SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Anton Lovrić

Usporedba monolitne i klijent-server arhitekture u razvoju web aplikacija

DIPLOMSKI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Anton Lovrić

JMBAG: 0016135356

Studij: Informatika u obrazovanju

Usporedba monolitne i klijent-server arhitekture u razvoju web aplikacija

DIPLOMSKI RAD

Mentor/Mentorica:

Prof. dr. sc. Neven Vrček

Varaždin, rujan 2024.

Anton Lovrić

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Ovaj diplomski rad će biti primarno orijentiran na analizu različitih arhitekturalnih pristupa prilikom izrade web aplikacija. Postoje brojni mogući pristupi prema dizajnu arhitekture web aplikacija te će u ovom radu najveći fokus biti postavljen na monolitnu i na klijent-server arhitekturu. Također će zbog svoje popularnosti ukratko biti objašnjenja i arhitektura mikroservisa koja je sve češća pojava u procesu razvoja web aplikacija.

Početak rada će biti orijentiran na predstavljanje alata koji će biti korišteni za izradu web aplikacija koje će se koristiti za usporedbu rezultata različitih arhitekturalnih pristupa. Alati koji će se koristiti su Next.js, Express.js i PostgreSQL. Next.js će biti korišten za izradu web aplikacije koristeći monolitnu arhitekturu zbog svoje mogućnosti izvođenja programskog koda na poslužitelju i na strani klijenta. Express.js će biti korišten za izradu vanjskog servisa na strani poslužitelja radi implementacije klijent-server arhitekture. U obje aplikacije će biti korištena PostgreSQL baza podataka zbog svoje brzine, skalabilnosti i sigurnosti.

Aplikacija koja će biti izrađena je *Tech Tales* – web blog koji će sadržavati različite članke vezane za tehnologiju. Nakon uspješne izrade dvije verzije ove web aplikacije od kojih svaka predstavlja drukčiju arhitekturu, provest će se analiza i usporedba monolitne i klijent-server arhitekture s gledišta performansi, složenosti razvoja i kvaliteti iskustva programera.

Ključne riječi: JavaScript, Node.js, web aplikacije, arhitektura softvera, Next.js

Sadržaj

1. Uvod

Razvoj programskih rješenja je dugotrajan proces kojem treba pažljivo pristupiti kako bi krajnji proizvod bio što uspješniji. Sve faze razvoja programskog proizvoda se mogu promatrati kroz životni ciklus softverskog razvoja, koji predstavlja proces za dizajn i izradu kvalitetnih programskih rješenja [1]. Cilj analize životnog ciklusa programskog proizvoda je izrada koraka razvoja i minimiziranje rizika pravilnim planiranjem [1]. Prije početka implementacije programskog rješenja, potrebno je pravilno isplanirati proces razvoja i dizajnirati ključne elemente rješenja kako bi završni proizvod bio što stabilniji. Jedan ključni element koji je potrebno odrediti rano u procesu razvoje proizvoda je arhitektura programskog rješenja jer izbor arhitekture određuje način organizacije rada tijekom faze implementacije i faze održavanja proizvoda.

Izbor pravilne arhitekture u razvoju web aplikacija igra značajnu ulogu u razini kvalitete konačnog proizvoda. Performanse web aplikacija u nekim slučajevima mogu pokazati rast od 30-50%, što je značajna razlika [2]. Arhitektura web aplikacija je okvir za rad koji definira skup principa na temelju kojih će aplikacija biti izrađena te nacrt koji određuje kako će korisničko sučelje komunicirati s poslužiteljem i bazom podataka [2]. Glavni cilj definiranja arhitekture je kreiranje skalabilnog sustava koji će omogućiti ispunjavanje svrhe web aplikacije i pružiti željene performanse i sigurnost svojim korisnicima [2].

Motivacija za izradu ovog diplomskog rada je bolje upoznavanje s arhitekturalnim obrascima koji se koriste u izradi modernih web aplikacija te primjena tih principa u razvoju aplikacija. Osim upoznavanja arhitekturalnih obrazaca, ovaj diplomski rad će biti koristan u upoznavanju temeljnih principa popularnih okvira za rad koji se danas često koriste za izradu web aplikacija. Zbog brojnih mogućih pristupa u izradi arhitekture sustava, u ovom diplomskom radu će fokus biti postavljen na 3 moguća pristupa. Pristupi koji će biti obrađeni su monolitna arhitektura, klijent-server arhitektura i arhitektura mikroservisa. Bit će izrađene web aplikacije pomoću monolitne i klijent-server arhitekture koje će biti detaljno uspoređene, a arhitektura mikroservisa će biti predstavljena kao alternativni pristup izradi web aplikacija koji je sve češći u modernom svijetu razvoja web aplikacija.

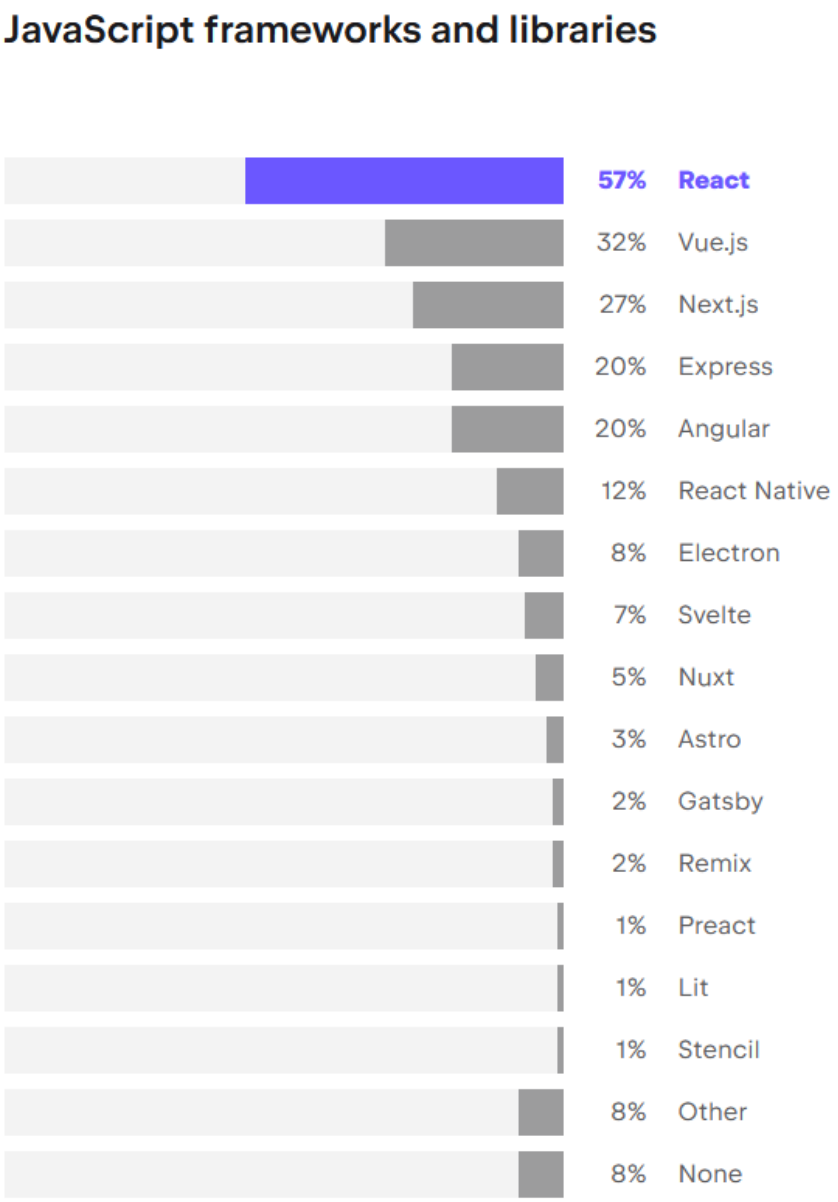
2. Metode i tehnike rada

Praktični dio ovog diplomskog rada predstavlja dizajniranje arhitekture web aplikacije i implementacija web aplikacije. Dizajn arhitekture se može napraviti u različitim alatima, poput Draw.io, Excalidraw, Visual Paradigm itd. Dizajn arhitekture u ovom diplomskom radu će biti izrađen u alatu Excalidraw zbog njegove jednostavnosti, prilagodljivosti i mogućnosti besplatnog korištenja. Izbor alata za izradu dizajna nije ključan faktor jer najveću važnost predstavlja teorijsko poznavanje različitih arhitektura i povezivanje tog znanja s poslovnom domenom.

Moderni ekosustav alata za razvoj web aplikacija je izrazito bogat i postoje brojni alati pomoću kojih je moguće izraditi web aplikacije. Najpoznatiji programski jezik za izradu web aplikacija je JavaScript. JavaScript je programski jezik koji je namijenjen omogućavanju interakcije korisnika s web mjestom koje posjećuje [3]. Programi napisani u JavaScriptu se nazivaju skriptama. Napisane skripte su obrađene JavaScript strojem (eng. engine) koji pretvara naredbe u strojni kod koji se izvršava velikim brzinama. Stroj također primjenjuje različite optimizacije na kod što osigurava maksimalne performanse [4]. JavaScript je inicijalno bio namijenjen izvršavanju na strani korisnika, odnosno izravno u prozoru web preglednika. Rad JavaScripta u web pregledniku se očituje u prepoznavanju brojnih događaja tijekom korištenja aplikacije poput klika miša, navigacija po web mjestu, dodavanje sadržaja na web stranicu itd. Priroda JavaScript-a je vidljiva u generiranju dinamičnog sadržaja koji je odmah vidljiv krajnjem korisniku [3]. Osim izvođenja na strani korisnika, moderni JavaScript se može izvršavati i na strani poslužitelja. Izvršavanje na strani poslužitelja predstavlja evoluciju JavaScript-a i omogućava izradu različitih servisa na strani poslužitelja (engl. backend). Izrada backend servisa je moguća koristeći Node.js engine i ovakav pristup je postao popularna opcija za razvoj modernih web aplikacija.

JavaScript i Node.js omogućavaju razvoj web aplikacija, ali postoje različiti okviri za rad (eng. frameworks) koji olakšavaju razvoj web aplikacija. JavaScript je poznat po svojoj fleksibilnosti i širokoj ponudi biblioteka i okvira za rad između kojih JavaScript programeri mogu birati. React.js je najpopularnija JavaScript biblioteka koju koriste JavaScript programeri [5]. Popularnost React.js biblioteke leži u tome što ju je razvila tvrtka Meta koja ulaže veliku količinu resursa u razvoj React-a. React omogućava podjelu web aplikacija u komponente i brojne optimizacije koje ubrzavaju rad aplikacije te je zbog toga najpopularniji izvor. Next.js je tzv. meta-framework, odnosno okvir za rad izrađen na temelju postojećeg okvira. Next.js je izgrađen na temelju React-a i nadograđuje ga brojnim elementima poput generiranja sadržaja na strani poslužitelja i omogućava korištenje elemenata koji će tek u budućnosti biti dostupni u React-u, kao što su serverske komponente. Serverske komponente su dijelovi koda koji su

generirani na strani poslužitelja, a uvođenjem serverskih komponenti se smanjuje razlika između programiranja na strani poslužitelja i programiranja na strani korisnika. Približavanje ovih pristupa je razlog zašto je Next.js izbor za izradu monolitne aplikacije u ovom diplomskom radu.



Slika 1 popularnost JavaScript okvira za rad (izvor: https://www.jetbrains.com/lp/devecosystem-2023/javascript/)

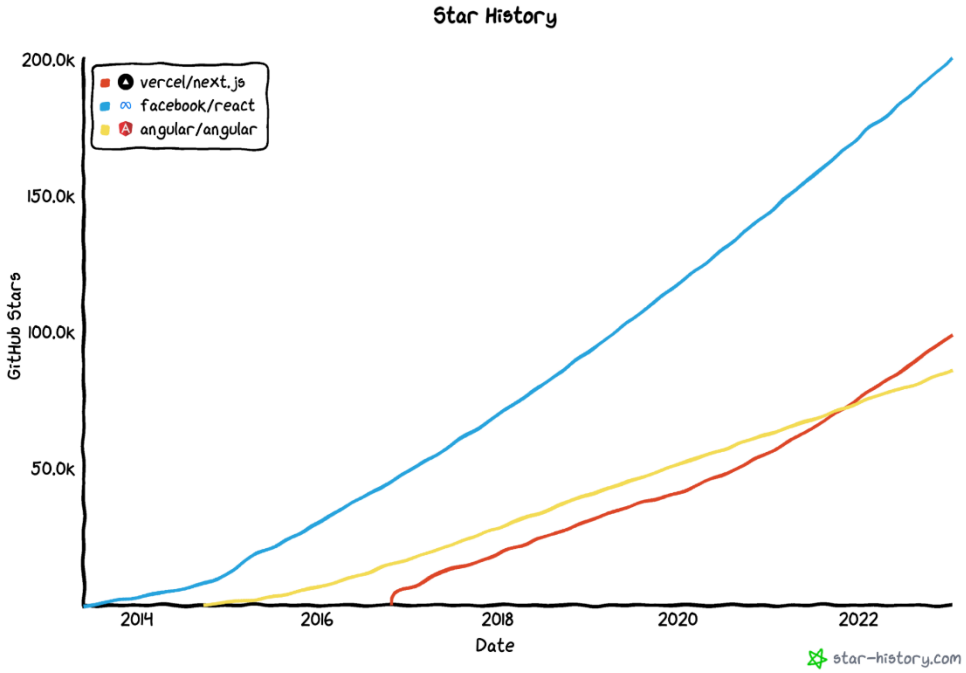
2.1. Next.js okvir za rad

Next.js je meta-framework koji služi za izradu React.js web aplikacija. Next.js je definiran kao full-stack framework jer omogućava pisanje koda koji će se izvoditi na strani

poslužitelja i za pisanje koda koji će se izvoditi na strani klijenta [6]. Next.js nadograđuje React.js brojnim značajkama, među kojima su:

- ruter pomoću kojeg se može lako ponovno iskorištavati programski kod, prikazivati stanja učitavanja, rukovanje greškama itd.
- renderiranje sadržaja na strani poslužitelja i na strani klijenta
- jednostavno dohvaćanje podataka s backend servisa
- podrška za korištenje različitih metoda stiliziranja web stranica (CSS, SCSS, CSS moduli, Tailwind CSS itd.)
- automatska optimizacija fotografija, fontova i skripti
- snažna podrška za korištenje TypeScript-a, koji omogućava statičku provjeru tipova i tako doprinosi održivosti projekta. [6]

Ranije navedene značajke su među glavnim razlozima zašto je Next.js najpopularniji JavaScript okvir za rad, što je moguće promatrati kroz broj zvjezdica na službenom Github repozitoriju..



Slika 2 popularnost JavaScript okvira za rad prema broju Github zvjezdica (Izvor: https://www.elpassion.com/blog/nextjs-what-is-it-and-why-should-you-use-it)

Prednost popularnih alata je u tome što često imaju veliku zajednicu koja koristi taj alat i lakše je pronaći pomoć tijekom razvoja aplikacije. Također je lako pronaći vanjske biblioteke koje proširuju postojeće mogućnosti alata kojeg koristimo što nam omogućava jednostavnije poboljšavanje aplikacije i dodavanje novih značajki.

Glavni razlog korištenja Next.js okvira za rad je njegov pristup serverskim komponentama. Serverske komponente omogućavaju razvoj koda koji će se izvoditi na strani poslužitelja i zbog toga ga je moguće cacheirati [7]. Cacheiranje je mehanizam koji sprema sadržaj koji je već poslan prema korisniku i brzo ga prikazuje ako korisnik ponovno zatraži taj sadržaj. Cacheiranjem ubrzavamo inicijalno učitavanje aplikacije i bržu navigaciju web mjestom jer možemo predvidjeti kako će izgledati sadržaj kojeg treba predstaviti korisniku i unaprijed ga pripremiti. Serverske komponente omogućavaju programeru istovremeni pristup svim resursima na poslužitelju i na klijentu unutar jedne aplikacije, što čini Next.js odličnim kandidatom za izradu web aplikacije monolitne arhitekture.

2.2. Express.js okvir za rad

Express.js je najpopularniji Node.js okvir za rad koji je često korišten za izradu backend servisa [9]. Express.js omogućava brz razvoj aplikacijsko programskih sučelja (API) koji služe za povezivanje aplikacija sa strane klijenta na stranu poslužitelja. Express je definiran kao minimalistički okvir za rad koji dolazi s malim brojem početnih značajki, ali je lako proširiv i lako se skalira potrebama složenijih aplikacija [10].

Glavne prednosti Express.js-a su njegova fleksibilnost i brzina. Zbog svoje minimalističke prirode, programeri mogu prilagođavati Express.js svojim potrebama i proširiti ga bibliotekama koje su njima potrebne [11]. Express je jako skalabilan okvir za rad koji efikasno rukuje brojnim zahtjevima sa strane klijenta. Zbog svoje popularnosti ima veliku zajednicu programera koji snažno podupiru ovaj alat i moguće je pronaći rješenja za brojne probleme [11].

Express.js radi na klijent-server modelu i zbog toga je prikladan alat za implementaciju klijent-server arhitekture u ovom diplomskom radu. Skalabilnost Express-a je njegova glavna prednost u ovom diplomskom radu jer će omogućiti rasterećivanje poslužitelja od dohvaćanja i obrade podataka sadržaja te će dio opterećenja biti prebačen na web preglednike koji mogu rukovati takvim zadacima.

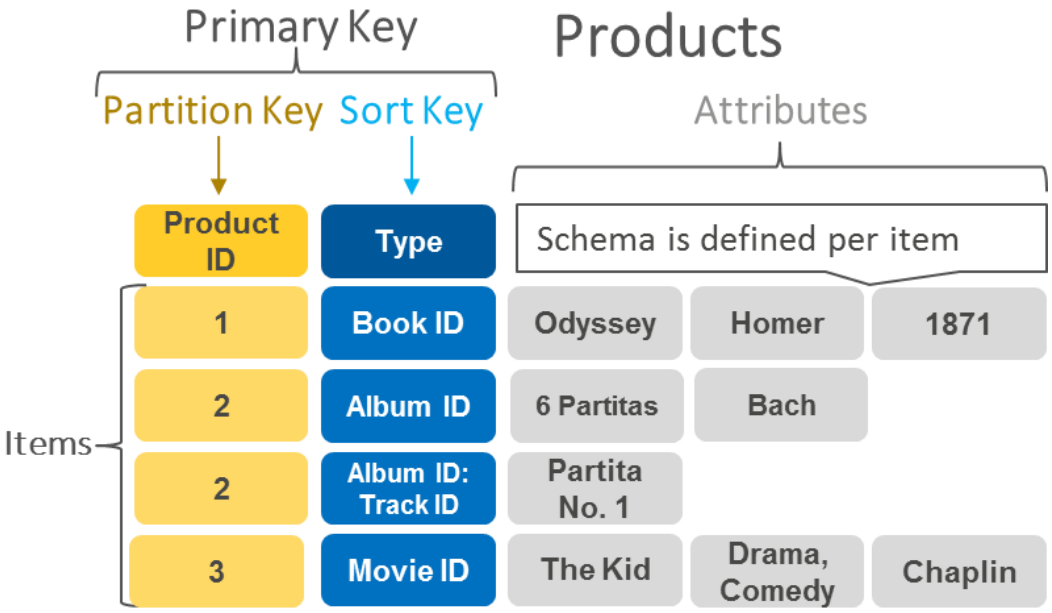
2.3. Baza podataka

Spremanje podataka je potreba većini web aplikacija i postoje brojni načini spremanja podatka. Podatke je moguće privremeno spremiti pomoću kolačića (eng. cookies), lokalnog spremišta (eng. local storage), spremište sesije (eng. session storage) i indeksiranih baza podataka (eng. indexed database). Svi ovi alati su dostupni unutar web preglednika ali ne pružaju mogućnost trajnog spremanja podataka. Trajno spremanje podataka je nužna potreba

web aplikacije koja će biti napravljena u sklopu ovog diplomskog rada kako bi korisnici mogli spremati svoje objave i pregledavati objave drugih korisnika. Zato je potrebno koristiti bazu podataka za spremanje svih podataka vezanih za ovu aplikaciju.

Glavne vrste baza podataka su relacijske i nerelacijske baze podataka. Relacijske baze podataka spremaju podatke u obliku tablica pomoću redaka i stupaca [10]. Tablice predstavljaju različite entitete (npr. korisnik), stupci predstavljaju svojstva entiteta (npr. šifra, ime), reci predstavljaju trajne zapise tih entiteta (npr. „1, Ivan“). Tablice je moguće međusobno povezati pomoći vanjskih ključeva i tako stvoriti vezu između podataka pomoću koje je moguće lako dohvaćati, uređivati i uklanjati podatke (npr. veza između korisnika i objave). Primjeri sustava za upravljanje relacijskim bazama podataka su PostgreSQL, MySQL i sl.

Nerelacijske baze podataka mogu spremati podatke na različite načine poput korištenja ključ-vrijednost parova, dokumentno orijentiranih baza podataka i baza podataka koji spremaju vrijednosti u obliku grafova [10]. Primjer sustava za upravljanje bazom podataka u obliku ključ-vrijednost parova je Redis, koji u navedenom formatu sprema sve podatke u radnu memoriju sustava na kojem je pokrenut. Dokumentno orijentirane baze podataka spremaju podatke u obliku JSON-a, što je jako čitljiv oblik strukturiranja podataka ali može uzrokovati redundanciju podataka [10]. Spremanje podataka u obliku grafa je namijenjeno za spremanje i navigaciju koristeći odnose podataka. Čvorovi predstavljaju entitete dok veze između čvorova definiraju odnose između podataka [10].



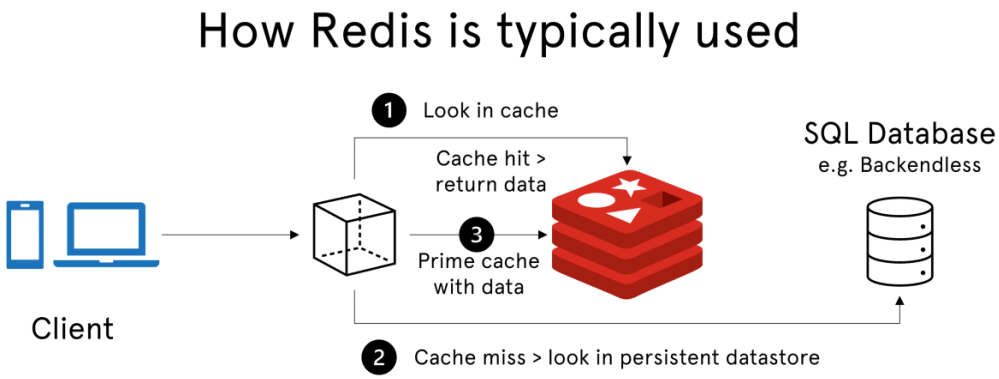
Slika 3 grafički prikaz dokumentno orijentirane baze podataka (izvor: https://aws.amazon.com/compare/the-difference-between-relational-and-non-relational-databases/)

2.4. PostgreSQL

Alat za trajnu pohranu podataka u ovoj aplikaciji će biti PostgreSQL. PostgreSQL je popularni sustav za upravljanje bazama podataka koji je poznat po svojoj pouzdanosti, sigurnosti, brzini i skalabilnosti [11]. PostgreSQL pruža veliki broj mogućnosti poput potpore različitih tipova podataka (npr. JSON i polja objekata), održavanja integriteta podataka i izvođenja složenih upita. PostgreSQL je odabran zbog svoje brzine, sigurnosti te popularnosti. PostgreSQL je jako dugo podržan te kontinuirano evoluira i razvija svoj ekosustav, čime je postao jedan od najboljih izbora za sustava za upravljanje bazom podataka.

2.5. Redis

PostgreSQL pokriva gotovo sve potrebe ove web aplikacije spremanjem podataka vezanih za objave, korisnike koji koriste aplikaciju. Problem kod korištenja PostgreSQL-a je u tome što predstavlja sporiji (ali stabilniji) način rukovanja podacima u usporedbi sa spremanjem u radnu memoriju. Redis je alat koji sprema podatke u memoriju u obliku ključ-vrijednost parova i tako omogućava iznimno brzo zapisivanje podataka i pristup podacima. Ovakav način rada je omogućio Redis-u da postane primarni alat u slučajevima u kojima je potreban brz pristup podacima. Slučajevi korištenja koji imaju ovakve zahtjeve mogu biti cacheiranje podataka, slanje poruka i pružanje pristupa podacima u stvarnom vremenu [12]. Oblik korištenja Redis-a kao alata za cacheiranje je prikazan slikom 4.



Slika 4 primjer korištenje Redis baze podataka u svrhu cacheiranja (izvor: https://backendless.com/redis-what-it-is-what-it-does-and-why-you-should-care/)

Nedostatak korištenja Redis-a je smanjena pouzdanost i integritet zapisa. Budući da su svi podaci spremljeni u radnu memoriju sustava na kojem je Redis instanca pokrenuta, svi podaci će biti izgubljeni nakon prekida rada sustava. Zato Redis nije prikladno rješenje za pohranu podataka ključnih za funkcioniranje sustava (npr. većina podataka o korisnicima i objavama) nego je često korišten kao potporni mehanizam nekom drugom, trajnom obliku spremanja podataka. Ovaj nedostatak Redis-a može biti umanjen povremenim stvaranjem

sigurnosnih kopija podataka, ali takav način rada zahtijeva dodatnu konfiguraciju te i dalje sa sobom nosi smanjeni rizik gubitka dijela podataka.

3. Razrada teme

Arhitektura softvera je složen koncept kojeg je teško definirati. Autori su pokušali definirati arhitekturu softvera, navodeći da arhitektura označava fundamentalnu organizaciju nekog sustava ili način na koji su organizirane najviše komponente u sustavu [13]. U ovom diplomskom radu će se arhitektura web aplikacije promatrati kao sustav najviših komponenti u sustavu i načina na koji te komponente surađuju, poput poslužitelja, klijenta, baze podataka itd.

Ovo poglavlje će biti posvećeno analizi arhitektura koje se koriste u web aplikacijama, praktičnoj primjeni monolitne i klijent-server arhitekture te usporedbi dobivenih programskih rješenja. Nakon općeg opisa arhitektura web aplikacija, posebna važnost će biti podarena analizi monolitne, klijent-server arhitekture i arhitekture mikroservisa. Svaki oblik arhitekture će biti prikazan na primjeru online knjižare radi lakšeg predstavljanja pojedine arhitekture. Bit će prikazan proces izrade web aplikacije koristeći monolitnu i klijent-server arhitekture te uspoređeno po različitim kriterijima, poput performanse aplikacije, procesa i složenosti razvoja te procesa isporuke.

3.1. Arhitekture web aplikacija

Arhitekturu softvera je najjednostavnije definirati kao nacrt određenog sustava [14]. Planiranje arhitekture softvera je ključan korak u procesu razvoja programskog proizvoda. Kvalitetnim planiranjem arhitekture je moguće predvidjeti potencijalne probleme koji mogu proizaći tijekom izrade programskog proizvoda, optimizirati sustav i posljedično ostvariti bolje performanse aplikacije. Početak izrade programskog proizvoda bez prethodnog definiranja arhitekture sustava nije rijetka pojava i većina aplikacija koje su izrađene na takav način imaju sličnu sudbinu. Aplikacije izrađene bez definirane arhitekture posjeduju elemente koji su jako usko povezani i teško zamjenjivi, krhku strukturu aplikacije i neuredan programski kod koji je nastao zbog nedostatka plana i vizije [15]. Nedostatak arhitekture sustava može imati katastrofalne posljedice za poduzeća. Dobar primjer posljedica loše definirane arhitekture je web aplikacija Pets.com, koja je planirala konkurirati popularnoj web trgovini Amazon. Ulagujući ogromnu količinu resursa u marketing, Pets.com se brzo probila na tržište i postigla veliku količinu popularnosti i postala vodećom online trgovinom za prodavanje potrepština za kućne ljubimce. Zbog ogromnog ulaganja u marketing, nije potrošeno dovoljno resursa na

pažljivu izradu infrastrukture aplikacija i ovo je uzrokovalo katastrofalne probleme u pogledu performansi sustava. Velika popularnost aplikacije je omogućila velike brojeve posjeta stranici, narudžbi i transakcija te infrastruktura nije bila dorasla ovom izazovu. Aplikacija je postala iznimno spora, transakcije su izgubljene i tvrtka je na kraju zatvorena. Pets.com služi kao primjer da prevelika razina uspjeha može biti kobna za aplikaciju koja nema pomno definiranu arhitekturu sustava da se nosi s obilnim prometom koji taj uspjeh donosi. [16]

Definiranje arhitekture programskog proizvoda ima ključnu ulogu u procesu razvoja, implementacije i održavanju programskog proizvoda. Arhitektura programskog proizvoda predstavljanja dijeljenu viziju i razumijevanje sustava koje posjeduju svi pojedinci uključeni u proces razvoja proizvoda. Tipovi pojedinaca koji su uključeni u proces razvoja programskog proizvoda su programeri, menadžeri, poslovni analitičari i svi ostali koji aktivno doprinose procesu razvoja proizvoda [15]. Budući da su svi ovi članovi tima uključeni u proces razvoja proizvoda, trebali bi biti uključeni u proces izrade arhitekture softvera i dobro biti upoznati s izrađenom arhitekturom. Poznavanje načina na koji funkcionira sustav je jako korisno svim članovima tima kako bi preciznije definirali viziju proizvoda koji izrađuju i bili efikasniji u komunikaciji i planiranju razvoja proizvoda. Arhitektura proizvoda predstavlja zajednički jezik svih članova tima koji razvija softver. Važnost poznavanja arhitekture sustava iz gledišta različitih članova tima je prikazana tablicom 1.

Tablica 1 prednosti poznavanja arhitekture sustava od strane članova tima

Pozicija	Važnost poznavanja arhitekture
Programer	• lakše dodavanje novih elemenata aplikacije • brže pronalaženje problema u aplikaciji i njihovo rješavanje
Menadžer	• poznavanje arhitekture omogućava kvalitetniju organizaciju rada • lakša koordinacija zadataka i rukovanje resursima
Poslovni analitičar	• detaljnija analiza pojedinog elementa sustava • identifikacija kritičnih elemenata sustava
DevOps stručnjak	• optimiziranje procesa isporuke aplikacija • rukovanja tehničkim resursima ključnih za rad aplikacije
Tester	• lakše identificiranje elemenata koje je potrebno testirati nakon svake isporuke • preciznije pisanje automatskih testova • ubrzavanje procesa testiranja aplikacije

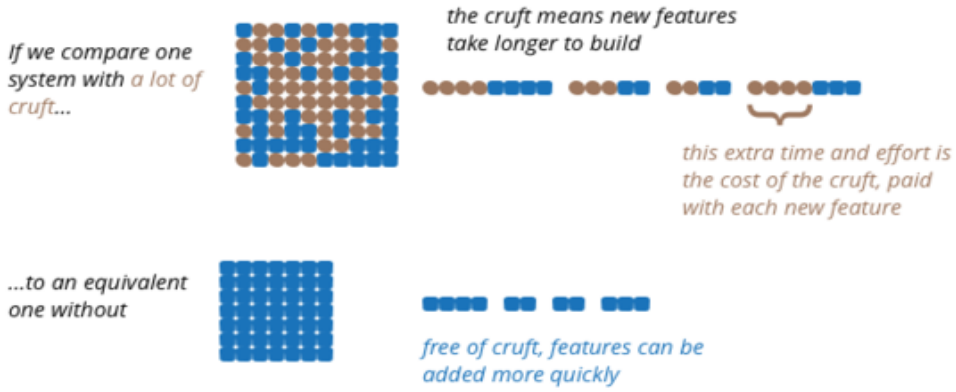
Jasno definiranje arhitekture sustava također pomaže u procesu donošenja tehničkih odluka i u održavanju konzistencije kroz programski kod [14]. Arhitekturalne odluke mogu izravno utjecati na različite tehničke detalje na aplikativnoj razini. Npr. korištenje vanjskog servisa za autentifikaciju i autorizaciju zahtijeva drukčiji način provođenja autentifikacije i autorizacije korisnika. Strogo definiranje arhitekture također utječe na konzistentno korištenje uzoraka tijekom implementacije proizvoda [14]. Konzistentnost u pisanju programskog koda značajno utječe na održivost i skalabilnost projekta, pogotovo ako postoji veliki broj programera koji rade na projektu. Miješanje stilova programiranja je jako česta pojava u sustavima u kojima nema jasno definiranih uzoraka razvoja softvera, a ispravljanje teškoća nastalih miješanjem uzoraka je dugotrajan proces koji usporava razvoj aplikacije. Zato je potrebno preventivno djelovati i spriječiti ove teškoće korištenjem jasno definirane strukture projekta.

Budući da su moderni sustavi izrazito kompleksni po svojoj prirodi, potrebno je aktivno nastojati smanjiti kompleksnost tih sustava pravilnim definiranjem arhitekture proizvoda. Izradom arhitekture sustava se sustav dijeli na manje jedinice, što olakšava nadgledanje sustava, njegovu analizu i rukovanje sustavom. Arhitektura sustava također jasno naglašava kako pojedine komponente zajedno funkcioniraju te je zbog toga lakše održavati sustav i omogućiti novim članovima tima da brzo razviju dovoljnu razinu poznavanja sustava kako bi počeli aktivno doprinositi njegovom razvoju [14].

Arhitektura softvera također ima značajan utjecaj i na njegovu kvalitetu. Metrike poput performanse, pouzdanosti, skalabilnosti i sigurnosti sustava su pod velikim utjecajem arhitekture sustava [14]. Npr. uvođenje dodatnog sloja autentifikacije koji implementira višefaktorsku autentifikaciju ima značajan utjecaj na poboljšanje sigurnosti tog sustava. Također, korištenje centraliziranih sustava može imati negativan utjecaj na pouzdanost sustava. Budući da je sustav dostupan na jednom mjestu nedostupnost s tog mjesta označava potpunu nedostupnost sustava, dok korištenje distribuiranih sustava uklanja ovaj problem jer zahtijeva korištenje modula koji ne ovise strogo jedni o drugima. U slučaju rušenja jednog dijela sustava, moguće je početi s rješavanjem tog problema dok korisnici koriste druge dijelove aplikacije koje nisu pod utjecajem tog neispravnog modula. Korištenje distribuiranih sustava je karakteristično za arhitekturu mikroservisa, što je jako čest pristup u modernom razvoju aplikacija.

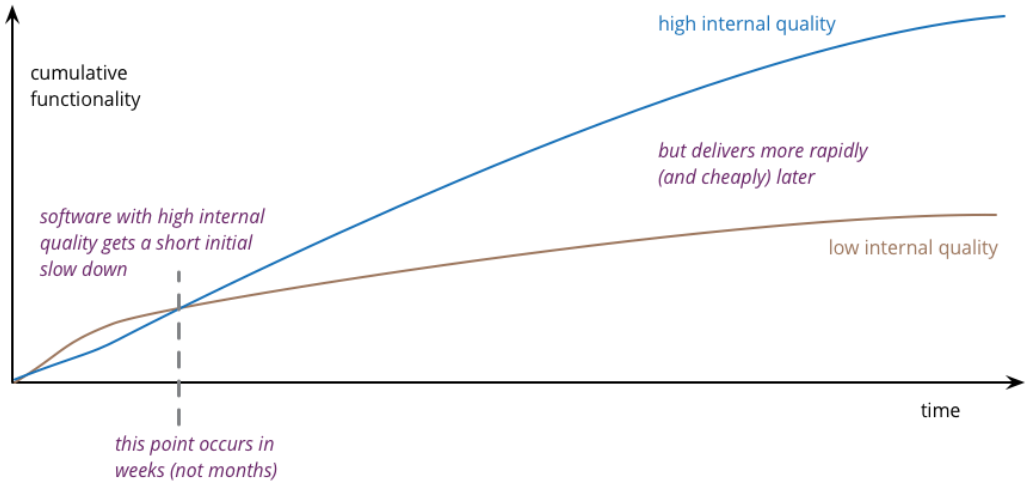
U procesu razvoja programskih proizvoda, najvažnija stavka je zadovoljstvo korisnika. Korisnici vjerojatno neće nikad upoznati arhitekturu većine aplikacija koje koriste, ali će svakako osjetiti negativne utjecaje koje stvara loše izrađena arhitektura. Osim problema vezanih za performanse, pouzdanost, skalabilnost i sigurnost koji su ranije spomenuti, loša arhitektura sustava također usporava proces izrade programskog proizvoda [13]. Loša

arhitektura značajno doprinosi stvaranju tehničkog duga kojeg je potrebno postupno rješavati. Rješavanje tehničkog duga je zahtjevan proces i otežava dodavanje novih značajki koje su potrebne korisnicima.



Slika 5 utjecaj tehničkog duga na dodavanje novih funkcionalnosti (izvor: https://martinfowler.com/architecture/)

Okruženja koja traže izrazito brz razvoj funkcionalnosti često zahtijevaju različite kompromise. Kvaliteta arhitekture sustava je često žrtvovana u svrhu bržeg razvoja programskog proizvoda jer naizgled ne utječe na korisničko iskustvo, ali manjak interne kvalitete sustava u konačnici stvara brojne probleme, kao što je prikazano i slikom 5. Uzimajući u obzir sve ranije navedene prednosti kvalitetne arhitekture sustava, važno je napomenuti da se nedostaci nekvalitetne arhitekture jako rano osjete, čak unutar nekoliko tjedana razvoja [13]. Pravilna izrada arhitekture sustava je vremenski zahtjevan proces te ga se iz tog razloga ponekad zanemaruje na početku procesa razvoja programskog proizvoda, ali važno je naglasiti da korištenje dobre arhitekture sustava donosi dugotrajne benefit koji nadmašuju vremenske zaostatke do kojih može doći na početku procesa razvoja proizvoda. Slika 6 prikazuje okvirni omjer između kumulativnog dodavanja funkcionalnosti i vremena razvoja proizvoda.

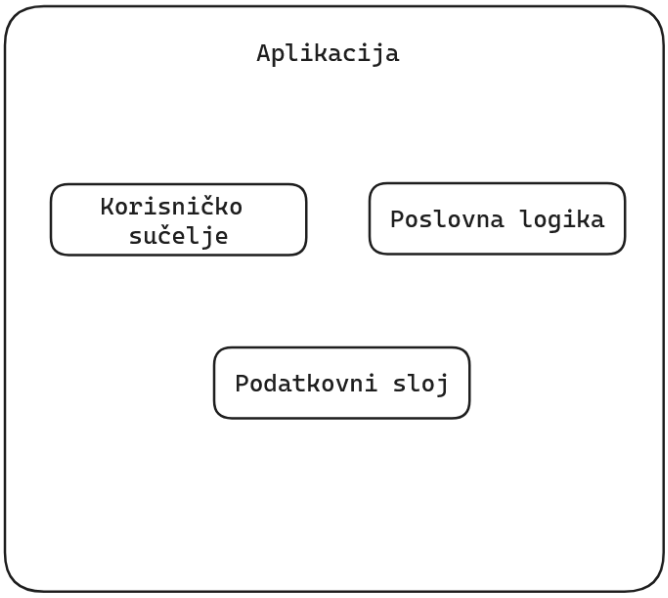


Slika 6 omjer između kumulativnog dodavanja funkcionalnost i vremena razvoja programskog proizvoda (izvor: https://martinfowler.com/articles/is-quality-worth-cost.html)

3.2. Monolitna arhitektura

Monolitna arhitektura je jedan od najranijih oblika arhitekture softverskih proizvoda. Teško je definirati tko je osmislio koncept monolitne arhitekture te se zasluge za njezino nastajanje mora dodijeliti većem broju autora. Prvi oblici monolitnih arhitektura su bile vidljive tijekom razvoja IBM računalnih sustava 1960.-ih i 1970.-ih godina [18]. Monolitne aplikacije su i dalje aktualne te su zato bitna stavka ovog diplomskog rada i nezaobilazne su u diskusijama o arhitekturama softvera. Veliki broj aplikacija automatski poprima izgled monolitne arhitekture zbog jednostavnog pristupa koji pruža monolitna arhitektura. Zbog svoje jednostavnosti je prikladna manjim timovima koji rade na aplikacijama od male do srednje veličine.

Riječ „monolit“ označava nešto cjelovito što je izrađeno iz jednog komada. Prema samoj definiciji riječi, moguće je zaključiti da monolitna arhitektura predstavlja cjeloviti sustav koji samostalno djeluje kao jedna cjelina. Monolitne aplikacije predstavljaju tradicionalni model softvera koji je izgrađen kao samostalna jedinica neovisna o drugim aplikacijama [18]. Monolitne aplikacije uglavnom imaju sav kod napisan unutar jednog repozitorija te je kod u većini slučajeva usko povezan. Uska povezanost koda može otežati proces dodavanja novih funkcionalnosti i zamjenu postojećih elemenata sustava. Shema monolitne arhitekture je prikazana sljedećom slikom.



Slika 7 monolitna arhitektura (izvor: vlastita izrada)

Cjelovita priroda monolitnih arhitektura zahtijeva podjelu resursa između svih komponenti. Umjesto pojedinačne dodjele resursa, komponente monolitne aplikacije moraju koristiti iste resurse koji pokreću aplikaciju te zato može doći do prekomjernog korištenja resursa. Radi izbjegavanja neočekivanih rušenja aplikacije, potrebno je konstantno pratiti potrošnju resursa. Postoje brojni mehanizmi u modernim web aplikacijama koji mogu prouzročiti neočekivanu razinu potrošnje resursa poput radne memorije. Neki od tih mehanizama su zauzimanje resursa koji se više ne koriste (npr. slušatelji događaja), neispravno cacheiranje, neoptimizirani programski kod itd. Ako se koristi monolitna arhitektura prilikom izrade web aplikacije, potrebno je uzeti sve ove elemente u obzir i pravilno isplanirati koji resursi su potrebni za neometano izvršavanje aplikacije.

Osim dijeljena resursa, sveobuhvatna priroda monolitnih arhitektura utječe i na način isporuke. Budući da monolitne aplikacije obuhvaćaju sve što im je potrebno za rad, njihova isporuka je uglavnom jednostavna. Jednostavnost ovog procesa je veliki benefit monolitnih aplikacija jer olakšana isporuka omogućava brzo rješavanje problema koji su nastali prethodnom isporukom. Također je lakše odrediti koji resursi su potrebni za rad aplikacije i potrebno je rukovati samo s jednim poslužiteljem na kojem se isporučuje aplikacija, za razliku od arhitekture mikroservisa koja zahtijeva posebnu jedinicu za svaki izrađeni servis.

3.2.1. Prednosti i nedostaci

Monolitna arhitektura je jako popularan oblik strukturiranja web aplikacija koji je duboko istražen. Glavna prednost monolitnih arhitektura je brzina razvoja softvera. Budući da postoji

jedan repozitorij programskog koda, nije potrebno trošiti vrijeme na pokretanje većeg broja servisa potrebnih za razvoj aplikacije [18]. Za razliku od arhitekture mikroservisa, promjene napravljene u monolitnoj arhitekturi je lakše nadgledati jer se događaju na jednom mjestu, što olakšava upravljanje projektom. Pisanje automatiziranih testova je također jednostavniji proces u sklopu monolitne arhitekture jer se sav kod kojeg je potrebno testirati nalazi u jednom repozitoriju. Osim pisanja automatiziranih testova, lakše je i ručno testirati aplikaciju u lokalnom okruženju zbog manjeg broja elemenata o kojima sustav ovisi [18]. Velika većina prednosti monolitnih aplikacija su uzrokovane jednostavnošću koju donosi monolitna arhitektura. Grupiranje svih komponenti sustava u jednu cjelinu omogućava ujednačen pristup aplikaciji i značajno ubrzava proces razvoja proizvoda na njegovom početku. Brzina početka razvoja je najčešće glavni razlog implementacije monolitne arhitekture i zato je popularan izbor firmama u svom začetku i manjim timovima koji surađuju na projektu. Ovakvim pristupom je moguće izbjeći intenzivno trošenje resursa i izdavanje inicijalne verzije aplikacije koja može biti iznimno važna za početak rada firme.

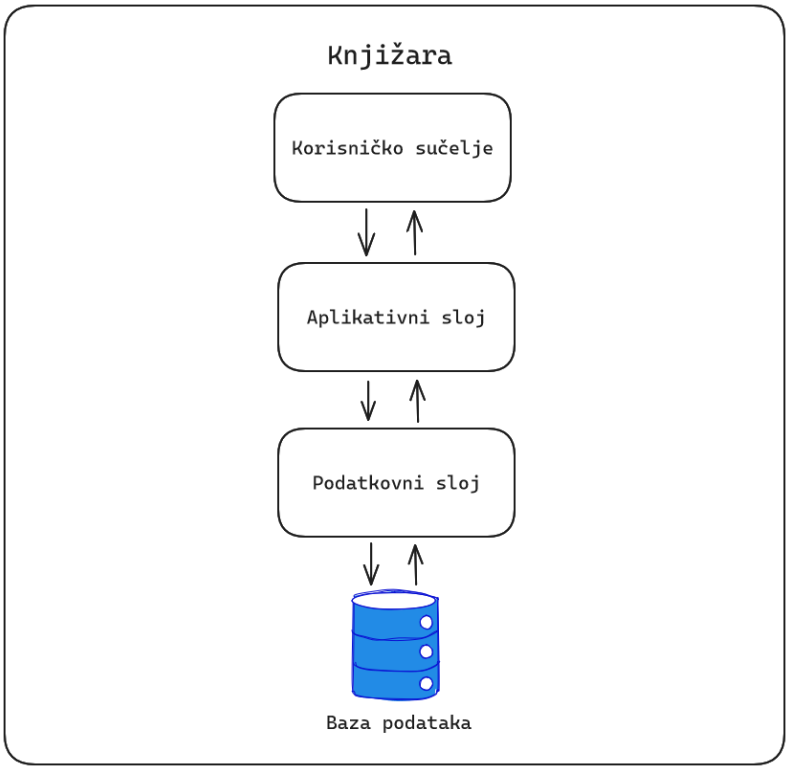
Unatoč jednostavnosti i efikasnosti monolitnih aplikacija, postoje i određeni nedostaci ovakvog pristupa. Nedostaci monolita se najviše očituju u kasnijim fazama životnog ciklusa proizvoda jer je s vremenom sve teže održavati složene aplikacije. Glavni problem koji nastaje je otežana skalabilnost proizvoda [18]. Budući da monolitnu aplikaciju tvori jedna cjelina koja obuhvaća sve komponente sustava, te komponente nije moguće samostalno razvijati. Dodana kompleksnost ovakvih aplikacija također usporava performanse aplikacija i brzinu razvoja što predstavlja značajan problem u svakoj organizaciji. Usko vezanje svih komponenti usporava razvoj aplikacije i dodatno usporava sve druge procese vezane za programski proizvod, poput testiranja. Budući da su svi elementi međusobno vezani, nakon uspješne isporuke nove aplikacije potrebno je ponovno testirati cijelu aplikaciju jer ne možemo sa sigurnošću utvrditi da nije došlo do neželjenih nuspojava tijekom razvoja određenog modula aplikacije. Također, rušenje jednog modula označava rušenje cijele aplikacije. Ovo je jako riskantan pristup i predstavlja veliku prijetnju pouzdanosti sustava. Zato je potrebno uložiti dodatan trud u zaštitu sustava od vanjskih napada i pravilno rukovati sve greške koje nastanu tijekom rada sustava.

Tablica 2 prednosti i nedostaci monolitne arhitekture (izvor: vlastita izrada)

Prednosti i nedostaci monolitne arhitekture	
Prednosti	**Nedostaci**
brzina razvoja	manjak skalabilnosti
jednostavnost	uska povezanost komponenti
lakše testiranje	smanjena pouzdanost

3.2.2. Primjer

Korištenje monolitne arhitekture za izradu aplikacije za vođenje poslovanja online knjižare je prikladan izbor zbog svoje jednostavnosti. Uključivanje svih funkcionalnosti koje traži ova aplikacija u jednu cjelinu nije prevelik izazov i predstavlja dobar način strukturiranja aplikacije.



Slika 8 model monolitne arhitekture knjižare (izvor: vlastita izrada)

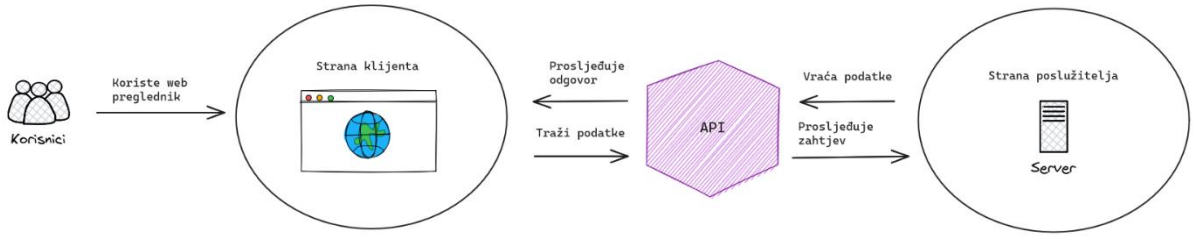
Budući da su monolitne arhitekture uglavnom jako rigidne i ne dopuštaju veliku fleksibilnost u svojoj izradi, dijagrami monolitnih arhitektura će izgledati jako slično za većinu aplikacija. Tako je i dijagram na slici 8 jako sličan konceptualnom dijagramu sa slike 7. Korisničko sučelje služi za prikaz podataka i komunicira s aplikativnim slojem kako bi dohvatio sve elemente potrebne za prikaz podataka. Aplikativni sloj dohvaća sve podatke iz baze podataka, obrađuje ih i prosljeđuje korisničkom sučelju.

3.3. Klijent-server arhitektura

Klijent-server arhitektura (ili razdvojena arhitektura), je oblik arhitekture web aplikacija u kojima se dio sustava odgovoran za prikaz sadržaja (frontend) i dio sustava koji se izvršava na poslužitelju (backend) promatraju kao dva neovisna člana sustava. Javlja se kao prirodna

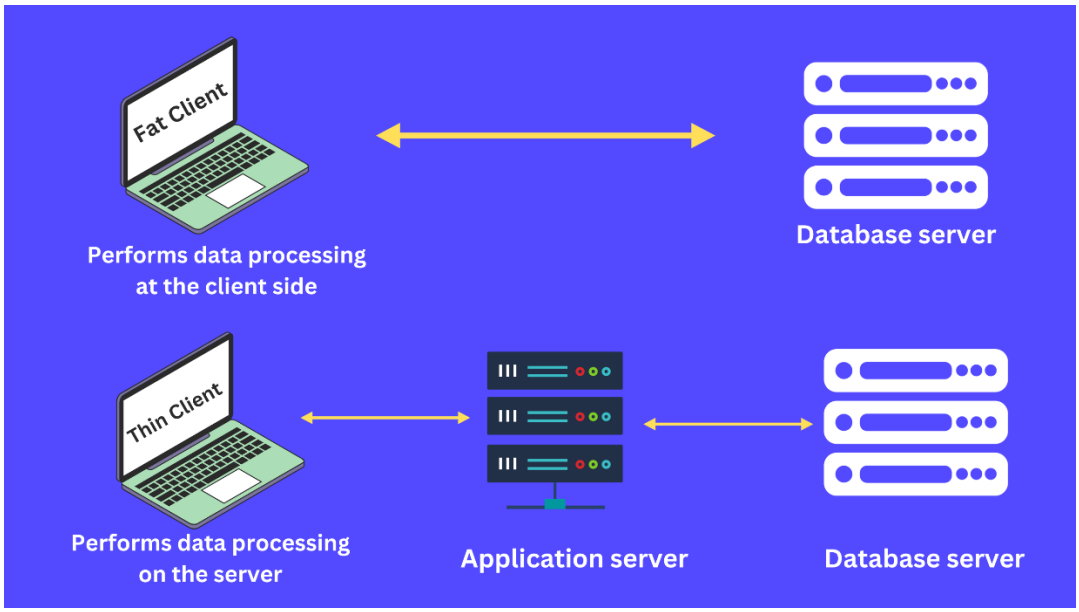
nadogradnja monolitne arhitekture i nastoji riješiti glavne nedostatke monolitne arhitekture. Razdvajanjem frontend i backend dijela aplikacije, komponente sustava više nisu tako strogo vezane i mogu biti tretirani kao odvojeni dijelovi sustava. Ovakav način rada omogućava znatno veću skalabilnost sustava i olakšava rad u timovima srednje veličine.

Razdvojena arhitektura je moderniji oblik izrade web aplikacija nastao potrebom za razdvajanjem web aplikacije na klijenta i poslužitelja. Budući da klijent i poslužitelj nisu pokrenuti na istom poslužitelju, potrebno je uspostaviti način komuniciranja između klijenta i poslužitelja. Klijent i poslužitelj najčešće komuniciraju putem sučelja za programiranje aplikacija (eng. application programming interface), odnosno API-ja. API je mehanizam koji omogućava komunikaciju s određenim resursom. API je implementiran kao dodatni sloj apstrakcije koji sakriva implementaciju od vanjskog korisnika ali mu omogućava pristup resursima koji su mu potrebni i tako omogućava lakšu izradu složenih funkcionalnosti [19]. Komunikacija između klijenta i poslužitelja se najčešće provodi HTTP zahtjevima, gdje API služi kao posrednik.



Slika 9 klijent-server arhitektura (izvor: vlastita izrada)

Budući da je monolitna arhitektura bila značajno orijentirana prema poslužitelju, klijent-server arhitektura gradi svoju popularnost razvojem tehnologija vezanih za programiranje na strani klijenta. JQuery je jedna od najstarijih biblioteka za programiranje na strani klijenta koja je omogućila programerima brojne pogodnosti prilikom razvoja aplikacija na strani klijenta. Dodanim mogućnostima dolazi do fleksibilnosti koja omogućava podjelu opterećenja između klijenta i servera. Javlja se koncept debelog i tankog klijenta. Ovi nazivi označavaju razinu opterećenja koja se prenosi na stranu klijenta. Debeli klijent sadrži znatno veću razinu obrade podataka te aplikacijske i poslovne logike, dok tanki klijent tvori samo prezentacijski sloj i resursi se ne troše na snagu obrade podataka [20]. Budući da se korištenjem tankog klijenta podaci obrađuju na strani poslužitelja i šalju nazad do klijenta, ovakav pristup štedi resurse klijenta ali dodatno opterećuje mrežu preko koje se šalju podaci, dok se prilikom korištenja debelog klijenta javlja suprotna situacija.



Slika 10 debeli i tanki klijent (izvor: https://www.designgurus.io/answers/detail/what-is-thick-client-vs-thin-client)

3.3.1. Prednosti i nedostaci

Glavnu prednost klijent-server arhitekture tvori razdvojenost klijenta i poslužitelja. Razdvajanjem sustava u dvije komponente je poboljšana skalabilnost sustava. Obje komponente je moguće razvijat drukčijom brzinom i prilagođavati potrebama aplikacije. Također je unaprijeđena fleksibilnost sustava i moguće je oblikovati arhitekturu na drukčije načine. Moguće je iskoristiti model debelog ili tankog klijenta ovisno o potrebama aplikacije. U slučajevima gdje su dostupni značajni resursi za obradu podataka na strani klijenta, debeli klijent je moguće koristiti za uštedu vremena razvoja strane poslužitelja i smanjivanje opterećenja na mrežu i poslužitelja. Primjena tankog klijenta je jako korisna ako se većina poslovne logike nalazi na poslužitelju i klijent je korišten kao prezentacijska stranica koja ne nosi odgovornost za obradu podataka. Klijent-server arhitektura također donosi tehničku fleksibilnost u smislu da aplikacija nije strogo vezana za jednu tehnologiju. Moguće je koristiti drukčije tehnologije na strani klijenta i na strani poslužitelja i mijenjati te tehnologije ovisno o potrebama. Korištenje odvojenih repozitorija programskog koda je prikladno za timove od male do srednje veličine jer je moguće podijeliti programere u manje timove koji razvijaju funkcionalnosti na odvojenim repozitorijima. Programski kod je pregledniji jer je logika podijeljena na dva mjesta i tako je moguće postići veću razinu modularnosti. Modularnošću se javlja podjela opterećenja te je moguće postići bolje performanse aplikacije u slučajevima gdje brzine mreže to podržavaju. Testiranje je također olakšano, jer je moguće lakše identificirati na što se odnose napravljene promjene i tako zaobići testiranje dijelova koda koji se nisu promijenili.

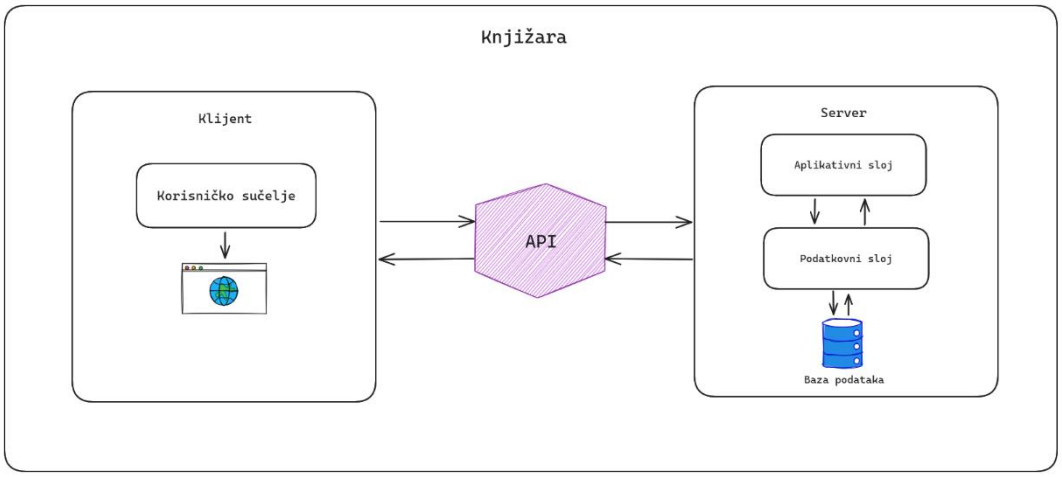
Nedostaci klijent-server arhitekture se uglavnom odnose na složenost održavanja koda, usporavanje razvoja, otežanu isporuku i više troškove resursa. Programski kod je kompliciranije održavati na dva repozitorija i potrebna je dobra koordiniranost i komunikacija između timova. Komunikacija je ključna kako bi svi članovi tima bili svjesni koji elementi se razvijaju i na kojem repozitoriju se nalaze kako ne bi došlo do dupliciranja koda ili pojave nekorištenog koda. Ovakav pristup usporava razvoj aplikacije jer programeri moraju obraćati pažnju na više mjesta. Proces razvoja je dodatno usporen zbog otežanog procesa pronalaženja i ispravljanja grešaka u kodu. Korištenje više repozitorija otvara mogućnost pojave grešaka na različitim mjestima te programeri moraju biti vješti u oba aspekta razvoja web aplikacija kako bi pravilno identificirali izvor grešaka. Praćenje promjena napravljenih u kodu je također otežano korištenjem više repozitorija i stvara se veće kognitivno opterećenje nad programerima koji prate i odobravaju napravljene promjene. Isporuka aplikacije je sada kompliciranija nego prilikom korištenja monolita jer postoje dva odvojena servisa koji je potrebno zasebno isporučiti na drugim strojevima i uspostaviti komunikaciju između njih. Isporuka također zahtijeva više strojeva na kojima će biti pokrenute aplikacije, što izaziva i veće troškove održavanja.

Tablica 3 prednosti i nedostaci klijent-server arhitekture (izvor: vlastita izrada)

Prednosti i nedostaci klijent-server arhitekture	
Prednosti	**Nedostaci**
skalabilnost	složenost održavanja
fleksibilnost	viši troškovi
mogućnost korištenja različitih jezika	sporiji razvoj
razdvojenost strane klijenta i poslužitelja (preglednost programskog koda)	zahtijeva bolje tehničko znanje nego monolitna arhitektura
	složenost isporuke

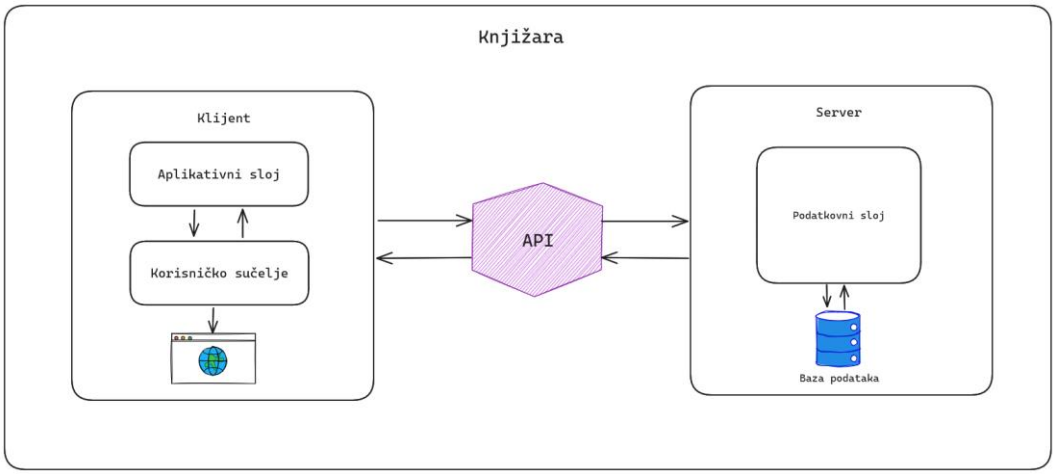
3.3.2. Primjer

Klijent-server arhitektura pruža veću fleksibilnost od monolitne arhitekture i zbog toga postoji više mogućnosti kako modelirati sustav knjižare. Moguće je koristiti model tankog ili debelog klijenta ovisno o potrebama aplikacije i infrastrukturi na kojoj gradimo sustav.



Slika 11 model klijent-server arhitekture knjižare (tanki klijent) (izvor: vlastita izrada)

Tanki klijent označava da će se na klijentu jedino izvršavati akcije koje se odnose na korisničko sučelje, odnosno sadržavat će samo prezentacijski sloj aplikacije. Aplikativni i podatkovni sloj aplikacije se nalazi na strani poslužitelja te se komunikacija između klijenta i poslužitelja izvodi putem API-ja i HTTP poziva.

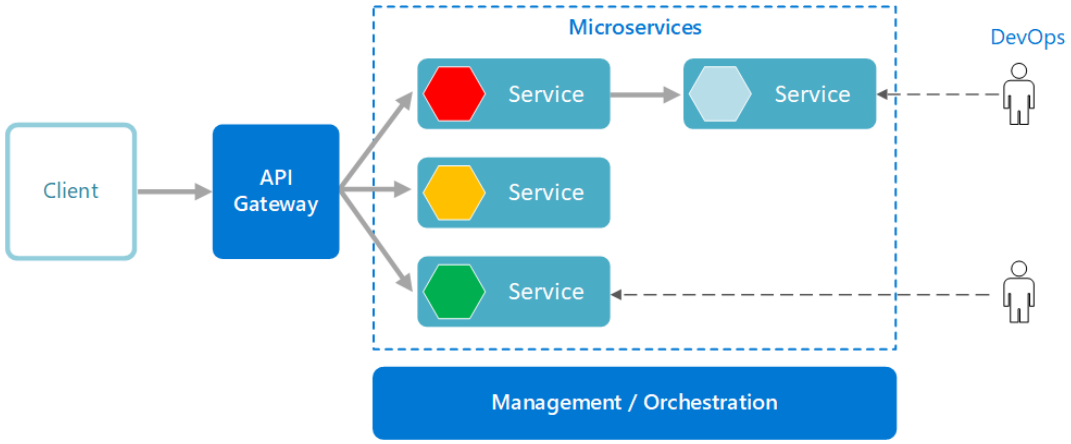


Slika 12 model klijent-server arhitekture knjižare (debeli klijent) (izvor: vlastita izrada)

Korištenjem modela debelog klijenta prebacujemo aplikativni sloj na stranu klijenta što označava da se većina operacija izvodi na klijentskoj strani. Podatkovni sloj vrši dohvaćanje i zapisivanje podataka u bazu podataka te putem API-ja vraća podatka klijentu koji obrađuje podatke prema definiranoj poslovnoj logici i prosljeđuje ih korisničkom sučelju na kojem se podaci prikazuju.

3.4. Arhitektura mikroservisa

Dr. Peter Rodgers 2005. godine postavlja snažne argumente protiv tradicionalnog načina strukturiranja softvera i prvi put koristi izraz „mikro-web servisi“. Ističe važnost duboke granulacije servisa i komponenti sustava radi postizanja fleksibilnosti i pojednostavljenja sustava [21]. Ova ideja je bila ključna za početak razvoja arhitekture mikroservisa, koja je prvi put predstavljena 2011. godine [21]. Arhitektura mikroservisa je oblik arhitekture koja sadrži veliki broj manjih, autonomnih servisa [22]. Organiziranje softvera u obliku mikroservisa je moderan i jako popularan oblik organizacije softvera. Servisi predstavljaju manje logičke cjeline aplikacije koji funkcioniraju neovisno o drugim servisima i tako su slabo vezani. Svaki servis je napisan u odvojenom repozitoriju i zasebno su isporučeni i pokrenuti.

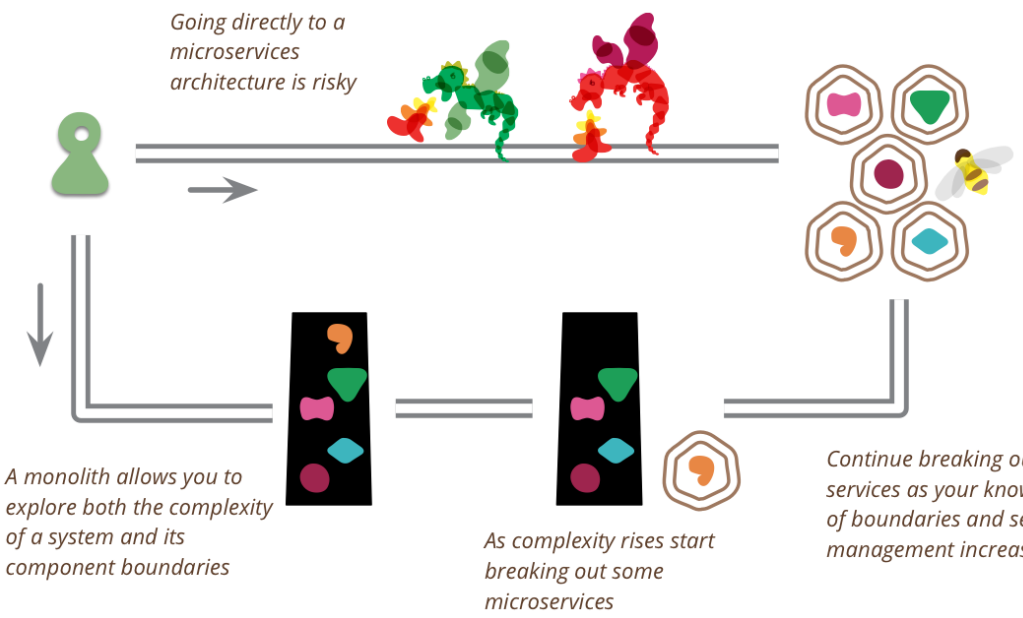


Slika 13 arhitektura mikroservisa (izvor: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices)

API pristupnik služi kao pristupna točka klijentu koji želi koristiti servise. Pristupnik prosljeđuje poziv odgovarajućim servisima koji obrađuju zahtjev klijenta. Osim korištenja API pristupnika, servisi mogu izravno komunicirati s drugim servisima. SOAP i REST protokoli su bili ključni za razvoj mikroservisa jer su omogućili jednostavan oblik pristupanja mikroservisima. SOAP (Simple Object Access Protocol) je oblik komuniciranja putem HTTP-a kojeg je izmislio Microsoft 1999. Korištenje SOAP protokola je bilo jako popularno zbog njihove prilagodljivosti i jednostavnosti, ali nije skalabilan oblik komunikacije [21]. Unatoč tome što je SOAP protokol i dalje korišten u nekim slučajevima, uglavnom je zamijenjen REST protokolom. REST (Representational State Transfer) je protokol prvi put predstavljen 2000. godine u Roy Fieldingovom djelu „Architectural Styles and the Design of Network-based Software Architectures“ [21]. Fielding u svom djelu ističe da je REST arhitekturalni stil namijenjen distribuiranim sustavima [23]. REST protokol naglašava važnost fleksibilnosti ii korištenja istog URL-a za pristup podacima i korištenje drukčijih HTTP metoda za korištenje drukčijih akcija. HTTP metode podržane u REST protokolu su POST, GET, PUT i DELETE [23]. Fleksibilnost REST-a ga čini kompatibilnim s konceptom mikroservisa i omogućava jednostavan način korištenja svih dostupnih resursa.

Mikroservisi svojom modularnošću predstavljaju potpunu suprotnost monolitnoj arhitekturi koja predstavlja softver kao cjelovitu i robusnu jedinicu. Prednost mikroservisa je u tome što predstavljaju skalabilniji sustav i zbog toga brojna poduzeća prelaze s monolitne arhitekture na arhitekturu mikroservisa. Jedno od tih poduzeća je Atlassian koji je 2018. godine odlučio preseliti sustave Jira i Confluence na mikroservisnu arhitekturu koja je pokrenuta putem AWS-a (Amazon Web Services) [18]. Osim Atlassian-a, neka druga poduzeća koja su napravila sličnu migraciju su Amazon, Netflix, eBay i Uber [24]. Sva ova poduzeća su se našla u istom arhitekturalnom problemu. Kreirali su aplikaciju u obliku monolita koja je postala iznimno popularna i bila suočena s velikim opterećenjem. Monolitna arhitektura im nije pružila skalabilnost koja je bila potrebna stoga su morali migrirati na pogodniji oblik arhitekture. Prijelazom na mikroservise su sva ova poduzeća smanjila svoje troškove, pojednostavnila programski kod i poboljšala performanse [24].

Promatranjem ranije spomenutih slučajeva u kojima su poduzeća uspješnom migracijom na arhitekturu mikroservisa ostvarila pogodne rezultate se postavlja pitanje: „Zašto ne započeti s arhitekturom mikroservisa umjesto monolita i tako izbjeći proces migracije?“. Razlog ovome je što rijetko koja aplikacija u svom začetku ima potrebu za skaliranjem, što je glavni problem koji mikroservisi rješavaju. Početnim razvojem aplikacije u obliku monolita nam omogućava brz razvoj aplikacije, otkrivanja složenosti aplikacije te ograničenja aplikacije i svake komponente te aplikacije. Tek nakon što dobijemo cijelu sliku o aplikaciji, možemo po potrebi početi s migracijom na arhitekturu koja više odgovara situaciji u kojoj se nalazi aplikacija.



Slika 14 prijelaz s monolitne arhitekture na arhitekturu mikroservisa (izvor: https://martinfowler.com/bliki/MonolithFirst.html)

Postoje različiti pristupi migracije s monolitne arhitekture na arhitekturu mikroservisa. Jedan pristup je pažljiva izrada monolita uzimajući u obzir modularnost sustava kako bi prijelaz na mikroservise bio jednostavniji. Drugi pristup je postepeno odvajanje komponenti sustava u odvojene servise i zadržavanje dijela monolita kao centralne jedinice. Monolit u ovakvoj situaciji poprima pasivniju ulogu i većina daljnjeg razvoja se odvija u odvojenim servisima. Treći pristup je potpuno napuštanje monolitne arhitekture i kreiranje novog projekta koji poprima oblik mikroservisa [25].

3.4.1. Prednosti i nedostaci

Postoje brojne prednosti mikroservisima, a jedna od najvažnijih je njihova skalabilnost. Budući da je svaki mikroservis neovisan, moguće ih je odvojeno skalirati ovisno o potrebama. Ako određeni servis zahtijeva veću količinu resursa moguće je skalirati taj servis bez utjecanja na ostatak aplikacije [22]. Skalabilnost mikroservisa je jedan od glavnih razloga zašto su ranije navedena poduzeća odlučila migrirati svoje aplikacije na arhitekturu mikroservisa i taj potez im je pružio veliku fleksibilnost u daljnjem radu. Budući da se isporuka svakog servisa provodi zasebno i svaki servis predstavlja autonomnu cjelinu, lakše je ispravljati greške i isporučivati nove verzije. Servis je moguće ažurirati bez ponovnog isporučivanja cijele aplikacije što ukazuje na agilnost arhitekture mikroservisa [22]. Budući da mikroservisi predstavljaju manje komponente aplikacije, repozitoriji koda uglavnom ne sadrže velike količine programskog koda i zbog toga je lakše upravljati tim repozitorijima [22]. Autonomnost repozitorija omogućava korištenje različitih tehnologija za svaki servis [22]. Fleksibilnost u izboru tehnologija je jako korisno u slučaju da je potrebno koristiti određenu tehnologiju koja bolje rješava određeni problem ili bolje odgovara članovima tima koji je zadužen za komponentu koji izrađuju. Pouzdanost sustava je drastično unaprijeđena korištenjem mikroservisa jer prestanak rada jednog mikroservisa ne označava prestanak rada cijele aplikacije nego samo mikroservis s greškom postaje nedostupan [22]. Nakon prestanka rada je moguće pravovremeno obavijestiti korisnike o prestanku rada značajke aplikacije i riješiti problem.

Arhitektura mikroservisa sa sobom nosi i određene izazove. Temeljni izazov s kojim se suočava svaka aplikacija s ovom arhitekturom je složenost sustava. Za razliku od monolitne arhitekture koja ima jednu temeljnu komponentu, mikroservisi imaju veliki broj komponenti koje je potrebno uskladiti na optimalan način kako bi postigli željene rezultate. Unatoč tome što servisi uglavnom nisu veliki dijelovi sustava, cjelokupnom sustavu raste složenost zbog većeg broja komponenti [22]. Pisanje manjih servisa zahtijeva drukčiji pristup nego pisanje monolitnih aplikacija što pruža određene izazove prilikom razvoja. Refaktoriranje često zahtijeva usklađivanje nekoliko servisa radi konzistencije te je pojedine servise potrebno prilagođavati jedne drugima radi očuvanja integriteta podataka [22]. Osim toga, postavljanje lokalne radne

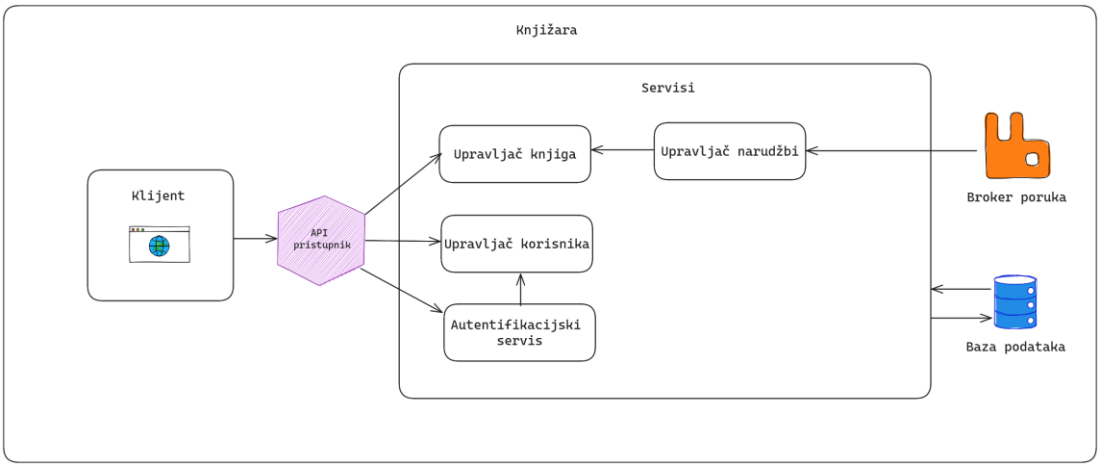
okoline je također otežano i zahtijeva pokretanje svih servisa potrebnih za aplikacije što može usporiti računalo na kojem se pokreće radna okolina. Korištenje manjih servisa može rezultirati komunikacijom između servisa što može stvoriti dugačke lance komunikacije. U ovakvim situacijama može doći do povećane latencija i usporavanje servisa. Zbog toga je potrebno pažljivo dizajnirati API-je na način da se prilagode serijalizirani formati podataka, izbjegavanje prečeste komunikacije između API-ja te korištenje asinkronih komunikacijskih obrazaca [22]. Upravljanje arhitekturom mikroservisa nije samo zahtjevna za programere koji su zaduženi za razvoj ovih servisa, nego zahtijeva iskusan DevOps tim koji će optimalno sinkronizirati mrežu mikroservisa na način da postignu najbolje moguće performanse i ispune sve zahtjeve aplikacije.

Tablica 4 prednosti i nedostaci arhitekture mikroservisa (izvor: vlastita izrada)

Prednosti i nedostaci arhitekture mikroservisa	
Prednosti	**Nedostaci**
visoka skalabilnost	složenost
brzo ažuriranje	zahtijeva visoko tehničko poznavanje arhitekture
agilnost	moguća latencija u komunikaciji
mogućnost korištenja različitih jezika i tehnologija	
pouzdanost	

3.4.2. Primjeri

Korištenje mikroservisa za strukturiranje aplikacije za knjižaru je najfleksibilniji izbor arhitekture. Moguće modelirati aplikaciju na veliki broj različitih načina ovisno o stupnju granulacije mikroservisa.



Slika 15 model arhitekture mikroservisa knjižare (izvor: vlastita izrada)

Slika 15 prikazuje način podjele sustava za upravljanjem knjižarom na 4 različita servisa i odvojeni broker poruka. Osim toga što klijent pristupa servisima putem API pristupnika, pojedini servisi međusobno komuniciraju. Servis za upravljanje korisnika i autentifikacijski servis komuniciraju jer su oboje vezani za domenu korisnika, jednako kao što servis za upravljanjem knjiga i servis za upravljanjem narudžbi međusobno komuniciraju jer se oboje odnose na domenu knjiga. Broker poruka direktno komunicira sa servisom za upravljanje narudžbi kako bi pravilno isporučivao poruke i narudžbama. U ovom primjeru svi mikroservisi dijele zajedničku bazu podataka, ali je moguće implementirati sustav u kojem servisi koriste odvojene baze podataka.

3.5. Razvoj web aplikacije

Ovo poglavlje se odnosi na razvoj dvije web aplikacije koje će biti izrađene koristeći monolitne i klijent-server arhitekture te potom biti uspoređene na temelju performansi, procesa razvoja i procesa isporuke. Aplikacija koja će biti izrađena je Tech Tales – blog koji sadrži članke vezane za tehnologiju. Aplikacija treba sadržavati brojne funkcionalnosti, među kojima su:

- prikaz članaka po kategorijama
- prikaz istaknutog članka
- prijava i registracija
- uređivanje profila korisnika
- kreiranje, uređivanje i brisanje članaka
- komentiranje članaka
- pregled, filtriranje i pretraživanje članaka

Pisanje članaka treba biti napravljeno na način da korisnici imaju veliku slobodu prilikom pisanja članaka, odnosno da mogu mijenjati veličinu teksta, formatirati tekst, prenositi fotografije itd.

Prije razvoja aplikacije će biti opisan proces dizajniranja baze podataka. Bit će opisane tablice koje su kreiranje, njihova svojstva te veze između tablica. Osim PostgreSQL baze podataka koja će biti izrađena, bit će opisana i Redis baza podataka koja će biti korištena za brzo pristupanje podacima.

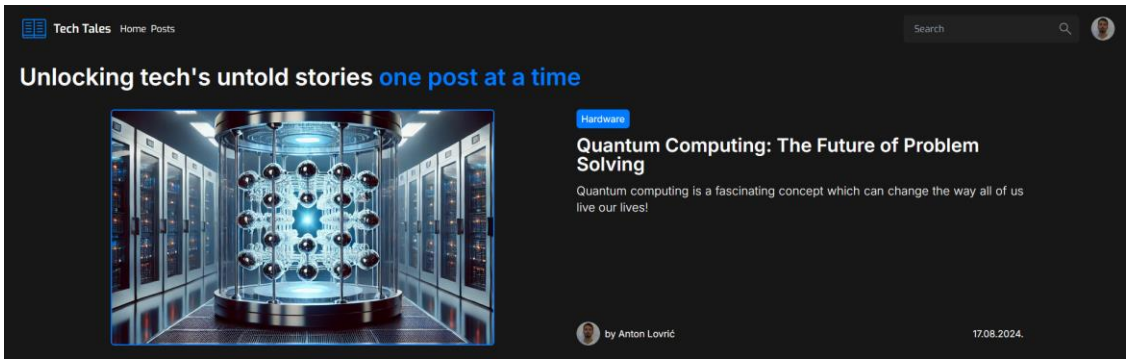
Nakon opisa baza podataka bit će prikazan razvoj aplikacije koristeći monolitne arhitekture. Bit će prikazan model arhitekture aplikacije te programski kod koji se koristio za pojedini dio aplikacije. Osim toga, bit će prikazan proces podjele stranica i pojedinačne komponente te će biti opisane pojedine ključne komponente Next.js okvira za rad. Nakon opisa razvoja aplikacije pomoću monolitne arhitekture, bit će opisan proces razvoja aplikacije koristeći klijent-server arhitekturu. Naglasak će biti postavljen na programiranje na strani poslužitelja budući da će strana klijenta biti jako slična primjeru u monolitnoj arhitekturi.

Nakon razvoja aplikacije potrebno je usporediti ove dvije aplikacije kroz cijeli njihov životni ciklus. Bit će uspoređen proces planiranja, razvoja, isporuke te održavanja aplikacija. Osim toga potrebno je usporediti performanse aplikacije budući da performanse aplikacije igraju ključnu ulogu u uspješnosti proizvoda.

3.5.1. Aplikacija Tech Tales

Kao što je ranije opisano, web aplikacija koja će biti razvijena u sklopu ovog diplomskog rada se zove *Tech Tales*. Ova aplikacija predstavlja online blog na kojem je moguće pronaći razne članke vezano za različita područja, npr. hardver, softver itd. Unutar ovog poglavlja će biti prikazani i opisani svi dijelovi ove aplikacije, uključujući određene aspekte razvoja aplikacije koje su zajedničke i monolitnoj i klijent-server arhitekturi.

Prvi dio aplikacije koji će biti opisan je naslovna stranica. Naslovna stranica sadržava stvari koje bi mogle biti zanimljive svim korisnicima ove aplikacija. Organizirana je na način da se na početku stranice prikazuje istaknuta objava. Istaknuta objava je objava s kojom su korisnici imali najviše interakcija i zbog toga se ističe kao reprezentativan članak u danom trenutku. Istaknuta obrana se određuje periodički, a proces određivanja istaknute objave je detaljnije opisano u poglavlju Redis. Osim istaknute objave, prikazano je posljednjih 6 objava iz svake kategorije. Ovi dijelovi omogućavaju korisnicima brzi pregled najnovijih članaka. Dio naslovne stranice s istaknutom objavom je prikazan sljedećom slikom zaslona.

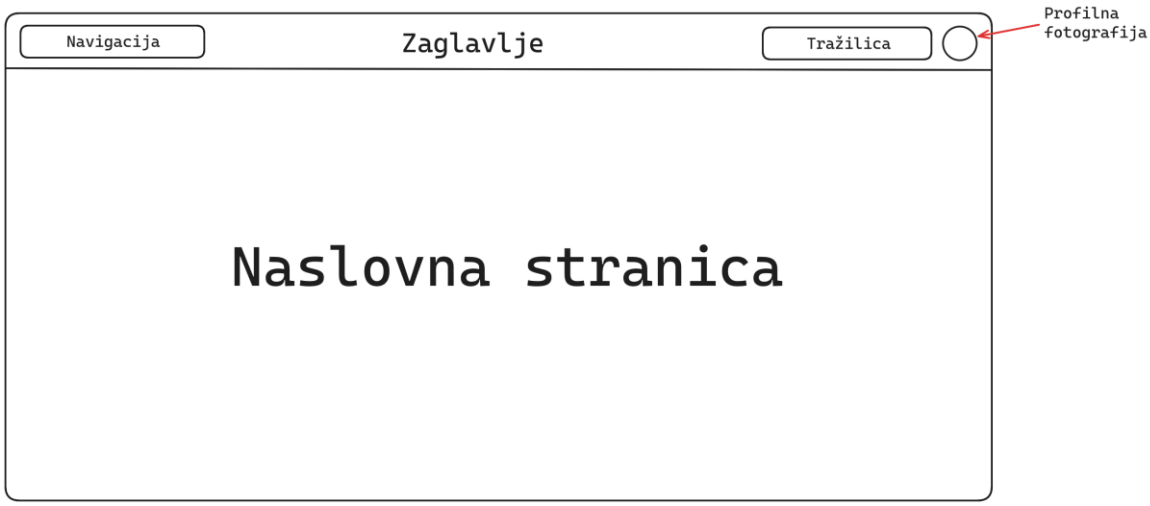


Slika 16 istaknuta objava na naslovnoj stranici (izvor: vlastita izrada)

Budući da Next.js okvir za rad funkcionira na temelju podjele velikih dijelova aplikacije na komponente, proces podjele aplikacije na komponente će biti opisan na ovom primjeru kroz nekoliko slika zaslona.

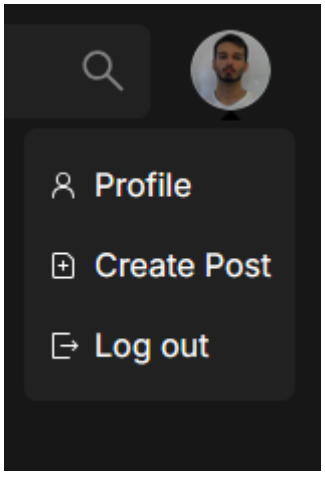
React komponenta je JavaScript funkcija koja vraća određenu HTML strukturu u obliku JSX-a [28]. Prikazivanje vraćenih HTML oznaka se naziva renderiranje sadržaja. JSX je ekstenzija JavaScript sintakse koja omogućava zajedničko korištenje HTML oznaka i JavaScript izraza [29]. Za kreiranje React komponente je dovoljno napisati funkciju koja vraća HTML te ju koristiti kao najobičniju HTML oznaku u ostalim komponentama i stranicama u aplikaciji. Na primjer, ako imamo komponentu *Zaglavlje*, tu komponentu možemo koristiti na svakom mjestu u aplikaciji korištenjem oznake <*Zaglavlje* />. S komponentama je moguće komunicirati tako da im proslijedimo određene argumente, baš kao i običnim JavaScript funkcijama. Primjer prosljeđivanja argumenata React komponentama je *<Zaglavlje argument={vrijednost} />.*

Podjela na komponente se provodi tako da odredimo koje dijelovi aplikacije mogu funkcionirati zasebno, neovisno o drugim dijelovima aplikacije. Prvi dio s prethodne slike zaslona koji je moguće odvojiti u posebnu komponentu je zaglavlje stranice. Koncept podjele ovog dijela aplikacije na manje komponente je prikazan sljedećom shemom.



Slika 17 podjela stranice na komponente (izvor: vlastita izrada)

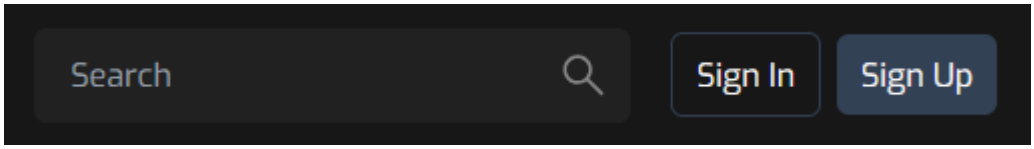
Prema ranije prikazanoj shemi je moguće zaključiti da su zaglavlje i naslovna stranica potpuno odvojene komponente koje zasebno funkcioniraju. Budući da će svi dijelovi aplikacije imati jednako zaglavlje, komponentu zaglavlja je moguće ponovno iskoristiti na svakom dijelu aplikacije. Zaglavlje je moguće još podijeliti na navigaciju, tražilicu i profilnu fotografiju prijavljenog korisnika, budući da svaki od ovih dijelova ima zasebno ponašanje. Daljnja podjela komponenti zaglavlja je pogotovo važna kod tražilice i profilne fotografije prijavljenog korisnika, budući da ti dijelovi imaju dodatne funkcionalnosti. Interakcija s tražilicom šalje zahtjev na poslužitelj i dohvaća objave koje imaju traženi upit u naslovu, dok klik na profilnu fotografiju otvara meni koju pruža dodatne akcije. Odvajanjem ovih dijelova aplikacije u zasebne komponente ima veliku važnost jer razdvajamo logiku što uvelike olakšava čitljivost koda i održavanje projekta.



Slika 18 akcije prijavljenih korisnika (izvor: vlastita izrada)

Prema ranije prikazanoj shemi je moguće zaključiti da su zaglavlje i naslovna stranica potpuno odvojene komponente koje zasebno funkcioniraju. Budući da će svi dijelovi aplikacije imati jednako zaglavlje, komponentu zaglavlja je moguće ponovno iskoristiti na svakom dijelu

aplikacije. Zaglavlje je moguće još podijeliti na navigaciju, tražilicu i profilnu fotografiju prijavljenog korisnika, budući da svaki od ovih dijelova ima zasebno ponašanje. Daljnja podjela komponenti zaglavlja je pogotovo važna kod tražilice i profilne fotografije prijavljenog korisnika, budući da ti dijelovi imaju dodatne funkcionalnosti. Interakcija s tražilicom šalje zahtjev na poslužitelj i dohvaća objave koje imaju traženi upit u naslovu, dok klik na profilnu fotografiju otvara meni koju pruža dodatne akcije. Odvajanjem ovih dijelova aplikacije u zasebne komponente ima veliku važnost jer razdvajamo logiku što uvelike olakšava čitljivost koda i održavanje projekta. Budući da u React komponentama možemo koristiti JavaScript logiku za renderiranje, unutar komponenti je moguće prikazivati drukčiju HTML strukturu ovisno o određenim vrijednostima. Ovakav pristup nam omogućava prikazivanje profilne fotografije samo ako je korisnik prijavljen. Ako korisnik nije prijavljen, potrebno je prikazati gumbe za prijavu i registraciju.



Slika 19 prikaz gumba za prijavu i registraciju (izvor: vlastita izrada)

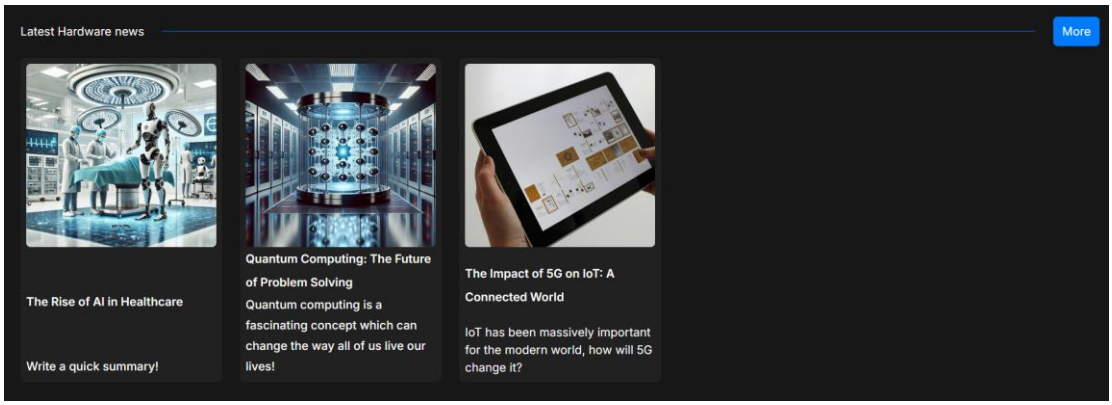
```
{isLoggedIn ? (
  <ProfileIcon logout={logout} user={userProfile} />
) : (
  <div className="flex items-center gap-2">
    <Link
      className="border border-slate-700 px-3 py-2 rounded-md"
      href={'/sign-in'}
    >
      <p>Sign In</p>
    </Link>
    <Link
      className="bg-slate-700 px-3 py-2 rounded-md"
      href={'/sign-up'}
    >
      <p>Sign Up</p>
    </Link>
  </div>
)}
```

Slika 20 uvjetovano renderiranje profilne fotografije (izvor: vlastita izrada)

Slika 20 prikazuje uvjetovano renderiranje profilne fotografije unutar komponente zaglavlja. Korištenjem ternarnog operatora provjerava se vrijednost varijable *isLoggedIn*, koja ukazuje na to ako je trenutni korisnik prijavljen. Ako je vrijednost *isLoggedIn* varijable istinita,

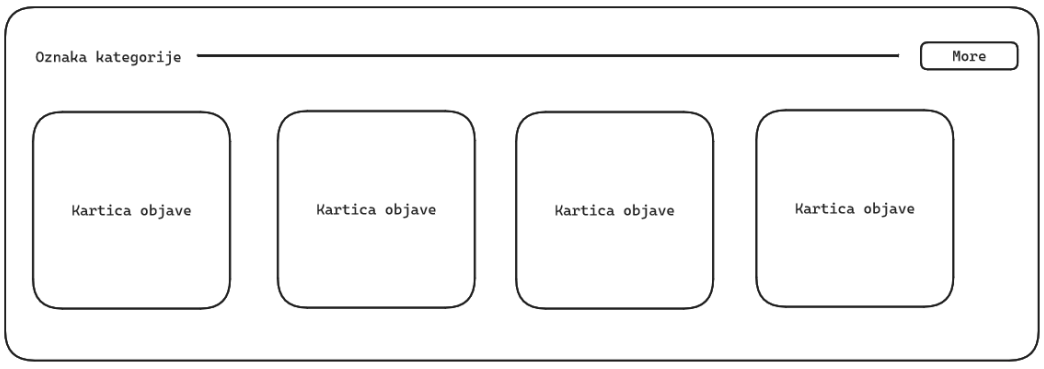
renderirat će se *ProfileIcon* komponenta, u protivnom će se renderirati pretinac koji sadrži poveznice na stranice za prijavu i registraciju.

Drugi dio naslovne stranice sadrži podjelu najnovijih objava po kategorijama. Za svaku kategoriju se prikazuje nove sekcija koja sadrži pet najnovijih objava. Za svaku objavu je prikazana kartica koja sadrži naslovnu fotografiju objave, naslov i sažetak. Klik na svaku karticu vodi na stranicu pregleda objave. Osim toga, iznad svake sekcije je prikazan *More* gumb koji vodi korisnika na stranicu koja sadrži sve objave u odabranoj kategoriji.



Slika 21 prikaz najnovijih objava unutar određene kategorije (izvor: vlastita izrada)

Ovaj dio stranice je potrebno dodatno podijeliti u komponente radi jednostavnijeg korištenja. Prednost podjele kartica objava u zasebno komponentu je povećana zato što su često korištene na ovoj stranicu, što značajno skraćuje količinu koda koja će biti korištena. Osim toga, odvajanje u komponentu omogućava razdvajanje i enkapsulaciju sve logike koja je vezana za te kartice. Budući da se format sekcija također više puta koristi na stranici, njih je također potrebno odvojiti u komponente koje sadrže kartice objava radi jednostavnijeg korištenja.



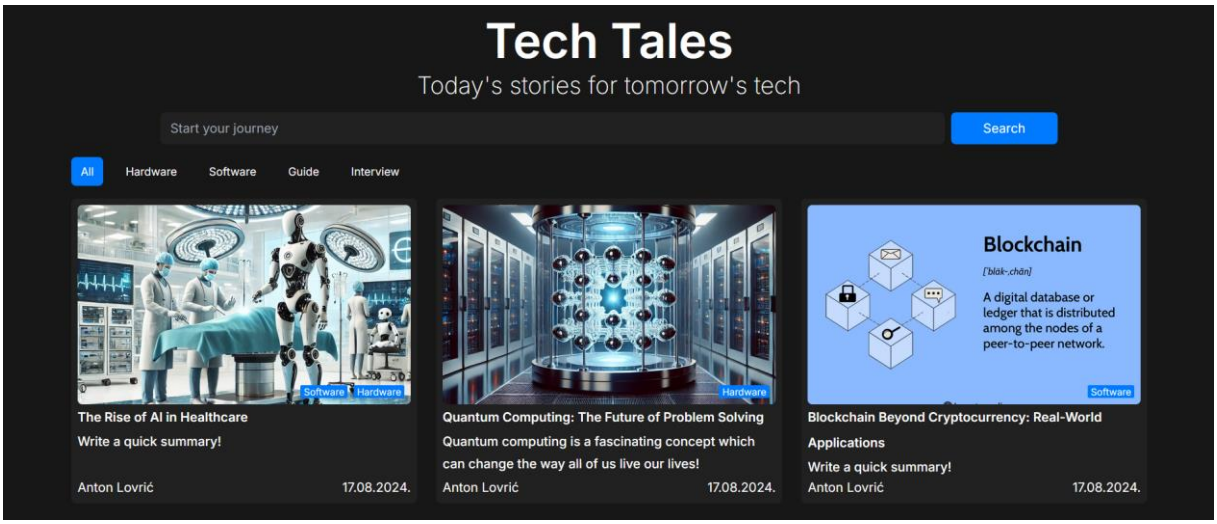
Slika 22 podjela sekcije objava na komponente (izvor: vlastita izrada)

Nakon naslovne stranice, potrebno je izraditi stranicu na kojoj će biti prikazani svi objavljeni članci. Budući da na aplikaciji može postojati veliki broj članaka potrebno je implementirati mehanizam pretraživanja i filtriranja članaka. Pretraživanje je implementirano

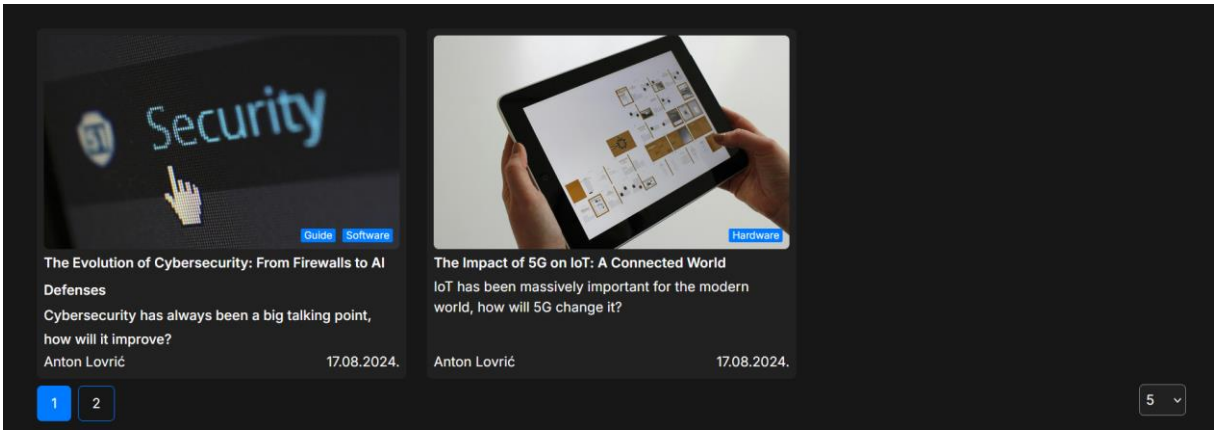
tako da korisnici mogu pretraživati naslove članaka, a mogu filtrirati članke prema različitim kategorijama koje se dodjeljuju člancima prilikom pisanja članaka. Korisnici mogu dodijeliti nula ili više kategorija svom članku, a moguće kategorije su:

- Hardware
- Software
- Guide
- Interview

Pretraživanje i filtriranje je moguće izvesti na strani klijenta (u web pregledniku) i na strani poslužitelja. U ovoj aplikaciji će obje funkcionalnosti biti implementirane na strani poslužitelja radi optimizacije aplikacije. Pretraživanje i filtriranje na strani poslužitelja osigurava bolje performanse zato što je provođenje ovih operacija najbrže kad se provodi izravno nad bazom podataka te nije potrebno slati sve objave na stranu klijenta. Slanje manje količine sadržaja do klijenta značajno ubrzava inicijalno učitavanje ove stranice što je jako bitno radi zadržavanja korisnika na web aplikaciji. Dio teksta koji se upisuje u tražilicu i šifre kategorija prema kojima se vrti filtriranje se šalju na poslužitelj putem parametara u URL-u, nakon čega poslužitelj dohvaća ove vrijednosti, pretražuje i filtrira objave prema ovim kriterijima te ih vraća klijentu. Poslužitelju se osim ovih parametara prosljeđuju broj objava koje je potrebno preskočiti prilikom obrade i broj objava koji je potrebno vratiti. Ovi parametri se koriste za paginaciju podataka, odnosno podjelu podataka na više stranica. Paginacijom omogućavamo korisnicima lakše pregled članaka i bolje performanse aplikacije.



Slika 23 stranica objava (izvor: vlastita izrada)

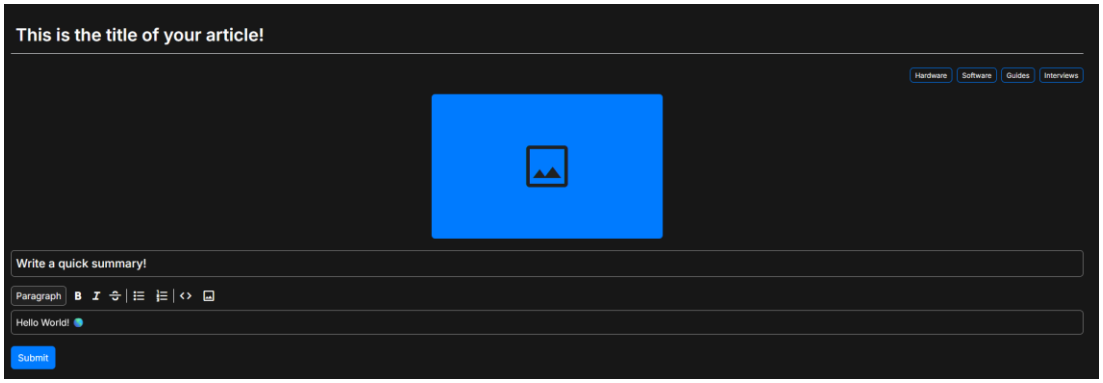


Slika 24 elementi za paginaciju i promjenu broja članaka ne jednoj stranici (izvor: vlastita izrada)

Klikom na svaku karticu objave se otvara nova stranica na kojoj je moguće pročitati sadržaj članka. Ove stranice su namijenjene da budu krajnje jednostavne. Potrebno je samo prikazati sadržaj članka, meta podatke (broj pozitivnih i negativnih glasova, broj komentara, datum objavljivanja itd.) i komentare na članku. Pod sadržaj članka se podrazumijevaju naslov, sažetak, naslovna fotografija članka, tekstualni sadržaj uključujući fotografije u članku te osnovni podaci o autoru članka. Također su prikazane i kategorije članka kako bi korisnici bolje razumjeli na koju domenu se odnosi članak.

Slika 25 stranica članka (izvor: vlastita izrada)

Članke je moguće kreirati klikom na *Create post* poveznicu prikazanu klikom pripadajući gumb u izborniku koji se otvara klikom na profilnu fotografiju prijavljenog korisnika prikazanog slikom 18. Navedena poveznica vodi korisnika na stranicu na kojoj je prikazana forma za kreiranje nove objave. Korisnik treba upisati naslov i sažetak objave te napisati sadržaj članka. Korisnik također može prenijeti naslovnu fotografiju članka koja će biti prikazana u svim karticama na aplikaciji. U prostoru predviđenom za pisanje sadržaja članka, korisnik može uređivati tekst prema svojoj volji. Postoji meni za uređivanje teksta pomoću kojeg korisnik može promijeniti razinu teksta (naslov, podnaslov, paragraf itd.), zadebljati, ukositi ili precrtati tekst, koristit uređenu ili neuređenu listu, umetnuti blok koda ili prenijeti fotografiju koja će biti prikazana zajedno sa tekstom članka.

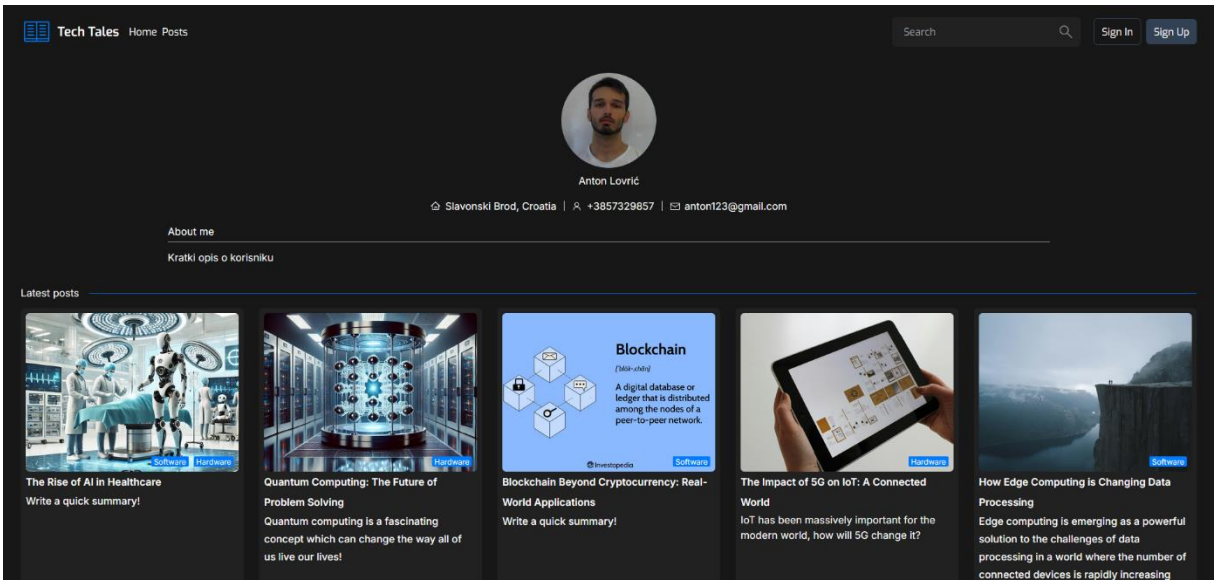


Slika 26 forma za kreiranje članka (izvor: vlastita izrada)

Svaki registrirani korisnik ima svoj profil koji je moguće pregledati. Na profilu korisnika je moguće vidjeti njegove informacije i objave koje je objavio. Informacije koje je moguće vidjeti na profilu su:

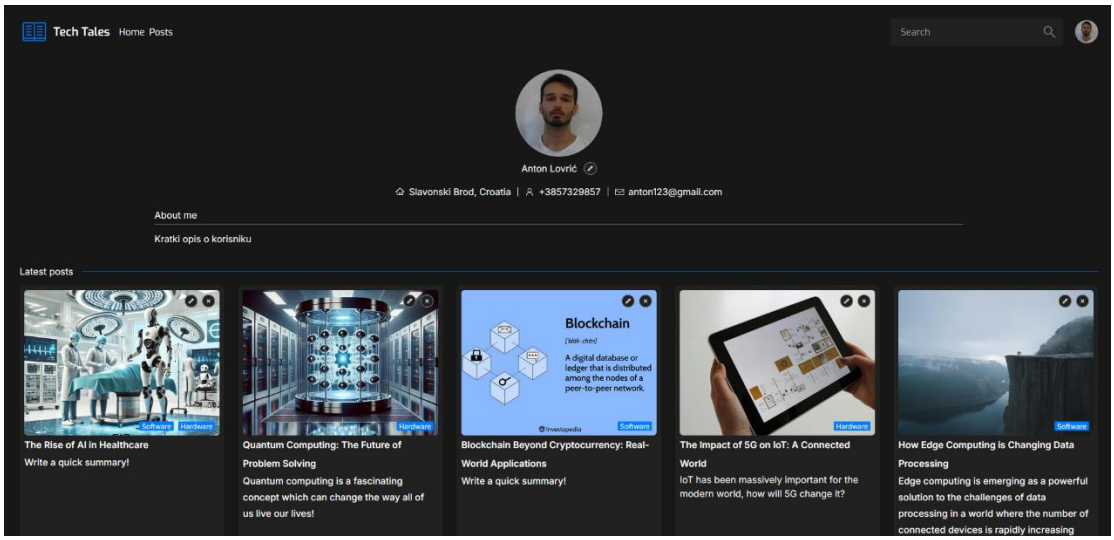
- ime
- prezime
- e-mail adresa
- opis
- lokacija
- broj telefona

Objave su prikazane u obliku kartica, kao i na drugim mjestima u aplikaciji.

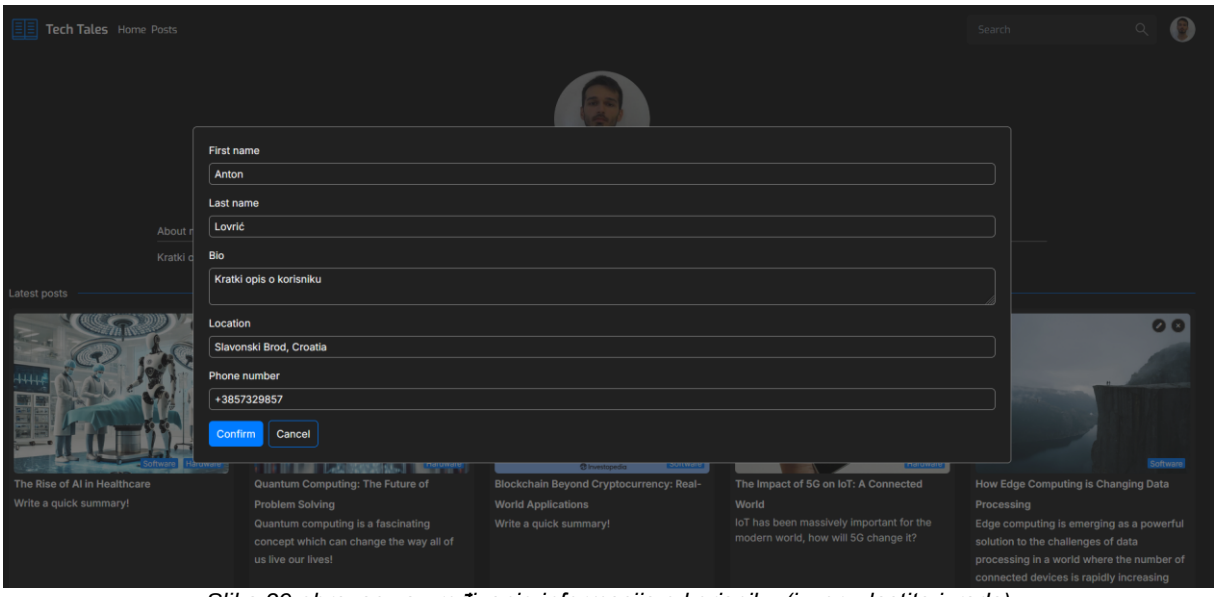


Slika 27 profil korisnika (izvor: vlastita izrada)

Na istoj stranici korisnici mogu i urediti svoj profil. Opcije uređivanja su omogućene samo ako se prijavljeni korisnik nalazi na stranici svog profila. Korisnici mogu klikom na profilnu fotografiju ažurirati svoju profilnu fotografiju prijenosom nove fotografije s uređaja. Moguće je i uređivati informacije o korisniku klikom na gumb za uređivanje pokraj imena korisnika. Osim uređivanja informacija o korisniku, korisnici na ovoj stranici mogu pristupiti uređivanju i brisanju svojih objava koristeći akcijske gumbe koji se nalaze u gornjem desnom uglu kartice objave.

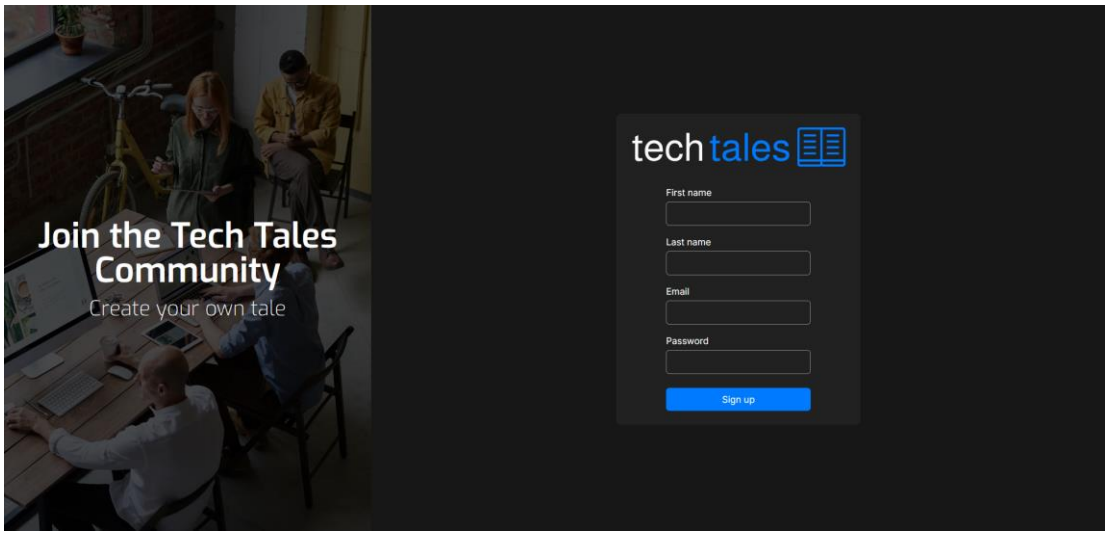


Slika 28 stranica profila prijavljenog korisnika (izvor: vlastita izrada)



Slika 29 obrazac za uređivanje informacija o korisniku (izvor: vlastita izrada)

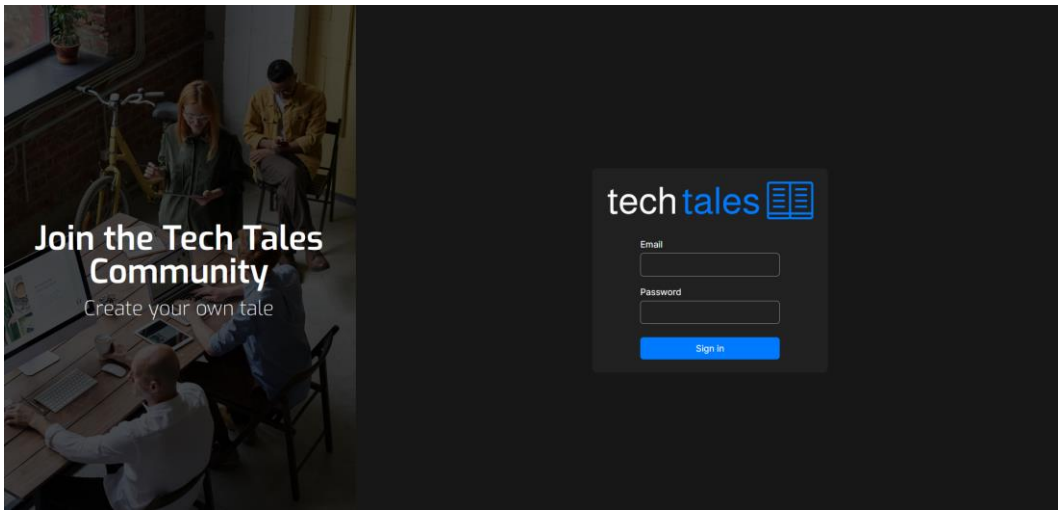
Aplikaciju je moguće koristiti bez kreiranja računa, ali kreiranje objava, komentiranje i glasanje zahtijeva kreiranja korisničkog računa. Korisnički račun je moguće kreirati pomoću odgovarajućeg obrasca koja se nalazi na zasebnoj stranici. Obrazac za registraciju traži unos osnovnih podataka potrebnih za korištenje aplikacije poput imena i prezimena, e-mail adrese i lozinke.



Slika 30 obrazac za registraciju (izvor: vlastita izrada)

Nakon uspješne registracije korisnici se mogu prijaviti i koristiti sve funkcionalnosti aplikacije. Obrazac za prijavu je jako sličan obrascu za registraciju, jedina je razlika u tome što obrazac za prijavu sadrži samo polja za e-mail i lozinku, budući da je e-mail jedinstven za sve korisnike a lozinka služi kao sigurnosna mjera. Implementiran je JWT oblik autentifikacije koji zahtijeva izdavanje dva ključa korisniku. Korisniku se nakon autentifikacije izdaju pristupni ključ (eng. access_token) i ključ za osvježavanje (eng. refresh_token) koji se na siguran način spremaju pomoću kolačića na strani poslužitelja. Pristupni ključ se šalje uz svaki zahtjev kako

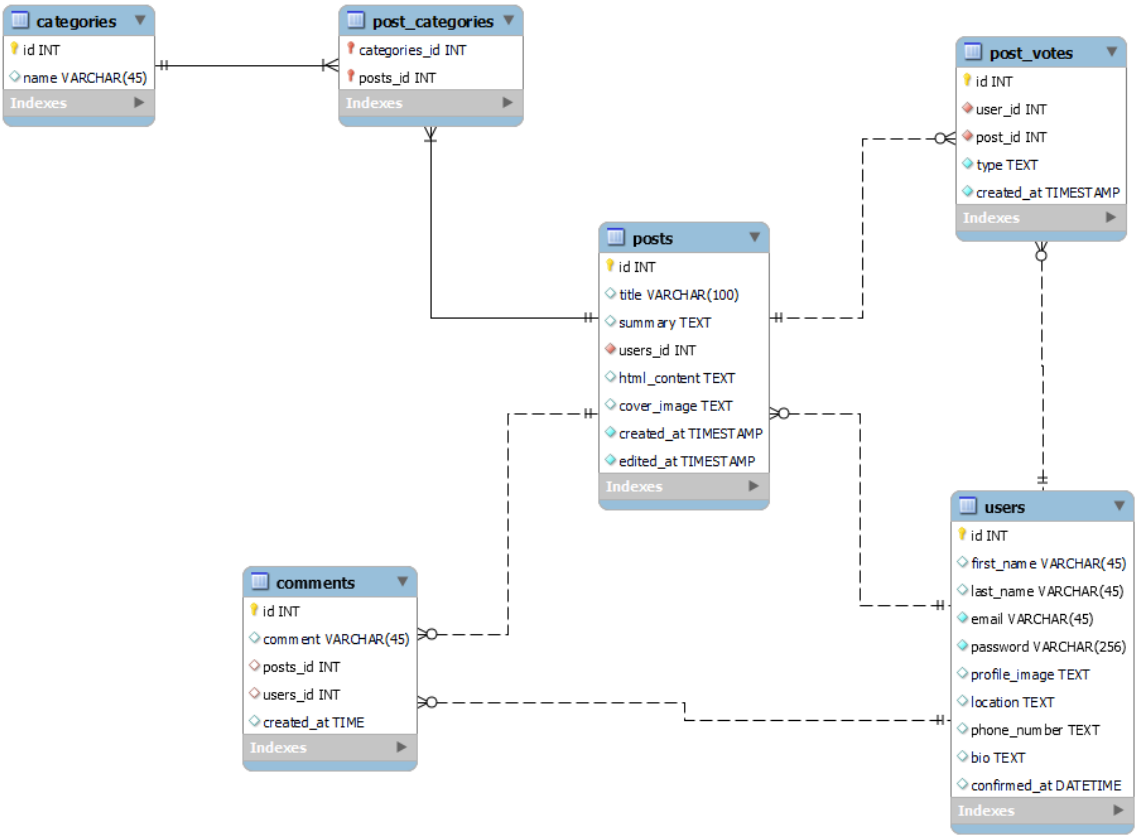
bi se potvrdio identitet korisnika prilikom obrade zahtjeva, ali pristupni ključ traje samo 2 sata. Kratko trajanje pristupnog ključa poboljšava sigurnost aplikacije budući da nakon 2 sata izdavanja više nije važeći ključ i postaje beskoristan napadaču u slučaju presretanja ključa. Nakon 2 sata je potrebno osvježiti ključ posebnim pozivom na stranu poslužitelja koristeći ključ za osvježavanje. Ključ za osvježavanje traje 10 dana te nakon isteka trajanja ključa za osvježavanje se korisnik mora ponovno prijaviti u aplikaciju.



Slika 31 obrazac za prijavu (izvor: vlastita izrada)

3.5.2. Dizajn baze podataka

Baza podataka je iznimno važna stavka svake web aplikacije koja zahtijeva trajnu pohranu podataka. Koristeći bazu podataka moguće je stvoriti trajne zapise koji se dalje koriste u aplikaciji. Ranije je navedeno da je izbor trajne baze podataka relacijski sustav za upravljanje bazom podataka PostgreSQL. Relacijske baze podataka organiziraju podatke uz pomoć tablica i veza između tablica. Tablice sadržavaju stupce i retke, odnosno atribute i zapise. Svaki atribut ima svoj tip podataka kako bi se održao integritet podataka te kako bi se ostvarila razina kontrole i sigurnosti nad podacima koji se zapisuju. Veze između tablica mogu biti veze 1:1 (jedan naprema 1), 1:N (jedan naprema više) i N:M (više naprema više). Pravilno postavljanje odnosa između relacija je neophodno za ispravan rad aplikacije jer je neophodno za dohvaćanje, kreiranje ažuriranje i uklanjanje podataka. Bez pravilno postavljenih veza u bazi podataka nije moguće prikazati podatke na način koji je razumljiv korisnicima. Prije izrade same baze podataka potrebno je izraditi ERA dijagram kako bi uspostavili plan korištenja baze podataka. ERA (entity-relations-attributes) dijagram je dijagram na kojem se nalaze sve tablice (entiteti), veze između tablica i svojstva (atributi) svake tablice. ERA dijagram koji se odnosi na aplikaciju Tech Tales je prikazan sljedećim modelom.



Slika 32 ERA dijagram aplikacije Tech Tales (izvor: vlastita izrada)

ERA dijagram prikazan slikom 16 sadrži 6 entiteta koji tvore bazu podataka:

- objave (posts)
- kategorije (categories)
- kategorije objava (post_categories)
- korisnici (users)
- komentari (comments)
- glasovi objava (post_votes)

Tablice za objave i korisnike su ključne tablice bez kojih aplikacija ne bi imala svrhu, dok ostale tablice doprinose funkcionalnostima aplikacije. Svaka objava, odnosno članak, u aplikaciji ima svoj naslov i sažetak, sadržaj, naslovnu fotografiju te informacije o tome kada je kreirana i uređena. Osim tih atributa, tablica sadrži i atribut *users_id* što predstavlja vanjski ključ na tablicu korisnika. Budući da je veza između ova dva entiteta jedan-naprema-više, vanjski ključ se mora nalaziti u tablici na strani „više“. Odnosno iz ove relacije možemo zaključiti da svaki korisnik može imati 0 ili više objava, dok svaka objava mora biti napravljena od strane jednog, i samo jednog, korisnika. Atributi vezani na tekstualni sadržaj nemaju ograničenja, ali atribut *title* ima ograničenje koje ističe da je maksimalna duljina naslova 100 znakova.

Tablica vezana za korisnike sadrži sve ključne informacije i korisnicima koji su registrirani u aplikaciji te imaju prava za kreiranje, ažuriranje, uklanjanje te komentiranje objava. Osim ključnih polja poput e-mail-a i lozinke koja ne smiju biti prazna, ostala polja se odnose na dodatne informacije o korisnicima koji su prikazani na profilu korisnika. Izrazito je važno da lozinka korisnika bude kriptirana u bazi podataka jer su u protivnom osjetljivi podaci izloženi napadima.

Korisnici mogu ostavljati komentare na objavama radi umrežavanja s drugim korisnicima, stvaranja doprinosa tuđim člancima i razvoja zajednice. Komentari sadrže vanjske ključeve na tablice korisnika i objava tako da objave mogu imati 0 ili više objava te korisnici mogu imati 0 ili više komentara u bazi podataka.

Tablica vezana za glasove objava je modeliranja na jako sličan naziv u smislu da sadrži vanjske ključeve na tablice korisnika i objava. Specifičan aspekt tablice za glasove objava je što stupac koji se odnosi na tip glasa sadrži tekst koji se upravlja izvana. Tipovi korišteni u aplikaciji Tech Tales su „up“ i „down“, odnosno varijanta glasova „sviđa mi se“ i „ne sviđa mi se“. Koristeći ovih glasova korisnici mogu izraziti svoje mišljenje o objavama. Prikazivanje glasova drugih korisnika je dobar oblik označavanja kvalitete članaka. Svaki prijavljeni korisnik može dati svoj glas o objavi i može samo jednom glasati za objavu. Osim toga što korisnik može glasati, korisnik može i poništiti svoj glas ili ga promijeniti.

Tablica *categories* označava kategorije u koje mogu biti svrstane objave. Korištenje kategorija služi za lakšu organizaciju podataka kroz cijelu aplikaciju te za filtriranje objava. Kategorije su implementirane na način da jedna objava može biti svrstana pod više kategorija. Ova funkcionalnost je implementirana korištenjem slabog entiteta, odnosno pomoćne tablice između kategorija i objava. Tablica *post_categories* predstavlja taj slabi entiteta i sadrži samo dva stupca i ti stupci su vanjski ključevi na tablice objava i kategorija. Zapisi u ovoj tablici predstavljaju vezu između objava i kategorije, koja je napravljena u obliku više-naprema-više. Veza nije obavezna prema ovoj tablici jer objave ne moraju biti svrstane pod određenu kategorije i mogu postojati kategorije pod koje nije svrstana nijedna objava.

3.5.2.1. Prisma

Unatoč tome što je moguće izravno upravljati bazom podataka, korištenje eksternih alata za komunikaciju s bazom podataka je jako česta pojava u modernim web aplikacijama. Postoje različiti alati ovakve prirode koji mogu biti korišteni u JavaScript / TypeScript aplikacijama kao što su Prisma, Drizzle, Mongoose, Sequelize itd. Za aplikaciju Tech Tales će biti korišten alat Prisma. Prisma je ORM (object relational mapper) otvorenog koda koji omogućava izvršavanje operacija nad bazom podataka [26]. Osim olakšanog pristupanja bazi podataka i manipuliranja podataka, Prisma automatski primjenjuje zaštitu od SQL injection

napada i tako unapređuje sigurnost sustava. Prisma također osigurava sigurnost tipova podataka, što daje programeru uvid u tip podataka koji su dohvaćeni tijekom manipuliranja podataka. ORM-ovi predstavljaju apstrakciju baze podataka visoke razine koji olakšavaju manipulaciju podataka na način da pruže intuitivno sučelje korisnicima pomoću kojeg je lako izvršavati naredbe nad bazom podataka [27]. Mapiranje podataka se izvodi na način da tablice iz baza podataka budu predstavljene u obliku klasa sa svojstvima koja odgovaraju stupcima pojedine tablice [27]. Svaki projekt koji koristi Prismu mora imati definiranu Prisma shemu u kojoj se nalaze temeljne informacije o bazi podataka i njezinim entitetima. Konfiguracija Prisme je prikazana sljedećom slikom zaslona.

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider  = "postgresql"
  url       = env("POSTGRES_PRISMA_URL")
  directUrl = env("POSTGRES_URL_NON_POOLING")
}
```

Slika 33 konfiguracija Prisma klijenta (izvor: vlastita izrada)

Konfiguracija prikazana prethodnom slikom zaslona prikazuje konfiguracijske ključeve baze podataka. Budući da su ovi podaci tajni, korištene su varijable okoline (eng. environment variables) kako bi bile skrivene od krajnjeg korisnika i radi lakše organizacije konfiguracija ovisno o okolini u kojoj je aplikacija pokrenuta.

```
model users {
  id            Int          @id @default(autoincrement())
  first_name    String?      @db.VarChar(45)
  last_name     String?      @db.VarChar(45)
  email         String       @db.VarChar(45)
  password      String       @db.VarChar(256)
  profile_image String?
  location      String?
  phone_number  String?
  bio           String?
  comments      comments[]
  post_votes    post_votes[]
  posts         posts[]      @relation("posts_users_idTousers")
}
```

Slika 34 Prisma model tablice korisnika (izvor: vlastita izrada)

```
model posts {
  id              Int              @id @default(autoincrement())
  title           String?          @db.VarChar(100)
  summary         String?
  users_id        Int
  html_content    String?
  cover_image     String?
  created_at      DateTime?        @default(now()) @db.Date
  edited_at       DateTime?        @default(now()) @db.Date
  comments        comments[]
  post_categories post_categories[]
  post_votes      post_votes[]
  author          users            @relation("posts_users_idTousers", fields: [users_id], references: [id], onDelete: NoAction, onUpdate: NoAction
}
```

Slika 35 Prisma model tablice objava (izvor: vlastita izrada)

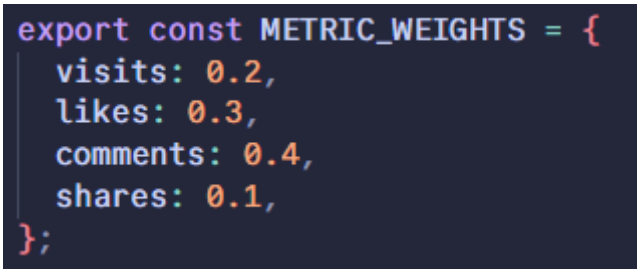
Slike 18 i 19 predstavljaju Prisma modele tablica korisnika i objava. Svaki model sadrži nazive stupaca i tip podataka koji je vezan za svaki stupac. Pojedini tipovi podataka su označeni znakom upitnik koji označava da ovaj stupac može ostati prazan u budućim zapisima. Osim tipova podataka postavljena su ograničenja poput primarnih ključeva, zadanih vrijednosti i vanjskih ključeva u sklopu kojih se nalazi naziv vanjskog ključa, stupci na koje se odnose te akcije koje će biti poduzete u slučaju ažuriranja i uklanjanja povezanih podataka.

3.5.2.2. Redis

Osim korištenja relacijske baze podataka za trajno spremanje podataka vezane za objave, korisnike i slično, u ovoj aplikaciji će biti korišten Redis za spremanje određenih metapodataka vezanih za objave. Budući da je jedan od zahtjeva web aplikacije prikazivanje istkanute objave, potrebno je pratiti nekoliko metrika o svakoj objavi kako bismo uspješno odredili koju objavu je potrebno istaknuti. Statistika koja će biti praćena za svaki članak je:

- broj posjeta
- broj komentara
- broj pozitivnih glasova
- broj podjela članka.

Svakoj ovoj metrici je dodijeljena težina koja označava važnost metrike za određivanje relevantnosti objave. Težine koje su korištene u aplikaciji su prikazane sljedećom slikom zaslona.



```
export const METRIC_WEIGHTS = {
  visits: 0.2,
  likes: 0.3,
  comments: 0.4,
  shares: 0.1,
};
```

Slika 36 težine korištene za izračun relevantnosti objave (izvor: vlastita izrada)

Redis je nerelacijska baza podataka koja sprema podatke u formatu ključ:vrijednost. Oblik ključa je definiran proizvoljno, a u ovoj aplikaciji je znak dvotočke korišten

kao separator. Upravljanje Redis bazom podataka kroz programski kod je prikazano sljedećom slikom zaslona.

```
export function incrementPostVisitCount(postId: number) {
  redisClient?.sAdd('post_ids', postId.toString());
  redisClient?.incr(`post:${postId}:visit`);
}

export function incrementPostLikeCount(postId: number) {
  redisClient?.sAdd('post_ids', postId.toString());
  redisClient?.incr(`post:${postId}:like`);
}

export function incrementPostCommentCount(postId: number) {
  redisClient?.sAdd('post_ids', postId.toString());
  redisClient?.incr(`post:${postId}:comment`);
}

export function incrementPostShareCount(postId: number) {
  redisClient?.sAdd('post_ids', postId.toString());
  redisClient?.incr(`post:${postId}:share`);
}
```

Slika 37 upravljanje Redis bazom podataka (izvor: vlastita izrada)

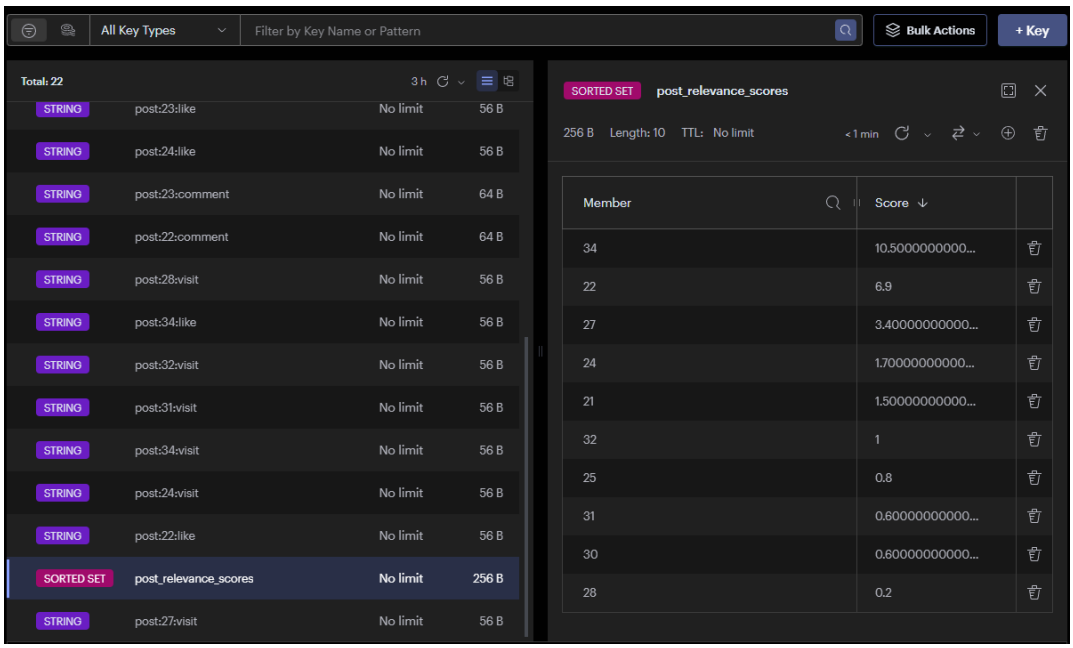
Ranije prikazanim funkcijama su dodijeljeni deskriptivni nazivi radi lakšeg razumijevanja njihovih akcija. Svaka funkcija prima šifru objave kao parametar te inkrementira vrijednost ključa za kojeg je odgovorna. Funkcija *incrementPostVisitCount* inkrementira broj posjeta svake objave, funkcije *incrementPostCommentCount* inkrementira broj komentara na svakoj objavi itd. Ove funkcije se pozivaju prilikom svake odgovarajuće akcije, odnosno nakon zapisa komentara u bazu poziva se funkcija za inkrementiranje broja komentara itd. Nakon prikupljanja podataka o objavama, poziva se funkcija za izračun relevantne objave. Relevantnost objave se računa na način da se vrijednost svake metrike pomnoži sa svojom težinom i zbroji vrijednostima vezanim za ostale metrike.

```
export function calculateRelevanceScore(metrics: IMetrics) {
  const { visits, likes, comments, shares } = metrics;
  return (
    visits * METRIC_WEIGHTS.visits +
    likes * METRIC_WEIGHTS.likes +
    comments * METRIC_WEIGHTS.comments +
    shares * METRIC_WEIGHTS.shares
  );
}
```

Slika 38 funkcija korištena za izračun relevantnosti objave (izvor: vlastita izrada)

Funkcija prikazana slikom 22 se poziva za svaku objavu u bazi podataka i vrijednost njezine relevantnosti se sprema u zasebnu listu u Redis bazi. Nakon spremanja relevantnosti,

po potrebi se dohvaća najrelevantnija objava koja se zatim može prikazati na naslovnoj stranici.



Slika 39 podaci spremljeni u Redis bazu podataka (izvor: vlastita izrada)

Ovakav oblik praćenja metrika je moguće implementirati u postojećoj PostgreSQL bazi podataka, ali budući da Redis sprema sve podatke u radnu memoriju stroja na kojem je pokrenut, sve operacije nad bazom, (dohvaćanje, zapisivanje i brisanje podataka) su iznimno brze te ne opterećuju sustav na kojem je pokrenuta Redis baza podataka. Zato je Redis česti izbor za ovakav oblik praćenja podataka, cacheiranje i slično.

3.5.3. Razvoj aplikacije koristeći monolitnu arhitekturu

Kao što je ranije opisano, monolitna arhitektura zahtijeva cjelovito implementiranje aplikacije i rukovanje svim elementima na jednom mjestu. Potrebno je unutar istog repozitorija koda obuhvatiti sav programski kod koji se izvršava na strani poslužitelja i na strani klijenta. Aplikacija se također isporučuje kao cjelina, odnosno nije potrebno zasebno implementirati više servisa nego jednom isporukom se poslužuje kompletna aplikacija korisnicima.

Uvođenje koncepta serverskih komponenti u sklopu Next.js okvira za rad je znatno poboljšalo proces korištenja monolitne arhitekture u izradi web aplikacija. Serverske komponente su dijelovi programskog koda koji su zaduženi za renderiranje korisničkog sučelja u sklopu kojih se renderiranje izvodi na strani poslužitelja uz opcionalno cacheiranje [7]. Korištenje serverskih komponenti ima brojne prednosti, među kojima je proces dohvaćanje podataka. Budući da se komponente izvršavaju na strani poslužitelja, moguće je izravno

pristupiti bazi podataka. Izravan pristup bazi podataka poboljšava performanse budući da nije potrebno trošiti vrijeme na prijenos podataka. Serverske komponente također poboljšavaju performanse cacheiranjem, smanjivanjem količine JavaSript koda kojeg je potrebno slati web pregledniku te SEO optimizacija pomoću koje se poboljšava dostupnost aplikacije korisnicima [7]. Korištenjem TypeScript-a i alata Prisma za dohvaćanje podataka je moguće na jednostavan način ostvariti potpunu sigurnost tipova kroz cijelu aplikaciju. Nakon dohvaćanja podataka pomoću alata Prisma, vraćena vrijednost automatski poprima strukturu podataka koja je definirana u bazi podataka. Automatsko nasljeđivanje tipova je jako korisna značajka ovog alata zato što olakšava proces razvoja aplikacije i unaprijed upozorava na probleme do kojih je moguće doći prilikom razvoja. Osim zaštite aplikacije od rušenja uslijed nepravilnog korištenja vrijednosti, korištenje ispravnih tipova podataka ubrzava proces razvoja i poboljšava iskustvo programera (eng. developer experience). Većina modernih uređivača teksta automatski predlaže moguće naredbe tijekom pisanja programskog koda, a ti prijedlozi su znatno precizniji ako su korišteni ispravni tipovi podataka.

Slika 40 prijedlozi koda tijekom korištenja alata prisma (izvor: vlastita izrada)

Još jedna stavka koja ubrzava proces razvoja aplikacije je objedinjavanje programskog koda koji se izvršava na strani poslužitelja i na strani klijenta. Tijekom razvoja aplikacije je iznimno jednostavno povezati potrebne podatke sa strane poslužitelja i klijenta budući da su ti elementi usko povezani. Pokretanje radne okoline je iznimno jednostavno budući da je potrebno pokrenuti samo jednu instancu aplikacije i svi potrebni elementi su odmah dostupni.

Tijekom razvoja aplikacije je lako osjetiti glavne nedostatke monolitnih arhitektura koji su ranije opisani, a to su uska povezanost komponenti i skalabilnost. Daljnjim rastom i razvojem aplikacije, razvoj postaje sve teži i sporiji. Usporavanje razvoja se može umanjiti korištenjem najboljih praksi i pravilnom organizacijom koda, ali pokretanje klijenta i poslužitelja u jednoj instanci s vremenom zahtijeva sve više resursa i postepeno se usporava. Aplikacija napravljena u sklopu ovog diplomskog rada nije toliko velika da se značajno osjete nedostaci

skalabilnosti, ali su očiti potencijalni problemi koji mogu nastati ovakvim pristupom. Također nije moguće koristiti drukčije tehnologije, pristup koji bi bio moguć korištenjem razdvojene arhitekture ili arhitekture mikroservisa. Zato je obavezno korištenje Next.js okvira za rad i Node.js servisa te vanjskih paketa koji su dostupni u JavaScript i TypeScript ekosustavu. Next.js je iznimno moćan alat i JavaScript ekosustav je jako bogat što umanjuje negativan utjecaj uske povezanosti. Programski kod monolitne aplikacije je dostupan ovdje.

3.5.4. Razvoj aplikacije koristeći klijent-server arhitekturu

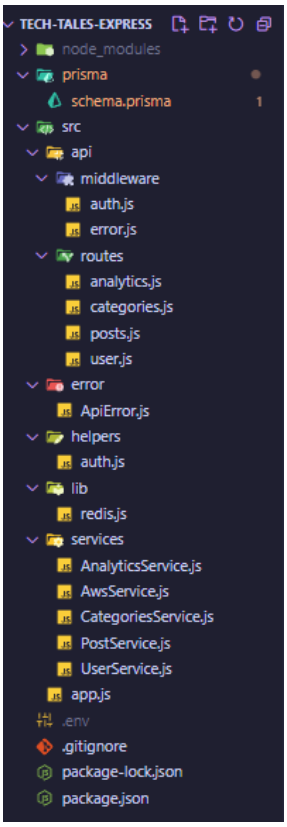
Klijent-server arhitektura podrazumijeva odvajanje servisa koji se pokreće na strani klijenta i servisa koji se pokreće na strani poslužitelja. Za razvoj na strani klijenta je se i dalje koristi Next.js, a na odvojenom poslužitelju je implementiran servis korištenjem Express.js okvira za rad. Express.js servis je napisan u programskom jeziku JavaScript te koristi alat Prisma za rukovanje bazom podataka. Dohvaćanje podataka na stranu klijenta se izvodi korištenjem ugrađenog *fetch* API-ja. Fetch API je sučelje dostupno u svim modernim web preglednicima te služi kao zamjena za XML HTTP zahtjeve [30]. Fetch API predstavlja fleksibilniju alternativu od XML HTTP zahtjeva koji su prije bili često korišteni te funkcioniraju na principu *Promise* objekata. Svaki Fetch poziv zahtijeva minimalno jedan argument, to je URL na koji se šalje poziv i predstavlja obavezan argument. Osim toga je moguće proširiti poziv većim brojem argumenata kao što su vrsta zahtjeva, tip HTTP zahtjeva i tijelo zahtjeva [30]. Next.js je proširio fetch API radi implementiranja cachiranja te tako dodatno ojačali već moćan oblik slanja zahtjeva poslužitelju [31].

Strana klijenta je ostala jako slična kao što je napravljena i u monolitnoj arhitekturi, samo što je bilo potrebno implementirati drukčiji način dohvaćanja podataka budući da aplikacija više nema izravan pristup bazi podataka. Implementirana je posebna *customFetch* funkcija pomoću koje se šalju zahtjevi na stranu poslužitelja uz automatsku dodjelu pristupnih ključeva i ključeva za osvježavanje. Osim toga je implementiran i mehanizam automatskog osvježavanja ključeva. Ovakav način olakšava autentifikaciju korisnika i korištenje autentifikacijskih ključeva. Programski kod klijent-server aplikacije je dostupan ovdje.

```
export const customFetch: typeof fetch = async (url, options) => {
  const accessToken = cookies().get('accessToken')?.value || '';
  const res = await fetch(url, {
    ...options,
    headers: new Headers({
      ...options?.headers,
      Authorization: `Bearer ${accessToken}`,
      Accept: 'application/json',
      'Content-Type': 'application/json',
    }),
  });
  if (res.status === 401) {
    const refreshToken = cookies().get('refreshToken')?.value || '';
    const tokensRes = await fetch(
      `${process.env.NEXT_PUBLIC_API_URL}/users/refresh-token`,
      {
        method: 'POST',
        body: JSON.stringify({
          refreshToken,
        }),
      }
    );
    if (tokensRes.status === 401) {
      logout();
      return res;
    }
    const tokens = await tokensRes.json();
    if (tokens.accessToken) {
      setAccessToken(tokens.accessToken);
    }
    if (tokens.refreshToken) {
      setAccessToken(tokens.refreshToken);
    }
  }
  return res;
};
```

Slika 41 automatsko osvježavanje autentifikacijskih ključeva (izvor: vlastita izrada)

Strana poslužitelja se značajno razlikuje od servisa implementiranog u sklopu monolitne arhitekture. Napisan je u Express.js okviru za rad koji je puno lakši i fleksibilniji okvir od Next.js-a. Express.js se zasniva na svojoj fleksibilnosti i proširivosti što ga čini dobrim izborom za ovaj oblik arhitekture. Strukturiranje Express projekta je proizvoljno ali je jako bitno radi lakšeg održavanja projekta. Struktura Express.js projekta implementiranog u sklopu ovog diplomskog rada je prikazana slikom 42.

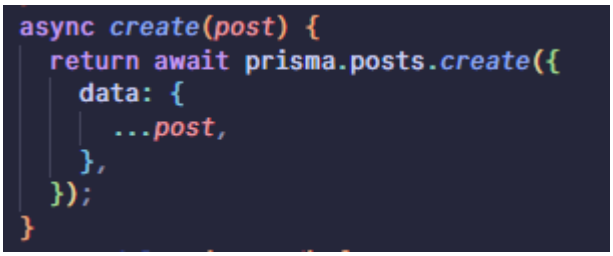


Slika 42 struktura Express.js projekta (izvor: vlastita izrada)

Glavna podjela projekta se vodi kroz mape *services* i *routes* te *app.js* datoteku u kojoj se pokreće cijeli servis. Unutar *app.js* datoteke se registriraju rute u ovom servisu pomoću kojih se razdjeljuju sve pristupne točke. Pristupne točke su definirane u datotekama koje se nalaze u *routes* mapi te svaka datoteka predstavlja jednu domenu aplikacije. Napravljene su zasebne datoteke za rukovanje pristupnim točkama za analitiku, kategorije, objave te korisnike u aplikaciji, a iste datoteke su napravljene i unutar mape *services*. U mapi *services* se nalaze datoteke unutar kojih se izravno rukuje s resursima aplikacije. Primjer povezivanja pristupnih točaka i resursa aplikacije je prikazan slikama 43 i 44.

```
postRouter.post('/', async (req, res, next) => {
  try {
    const post = req.body.post;
    const createdPost = await postService.create(post);
    res.send(createdPost);
  } catch (error) {
    next(error);
  }
});
```

Slika 43 pristupna točka za kreiranje objave (izvor: vlastita izrada)



Slika 44 funkcija za kreiranje objave (izvor: vlastita izrada)

Pristupna točka za kreiranje objava koristi POST metodu koja je namijenjena kreiranju novog resursa te je registrirano na izvorišnu rutu razdjelnika objava. Registriranje pristupne točke na tu rutu znači da će se u lokalnoj okolini ovoj točki pristupati putem veze *http://localhost:3000/api/posts*. U sklopu tijela zahtjeva se šalje objekt koji predstavlja objavu koja treba biti kreirana te se zapis te objave unosi u bazu putem alata Prisma.

Zasebna implementacija servisa na strani poslužitelja omogućava ravnomjerniju podjelu resursa nego što je u monolitnoj arhitekturi. Podjelom repozitorija koda na dva manja repozitorija omogućava znatno lakšu organizaciju programskog koda. Strana klijenta je također olakšana jer je odvojena sva poslovna logika na stranu poslužitelja. Prebacivanjem većine operacija na stranu poslužitelja je implementiran oblik klijent-server arhitekture koji se zove tanki klijent. Poslužitelj i klijent mogu odvojeno skalirati te nisu tako usko povezani kao u monolitnoj arhitekturi. Također je moguće koristiti drukčije tehnologije za razvoj na strani klijenta i na strani poslužitelja. Fleksibilnost koja je ostvarena razdvojenim pristupom pozitivno doprinosi skalabilnosti projekta, performansama te količini resursa koja je potrebna pojedinoj instanci.

Nedostatak klijent-server arhitekture je u tome što zahtijeva zasebno pokretanje servisa na strani klijenta i poslužitelja te konstantnog pridržavanja najboljih praksi kako projekt bi ostao dobro organiziran. U ovom obliku arhitekture nije moguće ostvariti potpunu sigurnost tipova kao što je ostvarena u implementaciji monolitne arhitekture. Manjak sigurnosti tipova podataka zahtijeva veću pažnju prilikom pristupanja resursima koji su razmijenjeni između klijenta i poslužitelja. Ovakve podatke je potrebno dodatno provjeriti kako bi se izbjegli brojni problemi, uključujući različite sigurnosne propuste kao što su SQL umetanje, XSS napadi itd.

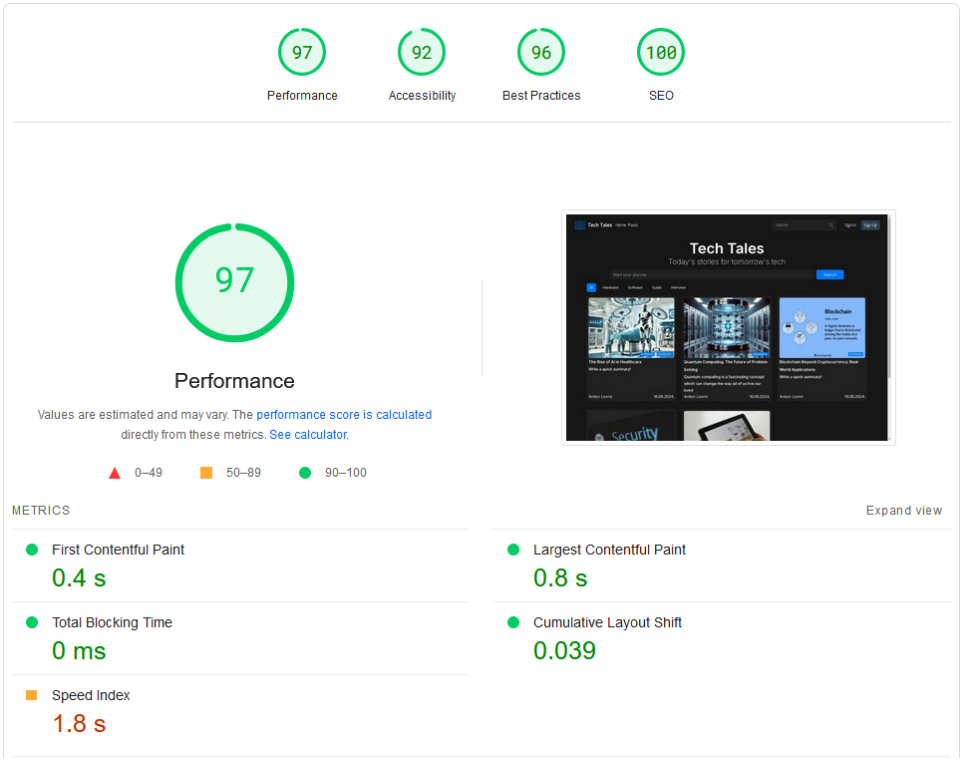
3.5.5. Usporedba razvijenih aplikacija

Nakon opisa aplikacije i arhitekturalnih razlika između dvije aplikacije koje su napravljene u sklopu ovog diplomskog rada, potrebno je usporediti razlike između dvije napravljene aplikacije. Aplikacije će biti uspoređene na temelju više kriterija, među kojima su performanse aplikacije, proces razvoja pojedine aplikacije te proces isporuke i proces održavanja pojedine aplikacije. Usporedba performansa aplikacija se odnosi na uspoređivanje

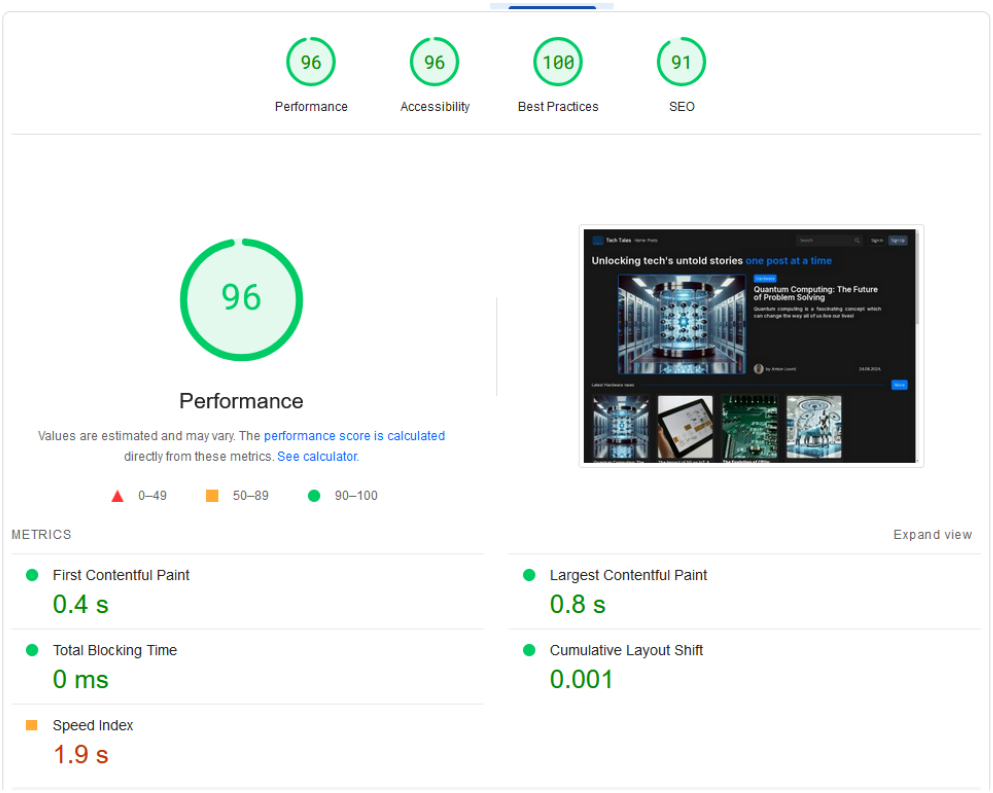
rezultata koji su postignuti testiranjem pomoću alata PageSpeed Insights, a testiranje je provedeno na produkcijskim verzijama aplikacije. Poglavlje koje se odnosi na usporedbu procesa razvoja sadržava opis razvoja pojedine aplikacije s gledišta programera koji razvija aplikaciju. Opisan je proces razvoja te prednosti i mane procesa razvoja aplikacija. Proces isporuke je uspoređen uzimajući u obzir složenost, cijenu isporuke te mogućnost konfiguracije. Proces održavanje je iznimno važan aspekt procesa razvoja softvera te je zbog toga također obuhvaćen ovim poglavljem. Kroz usporedbu procesa održavanja aplikacija bit će uspoređena složenost procesa održavanja pojedine aplikacije te kako parametri poput veličine timova mogu utjecati na proces održavanja.

3.5.5.1. Performanse

Performanse aplikacije su izmjerene za četiri stranice aplikacije koje zauzimaju najviše resursa. Četiri najintenzivnije stranice ove web aplikacije su naslovna stranica, stranica za pregled svih objava, stranica pregleda detalja objave te stranica profile korisnika. Mjerenje je provedeno pomoću aplikacije PageSpeed Insights. PageSpeed Insights mjeri performanse aplikacije na temelju različitih kriterija kao što su brzina odgovora sa strane poslužitelja, prvi preslik aplikacije, ukupno vrijeme renderiranja sadržaja itd. PageSpeed Insights analiza će biti pokrenuta za svaku stranicu na aplikacijama napravljenim u sklopu ovog završnog rada te će se usporediti rezultati između aplikacija s različitim arhitekturama.

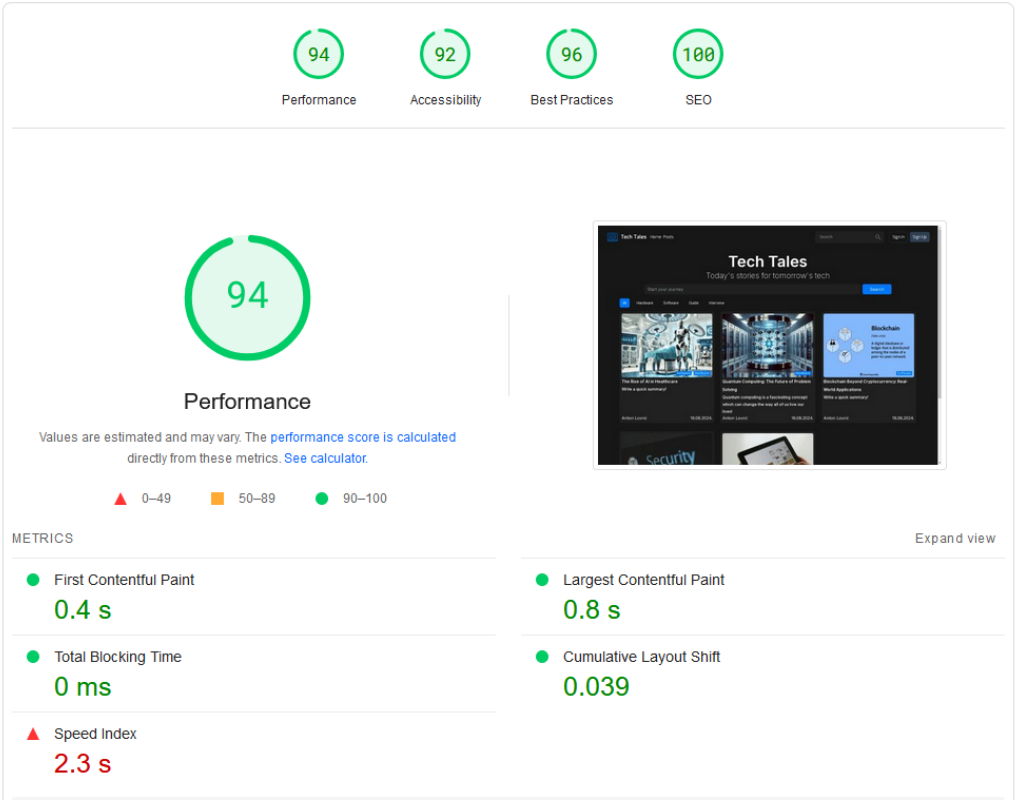


Slika 45 performanse naslovne stranice za monolitnu arhitekturu (izvor: vlastita izrada)

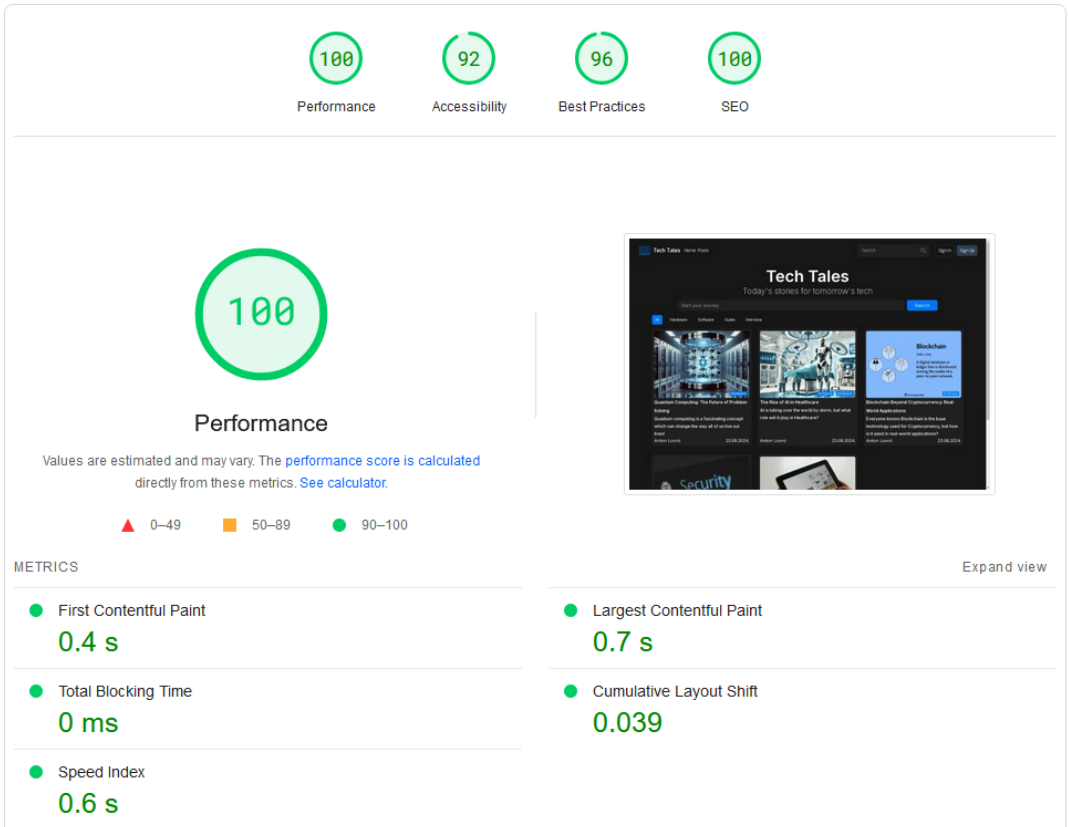


Slika 46 performanse naslovne stranice za klijent-server arhitekturu (izvor: vlastita izrada)

Performanse naslovne stranice pružaju jednake performanse u većini aspekata, a jedina osjetna razlika je u SEO kategoriji gdje prednost ima aplikacija s monolitnom arhitekturom. Također je potrebno uzeti u obzir da aplikacija računa relevantnost objava na svakom pokretanju naslovne stranice dok se izračun relevantnosti u klijent-server arhitekturi periodički pokreće jednom dnevno. Unatoč izgledom većem opterećenju na monolitnoj aplikaciji, performanse su jednake.



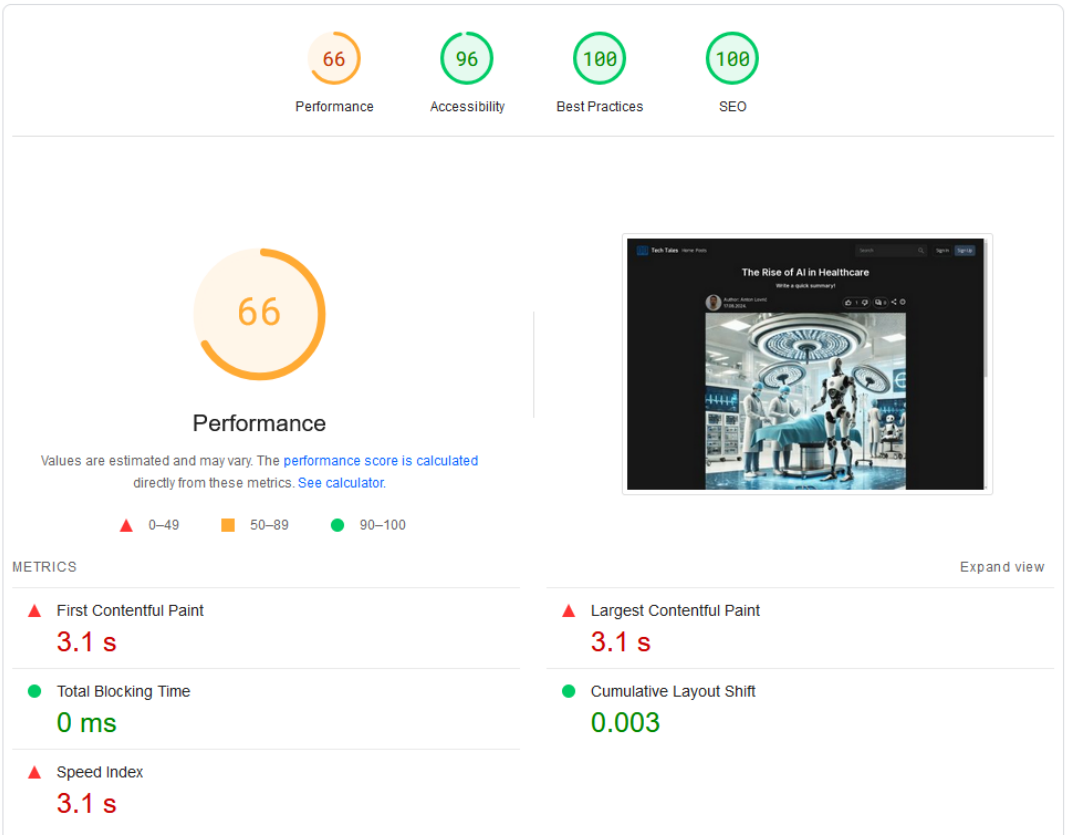
Slika 47 performanse stranice objava na aplikaciji s monolitnom arhitekturom (izvor: vlastita izrada)



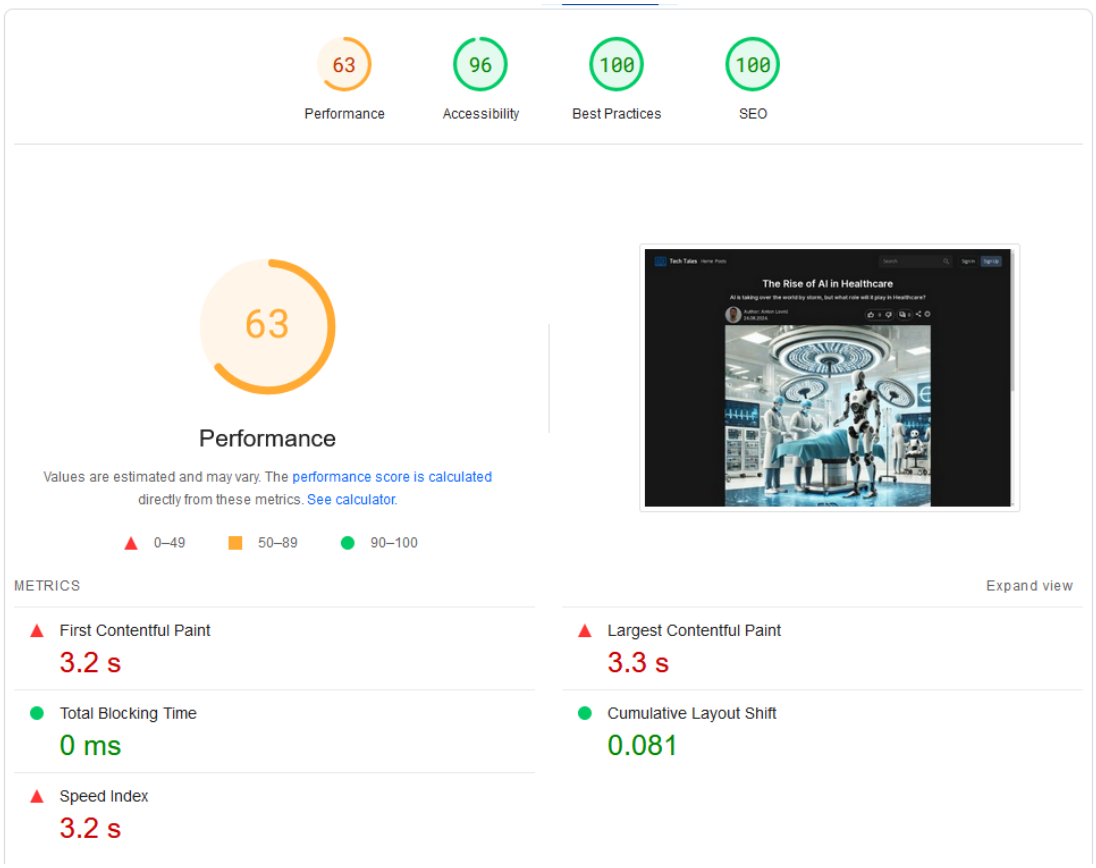
Slika 48 performanse stranice objava na aplikaciji s klijent-server arhitekturom (izvor: vlastita izrada)

Performanse na stranici objava su također poprilično jednaki, ali postoji velika razlika u brzini učitavanja stranice. Aplikacija s monolitnom arhitekturom je bila skoro dvije sekunde

sporija nego verzija aplikacije s klijent-server arhitekturom. Ovo je značajna razlika i igra veliku ulogu u korisničkom iskustvu.

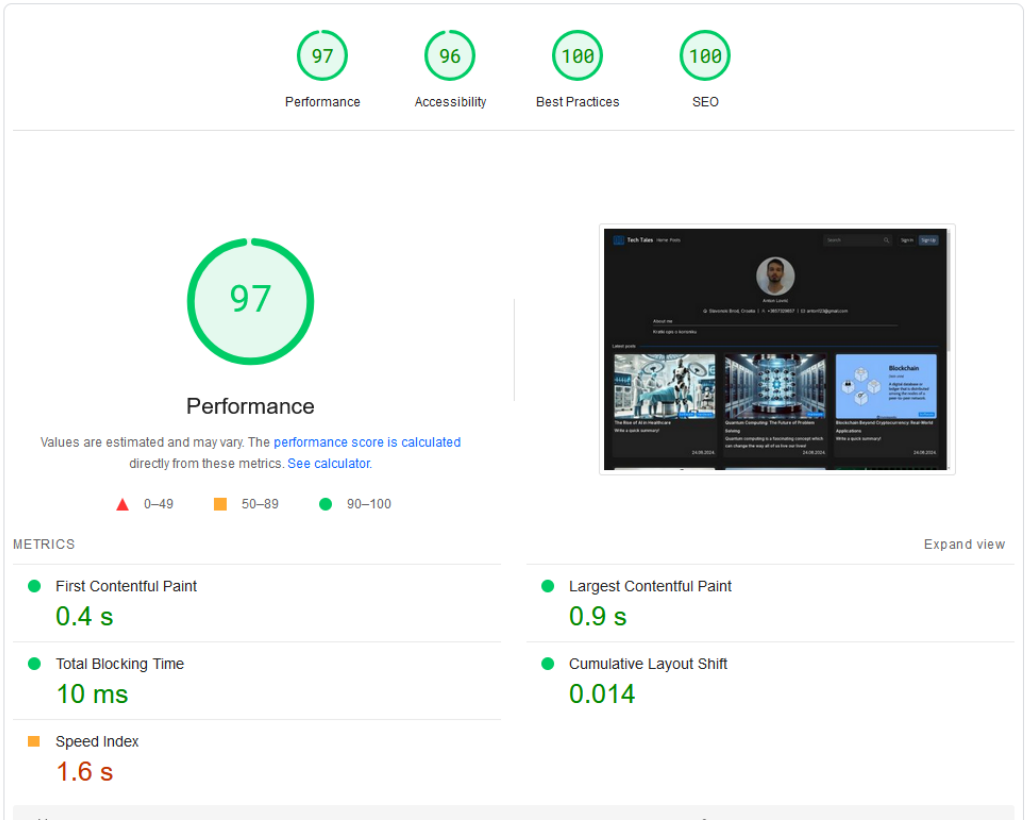


Slika 49 performanse stranice detalja objave na aplikaciji s monolitnom arhitekturom (izvor: vlastita izrada)

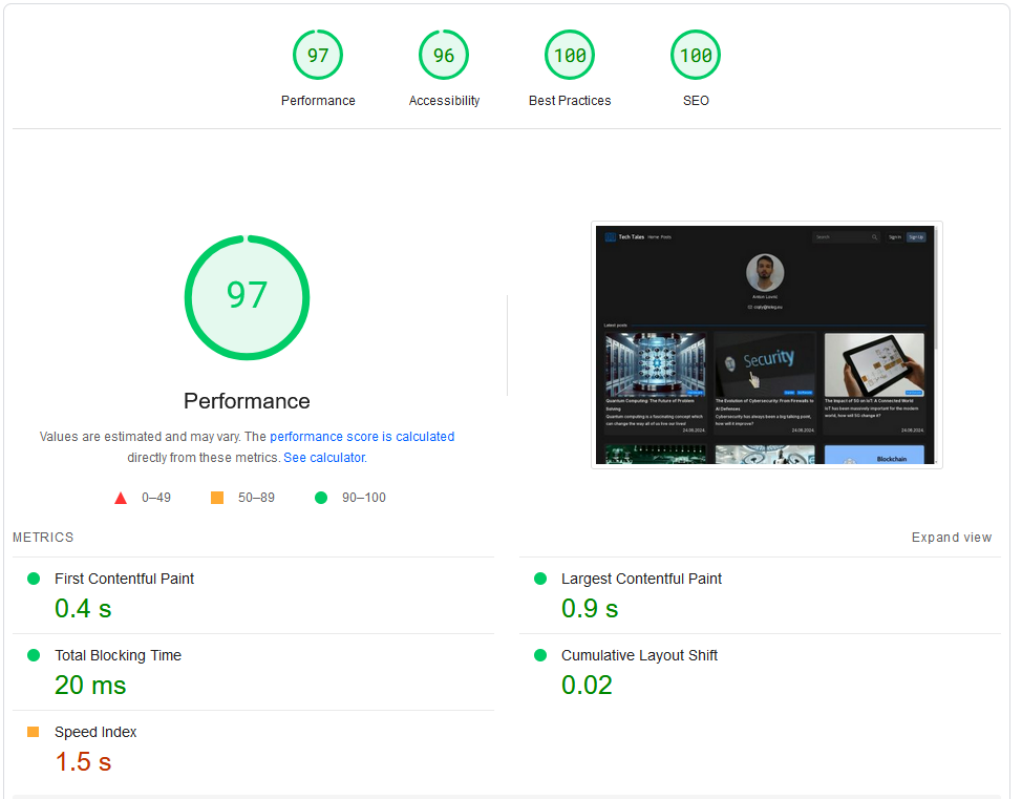


Slika 50 performanse stranice detalja objave na aplikaciji s klijent-server arhitekturom (izvor: vlastita izrada)

Performanse na stranicama detalja objave su jednake na obje aplikacije te su sporije od ostalih stranica zbog potrebe učitavanja cijelog sadržaja svake objave.



Slika 51 performanse stranice profila na aplikaciji s monolitnom arhitekturom) (izvor: vlastita izrada)



Slika 52 performanse stranice profila na aplikaciji s klijent-server arhitekturom (izvor: vlastita izrada)

Stranica profila je jednako brza na obje aplikacije stoga nije moguće dati prednost ijednoj arhitekturi.

Performanse su podjednake na svim analiziranim stranicama, uz jedinu veliku prednost aplikacije s klijent-server arhitekturom na stranici pregleda svih objava. Važno je napomenuti da je ovo jedna od češćih stranica koje će korisnici koristiti te će se razlika u brzini još više osjetiti primjenom filtera i pretraživanjem objava.

3.5.5.2. Proces razvoja

Razvoj aplikacije koristeći monolitnu arhitekturu je jako ugodan i zabavan proces jer su svi dijelovi aplikacije dostupni na jednom mjestu. Ovisno i potrebama radnje koja se trenutno provodi, moguće je pristupiti resursima i na strani klijenta i na strani poslužitelja. Dostupnost svih resursa unutar istog repozitorija olakšava razvoj u smislu da nije potrebno složiti veliki broj komponenti sustava kako bi taj sustav funkcionirao. Svi dijelovi aplikacije u objedinjeni na jednom mjestu te je jako lako pratiti takav način rada. Dostupnost svih resursa na istom mjestu nosi sa sobom i određene izazove, poput činjenice da repozitorij raste rapidnom brzinom budući da se na njemu nalazi logika vezana za stranu klijenta i za stranu poslužitelja. Centralizacija programskog koda također znatno ubrzava razvoj. Razvijajući aplikaciju s monolitnom arhitekturom nije potrebno mijenjati okolinu radi dodavanja novih funkcionalnosti. Svi resursi aplikaciju su dostupni unutar istog repozitorija te je moguće brzo doći do mjesta u kodu koje je potrebno proširiti. Ovakav način rada je pogotovo prikladan manjim timovima ili čak situacijama u kojima jedan programer samostalno razvija aplikaciju. Smanjena mogućnost pojave konflikata prilikom spajanja koda i značajno ubrzanje razvoja čini monolitnu arhitekturu dobrim izborom za manje timove koji trebaju brzo isporučiti svoj proizvod na tržište.

Budući da se u ovakvom repozitoriju nalazi velika količina programskog koda, potrebno je držati se najboljih praksi i pravilno organizirati kod. U slučaju da programski kod ne bude dobro organiziran, postoji velika mogućnost da će stradati performanse aplikacije i da će ju biti znatno teže održavati. Osim pogoršavanja performansi aplikacije, ne pridržavanje najboljim praksama lako može dovesti do situacije da repozitorij brzo postane zatrpan nepotrebnim i nečitkim kodom. Takva situacija značajno otežava daljnji razvoj i održavanje proizvoda. Postoji velika opasnost pojave tehničkog duga u monolitnim aplikacijama koja se samo povećava ubrzanim razvojem. Zato je važno pronaći ravnotežu između brzine razvoja i pisanja održivog koda.

Razvoj aplikacije koristeći klijent-server arhitekturu je također bio iznimno ugodno iskustvo. Inicijalno je bila razvijena aplikacija koristeći monolitnu arhitekturu i potom je izvršena migracija na klijent-server arhitekturu te se odmah osjetilo olakšanje odvajanjem logike poslužitelja na zaseban repozitorij. Kvaliteta aplikacije nije opala zbog kvalitetne potpore koju

Next.js pruža različitim oblicima razvoja aplikacije, ali je rasterećivanje repozitorija pozitivno doprinijelo cjelokupnom iskustvu razvoja aplikacije. Fleksibilnost ovog pristupa je bila očita jer postoji veliki broj mogućih tehnologija koje su se mogle koristiti za razvoj strane poslužitelja. Korištenje drukčijih tehnologija i drukčijih programskih jezika može donijeti brojne prednosti poput boljih performansi i lakše implementacije složenijih značajki aplikacije. Također izbor drukčijih tehnologija može biti korisno u situacijama gdje tim od nekoliko programera razvija aplikaciju zato što je moguće prilagoditi razvojnu okolinu postojećem znanju programera. Podjelom programskog koda na dvije odvojene domene također pomaže u organizaciji projekta. Puno je lakše organizirati projekta kada se rukuje s dvije manje komponente nego s jednom ogromnom komponentom koja tvori cijeli sustav. Također je lakše uskladiti rad cijelog tima programera zato što je podjelom na dva odvojena tima moguće glađe provoditi proces implementacije softvera. Klijent-server arhitektura je odlična alternativa monolitnoj arhitekturi zato što pridodaje fleksibilnosti proizvoda i otvara vrata različitim mogućnostima u budućnosti.

Klijent-server arhitektura sa sobom donosi i određene poteškoće. Određenim programerima može biti neugodno raditi na dva odvojena repozitorija. Radom na odvojenim repozitorijima se također blago usporava proces razvoja u manjim timovima ili u slučaju samostalnog razvoja softvera u usporedbi na monolitnu arhitekturu. Budući da monolitna arhitektura pruža izravan pristup svim resursima, razdvojeni pristup blago komplicira taj proces i tako usporava razvoj. Također je potrebno uvesti dodatan sloj apstrakcije koje se odnosi na dohvaćanje podataka s vanjskog servisa, ali to također omogućava drukčiji oblik pristupanja problemu koji čak može donijeti svoje prednosti.

3.5.5.3. Proces isporuke i održavanje

Postoje brojne platforme na kojima je moguće isporučiti web aplikacije. Jedna od tih platformi je Vercel. Vercel je također tvrtka koja je izradila Next.js okvir za rad i pruža jako dobru potporu svim Next.js aplikacijama za isporuku koristeći njihovu uslugu. Isporuka Next.js aplikacija ne zahtijeva nikakvu dodatnu konfiguraciju te pruža različite prednosti po pitanju skalabilnosti, dostupnosti i performansi [32]. Automatskom optimizacijom procesa renderiranja sadržaja na strani poslužitelja, korištenjem rubne mreže (eng. Edge Network) te optimizacijom fotografija i fontova je moguće ostvariti odlične performanse. Osim brojnih optimizacija, isporuka na Vercel je iznimno jednostavan proces koji omogućava brzu isporuku aplikacije korisnicima. Zbog ovih razloga je cijela monolitna aplikacija izrađena u ovom diplomskom radu isporučena na Vercel platformi te dio klijent-server aplikacije koji se odnosi na izvođenje na strani klijenta. Strana poslužitelja aplikacije izrađene pomoću klijent-server arhitekture je isporučena uz pomoć platforme Railway, koja pruža jednako jednostavan proces isporuke aplikacije. Railway pruža veću fleksibilnost u vrsti aplikacije koja se isporučuje te je zato odabran kao alat za isporuku Express.js servisa izrađenog u sklopu ovog diplomskog rada.

Važno je napomenuti da su ove aplikacije mogle biti isporučene na bilo kojem servisu i bilo kojem poslužitelju koji pruža slične usluge, ali su navedeni alati pružili najefikasniju opciju koja se može koristiti za ovu svrhu.

Procesi isporuke aplikacije s monolitnom arhitekturom i aplikacije s klijent-server arhitekturom su bili jako slični. Budući da je cijela monolitna aplikacija i dio klijent-server aplikacije isporučen na platformi Vercel, iskustvo je u potpunosti jednako po tom pitanju. Bitna razlika je u tome što je proces isporuke u klijent-server arhitekturi nešto duži i zahtijeva dužu konfiguraciju budući da se radi o dva različita servisa. Također je potrebno pratiti dvije različite isporuke i obraćati pažnju na njihov status. Osim toga, potrebno je i uskladiti dva isporučena servisa i tu se osjeti najveća razlika u procesu isporuke i održavanja. Održavanje aplikacije s klijent-server arhitekturom zahtijeva usklađenost dva različita servisa u svakom koraku procesa razvoja softvera, dok je na monolitnoj aplikaciji potrebno pratiti samo jedan repozitorij koji predstavlja cijelu aplikaciju. Aplikacija izrađena pomoću monolitne arhitekture je dostupna ovdje, dok je aplikacija izrađena pomoću klijent-server arhitekture dostupna ovdje.

4. Zaključak

Izbor arhitekture web aplikacije je iznimno važna odluka koju je potrebno pažljivo promišljati. Ne postoji jedinstveno rješenje za sve aplikacije te je odluku potrebno donijeti temeljem više različitih faktora. Neki od tih faktora su veličina aplikacije, potrebna brzina razvoja, karakteristike tima koji razvija aplikaciju (broj članova, vještine programera itd.), dostupni resursi i sl. Korisnici najviše osjećaju utjecaj ove odluke kod velikih web aplikacija koje prioritiziraju performanse, dok programeri osjete utjecaj ove odluke kroz proces održavanja aplikacija. Također je važno napomenuti da točan izbor web aplikacije utječe na mogućnost programera na isporuku novih značajki aplikacije, što posljedično utječe i na iskustvo korisnika.

U sklopu ovog diplomskog rada je najviše pažnje pridodano monolitnoj i klijent-server arhitekturi, uz teorijsku obradu arhitekture mikroservisa. Teorijska obrada arhitekture mikroservisa je uključena u ovaj diplomski rad zbog svoje važnosti u svijetu modernih web aplikacija. Osim teorijske obrade, monolitna i klijent-server arhitektura su dodatne obrađene u praktičnom obliku izradom web aplikacija koristeći ove dvije arhitekture. Izrada web aplikacija je bila najdugotrajniji proces tijekom izrade ovog diplomskog rada i donosi veliku vrijednost diplomskom radu zbog praktične primjene teorijskih koncepata koji su ranije obrađeni. Ove dvije arhitekture imaju svoje prednosti i nedostatke, ali mjerenjem performansi nije bilo moguće zaključiti koji pristup je bolji. Predviđam da bi se veća razlika mogla primijeniti u većim i složenijim aplikacijama, ali izbor arhitekture u ovom slučaju nije imao veliku razliku u pogledu performansi. Također je važno napomenuti da izrađena aplikacija sadrži većinu elemenata koji su korišteni u modernim web aplikacijama te se zato može promatrati kao reprezentativan uzorak. Unatoč sličnosti u performansama, moguće je donijeti osobni zaključak na temelju iskustva razvoja budući da je taj parametar u potpunosti subjektivan. Tijekom razvoja ovih aplikacija osjetio sam snažne prednosti izravnog pristupanja bazi podataka bez korištenja API-ja u monolitnoj arhitekturi te je ovakav pristup značajno pridonio brzini razvoja. Također sam uživao u razdvojenom pristupu klijent-server arhitekture te nije lako donijeti konačnu odluku. Istaknuo bih kako smatram da je izrada web aplikacija koristeći monolitnu arhitekturu uglavnom ispravan izbor, ali bih istaknuo da je jako važno uzeti u obzir potencijalne migracije u budućnosti zbog ograničenja monolitnih aplikacija u pogledu skalabilnosti.

Uživao sam u izradi ovog diplomskog rada jer sam dobio priliku detaljno analizirati arhitekture web aplikacija i upoznati različite pristupe ovom problemu. Smatram da sam značajno napredovao tijekom izrade ovog diplomskog rada u teorijskom poznavanju arhitektura web aplikacija ali i u tehničkim znanjima potrebnim za implementaciju aplikacija.

Popis literature

[1] “What is SDLC? - Software Development Lifecycle Explained - AWS,” Amazon Web Services, Inc. Dostupno: https://aws.amazon.com/what-is/sdlc/. [Pristupljeno: 25.08.2024.]

[2] V. Petrenko, “Web Application Architecture: Advanced Guide & Trends for 2024,” Litslink, Apr. 22, 2021. Dostupno: https://litslink.com/blog/web-application-architecture. [Pristupljeno: 25.08.2024.]

[3] “What is JavaScript? - JavaScript (JS) Explained - AWS,” Amazon Web Services, Inc. Dostupno: https://aws.amazon.com/what-is/javascript/. [Pristupljeno: 25.08.2024.]

[4] “An Introduction to JavaScript.” Dostupno: https://javascript.info/intro. [Pristupljeno: 25.08.2024.]

[5] “JavaScript Programming - The State of Developer Ecosystem in 2023 Infographic,” JetBrains: Developer Tools for Professionals and Teams. Dostupno: https://www.jetbrains.com/lp/devecosystem-2023. [Pristupljeno: 25.08.2024.]

[6] “Docs | Next.js.” Dostupno: https://nextjs.org/docs. [Pristupljeno: 25.08.2024.]

[7] “Rendering: Server Components | Next.js.” Dostupno: https://nextjs.org/docs/app/building-your-application/rendering/server-components. [Pristupljeno: 25.08.2024.]

[8] “What is Express.js?,” Codecademy. Dostupno: https://www.codecademy.com/article/what-is-express-js. [Pristupljeno: 25.08.2024.]

[9] “Express - Node.js web application framework.” Dostupno: https://expressjs.com/. [Pristupljeno: 25.08.2024.]

[10] “Relational vs Nonrelational Databases - Difference Between Types of Databases - AWS,” Amazon Web Services, Inc. Dostupno: https://aws.amazon.com/compare/the-difference-between-relational-and-non-relational-databases/. [Pristupljeno: 25.08.2024.]

[11] P. Scott, “What is PostgreSQL? Everything You Need to Know,” Percona Database Performance Blog, Feb. 02, 2024. Dostupno: https://www.percona.com/blog/what-is-postgresql-used-for/. [Pristupljeno: 25.08.2024.]

[12] “Redis: What It Is, What It Does, and Why You Should Care,” Backendless, Dec. 09, 2022. Dostupno: https://backendless.com/redis-what-it-is-what-it-does-and-why-you-should-care/. [Pristupljeno: 25.08.2024.]

[13] “Software Architecture Guide,” martinfowler.com, Aug. 01, 2019. Dostupno: https://martinfowler.com/architecture/. [Pristupljeno: 25.08.2024.]

[14] L. NORTON, Learning Software Architecture. IT Campus Academy, 2023.

[15] M. Richards, Software Architecture Patterns. O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2015. Dostupno: https://isip.piconepress.com/courses/temple/ece_1111/resources/articles/20211201_software_architecture_patterns.pdf

[16] M. Richards and N. Ford, Fundamentals of Software Architecture: An Engineering Approach. O'Reilly Media, Inc., 2020.

[17] "Software Evolution: Monoliths to Microservices - DZone," dzone.com. Dostupno: https://dzone.com/articles/evolution-of-software-architecture-from-monoliths. [Pristupljeno: 25.08.2024.]

[18] Atlassian, "Microservices vs. monolithic architecture," Atlassian. Dostupno: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith. [Pristupljeno: 25.08.2024.]

[19] "Introduction to web APIs - Learn web development | MDN," Aug. 05, 2024. Dostupno: https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Introduction. [Pristupljeno: 25.08.2024.]

[20] Dostupno: https://www.designgurus.io/answers/detail/what-is-thick-client-vs-thin-client. [Pristupljeno: 25.08.2024.]

[21] K. D. Foote, "A Brief History of Microservices," DATAVERSITY, Apr. 22, 2021. Dostupno: https://www.dataversity.net/a-brief-history-of-microservices/. [Pristupljeno: 25.08.2024.]

[22] martinekuan, "Microservice architecture style - Azure Architecture Center." Dostupno: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices. [Pristupljeno: 25.08.2024.]

[23] "Fielding Dissertation: CHAPTER 5: Representational State Transfer (REST)." Dostupno: https://ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm. [Pristupljeno: 25.08.2024.]

[24] M. Kubieniec, "Four Companies That Migrated From Monolith to Microservices," Kambu, Apr. 21, 2021. Dostupno: https://www.kambu.pl/blog/companies-that-migrated-from-monolith-to-microservices/. [Pristupljeno: 25.08.2024.]

[25] "bliki: Monolith First," martinfowler.com. Dostupno: https://martinfowler.com/bliki/MonolithFirst.html. [Pristupljeno: 25.08.2024.]

[26] "What is Prisma ORM? (Overview) | Prisma Documentation." Dostupno: https://www.prisma.io/docs/orm/overview/introduction/what-is-prisma. [Pristupljeno: 25.08.2024.]

[27] "Is Prisma ORM an ORM? | What is an ORM? | Prisma Documentation." Dostupno: https://www.prisma.io/docs/orm/overview/prisma-in-your-stack/is-prisma-an-orm. [Pristupljeno: 25.08.2024.]

[28] "Your First Component – React." Dostupno: https://react.dev/learn/your-first-component. [Pristupljeno: 25.08.2024.]

[29] "Writing Markup with JSX – React." Dostupno: https://react.dev/learn/writing-markup-with-jsx. [Pristupljeno: 25.08.2024.]

[30] “Fetch API - Web APIs | MDN,” Jul. 29, 2024. Dostupno: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API. [Pristupljeno: 25.08.2024.]

[31] “Functions: fetch | Next.js.” Dostupno: https://nextjs.org/docs/app/api-reference/functions/fetch. [Pristupljeno: 25.08.2024.]

[32] “Next.js on Vercel.” Dostupno: https://vercel.com/docs/frameworks/nextjs. [Pristupljeno: 25.08.2024.]

Popis slika

Popis tablica