# **HTTP1.1 vs HTTP2:**

## Background

### HTTP/1.1

Developed by Timothy Berners-Lee in 1989 as a communication standard for the World Wide Web, HTTP is a top-level application protocol that exchanges information between a client computer and a local or remote web server. In this process, a client sends a text-based request to a server by calling a *method* like GET or POST. In response, the server sends a resource like an HTML page back to the client.

For example, let’s say you are visiting a website at the domain www.example.com. When you navigate to this URL, the web browser on your computer sends an HTTP request in the form of a text-based message, similar to the one shown here:

GET /index.html HTTP/1.1

Host: www.example.com

### HTTP/2

HTTP/2 began as the SPDY protocol, developed primarily at Google with the intention of reducing web page load latency by using techniques such as compression, multiplexing, and prioritization. This protocol served as a template for HTTP/2 when the Hypertext Transfer Protocol working group httpbis of the [IETF (Internet Engineering Task Force)](https://www.ietf.org/) put the standard together, culminating in the publication of HTTP/2 in May 2015. From the beginning, many browsers supported this standardization effort, including Chrome, Opera, Internet Explorer, and Safari. Due in part to this browser support, there has been a significant adoption rate of the protocol since 2015, with especially high rates among new sites.

## Delivery Models

### HTTP/1.1 — Pipelining and Head-of-Line Blocking

The first response that a client receives on an HTTP GET request is often not the fully rendered page. Instead, it contains links to additional resources needed by the requested page. The client discovers that the full rendering of the page requires these additional resources from the server only after it downloads the page. Because of this, the client will have to make additional requests to retrieve these resources. In HTTP/1.0, the client had to break and remake the TCP connection with every new request, a costly affair in terms of both time and resources.

### HTTP/2 — Advantages of the Binary Framing Layer

In HTTP/2, the binary framing layer encodes requests/responses and cuts them up into smaller packets of information, greatly increasing the flexibility of data transfer.

Let’s take a closer look at how this works. As opposed to HTTP/1.1, which must make use of multiple TCP connections to lessen the effect of HOL blocking, HTTP/2 establishes a single connection object between the two machines. Within this connection there are multiple *streams* of data. Each stream consists of multiple messages in the familiar request/response format. Finally, each of these messages split into smaller units called *frames*:

# 

# **History of HTTP**

The Hypertext Transfer Protocol (HTTP) is one of the most ubiquitous and widely adopted application protocols on the Internet: it is the common language between clients and servers, enabling the modern web. From its simple beginnings as a single keyword and document path, it has become the protocol of choice not just for browsers, but for virtually every Internet-connected software and hardware application.

## **HTTP 0.9: The One-Line Protocol**

The original HTTP proposal by Tim Berners-Lee was designed with *simplicity in mind* as to help with the adoption of his other nascent idea: the World Wide Web. The strategy appears to have worked: aspiring protocol designers, take note.

In 1991, Berners-Lee outlined the motivation for the new protocol and listed several high-level design goals: file transfer functionality, ability to request an index search of a hypertext archive, format negotiation, and an ability to refer the client to another server. To prove the theory in action, a simple prototype was built, which implemented a small subset of the proposed functionality

## **HTTP/1.0: Rapid Growth and Informational RFC**

The period from 1991 to 1995 is one of rapid coevolution of the HTML specification, a new breed of software known as a "web browser," and the emergence and quick growth of the consumer-oriented public Internet infrastructure.

The growing list of desired capabilities of the nascent Web and their use cases on the public Web quickly exposed many of the fundamental limitations of HTTP 0.9: we needed a protocol that could serve more than just hypertext documents, provide richer metadata about the request and the response, enable content negotiation, and more. In turn, the nascent community of web developers responded by producing a large number of experimental HTTP server and client implementations through an ad hoc process: implement, deploy, and see if other people adopt it.

## **HTTP/1.1: Internet Standard**

The work on turning HTTP into an official IETF Internet standard proceeded in parallel with the documentation effort around HTTP/1.0 and happened over a period of roughly four years: between 1995 and 1999. In fact, the first official HTTP/1.1 standard is defined in RFC 2068, which was officially released in January 1997, roughly six months after the publication of HTTP/1.0. Then, two and a half years later, in June of 1999, a number of improvements and updates were incorporated into the standard and were released as RFC 2616.

## **HTTP/2: Improving Transport Performance**

Since its publication, RFC 2616 has served as a foundation for the unprecedented growth of the Internet: billions of devices of all shapes and sizes, from desktop computers to the tiny web devices in our pockets, speak HTTP every day to deliver news, video, and millions of other web applications we have all come to depend on in our lives.

What began as a simple, one-line protocol for retrieving hypertext quickly evolved into a generic hypermedia transport, and now a decade later can be used to power just about any use case you can imagine. Both the ubiquity of servers that can speak the protocol and the wide availability of clients to consume it means that many applications are now designed and deployed exclusively on top of HTTP.

# 

# **Differences between Node.js and the Browser**

Both the browser and Node.js use JavaScript as their programming language.

Building apps that run in the browser is a completely different thing than building a Node.js application.

Despite the fact that it's always JavaScript, there are some key differences that make the experience radically different.

From the perspective of a frontend developer who extensively uses JavaScript, Node.js apps bring with them a huge advantage: the comfort of programming everything - the frontend and the backend - in a single language.

You have a huge opportunity because we know how hard it is to fully, deeply learn a programming language, and by using the same language to perform all your work on the web - both on the client and on the server, you're in a unique position of advantage.

What changes is the ecosystem.

In the browser, most of the time what you are doing is interacting with the DOM, or other Web Platform APIs like Cookies. Those do not exist in Node.js, of course. You don't have the document, window and all the other objects that are provided by the browser.

And in the browser, we don't have all the nice APIs that Node.js provides through its modules, like the filesystem access functionality.

Another big difference is that in Node.js you control the environment. Unless you are building an open source application that anyone can deploy anywhere, you know which version of Node.js you will run the application on. Compared to the browser environment, where you don't get the luxury to choose what browser your visitors will use, this is very convenient.

This means that you can write all the modern ES6-7-8-9 JavaScript that your Node.js version supports.

Since JavaScript moves so fast, but browsers can be a bit slow to upgrade, sometimes on the web you are stuck with using older JavaScript / ECMAScript releases.

You can use Babel to transform your code to be ES5-compatible before shipping it to the browser, but in Node.js, you won't need that.

Another difference is that Node.js uses the CommonJS module system, while in the browser we are starting to see the ES Modules standard being implemented.

In practice, this means that for the time being you use require() in Node.js and import in the browser.

# **What Happens When You Type in a URL**

1. The browser looks up the IP address for the domain name via DNS
2. The browser sends a HTTP *request* to the server
3. The server sends back a HTTP *response*
4. The browser begins rendering the HTML
5. The browser sends requests for additional objects embedded in HTML (images, Css, JavaScript) and repeats steps 3-5.
6. Once the page is loaded, the browser sends further async requests as needed.

When you type “[https://wsvincent.com”](https://wsvincent.xn--com-9o0a/) into your browser the first thing that happens is a Domain Name Server (DNS) matches “[wsvincent.com](http://wsvincent.com/)” to an IP address. Then the browser sends an HTTP request to the server and the server sends back an HTTP response. The browser begins rendering the HTML on the page while also requesting any additional resources such as CSS, JavaScript, images, etc. Each subsequent request completes a request/response cycle and is rendered in turn by the browser. Then once the page is loaded some sites (though not mine) will make further asynchronous requests.

If I were asked to explain further I might start talking about *how* the server and browser connect via [TCP](https://en.wikipedia.org/wiki/Internet_protocol_suite). And we could discuss encryption via https too.

**Difference between copy by value and copy by reference.**

**1.Copy by value**

In JavaScript, we can divide data types into two different buckets, primitive data types and objects.

There are six primitive data types in JavaScript: string, number, boolean, undefined, null, and symbol as of ES6.

Primitive data types are passed, or copied, **by value** and are immutable, meaning that the existing value cannot be altered the way an array or an object can.

Ex.

Var x = 10

Var y = x

Console.log(y)

Output: 10

Here we have created two variables, x = 10 and y = x. Since 10 is a number and a primitive value, when we set y = x we are actually copying the value, i.e. 10, and assigning it to y.If we were to change the value of x, we would see that y retains its value of 10. Again, this is because primitive values are **copied**, so y's value is independent of x's value. Think of it as making a photocopy of a picture. After making the copy, you have two identical pictures: an original and a facsimile. If you were to cut the original in half, only the original would be altered and the facsimile would remain exactly the same.

Ex.

X =10

Y =x

Console.log(y)

X =20

Console.log(y)

Output: 10

10

**2.Copy by reference**

Objects, on the other hand, are passed**by reference** and point to a location in memory for the value

Ex.

Var x ={dog: “poodle”}

Var = x

Console.log(y.dog)

Output: “poodle”

In this example, x is now an object pointing to {dog: "poodle"}. When we create the variable y and assign it the value of x, we are now able to tap into the various properties of x which includes the value for dog, i.e. "poodle".

we see that the values for both x and y are not data types but **references** to an addresses in memory, the same address in fact! Now let’s take a look at what happens to x if we add a new property of size to y…

Ex.

var x= {dog: “poodle”}

var y=x

y.size = “large”

console.log(x)

output: {dog: “poodle”, size: “large”}

x still returns an object but now it has an additional property of size also! Again, this is because both x and y point to the same reference object, so any changes made to one variable, will be visible in the other.

To help me remember this concept, I like to think of reference values as a house and the variables as people who live in that house. All of the residents (variables) can say “I have a house” and point to the same house. If a single resident decides they want to paint the house yellow, then all the residents now have a yellow house because it is shared.

**How to copy by value a composite data type (array + objects)**

There are 3 ways to copy by value for composite data types.

1. Using the spread (...) operator
2. Using the Object.assign() method
3. Using the JSON.stringify() and JSON.parse() methods

# **1. Using Spread**

**Spread operator** allows an iterable to expand in places where 0+ arguments are expected. It is mostly used in the variable array where there is more than 1 values are expected. It allows us the privilege to obtain a list of parameters from an array.Using spread will clone your object.This will be a shallow copy.

# **2. Using Object.assign()**

The **Object.assign()** method copies all enumerable own properties from one or more source objects to a target object. It returns the target object.this will be a shallow copy.

# **3. Using JSON.parse() and JSON.stringify()**

The JSON object, available in all modern browsers, has two useful methods to deal with JSON-formatted content: parse and stringify. JSON.parse() takes a JSON string and transforms it into a JavaScript object. JSON.stringify() takes a JavaScript object and transforms it into a JSON string.Using JSON.parse() and JSON.stringify() for copy performs deep copy .

# **GUVI : Zen Code-Sprint : JavaScript Practice problems in JSON(Objects) and List**

# **Basic Tasks to play with JSON**

1. Add height and weight to Fluffy
2. Fluffy name is spelled wrongly. Update it to Fluffyy

var cat = {

    name: 'Fluffy',

    activities: ['play', 'eat cat food'],

    catFriends: [

    {

    name: 'bar',

    activities: ['be grumpy', 'eat bread omblet'],

    weight: 8,

    furcolor: 'white'

    },

    {

    name: 'foo',

    activities: ['sleep', 'pre-sleep naps'],

    weight: 3

    }

    ]

   }

   cat.name="fluffyy"

   console.log(cat)

1. List all the activities of Fluffyy’s catFriends.
2. var cat = {
3. name: 'Fluffy',
4. activities: ['play', 'eat cat food'],
5. catFriends: [
6. {
7. name: 'bar',
8. activities: ['be grumpy', 'eat bread omblet'],
9. weight: 8,
10. furcolor: 'white'
11. },
12. {
13. name: 'foo',
14. activities: ['sleep', 'pre-sleep naps'],
15. weight: 3
16. }
17. ]
18. }
20. for(var i in cat.catFriends){
22. for(var j in cat.catFriends[i]){
24. var b = cat.catFriends[i].activities
26. }
27. console.log(b)
28. }
30. Print the catFriends names.
31. var cat = {
32. name: 'Fluffy',
33. activities: ['play', 'eat cat food'],
34. catFriends: [
35. {
36. name: 'bar',
37. activities: ['be grumpy', 'eat bread omblet'],
38. weight: 8,
39. furcolor: 'white'
40. },
41. {
42. name: 'foo',
43. activities: ['sleep', 'pre-sleep naps'],
44. weight: 3
45. }
46. ]
47. }
49. for(var i in cat.catFriends){
51. for(var j in cat.catFriends[i]){
53. var b = cat.catFriends[i].name
55. }
56. console.log(b)
57. }

60. Print the total weight of catFriends
61. var cat = {
62. name: 'Fluffy',
63. activities: ['play', 'eat cat food'],
64. catFriends: [
65. {
66. name: 'bar',
67. activities: ['be grumpy', 'eat bread omblet'],
68. weight: 8,
69. furcolor: 'white'
70. },
71. {
72. name: 'foo',
73. activities: ['sleep', 'pre-sleep naps'],
74. weight: 3
75. }
76. ]
77. }

80. var a = 0
81. for(var i in cat.catFriends){
83. //for(var j in cat.catFriends[i]){
85. a =a+ +cat.catFriends[i].weight
87. }
88. console.log(a)
90. //}

# **Problem 0**

var myCar = {

   make: 'Bugatti',

   model: 'Bugatti La Voiture Noire',

   year: 2019,

   accidents: [

   {

   date: '3/15/2019',

   damage\_points: '5000',

   atFaultForAccident: true

   },

   {

   date: '7/4/2022',

   damage\_points: '2200',

   atFaultForAccident: true

   },

   {

   date: '6/22/2021',

   damage\_points: '7900',

   atFaultForAccident: true

   }

   ]

  }

  for(var i in myCar.accidents){

     for(var j in myCar.accidents[i]){

     var b=myCar.accidents[i].date

   }

   console.log(b)

  }

# **Problem 1**

var obj={name : 'RajiniKanth', age : 33, hasPets : false}

  function printAllValues(obj){

     console.log(Object.values(obj))

  }

  printAllValues(obj)

# **Problem 2**

var obj={name : 'RajiniKanth', age : 33, hasPets : false}

  function printAllValues(obj){

     console.log(Object.keys(obj))

  }

  printAllValues(obj)

# **Problem 4**

var arr = ["GUVI", "I", "am", "a geek"];

function transformFirstAndLast(arr) {

   let nobj={}

   let arrlg=arr.length

   nobj[arr[0]]=arr[arrlg-1]

 return nobj;

}

console.log(transformFirstAndLast(arr))

# **Problem 6**

var arraysample = [[["firstName", "Vasanth"], ["lastName", "Raja"], ["age", 24], ["role", "JSWizard"]], [["firstName", "Sri"], ["lastName", "Devi"], ["age", 28], ["role", "Coder"]]];

var result = arraysample.map(Object.fromEntries);

   console.log(result)

# **Problem 8**

var securityQuestions = [

    {

    question: "What was your first pet’s name?",

    expectedAnswer: "FlufferNutter"

    },

    {

    question: "What was the model year of your first car?",

    expectedAnswer: "1985"

    },

    {

    question: "What city were you born in?",

    expectedAnswer: "NYC"

    }

   ]

   function chksecurityQuestions(securityQuestions, ques, ans){

    for(let i=0;i<securityQuestions.length;i++){

      if(securityQuestions[i].question===ques){

        if(securityQuestions[i].expectedAnswer===ans){

          return true

        }

      }

    }

    return false

   }

   //Test case1:

   var ques = "What was your first pet’s name?";

   var ans  =  "FlufferNutter";

   var status = chksecurityQuestions(securityQuestions, ques, ans);

   console.log(status); // true

   //Test case2:

   var ques = "What was your first pet’s name?";

   var ans  =  "DufferNutter";

   var status = chksecurityQuestions(securityQuestions, ques, ans);

   console.log(status); // flase

# **Problem 9**

var students = [

        {

        name: "Siddharth Abhimanyu", age: 21}, { name: "Malar", age: 25},

        {name: "Maari",age: 18},{name: "Bhallala Deva",age: 17},

        {name: "Baahubali",age: 16},{name: "AAK chandran",age: 23},   {name:"Gabbar Singh",age: 33},{name: "Mogambo",age: 53},

        {name: "Munnabhai",age: 40},{name: "Sher Khan",age: 20},

        {name: "Chulbul Pandey",age: 19},{name: "Anthony",age: 28},

        {name: "Devdas",age: 56}

        ];

       function returnMinors(arr)

       {

           for(var i in students){

               if(students[i].age<20){

                   console.log(students[i].name)

               }

           }

       }

       console.log(returnMinors(students));

# **Problem 3**

var object = {name: "ISRO", age: 35, role: "Scientist"};

 const ver3 = Object.entries(object)

 console.log(ver3)