SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2653

Sustav za provođenje analize novinskih članaka i pridruženih komentara na društvenim mrežama

Iva Lovrec

Zagreb, lipanj 2021.

Umjesto ove stranice umetnite izvornik Vašeg rada.

Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

SADRŽAJ

1. Uvod

Internetski portali s vijestima i društvene mreže najbitniji su izvor informacija i najnovijih događanja po istraživanju Digital News Reporta te čak 91% ispitanika do vijesti dolazi online [16]. Portali svojim korisnicima omogućuju pregled vijesti iz različitih područja, a većina takvih portala uključuje i dio s komentarima gdje korisnici mogu izraziti svoja mišljenja o temama iz članka. Prikupljanje tih komentara može dati zanimljiv uvid u načine razmišljanja korisnika, njihovu podjelu političkih stavova, emocije koje korisnici izražavaju u komentarima te razne druge zanimljive statistike. Samo prikupljanje komentara daje veliku količinu podataka, no kako bi se ti podaci pretvorili u korisne informacije potrebno ih je analizirati i strukturirati u oblik koji prenosi određeno znanje o komentarima ili njihovim autorima. Ručna obrada tolike količine podataka nije moguća zbog čega je potrebno osmisliti sustav čiji je cilj obrada i analiza podataka prikupljenih s internetskih portala.

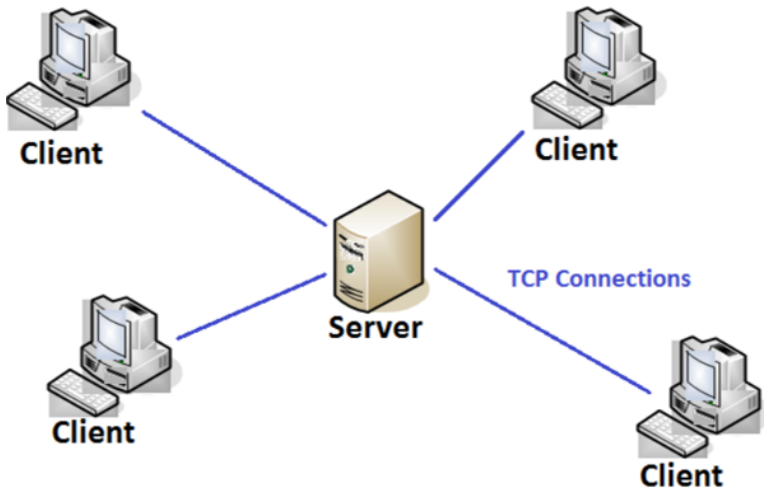
U ovom radu bit će objašnjen razvoj aplikacije koja se bavi upravo tom pretvorbom velike količine podataka o člancima u korisne statistike. Rad obrađuje četiri analize podataka: portali s najviše članaka, portali s najvećim prosjekom komentara po članku, korisnici s najviše komentara te korisnici s najagresivnijim komentarima. Osim analiza, razvijena aplikacija uključuje i pregled podataka o člancima i komentarima te njihovo filtriranje i sortiranje. Aplikacija je dio većeg sustava koji uključuje *scraper*,sustav za skupljanje i obradu internetskih stranica, te se temelji na podacima dobivenih iz njega. Uz rad s podacima, svrha aplikacije je također upravljanje scraperom uz mogućnost postavljanja nekih parametara te provjera rada scrapera kroz prikaz izvršenih testova.

Rad prvo obrađuje odabranu arhitekturu sustava te tehnologije koje su se koristile u razvoju i razlog njihovog odabira. Zatim se opisuje sama implementacija kroz potpoglavlja o procesu razvoja aplikacije te implementaciji arhitekture kroz odabrane tehnologije. Unutar opisa implementacije obrađeni su i načini poboljšanja kvalitete koda uz refaktoriranje te primijenjeni način razvoja novih funkcionalnosti i alternative takvom pristupu. Sljedeće se opisuju analize koje su izvedene, izgled ulaznih podataka, sama implementacija i oblik rezultata. Zadnje poglavlje obrađuje nadogradivost aplikacije te moguća poboljšanja.

2. Arhitektura sustava

Cilj rada bio je izrada aplikacije za pregled i analizu podataka sakupljenih na internetskim portalima s vijestima. Samo sakupljanje podataka nije obrađeno u radu, već se očekuje da je radom scraper dijela aplikacije baza napunjena podacima. Osim mogućnosti pregleda i analize, aplikacija mora omogućavati upravljanje određenim konfiguracijskim podacima scrapera te imati pregled testova kojima se verificira ispravan rad scrapera.

Aplikacija je izrađena prema arhitekturi klijent poslužitelj (engl. *client-server*). U toj arhitekturi poslužitelj sadrži sve podatke, a klijent ih dobavlja slanjem zahtjeva na poslužitelj. Specifičnost ove arhitekture je mogućnost odvajanja klijenta i poslužitelja na različita računala te mogućnost pristupa više klijenata istom poslužitelju kao što je prikazano na slici 2.1 [14].



Slika 2.1: Klijent poslužitelj arhitektura [14]

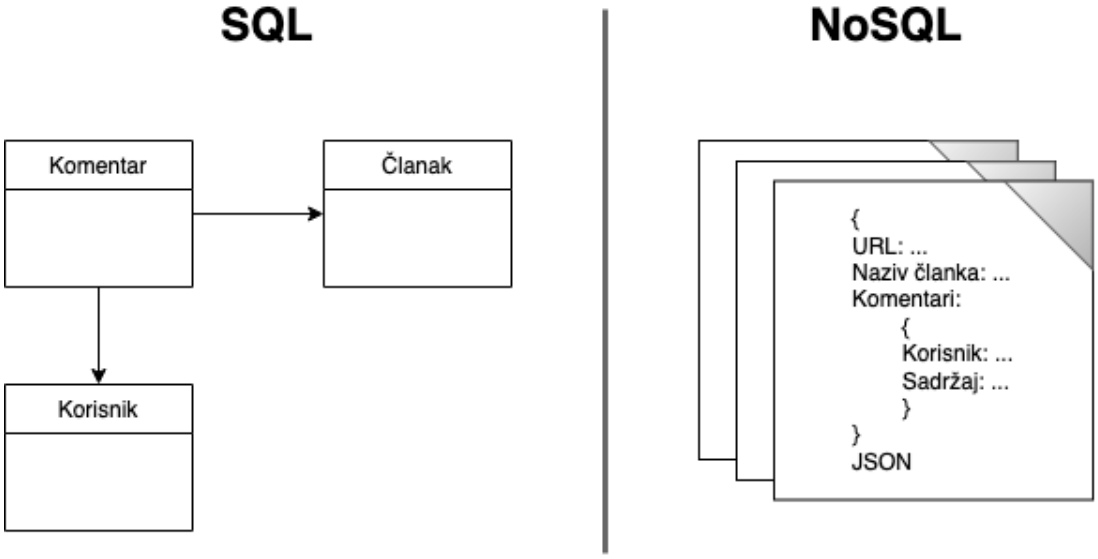
Iako se aplikacija do sada koristila samo na jednom računalu dobro je imati mogućnost odvajanja klijenta od poslužitelja ako će postojati zahtjevi za to. Drugi razlog odabira ovakve arhitekture jest mogućnost neovisnog razvoja što uključuje i razvoj u različitim tehnologijama. Komunikacija se odvija samo preko HTTP zahtjeva stoga osim mogućnosti njihovog primanja i slanja ne postoji nikakav dodatni zahtjev za uspješan rad poslužitelja i klijenta. Za trajno spremanje i dohvat podataka koristi se baza podataka, a poslužitelj komunicira s njom te dobivene podatke obrađuje i šalje u obliku JSON odgovora klijentu.

2.1. Korištene tehnologije

Sustav je razvijen koristeći tehnologije Django, Angular i MongoDB, a osim njih koristili su se alati Figma za razvoj makete (engl. *mock-up*) te Github za objavu razvijenog koda. Ovo poglavlje predstavlja tehnologije razvoja kroz kraće opise te za svaku tehnologiju daje razloge zašto je ona odabrana za razvoj aplikacije.

2.1.1. MongoDB

Baze podataka mogu se podijeliti na dvije vrste: SQL i NoSQL. Njihova glavna razlika je da su SQL baze relacijske, a NoSQL ne-relacijske. Relacijske baze podataka imaju predefinirane oblike za sve podatke i njihove međusobne veze što se postiže korištenjem tablica sa stupcima [3]. Takve strukture nema u NoSQL bazama te se u njima podaci spremaju unutar dokumenata. Oblik dokumenta koji se koristi ovisi o namjeni baze, a najčešće korišteni oblici su parovi ključeva i vrijednosti, JSON dokumenti te grafovi s rubovima i vrhovima [4]. Slikovita usporedba podataka u SQL i NoSQL bazi prikazana je na slici 2.2. Vidi se kako SQL baza koristi veze za pohranu različitih podataka dok je za povezivanje više podataka u NoSQL bazi potrebno da svi podaci budu pohranjeni u istom dokumentu.



Slika 2.2: Usporedba SQL i NoSQL baza

Razvijena aplikacija koristi MongoDB bazu koja je NoSQL tip baze te se koristi spremanje dokumenta u JSON obliku. Na odluku koja će se baza podatka koristiti najviše su utjecali struktura podataka i njihova količina. Predviđeno je da aplikacija radi sa zapisima koji sadrže članke preuzete s portala, a njih će količinski biti mnogo. S takvim velikim količinama podataka bolje rade NoSQL baze zbog svoje fleksibilnosti i

brzine. Veća brzina rada s bazom moguća je u NoSQL bazama jer nije potrebno održavati ACID(Atomicity, Consistency, Isolation, Durability) principe. Kada se baza ne brine o transakcijama, međusobnoj izoliranosti, konzistentnosti podataka te što se s podacima događa ako dođe do kvara, rad se može znatno ubrzati [1].

Osim količine, bitan faktor je bio i raznolikost strukture zapisa koja se javlja zbog drugačijih oblika podataka na različitim stranicama. Kako podaci nisu uvijek istog oblika, korištenje SQL baze podataka nije bilo moguće jer bi ono zahtijevalo zasebnu tablicu za svaki oblik zapisa što bi rezultiralo golemim i neodrživim brojem tablica. NoSQL baza omogućuje spremanje svih članaka u istu kolekciju iako su polja u zapisu različitih imena i tipa podataka. Osim raznolikosti zapisa drugi problem koji se javlja je promjenjivost. Portali mogu mijenjati kako se podaci zapisuju na njihovim stranicama te bi takve promjene uzrokovale velike probleme u sustavu sa SQL bazom. U tom slučaju bilo bi potrebno izraditi novu tablicu te za stare podatke napraviti konverziju u novi oblik ili odvojeno držati tablicu sa starim podacima. Unutar NoSQL baze takav problem neće se pojaviti jer ne postoji zadana struktura koju je potrebno zadovoljiti.

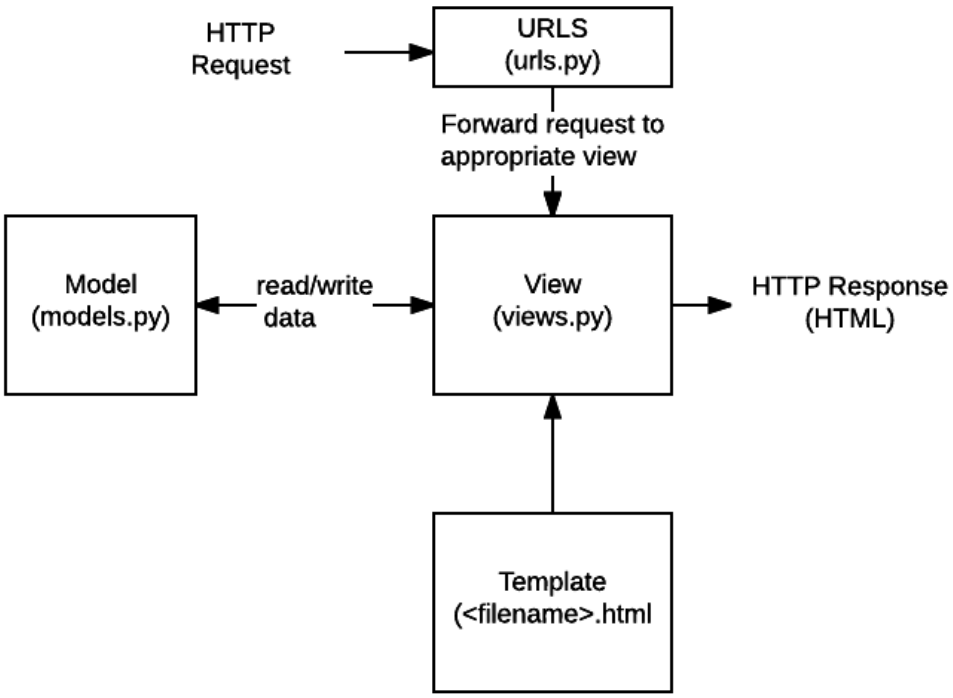
2.1.2. Django

Django je razvojni okvir za web aplikacije koji je baziran na programskom jeziku Python. Razvojni okvir je izradio skup iskusnih developera te sadrži sve komponente koje se standardno koriste u razvoju web aplikacija. Time se omogućuje lakše i brže razvijanje jer nije potrebno duplicirati standardni kod aplikacija (engl. *boilerplate code*). Django je nastao između 2003. i 2005. kada je tim koji je razvijao više različitih stranica za vijesti shvatio da se na svim stranicama ponavljao isti dio koda i neki oblikovni obrasci. Taj zajednički kod je bio izdvojen u generični razvojni okvir koji je objavljen 2005. pod nazivom Django [9].

Razvijena aplikacija koristi Django upravo zato što taj okvir ubrzava razvoj i dolazi s već gotovim dijelovima. Drugi razlog zašto je Django odabran jest zbog kompatibilnosti sa scraper dijelom aplikacije koja je napisana u Pythonu. Sličnost scrapera i aplikacije za upravljanje je omogućila da se ponovno iskoriste dijelovi koda vezani uz spajanje na bazu i dohvat iz nje što je objašnjeno u potpoglavlju 3.2.

Django projekti koriste organizaciju projekta u obliku *Model View Template* (MVT) arhitekture. Model predstavlja sloj koji radi s bazom podataka i on uključuje klase koje rade dohvat i spremanje u bazu. Model također definira strukturu podataka iz baze te se u praksi za svaku tablicu iz baze definira jedna klasa u modelu. View je funkcija za rukovanje s HTTP zahtjevima. Ona je poveznica između templatea i modela jer može raditi pozive na oba sloja. View iz modela dohvaća podatke, a zatim prosljeđuje oblikovanje odgovora templateu. Template je sloj koji definira strukturu datoteke, najčešće HTML stranice. On sadrži prazna rezervirana mjesta (engl. *placeholders*) koja se popunjavaju podacima koje view prosljeđuje iz poziva u model. Osim ova tri sloja bitna je i datoteka `urls.py` koja na temelju URL-a preusmjerava zahtjeve na pripadajući view [9]. Cijela struktura projekta i međusobne veze njegovih dijelova je prikazana na

slici 2.3.

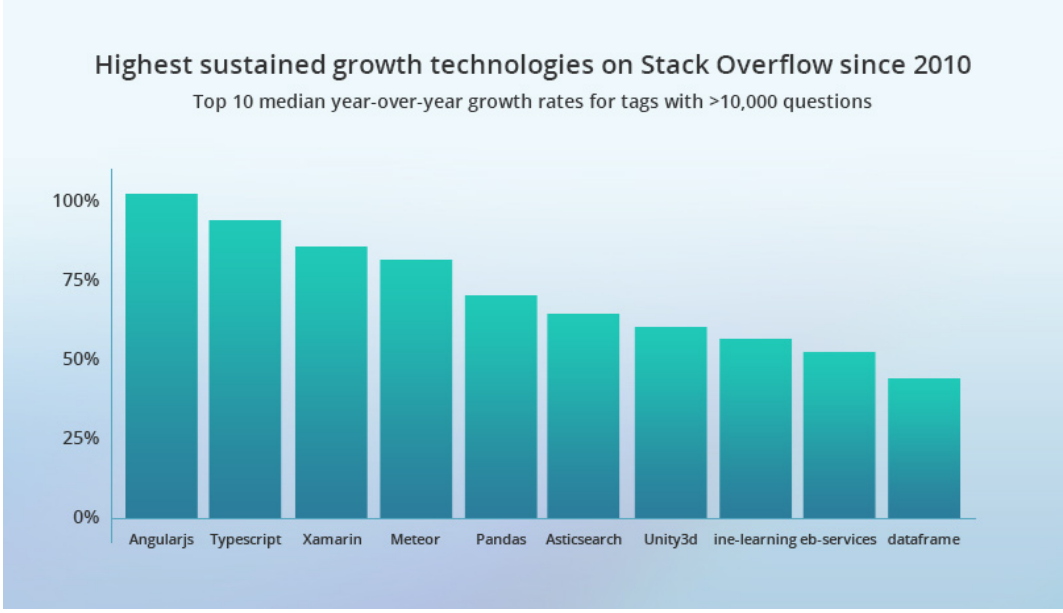


Slika 2.3: Struktura Django projekta [9]

Aplikacija razvijena u sklopu ovog rada ima nešto drugačiju strukturu. Razlog tome je to što se za razvoj sučelja (engl. *frontend*) koristi frontend razvojni okvir Angular, a ne template. Struktura razvijene aplikacije stoga ne sadrži template, već view vraća HTTP odgovor u JSON obliku te se Angular brine za ispravan prikaz takvih podataka. Druga razlika je u tome što razvijena aplikacija ne sadrži klase modela, već njih zamjenjuju servisi. Kako se koristila NoSQL baza podataka, klase modela nisu imale smisla jer one osim rada s bazom definiraju strukturu podataka koja u slučaju rada s kolekcijama ne postoji.

2.1.3. Angular

Angular je platforma za razvoj sučelja aplikacija koja se bazira na programskom jeziku Typescript te HTML-u. Platformu razvija Google kao projekt otvorenog koda (engl. *open-source*), a prvi put je predstavljen 2009. godine. Zbog jednostavnosti razvoja, Angular je danas jedna od vodećih platformi za razvoj korisničkih sučelja te ga koristi čak 44,3% softverskih inženjera [13]. Popularnost Angulara je vidljiva na grafu sa slike 2.4. gdje se nalazi prvi po prosječnom rastu među tehnologijama s više od deset tisuća pitanja postavljenih na stranici za pitanja o programskom inženjerstvu StackOverflow. Osnovni gradivni element u Angular aplikaciji je komponenta. Komponenta se definira Typescript klasom koja sadrži `@Component()` dekorator, HTML predloškom i CSS datotekom.



Slika 2.4: Tehnologije s najvećim konstantnim rastom [13]

Angular se temelji na *Model View Controller* (MVC) arhitekturi koja je ima neke sličnosti s već spomenutom MVT arhitekturom koju koristi Django. Razlika je u tome što u MVT arhitekturi controller ne postoji jer njegovu ulogu izvodi sam razvojni okvir te view sloj ima različito značenje. U MVT arhitekturi view ima ulogu "malog" kontrolera, to jest on prima HTTP zahtjeve i vraća odgovore, dok u MVC arhitekturi view sloj definira kako su podaci prikazani. Ekvivalent view sloja u MVC arhitekturi je template sloj u MVT arhitekturi [8]. U Angular aplikaciji model sadrži poslovnu logiku te se nalazi unutar komponenti koje se nazivaju servisi. View sloj definira prezentaciju podataka koristeći HTML i CSS datoteke koje se vežu uz pojedine komponente. Datoteke s nastavkom .ts sadrže logiku vezanu uz pojedine komponente te one čine controller sloj [11].

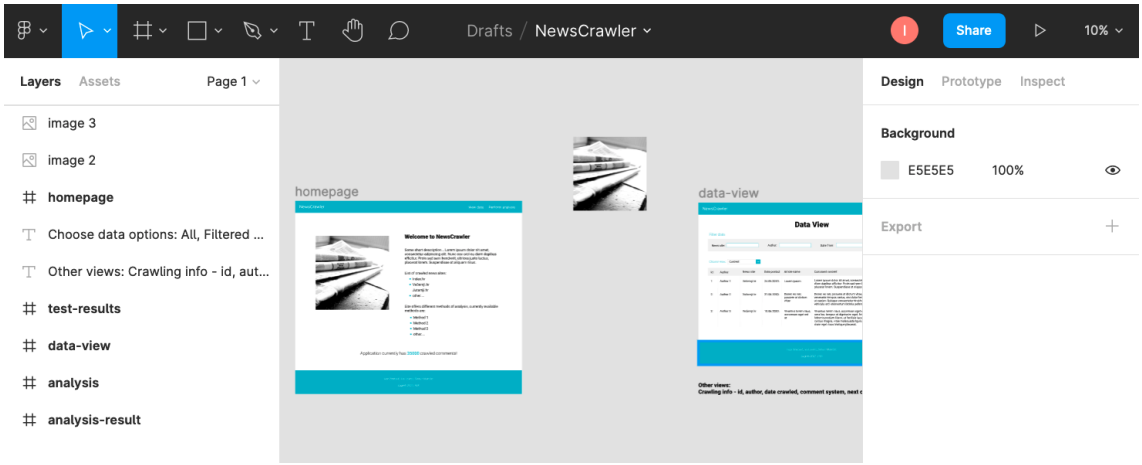
U razvoju aplikacije korišten je Angular zbog svoje jednostavnosti i popularnosti te zbog prethodnog rada i znanja o tehnologiji. Uz veliku popularnost se najčešće veže i aktivna zajednica pa je rad u takvoj tehnologiji sigurniji jer za gotovo svaki problem već postoji odgovor stoga je šansa za blokadu rada zbog problema mala.

2.1.4. Figma

Figma je online alat za uređivanje grafike i osmišljavanje dizajna korisničkih sučelja. Koristi se za razne zadatke vezane uz grafiku i dizajn kao što su makete web stranica, dizajniranje sučelja mobilnih aplikacija te prototipi dizajna. Prednost alata je to što je dostupan unutar preglednika te nije potrebno preuzimanje dodatnih aplikacija. Ovo omogućuje rad na istom projektu bez obzira na računalo ili operacijski sustav. Druga prednost Figme je trenutna kolaboracija više sudionika na istom projektu. Sve promjene su automatski i instantno spremljene u projekt što omogućuje zajednički rad u

stvarnom vremenu i osigurava istu, uvijek ažurnu verziju kod svih sudionika [10].

U ovom radu Figma je korištena za izradu maketa aplikacije. Iako nije bilo više sudionika koji istovremeno rade dizajn, mogućnost prikaza unutar preglednika je bila vrlo korisna. Time se izbjegla potreba za izvozom maketa ili instalacijom dodatnih aplikacija koje bi omogućile pregled. Osim toga, razlog za odabir Figme bio je prethodno znanje o alatu te lakoća korištenja. Jednostavno sučelje prikazano na slici 2.5. omogućuje odabir između stvaranja novog okvira koji služi za oblikovanje web stranice, dodavanja teksta ili geometrijskih oblika, a desni dio sučelja nudi dodatne opcije o trenutno odabranom objektu. Za razvoj jednostavne makete dovoljno je samo tih par funkcionalnosti stoga je korištenje intuitivno i lako.



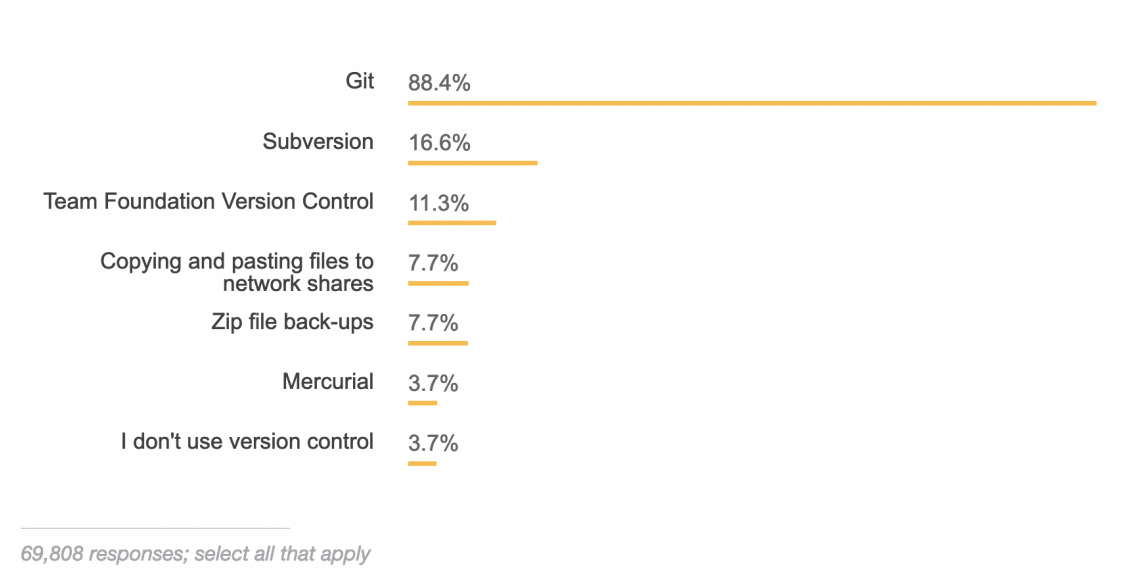
Slika 2.5: Sučelje alata Figme

2.1.5. Github

Github je stranica i web servis koji programerima omogućuje spremanje i upravljanje kodom preko kontrole verzija (engl. *Version control*). Kontrola verzija znači da se promjene ne rade izravno u originalnom kodu, već se nove funkcionalnosti i prepravke dodaju preko novih verzija. Verzije se razvijaju uz grananje (engl. *branching*) što je postupak koji duplicira dio izvornog koda, a ono omogućuje siguran rad koji neće utjecati na ostatak projekta. Drugi bitan koncept je spajanje grana (engl. *merging*) koje se provodi nakon što je novi kod razvijen i ispitan te ga je sigurno dodati u cijeli projekt [12].

Github je samo stranica koja omogućuje pohranu repozitorija, a Git je podloga koja omogućuje kontrolu verzija. Bitno je razlikovati ta dva pojma jer se Git koristi i na ostalim stranicama za održavanje (engl. *hosting*) repozitorija kao na primjer BitBucket, GitLab i SourceForge. Git je sustav otvorenog koda, a izradio ga je Linus Torvalds, kreator operacijskog sustava Linux, 2005. godine. Od tada je sustav razvio veliku popularnost te je danas najkorišteniji sustav za kontrolu verzija što je vidljivo na slici

2.6. Slika prikazuje rezultate istraživanja koje je proveo StackOverflow 2018. godine, a uključuje odgovore od skoro 70 000 profesionalnih softver inženjera.



Slika 2.6: Popularnost sustava za kontrolu verzija [7]

Github se koristio u razvoju aplikacije zbog mogućnosti kolaboracije više ljudi na istoj aplikaciji. Aplikacija za upravljanje i pregled članaka samo je dio veće aplikacije koja uključuje i skupljanje podataka s portala s vijestima. Kako bi istovremeni rad bio što lakši te kako ne bi došlo do neočekivanih problema u radu na istim datotekama, koristio se Github za kontrolu verzija.

3. Implementacija sustava

Ovo poglavlje obrađuje konkretnu izradu aplikacije. Prvo se objašnjavaju faze razvoja sustava to jest redoslijed aktivnosti koje su se provodile tijekom razvoja aplikacije te konkretna implementacija arhitekture klijent poslužitelj. Zatim se opisuju tehnike *refaktoriranja*, postupka za poboljšanje strukture koda, i daju primjeri provedenih tehnika, a na kraju se opisuje način razvijanja funkcionalnosti kroz grane (engl. *branch*) na sustavu kontrole verzija.

3.1. Faze razvoja sustava

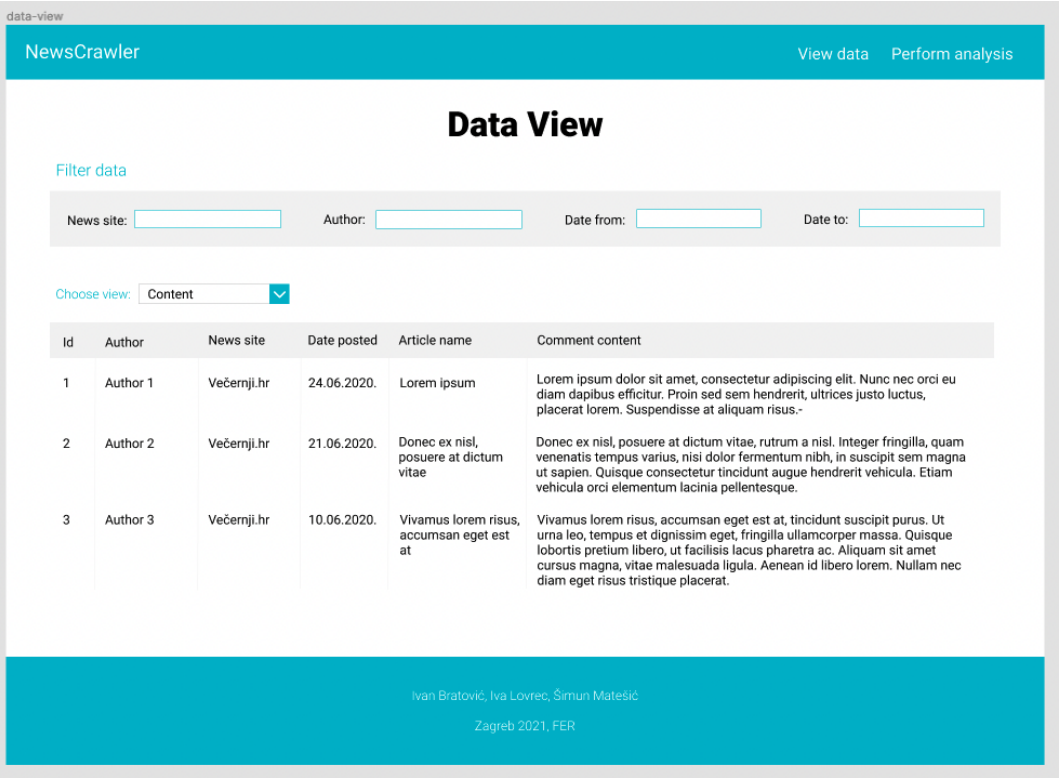
Razvoj aplikacije se događao u nekoliko različitih faza. Prva faza razvoja bilo je prikupljanje zahtjeva te su se u toj fazi utvrđivale funkcionalnosti aplikacije. Dvije najbitnije funkcionalnosti aplikacije su pregled podataka iz baze te mogućnost analiziranja podataka. Osim ovih zahtjeva u tom trenutku nisu postojali drugi zahtjevi pa je sljedeći korak bio osmisliti izgled sučelja.

Prvo je trebalo razmisliti koje će sve stranice aplikacija imati. Po zahtjevima su potrebne dvije stranice, stranica za pregled podataka te stranica za analizu podataka. Stranica za analizu treba uključivati pokretanje analize s parametrima te rezultate izvršene analize. S obzirom na to da su ovo dva odvojena zadatka, zamišljeno je da analiza bude podijeljena na stranicu za pokretanje analize te stranicu za rezultate. Osim te dvije stranice, javila se potreba i za naslovnom stranicom jer je uobičajena praksa da prva stranica nudi pregled mogućnosti, a ne konkretnu funkcionalnost aplikacije. Nakon što je određeno koje će stranice aplikacija sadržavati trebalo je analizirati svrhu svake od stranice te zaključiti koje će konkretne elemente sadržavati.

Analizom je zaključeno da naslovna stranica treba imati kratki opis aplikacije kako bi se odmah znala njezina svrha. Osim opisa, stranica će imati i dodatne informacije o aplikaciji kao što je broj članaka u bazi te podržane analize. Stranica za pregled podataka je zamišljena kao tablični prikaz. S obzirom na to da svaki zapis o članku ili komentaru ima više polja zanimljivih korisniku, najbolji oblik prikaza je uz pomoć tablice i stupaca. Stranica će sadržavati i mogućnost filtriranja jer sami prikaz tako velike količine podataka nema smisla bez opcije filtriranja zapisa. Stranica za pokretanje analize mora nuditi upis nekih parametara analize. Najbitniji parametar je naziv analize na temelju kojeg će se odabrati koja se analiza pokreće. Osim njega mogući su neki do-

datni parametri za određene analize, ali njih je moguće odrediti tek kada se znaju vrste analiza. Dodatno stranica za pokretanje analize treba sadržavati kratke opise analiza kako bi korisnik znao koja je njihova svrha i kako točno protumačiti rezultate. Pregled rezultata analize kao glavni element mora sadržavati vizualizaciju rezultata koja ovisi o vrsti analize, a također mora sadržavati i parametre s kojima je analiza pokrenuta.

Nakon što je analiza sadržaja stranica dovršena može se krenuti na izradu maketa. Makete su izrađene u online alatu Figma te su služile za vizualizaciju sadržaja stranica, a kasnije će poslužiti kao vodilja u implementaciji aplikacije. Na temelju maketa se kasnije izradio vrlo sličan dizajn korisničkog sučelja, a osim toga makete su služile kako bi se utvrdilo koje informacije je potrebno dohvatiti s poslužiteljske strane. Primjer izgleda makete stranice za pregled podataka prikazan je na slici 3.1.

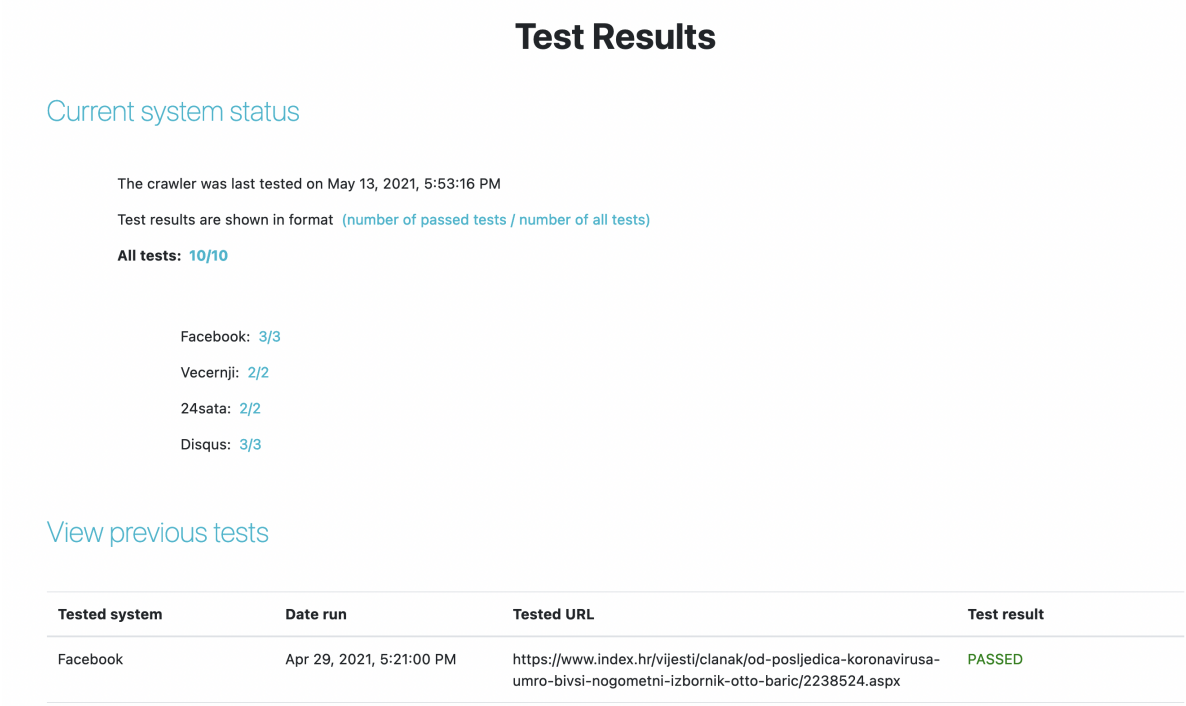


Slika 3.1: Maketa stranice za pregled podataka

Kod razvoja same aplikacije prvo je napravljena implementacija izrađenih maketa u korisničkom sučelju. Implementacija maketa je započela s izradom zaglavlja i podnožja koji su prisutni na svim stranicama. Sljedeće je bila implementirana stranica za pregled podataka. Kako poslužitelj još nije bio implementiran, podaci su bili *hardcodirani* kako bi se oponašao dohvat s poslužitelja. Uz stranicu je bila implementirana i komponenta za filter, ali je ona bila krivo izrađena jer je provodila filtriranje unutar sučelja aplikacije. Takvo filtriranje je dobro radilo u trenutnom slučaju gdje je bilo samo nekoliko zapisa, ali u stvarnom sustavu koji se razvijao bi takav način uzrokovao vrlo spore odgovore. Nakon toga je implementirana naslovna stranica s kratkim opisom aplikacije, popisom stranica čiji se članci nalaze u bazi, popisom implementiranih analiza te informacijom koliko ima komentara u bazi. Zadnje implementirane makete bile

su stranice za analizu. Kod implementacije je zaključeno da razdvajanje pokretanja i rezultata analize komplicira implementaciju jer je potrebno poslati puno parametara između komponenti. Smještanje komponente za rezultate unutar komponente za pokretanje analize omogućilo je korištenje *input* i *output* direktive koje Angular sadrži. Pomoću njih se mogu slati cijeli objekti među komponentama što je bilo vrlo korisno u ovom slučaju. Ovime je dovršena implementacija sučelja za prve zahtjeve, no u međuvremenu se javio novi zahtjev za prikazom rezultata testiranja scraper dijela aplikacije.

Nakon novog zahtjeva je prvo slijedila kratka analiza sadržaja nove stranice. Stranica bi trebala prikazivati podatke o testovima, a kako oni imaju više različitih polja koje je potrebno prikazati najbolji oblik prikaza je tablični. Testovi služe kako bi se znalo stanje sustava, to jest je li sve ispravno radi. Kako bi ta informacija bila što preglednije prikazana, zamišljeno je da se najnoviji rezultati prikazuju na vrhu stranice, odvojeno od ostalih podataka o prethodnim testovima. Nakon obavljene analize izrađena je maketa stranice, a zatim je uz male promjene ta maketa i implementirana. Slika 3.2. prikazuje izgled novo dodane stranice s testovima te se vidi kako su najbitniji podaci o trenutnom stanju sustava izdvojeni na vrhu stranice. Također na slici se vidi kako je se u tabličnom prikazu koriste boje za lakše raspoznavanje uspješnih i neuspješnih testova.



Slika 3.2: Stranica za prikaz rezultata testova

Kada je cijelo sučelje bilo implementirano krenulo se s razvojem poslužiteljske strane. Poslužitelj je već sadržavao neke funkcionalnosti koje su bile izrađene u sklopu isprobavanja tehnologija Django i Angular. Struktura projekta je bila razdijeljena u view sloj koji je pozivao model za dohvat podataka iz SQL baze. Tu strukturu je trebalo zamijeniti s novom koja je bazirana na MongoDB bazi koju puni scraper dio aplikacije.

Isprobano je nekoliko različitih načina za spajanje Django aplikacije i MongoDB baze, no niti jedan način nije radio ispravno i nije mogao koristiti već stvorenu bazu. Svi ovi načini su trebali omogućiti da se baza poziva preko Model sloja, no jedina uloga tog sloja uz upravljanje bazom je definiranje strukture podataka. Ta mogućnost nije ni bila potrebna u aplikaciji jer se struktura podataka razlikovala zbog heterogenosti izvora članaka.

Umjesto model sloja uvedeni su servisi koji su se s bazom spajali na identičan način kao i scraper. Kako bi se umjesto klasa modela mogla koristiti funkcija servisa, bilo je potrebno zamijeniti view baziran na klasi s funkcijskim. Ispis 3.1. prikazuje view baziran na klasi te se vidi kako u njemu nije moguće definirati neke kompliciranije obrade, već se samo može definirati klasa modela čiji se podaci dohvaćaju iz baze te `Serializer` koji se koristi za definiranje polja koja se šalju. Funkcijski view prikazan je u ispisu 3.2. i vidi se kako on omogućava dohvat parametara zahtjeva te poziv bilo koje funkcije čiji se rezultat zatim može poslati unutar HTTP odgovora. Nakon kreiranja novih funkcijskih viewova koji dohvaćaju podatke iz baze bilo je moguće sučelje povezati s poslužiteljem i s njega dohvatiti stvarne podatke. Sve funkcije koje su do tada vraćale *hardcodirane* podatke su bile zamijenjene s pozivima na poslužitelj te je aplikacija sada pokazivala stvarne podatke.

```
def get_data(request):
    data = Comment.objects.all()
    if request.method == 'GET':
        serializer = CommentSerializer(data, many=True)
        return JsonResponse(serializer.data, safe=False)
```

Ispis 3.1: Klasni view

```
def comments_on_domain(request):
    limit_articles = request.GET.get('limitArticle', "")
    result = analysis_service.get_domains_and_comment_number(
       limit_articles)
    return HttpResponse(result)
```

Ispis 3.2: Funkcijski view

Nakon povezivanja baze i omogućavanja prikaza stvarnih podatka u sučelju sljedeći je korak bio implementacija analiza. Prva implementirana analiza je bila broj članaka po domeni, to jest portalu s vijestima. Ovaj podatak se mogao dohvati direktno iz baze podatka uz agregaciju pa je implementacija bila jednostavna. Za prikaz rezultata u sučelju izrađena je komponenta `numbered-list` koju je moguće višestruko iskoristiti za bilo koju analizu čiji rezultat ima smisla prikazati u numeriranoj listi. Sljedeće je bila implementirana analiza prosječnog broja komentara po članku. Implementacija brojanja komentara je morala biti razdvojena na funkcije za svaki tip komentara zbog raznolikosti zapisa podataka. Za prikaz rezultata analize koristila se ista komponenta numbered-list, no ona se prvo trebala prepraviti kako bi polja u prikazu mogla biti dinamička te tako primiti različite oblike podataka.

Sljedeće je bilo potrebno urediti prikaz podataka. U tom trenutku je sučelje bilo pripremljeno za prikaz komentara, a iz baze su se dohvaćali članci pa su samo neka polja

bila popunjena. Sučelje se prenamijenilo za prikaz članaka, a uz to je bilo potrebno izraditi funkciju koja će obraditi članke tako da vrati broj komentara u članku što je korisna informacija u pregledu. Također je bilo potrebno promijeniti ispis datuma jer trenutni oblik nije čitljiv ljudima, već je to zapis u *epoch* formatu, to jest broj sekundi od 1.1.1970. Slika 3.3. prikazuje izgled tablice prije promjene, a slika 3.4. izgled nakon promjene. Vidi se kako tablica nakon prenosi više informacija koje bi mogle biti zanimljive korisniku koji pregledava članke.

Canonical URL	Comment type	Date posted	Content
https://www.tportal.hr/biznis/clanak/croatia-airlines-u-svibnju-jaca-mrezu-medunarodnih-letova-prema-jadranskim-odredistima-20210426	facebook	1619442720	
https://www.tportal.hr/autozona/clanak/foto-video-skoda-pokazala-nove-skice-fabije-sira-je-i-dulja-a-izgleda-opako-dobro-foto-20210426	facebook	1619442480	

Slika 3.3: Prikaz članaka prije promjene

Article URL	Comment type	Date posted	Number of comments
https://www.vecernji.hr/sport/srpski-reprezentativac-stize-u-nasice-nisam-previse-razmisljao-kad-su-me-zvali-1491245	vecernji	2021-05-10 16:15:00	1
https://www.24sata.hr/news/ante-3-ne-hoda-i-hrani-se-na-sondu-tri-puta-dnevno-jedva-smo-se-izborili-za-njegova-prava-761297	24sata	2021-05-10 16:15:00	173
https://www.vecernji.hr/vijesti/grlic-radman-nema-promjena-granica-na-zapadnom-balkanu-samo-treba-promjena-izbornog-zakona-u-bih-1491397	vecernji	2021-05-10 16:12:00	33

Slika 3.4: Prikaz članaka nakon promjene

Razvoj aplikacije je nastavljen s pripremom za implementaciju novih analiza. Zamišljeno je da se nove analize bave korisnicima i sadržajem komentara. Za to je bilo potrebno prvo izraditi funkciju koja će iz trenutnih zapisa članaka izvući podatke o komentaru te ih spojiti s pripadajućim korisnikom. Zbog raznolikosti podataka opet je bilo potrebno izraditi zasebnu funkciju za svaki tip komentara. Nakon što su implementirane funkcije koje obrađuju komentare mogla se dodati analiza koja broji komentare određenog korisnika te kao rezultat daje korisnike s najvećim brojem komentara.

Uz implementiranu funkciju za obradu komentara moglo se prijeći na analizu sadržaja komentara. Analiza bi trebala prepoznati agresivnost i ljutnju u komentarima te rezultirati popisom korisnika koji su se najagresivnije izražavali u svojim komentarima. Da bi se implementirala ovakva analiza bilo je potrebno istražiti kako se može prepoznati agresivnost unutar teksta. Nakon istraživanja je zaključeno da je najbolji odabir Emolex, rječnik koji povezuje riječi s emocijama uz određeni indeks. Analiza je implementirana uz korištenje Emolexa te je u sučelju također prikazana unutar numbered-list komponente.

U ovom trenutku se pojavio još jedan zahtjev, a to je razvoj stranice za prikaz i uređivanje konfiguracijskih podataka. Stranica bi trebala prikazivati konfiguraciju scraper dijela aplikacije te imati mogućnost izmjene nekih parametara. Pregledom konfiguracijskih datoteka zaključeno je da većina parametara ne bi trebala biti dostupna korisniku za promjenu te je potrebno implementirati samo pregled stranica koje se dohvaćaju scraperom. Na poslužitelju se izrađuju funkcije koja čitaju i uređuju konfiguracijsku datoteku s podacima o portalima, a na sučelju se implementira stranica koja prikazuje podatke i omogućuje njihovo uređivanje.

Na kraju je bilo potrebno napraviti neke prepravke unutar aplikacije kako bi bolje i brže radila. Prvo poboljšanje odnosi se na prikaz podataka. On trenutno sadrži samo tablicu te se automatski dohvaća prvih dvadeset komentara odnosno članaka. Da se lakše upravlja s dohvaćenim podacima dodana su polja za definiranje broja komentara i članaka, sortiranje i mogućnost odabira između dohvata iz baze i obrade podataka od scraper. Kod prikaza komentara dodana je i mogućnost filtriranja komentara po imenu autora. Time je omogućeno lakše snalaženje u prikazu podataka. Drugo poboljšanje aplikacije je izvedeno uz definiranje parametara analize. Za dugotrajnije analize je omogućeno upisivanje broja članaka koji će se obraditi pa se time riješio problem gdje kod baza s puno podataka analiza nije imala rezultat zbog preduge obrade.

3.2. Implementacija arhitekture sustava

Odabrana arhitektura klijent poslužitelj implementirana je uz korištenje tehnologija Django i Angular, a NoSQL baza MongoDB je korištena za implementaciju baze podataka. Ovo poglavlje obrađuje svaki sloj zasebno te opisuje njihovu strukturu i najbitnije datoteke.

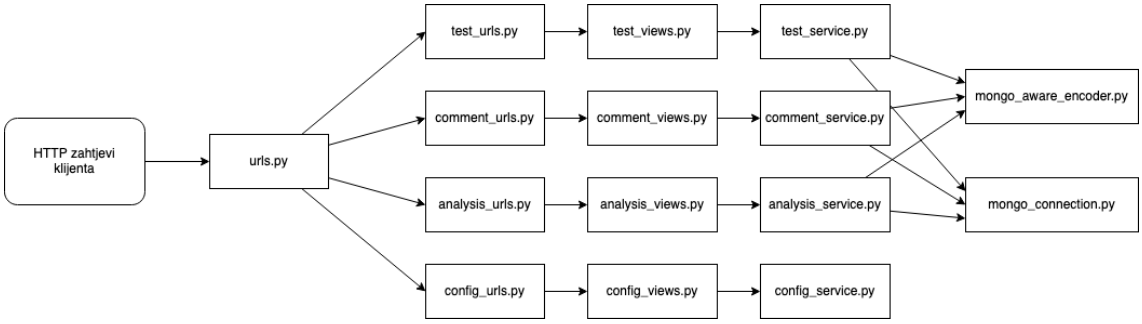
3.2.1. Baza podataka

Baza podataka se na početku rada aplikacije sastoji samo od kolekcija koje je izradio scraper, a to su kolekcije `comments` i `tests`. Kolekcija `comments` sadrži zapis za svaki članak te u sebi sadrži i podatke o komentarima tog članka. Kolekcija `tests` sadrži rezultate testova, a kako bi kolekcije bile prisutne u bazi potrebno je prvo pokrenuti scraper i skriptu za testiranje. Većina analiza i prikaza podataka stvaraju nove

kolekcije u bazi koje služe tome da se aplikacija ubrza. Jednom obrađeni podaci iz scrapera spremaju se u zasebne kolekcije kako za sljedeći dohvat ili analizu ne bi bilo potrebno opet obrađivati iste podatke. Dodatne kolekcije koje su tako stvorene su `articles`, `user_comments` i `user_aggression`. Spajanje baze s poslužiteljom je izvedeno uz korištenje `pymongo` knjižnice.

3.2.2. Poslužitelj

Poslužitelj je Django projekt te je njegova struktura podijeljena u nekoliko različitih slojeva. Glavni slojevi su servisi, viewovi i urlovi, a osim toga projekt sadrži konfiguraciju i paket za spajanje na bazu. Struktura projekta i kako su dijelovi povezani prikazana je na slici 3.5. Zbog bolje preglednosti izbačene su dodatne pomoćne klase analize.

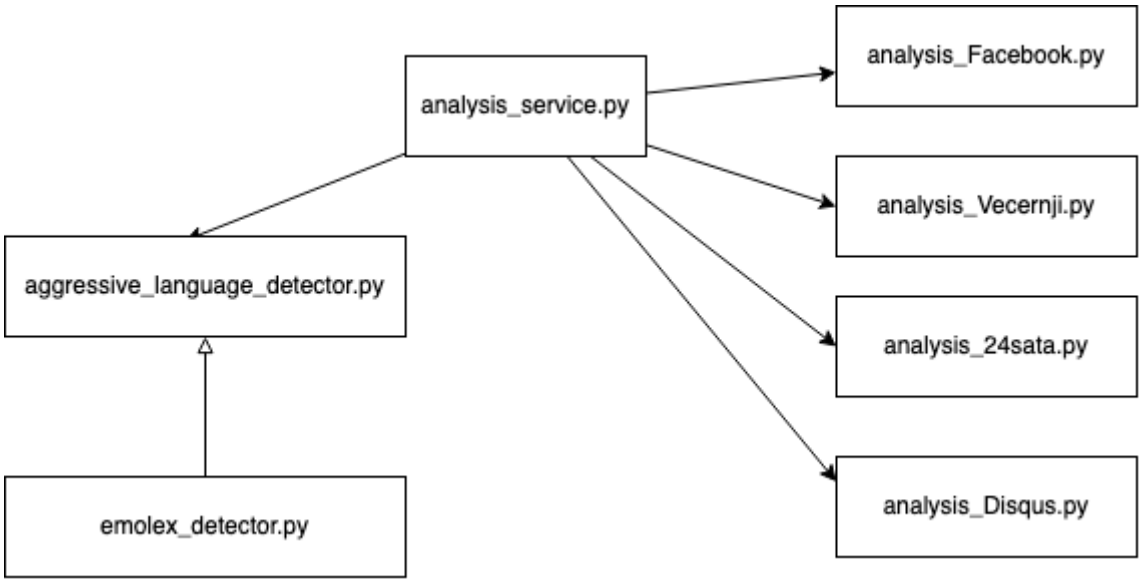


Slika 3.5: Dijagram strukture poslužitelja

Prvi sloj prema bazi je klasa `MongoConnection` koja se koristi za vezu s bazom podataka. Klasa definira samo jednu metodu koja vraća Mongo klijenta, a pri inicijalizaciji koristi konfiguracijsku datoteku `MongoService.ini`. Veza s bazom definirana je na isti način kao i u scraper dijelu.

Sloj servisa koristi klasu `MongoConnection` za pristup bazi, a taj sloj uključuje servis za svaki od bitnih entiteta u sustavu. Glavni entiteti sustava su komentari, članci, testovi, analize i konfiguracija stoga postoji servis za svaki od njih. Servisi za članke, komentare i testove služe za dohvat podataka iz baze, dok servis za konfiguraciju nema pristup bazi, već koristi `configparser` za čitanje i uređivanje konfiguracijske datoteke. Servis za analize radi obrade podataka iz baze te spremanje obrađenih podataka. Osim ovih servisa postoje još i pomoćni servisi. Oni uključuju servise za svaki od tipa komentara, servis za enkodiranje podataka i detektor agresivnog jezika. Servisi za komentare služe za obradu pojedinog tipa komentara i oblikovanje rezultata u isti format kako bi ga servis za analize mogao zajednički obraditi. Servis za enkodiranje podataka služi za pretvorbu iz MongoDB formata u JSON format koji se može poslati kao HTTP odgovor. Detektor agresivnog jezika se koristi u analizi najagresivnijih korisnika, a postoji nadklasa `AggressiveLanguageDetector` i podklasa `EmolexDetector` koja ju implementira. Odabrana je takva struktura tako da je lako zamijeniti `EmolexDetector` s nekim drugim detektorom, a jedini uvjet je imple-

mentacija metode `run_detection` koju definira nadklasa. Slika 3.6. daje detaljniji prikaz klase `analysis_service` i pomoćnih servisa koje ona koristi.



Slika 3.6: Dijagram klasa vezanih uz analizu

Iznad servisnog sloja nalazi se view sloj koji poziva funkcije servisa i njihove rezultate šalje unutar HTTP odgovora. Svaka klasa u view sloju postavlja varijablu s konekcijom na bazu i varijablu koja pohranjuje instancu servisa koji se poziva. Osim poziva servisa ovaj sloj služi za dohvat varijabli definiranih u URL-u. Varijable se na temelju imena izdvajaju iz primljenog zahtjeva, a moguće je definirati i pretpostavljene vrijednosti. Primjer dohvata varijabli iz URL-a prikazan je u ispisu 3.3. gdje se vidi dohvat tri varijable različitih tipova i definiranje njihovih pretpostavljenih vrijednosti. View sloj je kao i servisni sloj razdijeljen u datoteke ovisno o entitetu s kojim se bavi.

```
def fetch_user_comments(request):
    from_database = request.GET.get('fromDatabase', True) == 'true'
    limit_comment = request.GET.get('limitComment', 10)
    limit_articles = request.GET.get('limitArticle', "")
    result = analysis_service.get_users_and_comments(from_database,
        int(limit_comment), limit_articles)
```

Ispis 3.3: Dohvat varijabli iz URL-a

Koji se view poziva za određeni URL definirano je u datotekama direktorija `url_paths` koje su također podijeljene ovisno o entitetu. Urls datoteke sadrže samo relativni dio puta unutar URL-a za pojedini entitet. Te se datoteke zatim koriste u `urls.py` datoteci koja definira putanje unutar cijele aplikacije. Primjer `urls.py` datoteke koja uključuje putanje definirane u datotekama specifičnim za entitete je prikazan u ispisu 3.4.

```
urlpatterns = [
   path('', TemplateView.as_view(template_name='homepage.html')),
   path('comments', include('managementBackendApp.url_paths.
       comment_urls')),
```

```
    path('analysis', include('managementBackendApp.url_paths.
        analysis_urls')),
    path('tests', include('managementBackendApp.url_paths.test_urls')
        ),
    path('config', include('managementBackendApp.url_paths.
        config_urls'))
]
```

Ispis 3.4: Izgled aplikacijske urls.py datoteke

3.2.3. Sučelje

Za razvoj sučelja je korišten Angular te je projekt također razdijeljen na nekoliko dijelova. Sučelje se sastoji od servisa, zajedničkih komponenti, stranica, asseta, modela te predefiniranih Angular datoteka za cijelu aplikaciju. Struktura jedne komponente i svih najbitnijih dijelova vezanih uz nju prikazana je na slici 3.7. Model i asseti nisu prisutni na dijagramu jer ih ne koriste sve komponente.

Slika 3.7: Struktura Angular komponente i najbitnije veze

Servisi nude funkcije koje šalju zahtjeve na poslužitelj. Kao i na poslužitelju servisi su podijeljeni u datoteke ovisno o resursu s kojim rade. Svaki servis koristi bazni URL koji je zadan u `environment.ts` datoteci te na njega nadodaje entitet kojim se bavi. Svaka funkcija servisa zatim dodaje još dio puta potreban da se specificira funkcija unutar resursa te također može primati argumente koje postavlja u upit (engl. *Query string*) URL-a.

Direktorij `shared` služi za pohranu zajedničkih komponenti, to jest komponenti koje se koriste više puta na različitim stranicama. U ovom projektu to su samo dvije komponente, `header` i `footer`. Kako te komponente trebaju biti prisutne na svakoj stranici aplikacije, one nisu dodane unutar HTML koda zasebnih komponenti, već u HTML cijele aplikacije. On je definiran u datoteci `app.component.html` koja je prikazana na ispisu 3.5. Oznake `<app-header>` i `<app-footer>` označavaju implementi-

rano zaglavlje i podnožje, a ostatak stranice će biti prikazan unutar `router-outlet` oznake.

```
<body>
<app-header></app-header>
<div id="page-container">
  <div id="content-wrap">
    <router-outlet></router-outlet>
  </div>
  <footer id="footer">
    <app-footer></app-footer>
  </footer>
</div>
</body>
```

Ispis 3.5: Datoteka app.component.html

Unutar mape `pages` su smještene sve ostale stranice koje su jedinstvene i ne ponavljaju se unutar aplikacije. Za svaku stranicu je izrađen poddirektorij jer je svaka stranica zapravo Angular komponenta koja se sastoji od ponašanja definiranog u `.ts` datoteci, stila definiranog u `.css` datoteci te prikaza podataka definiranog u `.html` datoteci. Osim toga komponenta i ima .spec.ts datoteku, no ona služi testiranju koje se nije provodilo u sklopu ove aplikacije.

Modeli služe za definiciju strukture odgovora poslužitelja. Kada je odgovor poslužitelja uvijek istog oblika i ima određena polja dobra je praksa definirati taj oblik unutar sučelja. Time se postigla mogućnost provjere strukture i nadopune imena polja ako razvojno okruženje (engl. *integrated development environment*) to podržava. Uz definirane modele Angular javlja grešku ako se pokušava koristiti naziv polja koji nije definiran unutar modela što je prikazano na ispisu 3.6.

```
Error: src/app/pages/show-data/show-data.component.html:55:21 -
    error TS2551: Property 'publishedd' does not exist on type '
    Article'. Did you mean 'published'?

55        <th>{{comment.publishedd}}</th>
                        ~~~~~~~~~~

  src/app/pages/show-data/show-data.component.ts:7:16
    7   templateUrl: './show-data.component.html',
                     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    Error occurs in the template of component ShowDataComponent.
```

Ispis 3.6: Ispis Angular konzole u slučaju greške u modelu

`asset` mapa se koristi za sve datoteke koje se koriste na stranicama, ali nisu izvorni kod kao na primjer slike. Mapa dodatno ima podmapu `images` kako bi se datoteke odvojile po tipu. U ovoj aplikaciji su definirane samo dvije slike, `newspaper.png` koja se koristi za uljepšavanje sadržaja naslovne stranice te `dropdown.png` što je strelica za komponentu padajućeg izbornika koja je prilagođena ostatku dizajna.

Osim prethodno definiranih direktorija, Angular aplikacije imaju nekoliko datoteka koje se uvijek stvaraju uz aplikaciju. Najbitnije takve datoteke su `app.component.html`, `app.module.ts`, `app-routing.module.ts` i `styles.css`.

Datoteka `app.module.ts` služi za definiciju svih modula aplikacije. Ovdje je potrebno deklarirati sve izrađene komponente, a ako one nisu deklarirane Angular će prilikom poziva komponente dići iznimku da ona ne postoji. Osim vlastitih komponenti, datoteka služi za deklaraciju uvezenih modula koje aplikacija koristi kao na primjer `HttpClientModule` i `BrowserModule` koji su automatski prisutni u projektu nakon njegove kreacije. Datoteka `styles.css` služi za definiranje CSS stilova koji će se koristiti u cijeloj aplikaciji. Odabir između smještanja u stil komponente ili aplikacije ovisi o tome je li očekivano da će se taj stil koristiti u više različitih komponenti. Ako će se koristiti onda `styles` datoteka omogućuje definiranje tog stila na samo jednom mjestu, a ne u svakoj komponenti zasebno. Datoteka `app.component.html` definira HTML kod cijele aplikacije. Ako je potrebno da se neki element prikazuje na svakoj stranici onda se taj element dodaje u ovu datoteku. Bitan dio ove datoteke je oznaka `<router-outlet>` koja omogućuje prikaz različitih komponenti koje ovise o trenutnom URL putu unutar aplikacije. Same rute aplikacije se definiraju unutar datoteke `app-routing.module.ts` gdje se u polje ruta dodaje objekt s putom i nazivom komponente koju je potrebno prikazati kada je aktivna ta ruta.

3.3. Refaktoriranje

Refaktoriranje je restrukturiranje postojećeg izvornog koda tako da se mijenja samo njegova unutrašnja struktura dok ponašanje ostaje isto. Refaktoriranje najčešće čini niz manjih promjena koje sveukupno rade veliku promjenu na sustavu. Kako su promjene male, šansa da nešto pođe po krivu je također manja. Tijekom razvoja dobro je provoditi refaktoriranje kontinuirano te prije svake nove funkcionalnosti napraviti promjene ako se ona prirodno ne uklapa u postojeći kod, no to najčešće nije slučaj [6]. U ovom radu se refaktoriranje nije koristilo prije svake funkcionalnosti, već kada je struktura projekta postala loša, a izvorni kod nepregledan. Ovo poglavlje dati će pregled refaktoriranja provedenih na aplikaciji.

3.3.1. Tehnike refaktoriranja

Tijekom razvoja koristilo se nekoliko različitih tehnika refaktoriranja kako bi izvorni kod ostao čišći i pregledniji. Većinom se potreba za refaktoriranjem javljala kada su klase imale previše odgovornosti i sadržavale prevelik broj funkcija. Da bi se takve klase razdvojile na manje klase korištena je metoda vađenja klasa (engl. *Extract class*). Također je bilo potrebno neke dijelove metoda izdvojiti u zasebne metode (engl. *Extract method*) kako bi kod bio razumljiviji. Određene funkcije su bile premještene u druge klase (engl. *Move method*) kada se shvatilo da je nova struktura bolja. Zbog bolje nadogradivosti sustava koristila se i tehnika izdvajanja nadklase (engl. *Extract Superclass*), a za održavanje konzistentnosti i smanjenje broja instanciranja koristila se zamjena vrijednosti s varijablom (engl. *Extract Variable*) [5].

3.3.2. Primjeri refaktoriranja

Prvo veće refaktoriranje je provedeno nakon implementacije funkcija koje se spajaju na bazu. Servisi su već bili razdvojeni u ovisnosti o entitetu, ali se u svakom od njih ponavljala ista logika za izradu Mongo klijenta pa se odlučilo za tehniku vađenja klase. Ta se logika izdvojila u novu klasu `MongoConnection` koja pri inicijalizaciji izvodi čitanje konfiguracije i spajanje na bazu te ima metodu `getMongoClient` koja vraća klijenta. Sada servisi više ne moraju znati koja je konfiguracija baze, već se mogu samo spojiti na nju preko Mongo klijenta.

Drugi problem koji se javlja u aplikaciji jest to da je svaka funkcija u viewu stvarala novu instancu servisa koji joj je potreban. Time je za isti entitet postojalo nekoliko instanci servisa. Instanciranje servisa se stoga prebacilo u varijablu na razini klase, kako bi se koristila uvijek ista instanca te je time iskorištena tehnika zamjena vrijednosti s varijablom. Ispis 3.7. prikazuje funkciju prije refaktoriranja, a ispis 3.8. prikazuje izvorni kod nakon.

```
def articles_on_domain(request):
    result = AnalysisService(mongo_connection).
       get_domains_and_article_number()
    return HttpResponse(result)
```

Ispis 3.7: View funkcija prije refaktoriranja

```
analysis_service = AnalysisService(mongo_connection)

def articles_on_domain(request):
    result = analysis_service.get_domains_and_article_number()
    return HttpResponse(result)
```

Ispis 3.8: View funkcija nakon refaktoriranja

Dok je broj funkcionalnosti poslužitelja bio mali sve funkcije servisa su se pozivale iz istog viewa, te je on morao sadržavati sve korištene servise. Da se razvoj nastavio ovako ta view klasa bi brzo postala nepregledna i prevelika. Uz to se krši princip jedne odgovornosti jer je klasa radila sa svim entitetima i funkcionalnostima u aplikaciji. Bilo je potrebno provesti izdvajanje klasa te je podjela napravljena po entitetima kao i u servisima. Sada se za svaki view instancirao jedan servis i on je radio sa samo jednim entitetom. Za dodavanje novih funkcionalnosti je stoga bilo potrebno samo odrediti s kojim se entitetom radi te izmijeniti jedan pripadajući view i servis. Izmjena je riješila problem prenatrpane view klase, ali url datoteka je i dalje sadržavala sve puteve unutar aplikacije stoga se i ona razdvojila na više datoteka u ovisnosti o entitetu. Pojedine urls datoteke sadržavale su relativne puteve, a `urls.py` na razini aplikacije je definirala dio puta ispred relativnih puteva. U nju je trebalo uključiti sve nove urls datoteke kako bi one bile prepoznate u aplikaciji.

U početku razvoja sučelje je sadržavalo samo jedan servis koji je pozivao sve funkcije dostupne na poslužitelju te se opet javio problem prenatrpanosti. Slično kao i na poslužitelju izradio se zaseban servis za svaki entitet. Nakon razdvajanja u različite

servise bilo je potrebno izmijeniti pozive servisa u komponentama. Pozivi servisa koristili su isti bazni URL aplikacije te određeni nastavak koji je označavao entitet. Kako bi se osiguralo od nekonzistentnih promjena, definicija baznog URL-a se premješta u `environment` konstantu unutar polja `apiUrl` što je prikazano u ispisu 3.9. Servisi zatim umjesto definiranja cijelog URL-a u svakoj funkciji koriste konstantu kao što je prikazano u ispisu 3.10. koja se sastoji od environment konstante s baznim URL-om i nastavka za entitet. Za ovo refaktoriranje iskorištena je tehnika zamjene vrijednosti s varijablom.

```
export const environment = {
  production: false,
  apiUrl: 'http://127.0.0.1:8000/newsCrawler'
};
```

Ispis 3.9: Postavljanje environment konstante

```
const BASE_URL = environment.apiUrl + "/comments";

... definicija klase i konstruktor

getCommentTypes(): Observable<any>{
    return this.http.get<any>(BASE_URL+"/types");
  }
```

Ispis 3.10: Korištenje environment konstante

Tijekom implementacije analiza također se javila potreba za refaktoriranjem. Klasa `analysis_service` je sadržavala metode za obradu određenih vrsta komentara te se predvidjelo da će broj takvih metoda rasti s implementacijom novih analiza. Koristila se tehnika vađenja klase te je za svaki tip komentara izrađena pomoćna klasa. Servis za analize je onda oslobođen implementacijskih detalja i posebnosti svakog tipa komentara te klasa postaje preglednija.

Analiza najagresivnijih korisnika uvela je u `analysis_service` ovisnost o još jednoj datoteci te je sama implementacija bila komplicirana. Kako bi se detalji implementacije sakrili od servisa te kako bi se postigla nadogradivost, tehnikom vađenja klase je funkcija za detekciju izdvojena u novu klasu. Također je provedeno izdvajanje nadklase kako bi se trenutna klasa lako mogla zamijeniti s drugom implementacijom detekcije.

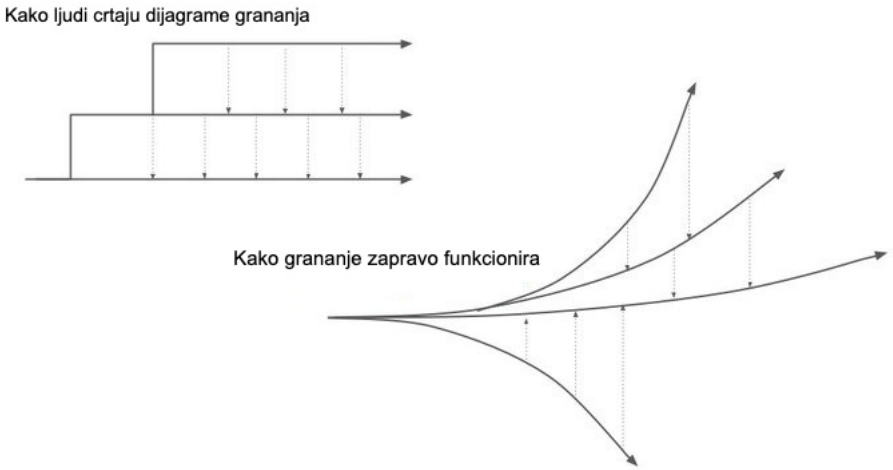
3.4. Dodavanje novih funkcionalnosti

U razvoju aplikacije korišten je Github repozitorij za spremanje koda, a razvijanje novih funkcionalnosti se radilo na zasebnim granama. Ovakav način razvoja naziva se *Feature Branch Workflow*, a glavna ideja je dodavanje funkcionalnosti preko grana umjesto s izravnim izmjenama glavne grane (engl. *main*). Jedna od najbitnijih prednosti ovog pristupa jest to da glavna grana uvijek sadrži kod koji ispravno radi jer se

spajanje provodi tek nakon detaljnog ispitivanja. Druga prednost je mogućnost istovremenog rada koja se najčešće provodi tako da svaki programer radi na svojoj grani. Kako grane međusobno ne utječu jedna na drugu, razvoj se može izvoditi neovisno. Prilikom spajanja grane s glavnom granom radi se *Pull Request*, zahtjev za spajanjem koji najčešće mora odobriti neki drugi programer te se tako radi dodatna provjera novo dodanog koda [2].

Način razvoja kroz grane ima i neke mane, a jedna od njih je onemogućavanje rane detekcije problema. Ako programeri razvijaju funkcionalnosti sasvim nezavisno, moguća preklapanja u promjenama će biti otkrivena tek kada se napravi *Pull Request*. Tada će trebati više vremena da se isprave nekonzistentnosti te će promjene najvjerojatnije trebati provesti u obje grane. Drugi nedostatak je što ovaj način razvoja otežava refaktoriranje jer postoji određeni strah od promjena kako se ne bi narušila funkcionalnost koja se razvija u drugoj grani. Rijetko provođenje refaktoriranja najčešće rezultira nečitkim i kompliciranim izvornim kodom nad kojim je teško raditi i predvidjeti posljedice promjena [15].

Kako bi se izbjegli navedeni problemi, potrebno je razmisliti o načinu rada i je li on kompatibilan s razvojem u granama. Ako se razvoj novih funkcionalnosti najčešće dovršava unutar nekoliko dana, mogu se izbjeći problemi s integracijom kod spajanja u glavnu granu, no za dulji razvoj ovaj način nije pogodan. Grane koje su se odvojile od glavne grane prije tjedan dana ili čak mjesec dana se u trenutku spajanja većinom jako razlikuju od glavne grane pa je usporedba promjena daleko zahtjevnija [15]. Slika 3.8. prikazuje zašto je razvoj u granama teško provoditi uz dugovječne grane. Vidi se kako s vremenom grane sve više divergiraju od početne točke stoga je integracija sve teža što više vremena prođe.



Slika 3.8: Prikaz grananja [18]

Postoje i drugi načini razvoja funkcionalnosti, a jedna od njih je *Trunk-based Development*. Taj način razvoja se bazira na radu u istoj grani, a kako bi se promjena stavila na tu granu prvo se mora lokalno testirati. Najčešće se koristi u manjim organizaci-

jama gdje programeri blisko surađuju kako bi se smanjila kompleksnost integracije. Nedostatak tehnike je mogućnost nepredviđenog defekta na jedinoj grani zbog čega je razvoj u zastoju dok se defekt ne riješi. Druga moguća strategija grananja je *Release Branching* gdje se svako izdanje (engl. *release*) razvija u zasebnoj grani, a strategija se najčešće koristi u vodopadnom i *Scrummerfall* modelu razvoja. Kod razvoja s više ljudi ovaj način može biti problematičan za upravljanje te osim za kratke periode između izdanja rezultira kašnjenjem i teškom integracijom. Zadnji način razvoja je *Story Branching* gdje se za svaki zadatak stvara nova grana. Strategija je slična razvoju po funkcionalnostima, no jedna korisnička priča može sadržavati više funkcionalnosti ili samo određeni dio neke funkcionalnosti. Primjer toga bi bio razvoj sučelja za novu funkcionalnost bez implementacije na poslužitelju. Ovakav razvoj se najviše koristi u organizacijama s agilnim razvojnim procesima gdje su novi zahtjevi prezentirani u obliku korisničkih priča [17].

Dodavanje nove funkcionalnosti u aplikaciju započinje sa stvaranjem nove grane. Grana se najčešće stvara iz main grane, ali moguće je stvaranje i iz druge grane ako nova funkcionalnost ovisi o nečemu što je implementirano na toj drugoj grani, a još ne postoji u glavnoj. Za stvaranje nove grane prvo je potrebno pozicionirati se u granu s koje se želi dalje razvijati uz naredbu `git checkout branch-name`. Nova grana se zatim stvara s naredbom `git checkout -b new-branch-name`. Nakon toga se u grani razvija nova funkcionalnost, a kada je ona gotova i ispitana može se spojiti u glavnu granu. Spajanje se ne radi uz korištenje Git naredbi, već je potrebno na Github otvoriti novi Pull Request za granu gdje je funkcionalnost implementirana. Tada je moguće dodati recenzente koji će dodatno provjeriti promjene u kodu i komentirati ako naiđu na problem ili ne razumiju svrhu nekog odsječka koda. Osim uz recenzente moguće je dodati provjeru uz različite alate za provjeru kvalitete koda kao što je Sonar ili alate za izgradnju aplikacije kao što su Travis ili Jenkins. Nakon što grana prođe sve provjere može se spojiti s glavnom granom.

4. Analiza podataka

Aplikacija sadrži ukupno četiri vrste analize. Prve dvije analize bave se domenama, to jest portalima s vijestima, a druge dvije analize se tiču podataka o komentarima i njihovim autorima. Samo jedna analiza,a to je analiza članaka na stranici je mogla biti provedena jedinstveno za sve tipove komentara, dok su ostale analize zahtijevale zasebnu obradu za svaki tip komentara.

4.1. Broj članaka po stranici

Prva i najjednostavnija analiza koja je bila implementirana je analiza broja članaka po domeni. Scraper u bazi stvara samo jednu kolekciju za podatke o člancima i komentarima, a to je kolekcija `comments` koja zapravo sadrži jedan zapis po članku. Svaki zapis u bazi sadrži URL, vrijeme objave, tip komentara i zapis komentara u nekom obliku, a ostala polja većinom ovise o tipu komentara. Jedina informacija koja je potrebna iz zapisa je URL jer se iz njega može iščitati domena po kojoj će se zapisi grupirati. Grupiranje je moguće uz korištenje agregata nad bazom. Unutar upita za bazu se izrađuje nova varijabla domain koja sadrži domenu. Kako su svi URL-ovi oblika `https://some-domain.com`, do naziva domene je moguće doći uz razdvajanje URL-a po znaku / te uzimanjem treće vrijednosti. Nova varijabla se zatim koristi kao `_id` po kojem se zapisi grupiraju, a za izračun broja članaka je potrebno samo uvećati varijablu `count` za jedan kod svakog pronađenog zapisa. Unutar upita je također moguće definirati sortiranje pa je rezultat sortiran po polju `count` u padajućem redoslijedu. Takav *sort* znači da će lista kretati od portala s najvećim brojem članaka što je i najzanimljivija informacija za prikazati korisniku. Upit i poziv na bazu koji se koriste prikazani su u ispisu 4.1.

```
query = [{"$project":
             {"domain": {"$arrayElemAt": [{"$split": ["$url", "/"]},
                  2]}}
          },
         {"$group":
             {"_id": "$domain", "count": {"$sum": 1}}
          },
         {"$sort":
             {"count": -1}
          }]
doc = self.comments_collection.aggregate(query)
```

Ispis 4.1: Upit za dohvat broja članaka po domeni

4.2. Broj komentara po članku

Sljedeća implementirana analiza bila je prosječan broj komentara po članku. Kako doći do informacije o broju članaka je već poznato iz prethodne analize, ali je sada uz tu informaciju bilo potrebno i saznati broj komentara. Način spremanja komentara unutar zapisa članka ovisi o tipu komentara zbog čega je bilo potrebno izraditi zasebnu funkciju za svaki od tipova komentara. Analiza je prvo bila implementirana tako da same funkcije dohvaćaju sve članke iz baze s tipom komentara za koji su zadužene. Ovakav način nije dobro funkcionirao za velike baze jer se nije mogao definirati neki manji broj članaka koji bi se obradio. Poboljšana analiza koristi funkciju za prikaz članaka i bazu koja je ta funkcija stvorila. Za prikaz članaka je bila potrebna obrada jer se smatralo da je broj komentara u članku zanimljiva informacija za prikazati. Obrada svoje rezultate sprema u bazu te ti rezultati sadrže URL i broj komentara što su jedine informacije koje su potrebne za provedbu ove analize. Iz tog razloga analiza prvo poziva funkciju koja obradi članke i spremi potrebne informacije u bazu u obliku prikazanom na ispisu 4.2.

```
{
    "_id": ObjectId("60c3a084603ba8a50e709362"),
    "canonicalUrl": "https://www.tportal.hr/biznis/clanak/naslov-
       nekog-clanka",
    "url": "https://www.tportal.hr/biznis/clanak/naslov-nekog-clanka
       ",
    "published": "2021-04-26 13:12:00",
    "commentCount": 0,
    "commentType": "facebook"
}
```

Ispis 4.2: Oblik zapisa u kolekciji articles

Nakon toga je moguće dohvatiti rezultate iz baze i prvo nad njima filtrirati sve članke koji su duplicirani što se može zaključiti na temelju istog URL-a. Filtrirane rezultate zatim treba grupirati po domeni te zbrojiti sve komentare koje se nalaze unutar članaka i prebrojati ukupan broj zapisa to jest članaka unutar pojedine domene. Uz dovršeno grupiranje su prisutni svi podaci potrebni za rezultat koji sadrži prosječan broj komentara po članku. Rezultat je prikazan u redoslijedu od portala s najvećim prosječnim brojem komentara, a uz tu informaciju prikazan je i podatak o ukupnom broju komentara i članaka što je vidljivo na slici 4.1.

Slika 4.1: Rezultat analize broja komentara po članku

4.3. Najaktivniji korisnici

Kako bi se dobila informacija o najaktivnijim korisnicima, potrebno je spojiti određeni komentar s korisničkim imenom te prebrojati koliko je komentara povezano s istim korisnikom. Da bi se došlo do ove informacije potrebno je opet obraditi podatke iz `comments` kolekcije koju je scraper stvorio. Kao i u prethodnoj analizi obrađuje se dio podataka koji sprema informacije o komentarima, a on je specifičan za svaku vrstu komentara pa je potrebno izraditi zasebne funkcije. Funkcije su prvo bile izrađene tako da same rade dohvat iz baze, no kasnije su prepravljene tako da primaju podatke iz baze u argumentu te ih zatim samo obrađuju. Funkcije vraćaju rezultate u obliku polja s objektima. Svaki objekt sadrži informacije o korisničkom imenu, URL-u članka, domeni, datumu objave i sadržaju. Rezultati se nakon svake obrade spremaju u bazu kako bi se omogućio prikaz rezultata i bez prethodne obrade. Nakon spremanja u bazu radi se dohvat iz nje koji grupira po korisničkom imenu te se tako dobije traženi rezultat, a upit za dohvat iz baze je prikazan u ispisu 4.3.

```
 query = [{"$group":
             {
             "_id": "$username",
             "count": {"$sum": 1}}
             },
         {"$sort": {"count": -1}},
         {"$limit": limit}]
doc = self.mongo_database.user_comments.aggregate(query)
```

Ispis 4.3: Upit za dohvat broja komentara po korisniku

Za Facebook vrstu komentara svi komentari se nalaze unutar polja `id_map`. Polje u sebi sadrži objekte tipa `user` i `comment`, a međusobno se mogu povezati preko `authorID` polja unutar komentara. Rezultati se spremaju u Python rječnik gdje `authorID` služi kao ključ. Prvi pronađeni objekt s određenim identifikatorom stvara zapis u rječniku, a ostali zapisi ga ažuriraju. Ako je zapis korisnik, onda ažurira postojeći zapis tako da mu dodaje korisničko ime. Ako je zapis komentar, ažurira se polje sa sadržajem komentara te se broj komentara uvećava za jedan. Dobiveni rječnik se na kraju pretvara u polje kakvo očekuje funkcija grupiranja rezultata.

Zapisi članaka za portal Večernji.hr informacije o komentarima sadrže unutar polja `comments`. `Comments` ne sadrži objekte kao u Facebook tipu komentara, već ima HTML kod stranice koji se nalazi u polju `page`. Uz kratku analizu strukture HTML-a primijećeno je da se potrebne informacije o sadržaju, korisničkom imenu i datumu objave nalaze unutar HTML elemenata sa specifičnim klasama. Za dohvat tih elemenata korištena je knjižnica `BeautifulSoup`. Ona omogućuje definiranje naziva elementa i klase na temelju koje se dohvate svi elementi iz HTML koda koji zadovoljavaju navedeni uvjet. Prvi element s korisničkom imenom će pripadati prvom elementu sa sadržajem komentara pa je stoga bilo bitno održavati indeks pojavljivanja koji se koristio kao ključ u rječniku. Kao i kod Facebook tipa komentara na kraju je bilo potrebno rječnik pretvoriti u polje kakvo se očekuje.

Tip komentara 24sata informacije o komentarima i korisnicima drži unutar dva odvojena polja, a zajednički podatak im je identifikator korisnika. Kako bi se podaci iz oba polja spojili, taj se identifikator koristi kao ključ unutar rječnika. Prvo se iterira po komentarima te se zapis u rječniku dodaje ili ažurira u ovisnosti je li već postoji ključ s identifikatorom korisnika. Zatim se iterira po polju korisnika i svaki se zapis ažurira s korisničkim imenom. Na kraju funkcije se kao i u prethodnim tipovima komentara radi pretvorba u polje.

Disqus tip komentara podatke o komentarima sprema unutar polja `comments` koje je oblika sličnog JSON-u. Da bi se došlo do traženih informacije bilo je potrebno trenutni oblik niza slova pretvoriti u JSON oblik unutar kojeg je moguće dohvaćati polja, no prvo su bile potrebne male izmjene. Da bi se podatak mogao parsirati u JSON oblik potrebno je zamijeniti jednostruke navodnike s dvostrukima, `None` pretvoriti u `null` i napraviti još nekoliko manjih preinaka. Nakon parsiranja je bilo potrebno samo povezati polje u objektu s pravilnim poljem unutar JSON-a jer su sve potrebne informacije bile unutar polja `comments`. Zbog toga što nije bilo potrebno nikakvo spajanje po identifikatoru ili nekom drugom polju, rezultat se mogao izravno spremati u polje koje se vraća iz funkcije.

Rezultat analize je jednostavni prikaz korisnika i broja komentara koji je sortiran od korisnika s najvećim brojem komentara.

4.4. Najagresivniji korisnici

Zadnje implementirana metoda analize bila je analiza najagresivnijih korisnika. Ova analiza obrađuje sadržaj komentara i svakom komentaru daje indeks agresivnosti. Najbolje prepoznavanje agresivnosti uključivalo bi neke od metoda strojnog učenja, no kako svrha ovog rada nije bila strojno učenje, odabrao se drugačiji pristup. Agresivnost pojedinog komentara se radila uz usporedbu riječi komentara s listom riječi iz rječnika Emolex. Emolex je rječnik koji za riječi daje indeks povezanosti s određenom emocijom. Emocije koje su dostupne su: ljutnja, strah, iščekivanje, povjerenje, iznenađenje, tuga, sreća i gađenje. Agresivnost je većinom povezana s ljutnjom pa će se iz rječnika iskoristiti samo dio koji povezuje riječi s emocijom ljutnje. Sve riječi vezane

uz ljutnju se izdvajaju u datoteku `Emolex.txt` te se ta datoteka smješta na poslužitelj unutar `conf` mape. Sadržaj datoteke prikazan je u ispisu 4.4. te se vidi kako riječi sa snažnom povezanošću s ljutnjom imaju veliki indeks. Taj indeks će se koristiti kao indikator agresivnosti te veća vrijednost označava veći broj agresivnih riječi.

```
word    Croatian-hr emotion emotion-intensity-score
outraged    pobjesnio   anger   0.964
brutality   brutalnost  anger   0.959
hateful mrzak   anger   0.940
terrorize   terorizirati    anger   0.939
infuriated  veoma ljut  anger   0.938
```

Ispis 4.4: Izgled `Emolex.txt` datoteke

Za provedbu analize potrebne su informacije o korisničkom imenu i sadržaju komentara, a te iste informacije već izdvaja analiza najaktivnijih korisnika. Ta analiza također sprema rezultate obrade u bazu pa se kolekcija `user_comments` iz baze može iskoristiti za provedbu analize. Ulazni podaci analize su stoga zapisi te kolekcije, a njihov oblik prikazan je u ispisu 4.5. Za dohvat se koristi agregacija po korisničkom imenu jer se agresivnost zaključuje na temelju svih komentara, a oni su u bazi spremljeni odvojeno.

```
{
   "_id": ObjectId("60c3a29e603ba8a50e70b0b1"),
   "username": "Vilim",
   "username_lower": "vilim",
   "datePosted": "2021-04-27 14:06:00",
   "domain": "vecernji",
   "url": "https://www.vecernji.hr/vijesti/naslov-neke-vijesti",
   "content": "Sadrzaj komentara"
}
```

Ispis 4.5: Oblik zapisa u kolekciji `user_comments`

Izračun indeksa agresivnosti jedinstvena je funkcija bez obzira na tip komentara jer radi na već objedinjenim rezultatima. Iako je potrebna samo jedna funkcija ona će biti dulja i mogla bi zatrpati sadržaj servisa za analize. Iz tog razloga se izrađuje nova klasa `EmolexDetector` koja će sadržavati logiku izračuna agresivnosti. Ta klasa implementira nadklasu `AggressiveLanguageDetector` tako da se lako može trenutni detektor zamijeniti s nekim novim. Jedini zahtjevi zamjene su da se nasljeđuje ista nadklasa te da se implementira funkcija `run_detection` koja vrši izračun.

Kod inicijalizacije Emolex detektora izvodi se obrada i spremanje riječi iz `Emolex.txt` u varijablu klase. Riječi su spremljene u rječnik gdje je riječ ključ, a vrijednost je indeks povezanosti riječi s emocijom ljutnje. Ovakvo spremanje omogućuje brže dohvaćanje indeksa jer nema iteracije po svim elementima nego se traži po ključu.

Funkcija `run_detection` radi tako da iterira po svim komentarima jednog korisnika te svaki komentar razlaže na riječi. Definirana je varijabla `anger_sum` na razini svakog korisnika te se u nju dodaje izračunata vrijednost agresivnosti svakog komentara. Funkcija iterira po riječima i za svaku riječ pregledava postoji li zapis u emolex rječniku s tom riječi. Prva implementacija je pregledavala samo identične riječi, a kasnije je implementacija izmijenjena i poboljšana. Za svaku riječ traži se postoji li u

rječniku zapis koji sadrži tu riječ unutar sebe. Time se dobije polje zapisa, a iz polja se onda uzima prvi zapis iz rječnika za koji vrijedi da je riječ iz komentara barem 70%. Ovim preinakama se pokušalo postići prepoznavanje istih korijena riječi, ali u isto vrijeme su se uveli novi problemi. Tako se na primjer riječ "biti" povezivala s riječi "ubiti" i rezultirala velikim indeksom agresivnosti iako je sasvim uobičajena riječ. Zbog toga je uvedeno pravilo da zapis iz rječnika mora počinjati s riječi iz komentara. Uzima se indeks prve pronađene riječi za koju su zadovoljena pravila duljine od 70% te započinjanja s traženom riječi. Taj se indeks dodaje ukupnom indeksu agresivnost komentara te se iteracija provodi sve dok se ne obrade sve riječi iz komentara. Nakon obrade se agresivnosti korisnika dodaje agresivnost komentara. Odvajanje indeksa za komentar i korisnika nije bilo potrebno u ovom slučaju, no implementacija je ovakva kako bi se lako mogla prenamijeniti za dohvat agresivnih komentara umjesto korisnika.

Rezultat analize je oblika prikazanog na ispisu 4.6. te se taj rezultat sprema u kolekciju `user_aggression` kako ne bi bilo potrebno svaki puta raditi analizu ispočetka. Pregled rezultata analize daje samo manji broj najagresivnijih korisnika te se prikazuju informacije o prosječnoj agresivnosti komentara i ukupnoj agresivnosti korisnika. Spremljeni rezultat omogućuje i druge oblike rezultata pa bi se na primjer s istim rezultatima mogao napraviti prikaz najagresivnijih komentara.

```
{
    "_id": ObjectId("60c3a2aa83cd2c19dc64df99"),
    "comments": [{
            "content": "Sadrzaj nekog jako agresivnog komentara",
            "comment_aggressiveness": 1.5
        }, {
            "content": "Sadrzaj neagresivnog komentara",
            "comment_aggressiveness": 0
        },
        {
            "content": "Sadrzaj agresivnog komentara",
            "comment_aggressiveness": 0.5
        }
    ],
    "username": "crazy mind",
    "count": 3,
    "user_aggressiveness": 2,
    "average_aggressiveness": 0.667
}
```

Ispis 4.6: Oblik zapisa u kolekciji `user_aggression`

5. Nadogradivost i mogućnosti poboljšanja

Ovo poglavlje obrađuje nadogradivost aplikacije te poboljšanja koja bi se mogla uvesti. Nadogradivost je opisana kroz nekoliko različitih slučajeva nadogradnje gdje se opisuju promjene koje su potrebne za uvođenje nove funkcionalnosti. Kao moguća poboljšanja obrađuju se nove mogućnosti filtriranja i ubrzanje analiza te se opisuje trenutno stanje sustava i kako bi promjene poboljšale aplikaciju.

5.1. Nadogradivost aplikacije

Aplikacija bi se mogla nadograditi na više načina. Tijekom razvoja aplikacije većinom se pazilo na to da se aplikacija može nadograditi i da taj postupak bude čim lakši. Većina refaktoriranja je služila ili lakšoj mogućnosti nadogradnje ili boljoj čitljivosti i razumljivosti koda čime se osigurava lakše održavanje.

Jedan od načina nadogradnje je dodavanje novih funkcionalnosti vezanih uz entitete koji već ne postoje u sustavu. Primjer takve nadogradnje bila bi uvođenje korisnika i prijave u aplikaciju. Na poslužiteljskoj strani su pojedini entiteti sasvim razdvojeni jedni od drugih stoga uvođenje novog entiteta ne bi zahtijevalo promjenu u postojećem kodu. Za obradu novog objekta potrebno bi bilo samo dodati servis koji implementira nove funkcionalnosti te zatim dodati view koji će te funkcije pozivati i pripadajuću urls datoteku. Jedina promjena bila bi u urls datoteci na razini aplikacije gdje je potrebno uključiti novo izrađenu urls datoteku da bi se njezini putevi mogli prepoznati. Unutar sučelja aplikacije također ne bi bilo potrebno raditi promjene u postojećem kodu. Za novu funkcionalnost može se definirati nova stranica i novi servis koji može pozivati poslužitelj. Jedine datoteke koje je potrebno promijeniti su `app.module.ts` gdje se moraju dodati nove komponente, to jest stranice i `app-routing.module.ts` gdje je potrebno deklarirati rute za nove stranice.

Drugi mogući način nadogradnje je dodavanje novih vrsta analiza. Isprva je bila izrađena funkcija unutar `analysis_service` koja vraća JSON polje sa svim analizama i njihovim podacima. U tom slučaju bi se za nadogradnju trebao dodati novi zapis u to JSON polje te je potrebno mijenjati postojeći kod funkcije. Za bolju nadogradivost se funkcija izmijenila tako da dohvaća zapise s istim podacima iz baze. U tom slučaju

za novu analizu je potrebno samo dodati novi zapis u bazi podataka te implementirati novu funkciju. Zapis u bazi definira URL koji je potrebno pozvati za izvršavanje analize pa je samo potrebno da nova funkcija bude dostupna na tom definiranom URL-u. Kako nova funkcija ne mora biti u `analysis_service` i dohvat podataka o analizama se radi iz baze, nije potrebna promjena u postojećem izvornom kodu.

Implementacija analize za najagresivnije korisnike je također izrađena tako da omogućuje nadogradnju i zamjenu trenutne identifikacije agresivnosti. Trenutno se koristi Emolex detector koji nasljeđuje `AggressiveLanguageDetector`, a takva struktura se koristila kako bi se omogućila lakša nadogradnja. Ako se Emolex detektor želi zamijeniti s drugom implementacijom potrebno je samo da nova implementacija nasljeđuje nadklasu `AggressiveLanguageDetector` te implementira njezinu metodu `run_detection`. Onda se lako umjesto `EmolexDetectora` instancira drugi detektor pri inicijalizaciji servisa za analize te se taj detektor dalje koristi na isti način kao i `EmolexDetector`.

5.2. Bolje mogućnosti filtriranja i pregleda

Aplikacija trenutno ima dva pregleda podatka, pregledi članaka i komentara, te oba pregleda sadržavaju samo nekoliko osnovnih mogućnosti sortiranja i filtriranja. Osim filtera i sortiranja bitan je bio i podatak o broju zapisa koji se dohvaća te mogućnost dohvata iz baze. Obje vrste pregleda prije prikaza podataka zahtijevaju određenu obradu koja u slučaju velikog broja podataka može biti spora. Ograničavanje broja zapisa koji se obrađuju zato može biti od velike pomoći kod velikih baza podataka. Pregled članaka ne nudi nikakav filter, već samo definiranje broja zapisa koji se dohvaćaju, sortiranje po datumu i najvećem broju komentara te mogućnost dohvata iz baze. Time se može dobiti neki osnovni pregled članaka, ali ne postoji mogućnost nikakvog detaljnijeg pretraživanja.

Bolje mogućnosti dohvata bi se mogle ostvariti uz filtere datuma objave i domene. Filter datuma bi morao biti ostvaren kroz dva polja koja definiraju raspon jer korisnik nikada neće znati točan datum i vrijeme nekog članka niti će po toj informaciji pretraživati za određeni članak. Filter po domeni bi omogućio korisniku da pretražuje samo članke s određenog portala koji ga zanima. Pregled komentara bi se također mogao poboljšati s istim filterima. Osim njih, dodatni zanimljivi filter bi mogao biti filter sadržaja koji bi korisnik koristio da pretraži za komentare s određenom riječi ili frazom. Primjer takvog pretraživanja koje bi korisniku bilo zanimljivo je pretraživanje riječi "korona" ili "stožer" da se pregledaju komentari i vide reakcije na tu temu.

5.3. Ubrzanje analiza

Sve implementirane analize uključuju obradu podataka koje je scraper stvorio, a neke analize uključuju i rezultate drugih analiza, to jest višestruke obrade početnih podataka. Ovisno o analizi potrebna je jedna ili više iteracija kroz podatke i međurezultate te takve obrade mogu biti vrlo spore, pogotovo nad većim skupovima podataka. Taj se problem pokušao umanjiti uz spremanje rezultata analize u bazu čime se omogućuje dohvat iz baze umjesto ponovne obrade istih podataka. Ipak, bazu je potrebno napuniti i korisnik mora čekati na prvu obradu, a kasnije tek može koristiti rezultate iz baze.

Bolji način korištenja dohvata iz baze bio bi uz korištenje zakazanih akcija koje bi se provodile u određenim intervalima i punile bazu. Akcije bi pozivale sve funkcije analize i punile pripadajuće kolekcije tako da pri pokretanju analize u aplikaciji je potrebno samo dohvatiti podatke iz baze. U ovisnosti o tome je li potrebno na početku rada aplikacije imati dostupne sve podatke iz scrapera, implementirala bi se obrada cijele baze prije početka rada aplikacije ili bi se baza postepeno punila podacima. Ovim poboljšanjem se riješio problem čekanja na rezultate analize.

6. Zaključak

Portali s vijestima koji korisnicima omogućuju komentiranje mogu dati uvid u razmišljanja i stavove javnosti, no proučavanje takvih podataka ručno bilo bi vrlo nepraktično. Kako bi se iz čistih podataka o člancima i komentarima moglo doći do korisnih informacija, prvo je potrebna obrada. U ovom radu predstavljena je aplikacija čija je svrha obrada podataka kroz nekoliko različitih vrsta analize te prikaz dobivenih rezultata. Sama aplikacija ne sakuplja podatke, već kao temelj služe podaci prikupljeni u scraper dijelu aplikacije koji ovaj rad ne obrađuje.

Cilj rada je bio razvoj aplikacije koja služi za upravljanje i provjeru ispravnog rada sustava za dohvat članaka i komentara te prikaz podataka i njihova analiza. Rad opisuje arhitekturu izrađenog sustava, tehnologije koje su se koristile te korake razvoja sustava. Tijekom implementacije korištene su neke od uobičajenih praksi razvoja poput refaktoriranja i razvijanja funkcionalnosti u zasebnim granama, stoga rad obrađuje i korištene prakse te objašnjava njihovu svrhu u razvoju aplikacije. Aplikacija sadrži četiri vrste analize podataka: broj članaka po portalu, prosječan broj komentara po članku na portalu, korisnici s najviše komentara te korisnici s najagresivnijim komentarima. Za svaku od analiza je objašnjena njezina implementacija, oblik ulaznih podataka te rezultat. Na kraju se opisuje nadogradivost aplikacije te se iznose neke mogućnosti nadogradnje i poboljšanja.

LITERATURA

[3] Aws: What is a relational database? *Pristup ostvaren: 13.6.2021.* URL https://aws.amazon.com/relational-database/.

[4] Microsoft docs: Non-relational data and nosql. *Pristup ostvaren: 13.6.2021.* URL https://docs.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data.

[2] Atlassian: Git feature branch workflow. *Pristup ostvaren: 19.6.2021.* URL https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow.

[7] Stack overflow: Developer survey results 2018. *Pristup ostvaren: 13.6.2021.* URL https://insights.stackoverflow.com/survey/2018/#work-version-control.

[6] Refactoring.com: Refactoring. *Pristup ostvaren: 17.6.2021.*, . URL https://refactoring.com/.

[5] Refactoring guru: Refactoring techniques. *Pristup ostvaren: 18.6.2021.*, . URL https://refactoring.guru/refactoring/techniques.

[1] Analytics india magazine: Nosql vs sql — which database type is better for big data applications. *Pristup ostvaren: 13.6.2021.* URL https://analyticsindiamag.com/nosql-vs-sql-database-type-better-big-data-applications/.

[8] Geeksforgeeks: Difference between mvc and mvt design patterns. *Pristup ostvaren: 13.6.2021.*, Online: 12.4.2020. URL https://www.geeksforgeeks.org/difference-between-mvc-and-mvt-design-patterns/.

[9] Mdn web docs: Django introduction. *Pristup ostvaren: 13.6.2021.*, Online: 14.5.2021. URL https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction.

[10] Theme junkie: What is figma? (and how to use figma for beginners). *Pristup ostvaren: 14.6.2021.*, Online: 17.9.2020. URL https://www.theme-junkie.com/what-is-figma/.

[11] Scotch: Mvc in an angular world. *Pristup ostvaren: 13.6.2021.*, Online: 25.2.2019. URL https://scotch.io/tutorials/mvc-in-an-angular-world.

[12] Kinsta: What is github? a beginner's introduction to github. *Pristup ostvaren: 13.6.2021.*, Online: 28.5.2021. URL https://kinsta.com/knowledgebase/what-is-github/.

[13] Grazitti interactive: 8 proven reasons you need angular for your next development project. *Pristup ostvaren: 13.6.2021.*, Online: 5.6.20218. URL https://www.grazitti.com/blog/8-proven-reasons-you-need-angular-for-your-next-development-project/.

[14] Cio wiki: Client server architecture. *Pristup ostvaren: 14.6.2021.*, Online: 6.2.2021. URL https://cio-wiki.org/wiki/Client_Server_Architecture.

[15] martinfowler.com: Featurebranch. *Pristup ostvaren: 19.6.2021.*, Online: 7.5.2020. URL https://martinfowler.com/bliki/FeatureBranch.html.

[16] Hnd: Istraživanje digital news report: 91 posto hrvatskih građana vijesti traži na internetu. *Pristup ostvaren: 20.6.2021.*, Online: 7.7.2017. URL https://www.hnd.hr/istrazivanje-digital-news-report-91-posto-hrvatskih-gradana-vijesti-trazi-na-internetu.

[17] Alan Crouch. Agile connection: Picking the right branch-merge strategy. *Pristup ostvaren: 19.6.2021.*, Online: 14.11.2018. URL https://www.agileconnection.com/article/picking-right-branch-merge-strategy.

[18] Martin Fowler. martinfowler.com: Patterns for managing source code branches. *Pristup ostvaren: 19.6.2021.*, Online: 28.5.2020. URL https://martinfowler.com/articles/branching-patterns.html.

Sustav za provođenje analize novinskih članaka i pridruženih komentara na društvenim mrežama

Sažetak

Rad obrađuje razvoj aplikacije za upravljanje dijelovima scrapera, provjeru njegovog rada te za prikaz i analizu podataka o člancima i komentarima. Opisana je odabrana arhitektura te korištene tehnologije i razlozi za njihov odabir. Rad daje pregled vremenskog tijeka razvoja, konkretne implementacije odabrane arhitekture, a zatim opisuje procese refaktoriranja i dodavanja novih funkcionalnosti u granama koji su se koristili u razvoju. Rad opisuje implementirane analize, pripadajuće ulazne podatke i rezultate, a na kraju se opisuje nadogradivost aplikacije i moguća poboljšanja.

Ključne riječi: komentari, članci, analiza

System for analysis of news on portals and associated comments on social network

Abstract

This thesis describes the development of an application for managing parts of scraper, verification of correct running and view and analysis of articles and comments. The thesis first introduces system requirements and the chosen arhitecture and then explains technologies which were used and explains the reasoning behind made choices. The paper covers the timeline of development and implementation of the chosen arhitecture. During application development refactoring and Feature Branch Workflow were used, and the thesis explains them in detail. Finally the paper covers implemented analysis techniques, their input data and results and also provides insight of system's upgradeability and possible future improvements.

Keywords: comments, articles, analysis