain text format used previously. Less need for popular HTTP 1.1 optimizations.

21.1 HTTP/2 Specification

- **Started with SPDY - draft 3**
- Comprised out of two specifications
 - HTTP/2 – RFC7540
 - HPACK (header compression) – RFC7541

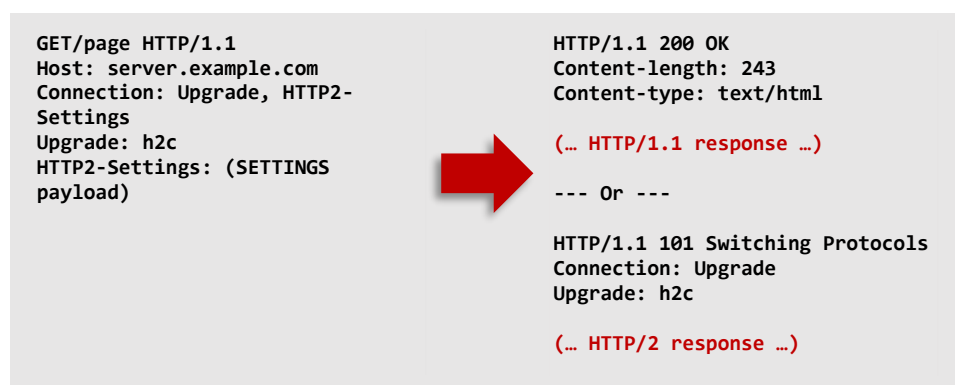
- Implementations
 - HTTP/2 over TLS (h2)
 - HTTP/2 over TCP (h2c)

21.2 HTTP/2 Over TLS (h2)

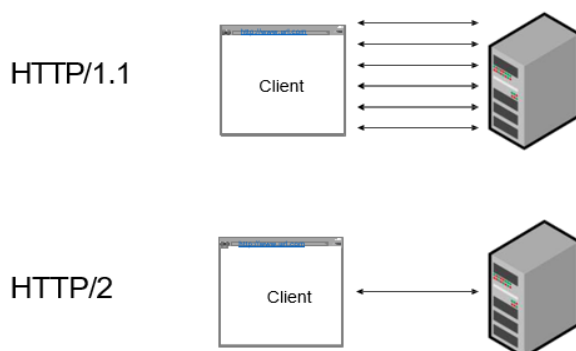
HTTP/2 shipped with TLS as optional. Firefox and Chrome developer teams stated they would only implement HTTP/2 over TLS. Today, only HTTPS:// is allowed for HTTP/2. TLS must be at least v1.2, with cipher suite restrictions.

21.3 HTTP/2 Over TCP (h2c)

It Uses the Upgrade header. It plans to support on IE, already supported in CURL.



21.4 TCP Connections - HTTP 1.1 versus HTTP/2



21.5 HTTP/2.0 Prioritized Requests

A connection may contain multiple streams (each of which consists of a sequence of frames). Each stream has a 31-bit identifier such as Odd for client-initiated and Even for server-initiated. Each stream has another 31-bit integer that expresses its relative priority. Frames from higher priority streams sent before those from lower priority streams and allow asynchronous stream processing (unlike HTTP/1.1 Pipelining).

21.6 HTTP/2.0 Compressed Headers

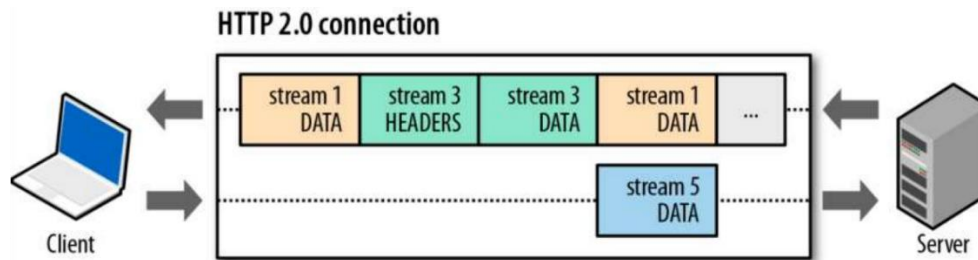
HTTP/1.1 can compress message bodies using gzip or deflate and sends headers in plain text. HTTP/2.0 also provides the ability to compress message headers.

21.7 HTTP/2.0 Push

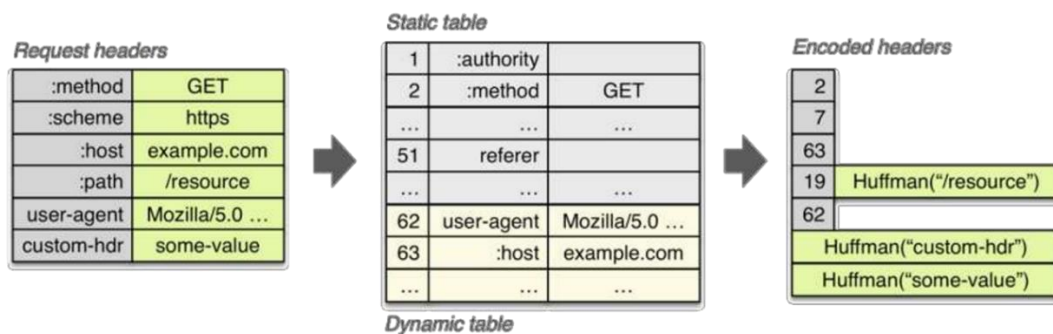
HTTP/1.1 servers only send messages in response to requests. HTTP/2.0 enables a server to pre-emptively send (or push) multiple associated resources to a client in response to a single request.

21.8 HTTP/2 Multiplexing

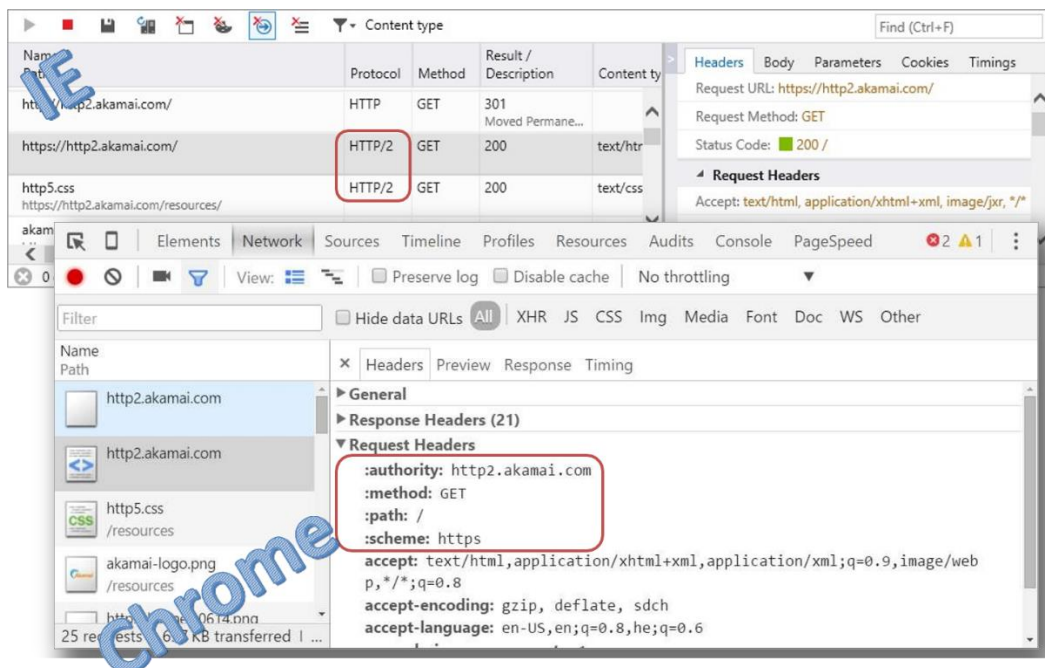
Each request/response stream has an ID. Streams comprise of frames (Header, Data...). A TCP connection can have multiple streams. Frames can be interleaved in the TCP channel. Stream dependencies control frame prioritization. Server (IIS/ASP.NET) sees streams as TCP Connection.



Header Compression (HPACK)



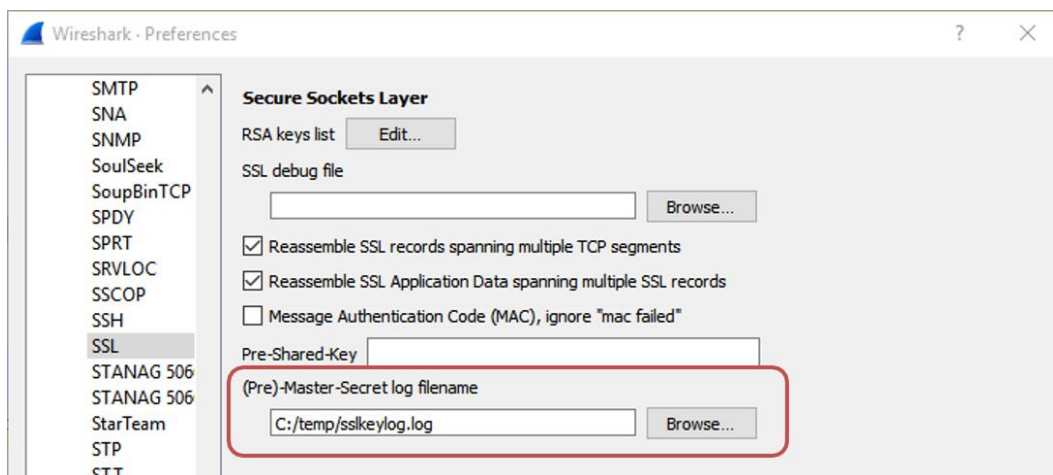
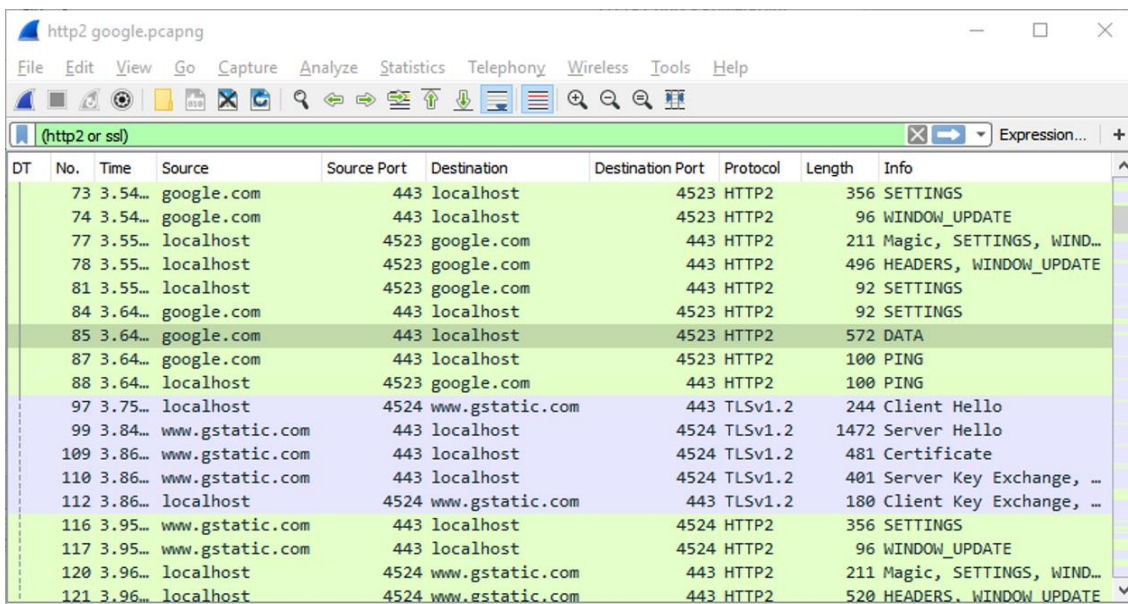
22. How to Troubleshoot - Dev Tools



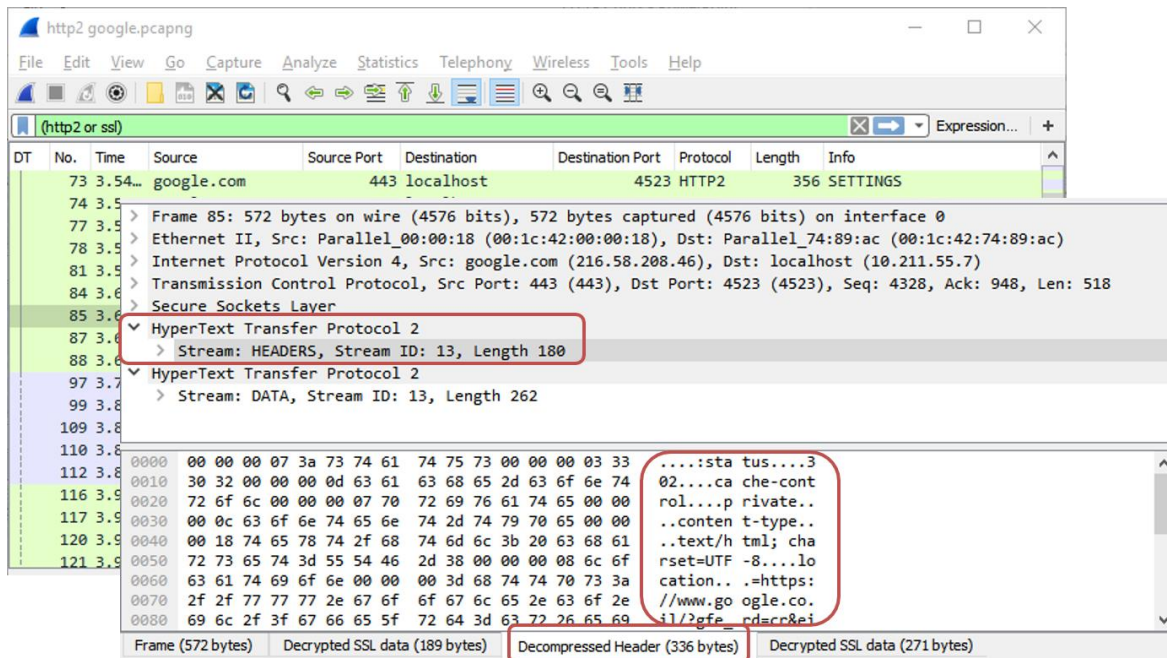
23. How to Troubleshoot – Wireshark

- Works with Chrome and Firefox only (Windows, Linux, Mac)
- Set **SSLKeyLogFile** for HTTPS sniffing
- Go to **Wireshark->Preferences->Protocols->SSL**

```
C: \> set SSLKEYLOGFILE=C:\temp\sslkeylog.log
```

DT	No.	Time	Source	Source Port	Destination	Destination Port	Protocol	Length	Info
	73	3.54...	google.com	443	localhost	4523	HTTP2	356	SETTINGS
	74	3.54...	google.com	443	localhost	4523	HTTP2	96	WINDOW_UPDATE
	77	3.55...	localhost	4523	google.com	443	HTTP2	211	Magic, SETTINGS, WIND...
	78	3.55...	localhost	4523	google.com	443	HTTP2	496	HEADERS, WINDOW_UPDATE
	81	3.55...	localhost	4523	google.com	443	HTTP2	92	SETTINGS
	84	3.64...	google.com	443	localhost	4523	HTTP2	92	SETTINGS
	85	3.64...	google.com	443	localhost	4523	HTTP2	572	DATA
	87	3.64...	google.com	443	localhost	4523	HTTP2	100	PING
	88	3.64...	localhost	4523	google.com	443	HTTP2	100	PING
	97	3.75...	localhost	4524	www.gstatic.com	443	TLSv1.2	244	Client Hello
	99	3.84...	www.gstatic.com	443	localhost	4524	TLSv1.2	1472	Server Hello
	109	3.86...	www.gstatic.com	443	localhost	4524	TLSv1.2	481	Certificate
	110	3.86...	www.gstatic.com	443	localhost	4524	TLSv1.2	401	Server Key Exchange, ...
	112	3.86...	localhost	4524	www.gstatic.com	443	TLSv1.2	180	Client Key Exchange, ...
	116	3.95...	www.gstatic.com	443	localhost	4524	HTTP2	356	SETTINGS
	117	3.95...	www.gstatic.com	443	localhost	4524	HTTP2	96	WINDOW_UPDATE
	120	3.96...	localhost	4524	www.gstatic.com	443	HTTP2	211	Magic, SETTINGS, WIND...
	121	3.96...	localhost	4524	www.gstatic.com	443	HTTP2	520	HEADERS, WINDOW_UPDATE



http2 google.pcapng

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

(http2 or ssl)

DT	No.	Time	Source	Source Port	Destination	Destination Port	Protocol	Length	Info
73	3.54...	google.com	443	localhost	4523	HTTP2	356	SETTINGS	

Frame 85: 572 bytes on wire (4576 bits), 572 bytes captured (4576 bits) on interface 0

Ethernet II, Src: Parallel_00:00:18 (00:1c:42:00:00:18), Dst: Parallel_74:89:ac (00:1c:42:74:89:ac)

Internet Protocol Version 4, Src: google.com (216.58.208.46), Dst: localhost (10.211.55.7)

Transmission Control Protocol, Src Port: 443 (443), Dst Port: 4523 (4523), Seq: 4328, Ack: 948, Len: 518

Secure Sockets Layer

HyperText Transfer Protocol 2

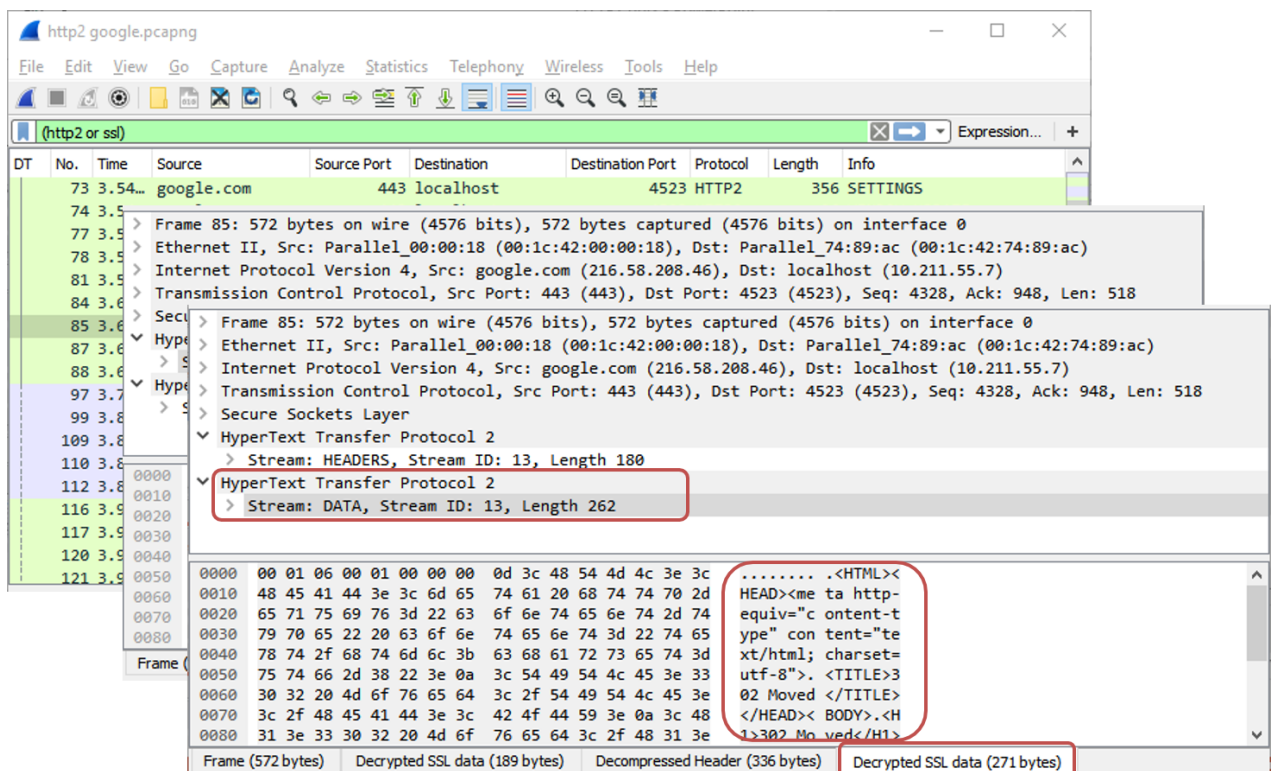
Stream: HEADERS, Stream ID: 13, Length 180

HyperText Transfer Protocol 2

Stream: DATA, Stream ID: 13, Length 262

0000 00 00 00 07 3a 73 74 61 74 75 73 00 00 00 03 33sta tus....3
0010 30 32 00 00 00 0d 63 61 63 68 65 2d 63 6f 6e 74 02....ca che-cont
0020 72 6f 6c 00 00 00 07 70 72 69 76 61 74 65 00 00 rol....p rivate..
0030 00 0c 63 6f 6e 74 65 6e 74 2d 74 79 70 65 00 00 ..conten t-type..
0040 00 18 74 65 78 74 2f 68 74 6d 6c 3b 20 63 68 61 ..text/h tml; cha
0050 72 73 65 74 3d 55 54 46 2d 38 00 00 00 08 6c 6f rset=UTF -8....lo
0060 63 61 74 69 6f 6e 00 00 00 3d 68 74 74 70 73 3a cation.. =https:
0070 2f 2f 77 77 77 2e 67 6f 6f 67 6c 65 2e 63 6f 2e //www.go ogle.co.
0080 69 6c 2f 3f 67 66 65 5f 72 64 3d 63 72 26 65 69 11/2gfe _ndacr8ei

Frame (572 bytes) Decrypted SSL data (189 bytes) Decompressed Header (336 bytes) Decrypted SSL data (271 bytes)



http2 google.pcapng

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

(http2 or ssl)

DT	No.	Time	Source	Source Port	Destination	Destination Port	Protocol	Length	Info
73	3.54...	google.com	443	localhost	4523	HTTP2	356	SETTINGS	

Frame 85: 572 bytes on wire (4576 bits), 572 bytes captured (4576 bits) on interface 0

Ethernet II, Src: Parallel_00:00:18 (00:1c:42:00:00:18), Dst: Parallel_74:89:ac (00:1c:42:74:89:ac)

Internet Protocol Version 4, Src: google.com (216.58.208.46), Dst: localhost (10.211.55.7)

Transmission Control Protocol, Src Port: 443 (443), Dst Port: 4523 (4523), Seq: 4328, Ack: 948, Len: 518

Secure Sockets Layer

HyperText Transfer Protocol 2

Stream: HEADERS, Stream ID: 13, Length 180

HyperText Transfer Protocol 2

Stream: DATA, Stream ID: 13, Length 262

0000 00 01 06 00 01 00 00 00 0d 3c 48 54 4d 4c 3e 3c<HTML><
0010 48 45 41 44 3e 3c 6d 65 74 61 20 68 74 74 70 2d HEAD><me ta http-
0020 65 71 75 69 76 3d 22 63 6f 6e 74 65 6e 74 2d 74 EQUIV="c ontent-t
0030 79 70 65 22 20 63 6f 6e 74 65 6e 74 3d 22 74 65 ype" con tent="te
0040 78 74 2f 68 74 6d 6c 3b 63 68 61 72 73 65 74 3d xt/html; charset=
0050 75 74 66 2d 38 22 3e 0a 3c 54 49 54 4c 45 3e 33 utf-8">. <TITLE>3
0060 30 32 20 4d 6f 76 65 64 3c 2f 54 49 54 4c 45 3e 02 Moved </TITLE>
0070 3c 2f 48 45 41 44 3e 3c 42 4f 44 59 3e 0a 3c 48 </HEAD>< BODY><.H
0080 31 3e 33 30 32 20 4d 6f 76 65 64 3c 2f 48 31 3e 1>302 Mo ved</H1>

Frame (572 bytes) Decrypted SSL data (189 bytes) Decompressed Header (336 bytes) Decrypted SSL data (271 bytes)

Server Push (Promise)

After the server responds with an HTML, it waits for requests to embedded resources. Server code knows which resources client needs such as JavaScript, CSS, Images, and HTML pages of future navigation.

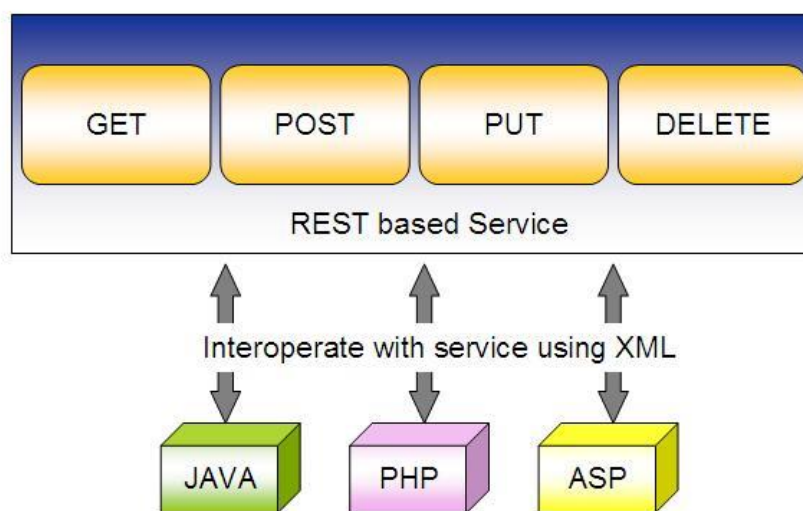
In ASP.NET, use `HttpResponse.PushPromise`

```
String path = Request.ApplicationPath;
Response.PushPromise(path + "/Images/1.png");
Response.PushPromise(path + "/Images/2.png");
```

24. REST (Representational State Transfer)

REST stands for **Representational State Transfer**. (It is sometimes spelled “ReST.”) It relies on a stateless, client-server, cacheable communications protocol, and in virtually all cases, the HTTP protocol is used. REST is *an architecture style* for designing networked applications. The idea is that, rather than using complex mechanisms such as CORBA, RPC or SOAP to connect between machines, simple HTTP is used to make calls between machines. In many ways, the World Wide Web itself, based on HTTP, viewed as a REST-based architecture.

RESTful applications use HTTP requests to post data (create and/or update), read data (e.g., make queries), and delete data. Thus, REST uses HTTP for all four CRUD (Create/Read/Update/Delete) operations.



25. REST HTTP Verbs

HTTP Verb	CRUD	Example	Response
POST	Create	POST http://www.example.com/customers POST http://www.example.com/customers/12345/orders .	201 (Created), 'Location' header with link to /customers/{id} containing new ID.
GET	Read	GET http://www.example.com/customers/12345 GET http://www.example.com/customers/12345/orders	200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists.
PUT	Update/Replace	PUT http://www.example.com/customers/12345 PUT http://www.example.com/customers/12345/orders/98765	405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection.
PATCH	Update/Modify	PATCH http://www.example.com/customers/12345 PATCH http://www.example.com/customers/12345/orders/98765	405 (Method Not Allowed), unless you want to modify the collection itself.
DELETE	Delete	DELETE http://www.example.com/customers/12345 DELETE http://www.example.com/customers/12345/orders	405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable.

25.1 GET

REQUEST	RESPONSE (JSON)
<pre>\$ curl -H "Accept:application/json" http://localhost:8888/demo-rest-jersey- spring/podcasts/1</pre>	<pre>{ "id": 1, "title": "- The Naked Scientists Podcast - Stripping Down Science", "linkOnPodcastpedia": "http://www.podcastpedia.org/podcasts/792/-The- Naked-Scientists-Podcast-Stripping-Down-Science", "feed": "feed_placeholder", "description": "The Scientists flagship science show brings you a lighthearted look at the latest scientific breakthroughs, interviews with the world top scientists, answers to your science questions and science experiments to try at home.", "insertionDate": "2014-10-29T10:46:02.00+0100" }</pre>

25.2 POST

REQUEST	RESPONSE
<pre>curl -i -X POST -H "Content-Type:application/json" http://localhost:8888/demo-rest-jersey- spring/podcasts/ -d '{"title":"- The Scientists Podcast - Stripping Down Science","Podcastpedia":"http://www.podcastpedia.o rg/podcasts/792/-The-Scientists-Podcast-Stripping- Down- Science","feed":"feed_placeholder","description":" The Scientists flagship science show brings you a lighthearted look at the latest scientific breakthroughs, interviews with the world top scientists, answers to your science questions and science experiments to try at home."}'</pre>	<pre>HTTP/1.1 201 Created Location: http://localhost:8888/demo-rest-jersey- spring/podcasts/2 Content-Type: text/html Access-Control-Allow-Origin: * Access-Control-Allow-Methods: GET, POST, DELETE, PUT Access-Control-Allow-Headers: X-Requested-With, Content-Type, X-Codingpedia Vary: Accept-Encoding Content-Length: 60 Server: Jetty(9.0.7.v20131107)</pre>

25.3 PUT

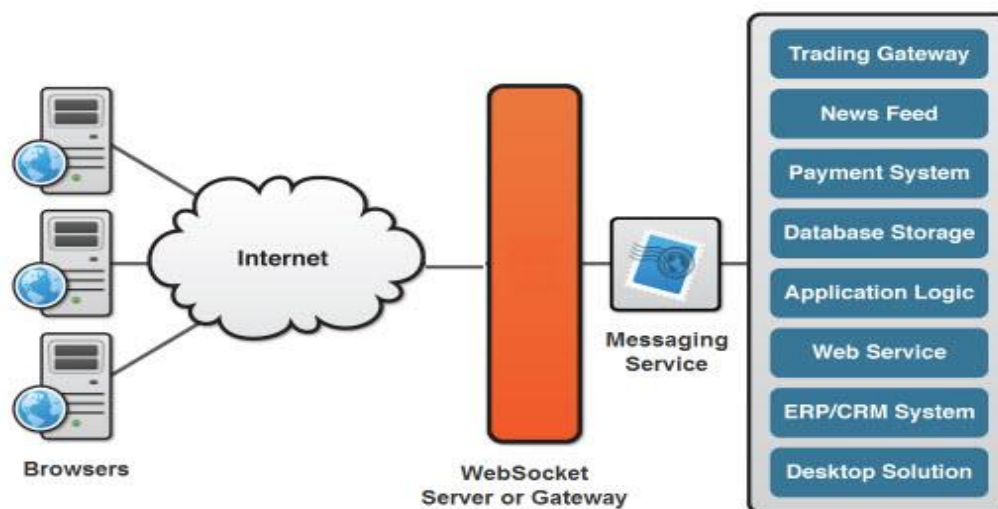
REQUEST	RESPONSE
<pre>curl -i -X PUT -H "Content-Type:application/json" http://localhost:8888/demo-rest-jersey- spring/podcasts/2 -d '{"id":2,"title":"Quarks & Co - zum Mitnehmen","linkOnPodcastpedia":"http://www.podcas tpedia.org/quarks","feed":"http://podcast.wdr.de/q uarks.xml","description":"Quarks & Co: Das Wissenschaftsmagazin"}'</pre>	<pre>HTTP/1.1 201 Created Location: http://localhost:8888/demo-rest-jersey- spring/podcasts/2 Content-Type: text/html Access-Control-Allow-Origin: * Access-Control-Allow-Methods: GET, POST, DELETE, PUT Access-Control-Allow-Headers: X-Requested-With, Content-Type, X-Codingpedia Vary: Accept-Encoding Content-Length: 60 Server: Jetty(9.0.7.v20131107)</pre>

25.4 DELETE

REQUEST	RESPONSE
<pre>curl -i -X DELETE http://localhost:8888/demo-rest-jersey-spring/podcasts/</pre>	<pre>HTTP/1.1 204 No Content Date: Tue, 25 Nov 2014 14:10:17 GMT Server: Jetty(9.0.7.v20131107) Content-Type: text/html Access-Control-Allow-Origin: * Access-Control-Allow-Methods: GET, POST, DELETE, PUT Access-Control-Allow-Headers: X-Requested-With, Content-Type, X-Codingpedia Vary: Accept-Encoding Via: 1.1 vldn680:8888 Content-Length: 0</pre>

26. WebSocket

WebSocket is a computer communications protocol, providing full-duplex communication channels over a single TCP connection. The WebSocket protocol was standardized by the IETF as RFC 6455 in 2011, and the WebSocket API in Web IDL is being standardized by the W3C. WebSocket is a different TCP protocol from HTTP.



27. Protocol handshake

To establish a WebSocket connection, the client sends a WebSocket handshake request, for which the server returns a WebSocket handshake response, as shown in the example below.

Client request (just like in HTTP, each line ends with `\r\n` and there must be an extra blank line at the end)

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
Origin: http://example.com
```

Server response

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sM1YUkAGmm50PpG2HaGWK=
Sec-WebSocket-Protocol: chat
```

28. HTTP Long Polling

Web app developers can implement a technique called HTTP long polling, where the client polls the server requesting new information. The server holds the request open until new data is available. Once available, the server responds and sends the new information. When the client receives the new information, it immediately sends another request, and the operation is repeated. This effectively emulates a server push feature.



29. Web DAV

Web Distributed Authoring and Versioning (WebDAV) is an extension of the Hypertext Transfer Protocol (HTTP) that allows clients to perform remote Web content authoring operations.

HTTP/1.1 still essentially a read-only protocol, as deployed

- Web Distributed Authoring and Versioning – HTTP extension
- The most recent version from 1999 – RFC2518

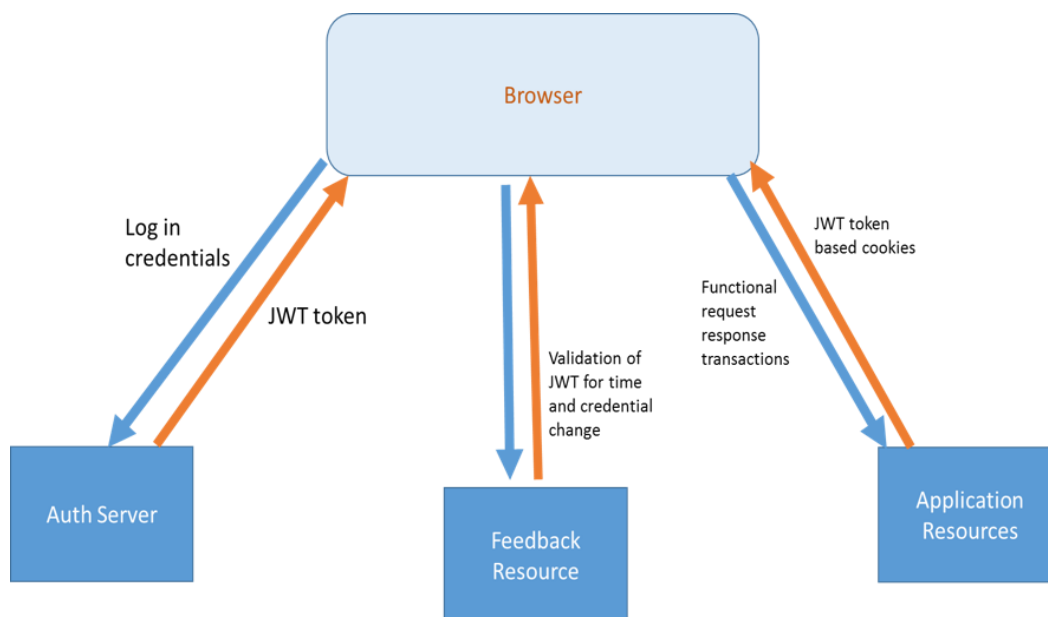
Extra methods:

- PROPFIND – retrieve resource metadata
- PROPPATCH – change/delete resource metadata
- MKCOL – create a collection (directory)
- COPY – copy resource
- MOVE – move the resource
- LOCK/UNLOCK – lock/release resource (so that others cannot change it)

30. JWT (JSON Web Token) – SSO

JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the **HMAC** algorithm) or a public/private key pair using **RSA**.

JSON Web Tokens consist of three parts separated by dots (.), which includes Header, Payload, and Signature. Therefore, a JWT typically looks like the following. **xxxxx.yyyyyy.zzzzz**



31. JWT (JSON Web Token)

➤ Header

The header *typically* consists of two parts: the type of the token, which is **JWT**, and the hashing algorithm being used, such as **HMAC SHA256** or **RSA**.

For example:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

➤ Payload

The second part of the token is the payload, which contains the claims. Claims are statements about an entity (typically, the user) and additional metadata. There are three types of claims: registered, public, and private claims.

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

➤ Signature

To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that. The signature is used to verify that the sender of the JWT and to ensure that the message was not changed along the way.

```

HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)

```

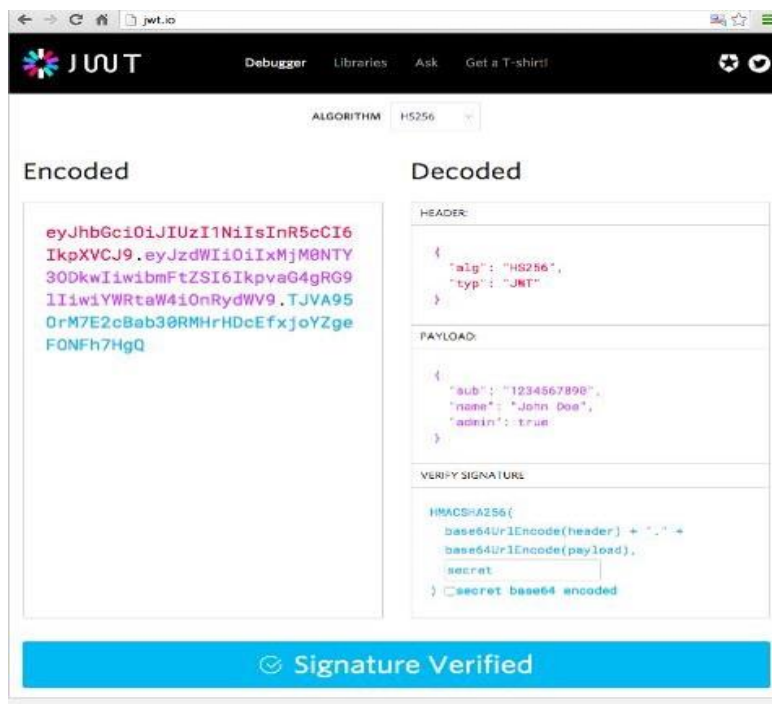
➤ Final Output

By putting all together, the final output shows as follows:

```

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaXNTb2NpYWwiOiJhbnRydWV9.TjVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

```



JWT

Debugger Libraries Ask Get a T-shirt!

ALGORITHM HS256

Encoded

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaXNTb2NpYWwiOiJhbnRydWV9.TjVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

Decoded

HEADER:

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
☐ secret base64 encoded
```

Signature Verified