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 9. August 2024

zur Erlangung des akademischen Grades eines Master of Science

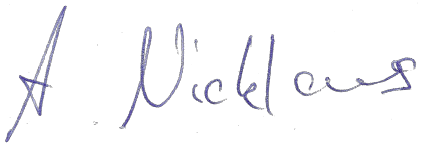
Erst-Prüfer: Prof. Dr. Fridtjof Toenniessen
Zweit-Prüfer: Stephan Soller

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 9. August 2024



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

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 brazingly 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 webiste'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 apparant 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 perceivable 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 are 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 skripts 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 indirecly 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 the performance of the page.

   (a) Components: All pages of the app have to consist of components that encapsulate reproducable HTML snippets and may project data onto the DOM.

   (b) List iteration: Because iterating long lists may decrease performance noticably, 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 intentionaly 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 (MDN Mozilla, 2024b; Google, 2019a). For this purpose, CSS shall be implemented for both pages and components.

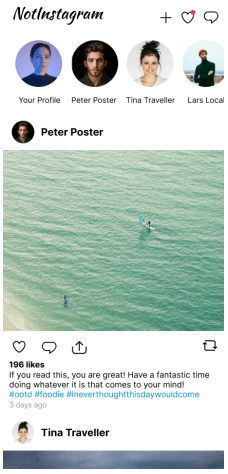
4. **Rendering time**: In addition to page type depending on the time of data load, the machine and time of composing the DOM is dependent on the content availability. For this paper, three different types are considered:
   (a) Client-side Rendering (CSR): The initial request gets a response with a 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 (Instagram from Meta, 2024). "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 latter 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.

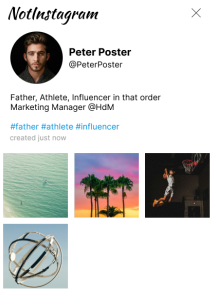
The Create page (see figure 1c) has three parts. The header contains the app's logo and a X icon linking to the feed. A form with three `<input>` elements and a `<button>` element allows the input of an multimedia source (image or video) and a text caption. The multimedia source can either be an URL 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 is 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.

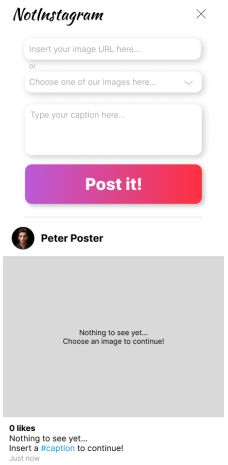
With these pages all page types are covered for testing. The About page and Create page are static, whereas the Feed page and Profile page are partly static (header and footer), but mostly delayed. The Create page is the only page with dynamic content.



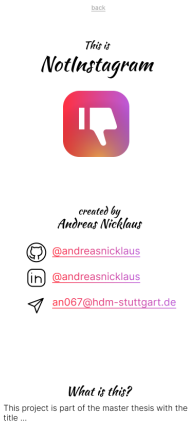
(a) Feed / Index Page (/)



(b) Profile Page (/user/@PeterPoster)



(c) Create Page (/create)



(d) About Page (/about)

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

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 `<img>` tag should be used. For external image links starting with "http://" or "https://" a less optimized or unoptimized `<img>` 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 referres 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.

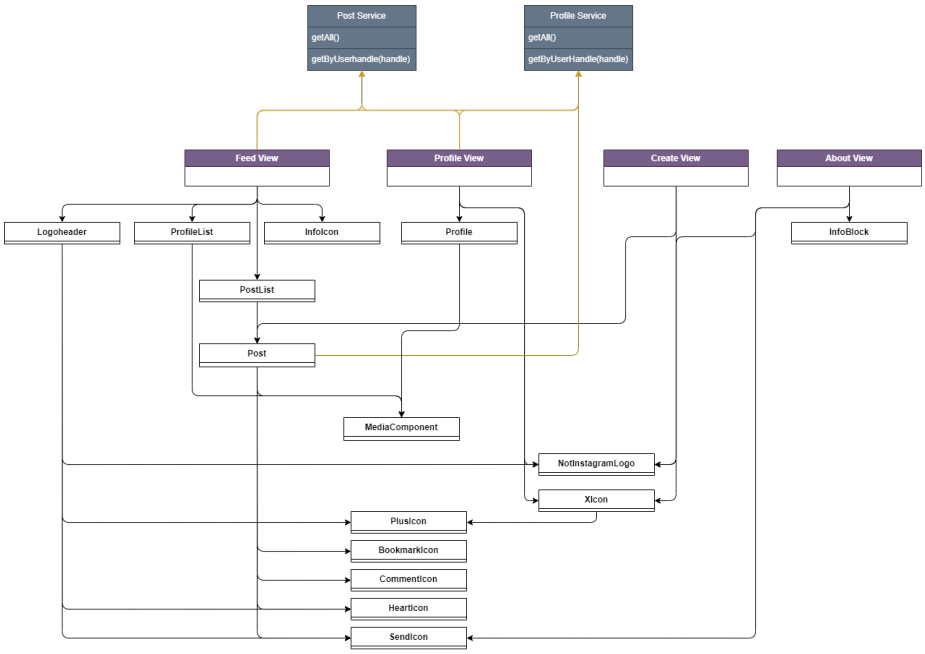


Figure 2: Pages, Components ands Services of the NotInstagram application

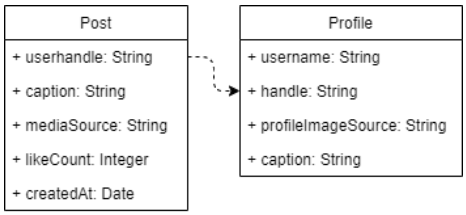


Figure 3: Classes used by the NotInstagram services

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.js 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

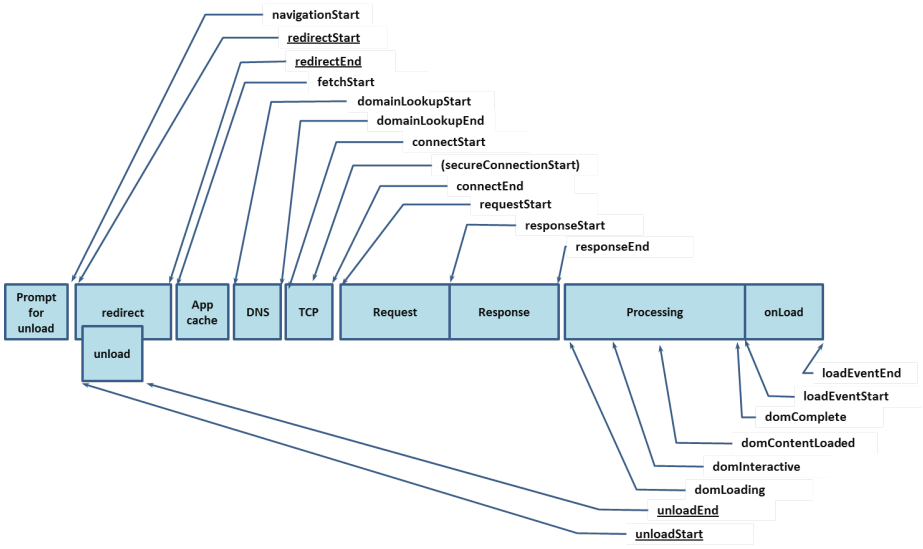


Figure 4: Timing attributes defined by the PerformanceTiming interface and the PerformanceNavigation interface (W3C, 2012)

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 fileserver 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 fo 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	`ng build`	`serve`
Astro	`astro build`	`astro preview`
Next.js	`next build`	`next start`
Nuxt	`nuxt build` `nuxt generate`	`nuxt preview` `nuxt preview`
React	`react-scripts build`	`serve`
Svelte	`vite build`	`vite preview`
Vue.js	`vite build`	`serve`

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 paraphrazed 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 (Web Hypertext Application Technology Working Group, 2024), 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 imidiately 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 priorization 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 seperable 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 itneractive elements, this metric is less important that the TTI.
- **Observed Last Visual Change (OLVC)**: The time of the last visual update to a web page. The metric is the most promisiing for this study as it indicates the end of the perceptable 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 littele 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 event 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 an 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>` elemets 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, deifferences 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.js**: Because these CSR frameworks have been competing for eight years at this point and they are still the most famous front-end frameworks (Devographics, 2024), 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 describe 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 behavior 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 (2019b) 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

This chapter contains details of the implementation and the strategies for creation of the project as well as for the separation of projects for each framework. The goal is to define taken steps to ensure reproducability and tracability of implementation choices and, as a result, interpretability of the results in the following chapter.

The implementation for each framework was started using the official "get started" guide on the framework's website (Google LLC, 2024; Schott, 2024; Vercel, Inc., 2024; Chopin et al., 2024; Meta Platforms, Inc., 2024; Svelte, 2024; You, Evan, 2024). Each website provides a command which creates a project directory and project files. The initialization options for the creation process were chosen with the following rules.

1. The project is to be created as empty as possible to ensure the focus on the framework "as is" rather than how it can be configured. No demo projcet is chosen if an option with fewer preconfigured files is available.

2. No testing tools is to be preconfigured except Playwright. If Playwright is not an option, then no testing tool should be chosen.

3. Otherwise the default options (recommended or first) should be chosen. If "none" is an option, it it should be selected.

After the initialization under these rules, the app's four pages and components as well as routing between the pages were configured. After creation of the Vue.js and React app, each component's template, code and style information was copied from either their Vue.js or React counterparts and adapted to the framework in question to speed up the creation process. Only after this process optimization efforts such as configuring image components (see section 4.1) and adaptation to the hosting environment were performed.

Additionally, project directories were separated into Github repositories. The separation is a requirement for hosting with Vercel as a maximum of three Vercel projects can be hosted from the same repository. This study exceeds this limit. This limiting condition entails that all testing configuration could not be centralized, but had to be duplicated across repositories. The setup of the testing environment has been the last step of the project creation (see section 4.2).

4.1 Components

While most of the design decisions for the components of the application have been made during the design of the application itself, the design choices concerning the implementation of said components are open to adaptation to the framework. The goals for this implementation period are few. First, the implementation for each framework should be as similar to the others as possible, meaning the HTML elements should be the same. Second, the implementation should follow the design language of the framework. Therefore no principles should translate from one implementation to another if they do not fit to the framework's design principles. Third, it has to follow the component design as described in section 3.1. If the design of the example application cannot be followed, changes are to be as minimal as possible. This section describes selected components and code snippets where they are either interesting for the performance, unforeseen choices or where they differ notably inbetween frameworks. The author of this study had had the most experience with Vue.js prior to this project. For this reason, code snippets in Vue.js have the most presentability and code snippets in this paper are shown in Vue.js wherever possible. The components described in this section are the About and the Create page, the Post component because it has two variations and the MediaComponent.

The About page is an interesting case because, as described in section 3.1, it is the only static page of the application. Its components and HTML children are therefore also static. Figure 5 shows a graphical overview of the page's contents from a DOM perspective. Because of its static nature, it is also the only page that can be fully prerendered. Notably, the lower part of the page consists of multiple subcomponents `<Infoblock>` with a title passed as a prop and a paragraph passed in a slot as a HTML child for the component. Functionally, its only purpose is styling and its only effect on the DOM is the addition of a `<h2>` and a `<p>` element. The other imported subcomponents `<NotInstagramLogo>` and `<SendIcon>` are also wrappers for a `<h1>` and a `<img>` element, respectively. Listing 1 demonstrates the static nature of the page view and the hard-coded addition of all text and multimedia in the template.

```
1  <!-- AboutView.vue -->
2  <template>
3    <div id="AboutView">
4      <RouterLink id="top-backlink" class="backlink" :to="{ name:
           'Feed' }"> back </RouterLink>
5
6      <p class="cursive">This is</p>
7      <NotInstagramLogo />
8      <img class="transparent logo" alt="" width="40%" height="240"
           loading="lazy" :src="Logo" />
9
10     <p class="cursive">created by</p>
```

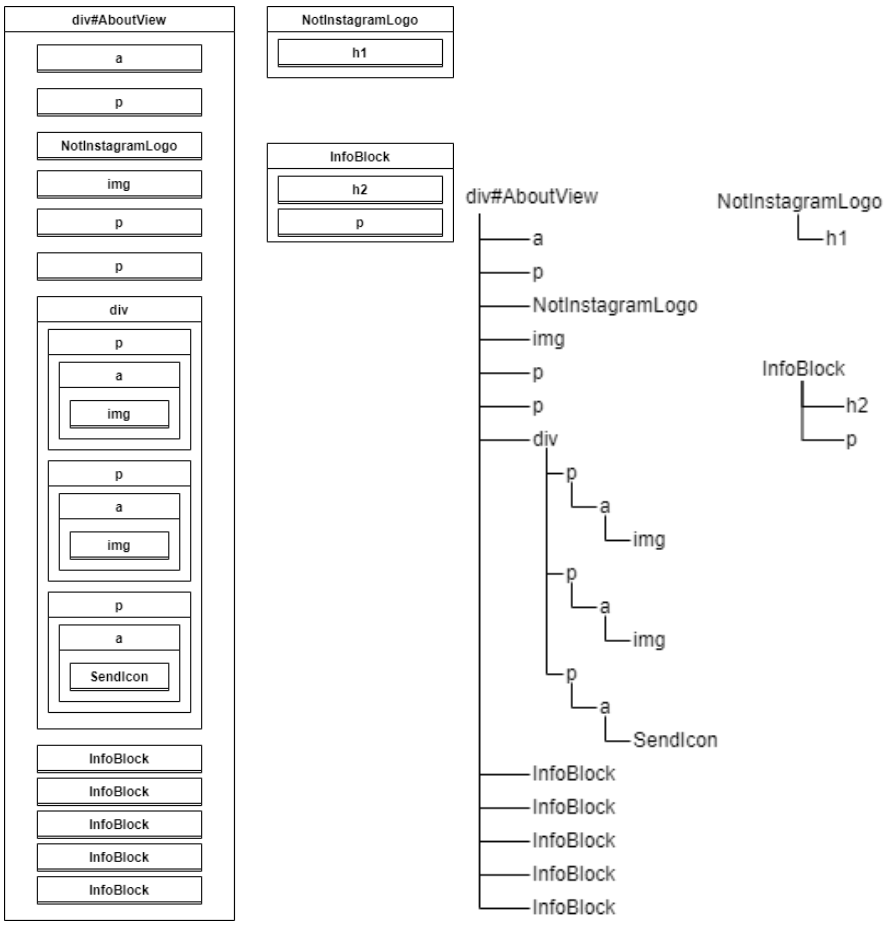


Figure 5: Graphical subdivision of the About page into components (Welches Diagramm ist besser?)

```
11      <p class="cursive big">Andreas Nicklaus</p>
12      <div id="socials">
13        <p>
14          <a href="https://github.com/andreasnicklaus"
                target="_blank">
15            <img class="transparent" width="29" height="29"
                  loading="lazy" :src="Github" />
16            @andreasnicklaus
17          </a>
18        </p>
19        <p>
20          <a href="https://www.linkedin.com/in/andreasnicklaus/"
                target="_blank">
21            <img class="transparent" width="29" height="29"
                  loading="lazy" :src="LinkedIn" />
22            @andreasnicklaus
23          </a>
24        </p>
25        <p>
26          <a href="mailto:an067@hdm-stuttgart.de">
27            <SendIcon/> an067@hdm-stuttgart.de
28          </a>
29        </p>
30      </div>
31
32      <InfoBlock title="What is this?">
33        This project is part of the master thesis by ...
34      </InfoBlock>
35      <InfoBlock title="Placeholder 1">
36        Lorem ipsum dolor sit amet, consectetur adipiscing ...
37      </InfoBlock>
38      <InfoBlock title="Placeholder 2">
39        Pellentesque diam volutpat commodo sed egestas ...
40      </InfoBlock>
41      <InfoBlock title="Placeholder 3">
42        Nec feugiat nisl pretium fusce. Sagittis id ...
43      </InfoBlock>
44      <InfoBlock title="Placeholder 4">
45        Ullamcorper malesuada proin libero nunc. Netus et ...
46      </InfoBlock>
47      <InfoBlock title="Placeholder 5">
48        Adipiscing elit pellentesque habitant morbi ...
49      </InfoBlock>
50
51      <RouterLink id="bottom-backlink" class="backlink" :to="{ name:
            'Feed' }"> back </RouterLink>
52    </div>
53  </template>
```

Listing 1: About page in Vue.js

The Create page poses an opposite to the About page. In contrast to a static page with non-changing content, the purpose of the Create page is to preview a new post. Its purpose is to update after user input. Listing 2 and 3 show the implementation of the Create page in Vue.js. The data of the component has four dynamic parts: The options and the choice for the selection of the post image in a `<select>` element, the caption of the new post and the media URL for the `<input>` element. The last data point for the component is the user handle, which is static for the preview in this example application. The computed property mediaSource (see listing 3, line 40) represents the logical choice between the media selection and source URL for the multimedia file

in the previewed post. This template contains a static `<header>`, the `<form>` with dynamic attributes and a Post component. This subcomponent hast to be dynamic and reactive to its props as they are changing throughout the process of post creation.

```
<!-- CreateView.vue -->
<template>
  <header>
    <RouterLink :to="{ name: 'Feed' }"> <NotInstagramLogo/>
        </RouterLink>
    <RouterLink :to="{ name: 'Feed' }"> <XIcon/> </RouterLink>
  </header>

  <form id="newPostForm" action="" method="post">
    <input type="url" name="mediaUrl" id="mediaUrl"
        placeholder="Insert your media URL here..."
        v-model="mediaUrl" />
    <p>or</p>
    <select name="preloaded-image" id="preloaded-image"
        v-model="mediaChoice">
      <option value="">Choose one of our media files
          here...</option>
      <option v-for="media in preloadedMedia" :key="media"
          :value="media">
        <span>{{ media }}</span>
      </option>
    </select>
    <textarea name="caption" id="caption" cols="30" rows="3"
        placeholder="Type your caption here" v-model="caption"/>
    <button type="submit" :disabled="!(caption && mediaSource)">
        Post it! </button>
  </form>

  <hr />

  <Post :userhandle="userhandle" :caption="caption" :likeCount="0"
      :mediaSource="mediaSource" :hideActionIcons="true" />
</template>
```

Listing 2: Create Page in Vue.js (Template)

```
// CreateView.vue
export default {
  name: "CreateView",
  data() {
    return {
      preloadedMedia: [
        "canyon.mp4", "abstract-circles.webp", ...
      ],
      userhandle: "@you",
      caption: "",
      mediaUrl: "",
      mediaChoice: "",
    };
  },
  computed: {
    mediaSource() {
      if (this.mediaUrl) return this.mediaUrl;
      return this.mediaChoice;
    },
  },
```

```
45 };
```

Listing 3: Create Page in Vue.js (Script)

Listings 4 and 5 show the implementation of the Post component in Vue.js. It requires seven props for the five data points of a post (see figure 3) and two additional props for the control over the design and loading behavior of the post's image or video. Additionally, the `mounted` method loads the user data through the `ProfileService` (see listing 5, line 43). The template of the component uses MediaComponent twice, once for the profile picture and once for the post image or video. The attributes for the profile picture are mainly static because the user data is not edited through the create form. The attributes of the post multimedia except the class, width and height are dynamic and editable. Additionally, the projection of the post's caption onto the DOM is dynamic. Every time the caption changes, the string is split by whitespaces and each word is projected onto a `<span>` element, so it can be styled as an hashtag if applicable. Afterwards, the list of `<span>` elements is joined using whitespaces. The purpose of this method for the projection of the caption is only for the styling of hashtags.

```
1  <!-- Post.vue -->
2  <template>
3    <div class="post">
4      <RouterLink v-if="user" :to="{ name: 'Profile', params: {
           handle: userhandle } }" class="postUserInfo" >
5        <MediaComponent class="profileImage"
             :src="user?.profileImageSource" alt="" width="44"
             height="44" />
6        <span class="username">{{ user?.username }}</span>
7      </RouterLink>
8      <MediaComponent class="postMedia" :src="mediaSource"
           :alt="caption" width="100%" height="100%"
           :eagerLoading="eagerLoading" />
9      <div class="actionIconRow" v-if="!hideActionIcons">
10       <div class="leftActionIcons">
11         <HeartIcon />
12         <CommentIcon />
13         <SendIcon />
14       </div>
15       <BookmarkIcon />
16     </div>
17     <p class="likeCount">{{ likeCount }} likes</p>
18     <p class="caption">
19       <span v-for="(word, i) in caption.split(' ')" :key="i"
             :style="word.startsWith('#') ? 'color: #0091E2' : ''">
20         {{ word }}{{ " " }}
21       </span>
22     </p>
23     <p class="creationTime">{{ creationTimeToString }}</p>
24   </div>
25 </template>
```

Listing 4: Post in Vue.js (Template)

```
26 // Post.vue
27 import ProfileService from "@/services/ProfileService";
28
29 export default {
30   name: "Post",
```

```
31    props: {
32      userhandle: String,
33      caption: String,
34      mediaSource: String,
35      likeCount: Number,
36      createdAt: Date,
37      hideActionIcons: Boolean,
38      eagerLoading: { type: Boolean, default: false },
39    },
40    data() {
41      return { user: null };
42    },
43    mounted() {
44      ProfileService.getByHandle(this.userhandle).then(
45        (user) => (this.user = user)
46      );
47    },
48    computed: {
49      creationTimeToString() {
50        ...
51      },
52    },
53  };
```

Listing 5: Post in Vue.js (Script)

Because the creation of such a dynamic component is an intended use case for Angular, Next.js, Nuxt, React, Svelte and Vue.js, their implementation is not unusual (see listings 2, 3, 31, 32, 33, 34, 35, 36, 37 and 38). Astro poses as an opposite to this. Because dynamic or reactive components are not implementable natively as Astro components, another framework has to be used in Astro Islands. For this reason, other components had to be invented in addition to the components as described in figure 2. `CreateForm` encapsulates the dynamic parts of the Create Page. It is a React component with the form and post preview. Because Astro components cannot be used in Islands, every subcomponent used here had to be implemented with React as a duplicate to an Astro component.

Listings 6, 7 and 8 show the implementation of this unique design in Astro. The Create component imports and inserts the React component `CreateForm` into HTML snippets for the page and marks it as a CSR component with `client:load` (see listing 7, line 18). From this component inwards, all HTML is generated on the client and purely as a React application. The CreateForm itself contains the form and Post subcomponent. Because of this structure, the components Post, MediaComponent, BookmarkIcon, CommentIcon, HeartIcon and SendIcon had to be implemented as Astro components and as React components. Figure 6 shows this updated component structure with Astro Islands.

```
1  // create.astro
2  export const prerender = false;
3  import HtmlLayout from "../Layouts/HtmlLayout.astro";
4
5  import NotInstagramLogo from
       "../components/NotInstagramLogo.astro";
6  import XIcon from "../components/icons/XIcon.astro";
7  import CreateForm from "../components/CreateForm.jsx";
8  import React from "react";
9
```

```
const userhandle = "@you";
```

Listing 6: Create page in Astro (Frontmatter)

```
<!-- create.astro -->
<HtmlLayout>
  <header>
    <a href="/"> <NotInstagramLogo /> </a>
    <a href="/"> <XIcon /> </a>
  </header>
  <React.StrictMode>
    <CreateForm userhandle={userhandle} client:load />
  </React.StrictMode>
</HtmlLayout>
```

Listing 7: Create page in Astro (HTML)

```
// CreateForm.jsx
import { useState } from "react";
import styles from "./CreatePost.module.css";
import Post from "./Post";

const preloadedMedia = [
  "canyon.mp4", "abstract-circles.webp", ...
];

const CreateForm = ({ userhandle }) => {
  const [caption, setCaption] = useState("");
  const [mediaUrl, setmediaUrl] = useState("");
  const [mediaChoice, setmediaChoice] = useState("");

  function mediaSource() {
    return mediaUrl || mediaChoice;
  }

  return (
    <>
      <form id={styles.newPostForm} action="" method="post">
        <input type="url" name="mediaUrl" id={styles.mediaUrl}
            placeholder="Insert your media URL here..."
            value={mediaUrl} onChange={(event) =>
            setmediaUrl(event.target.value)} />

        <p>or</p>

        <select name="preloaded-image" id={"preloaded-image"}
            value={mediaChoice} onChange={(event) =>
            setmediaChoice(event.target.value)}>
          <option value="">Choose one of our media files
              here...</option>
          {preloadedMedia.map((media) => (
            <option key={media} value={media}>{media}</option>
          ))}
        </select>
        <textarea name="caption" id={styles.caption} cols="30"
            rows="3" placeholder="Type your caption here"
            value={caption} onChange={(event) =>
            setCaption(event.target.value)}/>
        <button type="submit" disabled={!(caption &&
            mediaSource())}> Post it! </button>
      </form>
```

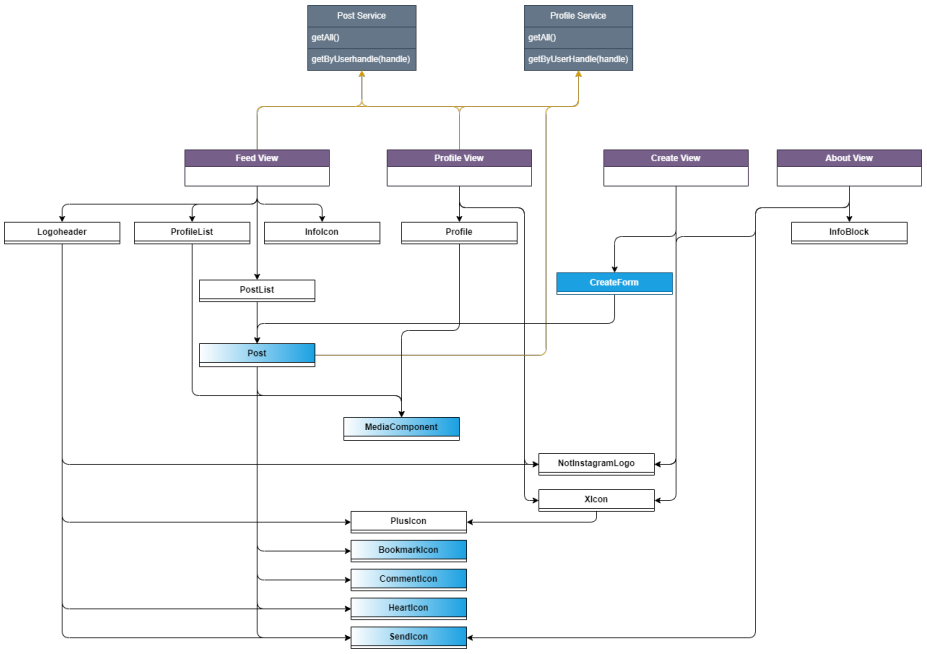


Figure 6: Adapted component structure for Astro Islands (React components are marked blue, duplicate components are white and blue)

```
35          <Post userhandle={userhandle} caption={caption}
                likeCount={0} mediaSource={mediaSource()}
                hideActionIcons={true} />
36        </>
37      );
38    };
39
40    export default CreateForm;
```

Listing 8: Create form in Astro

The MediaComponent is a presenter component for multimedia content, namely an image or a video. It is used within the ProfileList, Profile and Post components (see figure 2). As described in section 3.1, the main use of this component for a developer is to centralize the optimization of multimedia files and to ensure its correct size and style. As such, it is a catchall component for many kinds of multimedia sources. Listings 9 and 10 show parts of its implementation in Vue.js.

```
1   <!-- MediaComponent.vue -->
2   <template>
3     <img ref="image" class="postMedia"
          v-if="mediaSource.endsWith('webp')" :alt="alt"
          :width="width" :height="height" :loading="eagerLoading ?
          'eager' : 'lazy'" :src="mediaSource" />
4     <video ref="video" class="postMedia"
          v-else-if="mediaSource.endsWith('mp4')" :width="width"
          :preload="eagerLoading ? 'auto' : 'metadata'" controls
          controlslist="nodownload,nofullscreen,noremoteplayback"
          disablepictureinpicture loop muted >
```

```
5       <source :src="mediaSource" type="video/mp4" />
6     </video>
7     <div v-else class="mediaError" ref="mediaError">
8       <p>Nothing to see yet...<br />Choose an image to continue!</p>
9     </div>
10  </template>
11
12  <script>//see listing below</script>
```

Listing 9: MediaComponent in Vue.js (Template)

```
13  // MediaComponent.vue
14  import { playPauseVideo } from "@/utils/autoplay.js";
15
16  export default {
17    name: "MediaComponent",
18    props: {
19      src: { type: String },
20      alt: { type: String,  default: "" },
21      width: String,
22      height: String,
23      eagerLoading: { type: Boolean, default: false },
24    },
25    computed: {
26      mediaSource() {
27        if (
28          this.src == null ||
29          this.src == undefined ||
30          this.src.startsWith("http")
31        )
32          return this.src;
33        return new URL(`/src/assets/stock-footage/${this.src}`,
              import.meta.url)
34          .href;
35      },
36    },
37    mounted() {
38      const video = this.$refs.video;
39      if (video) playPauseVideo(video);
40    },
41  };
```

Listing 10: MediaComponent in Vue.js (Script)

First, the component takes five props that can be passed to it as HTML attributes (see listing 10, line 18 ff.). The `src` string contains either the file name or URL to the file. The `alt` prop is the alternative text for an image to simple pass to the `alt` attribtue of the `<img>` tag, as well as the width and height of the image or video. These props are primarily needed for optimization of layout shifts and to optionally tell the browser which image variant is needed from a source set on the page. Lastly, the `eagerLoading` prop is a boolean indicator for whether the file need to be loaded first for images or preloaded fully for videos.

Second, the computed property `mediaSource` returns the correct link to either the image or video source based on the start of the `src` prop. This allows the component to identify faulty or external source URLs and only import needed local multimedia files. This implementation design is unique to Vue.js and Nuxt. Looking at the implementation in React and Next.js, the same effect is achieved through the `useState` and `useEffect` hooks. The `ngOnChanges`

hook is used in Angular. In Svelte, the `mediaSource` is defined with a leading `$:`, making it reactive. Because of its non-dynamic nature the native Astro component defines `mediaSource` statically server-side. On the other hand, the dynamic component uses the same implementation as the React application.

Third, every framework uses conditional rendering to project either an image, a video or an error message onto the DOM. Additionally, the Svelte component checks another condition: external and internal images. For image source strings starting with "http", an HTML-native `<img>` element is used, whereas the Svelte-native `<enhanced:img>` tag is used for all other images. The remaining frameworks use either one or the other method to insert images. Vue.js, React and Angular do not support enhanced image elements. These frameworks only include images using the `<img>` tag. In contrast, Astro, Next.js, Nuxt and Svelte do have commponents that improve the performance of image elements. Astro natively supports an `<Image>` component that outputs an `<img>` tag with optimized attributes. Next.js comes with another `<Image>` component that optimizes images with a predefined width and height and Nuxt has a `<NuxtImg>` component to optimize images and define presets for its images. Svelte is the only one of that group that does not support full URLs to be passed to its image component.

Fourth, the attributes of the `<img>` elements are designed to optimize their load performance, size and image quality. While no way to optimize the size and quality of the source of simple `<img>` elements is apparent, the load performance is adapted to the usage of a `<MediaComponent>`. The first Post of a PostList is always eager-loaded, whereas all other images are lazy-loaded. The size of the bounding box of the image is also defined in order to prevent layout shifts during or after the loading of the image. Enhanced image components are configured to ideally optimize the size and quality of the requested image, as well as to insert placeholder images if possible.

The `<video>` elements are designed to optimize the load behavior of the browser and to change the default presentation and styling. Each video has a defined width and height, controls and playback behavior. In order to come as close to the application's model, Instagram, videos should autoplay, but stay muted. Each single behavior is a single attribute to set, but autoplaying every video requires every video to be loaded on page load. This network load bears a performance decrease. For this reason, only the metadata is preloaded unless it is the first post in the PostList. To ensure the wanted autoplay feature, each `<video>` element is referenced using the framework and custom code ensures videos play when they are in the viewport and pause when they are outside of it. This is achieved using an `IntersectionObserver` (MDN Mozilla, 2024a).

4.2 Tests

As described above, the implemention of tests and test configuration were the last step in the process of project creation. As such, tests were either left "as is" or not configured until the application could be considered "done". The test suite for this project can be split into two halves: Lighthouse CLI automation and Playwright tests (see section 3.5). Lighthouse is used to mostly cover aggregate metrics, while Playwright is used to export navigation and HTML event times.

To this end, a script was written to automate the execution of Lighthouse tests and to store Lighthouse reports in a comprehensive way. Listing 11 shows

parts of the implementation of the testing script. It reads project configurations from an external configuration file and iterates over them, executing the tests for every framework multiple times. Listing 12 contains an excerpt of the configuration file. Every project is built and hosted, if either a host command, e.g. using `npm run <script>`, or a serve command using `serve` is defined in the configuration file. While the application is hosted, a headless Google Chrome browser window is launched and multiple lighthouse tests are preformed. The report is generated using the URL as it is specified in the configuration and with static options. These options define among other things that an HTML report is to be generated, only performance metrics are to be collected and the HTTP status code is to be ignored. The last option is necessary because web servers started using `serve` return a 404 status code for files that do not exist in the hosted directory. For applications that rely on `index.html` to be returned if a requested resource is not available, this behavior is not desired. For example, requesting the defined path `/about` results in a 404 code with the `index.html` file as the response body. Withouth the option `ignoreStatusCode: true`, the Lighthouse test would fail as the page is considered to be unavailable.

```
1  // testing-script/index.js
2  import fs from 'fs';
3  import lighthouse from 'lighthouse';
4  import * as chromeLauncher from 'chrome-launcher';
5  import config from "./config.js"
6  import { exec, spawn } from 'child_process';
7
8  import { createLogger, transports, format } from "winston"
9
10 const logger = createLogger({...})
11
12 //...
13
14 function dateToUriSafeString(d) {...}
15
16 function build(projectConfig) {
17   return new Promise((resolve, reject) => {
18
19     if (projectConfig.buildCommand) {
20       logger.info("Starting build...")
21       exec(`${projectConfig.buildCommand}`, { cwd:
             projectConfig.projectPath, maxBuffer: 1024 * 1024 * 1024
             }, (error, stdout, stderr) => {
22         // ...
23       })
24     }
25     else {
26       logger.info("Skipping build because buildCommand was not
             specified")
27       resolve()
28     }
29   })
30
31 }
32
33 async function stopServer(hostProcess, projectConfig) {
34   return new Promise((resolve, reject) => exec(`taskkill /pid
         ${hostProcess.pid} /f /t`, (error, stdout, stderr) => {
35     // ...
36   }))
37 }
```

```
38
39  for (let projectConfig of config.projects) {
40    logger.info(`Testing project ${projectConfig.name}`)
41
42    // BUILD PHASE
43    await build(projectConfig)
44
45    // STARTING HOST PROCESS
46    // ...
47
48    if (serverCommand) {
49      logger.info("starting server...")
50      const [command, ...options] = serverCommand.split(" ");
51      hostProcess = spawn(command, options, { cwd:
          projectConfig.projectPath, shell: true })
52    }
53    else {
54      if (projectConfig.url.startsWith('http://localhost')) throw
          new Error("Server was not properly configured. Check
          prefferedServeCommand, hostCommand and/or serveCommand for
          project", projectConfig.name)
55      else ("Server was not started because no command was
          specified")
56    }
57
58    // ...
59
60    // START LIGHTHOUSE TEST
61    logger.info("Starting lighthouse tests...")
62    const url = projectConfig.url
63    const chrome = await chromeLauncher.launch( { chromeFlags:
        ['--headless'] } );
64    const options = { logLevel: 'warn', output: 'html',
        onlyCategories: ['performance'], port: chrome.port,
        ignoreStatusCode: true };
65
66    for (const route of (projectConfig.paths || ["/"])) {
67      // ...
68
69      for (let i = 0; i < config.runsPerProject; i++) {
70
71        const runnerResult = await lighthouse(url + route, options);
72
73        const { report: reportHtml, artifacts, lhr } = runnerResult;
74        const { timing, fetchTime, categories, ...rest } = lhr
75
76        fs.mkdirSync(`${projectConfig.reportDirectory}${route == "/"
              ? "/index" : route}`, { recursive: true }, (err) => {
77          if (err) throw err;
78        });
79        fs.writeFileSync(`${projectConfig.reportDirectory}${route ==
              "/" ? "/index" : route}/lighthouse-report-${new
              URL(url).hostname}-${dateToUriSafeString(new
              Date())}.html`, reportHtml);
80        fs.writeFileSync(`${projectConfig.reportDirectory}${route ==
              "/" ? "/index" : route}/lighthouse-report-${new
              URL(url).hostname}-${dateToUriSafeString(new
              Date())}.json`, JSON.stringify({ artifacts, lhr }, null,
              2));
81
82        // ...
83      }
```

```
84
85      // ...
86    }
87
88    await chrome.kill();
89    if (serverCommand) await stopServer(hostProcess, projectConfig)
90  }
91
92  logger.info("ALL DONE")
```

Listing 11: Automation script for Lighthouse tests

```
1   // testing-script/config.js
2   export default {
3     runsPerProject: 20,
4     preferredServeCommand: "serve",
5     projects: [
6       // ...
7       {
8         name: "Svelte on Vercel",
9         reportDirectory:
              "./lighthouse-reports/ig-clone-svelte/vercel",
10        url: "https://ig-clone-svelte.vercel.app",
11        paths: ["/", "/about", "/create", "/user/@PeterPoster"]
12      },
13      // ...
14      {
15        name: "Svelte",
16        projectPath: "../ig-clone/ig-clone-svelte",
17        buildCommand: "npm run build",
18        serveCommand: "npm run preview",
19        reportDirectory:
              "./lighthouse-reports/ig-clone-svelte/localhost",
20        url: "http://localhost:4173",
21        paths: ["/", "/about", "/create", "/user/@PeterPoster"]
22      },
23      // ...
24    ]
25  }
```

Listing 12: Test configuration for Lighthouse tests

Once the test results are available, the relevant metrics are collected, stored in a JSON file and the HTML report is stored as a means to debugging. After the tests are finished and results are stored, the Google Chrome window is killed and the webserver is stopped.

In order to evaluate and summarize the collection of tests performed using the automation script, another script was written so that test summaries are created. This report reader iterates over the list of JSON files and calculates the average per metric, route and project configuration from the configuration file. It makes it easier to compare the test results and interpret the performance of the frameworks (see chapter 5).

Similar to the test method for Lighthouse, Playwright tests can be triggered using a script to unify the output files. Listing 13 shows the implementation of this trigger script. Project directories are defined and the test command is executed in the directory with the configured environment variables. `PW_TEST_HTML_REPORT_OPEN` tells Playwright to not open a report even if a test fails, `PLAYWRIGHT_HTML_REPORT` defines the report directory as a directory with

a timestamp, so that no test results are overwritten, and `PLAYWRIGHT_JSON_OUTPUT_FILE` specifies the location where a JSON reports shall be stored.

```
// playwright-trigger.mjs
import { spawn } from 'child_process'

const projects = [
  // ...
  {
    name: "IG Clone Svelte",
    cwd: "ig-clone-svelte"
  },
  // ...
]

const testArguments = [
  // "/.*change\.spec\.js/"
]

function generateUriSafeTimestamp() {
  //...
}

console.log(`Found projects: ${projects.map(p =>
    `"${p.name}"`).join(', ')}`)

console.log(`Starting tests for ${projects.length}
    ${projects.length == 1 ? 'project' : 'projects'}...`)

for (const project of projects) {

  console.log(`Starting with "${project.name}"`)

  const now = new Date()
  const reportDirectory =
      `playwright-report-${generateUriSafeTimestamp()}`

  await new Promise(resolve => {

    const testProcess = spawn("npm", ["run", "test:e2e",
        ...testArguments], {
      cwd: project.cwd,
      shell: true,
      env: {
        ...process.env,
        PW_TEST_HTML_REPORT_OPEN: 'never',
        PLAYWRIGHT_HTML_REPORT: reportDirectory,
        PLAYWRIGHT_JSON_OUTPUT_FILE: reportDirectory +
            "/test-results.json"
      }
    })

    // ...
  }).then(() => {
    console.log(`Finished "${project.name}"
        (${projects.indexOf(project) + 1}/${projects.length})`)
  })
}

console.log("DONE")
```

Listing 13: Trigger script for Playwright tests

```
// ig-clone-vue/playwright.config.js
import process from 'node:process'
import { defineConfig, devices } from '@playwright/test'

export default defineConfig({
  testDir: './tests',
  timeout: 60 * 1000,
  expect: { timeout: 5000 },
  forbidOnly: !!process.env.CI,
  retries: 2,
  workers: 1,
  reporter: [['html'], ['json', { outputFile:
      'playwright-report/test-results.json' }]],
  use: {
    actionTimeout: 0,
    baseURL: 'http://localhost:3000',
    trace: 'on',
    headless: true
  },

  projects: [
    { name: 'Chromium', use: {...devices['Desktop Chrome']} },
    { name: 'Firefox', use: {...devices['Desktop Firefox']} },
    { name: 'Webkit', use: {...devices['Desktop Safari']} },

    /* Test against mobile viewports. */
    { name: 'Mobile Chrome', use: {...devices['Pixel 5']} },
    { name: 'Mobile Safari', use: {...devices['iPhone 12']} },

    /* Test against branded browsers. */
    { name: 'Microsoft Edge', use: {channel: 'msedge'} },
    { name: 'Google Chrome', use: {channel: 'chrome'} },
  ],

  webServer: {
    command: 'vite build && serve -sd dist',
    port: 3000,
    reuseExistingServer: !process.env.CI
  }
})
```

Listing 14: Playwright configuration for Vue.js

The tests and test configuration are similar for all frameworks. Listing 14 shows how the test suite is configured. Timeouts are defined for all tests so that even slowly loading pages are tested properly and retries are specified to repeat failing tests twice. The reason for this specification is that fluctuating timings close to the limit of failure should be tested multiple times to ensure that the test is supposed to fail. All test executions and repetitions are configured to run in sequence to minimize the influence of the availability of resources on the testing machine. This is especially important because Playwright both opens the application in a browser and runs a webserver for local tests. It is set to start a webserver, wait for its availability and then open the application under the specified `baseURL`. The webserver command, port and `baseURL` are different for every framework. The test configuration also specifies a list of browsers to test the application in. For this study, seven browsers were chosen based on the most used browsers (StatCounter, 2024) and their mobile versions. The browsers are Chromium, Google Chrome, Mobile Chrome, Safari, Mobile Safari, Microsoft Edge and Firefox.

The tests written for this application are threefold as they reflect the separation of performance metrics (see section 3.4). Listings 15, 17 and 19 show the test files.

First, page load times are measured using `page-load.spec.js` (see listing 15). Every defined route is opened in a browser window, the navigation timings are extracted through a `page.evaluate(<evalFunction>)` method and the timings are attached and annotated so that they can be read after the test execution. The test for every page is that the timings `loadEventEnd` and `domComplete` are faster than a time budget. The paths and time budget per page configed in `pages.js` (see listing 16). To ensure a fast performance, the time budgets are defined to be under two seconds for all pages. Because no network requests are made in the design of the application on the About page, the time budget was lowered to 1.5 seconds here.

```
1  // page-load.spec.js
2  import { test, expect } from '@playwright/test';
3  import routes from "./pages.js"
4
5  test.describe("Load Time", () => {
6    for (const route of routes) {
7      test(`${route.name} loads within the page load budget`, { tag:
           [`@${route.name}`, '@pageLoad'] }, async ({ page },
           TestInfo) => {
8
9        await page.goto(route.path)
10       await page.waitForLoadState()
11
12       const timing = await measurePerformance(page)
13       TestInfo.attach("timing.json", { body:
             JSON.stringify(timing, null, 2), contentType:
             "application/json" })
14
15       const [{ responseStart, responseEnd,
             domContentLoadedEventEnd, domComplete, loadEventEnd }] =
             timing;
16
17       test.info().annotations.push({ type: 'Page Load Budget',
             description: `The time budget for this page was
             ${route.pageLoadBudgetMs}ms` });
18       // ...
19
20       expect.soft(domComplete, `domComplete event should happen
             within ${route.pageLoadBudgetMs}
             ms`).toBeLessThanOrEqual(route.pageLoadBudgetMs)
21       expect.soft(loadEventEnd, `loadEventEnd event should happen
             within ${route.pageLoadBudgetMs}
             ms`).toBeLessThanOrEqual(route.pageLoadBudgetMs)
22     })
23   }
24 })
25
26 async function measurePerformance(page) {
27   return await page.evaluate(() =>
         performance.getEntriesByType('navigation'));
28 }
```

Listing 15: Test file for page load times

```
1  // pages.js
2  const routes = [
```

```
  { name: "Index page", path: "/", pageLoadBudgetMs: 2_000 },
  { name: "About page", path: "/about", pageLoadBudgetMs: 1_500 },
  { name: "Create page", path: "/create", pageLoadBudgetMs: 2_000
    },
  { name: "Profile page", path: "/user/@PeterPoster",
    pageLoadBudgetMs: 2_000 },
]

export default routes;
```

Listing 16: Test pages configuration

Second, component load times are measured with the help of `dynamic-performance.spec.js`. The same routes are opened after in initialization script is injected into the browser window. Listings 17 and 18 show parts of the test definition and the injected script. The latter waits for a specific element to appear in the DOM that does not appear in the HTML skeleton, if it exists. Afterwards, it initializes a `MutationObserver` on that element. Each observation is stored with an xpath, id and the last mutation time. The mutation time is overwritten every time so that only the latest update is recorded and the list of times is published as a member of the window object. Recorded mutations are added or removed children, addition or removal of the element itself and a changed attribute. Because the time of mutation is only measured as the time difference to the addition of the application-specific root element, the recorded times are an estimation of the execution time between framework initialization and the latest DOM mutation.

The test script waits for ten seconds after the injection of the recording script and then evaluates the recorded timings. The update times are also attached to the test as a JSON file so that they can be traced after the test context no longer exists. The test to pass for the page is that the latest DOM mutation happens within the page's load time budget. In order to trace the failing components more easily, screenshots are token of each slow HTML element. Additionally, a screenshot of the whole page is taken in which slow elements are colored. Every screenshot is then attached to the test. This method ensures that slow components can be identified visually even if xpath and id of the element change between component lifecycles or application builds.

```
// dynamic-performance.spec.js
import { test, expect } from '@playwright/test';
import routes from "./pages"

test.describe("Dynamic load time", () => {

  for (const route of routes) {

    test(`Dynamic components on ${route.name} load within the load
        budget`, { tag: [`@${route.name}`, '@componentLoad'] },
        async ({ page }, TestInfo) => {
      // Inject performance measurement script into the page
      await page.addInitScript({ path: './tests/performance.js' })

      // Go to the measured page
      await page.goto(route.path)
      await page.waitForLoadState('domcontentloaded')

      // Start evaluation
      const latestUpdateComponents = await new Promise(resolve =>
          setTimeout(resolve, 10_000)).then(() => {
```

```
19
20          // Return the sorted load times
21          return page.evaluate(() => {
22            if (!window.dynamic_component_performance) return null
23
24            // Sort the components by their latest dom update time
25            const sortedEntries =
                  Object.entries(window.dynamic_component_performance)
26              .map(([key, value]) => { return { id: key, ...value }
                    })
27              .sort((a, b) => a.lastUpdated - b.lastUpdated)
28
29            return sortedEntries
30          })
31        })
32
33        // Attach the measurements in JSON format
34        TestInfo.attach("update-times.json", { body:
              JSON.stringify(latestUpdateComponents, null, 2),
              contentType: "application/json" })
35
36        latestUpdateComponents.forEach(comp => {
37          const latestUpdateTime = comp.lastUpdated
38
39          // Assert the latest update occurs in time
40          return expect.soft(latestUpdateTime, `Component with
                identifier ${comp.id} should load within
                ${route.pageLoadBudgetMs}
                ms`).toBeLessThan(route.pageLoadBudgetMs)
41        })
42
43        // Create screenshots of slow components
44        const componentScreenshots = await Promise.all(
45          latestUpdateComponents.map((el) => {
46            if (el.lastUpdated > route.pageLoadBudgetMs) {
47              return
                    page.locator(el.id).screenshot().then(screenshot
                    => [el, screenshot])
48            }
49          }).filter(i => i)
50        )
51
52        // Capture a screenshot of the whole page with highlighted
              slow components
53        if (latestUpdateComponents.some(comp => comp.lastUpdated >
              route.pageLoadBudgetMs)){
54          // Hightlight slow components in HTML
55          await page.evaluateHandle(([latestUpdateComponents]) => {
56            //...
57           }, [latestUpdateComponents.filter(comp =>
                 comp.lastUpdated >= route.pageLoadBudgetMs)])
58
59          // Create the screenshot
60          const screenshot = await page.screenshot({fullPage: true})
61
62          // Attach the screenshot to the test
63          await TestInfo.attach("fullpage_screenshot.png",  {body:
                screenshot, contentType: 'image/png'})
64        }
65
66        // Attach the screenshots of the slow components to the test
67        await Promise.all(
```

```
68          componentScreenshots.map(([el, screenshot]) => {
69            return TestInfo.attach(
                  `${el.id}-${el.lastUpdated}ms.png`, {body:
                  screenshot, contentType: 'image/png'})
70          })
71        )
72      })
73
74    }
75  })
```

Listing 17: Test file for component load times

```
1   // performance.js
2   let loadTimes = {}
3   let startTime = Date.now()
4
5   function observe(targetNode) {
6     // Options for the observer (which mutations to observe)
7     const config = { attributes: true, childList: true, subtree:
          true };
8
9     // Callback function to execute when mutations are observed
10    const callback = (mutationList, observer) => {
11      for (const mutation of mutationList) {
12
13        if (mutation.type === "childList") {
14          const targetId = getId(mutation.target)
15
16          const skipAttribute =
17            mutation.target.attributes.skipperformance?.value ||
18            mutation.target.attributes.skipPerformance?.value
19
20          if (!(skipAttribute == true || skipAttribute == 'true')) {
21
22            if (mutation.addedNodes.length > 0) {
23              let addedElements =
                    Array.from(mutation.addedNodes).map(el =>
                    el.nodeName !== "#comment" && el.nodeName !==
                    "#text" ? getXPath(el) : el)
24              if (addedElements.length === 1) addedElements =
                    addedElements[0]
25
26              if (Array.from(mutation.addedNodes)
27              ) {
28                loadTimes[targetId] = { ...loadTimes[targetId],
                      lastUpdated: Date.now() - startTime, xpath:
                      loadTimes[targetId]?.xpath ||
                      getXPath(mutation.target) }
29
30                Array.from(mutation.addedNodes).forEach(node => {
31                  try {
32                    const nodeId = getId(node)
33                    loadTimes[nodeId] = { ...loadTimes[nodeId],
                          lastUpdated: Date.now() - startTime, xpath:
                          loadTimes[nodeId]?.xpath || getXPath(node)}
34                  } catch (e) {
35                    console.warn(e)
36                  }
37                })
38              }
39            }
40
```

```
41            else if (mutation.removedNodes.length > 0) {
42              let removedElements =
                    Array.from(mutation.removedNodes).map(el =>
                    el.nodeName !== "#comment" && el.nodeName !==
                    "#text" ? getXPath(el) : el)
43              if (removedElements.length === 1) removedElements =
                    removedElements[0]
44
45              if (Array.from(mutation.removedNodes)
46              ) {
47                loadTimes[targetId] = { ...loadTimes[targetId],
                      lastUpdated: Date.now() - startTime, xpath:
                      loadTimes[targetId]?.xpath ||
                      getXPath(mutation.target) }
48
49                Array.from(mutation.removedNodes).forEach(node => {
50                  try {
51                    const nodeId = getId(node)
52                    loadTimes[nodeId] = { ...loadTimes[nodeId],
                          lastUpdated: Date.now() - startTime, xpath:
                          loadTimes[nodeId]?.xpath || getXPath(node) }
53                  } catch (e) {}
54                })
55              }
56            }
57
58          }
59
60        } else if (mutation.type === "attributes") {
61          console.log(`The ${mutation.attributeName} attribute was
                modified.`, mutation);
62
63          const targetId = getId(mutation.target)
64
65          const skipAttribute =
66            mutation.target.attributes.skipperformance?.value ||
67            mutation.target.attributes.skipPerformance?.value
68
69          if (!(skipAttribute == true || skipAttribute == 'true')) {
70            loadTimes[targetId] = { ...loadTimes[targetId],
                  lastUpdated: Date.now() - startTime, xpath:
                  loadTimes[targetId]?.xpath ||
                  getXPath(mutation.target) }
71          }
72
73        }
74      }
75
76      window.dynamic_component_performance = loadTimes
77    };
78
79    // Create an observer instance linked to the callback function
80    const observer = new MutationObserver(callback);
81
82    // Start observing the target node for configured mutations
83    observer.observe(targetNode, config);
84  }
85
86  function getId(element) {
87    // ...
88  }
89
```

```
function reset() {
  loadTimes = {}
  startTime = Date.now()
}

/**
 * Get absolute xPath position from dom element
 * @param {Element} element element to get position
 * @returns {String} xPath string
 */
function getXPath(element) {
  // ...
}

let interval;

function initObservation() {
  // The id of the targetNode has to be adapted to the framework
      or application
  const targetNode = document.getElementById("app")
  if (targetNode) {
    observe(targetNode)
    if (interval) clearInterval(interval)
  }
}

interval = setInterval(initObservation, 100)

// initialize window.dynamic_component_performance
window.dynamic_component_performance = loadTimes
```

Listing 18: Injected mutation recorder script

Third, tests for the component update times are specified in `state-change.spec.js` (see listing 19). In this test specification two other time budgets are defined. The first update to the DOM and the slowest update to the DOM are tested. The idea behind these time budgets is that users may perceive the "reaction time" as the time frame in which their action had any effect or the time frame in which the effect of their actions finishes. To this end, user actions are defined in combination with a route to perform these actions on. For this work, four actions are defined on the Create page: The changing of the caption, the selection of an image, the insertion of a media source and the creation of a new post, which is a combination of caption change and media selection.

In order to evaluate the reaction speed to those user actions, the same mutation recording script is inserted as for component load times. The page is then opened and the recorded mutation timings are reset. Afterwards, the user action is performaned and the new mutation times are extracted, attached to the test and evaluated. The tests to pass are then that the earliest mutation timing is within 100 ms of the user input and the latest mutation timing is within 500 ms of the user input. Again, screenshots are taken of all HTML elements that were recorded as mutated and do not pass the tests. These screenshots are also attached to the test in order to debug applications that do no pass the tests.

```
// state-change.spec.js
import { test, expect } from '@playwright/test';

const minReactionTime = 100;
const maxUpdateTime = 500;

```

```
7  const actions = [
8    {
9      route: '/create',
10     inputActions: [
11       {
12         name: 'Caption Change',
13         action: async (page) => {
14           const captionInputField = page.getByPlaceholder('Type
                 your caption here')
15           return captionInputField.fill('Lorem ipsum ...')
16         }
17       },
18       {
19         name: 'Media Selection',
20         action: async (page) => {
21           const mediaSelector = page.locator('#preloaded-image')
22           return mediaSelector.selectOption('moon.webp')
23         }
24       },
25       {
26         name: 'Media Source Insert',
27         action: async (page) => {
28           const captionInputField = page.getByPlaceholder('Insert
                 your media URL here...')
29           return captionInputField.fill(`${new URL(await
                 page.url()).origin}/abstract-circles.webp`)
30         }
31       },
32       {
33         name: 'Post Creation',
34         action: async (page) => {
35           const mediaSelector = page.locator('#preloaded-image')
36           const captionInputField = page.getByPlaceholder('Type
                 your caption here')
37           await mediaSelector.selectOption('moon.webp')
38           return captionInputField.fill('Lorem ipsum ...')
39         }
40       }
41     ]
42   }
43 ]
44
45 for (const actionGroup of actions) {
46   for (const inputAction of actionGroup.inputActions) {
47
48     test.describe(`State Change DOM Update: ${inputAction.name}`,
         { tag: [`@${inputAction.name.replace(/\s/g, '')}`,
         '@stateChange'] }, () => {
49       let page;
50       let domUpdates = null;
51
52       test.beforeAll(async ({ browser }) => {
53         page = await browser.newPage();
54         await page.addInitScript({path: './tests/performance.js'})
55
56         await page.goto(actionGroup.route)
57         await page.waitForLoadState('domcontentloaded')
58
59         await new Promise(resolve => setTimeout(resolve, 3_000))
60         await page.evaluate(() => {reset()})
61
62         await inputAction.action(page)
```

```
63
64          await new Promise(resolve => setTimeout(resolve, 5_000))
65          domUpdates = await page.evaluate(() => {
66            if (!window.dynamic_component_performance) return null
67
68            // Sort the components by their latest dom update time
69            const sortedEntries =
                  Object.entries(window.dynamic_component_performance)
70              .map(([key, value]) => { return { id: key, ...value }
                    })
71              .sort((a, b) => a.lastUpdated - b.lastUpdated)
72            return sortedEntries
73          })
74        });
75
76        test.afterAll(async () => {
77          await page.close();
78        });
79
80        test(`User input triggers first update within
              ${minReactionTime} ms`, { tag: ['@minimalReactionTime']
              }, async ({ }, TestInfo) => {
81          expect(domUpdates).not.toBeNull()
82          expect(domUpdates).not.toEqual([])
83          const minReactionComp = domUpdates[0]
84
85          await TestInfo.attach(`domUpdates${TestInfo.retry > 0 ?
                `_retry_${TestInfo.retry}` : ''}.json`, { body:
                JSON.stringify(domUpdates, null, 2), contentType:
                "application/json" })
86          await test.info().annotations.push({ type: `Fastest Update
                ${TestInfo.retry > 0 ? `(retry #${TestInfo.retry})` :
                ''}`, description: `Component with id
                ${minReactionComp.id} loaded
                ${minReactionComp.lastUpdated}ms after user input
                (xPath: ${minReactionComp.xpath})` });
87          expect.soft(minReactionComp.lastUpdated, `Fastest updated
                component with identifier ${minReactionComp.id} should
                update within ${minReactionTime}
                ms`).toBeLessThanOrEqual(minReactionTime)
88
89          if (domUpdates.some(comp => comp.lastUpdated >=
                minReactionTime))
90            await test.info().annotations.push({ type: 'Hint',
                  description: `Screenshots below show slow updating
                  components` });
91
92          // take screenshots of all elements referenced in
                domUpdates
93          await Promise.all(
94            // ...
95          )
96        })
97
98        test(`DOM updates triggered by state change finish within
              ${maxUpdateTime} ms`, { tag: ['@maximalReactionTime'] },
              async ({ }, TestInfo) => {
99          expect(domUpdates).not.toBeNull()
100         expect(domUpdates).not.toEqual([])
101         const maxUpdateComp = domUpdates.at(-1)
102         await TestInfo.attach("domUpdates.json", { body:
                JSON.stringify(domUpdates, null, 2), contentType:
```

```
            "application/json" })
103         await test.info().annotations.push({ type: 'Slowest
                Update', description: `Component with id
                ${maxUpdateComp.id} loaded
                ${maxUpdateComp.lastUpdated}ms after user input
                (xPath: ${maxUpdateComp.xpath})` });
104
105         domUpdates.forEach(comp => {
106           expect.soft(comp.lastUpdated, `Component with identifier
                  ${comp.id} should finish updates within
                  ${maxUpdateTime}
                  ms`).toBeLessThanOrEqual(maxUpdateTime)
107         })
108
109         if (domUpdates.some(comp => comp.lastUpdated >=
                maxUpdateTime))
110           await test.info().annotations.push({ type: 'Hint',
                  description: `Screenshots below show slow updating
                  components` });
111
112         // take screenshots of all elements referenced in
                domUpdates
113         await Promise.all(
114           // ...
115         )
116       })
117     })
118   }
119 }
```

Listing 19: Test file for component update times

5 Evaluation

5.1 Page Load Times

5.2 Component Load Times

5.3 Component Update Times

6 Conclusion

7 Summary

A Listings


```
1 <!-- media-component.component.html -->
2 <img class="postMedia" *ngIf="mediaSource &&
      mediaSource.endsWith('webp'); else videoMedia"
      [ngSrc]="mediaSource" [alt]="alt"
      [width]="width?.endsWith('%') ? 600 : width" [height]="height
      || (width?.endsWith('%') ? 600 : width)" [id]="id"
      [class]="class" [sizes]="width!" [priority]="priority" />
3 <ng-template #videoMedia>
```

```
    <video #video class="postMedia" *ngIf="mediaSource &&
        mediaSource.endsWith('mp4'); else mediaError"
        [attr.width]="width" controls
        controlslist="nodownload,nofullscreen,noremoteplayback"
        disablepictureinpicture loop [muted]="true"
        preload="metadata" >
      <source [src]="mediaSource" type="video/mp4" />
    </video>
  </ng-template>
  <ng-template #mediaError>
    <div class="mediaError">
      <p>Nothing to see yet...<br />Choose an image to continue!</p>
    </div>
  </ng-template>
```

Listing 20: MediaComponent in Angular (Template)

```
// media-component.component.ts
import { NgIf, NgOptimizedImage } from '@angular/common';
import { Component, ElementRef, Input, ViewChild } from
    '@angular/core';
import { playPauseVideo } from "../../../utils/autoplay";

@Component({
  selector: 'app-media-component',
  standalone: true,
  imports: [NgIf, NgOptimizedImage],
  templateUrl: './media-component.component.html',
  styleUrl: './media-component.component.css',
})
export class MediaComponentComponent {
  @Input() src!: string;
  @Input() alt: string = "";
  @Input() width?: string;
  @Input() height?: string;
  @Input() id: string = "";
  @Input() class: string = "";
  @Input() priority: Boolean = false;

  @ViewChild('video') video?: ElementRef<HTMLVideoElement>;

  mediaSource: string = "";

  constructor() { }

  ngAfterViewInit() {
    if (this.video) playPauseVideo(this.video.nativeElement)
  }

  ngOnChanges(): void {
    if (
      this.src == null ||
      this.src == undefined ||
      this.src.startsWith("http")
    )
      this.mediaSource = this.src;

    else this.mediaSource = `assets/stock-footage/${this.src}`
  }

}
```

Listing 21: MediaComponent in Angular (Module)

```astro
// MediaComponent.astro
---
import { Image } from "astro:assets";
import styles from "./MediaComponent.module.css";

const { src, alt, width, height, className, id, priority = false,
    ...rest } = Astro.props;

let mediaSource = "";

const mediaFiles = await import.meta.glob(
  "/src/assets/stock-footage/*.{webp,mp4}"
);
const media = Object.fromEntries(
  Object.entries(mediaFiles).map(([key, value]) => [
    key.split("/")[key.split("/").length - 1],
    value,
  ])
);

if (src?.startsWith("http")) mediaSource = src;
else mediaSource = (await media[src]()).default;
---

<script>
  import { playPauseVideo } from "../utils/autoplay";

  playPauseVideo();
</script>
```

Listing 22: MediaComponent in pure Astro (Frontmatter & Script)

```astro
// MediaComponent.astro
{
  mediaSource && src?.endsWith("webp") && (
    <Image src={mediaSource} alt="" width={width} height={height}
        class={[className, styles.postMedia].join(" ")} id={id}
        loading={priority ? "eager" : "lazy"} {...rest} />
  )
}
{
  mediaSource && src.endsWith("mp4") && (
    <video class={[className, styles.postMedia].join(" ")} id={id}
        width={width} preload="metadata" controls
        controlslist="nodownload,nofullscreen,noremoteplayback"
        disable-picture-in-picture loop muted >
      <source src={mediaSource} type="video/mp4" />
    </video>
  )
}

{
  !(mediaSource && src?.endsWith("webp")) &&
    !(mediaSource && src.endsWith("mp4")) && (
      <div className={styles.mediaError}
        style={{
          minHeight: height ? height + "px" : "300px",
          maxWidth: width ? width + "px" : null,
          overflow: "hidden",
        }}
      >
        <p>
          Nothing to see yet...<br />
```

```
            Choose an image to continue!
          </p>
        </div>
    )
}
```

Listing 23: MediaComponent in pure Astro (Template)

```
// MediaComponent.jsx
import { createRef, useEffect, useState } from "react";
import styles from "./MediaComponent.module.css";
import { playPauseVideo } from "../utils/autoplay";

const mediaFiles =
    import.meta.glob("/src/assets/stock-footage/*.{webp,mp4}");
const media = Object.fromEntries(
  Object.entries(mediaFiles).map(([key, value]) => [
    key.split("/")[key.split("/").length - 1],
    value,
  ])
);

const MediaComponent = (props) => {
  const { src, alt, width, height, className, id, priority =
      false, ...rest } = props;
  const [mediaSource, setMediaSource] = useState("");
  const videoRef = createRef();

  useEffect(() => {
    if (videoRef.current) playPauseVideo(videoRef.current);

    if (src?.startsWith("http")) setMediaSource(src);
    else if (src && media[src])
      media[src]().then((mediaFile) => {
        if (src.endsWith("webp"))
            setMediaSource(mediaFile.default.src);
        else setMediaSource(mediaFile.default);
      });
  }, [src, mediaSource, videoRef]);

  if (mediaSource && src?.endsWith("webp"))
    return (
      <img key={src} src={mediaSource} alt={alt} width={width}
          height={height} className={[className,
          styles.postMedia].join(" ")} id={id} loading={priority
          ? "eager" : "lazy"} {...rest} />
    );
  else if (mediaSource && src.endsWith("mp4"))
    return (
      <video ref={videoRef} key={mediaSource}
          className={[className, styles.postMedia].join(" ")}
          id={id} width={width} preload="metadata" controls
          controlsList="nodownload,nofullscreen,noremoteplayback"
          disablePictureInPicture loop muted >
        <source src={mediaSource} type="video/mp4" />
      </video>
    );
  else
    return (
      <div className={styles.mediaError}
        style={{
          minHeight: height ? height + "px" : "300px",
          maxWidth: width ? width + "px" : null,
```

```
              overflow: "hidden",
            }}
          >
            <p>
              Nothing to see yet...<br />
              Choose an image to continue!
            </p>
          </div>
        );
    };

    export default MediaComponent;
```

Listing 24: MediaComponent Astro Island with React

```
// MediaComponent.js
import { createRef, useEffect, useState } from "react";
import styles from "./MediaComponent.module.css"
import Image from "next/image";
import { playPauseVideo } from "@/utils/autoplay";

const MediaComponent = ({ src, alt, width, height, className, id,
    priority = false }) => {
  let [mediaSource, setMediaSource] = useState("")
  let videoRef = createRef()

  useEffect(() => {
    if (videoRef.current) playPauseVideo(videoRef.current)
    try {
      if (src.startsWith('http')) setMediaSource(src)
      else
          setMediaSource(require(`@/assets/stock-footage/${src}`).default)
    } catch (error) {
      setMediaSource("")
    }
  }, [videoRef, src])


  if (
    mediaSource &&
    (
      (mediaSource.src && mediaSource.src.endsWith('jpg')) ||
      (src.startsWith('http') && src.endsWith('jpg'))
    )
  ) return (
    <div style={{ position: "relative", aspectRatio: 1, width:
        width == "100%" ? width : `${width}px`, overflow: "hidden"
        }} id={id} className={[className, styles.postMedia].join("
        ")}>
      <Image priority={priority}
          placeholder={src.startsWith('http') ? "empty" : "blur"}
          quality={50} src={mediaSource} alt={alt}
          width={width.endsWith("%") ? 600 : width} height={height
          || (width.endsWith("%") ? 600 : width)} />
    </div>
  )
  else if (mediaSource && mediaSource.endsWith('mp4')) return (
    <video ref={videoRef} key={mediaSource} className={[className,
        styles.postMedia].join(" ")} id={id} width={width}
        preload="metadata" controls
        controlsList="nodownload,nofullscreen,noremoteplayback"
        disablePictureInPicture loop muted >
      <source src={mediaSource} type="video/mp4" />
```

```
      </video>
    )
    else return (
      <div className={styles.mediaError}>
        <p>Nothing to see yet...<br />Choose an image to
            continue!</p>
      </div>)
}

export default MediaComponent
```

Listing 25: MediaComponent in Next.js

```
<!-- MediaComponent.vue -->
<template>
  <NuxtImg class="postMedia" v-if="mediaSource?.endsWith('jpg')"
      :src="mediaSource" :alt="alt" :preset="preset"
      :loading="priority ? 'eager' : 'lazy'" :preload="priority"
      :width="$config.public.image_presets[preset].modifiers.width"
      :height="$config.public.image_presets[preset].modifiers.height"/>
  <video ref="video" class="postMedia"
      v-else-if="mediaSource?.endsWith('mp4')" :width="width"
      preload="metadata" controls
      controlslist="nodownload,nofullscreen,noremoteplayback"
      disablepictureinpicture loop muted >
    <source :src="mediaSource" type="video/mp4" />
  </video>
  <div v-else class="mediaError">
    <p>Nothing to see yet...<br />Choose an image to continue!</p>
  </div>
</template>

<script>//see listing below</script>
```

Listing 26: MediaComponent in Nuxt (Template)

```
// MediaComponent.vue
const glob = import.meta.glob("~/assets/stock-footage/*.mp4", {
    eager: true });
const media = Object.fromEntries(
  Object.entries(glob).map(([key, value]) => [
    key.split("/")[key.split("/").length - 1],
    value.default,
  ])
);

export default {
  name: "MediaComponent",
  props: {
    src: { type: String },
    alt: { type: String, default: "" },
    width: String,
    height: String,
    preset: String,
    priority: { type: Boolean, default: false },
  },
  computed: {
    mediaSource() {
      if (this.src.endsWith(".mp4")) return media[this.src];
      return this.src;
    },
  },
```

```
  mounted() {
    const video = this.$refs.video;
    if (video) playPauseVideo(video);
  },
};
```

Listing 27: MediaComponent in Nuxt (Script)

```
// MediaComponent.js
import { createRef, useEffect, useState } from "react";
import styles from "./MediaComponent.module.css"
import { playPauseVideo } from "src/utils/autoplay";

const MediaComponent = ({ src, alt, width, height, className, id,
    priority = false }) => {
  let [mediaSource, setMediaSource] = useState("")
  const videoRef = createRef()

  useEffect(() => {
    if (videoRef.current) playPauseVideo(videoRef.current)
    try {
      setMediaSource(src.startsWith('http') ? src :
          require(`src/assets/stock-footage/${src}`))
    } catch (error) {
      setMediaSource("")
    }
  }, [src, mediaSource, videoRef])


  if (mediaSource.endsWith('webp')) return (
    <img loading={priority ? "eager" : "lazy"} src={mediaSource}
        alt={alt} width={width} height={height}
        className={[className, styles.postMedia].join(" ")}
        id={id} />
  )
  else if (mediaSource.endsWith('mp4')) return (
    <video ref={videoRef} className={[className,
        styles.postMedia].join(" ")} id={id} width={width}
        preload="metadata" controls
        controlsList="nodownload,nofullscreen,noremoteplayback"
        disablePictureInPicture loop muted >
      <source src={mediaSource} type="video/mp4" />
    </video>
  )
  else return (
    <div className={styles.mediaError} styles={{ height: (height ?
        height + 'px' : '300px'), width: width.endsWith("%") ?
        width : width + "px" }}>
      <p>Nothing to see yet...<br />Choose an image to
          continue!</p>
    </div>
  )
}

export default MediaComponent
```

Listing 28: MediaComponent in React

```
// MediaComponent.svelte
import { onMount } from 'svelte';
import { playPauseVideo } from '$lib/utils/autoplay';

```

```
const images = import.meta.glob('$lib/assets/stock-footage/*', {
  eager: true,
  query: { enhanced: true, quality: 50, w: 600 }
});
const media = Object.fromEntries(
  Object.entries(images).map(([key, value]) =>
      [key.split('/')[key.split('/').length - 1], value])
);
export let mediaSource: string = '';
$: mediaSource =
  $$props.src == null ||
  $$props.src == undefined ||
  $$props.src == '' ||
  $$props.src.startsWith('http')
    ? $$props.src
    : media[$$props.src].default;
export let video: HTMLVideoElement | undefined = undefined;
onMount(() => {
  if (video) playPauseVideo(video);
});
```

Listing 29: MediaComponent in Svelte (Script)

```
// MediaComponent.svelte
{#if $$props?.src?.endsWith('jpg')}
  {#if $$props?.src?.startsWith('http')}
    <img id={$$props.id} class="postMedia {$$props.class || ''}"
        alt={$$props.alt} src={mediaSource}
        loading={$$props.eagerLoading ? 'eager' : 'lazy'}
      style:width={$$props.width ? $$props.width.endsWith('%') ?
          $$props.width : $$props.width + 'px' : undefined}
      style:height={$$props.height ? $$props.height.endsWith('%')
          ? $$props.height : $$props.height + 'px' : $$props.width
          ? $$props.width + 'px' : undefined}
    />
  {:else}
    <enhanced:img id={$$props.id} class="postMedia {$$props.class
        || ''}" alt={$$props.alt} src={mediaSource}
        loading={$$props.eagerLoading ? 'eager' : 'lazy'}
      style:width={$$props.width ? $$props.width.endsWith('%') ?
          $$props.width : $$props.width + 'px' : undefined}
      style:height={$$props.height ? $$props.height.endsWith('%')
          ? $$props.height : $$props.height + 'px' : $$props.width
          ? $$props.width + 'px' : undefined}
    />
  {/if}
{:else if $$props?.src?.endsWith('mp4')}
  <video class="postMedia" width={$$props.width} controls
      controlslist="nodownload,nofullscreen,noremoteplayback"
      disablepictureinpicture loop muted
      preload={$$props.eagerLoading ? 'auto' : 'metadata'}
      bind:this={video}>
    <source src={mediaSource} type="video/mp4" />
  </video>
{:else}
  <div class="mediaError">
    <p>Nothing to see yet...<br />Choose an image to continue!</p>
  </div>
{/if}
```

Listing 30: MediaComponent in Svelte (Template)

```
<!-- create.component.html -->
<header>
  <a [routerLink]="['/']" routerLinkActive="router-link-active">
    <app-not-instagram-logo />
  </a>
  <a [routerLink]="['/']" routerLinkActive="router-link-active">
    <app-xicon />
  </a>
</header>

<form id="newPostForm" action="" method="post">
  <input [(ngModel)]="mediaUrl" type="url" name="mediaUrl"
      id="mediaUrl" placeholder="Insert your media URL here..." />
  <p>or</p>
  <select name="preloaded-image" id="preloaded-image"
      [(ngModel)]="mediaChoice">
    <option value="">Choose one of our media files here...</option>
    <option *ngFor="let media of preloadedMedia" [value]="media">
      <span> {{ media }} </span>
    </option>
  </select>
  <textarea [(ngModel)]="caption" name="caption" id="caption"
      cols="30" rows="3" placeholder="Type your caption here" />
  <button type="submit" [disabled]="!(caption && mediaSource())">
    Post it!
  </button>
</form>

<hr />

<app-post [userhandle]="userhandle" [caption]="caption"
    [likeCount]="0" [mediaSource]="mediaUrl || mediaChoice"
    [hideActionIcons]="true" />
```

Listing 31: Create page in Angular (Template)

```
// create.component.ts
import { Component } from '@angular/core';
import { RouterLink } from '@angular/router';
import { NotInstagramLogoComponent } from
    '../components/not-instagram-logo/not-instagram-logo.component';
import { XIconComponent } from
    '../components/icons/xicon/xicon.component';
import { PostComponent } from '../components/post/post.component';
import { NgFor } from '@angular/common';
import { FormsModule } from '@angular/forms';

@Component({
  selector: 'app-create',
  standalone: true,
  imports: [RouterLink, NotInstagramLogoComponent, XIconComponent,
      PostComponent, NgFor, FormsModule],
  templateUrl: './create.component.html',
  styleUrl: './create.component.css'
})

export class CreateComponent {
  preloadedMedia: string[] = [
    "canyon.mp4", "abstract-circles.webp", ...
  ]
  userhandle: string = "@you"
  caption: string = ""
  mediaUrl: string = ""
```

```
  mediaChoice: string = ""

  mediaSource(): string {
    if (this.mediaUrl) return this.mediaUrl;
    return this.mediaChoice;
  }
}
```

Listing 32: Create page in Angular (Module)

```
// create/page.tsx
"use client";

import Link from "next/link";
import styles from "./create.module.css";
import NotInstagramLogo from "@/components/NotInstagramLogo";
import Post from "@/components/Post";
import XIcon from "@/components/icons/XIcon";
import { useEffect, useState } from "react";

const preloadedMedia = [
  "canyon.mp4", "abstract-circles.webp", ...
];

const userhandle = "@you";

const CreatePost = () => {
  const [caption, setCaption] = useState("");
  const [mediaUrl, setmediaUrl] = useState("");
  const [mediaChoice, setmediaChoice] = useState("");
  const [mediaSource, setMediaSource] = useState(mediaChoice);

  useEffect(() => {
    setMediaSource(mediaUrl || mediaChoice);
  }, [mediaUrl, mediaChoice]);

  return (
    <>
      <header className={styles.createHeader}>
        <Link href="/">
          <NotInstagramLogo />
        </Link>
        <Link href="/">
          <XIcon />
        </Link>
      </header>

      <form id={styles.newPostForm} action="" method="post">
        <input type="url" name="mediaUrl" id={styles.mediaUrl}
            placeholder="Insert your media URL here..."
            value={mediaUrl} onChange={(event) =>
            setmediaUrl(event.target.value)} />
        <p>or</p>
        <select name="preloaded-image" id="preloaded-image"
            value={mediaChoice} onChange={(event) =>
            setmediaChoice(event.target.value)} >
          <option value="">Choose one of our media files
              here...</option>
          {preloadedMedia.map((media) => (
            <option key={media} value={media}>{media}</option>
          ))}
        </select>
```

```
47          <textarea name="caption" id={styles.caption} cols={30}
                rows={3} placeholder="Type your caption here"
                value={caption} onChange={(event) =>
                setCaption(event.target.value)} />
48          <button type="submit" disabled={!(caption && mediaSource)}>
49            Post it!
50          </button>
51        </form>
52
53        <hr />
54
55        <Post userhandle={userhandle} caption={caption}
              likeCount={0} mediaSource={mediaSource}
              hideActionIcons={true} />
56      </>
57    );
58  };
59
60    export default CreatePost;
```

Listing 33: Create page in Next.js

```
1  <!-- create.vue -->
2  <template>
3    <header>
4      <NuxtLink :to="{ name: 'index' }">
5        <NotInstagramLogo />
6      </NuxtLink>
7      <NuxtLink :to="{ name: 'index' }">
8        <XIcon />
9      </NuxtLink>
10   </header>
11
12   <form id="newPostForm" action="" method="post">
13     <input type="url" name="mediaUrl" id="mediaUrl"
           placeholder="Insert your media URL here..."
           v-model="mediaUrl" />
14     <p>or</p>
15     <select name="preloaded-image" id="preloaded-image"
           v-model="mediaChoice">
16       <option value="">Choose one of our media files
             here...</option>
17       <option v-for="media in preloadedMedia" :key="media"
             :value="media">
18         {{ media }}
19       </option>
20     </select>
21     <textarea name="caption" id="caption" cols="30" rows="3"
           placeholder="Type your caption here" v-model="caption" />
22     <button type="submit" :disabled="!(caption && mediaSource)">
23       Post it!
24     </button>
25   </form>
26
27   <hr />
28
29   <Post :userhandle="userhandle" :caption="caption" :likeCount="0"
         :mediaSource="mediaSource" :hideActionIcons="true" />
30 </template>
```

Listing 34: Create page in Nuxt (Template)

```
// create.vue
export default {
  name: "CreateView",
  data() {
    return {
      preloadedMedia: [
        "canyon.mp4", "abstract-circles.webp", ...
      ],
      userhandle: "@you",
      caption: "",
      mediaUrl: "",
      mediaChoice: "",
    };
  },
  computed: {
    mediaSource() {
      if (this.mediaUrl) return this.mediaUrl;
      return this.mediaChoice;
    },
  },
};
```

Listing 35: Create page in Nuxt (Script)

```
// CreatePost.js
import { useState } from "react"
import { Link } from "react-router-dom"
imort styles from "./CreatePost.module.css"
import NotInstagramLogo from "src/components/NotInstagramLogo"
import Post from "src/components/Post"
import XIcon from "src/components/icons/XIcon"

const preloadedMedia = [
  "canyon.mp4", "abstract-circles.webp", ...
]

const userhandle = "@you"

const CreatePost = () => {

  const [caption, setCaption] = useState("")
  const [mediaUrl, setmediaUrl] = useState("")
  const [mediaChoice, setmediaChoice] = useState("")

  function mediaSource() { return mediaUrl || mediaChoice }

  return <>
    <header>
      <Link to="/">
        <NotInstagramLogo />
      </Link>
      <Link to="/">
        <XIcon />
      </Link>
    </header>

    <form id={styles.newPostForm} action="" method="post">
      <input type="url" name="mediaUrl" id={styles.mediaUrl}
          placeholder="Insert your media URL here..."
          value={mediaUrl} onChange={(event) =>
          setmediaUrl(event.target.value)} />
      <p>or</p>
```

```
        <select name="preloaded-image" id={'preloaded-image'}
            value={mediaChoice} onChange={(event) =>
            setmediaChoice(event.target.value)}>
          <option value="">Choose one of our media files
              here...</option>
          {preloadedMedia.map(media => <option key={media}
              value={media}>
            {media}
          </option>
          )}
        </select>
        < name="caption" id={styles.caption} cols="30" rows="3"
            placeholder="Type your caption here" value={caption}
            onChange={(event) => setCaption(event.target.value)} />
        <button type="submit" disabled={!(caption && mediaSource())}>
          Post it!
        </button>
      </form>

      <hr />

      <Post userhandle={userhandle} caption={caption} likeCount={0}
          mediaSource={mediaSource()} hideActionIcons={true} />
    </>
}

export default CreatePost
```

Listing 36: Create in React

```
// create/+page.svelte
import NotInstagramLogo from
    '$lib/components/NotInstagramLogo.svelte';
import XIcon from '$lib/components/icons/XIcon.svelte';
import Post from '$lib/components/Post.svelte';

export const preloadedMedia = [
  "canyon.mp4", "abstract-circles.webp", ...
];

export const userhandle = '@you';
export let caption = '';
export let mediaUrl = '';
export let mediaChoice = '';
let mediaSource: string;
$: mediaSource = getMediaSource(mediaUrl, mediaChoice);

function getMediaSource(url: string, choice: string) {
  return url || choice;
}
```

Listing 37: Create in Svelte (Script)

```
<!-- create/+page.svelte -->
<header>
  <a href="/">
    <NotInstagramLogo />
  </a>
  <a href="/">
    <XIcon />
  </a>
</header>
```

```
29
30 <form id="newPostForm" action="" method="post">
31   <input type="url" name="mediaUrl" id="mediaUrl"
         placeholder="Insert your media URL here..."
         bind:value={mediaUrl}/>
32   <p>or</p>
33   <select name="preloaded-image" id="preloaded-image"
         bind:value={mediaChoice}>
34     <option value="">Choose one of our media files here...</option>
35     {#each preloadedMedia as media}
36       <option value={media}><span>{media}</span></option>
37     {/each}
38   </select>
39   <textarea name="caption" id="caption" cols="30" rows="3"
         placeholder="Type your caption here" bind:value={caption} />
40   <button type="submit" disabled={!(caption && mediaSource)}> Post
         it! </button>
41 </form>
42
43 <hr />
44
45 <Post {userhandle} {caption} likeCount={0} {mediaSource}
     hideActionIcons={true} />
```

Listing 38: Create in Svelte (Template)

B List of Figures

C List of Tables

D Acronyms

CI/CD	Continuous Integration and Continuous Delivery. 13, 59
CLI	Command Line Interface. 18, 19, 30, 59
CSR	Client-side Rendering. 7, 11, 12, 15, 16, 17, 18, 26, 59
CSS	Cascading Style Sheet. 6, 7, 9, 15, 16, 59
DOM	Document Object Model. 6, 7, 9, 13, 14, 15, 16, 17, 21, 25, 30, 37, 41, 59
FVC	First Visual Change. 16, 17, 20, 59
HTML	Hypertext Markup Language. 6, 7, 9, 14, 15, 16, 17, 19, 20, 21, 26, 29, 30, 31, 33, 37, 41, 59
HTTP	Hypertext Transfer Protocol. 31, 59
JS	JavaScript. 7, 15, 16, 17, 19, 20, 59
JSON	JavaScript Object Notion. 19, 33, 34, 37, 59
LCP	Largest Contentful Paint. 15, 20, 59
LVC	Last Visual Change. 15, 17, 59
OFVC	Observed First Visual Change. 16, 20, 59
OLVC	Observed Last Visual Change. 16, 20, 59

PWA	Progressive Web App. 18, 59
SEO	Search Engine Optimization. 18, 59
SSR	Server-side Rendering. 7, 11, 12, 15, 16, 17, 18, 59
SVG	Support Vector Graphic. 9, 59
TBT	Total Blocking Time. 15, 16, 17, 20, 59
TBW	Total Byte Weight. 14, 15, 20, 59
TTFB	Time To First Byte. 14, 15, 20, 59
TTI	Time To Interactive. 14, 15, 16, 17, 20, 59
URL	Uniform Resource Locator. 23, 29, 30, 31, 59

References

Chopin, S., Parsa, P., Roe, D., Fu, A., Lichter, A., Wilton, H., Lucie, and Huang, J. (2024). Installation. `https://nuxt.com/docs/getting-started/installation`. accessed 08/07/2024.

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

Google (2019a). Eliminate render-blocking resources. `https://developer.chrome.com/docs/lighthouse/performance/render-blocking-resources`. accessed 08/01/2024.

Google (2019b). 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.

Google LLC (2024). Setting up the local environment and workspace. `https://angular.dev/tools/cli/setup-local`. accessed 08/07/2024.

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

MDN Mozilla (2024a). Intersectionobserver. `https://developer.mozilla.org/en-US/docs/Web/API/IntersectionObserver`. accessed 08/06/2024.

MDN Mozilla (2024b). Render-blocking. `https://developer.mozilla.org/en-US/docs/Glossary/Render_blocking`. accessed 08/09/2024.

Meta Platforms, Inc. (2024). Getting started. `https://legacy.reactjs.org/docs/getting-started.html`. accessed 08/07/2024.

Schott, F. K. (2024). Install and set up astro. `https://docs.astro.build/en/install-and-setup/`. accessed 08/07/2024.

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

Svelte (2024). Introduction. https://svelte.dev/docs/introduction. accessed 08/07/2024.

Vercel, Inc. (2024). Installation. https://nextjs.org/docs/getting-started/installation. accessed 08/07/2024.

W3C (2012). Navigation timing. https://www.w3.org/TR/navigation-timing/. accessed 07/10/2024.

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.

You, Evan (2024). Quick start. https://vuejs.org/guide/quick-start.html. accessed 08/07/2024.

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