Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Technologies

Bachelor Thesis

Web application development using Next.Js

Kairat Nurakhmet

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

BACHELOR THESIS ASSIGNMENT

Kairat Nurakhmet

Informatics

Thesis title

Web application development using Next.Js

Objectives of thesis

The main aim of this thesis is to analyse the impact of the usage of Next.Js React framework for developing Web application and demonstrate possible use cases of this Framework on a prototype application utilizing communication via API services between front and back end.

The partial objectives are :
– To conduct a comprehensive literature review, exploring relevant use cases of the old ways of developing web applications and the use of the modern Next.Js framework.
– To examine the impact of Next.Js framework on the web application design and programming technique.
– Demonstrate the use of Next.Js on a prototype web application.

Methodology

The methodology of the thesis is based on the author's own research and the study of relevant information resources, using qualitative analysis of documents and external desk research focusing on Next.js framework. Based on the synthesis of the knowledge gained, a prototype web application will be developed and tested. On the basis of this practical example, conclusions and implications for Next.js use will be formulated.

The proposed extent of the thesis

40-50 pages

Keywords

Web development, Front-end, Next.js

Recommended information sources

De, Brajesh: API Management An Architect's Guide to Developing and Managing APIs for Your Organization, Berkeley, CA : Apress, 2017, ISBN: 978-1-4842-1305-6, 9781484213063

Elrom, Elad: React and libraries your complete guide to the React ecosystem, New York: Apress, 2021, ISBN: 978-1-4842-6696-0

FLANAGAN, David. *JavaScript : the definitive guide.* Sebastopol, CA: O'Reilly, 2002. ISBN 0-596-00048-0.

Griffiths, David: React cookbook: recipes for mastering the React framework, Beijing ; Boston ; Farnham ; Sebastopol ; Tokyo : O'Reilly, 2021, ISBN: 978-1-492-08584-3

Horton, Adam: Mastering React, master the art of building modern web applications using React, Birmingham, Mumbai: Packt publishing, 2016, ISBN: 978-1-78355-856-8.

Konshin, Kirill: Next.js quick start guide, Packt Publishing, 2018, ISBN: [1-78899-366-7; 1-78899-584-8]

Tyson, Matthew: The best new features in Next.js 13, InfoWorld.com; San Mateo: 2022, 2737142348.

Expected date of thesis defence

2022/23 SS – FEM

The Bachelor Thesis Supervisor

Ing. Petr Hanzlík, Ph.D.

Supervising department

Department of Information Engineering

Electronic approval: 7. 3. 2023

Ing. Martin Pelikán, Ph.D.

Head of department

Electronic approval: 13. 3. 2023

doc. Ing. Tomáš Šubrt, Ph.D.

Dean

Prague on 19. 02. 2024

Declaration

I declare that I have worked on my bachelor thesis titled "Web application development using Next.Js' by myself and I have used only the sources mentioned at the end of the thesis. As the author of the bachelor thesis, I declare that the thesis does not break any copyrights.

In Prague on 15/03/2024 _____

Acknowledgement

I would like to express my sincere gratitude to Ing. Petr Hanzlík, Ph.D. of the Department of Information Technologies at the Czech University of Life Sciences, Prague. He was always ready to help whenever I was in trouble and had any query regarding my thesis. He supported me with his relevant advice and encouragement in every step throughout the creation of this thesis.

I would like to thank all my previous and current colleagues that I worked with for their support, guidance, and encouragement throughout the completion of my bachelor's thesis.

Web application development using Next.Js

Abstract

This research work examines Next.js which is a revolutionary structure that has been built in relation to React and works towards making web application development better by improving efficiency, performance, and the experience of the developers. By leveraging React's extensive ecosystem and component-oriented architecture, Next.js provides advanced capabilities such as server-side rendering, static site generation and API communication which introduces modern web applications to a whole new level. Using numerous academic sources, this paper documents the evolution of frameworks in web development with emphasis on how React and Next.js overcome the limitations of traditional methods. This article goes further to examine one instance where an application was created using Next.js as well as its usefulness practically. Consequently, these have made it possible for Next.js based on underlying technology of React to become a simpler way leading to reducing time spent when developing scalable unique applications.

Keywords: Next.js, React, Framework, Web Development, Server-Side Rendering, Static Site Generation, API Communication, Modern Web Applications, Efficiency, Scalability, Developer Experience.

Vývoj webové aplikace pomocí Next.js

Abstrakt

Tato výzkumná práce zkoumá Next.js, což je revoluční struktura, která byla vytvořena v souvislosti s Reactem a pracuje na zlepšení vývoje webových aplikací zlepšením efektivity, výkonu a zkušeností vývojářů. Díky využití rozsáhlého ekosystému React a komponentově orientované architektury poskytuje Next.js pokročilé možnosti, jako je vykreslování na straně serveru, generování statických stránek a komunikace API, které zavádí moderní webové aplikace na zcela novou úroveň. Tento článek s využitím četných akademických zdrojů dokumentuje vývoj frameworků ve vývoji webových aplikací s důrazem na to, jak React a Next.js překonávají omezení tradičních metod. Tento článek dále zkoumá jeden případ, kdy byla aplikace vytvořena pomocí Next.js, a také jeho praktickou využitelnost. Ty následně umožnily, aby se Next.js založený na základní technologii React stal jednodušším způsobem vedoucím ke zkrácení času stráveného při vývoji škálovatelných unikátních aplikací.

Klíčová slova: Next.js, React, Vývoj Webu, Serverové Vykreslování, Generování Statických Stránek, Komunikace s API, Moderní Webové Aplikace, Efektivita, Škálovatelnost, Zkušenosti Vývojářů.

Table of content

1 Introduction

The quest for better, smarter, and more user-friendly web applications in today's world doesn't seem to have an ending. This chase has led to the improvement of web development methodologies and the adoption of different frameworks that are designed to simplify the development process while making sure there is a performance boost and enhanced user experience. Out of this bunch comes Next.js, a React framework that brings the best of server-side rendering (SSR), static site generation (SSG), and client-side rendering into one place. This thesis will explore the impact Next.js has on web application development, placing its modern approach side by side with traditional methodologies and showcasing its potential through the creation of a prototype application that makes use of API services to create smooth front-end and back-end communication.

The internet has changed everything about how business is done, how we communicate, and how innovation works. As such, it's not surprising at all how quickly technologies used for web development have evolved as well. Traditional methods often had distinct demarcations between front-ends - which were mostly focused on user interface and experience - and back-ends - which handled server logic and database interactions. These siloed processes required different skill sets, so they weren't always efficient. However, with Node.js came Next.js — a framework that allows developers to use JavaScript across both ends of a stack. This shift made things easier when it came to optimizing performance for users.

Next.js is a framework that makes it easy to build fast, scalable applications. It is useful for SEO in the digital world we live in now. Our pre-rendered servers make pages load faster and static site generation eliminates the need to rebuild every time they are visited – only rendering new pages when needed. Next.js also automatically splits code so users will only download what they need for the page they're visiting. However, one of our most powerful features has to be Next.js' ease of communication between front-ends and back-ends. So how does this modern approach compare to traditional development frameworks? To find out, I'll do a comparison based on qualitative analysis documents from external desk research. We'll get an even closer look at its efficiency and flexibility in practice by developing a prototype web application and testing it out. Hopefully, the tests can give us some concrete results that we can further analyze later on. In simple terms, this project aims

to explore Next.js' potential impact on design and programming techniques in modern web development environments. Using a prototype project as a base test, I hope to reveal its true functionality under practical circumstances. With our findings we aim to identify key strengths and any obstacles or weak points anyone considering Next.js may face — developers, businesses, or educators alike — so everyone can make informed decisions going forward.

2 Objectives and Methodology

2.1 Objectives

The main aim of this thesis is to analyse the impact of the usage of Next.Js React framework for developing Web application and demonstrate possible use cases of this Framework on a prototype application utilizing communication via API services between front and back end.

The partial objectives are:

- To conduct a comprehensive literature review, exploring relevant use cases of the old ways of developing web applications and the use of the modern Next.Js framework.
- To examine the impact of Next.Js framework on the web application design and programming technique.
- Demonstrate the use of Next.Js on a prototype web application.

2.2 Methodology

The methodology of the thesis is based on the author's own research and the study of relevant information resources, using qualitative analysis of documents and external desk research focusing on Next.js framework. Based on the synthesis of the knowledge gained, a prototype web application will be developed and tested. On the basis of this practical example, conclusions and implications for Next.js use will be formulated.

3 Literature Review

3.1 Javascript

JavaScript is a scripting language, which is used to dynamically manipulate the content of a website. The scripting language is characterized by the interpretation of the source code, rather than using a compiler to convert it into machine code. Since JavaScript is used in web pages, all web browsers contain a built-in engine, which can render JavaScript code. That is the reason, why JavaScript does not require any additional programs to run, and inserting the code into an HTML document is sufficient for it to get executed. JavaScript is a scripting language used to manipulate content on a website. Compared to compiled languages, it interprets source code, meaning it doesn't need a compiler to turn the code into machine code. All web browsers come with an engine that can parse JavaScript, so there's no need for additional software—just inserting the code into an HTML document is sufficient for it to get executed.

3.1.1 How Javascript Works in Browser

Once JavaScript receives the command from a web browser, it makes several transformations to turn into interactive components that are found on a Web page. First, the browser loads the HTML document and comes across JavaScript code which is usually embedded within <script> tags. The next step for the browser's JS engine is to parse the script meaning that it translates source code into a memory-based structure of abstract syntax tree (AST) representing its code organization. Thus, depending on the engine, this AST can be executed directly or compiled into bytecode and sometimes even optimized machine code. This process is incredibly efficient in that it allows just-in-time compilation and execution without any noticeable delays.

In other words, when running, this will enable the JavaScript Engine to interact with the DOM of web pages thus making them have dynamic content alteration possibilities. For instance, it can change existing HTML elements, add new ones, or remove some completely thereby altering what users see and the layout of that page at large. It is also capable of responding to user actions such as clicks on links or buttons; keyboard input like typing in a text field; or maybe page load events thus making a webpage more responsive and

interactive. What's amazing about this has been how well-integrated JavaScript has become with both HTML and CSS to enable the development of sophisticated web applications that run smoothly on today's browsers without extra plug-ins/software because they use built-in JS engines.

3.2 React.Js

React serves as a JavaScript library that helps to make user interfaces, concentrating mostly on single-page applications. It gives the possibility to make big web applications that can refresh data without reloading a page. React's main advantage is that it allows for the composition of complicated UIs through simple isolated code snippets known as components. It is built in such a way that developers describe what their user interface should look like by use of declarative-style programming and then React will effectively update the components and re-render them when any changes occur in data. The virtual DOM system of React optimizes modifications made within the actual DOM, thus increasing website efficiency and usability.

3.2.1 How React works

On top of handling the virtual DOM, React uses a process called reconciliation to find out which parts of the actual DOM need to be updated based on changes evident in the virtual DOM. For this reason, we compare the new virtual DOM with its previous version and determine what kind of minimal changes are required for such updates.

React components are small entities that can be reused over and again when developing user interfaces. The structure and behavior of these components combine to create complex UIs from smaller building blocks that developers can compose together. It is characterized by code reusability, maintainability as well as scalability within a given framework.

React's data flows in one direction only so that it can predictably manage states and help realize declarative UIs. Instead of directly changing their interface or even managing the state with an imperative language, such programmers must be able to decide how exactly each part should appear at any given time based on the current application state. By

abstracting away manual manipulation complexities from developers, React simplifies their work thus automatically updating the DOM whenever there is a change in them.

3.2.2 Advantages of React:

3.2.2.1 Component-Based Architecture

This encourages the use of reusable UI components that lead to more manageable code and faster development.

3.2.2.2 Declarative Approach

This simplifies code readability and maintenance since developers can specify how the user interface should appear under different conditions.

3.2.2.3 Virtual DOM

It increases application performance by reducing direct manipulations on the DOM thus making updates and rendering quicker.

3.2.2.4 Flexibility

By other frameworks or libraries incorporated in the tech stack, React can work with them hence offering flexibility in terms of development choices made. Also

3.2.3 Disadvantages of React:

3.2.3.1 Routing in React

Perhaps one of the most difficult parts of working with React is making routing work. This may seem like it is not a big deal but adding routing to your application can mean adding a lot of other things as well, and that can be a little complex. The React Router Dom NPM package is mostly used to implement this functionality in React. However, one must also keep up with current trends and industry standards on routing to avoid future issues that may arise as technology advances. Not doing so may result in navigation complications such as failed navigations, URLs being handled differently, or challenges faced in managing application states across various routes. Consequently, these difficulties may cause time lags

in project delivery or inconsistencies between the project requirements and routing implementations for it.

3.2.3.2 SEO Optimization Challenges

Client-side rendering, a common feature in traditional React applications, is often implemented where minimal initial HTML content is delivered to the browser and this is followed by JavaScript populating the contents of the page dynamically. This method provides an opportunity for interactivity as well as a smoother user experience but makes it hard for search engine bots. The first HTML sent to the browser may not contain substantial content; hence, search engine bots may find it difficult to index such pages.

Consequently, when compared with other web pages that are inaccessible or have less content on display, this sets back its visibility and ranking from being appropriately crawled and indexed. Therefore, traditional React applications that do not optimize SEO could be potentially non-existent on SERPs (search engine result pages).

3.3 Next.js

Next.js is an open-source framework built with React and intended for creating JavaScript applications that feature single-page applications (SPA). It is characterized by aspects like server-side rendering, static site generation, and incremental static regeneration which enable its performance as well as Search Engine Optimization. File-system-based routing, automatic code splitting, and optimized prefetching are among the ways Next.js makes it easy to develop high-performance and scalable web apps.

3.3.1 Advantages of Next.js

3.3.1.1 Server-Side Rendering (SSR)

Next.js has built-in support for server-side rendering, which means that a web page can be created on the server and then sent to the client with content already in it. SSR speeds

up initial load times, helps in SEO by providing pre-rendered HTML to search engines, and ensures better performance especially when there is lots of data.

3.3.1.2 Static-Site Generation (SSG)

Next.js supports static site generation where pages are rendered before they are built out. This way, when building the application, each page's HTML file is generated thus eliminating server-side rendering on each request. SSG results in faster page loads; less server load and improved SEO therefore ideal for sites with content or blogs.

3.3.1.3 Incremental Static Regeneration (ISR)

Incremental Static Regeneration is a new feature that combines the benefits of static site generation and dynamic content updating. This time-based method allows developers to keep the pages refreshed without doing full site rebuilds by just simply pre-rendering them in certain intervals or upon request. The Next.js ensures that static pages are created as they are requested to achieve a balance between static site efficiency and dynamic content flexibility which gives users up-to-date material while maintaining peak performance.

3.3.1.4 Automatic Code Splitting

Next.js automatically breaks down the JavaScript bundles based on the page boundaries thus ensuring that only what is needed is loaded for every other page individually. This will help reduce load time at start-up and optimize resource utilization thereby improving user experience, particularly on low network connections.

3.3.1.5 File-Based Routing

Routing becomes easier because Next.js uses a built-in file-based approach where page directory structure governs app routes directly. This simple routing scheme avoids complex configurations allowing easy organization and management of routes making the codebase more maintainable.

3.3.1.6 Built-in CSS and Sass Support

To style using CSS or SASS, developers can directly import stylesheets into components using Next.js. This makes styling modularized and scoped as well as enhances code organization while removing set-up or reliance on third-party libraries thereby boosting developer productivity.

3.3.1.7 API Routes

Within an application, developers can create an API route through which communication between client-side and server-side components can be seamless. These API routes could handle requests for fetching data, authenticating users, and other server-side operations that are important in building full-stack applications without additional backend frameworks.

3.3.1.8 TypeScript Support

In Next.js there is built-in support for Typescript which is a statically typed superset of JavaScript. Code quality is improved by TypeScript, developer productivity is enhanced, and the likelihood of runtime errors is reduced through the addition of type checking and smart suggestions during development.

3.4 Next.js framework environment setup

3.4.1 Node.js and NPM

Next.js applications use Node.js as their runtime environment; hence, the installation of Node.js is essential for Next.js development. It was bundled with NPM which among other things, takes care of libraries and dependencies that support Next.js framework features.

“npm is the world's largest software registry. Open source developers from every continent use npm to share and borrow packages, and many organizations use npm to manage private development as well.”[1]

[1] NPM documentation. Accessed 8 November 2023. Available from: https://docs.npmjs.com/about-npm

3.4.2 Source-code Editor

Microsoft develops Visual Studio Code (abbreviated VSCode) as a free open-source source-code editor for programming languages used by many developers. Syntax highlighting, intelligent code completion, snippets, code refactoring, and embedded Git control are some of its features supporting various programming languages. VSCode's extensibility allows the developers to extend it by adding different languages, themes, or even debuggers and also to connect with more services (VScode Official Documentation). The powerful tools available in it when designing web and cloud applications compared to its lightness make it common among programmers who work on both frontend and backend projects.

3.4.3 Create-next-app

Create-next-app command is designed to help developers streamline the process of creating a new Next.js project, by generating an optimized directory structure that enhances the performance and scalability of your projects. It also sets up a development environment with hot module replacement making it easy for you to test and iterate on your app.

3.4.4 Project Structure

Routes are built using files within the pages directory making it easy to locate them because each file represents a route as presented here: "Each file inside the pages directory becomes a route that gets automatically processed" (Next.js Official Documentation).

3.4.5 Package.json:

Package.json file is at the root of every Nextjs project through which you define the scripts necessary for running your development server, building your app ready for production, or starting your finished product again in case it goes down due to any reason i.e., crashes or something happens unexpectedly. In addition, this manifest file guarantees other developers on the team can install all the project dependencies and execute them consistently.

4 Practical Part

4.1 Architecture

4.1.1 Component-Based Architecture (CBA)

In Component-Based Architecture (CBA), web applications are broken down into components, which are independent and reusable pieces of application. Every component holds a part of the UI and logic for the application requiring a clear division of labor. This is different from a traditional monolithic approach which results in more organized development.

What Next.js adds server-side rendering abilities to React's component model, in addition to an extensive range of optimization features. In this framework, components do not only make up the UI but also form the basis of such important elements in the overall structure as routing, data fetching, and state management.

4.1.2 Document Object Model (DOM)

"The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects; that way, programming languages can interact with the page."[2]

The Virtual DOM in both Next.js and React is a lightweight clone of the real DOM. It facilitates fast web page updating by reducing direct changes on the actual DOM which may be slow. When a certain component's state changes, Next.js will first update the Virtual DOM, then compare this updated version with the previous one that was captured and finally analyze how to apply those changes to the original DOM more effectively. This technique called "diffing" ensures a performance boost owing to reduced excessive updates that are inevitable especially when it comes to dynamic applications like games among others.

[2] What is the DOM? Accessed 20 November 2023. Online Source. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction#what_is_the_dom

4.1.3 JSX

"JSX is a syntax extension of JavaScript that combines the JavaScript and HTML-like syntax to provide highly functional, reusable markup.

It's used to create DOM elements which are then rendered in the React DOM. While not required in React, JSX provides a neat visual representation of the application's UI.

A JavaScript file containing JSX will have to be compiled before it reaches a web browser."[3]

JSX allows developers to write HTML-look-like code in Javascript files. What seems like HTML in JSX is hardly that. Instead, you see a set of functions operating behind the scenes to make optimized JavaScript code for creating elements. The real magic of JSX is how it lets us mix rendering with UI logic, so developers can use logic inside views and before they're rendered. This kind of integration makes event handling easier, state management more responsive, and conditional content rendering flow like water — all making the development process more intuitive and efficient.

4.1.4 Routing

Routing in Next.js is file-based and the file-naming convention generates a route for each page in the *pages* directory. Developers need to include JS or JSX files in this folder. It'll automatically generate routes as required. Each file name will be a segment of the URL. For navigating between two pages Next.js provides a built-in component named *Link* of next/link library which supports client-side navigation.

4.1.5 Client-Side Rendering (CSR)

Client-Side Rendering HTML file in the user's browser, using JavaScript. Unlike Server-Side Rendering (SSR), where the complete HTML document is generated on the server, CSR involves sending only the bare minimum HTML, CSS, and JavaScript needed to load the page. Once these assets are loaded, JavaScript runs in the browser to generate the HTML content dynamically.

---

[3] Writing Markup with JSX. Accessed 12 November 2023. Online Source. Available from: https://react.dev/learn/writing-markup-with-jsx

4.1.6 Server-Side Rendering (SSR)

Server-Side Rendering generates the complete HTML document on the server for the requested page in response to a user, this is a completely different approach than client-side rendering, where the HTML document is generated on the client-side and may require more time and resources.

"Server-side rendering (SSR) is where the server sends a ready-to-render HTML page and the JS scripts that are required to make the page interactive."[4]

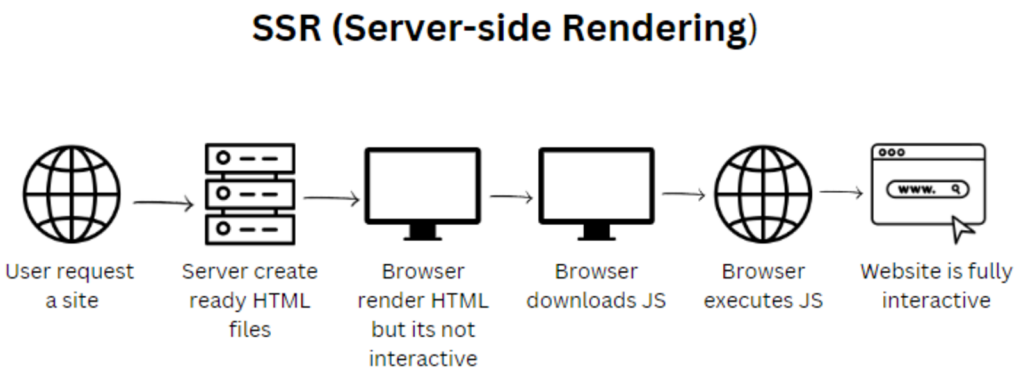


Figure 1: SSR (Server-Side Rendering)

4.1.7 Static-Site Generation (SSG)

"Static Site Generation describes the process of compiling and rendering a website at build time. The output is a cluster of static files, including the HTML, Javascript, and CSS files."[5]

Pre-rendering in Next.js is a Static Site Generator (SSG) technique where pages are built to be pre-generated at build time, hence resulting in some static HTML files. This process is good for pages whose content changes infrequently because it makes the website load very fast as well as helps in making the SEO better since this content can easily be found on search engines. It is in Next.Js that the developers use SSG to enhance performance, improve user experience through fast content delivery, and keep content still and secure, meeting the efficiency and dependability requirements of web application architecture.

---

[4] Server-side Rendering vs Static Site Generation in Next.js. Accessed 21 November 2023. Online Source. Available from: https://nextjs.org/docs/pages/building-your-application/rendering/static-site-generation
[5] Server-side Rendering vs Static Site Generation in Next.js. Accessed 21 November 2023. Online Source. Available from: https://nextjs.org/docs/pages/building-your-application/rendering/static-site-generation

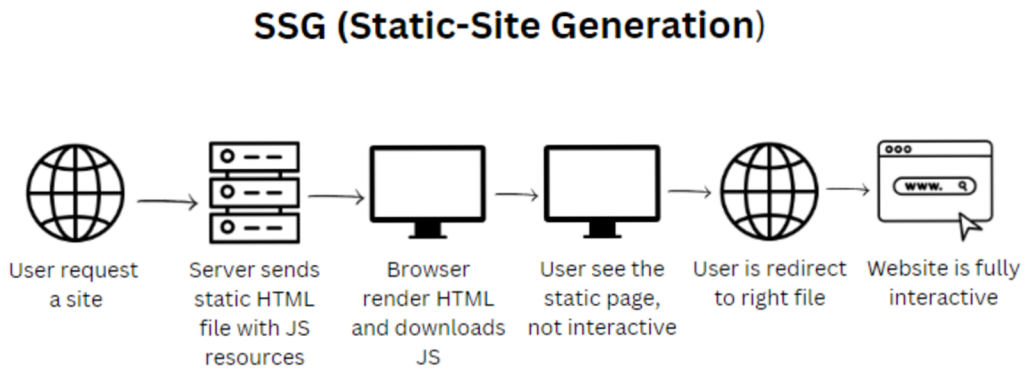


Figure 2: SSG (Static-Site Generation)

4.1.8 Incremental Static Regeneration (ISR)

In Next.js, the Incremental Static Regeneration (ISR) enables static pages to be regenerated at runtime to allow for dynamic updating of static content. Through ISR, developers can set situations where a page would be re-rendered; this enhances content freshness while holding on to the performance advantages provided by static generation. In web application development, it is a way of having websites display fresh information without incurring the server-side processing costs for each request at the same time striking a balance between performance and flexibility in terms of dynamic contents.

4.2 Environment and Conditions

This section describes the conditions under which the required tests were conducted, as well as the design of these tests and the metrics chosen to quantify the differences in the work done. This is to ensure reproducibility of results and overall transparency. To mimic real-world data interactions in both conventional and Next.js apps, will be used the **https://jsonplaceholder.typicode.com/posts** endpoint to fetch dummy data. Such standard datasets provided by this endpoint are like actual blog posts and can be used as a basis for comparing how well the two development styles perform in terms of data fetching functionalities.

4.2.1 Experimental Applications

The goal of the thesis will profoundly expound on the construction of two different web applications that had been designed from the same design file with three core pages, including Home, Blogs, and Blog details pages (as demonstrated in **Figures 3**, **4,** and **5**).

Both applications will share the same CSS file, disparities in styling will not account for any observed differences in performance, structure, and interactivity because these will be attributed to the development frameworks and methodologies used. This consistency in styling will maintain a uniform appearance of both applications such that it is easy to focus on the technical and architectural aspects of each development approach.

The first application is going to be built in a conventional way using HTML, CSS, and JavaScript thus creating an old-fashioned web structure. It will be used as a basic understanding of normal web development practices.

The second one will use Next.js on top of the **create-next-app** npm package for modern web development technologies. This app would help us understand how efficient or effective Next.js can be for building large-scale dynamic website applications. There is an analysis to compare such areas as project structure, API interaction, and routing to show the differences between traditional and Next.js based application developments.

The goal is to empirically determine what are the enhancements or differences related to handling several common tasks faced during web development while using Next.js compared to a usual approach.

4.2.1.1 Home page

All pages, including the Home page, will have a navigation bar with 3 links: Logo, Home, and Blogs.

The Home page will include a banner that represents the promoted blog post as demonstrated in **Figure 3** below.

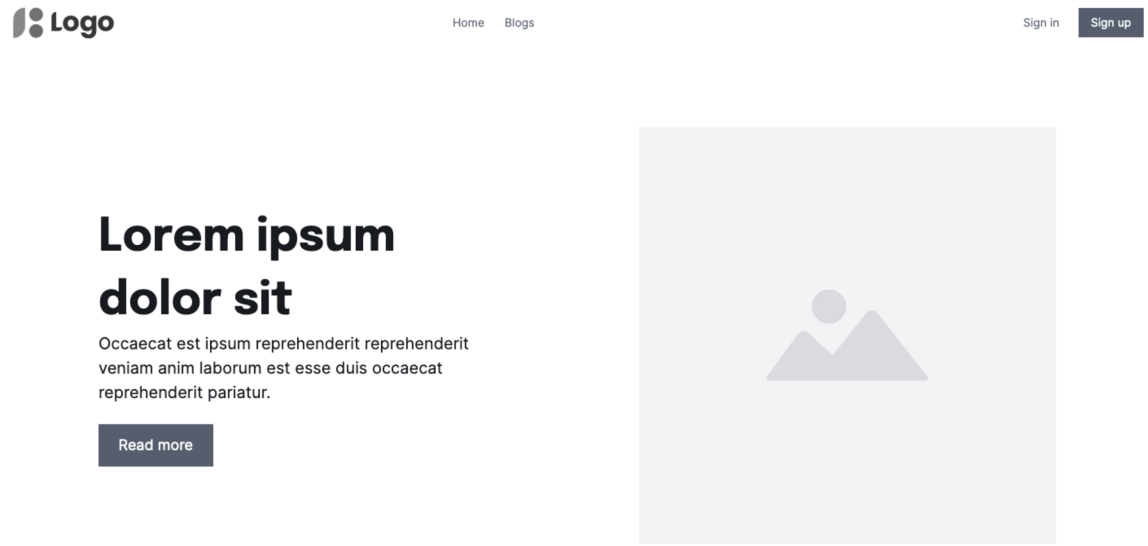


Figure 3: Home page picture

4.2.1.2 Blogs list page

The blog list page displays a grid of blog post cards, clicking on a card will redirect the user to the blog details page of that post. **Figure 4** below demonstrates the page:

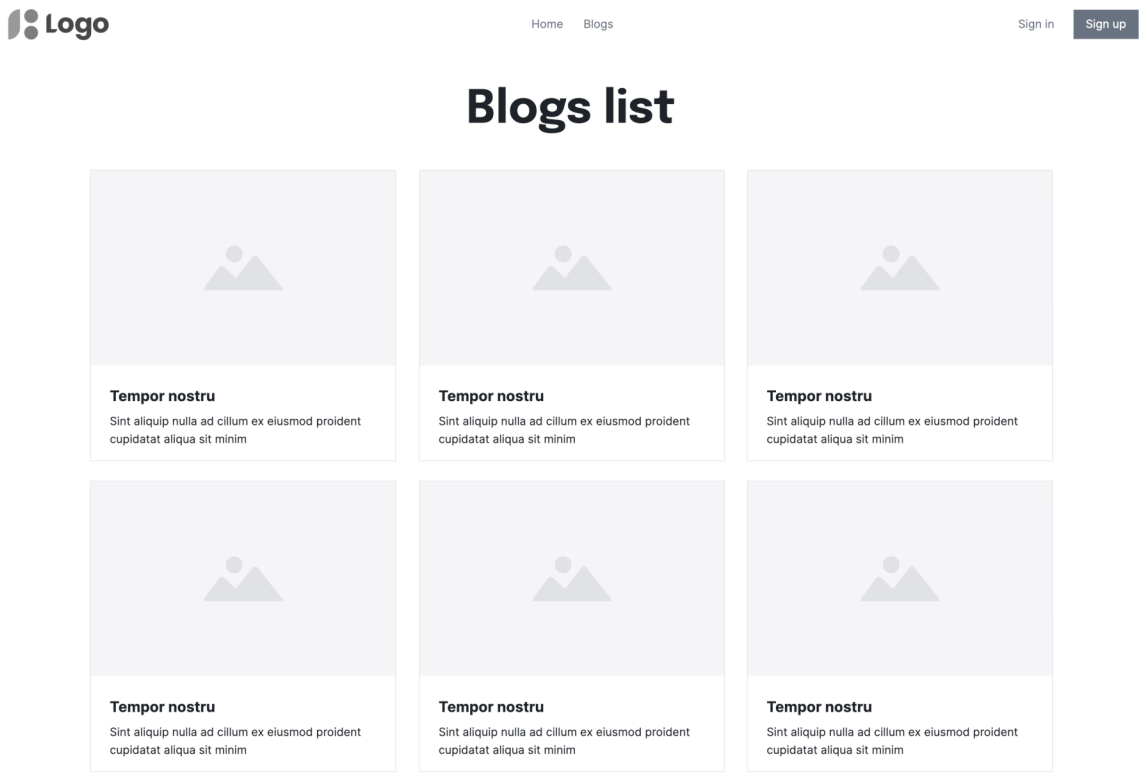


Figure 4: Blogs list picture

4.2.1.3 Blog details page

The blog details page displays the full content of a single blog post as demonstrated in **Figure 5** below.

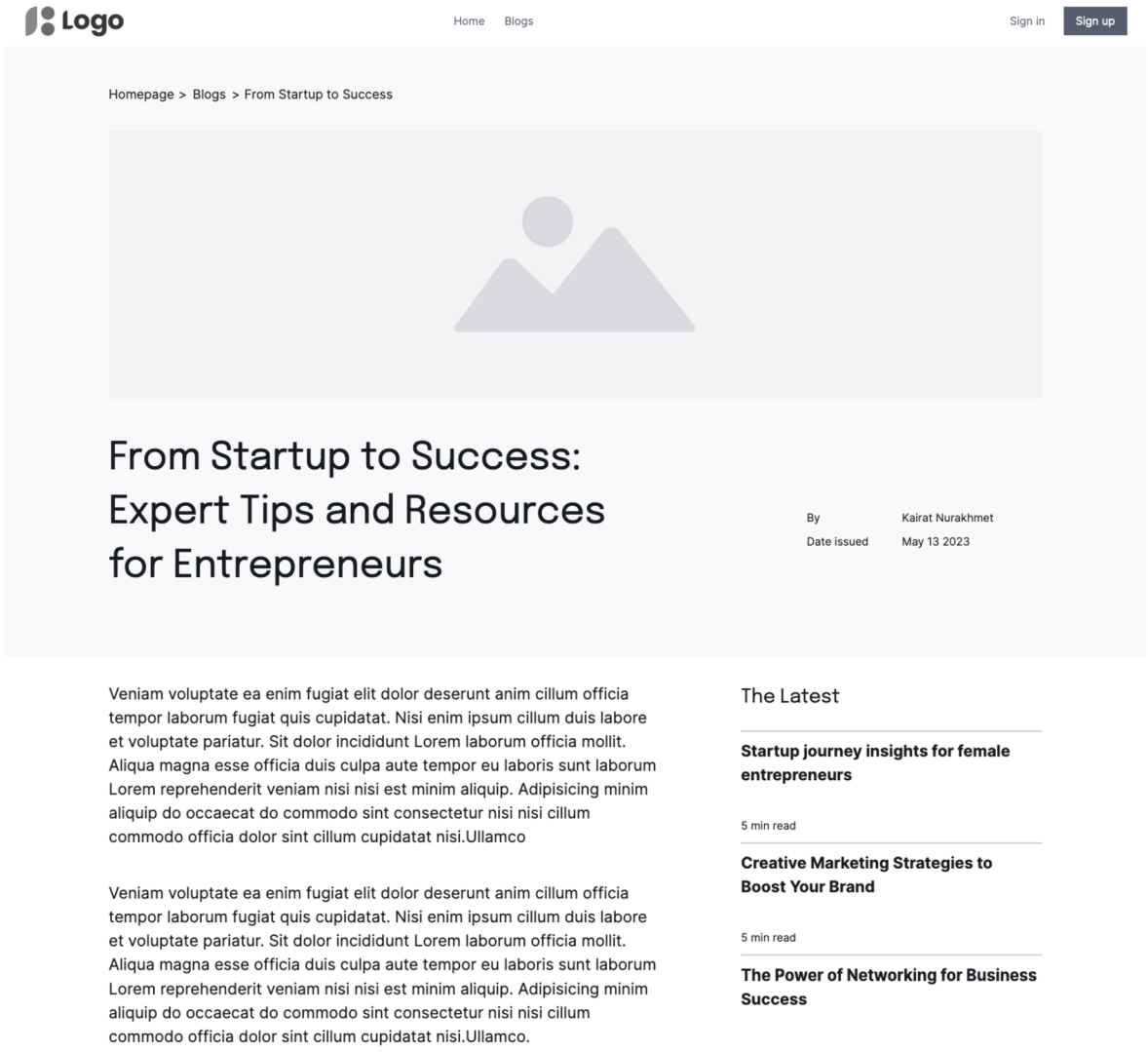


Figure 5: Blog details picture

4.3 Environment

The following is a table detailing the present versions of the tools due to be employed throughout unit testing phases:

Table 1: List of tools and respective versions.

TOOL	VERSIONS
Node	20.11.0
create-next-app	14.1.2
React	18.0
react-dom	18.0
Next	14.1.0
Npm	10.3.0
VS Code Counter	3.4.0
Prettier – Code formatter	10.1.0
Google Lighthouse	11.4.0

The following code demonstrated in **Source code** represents the **settings.json** file's content in the VS Code editor, highlighting how it ensures consistent formatting across the project's codebase.

Source code 1. Content of the settings.json file

```
{
  "editor.formatOnSave": true,
  "editor.tabSize": 2,
  "editor.autoClosingQuotes": "always",
  "editor.defaultFormatter": "esbenp.prettier-vscode",
  "[javascript]": {
    "editor.defaultFormatter": "esbenp.prettier-vscode"
  },
  "prettier.jsxSingleQuote": true,
  "prettier.singleQuote": true,
  "prettier.bracketSameLine": true,
  "prettier.printWidth": 120,
  "prettier.useEditorConfig": false,
  "editor.accessibilitySupport": "off",
  "workbench.colorTheme": "Default Light Modern"
}
```

In the practical part of the thesis, an extension tool VS Code Counter[6] tests a chosen code analysis utility. This is important for evaluating the lines of code (LOC) in both traditional applications and those built on Next.js. With this tool, developers can count lines of code in real-time and get statistics for entire directories or workspaces. Also, it is essential for its compatibility with language-specific extensions to ensure correct language identification while analyzing source codes. Therefore, results as shown by graphical representations via the CodeViz Stat extension bring visual and analytical dimensions to the evaluation of coding efforts in thesis projects. The tool also provides Real-time Counter Visibility which provides the ability to count the range of the selected code.

For formatting purposes, an extension tool Prettier – code formatter[7]. The code formatter will be employed to ensure code consistency across the project. This extension imposes a fixed set of regulations to automatically format code by deleting inconsistencies and enhancing ease of reading. The **settings.json** configuration sets Prettier as the default formatter and other coding standards such as style of quotes, line length, and where to put brackets for correct spacing. It makes sure that all codes have uniformity in their styles.

[6] VS Code Counter. Accessible from: https://github.com/uctakeoff/vscode-counter
[7] Prettier – Code formatter. Accessible from: https://github.com/prettier/prettier-vscode

4.4 Conventional Way website

4.4.1 File structure

Figure 6 illustrates the structure of the source code for the application in the project directory which is displayed in a tree structure. The **assets** folder is where all the images are used on the website. This small-scale application has a single CSS file at its root for styles. On larger production projects, however, styles would usually be separated into separate folders or files. Whenever users go to this site's root URL, they land on index.html as it is an entry point to that application.

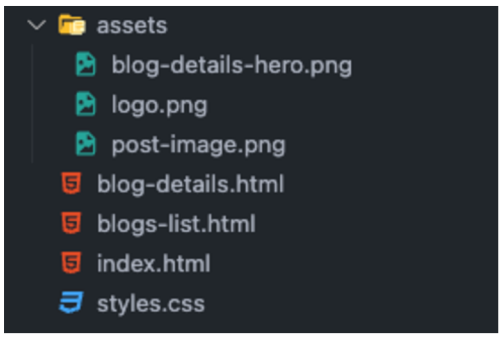


Figure 6: Conventional way file structure

4.4.2 Layout

To be consistent across pages, the layout of the application is designed in such a way that there is a header section that does not change. The unvarying nature of this heading promotes user-friendliness, so that the main content within the <main> tag differs from page to page to deliver unique content, but its overall structure and navigation elements are familiar to users wherever they are within the app. This makes it easier for people who use it since they only must concentrate on individual and simple contents being loaded after each page load of dynamic contents into their main sections. The layout of the website is demonstrated below in **Figure 7**.

```html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link rel="stylesheet" href="./styles.css" />
    <title><!-- Page title --></title>
    <meta name="description" content="Welcome to HTML" />
  </head>
  <body>
    <header>
      <a href="/"><img src="./assets/logo.png" alt="Logo of the website" /></a>
      <nav>
        <ul class="nav_pages">
          <li><a href="/">Home</a></li>
          <li><a href="blogs-list.html">Blogs</a></li>
        </ul>
        <ul class="nav_auth">
          <li><a href="">Sign in</a></li>
          <li><a href="" class="nav_sign-up">Sign up</a></li>
        </ul>
      </nav>
    </header>
    <main>
      <!-- Main content goes here. -->
    </main>
    <script>
      <!-- Script content goes here. -->
    </script>
  </body>
</html>
```

Figure 7: Website layout

4.4.3 Home page

This is an **index.html** file in a traditional project where we are establishing the structure of a simple web page. In the **body**, the article element is located which showcases the featured blog post in the **main** section. Initially, this post shows "Loading…" text which later changes into content fetched from an API endpoint using JavaScript's **fetch** method. This script dynamically modifies both the title and body of the post showcasing how JavaScript blends with static HTML for interactive user experience. Contents of the **main** and **script** tags are demonstrated below in **Figure 8**.

```
<main class="home_main">
  <article class="hero">
    <section class="hero_text">
      <h1 id="post-title">Loading...</h1>
      <p id="post-body">Loading...</p>
      <a href="/blog-detail.html?id=1" class="btn">Read more</a>
    </section>
    <section class="hero_img">
      <img src="./assets/post-image.png" alt="Post image mask" />
    </section>
  </article>
</main>
<script>
  fetch('https://jsonplaceholder.typicode.com/posts/1')
    .then((response) => response.json())
    .then((data) => {
      document.getElementById('post-title').textContent = data.title;
      document.getElementById('post-body').textContent = data.body;
    })
    .catch((error) => console.error('Error fetching post:', error));
</script>
```

Figure 8: Contents of main and script tags (index.html)

4.4.4 Blogs list page

In the **blogs-list.html** code snippet, JavaScript fetches a blog post list from placeholder API. Then it renders the posts dynamically throughout the webpage. Initially, the document's main section is empty and can be filled with content later on. The page loads and the script executes, fetching posts and iterating through them. An element **(<a>)** that has an image, title, and body of the post is created for each post. This is then appended to the articles-grid section. It demonstrates dynamic client-side rendering whereby the contents are loaded when a user interacts with it or when a page loads. Demonstration of the **main** and **script** contents are below in **Figure 9**.

```
<main class="blogs-list_main">
  <h1 class="blogs-list_heading">Blogs list</h1>
  <section class="articles-grid" id="articles-container">
    <!-- Dynamic content renders here. -->
  </section>
</main>
<script>
  fetch('https://jsonplaceholder.typicode.com/posts?_start=0&_limit=6')
    .then((response) => response.json())
    .then((posts) => {
      const container = document.getElementById('articles-container');
      container.innerHTML = ''; // Clear existing content
      posts.forEach((post) => {
        const article = document.createElement('a');
        article.classList.add('blog-card');
        article.href = `/blog-details.html?id=${post.id}`;
        article.innerHTML = `
        <article">
          <img src="./assets/post-image.png" alt="Post image" />
          <section>
            <h2>${post.title}</h2>
            <p>${post.body}</p>
          </section>
        </article>
      `;
        container.appendChild(article);
      });
    })
    .catch((error) => console.error('Error fetching posts:', error));
</script>
```

Figure 9: Contents of main and script tags

4.4.5 Blog details page

In the **main** tag of the **blog-details.html** file, the layout prominently features elements like a hero section with an image and breadcrumb navigation, a title (**h1** with id **article-title-h1**), and a paragraph (**p** with id **article-body**) that displays the blog post's content as demonstrated below in **Figure 10**. These elements are dynamically filled with data via JavaScript to provide detailed information about the blog post.

```html
<main class="blog-details_main">
  <article class="blog-details_article">
    <section class="blog-details_hero">
      <section class="blog-details_breadcrumb">
        <a href="/">Home</a>
        <span>&#x276F;</span>
        <a href="/blogs-list.html">Blogs list</a>
        <span>&#x276F;</span>
        <span id="article-title">From Startup to Success</span>
      </section>
      <img src="./assets/blog-details-hero.png" alt="Post image mask" />
      <section class="blog-details_hero-bottom-container">
        <h1 id="article-title-h1">Loading...</h1>
        <address class="blog-details_address">
          <ul>
            <li>By</li>
            <li>Date issued</li>
          </ul>
          <ul>
            <li>Kairat Nurakhmet</li>
            <li><time datetime="2023-05-13" title="May 13 2023">May 13 2023</time></li>
          </ul>
        </address>
      </section>
    </section>
    <section class="blog-details_content">
      <section>
        <p id="article-body">Loading...</p>
      </section>
      <section>
        <h3>The Latest</h3>
        <hr />
        <ul>
          <li>
            <a href="">Startup journey insights for female entrepreneurs</a>
            <span>5 min read</span>
          </li>
          <hr />
          <li>
            <a href="">Creative Marketing Strategies to Boost Your Brand</a>
            <span>5 min read</span>
          </li>
          <hr />
          <li>
            <a href="">The Power of Networking for Business Success</a>
            <span>5 min read</span>
          </li>
        </ul>
      </section>
    </section>
  </article>
  <section class="blog-details_additional-articles-grid" id="additional-articles-container">
    <!-- Dynamic content renders here. -->
  </section>
</main>
```

Figure 10: Content of main tag (blog-details.html)

The **script** tag contains JavaScript that fetches blog post data based on the post's ID from the URL and populates the content within the main tag, including the post's title and body. Additionally, it fetches a list of posts to populate a section (section with id **additional-articles-container**) at the bottom of the page, demonstrating dynamic data retrieval and

DOM manipulation to enhance user interactivity and engagement with the content. The content of Javascript code is demonstrated below in **Figure 11.**

```
<script>
  const queryParams = new URLSearchParams(window.location.search);
  const postId = queryParams.get('id');

  fetch(`https://jsonplaceholder.typicode.com/posts/${postId}`)
    .then((response) => response.json())
    .then((post) => {
      document.getElementById('article-title').textContent = post.title;
      document.getElementById('article-title-h1').textContent = post.title;
      document.getElementById('article-body').innerHTML = `<p>${post.body}</p>`;
    })
    .catch((error) => console.error('Error fetching post:', error));

  fetch('https://jsonplaceholder.typicode.com/posts?_start=0&_limit=6')
    .then((response) => response.json())
    .then((posts) => {
      const container = document.getElementById('additional-articles-container');
      container.innerHTML = '';
      posts.forEach((post) => {
        const article = document.createElement('a');
        article.classList.add('blog-card');
        article.href = `/blog-details.html?id=${post.id}`;
        article.innerHTML = `
        <article">
          <img src="./assets/post-image.png" alt="Post image" />
          <section>
            <h2>${post.title}</h2>
            <p>${post.body}</p>
          </section>
        </article>
      `;
        container.appendChild(article);
      });
    })
    .catch((error) => console.error('Error fetching posts:', error));
</script>
```

Figure 11: Content of script tag (blog-details.html)

4.4.6 Conventional application's code length

The following is a table detailing counts of lines of code for the html files responsible for the pages presented in this approach.

Table 2. Conventional way corresponding files and lines of code of the entire file.

FILE NAME	LINES OF CODE
index.html – Home page	46
blogs-list.html – Blogs list page	54
blog-details.html – Blog details page	122

The following table presents counts of lines of code to make an API request and render an element. The lines of code are counted line by line, not counting empty lines, formatting, and corrections added by code the editor.

Table 3. Conventional way count representing the purpose of each API endpoint in the files for

FILE NAME	PURPOSE	LINES OF CODE
index.html – Home page	Fetch featured blog post with ID 1	7
blogs-list.html – Blogs list page	Fetch 6 additional blog posts	22
blog-details.html – Blog details page	Fetch blog post by ID from the ID query parameter	10
blog-details.html – Blog details page	Fetch 6 additional blog posts	22

4.5 Next.js Way website

4.5.1 File structure

The Next.js file structure as demonstrated below in **Figure 12** consists of more files and folders compared to the conventional way.

Application created using the **npx create-next-app** command automatically created the following files and folders:

- **node_modules:** The node_modules folder is a key constituent of this structure because it contains libraries and dependencies that are managed by npm. It encompasses a wide range of reusable code that is important for modern web

development. At the root URL path of the app, static assets like images are in the public folder.

- **public:** The public folder contains static assets like images, and fonts which are accessible from any files in the application.
- **src:** The source (src) folder is the core where the application's source code resides.
- **src/app:** In this folder, the application contains page-specific JS or JSX files organized, according to route hierarchy. In Next.js routes are made by creating files in this folder, such as the **blogs-list** folder containing the **page.jsx** file by the meaning application has a blogs-list route.
- **.gitignore:** The file contains intentionally untracked files to ignore by Git, such as environmental keys or other configuration files of the project.
- **package.json:** Describes the application's metadata, scripts, and the list of dependencies.
- **package-lock.json:** Automatically created file based on package.json to lock down the versions of a package's dependency files.
- **README.md:** Contains the project description, usually developers describe the project and other information in this file.

Manually created files and folders:

- **components:** The components folder is the key of the React paradigm, containing reusable components inside any file of the application, such as the **Header.jsx**, **PostCard.jsx,** and other specific UI elements.

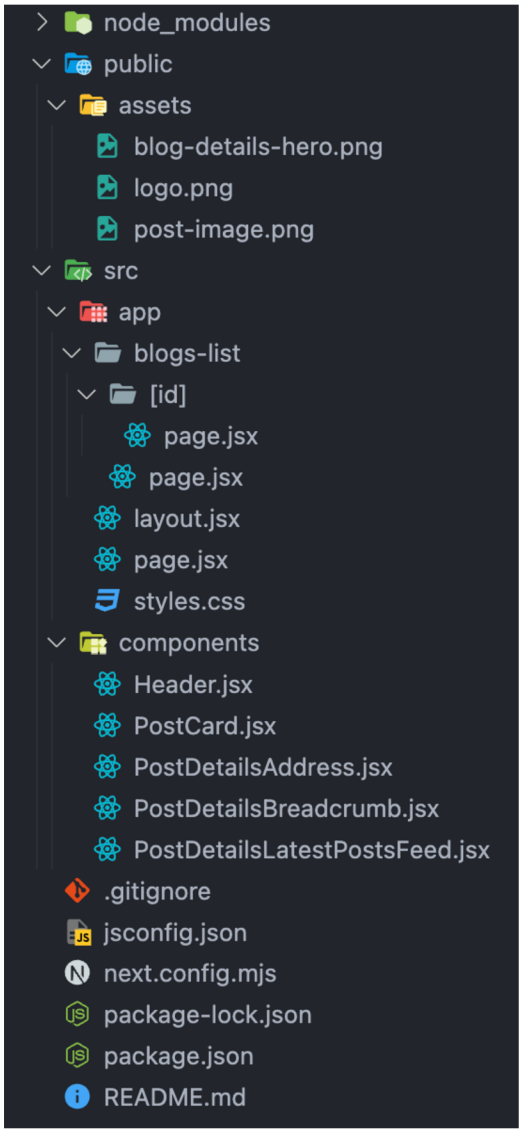


Figure 12: Next.js way file structure

4.5.2 Layout

In the Next.js framework, the **RootLayout** component serves as a wrapper component of the whole website and determines how the application's structure should look on the whole website as demonstrated in **Figure 13** below. The **children** prop inside the component is a special prop that is automatically provided by React. It is used for displaying whatever content is passed between the opening and closing curly brackets.

```jsx
import Header from '@/components/Header';
import './styles.css';

export const metadata = {
  title: 'Blogs website',
  description: 'Description of the website',
};

export default function RootLayout({ children }) {
  return (
    <html lang='en'>
      <body>
        <Header />
        {children}
      </body>
    </html
  );
}
```

Figure 13: RootLayout of the application (layout.jsx)

In **Figure 13** above, line **1** of the code demonstrates the import of the reusable **Header** component. The Header component is placed in line **13** and will be used across the website. React as a library and Next.js as a React framework insert the content of the Header component instead of line **13**. The content of the Header component is shown below in **Figure 14**.

```jsx
import Image from 'next/image';
import Link from 'next/link';

export default function Header() {
  return (
    <header>
      <Link href='/'>
        <Image src='/assets/logo.png' alt='Logo of the website' width={133} height={44} />
      </Link>
      <nav>
        <ul className='nav_pages'>
          <li>
            <Link href='/'>Home</Link>
          </li>
          <li>
            <Link href='/blogs-list'>Blogs</Link>
          </li>
        </ul>
        <ul className='nav_auth'>
          <li>
            <Link href=''>Sign in</Link>
          </li>
          <li>
            <Link href='' className='nav_sign-up'>
              Sign up
            </Link>
          </li>
        </ul>
      </nav>
    </header>
  );
}
```

Figure 14: Content of the Header component

4.5.3 Home page

In the Next.js application, the index of the website is located on the **page.jsx** file which is located within the **app** folder. In this file, the code demonstrated below in **Figure 15** illustrates the use of a built-in **Image** component for optimized image handling and a **Link** component for client-side routing, instead of <img> and <a> tags in HTML.

The page displays a Home page with a featured blog post, dynamically loaded from a placeholder API using the asynchronous function **getPosts**.

In the asynchronous getPost function the code retrieves data by the following URL: **https://jsonplaceholder.typicode.com/posts/1**. Where the number 1 indicates the ID of the featured post.

Inside the body of the Home component **post** variable is assigned to the result of the **getPost** function, to be later used on line **22** and line **23** to render the post title and post body.

```
1  import Image from 'next/image';
2  import Link from 'next/link';
3
4  export const metadata = {
5    title: 'Home',
6    description: 'Description of the website',
7  };
8
9  async function getPost() {
10   const res = await fetch('https://jsonplaceholder.typicode.com/posts/1');
11
12   return res.json();
13 }
14
15 export default async function Home() {
16   const post = await getPost();
17
18   return (
19     <main className='home_main'>
20       <article className='hero'>
21         <section className='hero_text'>
22           <h1>{post.title}</h1>
23           <p>{post.body}</p>
24           <Link href='/blogs-list/article-1' className='btn'>
25             Read more
26           </Link>
27         </section>
28         <section className='hero_img'>
29           <Image src='/assets/post-image.png' alt='Post image mask' fill />
30         </section>
31       </article>
32     </main>
33   );
34 }
```

Figure 15: Home page component (page.jsx file in the app folder)

4.5.4 Blogs list page

In the Next.js application, the **BlogsListPage** component presents a list of blog posts. The page's blog posts content is rendered by the **getPosts** function which fetches data asynchronously from a mock API, specifically from the endpoint **https://jsonplaceholder.typicode.com/posts?_start=0&_limit=6** using query parameters that limit results to six posts as demonstrated in the line **10** in **Figure 16** below. After that, the fetched data is stored in the **posts** variable, line **16**.

The page is structured semantically with a main tag capturing the header and a section for articles. Next.js's client-side routing capability can be seen in the Link component wrapping the PostCard component for each blog post. Here, JavaScript template literals are used within the href attribute to create links dynamically to detailed pages for each blog post using its unique ID.

```
1  import PostCard from '@/components/PostCard';
2  import Link from 'next/link';
3
4  export const metadata = {
5    title: 'Blogs list',
6    description: 'Description of the website',
7  };
8
9  async function getPosts() {
10   const res = await fetch('https://jsonplaceholder.typicode.com/posts?_start=0&_limit=6');
11
12   return res.json();
13 }
14
15 export default async function BlogsListPage() {
16   const posts = await getPosts();
17
18   return (
19     <main className='blogs-list_main'>
20       <h1 className='blogs-list_heading'>Blogs list</h1>
21       <section className='articles-grid'>
22         {posts &&
23           posts.map((post) => (
24             <Link key={post.id} href={`/blogs-list/${post.id}`} className='blog-card'>
25               <PostCard title={post.title} body={post.body} />
26             </Link>
27           ))}
28       </section>
29     </main>
30   );
31 }
```

Figure 16: Blogs list page (page.jsx file in the blogs-list folder)

The **PostCard** component is an example of a reusable component meant to display individual cards of posts. The component receives a **title** and **body** as props to display individual blog post card content as demonstrated in **Figure 17** below.

```jsx
import Image from 'next/image';

export default function PostCard({ title, body }) {
  return (
    <article>
      <Image src='/assets/post-image.png' alt='Post image mask' width={400} height={400} />
      <section>
        <h2>{title}</h2>
        <p>{body}</p>
      </section>
    </article>
  );
}
```

Figure 17: PostCardt.jsx components content

4.5.5 Blog details page

In Next.js, the use of square brackets ([ ]) in a folder or file name, such as [id], means it is a dynamic route. This practice enables the creation of a page that can show different content based on the URL path. The **BlogDetailsPage** function is built to respond to fetching and showing a blog post from an **ID** passed within the route parameters as demonstrated in **Figure 18** below.

There are two asynchronous functions; **getPostById** and **getPosts**, which have separate goals:

1. getPostById**:** retrieves single post details using its id.
2. getPosts**:** fetches a list of other blog posts to display as additional content on the page.

The **BlogDetailsPage** component gets an **ID** from its params object in the props and then uses it to fetch and render an appropriate post.

```jsx
import PostDetailsAdditionalPosts from '@/components/PostDetailsAdditionalPosts';
import PostDetailsAddress from '@/components/PostDetailsAddress';
import PostDetailsBreadcrumb from '@/components/PostDetailsBreadcrumb';
import PostDetailsLatestPostsFeed from '@/components/PostDetailsLatestPostsFeed';
import Image from 'next/image';

export const metadata = {
  title: 'Blog details',
  description: 'Description of the website',
};

async function getPostById(id) {
  const res = await fetch('https://jsonplaceholder.typicode.com/posts/' + id);
  return res.json();
}

async function getPosts() {
  const res = await fetch('https://jsonplaceholder.typicode.com/posts?_start=0&_limit=6');
  return res.json();
}

export default async function BlogDetailsPage({ params: { id } }) {
  const post = await getPostById(id);
  const additionalPosts = await getPosts();

  return (
    <main className='blog-details_main'>
      <article className='blog-details'>
        <section className='blog-details_hero'>
          <PostDetailsBreadcrumb title={post.title} />
          <Image src='/assets/blog-details-hero.png' alt='Post image mask' width={1200} height={400} />
          <section className='blog-details_hero-bottom-container'>
            <h1>{post.title}</h1>
            <PostDetailsAddress />
          </section>
        </section>
        <section className='blog-details_content'>
          <section>
            <p>{post.body}</p>
          </section>
          <PostDetailsLatestPostsFeed />
        </section>
      </article>
      <PostDetailsAdditionalPosts additionalPosts={additionalPosts} />
    </main>
  );
}
```

Figure 18: page.jsx file in the [id] folder indicating the Blog details page

The page is made up of various reusable components including PostCard to be used in the related additional posts section, and PostDetailsAdditionalPosts for the section rendering additional blog posts after the main content, the code is demonstrated below in **Figure 19.**

```jsx
import Link from 'next/link';
import React from 'react';
import PostCard from './PostCard';

export default function PostDetailsAdditionalPosts({ additionalPosts }) {
  return (
    <section className='blog-details_additional-articles-grid'>
      {additionalPosts &&
        additionalPosts.map((post) => (
          <Link key={post.id} href={'/blogs-list/' + post.id} className='blog-card'>
            <PostCard title={post.title} body={post.body} />
          </Link>
        ))}
    </section>
  );
}
```

Figure 19: PostDetailsAdditionalPosts.jsx file content

PostDetailsAddress for authorship and date information as demonstrated in **Figure 20** below.

```jsx
export default function PostDetailsAddress() {
  return (
    <address className='blog-details_address'>
      <ul>
        <li>By</li>
        <li>Date issued</li>
      </ul>
      <ul>
        <li>Kairat Nurakhmet</li>
        <li>
          <time dateTime='2023-05-13' title='May 13 2023'>
            May 13 2023
          </time>
        </li>
      </ul>
    </address>
  );
}
```

Figure 20: PostDetailsAddress.jsx component

The PostDetailsBreadcrumb for navigation assistance that reflects modular design ideologies in web development, to help users identify on which page they are located as illustrated in **Figure 21** below.

```jsx
import Link from 'next/link';

export default function PostDetailsBreadcrumb({ title }) {
  return (
    <section className='blog-details_breadcrumb'>
      <Link href='/'>Home</Link>
      <span>&#x276F;</span>
      <Link href='/blogs-list'>Blogs list</Link>
      <span>&#x276F;</span>
      <span>{title}</span>
    </section>
  );
}
```

Figure 21: PostDetailsBreadcrumb.jsx component

4.5.6 Next.js application's code length

The following is a table detailing lines of code for the page files presented in this approach.

Table 4. Next.js approach corresponding files and lines of code of the entire file.

FILE NAME	LINES OF CODE
src/app/page.jsx - Home page	29
src/blogs-list/page.jsx - Blogs list page	26
src/blogs-list/[id]/page.jsx – Blog details page	50

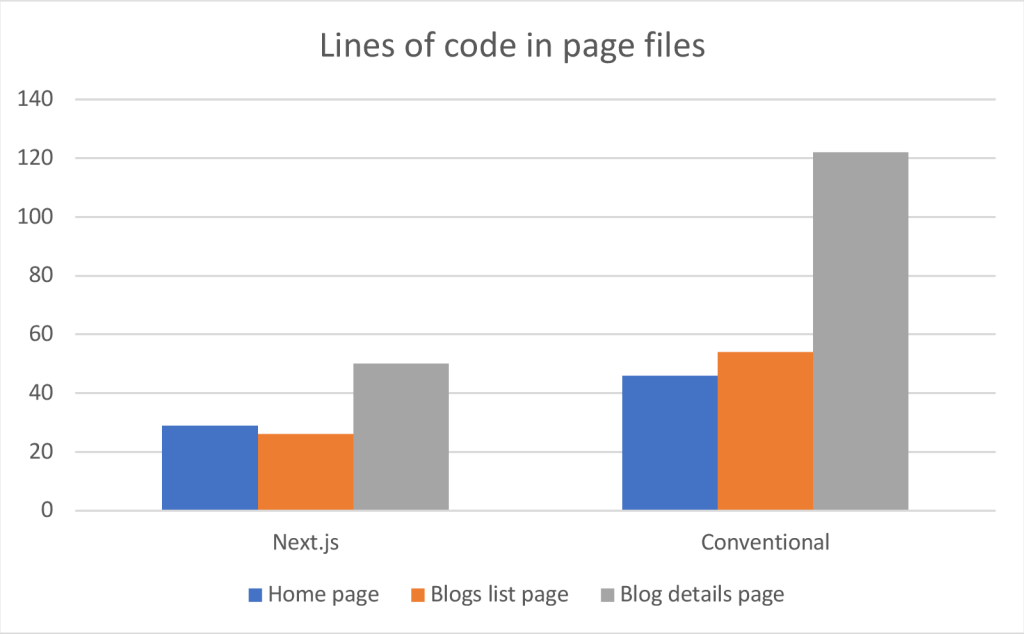
The following table displays the line count required for making an API call and assigning the retrieved data to corresponding elements. In this count, empty lines and lines added or formatted by the code editor are excluded, focusing solely on the actual code content.

Table 5. Next.js approach count represents the purpose of each API endpoint in the files.

FILE NAME	PURPOSE	LINES OF CODE
src/app/page.jsx - Home page	Fetch featured blog post with ID 1	7
src/blogs-list/page.jsx - Blogs list page	Fetch 6 additional blog posts	11
src/blogs-list/[id]/page.jsx – Blog details page	Fetch blog post by ID from the ID query parameter	10
src/blogs-list/[id]/page.jsx – Blog details page	Fetch 6 additional blog posts	14

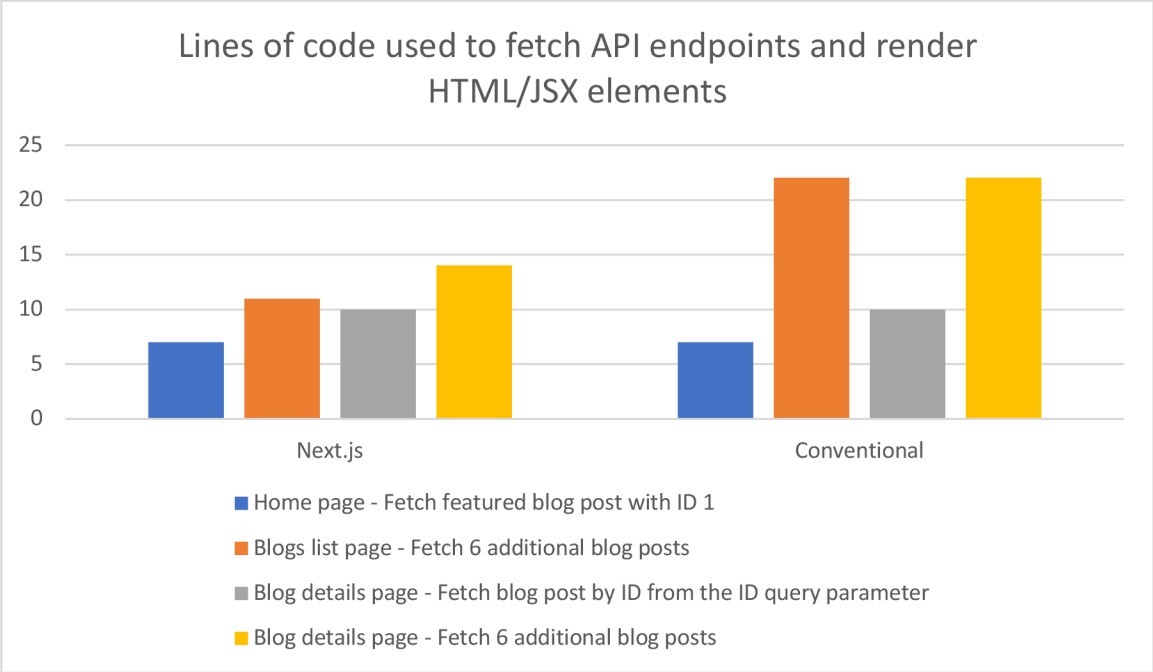
4.6 Lines of code

After putting the results of counting lines of code demonstrated in **Graph 1** below, it has been discovered, that the Next.js way application had a total of 105 lines of code and the Conventional way application had a total of 222 lines of code for the files representing pages.



Graph 1. Result of counting code lines.

The charts demonstrated below in **Graph 2** show the result of counting lines of codes that were used to fetch data using the API endpoint and then render elements with fetched data, that the Next.js application had a total of 42 lines of codes, and the conventional way application had a total of 61 lines of code.



Graph 2. API and render code lines

4.7 Experiment Structure

The practical part of the thesis comprised experiments that were done in five different ways.

1. Changes were made to the source code of the Next.js experimental application so that it will satisfy the test requirements for the chosen property.
2. The following **npm run build** command was used to compile new source code.
3. The built application is started in a private window on the browser.
4. Google Chrome web browser's Lighthouse tool was utilized by selecting a Mobile device to analyze the built application to collect performance data.
5. Each test started with a hard refresh of the page by pressing Ctrl + Shift + R from the keyboard on Windows or Cmd + Shift + R on MacOS devices.
6. The data was stored, and an average value was derived from it.

4.7.1 List of Chosen Metrics

1. **Speed Index:** A page load performance metric that evaluates the speed at which a page's content becomes visible, measured in seconds (s).

2. **Largest Contentful Paint:** Total Blocking Time is calculated by summing up the blocking durations of all significant tasks that occur between First Contentful Paint (FCP) and Time to Interactive (TTI), with the result expressed in milliseconds (ms).

4.7.2 Image Optimization

The conventional way of building an application does not support a built-in image optimizer, usually, developers use third-party services to optimize and maintain images in the application.

Next.js provides a built-in **Image** component, that optimizes image files based on the device size and resolution, but a default **img** tag also could be used. Next.js also provides lazy loading for all images that were used by the Image component, which loads images only when they come to the viewport of the browser. In this part will be used images from the Unsplash[8] free images source, the Next.js built-in Image component, and the default img tag.

To test the Next.js application the following **npm run build** command will be used to create a build file. Running this command will generate an optimized version of the application ready to be sent to production. Running the following **npm start** command in the terminal of the project will start the local server to test our application, by default server starts on this path – http://localhost:3000.

A single image[9] weighing 4.6 MB and the original size of 6496x4331 was used for all tests.

5 Results and Discussion

5.1 Analysis of two approaches

The conventional and Next.js approaches to developing the websites have been compared using identical style.css files and identical tags, except for Next.js' built-in Link and Image components to demonstrate the benefits of the Next.js approach. The results are significant, as they provide a specific comparison of the two approaches in the same scenario

[8] Unsplash. Available from: https://unsplash.com/
[9] That Prague bridge. Author: Anthony DELANOIX. Available from: https://unsplash.com/photos/people-walking-on-bridge-aDxmYZtYj7g

and can help with the decision, on which approach a developer should rely on when starting a project. The results also demonstrate the possible benefits of switching from one approach to another on an existing project.

5.2 Image optimization

The first test shows that the image used with the img tag has an unchanged weight of 4.59 mb and the image width is unchanged at the original 6496px.

The second test shows that the image used with the Image component has a modified compressed weight of 360.59 kb, and the image resolution has changed to the screen width of the device being used and is 1920px.

In analysing the data collected, it can be observed that a notable difference in image optimization performance can be achieved at a 92.34% improvement as shown in **Table of Results 1**.

This suggests that if the Image component tag is used, it will result in a noticeable increase in web page loading performance for the end user. This can be explained by the fact that the Image component is specifically developed to optimize image weight and resolution. Although, given that the tests were conducted on a mobile device, it can be assumed that users on desktop devices will experience a difference as desktop devices have different internet speeds than mobile devices.

Table 6. Table of Results 1.

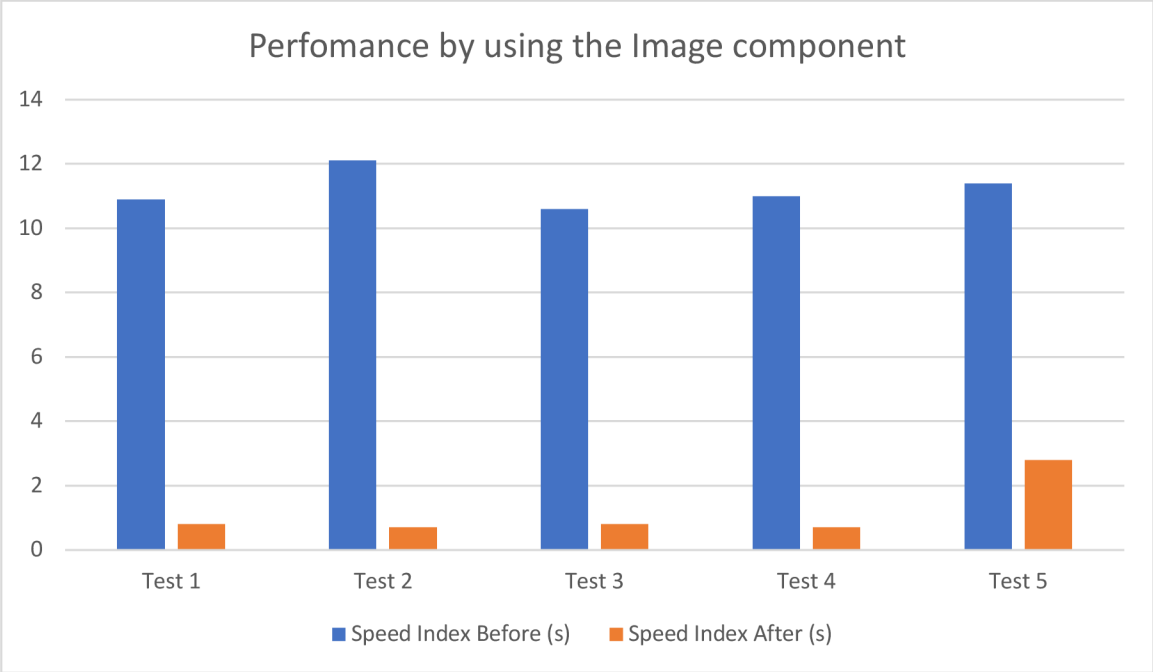
TAG	SIZE	WIDTH
img	4.6 MB	6496px(original)
Image	360.59 KB	1920px

5.3 Page speed

In analysing the collected data, it can be concluded that a significant improvement in page speed performance, was achieved at a 174.58% or at a 10.44-second improvement, as demonstrated in **Table of Results 2** below. The observed increase in page speed can be attributed to the use of the Image component in Next.js, which pre-optimizes locally stored images. This component implements optimization, increasing page speed by performing resource-intensive tasks ahead of time.

Table 7. Table of Results 2.

	Speed Index Before (s)	**Speed Index After (s)**
Test 1	10.9	0.8
Test 2	12.1	0.7
Test 3	10.6	0.8
Test 4	11.0	0.8
Test 5	11.4	0.7
Average	11.2	0.76



Graph 3. Effects of Image Component. Speed Index in Seconds.

5.4 Code length

A comparison has been made between Conventional and Next.js applications concerning the length of the code. All the files responsible for pages in the application in Conventional and Next.js folders are considered, and their lines of code are counted. As **Graph 1** and **Graph 2** demonstrate above, the number of lines in the Next.js application has decreased by over 52.7%.

5.5 Codebase

Developing an application using Next.js, it was discovered that the Component-Based Approach (CBA) in the application helps to reduce code length and allows to use of the same code in different parts of the application. It was also discovered that the nesting of components into each other can be infinite, which allows developers to create a flexible and convenient environment for developing and debugging the application.

6 Conclusion

In conclusion, the theoretical part covered the unique features and functional gains that are realized when Next.js is used for web application development as compared to traditional ways of developing websites. The framework built in the opening sections established a strong foundation for appreciating the basic technologies behind contemporary web design, HTML, CSS, and Javascript as well as the advanced features of Next.js.

Through creating two different applications with one relying on conventional methods while another using Next.js, it was evident that Next.js was efficient. This study focused mainly on project structure, API interaction, and routing and demonstrated how it made a difference by improving code organization, maintainability, and performance optimization.

Based on empirical evidence drawn from comparative analysis, it was evident that Next.js exhibited remarkable performance improvements and developer experience leapfrogging. This also validated its benefits in practice hence advocating for a radical shift towards more holistic development approaches as well as deepening insights into its applicability within real-life settings.

Consequently, this research paves the way for future studies on the expansive potential of Next.js in web development. Exploration should be done into how it connects to new internet concepts and can be applied at scale to big projects or across diverse domains of applications thereby expanding debates around methodologies employed during web development so that communities can develop better ways of creating user-friendly systems with less effort involved.

7 References

1. NPM documentation. Accessed 8 November 2023. Available from: https://docs.npmjs.com/about-npm
2. What is the DOM? Accessed 20 November 2023. Online Source. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction#what_is_the_dom
3. Writing Markup with JSX. React Dev Docs [online]. [Accessed 15 November 2023]. Available from: https://react.dev/learn/writing-markup-with-jsx
4. Server-side Rendering vs Static Site Generation in Next.js. Accessed 21 November 2023. Online Source. Available from: https://nextjs.org/docs/pages/building-your-application/rendering/static-site-generation
5. Server-side Rendering vs Static Site Generation in Next.js. Accessed 21 November 2023. Online Source. Available from: https://nextjs.org/docs/pages/building-your-application/rendering/static-site-generation
6. VS Code Counter. Accessible from: https://github.com/uctakeoff/vscode-counter
7. Prettier – Code formatter. Accessible from: https://github.com/prettier/prettier-vscode
8. Unsplash. Available from: https://unsplash.com/
9. That Prague bridge, Anthony DELANOIX. Unsplash [online]. [Accessed 5 January 2024]. Available from: https://unsplash.com/photos/people-walking-on-bridge-aDxmYZtYj7g
10. 8 Best Practices for React.js Component Design. Dev.to [online]. [Accessed 20 December 2023]. Available from: https://dev.to/blossom/8-best-practices-for-reactjs-component-design-4jn5
11. CSS Imports. MDN Web Docs [online]. [Accessed 25 October 2023]. Available from: https://developer.mozilla.org/en-US/docs/Web/CSS/@import
12. DUCKETT, Jon. HTML & CSS: Design and Build Websites. Wiley, 2011. ISBN 1118008189
13. Introduction to Programming in JavaScript. Launchschool [online]. [Accessed 25 October 2023]. Available from
14. Web Development Framework. TechTarget [online]. [Accessed 12 October 2023]. Available from: https://www.techtarget.com/searchcontentmanagement/definition/web-development-framework-WDF
15. Document Object Model. MDN Web Docs [online]. [Accessed 12 November 2023]. Available from: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Mode
16. How to Assure a Well-Formed Website. InformIT [online]. [Accessed 27 September 2023]. Available from: https://www.informit.com/articles/article.aspx?p=1193471
17. Working with JSON. MDN Web Docs [online]. [Accessed 27 September 2022]. Available from: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON Schema.org Markup. MDN Web Docs [online]. [Accessed 27 September 2022]. Available from: https://moz.com/learn/seo/schema-structured-data#:~:text=Schema.org%20(often%20called%20schema,represent%20your%20page%20in%20SERPs.

18. De, Brajesh: API Management An Architect's Guide to Developing and Managing APIs for Your Organization, Berkeley, CA : Apress, 2017, ISBN: 978-1-4842-1305-6, 9781484213063
19. Elrom, Elad: React and libraries your complete guide to the React ecosystem, New York: Apress, 2021, ISBN: 978-1-4842-6696-0
20. FLANAGAN, David. *JavaScript : the definitive guide*. Sebastopol, CA: O'Reilly, 2002. ISBN 0-596-00048-0.
21. Griffiths, David: React cookbook: recipes for mastering the React framework, Beijing ; Boston ; Farnham ; Sebastopol ; Tokyo : O'Reilly, 2021, ISBN: 978-1-492-08584-3
22. Horton, Adam: Mastering React, master the art of building modern web applications using React, Birmingham, Mumbai: Packt publishing, 2016, ISBN: 978-1-78355-856-8.
23. Konshin, Kirill: Next.js quick start guide, Packt Publishing, 2018, ISBN: [1-78899-366-7; 1-78899-584-8]
24. Tyson, Matthew: The best new features in Next.js 13, InfoWorld.com; San Mateo: 2022, 2737142348.
25. Google LightHouse. Google Developers [online]. [Accessed 10 January 2024]. Available from: https://developer.chrome.com/docs/lighthouse/overview/
26. About PageSpeed Insights [online]. Google Developers [Accessed 10 January 2024]. Available from: https://developers.google.com/speed/docs/insights/v5/about
27. React – A JavaScript Library for building user interfaces [online]. [Accessed 10 January 2023]. Available from: https://reactjs.org/
28. Visual Studio Code. [online]. [Accessed 10 January 2024]. Available from: https://code.visualstudio.com
29. How to use the Next.js Image component Effectively [online]. [Accessed 15 January 2024]. Available from: https://www.zachgollwitzer.com/posts/nextjs-image-component-tutorial
30. Next.js. Next.js Documentation [online]. [Accessed 14 February 2024]. Available from: https://nextjs.org/docs
31. How to Create a Full-Stack Application with Next.js – A Step-By-Step Tutorial for Beginners. [online]. [Accessed 21 July 2023]. Available from: https://www.freecodecamp.org/news/build-a-full-stack-application-with-nextjs/
32. Next.js tutorial with examples: Build better React apps with Next. [online]. [Accessed 14 September 2023]. Available from: https://www.educative.io/blog/nextjs-tutorial-examples
33. Client-Side Rendering vs. Server-Side Rendering. [online]. [Accessed 10 November 2023]. Available from: https://www.educative.io/courses/next-js-build-react-apps/client-side-rendering-vs-server-side-rendering
34. Incremental Static Regeneration with Next.js. [online]. [Accessed 11 November 2023]. Available from: https://blog.logrocket.com/incremental-static-regeneration-next-js/
35. Simple Guide to Client-Side Rendering (CSR) in React.js. [online]. [Accessed 1 March 2024]. Available from: https://javascript.plainenglish.io/simple-guide-to-client-side-rendering-csr-in-react-js-9eb019cb6d73
36. How To Optimize Images for Web and Performance. [online]. [Accessed 21 January 2024]. Available from: https://kinsta.com/blog/optimize-images-for-web/

List of pictures, tables, graphs, and abbreviations

7.1 List of pictures

7.2 List of tables

7.3 List of graphs

7.4 List of abbreviations

1. HTML: Hyper Text Markup Language
2. XML: Extensible Markup Language
3. JSX: Javascript XML
4. JS: Javascript
5. NPM: Node Package Manager
6. UI: User Interface
7. CSS: Cascading Style Sheets
8. DOM: Document Object Model
9. CBA: Component-Based Architecture
10. CSR: Client-Side Rendering
11. SSG: Static-Site Generation
12. SSR: Server-Side Rendering
13. ISR: Incremental Static Regeneration
14. AST: Abstract Syntax Tree
15. API: Application Program Interface
16. SPA: Single Page Application
17. LOC: Lines of Code
18. FCP: First Contentful Paint
19. TTI: Time to Interactive

7.5 List of Source Code Snippets