FOM Hochschule für Oekonomie & Management

Hochschulzentrum Berlin

Bachelor-Thesis

im Studiengang Wirtschaftsinformatik - Business Information Systems

zur Erlangung des Grades eines

Bachelor of Science (B.Sc.)

über das Thema

Fullstack Javascript Frameworks analysed using the Example of a Car Configurator

von

Emil Triest

Erstgutachter	Prof. Paul Acquaro Ed.D. (Columbia University)
Matrikelnummer	566224
Abgabedatum	2023-10-26

Abstract

Over the past decade, the evolution of web application development has been marked by the rise of full-stack JavaScript frameworks. The aim of this research was to dissect the technological underpinnings, performance metrics, and adaptability of four prominent frameworks: Next.js, Nuxt.js, Remix, and SvelteKit. These were assessed based on key technological differentiators and their subsequent impact on performance and suitability. Specifically, the study used the complex backdrop of a car configurator, known for its intricate data interactions and real-time feedback demands, as a benchmark.

Next.js stood out with its hybrid architecture catering to diverse rendering needs, while Nuxt.js highlighted its reactivity system building on Vue.js's strengths. Remix, although newer to the scene, distinguished itself with optimized server-side data fetching. SvelteKit's emphasis on compile-time optimizations offered a contrasting approach to runtime reactivity.

However, the notion of a "winner" among these frameworks proved to be fluid, shifting based on context. For instance, while SvelteKit showcased rapid feedback ideal for car configurators' initial loads, the robust server-side capabilities of Next.js and Nuxt.js made them more suitable for handling intricate data operations.

Conclusively, while technological prowess and performance metrics of a framework are crucial, the decision should be rooted in the specific demands of the targeted web application. As the intricacies of web applications continue to evolve, the strategic framework selection remains paramount. This study serves not only as an analytical guide on the contemporary frameworks but also underscores the importance of aligning technological choices with application-specific demands.

Table of Contents

Index of Tables

Index of Figures

List of Abbreviations

Abbreviation	Definition
URL	Uniform Resource Locator
UML	Unified Modeling Language
API	Application Programming Interface
PLT	Page Load Time
TTFB	Time To First Byte
FCP	First Contentful Paint
LCP	Largest Contentful Paint
DRAM	Dynamic Random Access Memory
HTML	HyperText Markup Language
SSR	Server Side Rendering
CDN	Content Delivery Network
CLI	Command Line Interface
SEO	Search Engine Optimization
PHP	PHP: Hypertext Preprocessor
JAM	JavaScript, APIs, and Markup
UI	User Interface
REST	Representational State Transfer
MEAN	MongoDB, Express.js, Angular, and Node.js
MERN	MongoDB, Express.js, React, and Node.js
MEVN	MongoDB, Express.js, Vue.js, and Node.js
JS	JavaScript
UUID	Universally Unique Identifier
SSG	Static Site Generation
AR	Augmented Reality
VR	Virtual Reality

1. Introduction

1.1 Background

This section aims to allow for a better contextual understanding of the subject of this study; "Full Stack JavaScript Frameworks analysed using the Example of a Car Configurator" and provide a foundational background to the reader.

1.1.1 The Tech Stack

"A **solution stack** or **software stack** [or **tech stack**] is a set of software subsystems or components needed to create a complete platform such that no additional software is needed to support applications"[1] A tech stack usually contains the following layers: the client layer, the server layer and the data layer. This is shown in the diagram below:

Figure 1 - A foundational tech stack

In the context of web application development, the three layers have the following roles: the client side is used to present the user with relevant information in the form of a website, the server side is used to handle all sorts of requests and information that should not be visible for the client. Lastly, the data layer contains all the software relevant to the data of the application. The term "Full-Stack" refers to the use/possibility of integration of all of these layers.

1.1.2 Full-Stack JavaScript

[1] *Computer Desktop Encyclopedia*. The Computer Language Company. 2015.

In the ever-evolving landscape of web development, full-stack JavaScript has emerged as a powerhouse, revolutionizing the way we create dynamic and interactive web applications. This modern approach leverages JavaScript not only for front-end user interfaces but also for back-end server-side operations and database connections, unifying the development process and enabling seamless communication between these two crucial aspects of web application development.

Full-stack JavaScript development has gained immense popularity due to its versatility, code reusability, and the vibrant ecosystem of tools and frameworks available. JavaScript, initially designed for client-side scripting, has transcended its original role to become a key player on both ends of the web application spectrum. With JavaScript powering the entire stack, developers can build robust, real-time applications with ease, streamlining the development process and enhancing overall efficiency.

1.1.3 Car Configurators

Car configurator web applications possess several technological characteristics that render them ideal case studies for comparing full-stack JavaScript frameworks. The following features make them a compelling choice for evaluating the capabilities of such frameworks in real-world scenarios.

1. Car configurators demand complex, dynamic user interfaces that respond swiftly to user inputs. This complexity necessitates robust front-end JavaScript frameworks to manage the dynamic rendering of vehicle models, colors, features, and prices.

2. Car configurators involve intricate data handling, including real-time updates of vehicle specifications, pricing, and availability. Effective data management and communication between the front-end and back-end are critical, requiring back-end frameworks to efficiently process and serve data.

3. Car configurators often involve integration with external services, such as inventory databases, pricing calculators, and payment gateways. Full-stack frameworks must support seamless integration with these external systems.

Additionally, these applications require excellent performance to deliver a smooth and immersive user experience, making performance optimization a vital consideration for the choice of the framework.

In summary, car configurator web applications present a comprehensive set of technological challenges, encompassing complex user interfaces, data management, external integrations and performance optimization. Evaluating full-stack JavaScript frameworks in this context offers valuable insights into their capabilities, scalability, and suitability for demanding real-world applications.

1.2 Literature Review

Diving into the vast landscape of web app frameworks, the wealth of insights and discussions present in the literature will build the foundation of this study. To contextualize the present research and to offer a foundation upon which to build, the Literature Review delves into previous scholarly works and examines knowledge of industry leaders like google, tracing the evolution, strengths, and shortcomings of existing full-stack JavaScript frameworks. By dissecting the wealth of knowledge amassed over the years, this section aims to provide a comprehensive understanding of the state-of-the-art, setting the stage for the subsequent analysis of the selected frameworks: Next.js, Nuxt.js, Remix, and SvelteKit.

1.2.1 Quality metrics (user-oriented) of a Full-Stack JavaScript Web Application

Quality metrics for a full-stack JavaScript application encompass measurements related to both the front-end (client-side) and the back-end (server-side). These metrics help developers understand the efficiency, speed, and reliability of their applications. The following table shows which quality metrics are important for a Full-Stack JavaScript web application and why:

Metric	Definition	Importance	Source
Load Time/Page Load Time (PLT)	The time taken for a webpage to fully load and render in a browser.	It directly affects user experience and engagement. A faster PLT can lead	Google's Web Fundamentals[i]

		to increased user satisfaction.	
Time to First Byte (TTFB)	The time it takes for a user's browser to receive the first byte of data from the server.	It provides insight into server response times and the initial stages of data retrieval.	Akamai's Performance Matters Survey[ii]
First Contentful Paint (FCP) and Largest Contentful Paint (LCP)	FCP is the time it takes for the first content to appear on the screen. LCP measures when the largest content element is rendered.	These metrics indicate how quickly users see meaningful content on the screen	Google's Web Vitals[iii]
JavaScript Execution Time	The time taken for the JavaScript code to execute.	Slow or blocking scripts can significantly hinder application performance.	Research at Google on JavaScript performance[iv]
API Response Time	The time taken for an API to process a request and return a response.	In full-stack applications, front-end components often rely on back-end APIs. Slow API responses can negatively impact user experience.	The Art of Monitoring by James Turnbull[v]
Memory Usage	The amount of memory consumed by the application.	Excessive memory consumption can lead to slowdowns, crashes, and reduced server capacity.	Memory Systems: Cache, DRAM, Disk by Bruce Jacob, Spencer Ng, David Wang.[vi]

Table 1 – Relevant User-Oriented Quality Metrics for a Full-Stack JavaScript Web Application

The sources used to justify the quality metrics used were a combination of academic sources and industry best practices and testimonials of renowned sources. This was done because while academic sources offer valuable theoretical insights, combining them with industry best practices and testimonials provides a more comprehensive understanding of what makes a full-stack JavaScript framework user-friendly.

1.2.2 Quality metrics (developer-oriented) of a Full-Stack JavaScript Web Application

Developer friendliness in a full-stack JavaScript framework is determined by a range of factors that make it easier for developers to understand, implement, maintain, and scale applications using that framework. The following table shows relevant factors that influence the developer friendliness of a full-stack JavaScript framework, along with an academic or reputable source for each:

Metric	Definition	Importance	Source
Documentation Quality	The clarity, comprehensiveness, and accuracy of official documentation, tutorials, and guides.	Good documentation is essential for onboarding new developers and assisting experienced ones.	Paper examining APIs from the standpoint of developers[vii]
Community Support and Ecosystem	The presence of an active developer community, available plugins/extensions, and third-party resources.	A vibrant community can provide solutions to common issues, share best practices, and develop tools or plugins that extend the framework's capabilities.	Conference excerpt on Collaborative Tools in Software Development[viii]
Consistency and Predictability	The uniformity in naming conventions, patterns, and behaviours within the framework.	Consistency allows developers to make educated guesses about unfamiliar parts of the API or framework.	Conference excerpt on a Web-Search Tool for API Components[ix]
Modularity and Flexibility	The ability of the framework to allow modular development and be flexible enough for various use cases.	Modular frameworks enable developers to use only the components they need, leading to more efficient and cleaner codebases.	Book on Adopting Product-Line Approaches in Software Architecture.[x]
Clear Error Messages and Debugging Tools	Informative error messages and powerful debugging tools incorporated within the framework.	Clear error messages and debugging tools help developers quickly identify and fix issues, reducing development time.	Book “Transactions on Software Engineering”[xi]

Scalability	The capability of the framework to adapt to growing application needs without significant modifications.	Scalability ensures that the framework can support the application as it grows in complexity and user base.	Conference excerpt on Elasticity in Cloud Computing.[xii]
Integration with Tools and Services	How easily the framework integrates with popular development tools and services.	Integration capabilities ensure that developers can use their preferred tools and services without friction.	Conference excerpt on selecting Software Requirements Prioritization Techniques.[xiii]

Table 2 – Relevant Developer-Oriented Metrics for Full-Stack JavaScript Frameworks

Again, while academic sources offer valuable theoretical insights, combining them with industry best practices and testimonials provides a more comprehensive understanding of what makes a full-stack JavaScript framework developer-friendly.

1.2.3 Short history of Full-Stack JavaScript

Jeff Atwood, co-founder of Stack Overflow, dubbed Atwood's law in 2007: "Any application that *can* be written in JavaScript, *will* eventually be written in JavaScript."[2] In 2023, JavaScript was found to be the most used programming language among software developers[3]. This 'law' seems to apply to full stack web applications as well.

While there may be reasons to use different programming languages for different parts of a web application, the full stack JavaScript approach has many benefits. These include promoting code reusability, simplifying the development process, and closing the gap between front-end and back-end development.

As of the writing of this thesis, there are ten full stack JavaScript frameworks with more than 10 thousand stars[4] on GitHub:

[2] Atwood, J. (2007). The principle of least power. *Coding Horror*. https://blog.codinghorror.com/the-principle-of-least-power/
[3] *Most used languages among software developers globally 2023 | Statista*. (2023, July 19). Statista. https://www.statista.com/statistics/793628/worldwide-developer-survey-most-used-languages/
[4] https://docs.github.com/en/get-started/exploring-projects-on-github/saving-repositories-with-stars

Name	Stars (in thousands)
Next.js	111.0
Nuxt	47.3
Meteor	43.7
Astro	34.2
Remix	24.5
Redwood	16.3
SvelteKit	16.0
Blitz	13.0
Amplication	11.5
Fresh	11.2

Table 3 - The most popular full stack JavaScript frameworks on GitHub.

Each framework has specific characteristics that differentiate it from the frameworks. Next.js uses React, Nuxt uses Vue, Meteor lets the developer choose their frontend, Astro focuses on content (useful for blogs for example), Remix tries to minimise loading by using native browser features, etc. Most of these frameworks have been around for a while. The first version of the most popular full stack framework, Next.js, was released in 2016[5]. However, not a lot of research has occurred regarding the benefits and drawbacks of these frameworks.

1.2.4 Which Frameworks and Why

Since examining all of these frameworks would explode the scope of this thesis, it was decided to examine four of the most popular frameworks Next.js, Nuxt, Remix and SvelteKit. Meteor, Astro and Redwood, although they are more popular than Remix and SvelteKit respectively, were skipped for separate reasons:

- Meteor can be implemented with many different frontends (including React, Vue and Svelte), causing a lot of overlap with other frameworks. Furthermore, with all of these implementations for just one framework it would severely limit the remaining scope available for the analysis and implementation of other frameworks.[6]
- Astro has a great reputation and is already being used by big names like Google and Microsoft. There is a very good reason for this popularity: it is a brilliant solution for

[5] https://github.com/vercel/next.js/releases/tag/1.0.0
[6] https://www.meteor.com/

content sites like blogs or documentation sites, however for the use case of a more complex site, like a car configurator, it is likely to be less suitable.[7]

- Redwood was skipped because like Next.js and Remix, it uses React as a frontend solution and is a lot less popular than the two frameworks. In order to avoid comparing three react full stack frameworks and have a more well-rounded scope, SvelteKit was picked for the analysis instead.[8]

Since adding more frameworks to the scope would probably result in a less in-depth review of each of these frameworks, it was decided to limit the scope and not add the remaining, less popular, frameworks (Blitz, Amplication and Fresh).

1.2.5 Next

Taken from the next.js website, "Next.js is a flexible **React framework** that gives you building blocks to create fast **web applications**."[9] Traditional React web applications employ client-side rendering, where the browser initially loads a bare HTML shell without content. Subsequently, it fetches a JavaScript file to dynamically render and enhance the page. However, this approach has drawbacks: content may not consistently index in search engines, and initial page load times can be lengthy.

Next.js addresses these issues with server-side rendering (SSR). It pre-renders content on the server, ensuring users and search bots encounter fully rendered HTML initially, transitioning to client-side rendering afterward, akin to conventional React apps.

Next.js structures directories to mirror URLs due to its integrated router. Its primary advantage lies in data fetching. Multiple server rendering strategies are available within one project:

1. Static Generation (Pre-rendering): Pages render during build time. The "getStaticProps" function fetches data, enabling local HTML rendering and CDN caching. Ideal for blogs or infrequently updated content.

[7] https://astro.build/
[8] https://redwoodjs.com/
[9] https://nextjs.org/learn/foundations/about-nextjs/what-is-nextjs

2. Server-Side Rendering (SSR): For frequently updated data, like that of car configurators, “getServerSideProps” fetches the latest server data per request, not at build time.

3. Incremental Static Regeneration: By adding a “revalidate” option to “getStaticProps”, Next.js regenerates pages within a specified time frame upon new requests, accommodating variable data update frequencies.

In summary, Next.js combines server-side rendering with versatile data fetching strategies, optimizing initial content load, and catering to applications with diverse data dynamics.

1.2.6 Nuxt

Similar to Next.js, Nuxt.js augments Vue.js. Nuxt.js incorporats its Nitro server engine, enabling diverse rendering modalities within a singular application context.

By default, Nuxt.js employs Universal rendering, commencing with server-side rendering and subsequently transitioning to client-side JavaScript for successive navigation. Developers wield the prerogative to meticulously configure rendering and caching paradigms for each route and to leverage deployment capabilities compatible with edge networks like Cloudflare. Notably, Nuxt.js encompasses an array of sophisticated built-in development tools and boasts a comprehensive ecosystem of modular extensions, facilitating a spectrum of functionalities such as image optimization, content management, and database integration.

Project initiation entails the utilization of a Command Line Interface (CLI), with the pivotal entry point manifesting in the "App.vue" file, which seamlessly accommodates TypeScript. Vue components are situated within the "Pages" directory, with file hierarchy directly dictating route definitions and Uniform Resource Locators (URLs). Of particular note, Nuxt.js streamlines code architecture through dedicated repositories for reusable components and functions, thereby automating component importation into pages. Consequently, extraneous import declarations are curtailed, and code partitioning is optimized.

Nuxt.js further enhances the development process by simplifying content administration through the "layouts" and "content" directories. Middleware capabilities, as presented within the "middleware" directory, facilitate pre-route code execution. The framework furnishes composable modules, such as "useFetch" for data retrieval and "useState" for seamless state management across client and server renders. Additionally, Nuxt.js contributes to search engine optimization through composables like "useHead" and "useSEOMeta," and enriches user interaction with fluid page transition animations via inbuilt transition mechanisms, thereby promoting an elevated user experience.

1.2.7 Remix

Remix is another React-based framework focused on server-side rendering, again offering potential advantages in performance and search engine optimization compared to client-side React applications. While addressing a well-known challenge, Remix introduces a distinctive approach reminiscent of traditional PHP or Ruby on Rails development.

One notable aspect of Remix is its exclusive focus on server-side rendering, distinguishing it from Next.js, which supports static site generation and incremental static regeneration. This decision sets Remix apart, as static generation has gained popularity for its rapid time-to-first-byte performance and straightforward deployment via static files. However, the drawback of static sites is the need for rebuilding pages upon data changes.

Remix forgoes the JAMstack approach, emphasizing server-side rendering, which necessitates a dedicated server to run the application. This choice aligns Remix with handling dynamic data more effectively. Ideal scenarios for Remix involve applications with numerous pages reliant on dynamic database-driven content.

1.2.8 SvelteKit

SvelteKit is a web application framework designed for server-rendered applications, seamlessly integrated with the popular Svelte UI Library. Created by Rich Harris, the mind behind Svelte, version 1.0 was released in 2022. A key feature of SvelteKit is its universal rendering capability, applicable to Node.js and various JavaScript Edge runtimes. This approach enables servers to pre-fetch data, delivering fully rendered HTML to browsers—a

crucial aspect for performance and search engine optimization (SEO). Subsequently, JavaScript takes charge, ensuring fast and smooth single-page application behavior.

SvelteKit incorporates a built-in router, with each page component corresponding to a unique URL. These components may have associated JavaScript files to handle data retrieval, exporting a "load" function whose return value becomes accessible to the component. SvelteKit promotes end-to-end type safety and offers data availability through Svelte stores, reducing the need for complex state management.

Pages can execute server-side actions, such as form submissions, without relying on client-side JavaScript. Layouts enable the sharing of UI components across multiple child routes, allowing them to fetch data independently. Additionally, SvelteKit supports server files for building RESTful APIs using functions like "get," "post," "patch," and "delete."

The framework's universality ensures compatibility across client and server environments, while the "server" extension permits exclusive server-side code execution, safeguarding sensitive resources like databases and environment variables.

1.3 Problem Description

As described in section 1.2.3 (Short History of Full-Stack JavaScript), full-stack JavaScript frameworks have become quite popular amongst developers in recent history. Scientific research on the topic, as shown below, has either focused on examining full-stack JavaScript by cherry picking every component of the tech stack, analysing the frontend frameworks or narrowing the scope of the research to only one full-stack framework.

Literature analysing cherry-picked stacks:

1. Study comparing the MEAN, MERN and MEVN stack[xiv]
2. In-depth study of the MEAN stack[xv]

Literature focusing on the frontend:

1. Study analysing the benefits of React[xvi]

2. Analysis of the usage of Vue for web apps[xvii]
3. Study examining Svelte[xviii]

Literature focusing on one single full-stack framework:

1. Book on Next.js[xix]
2. Book on Nuxt.js[xx]
3. Section on development with Sveltekit.js[xxi]

The absence of comprehensive research comparing the most popular full-stack JavaScript frameworks underscores a critical gap in the current body of knowledge within the field of web development. This gap necessitates the undertaking of this study, as it addresses a series of significant issues:

- **Evaluation of Framework Efficacy**: The rapid evolution and diversification of web development technologies have given rise to an abundance of full-stack JavaScript frameworks. Yet, the lack of comparative studies hinders developers and organizations from making informed decisions when selecting the most suitable framework for their specific needs. Understanding the comparative strengths and weaknesses of these frameworks is essential for optimizing development processes, improving application performance, and ensuring long-term sustainability.
- **Optimizing Resource Allocation**: The choice of a web development framework can have profound implications on resource allocation, including development time, human resources, and project budgets. An informed decision in this regard can significantly impact the efficiency of web development projects, saving both time and financial resources. Consequently, this study serves as a vital resource for organizations striving to make prudent investments in web development technologies.
- **Enhancing Software Quality**: The quality of web applications is inherently tied to the framework on which they are built. Suboptimal framework choices can lead to inefficiencies, performance bottlenecks, and scalability challenges. By comparing popular full-stack JavaScript frameworks, this study offers insights that can contribute to the development of higher quality, more reliable web applications, ultimately enhancing the user experience and reducing maintenance burdens.

- **Supporting Educational Initiatives**: As web development continues to be a popular field of study and professional pursuit, educational institutions and aspiring developers require up-to-date, comprehensive resources for learning and staying current with industry trends. This study contributes to the educational ecosystem by providing valuable insights into the most widely adopted full-stack JavaScript frameworks, aiding both educators and learners in their endeavors.

To summarize, this study's significance extends beyond the realm of web development. It addresses a pressing need within the industry, empowers decision-makers, improves resource allocation, enhances software quality, and supports educational initiatives. By comparing the most popular full-stack JavaScript frameworks and leveraging a relevant case study approach, this research seeks to bridge an existing knowledge gap and offer practical guidance to developers, organizations, and educators in the ever-evolving landscape of web development.

1.4 Objectives

The objectives of this study are twofold: to comprehensively evaluate and compare the capabilities of four prominent full-stack JavaScript frameworks—Next.js, Nuxt, Remix, and SvelteKit—and to ascertain their suitability for developing sophisticated web applications, particularly car configurators. To achieve this overarching goal, the study is guided by the following research questions:

Research Question 1: What are the key technological differentiators among popular JS full-stack frameworks and how do these differences impact their performance and suitability for web application development?

Research Question 2: How do the selected full-stack JavaScript frameworks address the challenges posed by the characteristics of the typical demands of a web application?

Research Question 3: Can clear winners be defined, both in terms of technological superiority and context suitability?

These research questions guide the investigation into the distinctive features and capabilities of each framework and their capacity to address the specific demands of car configurator

applications. Through a systematic exploration of these questions, this study aims to provide valuable insights for developers and decision-makers seeking to choose the most suitable full-stack JavaScript framework for web application development, especially in the context of car configurators.

1.5 Methodology

Mapping the course of any comprehensive research necessitates a robust and systematic methodology. This section delves into the intricate blueprint of procedures, tools, and analytical approaches that formed the foundation of this study. By outlining the chosen strategies and justifying the methods, it ensures a transparent, replicable, and coherent path for those keen on understanding the details of the research process.

1.5.1 Definition of Case Study Guidelines

The comparison of the specified frameworks using the case study of a car configurator will be conducted by implementing a simplified version of a car configurator in each of the frameworks. To allow for an unbiased and scientific comparison, the following set of guidelines were defined for each implementation:

Each implementation must contain the following features:

- The use of a dataset obtained from Mercedes-Benz developer API.
- Possibility to change every aspect (engine, wheels, color etc.) of the current configuration.
- Present an Image of the model.
- Show a price of the configuration.

Each Implementation must include the following pages:

- A main page allowing the user to configure the car.
- A confirmation page allowing the user to see the changes performed in the configuration.

1.5.2 Definition and Justification of Controlled Variables

The following controlled variables were defined as to not affect the comparison:

- **Dataset:** the usage of different sets of data in each implementation would be likely to have an impact on loading times.
- **Data API used:** the Mercedes-Benz developer API will be used to retrieve industry configuration data for all implementations.
- **Hosting platform:** the platform on which the applications run can have effects on the performance of the application.
- **Images:** as images can also affect some metrics of performance of a web application the same images must be used in each implementation.

1.5.3 Definition of Quality Metrics

The research conducted in sections 1.2.1 and 1.2.2 was used to choose a collection of relevant metrics to determine the quality of the frameworks. The following lists show which metrics were chosen for both aspects:

User-oriented metrics:

1. Load Time / Page Load Time (PLT)
2. Time to First Byte (TTFB)
3. First Contentful Paint (FCP) and Largest Contentful Paint (LCP)

For the user-oriented metrics, it was decided to use only PLT, TTFB and FCP/LCP since variations in the results for these metrics would have the greatest, most direct effect for the user. Furthermore, variances in these metrics are likely to directly reflect other metrics (such as API response time, JavaScript execution time etc.).

Developer-oriented metrics:

1. Documentation Quality
2. Community Support and Ecosystem

3. Consistency and Predictability
4. Modularity and Flexibility
5. Clear Error Messages and Debugging Tools
6. Scalability
7. Integration with Tools and Services

2. Analysis

The following section delves into the heart of the gathered data, breaking it down to discern clear patterns and insights. By connecting seemingly disparate data points, this segment offers a comprehensive understanding of the findings, turning raw information into a coherent narrative that holds implications for the broader field of web application development.

2.1 User stories & Acceptance Criteria

Before starting development, a user story and acceptance criteria was defined for each of the previously defined pages, to ensure uniformity and clarity regarding the development of each page for each implementation.

2.1.1 Page 1: Main Page

Title: Configuration Page

User Story:
As a user, I want to access the main page of the car configurator so that I can view and choose from a list of configurable components for my desired car model.

Acceptance Criteria:

1. The page should display a title.
2. The page should display a subtitle indicating the framework used for the implementation.
3. An image of a car should be prominently displayed.

4. The page should display a subtitle indicating the name of the car model being configured.
5. A list of configurable components should be presented, organized into predefined categories obtained from the data.
6. The user should be able to select and deselect components from the list.
7. The page should have a button that allows the user to proceed to the confirmation page with their current configuration.

2.1.2 Page 2: End Page

Title: Confirmation Page

User Story:
As a user, after configuring my car model, I want to view a summary of my choices and the total price so that I can confirm my desired configuration.

Acceptance Criteria:

1. The page should display a title.
2. The page should display a subtitle indicating the name of the car model.
3. The page should display a subtitle indicating the total price of the selected configuration.
4. A list of the selected equipment should be presented, organized into predefined categories obtained from the data.

2.2 Configuration Process

To visualise the configuration process ahead of development, the following diagram was created:

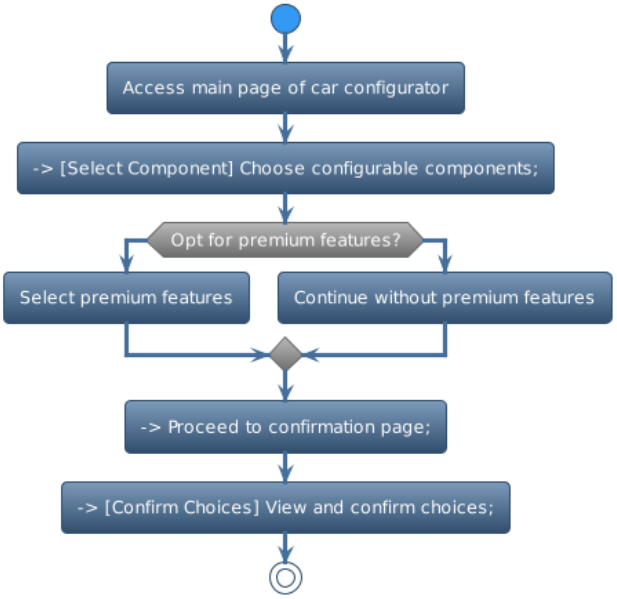

Figure 2 - UML Diagram showing the business process of the car configurator.

The process starts by accessing the main page of the car configurator, the user can then choose configurable components, and either opt for premium features or proceed with the default configuration, finally the user can proceed to the next page and view their configuration.

2.3 Development

Using the user stories defined above, the development process was similar for each implementation. A configuration page was implemented that used datasets obtained from the Mercedes-Benz developer API[10] to display a configurable list of categorised equipment. Each implementation also included another page which showed the configuration price and the equipment selected by the user. The following sections will dive deeper into the important aspects of the development of these implementations.

2.3.1 Datasets

[10] https://developer.mercedes-benz.com/products/car_configurator/

Three sets of data were used throughout the configurators: *selectableData*, *modelInfo*, and *initialData*. The data specification for all of these datasets can be found on the specifications page of the Mercedes-Benz developer API website[11].

The datasets were obtained from the following API endpoints:

selectableData:
/markets/{marketId}/models/{modelId}/configurations/{configurationId}/selectables
modelInfo:
/markets/{marketId}/models/{modelId}
initialData:
/markets/{marketId}/models/{modelId}/configurations/initial

Figure 3 - selectableData dataset used in each implementation, obtained from the Mercedes-Benz developer API.

As described on the site, the *selectableData* dataset lists relevant configuration data such as the possible components of the car, specific data pertaining to these components such as whether they were already included in the default configuration, price information, etc. and finally the different component categories of the car such as upholsteries and paints.

[11] https://developer.mercedes-benz.com/products/car_configurator/specifications/car_configurator

Figure 4 - modelInfo dataset used in each implementation, obtained from the Mercedes-Benz developer API.

The *modelInfo* dataset lists static data relevant to the specific model, as can be seen in the screenshot above. It includes general information like the name of the model, the position of the steering wheel and specific pricing information including relevant taxes.

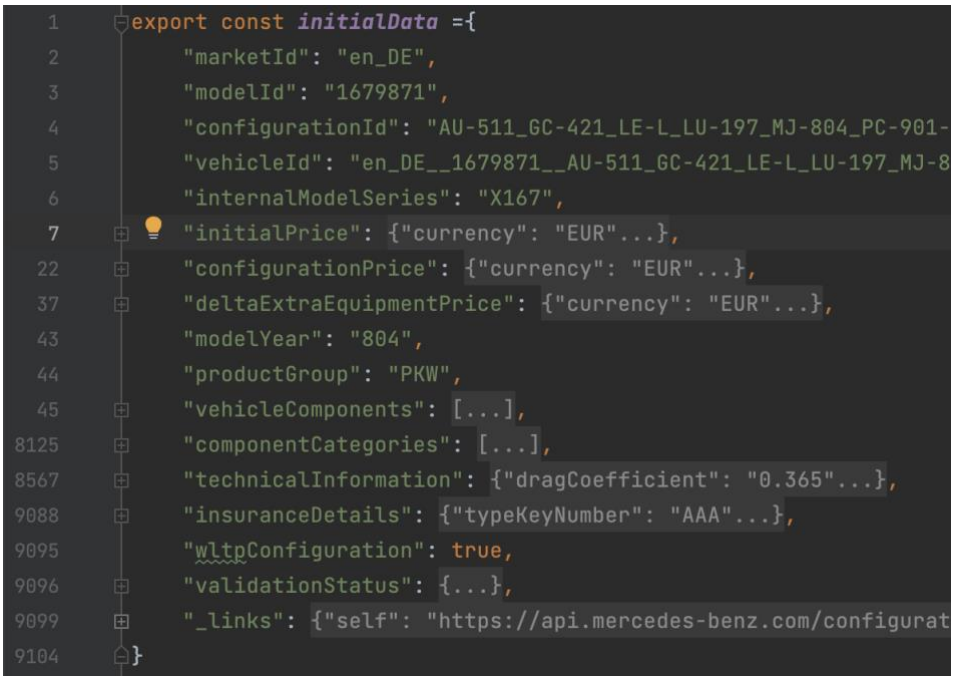

Figure 5 - initialData dataset used in each implementation, obtained from the Mercedes-Benz developer API.

The *initialData* dataset listed relevant information for the initial configuration of the model, including information regarding the price, technical information and different components included in the initial configuration. These datasets provided real-world industry data for each implementation.

2.3.2 Directory/Code Structure

This section will examine the directory/code structure of each implementation. For readers that want to delve deeper into the code and directories of these implementations, the links for the codebases can be accessed in *Table 4 – Relevant Resources of each Implementation.*

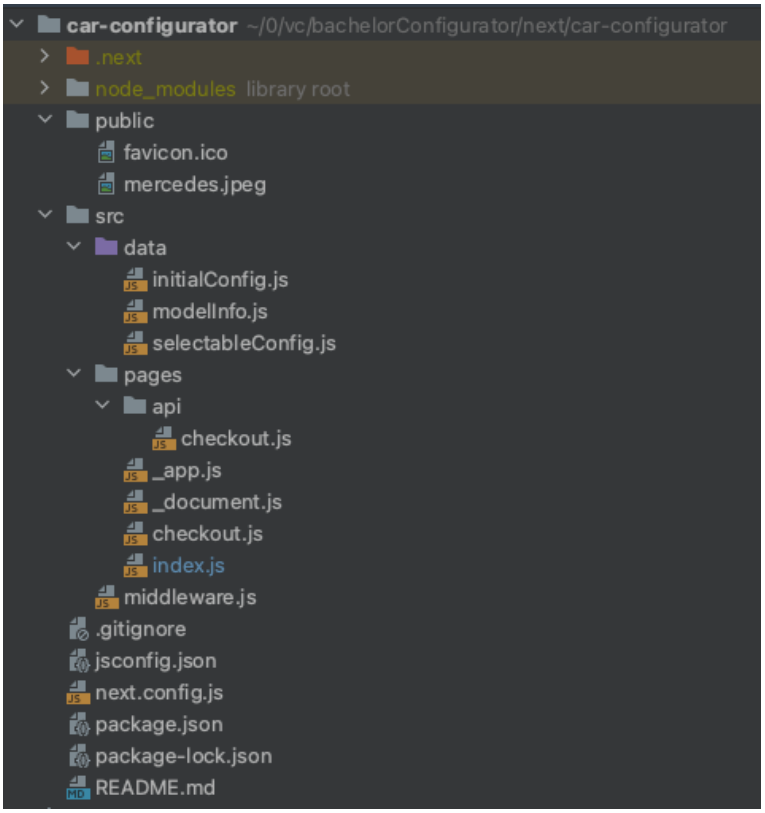

Figure 6 - Directory structure of the Next.js implementation.

The screenshot above shows the root directory of the Next.js implementation, which contains library directories (*/.next* and */node_modules*), a */public* directory for data such as images, a */src* directory which includes all the source code of the project and configuration files (*.gitignore*, *jsconfig.json*, etc.).

The source code directory includes the */data* directory, the */pages* directory and a *middleware.js* file. The */data* directory contains all relevant configurator data specified above. The */pages* directory includes an */api* directory for specifying API routes, next.js specific files (*_app.js* and *_document.js*) and page handlers (*index.js*, and *checkout.js*). Files that are placed in the */pages* directory are react components that are mapped to the corresponding route in the browser, i.e., the content defined in *checkout.js* can be accessed in the browser by accessing the */checkout* path. The *index.js* file handles the root path and any files placed in the */api* directory are mapped to the corresponding web path. The *middleware.js* file is used to define code that runs before a request is completed. In this implementation, the file is used to define a cookie to identify users and their corresponding configurations.

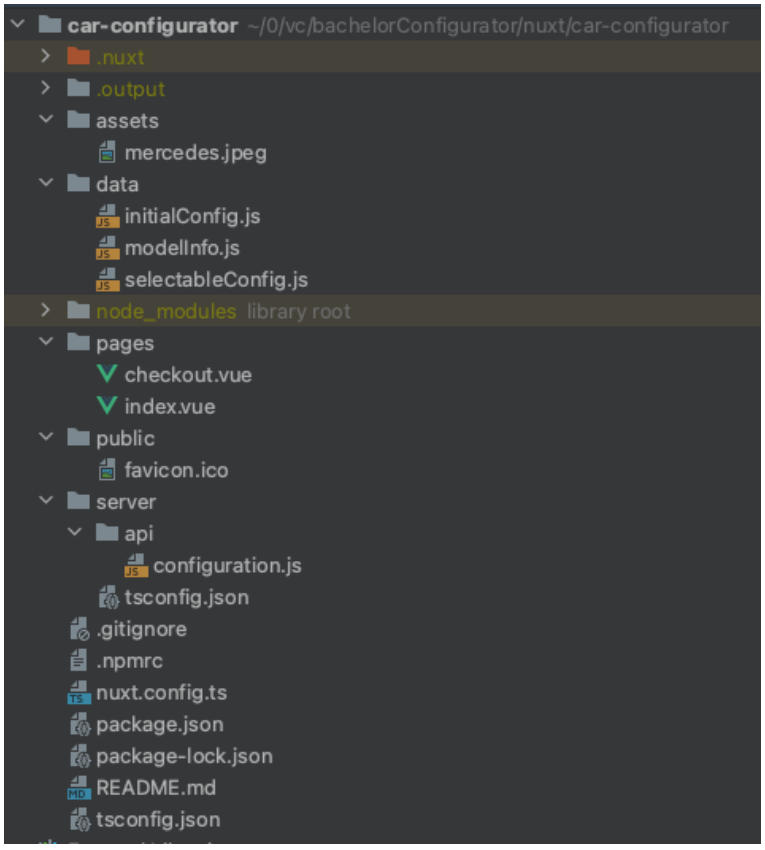

Figure 7 - Directory structure of the Nuxt.js implementation.

The screenshot above shows the root directory of the Nuxt.js implementation, which contains library directories (*/.nuxt* and */node_modules*), a build directory (*/.output*), an */assets* directory for content such as images, a */data* directory, a */pages* directory, a */public* directory, a */server* directory and configuration files (*.gitignore*, *.npmrc*, etc.). The */server/api* directory

and the */pages* directory work using the same mapping principle used in Next.js using vue components instead of React components.

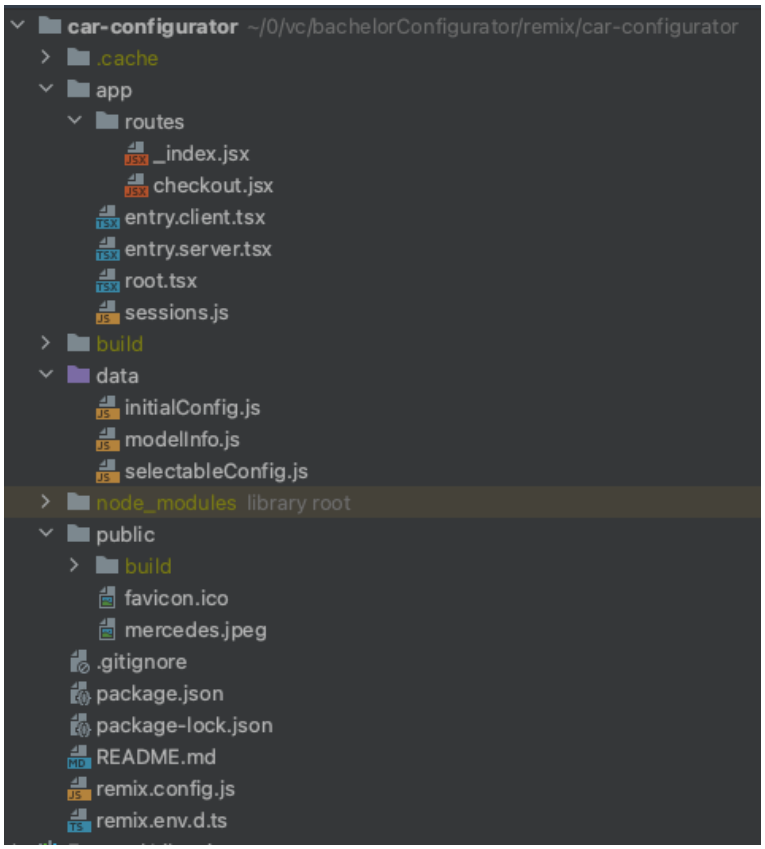

Figure 8 - Directory structure of the Remix implementation.

The screenshot above shows the root directory of the Remix implementation, which contains build directories (*/.cache*, */build* and */public/build*), a */public* directory, an */app* directory containing all the source code, a */data* directory, a library directory (*/node_modules*) a */public* directory and configuration files (*.gitignore*, *.remix.config.js*, etc.).

The */app/routes* directory works using the same route-to-path mapping principle used in the other two frameworks. The *entry.client.tsx*, *entry.server.tsx* and *root.tsx* files are all default remix files that come with a new application. The *sessions.js* is a remix file that can be used to identify requests from the same client, again to identify users and their corresponding configurations.

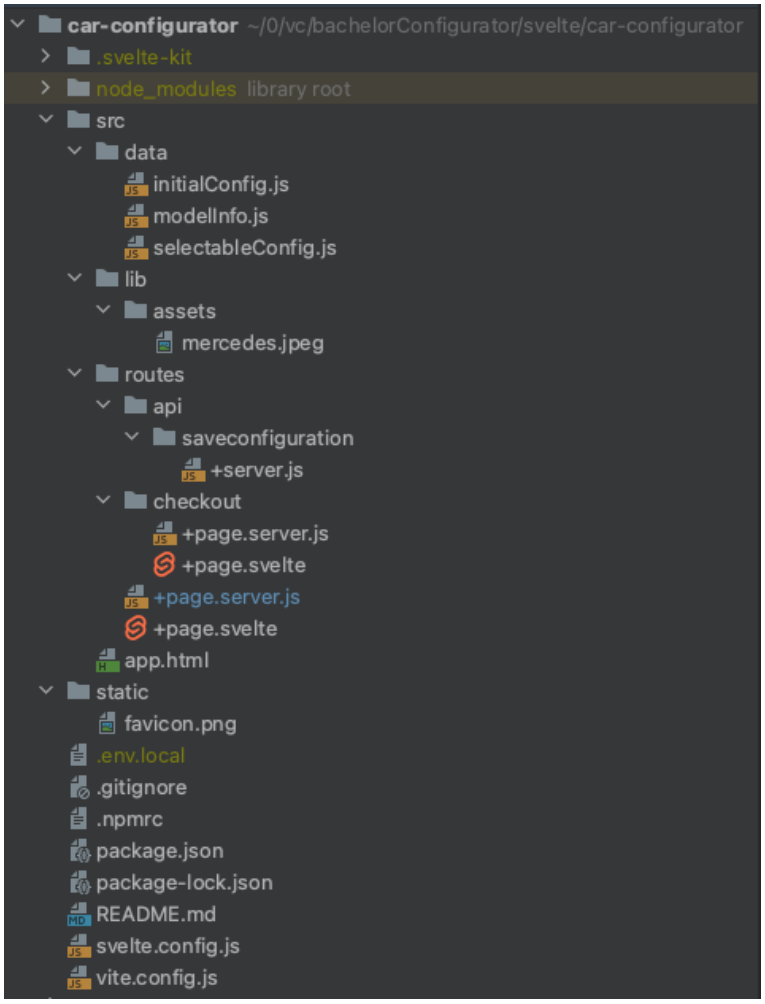

Figure 9 - Directory structure of the SvelteKit implementation.

Lastly, the SvelteKit implementation contains the following directories: library directories (/*.svelte-kit* and */node_modules*), a */src* directory, a */static* directory and configuration files (*.gitignore*, *.npmrc*, etc.).

The */src* directory contains the */data* directory, the */lib/assets* directory for content like images, the */routes* directory and the *app.html*. The */routes* directory in SvelteKit applications uses a similar mapping principal to the other frameworks. Directories within the /*routes* directory can be used to define paths, and default SvelteKit files can be placed inside these directories to define how SvelteKit should handle these paths (+*page.svelte* to define a svelte component and +*page.server.js* to define server-side code) The *app.html* is created when a new SvelteKit application is generated and contains boilerplate code.

2.3.3 Code

Since examining every line of code simply would not be possible within the scope of this thesis, it was decided to analyse how two typical use cases are handled in each implementation:

- Sending data from the backend to the frontend.
- Defining a session cookie.

Again, for readers that want to delve deeper into the code the repositories are public and can be accessed in *Table 4 – Relevant Resources of each Implementation.*

/src/pages/index.js

```
9   export async function getServerSideProps (context) {
10      return {
11          props: {
12              selectables: selectableData,
13              modelInfo: modelInfo,
14              initialData: initialData,
15              sessionid: context.req.cookies.sessionid
16          }
17      }
18  }
```

Figure 10 - Next.js Code Snippet showing data handling and session management.

In Next.js, the hybrid architecture seamlessly supports both server-rendering and static generation. The code presented above is an example of server-side rendering. The *getServerSideProps* function operates at request time and is responsible for fetching data. In the illustrated code, data elements like *selectables*, *modelInfo*, *initialData*, and the *sessionid* are conveyed as props to the Next.js page. This approach is typical in server-side rendering in Next.js, where data is retrieved and passed on as props.

The session is managed server-side through middleware. While Next.js doesn't have a built-in session management feature, it's highly compatible with standard Node.js middleware like

express-session or *cookie-session*. This setup allows for the efficient and secure management of session cookies, ensuring consistent data flow and state management across requests.

/server/api/configuration.js

```
38      const uuid = crypto.randomUUID();
39      setCookie(e, name: 'sessionid', uuid);
40
41      return {
42          sessionid: getCookie(e, name: 'sessionid'),
43          initialData: initialData,
44          selectables: selectableData,
45          modelData: modelInfo,
46      }
```

Figure 11 - Nuxt.js Code Snippet showing data handling and session management

Transitioning from the approach taken in Next.js to the methodology employed in Nuxt.js, it's evident that both frameworks emphasize efficient and reliable mechanisms for data handling. Nuxt.js, while sharing some similarities with Next.js, has its own distinct techniques, especially when it comes to the management of API routes.

While Next.js often relies on the context object within *getServerSideProps* for data retrieval and session management, Nuxt.js introduces methods such as *defineEventHandler* for robust event handling in API routes. The purpose of the *defineEventHandler* method is to delineate specific actions or responses for various request types or events. This structured approach allows for more granular control over the behavior of the API route.

In the context of session management, Nuxt.js employs functions like *getcookie* to extract session-related data directly from the request headers. This is somewhat parallel to how session data is accessed in Next.js through *context.req*. However, the modular design of Nuxt.js allows developers to define and utilize utility functions, like *getcookie*, which can be reused across different routes or modules, thereby promoting code reusability and maintainability.

/app/routes/_index.jsx

```
export async function loader({request}) {
    let session = await getSession(
        request.headers.get("Cookie")
    );

    let data = session.get("sessionid");
    if (data === undefined) data = crypto.randomUUID();
    session.set("sessionid", data);
    const cookie = await commitSession(session);
    console.log(data)

    return json(
        data: {
            selectables: selectableData,
            modelInfo: modelInfo,
            initialData: initialData,
            sessionid: data
        },
        init: {
            headers: {
                "Set-Cookie": cookie,
            }
        }
    );
}
```

Figure 12 - Remix Code Snippet showing data handling and session management.

Shifting the focus from Next.js and Nuxt.js, Remix emerges with its distinct strategies and methods. While there are commonalities in foundational web development principles across these frameworks, Remix stands out with its unique approach to data handling and session management.

In Remix, data transfer between the backend and frontend is orchestrated using the loader function. The *loader*, similar to the Next.js *getServerSideProps* function, acts as a conduit,

fetching data based on the incoming request and making it accessible to the frontend as props. As depicted in the code snippet, the *loader* function has access to the *request* object, which can be used to extract session or other contextual data.

When it comes to session management, Remix adopts a highly intuitive approach. The *session* object, as demonstrated, offers methods like *session.set* and *session.get* to respectively store and retrieve session-related data. Remix's encapsulation of session operations within a dedicated object provides a clear and structured approach to session handling.

This approach streamlines the process, making it more efficient and readable. Moreover, the seamless integration of session methods within the loader's context ensures a consistent and secure data flow between the backend and frontend, emphasizing Remix's focus on developer experience and robustness.

/src/routes/+page.server.js

```
6       if(!cookies.get('sessionid'))
7           cookies.set('sessionid', crypto.randomUUID());
8
9       return {
10          sessionid: cookies.get('sessionid'),
11          initialData: initialData,
12          selectables: selectableData,
13          modelData: modelInfo
14      }
```

Figure 13 - SvelteKit Code Snippet showing data handling and session management.

Finally, turning the attention to SvelteKit, yet another nuanced approach to backend-to-frontend data transmission and session management emerges. While each framework has its unique strengths and methodologies, SvelteKit's approach is characterized by its simplicity and directness.

In SvelteKit, the seamless communication between backend and frontend is facilitated by designated server routes, denoted by the + prefix in filenames like *+page.server.js*. These

server routes act as endpoints that the frontend can query or send data to. Unlike traditional server routes, SvelteKit's server routes coexist with the frontend, streamlining the development process. By placing server-side logic within these files, SvelteKit ensures a cohesive interaction between server and client components, allowing developers to manage both aspects in a unified environment.

In SvelteKit, cookie management plays a pivotal role in data flow and session handling. As demonstrated in the code snippet, methods such as *cookies.get* and *cookies.set* are employed to retrieve and store session-related data, respectively. The direct invocation of these methods underscores SvelteKit's lean and straightforward approach. The use of *cookies.set('session', randomUUID())* not only sets a session cookie but also assigns it a unique identifier, ensuring distinct sessions for different users or requests.

This method mirrors the session handling observed in the other frameworks but emphasizes SvelteKit's preference for minimalistic and efficient solutions. The inherent simplicity of SvelteKit ensures that developers can maintain a clear understanding of data flow and session states, reducing the potential for errors and simultaneously enhancing the overall development experience.

2.4 Deployment

In order to ensure that the metrics examined later in this paper would not be affected by variances in hosting providers, the same hosting platform was used for each implementation. Vercel[12] was chosen since the author was familiar with the platform, and its benefits with regards to ease of deployment and integration, prior to the writing of this thesis.

2.4.1 Vercel: A quick Overview

Vercel provides an optimized platform for hosting serverless applications. Serverless applications are applications that are designed to execute functions in response to events without the need for continuous server management, relying on cloud services to automatically manage the infrastructure, scaling, and execution.

[12] https://vercel.com

It supports over 35 frameworks and offers integration with the “Build Output API”[13], allowing seamless transformation of diverse codebases into deployable entities. Deployments can be initiated through the Command Line Interface (CLI) or an automated git-based workflow, which was used for the deployments performed in the context of this thesis. During deployment, files are uploaded to a robust data storage service, with system integrity checks ensuring unauthorized access prevention. Vercel's build container processes the code, accommodating numerous frontend frameworks or prebuilt projects.

The system dynamically serves static resources, invokes serverless functions, and optimizes images based on the requesting client's browser, all while caching responses to enhance subsequent requests' efficiency. In essence, Vercel offers a streamlined, secure, and globally optimized hosting infrastructure, mitigating traditional hosting challenges and ensuring peak performance for end-users.

2.5 Results

Using the guidelines (1.5.1) and controlled variables (1.5.2) defined in the methodology, the car configurator was implemented in each of the four frameworks and deployed to Vercel as outlined above. The table below shows the URLs and code repositories of each implementation.

Framework	URL	Code Repository
Next	https://car-configurator-seven.vercel.app/	https://github.com/supermancash/car-configurator
Nuxt	https://car-configurator-nuxt.vercel.app/	https://github.com/supermancash/car-configurator-nuxt
Remix	https://remix-car-configurator.vercel.app/	https://github.com/supermancash/remix-car-configurator
SvelteKit	https://svelte-car-configurator.vercel.app/	https://github.com/supermancash/svelte-car-configurator

Table 4 – Relevant Resources of each Implementation.

[13] https://vercel.com/blog/build-output-api

The images below show the configuration (main page) and confirmation page of the Nuxt implementation as a visual example. The other frameworks aren't shown since the sites are visually identical.

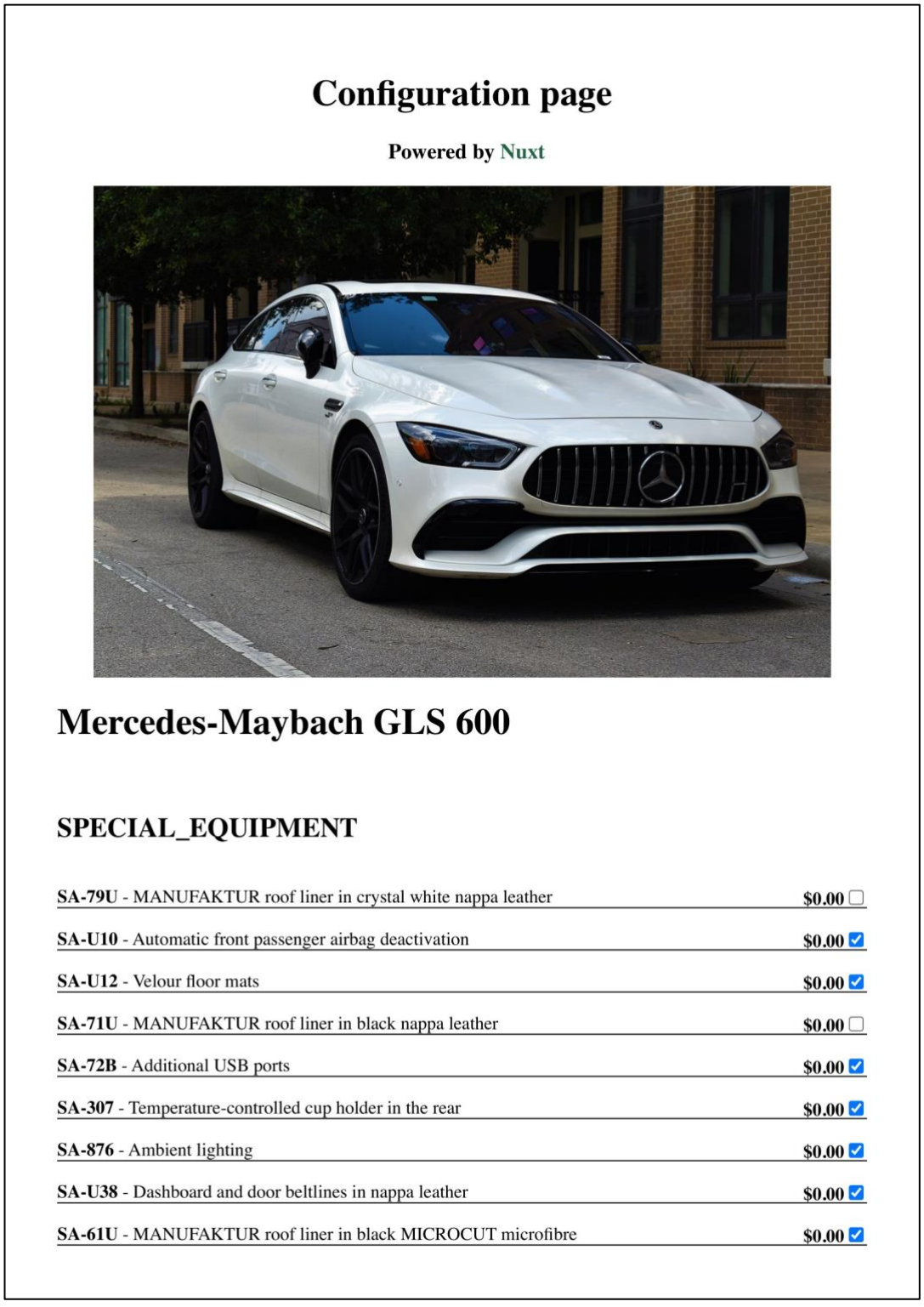

Figure 14 - The first part of the Configuration Page of the Nuxt Configurator

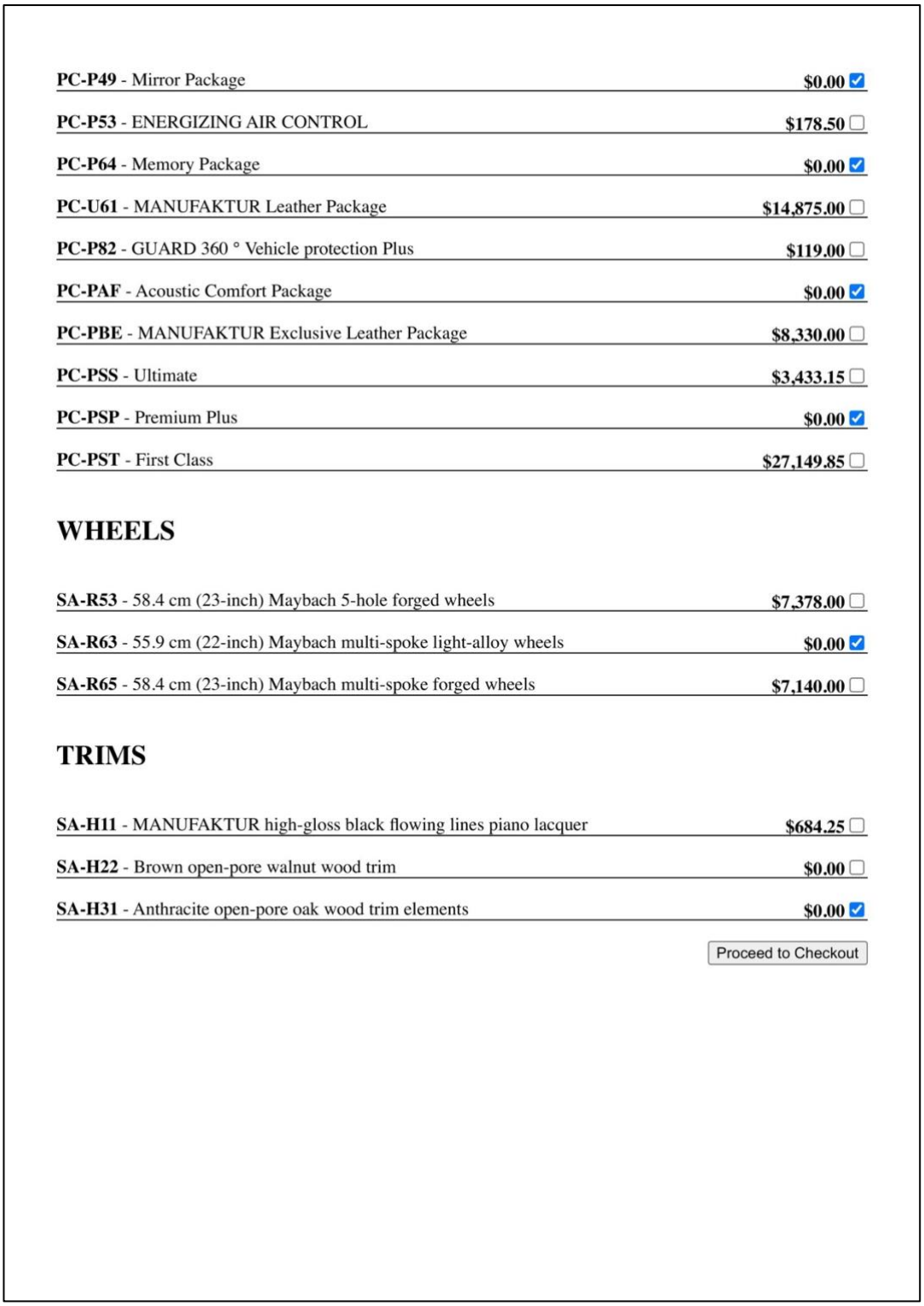

Figure 15 - The second part of the Configuration Page of the Nuxt Configurator

The configuration page, as described in the corresponding user story, allows the user to view an image of the car, basic details, create their own configuration by changing the checkboxes and proceeding to the next page using the button seen in the image above.

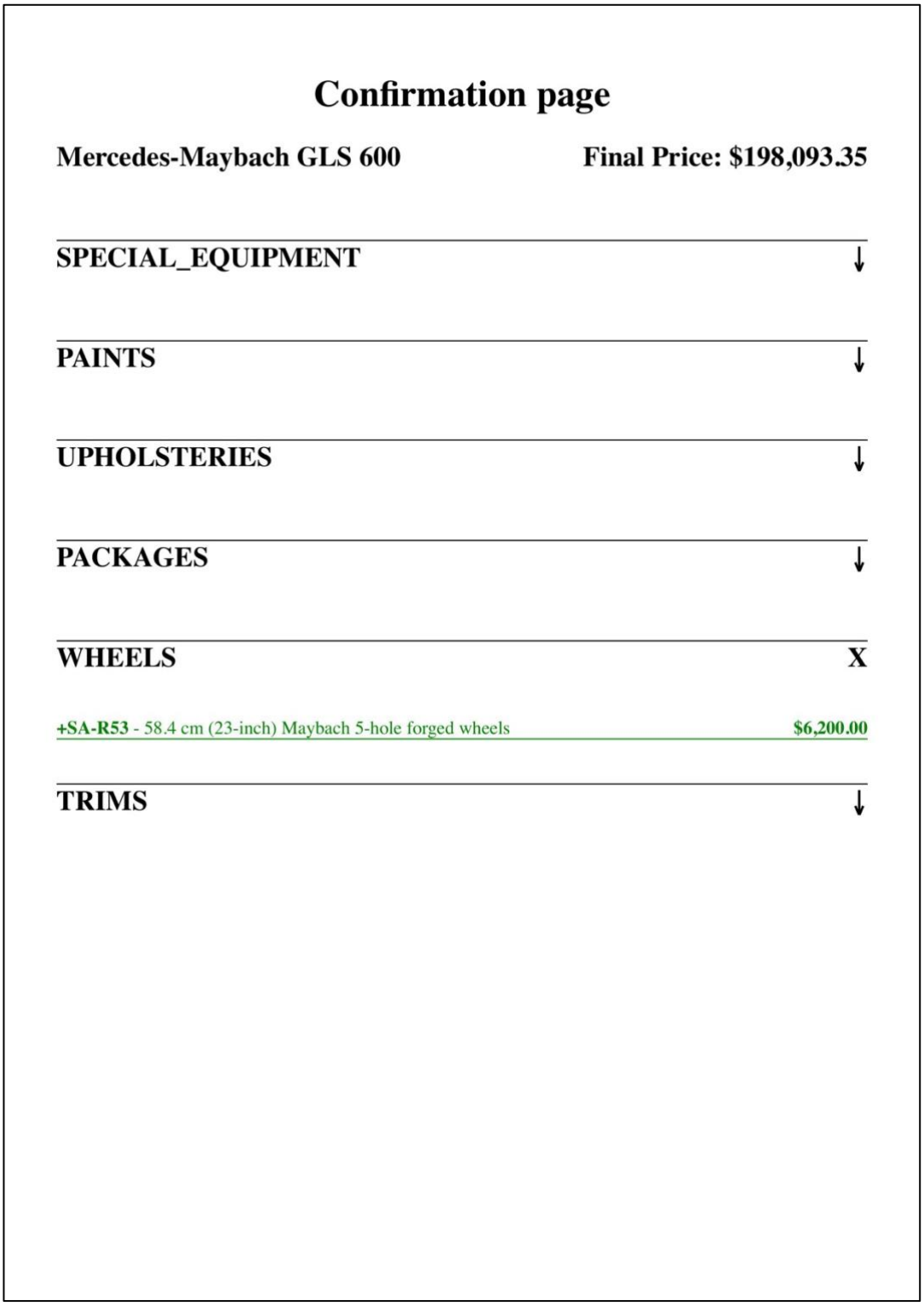

Figure 16 - The Confirmation Page of the Nuxt Configurator

The confirmation page, as also described by the corresponding user story, allows the user to see the final price and details of their configuration. As can be seen above, the different categories are presented as accordions, with the wheels accordion opened to show how an additional equipment is represented.

2.5.1 User Metrics

Framework	**PLT**(ms)	**TTFB**(ms)	**FCP/LCP**(s)
Next	752	443	0.4/0.8
Nuxt	641	268	0.3/0.9
Remix	740	250	0.3/0.6
SvelteKit	595	207	0.4/0.6

Table 5 - Results obtained for the selected user-oriented quality metrics of the four Frameworks.

Next.js is renowned for its hybrid rendering capabilities, enabling both server-side rendering (SSR) and static site generation (SSG). This flexibility can occasionally lead to slightly higher PLTs due to server-rendering overheads. However, its robust server-side capabilities are evident in the competitive TTFB.

Nuxt.js, akin to Next.js, also supports SSR and SSG but is optimized for Vue.js. Its slightly improved PLT and TTFB could be attributed to the Vue.js reactivity system, ensuring efficient updates and server responses. The framework's modular structure might also contribute to rapid content paints.

Remix, while demonstrating a PLT similar to Next.js, boasts a particularly impressive TTFB. This can be attributed to its optimized server-side data fetching and routing mechanisms, which prioritize delivering data to the client swiftly.

SvelteKit stands out with the lowest PLT and TTFB. Svelte's unique approach of compile-time optimization, as opposed to run-time reactivity, could be the driving factor behind these metrics. By minimizing the client-side JavaScript payload and maximizing efficiency, SvelteKit offers brisk initial loads and server responses.

In the context of car configurators, these performance metrics are paramount. Car configurators demand fluidity in interactions and immediate feedback. For instance, a user customizing vehicle features expects seamless transitions and rapid content updates. Here, a framework's TTFB and FCP/LCP play a pivotal role: a swift server response ensures timely data retrieval, while rapid content paints enhance perceived performance.

Furthermore, the architectural nuances of these frameworks have direct implications for car configurators. A configurator built with SvelteKit might benefit from its compile-time optimizations, ensuring smooth animations and transitions. In contrast, a Next.js or Nuxt.js-based configurator might leverage their robust server-side capabilities to efficiently handle complex configurations and data interactions.

While all the examined frameworks offer robust performance metrics suitable for car configurators, understanding the underlying characteristics of each framework can guide more informed decisions, ensuring optimal user experience and backend efficiency.

2.5.2 Developer Metrics

Developer metrics are crucial in understanding the ease and efficiency with which applications can be developed using different full-stack JavaScript frameworks. By examining the quality metrics defined earlier, the following insights were gained in the development process and associated research:

1. **Documentation Quality:**

    - Next.js: Renowned for its detailed and beginner-friendly documentation, it provides clear guides, tutorials, and API references. The documentation covers a vast array of topics, from basic setups to advanced configurations.
    - Nuxt.js: With comprehensive documentation, it includes extensive examples and use-case scenarios, catering to both newcomers and seasoned developers.
    - Remix: As a relatively newer framework, its documentation is evolving but is lauded for its clarity and directness.
    - SvelteKit: SvelteKit benefits from Svelte's well-established documentation, offering clear explanations and interactive examples. However, it would be difficult to confidently state that the documentation of concepts relating specifically to the SvelteKit framework, while comprehensive and extensive, can compete with how well Svelte itself is documented.

2. **Community Support and Ecosystem:**

- Next.js: Boasting a vibrant community, it has a plethora of plugins, extensions, and third-party integrations. Forums and discussion boards are active, with frequent contributions.
- Nuxt.js: Riding on the popularity of Vue.js, its community is vast and responsive. Numerous plugins and modules are available for extended functionalities.
- Remix: Its community is growing steadily, with an increasing number of contributions and discussions in online forums. The frameworks lack of history becomes evident when researching less common errors online.
- SvelteKit: With Svelte's backing, it has a supportive community and a burgeoning ecosystem of tools and extensions.

3. **Consistency and Predictability:**

- Next.js: Leveraging its hybrid architecture, Next.js maintains a consistent API and behavioral pattern, allowing developers to easily predict its behavior across various rendering modes.
- Nuxt.js: Building on Vue.js, Nuxt.js inherits its consistent component structure and naming conventions, making the framework predictable for developers familiar with Vue.
- Remix: Despite being relatively new, Remix has shown commitment to maintaining a uniform API, helping developers quickly get accustomed to its patterns.
- SvelteKit: SvelteKit, benefiting from Svelte's clear syntax, extends this consistency, making it easier for developers to anticipate framework behaviors.

4. **Modularity and Flexibility:**

- Next.js: Its plugin architecture and modular design mean developers can add functionalities as needed, without bloating the application.
- Nuxt.js: Built with Vue.js's component-based architecture in mind, it offers high modularity and flexibility for various application needs.
- Remix: Its design promotes modular development, with each route being its own isolated component, ensuring flexibility in application design.

- SvelteKit: The framework's component-based nature promotes modularity, and its compile-time approach ensures flexibility in shaping the final output.

5. **Clear Error Messages and Debugging Tools:**

- Next.js: Known for its intuitive error messages, it provides actionable feedback during development. Integrated debugging tools further assist in issue resolution.
- Nuxt.js: Detailed error messages coupled with Vue.js's devtools make debugging a streamlined process. In certain cases, the root cause of an error is difficult to decipher from the error message given.
- Remix: It emphasizes developer feedback, offering clear error messages that guide toward solutions.
- SvelteKit: Benefiting from Svelte's compiler-based architecture, it offers precise error messages at compile-time, aiding in early issue detection.

6. **Scalability:**

- Next.js: With its ability to switch between static generation and server-side rendering, it's poised to handle applications that scale both in user numbers and complexity.
- Nuxt.js: The framework's modular design and Vue's reactivity system ensure that applications can scale smoothly as they grow.
- Remix: The framework's architecture, emphasizing optimal server-side operations, ensures that applications remain performant as they scale.
- SvelteKit: Its compile-time optimization approach means that the resulting codebase is lean, ensuring applications scale efficiently.

7. **Integration with Tools and Services:**

- Next.js: Offers seamless integrations with a plethora of tools and services, making it developer-friendly.
- Nuxt.js: With Vue's ecosystem at its core, it supports a wide range of plugins and integrations, ensuring developers have the tools they need.

- Remix: Being newer, its integration ecosystem is growing, but the framework's design ensures ease of integration with popular tools.
- SvelteKit: The framework, backed by Svelte's ecosystem, provides robust integration capabilities with various development tools and services.

In the context of car configurators and similar complex web applications, these developer metrics take on added significance. Given the intricacies involved in developing configurators, clear documentation and strong community support can expedite development and problem resolution. Modular design ensures that the application remains maintainable and scalable, while consistency and predictability reduce the learning curve for developers. Furthermore, the ability to integrate seamlessly with other tools and services streamlines the development process. As configurators are meant to provide real-time feedback and updates, clear error messages and robust debugging tools are indispensable in ensuring a glitch-free user experience.

3. Conclusion

Web applications have experienced a significant shift in their developmental frameworks, tools, and methodologies in the past decade. Central to this evolution has been the rise of full-stack JavaScript frameworks, which promise to deliver seamless integration of frontend and backend development. Within this dynamic milieu, this research embarked on a quest to dissect the technological underpinnings, performance metrics, and adaptability of four prominent frameworks: Next.js, Nuxt.js, Remix, and SvelteKit.

3.1 Research Questions

1. **What are the key technological differentiators among popular JS full-stack frameworks and how do these differences impact their performance and suitability for web application development?**

The exploration commenced with an understanding of the distinct technological architectures that each framework brings to the table. Next.js, with its hybrid architecture, has showcased its adaptability in catering to diverse rendering needs. On the other hand, Nuxt.js, building upon Vue.js's robustness, presents a reactivity system that ensures swift and efficient updates.

Remix, albeit younger in its market presence, has carved a niche with its optimized server-side data fetching mechanisms. SvelteKit, pushing the boundaries of traditional frameworks, emphasizes compile-time optimizations over runtime reactivity.

When these technological differentiators are mapped to performance, each framework exhibits its strengths and potential areas of improvement. The analysis revealed that while SvelteKit offers impressive PLT and TTFB scores, likely attributed to its streamlined compile-time approach, Next.js, with its extensive server-rendering capabilities, offers a performance that's tailored to applications with complex data interactions.

2. **How do the selected full-stack JavaScript frameworks address the challenges posed by the characteristics of the typical demands of a web application?**

Delving into the heart of web application demands, the research adopted the lens of a car configurator - a complex, interactive, and data-intensive application. A car configurator, by its very nature, necessitates real-time feedback, intricate data interactions, smooth animations, and user-centric interactivity. Translating these demands to framework capabilities, several observations surfaced.

Next.js, with its adept server-rendering, stands as a strong contender for applications like car configurators that rely on real-time data interactions and feedback. Nuxt.js, leveraging Vue.js's modular structure, ensures swift content paints and updates, essential for applications demanding real-time feedback. Remix's structured approach to data handling, combined with its focus on delivering data to the client expeditiously, ensures a seamless user experience. SvelteKit, with its focus on compile-time efficiencies, promises brisk user feedback and fluid animations.

3. **Can clear winners be defined, both in terms of technological superiority and context suitability?**

In the world of technology, the term "winner" is often fluid and context-dependent. While SvelteKit's performance metrics hint at a technological edge, especially in the domain of car configurators that require rapid initial loads, Next.js's vast community support and in-depth

documentation make it an appealing choice for developers, especially those building complex web applications like car configurators.

However, when viewed through the prism of context, the landscape shifts. For web applications mirroring the intricacies of car configurators, where user interactions, real-time feedback, and data-intensive operations converge, the choice of framework becomes paramount. Here, while SvelteKit might offer rapid feedback, Next.js or Nuxt.js, with their robust server-side capabilities, might be more aligned for handling intricate data operations and interactions.

3.2 Conclusion and Forward path

The journey through this research has illuminated the multifaceted world of full-stack JavaScript frameworks. While performance metrics, technological architectures, and developer metrics offer a compass, the true north in selecting a framework lies in understanding the specific demands of the web application in question.

As the digital landscape continues to evolve, with web applications like car configurators pushing the boundaries of interactivity and user experience, the choice of framework will invariably impact the end product's success. It's imperative for developers and stakeholders to holistically assess not just the technological prowess of a framework but also its adaptability to the context at hand.

In the grand tapestry of web application development, this research serves as a beacon, guiding decisions, and offering insights into the dynamic world of full-stack JavaScript frameworks.

4. Discussion

In the swiftly evolving realm of web application development, the surge of full-stack JavaScript frameworks has paved the way for integrated solutions catering to both frontend and backend necessities. This transformation has been spearheaded by tools and methodologies that were previously non-existent or in their nascent stages a decade ago. As

elucidated in the conclusion, the versatility and adaptability of frameworks like Next.js, Nuxt.js, Remix, and SvelteKit have proven to be instrumental in this transition.

Each framework's intrinsic architectural design plays a pivotal role in determining its efficacy in specific scenarios. For instance, Next.js's hybrid architecture bestows it with the dexterity to adapt to varied rendering needs, while Nuxt.js capitalizes on Vue.js's robustness to render swift updates. These unique architectural elements not only differentiate the frameworks but also shape their suitability for diverse applications.

Contrasting frameworks in the context of demanding web applications, like car configurators, underscores their diverse strengths. While SvelteKit's compile-time optimizations cater to rapid initial loads, frameworks like Next.js and Nuxt.js bring forth a prowess in intricate data operations, a testament to their server-side capabilities. Such insights provide an empirical basis for developers, steering their choices based on the specific requirements of their projects.

While raw performance metrics are undeniably paramount in assessing the utility of a framework, other intangible assets must also be brought into the limelight. Next.js's expansive community support and comprehensive documentation, for example, are invaluable assets that can significantly smoothen a developer's journey. The integration of community wisdom and the availability of shared solutions can often expedite problem-solving and bolster the overall development process.

As reiterated in the research, the definition of a "winner" in the technological sphere is a confluence of multiple variables, and most critically, the context. The realm of web applications is incredibly diverse, and the same tool might resonate differently across projects. Hence, the pursuit is not just to chase technological superiority but to seek a symphony between the framework's capabilities and the contextual intricacies of the application in focus.

The current digital landscape is a dynamic one, with web applications continually pushing the thresholds of user interactivity and experience. As this landscape evolves, the intersection of technological advancements with the nuanced requirements of web applications will continually redefine the best practices and tools of choice. In such an ecosystem, periodic

reviews and research like the present one will remain invaluable, offering necessary recalibrations and guiding developers towards informed decisions.

In the intricate maze of web application development, choosing the right framework is less about absolutes and more about understanding the subtleties of each project's demands. As the industry marches ahead, research and discourse will remain the beacons, illuminating the path and ensuring that the digital realm remains responsive, intuitive, and above all, user centric.

Future research should pivot towards a longitudinal study, examining the adaptability and resilience of these frameworks in the face of emerging web technologies, such as WebAssembly, Edge Computing, and Progressive Web Applications. Moreover, as user expectations metamorphose, with an increasing demand for immersive experiences like Augmented Reality (AR) and Virtual Reality (VR) in web applications, it would be crucial to assess how these frameworks can be optimized or integrated with such advanced technologies. Additionally, the broader ecosystem, encompassing factors like developer training, community support, and security protocols, deserves in-depth exploration to truly discern which frameworks will stand the test of time and technological advancements. Finally methods that can be applied to examine the specific demands and requirements of a project should be constructed, that allow for informed decisions to be made when choosing a framework.

Declaration in lieu of oath

I hereby declare that I produced the submitted paper with no assistance from any other party and without the use of any unauthorized aids and, in particular, that I have marked as quotations all passages, which are reproduced verbatim or nearby verbatim from publications. Also, I declare that the submitted print version of this thesis is identical to its digital version. Further, I declare that this thesis has never been submitted before to any examination board in either its present form or in any other similar version. I herewith agree that this thesis may be published. I herewith consent that this thesis may be uploaded to the server of external contractors for the purpose of submitting it to the contractors' plagiarism detection systems. Uploading this thesis for the purpose of submitting it to plagiarism detection systems is not a form of publication.

Berlin, 26.10.2023

Signed, Emil Triest

Bibliography

[i] *Fast load times*. (n.d.). web.dev. https://web.dev/explore/fast

[ii] Akamai Technologies. (2014). *Consumer Web Performance Expectations Survey*

[iii] *Web Vitals*. (2020). https://web.dev/vitals/

[iv] Aigner, M., Hütter, T., Kirsch, C., Miller, A., Payer, H., & Preishuber, M. (2014). ACDC-JS. *Proceedings of the 10th ACM Symposium on Dynamic Languages*, 67–78. https://doi.org/10.1145/2661088.2661089

[v] Turnbull, J. (2014). *The Art of Monitoring*.

[vi] Jacob, B., Ng, S., & Wang, D. (2010). *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann.

[vii] Robillard, M. P. (2009). What makes APIs hard to learn? Answers from developers. *Software, IEEE*, 26(6), 27-34.

[viii] Storey, M. A. D., Cheng, L. T., Bull, I., & Rigby, P. C. (2006). Shared waypoints and social tagging to support collaboration in software development. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work* (pp. 181-190).

[ix] Stylos, J., & Myers, B. A. (2007, September). Mica: A web-search tool for finding API components and examples. In *2007 IEEE Symposium on Visual Languages and Human-Centric Computing* (pp. 195-202). IEEE.

[x] Bosch, J. (1999). Design and use of software architectures: Adopting and evolving a product-line approach. Pearson Education.

[xi] Ko, A. J., & Myers, B. A. (2005). A framework and methodology for studying the causes of software errors. *Transactions on Software Engineering*, 33(10), 685-700.

[xii] Herbst, N. R., Kounev, S., & Reussner, R. (2013). Elasticity in cloud computing: What it is, and what it is not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*.

[xiii] Wohlin, C., & Aurum, A. (2005). Criteria for selecting software requirements prioritization techniques. In *Proceedings of the 2005 conference on Software Engineering Research and Practice in Sweden* (pp. 139-146).

[xiv] Weber, N. (2022). *Evaluation and Comparison of Full-Stack JavaScript Technologies*.

[xv] Adhikari, A. (2016). *Full Stack JavaScript: Web Application Development with MEAN*.

[xvi] Swarali. (2021). *The benefits of using React JS*. https://doi.org/10.5281/zenodo.5383371

[xvii] Peter Pšenák, & Matúš Tibenský. (2020). The usage of Vue JS framework for web application

[xviii] Bhardwaz, S., & Godha, R. (2023). Svelte.js: The Most Loved Framework Today. *2023 2nd International*

[xix] Riva, M. (2022). *Real-World Next.js*.

[xx] Kok, L. T. (2020). *Hands-on Nuxt.js Web Development*.

[xxi] Julian Laubstein. (2022). Web-Apps mit SvelteKit erstellen. *IX*, *10*, 144–147.