Day 1

# Understanding the Evolution: HTTP/1.1 vs HTTP/2

In the ever-evolving landscape of web development, understanding the protocols that keep the internet running is crucial. Hypertext Transfer Protocol, or HTTP, is the backbone of data communication on the web. Since its inception, HTTP has undergone significant changes, with HTTP/1.1 and HTTP/2 being two of the major versions deployed across the internet. Let's delve into the differences between these two protocols to understand why and how the web has evolved from HTTP/1.1 to HTTP/2.

## HTTP/1.1: The Workhorse of the Web

Introduced in 1997, HTTP/1.1 has been the standard for web communication for over two decades. It introduced several improvements over its predecessor, HTTP/1.0, such as:

- \*\*Persistent Connections:\*\* HTTP/1.1 allows multiple requests and responses over the same TCP connection, reducing the overhead of setting up new connections.

- \*\*Pipelining:\*\* This feature enables multiple requests to be sent out before the first response is received, aiming to improve page load times.

- \*\*Chunked Transfers:\*\* Servers can start sending data before the entire response is ready, improving the perception of responsiveness.

Despite these enhancements, HTTP/1.1 has limitations, primarily due to how it handles requests and responses over a single connection.

## HTTP/2: A New Era of Efficiency

HTTP/2 was standardized in 2015 and brought about significant changes aimed at increasing web performance and efficiency. Here are some key features:

- \*\*Binary Framing Layer:\*\* HTTP/2 introduces a binary protocol that breaks down messages into smaller, manageable frames, improving parsing, multiplexing, and error detection.

- \*\*Multiplexing:\*\* Unlike HTTP/1.1, HTTP/2 allows multiple requests and responses to be in flight at the same time, over a single TCP connection. This dramatically reduces the issues related to multiple connections and head-of-line blocking found in HTTP/1.1.

- \*\*Stream Prioritization:\*\* Resources can be prioritized, allowing more critical resources to be sent earlier than less critical ones.

- \*\*Server Push:\*\* Servers can proactively send resources to the client before the client requests them, potentially reducing wait times.

- \*\*Header Compression:\*\* HTTP/2 uses the HPACK compression format to reduce overhead caused by HTTP headers.

## Practical Impact on Web Development and Performance

The differences between HTTP/1.1 and HTTP/2 have profound implications on web performance and development:

- \*\*Performance:\*\* HTTP/2’s ability to handle multiple requests in a single connection without the order affecting the speed (non-blocking) can significantly reduce page load times.

- \*\*Efficiency:\*\* With header compression and binary framing, HTTP/2 reduces the overhead on each request, allowing more efficient use of network resources.

- \*\*Complexity in Implementation:\*\* While HTTP/2 introduces complexity in terms of implementation (due to binary framing and stream prioritization), modern browsers and servers have largely abstracted these complexities for developers.

## Transitioning from HTTP/1.1 to HTTP/2

The shift from HTTP/1.1 to HTTP/2 does not require changes in the application code, as it is mostly a server-side upgrade. However, to fully utilize the features of HTTP/2, developers might need to rethink how they bundle and serve resources. For instance, techniques like domain sharding and concatenation may no longer provide benefits with HTTP/2 and could actually lead to decreased performance.

## Conclusion

The transition from HTTP/1.1 to HTTP/2 represents a significant leap forward in the efficiency and speed of web communication. As the internet continues to evolve, staying informed about these changes not only helps in leveraging the latest technologies but also ensures a faster, more secure user experience. For any developer or website owner, understanding and adopting HTTP/2 is not just beneficial; it's essential for keeping up with the modern web’s demands.

JavaScript Objects: A Deep Dive into Their Internal Representation JavaScript is one of the most widely used programming languages in the world, especially known for its role in web development. At the heart of JavaScript’s functionality are objects,

which are essentially collections of properties. Understanding how these objects are internally represented not only helps in optimizing code but also in debugging and maintaining it efficiently. This blog post explores the internal structure of JavaScript objects and how they operate behind the scenes. What are JavaScript Objects? In JavaScript, almost everything is an object.

From simple arrays to functions, and even complex date objects, all are treated as objects. At its simplest, a JavaScript object is a mapping between keys and values. Keys are strings (or Symbols), while values can be any data type, including other objects. let person = { name: "Alice", age: 25, greet: function() { console.log("Hello!"); } }; In the example above, person is an object with properties name, age, and greet. Internal Representation of Objects Internally, JavaScript engines like V8 (used in Chrome and Node.js), SpiderMonkey (Firefox), and JavaScriptCore (Safari) use various techniques to store and manage objects efficiently. While specifics can vary between different engines, the core ideas are often similar. Hidden Classes Despite being highly dynamic, JavaScript engines use an optimization called “hidden classes” (also known as “shapes” in some engines) to improve the performance of object property access. Hidden classes help in predicting the location of properties in memory,

making property access faster. When an object is created, it is associated with a hidden class that describes the layout of the object. If a new property is added to the object, the engine creates a new hidden class linked to the previous hidden class. This chain of hidden classes allows the engine to quickly find properties in objects. Inline Caching Inline caching is a technique used to speed up property access by caching the locations of object properties. When a property is accessed repeatedly, the engine does not have to perform a full lookup each time. Instead, it can refer to the cache, which holds the information about where the property value is stored. Property Storage Properties in JavaScript objects can be stored in two ways:

• In-object properties: These are stored directly within the object itself. Engines typically use this approach for properties that are added in the object constructor since their placement is predictable.

• Out-of-object properties: When objects have many properties or properties are added dynamically, they are stored in separate key-value storage to avoid resizing and moving the original object in memory. Optimizing Object Usage Understanding how objects are internally represented can guide developers in writing optimized code. Here are a few tips:

• Minimize dynamic property additions: Adding properties to an object after its creation can cause the hidden class to change, which may result in performance penalties due to hidden class transitions.

• Avoid deleting properties: Deleting properties can degrade performance because it changes the object’s shape, potentially causing more hidden class transitions.

• Instantiate objects consistently: If objects are instantiated in the same way, they share the same hidden class, which helps in optimizing property accesses. Conclusion Objects are a fundamental aspect of JavaScript, and their efficient management is crucial for high-performance applications. By leveraging concepts like hidden classes and inline caching, JavaScript engines are able to provide the dynamic flexibility that JavaScript is known for, while still ensuring that property accesses are fast. For developers, understanding these internal workings is not just about satisfying curiosity—it’s about writing better, faster JavaScript code. Whether you’re optimizing an existing application or starting a new project, keeping these principles in mind can lead to more efficient and effective codebases