Web aplikacija za vizualizaciju hijerarhijskih podataka

Bošnjak, Mario

Master's thesis / Diplomski rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:483880

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-20**

Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Mario Bošnjak

WEB APLIKACIJA ZA VIZUALIZACIJU HIJERARHIJSKIH PODATAKA

Diplomski rad

Voditelj rada:
prof. dr. sc. Ivica Nakić

Zagreb, rujan, 2019.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Mojim roditeljima i ženi, te svim dobrim ljudima koji su prošli kroz moj život.

Sadržaj

Sadržaj **iv**

Uvod **2**

1 Web aplikacija **3**
1.1 Aplikacijska logika 3
1.2 Implementacija aplikacije 5
1.2.1 Backend - poslužiteljski sloj 5
1.2.1.1 Operativni sustav 5
1.2.1.2 Baza podataka 5
1.2.1.3 Django Rest Framework 7
1.2.1.3.1 Userapp 8
1.2.1.3.2 Graphbuilder 16
1.2.2 Korisničko sučelje 28
1.2.2.1 Cytoscape.js 28
1.2.2.2 Angular 6 29
1.2.2.2.1 Authentication 32
1.2.2.2.2 Graphapp 37

2 Primjer korištenja aplikacije **51**

Bibliografija **55**

Uvod

Web aplikacija za vizualizaciju hijerarhijskih podataka je nastala iz ideje da se skup podataka koji sadrži međusobne relacije može prikazati u obliku grafa koji bi vizualizirao te podatke kao čvorove, a veze između njih kao bridove. Takvim prikazom podataka bi se olakšalo istraživanje podataka koji tvore skup kompleksnih veza. Kroz eksplicitne primjere i skupove podataka koji se mogu vizualizirati na taj način razvijala se općenitija slika o načinu na koji bi aplikacija trebala raditi i koje bi funkcionalnosti trebala sadržavati, sve dok se nije razvila do generičke razine koja omogućava dva smjera korištenja. Prvi smjer korištenja aplikacije je kreiranje skupa podataka korištenjem sučelja koji se izravno sprema u odgovarajuću strukturu koja pogoduje vizualizaciji u obliku grafa. Drugi smjer korištenja aplikacije je zamišljen na uzimanju skupa podataka koji u svojoj arhitekturi sadrži podatke i veze među njima. Taj skup se prilagođava strukturi podataka te se pomoću API-ja u backendu sprema u bazu podataka te se potom može prikazati na frontend segmentu web aplikacije.

Kroz ova dva smjera korištenja aplikacije moguće je pronaći i različite primjene same aplikacije pa je tako moguće koristiti aplikaciju za vizualizaciju različitih procesa, od poslovnih procesa koji su često prikazani i UML dijagramima pa do kulinarskih recepata. Isto vrijedi i za različite skupove podataka. Moguće je napraviti vizualizaciju društvene mreže ili kemijske strukture pojedine molekule. Sve primjere korištenja aplikacije je moguće postići kroz oba smjera korištenja. Razvojem aplikacije i dubljim razumijevanjem njene arhitekture i primjene dodatno su nastale i određene kombinacije pojmova kojima se tehnička implementacija aplikacije nastoji prikazati na jednostavan i pristupačan način koji bi bio svima razumljiv.

Pristup vizualizaciji i interpretaciji podataka koji je korišten u aplikaciji je komplementaran s metodama koje se koriste prilikom proučavanja podataka korištenjem pristupa za obradu velikih podataka (big data). U pristupu za obradu velikih podataka se većinom koriste statističke metode kojima se nastoji dobiti velika slika i općenita interpretacija, dok aplikacija pomaže smjestiti pojedinu informaciju u kontekst podataka koji na nju utječu te tako olakšava njenu tumačenje i shvaćanje. Implementacija aplikacije je podijeljena u dva osnovna segmenta, korisničko sučelje i backend (poslužiteljska strana aplikacije). Prilikom implementacije, ključnu ulogu je imala biblioteka Cytoscape.js. Uvidom u to kako

su u biblioteci implementirani elementi graf, čvor i brid pristupilo se razvoju backend segmenta aplikacije. Svi ti elementi su definirani kao klase i dodatno proširени kako bi mogli odražavati bitna svojstva koja su za njih namijenjena kroz ovu aplikaciju. Kroz razvoj aplikacije uočena i potreba se dodatno implementiraju dodatne klase koje omogućavaju veću fleksibilnost i funkcionalnost. Klase objekata su na sličan način implementirane i na korisničkom sučelju i na backend segmentu.
Prilikom implementacije su korištene tehnologije koje su javno dostupne, počevši od operativnog sustava, baze podataka, programskih jezika te frameworka u kojima je aplikacija pisana. Frameworci Angular i Django Rest Framework su odabrani prvenstveno zbog svoje modularnosti i mogućnosti scaffoldanja novih segmenata. Ovakve tehnologije ubrzavaju izradu prototipa i omogućavaju testiranje šireg spektra ideja. U kratkom roku je moguće isprobati više različitih pristupa, a segmenti koji se pokažu dobrima mogu se lako koristiti prilikom razvoja ostalih prototipova. U prednosti spadaju i dobra dokumentacija i mnogo primjera korištenja kao i velik broj korisnika ovih frameworka. Na taj način je lakše pronaći rješenja za specifične probleme kao i dobre prakse za razvoj aplikacije.

Poglavlje 1

Web aplikacija

Primjena i osnovna logička arhitektura web aplikacije za vizualizaciju hijerarhijskih podataka je prikazana kroz njenu nomenklaturu kojom se opisuju skupovi podataka koji se mogu vizualizirati u njoj. Sama implementacija aplikacije je postignuta kroz dva osnovna segmenta, a to su backend segment koji sadržava podatke te jedan dio aplikacijske logike, te frontend dijela koji se sastoji od korisničkog sučelja i vizualizacijskog dijela aplikacije. Zbog specifičnosti pojedinih funkcionalnosti, neke su implementirane na frontend dijelu, a neke na backend dijelu aplikacije, dok su neke ostvarene korištenjem oba segmenta.

1.1 Aplikacijska logika

Web aplikacija za vizualizaciju hijerarhijskih podataka se služi analogijom između skupa podataka i grafova. Svaka pojedina informacija se u grafu prikazuje kao čvor, dok su veze između pojedinih čvorova u grafu prikazane kao bridovi. Skup informacija koje posjeduju veze između sebe prikazujemo u aplikaciji kao graf. Svaki podatak bi trebao biti jedinstven te prema svojoj prirodi odgovarati nekoj činjenici, ali se zato može pojavljivati u različitim grafovima, koji predstavljaju različite kontekste u kojima se taj podatak može naći. Kontekst, to jest graf, je skup podataka i veza među njima.
Veze između podataka mogu biti samo simboličke, tako da se samo vizualizira njeno postojanje, a može joj se pridati i značenje i objašnjenje, što produbljuje kontekst i pobliže objašnjava strukturu podataka. Graf je vizualizacija konteksta između skupa podataka te njihovih veza. Svaki kontekst omogućava jasnije tumačenje pojedinog podatka, a svaki novi podatak i njegove veze s ostatkom konteksta mijenjaju cijeli kontekst te se on i svi podatci u skladu s tim iznova tumače.
Proširenje aplikacijske logike se postiže dodavanjem kategorija kojima se lakše može napraviti distinkcija između pojedinih podataka. Kategorije se kroz vizualizaciju mogu postići korištenjem različitih boja i oblika pa se time postiže da korisnik u kratkom periodu, bez

korištenja dodatne aplikacijske logike može vidjeti neke bitne segmente u skupu podataka te kako se oni odnose s ostalim podatcima.

Primjer 1.1.1. Objasnit ćemo gore navedenu aplikacijsku logiku na primjeru skripte ili knjige za pojedini kolegij na PMF-MO. Prvo ćemo definiramo graf koji se u ovom slučaju može zvati Kolegij. Za taj graf ćemo dalje definirati neke osnovne kategorije koje se redovito pojavljuju u matematičkim knjigama, a to su Definicija, Lema, Korolar, Propozicija, Teorem, Primjer. Jedna od kategorija koju ćemo ovaj put mimoići je Zadatak. Svakoj od tih kategorija dodijelimo jedinstveni oblik i boju tako da se na prvi pogled vrlo jednostavno razlikuju u našem grafu te da korisnik može odmah na prvi pogled shvatiti strukturu skupa podataka Kolegij.
Nakon definicije kategorija, krećemo s definiranjem podataka koji će biti prikazani u aplikaciji. Svaka pojedina definicija, lema, korolar, propozicija ili teorem imaju svoje ime ili jedinstvenu oznaku unutar knjige pa će tako svaki od njih biti kreiran kao zasebni podatak takvog imena, a dodatno se njihov tekst priloži tom podatku kao njegov sadržaj. Također će svakom podatku na temelju njegove kategorije biti pridruženi i odgovarajući oblik i boja.
Kada je kreiran skup podataka koji će predstavljati čvorove, kreira se skup veza koje će u grafu biti reprezentirane bridovima. Ukoliko neki podatak u svom sadržaju spominje drugi podatak, doda se veza između tih podataka. Kada se radi o ovakvom skupu podataka, dodaje se veza usmjerena od podatka koji je spomenut prema podatku koji ga spominje iz razloga što u hijerarhiji pojmova u matematičkim udžbenicima ne možemo koristiti pojmove koje nismo prethodno definirali. Na ovaj način se kreiraju posljedične veze koje kasnije mogu biti korisne kada želimo vidjeti koji su sve pojmovi i tvrdnje koriste u iskazu nekog teorema.
Nakon što dodamo sve veze, dobijemo graf Kolegij koji vizualizira pojmove i veze među njima koji čine osnovno znanje iz kolegija kojeg smo obrađivali. Struktura koju ovako dobijemo često nije stablo, nego ima više različitih korijena. U slučaju kolegija, korijeni bi trebali biti iz kategorije Definicija. Ovakva struktura podataka dozvoljava pisanje algoritama koji će za pojedini podatak pronaći sve njegove pretke i svaki korijen iz kojeg je potekao ovaj podatak. Ukoliko se u grafu Kolegiji nalaze nizovi podataka povezanih vezama koji su duljine veće ili jednake tri podatka, ovakav pristup može biti koristan kako bi se točno definiralo koje definicije i tvrdnje su uvjet za tvrdnju na kraju niza. Još je lakše vidjeti na koje sve tvrdnje neki od korijena utječe jer tada točno gradimo podstablo iz grafa kojem je korijen baš taj podatak. Uz to možemo vidjeti i sve susjede, a to su podatci koji izravno utječu na dani podatak ili pak podatci na koje taj podatak izravno utječe, to jest ima izravnu vezu s njima.

1.2 Implementacija aplikacije

Aplikacija je implementirana kroz dva osnovna sloja, backend i frontend koji se potom još sastoje od različitih dijelova i ključnih biblioteka koji omogućavaju potpuni rad aplikacije. Aplikacija je bazirana na Representational State Transfer (REST) arhitekturi pa je tako i komunikacija između slojeva postignuta REST servisima.
Osnovni dio backend segmenta čini *Django Rest Framework*[1] koji je optimizacija *Django Frameworka*[2], koji je pak framework MVC arhitekture za web aplikacije dostupan u *Python2* i *Python3* [3] programskim jezicima. Frontend sloj aplikacije je baziran oko funkcionalnosti *Cytoscape.js*[4] biblioteke koja omogućava vizualizaciju grafova, a čije su funkcionalnosti integrirane sa sučeljem koje je razvijeno korištenjem *Angular 6*[5] frameworka. Dva ključna programska jezika koja su korištena su Python3 za Django Rest Framework te *Typescript*[6] za Angular 6. Za verzioniranje i spremanje originalnog koda je korišten *git*[7] i to u kombinaciji s *Bitbucketom*[8].

1.2.1 Backend - poslužiteljski sloj

1.2.1.1 Operativni sustav

Za potrebe razvoja i testiranja aplikacije korišten je Linux Mint 18 te Linux Mint 19 koji su desktop verzije operativnog sustava bazirane na Ubuntu 16.04 LTS i Ubuntu 18.04 LTS distribucijama operativnog sustava Linux. Tijekom procesa razvoja, aplikacija je razvijana tako da se koristio osnovni i zadani korisnik operativnog sustava, a aplikacija je pokretana korištenjem razvojnih poslužitelja dostupnih kroz oba frameworka. Za potrebe postavljanja aplikacije, korištene su Ubuntu 16.04 LTS i Ubuntu 18.04 LTS verzije operativnog sustava namijenjene radu na poslužiteljima (serverima). Prilikom postavljanja aplikacije u produkcijsku okolinu, kreiran je posebni korisnik za pokretanje i posluživanje web aplikacije koji je imao posebno prilagođena prava.

1.2.1.2 Baza podataka

Prilikom razvoja i testiranja aplikacije korištene su dvije vrste baza podataka. Prilikom inicijalne faze pripremanja modela podataka i češćeg kreiranja novih podataka, aplika-

[1] https://www.django-rest-framework.org/
[2] https://www.djangoproject.com/
[3] https://www.python.org/
[4] http://js.cytoscape.org/
[5] https://angular.io/
[6] https://www.typescriptlang.org/
[7] https://en.wikipedia.org/wiki/Git
[8] https://bitbucket.org/product/

cija je razvijana na SQLite bazi podataka, koja dolazi uz sam DRF. Prilikom kreiranja i izvršavanja migracija podataka nad bazom sam framework bi kreirao datoteku za SQLite bazu podataka. Ovo je praktično iz razloga što se može lako koristiti i manipulirati s više različitih baza podataka koje u sebi sadrže različite podatke.
U kasnijim fazama razvoja, te pogotovo kroz testiranje i pripremu za postavljanje aplikacije na poslužitelj kako bi bila dostupna kao prava web aplikacija, korištena je PostgreSQL 10 baza podataka. Kod korištenja PostgreSQL baze podataka potrebno je podesiti dvije osnovne stavke kako bi backend segment mogao ispravno komunicirati s bazom te je koristiti. Potrebno je definirati korisnika, a to se može napraviti izravno iz linux terminala korištenjem postgres korisnika na sljedeći način korištenjem komande *createuser*[9]:

```
$ createuser -d -U postgres -R -S -P <imekorisnika>
```

Zastavice redom znače:

1. -d – označava da je korisniku dozvoljeno kreiranje novih baza unutar PostgreSQL 10 aplikacije

2. -U – ime korisnika pod kojim će se spojiti (ne označava ime korisnika kojeg kreiramo)

3. -R – novom korisniku nije dozvoljeno da kreira nove korisnike za aplikaciju (ovo je zbog sigurnosti, kako se u slučaju kompromitiranja korisnika, ne bi mogla kompromitirati cijela aplikacija PostgreSQL)

4. -S – novi korisnik neće biti super korisnik

5. -P – ova zastavica omogućava interaktivno kreiranje lozinke za novog korisnika, a to je poželjno ako se želi koristiti autorizacija korisnika

Baza nad PostgreSQL 10 se kreira korištenjem komande *createdb*[10], a također se kao postgres korisnik može koristiti izravno iz linux terminala bez potrebe da se ulazi u samu PostgreSQL 10 aplikaciju:

```
$ createdb -O <imekorisnika> <imebaze>
```

Pritom zastavica -O označava da će djangouser biti vlasnik baze djangodb. Ovime su stvoreni uvjeti kako bi se aplikacija mogla ispravno spojiti na bazu podataka u kojoj će čuvati sve potrebne podatke.

[9] https://www.postgresql.org/docs/10/app-createuser.html
[10] https://www.postgresql.org/docs/10/tutorial-createdb.html

1.2.1.3 Django Rest Framework

Za razvoj modela podataka i aplikacijske logike u backend sloju je korišten Django Rest Framework koji predstavlja optimizaciju Django Frameworka prilagođenu za pisanje backend logike za web aplikacije koje su usmjerene tome da funkcioniraju kao REST servisi. Ovakav pristup je odabran baš iz razloga što se logiku backend sloja željelo razdvojiti od logike korisničkog sučelja. To omogućava da se skupom podataka koji je spremljen u backendu može koristiti više različitih korisničkih sučelja.
Za pokretanje i postavljanje DRF-a je korišten *virtualenv*[11] koji omogućava kreiranje posebne Python3 okoline s kojom je lakše upravljati te je na njoj lakše ispraviti pojedine greške u odnosu na globalnu instalaciju Python3 programskog jezika. Virtualenv se može instalirati pomoću package managera dostupnog na operativnim sustavima baziranima na Ubuntu 16.04 i Ubuntu 18.04.
Nakon kreiranja i pokretanja virtualne okoline za rad na projektu, potrebno je instalirati osnovne biblioteke za kreiranje projekta a to su paketi django i djangorestframework. Cijeli popis paketa i njihovih verzija potrebnih za ispravan rad aplikacije je spremljen u datoteci koja se nalazi u direktoriju projekta. Biblioteke se iz liste dostupne u datoteci reqs.txt instaliraju u virtualnu okolinu pomoću komande:

```
$ pip install -r reqs.txt
```

Novi django projekt se kreira pomoću komande:

```
$ django-admin startproject <imeprojekta>
```

Nakon što django-admin pripremi strukturu direktorija i datoteka za ovaj projekt (*scaffold*[12]), može se kreirati i zasebnu aplikaciju unutar projekta. Za potrebe ovog projekta su korištene dvije aplikacije, a to su userapp i graphbuilder koje se kreiraju pomoću komande: django-admin startapp ¡imeaplikacije¿

```
$ django-admin startapp <imeaplikacije>
```

Uz navedeno, potrebno je napraviti i osnovne postavke aplikacije koje se definiraju u datoteci settings.py. Prethodno su izdvojena dva najbitnija segmenta postavki aplikacije, a to su `INSTALLED_APPS` u kojem su definirane sve instalirane aplikacije pa tako i nove dvije koje su kreirane, te DATABASES u kojem su definirane postavke za pristup bazi podataka.

[11] https://virtualenv.pypa.io/en/latest/
[12] https://en.wikipedia.org/wiki/Scaffold_(programming)

```
  INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'django_filters',
    'rest_framework',
    'rest_framework.authtoken',
    'graphbuilder.apps.GraphbuilderConfig',
    'userapp.apps.UserappConfig',
    'wikiapp.apps.WikiappConfig',
    'corsheaders',
]

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'djangodb',
        'USER': 'djangouser',
        'PASSWORD': 'django_user_pass',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
}
```

1.2.1.3.1 Userapp

Userapp je zasebna aplikacije u kojoj je implementirana korisnička logika za potrebe rada aplikacije. Izdvajanjem korisničke logike u zasebnu aplikaciju, postiže se veća modularnost te se budućim zasebnim aplikacijama unutar projekta olakšava pristup korisničkom modelu. *Struktura direktorija*[13] i datoteka userapp je ista kao i kod svih ostalih aplikacija unutar DRF projekta Backend, a koju generira django-admin. Zbog nekih posebnosti, struktura aplikacija unutar projekta je objašnjena kroz ovu aplikaciju. Za potrebe ove aplikacije je proširen model User dostupan u django.contrib.auth.models tako da je kreirana nova klasa UserProfile.

[13] https://docs.djangoproject.com/en/2.2/intro/tutorial01/#creating-a-project

```
from django.db import models
from django.contrib.auth.models import User
from django.db.models.signals import post_save
from django.dispatch import receiver
from rest_framework.authtoken.models import Token

class UserProfile(models.Model):
    """Profile Model. Has one2one connection with original django User model. Used to extend User model functionality"""
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    bio = models.TextField(max_length=500, blank=True)
    location = models.CharField(max_length=50, blank=True)
    birth_date = models.DateField(null=True, blank=True)

    def __str__(self):
        return "{} {}".format(str(self.user.first_name), str(self.user.last_name))

    class Meta:
        ordering = ('id',)

@receiver(post_save, sender=User)
def create_auth_token(sender, instance=None, created=False, **kwargs):
    if created:
        Token.objects.create(user=instance)
```

Klasa UserProfile ima vezu prema korisniku koja je vrste jedan prema jedan, što znači da za svaki korisnički profil postoji jedinstveni korisnik koji postoji u aplikaciji. Kako je User klasa sa svim svojim funkcionalnostima implementirana unutar Django Frameworka, na ovaj način se može koristiti sve ono što već postoji u frameworku, a to olakšava proces kreiranja korisnika, prijave u sustav te njihovog povezivanja s resursima koje kreiraju i stvaraju. S druge strane UserProfile klasa ima dodatna polja koja pobliže opisuju korisnika, a koja ne postoje na User klasi. Osim definicije novih atributa klase, dodatno se kreiraju nove klase i metode:

1. `__str__` – radi se override standardne metode `__str__` u Python3 te se specificira reprezentacija same klase kod poziva funkcije print()
2. Meta – u klasi Meta definira se redoslijed po kojem će se raditi raspoređivanje objekata u bazi

Korištenjem dekoratora na razini aplikacije postižu se posebne funkcionalnosti metoda, a podatci koje vraćaju u ovakvim situacijama se ponašaju kao atributi klase. Za potrebe kreiranja tokena kojim se rješava pitanje autorizacije i autentifikacije korisnika koristi se

dekorator receiver. U metodi `create_auth_token` kreira se autorizacijski token za korisnika, koji služi u komunikaciji između korisničkog sučelja i backenda. Token se koristi za autorizaciju korisnika kod kreiranja, čitanja, uređivanja i brisanja podataka (*CRUD* [14]). Ukoliko korisnik nije vlasnik nekog od resursa, a to se pomoću tokena može jednostavno ustanoviti, neće moći izvršavati akcije nad podatcima.

Nakon definicije modela podataka potrebno je definirati *serializere*[15] unutar aplikacije. Oni se koriste za pretvaranje *querysetova*[16] , kompleksnih skupova podataka koji dolaze iz baze podataka, u tipove podataka koji su prirodni za Python3.
Userapp aplikacija ima dva serializera, jedan je za klasu User, a drugi je za klasu UserProfile.

Klasa UserSerializer je serializer za klasu User. Radi mapiranje modela u *JSON*[17] format. U UserSerializer-u se definiraju četiri dodatna polja koja nisu prisutna u originalnom User modelu. To su polja projects, graphs, nodes, edges i ona su definirana u modelima iz aplikacije graphbuilder, u kojoj je implementirana aplikacijska logika. Svako od tih polja je definirano kao PrimaryKeyRelatedField. Na taj način je postignuto to da je svaki korisnik vlasnik svojih resursa. U klasi Meta je definirano na koju klasu se točno odnosi ovaj serializer i koji će se atributi iz definicije klase pojaviti kao polja u serializeru. Ukoliko se neko od polja ne definira u atributu fields klase Meta, tada taj podatak neće biti moguće dohvatiti pomoću REST-a.

Druga klasa u serializeru koju je potrebno kreirati je UserProfileSerializer. Definirana je nakon UserSerializera zato jer je on potreban kako bi se definirao atribut user. Kao i u prethodnoj klasi, definirana je klasa Meta koja prima referencu na klasu na koju se klasa UserProfileSerializer odnosi, a to je u ovom slučaju klasa UserProfile. Klasa Meta ima i atribut fields u kojem se definiraju sva polja koja će se biti dostupna u JSON formatu. Iznimka je atribut user koji predstavlja cijelu klasu User, dakle ne običan atribut, nego cijeli objekt.

U ovom primjeru se pojavljuje jedina iznimka kod definiranja serializera za jednu klasu, a to je definirana metoda create. U ostalim slučajevima Django i DRF, koji je baziran na njemu, nude generička rješenja za serijalizaciju objekata koja se stalno koriste. Budući su klase User i UserProfile vezane jedan na jedan, prilikom kreiranja instance jedne klase potrebno je definirati i instancu druge klase. Iz tog razloga je definirana i metoda create,

[14] https://en.wikipedia.org/wiki/Create,_read,_update_and_delete
[15] https://www.django-rest-framework.org/api-guide/serializers/
[16] https://docs.djangoproject.com/en/2.2/topics/db/queries/
[17] https://www.json.org/

koja je override generičke metode create iz samog frameworka, a dostupna je kroz serializere. U trenutku kreiranja instance klase UserProfile nastaje i instanca klase User i njihova jedinstvena veza.

```
from rest_framework import serializers, status
from userapp.models import UserProfile
from django.contrib.auth.models import User
from graphbuilder.models import Project, Graph, Node, Edge

class UserSerializer(serializers.ModelSerializer):

    """Serializers to map user model instance itno JSON format."""

    projects = serializers.PrimaryKeyRelatedField(many = True, queryset = Project.objects.all())
    graphs = serializers.PrimaryKeyRelatedField(many = True, queryset = Graph.objects.all())
    nodes = serializers.PrimaryKeyRelatedField(many = True, queryset = Node.objects.all())
    edges = serializers.PrimaryKeyRelatedField(many = True, queryset = Edge.objects.all())

    class Meta:
        model = User
        fields = ('id','username', 'first_name', 'last_name', 'email', 'projects','graphs', 'nodes', 'edges')



class UserProfileSerializer(serializers.ModelSerializer):

    """Serializers to map userprofile model instance into JSON format.
    """

    user = UserSerializer(required=True)

    class Meta:
        model = UserProfile
        fields = ('id', 'user', 'bio', 'location', 'birth_date',)


    def create(self, validated_data):
        """
        Overriding the default create method of the Model serializer.
        :param validated_data: data containing all the details of student
```

```
36          :return: returns a successfully created student record
37          """
38          user_data = validated_data.pop('user')
39          user = UserSerializer.create(UserSerializer(), validated_data=
    user_data)
40          userprofile, created = UserProfile.objects.update_or_create(
41                                  user=user,
42                                  bio=validated_data.pop('bio'),
43                                  location=validated_data.pop('location'),
44                                  birth_date=validated_data.pop('birth_date')
    ,)
45          return userprofile
```

Sljedeći korak kod izrade aplikacije za korisnika je definiranje *view*[18] metoda. Pomoću njih se definiraju skupovi podataka koji će biti dostupni na pojedinim putanjama REST aplikacije, a isto tako se definiraju i skupovi podataka koje aplikacija može primati na tim putanjama te ih kreirati i spremati u bazu.

```
1  from userapp.models import UserProfile
2  from userapp.serializers import UserProfileSerializer
3  from rest_framework import generics, filters, permissions, parsers
4  from rest_framework.authtoken.views import ObtainAuthToken
5  from rest_framework.authtoken.models import Token
6  from rest_framework.response import Response
7
8  class UserProfileList(generics.ListCreateAPIView):
9      parser_classes = (parsers.MultiPartParser, )
10     queryset = UserProfile.objects.all()
11     serializer_class = UserProfileSerializer
12     filter_backends = (filters.SearchFilter, )
13     search_fields = ('username', 'first_name', 'last_name')
14
15 class UserProfileDetail(generics.RetrieveUpdateAPIView):
16     parser_classes = (parsers.MultiPartParser, )
17     queryset = UserProfile.objects.all()
18     serializer_class = UserProfileSerializer
19
20 class CustomObtainAuthToken(ObtainAuthToken):
21     def post(self, request, *args, **kwargs):
22         response = super(CustomObtainAuthToken, self).post(request, *
    args, **kwargs)
23         token = Token.objects.get(key=response.data['token'])
24         return Response({'token': token.key, 'id': token.user_id})
```

[18] https://www.django-rest-framework.org/api-guide/viewsets/

Definirane su tri klase, od kojih su prve dvije izravno vezane za klasu UserProfile, dok je treća indirektno vezana za tu klasu. Ovdje, a isto je i s većinom view klasa u ostatku projekta, se prvenstveno koristi generics iz `rest_frameworka`. *Generics*[19] olakšava kreiranje klasa pomoću kojih definiramo viewove.
Klasa UserProfileList nasljeđivanjem ListCreateAPIView omogućava odgovor na dvije HTTP metode GET i POST. GET pozivom se dobije lista svih UserProfile objekata. Pri POST metodi je potrebno koristiti odgovarajuće formatiran set podataka koji će omogućiti kreiranje novog UserProfile-a. Klasa UserProfileDetail nasljeđuje RetrieveUpdateAPIView koji pomoću metoda iz genericsa stvara uvjete za odgovor na *HTTP metode*[20] GET, PUT i PATCH. Prva vraća instancu klase UserProfile s danim jedinstvenim id-om. Druge dvije metode rade ažuriranje instance klase UserProfile s danim jedinstvenim id-om.

Klasa CustomObtainAuthToken nasljeđuje klasu ObtainAuthToken. Nasljeđivanje i proširivanje funkcionalnosti ove klase je bilo potrebno iz razloga što se uz token prema korisničkom sučelju još trebalo slati i sam jedinstveni id korisnika kako bi se moglo dohvatiti korisnički profil te podatke iz aplikacije graphbuilder koji pripadaju tom korisniku, a to je vidljivo u definiciji povratnih podataka iz metode post.

Nakon definicije modela podataka kroz klase, ostvarivanja stvaranja uvjeta za serijalizaciju podataka iz baze u format koji je prirodan za obradu u Python3 programskom jeziku te potom i pripreme podataka za slanje u JSON formatu koji je prihvatljiv arhitekturi REST-a, potrebno je definirati putanje na kojima će bit dostupne klase definirane u views.py. To se definira u datoteci urls.py.

```
1  from django.conf.urls import url, include
2  from rest_framework.urlpatterns import format_suffix_patterns
3  from userapp import views
4  from rest_framework.authtoken.views import obtain_auth_token
5
6  urlpatterns = [
7      url(r'^profile/users/$', views.UserProfileList.as_view(), name='user
       -list'),
8      url(r'^profile/users/(?P<pk>[0-9]+)/$', views.UserProfileDetail.
       as_view(), name='user-detail'),
9      url(r'^profile/api_auth/', include('rest_framework.urls')),
10     url(r'^profile/get/token/', views.CustomObtainAuthToken.as_view())
11 ]
12 urlpatterns = format_suffix_patterns(urlpatterns)
```

[19] https://www.django-rest-framework.org/api-guide/generic-views/
[20] https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

Osnovni dio putanje je definiran ovisno o tome radi li se o aplikaciji postavljenoj na poslužitelj ili se radi o razvojnom serveru dostupnom unutar Django frameworka. U oba slučaja ostatak putanje izgleda kako je navedeno u regeximа na početku definicije url-a. Na putanjama na kojima nema dodatnih argumenata, registriramo klase iz viewova koje nasljeđuju ListCreateAPIView, dok na putanjama koje dodatno primaju i argument tipa PrimaryKey, to jest ID, registriramo viewove koji nasljeđuju RetrieveUpdateAPIView. Treći argument je name koji olakšava prepoznavanje i korištenje putanja kod testiranja aplikacije korisničkom sučelju koje je dostupno preko DRF testnog poslužitelja.
Postoje još dvije putanje. Prva putanja omogućava autorizaciju korisnika, dok druga putanja vraća korisnički token i id, a kako je prethodno navedeno to je potrebno za funkcionalnosti na korisničkom sučelju. Kreirane url putanje je potrebno registrirati na mjestu koje će sadržavati sve putanje u cijelom projektu. Radi se o url.py datoteci koja se nalazi u glavnom direktoriju backend. Registracijom putanja u tom djelu aplikacije omogućava se globalna dostupnost svih kreiranih putanja. Dovoljno je za svaku kreiranu aplikaciju u projektu napraviti registraciju njenih putanja na sljedeći način kako bi one bile dostupne.

```
"""backend URL Configuration

The ‘urlpatterns‘ list routes URLs to views. For more information please see:
    https://docs.djangoproject.com/en/2.1/topics/http/urls/
Examples:
Function views
    1. Add an import:  from my_app import views
    2. Add a URL to urlpatterns:  path('', views.home, name='home')
Class-based views
    1. Add an import:  from other_app.views import Home
    2. Add a URL to urlpatterns:  path('', Home.as_view(), name='home')
Including another URLconf
    1. Import the include() function: from django.urls import include, path
    2. Add a URL to urlpatterns:  path('blog/', include('blog.urls'))
"""
from django.contrib import admin
from django.urls import path
from django.conf.urls import url, include

urlpatterns = [
    path('admin/', admin.site.urls),
    url(r'^', include('graphbuilder.urls')),
    url(r'^', include('userapp.urls')),
]
```

Osim userapp i graphbuilder tu se nalazi i *admin*[21]. Radi se o dodatnom korisničkom sučelju koje omogućava administraciju aplikacije. Kako bi kreirane klase bile dostupne na tom sučelju, potrebno ih je unutar datoteke admin.py koja je dostupna u svakoj aplikaciji.

```
from django.contrib import admin
from userapp.models import UserProfile

admin.site.register(UserProfile)

# Register your models here.
```

Nakon ovog je moguće napravit prvo pokretanje cijelog projekta korištenjem sljedećeg niza komandi.

```
$ python3 manage.py makemigrations <imeaplikacije>
```

Manage.py je datoteka koja omogućava izvršavanje cijelog niza komandi koje ubrzavaju i olakšavaju razvoj i testiranje projekta. Zastavica makemigrations kreira *migracije*[22] koje definiraju kako će se strukturirati podatci u bazi podataka na temelju klasa definiranih u python datotekama unutar aplikacija. Ovaj poziv kreira posebni direktorij migrations unutar svake aplikacije u kojeg se spremaju podatci o svim migracijama. Sljedeći poziv će kreirati tablice unutar baze podataka koje će odgovarati onome što je definirano kroz python kod:

```
$ python3 manage.py migrate
```

Za pristupanje admin sučelju je potrebno kreirati superusera. Ovo se opet radi pomoću manage.py koja omogućava interaktivno kreiranje korisnika sljedećom komandom:

```
$ python3 manage.py createsuperuser
```

Potrebno je kreirati korisničko ime te zaporku kojom se potom može pristupiti na admin sučelje, ali isto tako i na sučelje koje pruža DRF za testiranje REST aplikacije. Za to je još potrebno pokrenuti testni poslužitelj pomoću sljedeće komande:

```
$ python3 manage.py runserver
```

Admin korisničko sučelje će biti dostupno na http://127.0.0.1:8000/admin/, dok će samo sučelje za testiranje REST aplikacije bit dostupno na http://127.0.0.1:8000/. Ovime je završen opis osnovnog procesa razvoja jedne aplikacije korištenjem Djanga i Django Rest Frameworka. Sljedeći dio govori više o aplikaciji graphbuilder koja omogućava vizualizaciju skupa podataka koji ima hijerarhijsku ili relacijsku strukturu.

[21] https://docs.djangoproject.com/en/2.2/ref/contrib/admin/
[22] https://docs.djangoproject.com/en/2.2/topics/migrations/

1.2.1.3.2 Graphbuilder

Metode izrade aplikacije korištenjem Django Rest Frameworka su objašnjenje u prethodnom djelu. Ovaj dio je posvećen strukturi podataka koja omogućava vizualizaciju u obliku grafa ili stabla. Jedina strukturalna razlika aplikacije graphbuilder u odnosu na aplikaciju userapp je u definiciji direktorija models i methods. U direktoriju models su u zasebnim datotekama definirane pojedine klase koje se navode niže, a u direktoriju methods su definirane odgovarajuće metode, specifične za pojedinu klasu, ali koje ne spadaju u kategoriju CRUD.
U graphbuilder aplikaciji postoji pet osnovnih klasa:

1. Project – definira se pojedini projekt koji se sastoji od jednog ili više grafova. Projekt je prvenstveno vezan na određeni skup podataka ili posebni korisnički slučaj koji se istražuje kroz više grafova

2. Graph – radi se o vizualizaciji skupa podataka kroz koju korisnik želi postići jedinstveni kontekst. Stavljanjem podataka u jedinstveni kontekst, dobiva se posebno tumačenje skupa podataka.

3. Node – svaki node predstavlja jednu jedinstvenu informaciju ili podatak koji korisnik želi prikazati u grafu.

4. Edge – veza između dvije informacije unutar jednog grafa

5. Category – kategorija ili vrsta pojedine informacije. Prilikom vizualizacije olakšava distinkciju između različitih vrsta podataka.

Prva klasa koja je definirana u aplikaciji graphbuilder je Project. Kao i svaki projekt, sastoji se od imena, sadržaja i vlasnika. Postoje dodatna dva polja koja koristimo u svim klasama, a to su `date_created` i `date_modified` koje koristimo za audit, kako bismo znali kada je neka instanca klase nastala i kada je modificirana. Nije potrebno pratiti tko je napravio izmjenu, budući samo vlasnik instance može mijenjati instancu.
Dodatno je definirana i klasa Meta koja ima atribut ordering. Pomoću ordering atributa se zadaje po kojim atributima će se sortirati skup objekata. U slučaju klase Project prvo će se sortirati po atributu id i to uzlazno, a potom po atributu name.
Nakon klase Project, definirana je klasa Graph. Svaka instanca klase Graph ima jedinstvenu vezu prema nekoj instanci klase Project. Ukratko, Graph može pripadati samo jednom Projectu. Dodatni atributi koje Graph ima su:

1. description – proširenje atributa content kako bi se bolje opisalo što se točno proučavalo u pojedinom Graphu.

2. layout – kako bi se moglo spremiti podatke vezane za layout vezan za Cytoscape.js biblioteku.

Klasa Category ima sličnu strukturu kao i prethodne dvije klase, uz dva dodatna bitna atributa, a to su color i shape. Svaka informacija, koju predstavlja jedan čvor u grafu, ima zasebnu kategoriju, a svaka kategorija ima posebnu boju i oblik. Na taj način se prilikom vizualizacije jednostavno postiže i uočava povezanost različite vrste podataka i grupiranje sličnih vrsta podataka.

1. shape – ima ponuđen skup oblika koje biblioteka Cytoscape.js može ispravno prikazati.

2. color – može biti bilo koja boja i njen zapis je u heksadecimalnoj bazi.

Preostale dvije klase definiraju graf koji želimo vizualizirati, a to su Node i Edge. Svaki Node predstavlja jednu informaciju, ali njegova struktura kao klase i kao objekta u bazi je i dalje slična prethodnim klasama. Osim standardnih atributa, klasa Node ima još i sljedeće atribute:

1. graphs – predstavlja skup grafova u kojima se neki Node može pojaviti

2. category – predstavlja kategoriju kojoj neki Node pripada. Kao što je navedeno u Primjeru, to može biti Definicija, Lema, Korolar itd.

Posljednja klasa koja se definira je Edge. Opet imamo sličnu strukturu kao i kod prethodnih klasa, s tim da klasa Edge ima specifična polja:

1. `node_start` – jedinstvena instanca klase Node koja označava početni vrh brida

2. `node_end` – jedinstvena instanca klase Node koja označava završni vrh brida

3. graph – jedinstvena instanca klase Graph u kojoj se brid pojavljuje

Na ovaj način je zaokružena cjelokupna struktura podataka u aplikaciji koja omogućava spremanje i dohvaćanje podataka te stvaranje struktura koje su potrebne za uspješnu vizualizaciju.

```
from django.db import models
from django.db.models.signals import post_save
from django.contrib.auth.models import  User
from rest_framework.authtoken.models import Token
from django.dispatch import receiver

class Project(models.Model):

    name = models.CharField(max_length=250, blank=False)
    content = models.TextField()
    date_created = models.DateTimeField(auto_now_add=True)
    date_modified = models.DateTimeField(auto_now=True)
    owner = models.ForeignKey('auth.User', related_name='projects',
   on_delete=models.CASCADE)

    def __unicode__(self):
        return self.name

    def __str__(self):
        return "{}".format(str(self.name))

    class Meta:
        ordering = ('id', 'name',)
```

```
class Graph(models.Model):

    name = models.CharField(max_length=250, blank=False)
    content = models.TextField()
    description = models.TextField()
    date_created = models.DateTimeField(auto_now_add=True)
    date_modified = models.DateTimeField(auto_now=True)
    project = models.ForeignKey(Project, related_name='graphs',
   on_delete=models.CASCADE)
    layout =  JSONField(default=dict)
    owner = models.ForeignKey('auth.User', related_name='graphs',
   on_delete=models.CASCADE)

    def __unicode__(self):
        return self.name

    def __str__(self):
        return "{}".format(str(self.name))

    class Meta:
        ordering = ('id', 'name',)
```

```
class Category(models.Model):
    """Class for describing the category of the Node"""

    ELLIPSE = 'ellipse'
    TRIANGLE = 'triangle'
    RECTANGLE = 'rectangle'
    ROUND_RECTANGLE = 'round-rectangle'
    BOTTOM_ROUND_RECTANGLE = 'bottom-round-rectangle'
    CUT_RECTANGLE = 'cut-rectangle'
    BARREL = 'barrel'
    RHOMBOID = 'rhomboid'
    DIAMOND = 'diamond'
    PENTAGON = 'pentagon'
    HEXAGON = 'hexagon'
    CONCAVE_HEXAGON = 'concave-hexagon'
    HEPTAGON = 'heptagon'
    OCTAGON = 'octagon'
    STAR = 'star'
    TAG = 'tag'
    VEE = 'vee'
    CIRCLE = 'circle'


    SHAPE_CHOICES = (
        ( ELLIPSE, 'Elipse'),
        ( TRIANGLE, 'Triangle' ),
        ( RECTANGLE, 'Rectangle'),
        ( CIRCLE, 'Circle'),
        ( ROUND_RECTANGLE, 'Round Rectangle', ),
        ( BOTTOM_ROUND_RECTANGLE, 'Bottom Round Rectangle'),
        ( CUT_RECTANGLE, 'Cut Rectangle', ),
        ( BARREL, 'Barrel'),
        ( RHOMBOID, 'Rhomboid', ),
        ( DIAMOND,'Diamond'),
        ( PENTAGON,'Pentagon' ),
        ( HEXAGON, 'Hexagon'),
        ( CONCAVE_HEXAGON, 'Concave Hexagon'),
        ( HEPTAGON, 'Heptagon'),
        ( OCTAGON, 'Octagon'),
        ( STAR, 'Star'),
        ( TAG, 'Tag'),
        ( VEE, 'Vee'),
    )

    name = models.CharField(max_length=250, blank=False)
    description = models.TextField()
    color = models.CharField(max_length=20)
```

```
48      shape = models.CharField(max_length=35, choices=SHAPE_CHOICES,
        default=RECTANGLE)
49      project = models.ManyToManyField(Project, related_name="categories")
50      date_created = models.DateTimeField(auto_now_add=True)
51      date_modified = models.DateTimeField(auto_now=True)
52
53      def __unicode__(self):
54          return self.name
55
56      def __str__(self):
57          return "{}".format(str(self.name))
58      class Meta:
59          ordering = ('id', 'name', )
```

```
1  class Node(models.Model):
2
3      name = models.CharField(max_length=250, blank=False)
4      content = models.TextField(blank=True)
5      description = models.TextField(blank=True)
6      date_created = models.DateTimeField(auto_now_add=True)
7      date_modified = models.DateTimeField(auto_now=True)
8      graphs = models.ManyToManyField(Graph, related_name='graph_nodes')
9      owner = models.ForeignKey('auth.User', related_name='nodes',
       on_delete=models.CASCADE)
10     category = models.ForeignKey(Category, related_name='nodes',
       on_delete=models.PROTECT)
11
12     def __unicode__(self):
13         return self.name
14
15     def __str__(self):
16         return "{}".format(str(self.name))
17
18     class Meta:
19         ordering = ('id', 'name',)
```

```
1  class Edge(models.Model):
2
3      name = models.CharField(max_length=250, blank=False)
4      content = models.TextField(blank=True)
5      node_start = models.ForeignKey(Node,  related_name='edge_start',
       null=False, on_delete=models.CASCADE)
6      node_end = models.ForeignKey(Node, related_name='edge_end', null=
       False, on_delete=models.CASCADE)
7      graph = models.ForeignKey(Graph, related_name='graph_edges', null=
       False, on_delete=models.CASCADE)
```

```
8     date_created = models.DateTimeField(auto_now_add=True)
9     date_modified = models.DateField(auto_now=True)
10    owner = models.ForeignKey('auth.User', related_name='edges',
    on_delete=models.CASCADE)
11
12    def __unicode__(self):
13        return self.name
14
15    def __str__(self):
16        return "{}".format(str(self.name))
17
18    class Meta:
19        ordering = ('id', 'name',)
```

Dio aplikacijske logike potreban da bi se ostvarile funkcionalnosti zamišljene u ovakvoj aplikaciji je ostvaren kroz metode koje su implementirane u modulu methods. Za svaki objekt su implementirane specifične metode potrebne za određeni prikaz na korisničkom sučelju.
U datoteci projectMethods.py su implementirane dvije metode:

1. projectList(`user_id`) – prima argument `user_id` te vraća listu svih projekata kojima je vlasnik korisnik s danim id-om.

2. projectStats(`project_id`) – metoda prima `project_id` kao argument te vraća statistiku za dashboard prilikom otvaranja pojedinog projekta, prvenstveno broj Nodova i Edgeva

U datoteci grapMethods.py definirana je samo jedna, ali ujedno i najbitnija metoda cijelog Backenda.

1. graphData(`graph_id`) – prima argument `graph_id`. Pretražuje bazu podataka i pronalazi Nodove i Edgeve koji su povezani s danim Graphom. Nakon toga iz njih priprema strukturu kakvu Cytoscape.js očekuje i koja se može vizualizirati na korisničkom sučelju. Podatci se spremaju u dictionary strukturu koja se šalje kao odgovor prilikom poziva odgovarajuće putanje. Pri tom prikuplja i dodatne podatke za svaki Node, a to su podatci vezani za:

   a) shape – pridružuje se oblik koji ima kategorija s kojom je povezan podatak

   b) color – pridružuje boju kategorije koju ima dana kategorija

   c) parents – zapisuje prvu liniju predaka za svaki Node. Predci Nodea su svi oni koji se nalaze na poziciji `node_start` u svim Edgevima u kojima se naš Node nalazi na poziciji `node_end`

Datoteka categoryMethods.py sadrži također samo jednu metodu:

1. listCategory(`project_id`) – metoda prima `project_id` te vraća listu instanci klase Category koja se koristi na korisničkom sučelju za pregled statistika danog Projecta

```
from graphbuilder.models import Project, Graph, Node, Edge, Category

def projectList(user_id):
    user = User.objects.get(pk=user_id)
    projects = Project.objects.filter(owner=user)
    return projects

def projectStats(project_id):

    nodes = []
    edges = []

    for graph_id in Graph.objects.filter(project= project_id).
    values_list('id', flat=True):
        eTemp = Edge.objects.filter(graph=graph_id).values_list('id',
    flat=True)
        nTemp = Node.objects.filter(graphs=graph_id).values_list('id',
    flat=True)

        edges.extend([edge for edge in eTemp])
        nodes.extend([node for node in nTemp])

    nodes = set(nodes)
    edges = set(edges)

    res = {
        'nodes': len(nodes),
        'edges': len(edges)
    }

    return res
```

```
from graphbuilder.models import Graph, Node, Edge
import pprint

def graphData(graph_id):

    pp = pprint.PrettyPrinter(indent=4)

    res = {}

```

```
10      graph = Graph.objects.get(pk=graph_id)
11
12      edgesData = graph.graph_edges.all()
13      nodesData = graph.graph_nodes.all()
14
15      edges = []
16      nodes = []
17
18      parents = {}
19
20      for edge in edgesData:
21          edges.append({'data':{'id': str(edge.id) + "_" + str(edge.
    node_start.id) +  "_" +  str(edge.node_end.id) , 'name':edge.name, '
    source': edge.node_start.id, 'target': edge.node_end.id}})
22          if edge.node_end.id in parents.keys():
23              parents[edge.node_end.id].add(str(edge.node_start.id))
24          else:
25              parents.update({edge.node_end.id: set()})
26              parents[edge.node_end.id].add(str(edge.node_start.id))
27
28
29      for node in nodesData:
30          if node.category:
31              if node.id in parents.keys():
32
33                  if str(node.id) in parents[node.id]:
34                      print(parents[node.id])
35                      parents[node.id].remove(str(node.id))
36                  nodes.append({'data':{'id': str(node.id), 'name':node.
    name, 'shape': node.category.shape, 'parents': parents[node.id], '
    color': node.category.color }})
37              else:
38                  nodes.append({'data':{'id': str(node.id), 'name':node.
    name, 'shape': node.category.shape, 'parents': set(), 'color': node.
    category.color }})
39          else:
40              if str(node.id) in parents[node.id]:
41                  (parents[node.id]).remove(str(node.id))
42              nodes.append({'data':{'id': str(node.id), 'name':node.name,
    'parents': parents[node.id], 'shape': 'elipse', 'color': '#d9d9d9'
    }})
43
44      res.update({
45          'id': graph.id,
46          'name': graph.name,
47          'decription': graph.description,
48          'content': graph.content,
```

```
        'layout': graph.layout,
        'cytoscape':{
            'elements':{
                'nodes': nodes,
                'edges': edges
                }}})

    return res
```

```
from graphbuilder.models import Category

def listCategory(project_id):

    categories = set(Category.objects.filter(project=project_id))
    result = []

    for cat in categories:
        #shape = '<div class=\"{}\"></div>'.format(cat.shape)
        temp = {
            'id': cat.id,
            'name': cat.name,
            'description': cat.description,
            'shape': cat.shape,
            'color': cat.color,
        }

        result.append(temp)

    return result
```

```
from rest_framework import permissions

class IsOwnerOrReadOnly(permissions.BasePermission):

    """This is the base permission to allow only the owners to change
    model instances"""

    def has_object_permission(self, request, view, obj):

        if request.method in permissions.SAFE_METHODS:
            return True

        return obj.owner == request.user
```

U aplikaciji graphbuilder postoji dodatna datoteka koja se zove permissions.py. U njoj su definirana prava pristupa za sve korisnike koji pokušavaju pristupiti podatcima iz aplika-

cije. Korisnici koji nisu vlasnici podataka, mogu ih vidjeti, međutim izmjene na podatcima mogu raditi samo vlasnici podataka.

Svi serializeri u aplikaciji graphbuilder su napravljeni po istim principima i zahtjevima kako su napravljeni i u aplikaciji userapp. Tako su definirani sljedeći serializeri:

1. ProjectSerializer
2. GraphSerializer
3. NodeSerializer
4. EdgeSerializer
5. CategorySerializer

Sve klase u serializer.py imaju dodatni atribut vezan za prava pristupa, a to je:

1. `permission_classes` – ima vrijednost (permissions.IsAuthenticatedOrReadOnly, IsOwnerOrReadOnly). Ovo odgovara pravima pristupa koja su definirana u datoteci permissions.py

U odnosu na datoteku views.py iz aplikacije userapp, postoje manje izmjene koje su vezane za metode koje su definirane u modulu methods. Ovo se odnosi na metode unutar views.py kojima prethodi dekorator `api_view`. Ovime se omogućava kreiranje view metode koja se može registrirati na putanjama i potom se tim putanjama može pristupiti kao i ostalim koje su generički definirane. Te metode odgovaraju na HTTP upite koji su odgovarajućeg tipa navedenog u dekoratoru koji im prethodi. Kako se vidi, te metode služe samo kako bi na ispravan način proslijedile podatke koje metode iz methods unaprijed pripreme. Dodatno je definirana metoda `api_root` koja omogućava da se u njenom odgovoru Response popišu sve bitne putanje. Ovo je korisno kada testiranje radi kroz sučelje dostupno nakon pokretanja testnog poslužitelja za Django. Tada se putanje aplikacije prikazuju kao linkovi na početnoj stranici te se klikom na pojedini link otvara putanja i forma za unos ili pregled podataka koja se nalazi iza nje.

```
1 from rest_framework import generics, filters, permissions
2 from rest_framework.renderers import JSONRenderer
3 from graphbuilder.permissions import IsOwnerOrReadOnly
4 from graphbuilder.methods import graphData, projectList, projectStats,
     listCategory
5 """Custom Views"""
6
7 @api_view(['GET'])
8 def usersProjects(request, user_id, format=None):
9
```

```
10      """Get all the data from this graph in order to create cytoscape
        object for rendering"""
11
12      renderer_classes = (JSONRenderer, )
13      return projectList(user_id)
14
15  @api_view(['GET'])
16  def graphCytoData(request, pk, format=None):
17
18      """Get all the data from this graph in order to create cytoscape
        object for rendering"""
19
20      renderer_classes = (JSONRenderer, )
21      return Response(graphData(pk))
22
23  @api_view(['GET'])
24  def projectStatistics(request, pk, format=None):
25      """Get the number of the nodes for one project"""
26
27      renderer_classes = (JSONRenderer, )
28      return Response(projectStats(pk))
29
30  @api_view(['GET'])
31  def categoryDashList(request, project_id, format=None):
32      renderer_classes = (JSONRenderer, )
33      return Response(listCategory(project_id))
34
35  @api_view(['GET'])
36  def api_root(request, format=None):
37      return Response({
38          'users': reverse('user-list', request=request, format=format),
39          'projects': reverse('project-list', request=request, format=
        format),
40          'graphs': reverse('graph-list', request=request, format=format),
41          'nodes': reverse('node-list', request=request, format=format),
42          'categories': reverse('category-list', request=request, format=
        format),
43          'edges': reverse('edge-list', request=request, format=format),
44      })
```

Na kraju je potrebno sve metode i klase iz views.py registrirati u urlpatterns iz datoteke urls.py. također je i te urlpatterne potrebno uvesti i registrirati u glavnoj datoteci za putanje cijelog projekta. Nakon novog kreiranja migracija, ovaj put za aplikaciju graphbuilder te migriranja podataka, backend sloj aplikacije može raditi ispravno. Bitno je napomenuti da zbog korištenja REST api arhitekture, podatke nije nužno dohvaćati kroz sučelje, kao ni kreirati i uređivati. Ovo je korisno za velike skupove podataka koji su zahtjevni za obradu,

jer se programskim rješenjem koje radi automatsku obradu velikog skupa podataka ti isti podatci mogu upisati u bazu i potom proučavati i istraživati kroz sučelje.

```
from django.conf.urls import url, include
from rest_framework.urlpatterns import format_suffix_patterns
from graphbuilder import views

urlpatterns = [
    url(r'^$', views.api_root),
    url(r'^graphapp/graphs/$', views.GraphList.as_view(), name='graph-list'),
    url(r'^graphapp/graphs/(?P<pk>[0-9]+)/$', views.GraphDetail.as_view(), name='graph-detail'),
    url(r'^graphapp/graphs/data/(?P<pk>[0-9]+)/$', views.graphCytoData, name='graph-data'),
    url(r'^graphapp/nodes/$', views.NodeList.as_view(), name='node-list'),
    url(r'^graphapp/nodes/(?P<pk>[0-9]+)/$', views.NodeDetail.as_view(), name='node-detail'),
    url(r'^graphapp/categories/$', views.CategoryList.as_view(), name='category-list'),
    url(r'^graphapp/categories/(?P<pk>[0-9]+)/$', views.CategoryDetail.as_view(), name='category-detail'),
    url(r'^graphapp/categories/dash/(?P<project_id>[0-9]+)/$', views.categoryDashList, name='category-dash-list'),
    url(r'^graphapp/edges/$', views.EdgeList.as_view(), name='edge-list'),
    url(r'^graphapp/edges/(?P<pk>[0-9]+)/$', views.EdgeDetail.as_view(), name='edge-detail'),
    url(r'^graphapp/projects/$', views.ProjectList.as_view(), name='project-list'),
    url(r'^graphapp/projects/(?P<pk>[0-9]+)/$', views.ProjectDetail.as_view(), name='project-detail'),
    url(r'^graphapp/projects/list/(?P<owner_id>[0-9]+)/$', views.ProjectUserList.as_view(), name='user-project-list'),
    url(r'^graphapp/projects/statistics/(?P<pk>[0-9]+)/$', views.projectStatistics, name='nodes-number'),

]

urlpatterns = format_suffix_patterns(urlpatterns)
```

1.2.2 Korisničko sučelje

Korisničko sučelje je razvijeno korištenjem:

1. Cytoscape.js – biblioteka koja omogućava vizualizaciju i manipulaciju grafovima, kao i određene standardne algoritme za teoriju grafova, prilagođena za implementaciju i korištenje u web aplikacijama.

2. Angular 6 – MVC framework koji je prilagođen za izradu korisničkog sučelja

3. Angular Material[23] – implementacija u Angularu Googleovog *Material Design Specificationa*[24] za razvoj korisničkog sučelja

4. Typescript – programski jezik koji je nastao iz *javascripta*[25] dodavanjem tipova varijabli (types)

1.2.2.1 Cytoscape.js

Cytoscape.js je biblioteka pisana originalno za programski jezik javascript, ali je u međuvremenu napravljena i njena verzija u typescriptu. Ova biblioteka omogućava vizualizaciju grafova u web aplikacijama tako da generira *canvas*[26] element unutar HTML strukture te na njemu crta struktura grafa.
Cytoscape.js ima specifičnu strukturu podataka koju prima pa je prema toj cijeloj strukturi i građena arhitektura aplikacije u backendu. Tu posebno dolazi do izražaja metoda graphData(`grap_id`) koja dohvaća podatke iz baze te ih prilagođava strukturi koja odgovara Cytoscapeu, a to je JSON struktura. Četiri najbitnija segmenta te strukture su:

1. container – radi se o elementu iz HTML strukture, a to je redovito element s oznakom div, u koji se smjesti vizualizacija grafa

2. elements – skup podataka koje će instanca Cytoscapea primiti i prikazati, a sastoji se od dva niza podataka:

   a) nodes – skup čvorova koji sadrži sve one podatke koje mu je dodijelila metoda iz backenda graphData(`graph_id`)

   b) edges – skup bridova nastao na isti način kao i nodes

[23] https://material.angular.io/
[24] https://material.io/design/
[25] https://www.javascript.com/
[26] https://www.w3schools.com/html/html5_canvas.asp

3. layout – označava osnovni raspored čvorova u prostoru definiranom pomoću osnovnog elementa container. Postoji više različitih layouta, a neki od njih su cose, breathdfirstsearch, circle, random itd.

4. style – JSON struktura koja omogućava dodavanje dodatnih informacija na elemente tipa node i edge, a to su boja, oblik, tekstualna oznaka i širina objekta.

Ovi elementi su dovoljni da bi Cytoscape mogao vizualizirati podatke koje imamo spremljene u bazi na backend sloju aplikacije.

Dodatna prednost biblioteke je što je graf koji nastane vizualizacijom interaktivan. U biblioteci su implementirani *event handleri*[27] koji nakon klika na pojedini element vrate taj cijeli element, bio to node ili edge. Ovo je iznimno korisno, jer se može koristiti za dohvat podataka iz baze. Naime, prilikom pripreme podataka u metodi graphData(`graph_id`) id samog Nodea iz backend sloja aplikacije je korišten i kao id u strukturi za Cytoscape. Ovo je olakšalo i ubrzalo izravni dohvat podataka iz baze, kako bi se prikazale dodatne informacije o node-u. Isto vrijedi i za edgeve koji se prikazuju u aplikaciji.

1.2.2.2 Angular 6

Angular 6 je MVC framework specijaliziran za izradu korisničkog sučelja. Kako je backend sloj aplikacije baziran na Django Rest Frameworku, za korisničko sučelje se moglo izabrati bilo koju tehnologiju ili framework koji podržavaju HTML upite. Angular je odabran zbog dobre dokumentacije i uputa koje su dostupne. Dodatno je korišten i Angular Material kako bi se olakšalo uređivanje korisničkog sučelja, jer se radi o već definiranim oblicima i paletama boja.
Za razvoj korisničkog sučelja u Angularu, dodatno su potrebni *NodeJS*[28] te *npm*[29] koji je dio samog NodeJS. NodeJS *instaliramo pomoću package managera*[30] dostupnog u operativnog sustavu Ubuntu. Osnovni alat pomoću kojeg se radi scaffold aplikacije i elementa u Angularu zove se *Angular CLI*[31] te funkcionira na sličnim principima kao i manage.py u Djangu. Pomoću CLI komande je moguće kreirati i dodatne specifične elemente unutar Angular projekta koji dodatno ubrzavaju razvoj aplikacije. Instalira ga se korištenjem npm-a pomoću sljedeće komande:

```
1  $ npm install -g @angular/cli
```

[27] http://js.cytoscape.org/#core/events
[28] https://nodejs.org/en/
[29] https://www.npmjs.com/
[30] https://github.com/nodesource/distributions/blob/master/README.md
[31] https://cli.angular.io/

Nakon instalacije CLI-a napravi se scaffold nove aplikacije i odmah instalacija typescript implementacije biblioteke Cytoscape:

```
$ ng new <imeaplikacije>
```

```
$ npm install -save @types/cytoscape
```

Ovime su zadovoljeni osnovni preduvjeti za razvoj korisničkog sučelja za aplikaciju. Korisničko sučelje se sastoji od dva osnova modula. Prvi modul je authentication čija je primarna zadaća komunikacija s aplikacijom userapp u backend sloju, dok modul graphapp komunicira s aplikacijom graphbuilder u backend sloju aplikacije.

Moduli se sastoje od tri osnovna elementa, a to su:

1. Modeli – definicije klasa koje se koriste u korisničkom sučelju. Modeli u korisničkom sučelju odgovaraju modelima u backend sloju aplikacije, s tim da na nekoliko mjesta postoje manje razlike.
2. Servisi – ovo su specijalizirane klase koje sadrže metode za komunikaciju s REST servisima na backend sloju aplikacije. Svaka od klasa definiranih u modelima ima odgovarajući servis s implementiranim svim vrstama poziva (CRUD)
3. Components – komponente su osnovni elementi za gradnju aplikacija i modula u Angularu.

Svaki od ovih segmenata modula se generira korištenjem Angular CLI-a i to jednom od sljedećih komandi:

1.
```
$ ng generate module --routing=true <putanjadolokacije>
```

2.
```
$ ng generate component <putanjadolokacije>
```

3.
```
$ ng generate   class <putanjadolokacije>
```

4.
```
$ ng generate service <putanjadolokacije>
```

Organizacija modula na strani korisničkog sučelja odražava organizaciju aplikacija u backend segmentu aplikacije. Dva osnovna modula su:

1. authentication – sadržava logiku za implementaciju i korištenje funkcionalnosti vezanih za samog korisnika. Izravno je povezana s aplikacijom userapp u backend segmentu aplikacije

2. graphapp – korisničko sučelje kojem je osnovna svrha upravljanje i vizualizacija podataka i aplikacijske logike dostupnih u backend segmentu aplikacije. Izravno je povezan s aplikacijom graphbuilder.

Struktura direktorija na korisničkom sučelju je zadana arhitekturom Angular 6 frameworka, ali je isto tako analogna logičkoj strukturi direktorija na backend segmentu, a to je posljedica toga što su i Angular i DRF bazirana na MVC arhitekturi.
Osnovna građevna komponenta svake aplikacije i modula razvijanog u Angularu je *komponenta*[32], koja se obično sastoji od četiri datoteke i sprema se u direktorij kojeg Angular CLI procesom scaffoldanja sam kreira i nazove prema komponenti koju kreiramo:

1. `<imekomponente>.component.spec.ts` – datoteka u kojoj je zadana generička definicija komponente kao elementa same aplikacije. Ovu datoteku u većini slučajeva nije potrebno mijenjati te je tako bilo i u slučaju cijele aplikacije.

2. `<imekomponente>.component.ts` – implementira *OnInit*[33] klasu dostupnu u Angularu. Ova datoteka sadržava logiku koja je vezana za cijelu komponentu. U njoj se nalaze deklaracije varijabli, objekata i funkcija koji su bitni za navedenu komponentu. Dodatno su tu još dvije metode:

   a) constructor – metoda koja služi kreiranju instance. Dodatno može primiti određene parametre i kreirati druge objekte koji su potrebni za ispravan rad navedene komponente. Argumenti koji se predaju konstruktoru su često vezani za druge komponente ili servise koji su potrebni za ispravan rad komponente

   b) ngOnInit – metoda kojom se definiraju osnovne akcije koje će komponenta poduzeti odmah nakon što je konstruirana, a prije nego li će biti prikazana. Ovo je vrlo korisno kada je potrebno dohvatiti podatke koji će se dinamički prikazati na korisničkom sučelju.

3. `<imekomponente>.component.html` – ova datoteka sadrži HTML definiciju pregleda kojeg će korisnički preglednik prikazati. Izravno je povezana s logikom koja je implementirana u datoteci ¡imekomponente¿.component.ts.

[32] https://v6.angular.io/api/core/Component
[33] https://v6.angular.io/api/core/OnInit

4. `<imekomponente>.component.css` – omogućava definiciju stila za korisničko sučelje za svaku komponentu. Budući je u ovoj aplikaciji korišten Angular Material za stiliziranje izgleda korisničkog sučelja, ova datoteka je rijetko korištena, osim u posebnim slučajevima.

Kako je u backend segmentu definicija klasa objekata izdvojena u zasebni direktorij, tako je napravljeno i u implementaciji korisničkog sučelja. Svi objekti koji postoje na backend segmentu su zrcalno definirani u odgovarajućem modulu korisničkog sučelja. Ovo je jedan od razloga zašto je korisničko sučelje rađeno u Angular 2+ frameworku, koji su svi redom pisani u Typescriptu, što olakšava povezivanje i provjeru vrste podataka koja se šalje.
Osim modela i komponenti, kod izrade aplikacije bazirane na REST servisima, bitna je i komunikacija s backend segmentom aplikacije. Angular tu komunikaciju rješava korištenjem *servisa*[34]. Standardna praksa je definirati servis za svaku klasu objekata koja se izmjenjuje s backend segmentom aplikacije. Svaki servis se pak sastoji od metoda koje implementiraju jednu od CRUD metoda. Time se kod daljnje implementacije funkcionalnosti, posebno onih vezanih za pregled podataka povećava jednostavnost, jer je dalje za akciju na nekom objektu dovoljno napisati jednu liniju koda koja će korištenjem servisa tu akciju izvršiti.
Datoteka imena `<imemodula>.routing.ts` sadrži podatke o putanjama i o tome koja će se komponenta nalaziti na kojoj putanji. Sličnog je koncepta kao i url datoteka u DRF-u, s tim da ona predstavlja ona omogućava korisničku navigaciju kroz aplikaciju tako da korisnik klika na pojedinu poveznicu.
Posljednji bitan segment modula je datoteka koja sadrži njegovu definiciju. U njoj se navode sve biblioteke i komponente koje modul uvozi i deklarira i koje su potom globalno dostupne na razini modula. U slučaju većih aplikacija koje koriste velik broj dodatnih biblioteka i modula ovo može biti korisno, jer će se uvoz pojedinog segmenta raditi samo onda kada je potreban i neće bespotrebno opterećivati aplikaciju i korisničko sučelje. To je izrazito bitno ako se želi postići veća fluidnost aplikacije i bolje korisničko iskustvo.

1.2.2.2.1 Authentication

Modul authentication dostupan na korisničkom sučelju ima primarnu svrhu omogućiti prijavu korisnika na sustav te potom omogućiti da se prilikom slanja zahtjeva prema backendu doda u zaglavlje dodaju podatci o korisniku. Svrha izdvajanja ovog segmenta u posebni modul je ista kao i kod Django Rest Frameworka, postizanje veće modularnosti i olakšavanje budućih nadogradnji aplikacije i proširenja funkcionalnosti s novim modulima. Budući je prethodno navedeno kako koji segment modula radi, ovdje će biti navedene komponente i servisi koji su definirani u ovom modulu, te će posebno biti prikazane specifičnosti koje odudaraju od standardnih implementacija. Samo je jedna klasa definirana u

[34]https://v6.angular.io/api/core/Injectable

ovom modulu, a to je Profile. Njeni atributi i organizacija podataka unutar nje odgovaraju onome što je definirano u backend segmentu aplikacije.

```
export class Profile {
    id: number;
    bio: string;
    location: string;
    birth_date: string;
    user: {
        id: number;
        username: string;
        first_name: string;
        last_name: string;
        email: string;
        projects: number[];
        graphs: number[];
        nodes: number[];
        edges: number[];
    };
}
```

Kako bi kreirali i mogli koristiti instancu klase Profile, potrebno je izmijeniti informacije s backend segmentom aplikacije te od njega dobiti točne podatke o korisniku i to se radi korištenjem servisa.

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { map } from 'rxjs/operators';
@Injectable({
  providedIn: 'root'
})
export class AuthenticationService {
  private userUrl = 'http://127.0.0.1:8000/profile/get/token/';
  constructor(
    private httpClient: HttpClient
    ) { }
  login(username: string, password: string) {
    return this.httpClient.post<any>(`${this.userUrl}`, {username:
    username, password: password})
    .pipe(map(user => {
      if (user && user.token ) {
        localStorage.setItem('currentUser', JSON.stringify(user));

        console.log('succesfull logged in ', user);
        return user.token;
```

```
      }
    }));
  }
  logout() {
    localStorage.removeItem('currentUser');
  }
}
```

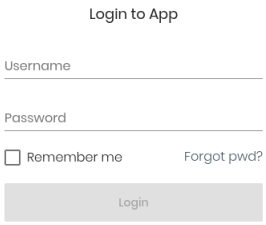
Podatci o korisniku se spremaju u lokalnu memoriju korisničkog računala, kako bi stalno bili dostupni za korištenje. Metoda login(username: string, password: string) prima dva argumenta, a to su username i password. Ti argumenti se šalju prema backendu, koji potom provjerava jesu li to ispravni podatci i vraća podatke o korisniku koji su potrebni za rad aplikacije.

```
import { Component, OnInit } from '@angular/core';
import { Router, ActivatedRoute } from '@angular/router';
import {
  FormBuilder,
  FormGroup,
  Validators,
  FormControl
} from '@angular/forms';
import { first } from 'rxjs/operators';

import { AuthenticationService } from '../services/authentication.
    service';
@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.scss']
})
export class LoginComponent implements OnInit {
  public form: FormGroup;
  loading = false;
  submitted = false;
  returnUrl: string;

  constructor(
    private fb: FormBuilder,
    private router: Router,
    private route: ActivatedRoute,
    private authService: AuthenticationService) {}
  ngOnInit() {
    this.form = this.fb.group({
      username: [null, Validators.compose([Validators.required])],
      password: [null, Validators.compose([Validators.required])]
```

```
32      });
33      this.authService.logout();
34      this.returnUrl = this.route.snapshot.queryParams['returnUrl'] || '/'
        ;
35    }
36    get func() {
37      return this.form.controls;
38    }
39
40
41    onSubmit() {
42
43      this.submitted = true;
44
45      if (this.form.invalid) {
46        return;
47      }
48
49      this.loading = true;
50      this.authService.login(this.func.username.value, this.func.password.
        value)
51          .pipe(first())
52          .subscribe(
53            data => {
54              this.router.navigate([this.returnUrl]);
55          },
56          error => {
57            this.loading = false;
58          }
59          );
60    }
61  }
```

Cijeli proces prijave korisnika se odvija kroz komponentu login koja je standardne arhitekture, dakle sastoji se od datoteka koje su prethodno navedene u opisu komponenti. Ovdje je prethodno prikazana samo datoteka koja implementira logiku korisničkog sučelja. Na korisničkom sučelju prozor prijave je vidljiv na Slika 1.1.

Ovaj modul ima i dvije komponente koje su specifične za rad aplikacije, a to su guards i helpers. Komponenta guards je klasa AuthGuard koja implementira CanActivate. Ukratko, tu se radi o mehanizmu zaštite pojedinih segmenta aplikacije od korisnika koji nisu registrirani i prijavljeni. Ako ne u lokalnoj memoriji ne postoji podatak o prijavljenom korisniku, tada se korisnika preusmjerava na prijavu koju se može vidjeti na Slika 1.1



Slika 1.1: Ekran za prijavu

```
import { Injectable } from '@angular/core';
import { Router, CanActivate, ActivatedRouteSnapshot,
    RouterStateSnapshot } from '@angular/router';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {

  constructor(private router: Router) {}

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot)
     {
    if (localStorage.getItem('currentUser')) {
      // console.log( localStorage.getItem('currentUser').name) ;
      return true;
    }

    this.router.navigate(['/authentication/login'], {queryParams: {
   returnUrl: state.url}});
    return false;
    }
}
```

Druga specifična komponenta je helpers, u kojoj je datoteka jwt.interceptor.ts

```
import { Injectable } from '@angular/core';
import { HttpRequest, HttpHandler, HttpEvent, HttpInterceptor } from '
    @angular/common/http';
import { Observable } from 'rxjs';
@Injectable()
export class JwtInterceptor implements HttpInterceptor {
    intercept(request: HttpRequest<any>, next: HttpHandler): Observable<
    HttpEvent<any>> {
        const currentUser = JSON.parse(localStorage.getItem('currentUser
    '));
        if (currentUser && currentUser.token) {
            request = request.clone({
                setHeaders: {
                    Authorization: `Token ${currentUser.token}`
                }});}
        return next.handle(request);
    }
}
```

Klasa JwtInterceptor implementira klasu HttpInterceptor. Ova pomoćna klasa je vrlo korisna jer olakšava pristupanje resursima u backend segmentu aplikacije. Kako je većina resursa zaštićena na način da ih čitati mogu samo registrirani korisnici, a uređivat samo vlasnici, prilikom pristupanja tim resursima trebalo bi u zaglavlje svake metode svakog servisa dodavati podatke o korisniku. JwtInterceptor to radi na učinkovit i efikasan način i time zaobilazi mnogo repetitivnog pisanja koda.

1.2.2.2.2 Graphapp

U graphapp modulu je implementirano korisničko sučelje i logika aplikacije koja je izravno povezana s backend segmentom aplikacije. Osim standardnih elemenata modula, ovdje postoje i specifične komponente koje se zovu dialog, koje su zamišljene kao čarobnjaci za kreiranje novih objekata. Graphapp ima pet definiranih klasa u direktoriju models. To su redom isti objekti kakvi su definirani i u backend segmentu aplikacije, a to su: project, graph, node, edge i category. Njihove definicije ne zahtijevaju dodatne metode ili podklase kakve zahtijeva DRF.

```
export class Project {
    id: number;
    name: string;
    content: string;
    graphs: number[];
    date_created: string;
    date_modified: string;
}
```

```
export class Graph {
    id: number;
    name: string;
    content: string;
    description: string;
    project: number;
    cytoscape: any;
    graph_nodes: number[];
    graph_edges: number[];
}
```

```
export class Node {
    id: number;
    name: string;
    content: string;
    description: string;
    category: number;
    parents: string[];
    graphs: number[];
    edge_start: number[];
    edge_end: number[];
    date_created: string;
    date_modified: string;
}
```

```
export class Edge {
    id: number;
    name: string;
    content: string;
    node_start: number;
    node_end: number;
    graph: number;
}
```

```
export class Category {
    id: number;
    name: string;
    description: string;
    color: string;
    shape: string;
    project: number[];
}
```

Svaka od navedenih klasa ima posebno definiranu klasu koja implementira servise. Dodatno, klase Project, Graph i Node imaju definirane i komponente tipa dialog. Svi servisi implementiraju istu logiku te su posloženi na isti način. Ovdje je prikazan servis za klasu Graph.

```
import { Injectable } from '@angular/core';
import { Observable, of } from 'rxjs';
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { catchError, map, tap } from 'rxjs/operators';

import { Graph } from '../models/graph';

const httpOptions = {
  headers: new HttpHeaders({ 'Content-Type': 'application/json'})
};

@Injectable({
  providedIn: 'root'
})
export class GraphService {

  private graphsUrl = 'http://127.0.0.1:8000/graphapp/graphs/';
  private dataUrl = 'http://127.0.0.1:8000/graphapp/graphs/data/';

  constructor(private http: HttpClient) { }

  getGraphs(): Observable<Graph[]> {
    return this.http.get<Graph[]>(this.graphsUrl);
  }

  getGraphsProject(project_id: number): Observable<Graph[]> {
    const url = `${this.graphsUrl}?project__id=${project_id}`;
    return this.http.get<Graph[]>(url);
  }

  getGraphNo404<Data>(id: number): Observable<Graph> {
```

```
32      const url = `${this.graphsUrl}${id}/`;
33      return this.http.get<Graph[]>(url).pipe(map(graphs => graphs[0]));
34    }
35
36    getGraph(id: number): Observable<Graph> {
37      const url = `${this.graphsUrl}${id}/`;
38      return this.http.get<Graph>(url);
39    }
40
41    getGraphData(id: number): Observable<Graph> {
42      const durl = `${this.dataUrl}${id}/`;
43      return this.http.get<Graph>(durl);
44    }
45      searchGraph(term: string): Observable<Graph[]> {
46      if (!term.trim()) {
47        return of([]);
48      }
49
50      return this.http.get<Graph[]>(`${this.graphsUrl}?search=${term}`);
51    }
52
53    addGraph(graph: Graph): Observable<Graph> {
54      return this.http.post<Graph>(this.graphsUrl, graph, httpOptions);
55    }
56
57    deleteGraph (graph: Graph | number): Observable<Graph> {
58      const id = typeof graph === 'number' ? graph : graph.id;
59      const url = `${this.graphsUrl}${id}/`;
60
61      return this.http.delete<Graph>(url, httpOptions);
62    }
63
64    updateGraph (graph: Graph): Observable<any> {
65      const id = typeof graph === 'number' ? graph : graph.id;
66      const url = `${this.graphsUrl}${id}`;
67      return this.http.put(url, graph, httpOptions);
68    }
69
70  }
```

Za kreiranje nove instance klasa Project, Graph ili Node korištena je komponenta tipa dialog. Dialog se poziva iz postojeće komponente te se radi injekcija u samo sučelje te se dialog pojavi u obliku čarobnjaka s određenim koracima, koji omogućava kreiranje novog objekta. U svakom od koraka se definira jedno obvezno polje koje korisnik mora unijeti. Većina je polja tekstualnog oblika pa je kroz korake istaknuta bitnost pojedinog atributa. Ostatak samog dialoga je definiran kao i kod ostalih komponenti.

```
import { Component, OnInit, Inject } from '@angular/core';
import { MatDialogRef, MAT_DIALOG_DATA } from '@angular/material';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';
import { Graph } from '../models/graph';
import { GraphService } from '../services/graph.service';

@Component({
  selector: 'app-graph-add-dialog',
  templateUrl: './graph-add-dialog.component.html',
  styleUrls: ['./graph-add-dialog.component.css']
})
export class GraphAddDialogComponent implements OnInit {

  firstFormGroup: FormGroup;
  secondFormGroup: FormGroup;
  thirdFormGroup: FormGroup;
  project_id: number;

  newGraph: Graph;

  constructor(
    public graphAddDialogRef: MatDialogRef<GraphAddDialogComponent>,
    private formBuilder: FormBuilder,
    private graphService: GraphService,
    @Inject(MAT_DIALOG_DATA) public data: any
  ) { }

  ngOnInit() {
    this.firstFormGroup = this.formBuilder.group({
      firstCtrl: ['', Validators.required]
    });
    this.secondFormGroup = this.formBuilder.group({
      secondCtrl: ['', Validators.required]
    });
    this.thirdFormGroup = this.formBuilder.group({
      thirdCtrl: ['', Validators.required]
    });

    this.project_id = this.data.project_id;

    console.log(this.project_id);
    }

  addGraph(name: string, content: string, description: string ): void {
    name = name.trim();
    description = description.trim();
```

```
47      content = content.trim();
48      const project = this.project_id;
49      if ( !name || !description || !content ) {
50        this.graphAddDialogRef.close(null);
51      }else {
52        this.graphService.addGraph({name, content, description, project}
      as Graph).subscribe( graph => {
53          this.newGraph = graph;
54        this.graphAddDialogRef.close(this.newGraph);
55      });}}
```

```
1  <div mat-dialog-content>
2    <p>Wizard for creating new graph</p>
3  </div>
4  <div mat-dialog-actions>
5    <mat-horizontal-stepper [linear]="true" #stepper>
6      <mat-step [stepControl]="firstFormGroup">
7        <form [formGroup]="firstFormGroup">
8          <ng-template matStepLabel>Graph Name</ng-template>
9          <mat-form-field>
10           <input #graphName matInput placeholder="Name Your Graph"
       formControlName="firstCtrl" required appearance="outline">
11         </mat-form-field>
12         <div>
13           <button mat-button matStepperNext>Next</button>
14         </div>
15       </form>
16     </mat-step>
17     <mat-step [stepControl]="secondFormGroup">
18       <form [formGroup]="secondFormGroup">
19         <ng-template matStepLabel>Content</ng-template>
20         <mat-form-field>
21           <textarea #graphCont matInput placeholder="Graph Content"
       formControlName="secondCtrl" required appearance="outline"></
       textarea>
22         </mat-form-field>
23         <div>
24           <button mat-button matStepperPrevious>Back</button>
25           <button mat-button matStepperNext>Next</button>
26         </div>
27       </form>
28     </mat-step>
29     <mat-step [stepControl]="thirdFormGroup">
30       <form [formGroup]="thirdFormGroup">
31         <ng-template matStepLabel>Description</ng-template>
32         <mat-form-field>
```

```
33          <textarea #graphDesc matInput placeholder="Short Graph
    Description" formControlName="thirdCtrl" required appearance="
    outline"></textarea>
34        </mat-form-field>
35        <div>
36          <button mat-button matStepperPrevious>Back</button>
37          <button mat-button matStepperNext>Next</button>
38        </div>
39      </form>
40    </mat-step>
41    <mat-step>
42      <ng-template matStepLabel>Done</ng-template>
43      You are now ready to create new graph
44      <div>
45        <button mat-button matStepperPrevious>Back</button>
46        <button mat-button (click)="stepper.reset()">Reset</button>
47        <button mat-button (click)="addGraph(graphName.value, graphCont.
    value, graphDesc.value)">Add</button>
48      </div>
49    </mat-step>
50  </mat-horizontal-stepper>
51 </div>
```

Najbitniji segment aplikacije je sama graph komponenta u kojoj se nalazi većina logike te cijelog sučelja za aplikaciju. Osim vizualizacije podataka kroz strukturu grafa ili stabla, kroz tu je komponentu moguće i dodavati nove čvorove i veze te raditi manipulaciju samim grafom na način da se mijenjaju algoritmi rasporeda čvorova u prostoru, izolacija čvorova koji su izravni prethodnici ili izravni prethodnici, te crtanje podgrafa na temelju svih prethodnika odabranog čvora. Dodatno je moguće napraviti i izvoz slike u .png formatu kako bi korisnik mogao dalje koristiti te podatke. Podatci se trajno čuvaju u bazi podataka te im korisnik uvijek može pristupiti.
Budući je sama implementacija komponente graph opširna, ovdje su prikazane samo bitne metode i pristupi koji su omogućili razvoj navedenih funkcionalnosti. Prilikom konstruiranja komponente odmah se dohvaćaju potrebni podatci za vizualizaciju grafa na temelju atributa id.

```
constructor(
    private dataService: DataService,
    private graphService: GraphService,
    private edgeService: EdgeService,
    private nodeService: NodeService,
    private catService: CategoryService,
    public graphDialog: MatDialog ) { }

    ngOnInit() {
      let grID: number;
      this.dataService.currentGraphID.subscribe(graph_id => {
        this.graph_id = graph_id
      });
      this.locallayout = layout.breadthfirst;
      this.getGraphData();
      this.getCategories(this.my_graph.project);
  }
```

Nakon pripreme i dohvata podataka, korisnik vidi prikazanu strukturu svog grafa. Graf je potpuno interaktivan, pa je moguće raditi povećavanje ili smanjivanje veličine slike kako bi se vidjelo više ili manje detalja. Također je moguće i ručno razmještati čvorove po dostupnome prostoru, a sve je to omogućeno kroz biblioteku cystoscape.js.
Osim interakcije kroz manipulaciju, moguće je dobiti i informacije o pojedinom čvoru (klasa Node), tako da se klikne na taj čvor. Budući biblioteka cytoscape.js ima implementirane event handlere, preko njih je moguće doći do podataka na koji je objekt korisnik kliknuo. Zbog ovoga je u metodi `graphData(graph_id)` prilikom definiranja skupa podataka za cytoscape, atribut id segmenta node postavljen točno na isti id tog objekta u bazi. Ovime se ubrzalo dohvat podataka jer su reference na podatke u bazi podataka već spremljene u samu strukturu cytoscape objekta bez da se zadiralo u njegovu definiciju. U klasi GraphComponent ovo je postignuto kroz metodu handleNode(event: Event). Navedena logika je iskorištena i za instance klase Edge kroz metodu handleEdge(event:Event).

```
handleNode(event: Event): void {
    if (this.edgeOptions) {
      this.cyt.on('click', 'node', function (event){
        const nodee = event.target;
        if (nodee.id().search('_') === -1) {
          global_node2 = nodee.id();
          // console.log('EdgeOptions and global_node2');
        }});
    } else {
      this.cyt.on('click', 'node', function(event) {
        const nodee = event.target;
```

```
        // console.log(nodee);
        if (nodee.id().search('_') === -1) {
          global_node = nodee.id();
          // console.log('Just the global_node');
        }});}
    if (this.edgeOptions && global_node2) {
      this.getNodeEvent(global_node2);
      // console.log('Prepared to add new Edge');
      global_node = null;
    }
    if (global_node) {
      this.getNodeEvent(global_node);
      // console.log('Sent out to get the node1');
      global_node = null;
    }}

handleEdge(event: Event): void {
    this.bLay = false;
      this.cyt.on('click', 'edge', function(event) {
        const edgee = event.target;
        if (edgee.id().search('_')) {
          global_edge = edgee.id().split('_', 1)[0];
        }});
      if (global_edge) {
        this.getEdgeEvent(global_edge);
        global_edge = null;
    }}
```

```
getEdgeEvent(id: number): void {
    this.bLay = false;
    this.my_node1 = null;
    this.edgeService.getEdge(id).subscribe(edge => this.my_edge1 = edge)
    ;
  }
getNodeEvent(id: number): void {
    this.bLay = false;
    if (!this.edgeOptions) {
      this.my_edge1 = null;
      this.nodeService.getNode(id).subscribe(node => this.my_node1 =
    node);
    } else {
      this.my_edge1 = null;
      this.nodeService.getNode(id).subscribe(node => this.my_node2 =
    node);
    }
  }
```

Dodatno su za ove dvije metode implementirane pomoćne metode, a to su getNodeEvent(id: number) i getEdgeEvent(id: number).
Mogućnost proučavanja podataka kroz graph dodatno olakšavaju sljedeće tri metode. Metoda `isolateSourceNodes(node_id: number)` vraća stablo neposrednih susjeda koji imaju svojstvo da veza polazi iz njih prema danom čvoru. `IsolateTargetNodes(node_id: number)` vraća stablo neposrednih susjeda koji imaju svojstvo da veza iz danog čvora završava u njima. Treća metoda u ovom segmentu je`isolateParents(node_id: number)`, a omogućava pronalazak svih predaka pojedinog čvora, sve do maksimalne dubine grafa koji se trenutno promatra. Za tu potrebu se dodatno koristi skup podataka parents koji je unaprijed pripremljen prilikom dohvata podataka.

```
isolateSourceNode(node_id: number): void {
     const edges = this.my_graph.cytoscape.elements.edges;
     const nodes = this.my_graph.cytoscape.elements.nodes;
     const edge_ids = new Array();
     const new_edges = new Array();
     const new_nodes = new Array();
     edges.forEach(element => {
       const source_node_id = parseInt(element.data.source, 10);
       const target_node_id = parseInt(element.data.target, 10);
       if (source_node_id === node_id) {
         new_edges.push(element);
         edge_ids.push(source_node_id);
         edge_ids.push(target_node_id);
       }
      });
    nodes.forEach(element => {
      const edge_node_id = parseInt(element.data.id, 10);
      if (edge_ids.includes(edge_node_id)){
        new_nodes.push(element);
      }
    })
    if (new_edges.length >= 1) {
      this.cyt = cytoscape({
        container: document.getElementById('cy'),
        elements:
        {
        nodes: new_nodes,
        edges: new_edges
      },
      layout: this.locallayout,
      style : this.cysheet,
    });}
 }

```

```
35
36 isolateTargetNode(node_id: number): void {
37       const edges = this.my_graph.cytoscape.elements.edges;
38       const nodes = this.my_graph.cytoscape.elements.nodes;
39       const edge_ids = new Array();
40       const new_edges = new Array();
41       const new_nodes = new Array();
42       edges.forEach(element => {
43         const source_node_id = parseInt(element.data.source, 10);
44         const target_node_id = parseInt(element.data.target, 10);
45         if (target_node_id === node_id) {
46           new_edges.push(element);
47           edge_ids.push(source_node_id);
48           edge_ids.push(target_node_id);
49         }
50        });
51      nodes.forEach(element => {
52        const edge_node_id = parseInt(element.data.id, 10);
53        if (edge_ids.includes(edge_node_id)) {
54          new_nodes.push(element);
55        }
56      });
57      if (new_edges.length >= 1) {
58        this.cyt = cytoscape({
59          container: document.getElementById('cy'),
60          elements:
61          {
62          nodes: new_nodes,
63          edges: new_edges
64        },
65        layout: this.locallayout,
66        style : this.cysheet,
67      });}
68
69 isolateParents(node: Node): void {
70       const my_nodes = this.my_graph.cytoscape.elements.nodes;
71       const my_edges = this.my_graph.cytoscape.elements.edges;
72       const new_nodes = new Array();
73       const new_edges = new Array();
74       const old_parents = new Array();
75       my_nodes.forEach(element => {
76         if (String(node.id) === element.data.id) {
77           const new_elem = element;
78           new_elem.data.shape = 'square';
79           new_elem.data.color = '#000099';
80           new_nodes.push(new_elem);
81           this.my_parents = element.data.parents;
```

```
82            old_parents.push(element.data.id);
83          }});
84        if (this.my_parents.length > 0) {
85          while (this.my_parents.length > 0) {
86            my_nodes.forEach(element => {
87              if (this.my_parents.includes(element.data.id) && !
    old_parents.includes(element.data.id)) {
88                new_nodes.push(element);
89                old_parents.push(element.data.id);
90                const index = this.my_parents.indexOf(element.data.id, 0);
91                if (index > -1) {
92                  this.my_parents.splice(index, 1);
93                }
94                element.data.parents.forEach(parent => {
95                  if (!old_parents.includes(parent)) {
96                    this.my_parents.push(parent);
97                  }
98                });
99                console.log('Array of parents: ', this.my_parents);
100               console.log('Array of OLD  parents: ', old_parents);
101               this.my_parents.forEach(my_par => {
102                 if (old_parents.includes(my_par)) {
103                   const loc_ind = this.my_parents.indexOf(my_par, 0);
104                   if (loc_ind > -1) {
105                     this.my_parents.splice(loc_ind, 1);
106                   }
107                 }
108               });}});}}
109       if (new_nodes.length > 0) {
110         this.cyt = cytoscape({
111           container: document.getElementById('cy'),
112           elements:
113           {
114           nodes: new_nodes,
115           edges: this.my_graph.cytoscape.elements.edges,
116         },
117         layout: this.locallayout,
118         style : this.cysheet,
119       });}
120       this.my_parents = null;
121   }
```

Izmjene parametara rasporeda čvorova i veza unutar vizualizacije dovode do dodatnog uvida u pojedini skup podataka. Kod pregleda prvih susjeda je korisno rasporediti podatke u krug, dok je kod pretrage predaka puno bolje rasporediti čvorove u strukturu stabla. Ako stablo ima mnogo čvorova s velikim nazivima i imenima, dobro je podesiti i minimalni

razmak. Sve se to postiže kroz metodu editLayout(type: string).

```
editLayout(type: string): void{
    if (type === 'cose'){
      const newLayout = this.cyt.layout(layout.cose);
      newLayout.run();
    }
    if (type === 'breadthfirst'){
      const newLayout = this.cyt.layout(layout.breadthfirst);
      newLayout.run();
    } else {
      const newLayout = this.cyt.layout(this.locallayout);
      newLayout.run();
    }
  }
```

Posljednja bitna funkcionalnost je izvoz podataka u sliku. To omogućava korisniku da podatke koje je proučavao ili kontekst koji mu je bitan, može kasnije opet pogledati i provjeriti. Navedeno se omogućava korištenjem metode exportImage().

```
exportImage(): void {
    const options = {
        bg: 'white',
        full: false,
        output: 'blob'
      };
    const image = this.cyt.png(options);
    const blob = new Blob( [image], {type: 'png'.toString()});
    const imgUrl = window.URL.createObjectURL(blob);
    const link = document.createElement('a');
    link.href = imgUrl;
    link.download = (this.my_graph.name + '.png').toString();

    link.dispatchEvent(new MouseEvent('click', { bubbles: true,
   cancelable: true, view: window }));
  }
```

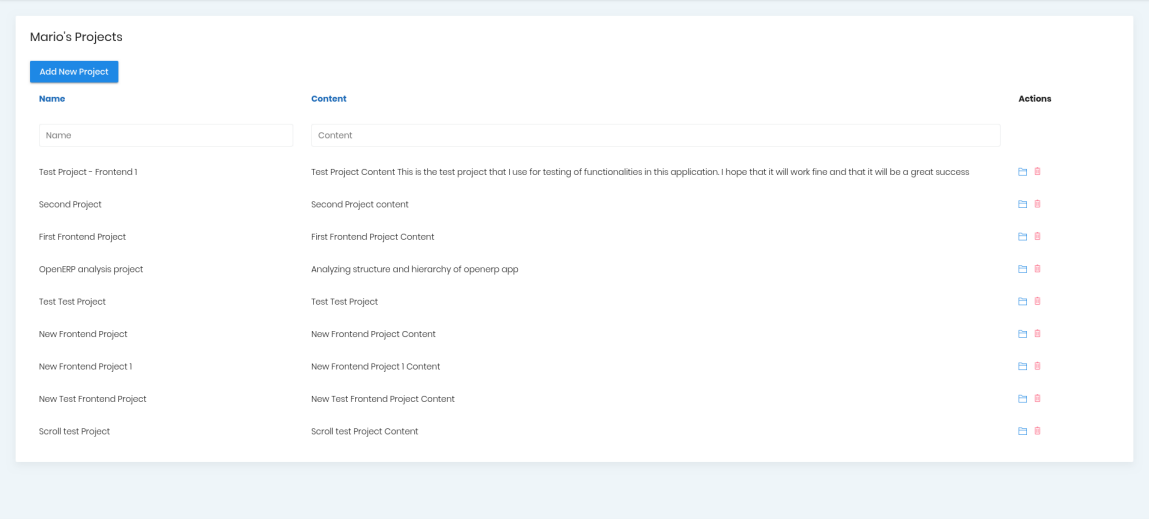
Osim uređivanja skupa podataka, korisnik može i proširivati postojeći skup podataka dodajući nove čvorove i veze. Ova funkcionalnost je implementirana i za klasu Node i za klasu Edge. Novi objekti se, kako je to već prethodno objašnjeno dodaju kroz čarobnjake. Primjer implementacije jednog takvog rješenja je dan kroz metodu openAddNodeDialog().

```
openAddNodeDialog(): void {
    const nodeAddDialogRef = this.graphDialog.open(
   NodeAddDialogComponent, {
      width: 'auto',
      data: { graph_id: this.my_graph.id }
    });
    nodeAddDialogRef.afterClosed().subscribe(
      node => {
        if (node) {
          this.my_node1 = node;
          this.getGraphData();
        }
      }
    );
  }
```

Poglavlje 2

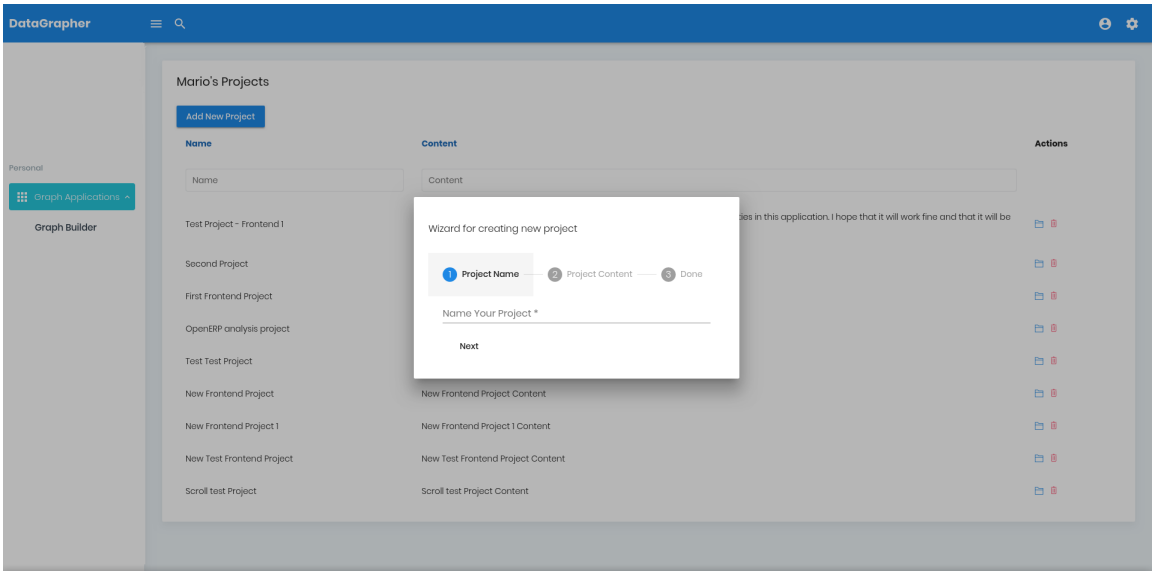
Primjer korištenja aplikacije

Aplikacija je rađena na način da bude što više intuitivna i jednostavna za korištenje, tako da korisnik relativno jednostavno može ovladati njome. Nakon prijave u aplikaciju, korisniku su pojavi popis njegovih projekata (Slika 2.1). Moguće je kreirati nove projekte ali i dodavati nove (Slika 2.2).



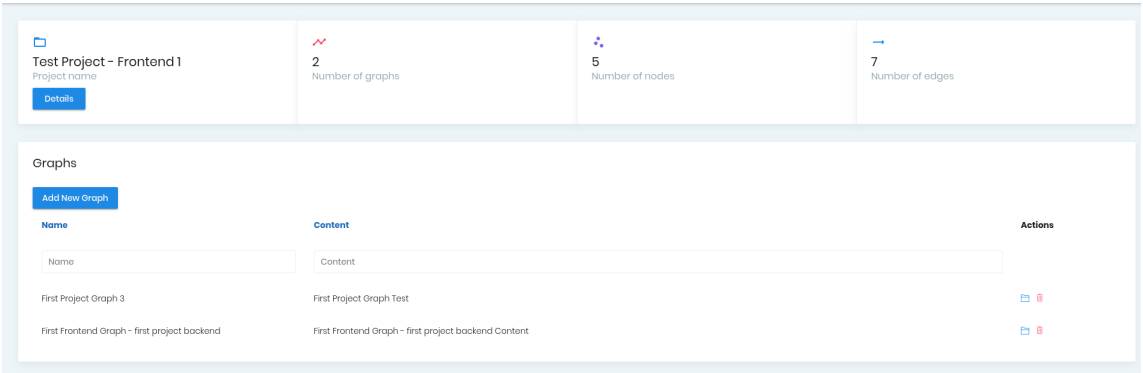
Slika 2.1: Lista Projekata

Nakon što korisnik odabere projekt na kojem želi nastaviti radit, moguće je taj projekt dalje proučavati kroz njegovu kontrolnu ploču (Slika 2.3) na kojoj se vidi i osnovna statistika tog projekta. Kontrolnu ploču je moguće proširiti i vidjeti dodatne podatke o samom projektu, kao i urediti opis projekta (Slika 2.4)

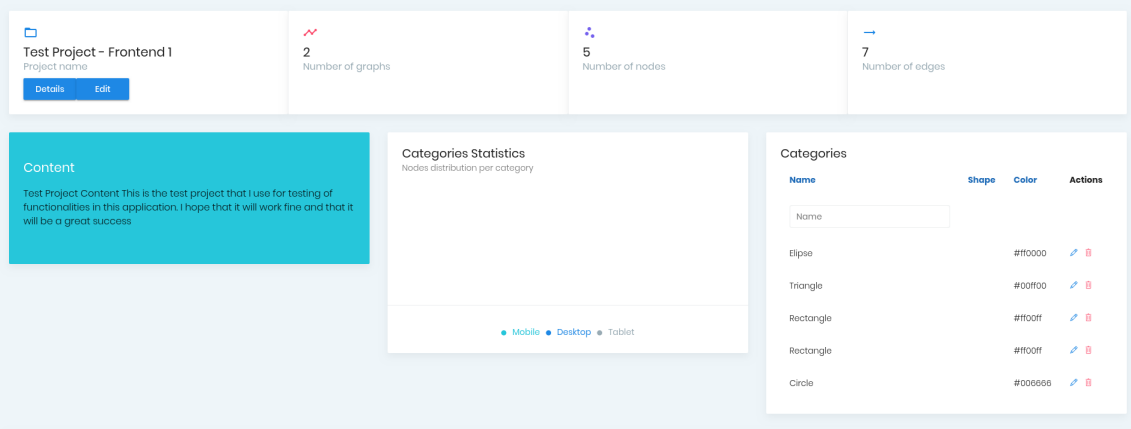


Slika 2.2: Dodavanje novog projekta

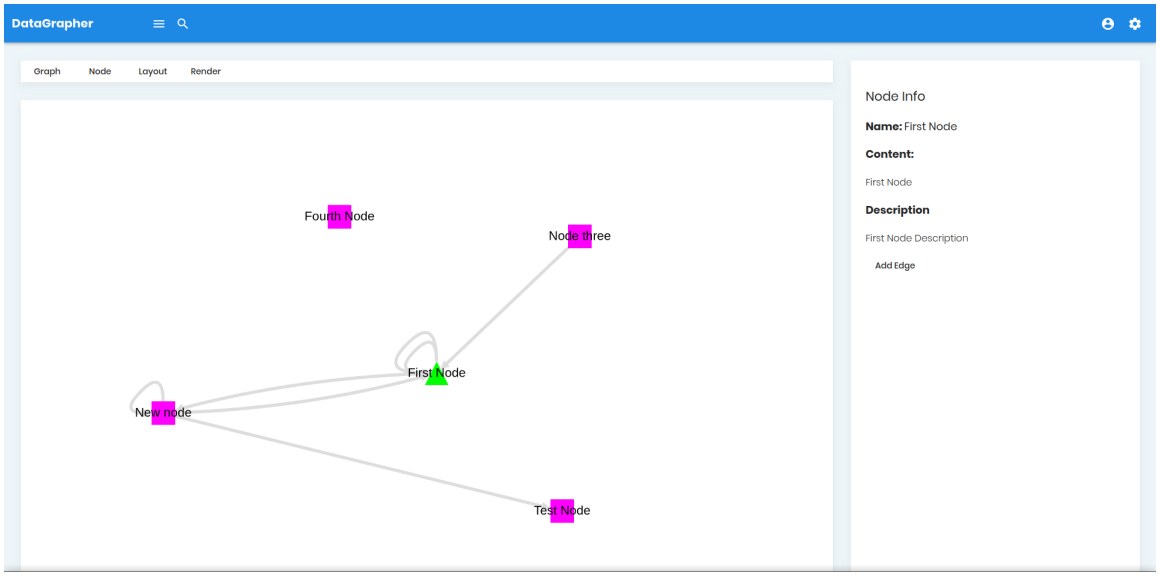
Osim podataka o projektu, korisnik na kontrolnoj ploči projekta vidi i sve grafove koji su povezani s tim projektom. Klikom na neki od grafova, otvaraju se podatci o tom grafu i dolazi do vizualizacije podataka (Slika 2.5). Korisnik na navedenom sučelju za vizualizaciju grafa može koristiti sve funkcionalnosti koje su opisane u dijelu vezanom za graphapp.



Slika 2.3: Pregled projekta



Slika 2.4: Prošireni pregled projekta



Slika 2.5: Vizualizacija grafa

Bibliografija

[1] D. B. Copeland, *Rails, Angular, Postgres, and Bootstrap - Powerful, Effective, Efficient, Full-Stack Web Development*, 2. izdanje, The Pragmatic Bookshelf, 2017.

[2] A. Freeman, *Pro Angular 6*, Apress, treće izdanje, 2018.

[3] Angular,
`https://angular.io/`, Angular Web Framework

[4] Cytoscape.js,
`http://js.cytoscape.org/`, Graph theory (network) library for visualisation and analysis

[5] Angular Material Admin Pro
`https://www.wrappixel.com/templates/materialpro-angular-dashboard/`, Angular Material Admin Pro dashboard for organization of frontend components

[6] Django,
`https://www.djangoproject.com/`, Django Web Framework

[7] DRF,
`https://www.django-rest-framework.org/`, Django Rest Framework

[8] PostgreSQL 10,
`https://www.postgresql.org/docs/10/index.html`, Open Source Relational Database

Sažetak

Web aplikacija za vizualizaciju hijerarhijskih podataka je razvijana kako bi se olakšalo vizualizaciju podataka u kojima postoji relacija ili hijerarhija. Mogućnost različitih prostornih raspoređivanja čvorova daje dodatni uvid podatke. Tako Breadthfirstsearch layout omogućava pregled hijerarhije unutar usmjerenog grafa, te jednostavnu izolaciju jednog ili više korijena, ako postoje unutar grafa. Izolacijom predaka pojedinog čvora možemo dobiti sve izravne prethodnike nekog čvora te tako izolirati podgraf koji sadrži samo one čvorove i bridove koji su bitni u kontekstu odabranog čvora.

Filtriranje po kategorijama čvorova pruža uvid u to kakve veze postoje unutar nekog podskupa podataka s danim obilježjem, to jest kategorijom.
Postoji veliki skup raznovrsnih podataka koji se mogu prilagoditi i prikazati pomoću ovakve aplikacije, a to može biti kao što je prikazano u primjeru, skup znanja iz nekog udžbenika. Dodatno se mogu prikazati skupovi podataka poput prometne, telekomunikacijske, plinske, vodovodne ili električne mreže. Osim toga može se koristiti u razvoju softvarea za prikaz *Entity-Relationship*[1] modela ili kao dijagram da se vide međuovisnosti pojedinih biblioteka ili paketa koji su potrebni za razvoj aplikacije. Tu svakako spada i mogućnost kreiranja grafa povezanosti između web stranica pa bi to bilo izrazito zanimljivo napraviti za Wikipediu, koja ima mnoštvo poveznica između stranica koje bi predstavljale čvorove. Aplikacija se može koristiti i za kreiranje skupa podataka tako da se korištenjem sučelja kreiraju novi projekti, grafovi, informacije i veze među njima koje se potom automatski spremaju u strukturu koja se može vizualizirati, a isto tako i uređivati i nadograđivati.

Prostor za širenje funkcionalnosti aplikacije postoji na svim razinama. Od optimizacije koda, do dodavanja novih funkcionalnosti koje se mogu koristiti za bolju vizualizaciju skupa podataka. Prostor leži i u povezivanju aplikacije s metodama strojnog učenja koje otvaraju jednu novu dimenziju. Primjer bi bio određeni skup pojmova, te klasični tekst bez ikakvih posebnih tagova. Tada bi se korištenjem tehnika vezanih za *NLP*[2] mogla napraviti obrada teksta te vizualizirati povezanost danih pojmova unutar teksta.

[1] https://en.wikipedia.org/wiki/Entity-relationship_model
[2] https://en.wikipedia.org/wiki/Natural_language_processing

Izazovi za ovakvu aplikaciju su prvenstveno u veličini podataka i njihovoj pripremi. Ostaju otvorena pitanja kako bi se aplikacija ponašala u slučaju vizualizacije grafova koji sadrže milijune čvorova te još više bridova. Skup podataka nad kojim bi se mogla provesti takva testiranja i proučavanja je Wikipedia, koja je javno dostupna za preuzimanje u obliku komprimirane xml datoteke.

Summary

Web application for visualization of hierarchically structured data was developed in order to enable visualization of any data that is structured in relational or hierarchical order. By using different graph layouts user can get an extra insight in the data. Breadthfirstsearch layout enables one to see if there is a hierarchy inside the directed graph and to isolate all root nodes inside it, if there is any. By isolating ancestors of one node, application can show all direct predecessors and by doing so user can see the subgraph that contains only the nodes and edges important for the chosen node.
Filtering data by categories will enable user to see if there are any connections inside a specific subset of data.
There is a vast array of different data sets that can be adapted and visualized by using an application such as this, and it can be done in the same way as it was shown in example by adapting the knowledge set from schoolbook. But it can also visualize data from traffic, telco, gas, electrical or water grid, or in software development for visualization of Entity-Relationship model, or as a diagram for modules dependency in software. There is also a potential for visualization of web, showing web pages and links between them and it would be extremely interesting to test on Wikipedia, as it has millions of articles and many more millions of links. Application can also be used for creation of data through user interface by creating projects, graphs, nodes, edges and categories, that will be automatically saved in relational or hierarchical structure that can be visualized, updated or upgraded.
There are many other functionalities that could be implemented in different parts of application and it is spread from optimizing code to adding new options for visualization. Other direction for improvement is connecting the application with machine learning solutions. For example, one could use a given set of data without any connections, and then create connections by extracting them with NLP from text. This would enable user to create edges between existing nodes, and by doing so build a new graph structure. There are also many challenges for application such as this one and they are primarily in size od data set and in preparation of that data set for use in application. There are still some open questions, some of them are directly related to visualization of data sets with millions of nodes and edges. Those questions could be answered by processing Wikipedia, which can be downloaded and processed.

Životopis

Mario Bošnjak je rođen 19.05.1988. u Sinju. Osnovnu školu je pohađao u Hrvacama od 1995. – 2003. nakon koje upisuje Franjevačku Klasičnu Gimnaziju s pravom javnosti u Sinju te srednjoškolsko školovanje završava 2007. Školovanje te iste godine nastavlja na inženjerskom smjeru preddiplomskog sveučilišnog studija Matematičkog odsjeka PMF-a u Zagrebu. Akademski naziv sveučilišnog prvostupnika stječe 2015. godine te iste godine upisuje diplomski studij Matematike i računarstva na PMF-u u Zagrebu.