



UNIVERSITATEA DE VEST DIN TIMIȘOARA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

DOMENIUL DE STUDII:
INFORMATICĂ APLICATĂ

LUCRARE DE DISERTAȚIE

COORDONATOR:

Conf. Dr. Cristina
MÎNDRUȚĂ

ABSOLVENT:

Nicolae SAVILENCU

Timișoara
2023

UNIVERSITATEA DE VEST DIN TIMIȘOARA

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

Tehnologii de randare a informațiilor în aplicațiile web moderne

COORDONATOR:

Conf. Dr. Cristina

MÎNDRUȚĂ

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 we had 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, Vue, 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*), etc.

Dissertation explores the impact of rendering technologies on development workflows, examines the challenges associated with 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 multiple 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	ii
Starea de artă	iv
1 Introducere	1
1.1 Motivație și scop	1
1.2 Contextul lucrării	2
1.2.1 Web 1.0	2
1.2.2 Web 2.0	4
1.2.3 Web 3.0	5
1.2.4 Paradigmele actuale de randare	7
1.2.5 Consecințele evoluției web	12
1.3 Evoluția tehnologiilor	13
1.4 De ce comparăm server side rendering cu client side rendering?	14
1.4.1 Lipsa literaturii	14
1.4.2 Dificultate pentru utilizatori	15
2 Tehnologii de randare	16
2.1 Server-side rendering	16
2.2 Static rendering	17
2.3 Client-side rendering	18
2.4 Rehydration	19
3 Aplicația practică	21
3.1 Metricile de performanță	21
3.2 Plugin-uri, extensii și aplicații pentru măsurarea performanței	24
3.3 Aplicația practică	25

3.3.1	Metodologie	25
3.3.2	Vanilla JavaScript	26
3.3.3	React framework	27
3.4	Analiza performanței	27
Bibliography		27
A Glosar		29
A.1	Acronime	29

Listă de figuri

1	Compararea între mai multe tipuri de randare a informațiilor. [11]	v
1.1	Arhitectura stratificată pentru semantic web	6
1.2	Server side rendering	8
1.3	Client side rendering. Paradigma SPA	9
1.4	Client side rendering. Paradigma SPA	10
1.5	Static site generation	10
1.6	Hidratarea parțială	11
1.7	Arhitectura pe insule	12
2.1	Server-side rendering	17
2.2	Static rendering	18
2.3	Client-side rendering	18
2.4	Tehnologia de hidratare	19
2.5	Dublicarea de cod la hidratare	20
3.1	Aplicația practica - vanilla JavaScript	26

Rendering-ul informațiilor a evoluat semnificativ pe parcursul timpului, în conformitate cu evoluția tehnologiilor web și a cerințelor utilizatorilor.

La început, majoritatea conținutului era generat pe server și 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 aplicații web cu interacțiuni complexe și randare dinamică a conținutului. Cu toate acestea, performanța aplicațiilor web bazate exclusiv pe randare client-side a putut fi limitată în condiții 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 și client-side. [5]

În prezent, tehnologiile de randare se bazează pe framework-uri și biblioteci moderne, cum ar fi React, Angular și Vue, care oferă unelte puternice pentru a construi aplicații web performante și cu o experiență utilizator remarcabilă. În plus, tehnologiile de randare moderne permit integrarea cu alte tehnologii precum WebAssembly și Web Workers, care au ca scop îmbunătățirea performanței aplicațiilor web. De asemenea, tehnologiile de randare permit utilizarea de animații și interacțiuni complexe, care rezultă într-un user experience sporit al aplicației.

Evoluția tehnologiei web și a cerințelor utilizatorilor continuă, iar randarea informațiilor este un domeniu în continuă schimbare și dezvoltare. În etapa curentă de dezvoltare a aplicațiilor web, predomină utilizarea largă a JavaScript și bibliotecilor JavaScript, cum ar fi React și Vue.js.

React, dezvoltat de Facebook, a devenit una dintre tehnologiile cele mai populare pentru construirea de aplicații web, datorită abilității sale de a oferi o experiență de randare eficientă și de înaltă performanță. Vue.js, dezvoltat de comunitate, se concentrează pe simplitate și ușurință de utilizare, fiind o alegere populară pentru proiecte mai mici sau pentru dezvoltatorii care încearcă să învețe tehnologii moderne de randare a informațiilor.

Există mai multe tipuri de randare a conținutului, cele mai populare fiind randarea *server-side*, randarea *client-side* și randarea *hibridă*: [10]

1. **Randarea server-side:** Procesarea datelor și generarea HTML se realizează pe server. Server-ul returnează apoi HTML-ul generat către browser, care îl afișează utilizatorului. Această metodă de randare este utilizată în mod traditional în aplicațiile web vechi. Avantajul acestei metode este că se poate efectua o procesare mai puternică a datelor pe server și se poate oferi o experiență utilizator

consistentă chiar și în condiții de conexiune lentă la internet.

2. **Randarea client-side:** Browser-ul primește date brute și le procesează prin intermediul unui script (cum ar fi JavaScript) pentru a genera HTML. Această metodă de randare permite o flexibilitate mai mare în prezentarea conținutului și oferă posibilitatea de a construi interacțiuni complexe cu utilizatorul, cum ar fi formulare dinamice și componente grafice interactive. Dezavantajul acestei metode este că necesită mai multă putere de procesare a clientului și poate fi mai puțin performantă în condiții de conexiune lentă la internet.
3. **Randarea hibridă:** Această metodă combină avantajele randării server-side și client-side. Datele sunt procesate inițial pe server și se returnează un HTML minim către browser. Ulterior, browser-ul folosește script-uri pentru a adăuga interacțiuni dinamice și a actualiza conținutul fără a fi necesară o reîncărcare completă a paginii.


	Server				Browser
					
	Server Rendering	"Static SSR"	SSR with (Re)hydration	CSR with Prerendering	Full CSR
Overview:	An application where input is navigation requests and the output is HTML in response to them.	Built as a Single Page App, but all pages prerendered to static HTML as a build step, and the JS is removed .	Built as a Single Page App. The server prerenders pages, but the full app is also booted on the client.	A Single Page App, where the initial shell/skeleton is prerendered to static HTML at build time.	A Single Page App. All logic, rendering and booting is done on the client. HTML is essentially just script & style tags.
Authoring:	Entirely server-side (request-response, HTML)	Built as if client-side (components, DOM*, fetch)	Built as client-side	Client-side	Client-side
Rendering:	Dynamic HTML	Static HTML	Dynamic HTML and JS/DOM	Partial static HTML, then JS/DOM	Entirely JS/DOM
Server role:	Controls all aspects. (thin client)	Delivers static HTML	Renders pages (navigation requests)	Delivers static HTML	Delivers static HTML
Pros:	👍 TTI = FCP 👍 Fully streaming	👍 Fast TTFB 👍 TTI = FCP 👍 Fully streaming	👍 Flexible	👍 Flexible 👍 Fast TTFB	👍 Flexible 👍 Fast TTFB
Cons:	👎 Slow TTFB 👎 Inflexible	👎 Inflexible 👎 Leads to hydration	👎 Slow TTFB 👎 TTI >>> FCP 👎 Usually buffered	👎 TTI > FCP 👎 Limited streaming	👎 TTI >>> FCP 👎 No streaming
Scales via:	Infra size / cost	build/deploy size	Infra size & JS size	JS size	JS size
Examples:	Gmail HTML, Hacker News	Docusaurus, Netflix*	Next.js, Razzle, etc	Gatsby, Vuepress, etc	Most apps

Figura 1: Compararea între mai multe tipuri de randare a informațiilor. [11]

În ceea ce privește performanța, tehnologiile moderne de randare a informațiilor au avansat mult în optimizarea vitezei de randare și a eficienței resurselor. Una dintre abordările moderne de optimizare a performanței 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 performanța aplicației. De asemenea, tehnologiile de randare permit utilizarea de tehnici de lazy loading, care încarcă conținutul doar atunci când este necesar, îmbunătățind astfel performanța aplicației și experiența utilizatorului. Alegerea metodei potrivite de randare depinde de cerințele specifice ale proiectului și poate include considerații legate de performanța, flexibilitatea și compatibilitatea cu echipamentele utilizatorilor. Este important să se înțeleagă avantajele și dezavantajele fiecărei metode și să se facă o evaluare corespunzătoare a acestora înainte de a lua o decizie.

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, în intermediu dezvoltării unei aplicații simple, cât și în intermediul unei aplicații relativ complexe, se vor identifica cele mai efective pattern-uri de dezvoltare.

Motivația alegerii acestei teme de disertație derivă din mai mulți factori. În primul rând, progresele tehnologice rapide în dezvoltarea web a dus la apariția mai multor paradigme de dezvoltare, iar ținerea pasului cu acestea a devenit o provocare pentru toți dezvoltatorii web. 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 un user experience mai bun și o performanță sporită.

În cele din urmă, această lucrare de disertație își propune să ofere informații despre 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 tehnosocial cu care interacționează oamenii prin intermediul rețelelor tehnologice. Noțiunea de sistem tehnosocial se referă la un sistem care îmbunătățește percepția umană, comunicarea și cooperarea. Percepția este condiția prealabilă necesară pentru a comunica și condiția prealabilă pentru a 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) și 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 concepția 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 pasivă.

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 monodirecțional, însemna că organizațiile împărtășeau informații precum broșuri, 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.

Browsersle erau foarte rudimentare și site-urile web erau mici. De aceea era destul să se creeze un fișier HTML static care ar conține toate datele necesare și designul, care nu

ar fi fost niciodată actualizate automat. Așa arată un fișier HTML:

```
!DOCTYPE html>

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

Mai târziu, când browser-ele au dezvoltat capacități vizuale, a apărut necesitatea de stilizare. Håkon W Lie a propus o prima versiune a CSS (Cascading Style Sheet), care este folosit și până în prezent. Un style sheet arată astfel:

```
body {
  background-color: black;
}

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

Caracteristici:

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

Limitații:

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

1.2.2 Web 2.0

A fost definit de Dale Dougherty în 2004. Tim O'Reilly a definit ulterior Web 2.0 astfel [66]: "Web 2.0 este revoluția afacerilor în industria calculatorului, cauzată de mutarea către internet ca platformă și încercarea de a înțelege regulile pentru succes pe această nouă platformă. Printre aceste reguli se numără următoarea: construirea de aplicații care să valorifice efectele de rețea pentru a deveni mai bune pe măsură ce mai mulți oameni le folosesc."

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 interacțiunea serverului web cu orice proces local. Mai târziu, limbi precum Perl, Java, PHP și ASP (și altele) au devenit populare pentru programare pe partea de server.

Site-urile web care devin mai bogate în funcții au nevoie de mai multă flexibilitate și 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, conținutul poate fi schimbat de către utilizatorii finali. Tehnologia cel mai utilizată pentru acest scop este Javascript. Când funcțiile client-side au devenit și mai importante, au fost concepute framework-uri pentru renderizarea (și controlarea) paginilor client-side. Cele mai cunoscute framework-uri sunt: Ember, Angular și 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

Limitații:

- 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

Web 3.0 este unul dintre subiectele moderne și evolutive asociate cu următoarele inițiative ale web 2.0. A fost pentru prima dată definit de John mark de la New York Times, în 2006. Versiunea 3.0 poate fi numită drept web semantic. Ideea ce se află la bază este de a defini structurile de date și de a le lega pentru o automatizare, întregare 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 user experience și să ajute la organizarea colaborării în rețelele sociale.

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 acceptă baze de date masive, la nivel mondial și arhitectură orientată web, care anterior era descrisă ca o rețea de documente.

Versiunea 3.0 este cunoscută sub numele de web semantic. Acesta a fost conceput de Tim Berners-Lee și reprezintă o mișcare colaboraționistă condusă de consorțiumul World Wide Web. Conform W3C [??] "Web-ul semantic oferă un framework comun care permite partajarea și reutilizarea datelor în limitele aplicațiilor, întreprinderilor și comunităților".

Tim Berners-lee a definit web-ul semantic ca o simbioză a tuturor paginilor web și informațiilor ce le conțin, într-o carte ce ar reprezenta o bază de date globală. Pentru aceasta el a propus o arhitectură stratificată pentru web semantic

Următorul pas în dezvoltarea web-ului este crearea aplicațiilor web autonome, care pot fi, de asemenea, utilizate fără conexiune la internet. Acestea sunt numite Aplicații Web Progressive (PWA). Crearea acestor aplicații nu este posibilă fără a avea control complet la nivelul clientului. Renderizarea pe partea de server nu este posibilă și se folosesc framework-uri complete pentru front-end pentru a construi acest fel de aplicații. Comunicarea se gestionează prin intermediul API-urilor.[2]

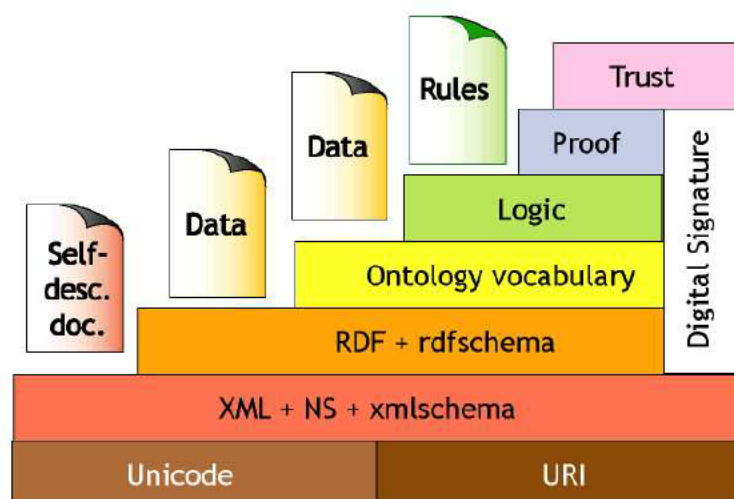


Figura 1.1: Arhitectura stratificată pentru semantic web

Caracteristici:

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

Provocări:

- *Vastitate* - conține miliarde de pagini, ce duce la redundanță și dublicate în date
- *Vaguitatea* - interogările imprecise 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
- *Inconsistența* - contradicții logice ce vor apărea inevitabil în timpul dezvoltării datorită diferențelor în interpretare a datelor
- *Înșelăciune* - momentul în care sursa de informații induce în eroare intenționat consumatorul de informații

1.2.4 Paradigmele actuale de randare

Rendering-ul este procesul de transformare a datelor și a codului în HTML care poate fi văzut de către utilizatorul final. Acest proces poate fi realizat pe server sau în browser și poate fi realizat integral sau parțial, iar toate acestea au compromisuri în ceea ce privește experiența utilizatorului, performanța și experiența dezvoltatorului. Paradigma de randare originală și cea mai de bază este un site web static: în această paradigmă de randare, toate paginile web sunt compuse în avans, apoi încărcate ca fișiere statice într-un recipient de stocare undeva în cloud și sunt legate de un nume de domeniu.

Acest lucru funcționează foarte bine chiar și în lumea de astăzi și există cadre precum Hugo, 11t și Jekyll care vă pot ajuta să le construiți în mod programatic. Dezavantajul este că nu sunt foarte bune pentru site-urile web în care datele se schimbă des, așa că sunt potrivite doar pentru site-urile foarte simple care nu necesită o tonă de interactivitate sau date dinamice. În cele din urmă, site-urile web trebuiau să fie mai dinamice, ceea ce ne-a adus aplicații multi-pagină în care HTML-ul și datele sunt puse împreună în mod dinamic pe un server de fiecare dată când vine o cerere de la un browser. Acest lucru înseamnă că aspectul site-ului web se poate schimba ori de câte ori se schimbă datele de bază. [7]

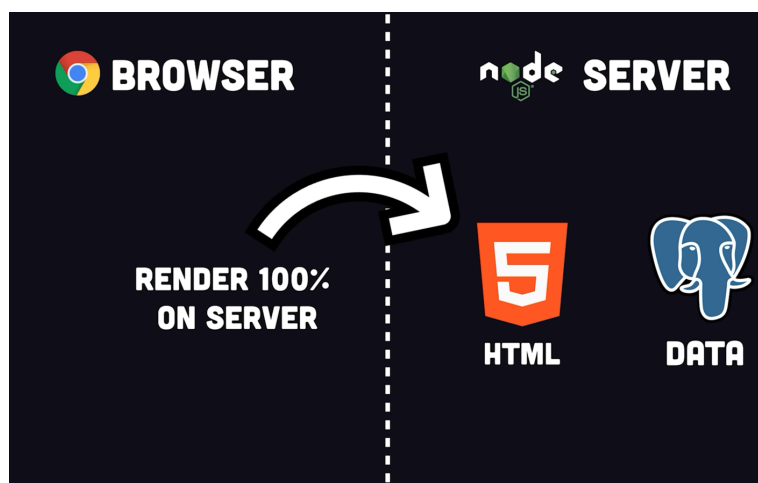


Figura 1.2: Server side rendering

Multe dintre cele mai mari aplicații web din ziua de azi încă folosesc această abordare, cum ar fi, de exemplu, amazon.com: de fiecare dată când faceți clic pe un link, obțineți o nouă pagină generată dinamic de pe serverele lor. În plus, există multe cadre populare pentru crearea de aplicații multi-pagină, cum ar fi Ruby on Rails, Django și Laravel, precum și sisteme de gestionare a conținutului, cum ar fi WordPress. Această abordare a funcționat foarte bine până la apariția iPhone. Atunci, oamenii au început să realizeze că reîncărcarea întregii pagini la fiecare URL pare cam greoaie în comparație cu aplicațiile super fluide de pe iPhone. De aceea, aproximativ în 2010, am asistat la apariția aplicațiilor cu o singură pagină, cu ajutorul unor framework-uri precum AngularJs și React. Câțiva ani mai târziu, în paradigma SPA, toată randarea interfeței se face în browser - începeți cu o pagină HTML ca un shell, apoi executați JavaScript pentru a reda interfața și pentru a prelua orice date necesare cu o cerere HTTP suplimentară.

Acum, chiar dacă este doar o singură pagină, aceasta poate avea mai multe rute.

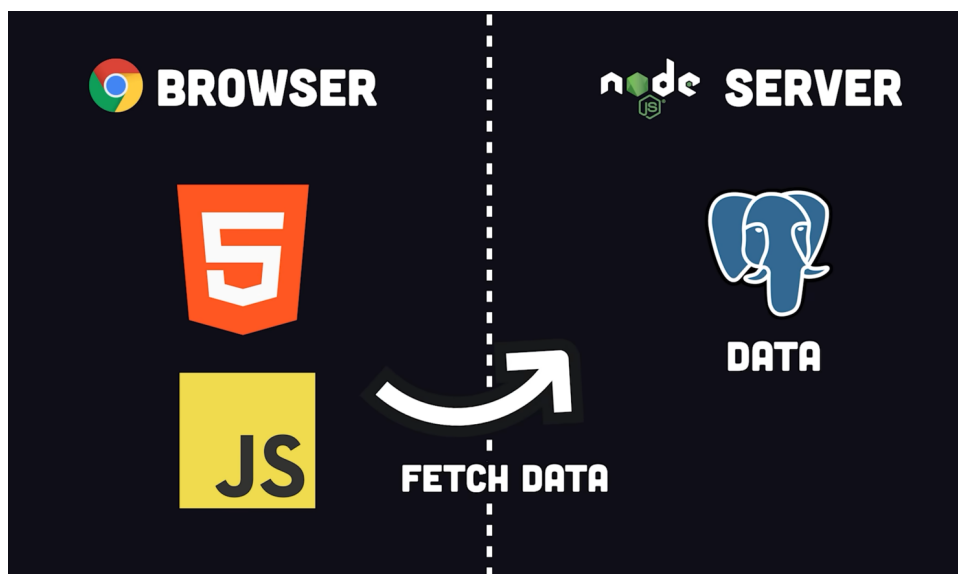


Figura 1.3: Client side rendering. Paradigma SPA

Aceste rute nu indică un server, ci sunt doar actualizate de JavaScript în browser. Acest lucru are avantajul imens de a fi instantaneu pentru utilizatorul final, spre deosebire de o aplicație cu mai multe pagini care ar putea dura cel puțin câteva sute de milisecunde sau mai mult pentru a reda pagina, dar există câteva dezavantaje mari: unul este că necesită un pachet mare de JavaScript, ceea ce poate face ca încărcarea inițială a paginii să fie destul de lentă și doi - deoarece redă doar o 'coajă', motoarele de căutare, chiar și în prezent, au dificultăți în a înțelege orice conținut de pe rutele dinamice, ceea ce nu este un lucru bun dacă aveți nevoie de o SEO bună sau dacă doriți ca oamenii să vă partajeze conținutul pe rețelele sociale. Câțiva ani mai târziu a venit timpul pentru un nou tip de framework, ceva care să poată reda HTML și date pe server sau la încărcarea inițială a paginii, apoi să se hidrateze la JavaScript pe partea clientului după aceea. Astăzi numim acest lucru SSR (server side rendering), dar ideea generală este că cererea inițială merge la un server și redă totul în mod dinamic, apoi, după încărcarea inițială a paginii, JavaScript preia controlul pentru a vă oferi o experiență asemănătoare cu cea a unei aplicații cu o singură pagină.

Această abordare best-of-both-worlds este utilizată de cadre precum Next JS, Nuxt, Svelt Kit și așa mai departe, care sunt adesea denumite meta Frameworks. Aceasta este probabil cea mai populară strategie de randare la ora actuală, dar există încă unele dezavantaje: un dezavantaj este că aveți nevoie de un server real, iar serverele costă bani. O ușoară variație a SSR este SSG sau generarea de site-uri statice. În această Paradigmă,

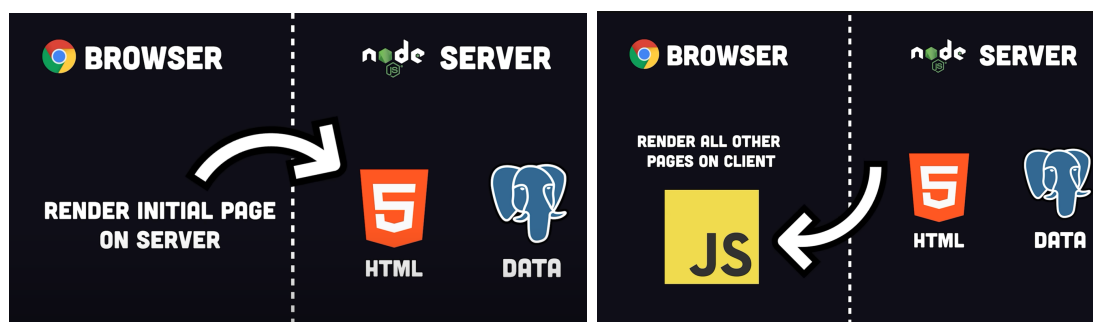


Figura 1.4: Client side rendering. Paradigma SPA

vă redați în avans tot HTML-ul, apoi îl încărcați pe o gazdă statică, cum ar fi un bucket de stocare, dar, la fel ca SSR, se va hidrata în JavaScript după încărcarea inițială a paginii. Site-urile web de acest tip sunt adesea numite Jam stack sites și sunt construite de obicei de aceleași meta Frameworks precum Nexjs și Swelt Kit, care obțin simplitatea și găzduirea cu costuri reduse a unui site static cu experiența de tip aplicație a unui SPA. Singurul lucru rău este că trebuie să redistribuiți site-ul ori de câte ori se schimbă datele și de aceea au inventat ISR sau regenerarea statică incrementală.

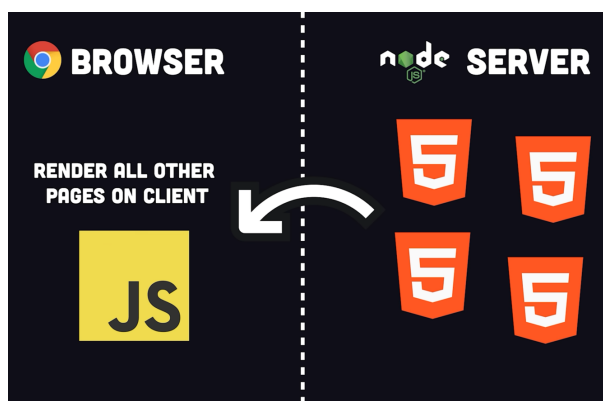


Figura 1.5: Static site generation

Ideea ce stă la bază este de a implemena un site static, cu reconstrucția paginilor individuale din mers. Pe server, când cache-ul este invalidat în mod normal, cu un site static poți doar să pui totul în cache permanent pe un CDN, ceea ce îl face extrem de rapid. Cu ISR, memoria cache poate fi invalidată pe baza anumitor reguli, cum ar fi o anumită perioadă de timp, iar atunci când se întâmplă acest lucru, paginile vor fi reconstruite, ceea ce vă permite să gestionați date dinamice fără a fi nevoie de o desfășurare efectivă a serverului, așa cum ar fi în cazul SSR. Obțineți ce este mai bun din ambele lumi între SSG și SSR, dar dezavantajul este că este mai complex de configurat pe cont

propriu, ceea ce înseamnă că probabil va trebui să găsiți o gazdă precum Vercel care să o suporte din start. Acum, o altă problemă despre care nu am vorbit cu nici un cadru care folosește hidratarea este că la încărcarea inițială a paginii, aplicația ar putea părea că este înghețată în timp ce JavaScript încă se execută pentru a prelua procesul de randare. Pentru a rezolva această problemă avem hidratarea parțială. [9]

Pe un site web de mari dimensiuni, JavaScript poate avea multe de făcut pentru lucruri care nici măcar nu sunt vizibile pentru utilizatorul final, cum ar fi, de exemplu, poate că aveți cel mai grozav și foarte interactiv subsol din lume, dar care umflă stiva de apeluri JavaScript. Cu o hidratare parțială, ați putea reda mai întâi componentele din partea de sus a paginii și apoi să așteptați până când utilizatorul derulează în jos înainte de a face acea componentă interactivă.

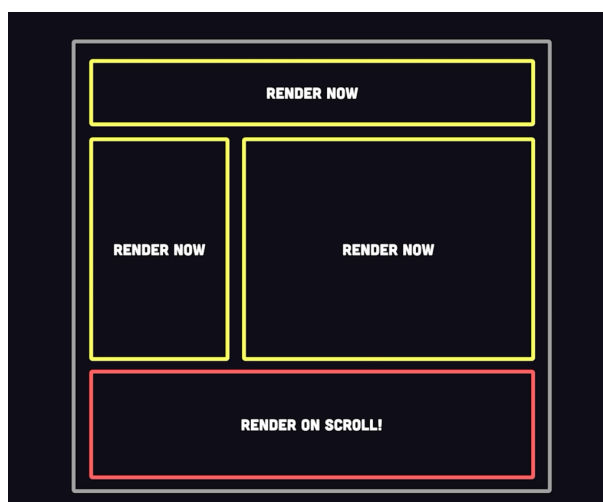


Figura 1.6: Hidratarea parțială

În prezent, multe instrumente suportă divizarea codului pentru a vă împărți aplicațiile în bucăți mai mici pentru a facilita modelele de lazy load de acest tip, dar ar putea fi posibilă o randare și mai eficientă cu arhitectura insulelor. În mod normal, atunci când se hidratează, JavaScript preia întreaga pagină, dar acest lucru nu este foarte eficient, deoarece multe componente sunt doar statice și neinteractive. Cu Islands se începe cu HTML static, apoi se folosește JavaScript doar pentru a hidrata componentele interactive. Acest lucru vă oferă insule de interactivitate. Cadre precum Astro facilitează acest model.

Un aspect pozitiv este că puteți avea o pagină care nu este deloc interactivă, caz în care nu se trimite niciun JavaScript către client, chiar dacă ați construit interfața cu un cadru JavaScript precum React. Acum, încă o altă modalitate de a aborda hidratarea inefficientă

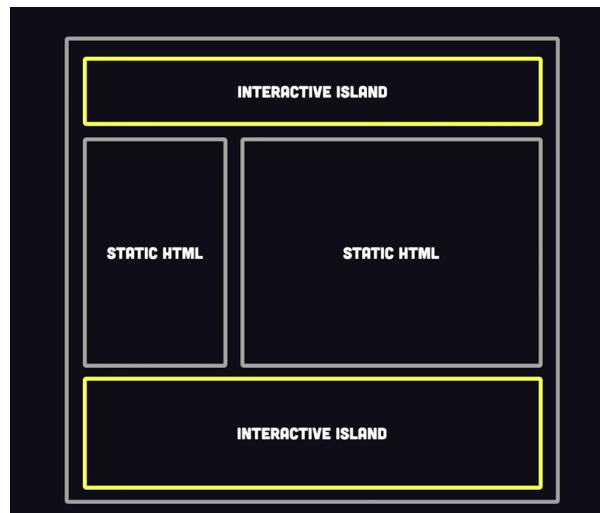


Figura 1.7: Arhitectura pe insule

este o paradigmă numită streaming SSR, care este susținută în cadre precum Next.js 13. Cu directorul de aplicații, datorită unor blocuri de construcție precum componentele serverului React, practic vă permite să redați conținutul de pe partea serverului în mod simultan în mai multe bucăți în loc de toate deodată. În cele din urmă, acest lucru înseamnă că interfața de utilizator devine interactivă mai rapid și se simte mai performant pentru utilizatorul final, dar dacă există o modalitate de a scăpa de hidratare cu totul, deoarece pare a fi sursa multor probleme. Ei bine, aici intervine rezumabilitatea, care este o nouă paradigmă de randare inițiată de quick framework.

Aceasta are o abordare interesantă, în care un site web și toate datele sale, chiar și lucruri cum ar fi ascultătorii de evenimente JavaScript, sunt serializate în HTML, apoi codul JavaScript real este împărțit în tone de bucăți mici. Asta înseamnă că încărcarea inițială a paginii este întotdeauna HTML static, nu este nevoie de hidratare, orice JavaScript necesar pentru interactivitate este lazy loaded în fundal.

1.2.5 Consecințele evoluției web

Evoluția 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 extinsă 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 aplicațiilor 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 ușurință, indiferent de locația utilizatorilor.

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

Dezvoltarea și popularizarea framework-urilor și bibliotecilor de cod, cum ar fi Angular, React și Vue, a simplificat dezvoltarea aplicațiilor web și a îmbunătățit 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, servicii web, API-uri și multe altele. În plus, aplicațiile web trebuie să funcționeze pe o varietate de platforme și dispozitive, ceea ce adaugă o altă dimensiune a complexității.

Pe măsură ce aplicațiile web devin mai complexe, devin mai dificil de dezvoltat și de întreținut. Developerii 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.

1.3 Evoluția 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 (*Asynchronous 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 receptiv.

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.

1.4 De ce comparăm server side rendering cu client side rendering?

Alegerea unei paradigme în detrimentul alteia are numeroase consecințe. De exemplu, preferința unei paradigme poate avea implicații asupra timpului de încărcare a paginii, a costurilor de dezvoltare și întreținere. Timpii crescuți de încărcare a paginii pot duce la abandonul clienților. De asemenea, optimizarea pentru motoarele de căutare și compatibilitatea cu browserul sunt, de asemenea, afectate de alegerea făcută.

1.4.1 Lipsa literaturii

Literatura despre aceste consecințe se găsește în principal în postări pe bloguri, în care se propune experiența personală și opinia unui developer sau a unei echipe. Deși există literatură despre tehnologiile moderne, inclusiv aplicațiile cu o singură pagină și randarea pe partea client, căutarea pentru literatură academică pe acest subiect nu dă rezultate.

1.4.2 Dificultate pentru utilizatori

Pe de altă parte, organizațiile se confruntă cu dificultăți pentru a decide care strategie de randare să aleagă pentru aplicațiile lor specifice. De exemplu, Twitter a folosit inițial randarea pe partea server și apoi a trecut la randarea pe partea client în 2010. Apoi, au trecut din nou la randarea pe partea server în 2012. Acest lucru sugerează că ar fi utilă o comparație pentru a ajuta la luarea unei decizii obiective privind alegerea uneia dintre cele două paradigme.

Capitolul 2

Tehnologii de randare

Rendering-ul este procesul de conversie a datelor într-o reprezentare vizuală și joacă un rol esențial în aplicațiile web moderne, permițând furnizarea de conținut dinamic și interactiv către utilizatori. De-a lungul anilor, tehnologiile de randare au evoluat semnificativ, determinate de progresele în dezvoltarea web și de cererea tot mai mare pentru aplicații de înaltă performanță.

Inițial a fost realizată în primul rând prin randarea directă a paginilor HTML. Serverul genera fișiere HTML statice, care erau transmise browserului clientului.

Odată cu apariția JavaScript, au apărut tehnicile de randare dinamică. Efectele și animațiile bazate pe JavaScript au contribuit semnificativ la evoluția experienței utilizatorului, acestea fiind randate pe partea clientului (utilizatorului).

2.1 Server-side rendering

SSR generează HTML complet pentru o pagină de pe server ca răspuns la navigare. Acest lucru evită navigarea dus-întors suplimentare pentru preluarea datelor și modelarea pe client, deoarece acestea sunt gestionate înainte ca browserul să primească un răspuns.

Server-side rendering produce, în general, un FCP rapid. Rularea logicii paginii și randarea pe server fac posibilă evitarea trimiterii unui JavaScript masiv către client. Acest lucru ajută la reducerea TBT-ului unei pagini, ceea ce poate duce și la un INP mai mic, deoarece thread-ul principal nu este blocat la fel de des în timpul încărcării paginii. Când thread-ul principal este blocat mai rar, interacțiunile utilizatorilor vor avea mai multe oportunități rulate mai devreme. Acest lucru are sens, deoarece cu randarea pe

partea serverului, textul și link-urile se trimit către browserul utilizatorului. Această abordare poate funcționa bine pentru un spectru larg de condiții de dispozitiv și de rețea și permite optimizarea prin intermediul analizării documentelor în flux.

Datorita la SSR, este mai puțin probabil ca utilizatorii să aștepte execuția JavaScript legat de CPU înainte de a putea folosi site-ul. Cu toate acestea, există un compromis potențial cu această abordare: generarea de pagini pe server necesită timp, ceea ce poate duce la un TTFB mai mare.

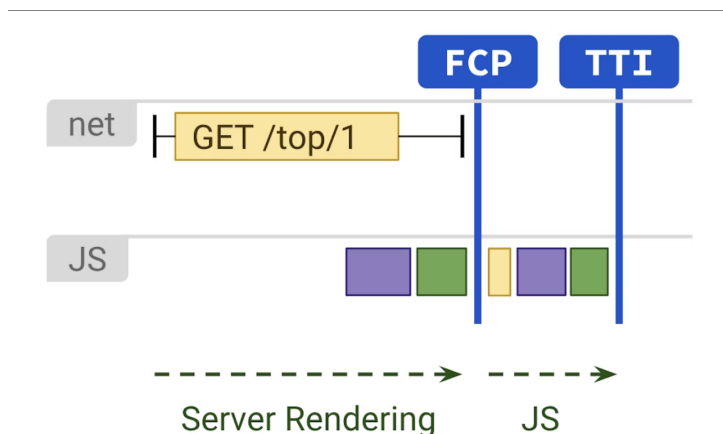


Figura 2.1: Server-side rendering

2.2 Static rendering

Randarea statică are loc în timpul build-ului aplicației. Această abordare oferă un FCP rapid și, de asemenea, un TBT și un INP mai scăzut - presupunând că cantitatea de JS la nivelul clientului este limitată. Spre deosebire de randarea pe server, reușește, de asemenea, să obțină un TTFB constant rapid, deoarece HTML pentru o pagină nu trebuie să fie generat dinamic pe server. În general, randarea statică înseamnă producerea unui fișier HTML separat pentru fiecare adresă URL în avans. Cu răspunsurile HTML generate în avans, randările statice pot fi implementate în mai multe CDN-uri pentru a profita de edge caching.

Unul dintre dezavantajele randării statice este că fișiere HTML individuale trebuie generate pentru fiecare URL posibil. Acest lucru poate fi dificil sau chiar imposibil atunci când nu puteți prezice din timp care vor fi acele adrese URL sau pentru site-uri cu un număr mare de pagini unice.

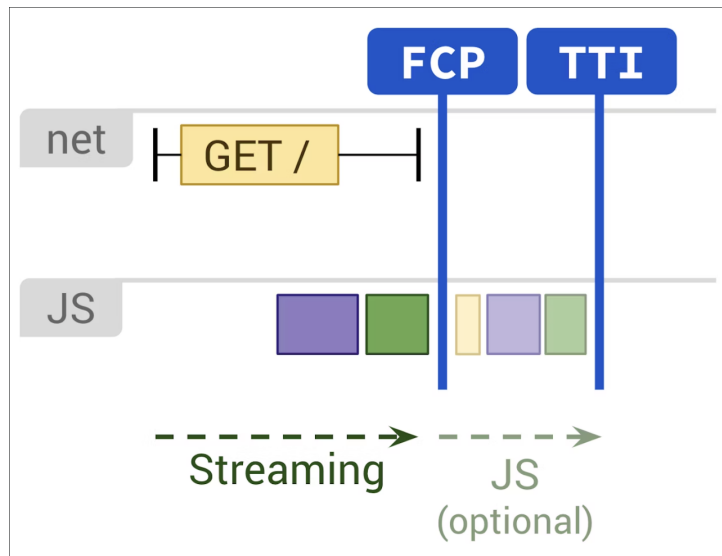


Figura 2.2: Static rendering

2.3 Client-side rendering

CSR reprezintă randarea paginilor direct în browser cu ajutorul la JavaScript. Toată logica, preluarea datelor, șablonarea și rutarea sunt gestionate preponderent de client.

Client-side rendering poate fi dificil de obținut și păstrat rapid pentru dispozitivele mobile. Randarea pe partea de client se poate apropia de performanța randării pe partea de serverului, păstrând un bundle JavaScript restrâns și oferind valoare în cât mai puține călătorii dus-întors. Scripturile și datele esențiale pot fi livrate mai devreme folosind *link rel=preload*, ceea ce face ca analizatorul să funcționeze mai repede pentru dvs. Modele precum PRPL merită, de asemenea, evaluate pentru a se asigura că navigațiile inițiale și ulterioare sunt instantanee.

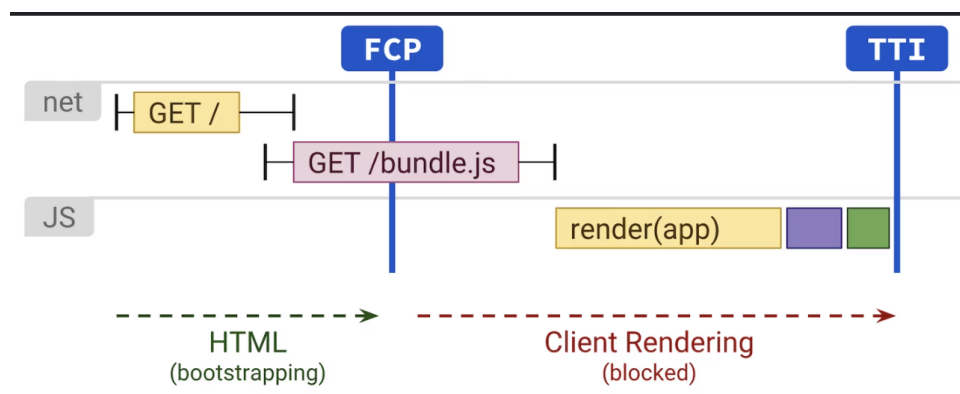


Figura 2.3: Client-side rendering

Principalul dezavantaj al CSR este că, cantitatea de JavaScript necesară tinde să crească pe măsură ce o aplicație crește, ceea ce poate avea efecte negative asupra INP-ului unei pagini. Acest lucru devine deosebit de dificil odată cu adăugarea de noi biblioteci JavaScript, polyfill-uri și cod third-party, care concurează pentru puterea de procesare și trebuie adesea procesate înainte ca conținutul unei pagini să poată fi randat.

2.4 Rehydration

Această abordare încearcă să negocieze compromisurile dintre randarea pe partea client și redarea pe partea serverului, făcând ambele. Solicitățile de navigare, cum ar fi încărcările întregii pagini sau reîncărcările, sunt gestionate de un server care randează pagina HTML, apoi JavaScript și datele utilizate pentru randare sunt încorporate în documentul rezultat. Acest lucru rezultă într-un FCP rapid la fel ca randarea pe partea serverului, apoi urmează rerandarea pe client, folosind o tehnică numită *(re)hidratare*. Aceasta este o soluție eficientă, dar poate veni cu dezavantaje considerabile de performanță.

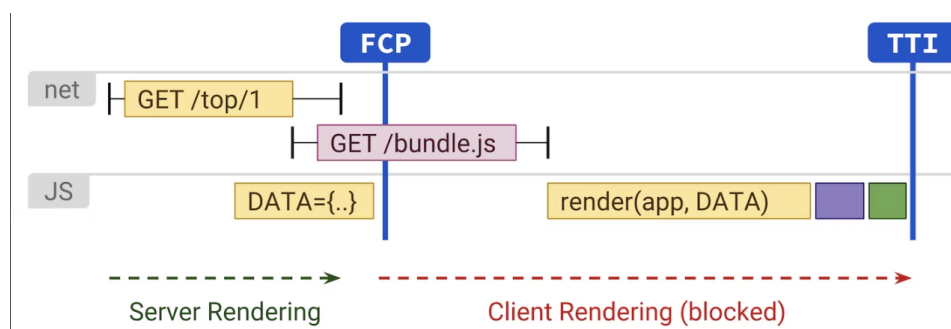


Figura 2.4: Tehnologia de hidratare

Principalul dezavantaj rehidratării este că poate avea un impact negativ semnificativ asupra TBT și INP, chiar dacă îmbunătățește FCP. Paginile randate de pe server pot părea înșelător a fi încărcate și interactive, dar nu pot fi utilizate efectiv până când scripturile de pe partea clientului pentru componente nu sunt executate și manipulatorii de evenimente au fost atașați. Acest lucru poate dura secunde sau chiar minute pe dispozitive mobile.

Problemele de rehidratare pot fi adesea mai grave decât interactivitatea întârziată din cauza JavaScript. Pentru ca client-side JavaScript să poată „relua” cu acuratețe de unde a rămas serverul, fără a fi nevoie să solicite din nou toate datele pe care serverul le-a folosit

pentru a randa HTML-ul, soluțiile actuale de SSR serializează în general răspunsul de la dependențele de date ale unei interfețe de utilizator în document ca etichete de script. Documentul HTML rezultat conține un nivel ridicat de duplicare:

<pre> <head> <title>ToDo</title> <link rel="stylesheet" href="/bundle.css"></script> </head> <body> </pre>	Head, generally not flushed early due to possible mutation by server rendering.
<pre> <h1>To Do's</h1> <input type="checkbox"> Wash dishes <input type="checkbox" checked> Mop floors <input type="checkbox"> Fold laundry <footer><input placeholder="Add To Do..."></footer> </pre>	Static HTML version of the requested page. Generally inert due to use of JS event handlers.
<pre> <script> var DATA = {"todos":[{"text":"Wash dishes","checked":false,"created":1546464530049}, {"text":"Mop floors","checked":true,"created":1546464571013}, {"text":"Fold laundry","checked":false,"created":1546424241610}]} </script> </pre>	Data required to render the view (which is already rendered above)
<pre> <script src="/bundle.js"></script> </body> </pre>	JS to boot up

Figura 2.5: Duplicarea de cod la hidratare

Totuși există o modalitate de a optimiza randarea pe server cu rehidratare. Utilizarea randării pe partea de server pentru conținut cu un grad ridicat de stocare în cache poate reduce TTFB, producând rezultate similare cu randarea preliminară. Rehidratarea progresivă sau parțială poate fi cheia pentru a face această tehnică mai viabilă în viitor.

Capitolul 3

Aplicația practică

Obiectivul acestui capitol este de a oferi 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.

Se va implementa aceeași aplicație cu ajutorul la mai multe framework-uri JavaScript, cât și în vanilla (*plain*) JavaScript, pentru a obține rezultate ce se vor putea compara, pentru a identifica cele mai optime abordări posibile.

3.1 Metricile 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 metrice 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.

- **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 fracțiunii de impact și fracțiunii de distanță a fiecărui eveniment de schimbare de aspect și apoi însumându-le pe întreaga încărcare a paginii. Fracțiunea de impact reprezintă proporția vizualizării afectată de schimbare, în timp ce fracțiunea de distanță reprezintă distanța maximă pe care elementul o parcurge în raport cu vizualizarea.

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 bune practici, cum ar fi stabilirea dimensiunilor explicite pentru elementele media, rezervarea spațiului pentru reclame sau conținut dinamic și evitarea injectării dinamice a conținutului deasupra elementelor 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.

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.

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 Plugin-uri, extensii și aplicații pentru măsurarea performanței

Există mai multe metode disponibile pentru măsurarea performanței unui 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** chrome extension - 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 de-a lungul anilor o serie de instrumente pentru a măsura și raporta performanța. Unii dezvoltatori sunt experți în utilizarea acestor instrumente, în timp ce alții au considerat că abundența atât a instrumentelor, cât și a metricilor este dificilă de urmărit. Inițiativa Web Vitals își propune să simplifice peisajul și să ajute site-urile să se concentreze pe metricile care contează cel mai mult, Core Web Vitals. Fiecare dintre Core Web Vitals reprezintă o componentă distinctă a experienței utilizatorului, este măsurabilă în teren și reflectă experiența reală a unui rezultat critic centrat pe utilizator.

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** chrome developer tool - 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, optimizare pentru motoarele de căutare (SEO) și altele. Lighthouse este folosit în mod frecvent de dezvoltatori pentru optimizarea site-urilor și aplicațiilor web în vederea obținerii unei experiențe mai bune pentru utilizatori.
- **PageSpeed Insights** website - unealtă de analiză a performanței web dezvoltată de Google. Aceasta oferă informații și recomandări pentru optimizarea vitezei și performanței unei pagini web. Prin analizarea conținutului unei adrese URL, PageSpeed Insights generează un raport care identifică probleme legate de performanță și sugerează îmbunătățiri.

3.3 Aplicația practică

Pentru a compara performanța dintre mai multe framework-uri JavaScript, este nevoie de o aplicație care va fi implementată în toate framework-urile, cu cât mai puține elemente distincte, 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 practica - 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 posturilor 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 posturilor

```
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 framework

3.4 Analiza performanței

Bibliografie

- [1] John H. Conway. Server-side rendering versus client-side rendering: A performance comparison. ACM, 2015
- [2] Maximiliano Firtman. Progressive Web Apps: The Future of Web Development. O'Reilly Media, 2017
- [3] Jason W. Bock. Server-Side Rendering with React and Redux. Packt Publishing, 2018
- [4] Tim Berners-Lee. The WorldWideWeb browser. W3C, 1990
- [5] Tim Berners-Lee, Daniel Connolly. Hypertext markup language (html). CERN, Geneva, Switzerland, 1993
- [6] Serge Demeyer. Software Metrics - ansymore.uantwerpen.be, 2017
- [7] 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
- [8] Benjamin Jakobus. VueJS vs Angular vs ReactJS with Demos - <http://www.dotnetcurry.com/vuejs/1372/vuejs-vs-angular-reactjs-compare>, 2017
- [9] Jeff Delaney. 10 Rendering Patterns for Web Apps, 2023
- [10] Sergey Laptick. Client Side vs Server Side UI Rendering. Advantages and Disadvantages - <https://blog.webix.com/client-side-vs-server-side-ui-rendering/>, 2017
- [11] Jason Miller, Addy Osmani. Rendering on the Web - <https://web.dev/rendering-on-the-web/>, 2019

Anexa A

Glosar

A.1 Acronime

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

Tabela A.1: Tabelă de acronime