Difference Between HTTP1.1 vs HTTP2

|  | HTTP 1.1 | HTTP 2 |
| --- | --- | --- |
| **Multiplexing** | Uses a series of serial connections. Each resource request is made sequentially, potentially causing head-of-line blocking. | Supports multiplexing, allowing multiple requests and responses to be sent and received in parallel over a single connection, eliminating head-of-line blocking. |
| **Header Compression** | Headers are sent as plain text and can be verbose, leading to higher overhead. | Employs header compression (HPACK) to reduce header size and improve efficiency. |
| **Binary Protocol** | Uses a text-based protocol, which requires parsing and is less efficient. | Utilizes a binary protocol for more efficient parsing, reducing errors and improving performance. |
| **Resource Prioritization** | Doesn't provide explicit mechanisms for resource prioritization. | Allows clients to specify the priority of resources, helping the server understand which resources to send first. |
| **Server Push** | Does not support server push. | Supports server push, enabling the server to proactively send resources to the client's cache before they are requested, reducing latency. |
| **Persistent Connection** | Supports persistent connections through the Keep-Alive header, but it's optional. | Persistent connections are the default behavior, eliminating the need for multiple connection setups. |
| **Security** | Does not provide security by default, additional protocols like HTTPS are required for encryption. | Encourages the use of encryption through Transport Layer Security (TLS/SSL) by default, enhancing security. |
| **Backward Compatibility** | Provides full support for HTTP/1.0 and HTTP/0.9, ensuring backward compatibility. | Requires upgrading to HTTP/2 for its benefits, but it maintains compatibility with HTTP/1.1 when necessary. |

Understanding Objects and Their Internal Representation in JavaScript

JavaScript is known for its versatility, allowing developers to work with various data types. Among these, objects stand out as a fundamental and powerful construct. In this blog, we'll dive into the world of objects in JavaScript and explore how they are internally represented.

# Objects: The Building Blocks

In JavaScript, objects are collections of key-value pairs, where keys are strings (or Symbols in ES6+), and values can be of any data type. Objects are used to represent complex data structures, organize code, and even simulate real-world entities. They are the cornerstone of the language, and understanding their internal representation is crucial for effective JavaScript development.

## Internal Representation

JavaScript engines, such as V8 (used in Chrome) or SpiderMonkey (used in Firefox), store objects differently from other data types like numbers or strings. Objects are typically implemented as dictionaries or hash maps. Let's break down the key aspects of an object's internal representation:

**Property Storage:** Objects use a hash table-like structure to store their properties. Each property (key-value pair) of an object is hashed, and the hash is used as an index to store and retrieve the property's value. This hash-based approach allows for efficient property access, even when dealing with a large number of properties.

**Hidden Classes:** JavaScript engines optimize object property access by using hidden classes, also known as shapes or maps. When an object is created, the engine assigns it a hidden class that defines the layout and properties of the object. When you add or modify properties, the engine updates the hidden class as needed. This optimization helps reduce the time required to access object properties.

**Property Attributes:** Each property in JavaScript objects has attributes that determine its behavior. These attributes include whether a property is enumerable (can be looped over), writable (can be modified), and configurable (can be deleted or have its attributes changed). These attributes affect how properties interact with JavaScript code.

**Prototypes:** Objects in JavaScript can have prototypes, which are other objects that they inherit properties and methods from. The prototype chain allows for object inheritance and is a key feature for creating reusable code structures.

## Practical Implications

Understanding the internal representation of objects in JavaScript has practical implications for developers. Here are a few:

**Performance Optimization:** Knowing how JavaScript engines store and access properties in objects can help you write more efficient code. Minimizing property lookups and adhering to hidden class optimizations can lead to faster execution.

**Object Manipulation:** Understanding property attributes can help you control how objects can be manipulated. For example, you can create read-only or non-enumerable properties to protect sensitive data.

**Prototypal Inheritance:** Mastery of prototype chains allows for creating sophisticated inheritance hierarchies, enabling you to build robust and maintainable codebases.

In conclusion, objects are a fundamental concept in JavaScript, and their internal representation is vital for efficient and effective programming. By understanding how objects are stored and accessed, you can leverage their full potential and write more performant and maintainable JavaScript code. Objects, with their flexible key-value pair structure and rich capabilities, empower developers to create dynamic and complex applications in the world of web development.