Universitatea de Vest din Timişoara

Facultatea de Matematică şi Informatică

Domeniul de Studii:
Informatică Aplicată

Lucrare de Disertaţie

COORDONATOR:

Conf. Dr. Cristina Mîndruţă

ABSOLVENT:

Nicolae Savilencu

Timişoara
2023

Universitatea de Vest din Timișoara

Facultatea de Matematică și Informatică

Tehnologii de randare a informaţiilor în aplicaţiile web moderne

COORDONATOR:

Conf. Dr. Cristina Mîndruţă

ABSOLVENT:

Nicolae Savilencu

Timișoara

2023

Rezumat

The web is evolving with high pace, and with the emergence of new web technologies, the way web applications are build has changed significantly, compared to the structures used in the past. Rendering is a critical aspect of web development, and modern web applications require faster, more efficient rendering techniques to deliver a better user experience. This dissertation aims to explore the rendering technologies used in modern web applications in order to define an optimal, yet flexible solution for developing applications of different scales.

The research begins by introducing the fundamental concepts of web rendering, including two of the most known approaches: server-side rendering and client-side rendering. It then examines the evolution of rendering technologies, from traditional rendering techniques with HTML, CSS, and JavaScript, to modern technologies like React, Gatsby, Next. The dissertation compares and contrasts these technologies and analyzes their performance and usability by some predefined metrics: LCP (*Largest Contentful Paint*), FID (*First Input Delay*), CLS (*Cumulative Layout Shift*), and other important factors such as development time and scalling ability.

Dissertation explores the impact of rendering technologies on development workflows, examines the challenges associated with adopting new technologies, such as learning curve, compatibility issues, performance bottlenecks, and SEO optimization.

Finally, the dissertation provides a case study of a web application build using multimple technologies, and comparing the performance metrics, individual technologies advantages and disadvantages.

Overall, the dissertation provides a comprehensive analysis of rendering technologies in modern web applications and their impact on web development, to indentify most optimal methods of creating performant, scallable, and user friendly applications.

Cuprins

Listă de figuri

Capitolul 1

Introducere

1.1 Motivație și scop

Obiectivul acestei lucrări este de a analiza tehnologiile de randare utilizate în aplicațiile web moderne, pentru a identifica metode de dezvoltare optime, din punct de vedere al performanței, experienței utilizatorului și SEO. Aceste rezultate vor fi obținute prin compararea mai multor framework-uri front-end după un anumit set de criterii, prestabilit. În urma comparării acestora, prin intermediu dezvoltării unei aplicații, se vor identifica cele mai eficiente șabloane de dezvoltare.

Motivația alegerii acestei teme de disertație derivă din mai mulți factori. În primul rând, progresul tehnologic rapid a dus la apariția mai multor paradigme de dezvoltare a aplicațiilor web, iar ținerea pasului cu acestea a devenit o provocare. În al doilea rând, odată cu crearea unor aplicații din ce în ce mai complexe, necesitatea de a cunoaște procesele interne precum tehnologiile și metodele de randare a devenit o componentă esențială a dezvoltării. Prin urmare, înțelegerea modului în care funcționează aceste tehnologii și a modului în care acestea pot fi optimizate poate ajuta la dezvoltarea unei aplicații care ar oferi o experiență de utilizare și o performanță sporită.

În cele din urmă, această lucrare de disertație își propune să ofere informații despe tehnologiile de randare utilizate în aplicațiile web moderne și despre impactul acestora asupra performanței, pentru a identifica cele mai eficiente framework-uri pentru a dezvolta o aplicație web ce poate fi scalată, bine optimizată și cu un user experience de nivel înalt.

1.2 Contextul lucrării

World Wide Web nu este un sinonim al internetului, dar este cea mai proeminentă parte a acestuia, care poate fi definită ca un sistem tehno-social cu care interacționează oamenii prin intermediul rețelelor tehnologice. Noțiunea de sistem tehno-social se referă la un sistem care îmbunătățește percepția umană, comunicarea și cooperarea. Percepția este condiția prealabilă necesară pentru a comunica și coopera.

Pe 12 Martie 1989, Tim Berners-Lee, informatician de origine britanică și un fost angajat CERN au scris o propunere pentru ceea ce va deveni ulterior World Wide Web. Acea propunere avea drept scop crearea unui sistem de comunicații mai eficient în cadrul CERN, însă Berners-Lee a realizat în cele din urmă că conceptul ar putea fi implementat la scară globală. Împreună cu informaticianul belgian Robert Cailliau au propus în 1990 să folosească hypertext pentru a lega și accesa diverse tipuri de informații dintr-o rețea de noduri în care utilizatorul poate naviga către rezultatele dorite.

Prima versiune a web-ului global de la CERN a fost construită pe primele versiuni ale HTTP (HyperText Transfer Protocol) şi HTML (HyperText Markup Language). Aceste pagini web au fost servite de primul server web. Berners-Lee a scris, de asemenea, primul browser web pentru a accesa acest nou web creat. Web-ul global s-a schimbat drastic de la concepţia sa originală din 1989.

1.2.1 Web 1.0

A fost creat în 1989 și utilizat până în 2005. Potrivit lui Tim Berners-Lee Web 1.0 a fost *read-only*, deoarece oferea foarte puțină interacțiune pentru utilizatori. Rolul web-ului a fost de natură pasaivă.

Web 1.0 a fost prima generație de World Wide Web și conținea doar pagini statice ce aveau doar un singur scop - de livrare a conținutului. Deoarece era monodirectional, însemna că organizaţiile împărtăşeau informaţii precum broşuri, cataloage doar pentru citire. Aceste date erau prezentate pe pagini HTML statice care se modificau manual. Utilizatorii nu puteau contribui la paginile web existente. Tehnologiile Web 1.0 includ protocoale web de bază, HTML, HTTP și URI. [1]

Browserele erau foarte rudimentare şi site-urile web erau mici. De aceea era destul să se creeze un fişier HTML static care ar conţine toate datele necesare şi designul, care nu ar fi fost niciodată actualizate automat.

```
!DOCTYPE html>
<html>
   <head>
      <title>Page Title</title>
   </head>
   <body>
      <h1>Heading example</h1>
      <p>My first paragraph.</p>
   </body>
</html>
```

Listing 1.1: Exemplu fişier HTML

Mai târziu, când browser-ele au dezvoltat capacităţi vizuale, a aparut necesitatea de stilizare. Håkon W Lie a propus o prima versiune a CSS (Cascading Style Sheet), care este folosit şi în prezent. Acesta este un limbaj de stilizare folosit pentru a descrie modul în care un document HTML ar trebui să fie prezentat. CSS a fost creat pentru a permite separarea conținutului și a prezentării. Acest lucru a permis paginilor web să fie mai accesibile, mai ușor de întreținut și mai flexibile.

```
body {
   background-color: black;
   margin: 10px 5px;
}

p {
   font-size: 12px;
   color: red;
}
```

Listing 1.2: Exemplu fişier CSS

Caracteristici:

- Conținut read-only
- Informațiile erau la dispoziția oricui și oricând
- Include pagini web statice și utilizează HTML

Limitări:

- Paginile puteau fi înțelese doar de oameni și nu aveau conținut compatibil pentru motoare de căutare
- Utilizatorii erau responsabili pentru responsabili pentru actualizarea și gestionarea conținutului
- Incapabilitatea de a reprezenta informații dinamice

1.2.2 Web 2.0

Tim O'Reilly a definit Web 2.0 drept fiind revoluţia afacerilor în industria calculatorului, cauzată de mutarea către internet ca platformă şi încercarea de a înţelege regulile pentru succes pe această nouă platformă. Printre aceste reguli se numără următoarea: construirea de aplicaţii care să valorifice efectele de reţea pentru a deveni mai bune pe măsură ce mai mulţi oameni le folosesc. [2]

Web 2.0 facilitează proprietăți majore cum ar fi practicile participative, colaborative și distribuite, care permit desfășurarea activităților zilnice formale și în sfere formale pe web. Web 2.0 este web-as-a-platform. Utilizatorii au mai multe unelte pentru interacțiune, însă cu mai puțin control. Versiunea 2.0 implică și un design web flexibil, reutilizare creativă, actualizări și creare de conținut colaborativ, care este considerată una dintre cele mai remarcabile caracteristici ale web 2.0.

Împreună cu primele site-uri web dinamice a venit nevoia pentru primele tehnologii de programare pe partea de server. La început, programarea pe partea de server se făcea direct în serverul web. Mai târziu, standardul CGI (Common Gateway Interface) a fost dezvoltat, ceea ce a făcut posibilă interacţiunea serverului web cu orice proces local. Mai târziu, limbi precum Perl, Java, PHP şi ASP (şi altele) au devenit populare pentru programare pe partea de server.

Site-urile web care devin mai bogate în funcţii au nevoie de mai multă flexibilitate şi de o utilizare mai bună. Acesta este momentul în care programarea client-side intră în joc. Fără a fi nevoie de o încărcare completă a paginii, conţinutul poate fi schimbat de către utilizatorii finali. Tehnologia cel mai utilizată pentru acest scop este JavaScript. Când funcţiile client-side au devenit şi mai importante, au fost concepute framework-uri pentru randarea (şi controlarea) paginilor client-side. Cele mai cunoscute framework-uri sunt: Ember, Angular şi React.

Caracteristici:

- Web-ul a devenit o platformă cu software peste nivelul unui singur dispozitiv
- Trecerea la internet-as-a-platform și încercarea de a înțelege regulile succesului în această nouă platformă
- Social Web este adesea folosit pentru a caracteriza site-urile care constau din comunități. Este bazat pe content management și identificarea unor noi metode de comunicare și interacționare dintre utilizatori. Aplicația web facilitează producerea colectivă de cunoștințe, rețele sociale și crește rata de schimb de informații dintre utilizatori

Limitări:

- Ciclu constant de iterații a schimbarilor și actualizărilor serviciilor
- Probleme etice privind construirea și utilizarea web 2.0
- Interconectivitatea și schimbul de cunoștințe între platforme dincolo de granițele comunității sunt încă limitate

1.2.3 Web 3.0

Cunoscut și sub numele de web semantic sau Web decentralizat, este următoarea generație a World Wide Web, care își propune să ofere o experiență mai inteligentă, interconectată și decentralizată. Reprezintă o viziune pentru viitorul web în care datele, aplicațiile și serviciile sunt interconectate în mod fluid, permițând mașinilor și oamenilor să înțeleagă și să interacționeze mai eficient cu informațiile.

Ideea ce se află la bază este de a defini structurile de date și de a le lega pentru o automatizare, intregare și reutilizare mai eficientă în diverse aplicații. Web 3.0 este capabil să îmbunătățească gestionarea datelor, să susțină accesibilitatea internetului mobil, să stimuleze creativitatea și inovația, să încurajeze fenomenul de globalizare, să sporească experiența utilizatorilor și să ajute la organizarea colaborării în rețelele sociale. [3]

În 3.0 conceptul de site web și pagină web dispare, datele nu sunt deținute, ci partajate, iar serviciile afișează diferite vederi, pe baza datelor. Aceste servicii pot fi aplicații și trebuie să se concentreze pe context și personalizare, ce vor fi atinse prin utilizarea

căutarilor verticale. Web 3.0 facilitează dezvoltarea de aplicații care utilizează inteligența artificială, învățarea automată și procesarea limbajului natural. Aceste tehnologii permit analiza avansată a datelor, automatizarea și capacitatea de luare a deciziilor, rezultând în servicii mai inteligente și personalizate pentru utilizatori.

Următorul pas în dezvoltarea web-ului este crearea aplicaţiilor web autonome, care pot fi, de asemenea, utilizate fără conexiune la internet. Acestea sunt numite Aplicaţii Web Progressive (PWA). Crearea acestor aplicaţii nu este posibilă fără a avea control complet la nivelul clientului. Randarea pe partea de server nu este posibilă şi se folosesc framework-uri complete pentru front-end pentru a construi acest fel de aplicaţii. Comunicarea se gestionează prin intermediul API-urilor.[4]

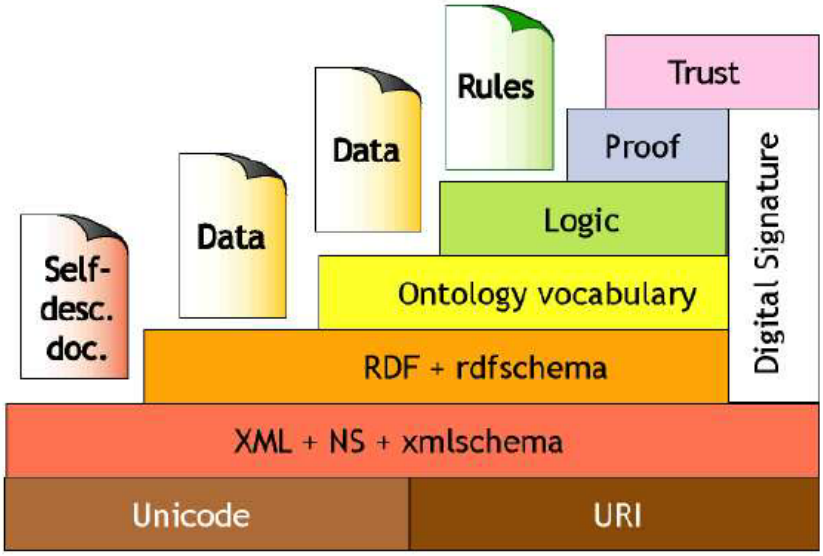

Figura 1.1: Arhitectura stratificată pentru semantic web [1]

Caracteristici:

- SaaS business model
- Platformă software open source
- Personalizarea aplicațiilor pentru utilizatori
- Pooling de resurse
- Web inteligent

[1] https://benchpartner.com/explain-semantic-web-layer-approach-stack-tower-with-example

Provocări:

- *Vastitate* - conține miliarde de pagini, ce duce la redundanța datelor
- *Vaguitatea* - interogările inprecise de către utilizatori, precum și modul în care furnizorii de conținut reprezintă conceptele pot face dificilă potrivirea termenilor de interogare cu termenii relevanți ai furnizorului. Acest lucru poate fi și mai agravat atunci când se încearcă integrarea mai multor baze de cunoștințe, care pot deține concepte similare, dar nu identice, ce poate rezulta în ambiguitate și incertitudine
- *Incoerența* - contradicții logice ce vor apărea inevitabil
- *Înșelăciune* - momentul în care sursa de informații induce în eroare intenționat consumatorul de informații

1.3 Evoluţia tehnologiilor

Pe măsură ce dezvoltarea aplicațiilor evolua, la fel au evoluat și tehnologiile utilizate în crearea aplicațiilor. De-a lungul anilor au apărut multe tehnologii care au revoluționat modul în care sunt dezvoltate aplicațiile. Pe măsură ce acestea deveneau mai complexe, necesitatea în tehnologii mai performante a devenit evidentă. La mijlocul anilor 2000 a fost introdus AJAX (*Asynchronus JavaScript and XML*), care a permis aplicațiilor web să comunice cu serverele, fără a reîncărca întreaga pagină. Acest lucru a rezultat în aplicații web mai rapide și receptive. [5]

În ultimii ani, au apărut mai multe framework-uri front-end, care au facilitat crearea aplicațiilor web cu o complexitate sporită. React, Vue, Next, Angular au devenit extrem de populare datorită ușurinței de utilizare și capacității lor de a gestiona cantități mari de date. Aceste framework-uri au făcut posibilă crearea unor aplicații cu un user experience și funcționalități ce nu au fost posibile anterior.

Pe lângă framework-urile front-end, au evoluat și tehnologiile back-end, în mod semnificativ. PHP, Ruby on Rails, Node.js au facilitat crearea aplicațiilor web moderne scalabile, care pot gestiona cantități mari de trafic în timp real și sarcini complexe de procesare a datelor.

Evoluția tehnologiilor web a fost determinată de necesitatea de a crea aplicații mai complexe, cu funcționalități avansate. De la HTML "curat" la framework-uri pentru

front-end și back-end, tehnologiile web continuă să evolueze într-un ritm rapid, fapt ce impune dezvoltatorii să fie la curent cu ultimele tehnologii și constant să-și actualizeze metodele de dezvoltare utilizate, pentru a deveni mai efectivi în dezvoltarea aplicațiilor moderne, care răspund nevoilor sporite ale utilizatorilor.

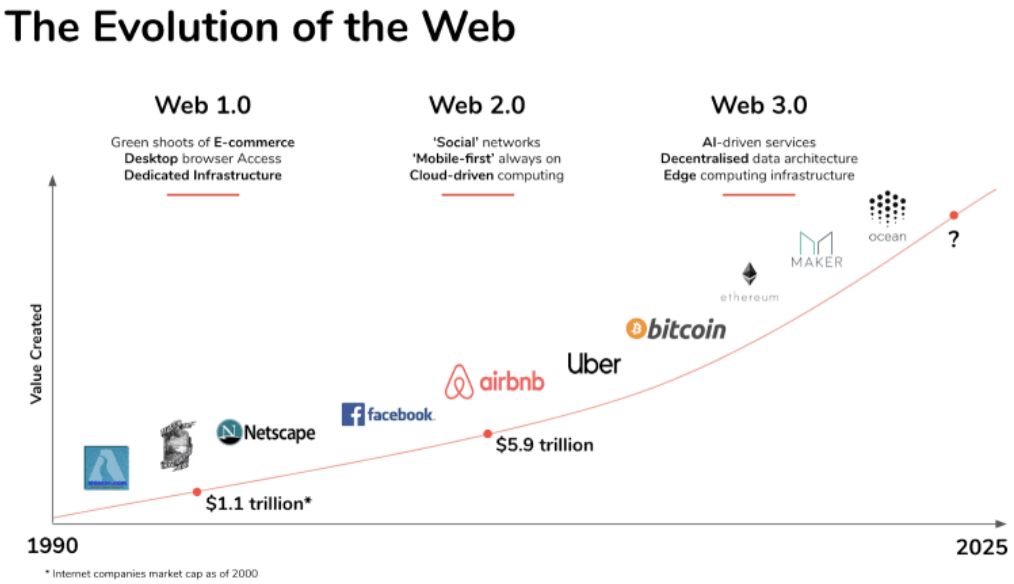

Figura 1.2: Evoluţia web-ului [2]

Evolutia web a avut un impact semnificativ asupra dezvoltării aplicațiilor web moderne. Creșterea vitezei de conectare la internet și a performanței dispozitivelor a permis dezvoltarea aplicațiilor web complexe, cu funcții avansate și user experience sporit. Utilizarea extinsa a standardelor web deschise, cum ar fi HTML, CSS și JavaScript, a dus la o mai mare interoperabilitate între diferite platforme și la o mai mare eficiență în dezvoltarea aplicațiilor.

Apariția și popularizarea API-urilor și a arhitecturilor RESTful au făcut posibilă integrarea mai ușoară a aplicatiilor web cu alte aplicații și servicii. Cloud computing-ul și serviciile web, cum ar fi Amazon Web Services și Microsoft Azure, au făcut posibilă scalabilitatea și distribuirea aplicațiilor web cu usurință, indiferent de locația utilizatorilor.

Creșterea numarului de utilizatori și a numarului de dispozitive conectate a dus la o creștere semnificativă a cerințelor de securitate și a necesității de a dezvolta aplicații web sigure.

[2] `https://dev.to/pragativerma18/evolution-of-web-42eh`

Dezvoltarea și popularizarea framework-urilor și bibliotecilor de cod, cum ar fi Angular, React și Vue, a simplificat dezvoltarea aplicațiilor web și a îmbunatățit productivitatea dezvoltatorilor.

Aplicațiile web moderne sunt mult mai complexe și pot include o varietate de componente, cum ar fi front-end, back-end, baze de date, API-uri și multe altele. În plus, aplicațiile web trebuie să functioneze pe o varietate de platforme și dispozitive, ceea ce adaugă o altă dimensiune a complexității.

Pe masură ce aplicațiile web devin mai complexe, devin mai dificil de dezvoltat și de întreținut. Dezvolatorii trebuie să se concentreze nu numai pe funcționalitatea aplicației, ci și pe asigurarea securității și performanței acesteia. În plus, este necesar să se mențină un echilibru între user experience, securitate și scalabilitatea aplicației.

Pentru a face față acestei complexități, dezvoltatorii utilizează framework-uri și biblioteci care simplifică procesul de dezvoltare și permite o gestionare mai bună a complexității aplicațiilor.

Capitolul 2

Paradigmele actuale de randare

Rendering-ul este procesul de conversie a datelor într-o reprezentare vizuală și joacă un rol esențial în aplicațiile web, permițând furnizarea de conținut dinamic și interactiv către utilizatori.În stadiile incipiente ale dezvoltării web-ului, randarea a fost realizată în principal pe server. Site-urile web erau construite folosind tehnologii precum PHP sau Java, unde serverul genera conținutul HTML în mod dinamic, pe baza solicitărilor utilizatorilor. Această abordare are un TTFB (Time to first byte) rapid, deoarece serverul poate genera HTML ce poate fi livrat rapid clientului. Dezavantajul acestei paradigme este că nu este la fel de efectivă pentru site-urile web în care datele se schimbă des, ci doar pentru site-urile care nu necesită interactivitate complexă sau date dinamice. [17]

Multe dintre cele mai mari aplicaţii web din ziua de azi încă folosesc această abordare, cum ar fi, de exemplu, amazon.com: de fiecare dată când faceţi click pe un link, obţineţi o nouă pagină generată dinamic de pe serverele lor. În plus, există multe framework-uri pentru crearea aplicaţiilor multi-page, cum ar fi Ruby on Rails, Django şi Laravel, precum şi sisteme de gestionare a conţinutului, cum ar fi WordPress.

2.1 Starea de artă

Rendering-ul informaţiilor a evoluat semnificativ pe parcursul timpului, în conformitate cu evoluţia tehnologiilor web şi a cerinţelor utilizatorilor.

La început, majoritatea conţinutului era generat pe server şi returnat către browser în formă de pagini HTML statice. Cu trecerea timpului, tehnologiile web au evoluat, ceea ce a permis procesarea dinamică a datelor în browser prin intermediul script-urilor. Acest

lucru a permis dezvoltarea de aplicaţii web cu interacţiuni complexe şi randare dinamică a conţinutului. Cu toate acestea, performanţa aplicaţiilor web bazate exclusiv pe randare client-side a putut fi limitată în condiţii de conexiune lentă la internet. Din acest motiv, tehnologiile de randare hibridă au devenit din ce în ce mai populare, combinând avantajele randării server-side şi client-side. [18]

În prezent, tehnologiile de randare se bazează pe framework-uri şi biblioteci moderne, cum ar fi React, Angular şi Vue, care oferă unelte puternice pentru a construi aplicaţii web performante şi cu o experienţă utilizator remarcabilă. În plus, tehnologiile de randare moderne permit integrarea cu alte tehnologii precum WebAssembly şi Web Workers, care au ca scop îmbunătăţirea performanţei aplicaţiilor web. De asemenea, tehnologiile de randare permit utilizarea de animaţii şi interacţiuni complexe, care rezultă într-un user experience sporit al aplicației.

Evoluţia tehnologiei web şi a cerinţelor utilizatorilor continuă, iar randarea informaţiilor este un domeniu în continuă schimbare şi dezvoltare. În etapa curentă de dezvoltare a aplicaţiilor web, predomină utilizarea largă a JavaScript şi bibliotecilor JavaScript, cum ar fi React şi Vue.js.

React, dezvoltat de Facebook, a devenit una dintre tehnologiile cele mai populare pentru construirea de aplicaţii web, datorită abilităţii sale de a oferi o experienţă de randare eficientă şi de înaltă performanţă. Vue.js, dezvoltat de comunitate, se concentrează pe simplitate şi uşurinţă de utilizare, fiind o alegere populară pentru proiecte mai mici sau pentru dezvoltatorii care încearcă să înveţe tehnologii moderne de randare a informaţiilor.

Există mai multe tipuri de randare a conţinutului, cele mai populare fiind randarea *server-side*, *client-side* şi *static server generation*: [19]

1. **Server-side rendering:** Procesarea datelor şi generarea HTML se realizează pe server. Server-ul returnează apoi HTML-ul generat către browser, care îl afişează utilizatorului. Această metodă de randare este utilizată în mod traditional în aplicaţiile web vechi. Avantajul acestei metode este că se poate efectua o procesare a datelor pe server şi se poate oferi o experienţă utilizator consistentă chiar şi în condiţii de conexiune lentă la internet. Afișarea inițială a paginii poate fi rapidă deoarece clientul nu trebuie să aștepte încărcarea sau executarea JavaScript-ului înainte de afișarea conținutului.

2. **Client-side rendering:** Browser-ul primeşte date brute şi le procesează prin inter-

mediul unui script (cum ar fi JavaScript) pentru a genera HTML. Această metodă de randare permite o flexibilitate mai mare în prezentarea conţinutului şi oferă posibilitatea de a construi interacţiuni complexe cu utilizatorul, cum ar fi formulare dinamice şi componente grafice interactive. Dezavantajul acestei metode este că necesită mai multă putere de procesare a clientului şi poate fi mai puţin performantă în condiţii de conexiune lentă la internet, deoarece clientul trebuie să aștepte descărcarea scripturilor și a datelor necesare înainte de a randa pagina.

3. **Static server generation:** Paginile web sunt pregenerate și randate în timpul procesului de build al aplicației, rezultând în fișiere HTML statice care pot fi furnizate către client fără a fi necesară procesarea la nivelul serverului. Cu SSG, conținutul rămâne static până la următorul build al aplicației.

În ceea ce priveşte performanţa, tehnologiile moderne de randare a informaţiilor au avansat mult în optimizarea vitezei de randare şi a eficienţei resurselor. Una dintre abordările moderne de optimizare a performanţei este utilizarea *"virtual DOM"* (Document Object Model), care se caracterizează printr-o rerandare eficientă doar a componentelor ce au suferit modificări. În plus, tehnologiile de randare moderne pot delimita componentele dinamice, care necesită rerandare constantă, de cele statice, care nu necesită recalculare, crescând astfel performanţa aplicaţiei. De asemenea, tehnologiile de randare permit utilizarea de tehnici de lazy loading, care încarcă conţinutul doar atunci când este necesar, îmbunătăţind astfel performanţa aplicaţiei şi experiența utilizatorului. [6] Alegerea metodei potrivite de randare depinde de cerinţele specifice ale proiectului şi poate include consideraţii legate de performanţa, flexibilitatea şi compatibilitatea cu echipamentele utilizatorilor. Este important să se înţeleagă avantajele şi dezavantajele fiecărei metode şi să se facă o evaluare corespunzătoare a acestora înainte de a lua o decizie.

2.2 Client side rendering

Reîncărcarea întregii pagini la fiecare schimbare de URL este un proces costisitor, de aceea pentru a reduce necesitatea de a regenera conţinutul au apărut soluţii precum React şi Angular - framework-uri JavaScript, ce permit modificarea componentelor fără a cauza o reîncărcare a paginii. Această arhitectură se numeşte SPA (*Single Page Application*).

Într-un SPA, resursele HTML, CSS și JavaScript inițiale sunt încărcate atunci când aplicația este accesată pentru prima dată. Interacțiunile ulterioare și modificările stării aplicației sunt gestionate prin efectuarea de cereri asincrone către server pentru a obține date, care sunt apoi randate pe partea de client (*client-side rendering*). Aceasta rezultă într-o experiență mai receptivă și interactivă, deoarece aplicația poate actualiza componente sau secțiuni specifice ale paginii fără a comunica cu serverul.

O aplicaţie ce utilizează CSR este caracterizată prin următoarele aspecte:

1. **Cererea HTTP a clientului** - când utilizatorul introduce URL-ul în bara de adrese a browserului, se stabilește o conexiune HTTP cu serverul.

2. **Răspunsul HTTP al serverului** - Serverul trimite înapoi fișierul HTML inițial, care conține referințe către fișiere JavaScript și CSS.

3. **Încărcarea paginii** - browserul clientului primește fișierul HTML inițial și începe randarea structurii de bază a paginii, inclusiv orice conținut static.

4. **Încărcarea fișierelor JavaScript** - browserul descarcă fișierele JavaScript referenţiate în fișierul HTML inițial.

5. **Preluarea datelor** - codul JavaScript executat de browserul clientului face cereri către server sau API-uri, pentru a prelua datele necesare pentru a randa conținutul dinamic.

6. **Randarea şi actualizarea** - odată ce datele sunt preluate, codul JavaScript procesează datele și generează dinamic structura HTML necesară. Browserul actualizează pagina randată cu conținutul HTML generat dinamic, care poate include liste, tabele, formulare sau alte componente

7. **Interactivitatea** - ascultătorii de evenimente și interacțiunile utilizatorului, cum ar fi click-urile sau introducerile de date, declanșează execuția de cod JavaScript ulterioară, care poate actualiza conținutul randat, fără a necesita o reîncărcare completă a paginii.

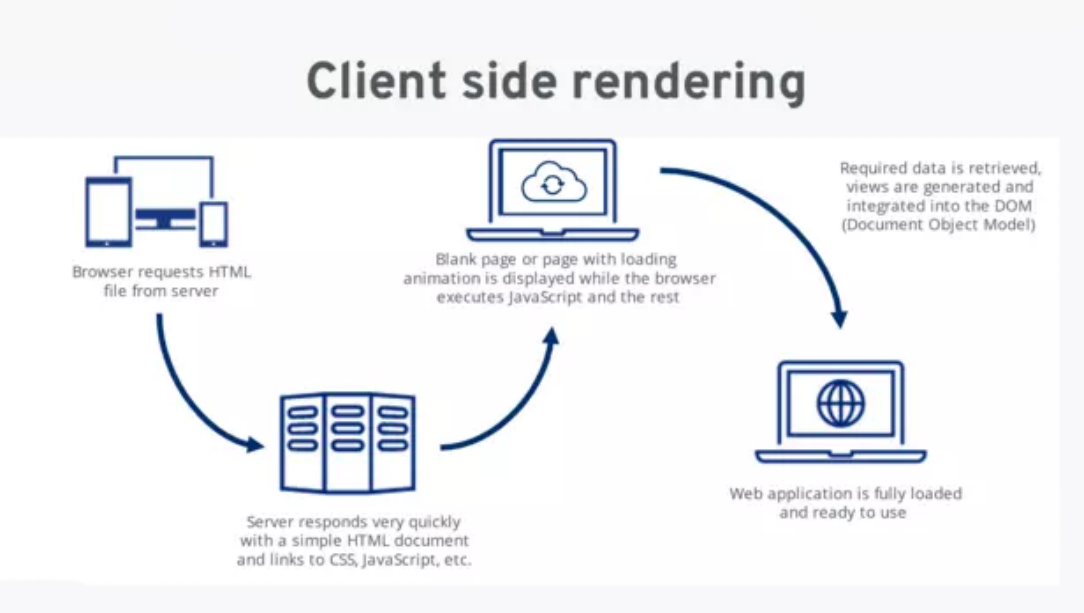

Figura 2.1: Client side rendering [1]

Deoarece browserul trebuie să descarce și să ruleze întregul cod al aplicației înainte ca conținutul să apară pe ecran, încărcarea inițială a paginii este de obicei lentă. Ca rezultat, utilizatorii văd o pagină goală sau un indicator de încărcare pentru o perioadă relativ lungă de timp. Aceasta duce la o experiență mai puțin plăcută pentru utilizatori și la o rată mai mare de abandonări a paginii (*bounce rate*). [21]

Cu toate acestea, pot exista provocări în ceea ce privește optimizarea pentru motoarele de căutare (SEO), deoarece toate resursele aplicației sunt încărcate la prima accesare, crawler-ii web tradiționali pot întâmpina dificultăți în parcurgerea și indexarea conținutului, iar volumul mare de scripturi ce sunt descărcate, bibliotecile externe şi dependențele pot rezulta într-un timp de încărcare inițial mai mare.

2.3 Server side rendering

O nouă abordare, ce avea scopul de a rezolva problemele aplicatiilor single page este server-side rendering (SSR).

SSR este o tehnică în dezvoltarea aplicațiilor web care implică generarea de HTML pe server și trimiterea acestuia către client. Spre deosebire de CSR, procesul de randare

[1] `https://www.ionos.ca/digitalguide/websites/web-development/server-side-and-client-side-scripting-the-differences/`

se execută pe server, ce reprezintă un avantaj semnificativ, datorită timpului mai rapid de încărcare inițială a paginilor. [17]

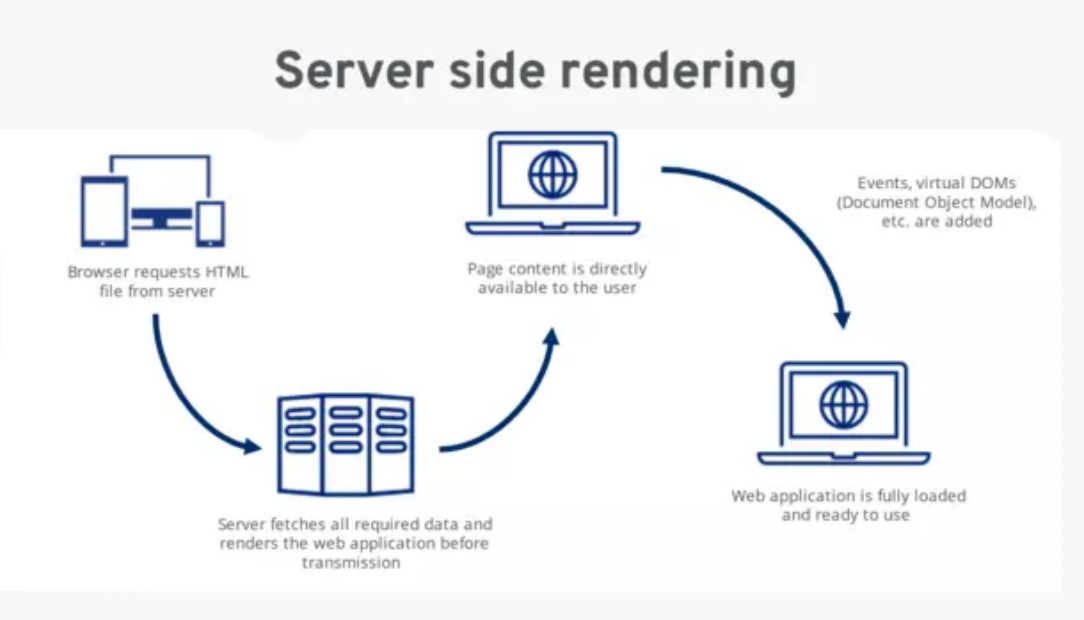

Figura 2.2: Server side rendering [2]

Un alt avantaj al SSR este îmbunătățirea optimizării pentru motoarele de căutare (SEO). Deoarece crawler-ii motoarelor de căutare se bazează în mod tipic pe conținutul HTML, serve-side rendering furnizează pagini complet randate motoarelor de căutare, care pot fi indexate ușor. Aceasta îmbunătățește vizibilitatea și accesibilitatea site-ului în rezultatele motoarelor de căutare.

SSR aduce beneficii și utilizatorilor cu conexiuni lente la internet sau dispozitive mai vechi. Prin pre-randarea HTML-ului pe server, SSR asigură utilizatorii cu conținut chiar și în cazul în care dispozitivele lor au putere de procesare limitată sau capacități de rețea reduse. Aceasta îmbunătățește accesibilitatea și experiența utilizatorului pentru o gamă mai largă de utilizatori.

Server-side rendering are câteva dezavantaje asociate. Deoarece fiecare solicitare necesită o nouă randare pe server, aceasta poate duce la o încărcare mai mare a serverului și la un timp de răspuns mai lent. De asemenea, este mai dificil de implementat și de întreținut, deoarece necesită o infrastructură de servere mai complexă. SSR creează o dependență de server, deoarece clientul primește pagini complet redată de la server. În

[2] `https://www.ionos.ca/digitalguide/websites/web-development/server-side-and-client-side-scripting-the-differences/`

cazul în care serverul întâmpină probleme sau este indisponibil, utilizatorii nu pot accesa sau utiliza aplicația. Gestionarea stării clientului poate deveni și mai complexă cu SSR. Sincronizarea și actualizarea stării între client și server pentru a menține coerența aplicației necesită o implementare bine gândită și poate fi dificil de gestionat.

O aplicație ce utilizează SSR este caracterizată prin următoarele aspecte:

1. **Cererea HTTP din partea clientului** - când utlizitorul introduce URL-ul în broswer, se stabileşte o conexiune HTTP cu serverul şi apoi se trimite serverului o cerere pentru a obţine documentul HTML.
2. **Preluarea datelor** - serverul preia datele necesare din baza de date sau API-uri.
3. **Pre-randarea pe partea de server** - serverul compilează componentele JavaScript în HTML static.
4. **Răspunsul HTTP al serverului** - serverul trimite documentul compilat către client.
5. **Încărcarea și afișarea paginii** - clientul descarcă fişierul HTML şi afişează componentele statice pe pagină.
6. **Hidratarea** - clientul descarcă fișierul sau fișierele JavaScript încorporate în HTML, procesează codul și atașează ascultători de evenimente componentelor.

[20]

2.4 Static server generation

SSG este o paradigma alternativă în dezvoltarea web care îmbină beneficiile SSR și CSR. Spre deosebire de server-side rendering, unde HTML-ul este generat dinamic la fiecare cerere, SSG generează fișiere HTML statice în timpul procesului de build, care sunt apoi servite către client.

Unul dintre avantajele majore ale SSG este performanța îmbunătățită și scalabilitatea. Deoarece HTML-ul este pre-randat, serverul poate servi fișiere statice direct, fără a necesita prelucrări suplimentare. Aceasta duce la încărcarea mai rapidă a paginilor și reducerea încărcării serverului, făcându-l potrivit pentru site-urile cu trafic intens.

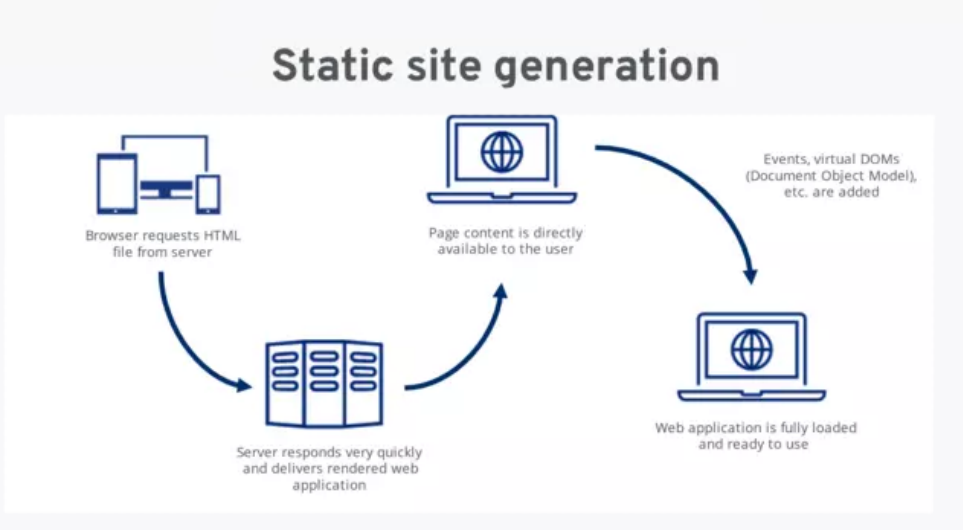

Figura 2.3: Static server generation [3]

O aplicație ce utilizează SSR este caracterizată prin următoarele aspecte:

1. **Procesul de build** - în timpul acestui proces codul sursă și fișierele de conținut ale site-ului web sunt analizate.

2. **Compilarea conținutului** - conţinutul fişierelor este procesat şi convertit din formatul sursă (Markdown sau YAML) în formatul HTML.

3. **Modelarea** - sunt aplicate şabloane pe fişierele compilate, permițând un aspect și o structură coerentă.

4. **Integrarea datelor** - sunt preluate date din diverse surse (baze de date, API-uri) şi sunt integrate în paginile generate anterior.

5. **Generarea HTML-ului static** - fişierele HTML sunt generate pe baza conţinutului compilat, a şabloanelor şi a datelor integrate.

6. **Optimizarea resurselor** - în timpul procesului de build, resursele precum CSS, JavaScript și imaginile se optimizează, reducând dimensiunile fișierelor și îmbunătățind performanța.

7. **Implementarea** - odată ce fișierele statice sunt generate, acestea pot fi păstrate în cache pe server sau pe CDN, permițând servirea instantanee a cererilor ulterioare

[3] `https://www.ionos.ca/digitalguide/websites/web-development/server-side-and-client-side-scripting-the-differences/`

fără a necesita regenerarea lor (*minimizează încărcarea serverului și îmbunătățește scalabilitatea generală, în special în scenarii în care actualizările de conținut sunt rare*).

8. **Cererea clientului** - când un utilizator vizitează o pagină, serverul web sau CDN-ul servește direct fișierul HTML static pre-generat.

9. **Afișarea paginii** - browserul clientului primește fișierul HTML static și afișează conținutul pe pagină.

10. **Interactivitatea** - Dacă există elemente interactive sau funcționalități dinamice pe pagină, codul JavaScript poate fi utilizat pentru a îmbunătăți experiența utilizatorului, de obicei prin preluarea de date suplimentare din API-uri sau prin gestionarea interacțiunilor utilizatorului.

Cu toate acestea are și câteva limitări. SSG nu este potrivit pentru aplicații ce necesită actualizări în timp real sau conținut dinamic, deoarece fișierele statice pre-randate nu se modifică până la următorul proces de build. Acest lucru poate introduce întârzieri în disponibilitatea conținutului și creșterea complexității gestionării actualizărilor de conținut. [7] Unul dintre cele mai mari dezavantaje ale acestui abordări este timpul de build. Dacă site-ul consă dintr-un număr mare de pagini, procesul de build al acestora va dura mult timp.

2.5 Alegerea paradigmei corespunzătoare

Alegerea între CSR, SSR și SSG este importantă în dezvoltarea unei aplicații web, deoarece are un impact direct asupra experienței utilizatorului, performanței și funcționalității generale a aplicației. Fiecare abordare are propriile sale puncte forte și considerații, iar alegerea depinde de mai mulţi factori, cum ar fi cerințele proiectului, scalabilitatea, complexitatea și obiectivele de performanță. Principalele aspecte de luat în considerare sunt:[8]

1. **Experiența utilizatorului** - Abordarea de randare afectează în mod semnificativ modul în care conținutul aplicației este livrat în browserul utilizatorului. CSR pune accent pe interactivitate, prin încărcarea inițială a unei structuri HTML minimale și apoi preluarea și randarea datelor pe partea clientului, ceea ce duce la o încărcare

inițială mai rapidă a paginii. SSR, pe de altă parte, furnizează o pagină HTML complet redată de pe server, fapt ce reduce timpul de încărcare inițial și asigură ca conținutul să fie imediat vizibil utilizatorilor. SSG generează fișiere HTML statice în timpul procesului de construcție, permitând încărcări de pagini ultra-rapide, deoarece nu este necesară prelucrarea în server.

2. **Optimizarea pentru motoarele de căutare** - Motoarele de căutare întâmpină adesea dificultăți în indexarea conținutului JavaScript dinamic. SSR și SSG, care livrează conținut HTML pre-randat în browser, sunt mai efective cu SEO în comparație cu CSR. Acest lucru se datorează faptului că motoarele de căutare pot parcurge și indexa ușor conținutul HTML static, ceea ce ajută la îmbunătățirea vizibilității și descoperirii.

3. **Performanța** - Abordarea de randare afectează performanța generală a aplicației. CSR poate duce la timp de încărcare mai lung, în special în cazul conexiunilor de rețea mai lente sau al unor cantități mari de date de preluat. SSR reduce timpul de randare inițial prin furnizarea unei pagini pre-randate de pe server, însă navigarea ulterioară poate necesita apeluri suplimentare către server. SSG oferă cea mai bună performanță, deoarece generează fișiere HTML statice care pot fi livrate direct utilizatorului, minimizând necesitatea de prelucrare pe server.

4. **Complexitatea și fluxul de dezvoltare** - Framework-urile CSR, cum ar fi React, Angular sau Vue.js, necesită ca dezvoltatorii să gestioneze logica de randare și stările componentelor, ceea ce poate introduce complexitate. SSR și SSG reduc, într-o anumită măsură, această complexitate prin transferul logicii de randare la server sau la procesul de build al aplicației. SSR necesită configurare pe partea de server și cunoștințe despre tehnologiile backend, în timp ce SSG simplifică fluxul de lucru în dezvoltare prin generarea de active statice care pot fi ușor implementate pe orice server web.

5. **Conținutul dinamic și actualizări în timp real** - CSR se evidențiază în aplicațiile extrem de interactive care se bazează pe actualizări frecvente ale datelor sau pe funcționalități în timp real. Cu CSR, datele pot fi actualizate fără necesitatea unei reîncărcări complete a paginii. SSR și SSG sunt mai potrivite pentru aplicațiile axate pe conținut mai puțin dinamic, deoarece necesită o solicitare către server

pentru a actualiza datele.

Alegerea paradigmei potrivite este esențială pentru a furniza o aplicație web rapidă, interactivă și optimizată pentru motoarele de căutare. De asemenea, este important să se ia în considerare cerințele proiectului, complexitatea și obiectivele de performanță pentru a determina cea mai bună abordare de randare. Aceasta este una dintre primele decizii importante ce trebuie luată în dezvoltarea unei aplicații web, deoarece este dificil de schimbat ulterior.

Capitolul 3

Aplicația practică

Obiectivul acestui capitol este de a oferi o experiență practică cu tehnologiile de randare și de a evalua performanța și eficiența acestora. Rezultatele obținute vor fi valoroase pentru dezvoltatori și arhitecți care doresc să optimizeze procesele de randare și să îmbunătățească experiența utilizatorului.

Pentru o obiectivitate sporită a fost implementată aceeaşi aplicaţie, cu ajutorul mai multor framework-uri JavaScript, fiecare dintre ele reprezentând o abordare diferită de randare (CSR, SSR, SSG). De asemenea, aplicația a fost implementată şi cu ajutorul vanilla JavaScript, pentru a compara aplicațiile după un anumit set de criterii cu scopul de a identifica care sunt aspectele cheie care diferă între aplicațiile construite cu ajutorul framework-urilor JavaScript și cea construită folosind vanilla JavaScript. Rezultatul acestui studiu comparativ oferă o imagine de ansamblu asupra avantajelor și dezavantajelor fiecărei tehnici de randare, precum și a celor mai optime abordări posibile.

3.1 Metrici de performanță

Pentru compararea efectivă a metodelor de implementare, este nevoie de a compara rezultatele după anumite criterii care reprezintă performanța generală a aplicației. Metricile care vor fi utilizate în evaluarea performanței pentru toate implementările sunt:

- **Largest Contentful Paint (*LCP*)** - metrică de performanță utilizată pentru a măsura viteza de încărcare și stabilitatea vizuală percepută a unei pagini web, introdusă de Google. LCP măsoară timpul necesar ca cel mai mare element vizibil de pe o pagină web să fie afișat în zona de vizualizare a utilizatorului. Acest element poate

fi o imagine, un videoclip sau un element de tip block, cum ar fi un paragraf sau un antet. Cu cât LCP este mai rapid, cu atât este mai bună experiența utilizatorului, deoarece indică faptul că conținutul principal al paginii este afișat rapid. Un scor bun al LCP este considerat în general sub 2,5 secunde. Dacă LCP depășește acest prag, poate duce la o rată mai mare de respingere și poate afecta negativ angajamentul utilizatorului. Pentru a îmbunătăți LCP, dezvoltatorii web pot optimiza performanța site-ului prin minimizarea resurselor care blochează afișarea, optimizarea imaginilor și videoclipurilor și implementarea tehnicilor precum lazy-load și code splitting.

LCP este una dintre principalele metrici luate în considerare de motoarele de căutare, inclusiv Google, pentru a evalua experiența utilizatorului pe un site web. Site-urile cu scoruri bune ale LCP sunt mai susceptibile de a obține un rang mai înalt în rezultatele căutării și de a oferi o experiență de navigare mai bună utilizatorilor. [9]

- **Cumulative Layout Shift (*CLS*)** - este utilizată pentru a măsura stabilitatea vizuală a unei pagini web. Aceasta cuantifică cantitatea de schimbări de aspect neașteptate care apar în timpul încărcării paginii. Schimbarea de aspect se referă la mișcarea elementelor paginii, cum ar fi imagini, butoane sau text, într-un mod care perturbă experiența de citire sau de interacțiune a utilizatorului. Aceste schimbări pot fi frustrante pentru utilizatori, în special atunci când cauzează click-uri accidentale sau fac conținutul dificil de citit.

  CLS este calculat prin măsurarea fracţiunii de impact și fracţiunii de distanță a fiecărui eveniment de schimbare de aspect și apoi însumându-le pe întreaga încărcare a paginii. Fracţiunea de impact reprezintă proporția vizualizării afectată de schimbare, în timp ce fracţiunea de distanță reprezintă distanța maximă pe care elementul o parcurge în raport cu ecranul utilizatorului. [10]

  Pentru a oferi o bună experiență utilizatorului, o pagină web ar trebui să aibă un scor CLS scăzut. Un scor CLS sub 0.1 este considerat excelent, între 0.1 și 0.25 este considerat bun, în timp ce scorurile peste 0.25 sunt considerate slabe.

  Pentru a reduce CLS, dezvoltatorii web pot urma practici, cum ar fi stabilirea dimensiunilor explicite pentru elementele media, rezervarea spațiului pentru reclame sau conținut dinamic și evitarea injectării dinamice a conținutului deasupra elemen-

telor existente. Prin minimizarea schimbărilor de aspect neașteptate, dezvoltatorii pot îmbunătăți stabilitatea vizuală a paginilor web și experiența utilizatorilor.

- **First Input Delay (*FID*)** - măsoară reactivitatea și interactivitatea unei pagini web. Ea cuantifică timpul necesar ca o pagină web să răspundă la prima interacțiune a utilizatorului, cum ar fi apăsarea unui buton sau selectarea unui meniu derulant.

  FID se concentrează în mod specific pe întârzierea dintre prima interacțiune a utilizatorului și capacitatea browserului de a răspunde la acea interacțiune. Este importantă deoarece reflectă reactivitatea percepută a unui site web și are un impact direct asupra experienței utilizatorului. Un FID redus indică un site web mai interactiv și mai receptiv, în timp ce un FID ridicat poate face un site web să pară lent și neinteractiv. [11]

  FID-ul este măsurat în milisecunde (ms), iar un scor bun este considerat a fi mai mic de 100 de milisecunde. Un scor peste 300 de milisecunde este considerat slab și poate rezulta într-o experiență frustrantă pentru utilizatori.

  Pentru a îmbunătăți FID, dezvoltatorii web pot optimiza codul lor, reduce timpul de execuție al JavaScript-ului și prioritiza sarcinile critice pentru a asigura timp de răspuns mai rapid la interacțiunile utilizatorului. Tehnici precum minimizarea codului, împărțirea codului și încărcarea asincronă a scripturilor pot contribui la reducerea impactului JavaScript-ului asupra FID-ului.

  FID-ul este una dintre metricile centrale ale aspectelor vitale ale webului utilizate de motoarele de căutare, inclusiv Google, pentru a evalua experiența utilizatorului pe un site web. Site-urile cu scoruri bune de FID au mai multe șanse să obțină un rang mai înalt în rezultatele căutării și să ofere o experiență de navigare mai fluidă și mai captivantă utilizatorilor.

- **Interaction to Next Paint (*INP*)** - metrică a Core Web Vitals, care va înlocui First Input Delay (FID) din martie 2024. INP evaluează reactivitatea folosind date din Event Timing API. Atunci când o interacțiune determină ca o pagină să devină neproductivă, aceasta reprezintă o experiență utilizator slabă. INP observă latența tuturor interacțiunilor pe care le-a realizat utilizatorul cu pagina și raportează o valoare unică sub care au fost toate (sau aproape toate) interacțiunile. Un INP scăzut înseamnă că pagina a putut răspunde în mod constant și rapid tuturor sau

unei mari majorități a interacțiunilor utilizatorului.

Unele interacțiuni vor dura mai mult decât altele, dar în special pentru interacțiunile complexe, este important să prezentați un feedback vizual inițial ca indiciu pentru utilizator că se întâmplă ceva în background. Timpul până la următoarea desenare (*paint*) este cea mai timpurie oportunitate pentru a face acest lucru. Prin urmare, intenția INP nu este de a măsura toate efectele ulterioare ale interacțiunii (cum ar fi fetch-urile și actualizările UI din alte operații asincrone), ci timpul în care următoarea desenare este blocată. Prin întârzierea feedbackului vizual, puteți da utilizatorilor impresia că pagina nu răspunde acțiunilor lor. [12]

Scopul INP este de a asigura ca timpul de la inițierea unei interacțiuni de către utilizator până la următoarea cadru desenat să fie cât mai scurt posibil, pentru toate sau majoritatea interacțiunilor realizate de utilizator.

3.2 Metode de evaluare a performanței

Există mai multe metode disponibile pentru măsurarea performanței unei aplicații web. Aceste instrumente pot ajuta dezvoltatorii să monitorizeze și să îmbunătățească performanța și experiența utilizatorilor.

În evaluarea aplicației curente au fost folosite următoarele metode de evaluare a performanței:

- **Web Vitals** (extensie pentru browser) - inițiativă de la Google, care oferă ghidare unitară pentru semnalele de calitate care sunt esențiale pentru a oferi o experiență excelentă utilizatorilor pe web.

  Google a furnizat o serie de instrumente pentru a măsura și raporta performanța. Inițiativa Web Vitals se concentrează pe metricile Core Web Vitals. Fiecare metrică reprezintă o componentă distinctă a experienței utilizatorului, este măsurabilă în teren și reflectă experiența reală a unui rezultat critic centrat pe utilizator. [13]

  Setul actual se concentrează pe trei aspecte ale experienței utilizatorului - încărcare, interactivitate și stabilitate vizuală și include următoarele metrici: Largest Contentful Paint (LCP), First Input Delay (FID), Cumulative Layout Shift (CLS).

- **Lighthouse** - unealtă de testare automată, open-source, dezvoltată de Google, care ajută la măsurarea și îmbunătățirea calității și performanței paginilor web.

Aceasta oferă o analiză cuprinzătoare a diferitelor aspecte ale unei pagini web precum performanța, accesibilitatea, optimizarea pentru motoarele de căutare (SEO). Lighthouse este folosit în mod frecvent de dezvoltatori pentru optimizarea site-urilor și aplicațiilor web în vederea obținerii unei experiențe mai bune pentru utilizatori, iar datorită rapoartelor generate, poate oferi informaţii despre punctele slabe ale unei aplicaţii web şi recomandări necesare pentru a le remedia.

- **PageSpeed Insights** - unealtă de analiză a performanței web dezvoltată de Google. Oferă informații și recomandări pentru optimizarea vitezei și performanței unei pagini web. Prin analizarea conținutului unei adrese URL, PageSpeed Insights generează un raport în care sunt identificate problemele legate de performanță și sugerează opţiuni pentru a le remedia. Fiecare sugestie de optimizare este clasificată ca fiind ușor de implementat, medie sau dificilă, iar utilizatorii pot alege să filtreze rezultatele după complexitate.

3.3 Comparaţia şi analiza practică a aplicaţiilor

Pentru a compara performanța dintre mai multe framework-uri JavaScript, este nevoie de o aplicație care va fi implementată în toate framework-urile, fiind identice din punct de vedere al interfec tei cu utilizatorul, pentru o evaluare cât mai obiectivă. Scopul acestei analize este de a determina care abordare oferă cea mai bună performanță în ceea ce privește timpul de încărcare al paginii, viteza de randare și experiența generală a utilizatorului. Totodată se vor lua în considerare ușurința implementării, documentația disponibilă și scalabilitatea efectivă a aplicației.

3.3.1 Metodologie

Pentru această comparație a performanței, am selectat vanilla JavaScript și React, datorită popularității și utilizării extinse. Am creat o aplicație demonstrativă care include o imagine de fundal aleatorie preluată de la un API extern, o listă de utilizatori și postări asociate preluate de pe un alt API, elemente ce conțin animații și stilizare a componentelor, un input care permite filtrarea utilizatorilor.

Performanța aplicației a fost testată atât în mediul local, cât și după deploy pe web hosting.

3.3.2 Vanilla JavaScript

Structura aplicație constă din trei fișiere:

- **index.html** - conține markup-ul aplicației
- **app.js** - conține toată logica aplicației (data fetching, data rendering, data mutations)
- **style.css** - conține stilizarea elementelor html

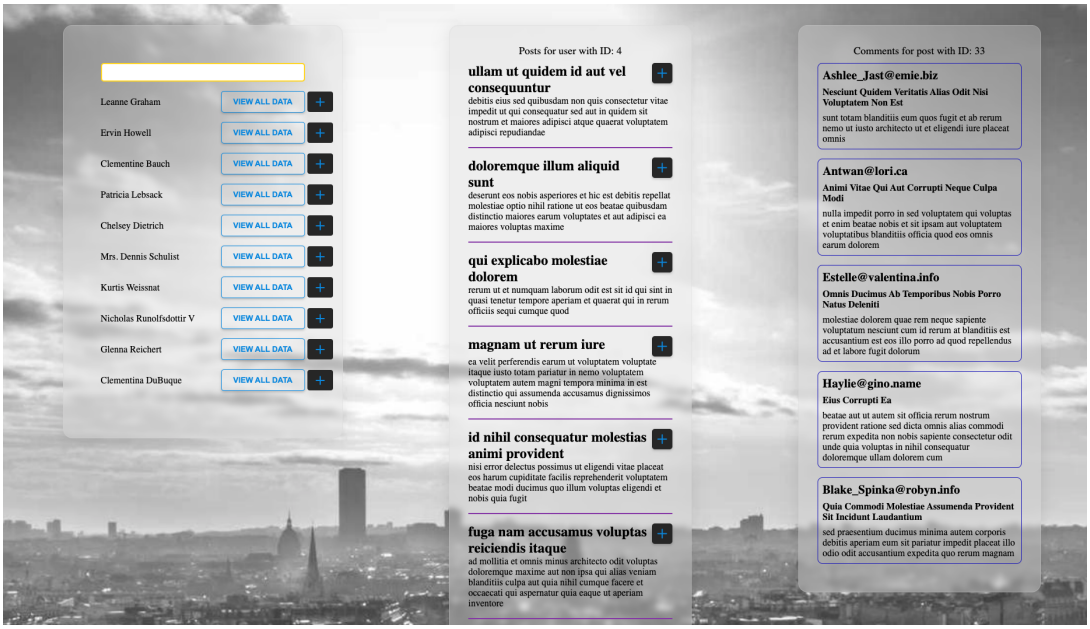

Figura 3.1: Aplicația practică - vanilla JavaScript

Pentru a obține datele de la API, am utilizat funcția *fetch*:

```
const usersRes = await fetch("https://jsonplaceholder.typicode.com/users");
const postsRes = await fetch("https://jsonplaceholder.typicode.com/posts");
```

Pentru a implementa funcționalitatea de afișare a postărilor asociate unui utilizator, am utilizat localstorage pentru a stoca Id-ul și numele utilizatorului curent, pentru a folosi aceste date în funcția de filtrare a postărilor:

```
function handleSaveButtonClick(event) {
  localStorage.setItem("selectedUser", JSON.stringify(selectedUser));
}
```

```
async function renderPosts(data) {
   const postsUl = document.getElementById("posts");
   const selectedName = localStorage.getItem("selectedName");
   const selectedUser = ALL_USERS.find((user) => user.name === selectedName);

   const html = data.map( (posts) =>
      `<li class="list-item-post">
      <h2 class='post-header'> ${post.title} </h2>
      <p> ${post.body} </p>
      </li>`).join("");

   postsUl.innerHTML += html;
}
```

3.3.3 React

Cu ajutorul comenzii *npx create-react-app 02-react* se generează un proiect React, cu numele 02-react, care va conține fișierele configurate și folderele necesare pentru aplicația React, inclusiv structura de bază a proiectului, fișierele de configurare implicite și dependențele inițiale.

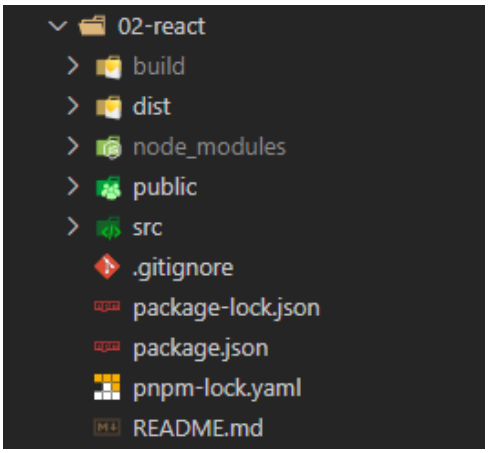

Figura 3.2: Structura unei aplicații React

Directorul "src" dintr-un proiect conține fișierele de cod sursă pentru aplicația React. Aici sunt scrise și organizate componentele, stilurile și alte fișiere relevante ale aplicației.

Aplicația construită cu React are câteva avantaje față de o aplicație construită cu vanilla JavaScript:

1. **Eficiență și performanță**: Virtual DOM permite re-randarea selectivă a componentelor. Virtual DOM-ul este o copie a DOM-ului real, stocată în memorie. Atunci când apar modificări în starea sau proprietățile unei componente, React declanșează un proces de re-randare - se compara Virtual DOM-ul anterior cu cel nou generat pentru componenta afectate şi descendenţii acesteia, folosind procesul de reconciliere. Prin calcularea diferențelor (diffing) între cele două Virtual DOM-uri, React identifică părțile specifice care necesită actualizare, apoi aplică doar actualizările necesare componentelor afectate în DOM-ul real, minimizând randările excesive ale componentelor nemodificate. Capacitatea Virtual DOM-ului de a efectua actualizări selective contribuie la viteza și eficiența React, în special în gestionarea structurilor complexe de interfață de utilizator și a modificărilor frecvente ale stării sau proprietăților.

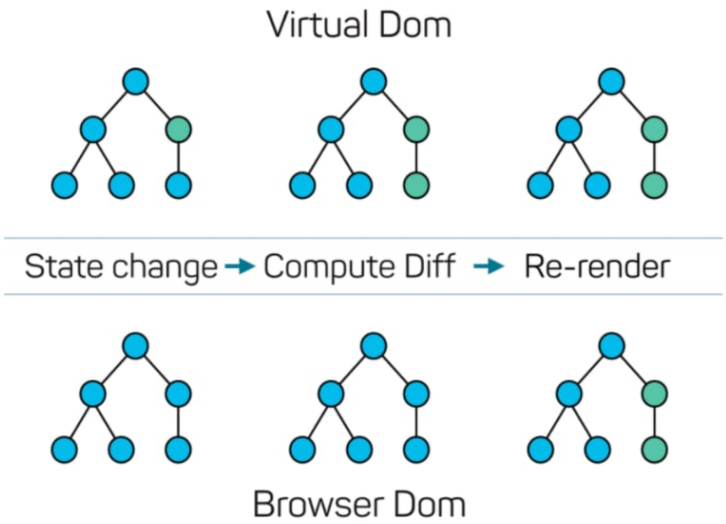

Figura 3.3: Ilustrarea procesului de reconciliere

[1]

2. **Sintaxă declarativă**: React urmează o paradigma de programare declarativă, ce permite dezvoltatorilor să descrie cum ar trebui să arate interfața utilizator pe baza stării curente a componentelor, în loc să manipuleze manual DOM-ul. Aceasta

[1] `https://dev.to/adityasharan01/react-virtual-dom-explained-in-simple-english-10j6`

simplifică dezvoltarea, îmbunătățește lizibilitatea codului și reduce probabilitatea apariției erorilor.

3. **React hooks**: oferă o modalitate mai concisă și intuitivă de a gestiona starea și efectele secundare. Cu ajutorul hooks-urilor, componentele funcționale pot avea starea locală (*hook-ul useState*), pot efectua efecte secundare(*hook-ul useEffect*), pot accesa valorile contextului (*hook-ul useContext*), pot crea referințe mutable folosind hook-ul useRef, pot memoiza calculele folosind hook-ul useMemo și pot memoiza funcțiile de callback folosind hook-ul useCallback. Hooks-urile promovează reutilizabilitatea codului, îmbunătățesc lizibilitatea și experiența de dezvoltare, permițând dezvoltatorilor să scrie componente mai modulare și funcționale fără a fi nevoie de componente de tip clasă. Acestea au devenit o parte esențială a React-ului, facilitând gestionarea logicii și a efectelor secundare în cadrul componentelor funcționale. [16]

Pentru a reține anumite date, precum utilizatorul curent, masivul de postări sau valoarea curentă a input-ului putem utiliza hook-ul *useState*, ce permite gestionarea și actualizarea stării interne a unei componente, iar pentru a obține listele de utilizatori și postrări într-un mod efectiv, utilizăm hook-ul *useEffect* pentru a ne asigura că acestea sunt executate la momentele potrivite în timpul ciclului de viață al componentei.

```
useEffect(() => {
 async function fetchData() {
   const usersRes = await fetch("https://jsonplaceholder.typicode.com/users");
   const postsRes = await fetch("https://jsonplaceholder.typicode.com/posts");

   const usersData = await usersRes.json();
   const postsData = await postsRes.json();

   allUsers.current = usersData;
   allPosts.current = postsData;
   setUsers(usersData);
   setPosts(postsData);
 }
 fetchData();
}, []);
```

3.3.4 Next.js

Este un framework pentru backend, bazat pe React. Tot ce se poate realiza în React se poate realiza şi în Next.js, însă acesta vine cu o serie de caracteristici suplimentare, precum rutarea, apeluri API, autentificare, pre-rendering, SEO, etc. Aceste funcţionalităţi nu sunt disponibile în React în mod implicit şi trebuie instalate biblioteci și dependințe externe, precum React Router pentru a obţine funcţionaităţi similare.

Cu ajutorul comenzii *npx create-next-app@latest 03-next* putem crea o aplicaţie Next.js. Structura iniţială a unei aplicaţii este similară cu cea a unei aplicaţii React, cu excepţia unor fişiere suplimentare:

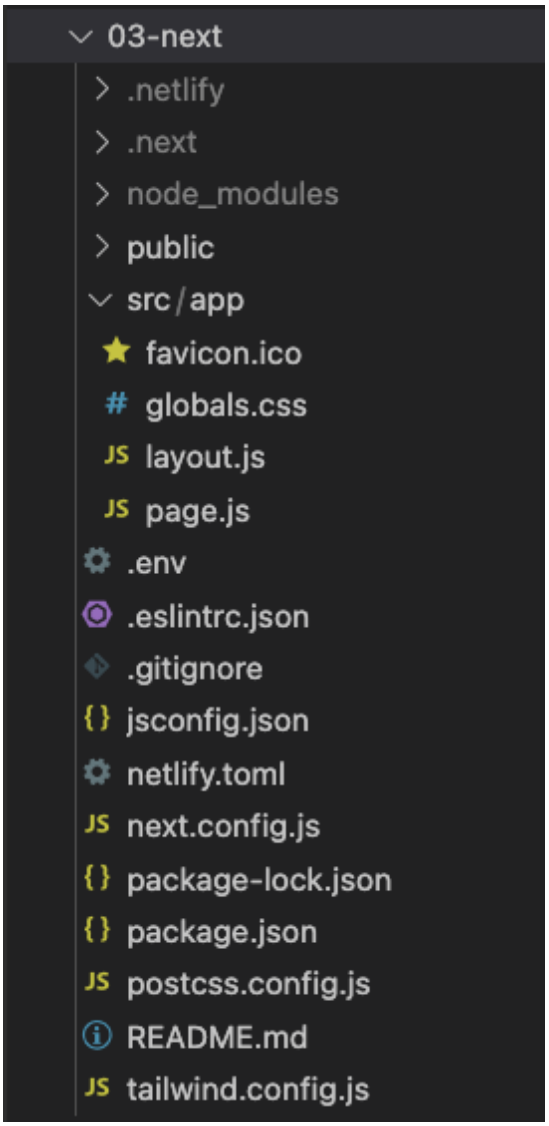

Figura 3.4: Structura unei aplicații Next.js

3.3.5 Gatsby

De asemenea este bazat pe React, dar spre deosebire de Next.js este un framework pentru frontend. Gatsby este un generator de site-uri statice, ce permite dezvoltatorilor să creeze site-uri web performante, ce pot fi scalate şi cu optimizate pentru motoarele de căutare. În ceea ce privește capacitatea de gestionare a datelor, Gatsby impune un mod exact în care datele trebuie preluate şi gestionate, cu ajutorul la GraphQL, în timp ce Next.js este mai flexibil din acest punct de vedere.

Pentru a crea o aplicaţie Gatsby, este nevoie de comanda *npm init gatsby*. Structura iniţială a unei aplicaţii:

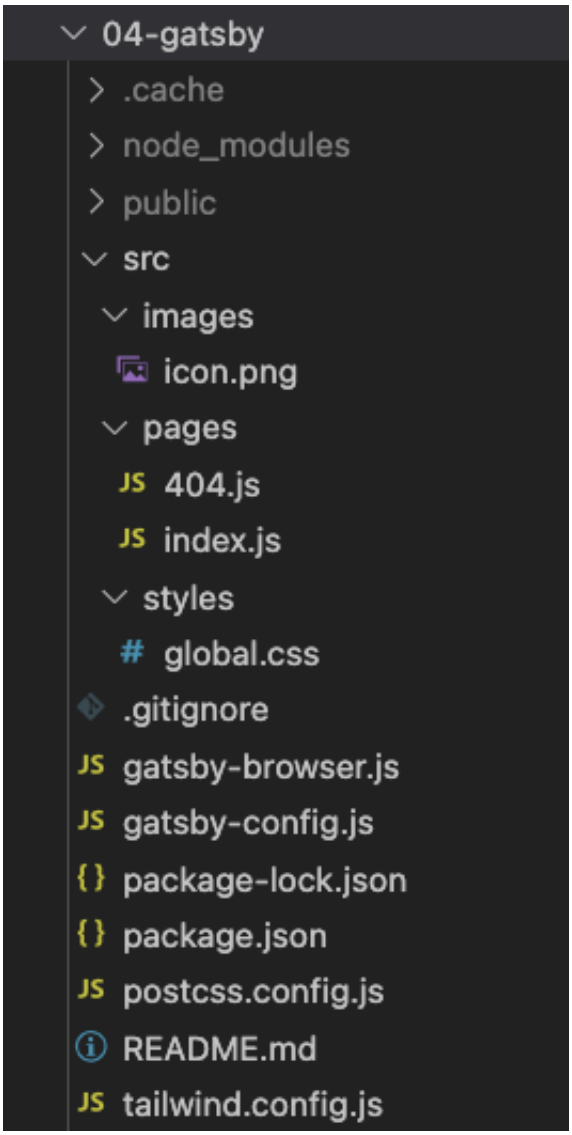

Figura 3.5: Structura unei aplicații Gatsby

Folderul *src* este compus din *images*, unde sunt stocate imaginile utilizate, *pages*, unde sunt definite paginile aplicaţiei şi folderul *styles*, unde se află fişierele de stil. Denumirea unui fişier din folderul *pages* reprezintă ruta paginii respective. De exemplu, fişierul *index.js* reprezintă ruta /, iar fişierul *about.js* reprezintă ruta */about*.

Gatsby optimizează aplicațiile prin utilizarea de plugin-uri, care sunt pachete sau module ce extind funcționalitatea Gatsby și oferă optimizări suplimentare. Aceste plugin-uri pot fi folosite pentru îmbunătățirea diverselor aspecte ale aplicației, inclusiv performanța, optimizarea imaginilor 3.5.1, împărțirea codului (*code splitting*), optimizarea SEO.

3.4 Analiza performanței

Performanța aplicațiilor a fost testată cu ajutorul Web Vitals, Lighthouse, PageSpeed Insights, după ce aplicaţiile au fost încărcate pe un web hosting, pentru a obţine o imagine completă şi obiectivă asupra performanţei acestora. Rezultatele obținute oferă informații detaliate despre performanța aplicațiilor, inclusiv timpul de încărcare a paginii, interactivitatea, stabilitatea vizuală, metode de îmbunătățire a acestora şi o comparaţie subiectivă a procesului de configurare pentru deploy-ul aplicaţiilor pe hosting.

3.4.1 Web Vitals

Toate aplicaţiile au fost testate după ce au fost configurate pe acelaşi web hosting - Netlify, în condiţii de conexiune stabilă la internet şi cu simularea unei conexiuni lente (slow 3G).

Rezultatele medii obţinute sunt prezentate în tabelul următor:

Aplicaţie	**Conexiune normală**				**3G lent**			
	LCP	**CLS**	**FID**	**INP**	**LCP**	**CLS**	**FID**	**INP**
JavaScript	0.241s	0.006	0.500ms	32.000ms	7.286s	0.006	0.800ms	16.000ms
React	0.787s	0.002	0.900ms	32.000ms	8.394s	0.002	1.000ms	16.000ms
Next	0.448s	0.002	1.100ms	32.000ms	10.849s	0.002	1.300ms	40.000ms
Gatsby	0.084s	0.002	0.900ms	16.000ms	2.127s	0.002	0.900ms	40.000ms

Tabela 3.1: Rezultatele generate de Web Vitals

Conform rezultatelor obţinute, aplicaţia construită cu Gatsby este cea mai eficientă din punct de vedere al randării, urmată de aplicaţia vanilla JavaScript. Aplicaţiile construite cu React şi Next.js au rezultate similare, dar performanţele acestora sunt mai slabe. Datorită optimizărilor aduse, în special prin utilizarea Gatsby *StaticImage* şi a *prefetch*-ului, aplicaţia Gatsby este cea mai rapidă din punct de vedere al timpului de randare, atât pe conexiunea normală, cât şi pe cea lentă.

3.4.2 Lighthouse

Cu ajutorul acestui tool, care este prezent în browserele Google Chrome şi Mozilla Firefox, au fost generate rapoartele de performanţă pentru aplicațiile dezvoltate. Un raport Lighthouse este compus din mai multe secțiuni, fiecare dintre acestea având un scor şi o descriere a metricilor evaluate. Scorul este calculat pe baza unor ponderi, care sunt diferite pentru fiecare secțiune. Ponderile sunt calculate pe baza unor studii de utilizare, care au fost efectuate de Google.

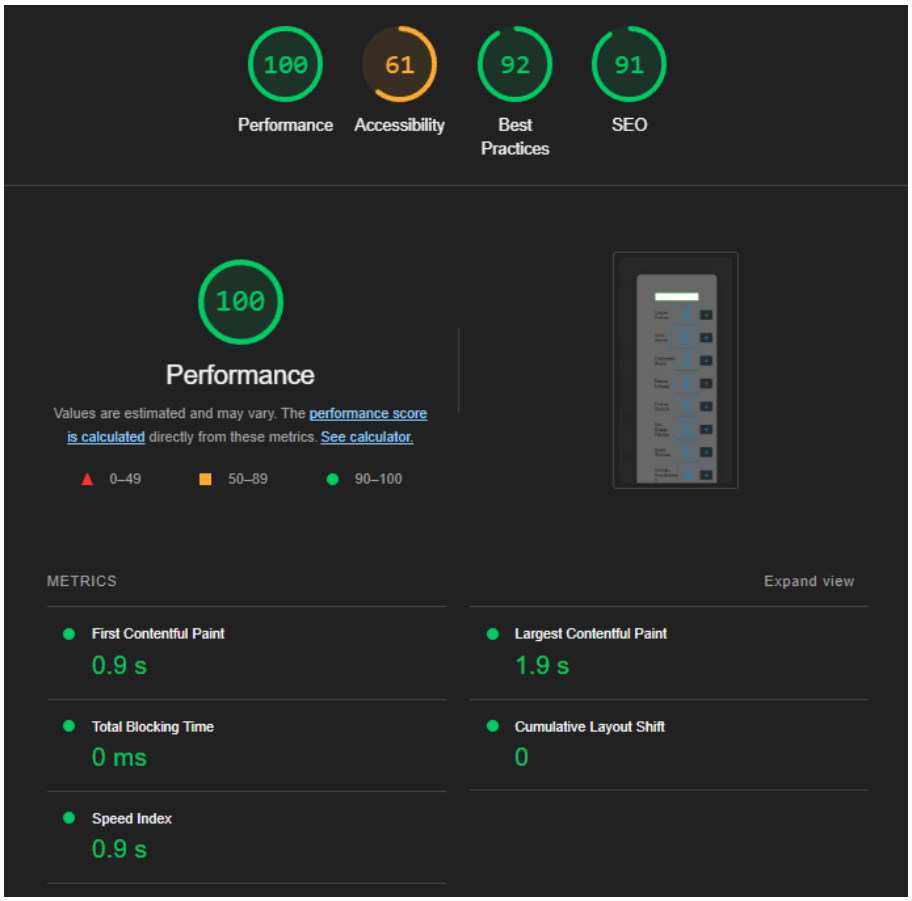

Figura 3.6: Exemplu raport generat Lighthouse

Pentru fiecare dintre aplicaţiile dezvoltate au fost generate rapoarte Lighthouse, din case s-au extras cele mai importante metrici, relevante pentru performanţa aplicaţiei (***FCP*** *- First Contentful Paint,* ***LCP*** *- Largest Contentful Paint,* ***TBT*** *- Total Blocking Time,* ***CLS*** *- Cumulative Layout Shift,* ***SI*** *- Speed Index)*, experienței utilizatorului (**Accessibillity, Best practices**) şi optimizarea pentru motoarele de căutare (**SEO**). Rezultatele sunt prezentate în tabelul următor:

Aplicaţie	**Performanţă**					**Alte metrici**		
	FCP	**LCP**	**TBT**	**CLS**	**SI**	**Accessib.**	**Best Practices**	**SEO**
JavaScript	0.3s	0.6s	0	0	0.9s	65	100	89
React	0.4s	0.5s	0	0	0.8s	83	100	100
Next	0.3s	0.4s	0	0	0.4s	92	92	100
Gatsby	0.2s	0.2s	0	0	0.3s	82	92	80

Tabela 3.2: Rezultatele rapoartelor Lighthouse

Conform rezultatelor obţinute în urma testării, aplicaţiile dezvoltate cu Next.js şi Gatsby.js au obţinut cele mai bune rezultate. Aceste scoruri sunt influenţate atât de optimizările prezente în aceste framework-uri, cât şi de hosting-ul ales, care oferă un mediu de rulare optimizat pentru aplicaţiile dezvoltate cu Next.js şi Gatsby.

Pe baza metricilor analizate, aplicația "Gatsby" demonstrează o performanță mai bună în comparație cu celelalte aplicații. Aceasta obține constant valori mai mici pentru FCP și LCP, ce rezultă într-o randare mai rapidă a conținutului.

3.4.3 PageSpeed Insights

Evaluează pagina atât pe dispozitive mobile, cât și pe desktop, atribuie un scor în funcție de performanța paginii și se oferă recomandări pentru îmbunătățire. Acest tool ia în considerare mai mulți factori care pot afecta performanța unei pagini, cum ar fi timpul de răspuns al serverului, cache-ul resurselor, optimizarea imaginilor, minificarea JavaScript și CSS. De asemenea, acesta ia în considerare utilizarea celor mai bune practici și a ghidurilor de performanță web, cum ar fi reducerea resurselor care blochează randarea, optimizarea căii critice de randare și folosirea cache-ului browserului.

În urma testării aplicaţiilor dezvoltate, au fost obţinute următoarele rezultate:

Desktop	Performanţă					Alte metrici		
	FCP	LCP	TBT	CLS	SI	Accessib.	Best Practices	SEO
JavaScript	0.3s	0.6s	0	0	0.9s	65	100	89
React	0.2s	0.3s	0	0	1.1s	83	100	100
Next	0.2s	0.3s	0	0	0.7s	92	92	100
Gatsby	0.2s	0.2s	0	0	0.6s	85	92	80

Tabela 3.3: Rezultatele raport PageSpeed Insights - Desktop

Mobile	Performanţă					Alte metrici		
	FCP	LCP	TBT	CLS	SI	Accessib.	Best Practices	SEO
JavaScript	0.8s	2.0s	80ms	0.102	2.2s	65	100	91
React	0.9s	1.3s	250ms	0	2.3s	83	100	100
Next	0.9s	0.9s	190ms	0	1.5s	92	92	100
Gatsby	0.8s	0.8s	50ms	0	1.7s	85	92	83

Tabela 3.4: Rezultatele raport PageSpeed Insights - Mobile

3.5 Analiza rezultatelor

3.5.1 Uşurinţa implementării

Din punct de vedere al utilizării, vanilla JS este cel mai proeminent, deoarece nu necesită dependențe sau framework-uri suplimentare. Pentru a executa un cod JavaScript este destulă adăugarea unui tag *<script>* în pagina HTML:

```
<script>
   const exemplu = "Hello World!";
</script>
```

Listing 3.1: Exemplu de adăugare a unui script în pagina HTML

sau adaugarea unui fişier JavaScript separat, care poate fi inclus în pagina HTML cu ajutorul aceluiaşi tag, dar cu specificarea locaţiei fişierului:

```
<script src="path/to/file.js"></script>
```

Listing 3.2: Exemplu de adăugare a unui script dintr-un fişier extern

Aplicaţiile create cu ajutorul framework-urilor React, necesită paşi suplimentari pentru configurarea inițială a proiectului folosind *create-react-app* sau configurarea manuală a unui mediu de dezvoltare. De asemenea, este necesară instalarea dependențelor proiectului, care pot fi gestionate cu ajutorul unui package manager, cum ar fi *npm* sau *yarn*. React utilizează JSX - o sintaxă care combină JavaScript cu HTML, care necesită un preprocesor pentru a fi transformat în JavaScript valid.

```
--- cod JSX
const numbers = [1, 2, 3, 4, 5];
const list = (
   <ul>
      {numbers.map((number) => (
      <li key={number}>{number}</li>
      ))}
   </ul>
);
--- cod JavaScript
const numbers = [1, 2, 3, 4, 5];
const list = React.createElement("ul",null,
   numbers.map((number) =>
      React.createElement("li", { key: number }, number)
   )
);
```

Listing 3.3: Comparare sintaxă JSX şi JavaScript

JSX îmbunătățește lizibilitatea codului prin utilizarea de cod similar HTML în cadrul JavaScript. Reprezentarea vizuală a structurii codului sporește ușurința de întreținere și debugging-ul.

Promovează o abordare modulară și bazată pe componente, care poate fi reutilizate, facilitând crearea și gestionarea elementelor de interfață. Componentele pot fi scrise direct în cadrul codului JavaScript, permițând o organizare eficientă a codului și reutilizabilitate.

```
function Welcome(props) {
   return <h1>Hello, {props.name}</h1>;
}
```

Listing 3.4: Exemplu de componentă React

Next.js şi Gatsby sunt framework-uri care se bazează pe React, oferind un mediu de dezvoltare mai uşor, cu un set de funcționalități predefinite, cum ar fi server-side rendering, generarea de pagini statice, optimizarea imaginilor:

```
--- Next.js
import Image from 'next/image'
function HomeImage() {
   return (
         <Image
            src="https://picsum.photos/1920/1080?grayscale"
            alt="Background picture"
         />
   )
}
export default HomeImage


--- Gatsby
import React from "react"
import { StaticImage } from "gatsby-plugin-image"
const HomeImage = () => {
   return (
         <StaticImage
            alt="Background picture"
            src="https://picsum.photos/1920/1080?grayscale"
         />
   )
}
```

Listing 3.5: Exemplu de utilizare a unei imagini optimizate automat în Next.js şi Gatsby

Acestea sunt utile pentru crearea de aplicații web complexe, care necesită o structură bine definită și o arhitectură scalabilă, însă necesită o perioadă de timp mai mare pentru a fi învățate şi implementate.

3.5.2 Performanța

În urma analizei rezultatelor obţinute cu ajutorul Web Vitals, rapoartelor generate de Lighthouse şi PageSpeed Insights se poate observa faptul că aplicaţia construită cu Gatsby este cea mai performantă, datorită optimizării automate a imaginilor şi a generării de pagini statice. În cazul aplicaţiei construite cu Next.js, performanţa este influenţată de modul de implementare, deoarece nu există o optimizare automată a imaginilor, iar paginile sunt generate dinamic. Aplicaţia construită cu vanilla JS este cea mai puţin performantă, deoarece nu există o optimizare a imaginilor, iar codul este scris într-un singur fişier, ceea ce duce la un timp de încărcare mai mare.

Pe desktop, Gatsby a reușit să ofere o performanță deosebită, cu valori mici pentru FCP (0.2s), LCP (0.2s) și o lipsă totală a timpului de blocare semnificativ (TBT = 0). Aceste rezultate indică o experiență rapidă și fluidă pentru utilizatorii care accesează aplicația Gatsby pe desktop.

În ceea ce privește dispozitivele mobile, Gatsby continuă să ofere o performanță excelentă, obținând valori competitive pentru FCP (0.8s) și LCP (0.8s). Deși valorile sunt ușor mai ridicate decât în cazul desktopului, Gatsby încă se menține în fruntea clasamentului, asigurând o încărcare rapidă și o experiență plăcută pe dispozitivele mobile.

3.5.3 Dimensiunea aplicaţiei

Dimensiunea aplicaţiei este un alt factor important care influenţează performanţa. Cu cât mai mare este dimensiunea aplicaţiei, cu atât mai multe resurse trebuie descărcate de către browser, ceea ce duce la un timp de încărcare mai mare. Dimensiunea aplicaţiei este influenţată de framework-ul folosit, bibliotecile suplimentare, dependenţele aplicaţiei şi de modul de implementare. În urma analizei fişierelor descărcate de către browser, în tab-ul de Network din DevTools, am obţinut următoarele rezultate:

- Vanilla JavaScript $\sim$ 198.1 KB
- React $\sim$ 255.5 KB
- Next.js $\sim$ 202.2 KB (*fără optimizarea imaginilor*) şi 138.9 KB (*cu optimizarea imaginilor*)
- Gatsby $\sim$ 205.3 KB (*fără optimizarea imaginilor*) şi 124.7 KB (*cu optimizarea imaginilor*)

Framework-urile Next.js și Gatsby au adus o îmbunătățire semnificativă în ceea ce privește optimizarea imaginilor, ceea ce a dus la o dimensiune mai mică a aplicațiilor în comparație cu utilizarea doar a Vanilla JS și React. Deși framework-urile includ mai multe librării suplimentare, care pot crește dimensiunea generală a aplicației, funcționalitățile pe care le oferă acestea sunt optimizate pentru a reduce dimensiunea finală a aplicației.

3.5.4 SEO

Vanilla JS nu oferă suport direct pentru optimizarea pentru motoarele de căutare. Implementarea SEO trebuie făcută manual, inclusiv gestionarea meta tag-urilor, structurarea conținutului și gestionarea URL-urilor.

React oferă suport pentru optimizarea pentru motoarele de căutare, prin intermediul unor biblioteci dedicate şi necesită atenție suplimentară și cunoștințe SEO pentru a implementa aceste optimizări. Prin utilizarea React Helmet sau a altor biblioteci similare, este posibiă adăugarea meta tag-urilor relevante.

Next.js are abilități de optimizare pentru SEO încorporate. Oferă suport nativ pentru server-side rendering (SSR) - paginile vor fi pregătite și livrate către motoarele de căutare cu conținutul complet generat, facilitând astfel indexarea și clasificarea mai bună în rezultatele căutării. Next.js gestionează automat meta tag-urile dinamice pentru fiecare pagină și, de asemenea, facilitează generarea paginilor statice, care pot fi extrem de benefice pentru SEO.

Gatsby este un framework care pune accent pe performanță și optimizare pentru SEO. Prin generarea paginilor statice în avans, Gatsby oferă un avantaj semnificativ în ceea ce privește viteza de încărcare a paginilor și indexarea lor de către motoarele de căutare. Gatsby are, de asemenea, suport nativ pentru gestionarea meta tag-urilor și structurarea conținutului pentru a obține rezultate optime în SEO. De asemenea, Gatsby beneficiază de integrarea cu GraphQL, permițând încărcarea eficientă a datelor și gestionarea optimă a conținutului pentru SEO.

Capitolul 4

Concluzii

Această lucrare oferă o perspectivă asupra tehnologiilor de randare și a impactului lor asupra aplicațiilor web din punct de vedere al performanței, optimizării şi experienței utilizatorilor. Este important de menționat faptul că performanța unei aplicații web nu se rezumă doar la tehnologia de randare utilizată, ci și la practicile de optimizare și îmbunătățire continuă. Utilizarea instrumentelor precum Lighthouse și PageSpeed Insights poate oferi informaţii valoroase pentru identificarea punctelor slabe și îmbunătățirea performanței unei aplicații web.

Alegerea tehnologiei de randare depinde de cerințele specifice ale aplicației proiectate. SSR *(Server Side Rendering)* oferă avantaje din punct de vedere al SEO, datorită faptului că conţinutul HTML este generat direct pe server şi este furnizat către motoarele de căutare, îmbunătățirea performanței încărcării inițiale a paginii, dar are dezavantaje în ceea ce privește interactivitatea, deoarece întreaga pagină trebuie reîncărcată pentru a actualiza conținutul.

CSR *(Client Side Rendering)*, pe de altă parte, oferă o experiență mai interactivă, deoarece conținutul este generat în browser, dar are dezavantaje în ceea ce privește performanța încărcării inițiale a paginii, deoarece întregul JavaScript trebuie descărcat și executat înainte ca pagina să poată fi afișată.

SSG *(Static Server Generation)* este o alternativă la SSR, care oferă avantajele SEO și performanța încărcării inițiale a paginii, dar nu oferă interactivitatea oferită de CSR.

În ceea ce privește tehnologiile de randare, React oferă o experiență de dezvoltare mai bună, datorită faptului că este mai ușor de învățat și de utilizat, oferind unelte *precum create-react-app*, care oferă un mediu de dezvoltare preconfigurat şi sintaxă cunoscută,

deoarece este bazat pe JavaScript. Beneficiile care le oferă React sunt cel mai evidenţiate în aplicațiile complexe cu interfețe de utilizator dinamice și actualizări frecvente, unde virtual DOM și mecanismele eficiente de randare oferă un UX mai bun.

Vanilla JS, pe de altă parte, oferă o performanță mai bună, deoarece nu necesită biblioteci suplimentare, dar are o curba de învățare mai mare și necesită mai mult cod pentru a realiza aceleași funcționalități.

Cu toate acestea, este important de menționat că în aplicațiile mai simple sau în cazurile în care interactivitatea nu este un aspect major, JavaScript simplu poate fi mai performant datorită naturii sale. Vanilla JavaScript evită costurile suplimentare ale unui framework precum React și poate fi optimizat pentru cazuri de utilizare specifice, rezultând într-o execuție mai rapidă.

Fiecare dintre metodele abordate are avantajele şi dezavantajele sale. Vanilla JS este cel mai simplu de implementat, deoarece nu necesită dependenţe sau biblioteci suplimentare. React este un framework popular, care oferă o abordare mai structurată a dezvoltării aplicaţiilor web, dar necesită un timp mai mare de implementare şi configurare. Next.js şi Gatsby sunt framework-uri construite pe React, care oferă un set de funcţionalităţi suplimentare, ce pot fi utile în dezvoltarea unei aplicaţii web, dar necesită cunoştinţe mai avansate de React şi a specificaţiilor interne ale acestor framework-uri.

4.1 Rezultatele obținute

În urma analizei perfomanţei aplicaţiilor construite, aplicaţia dezvoltată cu ajutorul framework-ului Gatsby este cea mai performantă, deoarece utilizează paradigma SSG (Static Server Generation), care este caracterizată prin generarea conţinutului HTML pe server, înainte de a fi livrat către utilizatori. Această abordare oferă o performanţă sporită în ceea ce priveşte timpul de încărcare a paginii, deoarece nu necesită randarea conţinutului în browser, ci doar livrarea acestuia către utilizator. De asemenea, datorită optimizărilor oferite de Gatsby, imaginile sunt comprimate şi redimensionate, iar codul este minificat, ceea ce duce la o performanţă sporită a aplicaţiei.

Aplicaţiile ce au fost construite cu ajutorul React, Next.js şi Gatsby au o dimensiune mai mare, deoarece necesită biblioteci suplimentare pentru a funcţiona şi trebuie configurate pentru a putea fi hostate pe web, fapt ce poate rezulta într-o experienţă de dezvoltare mai complexă, însă aceasta este compensată de beneficiile oferite de aceste

framework-uri.

În aplicația ce utilizeaza vanilla JS, manipulările DOM sunt realizate direct, iar în cazul unei aplicaţii simple această abordare directă este mai eficientă decât utilizarea React şi virtual DOM, ce necesită paşi suplimentari pentru actualizarea conţinutului.

Odată cu scalarea aplicaţiei, framework-urile devin mai performante, iar codul mai uşor de menţinut, datorită arhitecturii predefinite, bazate pe componente reutilizabile şi alte unelte de dezvoltare precum hook-urile, ce simplifică implementarea, actualizarea şi scalarea optimă a funcţionalităţilor necesare.

4.2 Direcții viitoare de cercetare

În timp ce studiul actual s-a concentrat pe compararea performanței JavaScript-ului simplu (Vanilla), React, Next.js și Gatsby într-o aplicație web relativ simplă, există mai multe direcții pentru cercetări ulterioare, care pot extinde înțelegerea asupra acestor tehnologii de randare în scenarii mai complexe. Prin investigarea acestor direcții, putem obține o perspectivă mai profundă asupra punctelor forte și limitărilor fiecărei tehnologii și a performanței acestora în aplicații web complexe, cu conținut dinamic şi actualizări frecvente ale componentelor.

Prin creșterea complexității și a dimensiunii aplicației, inclusiv adăugarea mai multor pagini, va fi posibiă evaluarea modului în care fiecare tehnologie gestionează sarcina de lucru suplimentară. Această analiză ar ajuta la determinarea dacă anumite tehnologii sunt mai potrivite pentru aplicații de mari dimensiuni și ar dezvălui eventuale restricții de performanță care pot apărea.

Acest studiu s-a axat pe utilizarea vanilla JS, React, Next.js și Gatsby, însă există numeroase framework-uri şi tehnologii de randare. Cercetările ulterioare ar putea extinde comparația pentru a include alte framework-uri sau biblioteci, cum ar fi Angular, Vue.js sau Svelte şi tehnologii de randare precum ISR (*Incremental Static Regeneration*), care este o variantă îmbunătăţită a SSG, în care paginile statice se pot regenera pe server la cerere, fără necesitatea de rebuild a întregii aplicaţii.

Prin includerea unui spectru mai larg de tehnologii de randare, putem obține o perspectivă mai largă asupra performanței acestora și a modului în care se compară cu cele evaluate în acest studiu.

Bibliografie

[1] Mike Evans. ***"The Evolution of the Web - From Web 1.0 to Web 4.0"*** (2008)

[2] Tim O'Reilly. ***"What Is Web 2.0"*** - `https://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html` (2005)

[3] Sean B. Palmer. ***"The Semantic Web: An Introduction"*** (2001)

[4] Dennis Sheppard. ***"Introduction to Progressive Web Apps"*** (2017)

[5] Jesse James Garrett. ***"Ajax: A New Approach to Web Applications"*** (2007)

[6] A. Javeed. ***"Performance Optimization Techniques for ReactJS"*** (2019)

[7] German Cocca. ***"Rendering Patterns for Web Apps - Server-Side, Client-Side, and SSG Explained"*** - `https://www.freecodecamp.org/news/rendering-patterns/` (2023)

[8] ***"Choose the Rendering Method (SPA vs SSR vs SSG)"*** - `https://dev.to/crunchstack/choose-the-rendering-method-spa-vs-ssr-vs-ssg-121f` (2021)

[9] Philip Walton, Barry Pollard. ***"Largest Contentful Paint (LCP)"*** - `https://web.dev/lcp/` (2019)

[10] Philip Walton, Milica Mihajlija. ***"Cumulative Layout Shift (CLS)"*** - `https://web.dev/cls/` (2019)

[11] Philip Walton. ***"First Input Delay (FID)"*** - `https://web.dev/fid/` (2019)

[12] Jeremy Wagner. ***"Interaction to Next Paint (INP)"*** - `https://web.dev/inp/` (2019)

[13] Barry Pollard, Michal Mocny, Rick Viscomi, Brendan Kenny. ***"Using the Web Vitals extension to debug Core Web Vitals issues"*** - https://web.dev/debug-cwvs-with-web-vitals-extension/(2023)

[14] Tushar Pol. ***"Google Lighthouse: What It Is & How to Use It"*** - https://www.semrush.com/blog/google-lighthouse/(2023)

[15] Google documentation. ***"About PageSpeed Insights"*** - https://developers.google.com/speed/docs/insights/v5/about

[16] John Larsen. ***"React Hooks in Action: With Suspense and Concurrent Mode"***, p.[54,96,121] (2021)

[17] Alex Grigoryan. ***"The Benefits of Server Side Rendering Over Client Side Rendering"*** - https://medium.com/walmartlabs/the-benefits-of-server-side-rendering-over-client-side-rendering-5d07ff2cefe8 (2017)

[18] Tim Berners-Lee, Daniel Connolly. ***"Hypertext markup language (html)"*** (1993)

[19] Sergey Laptick. ***"Client Side vs Server Side UI Rendering. Advantages and Disadvantages"*** - https://blog.webix.com/client-side-vs-server-side-ui-rendering/ (2017)

[20] Anna Monus. ***"What Is Server-side Rendering And How Does It Improve Site Speed?"*** - https://www.debugbear.com/blog/server-side-rendering (2019)

[21] Google Research - https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks-load-time-vs-bounce/ (2017)

Anexa A

Glosar

A.1 Acronime

SSR	Server side rendering
CSR	Client side rendering
SSG	Static server generation
UX	User experience
UI	User interface
LCP	Largest contentful paint
FID	First input delay
CLS	Comulative layout shift
SEO	Search engine optimization
JS	JavaScript

Tabela A.1: Tabelă de acronime