DAY 1 TASK

Write a blog on the Difference between HTTP1.1 and HTTP2

## The Difference Between HTTP/1.1 and HTTP/2: A Detailed Comparison

The \*\*Hypertext Transfer Protocol (HTTP)\*\* is the foundation of data communication on the web. Over the years, it has evolved to meet the growing demands of web performance and user experience. Two major versions that are widely used today are \*\*HTTP/1.1\*\* and \*\*HTTP/2\*\*. While both serve the same purpose—transmitting web data—they differ significantly in terms of performance and features. Let’s dive into the key differences between HTTP/1.1 and HTTP/2 and understand how these advancements improve the modern web.

Multiplexing: Parallel vs. Sequential Requests\*\*

In HTTP/1.1, only one request can be made per TCP connection at a time. This means that each request must wait for the previous one to complete before the next can begin, leading to a "head-of-line blocking" issue. Browsers often open multiple connections (up to 6 per domain) to counteract this bottleneck, but it’s inefficient.

One of the biggest improvements in HTTP/2 is \*\*multiplexing\*\*, which allows multiple requests to be sent over a single TCP connection concurrently. This means that multiple resources, such as HTML, CSS, and JavaScript, can be downloaded at the same time without waiting for the other to finish. This drastically reduces latency and improves page load speed.

Header Compression: Reducing Redundancy

In HTTP/1.1, headers (metadata about the request/response) are sent in plain text, and they are often repetitive across requests. For large pages with numerous resources, this can result in substantial overhead.

HTTP/2 introduces \*\*HPACK compression\*\* for headers. This technique compresses the headers, reducing the size of the data being transmitted. Additionally, HTTP/2 allows headers to be remembered from previous requests, further minimizing the amount of redundant data sent.

Server Push: Proactive Resource Delivery

With HTTP/1.1, the server waits for the client (browser) to request each resource. For instance, after receiving the HTML of a webpage, the browser will request associated resources like stylesheets, scripts, and images one by one.

HTTP/2 supports \*\*Server Push\*\*, which allows the server to proactively send resources to the client without waiting for the client to request them. If the server knows that the client will need certain files (like CSS or JS), it can send them immediately after the initial HTML response. This reduces round-trip delays and speeds up page rendering.

Binary vs. Textual Transmission

HTTP/1.1 uses a plain-text format for transmitting data, making it more readable for humans but less efficient for machines. Parsing plain text adds overhead.

HTTP/2 uses a \*\*binary protocol\*\*, which is more efficient for machines to parse and interpret. This change helps improve the performance and speed of data transfer since computers handle binary data much more quickly than text-based data.

Stream Prioritization

HTTP/1.1 does not offer any built-in mechanism for prioritizing the transmission of certain resources over others. This can result in slower loading times for critical resources like CSS or JavaScript files, affecting user experience.

HTTP/2 allows the client to assign priorities to different streams. This means important resources, such as the main CSS or JavaScript files, can be delivered first, ensuring faster rendering of the page. Non-critical resources like images can be loaded later.

Connection Management: Fewer Connections, Better Performance

- \*\*HTTP/1.1:\*\* To circumvent the limitations of single-request-per-connection, browsers open multiple TCP connections to the same server. However, this can create unnecessary load on both the client and server, reducing overall efficiency.

- \*\*HTTP/2:\*\* Since HTTP/2 supports multiplexing, all communication between the client and server can happen over a single connection. This reduces the overhead of opening and managing multiple connections, making it much more efficient.

Security: HTTPS by Default

HTTP/1.1: HTTP/1.1 does not enforce encryption, though websites can implement HTTPS (HTTP Secure) to add a layer of security.

HTTP/2: While HTTP/2 itself does not mandate encryption, most implementations require HTTPS. As a result, HTTP/2 is generally used over secure connections, ensuring safer data transmission and protecting against man-in-the-middle attacks.

Adoption and Compatibility

HTTP/1.1HTTP/1.1 has been around since 1999 and is supported by all browsers and servers. While it is slower and less efficient than HTTP/2, it remains widely used, especially for legacy systems.

HTTP/2, introduced in 2015, has seen broad adoption. Most modern browsers and servers now support HTTP/2, and many websites have migrated to this newer protocol to take advantage of its performance benefits. However, it may not be supported by some older systems.

Write a blog about objects and its internal representation in Javascript

Objects, in JavaScript, is it’s most important data-type and forms the building blocks for modern JavaScript. These objects are quite different from JavaScript’s primitive data-types(Number, String, Boolean, null, undefined and symbol) in the sense that while these primitive data-types all store a single value each (depending on their types).

Objects are more complex and each object may contain any combination of these primitive data-types as well as reference data-types.  
An object, is a reference data type. Variables that are assigned a reference value are given a reference or a pointer to that value. That reference or pointer points to the location in memory where the object is stored. The variables don’t actually store the value.

Loosely speaking, objects in JavaScript may be defined as an unordered collection of related data, of primitive or reference types, in the form of “key: value” pairs. These keys can be variables or functions and are called properties and methods, respectively, in the context of an object.

For Eg. If your object is a student, it will have properties like name, age, address, id, etc and methods like updateAddress, updateNam, etc.

**Objects and properties**

A JavaScript object has properties associated with it. A property of an object can be explained as a variable that is attached to the object. Object properties are basically the same as ordinary JavaScript variables, except for the attachment to objects. The properties of an object define the characteristics of the object. You access the properties of an object with a simple dot-notation:

objectName.propertyName

Like all JavaScript variables, both the object name (which could be a normal variable) and property name are case sensitive. You can define a property by assigning it a value. For example, let’s create an object named myCar and give it properties named make, model, and year as follows:

var myCar = new Object();  
myCar.make = 'Ford';  
myCar.model = 'Mustang';  
myCar.year = 1969;

Unassigned properties of an object are [undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined) (and not [null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/null)).

myCar.color; // undefined

Properties of JavaScript objects can also be accessed or set using a bracket notation (for more details see [property accessors](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Property_Accessors)). Objects are sometimes called *associative arrays*, since each property is associated with a string value that can be used to access it. So, for example, you could access the properties of the myCar object as follows:

myCar['make'] = 'Ford';  
myCar['model'] = 'Mustang';  
myCar['year'] = 1969;

An object property name can be any valid JavaScript string, or anything that can be converted to a string, including the empty string. However, any property name that is not a valid JavaScript identifier (for example, a property name that has a space or a hyphen, or that starts with a number) can only be accessed using the square bracket notation. This notation is also very useful when property names are to be dynamically determined (when the property name is not determined until runtime). Examples are as follows:

// four variables are created and assigned in a single go,   
// separated by commas  
var myObj = new Object(),  
 str = 'myString',  
 rand = Math.random(),  
 obj = new Object();  
myObj.type = 'Dot syntax';  
myObj['date created'] = 'String with space';  
myObj[str] = 'String value';  
myObj[rand] = 'Random Number';  
myObj[obj] = 'Object';  
myObj[''] = 'Even an empty string';console.log(myObj);

You can also access properties by using a string value that is stored in a variable:

var propertyName = 'make';  
myCar[propertyName] = 'Ford';propertyName = 'model';  
myCar[propertyName] = 'Mustang';

You can use the bracket notation with [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) to iterate over all the enumerable properties of an object. To illustrate how this works, the following function displays the properties of the object when you pass the object and the object's name as arguments to the function:

function showProps(obj, objName) {  
 var result = ``;  
 for (var i in obj) {  
 // obj.hasOwnProperty() is used to filter out properties from the object's prototype chain  
 if (obj.hasOwnProperty(i)) {  
 result += `${objName}.${i} = ${obj[i]}\n`;  
 }  
 }  
 return result;  
}

So, the function call showProps(myCar, "myCar") would return the following:

myCar.make = Ford  
myCar.model = Mustang  
myCar.year = 1969

**Creating Objects In JavaScript :**

**Create JavaScript Object with Object Literal**

One of easiest way to create a javascript object is object literal, simply define the property and values inside curly braces as shown below

let bike = {name: 'SuperSport', maker:'Ducati', engine:'937cc'};

**Create JavaScript Object with Constructor**

Constructor is nothing but a function and with help of new keyword, constructor function allows to create multiple objects of same flavor as shown below

function Vehicle(name, maker) {  
 this.name = name;  
 this.maker = maker;  
}  
let car1 = new Vehicle(’Fiesta’, 'Ford’);  
let car2 = new Vehicle(’Santa Fe’, 'Hyundai’)  
console.log(car1.name); //Output: Fiesta  
console.log(car2.name); //Output: Santa Fe

**Using the JavaScript Keyword new**

The following example also creates a new JavaScript object with four properties:

Example

var person = new Object();  
person.firstName = “John”;  
person.lastName = “Doe”;  
person.age = 50;  
person.eyeColor = “blue”;

**Using the**Object.create**method**

Objects can also be created using the [Object.create()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/create" \t "_blank) method. This method can be very useful, because it allows you to choose the prototype object for the object you want to create, without having to define a constructor function.

// Animal properties and method encapsulation  
var Animal = {  
 type: 'Invertebrates', // Default value of properties  
 displayType: function() { // Method which will display type of Animal  
 console.log(this.type);  
 }  
};  
// Create new animal type called animal1   
var animal1 = Object.create(Animal);  
animal1.displayType(); // Output:Invertebrates  
// Create new animal type called Fishes  
var fish = Object.create(Animal);  
fish.type = 'Fishes';  
fish.displayType(); // Output:Fishes