1. Write a blog on Difference between HTTP1.1 vs HTTP2

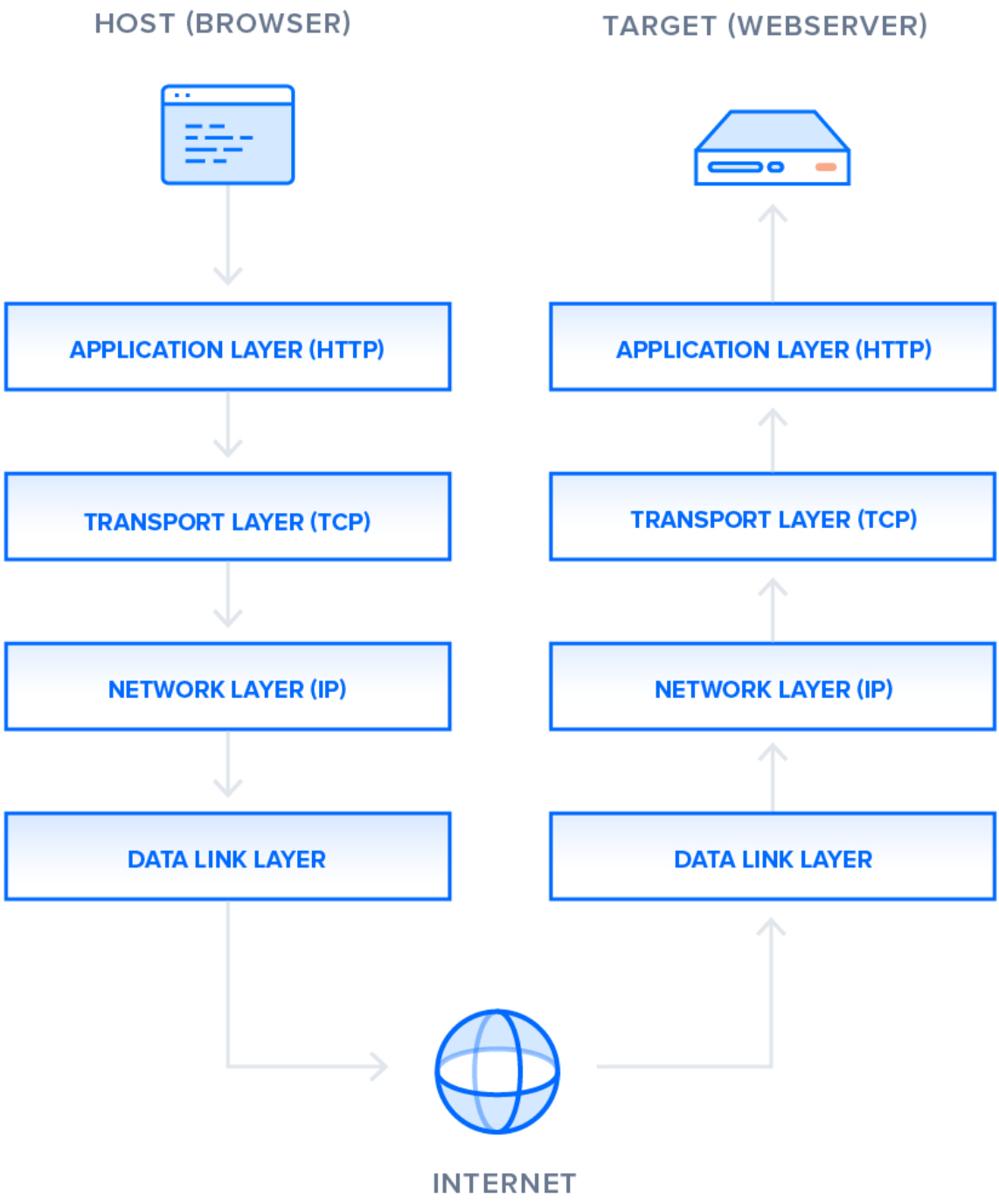
Introduction

The Hypertext Transfer Protocol, or HTTP, is an application protocol that has been the de facto standard for communication on the World Wide Web since its invention in 1989. From the release of HTTP/1.1 in 1997 until recently, there have been few revisions to the protocol. But in 2015, a reimagined version called HTTP/2 came into use, which offered several methods to decrease latency, especially when dealing with mobile platforms and server-intensive graphics and videos. HTTP/2 has since become increasingly popular, with some estimates suggesting that around a third of all websites in the world support it. In this changing landscape, web developers can benefit from understanding the technical differences between HTTP/1.1 and HTTP/2, allowing them to make informed and efficient decisions about evolving best practices.

After reading this article, you will understand the main differences between HTTP/1.1 and HTTP/2, concentrating on the technical changes HTTP/2 has adopted to achieve a more efficient Web protocol.

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:


HOST (BROWSER)
TARGET (WEBSERVER)
APPLICATION LAYER (HTTP)
TRANSPORT LAYER (TCP)
NETWORK LAYER (IP)
DATA LINK LAYER
APPLICATION LAYER (HTTP)
TRANSPORT LAYER (TCP)
NETWORK LAYER (IP)
DATA LINK LAYER
INTERNET

There is much to discuss about the lower levels of this stack, but in order to gain a high-level understanding of HTTP/2, you only need to know this abstracted layer model and where HTTP figures into it.

With this basic overview of HTTP/1.1 out of the way, we can now move on to recounting the early development of HTTP/2.

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) 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.

From a technical point of view, one of the most significant features that distinguishes HTTP/1.1 and HTTP/2 is the binary framing layer, which can be thought of as a part of the application layer in the internet protocol stack. As opposed to HTTP/1.1, which keeps all requests and responses in plain text format, HTTP/2 uses the binary framing layer to encapsulate all messages in binary format, while still maintaining HTTP semantics, such as verbs, methods, and headers. An application level API would still create messages in the conventional HTTP formats, but the underlying layer would then convert these messages into binary. This ensures that web applications created before HTTP/2 can continue functioning as normal when interacting with the new protocol.

The conversion of messages into binary allows HTTP/2 to try new approaches to data delivery not available in HTTP/1.1, a contrast that is at the root of the

practical differences between the two protocols. The next section will take a look at the delivery model of HTTP/1.1, followed by what new models are made possible by HTTP/2.

Delivery Models

As mentioned in the previous section, HTTP/1.1 and HTTP/2 share semantics, ensuring that the requests and responses traveling between the server and client in both protocols reach their destinations as traditionally formatted messages with headers and bodies, using familiar methods like `GET` and `POST`. But while HTTP/1.1 transfers these in plain-text messages, HTTP/2 encodes these into binary, allowing for significantly different delivery model possibilities. In this section, we will first briefly examine how HTTP/1.1 tries to optimize efficiency with its delivery model and the problems that come up from this, followed by the advantages of the binary framing layer of HTTP/2 and a description of how it prioritizes requests.

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/1.1 takes care of this problem by introducing persistent connections and pipelining. With persistent connections, HTTP/1.1 assumes that a TCP connection should be kept open unless directly told to close. This allows the client to send multiple requests along the same connection without waiting for a response to each, greatly improving the performance of HTTP/1.1 over HTTP/1.0.

Unfortunately, there is a natural bottleneck to this optimization strategy. Since multiple data packets cannot pass each other when traveling to the same destination, there are situations in which a request at the head of the queue that cannot retrieve its required resource will block all the requests behind it. This is known as *head-of-line (HOL) blocking*, and is a significant problem with optimizing connection efficiency in HTTP/1.1. Adding separate, parallel TCP connections could alleviate this issue, but there are limits to the number of concurrent TCP connections possible between a client and server, and each new connection requires significant resources.

These problems were at the forefront of the minds of HTTP/2 developers, who proposed to use the aforementioned binary framing layer to fix these issues, a topic you will learn more about in the next section.

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*:

HTTP/2 — Stream Prioritization

Stream prioritization not only solves the possible issue of requests competing for the same resource, but also allows developers to customize the relative weight of requests to better optimize application performance. In this section, we will break down the process of this prioritization in order to provide better insight into how you can leverage this feature of HTTP/2.

As you know now, the binary framing layer organizes messages into parallel streams of data. When a client sends concurrent requests to a server, it can prioritize the responses it is requesting by assigning a weight between 1 and 256 to each stream. The higher number indicates higher priority. In addition to this, the client also states each stream's dependency on another stream by specifying the ID of the stream on which it depends. If the parent identifier is omitted, the stream is considered to be dependent on the root stream. This is illustrated in the following figure:

Conclusion

As you can see from this point-by-point analysis, HTTP/2 differs from HTTP/1.1 in many ways, with some features providing greater levels of control that can be used to better optimize web application performance and other features simply improving upon the previous protocol. Now that you have gained a high-level perspective on the variations between the two protocols, you can consider how such factors as multiplexing, stream prioritization, flow control, server push, and compression in HTTP/2 will affect the changing landscape of web development.

If you would like to see a performance comparison between HTTP/1.1 and HTTP/2, check out this Google demo that compares the protocols for different latencies. Note that when you run the test on your computer, page load times may vary depending on several factors such as bandwidth, client and server resources available at the time of testing, and so on. If you'd like to study the results of more exhaustive testing, take a look at the article HTTP/2 – A Real-World Performance Test and Analysis. Finally, if you would like to explore how to build a modern web application, you could follow our How To Build a Modern Web Application to Manage Customer Information with Django and React on Ubuntu 18.04 tutorial, or set up your own HTTP/2 server with our How To Set Up Nginx with HTTP/2 Support on Ubuntu 20.04 tutorial.

2.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` (and not `null`).

```
myCar.color; // undefined
```

Properties of JavaScript objects can also be accessed or set using a bracket notation (for more details see 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` 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()` 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();
```