

From Files to Streams: Revisiting Web History and Exploring Potentials for Future Prospects

Lucas Vogel
lucas.vogel2@tu-dresden.de
TU Dresden
Germany

Thomas Springer
thomas.springer@tu-dresden.de
TU Dresden
Germany

Matthias Wählisch
m.waehlich@tu-dresden.de
TU Dresden
Germany

ABSTRACT

Over the last 30 years, the World Wide Web has changed significantly. In this paper, we argue that common practices to prepare web pages for delivery conflict with many efforts to present content with minimal latency, one fundamental goal that pushed changes in the WWW. To bolster our arguments, we revisit reasons that led to changes of HTTP and compare them systematically with techniques to prepare web pages. We found that the structure of many web pages leverages features of HTTP/1.1 but hinders the use of recent HTTP features to present content quickly. To improve the situation in the future, we propose fine-grained content segmentation. This would allow to exploit streaming capabilities of recent HTTP versions and to render content as quickly as possible without changing underlying protocols or web browsers.

CCS CONCEPTS

• Information systems → World Wide Web; • Social and professional topics → History of computing.

ACM Reference Format:

Lucas Vogel, Thomas Springer, and Matthias Wählisch. 2024. From Files to Streams: Revisiting Web History and Exploring Potentials for Future Prospects. In *Companion Proceedings of the ACM Web Conference 2024 (WWW '24 Companion)*, May 13–17, 2024, Singapore, Singapore. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3589335.3652001>

1 INTRODUCTION

From its first implementation to now, the World Wide Web (WWW) has changed significantly. Initially, the Web was considered “a set of associations, and in a way, the Web is a representation of mankind’s knowledge” [7, Tim Berners Lee]. Thirty years later, “the Web is now a gigantic global software platform” [45, Michael Janiak]. The technologies and principles that enabled different applications and services on top of the WWW, however, stayed fundamentally the same.

The WWW is more than presenting a web page. It is a system that leverages multiple protocols on different layers to deliver (HTTP) and render (HTML, CSS, and JavaScript) different types of content supported by common tools. Many innovations that led to changes in each part were driven by creating content more efficiently and

delivering content faster—scalable content dissemination with minimal latency was and still is critical for successful web applications. These changes have been, so far, mainly considered independently of each other, even though they could achieve higher performance gains if they are considered together. One example is the lack of preparing web pages such that content is segmented into pieces to benefit most from the streaming capabilities of HTTP/3.

In this paper, we systematically analyze why specific changes have been made to help our research community identify room for further improvements. We reflect on the different design decisions of HTTP (§ 2) and common approaches to creating content (§ 3) based on historical documents and discussions with key stakeholders. We find that the lack of full advantages is a rather bad coincidence (§ 4) and derive opportunities for future improvements (§ 5).

2 ON THE HISTORY OF HTTP

HTTP has evolved significantly since its inception. Originally designed to deliver linked, mainly text-based documents, it has evolved into an optimized protocol that enables various applications, including complex real-time communication. Design decisions of HTTP not only reflect the needs of emerging web applications and services but also align with the state of the larger Internet ecosystem. For instance, when the first web server implementation was released in 1991 [17], HTTP was tailored to transfer linked files, and each web page consisted of relatively few content pieces. This design choice was inspired by FTP, a popular standard at the time for downloading files [62, 70]. HTTP underwent rapid changes as the Web expanded over the following years. Now, modern web pages consist of multiple content pieces, necessitating parallel delivery instead of downloading them sequentially to form the final page—calling for streams, supported in more recent HTTP versions.

HTTP/0.9. In 1991, Tim Berners-Lee designed HTTP “with simplicity in mind” [10, 33]. A client requested a web page via a GET request. The original proposal explicitly states that the server response is HTML [10]. The server then responded with ASCII characters representing the content of the resource, in this case, a file on the server [59]. The design lacked error handling, so clients had to examine the HTML output to determine if the data was received successfully [58].

HTML is based on SGML syntax, as ISO already standardized SGML in 1986 [41]. SGML was a file-based format, and to differentiate both files, the file ending was designed so it could be renamed from .sgml to .html [58]. From the beginning, HTML allows to separate structure from layout, differentiating it from SGML’s more generalized and integrated text processing approach [16, 41].

In summary, due to prior protocols such as FTP and existing file formats at the time, the Web was designed to work with files. The

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

WWW '24 Companion, May 13–17, 2024, Singapore, Singapore

© 2024 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-0172-6/24/05.

<https://doi.org/10.1145/3589335.3652001>

lack of standardization of HTTP and issues, such as the inability to recognize errors, required further protocol improvements, which will be discussed next.

HTTP/1.0. Between 1993 and 1997, the Web grew exponentially [31]. This growth motivated clearer documentation and some clean-ups. In May 1996, HTTP/1.0 was published in RFC 1945 [12], an informational document rather than a standard, collecting the common usage of existing implementations [33]. In HTTP/1.0, every request requires an individual TCP handshake, which is closed after the answer is transferred [12], see Figure 1. The increase in web pages led to a stronger focus on the presentation layer (see § 3). HTTP now supported Content-Types [12], allowing for different media to be transferred since the type of content was specified explicitly. This shift in perspective could have marked a turning point for the Web, enabling truly mixed content in a single request. Until today, a significant number of media types have been registered [61], maintained by IANA [40]. However, potentially because text-based implementations already existed, and with document formats such as images or CSS files, the file-based structure prevailed.

HTTP/1.1. HTTP/1.1, specified in 1997 in RFC 2068 [23] and later RFC 2616 [24], addressed a fundamental performance problem of HTTP/1.0 by introducing two features. First, instead of using individual TCP requests for every resource, which slowed down the transmission because resources (e.g., images) required a handshake as well, HTTP/1.1 allowed reusing one TCP connection for multiple resources (see Figure 1) based on the Keep-Alive mechanism. Second, HTTP/1.1 introduced request pipelining, where multiple requests could be sent via a single TCP connection, *i.e.*, a client can send requests in order without waiting for the responses [25].

Session reuse and request pipelining directly addressed protocol-level needs to present content faster. Using a single connection for multiple types of content, such as CSS or HTML, improved loading times, for example, by preventing the execution of the TCP slow start algorithm multiple times. However, pipelining was deactivated by default in a majority of modern HTTP clients [52] due to multiple problems such as buggy proxies and the complexity of implementation.

Starting in July 1997, the newly formed HTTP-NG Working Group proposed several ideas for a new generation of HTTP [34]. They used the concept of multiplexing data into one single stream. This concept would allow for faster transfer, as HTTP pipelining must wait until a response is fully finished before new data can be sent [52]. In 1999, the working group stopped and transferred the outcome to the IETF [34].

HTTP/2. In 2009, Google announced SPDY, a project aiming to improve web page loading speed by minimizing latency [8]. This approach also utilized multiplexing, with Google claiming up to a 55% reduction in page loading time over SSL [1]. Some of the inspiration was drawn from HTTP-NG [1]. As SPDY gained more traction, the HTTP Working Group specified the HTTP/2 protocol in 2012, inspired by SPDY [30, 55]. In May 2015, HTTP/2 was published as RFC 7540 [9]. Some major technical improvements include data compression, request prioritization, server push, and, most notably, multiplexing requests. They allow for transferring binary data in parallel in a different order based on prioritization, as shown in Figure 1. Multiplexing is a significant step because it

changes the original design of a file-based protocol to streams. On a protocol level, streams were the new way of transferring data on the Web. Despite speed improvements, streams were primarily used for transferring files, such as CSS, HTML, or JavaScript.

HTTP/3. Before HTTP/3, TCP was the preferred protocol for transmitting HTTP [9, 25]. However, TCP acknowledges every packet, causing delays if a packet is lost, as all streams are stopped. This issue is known as HOL-blocking (head-of-line blocking) [5, 19]. Google started working on QUIC in 2012, an alternative to TCP, TLS, and HTTP/2 [18]. QUIC is based on UDP and implements a larger part of the stack, allowing for faster protocol updates.

In 2018, the IETF decided that “HTTP/QUIC” should be named HTTP/3 [14, 39]. Later, QUIC was published as RFC 9000 [42] in 2021 and HTTP/3 as RFC 9114 [13] in 2022. HTTP/3 retains the stream-based approach but introduces new features, such as per-stream flow control. With the replacement of HTTP/2 by HTTP/3 and TLS and TCP by QUIC, HTTP has become a specialized protocol designed to transfer client content based on low-latency connection setup and better system handling of lost packets [13].

In summary, HTTP has changed over the last 30 years. Web content, however, has also changed, partly independently of HTTP, partly to cope with its limitations. We discuss these changes next.

3 WEB CONTENT CREATION HISTORY

The limitations of HTTP also influenced the creation of web content. In the context of this paper, the term “content” refers to code that is sent to the user and processed there to be displayed as a web page. The design choices for preparing code are sparsely documented compared to protocol changes.

Beginning of Scripts and Styles. In 1990, the Web Browser written by Tim Berners-Lee already included an editor for users to modify content themselves [11]. Subsequent browsers, such as the Line Mode Browser [26] or Viola [80], did not have a built-in editor. Since then, the creation and consumption of web content have diverged.

In October 1994, Håkon Wium Lie proposed Cascading HTML style sheets (CHSS) just three days before Netscape’s announcement [16, 47]. After collaborating with Bert Bos and significant debates at WWW conferences, the first version of Cascading Style Sheets (CSS1) was published in 1996 [16, 48]. Meanwhile, JavaScript emerged in 1995, when Netscape added scripting into their browser, realizing the need for a more dynamic web [54, 65]. This involved two strategies. First, Brendan Eich was hired to work on a Scheme at Netscape [22]. Secondly, Netscape collaborated with Sun (now part of Oracle) to provide Java Applets [54, 65]. After Eich wrote a prototype in 1995, it was named “LiveScript” by Netscape marketing but later renamed to JavaScript [49, 65]. JavaScript was, and still is, the prominent scripting language of the Web [49].

Early Development of Websites with JavaScript. The release of the first DOM specification [6] in 1998 set the starting point for extended JavaScript development, enabling dynamic web applications. According to various informal sources, developing JavaScript with multiple files (separated for an improved developer experience) often used concatenation to produce one output file, which could then be sent to the user [28, 43, 46]. These were individual, large global scripts [28]. One reason for this combination of

code, such as JavaScript or CSS, was loading time optimization. Since HTTP/1.0 required a new TCP connection for every request, combined with the various issues with HTTP/1.1 implementations meant that combining resources could generally improve performance [52]. Additionally, simultaneous connections to a single endpoint were generally limited to six TCP connections per individual origin (a host name and port) by browsers supporting HTTP/1.1 [32, 69]. Consequently, combining resources was necessary, or Domain Sharding was required where multiple subdomains are used to host a larger number of files [32]. Despite these restrictions, JavaScript and CSS grew significantly in popularity, partially due to the dotcom boom at the time [60]. Resource concatenation was therefore utilized to improve performance.

The Dynamic Web. Since 1993, CGI scripts [67] have been a standardized way of adding dynamic functionality to web pages by running scripts on the server. However, security concerns emerged, e.g., [36, 77]. Client-side code can avoid this issue by being executed in the browser. Therefore, tools such as Java Applets, Adobe Flash, and Shockwave gained popularity with the release of the Netscape Plug-in API in 1996 [4, 53].

In 2005, Garrett introduced “Ajax” [27, 64], a new way of loading content asynchronously. Before Ajax, presenting updated content under the same URL often required a reload of the web page. Given the connection overhead of HTTP/1.1, a complete reload of a page slowed down the interaction speed [27]. The Ajax concept paved the way for “Web 2.0.” [15, 64]. Partially to simplify the development and implementation of Ajax, Resig introduced jQuery [66] in 2006, which since then has been a popular library based on JavaScript. jQuery is still used by 77% of all websites [63].

With the increasing popularity of jQuery, prior approaches were less deployed. Java Applets were deprecated in 2017, and Shockwave and Adobe Flash in 2019 and 2020 [3, 81]. Libraries such as jQuery, Bootstrap, and Underscore simplify the developer experience [81] and their success demonstrates that the broad adoption of new technologies depends on an easy-to-use development. The downside is that such libraries can increase loading times as they must be loaded completely before use. By default, this is render-blocking, preventing the browser from displaying the page. Furthermore, the use of libraries and an increasing amount of JavaScript also result in web pages continuously growing in size. Even though full page reloads are no longer necessary, the overhead of loading large render-blocking libraries can still impact loading speed, even with improved connection times and transfer speeds of HTTP/2 or HTTP/3.

Start of Node.js and Bundles. In order to organize the increasing amount of JavaScript code, developers separated code into individual files, which were then concatenated and used as global scripts [28, 43, 46].

That changed when Node.js was introduced by Ryan Dahl in 2009 [20]. Node.js allowed JavaScript to run on a server and shipped with a package manager called *npm*. Npm allowed developers to install other packages and import (“require”) code when needed. In contrast, JavaScript did not have a native module system until 2015 [28, 57]. This module system of Node.js enabled significantly larger code bases. Node.js, however, was not intended to be used in a browser. This changed when Browserify emerged in 2010 and

became popular in 2013 [29, 35]. Browserify allows using the CommonJS “require” syntax, allowing developers to use and require npm packages while creating websites [28]. However, browsers do not understand the “require” keyword, so Browserify transforms all code into a browser-compatible version. All dependencies are resolved and combined into a single file, called a *bundle* [28]. Even though the primary goal of Browserify was different, it is often cited as being the first influential bundler [28, 43, 46]. In summary, bundling tools allow combining multiple code files. Before the reduction of the individual request overhead starting at HTTP/1.1, bundling improved loading times, as fewer requests were required to fetch all necessary data for page load while also improving the developer experience.

Webpack. While Browserify could compile and bundle code, the purpose-built compiler and bundler Webpack started development in 2012 and gained popularity in \approx 2015 [28, 29, 68]. In contrast to Browserify, it was also designed to bundle other resources, such as CSS or images [2]. This enabled the emergence of popular modern frontend frameworks such as React or Angular, both of which use Webpack. Especially with React gaining popularity, Webpack overtook Browserify in popularity [29]. Splitting code is possible with Webpack as a manual opt-in feature but is disabled by default [78]. Webpack was designed for large projects [79]. However, it was stated that for significantly large projects, it is suboptimal to load the bundle as one large file [79].

Therefore, to this day, popular frontend frameworks and bundlers like Webpack will produce large, individual files that, by default, are render-blocking [29].

4 HISTORICAL OVERLAP OF PROTOCOL CHANGES AND CONTENT CREATION

Contextualizing HTTP and content creation into historical context, it becomes evident that changes to HTTP were misaligned in time with changes to languages and content creation frameworks, as shown in Figure 1. Until 2015, HTTP/1.1 (and predecessors) on top of TCP led to an environment where larger but fewer requests improved performance due to the recurring overhead of connection establishment per request. To overcome the performance drawbacks of HTTP and TCP, bundlers such as Webpack gained popularity because using large individual files (bundles) fit the needs of developers and reduced performance penalties [28, 29, 57]. The major drawback of bundling is code efficiency. If a project uses JavaScript or CSS frameworks, the entire code base of the framework is usually fully included in a bundle. This increases the total size of a web page and, if loaded in the default render-blocking way, slows down the loading time of web pages. A recent study revealed that the majority of the most popular web pages still have this issue [73].

In 2015, with the advent of HTTP/2, the system changed. Now, a single connection can multiplex various resources with different priorities, allowing for the asynchronous transfer of smaller files [9]. This change is in direct contrast to the concept of bundling and the emergence of libraries, such as jQuery. Hofman, a Web performance expert, called bundling an “anti-pattern” in HTTP/2 [37]. In principle, the content of a web page could be adapted again to benefit from the features of HTTP/2. We describe one possible approach next.

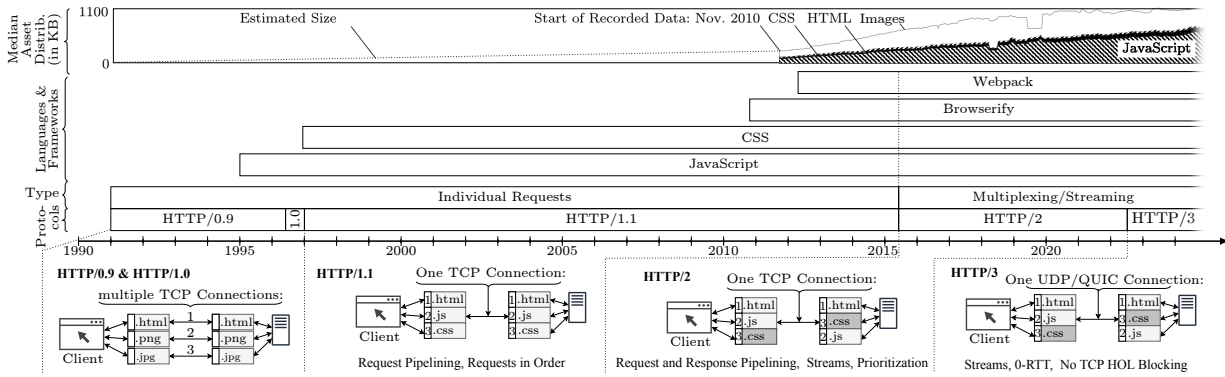


Figure 1: Historical comparison of Web technologies, including protocols and content creation and presentation features such as bundlers, JavaScript, and CSS. A major change occurred when HTTP/2 was released (dotted line). Protocol deprecations are not shown. Source for asset distribution data: HTTP Archive [38]. Images sizes are not stacked, as images are not render-blocking.

5 RE-IMAGINING THE CONTENT STRUCTURE BASED ON HTTP/3

Most of the most popular web pages are still based on large, render-blocking bundles, see § 4. This might be due to the lack of alternatives. In this section, we introduce one potential option to align content creation with the potentials of emerging Web standards.

Basic Concept. While bundling uses the central concept of combining resources, the next generation of web content should do the opposite: split the content as much as possible. The main reason for this design choice is code efficiency. If code is split into small, individual pieces, then it can be loaded asynchronously on demand. If planned correctly, the transferred code only contains data required for rendering, which can significantly improve loading times. This also matches recommendations by Hofman [37], who suggested splitting as a solution. For example, jQuery or Bootstrap include a large set of functions. Many web pages only need parts of these functions, resulting in loading a large amount of unused code. Splitting is relevant for CSS and JavaScript, as both can be loaded externally and both can be render-blocking. Adding to Hofman, the next generation of code processors should also take advantage of the prevailing streaming technology available in HTTP/2 and HTTP/3. This moves loading from an all-or-nothing approach to the behavior of loading a web page as an ongoing process over time. Such a concept has been highlighted in a study by Jahromi *et al.* [44], stating that users do not wait for a complete result after an interaction. Streaming can be used as a major advantage, presenting the user with a continuously rendered version as the page loads to enable subsequent actions. This, combined with the maximum splitting approach, has the additional positive effect that a First Contentful Paint (FCP) of a page can be fast, even in challenging network conditions. Three main challenges remain to provide a similar, fully automated system like bundlers:

Challenge 1: Content usage detection. The first step is the automatic generation of information about the type of code used and where to split the code. For CSS, techniques such as Critical [56] exist but are limited to the content above-the-fold. Other solutions include the Essential framework, which addresses the issues of Critical [75]. For JavaScript, approaches like tree shaking are

available and already in use by bundlers today [21]. They remove dead code but identify dead code only on file or function level and are bound to specific frameworks. One approach to improve the situation is resumeability, where JavaScript code is loaded on demand without breaking the web page. This approach requires less developer effort and is framework-independent [72, 76].

Challenge 2: Content usage location and order. After detecting and splitting the necessary code, it needs to be ordered and interleaved so that only neighboring pieces of HTML, CSS, and JavaScript depend on each other and, thus, only minimally block rendering. For CSS, this involves matching selectors with the DOM of a document. For JavaScript, this is still an open challenge, but promising automation approaches have been proposed [50, 76].

Challenge 3: Streaming a web page. Lastly, the processed data needs to be transferred by utilizing the streaming capability of HTTP/2 and HTTP/3. Frameworks such as Turbo [71] and Marco [51] demonstrated that streaming of web page chunks is feasible and ready for production. A demo of a fully streamed page was presented at The Web Conference 2023 [74].

It is worth noting that streaming is just one possible solution but—looking at the continuous trend of ever-increasing web pages—it appears a promising approach [38]. Prior work bolsters this argument, showing that streamed web pages significantly improve loading times due to fine-grained page chunks and selecting only required code [74].

6 CONCLUSION

In this paper, we presented a historical analysis of two major building blocks crucial for a fast World Wide Web: the content delivery layer and the content presentation layer. We found that the design of both building blocks was mostly considered independently, despite the potential benefits from joint design decisions. To improve the situation, we sketched content segmentation aiming for minimal render blocking. This proposal is transparent to content creators and enables the use of streaming features of underlying protocols. We hope that the insights of this paper do not only serve as a reminder of the reasons why Web technologies evolved in specific directions but also as path forward.

REFERENCES

- [1] 2023. *SPDY: An experimental protocol for a faster web*. White Paper. Chromium. <https://www.chromium.org/spdy/spdy-whitepaper> [Online; accessed 10. Oct. 2023].
- [2] 2023. *webpack*. <https://webpack.js.org> [Online; accessed 6. Nov. 2023].
- [3] Adobe. 2024. *End of Life (EOL) for Adobe Shockwave*. Technical Report. <https://web.archive.org/web/20220308095014/https://helpx.adobe.com/shockwave/shockwave-end-of-life-faq.html> [Online; accessed 29. Feb. 2024].
- [4] Adobe Systems Incorporated. 2024. *Adobe - Products : Adobe Shockwave Player*. Technical Report. <https://web.archive.org/web/20120110005639/http://www.adobe.com/products/shockwaveplayer> [Online; accessed 29. Feb. 2024].
- [5] M. Allman, V. Paxson, and E. Blanton. 2009. *TCP Congestion Control*. RFC 5681. IETF. <https://doi.org/10.17487/RFC5681>
- [6] Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Arnaud Le Hors, Gavin Nicol, Jonathan Robie, Chris Sutor, Robert Wilson, and Lauren (Eds.) Wood. 1998. *Document Object Model (DOM) Level 1 Specification*. W3C recommendation REC-DOM-Level-1-19981001. W3C. <https://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/>
- [7] Frank Bajak. 1993. A World of Data Coming to Your Fintertips. Wiring the Planet. Part 2. *San Francisco Examiner* (May 1993), D1. https://sfexaminer.newspapers.com/browse/the-san-francisco-examiner_9317/1993/05/31/
- [8] Mike Belshe and Roberto Peon. 2009. *A 2x Faster Web*. Technical Report. Chromium. <https://blog.chromium.org/2009/11/2x-faster-web.html> [Online; accessed 10. Oct. 2023].
- [9] M. Belshe, R. Peon, and M. Thomson. 2015. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. IETF. <https://doi.org/10.17487/RFC7540>
- [10] Tim Berners-Lee. 1999. *The Original HTTP as defined in 1991*. Technical Report. W3C. <https://www.w3.org/Protocols/HTTP/AsImplemented.html> [Online; accessed 14. Aug. 2023].
- [11] Tim Berners-Lee. 2017. *Tim Berners-Lee: WorldWideWeb, the first Web client*. Technical Report. <https://www.w3.org/People/Berners-Lee/WorldWideWeb.html> [Online; accessed 2. Oct. 2023].
- [12] T. Berners-Lee, R. Fielding, and H. Frystyk. 1996. *Hypertext Transfer Protocol - HTTP/1.0*. RFC 1945. IETF. <https://doi.org/10.17487/RFC1945>
- [13] M. Bishop. 2022. *HTTP/3*. RFC 9114. IETF. <https://doi.org/10.17487/RFC9114>
- [14] Mike Bishop. 2023. *HTTP/QUIC. What's in a Name?*. In *IETF 103 Proceedings*. IETF. <https://datatracker.ietf.org/meeting/103/materials/slides-103-httpbis-httpquic-02>
- [15] Grant Blank and Bianca C Reisdorf. 2012. The participatory web: A user perspective on Web 2.0. *Information, Communication & Society* 15, 4 (2012), 537–554.
- [16] Bert Bos. 2017. *A brief history of CSS until 2016*. Technical Report. <https://www.w3.org/Style/CSS20/history.html> [Online; accessed 2. Oct. 2023].
- [17] CERN. 2016. *Change History of W3C httpd*. Technical Report. <https://www.w3.org/Daemon/Features.html> [Online; accessed 14. Aug. 2023].
- [18] Chromium. 2023. *QUIC, a multiplexed transport over UDP*. Technical Report. <https://www.chromium.org/quic> [Online; accessed 16. Oct. 2023].
- [19] Cloudflare. 2023. *What is HTTP/3?* Technical Report. <https://www.cloudflare.com/learning/performance/what-is-http3> [Online; accessed 16. Oct. 2023].
- [20] Ryan Dahl. 2009. *node-v0.x-archive: Major refactoring: program name now 'node'*. Technical Report. <https://github.com/nodejs/node-v0.x-archive/commit/19478ed4b14263c489e872156ca55ff16a07ebe0> [Online; accessed 6. Nov. 2023].
- [21] MDN Web Docs. 2024. *Tree shaking - MDN Web Docs Glossary: Definitions of Web-related terms | MDN*. Technical Report. https://developer.mozilla.org/en-US/docs/Glossary/Tree_shaking [Online; accessed 4. Jan. 2024].
- [22] Brendan Eich. 2023. *Popularity - Brendan Eich*. Technical Report. <https://brendaneich.com/2008/04/popularity> [Online; accessed 30. Oct. 2023].
- [23] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. 1997. *Hypertext Transfer Protocol - HTTP/1.1*. RFC 2068. IETF. <https://doi.org/10.17487/RFC2068>
- [24] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. 1999. *Hypertext Transfer Protocol - HTTP/1.1*. RFC 2616. IETF. <https://doi.org/10.17487/RFC2616>
- [25] R. Fielding and J. Reschke. 2014. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. IETF. <https://doi.org/10.17487/RFC7230>
- [26] Henrik Frystyk Nielsen. 1998. *Command Line Syntax for the Line Mode browser*. Technical Report. <https://www.w3.org/LineMode/User/CommandLine.html> [Online; accessed 30. Oct. 2023].
- [27] Jesse James Garrett. 2005. *Ajax: A New Approach to Web Applications | Adaptive Path*. Technical Report. <https://web.archive.org/web/20190119022701/https://adaptivepath.org/ideas/ajax-new-approach-web-applications> [Online; accessed 29. Feb. 2024].
- [28] Devlin Glasman. 2023. *A History of JavaScript Modules and Bundling. For the Post-ES6 Developer | 8th Light*. Technical Report. <https://8thlight.com/insights/a-history-of-javascript-modules-and-bundling-for-the-post-es6-developer> [Online; accessed 30. Oct. 2023].
- [29] Google. 2023. *Google Trends - Browserify, Webpack, React, Angular*. Technical Report. <https://trends.google.de/trends/explore?date=all&q=Browserify,Webpack,React%20JS,Angular%20JS> [Online; accessed 6. Nov. 2023].
- [30] Google. 2023. *Google Trends - SPDY*. <https://trends.google.de/trends/explore?date=2009-01-01%202023-10-10&q=SPDY> [Online; accessed 10. Oct. 2023].
- [31] Mathew Gray. 1997. *Web Growth Summary*. Technical Report. <https://stuff.mit.edu/people/mkgray/net/web-growth-summary.html> [Online; accessed 02. Oct. 2023].
- [32] Ilya Grigorik. 2015. *HTTP: HTTP/1.X - High Performance Browser Networking* (O'Reilly). *High Performance Browser Networking* (Nov. 2015). <https://hpbn.co/http1x/#domain-sharding>
- [33] Ilya Grigorik. 2015. *HTTP protocols*. O'Reilly Media, Inc., Sebastopol, CA, USA. <https://www.oreilly.com/library/view/http-protocols/9781492030478/ch01.html>
- [34] W3C HTTP-NG Working Group. 1999. (Proposed) HTTP-NG Working Group. <https://www.w3.org/Protocols/HTTP-NG> [Online; accessed 10. Oct. 2023].
- [35] James Halliday. 2010. *browserify - Initial Commit*. Git Commit. GitHub. <https://github.com/browserify/browserify/commit/b0363ae3d5749b3f7e722b21c65484fd6f34ac3> [Online; accessed 13. Nov. 2023].
- [36] Shawn V. Hernan. 2001. *phf CGI Script fails to guard against newline characters*. Vulnerability Note VU#20276. CERT Coordination Center. <https://www.kb.cert.org/vuls/id/20276> [Online; accessed 29. Feb. 2024].
- [37] Erwin Hofman. 2022. *The 2 main performance debts of HTTP/1*. <https://www.erwinhofman.com/blog/two-main-performance-debts-of-http1> [Online; accessed 3. Jan. 2024].
- [38] HTTP Archive. 2024. *HTTP Archive: Page Weight*. Technical Report. <https://httparchive.org/reports/page-weight> [Online; accessed 28. Feb. 2024].
- [39] HTTP (httpbis) Working Group. 2023. *Working Group Minutes. QUIC and HTTP*. In *IETF 103 Proceedings*. IETF. <https://datatracker.ietf.org/meeting/103/materials/minutes-103-httpbis-00>
- [40] IANA. 2023. *Media Types*. <https://www.iana.org/assignments/media-types/media-types.xhtml> [Online; accessed 2. Oct. 2023].
- [41] ISO. 2023. *Information processing - Text and office systems - Standard Generalized Markup Language (SGML)*. Standard ISO 8879:1986. ISO. <https://www.iso.org/obp/ui/en/#iso:std:iso:8879:ed-1:v1:en> [Online; accessed 2. Oct. 2023].
- [42] J. Iyengar and M. Thomson. 2021. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. IETF. <https://doi.org/10.17487/RFC9000>
- [43] Jacob Jackson. 2022. *State of the Web: Bundlers & Build Tools*. Technical Report. <https://byteofdev.com/posts/bundlers> [Online; accessed 30. Oct. 2023].
- [44] Hamed Z Jahromi, Declan T Delaney, and Andrew Hines. 2020. *Beyond first impressions: Estimating quality of experience for interactive web applications*. *IEEE Access* 8 (2020), 47741–47755.
- [45] Michael Janiak. 2023. *Why Modern Web Design Is No More: The New Era of 'Product Design'*. *Website Services, Inc.* (May 2023). <https://www.websitemagazine.com/web-design/why-modern-web-design-is-no-more-the-new-era-of-product-design>
- [46] Nolan Lawson. 2017. *A brief and incomplete history of JavaScript bundlers*. Technical Report. <https://nolanlawson.com/2017/05/22/a-brief-and-incomplete-history-of-javascript-bundlers> [Online; accessed 30. Oct. 2023].
- [47] Håkon Wium Lie. 1995. *Cascading HTML Style Sheets - A Proposal*. individual proposal. W3C. <https://www.w3.org/People/howcome/p/cascade.html> [Online; accessed 2. Oct. 2023].
- [48] Håkon Wium Lie and Bert Bos. 1996. *Cascading style sheets, level 1*. (Sept. 1996).
- [49] Richard Macmanus. 2020. *1995: The Birth of JavaScript*. Technical Report. Web Development History. <https://webdevelopmenthistory.com/1995-the-birth-of-javascript> [Online; accessed 30. Oct. 2023].
- [50] Shaghayegh Mardani, Mayank Singh, and Ravi Netravali. 2020. *Fawkes: Faster Mobile Page Loads via App-Inspired Static Templating*. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 879–894. <https://www.usenix.org/conference/nsdi20/presentation/mardani>
- [51] Marko. 2023. *Marko*. <https://markojs.com> [Online; accessed 9. Jan. 2024].
- [52] MDN. 2023. *Connection management in HTTP/1.x*. MDN Web Docs. Mozilla. https://developer.mozilla.org/en-US/docs/Web/HTTP/Connection_management_in_HTTP_1.x#http_pipelining [Online; accessed 10. Oct. 2023].
- [53] Naif Mehanna and Walter Rudametkin. 2023. *Caught in the Game: On the History and Evolution of Web Browser Gaming*. In *Companion Proceedings of The ACM Web Conference 2023*. ACM, New York, NY, USA, 601–609. <https://doi.org/10.1145/3543873.3585572>
- [54] Netscape Communications Corporation. 2023. *Introducing Netscape Navigator 2.0*. Technical Report. https://web.archive.org/web/19961026223230/http://www3.netscape.com/comprod/products/navigator/version_2.0/index.html [Online; accessed 30. Oct. 2023].
- [55] Mark Nottingham. 2012. *Rechartering HTTPbis from Mark Nottingham on 2012-01-24 (ietf-http-wg@w3.org from January to March 2012)*. <https://lists.w3.org/Archives/Public/ietf-http-wg/2012JanMar/0098.html> [Online; accessed 10. Oct. 2023].
- [56] Addy Osmani. 2024. *critical*. Git repository. GitHub. <https://github.com/addyosmani/critical> [Online; accessed 4. Jan. 2024].
- [57] Addy Osmani and Mathias Bynens. 2018. *JavaScript modules · V8*. Technical Report. <https://v8.dev/features/modules> [Online; accessed 6. Nov. 2023].

- [58] Oxford Brookes University. 2002. *History of the Web*. Technical Report. <https://web.archive.org/web/20111226012516/http://www.weblab.isti.cnr.it/education/documents/origins.pdf> [Online; accessed 14. Aug. 2023].
- [59] Steven Pemberton. 2022. On the Design of the URL. <https://homepages.cwi.nl/~steven/Talks/2020/10-09-urls/design.html> [Online; accessed 14. Aug. 2023].
- [60] Political Calculations . 2010. Here's Why The Dot Com Bubble Began And Why It Popped - Business Insider. *Business Insider* (Dec. 2010). <https://web.archive.org/web/20200406151705/https://www.businessinsider.com/heres-why-the-dot-com-bubble-began-and-why-it-popped-2010-12>
- [61] J. Postel. 1994. *Media Type Registration Procedure*. RFC 1590. IETF. <https://doi.org/10.17487/RFC1590>
- [62] J. Postel and J. Reynolds. 1985. *File Transfer Protocol*. RFC 959. IETF. <https://doi.org/10.17487/RFC0959>
- [63] Q-Success. 2024. *Usage Statistics and Market Share of JavaScript Libraries for Websites, February 2024*. Technical Report. https://w3techs.com/technologies/overview/javascript_library [Online; accessed 29. Feb. 2024].
- [64] Mike Ralphson. 2019. A brief history of Web APIs. *DEV Community* (Jan. 2019). <https://dev.to/mikeralphson/a-brief-history-of-web-apis-47k4>
- [65] Axel Rauschmayer. 2023. *Chapter 4. How JavaScript Was Created*. Technical Report. <https://web.archive.org/web/20221009110102/https://speakingjs.com/es5/ch04.html> [Online; accessed 30. Oct. 2023].
- [66] John Resig. 2006. *BarCampNYC Wrap-up*. Blog Post. <https://johnresig.com/blog/barcampnyc-wrap-up/> [Online; accessed 11. Mar. 2024].
- [67] D. Robinson and K. Coar. 2004. *The Common Gateway Interface (CGI) Version 1.1*. RFC 3875. IETF. <https://doi.org/10.17487/RFC3875>
- [68] sokra. 2012. *Webpack - Initial Commit*. Git Commit. GitHub. <https://github.com/webpack/webpack/commit/2e1460036c5349551da86c582006c7787c56c543> [Online; accessed 13. Nov. 2023].
- [69] Daniel Stenberg. 2016. *HTTP/2 connection coalescing* | *daniel.haxx.se*. Technical Report. <https://daniel.haxx.se/blog/2016/08/18/http2-connection-coalescing> [Online; accessed 17. Jan. 2024].
- [70] Daniel Stenberg. 2021. *FTP vs HTTP*. <https://daniel.haxx.se/docs/ftp-vs-http.html> [Online; accessed 14. Aug. 2023].
- [71] Turbo. 2024. Turbo: The speed of a single-page web application without having to write any JavaScript. <https://turbo.hotwired.dev> [Online; accessed 9. Jan. 2024].
- [72] Juho Vepsäläinen, Miško Hevery, and Petri Vuorimaa. 2024. Resumability-a New Primitive for Developing Web Applications. *IEEE Access* (2024).
- [73] Lucas Vogel and Thomas Springer. 2022. An in-depth analysis of web page structure and efficiency with focus on optimization potential for initial page load. In *International Conference on Web Engineering*. Springer, Switzerland, 101–116.
- [74] Lucas Vogel and Thomas Springer. 2023. How Streaming Can Improve the World (Wide Web). In *Companion Proceedings of the ACM Web Conference 2023*. 140–143.
- [75] Lucas Vogel and Thomas Springer. 2023. Speed Up the Web with Universal CSS Rendering. In *International Conference on Web Engineering*. Springer, 191–205.
- [76] Lucas Vogel and Thomas Springer. 2023. Waiter and AUTRATAC: Don't Throw It Away, Just Delay!. In *International Conference on Web Engineering*. Springer, 278–292.
- [77] Bernhard Wagner. 1998. Controlling Cgi Programs. *SIGOPS Oper. Syst. Rev.* 32, 4 (Oct. 1998), 40–46. <https://doi.org/10.1145/302350.302360>
- [78] Webpack Contributors. 2017. *code splitting*. Documentation. GitHub. <https://github.com/webpack/docs/wiki/code-splitting> [Online; accessed 6. Nov. 2023].
- [79] Webpack Contributors. 2017. *What is webpack*. Documentation. GitHub. <https://github.com/webpack/docs/wiki/what-is-webpack> [Online; accessed 6. Nov. 2023].
- [80] Pei-Yuan Wei. 1992. Overview, ViolaWWW. <https://www.w3.org/History/19921103-hypertext/hypertext/WWW/Viola/violaWWWAbout.html> [Online; accessed 30. Oct. 2023].
- [81] Richard York. 2011. *Beginning JavaScript and CSS development with jQuery*. John Wiley & Sons.