Day 01 Assignment

1.Write a blog on Difference between HTTP1.1 vs HTTP2

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 could be reused, which saved time. It no longer needed to be opened multiple times to display the resources embedded in the single original document.
* Pipelining was added. This allowed a second request to be sent before the answer to the first one was fully transmitted. This lowered the latency of the communication.
* Chunked responses were also supported.
* Additional cache control mechanisms were introduced.
* Content negotiation, including language, encoding, and type, was introduced. A client and a server could now agree on which content to exchange.
* Added host header, the ability to host different domains from the same IP address allowed server collocation.

Over the years, web pages became more complex. Some of them were even applications in their own right. More visual media was displayed and the volume and size of scripts adding interactivity also increased. Much more data was transmitted over significantly more HTTP requests and this created more complexity and overhead for HTTP/1.1 connections. To account for this, Google implemented an experimental protocol SPDY in the early 2010s. This alternative way of exchanging data between client and server amassed interest from developers working on both browsers and servers. SPDY defined an increase in responsiveness and solved the problem of duplicate data transmission, serving as the foundation for the HTTP/2 protocol.

The HTTP/2 protocol differs from HTTP/1.1 in a few ways:

* It's a binary protocol rather than a text protocol. It can't be read and created manually. Despite this hurdle, it allows for the implementation of improved optimization techniques.
* It's a multiplexed protocol. Parallel requests can be made over the same connection, removing the constraints of the HTTP/1.x protocol.
* It compresses headers. As these are often similar among a set of requests, this removes the duplication and overhead of data transmitted.
* It allows a server to populate data in a client cache through a mechanism called the server push.

2. Write a blog about objects and its internal representation in Javascript

### Understanding Objects in JavaScript

In JavaScript, objects are collections of key-value pairs, where keys are strings (or symbols) and values can be any data type, including other objects, functions, arrays, or primitive values like strings, numbers, and booleans. This flexible structure allows for complex data modeling and manipulation.

// Example of a simple JavaScript object

const person = {

name: "John Doe",

age: 30,

profession: "Developer"

};

### Internal Representation of Objects

Internally, JavaScript engines like V8 (used in Chrome and Node.js) store objects in memory using various data structures and optimization techniques. While the exact implementation details may vary between engines, here's a high-level overview of how objects are typically represented:

1. **Hash Tables**: JavaScript engines often use hash tables (or similar data structures) to store object properties efficiently. Each property key is hashed to determine its storage location in memory, providing fast access time (usually O(1)).
2. **Hidden Classes (or Shapes)**: To optimize property access and memory usage, JavaScript engines may use hidden classes or shapes to represent the structure of objects. When an object is created, the engine assigns it a hidden class based on its properties and their order of declaration. Subsequent objects with the same structure share the same hidden class, allowing for faster property access.
3. **Property Descriptors**: Each property in an object is associated with a property descriptor, which contains metadata such as property attributes (e.g., configurable, enumerable, writable) and the value of the property. JavaScript engines use these descriptors to manage property behavior and access control.
4. **Prototype Chain**: Objects in JavaScript are linked together through a prototype chain, allowing them to inherit properties and methods from their prototype objects. This chain enables prototypal inheritance, a fundamental feature of the language.

Since Stack can only store fixed-size or static data, where does JavaScript store dynamic data? The answer is Heap. The heap memory, unlike stack memory, doesn’t have a fixed-size limitation, i.e., the memory gets allocated dynamically.

JavaScript uses a heap for storing variables whose size is unknown at compile time or may vary at the run time, such as objects, arrays, functions, etc.

The JavaScript Engine dynamically allocates memory to the heap. Initially, the heap size depends on available system memory, and it can be dynamically increased/decreased based on the need. You can check the current memory usage of your Node.js application using the process.memoryUsage() method.

JavaScript allocates memory for objects within the heap, but we must have a reference to that memory location to access the value. The reference to the memory location resides in the stack memory.

Since the variable name contains a reference to the heap object, whenever we refer to the variable in our code, it’s the reference and not the actual object. Let’s take an example:

let box1 = {

width: 100,

height: 50

};

let box2 = box1;

box2.width = 200;

console.log('Box 1 width: ', box1.width);

console.log('Box 2 width: ', box2.width);

## Garbage collection

In languages like C, C++, etc., it’s our job to allocate memory and safely deallocate it after use. Therefore, if the memory is not freed up after use, the program will run with an unused allocated memory block. We term this condition a memory leak. Therefore, to prevent such conditions, modern programming languages, like JavaScript, come with a Garbage collector.

A Garbage collector is a part of the JavaScript Engine whose job is to free up unused memory from the heap using a garbage collection process. It ensures proper memory management while the application is running. Therefore, the programmer doesn’t have to worry about manual memory management, the garbage collector does it.

It seems pretty straightforward, right? Well, No. There are a lot of challenges to achieving this. There isn’t any algorithm that guarantees that it frees up all the unused memory at the current moment. But, there are algorithms such as the mark and sweep, that perform pretty well and come close to removing most of the unwanted pieces of memory from the heap.

### Mark and sweep

It’s one of the most common garbage collection algorithms out there. Basically, it marks the objects not reachable by the root object, i.e., the window object as trash or garbage. Later, the garbage collector removes these marked objects from the heap. Have a look at the example below.

In the above example, the employee object is reachable from the window object. Now, because the company and person objects are reachable through the employee object, they are ultimately reachable by the window object. But, the car object and other objects connected to it are not accessible through the window object. Therefore, the algorithm marks these objects and later free up the memory used by them.

One tradeoff though is the global variables declared using var. When declaring a variable with var, the JavaScript Engine attaches them to the window object. Now, since these variables are always reachable by window objects, there are never removed by the garbage collector.

