MASTER RAD

1. UVOD

Zadatak rada je izrada web aplikacije koja olakšava unos i obradu rezultata ispita na predmetu Programski jezici i strukture podataka. Aplikacija je razvijena korišćenjem Django i Angular framework-a, Google Drive API-ja, kao i Docker alata za kontejnerizaciju. Razlog za kreiranje aplikacije je potreba standardizovanja metode za unos i obradu podataka o rezultatima ispita i kolokvijuma.

2. PREGLED KORIŠĆENIH TEHNOLOGIJA

2.1 DJANGO

Django je framework za izradu web aplikacija baziran na Python programskom jeziku. Koristi MVT (model-view-template) arhitekturalni obrazac. Uz Django, u izradi rada, korišćena je i biblioteka djangorestframework, koja omogućuje slanje rest zahteva aplikaciji.

2.1.1 Struktura Django aplikacije

Django aplikacija se, po pravilu, sastoji od različitih modula, organizovanih tako da su modeli i kontroleri odvojeni, u razlicitim fajlovima.
Postoji glavni modul koji se koristi za upravljanje ostatkom aplikacije. U tom modulu se nalaze izmedju ostalog podesavanja aplikacije (kredencijali za konekciju sa bazom, opis trenutne strukture aplikacije, putanja do statickih fajlova, potreban midleware…). Glavni modul takodje sadrzi i skripte za pokretanje debug servera kao i mapiranje ruta na funkcije u aplikaciji.

2.1.2 Modeli u Djangu

Dajngo dolazi sa implementiranim objekt relacionim mapiranjem (ORM), koje podržava za mnogobrojne baze podataka. Objekti koji podržavaju ORM nasleđuju klasu django.db.models.model, Ova klasa dozvoljava implementiranje polja razlicitih tipova kao i definisanje relacija koje ce biti mapirane na bazu podataka i čuva se u fajlu models.py.

Primer klase koja implementira model Person:

```
from django.db import models

class Person(models.Model):
        first_name = models.CharField(max_length=30)
        last_name = models.CharField(max_length=30)
```

Klasa models nudi API koji podržava kreiranje, pretragu, izmenu i brisanje podataka iz baze. Čuvanje objekta u bazi obavlja se pozivom funkcije save() nad njim.

2.1.3 REST pozivi

Django ne podržava REST pozive bez instaliranja dodatnih biblioteka. Jedna od biblioteka koja omogućava slanje REST poziva u Django aplikaciji je djangorestframework. Da bi se REST pozivi mogli koristiti djangorestframework nudi klasu rest_framework.serializers.ModelSerializer. Instance objekata koje implementiraju ovu klasu podržavaju mapiranje JSON objekata na primitivne tipove podataka u Python-u i obrnuto. Instance ove klase mogu da sadrže validatore podataka koji se definišu u unutrašnjoj klasi Meta. Serijalizatori se čuvaju u serializers.py fajlu.

Primer klase za serijalizaciju koja serijalizuje i validira sva polja klase Person:

```
class PersonSerializer(serializers.ModelSerializer):
    class Meta:
        model = Person
        fields = "__all__"
```

2.1.4 Mapiranje zahteva

Po konvenciji, zahtevi se mapiraju putem funkcija definisanih u fajlu views.py. Funkcije se mapiraju na url-ove definisane u fajlu urls.py. Kad se zahtev uputi, funkcija na koju se šalje zahtev kao svoj parametar uzima podatke poslate zahtevom. Parametar u sebi, između ostalog, sadrži telo zahteva i metod kojim se zahtev šalje (GET, POST, PUT, DELETE…)

Django poseduje klase django.http.HttpResponse i django.http.JsonResponse koje kao povratnu vrednost funkcije vraćaju odgovor aplikaciji u zahtevanom formatu.

Da bi se podaci mogli vratiti u JSON formatu potrebno je prethodno transformisati ih u odgovarajući format, korišćenjem ili Python-ove json biblioteke ili već opisane serijalizatore.

2.1.5 Gunicorn server

Gunicorn server je WSGI (web server gateway interface) koji može da se koristi za implementaciju (deployment) Django aplikacije. WSGI predstavlja konvenciju koja definiše standardni interfejs između web servera i Python web aplikacije. Fajl wsgi.py sadrži skript sa instrukcijama za pokretanje aplikacije na WSGI serveru.

2.1.6 Pandas biblioteka

Pandas je Python biblioteka za analizu podataka. Osnovna struktura podataka kojom ova biblioteka manipuliše je tabela, odnosno DataFrame. Pandas ima implementirane fukcije za preuzimanje i konverziju podataka iz različitih vrsta fajlova (csv, xls, json…). Funkcija pandas.DataFrame() kreira tabelu. Ovako kreirana table čuva podatke o kolonama i podacima koji se nalaze u njoj. Da bise podacima pristupilo može se koristiti funkcija pandas.DataFrame.loc[]. pandasDataFrame.loc[] pronalazi podatke na osnovu

unete labele kolone. Pandas podržava agregaciju podataka, kao i spajanje podataka na osnovu vrednosti polja.

2.2 Angular

Angular je framework za pravljenje single-page aplikacija u TypeScript programskom jeziku Single-page aplikacije imaju osobinu da se prilikom interakcije, umesto učitavanja celokupnog sadržaja stranice, trenutna web stranica samo ažurira potrebnim sadržajem. U okviru framework-a nalazi se i debug server. Framework omogućava konverziju sadržaja u čist HTML i JavaScript format.

2.2.1TypeScript

TypeScript je nadskup JavaScript programskog jezika u kome je uvedena tipizacija. Kao i JavaScript, TypeScript korisnicima omogućava kreiranje klasa, interfejsa i funkcija, ali, za razliku od JavaScript-a, u TypeScript-u postoji provera tipa podataka prilikom dodele.

Primer definisanja promenljivih u TypeScript-u:

```
let someString = “Hello”
var anotherString = “Hello”
someOtherString:string = “Hello”
```

Definisanje klase u TypeScript-u

```
class Person{
name: string’
surname: string

constructor(name:string, surname:string){
        this.name = name;
        this.surname = surname;

}}
```

Definisanje funkcija u TypeScript-u

```
function sayHello(): string{
return “Hello”;
}
```

2.2.1.1 Kontrola toka

Kontrola toka u TypeScript programskom jeziku obavlja se if/else grananjem, for, while i do/while petljama. For petlja ima tri formata; Prvi format (for(...;...;...)) izvršava deo programa na osnovu zadatog početnog i krajnjeg uslova i koraka kojim se petlja pokreće. Kao i u većini programskih jezika if/else grananje služi za jednostavnu kontrolu toka programa na osnovu zadatih uslov

Primer if/else grananja:

```
if (a > 5 && a <=10){
```

```
        console.log(“Passed”);
}else{
        console.log(“Failed”);
}
```

Drugi format (for(... in …)) koristi se za prolazak kroz elemente liste, niza ili rečnika pri čemu vraća indeks elementa strukture u toku iteracije.
Treći format (for(... of …)) takođe se koristi za prolazak kroz elemente liste, niza ili rečnika tako što čita vrednost elemenata tih struktura.

Primer ispisa i-tog elementa niza a korišćenjem for petlje:

```
for(var i = 0; i < 5; i++){
        console.log(a[i]);
}

for(var a_index in a){
        console.log(a[a_index);
}

for(var element in a){
        console.log(element);
}
```

Do/while i while petlja se koristi za izvršavanje naredbi dok je odredjen uslov ispunjen. Razlika između while i do/while petlje je u tome što do/while petlja sigurno izvršava instrukcije bar jednom.
Primer do/while petlje:

```
let a = 0;
while(a < 5){
        console.log(a);
        a++;
}

let a = 0;
do{
console.log(a);
        a++;
} while(a<5);
```

2.2.2 Struktura Angular aplikacije

Angular aplikacija se sastoji od jedne ili više komponenti. Komponenta u sebi sadrži klasu definisanu u TypeScript-u (koja je anotirana @Component() dekoratorom), HTML template i CSS stilove. Svaka komponenta ima definisan HTML template tag, koji je

moguće upotrebiti u okviru drugih komponenti u aplikaciji. HTML template je nadogradnja HTML markup-a koja omogućava dinamičko dodavanje sadržaja u HTML dokumentu.

Primer HTML template-a i propratne TypeScript klase:

```
<button type=”button” (click)=”sayHello()”>{{this.buttonText}}</button>
```

```
import { Component } from '@angular/core';

@Component ({
  selector: 'hello-button’,
  templateUrl: './hello-button.component.html'
})
export class HelloWorldInterpolationComponent {
        buttonText:string = “Click me”;

        sayHello():void{
        console.log(“Hello”);
        }
}
```

2.3 Docker

Docker je alat za virtuelizaciju na nivou operativnog sistema. Arhitektura Docker alata izgrađena na Linux operativnom sistemu koristi njegove već implementirane funkcionalnosti za izolaciju procesa (kernel namespaces i cgroups).
Kontejner je standardna jedinica softvera koja pakuje kod i sve njegove zavisnosti. Slika Docker kontejnera je samostalan, izvršni paket softvera koji uključuje sve što je potrebno za pokretanje aplikacije. Slike kontejnera postaju kontejneri tokom izvršavanja, a u slučaju Docker kontejnera – slike postaju kontejneri kada se pokreću na Docker Engine-u. Kontejneri izoluju softver od njegovog okruženja.

2.3.1 Docker Volumes i bind mounts

Bind mount je vezivanje strukture fajl sistema docker kontejnera sa folderom na mašini na kojoj je pokrenut kontejner. Da bi bind mount mogao da radi, potrebno je da mašina ima istu fajl strukturu kao i docker kontejner.
Docker Volume (Docker disk) je struktura u kojoj Docker čuva podatke koji se generišu u okviru kontejnera. Za razliku od bind mount-a, Docker disk ne zahteva da mašina ima istu fajl strukturu kao Docker kontejner. Kreiranje i upravljanje diskovima je moguće uz pomoć ugrađenih docker komandi. Zbog osobine da diskovi ne zavise od fajl sistema, moguće ih je pokrenuti na bilo kojoj mašini.

2.3.2 Docker Network

Docker koristi iptables pravila za podešavanje i izolaciju mreže u okviru kontejnera. Zbog toga, aplikacije pokrenute u okviru kontejnera ne zavise od podešavanja mreže na mašini na kojoj su pokrenute. Različiti drajveri koje implementira docker daju različite vrste podešavanja.
Glavni drajver je bridge. Ovaj drajver se koristi za kontejnere koji ne moraju međusobno da komuniciraju. Za druge, naprednije vrste komunikacija, postoje i host, overlay, ipvlan i macvlan drajveri. Ovi drajveri omogućavaju komunikaciju između više kontejnera, kao i napredna podešavanja i mac adresa. Ako se specificira da je drajver none, docker kontejner neće imati mogućnost komunikacije preko mreže.

2.3.3 Dockerfile

Da bi Docker mogao da kreira sliku instrukcije potrebne za kreiranje se pišu u tekstualnoj datoteci, Dockerfile. Naredba docker build pokreće kreiranje slike. U Docker fajlu koriste se direktive koje opisuju kako će se slika kreirati. Neke od bitnijih direktiva su:

Direktiva	Opis
FROM *naziv_slike*	Određuje osnovnu sliku za kreiranje tekuće slike
RUN *komanda*	Pokreće komande u toku stvaranja slike
COPY *fajl odredište*	Kopira datoteku u odredišni folder u okviru Docker slike
EXPOSE *port*	Otvara port u okviru Docker Network-a ka ostalim procesima operativnog sistema
WORKDIR *folder*	Određuje radnu datoteku slike
CMD [“*komanda*”]	Određuje komandu koja će se izvršiti na početku rada kontejnera

2.3.4 Docker Compose

Docker compose je alat koji se koristi za definisanje i pokretanje aplikacija koje koriste više Docker kontejnera. Definisanje instrukcija za pokretanje kontejnera pravi se u YAML fajlu. docker-compose build i docker-compose up komande se koriste za build-ovanje, odnosno pokretanje kontejnera. Podrazumevani fajl u kom su definisane instrukcije je docker-compose.yml. Neke od instrukcija koje se koriste u docker compose fajlu su.

version	verzija Docker Compose alata koja se koristi
services	imena servisa (kontejnera) koji treba da budu pokrenuti
image	ime slike koja treba da bude pokrenuta

build	putanja do docker fajla ako je potrebno build-ovati sliku
volumes	spisak imena diskova koje compose koristi
ports	port-ovi koji su otvoreni ka ostatku sistema
depends_on	određuje redosled kojim kontejneri treba da se pokrenu

Primer docker-compose.yml fajla

```
version: "3.7"
services:
    database:
        image: postgres:latest
        ports:
          - 5432:5432

    adminer:
        image: adminer:latest
        depends_on: database
        ports:
          - 80:8080
```

2.4 NGINX

NGINX je web server i load balancer. Koristi se i za serviranje statičkih fajlova. Podešavanje servera se vrši preko datoteke nginx.conf. Podrazumevano, ova datoteka se nalazi na putanji usr/local/nginx/conf, /etc/nginx ili /usr/local/etc/nginx.
nginx.conf datoteka sadrži direktive potrebne serveru da bi izvršio operacije. Kontekst neke direktive smešten je u hijerarhiju preko vitičastih zagrada.
Direktive koje se mogu naći u nginx fajlu su:

server	definiše adresu i druge parametre servera
location	mapira url na određenu putanju
root	mapira lokaciju izvornog fajla

2.5 Google Drive API

Google Drive API se koristi za interakciju sa servisima koje nudi Google Drive. API je dostupan u nekoliko programskih jezika: JavaScript, Java, Python… Funkcije koje API pruža su kreiranje, brisanje i izmena fajlova na Google Drive-u, kao i rad sa Google Docs Editors aplikacijama.

2.6 AWS - biće dodato

3. Implementacija sistema

3.1 Implementacija serverske aplikacije

Serverska aplikacija napravljena je korišćenjem Django framework-a, biblioteke djangorestframework i Google Drive API. Kontejnerizacija aplikacije je obavljena u Docker-u. U daljem tekstu dat je opis implementacije rešenja.

3.1.1 Model podataka

U programu postoje sledeći modeli:

- AppUser - modeluje korisnika aplikacije. Nasleđuje klasu django.contrib.auth.models.AbstractBaseUser. Ova klasa sadrži prototip za kreiranje korisnika aplikacije.
- LogEntry - modeluje podatke unete ili izmenjene od strane korisnika
- FileUpload - modeluje fajl koji je upload-ovan na server

Polja koja su korišćena za opis modela su:

- CharField(): Modeluje polje koje sarži tekst
- EmailField(): Modeluje polje koje sadrži email adresu
- DateTimeField(): Modeluje polje koje u sebi sadrži podatak o vremenu
- BooleanField(): Modeluje polje koje može da ima vrednost true ili false
- IntegerField(): Modeluje polje koje sadrži brojnu vrednost
- FileField(): Modeluje putanju fajla upload-ovanog na server
- ForeignKey(): Modeluje relaciju jedan prema više

Klase AppUser i LogEntry čuvaju se trajno u bazi podataka, dok klasa FileUpload služi za mapiranje i privremeno čuvanje putanje fajla na serveru.

```
class AppUser(AbstractBaseUser):
  email = models.EmailField(max_length=100, unique=True)
  first_name = models.CharField(max_length=100, unique=False)
  last_name = models.CharField(max_length=100, unique=False)
  date_joined = models.DateTimeField(auto_now_add=True)
  last_login = models.DateTimeField(auto_now=True)
  is_admin = models.BooleanField(default=False)
  is_active = models.BooleanField(default=True)
  is_staff = models.BooleanField(default=False)
```

```
    is_superuser = models.BooleanField(default=False)

class LogEntry(models.Model):
    user = models.ForeignKey(AppUser, on_delete=models.CASCADE)
    date = models.DateTimeField(auto_now=True)
    index_number = models.CharField(max_length=15)
    last_name = models.CharField(max_length=100)
    first_name = models.CharField(max_length=100)
    t1234 = models.IntegerField()
    sov = models.IntegerField()
    p1 = models.IntegerField()
    p2 = models.IntegerField()
    exam = models.IntegerField()

class FileUpload(models.Model):
    file = models.FileField(upload_to='files')
```

Za korisnika aplikacije kreirana je i AppUserManager klasa. Klasa nasleđuje ugrađenu klasu django.contrib.auth.models.BaseUserManager. AppUserManager klasa definiše način na koji će se kreirati korisnik i superuser aplikacije. Funkcijom django.contrib.hashers.make_password() omogućeno je čuvanje heširane lozinke.

```
class AppUserManager(BaseUserManager):

  def create_user(self, email, first_name, last_name, password):
      …

            user = self.model(email=self.normalize_email(email), first_name=first_name, last_name=last_name)
      user.set_password(make_password(password))
      user.save(using=self._db)
      return user

  def create_superuser(self, user):
      user.is_superuser = True
      user.is_admin = True
      user.is_staff = True
      user.save(using=self._db)

      return user
```

3.1.2 Serijalizatori podataka

S obzirom da se podaci potrebni za kreiranje instance klase FileUpload ne prosleđuju putem REST zahteva, za ovu klasu nije bilo potrebno kreirati serijalizatore.
Klasa AppUser ima implementirane tri vrste serijalizatora: UserSerializer, EmailSerializer, SuperUserSerializer. UserSerializer se koristi za serijalizaciju podataka vezanih za običnog korisnika. EmailSerializer serializuje samo email i id korisnika. SuperUserSerializer ima mogućnost da serializuje sva polja vezana za superuser-a aplikacije.

```
class UserSerializer(serializers.ModelSerializer):
  class Meta:
     model = AppUser
     fields = ('id', 'email', 'first_name', 'last_name', 'password')
     extra_kwargs = {'password':{'write_only': True}}

  def create(self, validated_data):
     validated_data['password'] = make_password(validated_data.get('password'))
     return super(UserSerializer, self).create(validated_data)


class EmailSerializer(serializers.ModelSerializer):
  class Meta:
 class Meta:
     model = AppUser
     fields = ['id', 'email']


class SuperUserSerializer(serializers.ModelSerializer):     model = AppUser
     fields = ['id', 'email']

  class Meta:
     model = AppUser
      fields = ('id', 'email', 'first_name', 'last_name', 'password', 'is_admin', 'is_superuser',
'is_staff')
     extra_kwargs = {'password': {'write_only': True}}

  def create(self, validated_data):
     validated_data['password'] = make_password(validated_data.get('password'))
     return super(SuperUserSerializer, self).create(validated_data)
```

Klasa LogEntriy ima implementirana dva serijalizatora: LogEntrySerializer i TableSerializer. Razlika između ova dva serijalizatora je samo u poljima user i date. LogEntrySerializer serijalizuje podatke vezane za podatke o izmenama. TableSerializer se koristi za statistiku izmena koju vidi korisnik, pa podaci o datumu izmene i korisniku nisu potrebni.

```
class LogEntrySerializer(serializers.ModelSerializer):
  user = EmailSerializer()

  class Meta:
    model = LogEntry
    fields = "__all__"


class TableSerializer(serializers.ModelSerializer):
  class Meta:
    model = LogEntry
    fields = ['index_number', 'first_name', 'last_name', 't1234', 'sov', 'p1', 'p2', 'exam']
```

3.1.3 Rad sa korisnicima programa

Kao što je već rečeno, u aplikaciji postoje dve vrste korisnika: običan korisnik i superuser. Superuser ima mogućnost da vidi podatke koji se čuvaju u log-u, kao i da dodaje nove korisnike aplikacije. Prilikom pokretanja aplikacije prvi put, potrebno je ručno dodati superuser-a aplikacije. Funkcija user_count() proverava broj korisnika aplikacije. Operacije nad korisnicima, koje nisu dodavanje prvog superusera, nisu moguć ako u aplikaciji ne postoji ni jedan korisnik. Broj korisnika aplikacije se pamti u request sesiji.

```
def user_count():
  return AppUser.objects.all().count()


def check_user_initialization(request):
  if request.method == "GET":
    users = AppUser.objects.all()
    count = users.count()
    request.session['count'] = count
    return JsonResponse(data={"count": count}, status=200)

  return HttpResponse(status=400)


def create_superuser(request):
  count = request.session.get('count')
  if count is None:
    request.session['count'] = user_count()
    count = request.session.get('count')

  if count != 0:
    return HttpResponse(status=403)
```

```
    if request.method == "POST":
        data = JSONParser().parse(request)
        serializer = SuperUserSerializer(data=data)
        if serializer.is_valid():
            serializer.save()
            request.session['count'] = request.session.get('count')+1
            return JsonResponse(serializer.data, status=200)

    return HttpResponse(status=400)

def create_user(request):
    count = request.session.get('count')
    if count is None:
        request.session['count'] = user_count()
        count = request.session.get('count')

    if request.method == "POST":
        body = request.body.decode('utf-8')
        body = loads(body)
        new_user = body[0]
        admin = body[1]

        try:
            admin_user = AppUser.objects.get(email=admin['email'])
        except AppUser.DoesNotExist:
            return HttpResponse(status=404)

        if not admin_user.is_admin:
            return HttpResponse(status=403)

        if check_password(admin['password'], admin_user.password):
            serializer = UserSerializer(data=new_user)
            if serializer.is_valid():
                serializer.save()
                return JsonResponse(serializer.data, status=201)
            else:
                return HttpResponse(status=400)

        return HttpResponse(status=403)
```

Sa serverske strane prijavljivanje na aplikaciju se vrši preko funkcije user_login(). Nakon slanja zahteva proveravaju se email i password korisnika. Funkcija django.contrib.auth.hashers.check_password() proverava ispravnost heša unete lozinke.

```
@csrf_exempt
def user_login(request):
  if request.method == "POST":
     body = request.body.decode('utf-8')
     body = loads(body)
     email = body['email']
     password = body['password']
     try:
        user = AppUser.objects.get(email=email)
     except AppUser.DoesNotExist:
        return HttpResponse(status=404)

     if check_password(password, user.password):
        print(user.email)
        if user.is_admin:
           serializer = SuperUserSerializer(user)
        else:
           serializer = UserSerializer(user)

        return JsonResponse(serializer.data, status=200)

  return HttpResponse(status=403)
```

3.1.4 Rad sa unetim podacima

Rad sa unetim podacima se sastoji iz tri dela: preuzimanje i parsiranje podataka poslatih u zahtevu, obrada unetih podataka i postavljanje podataka na Google Sheet putem Google Drive API-ja.

3.1.4.1 Rad sa csv i zip fajlovima

Korisnik, u zavisnosti od vrste testa čije rezultate unosi, na server šalje ili .csv ili .zip datoteku. Datoteke se šalju u string base64 formatu. Nakon što se datoteka prevede iz string u binarni format, privremeno se čuva kao statički fajl, a njena putanja u bazi podataka. Nakon što je završena obrada datoteke, ona se briše iz baze podataka i sa servera. Pandas dataframe podržava direktno učitavanje .csv datoteka funkcijom pandas.read_csv(). Struktura svakog .zip fajla je takva da u njoj postoji jedna .csv datoteka. Pošto pandas ne podržava direkno čitanje .zip datoteke, potrebno je prvo ekstrahovati .csv datoteku. Za ekstrakciju .csv datoteke korišćena je biblioteka ZipFile. Ona poseduje funkcije za čitanje, pregled strukture i ekstrahovanje .zip fajlova.

Funkcija parse_file_data, koja pravi fajl i stavlja ga na server:

```
def parse_file_data(file_data, user, file_type):
  file = FileUpload.objects.create()
  format, file_str = file_data.split(';base64,')
  ext = format.split('/')[-1]
  file_name = user + file_type
  file.file = ContentFile(base64.b64decode(file_str), name=file_name)
  file.save()

  file_path = 'files/' + file_name

  return file, file_path
```

Rad sa .zip datotekom i pronalaženje .csv datoteke:

```
   for file_name in list_of_files:
        if file_name.endswith('.csv'):
           csv_path = 'files/' + file_name
           zf.extract(file_name, 'files')
           break

     zf.close()
     file.delete()

     if not csv_path:
        return HttpResponse(status=400)
     exam_type = int(body[0]['examType'])
     df = pd.read_csv(csv_path)
     dfs = prepare_df_data(df, exam_type, False)
```

3.1.4.2 Obrada unetih podataka

Kao što je već rečeno, obrada unetih podataka obavlja se putem pandas biblioteke. Nakon što se od unete datoteke napravi DataFrame nad njim se vrši niz neophodnih operacija. Tabela koja se čuva na centralnom Google Sheet dokumentu ima različit način imenovanja kolona. Zbog toga je potrebno prvo formatirati imena kolona i ukloniti nepotrebne kolone iz DataFrame-a. Format broja indeksa, koji se koristi kao ključ po kome se podaci upisuju u tabelu, je drugačiji na centralnom Google Sheet-u i u .csv fajul. Parsiranje i reformatiranje broja indeksa izvršeno je putem regex izraza. Regex izraz očitava smer, broj upisa i godinu iz indeksa i čuva te podatke za dalju upotrebu. Na osnovu godine upisa određuje se na koji list u centralnom Google Sheet-u je potrebno upisati podatke. Program u svom settings.py fajlu sadrži promenljivu

CURRENT_YEAR koja čuva podatak o tekućoj školskoj godini. Studenti upisani pre ove školske godine se razrstavaju na poseban list. Ostali studenti se razvrstavaju na osnovu smera na koji su upisani.

Funkcija prepare_df_data koja vrši prpremu podataka za upis u Google Sheet.

```
def prepare_df_data(df, exam_type, test):
  …

  if test:
     df['04. Ime'] = df['Ime i prezime'].str.split(' ', expand=True)[0]
     df['03. Prezime'] = df['Ime i prezime'].str.split(' ', expand=True)[1]
     df = df.rename(columns={"Broj poena": et, "Broj indeksa":"02. Broj indeksa"})
     df = df.loc[:, df.columns.intersection(['04. Ime', '03. Prezime', et, "02. Broj indeksa"])]

  else:
       df = df.rename(columns={"indeks": "02. Broj indeksa", "ime": "04. Ime", "prezime":
"03. Prezime", "poeni": et})
     df = df.drop(columns=['ukupno_poena', 'ip', 'datum'])

  for index, row in df.iterrows():
     section = re.split(r'\d{1,3}-\d{4}', df['02. Broj indeksa'][index])[0]
     no = re.split(r'-\d{4}', df['02. Broj indeksa'][index])[0]
     no = re.split(r'[aA-zZ]{2}', no)[1]
     year = re.split(r'\d{1,3}-', df['02. Broj indeksa'][index])[1]
     index_format = section.upper() + ' ' + no + '/' + year
     df.loc[index, '02. Broj indeksa'] = index_format

  current_year = settings.CURRENT_YEAR

        df_ra = df[df['02. Broj indeksa'].str.contains(fr'(?=RA.*)(?=.*{current_year})',
regex=True)]
       df_psi = df[df['02. Broj indeksa'].str.contains(fr'(?=PR.*)(?=.*{current_year})',
regex=True)]
  df_old = df[~df['02. Broj indeksa'].str.contains(fr'(?=.*{current_year})', regex=True)]

 …
```

Nakon očitavanja podataka iz centralnog Google Sheet-a i njihovog prevođenja u DataFrame, uneti podaci se upoređuju sa njima. Funkcija pandas.merge() i pozivanje agregatora .groupby() vrši spajanje podataka dva DataFrame-a na osnovu broja indeksa. Ovakvim spajanjem se ažuriraju podaci u odgovarajućem redu tabele.

Funkcija prepare_sheet_writing koja spaja dve tabele i vraća podatke u odogovarajućem obliku.

```
def prepare_sheet_writing(sheet_vals, df):
  sheet_df = pd.DataFrame(sheet_vals[1:], columns=sheet_vals[0])
   new_df_outer = pd.merge(sheet_df, df, left_on="02. Broj indeksa", right_on="02. Broj indeksa",
                           how="outer").groupby(
     lambda x: x.split('_')[0], axis=1).last()
  new_df_outer = new_df_outer.where(pd.notnull(new_df_outer), "")

  sheet_body = new_df_outer.columns.tolist()
  vals = new_df_outer.values.tolist()
  vals.insert(0, sheet_body)

  return {'values': vals}
```

3.1.4.3 Rad sa Google Sheet tabelama

Za rad sa Google Sheet tabelama potrebno je koristiti Google Drive API. google_auth_oauthlib.flow biblioteka sadrži funkcije i klase za autorizaciju korisnika. Klasa InstalledAppFlow iz ove biblioteke implementira funkciju from_client_secrets_file uzima putanju do ključa kojim se vrši autentifikacija korisnika. Putanja do ovog fajla čita se iz environment variable. Nakon što je autentifikacija uspešno izvšena moguće je vršiti operacije nad Google Sheet tabelom. Funkcije open_sheet_for_reading i open_sheet_for_writing implementirane su tako da omoguće čitanje tabele prema unetom id-ju. Upis i čitanje tabele se vrši na osnovu imena listova u tabelama.

Funkcije za čitanje i upis u tabelu.

```
def open_sheet_for_reading(service, range, sheet_id):
  sheet = service.spreadsheets()
  result = sheet.values().get(spreadsheetId=sheet_id, range=range).execute()
  return result.get('values', [])


def open_sheet_for_writing(service, range, sheet_id, body):
  sheet = service.spreadsheets()
  sheet.values().update(spreadsheetId=sheet_id, range=range,
                        body=body, valueInputOption="RAW").execute()
```

Funkcija koja nakon obrade rezultata testa vrši upis podataka u tabelu na Google Drive-u.

```
def test_results(request):
        …

        scopes = ['https://www.googleapis.com/auth/drive']
        key_path = environ.get("GOOGLE_KEY_PATH")
        api_port = environ.get("GOOGLE_API_PORT")
        if api_port:
           api_port = int(api_port)
        else:
           api_port = 8001
        flow = InstalledAppFlow.from_client_secrets_file(key_path, scopes)
        flow.redirect_uri = environ.get("REDIRECT_URI")
        credentials = flow.run_local_server(port=api_port)
        service = build('sheets', 'v4', credentials=credentials)
                            sheet_vals_ra = open_sheet_for_reading(service, range_ra,
environ.get("SHEET_ID"))
                           sheet_vals_psi = open_sheet_for_reading(service, range_psi,
environ.get("SHEET_ID"))
                          sheet_vals_old = open_sheet_for_reading(service, range_old,
environ.get("SHEET_ID"))

        sheet_body_ra = prepare_sheet_writing(sheet_vals_ra, dfs[0])
        sheet_body_psi = prepare_sheet_writing(sheet_vals_psi, dfs[1])
        sheet_body_old = prepare_sheet_writing(sheet_vals_old, dfs[2])

                        open_sheet_for_writing(service, range_ra, environ.get("SHEET_ID"),
sheet_body_ra)
                      open_sheet_for_writing(service, range_psi, environ.get("SHEET_ID"),
sheet_body_psi)
                      open_sheet_for_writing(service, range_old, environ.get("SHEET_ID"),
sheet_body_old)
        …
```

3.1.5 Log i statistika

Log korišćenja aplikacije se čuva nakon upisa podataka na centralni Google Sheet. Log sadrži sve unete izmene. Pošto je potrebno kreirati više od jedne instance klase LogEntry, koristi se funkcija bulk_create koju implementiraju objekti modela.

Funkcija koja čuva izmene u programu.

```
def add_log(dfs, user_email):
  …
```

```
for df in dfs:
    for index, row in df.iterrows():

        …

        table_instances.append(LogEntry(index_number=row['02. Broj indeksa'],
                                        user=user.first(),
                                        last_name=row['03. Prezime'],
                                        first_name=row['04. Ime'],
                                        t1234=t1234,
                                        sov=sov,
                                        p1=p1,
                                        p2=p2,
                                        exam=exam))

    LogEntry.objects.bulk_create(table_instances)
```

Statistika koju korisnik vidi su svi podaci koji postoje u centralnom Google Sheet-u. Na klijentski deo aplikacije potrebno je poslati sve podatke iz centralnog Google Sheet-a. Funkcija get_statistics se svodi na čitanje podataka iz Google Sheet tabele.

Funkcija za slanje statistike na klijentski deo aplikacije:

```
def get_statistics(request):
        …
                        sheet_vals_ra = open_sheet_for_reading(service, range_ra, environ.get("SHEET_ID"))
                        sheet_vals_psi = open_sheet_for_reading(service, range_psi, environ.get("SHEET_ID"))
                        sheet_vals_old = open_sheet_for_reading(service, range_old, environ.get("SHEET_ID"))

    sheet_df_ra = pd.DataFrame(sheet_vals_ra[1:], columns=sheet_vals_ra[0])
    sheet_df_psi = pd.DataFrame(sheet_vals_psi[1:], columns=sheet_vals_psi[0])
    sheet_df_old = pd.DataFrame(sheet_vals_old[1:], columns=sheet_vals_old[0])

    dfs = [sheet_df_ra, sheet_df_psi, sheet_df_old]
    stats = stats_serializer(dfs)
    stats = TableSerializer(stats, many=True)
    return JsonResponse(data=stats.data, safe=False, status=200)
```

3.2 Implementacija klijentske aplikacije

Klijentska aplikacija implementirana je upotrebom Angular framework-a. Ona služi za posrednu interakciju korisnika sa centralnim Google Sheet-om. U klijentskoj aplikaciji implementirane su funkcije za dodavanje korisnika, unos i upload podataka na server, kao i prikaz logova i statistike.

3.2.1 Model podataka

Model podataka implementiran je putem interfejsa AppUser, Log, Count i ResultsFormat. Osim interfejsa Count, model se poklapa sa modelom na serverskoj strani. Interfejs Count služi za proveru trenutnog broja korisnika u aplikaciji.

Modeli implementirani u klijentskoj aplikaciji

```
export interface AppUser{
 id?:number;
 email:string;
 first_name:string;
 last_name:string;
 password?:string;
 is_admin?:boolean;
 is_staff?:boolean;
 is_superuser?:boolean;

}


export interface Log{
 id?:number,
 date:Date,
 index_number:string,
 last_name:string,
 first_name:string,
 t1234?:number,
 sov?:number,
 p1?:number,
 p2?:number,
 exam?:number,
 user?:AppUser,
 userStr?:string


}

export interface Count{
```

```
 count:number;
}

export interface ResultsFormat{
 resultType:string;
 examType?:string;
 testType?:string;
 file: string | ArrayBuffer | null;
}
```

3.2.2 Rad sa korisnicima

Da bi bio omogućen rad sa ostalim funkcionalnostima aplikacije, tokom prvog pokretanja, potrebno je dodati superuser-a. Unos podataka za kreiranje superuser-a vrši se putem forme. Forma je mapirana na model korisnika. U TypeScript fajlu komponente definisan je način slanja REST zahteva na serverski deo aplikacije. REST zahtev se šalje putem klase HttpClient. Ova klasa stvara publish subscribe relaciju za REST zahtev.

Forma za unos superuser-a:

```
 <div class = "row">
   <div class="offset-3"></div>
   <div class="col-md-6">
     <br/>
    <form name = "userForm"  (ngSubmit) = "sendUser()" novalidate>
      <div class = "form-group">
       <label>Name: </label>
          <input type = "text" [(ngModel)] = "userModel.first_name" id = "gName" class = "form-control" name = "name"/>
      </div>
      <div class = "form-group">
       <label>Surname: </label>
         <input type = "text" [(ngModel)]="userModel.last_name" id = "gSurname" class = "form-control" name = "surname"/>
      </div>
      <div class = "form-group">
       <label>Email: </label>
            <input type = "email" [(ngModel)]="userModel.email" id = "gEmail" class = "form-control" name = "email"/>
      </div>
      <div class = "form-group">
       <label>Password: </label>
          <input type = "password" [(ngModel)]="userModel.password" id = "gPassword" class = "form-control" name = "password"/>
      </div>
```

```
      <br/>
      <button type = "submit" class = "btn btn-primary">Add</button>
    </form>
  </div>
 </div>
```

Klasa kojom se šalje zahtev na serverski deo aplikacije:

```
export class SuperuserComponent implements OnInit {
…

 sendUser():void{
   this.userObservable(this.userModel).subscribe(
     (res:AppUser) => {
       console.log(res);
       location.href='index';
     },
     err => {
       alert("Something went wrong");
       console.log(err.message);
     }
   )
 }



 userObservable(appUser:AppUser){
   let url = "http://";
   let server = "localhost"
   let port = "8000";
   url = url + server + ":" + port + "/create_superuser/";
   return this.http.post<AppUser>(url, appUser);
 }
}
```

Superuser aplikacije ima mogućnost da dodaje druge korisnike aplikacije. Forma slična onoj za dodavanje superuser-a postoji i za dodavanje drugih korisnika aplikacije. Tokom dodavanja korisnika traži se autentifikacija superuser-a koji dodaje novog korisnika.

3.2.3 Unos podataka

Forma za unos podataka sadrži polje i radio button-e za za izbor tipa testa za koji se unose rezultati i polje za upload fajla. Polje za upload fajla učitava fajl sa uređaja korisnika i prosleđuje ga u tekstualnom formatu na server.

Deo forme u kome je implementiran izbor ispita i upload fajla.

```
…
      <label>Results Type</label>
                  <select id="sType" class="form-control" [(ngModel)]="selectedType" [ngModelOptions]="{standalone:true}">
        <option *ngFor="let type of resultTypes; let i = index;" [value]="resultTypes[i]">
         {{resultTypes[i]}}
        </option>
      </select>
     </div>
     <div class = "form-group">
      <label>File: </label>
                   <input name="file" type = "file" id="gFile" class="form-control" (change)="fileChange($event)"/>
     </div>
     <div class="form-group" *ngIf="selectedType === 'Exam Result' ">
      <label>Partial 1</label>
      <br/>
                                <input type="radio" name="examType" [value]="1" [(ngModel)]="selectedExamType" id="partial1" [disabled]="selectedType!=='Exam Result'">
      <br/>
      <label>Partial 2</label>
      <br/>
                                <input type="radio" name="examType" [value]="2" [(ngModel)]="selectedExamType" id="partial2" [disabled]="selectedType!=='Exam Result'">
      <br/>
      <label>Full Exam</label>
      <br/>
                    <input type="radio" name="examType" id="fullExam" [value]="3" [(ngModel)]="selectedExamType" [disabled]="selectedType!=='Exam Result'">
      <br/>
     </div>
…
```

3.2.4 Prikaz statistike i log-a

Statistiku se računa na osnovu svih podataka iz centralnog Google Sheet-a. Korisnik ima mogućnost da vidi dve vrste podataka. Sve podatke o studentu, zajedno sa ocenama ili

statistiku na nivou predmeta (prolaznost, prosečna ocena). Na klijentskoj strani aplikacije su implementirane funkcije za računanje prolaznosti i prosečne ocene

Tabela koja prikazuje podatke o studentima:

```
…
<div class="container" *ngIf="selectedOption==='Student Data'">
 <div class = "container">
  <table class="table table-striped table-responsive-md btn-table" border="1">
    <thead>
    <title>Changelog</title>
    <tr>
      <th scope="col">Index number</th>
      <th scope="col">Last name</th>
      <th scope="col">First name</th>
      <th scope="col">T1234</th>
      <th scope="col">SOV</th>
      <th scope="col">P1</th>
      <th scope="col">P2</th>
      <th scope="col">Exam</th>
      <th scope="col">Grade</th>

    </tr>
    </thead>
    <tbody>
    <tr *ngFor="let d of allData">
      <td>{{d.index_number}}</td>
      <td>{{d.last_name}}</td>
      <td>{{d.first_name}}</td>
      <td>{{d.t1234}}</td>
      <td>{{d.sov}}</td>
      <td>{{d.p1}}</td>
      <td>{{d.p2}}</td>
      <td>{{d.exam}}</td>
      <td>{{calculateGrade(d)}}</td>
    </tr>
    </tbody>
  </table>
 </div>
```

Prikaz statističkih podataka na nivou predmeta:

```
</div>
<div class="container" *ngIf="selectedOption==='Averages'">
 <p>Average Grade: <p>
```

```
 <p>{{this.averageGrade()}}</p>
 <p>Passed: <p>
 <p>{{this.passed()}}</p>
 <p>Failed: <p>
 <p>{{this.failed()}}</p>
 <p>Percent Passed: <p>
 <p>{{this.percentPassed()}}</p>
…
</div>
```

Računanje statističkih podataka:

```
export class StatisticsPageComponent implements OnInit {

…
 calculateGrade(d:Log):number{
   let sum = 0;
   let i = 0;

   if(d.exam)
     if (d.exam > 0)
       sum += d.exam;

   if(d.p1)
     if(d.p1 >0)
       sum += d.p1;

   if(d.p2)
     if(d.p2>0)
       sum += d.p2;

   if(d.sov)
     if(d.sov>0)
       sum += d.sov;

   if(d.t1234)
     if(d.t1234>0)
       sum+= d.t1234;


   if (sum >= 91 && sum <= 100) {
     this.grades[5]++;
     return 10;
   }
```

```
  else if(sum >= 81 && sum <= 90) {
    this.grades[4]++;
    return 9;
  }
  else if(sum >= 71 && sum <= 80) {
    this.grades[3]++;
    return 8;
  }
  else if(sum >= 61 && sum <=70) {
    this.grades[2]++;
    return 7;
  }
  else if(sum >=51 && sum <=60) {
    this.grades[1]++;
    return 6;
  }
  else {
    this.grades[0]++;
    return 5
  }
}

 averageGrade(){
       let total = this.grades[5] + this.grades[4] + this.grades[3] +this.grades[2] +
this.grades[1];
    let sum = this.grades[5]*10 + this.grades[4]*9 + this.grades[3]*8 +this.grades[2]*7 +
this.grades[1]*6;
  return sum/total;
 }

 passed(){
  return  this.grades[5] + this.grades[4] + this.grades[3] +this.grades[2] + this.grades[1];
 }

 failed(){
  return this.grades[0]
 }

 percentPassed(){
       let total = this.grades[5] + this.grades[4] + this.grades[3] +this.grades[2] +
this.grades[1] + this.grades[0];
      let passed = this.grades[5] + this.grades[4] + this.grades[3] +this.grades[2] +
this.grades[1];
  return Math.round(((passed/total)*100)*100)/100;
```

```
 }
}
```

Log podatke može da vidi samo superuser aplikacije. Podaci koji se dobiju sa serverske strane aplikacije se smeštaju u tabelu.

Tabela za prikaz log fajla:

```
<div class = "container">
 <table class="table table-striped table-responsive-md btn-table" border="1">
  <thead>
  <title>Changelog</title>
  <tr>
    <th scope="col">Index number</th>
    <th scope="col">Last name</th>
    <th scope="col">First name</th>
    <th scope="col">T1234</th>
    <th scope="col">SOV</th>
    <th scope="col">P1</th>
    <th scope="col">P2</th>
    <th scope="col">Exam</th>
    <th scope="col">User</th>
    <th scope="col">Date</th>

  </tr>
  </thead>
  <tbody>
  <tr *ngFor="let log of logs">
    <td>{{log.index_number}}</td>
    <td>{{log.last_name}}</td>
    <td>{{log.first_name}}</td>
    <td>{{log.t1234}}</td>
    <td>{{log.sov}}</td>
    <td>{{log.p1}}</td>
    <td>{{log.p2}}</td>
    <td>{{log.exam}}</td>
    <td>{{log.userStr}}</td>
    <td>{{log.date}}</td>

  </tr>
  </tbody>
 </table>
```

** **

4. Deployment

4.1 Kontejnerizacija rešenja

Serverska i klijentska aplikacija su kontejnerizovane uz pomoć Docker i Docker compose alata. U CMD direktivi Dockerfile-a serverske aplikacije poziva se skript fajl koji vrši pokretanje produkcionog servera (Gunicorn) i migraciju modela u bazu podataka.

Dockerfile serverske aplikacije:

```
FROM python:3.10-bullseye

EXPOSE 8001

WORKDIR /usr/src

COPY requirements.txt .
RUN apt update
RUN pip install -r requirements.txt
COPY . /usr/src
COPY start_server.sh .

CMD ["./start_server.sh"]
```

Skript fajl za pokretanje produkcionog servera:

```
#!/bin/bash

cd pjisp &&
python manage.py makemigrations && python manage.py migrate --run-syncdb &&
python manage.py runserver 0.0.0.0:8000
```

Pre nego što je kontejnerizovana klijentska aplikacija, pozvana je komanda ng build, Angular framework-a. Ova komanda vrši prevođenje svih fajlova Angular aplikacije u HTML i JavaScript format. Produkcioni server koji se koristi za serviranje statičkih fajlova je NGINX. U konfiguracionom fajlu navedena je putanja do početne stranice. Takođe, navedena je i direktiva za rutiranje adresa.

Dockerfile klijentske aplikacije:

```
FROM nginx:1.23.1

EXPOSE 8080

COPY nginx.conf /etc/nginx/conf.d
RUN rm -rf /usr/share/nginx/html/*
COPY pjisp/dist/pjisp/ /usr/share/nginx/html
```

NGINX konfiguracioni fajl:

```
server {
 listen 8080;
 server_name localhost;
 root /usr/share/nginx/html;

 location /{
   try_files $uri $uri/ /index.html;

  }

}
```

Docker compose fajl objedinjuhje serversku i klijentsku aplikaciju, kao i bazu podataka i interfejs za upit podataka. Svi osetljivi podaci iz aplikacija stavljeni su u docker_vars.env fajl. Nalaze se u formatu ključ=vrednost.

Docker compose fajl:

```
version: "3.7"

services:
 pjisp_server:
   build: ./pjisp_server
   container_name: pjisp_server
   restart: on-failure:10
   depends_on:
     - postgres_db
   env_file:
     - ./secrets_folder/docker_vars.env
   ports:
     - "8000:8000"

 pjisp_ng:
   build: ./pjisp_ng
```

```
  container_name: pjisp_ng
  restart: on-failure:10
  links:
    - pjisp_server
  ports:
    - "8080:8080"

postgres_db:
  image: postgres:14-bullseye
  restart: always
  env_file:
    - ./secrets_folder/docker_vars.env
  ports:
    - "5432:5432"
  volumes:
    - ./data/postgres_data:/var/lib/postgresq/data

adminer:
  image: adminer
  restart: always
  ports:
    - "8081:8080"
```