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.waehlisch@tu-dresden.de
TU Dresden
Germany

ABSTRACT

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.

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, 4 pages. https://doi.org/10.1145/3589335.3652001

1 INTRODUCTION

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"* [2, Tim Berners Lee]. Thirty years later, *"the Web is now a gigantic global software platform"* [24, Michael Janiak]. 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 We reflect on the different design decisions of HTTP (§ 2) and common approaches to creating content (§ 3) based on historical documents and discussions with key stakeholders.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).

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

ACM ISBN 979-8-4007-0172-6/24/05.
https://doi.org/10.1145/3589335.3652001

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 [9], 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 (RFC 959). 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" [4]. A client requested a web page via a GET request. The original proposal explicitly states that the server response is HTML [4]. The server then responded with ASCII characters representing the content of the resource, in this case, a file on the server [33]. The design lacked error handling. Clients had to examine the HTML output to determine if the data was correct.

HTML is based on SGML syntax, as ISO already standardized SGML in 1986. 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`. From the beginning, HTML allows to separate structure from layout, differentiating it from SGML's more generalized and integrated text processing approach [8]. 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 lack of standardization of HTTP and issues, such as the inability to recognize errors, required further protocol improvements.

HTTP/1.0. Between 1993 and 1997, the Web grew exponentially [16]. This growth motivated clearer documentation and some clean-ups. In May 1996, HTTP/1.0 was published in RFC 1945, an informational document rather than a standard, collecting the common usage of existing implementations. In HTTP/1.0 , every request requires an individual TCP handshake, which is closed after the answer is transferred, see Figure 1. The increase in web pages led to a stronger focus on the presentation layer (see § 3). HTTP now supported Content-Types, 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, maintained by IANA. 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 and later RFC 2616, 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.

Session reuse and request pipelining directly addressed protocol-level needs to present content faster. Using a single connection for multiple types of content, *e.g.,* CSS or HTML, improved loading times, for example, by preventing the execution of TCP slow start multiple times. However, pipelining was deactivated by default in a majority of modern HTTP clients [29] 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 [18]. They used the concept of multiplexing data into one single stream. This would allow for faster transfer, as HTTP pipelining must wait until a response is fully finished before new data can be sent [29]. In 1999, the outcome was transferred to the IETF [18].

HTTP/2. In 2009, Google announced SPDY, a project aiming to improve web page loading speed by minimizing latency [3]. This approach also utilized multiplexing, with Google claiming up to a 55% reduction in page loading time over SSL. Some of the inspiration was drawn from HTTP-NG. As SPDY gained more traction, the HTTP Working Group specified the HTTP/2 protocol in 2012, inspired by SPDY [30]. In May 2015, HTTP/2 was published as RFC 7540. 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. 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). Google started working on QUIC in 2012, an alternative to TCP, TLS, and HTTP/2. 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 [6, 21]. Later, QUIC was published as RFC 9000 in 2021 and HTTP/3 as RFC 9114 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.

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 [5]. Subsequent browsers, such as the Line Mode Browser or Viola, 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 [8, 25]. After collaborating with Bert Bos and significant debates at WWW conferences, the first version of Cascading Style Sheets (CSS1) was published in 1996 [8, 26]. Meanwhile, JavaScript emerged in 1995, when Netscape added scripting into their browser, realizing the need for a more dynamic Web. This involved two strategies. First, Brendan Eich was hired to work on a Scheme at Netscape [12]. Secondly, Netscape collaborated with Sun (now part of Oracle) to provide Java Applets. After Eich wrote a prototype in 1995, it was named "LiveScript" by Netscape marketing but later renamed to JavaScript [27]. JavaScript was, and still is, the prominent scripting language of the Web.

Early Development of Websites with JavaScript. The release of the first DOM specification 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 [14, 22]. These were individual, large global scripts [14]. 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 [29]. 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 [17]. Consequently, combining resources was necessary, or Domain Sharding was required where multiple subdomains are used to host a larger number of files [17]. Despite these restrictions, JavaScript and CSS grew significantly in popularity, partially due to the dotcom boom at the time. Resource concatenation was therefore utilized to improve performance.

The Dynamic Web. Since 1993, CGI (RFC 3875) has been a standardized way of adding dynamic functionality to web pages by running scripts on servers. However, security concerns emerged [40]. Client-side code executed in the browser can avoid this issue. Therefore, tools such as Java Applets, Adobe Flash, and Shockwave gained popularity when the Netscape Plug-in API was released in 1996.

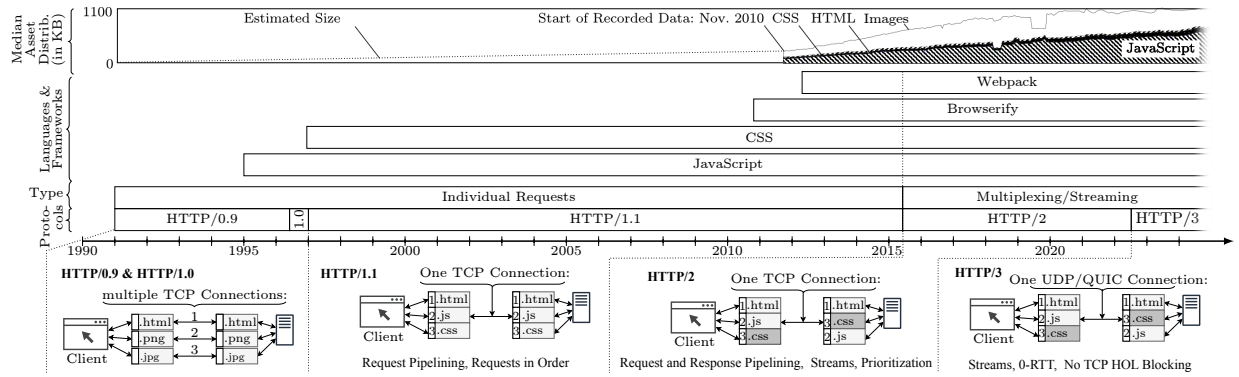

Figure 1: Historical comparison of Web technologies, including protocols and content creation and presentation features such as bundlers, JavaScript, and CSS. A major change occurred when HTTP/2 was released (dotted line). Protocol deprecations are not shown. Source for asset distribution data: HTTP Archive [20]. Images sizes are not stacked, as images are not render-blocking.

In 2005, Garrett introduced "Ajax" [13], 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 [13]. The Ajax concept paved the way for "Web 2.0." [7]. Partially to simplify the development and implementation of Ajax, Resig introduced jQuery [35] in 2006, which since then has been a popular library based on JavaScript. jQuery is still used by 77% of all websites [34].

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. Libraries such as jQuery, Bootstrap, and Underscore simplify the developer experience 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 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 [14, 22]. That changed when Node.js was introduced by Ryan Dahl in 2009 [10]. 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 [14, 32]. 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 [15]. Browserify allows using the CommonJS "require" syntax, allowing developers to use and require npm packages while creating websites [14]. 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* [14]. Even though the primary goal of Browserify was different, it is often cited as being the first influential bundler [14, 22]. In summary, bundling tools allow combining multiple code files. Before reducing the overhead of individual requests started in HTTP/1.1, bundling improved loading, as fewer requests were required to fetch necessary data for page load while also improving developer experience.

Webpack. While Browserify could compile and bundle code, the purpose-built compiler and bundler Webpack [1], started in 2012, gained popularity in ≈2015 [14, 15]. In contrast to Browserify, it was also designed to bundle other resources, such as CSS or images. 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 became more popular than Browserify [15]. Splitting code is possible with Webpack as a manual opt-in feature but is disabled by default [41]. Webpack was designed for large projects, but for significantly large projects it is not advised to load the bundle as one large file. Therefore, popular frontend frameworks and bundlers like Webpack will produce large, individual files that, by default, are render-blocking [15].

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 [14, 15, 32]. 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 [37].

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. 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 [19]. In principle, the content of a web page could be adapted again to benefit from the features of HTTP/2.

5 NEW CONTENT STRUCTURE AND HTTP/3

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 [19]. 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. Splitting is relevant for CSS and JavaScript, as both can be loaded externally and both can be render-blocking. The next generation of code processors should 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 studied [23], 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 [31] exist but are limited to the content above-the-fold. Other solutions include the Essential framework, which addresses the issues of Critical [39]. For JavaScript, approaches like tree shaking are available and already in use by bundlers today [11]. 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.

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 [28].

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. Streaming of web page chunks [36] as well as complete web pages [38] is feasible.

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 [20] to reduce loading times [38].

REFERENCES

[1] 2023. webpack. https://webpack.js.org
[2] Frank Bajak. 1993. A World of Data Coming to Your Fingertips. Wiring the Planet. Part 2. *San Francisco Examiner* (May 1993), D1. https://sfexaminer.newspapers.com/browse/the-san-francisco-examiner_9317/1993/05/31/
[3] Mike Belshe and Roberto Peon. 2009. *A 2x Faster Web*. Chromium. https://blog.chromium.org/2009/11/2x-faster-web.html
[4] Tim Berners-Lee. 1999. *The Original HTTP as defined in 1991*. W3C. https://www.w3.org/Protocols/HTTP/AsImplemented.html
[5] Tim Berners-Lee. 2017. *Tim Berners-Lee: WorldWideWeb, the first Web client*. https://www.w3.org/People/Berners-Lee/WorldWideWeb.html
[6] Mike Bishop. 2023. HTTP/QUIC. What's in a Name?. In *IETF 103 Proc.* IETF. https://datatracker.ietf.org/meeting/103/materials/slides-103-httpbis-httpquic-02
[7] 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.
[8] Bert Bos. 2017. *A brief history of CSS until 2016*. https://www.w3.org/Style/CSS20/history.html
[9] CERN. 2016. *Change History of W3C httpd*. www.w3.org/Daemon/Features.html
[10] Ryan Dahl. 2009. *node-v0.x-archive*. https://github.com/nodejs/node-v0.x-archive/commit/19478ed4b14263c489e872156ca55ff16a07ebe0
[11] MDN Web Docs. 2024. *Tree shaking*. https://developer.mozilla.org/en-US/docs/Glossary/Tree_shaking
[12] Brendan Eich. 2023. *Popularity*. https://brendaneich.com/2008/04/popularity
[13] J. J. Garrett. 2005. *Ajax*. https://web.archive.org/web/20190119022701/https://adaptivepath.org/ideas/ajax-new-approach-web-applications
[14] D. Glasman. https://8thlight.com/insights/a-history-of-javascript-modules-and-bundling-for-the-post-es6-developer
[15] Google. 2023. *Trends - Browserify, Webpack, React, Angular*. https://trends.google.de/trends/explore?date=all&q=Browserify,Webpack,React%20JS,Angular%20JS
[16] Mathew Gray. 1997. *Web Growth Summary*. https://stuff.mit.edu/people/mkgray/net/web-growth-summary.html
[17] Ilya Grigorik. 2015. HTTP: HTTP/1.X. *High Performance Browser Networking* (Nov. 2015). https://hpbn.co/http1x/#domain-sharding
[18] W3C HTTP-NG Working Group. 1999. https://www.w3.org/Protocols/HTTP-NG
[19] Erwin Hofman. 2022. The 2 main performance debts of HTTP/1. https://www.erwinhofman.com/blog/two-main-performance-debts-of-http1
[20] HTTP Archive. 2024. *HTTP Archive: Page Weight*. Technical Report. https://httparchive.org/reports/page-weight
[21] HTTP (httpbis) Working Group. 2023. Minutes. QUIC and HTTP. In *IETF 103 Proc.* IETF. https://datatracker.ietf.org/meeting/103/materials/minutes-103-httpbis-00
[22] J. Jackson. 2022. *State of the Web*. https://byteofdev.com/posts/bundlers
[23] Hamed Z Jahromi, *et al.* 2020. Beyond first impressions: Estimating quality of experience for interactive web applications. *IEEE Access* 8 (2020), 47741–47755.
[24] Michael Janiak. 2023. Why Modern Web Design Is No More: The New Era of 'Product Design'. https://www.websitemagazine.com/web-design/why-modern-web-design-is-no-more-the-new-era-of-product-design
[25] Håkon Wium Lie. 1995. *Cascading HTML Style Sheets – A Proposal*. individual proposal. W3C. https://www.w3.org/People/howcome/p/cascade.html
[26] Håkon Wium Lie and Bert Bos. 1996. Cascading style sheets, level 1. (Sept. 1996).
[27] Richard Macmanus. 2020. *1995: The Birth of JavaScript*. Web Development History. https://webdevelopmenthistory.com/1995-the-birth-of-javascript
[28] Shaghayegh Mardani, *et al.*. 2020. Fawkes: Faster Mobile Page Loads via App-Inspired Static Templating. In *Proc. of USENIX NSDI*. 879–894.
[29] MDN. 2023. *Connection management*. https://developer.mozilla.org/en-US/docs/Web/HTTP/Connection_management_in_HTTP_1.x#http_pipelining
[30] Mark Nottingham. 2012. Rechartering HTTPbis. https://lists.w3.org/Archives/Public/ietf-http-wg/2012JanMar/0098.html
[31] Addy Osmani. 2024. *critical*. https://github.com/addyosmani/critical
[32] A. Osmani*et al.* 2018. *JavaScript modules*. https://v8.dev/features/modules
[33] Steven Pemberton. 2022. On the Design of the URL. https://homepages.cwi.nl/~steven/Talks/2020/10-09-urls/design.html
[34] Q-Success. 2024. *Usage Statistics and Market Share of JavaScript Libraries for Websites, 2024*. https://w3techs.com/technologies/overview/javascript_library
[35] John Resig. 2006. *BarCampNYC Wrap-up*. Blog Post. https://johnresig.com/blog/barcampnyc-wrap-up/
[36] Turbo. 2024. Turbo: The speed of a single-page web application without having to write any JavaScript. https://turbo.hotwired.dev
[37] 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.
[38] 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.
[39] Lucas Vogel and Thomas Springer. 2023. Speed Up the Web with Universal CSS Rendering. In *International Conference on Web Engineering*. Springer, 191–205.
[40] B. Wagner. 1998. Controlling Cgi Programs. *SIGOPS Oper. Syst. Rev.* 32, 4, 40–46.
[41] Webpack Contributors. 2017. *code splitting*. Documentation. GitHub. https://github.com/webpack/docs/wiki/code-splitting