Mega-fast or just super-fast? Performance differences of mainstream JavaScript frameworks for web applications

Andreas Nicklaus
Hochschule der Medien Stuttgart
Matrikelnummer: 44835

Prüfer: Prof. Dr. Fridtjof Toenniessen & Stephan Soller

Abstract

Part of every web application project is the choice of tooling, especially the choice of framework for the development team. Unfortunately, this discussion has evolved into a sentiment matter rather than a factual one. This thesis presents a study of a single examplary web application created identically with seven mainstream JavaScript web development frameworks: Angular, Astro, Next.js, Nuxt, React, Svelte and Vue.js. A testing suite is proposed using Lighthouse and Playwright to cover the classic page load, the load of JavaScript components and the update of JavaScript components. The evaluation of the measurements include the usage of two new derivative metrics: the Observed Visual Change Duration (OVCD) and a loadEventEnd metric balanced towards the requestStart event of NavigationTiming measurements. The results show no clear-cut overall advantage of one single development framework, but outline strengths and weaknesses of all tested frameworks. Component update times indicate Nuxt as the fastest and Next.js as the slowest framework for update times. Similarly, Google Chrome appears to be the fastest and Desktop Safari the slowest browsers for updating the DOM after user input.

1 Introduction

Throughout the evolution of the world wide web, development of website became more complex, both of the page content and the functionality. This complexity resulted in higher demand for technical sophistication in networking, hosting services and development tools. Although modern frameworks provide technical advancements to increase the speed of page and content generation and arguably a better developer experience, there is no apparant way to objectively determine a “best framework” in terms of developer experience.

When it comes to user experience and perceived performance however, there are plentiful collections of metrics and criteria to choose from so as to determine the performance of websites, not frameworks. The optimization of websites has become a goal during development because it has a real effect on Search Engine Optimization and user behaviour.

Both effects create business interests and financial incentives to invest resources into performance optimization (Li et al., 2010; Zhou et al., 2013). Past research and existing tools as well as guide give direction to optimize websites according to stakeholders' and users' expectations and in most case only focus on specific websites or specific frameworks or give general advice.

However, the lack of research on the effect of the framework on website performance indicates a need for research on the topic. Relying on marketing material for the choice of framework is questionable because most modern frameworks claim to be fast, easy to use and performance efficient. This suggests that each would be a great choice. Comparing frameworks presents a challenge because no ideal set of metrics for this use case apparant and ther are no publicy available replicas of web applications built with diffent frameworks. Therefore, a comparative study between the same website built with different web development frameworks is needed. With this data, an informed choice might be made for projects in the future. The goals of this thesis are to propose a set of metrics that allow comparing mainstream JavaScript (JS) frameworks fro web applications, to provide a comparative study between selected frameworks and to create a tools to compare the rendering performance a web page as a whole and of dynamic components within a page.

2 Setup of the application and testing environment

One of the choices for the setup of the study is which frameworks to implement the application in and compare. The selected frameworks have to support the designed web application without the help of another tool or framework unless intended by the developers of the framework. Plus, the frameworks have to use JavaScript (JS) in order to narrow down the scope the study. TypeScript frameworks are allowed because they support JS (Bierman et al., 2014).

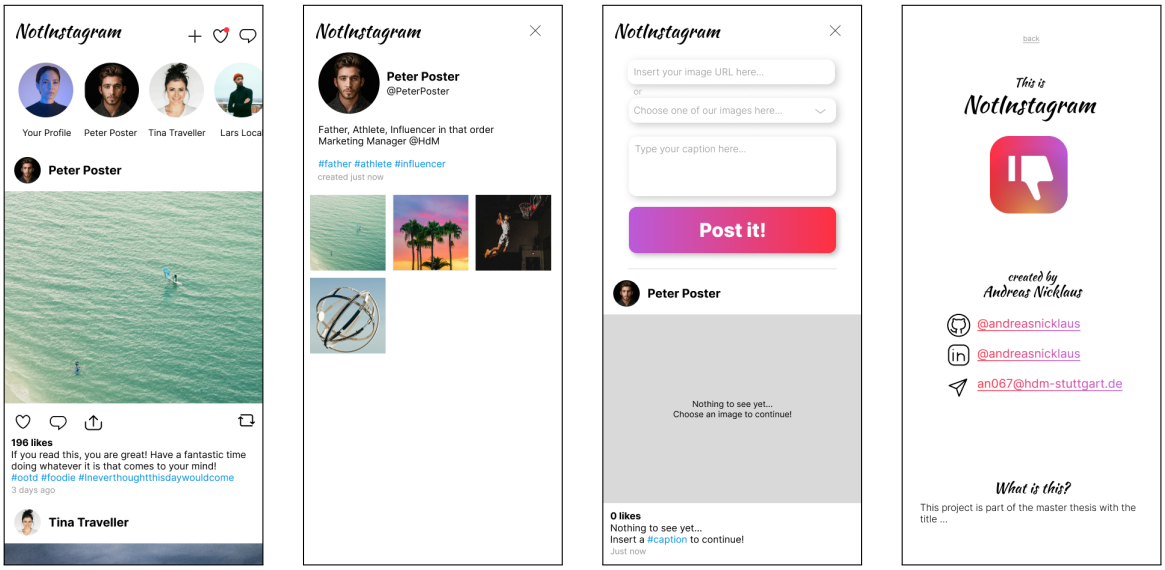
Basis for the framework selection is the rankings of most-used, most-liked and most-interesting web development frameworks and tools (Devographics, 2024). The following frameworks were selected for this framework:

1. Angular
2. Astro
3. Next.js
4. Nuxt
5. React
6. Svelte
7. Vue.js

In addition, Preact, Solid and Qwik were considered to be included in this study, but were dropped because of negative sentiment or low usage among developers that have experience with the tools.

The web application used for this study is designed to be subject of comparisons between frameworks. Its look is derived from the Android app of Instagram (Instagram from Meta, 2024) and it has four pages (see figure 1). The four pages cover three generally

valid page types identified in the design process. The About page is a "Static page" as it does not change its content after the initial response from the web server. No additional data query is needed to build the finished DOM structure. The Feed page and the Profile page are "Delayed pages". Their defining characteristic is that the DOM cannot be fully built from the initial HTML document, but need data queries to complete before all content can be displayed. These data queries are triggered immediately after the initial page request. The Create page is the only "Dynamic page". Event its initial features indicate it being either a static or delayed page, depending on the implementation, it has dynamic components that update through user input. Mutations to the DOM are therefore not triggered by the initial page request but user action. The time of such mutations is therefore not predictable.



(a) Feed page (b) Profile page (c) Create page (d) About page

Figure 1: Screenshots of the NotInstagram application's pages (path in parentheses)

These four pages are comprised of 15 components, most of which are wrappers to encapsulate image components, styled text or iterations over lists with subcomponents. However, two components stand out because of their special purpose and implementation differences between frameworks.

1. The MediaComponent is designed to present both internal and external image and video sources in a single component. It is used to display Profile images and Post content. Its main purpose is to decide - based on the passed source string - how to project the multimedia file onto the DOM. As such, a decision fro enhanced image or video elements had to be made per framework during the implementation of the application. Svelte, Astro, Next.js and Nuxt provide such an enhanced image component. In contrast, video elements are inserted to the DOM as-is, but the browser behaviour is adapted identically for all frameworks using attributes on the `<video>` element and JavaScript. In addition, the import of local images differs between frameworks because the load behaviour differs. As such some frameworks require importing all local images in order to select the requested image.

2. The Create page poses a challenge to Astro because it does not natively support dynamic components. The intended solution is to implement so-called "Islands" using another framework. React is chose for its high usage rate among web developers (Devographics, 2024). As a result, two implementations are compared in this study: Using the React components that are needed for Astro Islands everywhere, even if the component in question is not dynamic, and creating duplicate Astro components for when a component is not required to be dynamic. One additional React component "CreateForm" was created in order to encapsulate React subcomponents and six components were implemented in React because they are part of the form and the Post preview on the Create page.

In order to test the end-products of the frameworks, at least one web server is needed to host the application. Previous work suggest that network delay is a great part of render delay and performance issues (Grigorik, 2013). For this reason, the tests for this study are performed on two different web servers: An online hosting service and the local testing machine.

1. **Vercel** was chosen for hosting the applications on distant servers based on its popularity, capabilities for Server-side Rendering (SSR), support for both a free and paid version and its integration into CI/CD pipelines. Each Vercel project was connected to a Github repository, one per framework. Only required project configuration options were changed per project on the plattform to ensure its state as as-is.

2. A **local host** was chosen to minimize the effect of network delay and related delays, e.g. domain name resolving, in this study. The application is hosted on the testing machine. This client device is a HP Envy x360 with an AMD Ryzen 5 5500U processer and 16 GB RAM. The OS on the device is Windows 11 Home (Version 10.0.22631) during testing. The application was build before every test and hosted using either built-io commands for the framework or using the `serve` command (see table 1)

Framework	Build Command	Host Command
Angular	`ng build`	`serve`
Astro	`astro build`	`astro preview`
Next.js	`next build`	`next start`
Nuxt	`nuxt build`	`nuxt preview`
	`nuxt generate`	`nuxt preview`
React	`react-scripts build`	`serve`
Svelte	`vite build`	`vite preview`
Vue.js	`vite build`	`serve`

Table 1: Build and host command for each used framework as used for testing the applications hosted locally

To identify weaknesses of the frameworks and simplify the evaluation of the results, eleven metrics were chosen in three to test the frameworks (see table 2). The page load

covers the classic load time of web pages and is specified to outline the load speed from `requestStart` to the last change to the page. The Component Load is defined as the time frame frame in which any changes to the DOM with JS can be identified and the rendering process of JS components are shown. The Component Update Time is defined as the time between a user interaction and a DOM mutation. This time frame describes the speed of feedback to the user that the interaction has been registered and something is happening as well as the speed until that something finishes happening. Especially DOM mutation times are expected to show differences inbetween frameworks and implementations as the elements and internal implementation changes from one framework to another.

Page	**Component**	
Load Time	**Load Time**	**Update Time**
Total Byte Weight	Observed Last Visual Change	
Time To First Byte	Observed First Visual Change	
Time To Interactive		
Total Blocking Time		
LoadEventEnd		
DomContentLoaded	DOM Mutation Times	
Last Visual Change		
Largest Contentful Paint		

Table 2: Assignment of metrics to the metric categories

The requirements for testing tools created by hosting the application on two different web servers and by the list list of metrics are fulfilled by the Lighthouse CLI and Playwright. Both are required to output their results in human-readable and machine-readable format to support easy debugging and the creation of aggregate metrics.

1. Using the **Lighthouse CLI**, a script for starting the web server and running Lighthouse tests on the web application. These tests run 20 times and only cover the performance measurements of Lighthouse. Reports are created in both HTML and JSON format in order to debug the tests and create the mean average of every measurement.

2. Tests with **Playwright** focus on the measurement of DOM mutations and the adherence to time budgets. To that end, a JS script is injected into the browser context before tests. This recording script initializes a MutationObserver on a specific HTML element that is created by the framework. This way, all DOM such as element addition, element removal and attribute change are recorded with the id and xpath of the element a time of the mutation.

Which metric is measured by which tool can be seen in table 3.

3 Test results

Both metrics for th page load and for the component load times show no clear generally applicable evidence for a single framework being faster than the others. Such a distinction

Lighthouse	Playwright
Total Byte Weight (TBW)	domContentLoaded
Time To First Byte (TTFB)	loadEventEnd
Time To Interactive (TTI)	User Input Times
Total Blocking Time (TBT)	Mutation Times
Largest Contentful Paint (LCP)	
First Visual Change (FVC)	
Observed First Visual Change (OFVC)	
Observed Last Visual Change (OLVC)	

Table 3: Assignment of metrics to the test tools

can only be made on aper metric basis. Figure 2 presents the averages of measurements from the Lighthouse reports per page and framework.

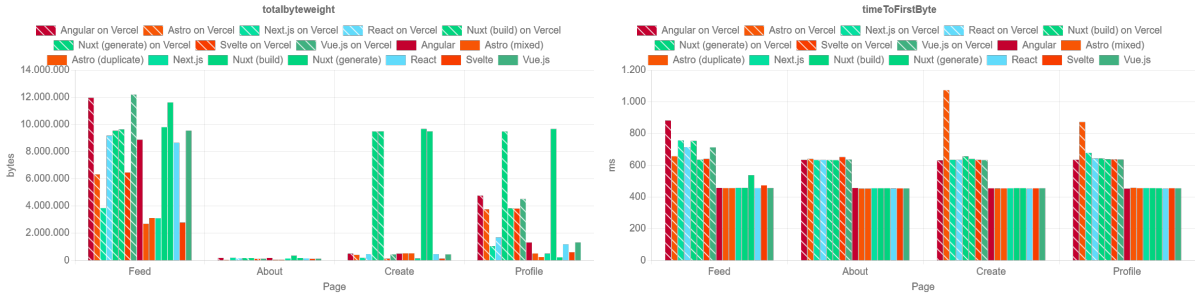
Next.js, Astro and Svelte are the leading frameworks in TBW and Svelte, Next.js, Vue.js and especially astro have fast results in their TTI. In addition, Astro, Angular, Svelte, Nuxt and Vue.js stand out through little fluctuations in TTI across the four pages and the test repetitions. The results of measurements for the TBT also favors Astro and Svelte. In contrast, Astro and Svelte perform poorly in DomContentLoaded and balanced LoadEventEnd. These metrics are strengths of Vue.js, React and Nuxt. The balanced LoadEventEnd is balanced towards the requestStart (see equation 1). Vue.js and React are also the fastest frameworks in OLVC. The TTFB does not support a ranking of frameworks. Instead, it is more dependent on the page and the host, which influence the results more than the framework. However, Astro, Next.js and Angular stand out through slow results in this metric. The balanced LoadEventEnd hightlights Vue.js and React positively, but also demonstrates a high dependency on the browser.

$$loadEventEnd_{balanced} = loadEventEnd_{raw} - requestStart \tag{1}$$

The metrics for the component load time have similar characteristics. The OFVC of the applications are early in Astro, React and Next.js, which indicates a strength of React-based frameworks. React, Vue.js and Angular also naturally have a short Observed Visual Change Duration (OVCD), which is unsurprising. The OVCD is defined as the time difference between OFVC and OLVC (see equation 2). The recordings of early DOM mutations are also very fast for Astro, Vue.js and React, whereays recordings are missing completely for Angular. This is most likely due to a faulty initialization of the MutationObserver that is responsible for recording mutation times.

$$observedVisualChangeDuration = observedLastVisualChange - observedFirstVisualChange \tag{2}$$

In contrast, the measurements made for the component update times suggest clear rankings of the frameworks and of browsers. The times of the DOM mutations are quite similar to each other except in Mobile Safari and Desktop Safari. In these browsers, Next.js is the slowest and Nuxt is the fastest framework. Across all pages and frameworks, the ranking of browsers from fastest to slowest is Google Chrome, Microsoft Edge, Chromium, Mobile Chrome, Firefox, Mobile Safari and Desktop Safari. This means that

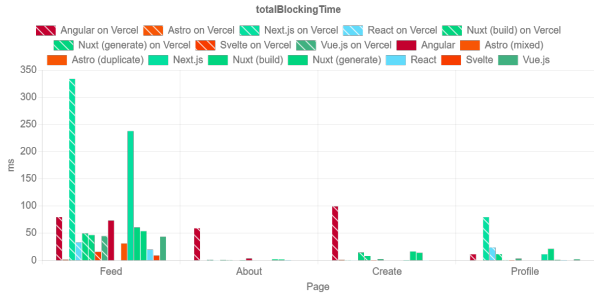


(a) Total Byte Weight (TBW)

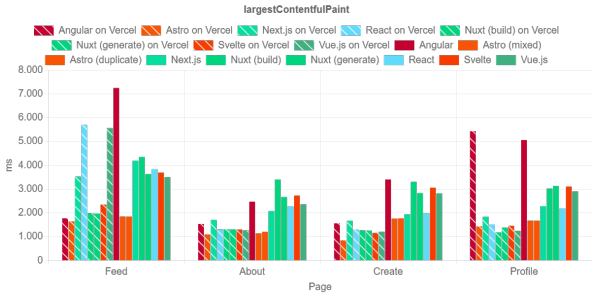
(b) Time To First Byte (TTFB)

(c) Time To Interactive (TTI)

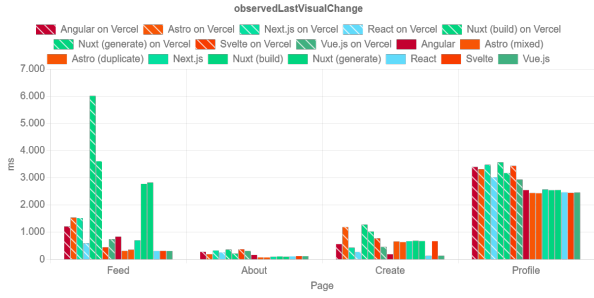
(d) Observed DomContentLoaded



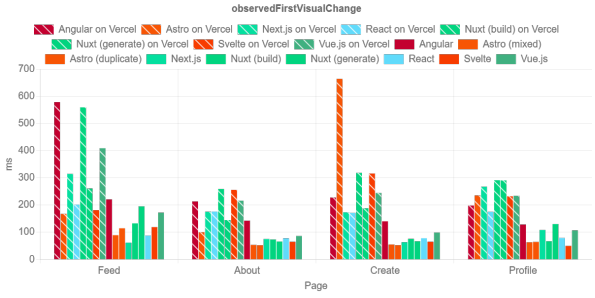
(e) Total Blocking Time (TBT)



(f) Largest Contentful Paint (LCP)



(g) Observed Last Visual Change (OLVC)



(h) Observed Last Visual Change (OLVC)

Figure 2: Lighthouse test results in Google Chrome

time budgets are easiest to keep to in Google Chrome and that testing in Desktop Safari is more challenging for the test subject. The ranking of frameworks is from fastest to slowest across pages and browsers Nuxt, Angular, Vue.js, React, Astro/Svelte and Next.js. In addition, Nuxt, Vue.js and Svelte are economical with DOM mutations after user interaction, whereas the other frameworks update the DOM after user interaction

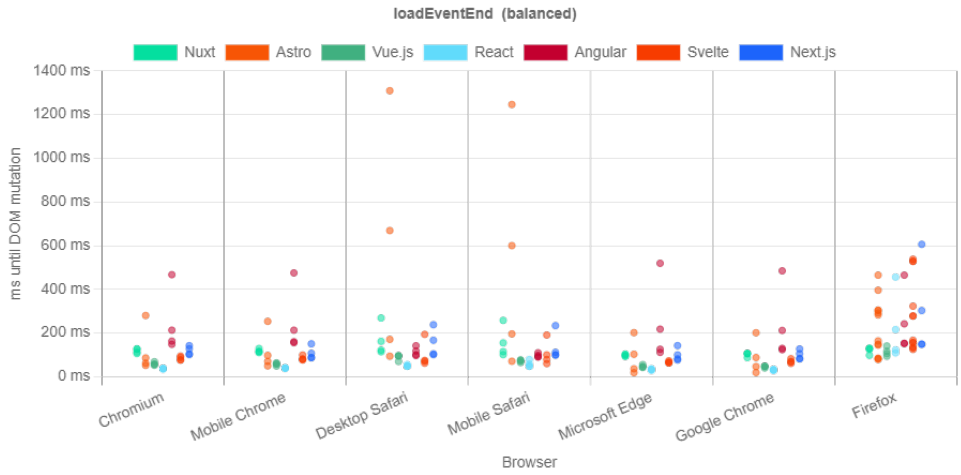


Figure 3: Balanced loadEventEnd timings

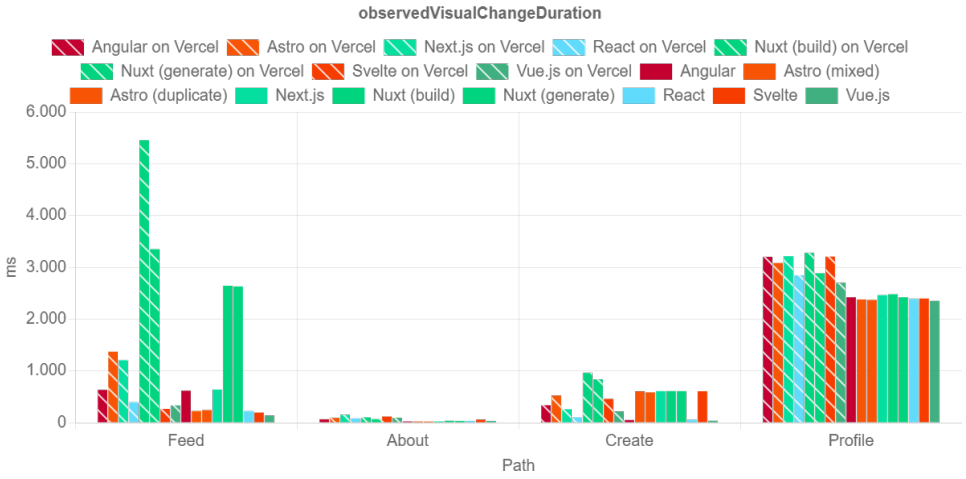


Figure 4: Observed Visual Change Duration (OVCD)

in more different ways. This ranking can influence the choice of framework for user input heavy applications. For this kind of application, Nuxt, Angular, Vue.js and React present themselves as the best choices relating to component update times.

4 Summary

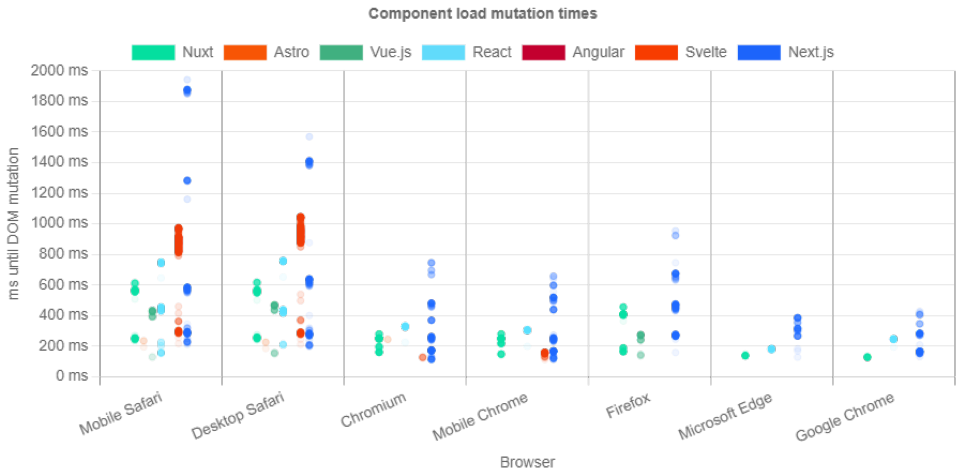


Figure 5: Component load mutation times

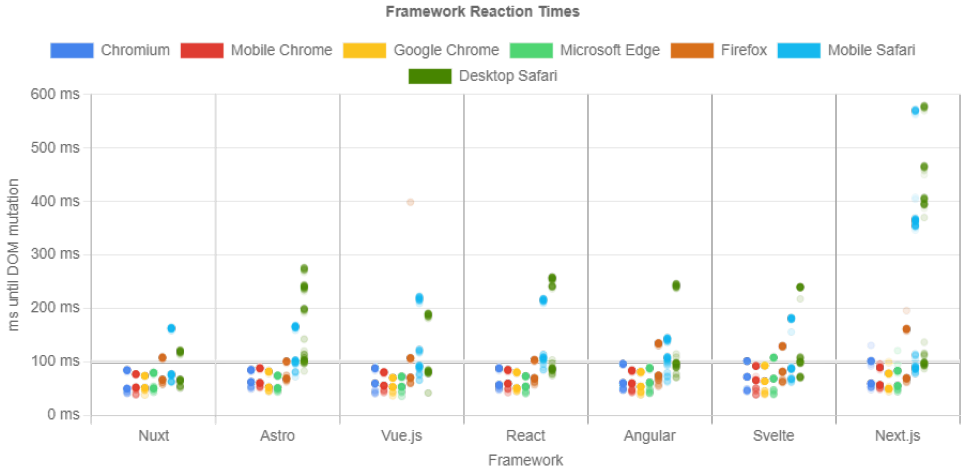


Figure 6: Recorded DOM mutation timings after user actions

A Acronyms

CI/CD Continuous Integration and Continuous Delivery.

DOM Document Object Model.

FVC First Visual Change.

HTML Hypertext Markup Language.

JS JavaScript.

JSON JavaScript Object Notion.

LCP Largest Contentful Paint.

LVC Last Visual Change.

OFVC Observed First Visual Change.

OLVC Observed Last Visual Change.

Browser	Usage quota
Google Chrome	65.68 %
Desktop Safari	17.96 %
Microsoft Edge	5.26 %
Firefox	2,75 %
Chromium	NA
Mobile Chrome	NA
Mobile Safari	NA

Table 4: Browser usage (StatCounter, 2024)

OVCD Observed Visual Change Duration.

SEO Search Engine Optimization.

SSR Server-side Rendering.

TBT Total Blocking Time.

TBW Total Byte Weight.

TTFB Time To First Byte.

TTI Time To Interactive.

B References

Bierman, G., Abadi, M., and Torgersen, M. (2014). Understanding typescript. In Jones, R., editor, *ECOOP 2014 – Object-Oriented Programming*, pages 257–281, Berlin, Heidelberg. Springer Berlin Heidelberg.

Devographics (2024). State of javascript 2023. `https://2023.stateofjs.com/en-US/libraries/front-end-frameworks/`. accessed 07/29/2024.

Grigorik, I. (2013). *High Performance Browser Networking*. O'Reilly Media, Inc., 1005 Gravensetin Highwy North, Sebastopol, CA 95472.

Instagram from Meta (2024). Instagram. `https://www.instagram.com/`. accessed 08/02/2024.

Li, Z., Zhang, M., Zhu, Z., Chen, Y., Greenberg, A., and Wang, Y.-M. (2010). Webprophet: automating performance prediction for web services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, page 10, USA. USENIX Association.

StatCounter (2024). Quick start. `https://gs.statcounter.com/`. accessed 07/18/2024.

Zhou, M., Giyane, M., and Nyasha, M. (2013). Effects of web page contents on load time over the internet. *International Journal of Science and Research (IJSR)*, pages 2319–7064.

	Angular	Astro	Next.js	Nuxt	React	Svelte	Vue.js	Average
Chromium	44 68,9 95	51 71,75 89	47 75,41 108	39 66,4 94	44 57,87 85	38 73,65 95	51 77,18 104	44,857 70,166 95,714
Firefox	54 88,7 123	63 99,34 142	59 142,18 235	59 82,78 108	54 83,72 181	60 93,67 129	52 82,24 103	57,286 96,090 145,857
Desktop Safari	77 122,7 172	87 170,1 270	79 303,54 493	51 86,02 124	84 169 280	70 164,23 283	47 135,72 200	70,714 164,473 260,286
Mobile Chrome	44 66,84 90	49 68,96 85	47 93,7 143	42 61,14 82	44 67,41 82	45 80,81 116	46 69,04 89	45,286 72,557 98,143
Mobile Safari	52 105,61 152	78 154,28 254	73 196,39 372	47 110,32 167	67 125,67 183	56 126,19 208	52 133,03 206	60,714 135,927 220,286
Microsoft Edge	43 70,38 90	44 64,25 80	46 72,77 134	37 60,88 85	41 62,13 75	40 74,46 102	40 60,9 79	41,571 66,539 93,143
Google Chrome	41 62 84	43 57,49 72	41 68,75 99	34 59,92 77	40 59,48 77	39 64,24 89	37 61,1 77	39,286 61,854 82,143
Average	50,714 83,59 115,143	59,286 98,024 141,714	56 136,106 226,286	44,143 75,351 105,286	53,429 89,326 137,571	49,714 96,75 146	46,429 88,459 122,571	
Weighted average	44,505 68,9 94,179	47,915 74,421 103,895	45,159 107,408 167,077	35,06 60,283 80,285	45 74,989 109,784	41,941 78,181 118,195	36,277 69,971 93,482	

Table 5: Minimum, average and maximum of recorded mutation times after user input in milliseconds (fastest times are highlighted green, slowest red)

Github repository: All code and additional meterial can be found under `https://github.com/andreasnicklaus/master`.

	Passed	**Flaky**	**Failed**
Angular	112	0	0
Nuxt	112	0	0
Next.js	111	0	1
React	110	2	0
Vue.js	110	2	0
Svelte	108	0	4
Astro	103	3	6

Table 6: Total passed, flaky and failed Playwright tests per framework