**Difference between HTTP1.1 vs HTTP2**

HTTP/1.1 and HTTP/2 are both protocols used for communication between web browsers and web servers, but they differ significantly in terms of performance, efficiency, and features. Here are some key differences between HTTP/1.1 and HTTP/2:

|  |  |  |
| --- | --- | --- |
|  | **HTTP/1.1** | **HTTP/2** |
| **Multiplexing** | HTTP/1.1: It uses a single connection to transfer one resource at a time. This means that if multiple resources (such as images, scripts, stylesheets) need to be loaded, separate connections are established for each resource. This can lead to inefficient resource loading, also known as the "head-of-line blocking" problem. | HTTP/2: It supports multiplexing, allowing multiple resources to be sent over a single connection simultaneously. This eliminates the head-of-line blocking issue, making the overall loading of resources more efficient. |
| **Header Compression** | HTTP/1.1: Headers are sent in plain text with each request and response. This can result in a significant overhead when transferring small payloads or multiple resources. | HTTP/2: It uses header compression, which reduces the size of headers before transmission, thereby reducing overhead and improving performance. |
| **Server Push** | HTTP/1.1: The server cannot proactively push resources to the client. The client must explicitly request each resource. | HTTP/2: It supports server push, where the server can push resources to the client's cache without waiting for explicit requests. This can improve page load times by preloading resources that the server predicts will be needed. |
| **Binary Protocol** | HTTP/1.1: It uses plain text for both headers and data, which can lead to higher latency and increased data transfer sizes due to textual representation. | HTTP/2: It uses a binary protocol for more efficient data transmission. This reduces the amount of data that needs to be transmitted and parsed, resulting in better performance. |
| **Flow Control** | HTTP/1.1: It lacks flow control mechanisms at the protocol level. This means that if a client receives data too quickly, it might overwhelm the client's resources. | HTTP/2: It includes built-in flow control mechanisms, allowing both the client and server to control the pace of data transfer to match their respective processing speeds. |
| **Connection Reuse** | HTTP/1.1: Each resource transfer typically requires opening a new connection, incurring connection setup overhead. | HTTP/2: It promotes connection reuse, allowing multiple requests and responses to share a single connection. This minimizes connection setup overhead |

In summary, HTTP/2 is designed to address the performance limitations of HTTP/1.1 by introducing features like multiplexing, header compression, server push, binary framing, and flow control. These improvements collectively lead to faster and more efficient web page loading, reduced latency, and better utilization of network resources. As a result, many modern websites and applications have transitioned to using HTTP/2 to provide a better user experience.

Top of Form

**Exploring Objects and Their Internal Representation in JavaScript**

**Introduction**

JavaScript, the versatile and widely-used programming language, owes much of its power to its ability to work with objects. Objects are at the heart of JavaScript, allowing developers to organize and manipulate data in a structured and efficient manner. In this blog post, we'll delve into the concept of objects and their internal representation in JavaScript, providing you with a clearer understanding of how they function and how you can leverage them effectively in your code.

**Understanding Objects in JavaScript**

At its core, JavaScript is an object-oriented language. An object is a complex data type that can store a collection of key-value pairs, where each key represents a property name and each value is associated with that property. This allows developers to represent real-world entities or abstract concepts in their code. Objects can be created using either the literal syntax or the constructor syntax:

javascript

// Creating an object using literal syntax

const person = {

firstName: "John",

lastName: "Doe",

age: 30

};

// Creating an object using constructor syntax

const car = new Object();

car.make = "Toyota";

car.model = "Camry";

car.year = 2023;

**Internal Representation of Objects**

Behind the scenes, JavaScript engines like V8 use a combination of hash maps and linked lists to implement objects efficiently. When you create an object, the engine allocates memory for it and maintains a hidden class or internal structure that defines the object's properties and their positions in memory. This helps with quick property access and reduces memory overhead.

**Properties and Methods**

Objects can have properties and methods. Properties are values associated with keys, while methods are functions associated with keys. Here's an example that demonstrates both:

javascript

const circle = {

radius: 5,

calculateArea: function() {

return Math.PI \* this.radius \* this.radius;

}

};

console.log(circle.calculateArea()); // Output: 78.53981633974483

In modern JavaScript, you can also use shorthand syntax to define methods within objects:

javascript

const square = {

sideLength: 4,

calculateArea() {

return this.sideLength \* this.sideLength;

}

};

**Object Prototypes and Inheritance**

JavaScript implements inheritance using a mechanism known as prototypes. Each object in JavaScript has an associated prototype object, which allows properties and methods to be inherited from one object to another. This forms the basis of a powerful feature called prototype-based inheritance.

javascript

// Creating a parent object

const animal = {

sound: "unknown",

makeSound() {

console.log(`The animal makes a ${this.sound} sound.`);

}

};

// Creating a child object that inherits from the parent

const dog = Object.create(animal);

dog.sound = "bark";

dog.makeSound(); // Output: The animal makes a bark sound.

**Working with Object Constructors**

Object constructors are functions that serve as blueprints for creating objects with similar properties and methods. The this keyword inside a constructor refers to the object being created. Constructors are commonly used with the new keyword.

javascript

function Book(title, author) {

this.title = title;

this.author = author;

}

const myBook = new Book("The Great Gatsby", "F. Scott Fitzgerald");

**ES6 Classes**

With the introduction of ECMAScript 2015 (ES6), JavaScript also introduced a more traditional class syntax, making object-oriented programming more intuitive for developers coming from other programming languages.

javascript

class Rectangle {

constructor(width, height) {

this.width = width;

this.height = height;

}

calculateArea() {

return this.width \* this.height;

}

}

const myRectangle = new Rectangle(5, 10);

**Conclusion**

Objects lie at the heart of JavaScript's versatility and power. Their internal representation is an intricate blend of memory allocation, hidden classes, and efficient data structures, all designed to optimize property access and memory usage. By understanding how objects work in JavaScript, you can create well-organized, maintainable code and harness the language's full potential for building dynamic and interactive web applications. Whether you're using object literals, constructors, prototypes, or ES6 classes, objects remain an essential tool in your JavaScript toolkit.

Bottom of Form