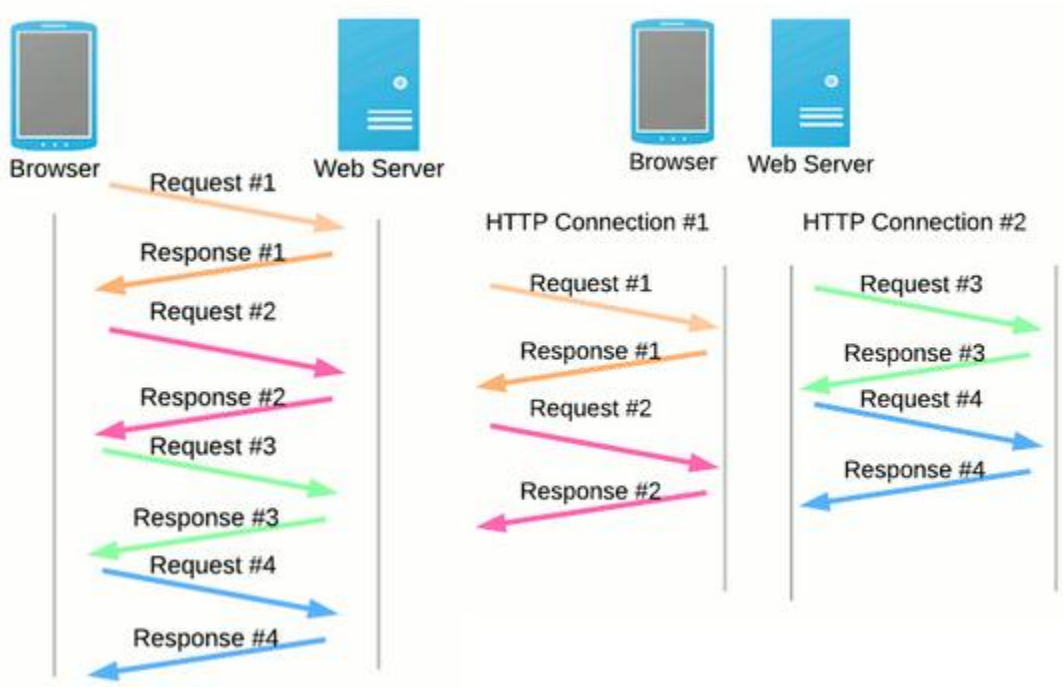
Task - 01

Armaan.K

1. Difference between HTTP1.1 vs HTTP2

HTTP2, the new Web protocol slated to go live any day now, aims to be a faster, more efficient protocol. HTTP1.1 is the current predecessor and has been around for about 15 years. The problem with HTTP1.1 is that can only load requests one at a time, one request per one TCP connection. Basically, this made browsers run parallel requests to multiple TCPs for the same Web asset. This clogs up “the wire” with multiple duplicate data requests, and can hurt performance if too many requests are made.



In short, HTTP1.1 makes loading a web page more resource-intensive than ever. This led to the industry utilizing Best Practices like concatenation, data inlining, and domain sharding in an attempt to fix the underlying problems of the protocol.

HTTP & SPDY: The Basis Of HTTP2

Enter SPDY in 2012. SPDY was the next open-source protocol that was developed, this time by Google, in an attempt to reduce web page load latency and increase security. SPDY modified the way HTTP requests and responses are sent over the wire and made an effective base for HTTP2. Just recently Google announced it will be removing SPDY in favor of HTTP2 and that we can expect HTTP2 to come to browsers in a few weeks.

These are the high-level differences between HTTP1 and HTTP2:

- HTTP2 is binary, instead of textual
- HTTP2 is fully multiplexed, instead of ordered and blocking
- HTTP2 can, therefore, use one connection for parallelism
- HTP2 uses header compression to reduce overhead
- HTTP2 allows servers to “push” responses proactively into client caches

To sum up: HTTP2 is already available. It’s recommended to use HTTP2 as support for a faster website and better user experience, but the old standby recommendations of losslessly optimizing images, etc. still hold true.

2. http version history

Evolution of HTTP :

HTTP (HyperText Transfer Protocol) is the underlying protocol of the World Wide Web. Developed by Tim Berners-Lee and his team between 1989-1991, HTTP has seen many changes, keeping most of the simplicity and further shaping its flexibility. HTTP has evolved from an early protocol to exchange files in a semi-trusted laboratory environment, to the modern maze of the Internet, now carrying images, videos in high resolution and 3D.

HTTP/0.9 – The one-line protocol

The initial version of HTTP had no version number; it has been later called 0.9 to differentiate it from the later versions. HTTP/0.9 is extremely simple: requests consist of a single line and start with the only possible method `GET` followed by the path to the resource (not the URL as both the protocol, server, and port are unnecessary once connected to the server).

Unlike subsequent evolutions, there were no HTTP headers, meaning that only HTML files could be transmitted, but no other type of documents. There were no status or error codes: in case of a problem, a specific HTML file was send back with the description of the problem contained in it, for human consumption.

HTTP/1.0 – Building extensibility

HTTP/0.9 was very limited and both browsers and servers quickly extended it to be more versatile:

- Versioning information is now sent within each request (`HTTP/1.0` is appended to the `GET` line)
- A status code line is also sent at the beginning of the response, allowing the browser itself to understand the success or failure of the request and to adapt its behavior in consequence (like in updating or using its local cache in a specific way)
- The notion of HTTP headers has been introduced, both for the requests and the responses, allowing metadata to be transmitted and making the protocol extremely flexible and extensible.
- With the help of the new HTTP headers, the ability to transmit other documents than plain HTML files has been added (thanks to the `Content-Type` header).

At this point, a typical request and response looked like this:

```
GET /mypage.html HTTP/1.0
User-Agent: NCSA_Mosaic/2.0 (Windows 3.1)

200 OK
Date: Tue, 15 Nov 1994 08:12:31 GMT
Server: CERN/3.0 libwww/2.17
Content-Type: text/html
```

```
<HTML>
A page with an image
  <IMG SRC="/myimage.gif">

</HTML>
```

Followed by a second connection and request to fetch the image (followed by a response to that request):

```
GET /myimage.gif HTTP/1.0
User-Agent: NCSA_Mosaic/2.0 (Windows 3.1)

200 OK
Date: Tue, 15 Nov 1994 08:12:32 GMT
Server: CERN/3.0 libwww/2.17
Content-Type: text/gif

(image content)
```

These novelties have not been introduced as concerted effort, but as a try-and-see approach over the 1991-1995 period: a server and a browser added one feature and it saw if it got traction. A lot of interoperability problems were common. In November 1996, in order to solve these annoyances, an informational document describing the common practices has been published, RFC 1945. This is the definition of HTTP/1.0 and it is notable that, in the narrow sense of the term, it isn't an official standard.

HTTP/1.1 – The standardized protocol

In parallel to the somewhat chaotic use of the diverse implementations of HTTP/1.0, and since 1995, well before the publication of HTTP/1.0 document the next year, proper standardization was in progress. The first standardized version of HTTP, HTTP/1.1 was published in early 1997, only a few months after HTTP/1.0.

HTTP/1.1 clarified ambiguities and introduced numerous improvements:

- A connection can be reused, saving the time to reopen it numerous times to display the resources embedded into the single original document retrieved.

- Pipelining has been added, allowing to send a second request before the answer for the first one is fully transmitted, lowering the latency of the communication.
- Chunked responses are now also supported.
- Additional cache control mechanisms have been introduced.
- Content negotiation, including language, encoding, or type, has been introduced, and allows a client and a server to agree on the most adequate content to exchange.
- Thanks to the `Host` header, the ability to host different domains at the same IP address now allows server colocation.

A typical flow of requests, all through one single connection is now looking like this:

```
GET /en-US/docs/Glossary/Simple_header HTTP/1.1
Host: developer.mozilla.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0) Gecko/20100101 Firefox/50.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://developer.mozilla.org/en-US/docs/Glossary/Simple_header

200 OK
Connection: Keep-Alive
Content-Encoding: gzip
Content-Type: text/html; charset=utf-8
Date: Wed, 20 Jul 2016 10:55:30 GMT
Etag: "547fa7e369ef56031dd3bff2ace9fc0832eb251a"
Keep-Alive: timeout=5, max=1000
Last-Modified: Tue, 19 Jul 2016 00:59:33 GMT
Server: Apache
Transfer-Encoding: chunked
Vary: Cookie, Accept-Encoding

(content)

GET /static/img/header-background.png HTTP/1.1
Host: developer.mozilla.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:50.0) Gecko/20100101 Firefox/50.0
Accept: */*
```

```
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: https://developer.mozilla.org/en-US/docs/Glossary/Simple_header

200 OK
Age: 9578461
Cache-Control: public, max-age=315360000
Connection: keep-alive
Content-Length: 3077
Content-Type: image/png
Date: Thu, 31 Mar 2016 13:34:46 GMT
Last-Modified: Wed, 21 Oct 2015 18:27:50 GMT
Server: Apache

(image content of 3077 bytes)
```

HTTP/1.1 was first published as RFC 2068 in January 1997.

More than 15 years of extensions

Thanks to its extensibility – creating new headers or methods is easy – and even if the HTTP/1.1 protocol was refined over two revisions, RFC 2616 published in June 1999 and the series of RFC 7230-RFC 7235 published in June 2014 in prevision of the release of HTTP/2, this protocol has been extremely stable over more than 15 years.

Using HTTP for secure transmissions

The largest change that happened to HTTP was done as early as end of 1994. Instead of sending HTTP over a basic TCP/IP stack, Netscape Communications created an additional encrypted transmission layer on top of it: SSL. SSL 1.0 was never released outside the company, but SSL 2.0 and its successor SSL 3.0 allowed for the creation of e-commerce Web sites by encrypting and guaranteeing the authenticity of the messages exchanged between the server and client. SSL was put on the standards track and eventually became TLS, with versions 1.0, 1.1, 1.2, and 1.3 appearing successfully to close vulnerabilities.

During the same time, the need for an encrypted transport layer raised: the Web left the relative trustiness of a mostly academic network, to a jungle where advertisers, random

individuals or criminals compete to get as much private information about people, try to impersonate them or even to replace data transmitted by altered ones. As the applications built over HTTP became more and more powerful, having access to more and more private information like address books, e-mail, or the geographic position of the user, the need to have TLS became ubiquitous even outside the e-commerce use case.

Using HTTP for complex applications

The original vision of Tim Berners-Lee for the Web wasn't a read-only medium. He envisioned a Web where people can add and move documents remotely, a kind of distributed file system. Around 1996, HTTP has been extended to allow authoring, and a standard called WebDAV was created. It has been further extended for specific applications like CardDAV to handle address book entries and CalDAV to deal with calendars. But all these *DAV extensions had a flaw: they had to be implemented by the servers to be used, which was quite complex. Their use on Web realms stayed confidential.

In 2000, a new pattern for using HTTP was designed: representational state transfer (or REST). The actions induced by the API were no more conveyed by new HTTP methods, but only by accessing specific URIs with basic HTTP/1.1 methods. This allowed any Web application to provide an API to allow retrieval and modification of its data without having to update the browsers or the servers: all what is needed was embedded in the files served by the Web sites through standard HTTP/1.1. The drawback of the REST model resides in the fact that each website defines its own non-standard RESTful API and has total control on it; unlike the *DAV extensions where clients and servers are interoperable. RESTful APIs became very common in the 2010s.

Since 2005, the set of APIs available to Web pages greatly increased and several of these APIs created extensions, mostly new specific HTTP headers, to the HTTP protocol for specific purposes:

- Server-sent events, where the server can push occasional messages to the browser.
- WebSocket, a new protocol that can be set up by upgrading an existing HTTP connection.

Relaxing the security-model of the Web

HTTP is independent of the security model of the Web, the same-origin policy. In fact, the current Web security model has been developed after the creation of HTTP! Over the years, it has proved useful to be able to be more lenient, by allowing under certain constraints to lift some of the restriction of this policy. How much and when such restrictions are lifted is transmitted by the server to the client using a new bunch of HTTP headers. These are defined in specifications like Cross-Origin Resource Sharing (CORS) or the Content Security Policy (CSP).

In addition to these large extensions, numerous other headers have been added, sometimes experimentally only. Notable headers are Do Not Track (`DNT`) header to control privacy, `X-Frame-Options`, or `Upgrade-Insecure-Requests` but many more exist.

HTTP/2 – A protocol for greater performance

Over the years, Web pages have become much more complex, even becoming applications in their own right. The amount of visual media displayed, the volume and size of scripts adding interactivity, has also increased: much more data is transmitted over significantly more HTTP requests. HTTP/1.1 connections need requests sent in the correct order. Theoretically, several parallel connections could be used (typically between 5 and 8), bringing considerable overhead and complexity. For example, HTTP pipelining has emerged as a resource burden in Web development.

In the first half of the 2010s, Google demonstrated an alternative way of exchanging data between client and server, by implementing an experimental protocol SPDY. This amassed interest from developers working on both browsers and servers. Defining an increase in responsiveness, and solving the problem of duplication of data transmitted, SPDY served as the foundations of the HTTP/2 protocol.

The HTTP/2 protocol has several prime differences from the HTTP/1.1 version:

- It is a binary protocol rather than text. It can no longer be read and created manually. Despite this hurdle, improved optimization techniques can now be implemented.
- It is a multiplexed protocol. Parallel requests can be handled over the same connection, removing the order and blocking constraints of the HTTP/1.x protocol.

- It compresses headers. As these are often similar among a set of requests, this removes duplication and overhead of data transmitted.
- It allows a server to populate data in a client cache, in advance of it being required, through a mechanism called the server push.

Officially standardized, in May 2015, HTTP/2 has had much success. By July 2016, 8.7% of all Web sites[1] were already using it, representing more than 68% of all requests[2]. High-traffic Web sites showed the most rapid adoption, saving considerably on data transfer overheads and subsequent budgets.

This rapid adoption rate was likely as HTTP/2 does not require adaptation of Web sites and applications: using HTTP/1.1 or HTTP/2 is transparent for them. Having an up-to-date server communicating with a recent browser is enough to enable its use: only a limited set of groups were needed to trigger adoption, and as legacy browser and server versions are renewed, usage has naturally increased, without further Web developer efforts.

Post-HTTP/2 evolution

HTTP didn't stop evolving upon the release of HTTP/2. Like with HTTP/1.x previously, HTTP's extensibility is still being used to add new features. Notably, we can cite new extensions of the HTTP protocol appearing in 2016:

- Support of `Alt-Svc` allows the dissociation of the identification and the location of a given resource, allowing for a smarter CDN caching mechanism.
- The introduction of `Client-Hints` allows the browser, or client, to proactively communicate information about its requirements, or hardware constraints, to the server.
- The introduction of security-related prefixes in the `Cookie` header, now helps guarantee a secure cookie has not been altered.

This evolution of HTTP proves its extensibility and simplicity, liberating creation of many applications and compelling the adoption of the protocol. The environment in which HTTP is used today is quite different from that seen in the early 1990s. HTTP's original design proved to be a masterpiece, allowing the Web to evolve over a quarter of a century, without the need of a mutiny. By healing flaws, yet retaining the flexibility and extensibility which made HTTP such a success, the adoption of HTTP/2 hints at a bright future for the protocol.

HTTP/3 - HTTP over QUIC

Draft

This page is not complete.

Experimental

This is an experimental technology

Check the Browser compatibility table carefully before using this in production.

The next major version of HTTP, HTTP/3, will use QUIC instead TCP/TLS for the transport layer portion.

3. List 5 difference between Browser JS(console) vs Nodejs

1. In the browser, most of the time what you are doing is interacting with the DOM, or other Web Platform APIs like Cookies. Those do not exist in Node.js, of course. You don't have the `document`, `window` and all the other objects that are provided by the browser.
2. And in the browser, we don't have all the nice APIs that Node.js provides through its modules, like the filesystem access functionality.
3. Another big difference is that in Node.js you control the environment. Unless you are building an open source application that anyone can deploy anywhere, you know which version of Node.js you will run the application on. Compared to the browser environment, where you don't get the luxury to choose what browser your visitors will use, this is very convenient.This means that you can write all the modern ES6-7-8-9 JavaScript that your Node.js version supports.
4. Since JavaScript moves so fast, but browsers can be a bit slow to upgrade, sometimes on the web you are stuck with using older JavaScript / ECMAScript releases.You can use Babel to transform your code to be ES5-compatible before shipping it to the browser, but in Node.js, you won't need that.

5. Another difference is that Node.js uses the CommonJS module system, while in the browser we are starting to see the ES Modules standard being implemented.In practice, this means that for the time being you use `require()` in Node.js and `import` in the browser.

4. what happens when you type a URL in the address bar in the browser?

“What happens when you type in a URL” is a deceptive question commonly asked in tech interviews. If you look online, there are many very detailed resources but few concise explanations of how a web browser, a server, and the general internet work together.

This is how I would explain it:

1. You enter a URL into a web browser
2. The browser looks up the IP address for the domain name via DNS
3. The browser sends a HTTP *request* to the server
4. The server sends back a HTTP *response*
5. The browser begins rendering the HTML
6. The browser sends requests for additional objects embedded in HTML (images, css, JavaScript) and repeats steps 3-5.
7. Once the page is loaded, the browser sends further async requests as needed.

That’s really it. Here’s a description in words for this site.

When you type “https://wsvincent.com” into your browser the first thing that happens is a Domain Name Server (DNS) matches “wsvincent.com” to an IP address. Then the browser sends an HTTP request to the server and the server sends back an HTTP response. The browser begins rendering the HTML on the page while also requesting any additional resources such as CSS, JavaScript, images,

etc. Each subsequent request completes a request/response cycle and is rendered in turn by the browser. Then once the page is loaded some sites (though not mine) will make further asynchronous requests.

If I were asked to explain further I might start talking about *how* the server and browser connect via TCP. And we could discuss encryption via https, too.

But fundamentally, it's only 7 steps. I hope this clears up any confusion for readers.