**DAY-1 TASK**

**Difference between HTTP/1.1 and HTTP/2.0**

1. One of the most significant features that distinguishes HTTP/1.1 and HTTP/2 is the binary framing layer, which can be thought of as a part of the application layer in the internet protocol stack. As opposed to HTTP/1.1, which keeps all requests and responses in plain text format, HTTP/2 uses the binary framing layer to encapsulate all messages in binary format, while still maintaining HTTP semantics, such as verbs, methods, and headers.
2. HTTP/1.1 allows persistent connections and pipelining. With persistent connections, HTTP/1.1 assumes that a TCP connection should be kept open unless directly told to close. This allows the client to send multiple requests along the same connection without waiting for a response to each. Since multiple data packets cannot pass each other when traveling to the same destination, there are situations in which a request at the head of the queue that cannot retrieve its required resource will block all the requests behind it. This is known as head-of-line (HOL) blocking, and is a significant problem with optimizing connection efficiency in HTTP/1.1. Adding separate, parallel TCP connections could alleviate this issue, but there are limits to the number of concurrent TCP connections possible between a client and server, and each new connection requires significant resources.

HTTP/2 tackles this issue by introducing the binary framing layer which encodes requests/responses and cuts them up into smaller packets of information, greatly increasing the flexibility of data transfer. As opposed to HTTP/1.1, which must make use of multiple TCP connections to lessen the effect of HOL blocking, HTTP/2 establishes a single connection object between the two machines. Within this connection there are multiple streams of data. Each stream consists of multiple messages in the familiar request/response format. Finally, each of these messages split into smaller units called frames.

At the most granular level, the communication channel consists of a bunch of binary-encoded frames, each tagged to a particular stream. The identifying tags allow the connection to interleave these frames during transfer and reassemble them at the other end. The interleaved requests and responses can run in parallel without blocking the messages behind them, a process called multiplexing. Multiplexing resolves the head-of-line blocking issue in HTTP/1.1 by ensuring that no message has to wait for another to finish. This also means that servers and clients can send concurrent requests and responses, allowing for greater control and more efficient connection management.

1. We can set the weights in our requests based on your needs in HTTP/2 which is not available in HTTP/1.1. For example, we may assign a lower priority for loading an image with high resolution after providing a thumbnail image on the web page. By providing this facility of weight assignment, HTTP/2 enables developers to gain better control over web page rendering. The protocol also allows the client to change dependencies and reallocate weights at runtime in response to user interaction. It is important to note, however, that a server may change assigned priorities on its own if a certain stream is blocked from accessing a specific resource.
2. HTTP/1.1 relies on the transport layer to avoid buffer overflow, each new TCP connection requires a separate flow control mechanism. HTTP/2, however, multiplexes streams within a single TCP connection. HTTP/2 allows the client and server to implement their own flow controls, rather than relying on the transport layer. The application layer communicates the available buffer space, allowing the client and server to set the receive window on the level of the multiplexed streams.
3. In HTTP/1.1, if the developer knows in advance which additional resources the client machine will need to render the page, they can use a technique called resource inlining to include the required resource directly within the HTML document that the server sends in response to the initial GET request. A major drawback of resource inlining is that the client cannot separate the resource and the document. A finer level of control is needed to optimize the connection, a need that HTTP/2 seeks to meet with server push.

Since HTTP/2 enables multiple concurrent responses to a client’s initial GET request, a server can send a resource to a client along with the requested HTML page, providing the resource before the client asks for it. This process is called server push. In this way, an HTTP/2 connection can accomplish the same goal of resource inlining while maintaining the separation between the pushed resource and the document. This means that the client can decide to cache or decline the pushed resource separate from the main HTML document, fixing the major drawback of resource inlining.

1. HTTP/1.1 uses programs like gzip to compress the data sent in HTTP messages, especially to decrease the size of CSS and JavaScript files. The header component of a message, however, is always sent as plain text. Although each header is quite small, the burden of this uncompressed data weighs heavier and heavier on the connection as more requests are made, particularly penalizing complicated, API-heavy web applications that require many different resources and thus many different resource requests. Additionally, the use of cookies can sometimes make headers much larger, increasing the need for compression.

In order to solve this bottleneck, HTTP/2 uses HPACK compression to shrink the size of headers. HTTP/2 can split headers from their data, resulting in a header frame and a data frame. The HTTP/2-specific compression program HPACK can then compress this header frame. This algorithm can encode the header metadata using Huffman coding, thereby greatly decreasing its size. Additionally, HPACK can keep track of previously conveyed metadata fields and further compress them according to a dynamically altered index shared between the client and the server. This feature greatly helps in reducing client-server latency.

**Javascript Objects and it’s internal representation:**

In JavaScript, almost "everything" is an object.

* Booleans can be objects (if defined with the new keyword)
* Numbers can be objects (if defined with the new keyword)
* Strings can be objects (if defined with the new keyword)
* Dates are always objects
* Maths are always objects
* Regular expressions are always objects
* Arrays are always objects
* Functions are always objects
* Objects are always objects

All JavaScript values, except primitives, are objects. JavaScript defines 7 types of primitive data types: string, number, Boolean, null, undefined, symbol & bigint.

Objects are also considered as variables. Objects can contain many values. Object values are written as name : value pairs (name and value separated by a colon). For example:

const person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

A JavaScript object is a collection of named values. The named values, in JavaScript objects, are called properties. In the above example “firstName” is the property name and “John” is the value.

There are different ways to create new objects:

* Create a single object, using an object literal.
* Create a single object, with the keyword new.
* Define an object constructor, and then create objects of the constructed type.
* Create an object using Object.create().

Objects are mutable: They are addressed by reference, not by value.