Write a blog on Difference between HTTP1.1 vs HTTP2

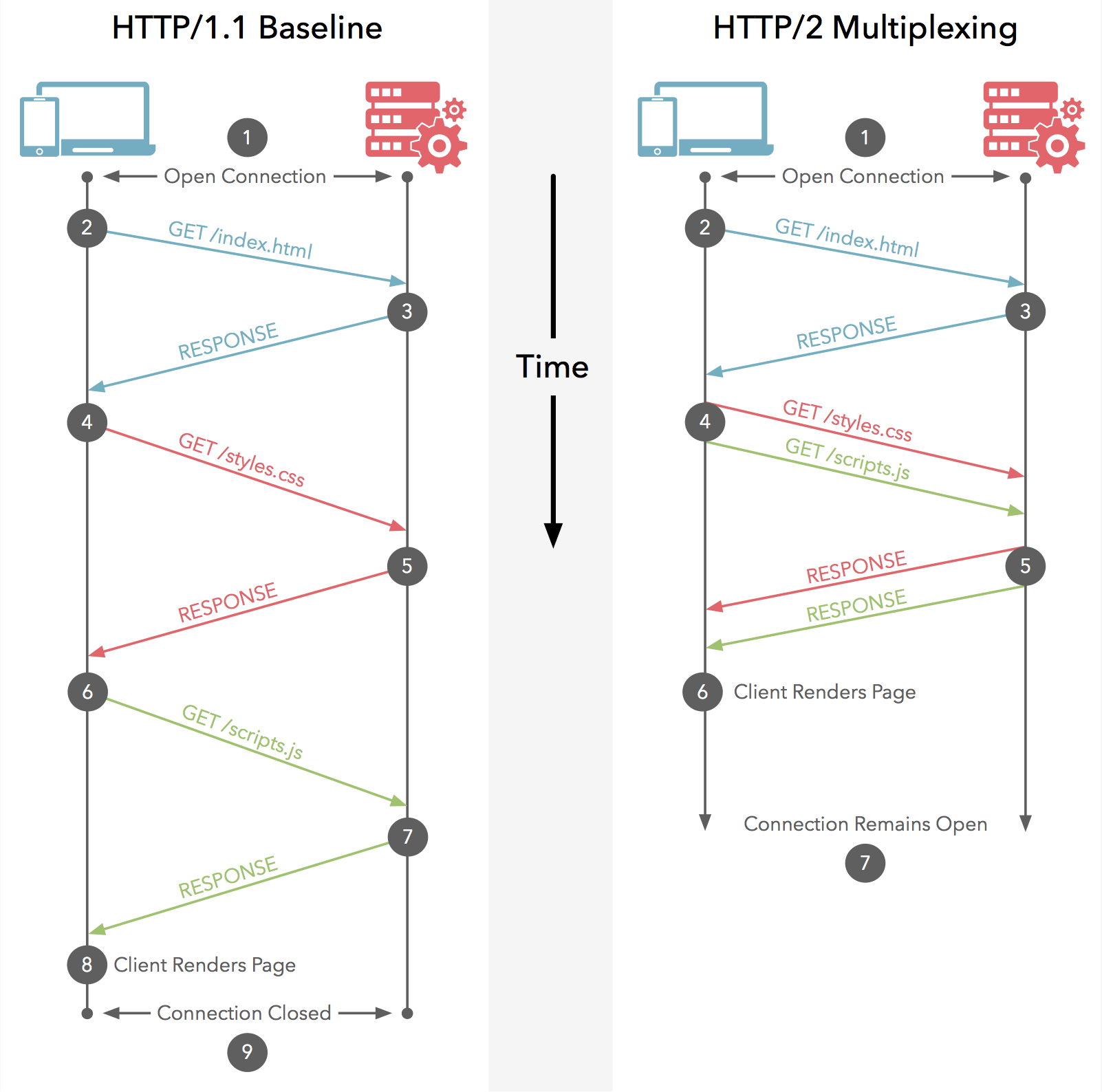
HTTP/1.1 has been around for more than a decade. With Google’s SPDY leading the way in 2015, the IETF (Internet Engineering Task Force) gave us HTTP/2, which introduces several features to reduce page load times.

HTTP/2 achieves faster webpage loading without performance optimizations that require extensive human efforts in terms of development. It significantly reduces the complexities that had crept into HTTP/1.1 and gives us a robust protocol which, though not without its flaws, will perhaps stand the test of time. Before making this leap forward, let’s trace our steps back to when the internet was in its infancy to understand how the different versions evolved into the current form.

HTTP2 is much faster and more reliable than HTTP1. HTTP1 loads a single request for every TCP connection, while HTTP2 avoids network delay by using multiplexing. HTTP is a network delay sensitive protocol in the sense that if there is less network delay, then the page loads faster.

http/2 is binary, instead of textual

binary protocols are more efficient to parse, more compact "on the wire," and, most importantly, they are much less error-prone compared to textual protocols like http/1.x, because they often have a number of affordances to "help" with things like whitespace handling, capitalization, line endings, blank lines, and so on.



====================================================================

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.  
An object can be created with figure brackets {…} with an optional list of properties. A property is a “key: value” pair, where a key is a string (also called a “property name”), and value can be anything.

To understand this rather abstract definition, let us look at an example of a JavaScript Object :

let school = {  
name : “Vivekananda School”,  
location : “Delhi”,  
established : “1971”  
}  
In the above example “name”, “location”, “established” are all “keys” and “Vivekananda School”, “Delhi” and 1971 are values of these keys respectively.

Each of these keys is referred to as properties of the object. An object in JavaScript may also have a function as a member, in which case it will be known as a method of that object.

Let us see such an example :  
// javascript code demonstrating a simple object  
let school = {  
name: ‘Vivekananda School’,  
location : ‘Delhi’,  
established : ‘1971’,  
displayInfo : function(){  
console.log(${school.name} was established  
in ${school.established} at ${school.location});  
}  
}  
school.displayInfo();

Output:  
In the above example, “displayinfo” is a method of the school object that is being used to work with the object’s data, stored in its properties.  
Properties of JavaScript Object  
The property names can be strings or numbers. In case the property names are numbers, they must be accessed using the “bracket notation” like this :  
let school = {  
name: ‘Vivekananda School’,  
location : ‘Delhi’,  
established : ‘1971’,  
20 : 1000,  
displayInfo : function(){  
console.log(The value of the key 20 is ${school[‘20’]});  
}  
}  
school.displayInfo();

Output:  
But more on the bracket notation later.  
Property names can also be strings with more than one space separated words. In which case, these property names must be enclosed in quotes :  
let school = {  
“school name” : “Vivekananda School”,  
}  
Like property names which are numbers, they must also be accessed using the bracket notation.

Like if we want to access the ‘Vivekananda’ from ‘Vivekananda School’ we can do something like this:  
// bracket notation  
let school = {  
name: ‘Vivekananda School’,  
displayInfo : function(){  
console.log(${school.name.split(‘ ‘)[0]});  
}  
}  
school.displayInfo(); // Vivekananda

Output:  
In the above code, we made use of bracket notation and also split method provided by javascript which you will learn about in strings article.  
Creating Objects  
There are several ways or syntax’s to create objects. One of which, known as the Object literal syntax, we have already used. Besides the object literal syntax, objects in JavaScript may also be created using the constructors, Object Constructor or the prototype pattern.  
Using the Object literal syntax : Object literal syntax uses the {…} notation to initialize an object an its methods/properties directly.

Let us look at an example of creating objects using this method :  
var obj = {  
member1 : value1,  
member2 : value2,  
};  
These members can be anything — strings, numbers, functions, arrays or even other objects. An object like this is referred to as an object literal. This is different from other methods of object creation which involve using constructors and classes or prototypes, which have been discussed below.

Object Constructor : Another way to create objects in JavaScript involves using the “Object” constructor. The Object constructor creates an object wrapper for the given value. This, used in conjunction with the “new” keyword allows us to initialize new objects.  
Example :  
const school = new Object();  
school.name = ‘Vivekanada school’;  
school.location = ‘Delhi’;  
school.established = 1971;

school.displayInfo = function(){  
console.log(${school.name} was established  
in ${school.established} at ${school.location});  
}

school.displayInfo();  
Output:  
The two methods mentioned above are not well suited to programs that require the creation of multiple objects of the same kind, as it would involve repeatedly writing the above lines of code for each such object. To deal with this problem, we can make use of two other methods of object creation in JavaScript that reduces this burden significantly, as mentioned below:  
Constructors: Constructors in JavaScript, like in most other OOP languages, provides a template for creation of objects. In other words, it defines a set of properties and methods that would be common to all objects initialized using the constructor.

Let us see an example :  
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

Output:  
Notice the usage of the “new” keyword before the function Vehicle. Using the “new” keyword in this manner before any function turns it into a constructor. What the “new Vehicle()” actually does is :  
It creates a new object and sets the constructor property of the object to schools (It is important to note that this property is a special default property that is not enumerable and cannot be changed by setting a “constructor: someFunction” property manually).

Then, it sets up the object to work with the Vehicle function’s prototype object ( Each function in JavaScript gets a prototype object, which is initially just an empty object but can be modified.The object, when instantiated inherits all properties from its constructor’s prototype object).

Then calls Vehicle() in the context of the new object, which means that when the “this” keyword is encountered in the constructor(vehicle()), it refers to the new object that was created in the first step.  
Once this is finished, the newly created object is returned to car1 and car2(in the above example).

Inside classes, there can be special methods named constructor().  
class people {  
constructor()  
{  
this.name = “Adam”;  
}  
}

let person1 = new people();

// Output : Adam  
console.log(person1.name);

Output:  
Having more than one function in a class with the name of constructor() results in an error.  
Prototypes : Another way to create objects involves using prototypes. Every JavaScript function has a prototype object property by default(it is empty by default). Methods or properties may be attached to this property. A detailed description of prototypes is beyond the scope of this introduction to objects.

However you may familiarize yourself with the basic syntax used as below:  
let obj = Object.create(prototype\_object, propertiesObject)  
// the second propertiesObject argument is optional  
An example of making use of the Object.create() method is:  
let footballers = {  
position: “Striker”  
}

let footballer1 = Object.create(footballers);

// Output : Striker  
console.log(footballer1.position);  
Output:  
In the above example footballers served as a prototype for creating the object “footballer1”.

All objects created in this way inherits all properties and methods from its prototype objects. Prototypes can have prototypes and those can have prototypes and so on. This is referred to as prototype chaining in JavaScript. This chain terminates with the Object.prototype which is the default prototype fallback for all objects. Javascript objects, by default, inherit properties and methods from Object.prototype but these may easily be overridden. It is also interesting to note that the default prototype is not always Object.prototype.For example Strings and Arrays have their own default prototypes — String.prototype and Array.prototype respectively.

Accessing Object Members  
Object members(properties or methods) can be accessed using the  
dot notation :  
(objectName.memberName)  
let school = {  
name : “Vivekanada”,  
location : “Delhi”,  
established : 1971,  
20 : 1000,  
displayinfo : function() {  
console.log(${school.name} was established  
in ${school.established} at ${school.location});  
}

}

console.log(school.name);

console.log(school.established);  
Output:

Bracket Notation :  
objectName[“memberName”]  
let school = {  
name : “Vivekanada School”,  
location : “Delhi”,  
established : 1995,  
20 : 1000,  
displayinfo : function() {  
document.write(${school.name} was established  
in ${school.established} at ${school.location});  
}  
}

// Output : Vivekanada School  
console.log(school[‘name’]);

// Output: 1000  
console.log(school[‘20’]);  
Output:  
Unlike the dot notation, the bracket keyword works with any string combination, including, but not limited to multi-word strings.

For example:  
somePerson.first name // invalid  
somePerson[“first name”] // valid  
Unlike the dot notation, the bracket notation can also contain names which are results of any expressions variables whose values are computed at run-time.  
For instance :  
let key = “first name” somePerson[key] = “Name Surname”  
Similar operations are not possible while using the dot notation.