1. HTTP1.1 VS HTTP2

The Hypertext Transfer Protocol (HTTP) is the backbone of data communication on the World Wide Web. Over time, as the demands of web applications and the internet landscape evolved, so did the protocol itself. This evolution led to the introduction of HTTP/2, an enhanced version of its predecessor, HTTP/1.1. In this blog post, we will delve into the key differences between HTTP/1.1 and HTTP/2, exploring how each version impacts web performance, efficiency, and user experience.

HTTP/1.1 Overview:

HTTP/1.1, the predecessor to HTTP/2, has been the dominant protocol for over a decade. While it has played a crucial role in the growth of the internet, it does have its limitations. One notable limitation is the way it handles multiple requests.

Connection Handling: Serial Processing: HTTP/1.1 relies on a single, persistent connection to process one request at a time. Multiple requests must wait in line, leading to potential delays in loading web pages.

Header Overhead: Redundant Headers: Each request and response in HTTP/1.1 carries a set of headers. Unfortunately, many of these headers are redundant, resulting in increased overhead and slower performance.

Resource Bundling: Inefficient Bundling: To reduce the number of requests, developers often resort to techniques like resource bundling (concatenating multiple files into one). However, this can lead to inefficiencies when only a subset of the bundled resources is needed.

HTTP/2 Advancements:

Enter HTTP/2, designed to address the shortcomings of HTTP/1.1 and provide a more efficient and optimized communication protocol.

Multiplexing: Parallel Processing: HTTP/2 introduces multiplexing, allowing multiple requests and responses to be sent concurrently over a single connection. This significantly reduces latency and speeds up the loading of web pages.

Header Compression: HPACK Algorithm: Unlike HTTP/1.1, HTTP/2 employs header compression using the HPACK algorithm. This minimizes redundant headers, reducing the overhead associated with each request and response.

Binary Protocol: Efficient Data Transfer: HTTP/2 switches from a text-based protocol to a binary protocol. This change makes it more efficient in terms of parsing and reduces errors that can occur due to whitespace or typos.

Server Push: Proactive Resource Push: One of the standout features of HTTP/2 is server push. With this capability, the server can push resources to the client before they are explicitly requested, optimizing page load times.

Prioritization: Stream Prioritization: HTTP/2 introduces stream prioritization, allowing the client to specify the importance of different resources. This ensures that critical resources are fetched first, further enhancing performance.

Conclusion:

In summary, the transition from HTTP/1.1 to HTTP/2 represents a significant leap in the efficiency and performance of web communication. HTTP/2's adoption of multiplexing, header compression, binary protocol, server push, and prioritization addresses the challenges posed by its predecessor, resulting in faster, more responsive web applications. As the internet continues to evolve, HTTP/2 stands as a testament to the importance of adapting protocols to meet the demands of modern web development and user expectations.

2. JavaScript Objects and its Internal Representation:

JavaScript, as a versatile and dynamic programming language, relies heavily on objects for structuring and organizing data. Objects are a fundamental part of the language, offering a flexible and intuitive way to model real-world entities. In this blog post, we'll explore the internal representation of objects in JavaScript, shedding light on how they are stored and accessed under the hood.

Understanding Objects in JavaScript:

In JavaScript, objects are collections of key-value pairs, where each key is a string or symbol, and each value can be of any data type, including other objects. This key-value structure provides a powerful mechanism for organizing and manipulating data.

Object Creation:

Literal Notation: Objects can be created using literal notation, where key-value pairs are defined within curly braces.

```
let person = { name: 'John', age: 30, city: 'Example' };
```

Constructor Function: Objects can also be created using constructor functions, providing a blueprint for creating multiple instances.

```
function Person(name, age, city) {
  this.name = name;
  this.age = age;
  this.city = city;
}
let person = new Person('John', 30, 'Example');
```

Internal Representation of Objects:

Property Storage:

Hash Tables: JavaScript engines use hash tables to store object properties efficiently. When a new property is added to an object, the engine calculates a hash value for the property name, enabling fast lookup and retrieval.

Hidden Classes and Shapes:

Optimizing Property Access: JavaScript engines employ techniques like hidden classes and shapes to optimize property access. As an object's structure evolves, the engine dynamically adjusts its internal representation to enhance performance.

Proto and Prototypal Inheritance:

Prototype Chain: Objects in JavaScript are linked through a prototype chain. The __proto__ property allows an object to inherit properties and methods from its prototype. This mechanism facilitates prototypal inheritance, a distinctive feature of JavaScript.

Garbage Collection:

Memory Management: JavaScript engines utilize garbage collection to reclaim memory occupied by objects that are no longer reachable. This automatic process ensures efficient memory usage and prevents memory leaks.

Primitive Wrapper Objects:

Automatic Wrapping: JavaScript automatically wraps primitive values (like numbers and strings) with their respective object wrapper counterparts when necessary. This behavior allows primitive values to be treated as objects in certain contexts.

Conclusion:

Objects lie at the core of JavaScript's expressive and flexible nature. Understanding their internal representation provides insights into how JavaScript engines optimize property access, manage memory, and implement prototypal inheritance. As you continue your journey in JavaScript development, a deeper appreciation for the intricacies of object representation empowers you to write more efficient and effective code. Whether you're creating complex data structures or leveraging prototypal inheritance for code organization, the versatility of objects in JavaScript remains a cornerstone of modern web development.