Optimizacija spletne rešitve v ogrodju Next.js z Google Lighthouse

Leon Pahole

Povio Inc., San Francisco, United States of America
leon.pahole@povio.com

Raziskave kažejo, da so strani, ki se hitro naložijo, bolj optimizirane za spletne iskalnike, prijaznejše za uporabnike in bolj profitabilne. Google Lighthouse je orodje za analizo kvalitete spletnih strani, s katerim lahko med drugim izmerimo učinkovitost nalaganja strani. Učinkovitost je izmerjena na podlagi metrik, ki izražajo uporabniško izkušnjo ob nalaganju. Next.js je ogrodje za razvoj spletnih aplikacij, ki temelji na knjižnici React, pri čemer še posebej poudarja optimizacijo delovanja spletnih aplikacij, predvsem nalaganje le-teh. V prispevku smo na kratko razložili performančne metrike, ki jih meri Google Lighthouse v sklopu ocene učinkovitosti spletne strani, nato pa smo na podlagi praktičnih izkušenj predstavili naslednje Next.js tehnike za optimizacijo teh metrik: izrisovanje na strežniku, statično izrisovanje, inkrementalno statično izrisovanje, ločevanje kode, optimizacija slik, optimizacija pisav, strežniške komponente in nalaganje na zahtevo. Izkaže se, da lahko v praksi s pomočjo navedenih tehnik, v kombinaciji z meritvami iz Google Lighthouse, na enostaven in intuitiven način izboljšamo nalaganje naše spletne strani, kar vodi k boljši uporabniški izkušnji.

Ključne besede:
Next.js
Google Lighthouse
optimizacija nalaganja
kakovost informacijskih rešitev
spletne aplikacije

Univerzitetna založba Univerze v Mariboru
ISBN 978-961-286-772-0
DOI https://doi.org/10.18690/um.feri.8.2023.15

1 Uvod

V modernem razvoju spletnih rešitev ni pomembno le, da so naše spletne strani privlačne in funkcionalne, temveč tudi to, da so učinkovite. Raziskave kažejo, da so strani, ki se hitro naložijo, bolj optimizirane za spletne iskalnike (angl. search engine optimization, v nadaljevanju SEO), prijaznejše za uporabnike in bolj profitabilne [1,2,3].

Google Lighthouse (v nadaljevanju Lighthouse) je eno izmed orodij za analizo učinkovitosti nalaganja spletnih strani. Deluje na podlagi ocen metrik, ki so bile iz strani Lighthouse ekipe določene kot pomembne za hitro nalaganje in dobro uporabniško izkušnjo.

V tem prispevku smo predstavili lastne izkušnje in s tem povezano znanje za optimizacijo spletnih strani, napisanih v ogrodju Next.js, glede na metrike iz Google Lighthouse:

- V prvem delu smo opisali metrike, ki jih Lighthouse definira kot pomembne za učinkovitost.
- V drugem delu smo na podlagi analize in lastnih izkušenj predstavili tehnike v ogrodju Next.js, s katerimi lahko izboljšamo Lighthouse oceno in posledično pohitrimo delovanje spletne strani.

Kljub temu da v prispevku obravnavamo ogrodje Next.js, je prispevek zasnovan konceptualno, zato upamo, da bodo podane informacije koristile tudi spletnim razvijalcem, ki ne uporabljajo ogrodja Next.js.

2 Google Lighthouse

Lighthouse je odprtokodno orodje za analizo kvalitete spletnih strani. Kvaliteta je izražena v petih kategorijah: učinkovitost (angl. performance), SEO, dostopnost (angl. accessibility), dobre prakse (angl. best practices) in progresivne spletne aplikacije (angl. progressive web apps ali PWA) [4].

Rezultat Lighthouse analize je numerična ocena (med 1 in 100) za vsako izmed kategorij in seznam predlogov za izboljšavo ocene. Ker je teste možno izvesti hitro in pogosto, lahko razvijalci Lighthouse uporabijo kot pripomoček pri izboljšavi kvalitete spletne strani.

V tem prispevku smo se osredotočili na oceno učinkovitosti, s pomočjo katere lahko merimo hitrost nalaganja spletne strani. Pri tem posebej poudarjamo, da gre za optimizacijo začetnega nalaganja spletne strani in ne optimizacijo delovanja strani, ko je le-ta že naložena.

2.1 Izračun Lighthouse ocene učinkovitosti

Kriterij izračuna Lighthouse ocene1 ni stalen, temveč se lahko skozi čas spreminja glede na to, katere metrike se na podlagi raziskav ekipi Lighthouse zdijo najprimernejše kot izraz dobre uporabniške izkušnje [5].

V času pisanja tega prispevka je aktualna Lighthouse verzija 10 [6], ki se osredotoča na naslednje metrike2: First Contentful Paint (FCP), Speed Index (SI), Largest Contentful Paint (LCP), Total Blocking Time (TBT) in Cumulative Layout Shift (CLS).

Vsaka izmed metrik prispeva h končni oceni na podlagi uteži, ki predstavlja njeno pomembnost. Uteži so določene na podlagi podatkov iz realnih strani in so porazdeljene po logaritemsko normalni porazdelitvi. Za lažjo predstavo si lahko pomagamo s kalkulatorjem uteži in ocene [7].

[1] seznam sestavnih delov – izvorno "Software Bill of Materials", okrajšava "SBOM"
[1] Repozitorij, https://www.sona

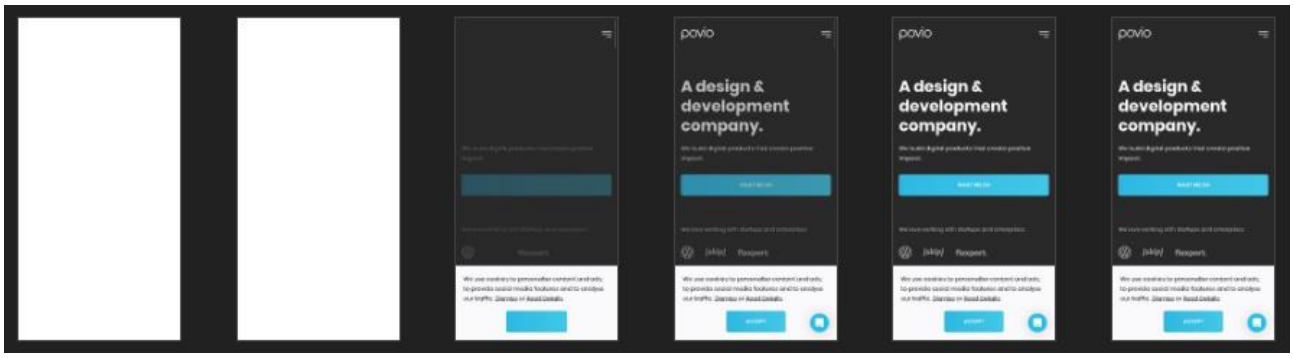
V nadaljevanju so predstavljene performančne metrike. Ker se v prihodnosti lahko spremenijo, smo metrike opisali na kratko, bralce pa pozivamo k rednemu spremljanju novih verzij Lighthouse [6]. Prav tako je pomembno razumevanje dejstva, da so vse metrike izbrane na podlagi vizualne uporabniške izkušnje med procesom nalaganja.

2.1.1 First Contentful Paint (FCP)

FCP meri čas (v sekundah), v katerem se na strani prikaže prvi vizualni element - tekst, slika, ali katerikoli drug element, ki je viden [8].

Slika 1 prikazuje korake nalaganja spletne strani - v tem primeru se FCP zgodi na tretjem koraku, ko se na zaslon izrišejo prvi elementi.

Metrika FCP je pomembna, saj izraža dobro uporabniško izkušnjo pri nalaganju strani - prej, ko se na strani začne izrisovati katerikoli vizualni element, prej uporabnik dobi indikacijo, da se stran dejansko nalaga.



Slika 1: Izris spletne strani.

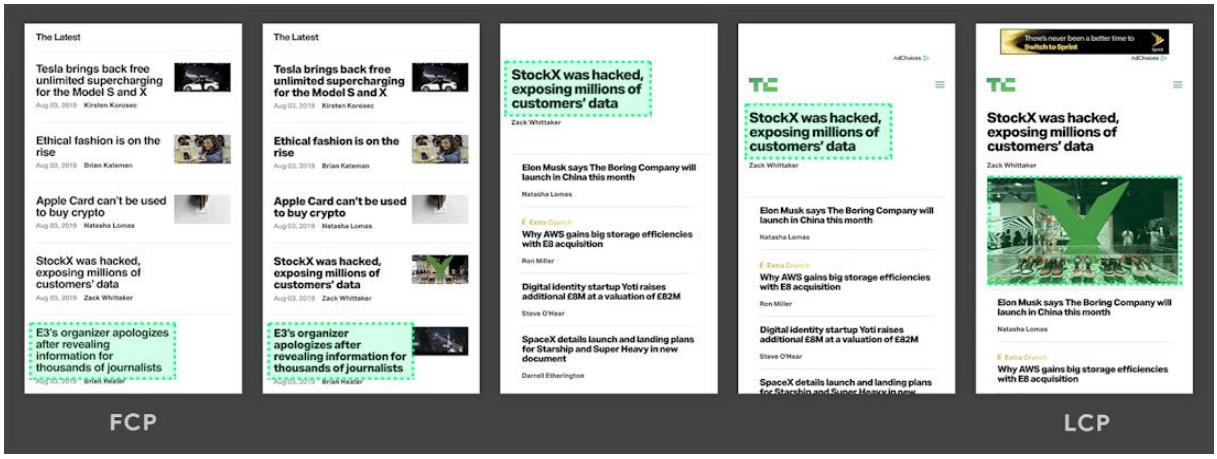
2.1.2 Speed Index (SI)

SI meri hitrost (v sekundah), s katero se vizualni elementi na strani izrisujejo med nalaganjem in kako hitro se na strani izrišejo vsi vizualni elementi. V nasprotju s FCP, je SI bolj celovita metrika, saj obravnava celotno časovnico nalaganja spletne strani, vključno s postopnim nalaganjem elementov in enakomernostjo nalaganja [9,10]. Metrika se izračuna na podlagi videa nalaganja strani, nad katerim se izvede algoritem Speedline [11].

2.1.3 Largest Contentful Paint (LCP)

LCP meri čas (v sekundah), v katerem se na strani prikaže največji in najpomembnejši vizualni element. Ta element določi hevristični algoritem [12] in je lahko, na primer slika, video, ali velik tekst s pomembnimi informacijami [12]. Pri tem je pomembno poudariti, da se skozi nalaganje strani ta element lahko spremeni, pri merjenju LCP pa je pomemben samo zadnji tak element. Slika 2 prikazuje primer spreminjanja najpomembnejšega elementa (označen s pravokotnikom) skozi nalaganje spletne strani.

Podobno kot FCP in SI, tudi LCP poudarja dobro uporabniško izkušnjo pri nalaganju spletne strani; uporabniki želijo, da se najpomembnejši del spletne strani naloži čimprej.



Slika 2: Izris spletne strani in spreminjanje najpomembnejšega elementa (označen s pravokotnikom)
Vir: [12].

2.1.4 Total Blocking Time (TBT)

Glavna nit (angl. main thread) v brskalniku je zadolžena za izrisovanje spletne strani, izvajanje JavaScript kode in obravnavanje uporabniškega vnosa [13]. Če se ob nalaganju prenese veliko JavaScript kode, je treba le-to prevesti (angl. parse) in izvesti. Med tem časom je glavna nit blokirana in ni zmožna obravnavati uporabniškega vnosa. TBT meri trajanje (v milisekundah), v katerem je bila glavna nit na tak način blokirana, in sicer med FCP in trenutkom, ko je bila stran naložena in je postala interaktivna [14].

TBT meri interaktivnost spletne strani med nalaganjem. Velik TBT pomeni, da je stran med nalaganjem neinteraktivna in se vsi uporabniški vnosi obravnavajo z zamikom, kar nakazuje na slabo uporabniško izkušnjo.

2.1.5 Cumulative Layout Shift (CLS)

CLS meri vizualno stabilnost spletne strani na podlagi nepričakovanih zamikov v postavitvi strani (angl. layout shift, v nadaljevanju zamik). Zamik se zgodi, ko se vsebina spletne strani nepričakovano spremeni ali prestavi, kar povzroči zamik celotne vsebine strani. Takšen zamik negativno vpliva na uporabniško izkušnjo, saj ni vizualno prijeten, prav tako pa lahko pride do klikov na nepredvidene elemente. Metrika je izražena z oceno med 0 in 1, kjer 0 predstavlja stran brez zamikov, 1 pa stran z veliko motečimi zamiki [15].

2.1.6 Velikost JavaScript paketa

Velikost JavaScript paketa (v kB), ki se ob nalaganju prenese na odjemalca (angl. bundle size) ni metrika, ki se neposredno uporablja pri merjenju ocene učinkovitosti spletne strani, jo je pa kljub temu smiselno omeniti, saj posredno vpliva na večino metrik. Hkrati so v sodobni praksi popularna ogrodja, ki uporabljajo veliko količino JavaScript kode, zato je treba velikost JavaScript paketa nenehno spremljati.

V poglavju 2.1.4 smo omenili, da je potrebno JavaScript kodo ob nalaganju prenesti, prevesti in izvesti. Večji, kot je JavaScript paket, dlje časa bo ta proces trajal, kar bo posledično povzročilo vpliv na metrike FCP, LCP, SI in TBT. Kot bomo spoznali v poglavju 3, je veliko število tehnik za optimizacijo namenjenih ravno zmanjševanju velikost JavaScript paketa.

2.2 Merjenje metrik

Učinkovitost naše strani lahko ocenimo na podlagi dveh tipov podatkov: laboratorijski (angl. lab) in terenski (angl. field) [16].

Laboratorijski podatki so pridobljeni s strani razvijalcev v kontroliranem okolju - ponavadi v brskalniku na lokalni napravi razvijalca. Takšni testi so zelo hitri in ponovljivi, zato jih je smiselno zaganjati pogosto, za namene spremljanja in izboljševanja učinkovitosti strani.

Kljub temu da nam laboratorijski podatki lahko podajo približno oceno učinkovitosti spletne strani, pa le-ti niso dovolj reprezentativni za vse uporabnike, ki obiščejo našo spletno stran; uporabniki prihajajo iz različnih geografskih lokacij, z različnimi napravami in različnimi internetnimi hitrostmi. Da dobimo vpogled v to, kako učinkovita je naša stran za realne uporabnike, je potrebno analizirati terenske podatke. Ti podatki se zberejo, medtem ko uporabniki uporabljajo stran in se pošljejo v centralizirano analitiko (npr. Google Analytics).

V praksi predlagamo, da razvijalci uporabljajo oba tipa podatkov.

- Laboratorijski podatki so uporabni v začetnih fazah razvoja, ko še nimamo terenskih podatkov ali pa takrat, ko na podlagi terenskih podatkov vemo, da imamo težavo z učinkovitostjo in moramo težavo podrobneje analizirati v lokalnem razvoju.
- Terenski podatki so nujni za realno sliko o tem, kako učinkovita je naša spletna stran. Dobra Lighthouse ocena še ne pomeni, da je naša stran popolnoma učinkovita za vse uporabnike. Metrike iz terenskih podatkov je potrebno nenehno spremljati.

2.2.1 Izvedba Lighthouse testa

Lighthouse test najlažje izvedemo neposredno v brskalniku Google Chrome ali pa s pomočjo vtičnika Google Lighthouse [4,17]. Obstajajo tudi spletna orodja, s katerimi lahko Lighthouse test zaženemo iz različnih lokacij in v različnih simuliranih okoljih (npr. mobilna naprava s slabo povezavo). Pri testiranju je potrebno omeniti slednje:

- Lighthouse test bo zbral laboratorijske podatke.
- Test je potrebno zagnati v brskalniku v privatnem načinu (angl. incognito), saj s tem preprečimo, da bi predpomnjeni podatki ali vtičniki vplivali na oceno.
- Če teste zaganjamo na lokalni strani (angl. localhost), je smiselno uporabiti Google Lighthouse v terminalu, s čimer lahko simuliramo omrežno latenco [18].
- Testiramo lahko v mobilnem ali navadnem načinu. Mobilni način simulira določene omejitve mobilnih naprav, kot so znižana procesorska zmogljivost in slaba omrežna povezava. Večina spletnih strani cilja tako mobilne naprave kot tudi računalnike, zato sta pomembni obe oceni, pri čemer pa lahko mobilno oceno interpretiramo kot najslabši možni scenarij in stran optimiziramo glede na le-to.
- Poleg hitrosti nalaganja spletne strani je s pomočjo izbire načine (angl. mode) možno testirati tudi druge faze delovanja strani [19]. V tem prispevku se na druge načine nismo osredotočili.
- Lighthouse analizira zgolj eno stran. Večina spletnih strani vsebuje več strani in idealno je, da test zaženemo na vseh. To lahko naredimo z orodjem, kot je Unlighthouse [36].

3 Tehnike za izboljšanje učinkovitosti v Next.js

Next.js je popularno (več kot 4 milijoni prenosov na teden v času pisanja prispevka [20]) ogrodje za razvoj spletnih aplikacij, ki temelji na ogrodju React. Ogrodje postavlja velik poudarek na dobri razvijalski izkušnji (angl. developer experience ali DX) in velikemu naboru tehnik za izboljšanje učinkovitosti spletnih aplikacij, predvsem nalaganja le-teh. V tem poglavju smo predstavili nekaj izmed teh tehnik, analizirali njihov vpliv na performančne metrike in podali praktične nasvete za uporabo.

Za namene jedrnatosti smo vsako tehniko opisali zgolj na konceptualnem nivoju. Ogrodje Next.js se namreč vseskozi razvija, tako da bo lahko v prihodnosti koda drugačna, kot je v času pisanja prispevka, koncepti pa bodo ostali enaki.

3.1 Izrisovanje na strežniku

Izrisovanje na strežniku (angl. server-side rendering ali SSR) je tehnika, pri kateri se vsebina strani v obliki HTML kode generira na strežniku, nato pa se generirana HTML koda pošlje na odjemalca [21]. To je v nasprotju z izrisovanjem na odjemalcu (angl. client-side rendering ali CSR), kjer se spletna aplikacija naloži kot prazna stran, nato pa se izrisuje s pomočjo JavaScripta - to počnejo ogrodja za enostranske aplikacije (angl. single-page application ali SPA), kot so React, Angular in Vue. Next.js privzeto uporablja SSR za vse strani, ki so odvisne od zunanjih podatkov.

SSR v ogrodju Next.js deluje drugače kot v tradicionalnih ogrodjih, kot je npr. PHP. Ko se stran v Next.js aplikaciji naloži na odjemalca, bo zagonsko okolje React prevzelo delovanje spletne strani. To pomeni, da bo React obravnaval uporabniške vnose, izrisovanje in navigacijo, kar praktično pomeni, da se stran prične obnašati enako, kot se obnaša SPA stran. Lahko bi torej rekli, da je Next.js hibriden pristop med SSR in SPA - ob nalaganju se uporabi SSR, ob nadaljnji interakciji pa SPA. To je ponavadi zaželjeno, saj so SPA aplikacije po nalaganju hitrejše in bolj interaktivne, ker ni potrebe po osveževanju strani.

Proces, v katerem Next.js preda izvajanje aplikacije ogrodju React, ni zastonj. Potrebno je namreč inicializirati React zagonsko okolje, zgraditi vse podatkovne strukture (na primer drevo komponent) in povezati vse poslušalce dogodkov (angl. event handler). Ta proces se imenuje hidracija (angl. hydration) [22,23]. Med hidracijo je stran neodzivna za uporabniško interakcijo (šteje se kot eno izmed opravil, ki povečujejo TBT). To pomeni tudi, da je ob nalaganju strani potreben JavaScript.

3.1.1 Vpliv SSR na performančne metrike

Tabela 1 prikazuje vpliv SSR na performančne metrike v primerjavi s SPA. Pojasnimo:

- FCP, LCP in SI so manjše pri SSR, saj se iz strežnika pošlje že pripravljen HTML, ki se lahko takoj prične izrisovati. Zaradi tega se prvi vizualni izris in izris najpomembnejšega elementa zgodita prej. V primeru SPA pa se naloži prazna stran, ki se mora nato še prvič izrisati.
- TBT je manjši za SSR. Čeprav mora SSR pristop izvesti hidracijo, pa mora mora SPA poleg tega izvesti še prvi izris.
- Opomniti moramo, da SSR primer uporablja ločevanje kode (poglavje 3.4), ki dodatno pozitivno vpliva na rezultate.

3.1.2 Kdaj uporabiti SSR

V preteklosti je bilo izbiro med SSR in SPA nujno opraviti na začetku projekta ter izjemno pomembna, saj je napačna odločitev lahko pomenila dolgotrajno migracijo v prihodnosti. Next.js to odločitev nekoliko poenostavi, saj lahko na nivoju posamezne strani (in celo komponente) določimo, katero metodo izrisa bomo uporabili.

SSR ima nekaj prednosti pred SPA:

- Je optimiziran za SEO, saj strežnik ob nalaganju vrne izrisan HTML, namesto prazne strani.
- Pospeši nalaganje strani, saj ni potrebno izvesti začetnega izrisa na odjemalcu. To je pomembno predvsem za mobilne naprave, ki imajo manj procesorske moči.
- Če je naša aplikacija odvisna od podatkov, lahko le-te pridobimo že na strežniku, kar lahko pohitri izvajanje, saj je komunikacija med strežniki ponavadi hitrejša kot komunikacija med odjemalcem in strežnikom. Prav tako je tak način komunikacije varnejši, saj lahko dostop do vira podatkov omejimo zgolj na naš Next.js strežnik.

SPA ima pa naslednje prednosti pred SSR:

- SSR za delovanje potrebuje relativno močan strežnik, medtem ko SPA lahko serviramo na omrežju za dostavo vsebine (angl. content delivery network ali CDN). To pomeni, da je vzdrževanje SPA aplikacij potencialno cenejše, prav tako pa se lahko hitreje naložijo s strežnika (saj je CDN geografsko bližje uporabniku).
- Ker SPA izrisuje zgolj na odjemalcu, je strežnik manj obremenjen kot pri SSR. To lahko naredi razliko pri veliki količini uporabnikov.

Naš predlog za izbiro tehnike izrisovanja temelji predvsem na tem, koliko denarnih sredstev imamo na razpolago in kako pomemben je SEO. Poudariti je treba, da SSR optimizira zgolj nalaganje spletne strani.

- Če razvijamo javno stran, kjer je pomemben SEO in hitrost nalaganja, uporabimo SSR.
- Če razvijamo interno stran (na primer nadzorno ploščo), ki ne potrebuje SEO ali hitrega nalaganja, uporabimo SPA.
- Next.js aplikacijo je možno pretvoriti v SPA aplikacijo, zato lahko v vsakem primeru uporabimo Next.js [24].

3.2 Statično izrisovanje

Statično izrisovanje (angl. static-site generation ali SSG) je tehnika, pri kateri se statične HTML datoteke naše strani generirajo v času prevajanja (angl. build time) naše aplikacije. Generirane HTML datoteke se nato v času izvajanja servirajo statično, brez potrebe po SSR [25]. Če naša stran vsebuje podatke iz zunanjih virov, se podatki pridobijo v času prevajanja. Prav tako se generirajo vse dinamične strani (npr. /blog/2 za drugo stran bloga).

Pomembno je poudariti, da je edina razlika med SSR in SSG hitrost, pri kateri strežnik odgovori na zahtevo odjemalca - pri SSR mora strežnik najprej generirati HTML, pri SSG pa lahko samo vrne vnaprej generiran HTML. Ko se stran naloži, se SSG stran prične obnašati enako, kot SSR (torej, kot SPA), zato hidracija je potrebna.

Next.js privzeto uporablja SSG za vse strani, ki niso odvisne od zunanjih podatkov.

3.2.1 Vpliv SSG na performančne metrike

Tabela 1 prikazuje vpliv SSG na performančne metrike v primerjavi s SSR. Pojasnimo:

- Pričakovano je, da so FCP, LCP in SI boljše v primeru SSG, saj se strežnik hitreje odzove na uporabniško zahtevo in se posledično stran lahko prične hitreje izrisovati.
- Velikost paketa JavaScript in TBT ste enaka, saj tako SSR in SSG izvedeta hidracijo in potrebujeta enako količino JavaScripta za delovanje. Razlika je torej samo v hitrosti odgovora.

3.2.2 Kdaj uporabiti SSG

V Next.js lahko za vsako stran izberemo, ali želimo, da se izrisuje s tehniko SSG. Kako sprejeti to odločitev?

SSG ima naslednje prednosti pred SSR:

- Je hitrejši način izrisovanja, saj serviramo zgolj statični HTML.
- Statične strani lahko serviramo iz CDN, kar dodatno pohitri zahtevke.

SSR ima kljub temu nekaj prednosti pred SSG:

- Največja slabost SSG je nespremenljivost; strani se generirajo med fazo prevajanja in se med izvajanjem ne morejo spreminjati. Če se na naši strani vsebina spreminja (npr. socialno omrežje, kot je Twitter, ali stran, ki je povezana na sistem za upravljanje vsebin /angl. content management system ali CMS/), SSG

najverjetneje ni primeren. Ena izmed rešitev za to težavo je, da v času prevajanja generiramo HTML za statične elemente (navigacija, ikone, ipd.), nato pa dinamične podatke naložimo v času izvajanja na odjemalcu, po tem, ko se zgodi hidracija, vendar ima to slab SEO in je počasnejše od SSR.

- Če imamo veliko strani, se bo SSG stran prevajala dolgo časa, prav tako pa bodo te strani potrebovale veliko prostora na strežniku.

Izbira SSG je torej pogojena z naravo vsebine spletne strani:

- Če se vsebina ne spreminja (npr. statična informativna stran), uporabimo SSG.
- Če se vsebina spreminja, ampak dovolj redko, da si lahko privoščimo ponovni zagon prevajanja ob vsaki spremembi (npr. osebni blog, na katerega objavljamo 1-krat na teden), uporabimo SSG.
- Če se podatki spreminjajo zelo hitro (npr. stran s spremljanjem športnega rezultata v živo) ali pa so podatki personalizirani na uporabnika (npr. Facebook časovnica), uporabimo SSR.
- Če se podatki spreminjajo relativno redko (npr. enkrat na 10 minut), ampak si ne moremo privoščiti ponovnega prevajanja, lahko uporabimo ISR (opisan v poglavju 3.3).

3.3 Inkrementalno statično izrisovanje

V poglavju 3.2 smo omenili, da lahko SSG uporabimo za osebni blog, ker si lahko privoščimo ponovno prevajanje strani ob vsaki objavi. Kaj pa, če ima naš blog več urednikov, ki občasno pišejo objave, ampak nimajo tehničnega znanja, da bi zagnali ponovno prevajanje? V tem primeru moramo uporabiti SSR, da so objavljeni podatki ažurno prikazani na strani.

Vendar ker je SSG tako hiter, bi bilo kljub temu optimalno, da bi ga lahko nekako uporabili v primerih, ko se podatki ne spreminjajo zelo hitro (nekajkrat na minuto). To nam omogoča tehnika inkrementalnega statičnega izrisovanja.

Inkrementalno statično izrisovanje (angl. incremental static rendering ali ISR) nam omogoča, da med izvajanjem aplikacije selektivno posodobimo statične HTML datoteke, ki smo jih generirali v času prevajanja. Prav tako lahko na novo generiramo strani, ki sploh niso bile generirane v času prevajanja: ko uporabnik prvič zahteva stran, ki še ni generirana, jo izrišemo s pomočjo SSR, rezultat shranimo in ob naslednjih zahtevkih serviramo to HTML datoteko iz predpomnilnika. ISR torej v bistvu kombinira SSR (prvi zahtevek) in SSG (naslednji zahtevki) [26].

V ISR je pomembna nastavitev, kako pogosto (če sploh) bomo posodabljali predpomnilnik:

- Na interval, npr. po 5 minutah, bo stran označena kot neažurna in se bo ob naslednjem zahtevku ponovno zgenerirala (s pomočjo SSR).
- Na določen dogodek, npr., ko se spremenijo podatki v CMS. V začetku tega poglavja smo omenili primer z blogom - v tem primeru je dogodek objava.

Vidimo lahko, kako ISR kombinira prednosti SSR (dinamičnost) in SSG (hitrost nalaganja).

3.3.1 Vpliv ISR na performančne metrike

ISR ne počne nič novega, kar nismo analizirali že v prejšnjih poglavjih - ob prvem zahtevku bodo metrike podobne, kot pri SSR (poglavje 3.1), ob naslednjih zahtevkih (do posodobitve predpomnilnika) pa bodo podobne, kot pri SSG (poglavje 3.2).

3.3.2 Kdaj uporabiti ISR

Uporabo ISR predlagamo vedno, ko strani niso statične, ampak se podatki ne spreminjajo tako pogosto, da bi bil strežnik preobremenjen ob ponovnem generiranju HTML datotek.

Dober primer strani, kjer lahko uporabimo ISR, so strani, ki so povezane na CMS ali blogi. Tam pričakujemo relativno redke posodobitve, npr. enkrat na 10 minut.

ISR ni dobra izbira za strani, ki vsebujejo zelo hitro spreminjajoče se podatke (npr. spremljanje lokacije vozila), ali pa so podatki personalizirani za uporabnika (npr. seznam predlaganih izdelkov za uporabnika). Prav tako moramo v obzir vzeti dejstvo, da ISR potrebuje nekaj časa, da izvede generiranje datotek, zato spremenjeni podatki ne bodo takoj vidni na strani. V določenih primerih si tega ne moremo privoščiti (npr. realno-časovno spremljanje novic).

3.4 Ločevanje kode

Naše spletne strani ponavadi sestavlja več strani. V tipični SPA (npr. React) aplikaciji se JavaScript koda za celotno spletno stran (torej za vse strani) prenese ob nalaganju. To je seveda precej potratno, saj stran, ki se nalaga, najverjetneje ne potrebuje celotne kode.

Ločevanje kode je tehnika, pri kateri v času prevajanja JavaScript kodo razdelimo na več datotek (angl. chunk), ponavadi eno za vsako stran. Cilj je zmanjšati nalaganje spletne strani tako, da prenesemo zgolj JavaScript kodo, ki je potrebna za izvajanje zahtevane strani - koda za ostale strani pa se lahko prenese v ozadju, ko se nalaganje zaključi [27].

3.4.1 Vpliv ločevanja kode na performančne metrike

Ločevanje kode prispeva k razliki v metrikah, ki je vidna v tabeli 1, kjer smo primerjali React (brez ločevanja kode) in Next.js (z ločevanjem kode). Ločevanje kode zagotovo izboljša vse metrike, povezane z nalaganjem strani (predvsem TBT), saj zniža količino kode, ki se mora ob nalaganju prenesti in izvesti.

3.4.2 Kdaj uporabiti ločevanje kode

Next.js privzeto uporablja ločevanje kode in v času pisanja tega prispevka ne obstaja način, da bi ga lahko izklopili. Menimo, da je ločevanje kode uporabno na vseh spletnih straneh, zato se strinjamo s tem, da je vedno vključeno.

3.5 Optimizacija slik

Slike so pomemben del vsebine spletnih strani. Ponavadi jih vključimo z značko img, v Next.js pa lahko uporabimo tudi komponento Image, ki je zgrajena nad img značko in za sliko omogoči naslednje optimizacije [28]:

- Nalaganje na zahtevo (angl. lazy loading): slike se naložijo šele, ko so v vidnem polju.
- Optimizacija velikosti: Next.js za vsako sliko generira več velikosti in nato odjemalcu pošlje tisto, ki je najprimernejša glede na velikost in zmogljivost naprave. Na primer, na mobilnih napravah slike v visoki ločljivosti niso potrebne, zato se pošljejo manjše slike, kar pohitri nalaganje.
- Uporaba modernih formatov: v kolikor ga uporabnikov brskalnik podpira, bo Next.js na odjemalca poslal sliko v modernem slikovnem formatu, kot je na primer WebP. Moderni formati so namreč pri isti kvaliteti manjši kot tradicionalni formati, kot sta npr. PNG ali JPEG [29].
- Preprečitev zamikov: Image komponenta preprečuje razvijalcem, da sliki ne bi določili ustreznih dimenzij, kar pripomore k boljši oceni za metriko CLS.

Image komponenta deluje tako za statične, vnaprej določene slike, kot tudi za dinamične, oddaljene (angl. remote) slike. V ozadju Next.js pridobi sliko, jo pretvori v ustrezen format in velikost, jo predpomni in nato pošlje na odjemalca (v tem pogledu deluje podobno kot ISR za HTML datoteke).

3.5.1 Vpliv optimizacije slik na performančne metrike

Tabela 1 prikazuje vpliv optimizacije slik na performančne metrike, v primerjavi z img značko brez dimenzij. Pojasnimo:

- Slikam je vedno potrebno definirati dimenzije. Če uporabimo značko img brez dimenzij (atributa width in height), brskalnik zaradi manjkajočih dimenzij ne bo rezerviral prostora za sliko, kar povzroči zamik ob nalaganju. Zaradi tega je CLS večji, kot je pri Image komponenti, ki nam preprečuje, da slikam ne bi določili dimenzij.
- Image komponenta kaže izboljšavo v vseh metrikah, povezanih s hitrostjo, saj so dostavljene slike manjše in v optimalnem formatu.
- Image komponente se prikažejo hitreje, kar izboljša FCP.
- Image komponenta zahteva več logike na odjemalcu, zato se velikost JavaScript paketa poveča.

3.5.2 Kdaj uporabiti optimizacijo slik

Izgleda, kot da Image komponenta nima slabih lastnosti in se lahko vedno uporablja namesto img značke. S tem se strinjamo, ko gre za statične slike (na primer logotipi, ozadja in ostale dekorative slike).

Ko pa gre za dinamične, oddaljene slike (na primer slike iz strežnika, profilne slike ali druge slike, ki so bile naložene s strani uporabnikov), moramo biti previdni. Spomnimo, da mora vsaka slika v Image komponenti biti prenešena z izvornega strežnika na predpomnilnik Next.js strežnika, kar bo postopoma povečevalo količino shranjenih podatkov in obremenilo strežnik. Zaradi tega slike na večini spletnih ponudnikov za gostovanje Next.js aplikacij niso zastonj. Na primer, v času pisanja prispevka, Vercel gostovanje ponuja zgolj 1000 slik na periodo zastonj [37].

Če stran vsebuje veliko število slik, je potrebno pretehtati naslednje:

- Če razvijamo javno stran, kjer je hitrost nalaganja zelo pomembna, moramo za optimalno nalaganje zagotovo uporabiti Image komponento.
- Če razvijamo interno nadzorno ploščo, ki vsebuje veliko slik, nalaganje pa ni bistvenega pomena, je verjetno bolje uporabiti img značko, s čimer znižamo stroške in obremenjenost strežnika.

3.5.3 Kako izbrati pravilno dimenzijo slik

Image komponenta razvijalce prisili v to, da za sliko definiramo dimenzije. Sliki lahko nastavimo statično velikost (preko width in height atributov) ali pa ji povemo, da se razširi čez celotnega starša.

V kolikor se dimenzije slike ne spreminjajo skozi različne dimenzije zaslonov (npr. logotipi, profilne slike in ikone), vedno uporabimo statične dimenzije. S tem bomo zagotovili optimalno nalaganje brez zamika.

Ko delamo s slikami, katerih dimenzije se spreminjajo na podlagi širine zaslona, moramo dimenzije nastaviti na drug način. Slika 3 prikazuje referenčno implementacijo za primer, kjer želimo, da je slika širine 500px na računalniku, polovico širine zaslona (50vw) na tablici in celotno širino zaslona (100vw) na telefonih. Storili smo slednje:

- Sliko smo nastavili tako, da se raztegne čez celotnega starša (element div).
- Starša smo s pomočjo CSS nastavili na ustrezne širine za vsako izmed širin zaslona.
- Na sliki smo uporabili atribut sizes, da za vsako širino zaslona odjemalec zahteva ustrezno velikost slike.

HTML

```
<div className={crewMemberPageStyles.imageWrapper}>
  <Image
    src={crewMember.images.png}
    alt={`Image of ${crewMember.name} crew member`}
    sizes={`
      (max-width: ${Breakpoints.mobile}) 100vw,
      (max-width: ${Breakpoints.tablet}) 50vw,
      500px
    `}
    fill
    className={crewMemberPageStyles.image}
  />
</div>
```

CSS

```
.imageWrapper {
  position: relative;

  width: 100vw;
  height: 400px;

  @include breakpoints.tablet {
    width: 50vw;
    height: 200px;
  }

  @include breakpoints.desktop {
    width: 500px;
    height: 300px;
  }
}
```

Slika 3: Koda za določitev dimenzij slike, ki ima spreminjajoče se dimenzije na različnih širinah zaslonov.

V tem primeru je treba posebej poudariti atribut sizes. Atribut določa širino slike, ki jo bo odjemalec zahteval na podlagi širine zaslona. Če se bo stran npr. naložila na telefonu, bo odjemalec od strežnika zahteval sliko s širino, ki je enaka širini zaslona (100vw; npr. če je telefon širine 450px, bo to 450px). Na tablici bo polovico zaslona (50vw), na računalniku pa 500px. Višina slike se na strežniku samodejno izračuna iz širine na podlagi razmerja med širino in višino slike (angl. aspect ratio). Pomembno je poudariti, da Next.js najverjetneje ne bo vrnil slike s točno zahtevano dimenzijo, ampak z dimenzijo, ki je podobna, ampak večja od zahtevane. Next.js si namreč v ozadju zgenerira končno število slik z dimenzijami in ne generira dodatnih slik za vsak zahtevek.

3.6 Optimizacija pisav

Pisave so vključene v praktično vsaki moderni spletni strani, vendar pa optimalno nalaganje pisav ponavadi povzroči določene izzive za razvijalce [30]:

- Blokiranje nalaganja: če nismo previdni, lahko nalaganje pisav blokira glavno nit brskalnika, kar pomeni, da se stran ne bo prikazala, dokler se pisava ne bo naložila. To lahko rešimo za asinhronim nalaganjem.
- Prikaz nestiliziranega besedila (angl. flash of unstyled text ali FOUT): če pisave nalagamo asinhrono, se lahko zgodi, da se bo ob nalaganju za nekaj trenutkov prikazala rezervna pisava (angl. fallback font), preden se bo zamenjala s pravo pisavo. To povzroči vidni zamik, ki poslabša CLS. To lahko rešimo tako, da za rezervno pisavo uporabimo čim bolj podobno pisavo, kot je pisava, ki jo nalagamo, nato pa jo še dodatno približamo s pomočjo CSS lastnosti, kot so line-height, letter-spacing in font-size.

Optimizacija pisav (angl. Font optimization), ki jo ponuja Next.js, samodejno reši vse zgoraj opisane težave. Kot razvijalci moramo samo določiti ustrezno pisavo, ki je lahko lokalna ali pa naložena iz Google fonts, Next.js pa bo poskrbel za pravilno nalaganje in ustrezno prilagojeno rezervno pisavo [31].

3.6.1 Vpliv optimizacije pisav na performančne metrike

Tabela 1 prikazuje vpliv optimizacije pisav na performančne metrike, v primerjavi z načinom, ki blokira glavno nit (blokiranje) in z asinhronim nalaganjem s FOUT (async). Pojasnimo:

- V primerjavi z blokiranjem se izboljšajo metrike, povezane s hitrostjo nalaganja, saj pri blokiranju nalaganje strani blokiramo, dokler pisava ni naložena.
- V primerjavi z asinhronim načinom se izboljša predvsem CLS metrika, saj optimizacija pisav preprečuje pojav FOUT. Poleg tega so hitrostne metrike boljše, če uporabljamo Google fonts, saj Next.js v času prevajanja lokalno shrani pisavo, zato povezava z Google fonts strežnikom ni potrebna.

3.6.2 Kdaj uporabiti optimizacijo pisav

Smo mnenja, da optimizacija pisav nima slabosti, saj ne povzroča dodatnih stroškov, temveč zgolj poenostavi vključevanje pisav, ki bi ga razvijalci v vsakem primeru morali narediti. Zato priporočamo, da se vedno uporabi.

3.7 Strežniške komponente

Spletne strani ponavadi vsebujejo več vizualnih komponent. Glede na obnašanje, lahko komponento ločimo na:

- Interaktivne: komponente, ki se odzivajo na uporabniške interakcije in hranijo stanje, npr. vnosno polje.
- Neinteraktivne: komponente, ki zgolj prikazujejo določeno vsebino, npr. objava na socialnem omrežju. Ne hranijo stanja in ne obravnavajo uporabniških vnosov.

V Next.js se vse komponente ponavadi izrišejo na strežniku (SSR), nato pa se hidrirajo na odjemalcu. To zahteva prenos JavaScript kode za vsako izmed komponent iz strežnika na odjemalca. Tukaj pa obstaja priložnost za optimizacijo; v kolikor je komponenta neinteraktivna, namreč hidracija ne bi bila potrebna, saj neinteraktivne komponente ne hranijo stanja in ne obravnavajo uporabniških vnosov. To je motivacija za idejo strežniških komponent.

Next.js nam omogoča, da za vsako komponento določimo, ali je strežniška (angl. server component) ali odjemalčeva (angl. client component). V kolikor je komponenta neinteraktivna, jo lahko označimo kot strežniško, kar bo pomenilo, da se JavaScript koda za komponento ne bo prenesla na odjemalca, saj ni potrebe po hidraciji. To lahko občutno pohitri nalaganje, saj večina spletnih strani vsebuje več neinteraktivnih komponent, kot interaktivnih [32,33].

3.7.1 Vpliv strežniških komponent na performančne metrike

Tabela 1 prikazuje vpliv strežniških komponent na performančne metrike, v primerjavi z implementacijo, kjer so vse komponente odjemalčeve. Pričakovano je JavaScript paket manjši, prav tako pa so posledično izboljšane vse metrike, povezane s hitrostjo nalaganja.

3.7.2 Kdaj uporabiti strežniške komponente

Next.js nam omogoča, da se na nivoju vsake komponente odločimo, ali bo le-ta strežniška ali odjemalčeva. Seveda je zaželjeno, da je čim več komponent strežniških.

Pomembno je poudariti, da potomci odjemalčevih komponent ne morejo biti strežniške komponente. To pomeni, da si moramo za optimalno delovanje drevo React komponent ustrezno izoblikovati tako, da so odjemalčeve komponente čim bližje listom drevesa [33]. To morebiti zahteva nekaj preurejanja kode.

3.8 Nalaganje na zahtevo

Glede na naravo naše spletne strani, določenih komponent morda ni treba naložiti že ob nalaganju strani. Kot primer vzamimo modalno okno, ki se prikaže le, če uporabnik klikne na gumb. JavaScript kode za takšno komponento ne bi bilo potrebno nalagati takoj, ampak bi nalaganje lahko preložili na kasnejši trenutek (npr. ko je stran dokončno naložena, ali pa šele, ko uporabnik postavi kurzor na gumb).

Nalaganje na zahtevo (angl. lazy loading) nam omogoča, da preložimo nalaganje določenih komponent [34]. V zgornjem primeru bi torej lahko preložili nalaganje kode za modalno okno, kar bi kodo za to komponento izločilo iz začetnega JavaScript paketa. S tem pohitrimo nalaganje in posledično izboljšamo performančne metrike.

3.8.1 Vpliv nalaganja na zahtevo na performančne metrike

Tabela 1 prikazuje vpliv nalaganja na zahtevo na performančne metrike, v primerjavi z implementacijo, kjer se nobena komponenta ne nalaga na zahtevo. Rezultati so pričakovani: nalaganje na zahtevo izboljša metrike hitrosti in zmanjša velikost JavaScript paketa, saj se koda za komponente prenese šele po tem, ko se stran naloži.

3.8.2 Kdaj uporabiti nalaganje na zahtevo

Nalaganje na zahtevo lahko občutno izboljša hitrost nalaganja strani, ampak ima nekaj slabosti:

- Režija: prenašanje dodatne kode po nalaganju strani je bolj potratno, kot če bi celoten paket naložili takoj, saj moramo po tem, ko se stran naloži, poslati zahtevke še za ostale komponente. Če je koda za komponento, ki jo nalagamo na zahtevo, zelo majhna, se to morda ne splača.
- Med nalaganjem komponente lahko pride do napake, ki jo moramo nato ustrezno obravnavati.
- Komponenta se ne naloži takoj, kar lahko vodi v počasno delovanje za uporabnike.
- SEO: komponente, ki se nalagajo na zahtevo, ne bodo del vsebine, ki jo vidijo spletni iskalniki.

Odločitev, ali uporabimo nalaganje na zahtevo ali ne, je torej odvisna od narave vsebine naše spletne strani:

- Komponente, ki morajo biti vidne takoj ob nalaganju, ali pa vsebujejo pomembno vsebino za SEO, se naj ne nalagajo na zahtevo.
- Primeri komponent, ki se lahko nalagajo na zahtevo, so modalna okna ali prikazna okna, ki se prikažejo šele nekaj časa po tem, ko se stran naloži (na primer obvestilo o piškotkih).
- Opomba: strežniških komponent (poglavje 3.7) ni potrebno nalagati na zahtevo, saj niso del JavaScript paketa, ki se pošlje na odjemalca. Gre torej zgolj za komponente na odjemalcu.

3.9 Povzetek: vpliv tehnik na performančne metrike

Tabela 1 povzema vpliv opisanih Next.js tehnik na performančne metrike na podlagi testov, izvedenih na različnih velikostih projektov. Vsako tehniko smo primerjali z referenčno implementacijo in zabeležili razliko v metrikah. Legenda znakov:

- =: tehnika ne vpliva na metriko
- +: metrika je večja v primerjavi z referenčno implementacijo
- -: metrika je manjša v primerjavi z referenčno implementacijo

Tabela 1: Povzetek vpliva Next.js tehnik na performančne metrike.

	SSR	SSG	Loč. kode	Opt. slik	Opt. pisav	Opt. pisav	Strežniške komp.	Nal. na zahtevo
Primerjava z	SPA	SSR	SPA brez loč. kode	Img značka brez dimenzij	Blokiranje	Async	Odjemalčeve komp.	Pristop brez nal. na zahtevo
FCP (s)	-	-	-	-	-	-	-	-
SI (s)	-	-	-	-	-	-	-	-
LCP (s)	-	-	-	-	-	-	-	-
TBT (ms)	-	=	-	=	-	-	-	-
CLS	=	=	=	- (dimenzije slik)	=	- (ni FOUT)	=	=
Velikost JavaScript paketa (kB)	= (hidracija)	= (hidracija)	-	+(dodatna logika)	=	=	-	-

3.10 Spremljanje terenskih podatkov

Next.js omogoča enostavno zbiranje terenskih podatkov performančnih metrik (slika 4) [35].

```
export function reportWebVitals(metric) {
  console.log(metric);
}
```

Slika 4: Koda za pridobivanje terenskih podatkov performančnih metrik (web vitals) v Next.js.

Funkcija reportWebVitals bo samodejno klicana, ko bo brskalnik izmeril metrike. Nato jih lahko pošljemo v poljubno orodje za analizo podatkov, npr. Google Analytics ali Datadog.

4 Zaključek

V prispevku smo predstavili praktično znanje in izkušnje na področju optimizacije učinkovitosti nalaganja spletnih strani, razvitih v ogrodju Next.js. Opisali smo metrike, ki jih Google Lighthouse izmeri in uporabi za izračun performančne ocene spletne strani. Nato smo navedli tehnike v ogrodju Next.js, ki jih v praksi uporabljamo za izboljšanje performančnik metrik, pri čemer smo vsako tehniko na kratko opisali, poročali vpliv na metrike in navedli lastno mnenje glede prednosti in slabosti uporabe tehnik.

Tehnike v ogrodju Next.js se v praksi izkažejo za zelo uporabne pri optimizaciji nalaganja spletnih strani, saj so intuitive, enostavne za uporabo in efektivne. V praksi optimizacija poteka tako, da spremljamo terenske podatke, z Lighthouse preverimo, kje lahko izboljšamo nalaganje naše spletne strani, nato pa uporabimo ustrezno tehniko za optimizacijo.

Literatura

[1] https://web.dev/tags/case-study/, Case Study, obiskano 23. 7. 2023.

[2] https://web.dev/why-speed-matters/, Why speed matters, obiskano 23. 7. 2023.

[3] https://www.conductor.com/academy/page-speed-resources/, Steven van Vessum, Why Page Speed Matters: 10 Case Studies Show How, obiskano 23. 7. 2023.

[4] https://developer.chrome.com/docs/lighthouse/overview/, Lighthouse overview, obiskano 23. 7. 2023.

[5] https://developer.chrome.com/en/docs/lighthouse/performance/performance-scoring/, Lighthouse performance scoring, obiskano 23. 7. 2023

[6] https://github.com/GoogleChrome/lighthouse/releases, Lighthouse releases, obiskano 23. 7. 2023

[7] https://googlechrome.github.io/lighthouse/scorecalc/, Lighthouse Scoring Calculator, obiskano 23. 7. 2023

[8] https://web.dev/i18n/en/fcp/, Philip Walton, First Contentful Paint (FCP), obiskano 23. 7. 2023

[9] https://developer.mozilla.org/en-US/docs/Glossary/Speed_index, Speed index, obiskano 23. 7. 2023

[10] https://developer.chrome.com/en/docs/lighthouse/performance/speed-index/, Speed index, obiskano 23. 7. 2023

[11] https://github.com/paulirish/speedline, Speedline, obiskano 23. 7. 2023

[12] https://web.dev/i18n/en/lcp/, Philip Walton, Barry Pollard, Largest Contentful Paint (LCP), obiskano 23. 7. 2023

[13] https://developer.mozilla.org/en-US/docs/Glossary/Main_thread, Main thread, obiskano 23. 7. 2023

[14] https://web.dev/i18n/en/tbt/, Philip Walton, Total Blocking Time (TBT), obiskano 23. 7. 2023

[15] https://web.dev/i18n/en/cls/, Philip Walton, Milica Mihajlija, Cumulative Layout Shift (CLS), obiskano 23. 7. 2023

[16] https://web.dev/lab-and-field-data-differences/, Philip Walton, Why lab and field data can be different (and what to do about it, obiskano 23. 7. 2023

[17] https://chrome.google.com/webstore/detail/lighthouse/blipmdconlkpinefehnmjammfjpmpbjk, Lighthouse, obiskano 23. 7. 2023

[18] https://github.com/GoogleChrome/lighthouse#cli-options, Lighthouse, obiskano 23. 7. 2023

[19] https://github.com/GoogleChrome/lighthouse/blob/HEAD/docs/user-flows.md, User Flows in Lighthouse, obiskano 23. 7. 2023

[20] https://www.npmjs.com/package/next, Tim Neutkens, Naoyuki Kanezawa, Guillermo Rauch, Arunoda Susiripala, Tony Kovanen, Dan Zajdband, Next.js„ obiskano 23. 7. 2023

[21] https://nextjs.org/docs/pages/building-your-application/rendering/server-side-rendering, Server-side Rendering (SSR), obiskano 23. 7. 2023

[22] https://nextjs.org/learn/foundations/how-nextjs-works/rendering, What is Rendering?, obiskano 23. 7. 2023

[23] https://qwik.builder.io/docs/concepts/resumable/, Resumable vs. Hydration, obiskano 23. 7. 2023

[24] https://gist.github.com/gaearon/9d6b8eddc7f5e647a054d7b333434ef6, Next.js client-only SPA example, obiskano 23. 7. 2023

[25] https://nextjs.org/docs/pages/building-your-application/rendering/static-site-generation, Static Site Generation (SSG), obiskano 23. 7. 2023

[26] https://nextjs.org/docs/pages/building-your-application/rendering/incremental-static-regeneration, Incremental Static Regeneration (ISR), obiskano 23. 7. 2023

[27] https://nextjs.org/learn/foundations/how-nextjs-works/code-splitting, What is Code Splitting?, obiskano 23. 7. 2023

[28] https://nextjs.org/docs/pages/building-your-application/optimizing/images, Image Optimization, obiskano 23. 7. 2023

[29] https://www.adobe.com/creativecloud/file-types/image/raster/webp-file.html, WebP files, obiskano 23. 7. 2023

[30] https://css-tricks.com/how-to-load-fonts-in-a-way-that-fights-fout-and-makes-lighthouse-happy/, Adrian Bece, How to Load Fonts in a Way That Fights FOUT and Makes Lighthouse Happy, obiskano 23. 7. 2023

[31] https://nextjs.org/docs/pages/building-your-application/optimizing/fonts, Font Optimization, obiskano 23. 7. 2023

[32] https://adhithiravi.medium.com/what-are-server-components-and-client-components-in-react-18-and-next-js-13-6f869c0c66b0, Adhithi Ravichandran, Understanding Server Components in React 18 and Next.js 13, obiskano 23. 7. 2023

[33] https://nextjs.org/docs/getting-started/react-essentials, React Essentials, obiskano 23. 7. 2023

[34] https://nextjs.org/docs/app/building-your-application/optimizing/lazy-loading, Lazy Loading, obiskano 23. 7. 2023

[35] https://nextjs.org/docs/pages/building-your-application/optimizing/analytics, Analytics, obiskano 23. 7. 2023

[36] https://unlighthouse.dev/, Unlighthouse - Site-wide Google Lighthouse, obiskano 26. 7. 2023

[37] https://vercel.com/pricing, Pricing - Vercel, obiskano 26. 7. 2023