



Masterarbeit im Studiengang Computer Science and Media

WIP: Mega-fast or just super-fast? Performance differences of mainstream JavaScript frameworks for web application

vorgelegt von

Andreas Nicklaus
Matrikelnummer 44835

an der Hochschule der Medien Stuttgart

am 1. August 2024

zur Erlangung des akademischen Grades eines Master of Science

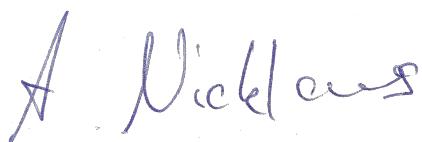
Erst-Prüfer: Prof. Dr. Fridtjof Toennies
Zweit-Prüfer: Stephan Soller

Ehrenwörtliche Erklärung

Hiermit versichere ich, Andreas Nicklaus, ehrenwörtlich, dass ich die vorliegende Masterarbeit mit dem Titel: „WIP: Mega-fast or just super-fast? Performance differences of mainstream JavaScript frameworks for web application“ selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§26 Abs. 2 Bachelor-SPO (6 Semester), § 24 Abs. 2 Bachelor-SPO (7 Semester), § 23 Abs. 2 Master-SPO (3 Semester) bzw. § 19 Abs. 2 Master-SPO (4 Semester und berufsbegleitend) der HdM) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Eislingen, den 1. August 2024

A handwritten signature in blue ink, consisting of a stylized 'A' followed by the name 'Nicklaus' in a cursive script.

Andreas Nicklaus

Zusammenfassung

Diese Arbeit kurz und knackig.

Abstract

This work in a nutshell.

Disclaimer: This paper has been written with the help of AI tools for translating sources and outlining parts of the written content. All content has been written or created by the author unless marked otherwise.

Contents

1	Introduction	4
2	Related Work	5
3	Design	5
3.1	Example Application	5
3.2	Choice of frameworks	11
3.3	Hosting Environments	11
3.4	Performance Metrics	13
3.4.1	Page Load Times	14
3.4.2	Component Load Times	16
3.4.3	Component Update Times	17
3.5	Testing Tools	18
4	Implementation	20
4.1	Components	20
4.2	Tests	20
5	Evaluation	20
5.1	Page Load Times	20
5.2	Component Load Times	20
5.3	Component Update Times	20
6	Conclusion	20
7	Summary	20
A	List of Figures	21
B	List of Tables	21
C	Acronyms	21

1 Introduction

Throughout the evolution of the world wide web, many changes have disrupted the way websites are created. From simple file servers run by few selected institutions, simple static web pages and dynamic services like blogs and forums to websites created with the help UI tools and web development frameworks, mainly written in JavaScript, development has changed drastically since its beginning.

Older web pages often lacked features, that developers today work with as a matter of course. Yet their load and rendering most likely would be brazenly fast with today's technological advancements in networking, browser functionalities and user equipment. Modern websites though are often bigger in size, have a lot more features and are in many respects more complex. Due to the increased complexity, the mere volume of a website's data has increased, especially with more and more multimedia files. That in return has increased the demand for better performance on all components of the load and rendering process. This technological advancement has upped the technological sophistication for development tools as well. Today's modern web development frameworks support developers with tools to create sites and applications through terminal commands. They often increase the content-per-line-of-code quota through implicit page generation in contrast to the explicit writing of source code from earlier times. Many frameworks even feature configuration options for directly hosting the webpage.

As the generation process changed from writing code manually to automatically, this implicit page generation undoubtedly increased speed through faster content generation and a greater developer experience for some developers. Because developer experience varies between different frameworks and some approaches are more intuitive to respective developers, a current trend has evolved for developers to become experts in a single framework rather than many. This trend leads to a tribal conflict as to which framework is better than others with each tribe being convinced that their framework is the best. There is no apparent way to determine a "best framework" in terms of Developer Experience because it is a subjective criterion. The performance of a framework as assessed by the developer can be similar or greatly different, depending on the frameworks and the interviewees.

When it comes to User Experience and especially the Perceived User Experience 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 both the ranking of web pages in search engines and the user behavior. Both effects create business interests and financial incentives to invest resources into performance optimization. However, the lack of research on the topic suggests either a consensus for a negligible effect of the development framework on the website's performance or a lack of knowledge of the effect. Measurements on the effect of the development framework are a major convoluted task simply because the performance of a specific website can be dependent on many other factors such as the user's device, browser, networking hardware or server-side hardware. The number of possible combinations of factors and their reliability makes it difficult to measure a single performance run with a reliable result. Ev-

ery single result is only a small part of a large number of possible performances the same application could achieve with different parameters. It is therefore conceivable that a “perfect combination” of hard- and software exists for each framework or in general, but it is currently not possible to find such a combination because the necessary data is missing.

Many modern web tracking services provide data about the user, the user’s devices, current page load times and so on. This data is helpful in determining current poor performances and therefore possible starting points for optimization efforts. But it gives very little information about recommended actions or recommended choice of frameworks for a redesign of a web application. Relying on marketing material for choice of frameworks is equally questionable because most modern frameworks claim to be fast, easy to use and performance efficient. This suggests that each would be a great choice for developers.

In order to find a suitable framework for an application, a set of metrics needs to be at least outlined for comparison. Many former studies suggest metrics to be relevant for the User Experience or Search Engine Optimization. Content metrics such as word count or presence of meta tags might be important for some performance measurements, but might also have no effect on the User Experience. In contrast, rendering metrics such as page load time or page weight might be ascribed to the framework used during development.

The performance of a framework towards the user can very rarely be compared because there are no publicly available comparisons between exact replicas of web applications built with different frameworks. Therefore, a comparative study between the same website built with different frameworks is needed to get as close as possible to an exact website replica. With this data, an informed choice might be made for other projects.

The goals of this paper are to propose a set of metrics that allow comparing mainstream JavaScript frameworks for web applications, to provide a comparative study between selected frameworks and create a tool to compare the rendering performance of a page as a whole and of dynamic components within a page.

2 Related Work

3 Design

Whereas the following chapters cover the implementation of testing and evaluation of results, this section introduces the concept of the comparative study. The goals of and requirements for the example application, the differences and choices for the hosting environments for testing and the tools for testing as well as selected metrics will be described here.

3.1 Example Application

The example application for the study is designed to be a benchmark application for testing. The following goals were considered during the design process:

1. **Page types:** With the goal of covering most kinds of webpages, three types were identified based on the time of data loading. These types differ

in timing at which the DOM content is loaded or updated. The definition of a finished load or update for this work is that the linking of resources does constitute a finished load or update of the webpage regardless of the load time of said resource. The only condition for that is that any linked resource does not update the DOM in any way. If a resource does, then the load or update is considered not finished.

- (a) Static pages are webpages which do not change their content after the initial response from the web server. The initial HTML document already is the only resource that is necessary to create a complete DOM. If inline scripts update the DOM, they are considered external resources.
- (b) Delayed pages do not have a complete DOM after loading and parsing of the initial HTML document. Some data or content is loaded and inserted (or removed) into the DOM after the initial render. The time of these updates can be any time after the initial render, but the execution of code or start of request for the resource that is responsible for the update has to be directly or indirectly triggered by the content of the initial DOM or HTML document.
- (c) Dynamic pages can be updated or update themselves by events that are not triggered by the content of the initial DOM or HTML document. These events can either be triggered by user interaction or other events such as websocket messages. The time of such updates is by their nature not predictable. Dynamic pages are either static or delayed with additional possibilities for updates.

This list is created with the knowledge that frameworks or other technology such as caching may move a webpage from one type to another.

2. **Modern Development Practices:** The example application should contain modern development practices that do project onto the DOM. Practices that have no effect on either the projection of data or user interaction, such as coding styles or project management, are considered to have no effect on the performance of the page.
 - (a) Components: All pages of the app have to consist of components that encapsulate reproducible HTML snippets and may project data onto the DOM.
 - (b) List iteration: Because iterating long lists may decrease performance noticeably, some components or pages should implement list iteration.
 - (c) String interpolation: Although it is not considered a performance issue before testing, string interpolation is prevalent in all modern frameworks known to the author.
 - (d) Services: Separation of functions in services is wide spread practice to reduce code duplicates and easy refactoring. In this case, services also allow to intentionally implement delays for testing purposes.
3. **CSS:** Even though the usage of CSS can in no way be considered a modern practice, it is still used on effectively every webpage. Additionally,

stylesheets are considered render-blocking resources that impact performance negatively. For this purpose, CSS shall be implemented for both pages and components.

4. **Rendering time:** In addition to page type depending on the time of data load, the machine and time of composing the DOM is dependent on the content availability. For this paper, three different types are considered:
 - (a) Client-side Rendering (CSR): The initial request gets a response with a mostly empty HTML document (“skeleton”) except linked CSS and JS resources which after loading, parsing and execution update the DOM.
 - (b) Server-side Rendering (SSR): Updates that happen after receiving the skeleton through JS code execution on CSR, happen before the initial request is responded to on the web server. The initial HTML document is filled and no longer a skeleton with SSR. Therefore, it has greater byte size. Server-side Rendering requires an “active” front-end server rather than only a file server to execute code.
 - (c) Prerendering: Rendering happens during build time of the application. This increases the build time and the byte size of the initial HTML document, but only a file server is needed for hosting.
5. **Multimedia:** Most of network load and therefore network delay is made up by multimedia files. Although compression has gotten better over time, the byte size made up by multimedia files of a webpage has gotten larger over the last years. Therefore, size optimization of image and video files is considered a major part of performance optimization and a great potential for a performance increase by the used framework.

Based on these considerations, the application “NotInstagram” was designed as a comparable example application. It is heavily inspired by Instagram and a partial reproduction of its app design. “NotInstagram” consists of four pages (see figure 1). 1a shows the design of the Feed page. It is the start page of the app and contains 4 parts: the header, the profile list, the post list and a footer. Each item of the feed page is to be implemented as its own component or components. The plus icon in the header links to the create page, the footer links to the about page and every instance of a profile picture and profile name links to a profile page. The later contains profile information including a profile picture, name, user handle / ID, profile creation time, caption and a grid of all the user’s posts (see figure 1b). The profile component encapsulates all HTML elements of that page except the header containing the app logo and X icon, which both link back to the feed page. Both the feed page and the profile page are generally expected to classify as delayed pages, because the content of profile and posts lists can only be loaded after the page load.

The Create page (see figure 1c) has three parts. The header contains the app’s logo and a X icon linking to the feed. A form with three `<input>` elements and a `<button>` element allows the input of an multimedia source (image or video) and a text caption. The multimedia source can either be an URL or a selection from a list of preuploaded files. The post caption is a pure text input. The lower part of the page is the post preview, in which some predefined

information such as user profile and the user inputs are combined. As such, the profile page is a static page until the user uses the creation form, at which point it has to be considered a dynamic page. The About page (see figure 1d) is designed to statically display information about the application. It is a static page because no content is loaded after a delay and no user inputs are possible.

With these pages all page types are covered for testing. The About page and Create page are static, whereas the Feed page and Profile page are partly static (header and footer), but mostly delayed. The Create page is the only page with dynamic content.

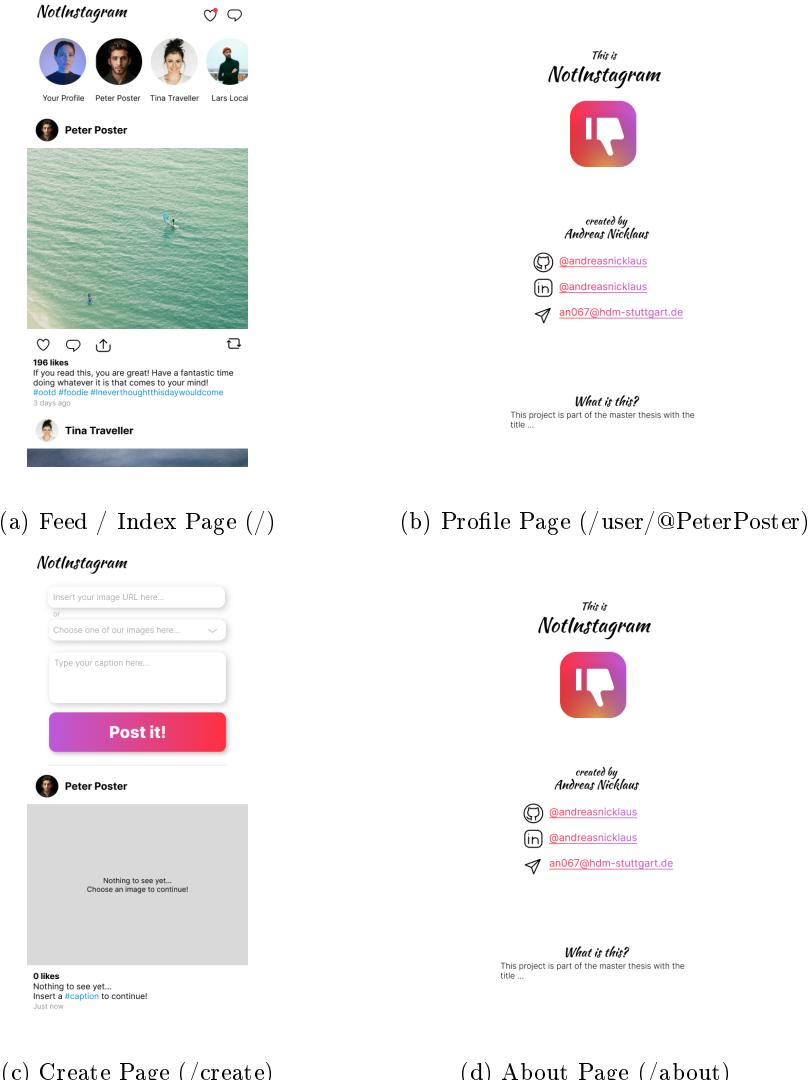


Figure 1: Screenshots of the NotInstagram application's pages (path in parentheses) **(Bilder müssen noch geändert werden)**

The data fetching and loading is designed to be implemented as services. For NotInstagram two different services are needed. The PostService is a service for

all components to query posts. The method `getAll()` returns a list of all posts by all users and `getByUserHandle(handle)` returns the same list filtered by those posted by a user with the handle equal to the function parameter. ProfileService is a service to query user profiles. It has the same two methods which return all user profiles and only one user profile respectively. Services are designed asynchronous, but the data is not queried from a server external to the browser, but hard coded. This design decision is based on the premise that delay can be coded into or out of asynchronous functions to mimic network delay for testing purposes if necessary.

Figure 2 describes the usage of components and services within page views. It displays the four pages of NotInstagram as views, the two services and 15 components. Seven of those components are icon components. Those components serve as wrappers for SVGs to ensure their correct scale and style. `XIcon` poses an exception to the design as it is a wrapper for a `PlusIcon` component rotated by 45°. The colored arrows show the usage of one of the services. Both `FeedView` and `ProfileView` use both services to load data. For the Feed page, both `PostService.getAll()` and `ProfileServices.getAll()` are needed to pass the data to `PostList` and `ProfileList`. Notably, each `Post` component accesses the `ProfileService` again, to get the profile image and name for its headline, even if the information is available in a parent or grandparent component. Figure 3 displays the connections between post and profile object instances. The member `userhandle` of a post references the member `handle` of a user profile. The `Profile` page needs access to the service to get the information of the requested profile and a list of posts from the `getByUserHandle` methods to pass into the `Profile` component. `LogoHeader`, `NotInstagramLogo` and `InfoBlock` are not data-presenting components, but rather styling components. Their only function is styling text or projecting HTML elements with CSS information.

In contrast, the `MediaComponent` is designed as a way to allow both internal and external images and video source. It is used by `ProfileList`, `Post` and `Profile` to display posts and profile images. It's main goals is to decide based on the passed image source string how to project the multimedia file onto the DOM. The component accepts source strings for images and videos, differentiated against by the string's ending and therefore the file's extension. If it is a local image, namely an image that was available for optimization at build time, the best available form of optimized `` tag should be used. For external image links starting with “`http://`” or “`https://`” a less optimized or unoptimized `` tag shall be inserted into the DOM. For videos, any source string is to be projected onto a `<source>` tag with identical `<video>` wrapper.

The application refers to local images, which can possibly be optimized, and external images, which cannot be optimized. The reason for this is the assumption for this project that optimizing multimedia files uploaded by a user and referencing them in a manner suitable for this application is not suitable for this work. Rather, the better alternative for serving the use case of the application would be a dedicated server for encoding, decoding and generally optimizing multimedia files. Since this solution would be independent from the front-end framework's performance and it would outgrow the scope of this work, a distinction is only made between static images, called local images here, and external images with full URLs.

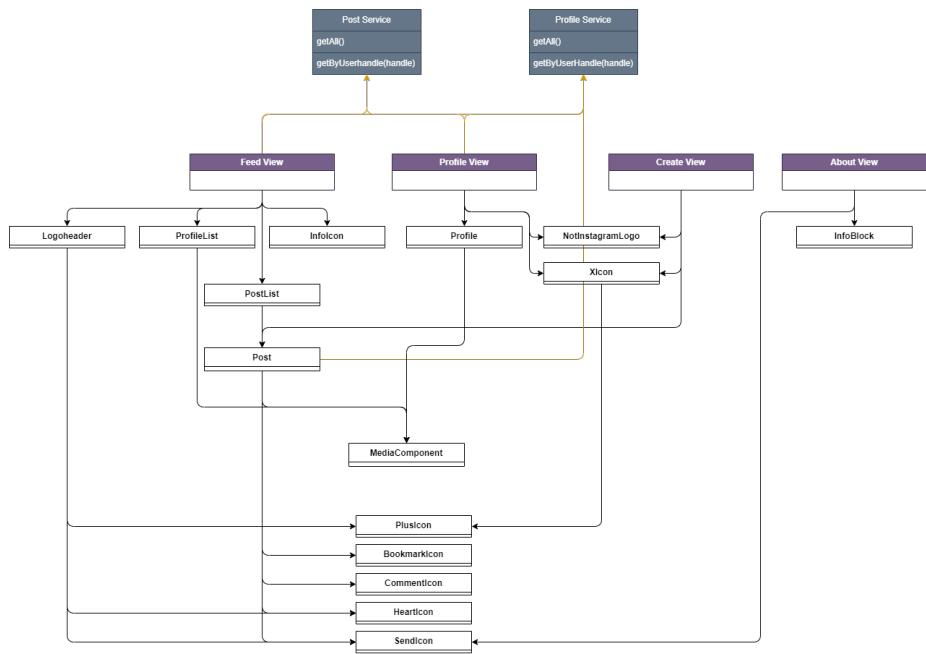


Figure 2: Pages, Components and Services of the NotInstagram application

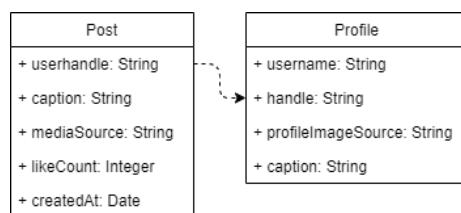


Figure 3: Classes used by the NotInstagram services

3.2 Choice of frameworks

The choice of tested frameworks for this study is the choice for which frameworks the application will be implemented in and tested. The requirements for this selection are twofold. The application has to be implementable as designed above with the framework without the use of any other non-native tool to the framework or any tool that was not officially intended to be used in combination by the developers of the primary framework. Additionally, the application must be implementable in JavaScript. This requirement includes TypeScript frameworks because it is possible to use JavaScript in TypeScript applications. Ease of use and developer experience should explicitly not influence the selection process because it is part of the evaluation of the frameworks.

Because research revealed in early stages of the study that many frameworks fulfil those requirements, the long list of candidates had to be sorted. The deciding factor for this selection was usage, awareness of and positive sentiment towards tools among developers because the evaluation of mainstream and general-purpose frameworks appear more valuable than lesser known or specialised tools. A ranking of the most-used JavaScript front-end frameworks of 2023 (Devographics, 2024) lists the four frameworks with the most developers who have used it before: React (84%), Vue.js (50%), Angular (45%) and Svelte (25%). In addition, Astro was chosen for its especially high awareness among the category “other front-end tools” (30%), as well as its usage (19%) and interest (62%) in the category “meta-frameworks”. From the last category of tools, two other frameworks were selected: Next.js and Nuxt. Both tools are highly-used frameworks and have the appearance and goal of improving React and Vue.js, respectively. For this reason, they are interesting choices for this study. All selected frameworks fulfil the requirements. The application is implementable with all frameworks or intended addition of tools. Next.js and Nuxt require the usage of React or Vue tools and dynamic components cannot be written in pure Astro. Astro intends the usage of other frameworks to implement so-called “islands”. For those components, React was chosen for its top usage rate.

Other frameworks were also considered for selection. Solid and Qwik seemed fitting candidates in this study because of high interest among developers and apparent potential for fast performance of their end product. Additionally, from the ranking of most-used front-end frameworks Preact was considered with a usage percentage of 13%. Ultimately, all three were not chosen because of negative sentiment or low usage among developers. This concludes the framework selection for this study. Table 1 list the selection and categorizes them into groups with and without CSR and SSR. It also states whether the developer for the application had any previous experience working with the framework. This information is important for the unintended performance optimizations and can later be used for interpretation of the frameworks performance measurements. Plus, it will influence the assessment of ease of use and developer experience.

3.3 Hosting Environments

After designing the application, the next step in the study process was to decide on where the application is to be hosted for testing. Network delay is a great part of render delay and performance issues because loading files in sequence will block rendering if parsing documents and executing code is dependent on

Framework	CSR	SSR	Previous Experience
Angular	yes	no	yes
Astro	yes	yes	yes
Next.js	no	yes	no
Nuxt	yes (generate)	no (build)	no
React	yes	no	yes
Svelte	yes	no	no
Vue.js	yes	no	yes

Table 1: List of selected frameworks. Items with both CSR and SSR render some pages or components upon request, but also require CSR

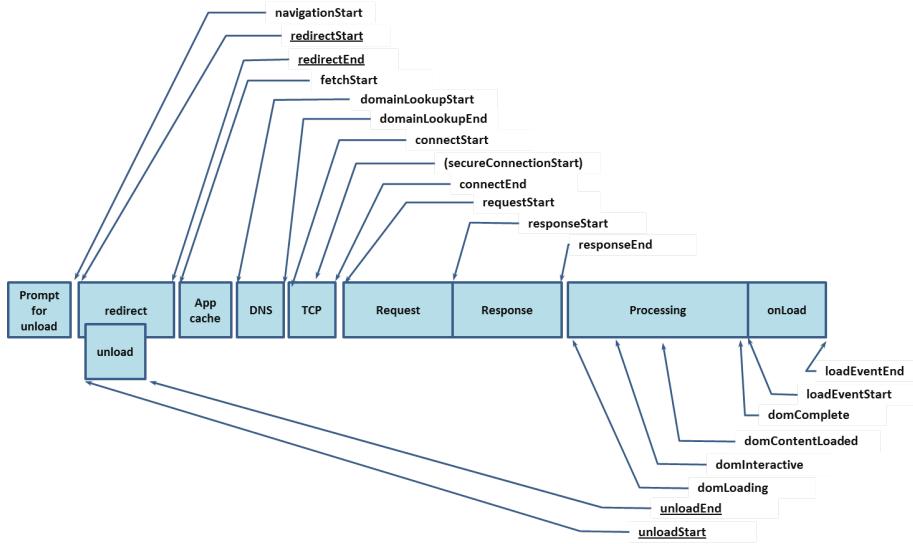


Figure 4: Timing attributes defined by `PerformanceTiming` interface and the `PerformanceNavigation` interface

network requests. The request delay is based on the speed of the web server, the size of the generated file, request and response and the network speed. Therefore the time needed for fulfilling network requests should be considered in the choice of hosting environment or service.

Figure 4 illustrates how a slow network may delay the rendering process of a webpage. The tests for this study shall cover real-world hosting using publicly available services and local hosting to both test the network delay and test the application without interference of network speeds. Additionally, tests can not be done only on local servers because tests shall include timings before `responseEnd`. Requirements for the distant hosting environment or service are threefold. The service shall have “active server capabilities”, meaning capabilities that exceed pure static files server functions for Server-side Rendering and similar functionalities. Furthermore, it is required to be a widely used hosting service to ensure the real-world applicability of the study. Since this requirement

is not clearly applicable, it is considered a guideline. Lastly, to be applicable for small projects as well as established larger websites the service chosen for the study is required to support free usage and integration into a Continuous Integration and Continuous Delivery (CI/CD) configuration because it is a widely used development practice. As such, the integration is important to require the least possible manual configuration for hosting the application because this study is not supposed to be about the configurability. Rather, the study shall focus on the "out of the box" performance of the frameworks. Continuing with that sentiment, the optimization and therefore configuration of the hosting environment is not part of this work. This is the methodology for answering the question: With which framework do developers get the best performance for their web applications without spending much or any time with optimization and configuration?

Based on these considerations and personal experience with the service, Vercel was chosen for hosting the application for this study. Vercel supports predefined configurations and automatic recognition of all frameworks chosen for this study. Also, Vercel projects integrate seamlessly into a CI/CD process based on its integration with Github. A Github repository was created for each framework and connected to a Vercel project. During initialization of the Vercel projects and first preliminary tests, one problem with Vercel's free account quickly became apparent: The bandwidth limitation of 100GB per month and account was reached after two weeks of testing unoptimized and unfinished versions of the applications with large image and video files. Because no information was found on the effect of a reached limit, the account was deemed dead for the month. The solution to this problem was the creation a second free Vercel account and the plan to create another account every time the limit would be reached in the future, which it did not.

The second hosting environment for this study is hosting the application locally on the testing machine. This environment ensures minimal network load times and eliminates every other connected delays such as resolving domain names. If the framework supports a "preview" mode, it was used for hosting the application. Otherwise, the application would be build and hosted using the `serve` command or the active server would be started with `node <filename>`. If neither of the two options would be available, the "dev" mode of the application would be used and tested. Table 2 lists the used commands for building and starting the webserver per framework.

3.4 Performance Metrics

The load time and reactivity of a web page and its user interface decreases user retention and continuing user actions over time independently from the content. The "reaction time" is interpreted in three separate ways for this study: The page load time, meaning the time from navigation start to DOM mutation, the time from a state change, e.g. data query end, to DOM mutation, here called component load time, and the time between a user input to finished DOM mutation, called component update time for this study. Nearly most of these times can be combined from or described using navigation events (see figure 4). These timing categories are not exclusive, but measurements for these time categories do overlap.

Naturally, other metrics than the navigation timings were also considered.

Framework	Build Command	Host Command
Angular	<code>ng build</code>	<code>serve</code>
Astro	<code>astro build</code>	<code>astro preview</code>
Next	<code>next build</code>	<code>next start</code>
Nuxt	<code>nuxt build</code> <code>nuxt generate</code>	<code>nuxt preview</code> <code>nuxt preview</code>
React	<code>react-scripts build</code>	<code>serve</code>
Svelte	<code>vite build</code>	<code>vite preview</code>
Vue	<code>vite build</code>	<code>serve</code>

Table 2: Build and host command for each used framework as used for testing the applications hosted locally

From the list of measurements in Lighthouse reports (see chapter 3.5), sublists with relevant metrics were created to properly represent the time measurements of the described render sections and DOM mutation events. These reports cover the initial load of a page and visual content presentation after initial load. None of the Lighthouse metrics cover the time of DOM mutations after user input events. Therefore, yet additional measurements have to be considered to describe the performance of mutations. To this end, some self-written code is injected through Playwright (see chapter 3.5) to measure the time of updates to the DOM. The following sections describe which measurements are needed for each render section in detail.

3.4.1 Page Load Times

In the context of this study, the first contact point for a user to a web page is considered to be the first page load or initial page load. Within the initial load, the user’s main expectations and goals are assumed to be finding a page with the wanted information or input rather than finding the information itself. As a result, the aim of the client’s browser and render engine for this first time frame, called “page load” here, is to both parse HTML and project the content of the page onto the DOM. In order to focus on this time frame, these metrics describe the application’s performance.

- **Total Byte Weight (TBW):** The total size of files or content of response directly increases either the App Cache time between `fetchStart` and `domLoading` or `domContentLoaded` if the resource can be cached in the client or the response time between `responseStart` and `responseEnd` otherwise.
- **Time To First Byte (TTFB):** The time between `navigationStart` and `responseStart`. Most of the network delay can be described by the TTFB. Often inaccurately paraphrased as “ping”.
- **Time To Interactive (TTI):** The time until the page can be interactive, described by the DOM’s loading state “interactive”. By navigation events described as the time between `navigationStart` and `domInteractive`. Notably, the timing of `domInteractive` is not reliable because a DOM

may become interactive, but the browser may not be interactive yet. Additionally, resources may still be loading. For example, a DOM from a HTML skeleton may be “interactive” after a few milliseconds, but no content may be rendered for the user to see, because CSR code is still loading (Web Hypertext Application Technology Working Group, 2024).

- **DomContentLoaded:** Similar to TTI, DomContentLoaded measures the time between `navigationStart` and `domContentLoaded`. At this point in time, “all subresources apart from `async` script elements have loaded” (Web Hypertext Application Technology Working Group, 2024). A large difference between TTFB and DomContentLoaded indicates a great size or at least long load time of subresources.
- **LoadEventEnd:** Total time spent immediately after initial load of a page until the DOM’s `onload` event is finished. This is the time from `navigationStart` to `loadEventEnd`. The time represents both the capability of the used framework to optimize the usage of a client’s and network’s resources on initial load and the prioritization of JavaScript execution by splitting not immediately needed code into `async` scripts.
- **Total Blocking Time (TBT):** The total time spent by a browser with parsing and optionally resources that block the rendering process from finishing. This includes stylesheets and scripts without the `async` or `defer` tag. The metric directly represent the time before the browser can fulfil the user’s goal on initial load.
- **Last Visual Change (LVC):** Time from `navigationStart` until the last visual change above the fold, meaning within the viewport of the user.
- **Largest Contentful Paint (LCP):** The time between navigation to the page and the time of rendering for the visually largest text or image element in the user’s viewport (Google, 2020). Optimization of this metric requires and understanding of the page’s content and element size within the viewport.

From this list of relevant metrics, some expectations can be formulated before testing for them. First, TBT is most likely slower with CSR frameworks because the code execution filling the HTML skeleton takes some time that is not necessary in client with SSR and Prerendered pages. On delayed pages this difference is expected to be very slight or nonexistent. Second, the LCP probably will not differ across frameworks, but naturally across pages. In contrast, if a framework does create a faster result for its LCP, it is expected to be a SSR or Prerendering framework because of its expected shorter TBT. Third, CSR frameworks differ from SSR and Prerendering frameworks by Total Byte Weight similar to Largest Contentful Paint. Although the HTML document is much slimmer with CSR, the JS files are expected to be equally larger than server-side rendered and prerendered pages. It is probably nearly equal in sum because the byte size of the page is likely mostly made up from multimedia files such as images and videos, CSS and JavaScript files. Fourth, the selected frameworks should be inversely separable into groups by the Time To First Byte. Most likely CSR and Prerendering frameworks will be faster for this metric because the web server can serve as a static fileserver and does not have to execute any additional

code. Fifth, because CSR pages consist of only nearly empty HTML skeletons and links to JS and CSS files, the TTI is expected to be much faster for CSR pages. Lastly, the timing of the `loadEventEnd` is not clearly predictable before testing. The only expectation is that newer framework perform better in this metric simply because they are newer and are expected to make optimizations that go into a faster parsing and rendering of a web page.

With these expectations it would be most interesting to see the differences between CSR and SSR frameworks. From the list of selected frameworks for this study, those frameworks with direct competitors are Nuxt compared with Vue.js as well as Next.js in comparison to React. Additionally, Angular and Svelte in the group of CSR frameworks shall be compared with the SSR framework group with Astro.

3.4.2 Component Load Times

As a second category of relevant metrics, measurements for the separation of the app into components are grouped together. This category is designed to reflect the performance of the JavaScript that was generated by the framework. This stands in contrast to how much content can be rendered by the time of `responseEnd`. To this end, only measurements after `responseEnd` can be taken into consideration. Each mutation from the initial DOM has to be interpreted as a update to a component. The following metrics are part of this category.

- **LoadEventEnd:** The time between `responseEnd` and `loadEventEnd`. It combines all render-blocking parsing and synchronized code execution. Therefore, it is a combined indicator from the code performance and general optimization.
- **Observed First Visual Change (OFVC):** The time of the first visual update from a blank canvas. It is an indicator for the start of visual rendering and a signal to a user that the page is working or loading. For pages with interactive elements, this metric is less important than the TTI.
- **Observed Last Visual Change (OLVC):** The time of the last visual update to a web page. The metric is the most promising for this study as it indicates the end of the perceptible rendering process and therefore perceived load speed.
- **Mutation Times:** Time from initialization of the app with a predetermined HTML element such as `<main>` to a DOM mutation. See section 3.4.3 for more info on this.
- **TBT**
- **TTI**

Based on the intention for testing these metrics, comparing or optimizing JavaScript frameworks, the following expectations were presented before tests. First, prerendered and SSR pages are expected to show a earlier FVC because the execution of any code for delay components can start earlier. This expectation comes from the added code of CSR applications to add static elements to the DOM through JS. Second, CSR applications probably finish their LVC

slightly earlier than other applications. The assumption for this prediction is that every application starts long tasks only after the HTML was parsed which takes longer for SSR or prerendered pages. As a result of these two expectations the observations of a `MutationObserver` most likely have a lower maximum and are less spread out for SSR and prerendered pages, but start later than CSR pages. Third, as described above, the TBT is expected to be slightly later for CSR than for SSR or prerendered applications and fourth, CSR apps should have a slower TTI.

With these metrics, identifying bloated applications and components is the goal. JavaScript that is loaded, parsed and executed that increases the initial load time of a page should be indicated through these tests. Such unnecessary or render-blocking scripts are pointed out through TBT and little difference between FVC and LVC. For example, a script can be considered unnecessary for initial load if it is executed before rendering that only defines functions, initializes objects that are not yet needed or creates a blocking dependency chain, e.g. through importing another script.

3.4.3 Component Update Times

For the third category of relevant metrics, DOM mutation stemming from events triggered by the user are grouped together. These events influence the user experience on the condition that they lead to DOM mutations. Only two kinds of measurements can be made to gain insight into update speed although three measurements are perceivable.

- **User Input Times:** The time of a user input. The kind of user input is not restricted to `onInput` or `onChange` events, but rather any event triggered by the user.
- **State Change Times:** The time a state changes after user input. This is usually not automatically directly testable because the internal functionality of the framework is not always observable.
- **Mutation Times:** Time of a mutation from user input within a predetermined HTML element such as `<main>` to another DOM mutation. A `MutationObserver` is initialized and all mutations are recorded. Designated mutations to the DOM are added child elements, removed child elements and attribute updates (added, edited and removed).

For these metrics no expectations could be formulated before testing because the speed of a mutation is purely based on the implementation of the framework itself. These implementations are not openly accessible without knowledge of the frameworks' source code. Still, some prediction can be made independently from a specific framework. Apps that represent their state in the DOM, e.g. an "edited" state for a user input or an updated value attribute of an `<input>` element, will most likely have...

1. more entries in the recorded DOM mutations and...
2. a later last entry in the recorded DOM mutations.

Also, the implementation of the app shows differences here as additional elements, such as <div> elements as wrappers for each component can influence the time and number of updated elements in either direction, dependent on the use case. To summarize some comparisons between frameworks or groups of frameworks, the most appealing for the evaluation are the following.

1. **CSR - SSR:** Before testing, differences between CSR and prerendered pages are expected, but the metrics and amount of differences are a probable subject of interest. Because there is no perceivable differences between prerendered pages and server-side rendered pages from a client perspective, they are grouped together in this context.
2. **Angular - React - Vue:** Because these CSR frameworks have been competing for **X** years at this point and they are still the most famous front-end frameworks, the comparison of these frameworks is relevant for this study.
3. **Nuxt - Vue.js:** As a next generation of the Vue.js framework, the actual performance increase of Nuxt is interesting for developers.
4. **Next.js - React:** Same as above
5. **Vue-based - React-based:** Because a direct comparison of frameworks based on React and based on Vue.js is possible with multiple candidates, a difference in performance is an actual relevant factor for the choice between the ecosystems.
6. **Svelte - Astro:** As the most modernly popular frameworks in the selection of frameworks, Astro and Svelte have the potential to both outdo their contenders and outdo each other. This comparison is most interesting for fans of new tools and the development teams of the frameworks themselves.

3.5 Testing Tools

In order to test for these metrics, a set of multiple testing tools is needed. These testing tools are required to cover the measurements described above and the tools have to work with similar configuration for all selected frameworks. Test reports have to be generated in a machine-readable format in order to evaluate the results and create aggregate metrics from them. This is a requirement because from previous experience it is known that performance values in the web development context have a considerable variance. To this end, two different tools for automating tests were chosen:

1. **Lighthouse CLI:** The Lighthouse CLI makes it possible to automate the generation of Lighthouse reports. Tests for these reports combine measurements with weights in categories and reduce them to a single score, as well as five main category scores. These categories are performance, accessibility, best practices, SEO and PWA. Additionally, Lighthouse reports contain recommendations for optimizing metrics and increasing the scores. It is a popular tool for measuring the initial page loads, page content and meta information for a web site. Changes after the initial page load are

not possible to test with the Lighthouse CLI. Reports are by default generated as HTML files, but the tool was configured to generate both HTML and JSON reports for this study. Since Lighthouse is designed to test live websites in production, the tool does not support starting a local development server. Testing with Lighthouse therefore needs to include building and hosting the application locally while tests are running.

2. **Playwright**: Playwright is a front-end testing tools for web applications in development. It mainly supports checking page content, but also supports the execution of injected JavaScript and full control over the browser. This also means that the control over the user inputs enables measurement of timings connected to user behaviour such as clicking links and buttons, hover the mouse over elements or using `<input>` elements. Such options are needed to evaluate the timings of interactive elements. The development-focused design also bears the advantage of its initialization being included in some framework's initialization options. Both Svelte and Vue.js support installing and initializing configuration for Playwright in their own initialization (see chapter 4 for more on this). Similar to Lighthouse, reports can be created as HTML and JSON files. For this study, only JSON reports were used for the results, but HTML reports were used for debugging tests.

Although all requirements can be fulfilled with these tools, multiple problems were found with them. Because Lighthouse reports include data that is influenced by all actors and constraints regarding the web page, many factors contribute to the variability of its results. Google (2019) contains a list of sources of variability. The relevant sublist of factors for this study contains for local tests client resource contention, client hardware variability and browser nondeterminism. Client hardware variability is mitigated through the usage of the same client device for all tests. The client device in question is a HP Envy x360 Convertible 15-eu0xxx with an AMD Ryzen 5 5500U processor and 16GB RAM. The operating system on the device is Windows 11 Home (Version 10.0.22631) during testing. Client resource contention could not be fully mitigated. Attempts to keep a lid on client resources was killing the most hardware intensive background tasks and services on the test machine before starting tests. Browser nondeterminism was taken into account and adopted as a test dimension because the target group of an application should be factor for the choice of framework, especially for purely desktop or mobile applications. To this end, tests were executed with the most commonly used browsers wherever possible. For Lighthouse tests, such an option was not found. Instead, all tests were explicitly executed on Google Chrome for desktop. A Lighthouse report was not generated on other browsers.

For tests on a distant server, other factors contribute to the fluctuation of Lighthouse test results in addition. Local network variability, tier-1 network variability and web server variability have to be considered for the tests. The first two could not be mitigated. The internet connection speed at the test location was 100 Mbit/s to simulate common modern consumer internet connections. Web server variability could not be mitigated as well. For this reason, a hosting service was explicitly chosen for all tests to minimize the variability across frameworks (see section 3.3).

For mitigation of all factors of variability, Lighthouse tests were executed 20 times to gain an average of all measurements. The repetitions were configured with the same browser context and web server for local tests for each run. The reason for this decision is that fluctuations based on the first requests within the client or the server should be mitigated with this method.

Two additional problems with Playwright were found before the start of the test phase. The time of injection for JS script could not be properly determined. This fluctuation could not be mitigated. Also, reading data from the window context after the fact proved to be difficult because the context closes after the test ends and the report only contains explicitly tested values. Objects such as the needed navigation timings are no longer available after the fact. The solution to this problem was to attach all necessary information as a file to the report so it is readable after the context closed.

With all tools and workarounds in place, the data needed for the study could be collected. Lighthouse covers TBW, TTFB, TTI, TBT, LCP, FVC, OFVC and OLVC whereas Playwright cover all navigation and HTML event times, namely DomContentLoaded, LoadEventEnd, user input times, state change times and mutation times.

4 Implementation

4.1 Components

4.2 Tests

5 Evaluation

5.1 Page Load Times

5.2 Component Load Times

5.3 Component Update Times

6 Conclusion

7 Summary

A List of Figures

1	Screenshots of the NotInstagram application's pages (path in parentheses) (Bilder müssen noch geändert werden)	8
2	Pages, Components and Services of the NotInstagram application	10
3	Classes used by the NotInstagram services	10
4	Timing attributes defined by PerformanceTiming interface and the PerformanceNavigation interface	12

B List of Tables

1	List of selected frameworks. Items with both CSR and SSR render some pages or components upon request, but also require CSR	12
2	Build and host command for each used framework as used for testing the applications hosted locally	14

C Acronyms

CI/CD	Continuous Integration and Continuous Delivery.
CLI	Command Line Interface.
CSR	Client-side Rendering.
CSS	Cascading Style Sheet.
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.
PWA	Progressive Web App.
SEO	Search Engine Optimization.
SSR	Server-side Rendering.
SVG	Support Vector Graphic.
TBT	Total Blocking Time.

TBW Total Byte Weight.
TTFB Time To First Byte.
TTI Time To Interactive.

References

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

Google (2019). Lighthouse variability. <https://developers.google.com/web/tools/lighthouse/variability>. accessed=08/01/2024.

Google (2020). Largest contentful paint. <https://developer.chrome.com/docs/lighthouse/performance/lighthouse-largest-contentful-paint>. accessed=07/28/2024.

Web Hypertext Application Technology Working Group (2024). Html living standard. <https://html.spec.whatwg.org/multipage/dom.html#current-document-readiness>. accessed=07/30/2024.