Razvoj društvene mreže "Mrvica"

Gradečak, Andrija

Sveučilište u Rijeci
Fakultet informatike i digitalnih tehnologija
Preddiplomski studij informatike

Andrija Gradečak

Razvoj društvene mreže „Mrvica“

Završni rad

Mentor: izv. prof. dr. sc. Sanja Čandrlić

Rijeka, rujan 2022.

FAKULTET INFORMATIKE I DIGITALNIH TEHNOLOGIJA
SVEUČILIŠTE U RIJECI

Rijeka, 1.6.2022.

Zadatak za završni rad

Pristupnik: Andrija Gradečak

Naziv završnog rada: Razvoj društvene mreže "Mrvica"

Naziv završnog rada na eng. jeziku: Development of a social network "Mrvica"

Sadržaj zadatka:

Zadatak studenta je osmisliti i izraditi aplikaciju – društvenu mrežu pod nazivom „Mrvica“. Ona bi okupljala korisnike koji imaju želju dijeliti i objavljivati svoje recepte i radove. Razvojno okruženje nije definirano i student ga može sam odabrati.

Mentor

Izv. prof. dr. sc. Sanja Čandrlić

Voditelj za završne radove

Doc. dr. sc. Miran Pobar

Zadatak preuzet: 2.6.2022.

Andrija Gradečak

(potpis pristupnika)

Sadržaj

Sažetak

Cilj ovog rada je razvitak društvene mreže u razvojnom okviru Django u svrhu povezivanja pekara i entuzijasta sa područja Hrvatske. Prije opisa koraka implementacije aplikacije detaljno je objašnjena struktura podataka aplikacije te postupak inicijalizacije Django projekta. Dio rada o samoj implementaciji uključuje detaljne korake razvoja nekih od stranica aplikacije, spajajući koncepte Django-a poput modela, obrazaca, pogleda i predložaka.

Ključne riječi: Django, društvena mreža, web aplikacija, paketi, kontrola verzija, modeli, obrasci, pogledi, predlošci

1. Uvod

Web aplikacije su aplikacije razvijene uz pomoć web tehnologija poput HTML-a, CSS-a i JavaScript-a koje se pokreću u web pregledniku. Ovoj vrsti aplikacija korisnici mogu pristupiti preko Interneta koristeći se web preglednicima poput Internet Explorera, Microsoft Edge-a, Firefox-a ili Chrome-a [1]. Web aplikacije najčešće se sastoje od prednjeg (engl. *frontend*) i pozadinskog (engl *backend*) dijela. Frontend web aplikacije vezan je uz korisničku interakciju i iskustvo. Na frontend dijelu web aplikacije dinamički se prikazuju podaci koje šalje backend dio. Ukratko, backend rukuje svim aspektima upravljanja podataka uključujući njihovo spremanje, organizaciju i dostavljanje. Drugim riječima, frontend je dio razvoja s klijentske strane dok je backend dio razvoja sa poslužiteljske strane [2].

Razvoju web aplikacija može se pristupiti na različite načine. U početku su se koristile već spomenute tehnologije poput HTML-a, CSS-a i JavaScript-a/PHP-a, dok se danas češće koriste razni razvojni okviri koji su na višoj razini apstrakcije [3]. Popularan smjer razvoja web aplikacija danas je korištenje jednog razvojnog okvira za frontend te drugog za backend dio aplikacije [4]. Jedan primjer takvog razvoja je takozvani „MERN“ (MongoDB, Express.js, React i Node.js) stog tehnologija. MongoDB služi kao NoSQL dokumentna baza podataka, u Express.js-u piše se serverski JavaScript kod koji Node.js zatim pokreće na serveru, a React.js služi za definiranje interaktivnog korisničkog sučelja [5].

Razvoju web aplikacije „Mrvica“ pristupiti će se na nešto drugačiji način – korištenjem razvojnog okvira Django. Django je besplatan Python razvojni okvir otvorenog koda koji omogućava razvoj web aplikacija od početka do kraja [6]. Django prati „MVT“ (engl. *model–view–template*) obrazac softverske arhitekture (engl. *software architectural pattern*) koji je varijacija MVC obrasca. Ukratko, u MVT obrascu modeli predstavljaju sučelje prema podacima, pogledi definiraju koji podaci će činiti kontekst, a predlošci definiraju kako će se taj kontekst prikazati korisniku [7].

U nastavku rada slijedi detaljno pojašnjenje načina rada Django razvojnog okvira, kratki osvrt na alternativne razvojne okvire, definicija strukture podataka te sam proces razvoja web aplikacije „Mrvica“.

2. Razvojni okvir Django

Ono što Django čini jednim od najpopularnijih razvojnih okvira za razvijače web aplikacija je veliki broj korisnih alata koji su ugrađeni u sami okvir [8]. Ovime je u Django-u omogućen razvoj web aplikacija gdje svi dijelovi aplikacije rade zajedno na unaprijed poznati način. Django se stoga može koristiti za razvoj raznih vrsta projekata, od društvenih mreža i wiki stranica do stranica za online kupovinu [9]. Razvoj u Django-u omogućava i kombiniranje sa raznim frontend razvojnim okvirima [10], a može se proširiti i velikim brojem paketa koji imaju specifične namjene npr. paketom „Pillow“ koji se u ovom projektu koristi za učitavanje i manipuliranje slika [11]. Unatoč tome što Django dolazi upakiran sa specifičnom bazom podataka ili serverom za pokretanje Python koda, razvijači u bilo kojem trenutku mogu donijeti odluku zamijeniti iste sa komponentama koje im više odgovaraju.

Velika popularnost Django-a zasigurno proizlazi iz činjenice da se u razvoju web aplikacija on koristi Python-om. Python je vrlo popularan programski jezik čije izvođenje podržavaju mnoge platforme, od Windows-a do Linux-a i MacOS-a. Za Python je na Internetu moguće naći veliki broj pomoćnih resursa, a oni će u većem dijelu biti korisni i za Django projekte. Isključivši Python, Django kao razvojni okvir iza sebe ima jednu od većih Internetskih zajednica [12].

Samu srž razvoja web aplikacije u Django-u čini MVT obrazac arhitekture softvera koji olakšava proces razvoja i omogućava ponovnu iskoristivost velikog broja komponenti. MVT je Djangov odgovor na MVC obrazac gdje se kod aplikacije razdvaja u nekoliko slojeva. U MVT obrascu modeli (engl. *models*) su sačinjeni od Python objekata koji definiraju strukturu podataka aplikacije, pružaju različite mehanizme za upravljanje istim podacima te olakšavaju postavljanje upita nad bazom podataka. Pogledi (engl. *view*) su Python funkcije koje rukuju HTTP zahtjevima i vraćaju HTTP odgovore, pristupajući potrebnim podacima kroz definirane modele. Pogledi pak delegiraju proces prikaza informacija predlošcima (engl. *templates*). Predlošci definiraju izgled samih stranica aplikacije te prikaz podataka koji se dinamički ubrizgavaju iz konteksta proslijeđenog pogledima [13].

Velika prednost korištenja Django-a kao razvojnog okvira za razvoj web aplikacija je i njegova fokusiranost na sigurnost. Django pruža veliki broj alata i načina kojima se aplikacija na jednostavan način može osigurati od napada sa strane vanjskih aktera. Dva primjera čestih napada od kojih Django štiti korištenjem jednostavnih alata su injekcija SQL-a (engl. *SQL injection*) i skriptiranje na više stranica (engl. *cross-site scripting, CSRF*) [14]. Nadalje, neke od sigurnosnih značajki u Django-u su već automatski omogućene, a jedan takav primjer je haširanje korisničkih zaporki široko korištenom kriptografskom funkcijom SHA256 [15].

3. Alternativni razvojni okviri

Django je jedan od mnogo razvojnih okvira koji omogućuju razvoj web aplikacija od početka do kraja. Neki primjeri Django-u sličnih okvira su Laravel (PHP), Ruby on Rails (Ruby), Symfony (PHP), Spring (Java) te ASP.NET (C#) [16]. U nastavku poglavlja će više riječi biti o Laravelu i Ruby on Rails.

3.1 Laravel

Laravel je besplatan PHP razvojni okvir otvorenog koda baziran na Symfony PHP okviru. Ovaj razvojni okvir koristi se za razvoj web aplikacija koje prate MVC obrazac arhitekture softvera. Poput Django-a, Laravel nudi veliki izbor ugrađenih alata koji olakšavaju razvoj, podizanje te održavanje web aplikacija. Nadalje, Laravel također omogućava izmjenu željenih komponenti poput npr. izmjene korištene baze podataka. Funkcionalnosti aplikacija razvijenih u Laravelu dodatno se mogu proširiti instalacijom raznih modula koje Laravel nudi „Composer“ upraviteljem paketa.

U nastavku slijedi nekoliko primjera ugrađenih funkcionalnosti kojima se Laravel ističe među konkurencijom te olakšava razvoj web aplikacija. Kod implementacije usmjeravanja, definirane rute aplikacije mogu se jednostavno ispisati koristeći ugrađeni CLI alat. Postavljanje upita nad bazom podataka u Laravelu također je intuitivno, a radi na način da se upit slaže ulančavanjem poziva željenih funkcija. Za stvaranje predložaka Laravel se služi „Blade Template Engine-om“, malim i jednostavnim jezikom koji se koristi hijerarhijom blokova kako bi definirao raspored elemenata i omogućio ubacivanje dinamičnog sadržaja. Autentikacija korisnika također se lako implementira u Laravelu, a nudi velik broj alata kojima se aplikacija osigurava od vanjskih napada. Kako bi se proces razvoja dodatno olakšao, Laravel automatski omogućava neke od tih zaštitnih mjera [17].

3.2 Ruby on Rails

Ruby on Rails je besplatan Ruby razvojni okvir otvorenog koda koji se, poput Laravela, služi MVC obrascem arhitekture softvera. Ruby on Rails je razvojni okvir koji također omogućava razvoj frontenda i backenda web aplikacija (engl. *full-stack*), ali se najčešće koristi za implementaciju serverskog dijela. Ruby on Rails potiče korištenje popularnih web standarda poput JSON-a, XML-a i web tehnologija poput HTML-a, CSS-a i JavaScripta za implementaciju korisničkog sučelja i interakcije. Ovaj razvojni okvir prvi je predstavio koncepte poput migracija i stvaranja baza podataka koristeći se kodom. Veliki broj danas popularnih razvojnih okvira za web aplikacije posudilo je ideje Ruby on Railsa. Među te razvojne okvire spadaju Django i Laravel, ali i neki drugi poput npr. Catalysta, Grailsa, Phoenixa, Sails.js-a itd.[18].

4. Potrebe aplikacije

Društvena mreža „Mrvica“ zamišljena je kao web aplikacija preko koje pekari i entuzijasti mogu podijeliti svoje recepte, upoznati druge korisnike te komentirati njihove radove. Kako bi se „Mrvica“ smatrala društvenom mrežom, ona mora imati elemente koji su zajednički većini drugih društvenih mreža poput npr. Instagrama, Twittera, Facebooka itd. U nastavku su opisani potrebni dijelovi aplikacije i njezine glavne funkcionalnosti.

Kao i svaka druga društvena mreža, „Mrvica“ mora imati mogućnost registracije novih i prijave postojećih računa. Registracijom korisnik mora biti u mogućnosti zadati svoje korisničko ime i postaviti željenu lozinku. Korisnik dodatno mora definirati i svoju e-mail adresu, a to će mu omogućiti resetiranje lozinke u slučaju gubitka ili zaborava iste. Za svakog korisnika veže se i jedinstven profil kojeg nakon registracije ispunjava dodatnim informacijama poput imenom, opisom i lokacijom. Kako bi korisnik koristio samu aplikaciju potrebno je omogućiti prijavu u sustav točno onim informacijama koje su upisane u procesu registracije. Korisnik se iz sustava mora moći i odjaviti, a proces odjave mora biti jednostavan npr. klikom gumba odjave koji je lako vidljiv i dostupan na svim stranicama aplikacije.

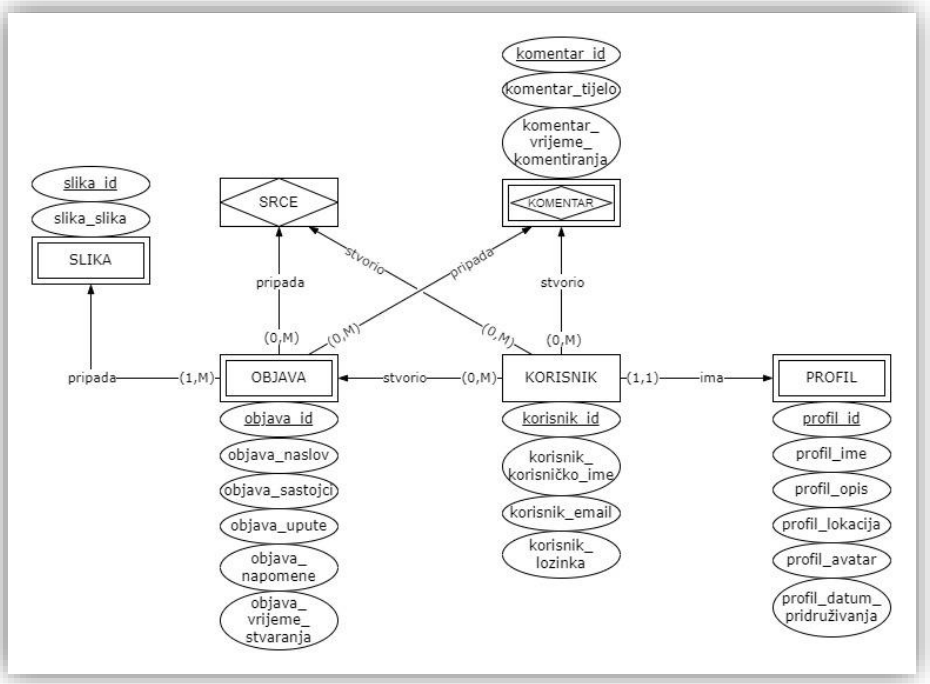
Objavljivanje sadržaja centralni je dio svake društvene mreže. U „Mrvici“ ovo konkretno znači mogućnost objavljivanja novih recepata. Korisniku kroz jednostavan obrazac mora biti omogućeno objavljivanje željenih recepata, a polja unosa moraju sadržavati informacije poput sastojaka, uputa, napomena i slika finalnog proizvoda. Korisniku također mora biti omogućeno otkrivanje i pregledavanje objava drugih korisnika mreže.

Još jedan bitan aspekt svake društvene mreže je i mogućnost interakcije sa drugim korisnicima. Kako bi se sadržaji objavljeni u „Mrvici“ obogatili interakcijom, korisnicima se moraju omogućiti funkcionalnosti poput komentiranja i „sviđanja“ objava. Svaki korisnik mora imati mogućnost komentiranja objava mreže, uključujući i svoje i tuđe objave. Svaki komentar mora prikazati minimalan broj informacija o korisniku koji ga je stvorio te omogućiti preusmjeravanje na profil klikom profilne slika ili korisničkog imena. Korisnicima mreže mora biti omogućeno i označavanje svih objava mreže srcima. Obrnuta akcija ili oduzimanje srca također mora biti omogućeno. Za kraj, korisnicima na određenim stranicama mora biti prikazan broj srca i komentara za sve objave stranice. Konkretno, u „Mrvici“ te informacije moraju biti prikazane na naslovnoj stranici, stranicama profila te stranicama samih objava.

Kako bi korisnik uopće stupio u interakciju sa drugim korisnicima i počeo otkrivati nove sadržaje, mora se omogućiti otkrivanje i „praćenje“ profila. Korisnik na naslovnoj stranici mora moći doći do svih korisničkih profila mreže. Klikom na te profile o istima može saznati više informacija ili ih „zapratiti“. Također, korisniku na vlastitom profilu moraju biti prikazane lista korisnika koje prati te lista korisnika koji prate njega. Kako se korisnika ne bi zasulo dugačkim listama profila mreže, treba mu omogućiti prikaz manjeg broja profila. Uz svaku od spomenutih lista treba se nalaziti i gumb koji, ukoliko korisnik to želi, preusmjerava na stranicu gdje se nalazi potpuna lista željenih profila. Na kraju, izvršavanje akcija „praćenja“ i „otpraćivanja“ profila mora biti jednostavno npr. klikom gumba koji se nalazi u bloku informacija određenog profila.

5. Struktura podataka

Za potrebe web aplikacije „Mrvica“ moraju se stvoriti slijedeće tablice: *Korisnik, Profil, Objava, Komentar, Srce* i *Slika.* Na slici ispod možemo vidjeti pripadni dijagram entiteta i veza.



Slika 1: Dijagram entiteta i veza

Za gore prikazani DEV definira se i relacijska shema baze podataka. Django ne stvara gore navede tablice istom logikom, ali o tome će više riječi biti kasnije. Relacijsku shemu svejedno je korisno izraditi kako bi dobili bolju ideju o strukturi tablica i vezama između njih.

KORISNIK (korisnik_id, korisnik_korisničko_ime, korisnik_email, korisnik_lozinka,
profil_id, profil_ime, profil_opis, profil_lokacija, profil_avatar, profil_datum_pridruzivanja)
OBJAVA (korisnik_id, objava_id, objava_naslov, objava_sastojci, objava_upute, objava_napomene, objava_vrijeme_stvaranja)
KOMENTAR (korisnik_id, objava_id, komentar_id, komentar_tijelo, komentar_vrijeme_stvaranja)
SRCE (korisnik_id, objava_id)
SLIKA (korisnik_id, objava_id, slika_id, slika_slika)

Slika 2: Relacijska shema

Tablica *Korisnik* glavna je tablica aplikacije te sprema podatke o računima registriranih korisnika. Definicija ove tablice nasljeđuje se iz Djangove *User* tablice, a ovime se postiže biranje samo onih atributa koji će se zaista koristiti u aplikaciji. Tablica *Korisnik* se sastoji od slijedećih atributa: *id*, *korisničko_ime*, *email* te *lozinka*. Ova

tablica/entitet je jakog tipa, što znači da zasebni korisnici mogu postojati samostalno u bazi podataka.

Tablica *Profil* sprema podatke o profilima registriranih korisnika. Ova tablica sastoji je od slijedećih atributa: *id, ime, opis, lokacija, avatar i datum_pridruživanja*. Ime označava ime i prezime korisnika npr. „Ivo Ivić“, opis označava dodatne informacije o profilu npr. „Pekar kruhova i krušnih proizvoda“, lokacija označava mjesto u kojem korisnik prebiva npr. „Rijeka“, avatar označava sliku profila korisnika, a datum pridruživanja točan datum kada je korisnik stvorio profil na mreži. *Profil* je slabi tip entiteta što znači da ne može postojati bez pripadajućeg korisnika tj. brisanjem korisničkog računa briše se i profil.

Tablica *Objava* sprema podatke o objavama koje su korisnici stvorili. Sastoji se od slijedećih atributa*: id, naslov, sastojci, upute, napomene i vrijeme_stvaranja*. Naslov označava ime recepta, sastojci označavaju sve potrebne sastojke recepta, upute označavaju korake potrebne za pripremanje recepta, napomene označavaju dodatne upute koje ukazuju na oprez kod pripreme, a vrijeme stvaranja u kojem momentu je recept objavljen. Vremenom stvaranja aplikacija se služi za sortiranje objava na stranicama. Nalik tablici *Profil, Objava* je slabi tip entiteta jer ni jedna objava ne može postojati bez pripadajućeg korisnika.

Tablica *Komentar* sprema podatke o komentarima koje su korisnici stvorili na određenim objavama. Sastoji se od slijedećih atributa: *id, tijelo* i *vrijeme_komentiranja*. Tijelo označava sam tekstualni sadržaj komentara, a vrijeme komentiranja trenutak u kojem je komentar stvoren. Vremenom komentiranja aplikacija se služi za sortiranje komentara na stranici objave. *Komentar* je slaba agregacija te ne može postojati bez pripadajućeg korisnika i objave tj. jedan korisnik na određenoj objavi može ostaviti više od jednog komentara.

Tablica *Srce* sprema srca kojima su korisnici označili objave koje im se sviđaju. Srca u objavama samo se broje zbog čega nisu potrebni drugi atributi. *Srce* je agregacija te ne može postojati bez pripadajućeg korisnika i objave tj. jedan korisnik na određenoj objavi može ostaviti samo jedno srce.

Na kraju, tablica *Slika* sprema podatke o slikama koje su dio galerije neke objave. Sastoji se od atributa *id* i *slika*, gdje slika označava same bajtove slike. *Slika* je slabi tip entiteta te ne može postojati bez pripadajuće objave.

6. Faze razvoja

Vodopadni model jedna je od najranije osmišljenih metodologija razvoja softvera. Ovu metodologiju razvoja lako je shvatiti i koristiti, a to je zato što se cijeli proces razvoja softvera dijeli na nekoliko odvojenih faza. Završetak i rezultati jedna faze služe kao ulaz u drugu fazu, a kroz faze se prolazi sekvencijalno. Vodopadni model sastoji se od slijedećih faza – analize zahtjeva, dizajna sustava, implementacije, testiranja, uporabe sustava te održavanja [19].

Faza analize zahtjeva prva je faza modela, a oslanja se na ideji da se svi zahtjevi projekta mogu prikupiti i razlučiti unaprijed. Za veći dio ove faze odgovoran je voditelj projekta čiji je posao detaljno proučiti i opisati zahtjeve projekta.

Druga faza je dizajn sustava čiji je cilj kreirati tehničko rješenje na probleme i zahtjeve opisane u fazi analize. Dizajnira se logika sustava, opisujući njegovu svrhu i opseg, a zatim i fizički dizajn koji uključuje specifikacije potrebnog hardvera i tehnologija.

Treća faza vodopadnog modela je implementacija. U ovoj fazi odvija se tehnološka implementacija dizajniranog sustava. Ova faza je često najkraća zbog toga što se najviše vremena i resursa izgubi na fazama analize i dizajna.

Testiranje je četvrta faza vodopadnog modela. Razvijeni softver testira se u svrhu identifikacije pogrešaka ili neispunjenja zahtjeva. Glavni cilj ove faze je utvrditi udovoljava li softversko rješenje korisnikovim zahtjevima.

Zadnje i najvažnije faze vodopadnog modela su uporaba i održavanje softvera. Ove dvije faze često se odvijaju paralelno jer korisnici korištenjem softvera nailaze na nove greške ili nedostatke sustava. Za održavanje se unutar razvojnog tima često formira novi tim čija će odgovornost biti pružati podršku korisnicima i implementirati potrebne promjene u softveru [20].

Kod razvoja društvene mreže „Mrvica“, u fokusu su bile dvije od šest spomenutih faza - dizajn i implementacija aplikacije.

Pod dizajnom aplikacije „Mrvica“ podrazumijeva se promišljanje i stvaranje strukture podataka tj. dijagrama entiteta i veza te relacijske sheme. U dizajn spada i promišljanje o modelima Django aplikacije jer se oni procesom migracije prevode u fizičke tablice baze podataka. U ovoj fazi nije bilo aktivnosti poput dizajna sučelja i interakcije jer se inspiracija vukla iz već postojećih web aplikacija.

Druga faza vodopadnog modela na kojoj je bio fokus u aplikaciji „Mrvica“ je implementacija. Kod razvoja „Mrvice“ ova faza trajala je najduže s obzirom da je u nekoliko navrata bilo potrebno napraviti promjene modela ili predložaka. U ovoj fazi se uzela struktura podataka te krenulo sa prevođenjem u modele aplikacije. Zatim je slijedio razvoj stranica i potrebnih funkcionalnosti. U ovom fazi spajaju se već spomenuti koncepti - modeli, obrasci, pogledi i predlošci. Ovdje su se radile promjene modela ako isti nisu odgovarali novo implementiranim funkcionalnostima. Druge promjene ticale pak su se ticale izgleda same aplikacije, a provodile su se izmjenom strukture *.html* datoteka tj. predložaka.

7. Razvoj

U ovom poglavlju detaljno su pojašnjene faze razvoja Django projekta. Razvoj aplikacije „Mrvica“ slijedi korake preporučene na stranicama Django dokumentacije. Neki od tih koraka uključuju inicijalnu postavu projekta, postavljanje kontrole verzija, definiranje ruta, stvaranje modela itd. [21].

7.1 Inicijalna postava projekta

Prije samog razvoja aplikacije potrebno je napraviti inicijalnu postavu projekta. Inicijalna postava uključuje nekoliko koraka, a prvi korak je instalacija Django-a i pomoćnih paketa. Potom slijedi inicijalizacija direktorija projekta uz pomoć Django CLI naredbi. Za projekt svakako treba postaviti i kontrolu verzija, a to uključuje korake poput stvaranja udaljenog repozitorija na stranici GitHub, inicijalizaciju Gita u direktoriju projekta te podizanje početne verzije projekta na udaljeni repozitorij. Na kraju slijedi i podešavanje postavki Django projekta kako bi one odgovarale zahtjevima aplikacije.

7.1.1 Instalacija Django-a i pomoćnih paketa

Jedini preduvjet instalacije je postojanje Python paketa na računalu. Django instalaciju radimo naredbom [22]:

```
python -m pip install Django
```

Zatim treba instalirati sve pomoćne pakete, a u slučaju aplikacije „Mrvica“ to je samo paket *Pillow* koji implementira sučelje za učitavanje i manipulaciju slikama:

```
python -m pip install Pillow --user
```

7.1.2 Inicijalizacija i pokretanje projekta

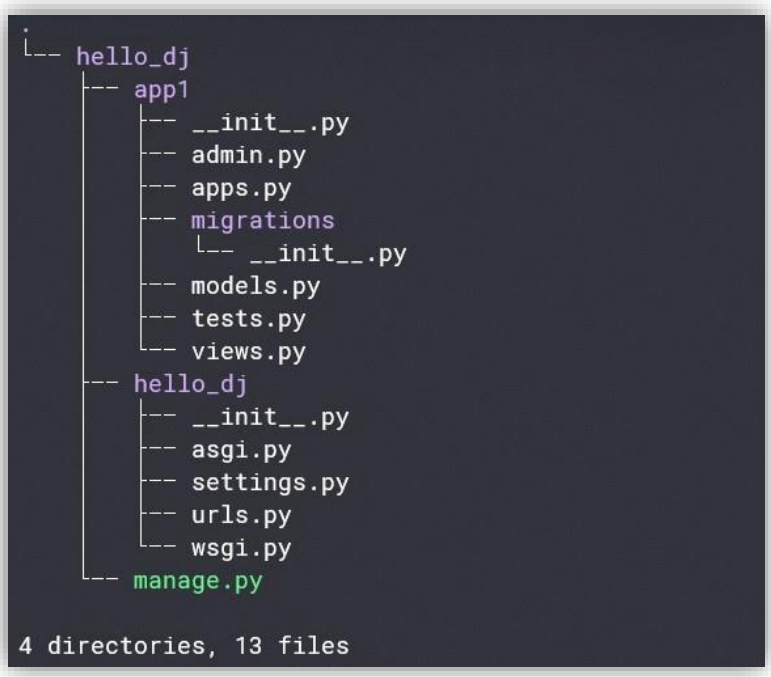
Slijedeći korak je inicijalizacija direktorija projekta. Stvaramo prazan lokalni direktorij unutar kojega će se nalaziti sve datoteke projekta. Zatim pokrećemo Django CLI naredbu koja taj direktorij inicijalizira potrebnim projektnim datotekama:

```
django-admin startproject mrvica
```

Projektni direktorij sada je inicijaliziran, ali treba inicijalizirati i direktorij same aplikacije. To činimo pokretanjem slijedeće Django CLI naredbe:

```
django-admin startapp app
```

Sada imamo projektno stablo koje izgleda ovako:



Slika 3: Početno projektno stablo Django projekta

Pokretanje aplikacije iz korijenskog direktorija projekta zatim izvršavamo sljedećom naredbom [23]:

```
python manage.py runserver
```

7.1.3 Kontrola verzija pomoću Gita i GitHuba

Radimo Git inicijalizaciju koja u glavnom direktoriju projekta stvara okruženje za kontrolu verzija:

```
git init
```

Zatim dodajemo novu *remote* lokaciju koja projekt spaja sa udaljenim repozitorijem.

```
git remote add origin https://github.com/agradecak/mrvica.git
```

Sljedećim skupom Git naredbi sve datoteke direktorija dodajemo u zonu za priređivanje (engl. *staging area*), nove datoteke i promjene „posvećujemo“ (engl. *commit*) u lokalnu sliku (engl. *snapshot*) i na kraju sve promjene „guramo“ (engl. *push*) na udaljeni repozitorij (engl. *remote repository*). Isti proces ponavljamo nakon završetka implementacije svake nove funkcionalnosti [24]:

```
git add --all
git commit -m „Inicijalno podizanje projekta.“
```

```
git push origin main
```

7.1.4 Podešavanje settings.py datoteke

Kako bi postava projekta odgovarala zahtjevima aplikacije, potrebno je napraviti neke promjene u konfiguracijskoj datoteci *settings.py*. Aplikaciju koju smo generirali u prijašnjim koracima treba instalirati na projektnoj razini dodajući njen naziv u listu *INSTALLED_APPS*:

```
INSTALLED_APPS = [
    ...
    'app'
]
```

Definiramo *LANGUAGE_CODE* kojim se Django služi da bi ispisivao poruke sustava u odabranom jeziku, a ovom slučaju to je hrvatski [25]:

```
LANGUAGE_CODE = 'hr-hr'
```

U projektima razvoja web aplikacija često se koriste statične datoteke poput CSS datoteka, fontova, ikona itd. Kod podizanja aplikacije, statične datoteke otpremaju se zajedno sa datotekama koje sadrže izvorni kod aplikacije. Definiranjem lokacije *STATIC* Django-u govorimo sa koje lokacije će se statične datoteke učitavati:

```
STATIC_URL = '/static/'
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, 'static')
]
```

Datoteke koje učitavaju korisnici, npr. slike, ne otpremaju se zajedno sa drugim datotekama aplikacije. Kako bi se spriječio nastanak kolizija između imena korisnički učitanih i statičnih datoteka, korisnički učitane datoteke često se spremaju na drugu lokaciju. Ova lokacija u Django-u zove se *MEDIA*. Kako se za spremanje slika u ovoj aplikaciji ne služimo vanjskim servisom već lokalnim diskom, *MEDIA* lokaciju možemo definirati unutar spomenutog *STATIC* direktorija [26]:

```
MEDIA_URL = '/images/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'static/images')
```

U aplikaciji ćemo se za prijavu, odjavu i registraciju novih računa koristiti Django-vim ugrađenim sustavom za autentikaciju. Prednost toga je što na jednostavan i deklarativan način možemo specificirati rute na koje će se korisnici preusmjeriti u slučaju prijave ili odjave s aplikacije [27]:

```
LOGIN_URL = '/'
LOGIN_REDIRECT_URL = '/naslovna/'
LOGOUT_REDIRECT_URL = '/'
```

Za kraj, postojećim korisnicima aplikacije omogućeno je resetiranje lozinke slanjem e-pošte koja sadrži poveznicu na obrazac za promjenu lozinke. Kako se za pokretanje i razvoj aplikacije služimo lokalnim serverom, Django-u moramo dati do znanja da tu e-poštu šalje na konzolu lokalnog servera, a ne na vanjski servis za e-poštu [28]:

```
EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'
```

7.1.5 Definiranje ruta

Django dolazi sa ugrađenim sustavom za definiranje ruta aplikacije. U projektnom direktoriju već se nalazi *urls.py* datoteka, a sadrži definiciju rute za ugrađenu Django admin stranicu:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
]
```

Kako bi definirali rute za određenu aplikaciju, potrebno je u varijablu *urlpatterns* projektne *urls.py* datoteke uključiti lokaciju aplikacijske *urls.py* datoteke:

```
    path('', include('app.urls')),
```

Zatim stvaramo samu aplikacijsku *urls*.py datoteku u direktoriju aplikacije. Uključujemo potrebne module poput npr. *path, settings i static* te definiramo imenski prostor (engl. *namespace*) aplikacije varijablom *app_name*. Stvaramo novu praznu listu *urlpatterns*, a na njen kraj dodajemo i prethodno definiranu lokaciju *MEDIA* kako bi se korisnički učitanim slikama moglo pristupiti preko URL-a:

```
from django.urls import path
from django.conf import settings
from django.conf.urls.static import static
from .views import *

app_name = 'app'

urlpatterns = []
urlpatterns += static(settings.MEDIA_URL,
document_root=settings.MEDIA_ROOT)
```

Ovime je završena osnovna postava usmjeravanja (engl. *routing*) aplikacije, a nove rute sada možemo definirati dodavanjem novih putanja (engl. *path*) u listu *urlpatterns*. Za svaku putanju te liste odabiremo naziv rute, pogled koji će se izvršiti te simbolično ime rute. Na primjer, rutu naslovnice definiramo na slijedeći način [29]:

```
    path('naslovna/', naslovna, name='naslovna'),
```

7.2 Modeli

Modelima se u Django-u definira struktura podataka tj. shema baze podataka i struktura njezinih tablica. Kako bismo definirali modele, u direktoriju aplikacije potrebno je stvoriti datoteku *models.py*. Modeli su obične Python klase koje nadjačavaju Djangovu ugrađenu klasu *models.Model*, a svaki model najčešće se preslikava u točno jednu tablicu baze. Modeli se sastoje od atributa tj. varijabli određenih tipova koje se preslikavaju u stupce tablica. U Django-u se modeli pretvaraju u finalne tablice procesom migracije. Pri izvođenju inicijalne migracije, Django za svaku tablicu automatski kreira primarni ključ *id*. Ovisno o vezama koje su definirane između modela, Django može stvoriti dodatne i posredničke tablice. Na primjer, za spremanje veze „praćenja“ među profilima stvara se dodatna tablica *profil_prati* [30].

7.2.1 Postupak konstrukcije modela na primjeru modela Profil

Na primjeru modela *Profil* biti će objašnjeni neki osnovni koncepti konstrukcije modela.

Kod modela *Profil*:

```
class Profil(models.Model):
    korisnik = models.OneToOneField(User, on_delete=models.CASCADE)
    ime = models.CharField(max_length=50)
    opis = models.CharField(max_length=255)
    lokacija = models.CharField(max_length=50)
    avatar = models.ImageField(default='profile.jpg')
    datum_pridruzivanja = models.DateField(auto_now_add=True)
    prati = models.ManyToManyField('self',
related_name='pracen_od', symmetrical=False, blank=True)

    class Meta:
        verbose_name_plural = 'Profili'

    def __str__(self):
        return "{}({})".format(self.korisnik.username, self.id)
```

U prvom bloku, ispod definicije imena klase, možemo vidjeti kako se definiraju atributi modela tj. budući stupci tablica. Svaki atribut klase je varijabla, a tip atributa definira se nadjačavanjem ugrađenih Django klasa npr. lokacija je skup znakova pa nadjačava *models.CharField*. Tipovi se dalje mogu prilagoditi dodavanjem željenih parametara npr. broj znakova želimo ograničiti na 50:

```
    lokacija = models.CharField(max_length=50)
```

Atributi ne moraju biti samo tekstualnog tipa. Na primjer, atribut 'avatar' omogućava unos slike, sa definiranom zadanom slikom u slučaju da korisnik nije učitao svoju.

```
    avatar = models.ImageField(default='profile.jpg')
```

Tablica *Profil* je slabog tipa što znači da ukoliko obrišemo redak tablice *User*, na koju se vanjski ključ određenog profila referira, briše se i ovaj redak tablice *Profil*. Za označavanje modela slabim, u tijelo funkcije atributa koji označava vanjski ključ prosljeđujemo parametar *on_delete=models.CASCADE*:

```
    korisnik = models.OneToOneField(User, on_delete=models.CASCADE)
```

Jedan profil može pratiti niti jedan ili više drugih profila, a isti profil može biti praćen od niti jednog ili više profila. Za definiranje takve M:M povratne veze za atribut modela *prati* koristimo Djangov tip *models.ManyToManyField*. Kako bi se olakšao pristup svim profilima koji prate ovaj profil, definira se i reverzibilno ime parametrom *related_name='pracen_od'*. Nadalje, profili ne „zapraćuju“ jedan drugoga automatski tj. jedan profil može pratiti drugi, ali drugi ne mora pratiti prvog. Takvu asimetriju definiramo parametrom *symmetrical=False*. Na kraju, profil ne mora pratiti niti jedan drugi profil, a takvo ponašanje definiramo parametrom *blank=True*:

```
    prati = models.ManyToManyField('self',
related_name='pracen_od', symmetrical=False, blank=True)
```

Definiranjem *Meta* klase unutar klase *Profil* Django-u dajemo do znanja da ćemo neke meta podatke, npr. ime klase u množini, nadjačati svojim vrijednostima. Na ugrađenoj administracijskoj stranici ime klase u množini više neće biti „Profils“ nego „Profili“*:*

```
    class Meta:
        verbose_name_plural = 'Profili'
```

Implementiramo i *__str__* funkciju koja za svaki redak tablice *Profil* na administracijskoj stranici (engl. *admin panel*) ispisuje kratki opis u svrhu identifikacije samog retka. Na administracijskoj ploči aplikacije moguće je pregledavati i rukovati podacima aplikacije u obliku jednostavnog, vizualnog sučelja.

```
    def __str__(self):
        return "{}({})".format(self.korisnik.username, self.id)
```

Na kraju, profil korisnika želimo stvoriti automatski, nakon završetka procesa registracije novog računa. Takvu funkcionalnost postižemo definiranjem funkcije *stvori_profil()* na kraju datoteke *models.py*. Ova funkcija je „prijamnik“ tj. hvata *post_save* signal koji svaki *User* šalje nakon završetka procesa registracije. Funkcija za novo registriranog korisnika stvara novi profil na način da proslijedi *id* korisnika u vanjski ključ *stvorio* novo stvorenog profila:

```
@receiver(post_save, sender=User)
def stvori_profil(sender, instance, created, **kwargs):
    if created and not instance.is_superuser:
        korisnik_profil = Profil(korisnik=instance)
        korisnik_profil.save()
```

7.2.2 Objava

Kod modela *Objava*:

```
class Objava(models.Model):
    naslov = models.CharField(max_length=100)
    sastojci = models.TextField()
    upute = models.TextField()
    napomene = models.TextField()
    stvorio = models.ForeignKey(User, related_name='objave',
on_delete=models.CASCADE)
    objava_srca = models.ManyToManyField(User,
related_name='srce_korisnik', blank=True, through=Srce)
    vrijeme_stvaranja = models.DateTimeField(auto_now_add=True)

    class Meta:
        verbose_name_plural = 'Objave'
        ordering = ('-vrijeme_stvaranja',)

    def __str__(self):
        return "{} {} {} {}".format(self.id, self.stvorio.username,
self.vrijeme_stvaranja, self.naslov[0:20])

    def sastojci_as_list(self):
        return self.sastojci.split('\n')

    def upute_as_list(self):
        return self.upute.split('\n')

    def prva_slika(self):
        return self.objava_slike.first()
```

Osim atributa tekstualnog tipa koji imaju očite svrhe, *Objava* se sastoji i od vanjskog ključa *stvorio*, povezujući objavu s korisnikom koji ju je stvorio:

```
class Objava(models.Model):
    naslov = models.CharField(max_length=100)
    sastojci = models.TextField()
    upute = models.TextField()
    napomene = models.TextField()
    stvorio = models.ForeignKey(User, related_name='objave',
on_delete=models.CASCADE)
```

Jedna objava može biti označena srcima više korisnika, a jedan korisnik može više objava označiti srcima. Takvu M:M vezu kojom posreduje tablica *Srce* definiramo parametrom *through=Srce*:

```
    objava_srca = models.ManyToManyField(User,
related_name='srce_korisnik', blank=True, through=Srce)
```

Objava ima i atribut *vrijeme_stvaranja*, a tip datuma definiramo nadjačavajući *model.DateTimeField*. Dodavanjem parametra *auto_add_now=True* Django-u naznačujemo da taj atribut treba automatski popuniti u trenutku stvaranja:

```
    vrijeme_stvaranja = models.DateTimeField(auto_now_add=True)
```

U *Meta* klasi dodatno definiramo sortiranje objava na temelju vremena stvaranja tj. najnovije objave biti će na vrhu stranice, dok će starije biti na dnu:

```
    class Meta:
        verbose_name_plural = 'Objave'
        ordering = ('-vrijeme_stvaranja',)
```

Za model *Objava* definiramo i tri nove funkcije: *sastojci_as_list*, *upute_as_list* i *prva_slika*. Definiranjem funkcije u nekom modelu možemo izvesti nove informacije iz postojećih podataka.

Na primjeru modela *Objava*, atribute *sastojci* i *upute* na stranici želimo ispisati u obliku listi. U obje funkcije *string* režemo kod znaka novog reda, vraćajući podatke u formatu liste:

```
    def sastojci_as_list(self):
        return self.sastojci.split('\n')

    def upute_as_list(self):
        return self.upute.split('\n')
```

Na kraju definiramo i funkciju *prva_slika* kojom se služimo za prikaz naslovne slike neke objave tj. recepta. Ova funkcija vraća prvu sliku iz liste svih slika koje su povezane sa određenom objavom:

```
    def prva_slika(self):
        return self.objava_slike.first()
```

7.2.3 Komentar

Kod modela *Komentar*:

```
class Komentar(models.Model):
    objava = models.ForeignKey('Objava', related_name='komentari',
on_delete=models.CASCADE)
    tijelo = models.CharField(max_length=255, blank=False)
    stvorio = models.ForeignKey(User, related_name='komentari',
on_delete=models.CASCADE)
    vrijeme_komentiranja = models.DateTimeField(auto_now_add=True)

    class Meta:
        verbose_name_plural = 'Komentari'
        ordering = ('-vrijeme_komentiranja',)

    def __str__(self):
        return "{} {} {}".format(self.id, self.stvorio.username,
self.tijelo[0:20])
```

Komentar se sastoji od tekstualnog tijela te vanjskih ključeva na tablice *Objava* i *User*.

Poput vremena stvaranja objave, u *Meta* klasi definiramo sortiranje komentara na temelju trenutka njihovog stvaranja. Stariji komentari biti će na dnu bloka komentara objave, a novi će biti na vrhu.

7.2.4 Srce

Kod modela *Srce*:

```
class Srce(models.Model):
    objava = models.ForeignKey('Objava', related_name='srca',
on_delete=models.CASCADE)
    stvorio = models.ForeignKey(User, related_name='srca',
on_delete=models.CASCADE)

    class Meta:
        verbose_name_plural = 'Srca'

    def __str__(self):
        return "{} {}({})".format(self.id,
self.objava.naslov[0:20], self.objava.id)
```

Srce osim vanjskih ključeva na tablice *Objava* i *User* nema niti jedan drugi atribut. Drugi atributi nisu potrebni jer se u aplikaciji srcima služimo jedino za brojanje tj. za svaku objavu želimo doznati broj srca koje ima.

7.2.5 Slika

Kod modela *Slika*:

```
class Slika(models.Model):
    objava = models.ForeignKey(Objava,
related_name='objava_slike', on_delete=models.CASCADE)
    slika = models.ImageField()

    class Meta:
        verbose_name_plural = "Slike"

    def __str__(self):
        return "{} {}({})".format(self.id,
self.objava.naslov[0:20], self.objava.id)
```

Model *Slika* odnosi se na slike koje korisnik učitava za određenu objavu. Osim vanjskog ključa na tablicu *Objava*, *Slika* se sastoji i od atributa *slika* koji je tipa *models.*ImageField. Ovaj atribut služi za spremanje samih bajtova učitane slike.

7.3 Primjer složenog algoritma

Funkcija *save* je ugrađena Django funkcija, a dio je svakog modela koji nasljeđuje iz *models.Model*. Njezinim nadjačavanjem možemo manipulirati podacima prije nego što se oni spreme u bazu podataka [31]. Svrha nadjačavanja slijedeće *save* funkcije modela *Slika* je promjena veličine i rezanje svake korisnički učitane slike prije njezinog spremanja na disk. Istu *save* funkciju, ali sa drugim zadanim dimenzijama ima i atribut *avatar* modela *Profil*.

Kod nadjačane funkcije *save*:

```
    def save(self, *args, **kwargs):
        slika_temp = Img.open(io.BytesIO(self.slika.read()))
        sirina, visina = slika_temp.size
        nova_sirina, nova_visina = 480, 480

        if slika_temp.mode != 'RGB':
            slika_temp = slika_temp.convert('RGB')

        if sirina > visina:
            x1 = (sirina - visina) / 2
            y1 = 0
            x2 = (sirina + visina) / 2
            y2 = visina
            slika_temp = slika_temp.crop((x1, y1, x2, y2))

        elif visina > sirina:
            x1 = 0
```

```
            y1 = (visina - sirina) / 2
            x2 = sirina
            y2 = (visina + sirina) / 2
            slika_temp = slika_temp.crop((x1, y1, x2, y2))

        if sirina > nova_sirina and visina > nova_visina:
            slika_temp.thumbnail((nova_sirina, nova_visina),
Img.ANTIALIAS)

        output = io.BytesIO()
        slika_temp.save(output, format='JPEG')
        output.seek(0)
        self.slika = InMemoryUploadedFile(output,'ImageField',
"%s.jpg" %self.slika.name.split('.')[0], 'image/jpeg',"Content-
Type: charset=utf-8", None)
        super(Slika, self).save(*args, **kwargs)
```

Slika se otvara, a njena kopija se potom sprema u varijablu *slika_temp*. Sve ostale operacije izvode se na toj kopiji slike. Iz podataka otvorene slike izdvaja se njezina veličina i sprema u dvije varijable – *sirina* i *visina*. Definiraju se i željene dimenzije slike – *nova_sirina* i *nova_visina*.

```
def save(self, *args, **kwargs):
    slika_temp = Img.open(io.BytesIO(self.slika.read()))
    sirina, visina = slika_temp.size
    nova_sirina, nova_visina = 480, 480
```

Prvo se provjerava je li model boja slike (engl. *color mode*) jednak modelu RGB. Ukoliko nije, model boja slike pretvara se u RGB.

```
    if slika_temp.mode != 'RGB':
        slika_temp = slika_temp.convert('RGB')
```

Slijedeća dva blok kondicionala sadrže logiku za rezanje (engl. *crop*) slike. Cilj je odrediti i izrezati središnji dio slike ukoliko su dimenzije slike (visina i širina) različite. Kod rezanja se služimo koordinatama dviju točki – gornjeg lijevog i donjeg desnog kuta slike.

Ako je širina slike veća od visine (slika je široka), gornja i donja strana slike se ne diraju tj. y koordinate ostavljamo istima. Za dobivanje kvadrata potrebno je rezati samo lijevu i desnu stranu tj. x obje koordinate pomaknuti za jednake vrijednosti s obje strane.

```
    if sirina > visina:
        x1 = (sirina - visina) / 2
        y1 = 0
        x2 = (sirina + visina) / 2
        y2 = visina
        slika_temp = slika_temp.crop((x1, y1, x2, y2))
```

Ako je pak visina slike veća od širine (slika je visoka), lijeva i desna strana slike se ne diraju tj. x koordinate ostavljamo istima. Za dobivanje kvadrata potrebno je rezati samo gornju i donju stranu tj. y obje koordinate pomaknuti za jednake vrijednosti s obje strane.

```
    elif visina > sirina:
        x1 = 0
        y1 = (visina - sirina) / 2
        x2 = sirina
        y2 = (visina + sirina) / 2
        slika_temp = slika_temp.crop((x1, y1, x2, y2))
```

Promjenu veličine slike, u slučaju da su dimenzije veće od željenih, izvršavamo pozivanjem *thumbnail* metode nad slikom. Istovremeno primjenjujemo i zaglađivanje slike (engl. *anti-aliasing*):

```
    if sirina > nova_sirina and visina > nova_visina:
        slika_temp.thumbnail((nova_sirina, nova_visina),
Img.ANTIALIAS)
```

Na kraju, bajtove procesirane slike izvodimo u sliku *.jpeg* formata te spremamo u varijablu *output*. Spremanje same slike u bazu izvršavamo „izvlačenjem“ procesirane slike iz radne memorije te pozivanjem metode *super()*.

```
    output = io.BytesIO()
    slika_temp.save(output, format='JPEG')
    output.seek(0)
    self.slika = InMemoryUploadedFile(output,'ImageField', "%s.jpg"
%self.slika.name.split('.')[0], 'image/jpeg',"Content-Type:
charset=utf-8", None)
    super(Slika, self).save(*args, **kwargs)
```

7.4 Obrasci

Obrasci (engl. *forms*) korisniku omogućavaju unos podataka za objave, komentare, svoj profil itd. U Django-u se obrasci definiraju u datoteci *forms.py* koju treba prethodno stvoriti u direktoriju aplikacije. Obrasci su obične Python klase koje nadjačavaju ugrađenu Django klasu *forms.ModelForm*. Nalik pisanju modela, svako polje obrasca je varijabla kojoj se tip definira nadjačavanjem željene *Field* klase. Struktura polja obrazaca mogu se dodatno specificirati dodavanjem parametara pripadajućim klasama. Na primjer, dodavanjem parametra *widget* definira se izgled pripadajuće HTML oznake [32].

7.4.1 Postupak konstrukcije obrazaca na primjeru obrasca ObjavaForm

Na primjeru obrasca za unos podataka o novoj objavi objasniti ćemo osnovne koncepte definiranja obrazaca.

Definiciju obrasca započinjemo imenovanjem klase kojom nadjačavamo ugrađenu Django klasu *forms.ModelForm*:

```
class ObjavaForm(forms.ModelForm):
```

Svako polje obrasca koje želimo prikazati korisniku definiramo stvaranjem varijable koju inicijaliziramo željenom klasom, ovisno o tipu unosa koji očekujemo od korisnika. Ovoj klasi možemo proslijediti dodatne parametre npr. *required* za označavanje obaveznog unosa. Kao parametar možemo proslijediti i *widget*, a njime označavamo koji tip polja želimo da Django stvori u obrascu pripadajuće .html datoteke. *Widgetu* također možemo proslijediti dodatne parametre npr. željene CSS klase polja. Donja slika prikazuje definiciju polja *naslov*:

```
    naslov = forms.CharField(
        label="Naslov",
        required=True,
        widget=forms.widgets.Textarea(
            attrs={
                "placeholder": "Tijesto za pizzu",
                "class": "input is-grey-light is-medium",
            }
        ),
    )
```

Nalik definiciji modela, u klasi obrasca definiramo i *Meta* klasu. Ona sadrži naziv modela na temelju kojeg će se obrazac stvoriti, a varijablama *fields* i *exclude* može se definirati točno koja polja obrazac treba imati. Obrazac *ObjavaForm* temelji se na modelu *Objava*, a polja čije unose ne želimo omogućiti, npr. biranje korisnika koji je stvorio objavu, definiramo u varijabli *exclude*:

```
    class Meta:
        model = Objava
        exclude = ("stvorio", "objava_srca", )
```

Pozivanje nekog obrasca radimo u željenom pogledu. Na primjer, u pogledu koji vraća stranicu gdje korisnik unosi podatke o objavi pozivamo obrazac *ObjavaForm*:

```
objava_form = ObjavaForm()
```

Slanje obrasca u predložak odvija se definiranjem konteksta o kojemu će više riječi biti u slijedećem poglavlju. Na predlošku pomoću proslijeđenog konteksta izvlačimo pojedina polja, a to obavezno radimo unutar <*form*> elementa sa oznakom metode *POST*.

Svaki obrazac dodatno se osigurava *csrf_token* dekoratorom koji validira da uneseni podaci nisu zlućudni:

```
<form method="POST" enctype="multipart/form-data" class="box">
    <div class="title">
        <p>Nova objava</p>
    </div>
    {% csrf_token %}
    <div class="field">
        <label class="label" for="{{
objava_form.naslov.id_for_label }}">Naslov:.</label>
        <div class="control">
            {{ objava_form.naslov }}
            <span class="icon">
            <i class="fas fa-user"></i>
            </span>
        </div>
        {{ objava_form.naslov.errors }}
    </div>
...
</form>
```

Kako bi se uneseni podaci proslijedili POST zahtjevom, u obrazac obavezno treba uključiti i <*button*> element tipa *submit:*

```
<button class="button" type="submit">
    ...
</button>
```

Svi ostali obrasci aplikacije definiraju se istom logikom.

7.5 Pogledi i predlošci

Pogledi i predlošci centralni su dio svake Django aplikacije. Pogledi se definiraju u datoteci *views.py*, a određuju koji podaci će se prikazati korisniku. Pogledi zovu predloške koji definiraju na koji način će se ti podaci prikazivati. Konkretno, pogled je obična Python funkcija koje vraća određeni predložak te mu šalje neki kontekst. Kontekst je Python rječnik (engl. *dictionary*) sastavljen od podataka i pripadajućih naziv. Predlošci su pak obične *.html* datoteke koje se koriste kontekstom kako bi korisniku dinamično prikazali odabrane podatke. Django se pri definiranju predložaka koristi posebnim jezikom zvanim „Django template language“ (kratica *DTL*). U Django aplikaciji moguće je koristiti i neki drugi jezik za predloške, a popularna alternativa *DTL-u* je jezik zvan *Jinja2* [33].

7.5.1 Nasljeđivanje predložaka

Za neke stranice aplikacije želimo koristiti iste elemente aplikacije. Na primjer, na svim stranicama kojima mogu navigirati ulogirani korisnici želimo prikazati istu navigacijsku traku sa gumbima koji vode na naslovnicu, profil ili odjavu. Potrebno je stvoriti takozvani „base“ predložak koji sadrži elemente zajedničke tim stranicama. Predložak *base_logged_in.html* stvaramo u direktoriju *./app/templates*, a njegov kod izgleda ovako:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="...">
    <link rel="stylesheet" href="...“ />
    <title>Mrvica</title>
</head>
<body>
    ...
    <div class="container">
        {% block content %}
        {% endblock %}
    </div>
</body>
</html>
```

U *body* element možemo definirati željene elemente koji će se prikazivati na svim stranicama koje nasljeđuju iz ovog predloška. Mjesto na kojemu se pak „ubacuje“ sadržaj neke druge stranice definiran je slijedećim *DTL* oznakama:

```
        {% block content %}
        {% endblock %}
```

Na svakoj stranici aplikacije koja nasljeđuje gornji predložak treba dodati oznaku *{% extends 'some_template.html' %},* a HTML tagovi koji definiraju strukturu samih stranica dodaju se između gore već navedenih DTL oznaka. Na primjer, predložak *naslovna.html* izgledati će ovako [34]:

```
{% extends 'base_logged_in.html' %}

{% block content %}
<section class="hero">
    Ostale HTML oznake ...
</section>

<div class="columns">
    Ostale HTML oznake ...
</div>
{% endblock %}
```

7.5.2 Stiliziranje predložaka

Iako fokus rada nije na procesima strukturiranja i stiliziranja *.html* datoteka aplikacije tj. predložaka, dobro je navesti o kakvim tehnologijama se radi.

Za stiliziranje predložaka koristi se CSS razvojni okvir „Bulma“. Ovaj razvojni okvir olakšava proces stiliziranja *.html* datoteka na način da nudi predefinirane CSS klase. Cilj Bulme je omogućiti jednostavno stiliziranje stranica klasama koje su responzivno orijentirane te koje korisnicima pružaju ujednačeno i vizualno estetično iskustvo [35]. Bulmu nije potrebno instalirati kao druge pakete projekta, već se online link na cijelu CSS *bulma.css* datoteku ubacuje direktno u *base* predloške aplikacije:

```
<link rel="stylesheet"
href="https://cdn.jsdelivr.net/npm/bulma@0.9.4/css/bulma.min.css">
```

U aplikaciji „Mrvica“ koristimo se i ikonama. Ikone služe kako bi korisnicima na simboličan način ukazali na kakve unose očekujemo u obrascima, čemu određeni gumbi služe itd. Ikone kojima se aplikacija služi dohvaćaju se sa internetske stranice „Font Awesome“ (kratica *FA*). Nalik postavi Bulme, korištenje *FA* ikona omogućavamo ubacivanjem online linka CSS datoteke *font_awesome.css* direktno u *base* predloške:

```
    <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/font-
awesome/6.1.2/css/all.min.css" integrity="sha512-
1sCRPdkRXhBV2PBLUdRb4tMg1w2YPf37qatUFeS7zlBy7jJI8Lf4VHwWfZZfpXtYSLy
85pkm9GaYVYMfw5BC1A==" crossorigin="anonymous" referrerpolicy="no-
referrer" />
```

7.6 Stranice aplikacije

U ovom poglavlju opširnije će se opisati proces implementacije nekih od glavnih stranica aplikacije. Stranice su cjeline koje spajaju modele, obrasce, poglede i predloške, pritom definirajući željene funkcionalnosti aplikacije.

7.6.1 Prijava

Stranicu prijave *login* koristi korisnik kako bi se prijavio u sustav. Sastoji se od obrasca za unos podataka te poveznica na stranicu za resetiranje lozinke i registraciju:

Slika 4: Stranica prijave u aplikaciju

U listu *urlpatterns* datoteke *urls.py* prvo definiramo rutu preko koje će korisnik doći do prijave. Zamišljeno je da se stranica prijave otvara odmah pri ulasku u aplikaciju. Međutim, ova aplikacija služi se Djangovim ugrađenim sustavom za autentikaciju pa ruta treba izgledati točno ovako:

```
    path('', login_request, name='login'),
```

Pogled koji ruta poziva zove se *login_request*, a njegov kod izgleda ovako:

```
def login_request(request):
    if request.method == "POST":
        obrazac_prijave = UserLoginForm(request, data=request.POST)
        if obrazac_prijave.is_valid():
            username = obrazac_prijave.cleaned_data.get('username')
            password = obrazac_prijave.cleaned_data.get('password')
            user = authenticate(username=username,
password=password)
            if user is not None:
                login(request, user)
                return redirect("app:naslovna")
    else:
        obrazac_prijave = UserLoginForm()
    return render(request, "app/login.html", {"obrazac_prijave":
obrazac_prijave})
```

Ova funkcija vraća predložak *login.html* koji definira izgled stranice za prijavu te kontekst koji je sastavljen od obrasca za prijavu:

```
    return render(request, "app/login.html", {"obrazac_prijave":
obrazac_prijave})
```

Pri prvom ulasku na stranicu prijave zahtjev se šalje metodom *GET*, a ona u kontekst šalje prazni obrazac za prijavu.

```
    else:
        obrazac_prijave = UserLoginForm()
```

Ako je zahtjev poslan metodom *POST*, korisnički uneseni podaci iz zahtjeva se prosljeđuju u varijablu *obrazac_prijave*.

```
    if request.method == "POST":
        obrazac_prijave = UserLoginForm(request, data=request.POST)
```

Provjerava se jesu li uneseni podaci validni. Ako jesu, u varijable *username* i *password* dohvaćaju se korisničko ime i lozinka. Zatim se u novu varijablu *user* sprema rezultat autentikacije tj. onaj korisnik čiji su podaci uneseni u obrascu:

```
        if obrazac_prijave.is_valid():
            username = obrazac_prijave.cleaned_data.get('username')
            password = obrazac_prijave.cleaned_data.get('password')
            user = authenticate(username=username,
password=password)
```

Ako je autentikacija uspješna tj. podaci obrasca odgovaraju podacima nekog korisnika, korisnika se prijavljuje u sustav te usmjerava na naslovnu stranicu:

```
            if user is not None:
                login(request, user)
                return redirect("app:naslovna")
```

U predlošku *login.html* nalazi se obrazac za prijavu. Ovaj predložak iz konteksta *obrazac_prijave* izvlači polja definirana klasom UserLoginForm, a koja je ugrađena klasa Djangovog sustava za autentikaciju:

```
<form method="POST">
    <div class="title">
        <p >Prijava</p>
    </div>
    {% csrf_token %}
    {{ obrazac_prijave.non_field_errors }}
    <div class="field">
        <label class="label" for="{{
obrazac_prijave.username.id_for_label }}">Korisničko ime:</label>
        <div class="control">
            {{ obrazac_prijave.username }}
            <span class="icon">
```

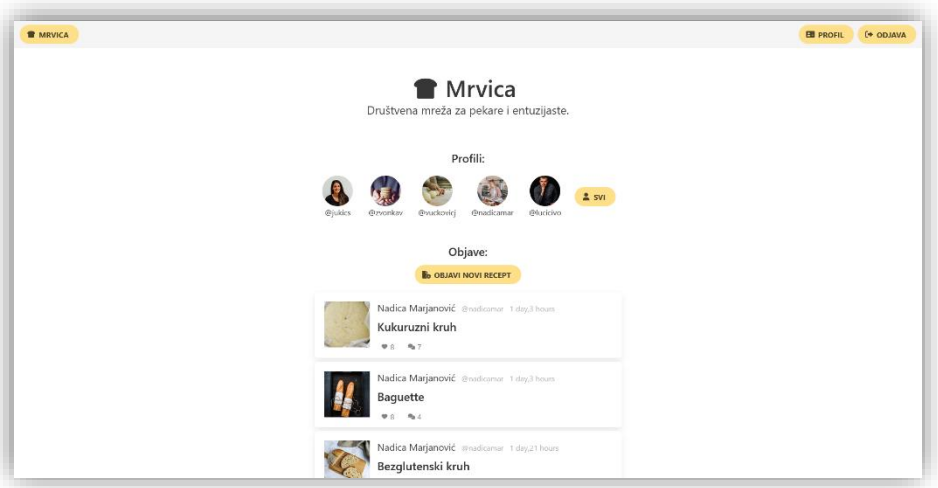
```
            <i class="fas fa-user"></i>
            </span>
        </div>
        {{ obrazac_prijave.username.errors }}
    </div>
    ...
</form>
```

Na dnu obrasca nalaze se i dvije poveznice - jedna koja vodi na stranicu za resetiranje lozinke i druga koja vodi na stranicu za registraciju novog korisnika:

```
<div class="field">
    <a href="/password_reset">Zaboravili ste lozinku?</a>
    </br>
    <a href="/register">Nemate račun?</a>
</div>
```

7.6.2 Naslovnica

Stranica naslovnice *naslovna* otvara se nakon prijave korisnika u sustav. Ona se sastoji od popisa svih profila mreže, gumba za stvaranje nove objave tj. recepta i svih objava onih profila koje korisnik prati:



Slika 5: Naslovna stranica

U listu *urlpatterns* datoteke *urls.py* dodajemo slijedeću rutu:

```
    path('naslovna/', naslovna, name='naslovna'),
```

Ova ruta poziva pogled *naslovna* čiji je kod:

```
@login_required
def naslovna(request):
    profili = list(Profil.objects.exclude(korisnik=request.user))
    if len(profili) > 5:
        random_profili = random.sample(profili, 5)
    else:
        random_profili = profili
    return render(request, 'app/naslovna.html', {'profili':
random_profili})
```

Dekorator *@login_required* omogućava pristup naslovnici samo ako je neki korisnik prijavljen u sustav. Anonimnim tj. neprijavljenim korisnicima pristup naslovnici nije omogućen:

```
@login_required
```

Pogled *naslovna* vraća predložak *naslovna.html* u kojemu je definiran izgled naslovnice te kontekst koji se sastoji od ostalih profila aplikacije:

```
    return render(request, 'app/naslovna.html', {'profili':
random_profili})
```

Slanjem *GET* zahtjeva za naslovnicom, pogled u varijablu *profili* sprema listu svih profila osim onog koji je vezan uz prijavljenog korisnika. Na naslovnici želimo prikazati nekoliko profila koje bi korisnik htio zapratiti, a njih nasumično dohvaćamo iz liste. Ako je broj registriranih profila manji od zadanog broja, dohvaćamo sve profile iz te liste:

```
    profili = list(Profil.objects.exclude(korisnik=request.user))
    if len(profili) > 5:
        random_profili = random.sample(profili, 5)
    else:
        random_profili = profili
```

Profili su jedini kontekst koji pogledom šaljemo u predložak *naslovna.html*. Postojanje bloka profila provjeravamo postavljanjem kondicionala tj. ako u kontekstu nema profila, blok profila neće se stvoriti na stranici:

```
{% if profili%}
<div class="block">
    ...
</div>
{% endif %}
```

Ako se kontekst pak sastoji od nekog broja profila, njih ispisujemo definiranjem for petlje unutar gornjeg kondicionala. For petljom prolazimo kroz svaki profil iz konteksta i prikazujemo njegove podatke unutar definiranih HTML oznaka:

```
    {% for profil in profili %}
    <div class="column">
        ...
    </div>
    {% endfor %}
```

Podatke o profilima ispisujemo odabirom atributa preko DTL oznaka: Na primjer, ispisivanje imena, korisničkog imena i datuma stvaranja računa u aplikaciji izgleda ovako:

```
<div class="content">
    <p>
        <span class="title"> {{ objava.stvorio.profil.ime }}
</span>
        <span class="subtitle"> @{{ objava.stvorio.username }}
</span>
        <span class="subtitle"> {{
objava.vrijeme_stvaranja|timesince }}</span>
    </p>
</div>
```

Na naslovnici se nalaze i objave onih profila koje korisnik prati. Međutim, podaci o tim objavama ne izvlače se iz definiranog konteksta. To je zato što se u predlošku možemo koristiti podacima prijavljenog korisnika kroz Djangov *DTL* jezik. Kako bi izvukli objave praćenih korisnika potrebno je ugnijezditi dvije petlje. Prva petlja iz podataka prijavljenog korisnika izvlači podatke o svim profilima koje on prati, a druga petlja za svaki od tih profila izvlači sve objave:

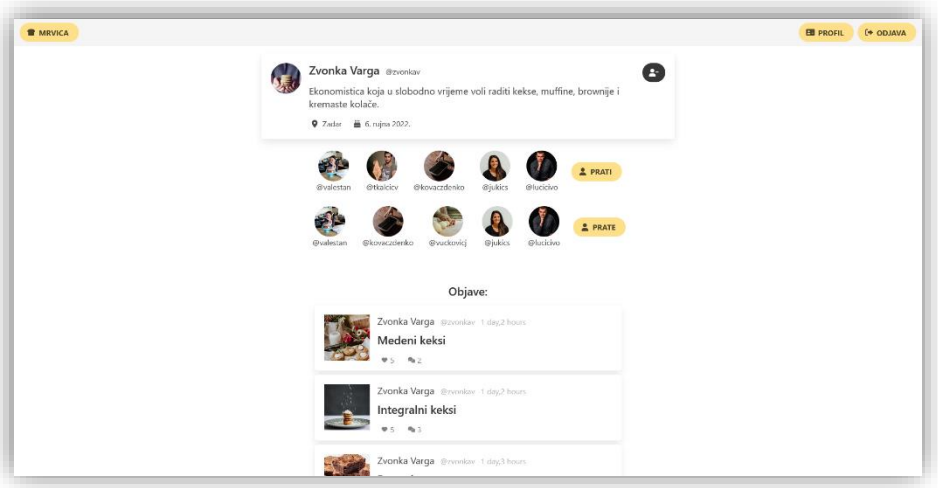
```
{% for praceni_profil in user.profil.prati.all %}
    {% for objava in praceni_profil.korisnik.objave.all %}
        <div class="block">
            ...
        </div>
    {% endfor %}
{% endfor %}
```

Podatke o objavama ispisujemo istim postupkom kao i za profile. Međutim, atributi objave poput sastojaka i uputa vraćaju nam se u obliku liste pa je njome potrebno proći for petljom. Dodatno, preusmjeravanje klikom na specifičnu objavu omogućavamo definiranjem HTML oznake *<a>*. Vrijednost *href* sadrži DTL oznaku *url* kojom Djangu govorimo da se pozove pogled *objava*, a njemu proslijedi *id* objave. Taj dio predloška izgleda ovako:

```
<div class="content">
    <p>
        <a href="{% url 'app:objava' objava.id %}">
            <span class="title"> {{ objava.naslov }} </span>
        </a>
    </p>
    <ul>
        {% for sastojak in objava.sastojci_as_list %}
                <li>{{ sastojak }}</li>
        {% endfor %}
    </ul>
    <ol>
        {% for uputa in objava.upute_as_list %}
                <li>{{ uputa }}</li>
        {% endfor %}
    </ol>
    <p>
        {{ objava.napomene }}
    </p>
</div>
```

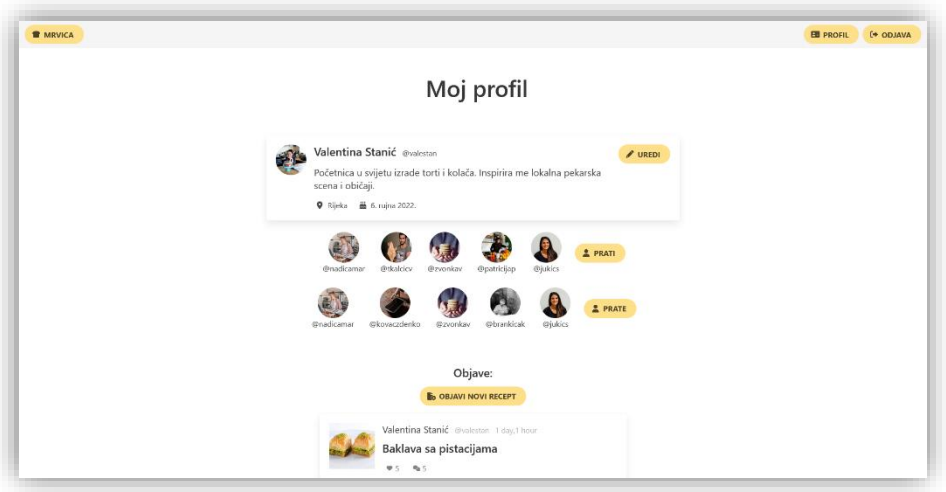
7.6.3 Profil

Stranica profila *profil* prikazuje informacije o specifičnom profilu. Sastoji se od bloka sa informacijama o profilu, gumba za praćenje, liste profila koje prate ovaj profil, liste profila koje ovaj profil prati te svih objava profila:



Slika 6: Stranica profila

Stranica profila *profil* trenutno prijavljenog korisnika dodatno sadrži gumb na stranicu za uređivanje profila te gumb za objavljivanje novog recepta:



Slika 7: Stranica profila prijavljenog korisnika

U listu *urlpatterns* datoteke *urls.py* dodajemo rutu. Ruta na određeni profil sadrži i njegov primarni ključ. Primarni ključ se pogledu prosljeđuje iz predloška na kojemu je korisnik kliknuo na određeni profil:

```
    path('profil/<int:pk>', profil, name='profil'),
```

Ruta poziva pogled *profil* koji izgleda ovako:

```
@login_required
def profil(request, pk):
    profil = Profil.objects.get(pk=pk)

    if request.method == "POST":
        logirani_profil = request.user.profil
        radnja_pracenje = request.POST.get("pracenje")

        if radnja_pracenje == "prati":
            logirani_profil.prati.add(profil)
        elif radnja_pracenje == "neprati":
            logirani_profil.prati.remove(profil)

        logirani_profil.save()

    return render(request, "app/profil.html", {"profil": profil})
```

Pogled *profil* vraća predložak *profil.html* u kojemu je definiran izgled stranice profila i kontekst koji se sastoji od informacija o profilu:

```
    return render(request, "app/profil.html", {"profil": profil})
```

Kako se ruta sastoji i od primarnog ključa specifičnog profila, taj ključ hvatamo u listi parametra funkcije:

```
def profil(request, pk):
```

Ovaj primarni ključ zatim koristimo kako bi iz baze izvukli taj profil:

```
    profil = Profil.objects.get(pk=pk)
```

Za otvaranje profila zahtjev se šalje metodom *GET*. Međutim, ako smo na profilu podnijeli neku akciju npr. zapratili profil, zahtjev se šalje metodom *POST* zajedno sa podacima akcije.

```
    if request.method == "POST":
```

Na stranici profila omogućeno je praćenje istog tj. korisnik može zapratiti ili otpratiti profil. Kako bi korisnik izveo tu akciju nad profilom, potrebno je prvo izvući podatke o njegovom vlastitom profilu i akciju koja se proslijedila zahtjevu. Akcija *praćenje* može imati dvije vrijednosti – *prati* i *neprati*. Ovisno o poslanoj akciji, korisniku se u listu praćenih profila dodaje ili uklanja ovaj profil:

```
    if request.method == "POST":
        logirani_profil = request.user.profil
        radnja_pracenje = request.POST.get("pracenje")

        if radnja_pracenje == "prati":
            logirani_profil.prati.add(profil)
        elif radnja_pracenje == "neprati":
            logirani_profil.prati.remove(profil)

        logirani_profil.save()
```

Gumb za praćenje se na predlošku nalazi ovisno o tome je li trenutni profil korisnikov ili ne. Ako je korisnik na profilu drugog korisnika, gumb za praćenje se prikazuje:

```
<form method="post">
    {% csrf_token %}
    {% if profil != user.profil %}
        ...
    {% endif %}
</form>
```

Dodaje se nova *if-else* izjava koja prikazuje gumb ovisno o statusu praćenja profila. Ako korisnik prati ovaj profil, gumb koji se prikazuje je *odprati*, a akcija koja se šalje odabirom gumba je *neprati*. Ako korisnik ne prati ovaj profil, prikazuje se gumb *zaprati*, a akcija koje se šalje odabirom gumba je *prati*:

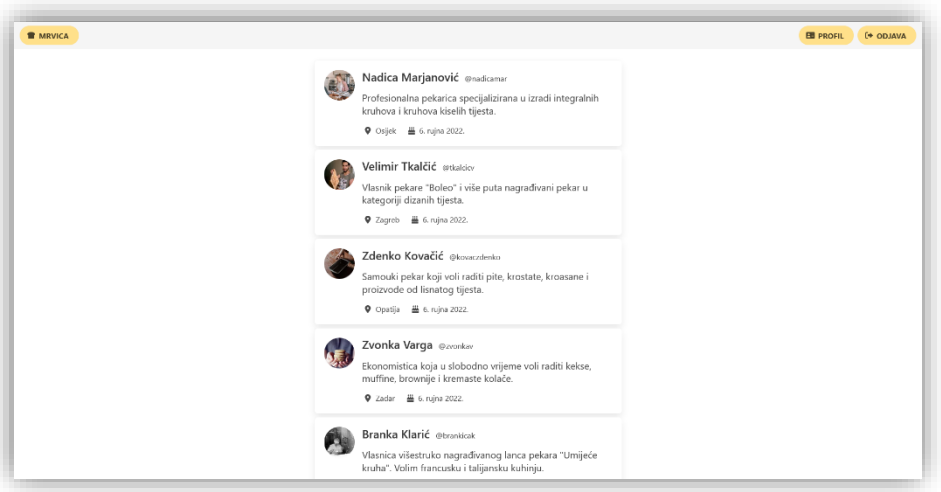
```
        {% if profil in user.profil.prati.all %}
            <button class="button" name="pracenje" value="neprati">
                <span class="icon"> <i class="fa-solid fa-user-
minus"></i> </span>
            </button>
        {% else %}
            <button class="button" name="pracenje" value="prati">
                <span class="icon"> <i class="fa-solid fa-user-
plus"></i> </span>
            </button>
        {% endif %}
```

Ako je korisnik pak na vlastitom profilu, prikazuje se gumb *Uredi profil* koji vodi na stranicu sa obrascem za promjenu informacija. Za preusmjeravanje se koristi *DTL* oznaka *url* sa nazivom samog pogleda:

```
{% if profil == user.profil %}
<a class="button" href="{% url 'app:uredi_profil' %}">
    <span class="icon"> <i class="fa-solid fa-address-card"></i>
</span>
    <span class=""> uredi profil </span>
</a>
{% endif %}
```

7.6.4 Profili

Stranica *profili* prikazuje informacije o svim profilima mreže osim profila trenutno prijavljenog korisnika. Sastoji se od blokova informacija za svaki od profila mreže:



Slika 8: Stranica svih profila mreže

U listu *urlpatterns* datoteke *urls.py* dodajemo rutu:

```
    path('profili/', profili, name='profili'),
```

Ruta poziva pogled *profili* koji izgleda ovako:

```
@login_required
def profili(request):
    profili = Profil.objects.exclude(korisnik=request.user)
    return render(request, 'app/profili.html', {'profili':
profili})
```

Pogled *profil* vraća predložak *profiil.html* u kojemu je definiran izgled stranice i kontekst koji se sastoji od informacija o svim profilima mreže, osim profila trenutno prijavljenog korisnika:

```
    return render(request, 'app/profili.html', {'profili':
profili})
```

Sve profile mreže, osim korisnikovog, dohvaćamo koristeći Djangovu *exclude* metodu. Ona dohvaća sve retke tablice *Profil*, izuzimajući onaj redak čija je vrijednost stranog ključa *korisnik* jednaka ključu koji je proslijeđen metodi kao parametar. U ovom slučaju to je primarni ključ trenutno prijavljenog korisnika:

```
    profili = Profil.objects.exclude(korisnik=request.user)
```

Na predlošku *profili.html* for petljom prolazimo kroz kontekst tj. sve dohvaćene profile, stvarajući blok informacija za svaki profil:

```
{% for profil in profili %}
    <div class="block">
        <div class="box">
            ...
        </div>
    </div>
{% endfor %}
```

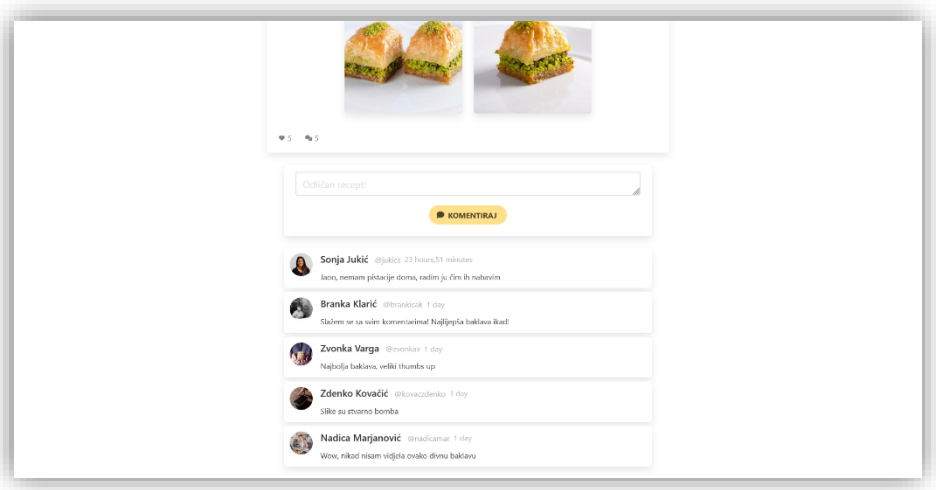
Za svaki blok profila omogućeno je i preusmjeravanje korisnika na taj profil. To je implementirano definiranjem HTML oznake *<a>* i DTL oznake *url*. DTL oznaci prosljeđujemo i dohvaćeni id profila kako bi se prikazala stranica specifičnog profila:

```
<a href='{% url 'app:profil' profil.id %}'>
    <span class="title">{{ profil.ime }}</span>
    <span class="subtitle">@{{ profil.korisnik.username }}</span>
</a>
```

7.6.5 Objava

Stranica *objava* sadrži podatke i slike određene objave, mogućnost lajkanja objave, blok za komentare i obrazac za stvaranje novog komentara:

Slika 9: Stranica objave



Slika 10: Stranica objave, komentari

U listu *urlpatterns* datoteke *urls.py* dodajemo rutu. Nalik stranici *profil*, ruta sadrži i primarni ključ objave. Primarni ključ iz predloška prosljeđuje se pogledu koji zatim vraća stranicu specifične objave:

```
    path('objava/<int:pk>', objava, name='objava'),
```

Ruta poziva pogled *objava* koji izgleda ovako:

```
@login_required
def objava(request, pk):
    objava = Objava.objects.get(pk=pk)
    trenutni_path = request.path

    if request.method == "POST":
        obrazac_komentara = KomentarForm(request.POST)
        if obrazac_komentara.is_valid():
            komentar = obrazac_komentara.save(commit=False)
            komentar.objava = objava
            komentar.stvorio = request.user
            komentar.save()
            return redirect(trenutni_path)
        logirani_korisnik = request.user
        radnja_srce = request.POST.get("srce")

        if radnja_srce == "voli":
            objava.objava_srca.add(logirani_korisnik)
        elif radnja_srce == "nevoli":
            objava.objava_srca.remove(logirani_korisnik)

        objava.save()
        return redirect(trenutni_path)
    else:
        obrazac_komentara = KomentarForm()
    return render(request, "app/objava.html", {"objava": objava,
"obrazac_komentara": obrazac_komentara, })
```

Pogled *objava* vraća predložak *objava.html* te kontekst sačinjen od podataka same objave i obrasca za unos novog komentara.:

```
    return render(request, "app/objava.html", {"objava": objava,
"obrazac_komentara": obrazac_komentara, })
```

Pogledu se dodatno prosljeđuje i priparni ključ selektirani objave, kako bi se iz redaka tablice *Objava* mogla dohvatiti specifična objava:

```
def objava(request, pk):
```

Objava se zatim dohvaća uz pomoć primarnog ključa. U isto vrijeme dohvaćamo i trenutnu rutu korisnika u slučaju da objavljuje komentar. Kada se komentar objavi, potrebno je korisnika preusmjeriti na istu stranicu kako se komentar ne bi više puta zapisao (engl. *submit*):

```
    objava = Objava.objects.get(pk=pk)
    trenutni_path = request.path
```

Ako se zahtjev za stranicom poslao metodom *GET*, ide se na *else* blok glavne if-else izjave, prosljeđujući podatke objave i prazni obrazac za unos novog komentara:

```
    else:
        obrazac_komentara = KomentarForm()
```

Ako se zahtjev pak poslao *POST* metodom, potrebno je iz zahtjeva izvući podatke. Mogući podaci uključuju sadržaj novog komentara i vrijednost akcije lajkanja.

Ukoliko su podaci uneseni u obrazac komentara validni, komentar se sprema. To radimo tako da vanjski ključ komentara *objava*, koji ga povezuje sa nekom objavom, postavljamo na vrijednost id-a trenutne objave. Dodatno, u polje *stvorio* spremamo id korisnika koji je taj komentar napisao. Na kraju, korisnika treba preusmjeriti na istu stranicu kako se komentar ne bi zapisao više puta:

```
        obrazac_komentara = KomentarForm(request.POST)
        if obrazac_komentara.is_valid():
            komentar = obrazac_komentara.save(commit=False)
            komentar.objava = objava
            komentar.stvorio = request.user
            komentar.save()
            return redirect(trenutni_path)
```

U podacima zahtjeva poslanog *POST* metodom mogu se nalaziti i podaci o *lajkanju* tj. je li korisnik poslao akciju označavanja objave srcem ili akciju brisanja srca. Ukoliko je korisnik podnio jednu od tih dviju akcija, rezultat se iz predloška šalje u varijablu *radnja_srce*. Dodatno, id trenutno prijavljenog korisnika sprema se u varijablu *logirani*_korisnik. Podnesena radnja se zatim provjerava if-else izjavom, a vanjski ključ objave i korisnika spremaju se u atribute srca:

```
        logirani_korisnik = request.user
        radnja_srce = request.POST.get("srce")

        if radnja_srce == "voli":
            objava.objava_srca.add(logirani_korisnik)
        elif radnja_srce == "nevoli":
            objava.objava_srca.remove(logirani_korisnik)
```

Na predlošku *objava.html* podatke objave možemo prikazati pozivanjem atributa nad prethodno definiranim kontekstom *objava*. Sastojci i upute dolaze u listama stoga je kroz njih potrebno proći *for* petljom. Kako unos *napomena* nije obavezan u obrascu objave, *if* izrazom provjerava se postoje li napomene prije stvaranja njihova bloka:

```
<div class="content">
    <p>
        <a href='{% url 'app:profil' objava.stvorio.profil.id %}'>
            <span class="title">{{ objava.stvorio.profil.ime
}}</span>
            <span class="subtitle">@{{ objava.stvorio.username
}}</span>
        </a>
        <span class="subtitle">{{
objava.vrijeme_stvaranja|timesince }}</span>
    </p>
    <p class="title">
        Sastojci:
    </p>
    <ul>
        {% for sastojak in objava.sastojci_as_list %}
            <li>{{ sastojak }}</li>
        {% endfor %}
    </ul>
    <p class="title">
        Upute:
    </p>
    <ol>
        {% for uputa in objava.upute_as_list %}
            <li>{{ uputa }}</li>
        {% endfor %}
    </ol>
    {% if objava.napomene %}
        <p class="title">Napomene:</p>
        <p>{{ objava.napomene }}</p>
    {% endif %}
</div>
```

Kako bi prikazali sve slike vezane uz trenutnu objavu, potrebno je pristupiti svim njezinim slikama. Django kod definiranja modela *Objava* omogućava stvaranje reverzibilnog imena kojim do svih slika možemo doći kroz samu objavu. Zbog ovoga nije potrebno filtrirati sve slike po *id-u* objave već ih možemo samo pozvati kroz atribut objave *objava_slike*. For petljom zatim prolazimo kroz slike i stvaramo blok za svaku od njih:

```
{% for objava_slika in objava.objava_slike.all %}
    <div class="column">
        ...
    </div>
{% endfor %}
```

Svaka objava ima prikaz broja srca, mogućnost označavanja iste srcem te broj komentara. Broj srca prikazuje se na gumbu, a do njih možemo doći pozivanjem atributa *objava_srca* trenutne objave. Glavni *if-else* provjerava je li to objava trenutno prijavljenog

korisnika te ako je, onemogućava lajkanje tj. samo prikazuje broj srca. Unutarnji *if-else* izraz definira različit stil gumba ovisno o tome je li korisnik označio objavu srcem ili nije. Dodatno, prva vrsta gumba šalje akciju *voli*, a druga akciju *nevoli*. Ove akcije šalju se *POST* zahtjevom u varijabli *srce*. Ispod ovi *if-else* blokova nalazi se i onemogućeni gumb koji prikazuje broj komentara:

```
<form method="post">
    {% csrf_token %}
    <div class="buttons">
        {% if  objava.stvorio == user %}
        <button class="button" disabled>
            <span class="icon"><i class="fas“></i></span>
            <span class="is-size-6"> {{ objava.srca.count }}
</span>
        </button>
        {% else %}
            {% if user in objava.objava_srca.all %}
                <button class="button" name="srce" value="nevoli">
                    <span class="icon"><i class="fas“></i></span>
                    <span class="is-size-6"> {{ objava.srca.count
}} </span>
                </button>
            {% else %}
                <button class="button" name="srce" value="voli">
                    <span class="icon"><i class="fas“></i> span>
                    <span class="is-size-6"> {{ objava.srca.count
}} </span>
                </button>
            {% endif %}
        {% endif %}
        <button class="button" disabled>
            <span class="icon"><i class="fas“></i></span>
            <span class="is-size-6"> {{ objava.komentari.count }}
</span>
        </button>
    </div>
</form>
```

U bloku objave može se naći i gumb koji omogućava brisanje objave. Ako su id korisnika koji je stvorio objavu i korisnika koji je trenutno prijavljen, gumb se prikazuje. Brisanje se izvršava stvaranjem poveznice sa DTL oznakom *url* koja poziva pogled *brisi_objavu*, prosljeđujući i *id* objave.

```
{% if objava.stvorio == user %}
    <div class="level-right">
        <div class="buttons">
            <a class="button" href="{% url 'app:brisi_objavu'
objava.id %}">
                <span class="icon"> <i class="fa-solid fa-ban"></i>
</span>
                <span>briši</span>
            </a>
        </div>
    </div>
{% endif %}
```

Ispod bloka objave nalazi se obrazac za komentiranje. Ovaj obrazac proslijeđen je kontekstom pogleda *objava*, a sastoji se samo od tijela komentara. Kada korisnik klikne gumb „komentiraj“, podaci se proslijede pogledu te se komentar sprema i prikazuje:

```
<form method="post">
    {% csrf_token %}
    {{ obrazac_komentara.tijelo }}
    <div class="buttons">
        <button class="button" type="submit">
            <span class="icon"> <i class="fa-solid fa-comment"></i>
</span>
            <span>komentiraj</span>
        </button>
    </div>
</form>
```

Na kraju stranice nalaze se svi komentari objave. Odabirom reverzibilnog atributa *komentari* ove objave, *for* petljom prolazimo kroz sve povezane komentare i ispisujemo njihov sadržaj:

```
{% for komentar in objava.komentari.all %}
    <div class="box">
        ...
        <div class="content">
            {{ komentar.tijelo }}
        </div>
    </div>
{% endfor %}
```

Ako je *id* korisnika koji je stvorio određeni komentar jednak *id-u* prijavljenog korisnika, na desnoj strani komentara prikazuje se gumb za brisanje komentara. Klikom na gumb korisnika se preusmjerava na pogled *brisi_komentar*, zajedno sa *id-em* objave i *id-em* komentara.

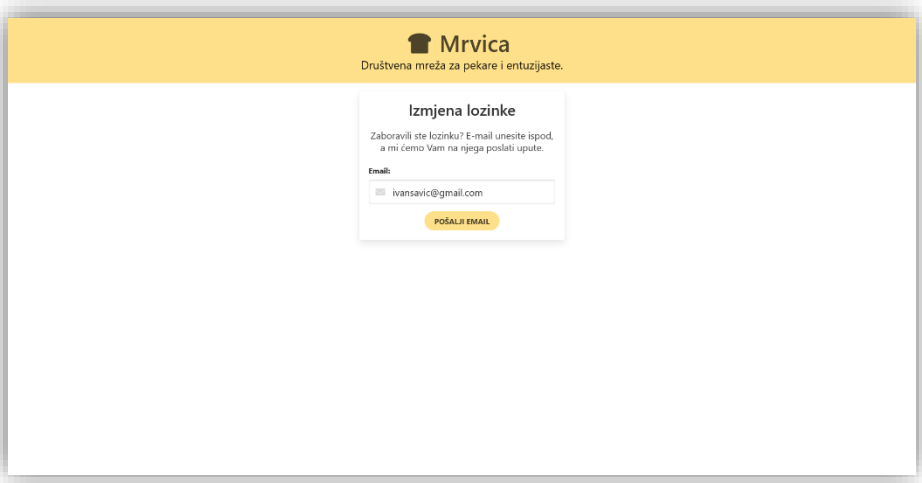
```
{% if komentar.stvorio == user %}
    <div class="level-right">
        <a href="{% url 'app:brisi_komentar' objava.id komentar.id
%}">
            <button class="button">
                <span class="icon"> <i class="fa-solid fa-comment-
slash"></i> </span>
                <span>briši</span>
            </button>
        </a>
    </div>
{% endif %}
```

7.6.6 Ostale stranice aplikacije

Stranica za registraciju *register* sadrži obrazac za unos podataka novog korisnika te poveznicu na stranicu prijave ukoliko korisnik već ima račun.

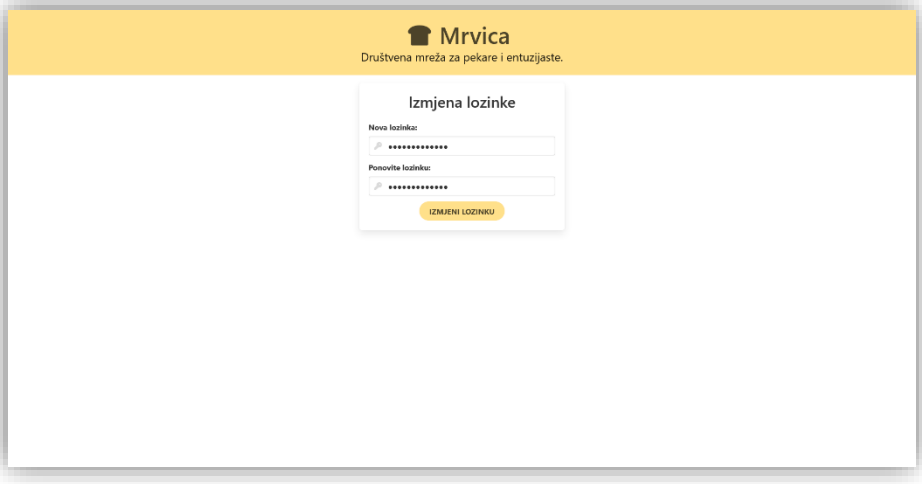
Slika 11: Stranica registracije novog korisnika

Stranice za resetiranje lozinke postojećeg korisnika omogućavaju promjenu lozinke korisniku ukoliko ju je zaboravio ili izgubio.



Slika 12: Stranica za izmjenu lozinke, unos email adrese

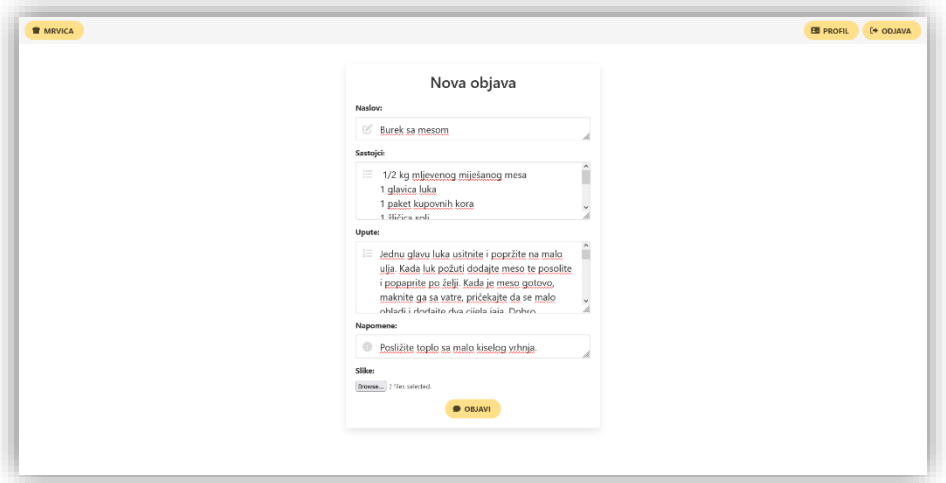
Slika 13: Stranica za izmjenu lozinke, slanje email poveznice



Slika 14: Stranica za izmjenu lozinke, izmjena lozinke

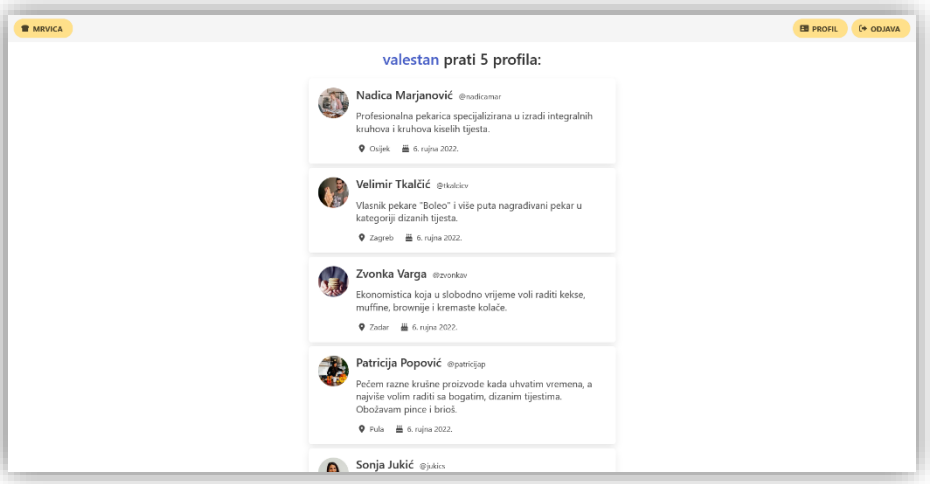
Slika 15: Stranica za izmjenu lozinke, poruka uspjeha

Stranica *nova_objava* sadrži obrazac za unos podataka o novoj objavi tj. receptu:

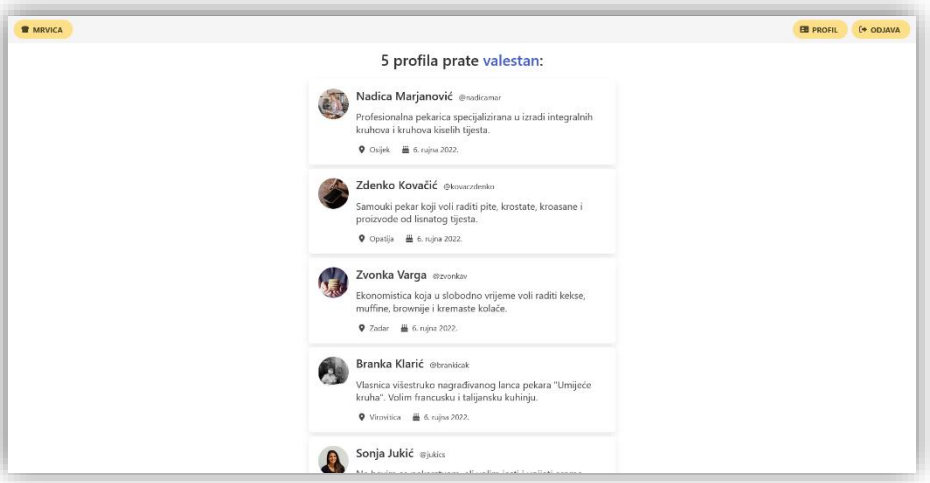


Slika 16: Stranica za stvaranje nove objave

Stranice *prati* i *prate* sadrže popise profila koje trenutno prijavljeni korisnik prati i profila koji prate trenutno prijavljenog korisnika.

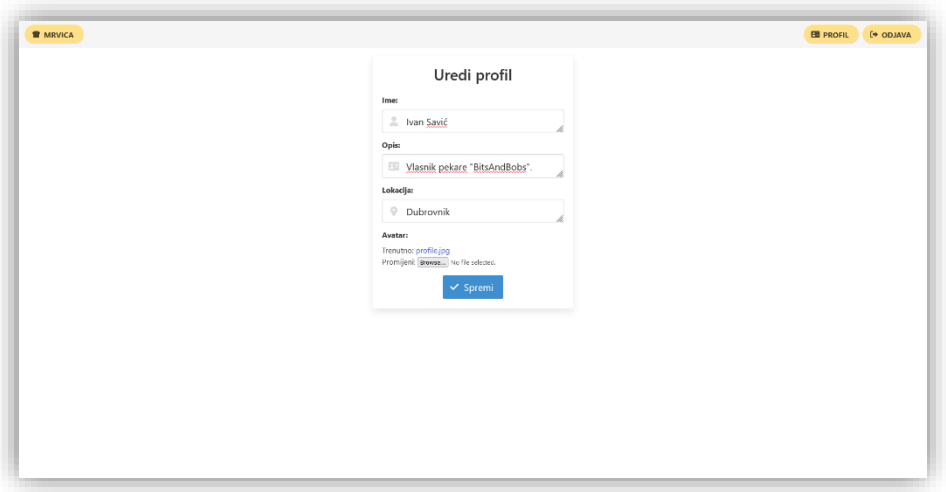


Slika 17: Stranica profila koje korisnik prati



Slika 18: Stranica profila koji prate korisnika

Stranica za uređivanje profila trenutno prijavljenog korisnika *uredi_profil* sadrži obrazac za promjenu informacija profila.



Slika 19: Stranica za uređivanje profila korisnika

8. Zaključak

Društvena mreža „Mrvica“ je web aplikacija izrađena u Django-u koja omogućava povezivanje pekara i entuzijasta na području Hrvatske. Ono što ju čini dobrim izborom za takve korisnike je mogućnost jednostavnog objavljivanja recepata, praćenja drugih korisnika, komentiranja drugih radova te jednostavnog učitavanja slika tj. rezultata radova. Sučelje je jednostavno, a inspirirano je stranicama poput Twittera, Facebooka, Instagrama itd. Ovime se izbjegava potreba korisnika da uči rad u novom sučelju, a omogućava nesmetano sudjelovanje u širenju sadržaja i utjecaja mreže.

Unatoč tome što „Mrvica“ sadrži sve osnovne funkcionalnosti jedne društvene mreže, za buduća poboljšanja ima mnogo prostora. Prvi korak bio bi podizanje aplikacije na Internet preko platforme poput npr. Heroku. Bilo bi pametno zamijeniti ugrađenu SQLite bazu podataka nekom drugom poput npr. PostgreSQL bazom koja nudi više funkcionalnosti. Proces autentikacije također bi bilo dobro proširiti mogućnošću prijave preko računa neke druge mreže npr. Google, Facebook itd. Za kraj, lokaliziranjem društvene mreže drugim jezicima omogućilo bi proširivanje područja korištenja aplikacije na brojne druge države.

Zaključno, zadovoljan sam rezultatom rada te ću nastaviti raditi na „Mrvici“ kako bi ideju koja ona nosi realizirao na Internetu.

Popis literature

[1] Web App Definition, zadnji pristup
https://v2cloud.com/glossary/web-app-definition

[2] Backend vs Frontend: How Are They Different?, zadnji pristup
https://kinsta.com/blog/backend-vs-frontend/

[3] Websites: past and present, zadnji pristup
https://enonic.com/blog/websites-past-and-present

[4] Web App Development in 2022: Everything You Need to Know, zadnji pristup
https://www.trio.dev/blog/web-app-development

[5] MERN Stack Explained, zadnji pristup
https://www.mongodb.com/mern-stack

[6] What is Django?, zadnji pristup
https://tutorial.djangogirls.org/en/django/

[7] Difference between MVC and MVT architecture, zadnji pristup
https://www.geekinsta.com/difference-between-mvc-and-mvt/

[8] Most popular web frameworks among developers worldwide 2022, zadnji pristup
https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web/

[9] Django (web framework), zadnji pristup
https://en.wikipedia.org/wiki/Django_(web_framework)

[10] When and Why Would I Need a JS Framework for Django?, zadnji pristup
https://vsupalov.com/when-and-why-js-framework-for-django/

[11] 6 Python Packages you should be using in every Django Web App, zadnji pristup
https://ordinarycoders.com/blog/article/django-python-packages

[12] Django Developers Survey 2021, zadnji pristup
https://lp.jetbrains.com/django-developer-survey-2021-486/

[13] Django MVT Architecture, zadnji pristup
https://www.askpython.com/django/django-mvt-architecture

[14] Security in Django¶, zadnji pristup
https://docs.djangoproject.com/en/4.1/topics/security/

[15] Password management in Django, zadnji pristup
https://docs.djangoproject.com/en/4.1/topics/auth/passwords/

[16] Top 10 Django Alternatives, zadnji pristup
https://blog.back4app.com/top-10-django-alternatives/

[17] Laravel Documentation, zadnji pristup
https://laravel.com/docs/9.x

[18] Ruby on Rails, zadnji pristup
https://en.wikipedia.org/wiki/Ruby_on_Rails

[19] SDLC - Waterfall Model, zadnji pristup
https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm

[20] Waterfall Methodology, zadnji pristup
https://www.workfront.com/project-management/methodologies/waterfall

[21] Getting started with Django, zadnji pristup
https://www.djangoproject.com/start/

[22] Quick install guide, zadnji pristup
https://docs.djangoproject.com/en/4.1/intro/install/

[23] Writing your first Django app, part 1, zadnji pristup
https://docs.djangoproject.com/en/4.1/intro/tutorial01/

[24] Django Deployment to Github, zadnji pristup
https://www.javatpoint.com/django-deploy-on-github

[25] Django settings, zadnji pristup
https://docs.djangoproject.com/en/4.1/topics/settings/

[26] Working with Static and Media Files in Django, zadnji pristup
https://testdriven.io/blog/django-static-files/

[27] Using the Django authentication system, zadnji pristup
https://docs.djangoproject.com/en/4.1/topics/auth/default/

[28] Send, Receive, and Test Emails in Django, zadnji pristup
https://mailtrap.io/blog/django-send-email/

[29] URL dispatcher, zadnji pristup
https://docs.djangoproject.com/en/4.0/topics/http/urls/

[30] Models, zadnji pristup
https://docs.djangoproject.com/en/4.1/topics/db/models/

[31] Signals, zadnji pristup
https://docs.djangoproject.com/en/3.1/ref/signals/

[32] Working with forms, zadnji pristup
https://docs.djangoproject.com/en/4.1/topics/forms/

[33] Writing your first Django app, part 3, zadnji pristup
https://docs.djangoproject.com/en/4.1/intro/tutorial03/

[34] Template extending, zadnji pristup
https://tutorial.djangogirls.org/en/template_extending/#create-a-base-template

[35] Bulma Documentation, Overview, zadnji pristup
https://bulma.io/documentation/overview/

Popis slika

Popis priloga

Uz rad je priložena aplikacija „Mrvica“.