

Sveučilište Jurja Dobrile u Puli

Fakultet Informatike u Puli

Antonio Labinjan

COMPUTER VISION ATTENDANCE SYSTEM

Dokumentacija

Pula, 2024.

Sveučilište Jurja Dobrile u Puli

Fakultet Informatike u Puli

COMPUTER VISION ATTENDANCE SYSTEM

Dokumentacija

Antonio Labinjan

JMBAG: 0303106891 redovan student

Kolegij: **Web-aplikacije**

Mentor: **doc. dr. sc. Nikola Tanković**

Pula, prosinac 2024.

Sadržaj

1. Sažetak	4
2. Uvod i motivacija	5
3. Use case i Class dijagrami sustava	6
4. Razrada funkcionalnosti	8
5. Moduli aplikacije	17
6. Sažeta specifikacija backenda	63
7. Popis tehnologija...	78
8. Upute za korištenje	79

1. Sažetak

Computer vision attendance system je aplikacija za evidentiranje prisutnosti pomoću prepoznavanja lica.

Ova aplikacija koristi computer vision za pojednostavljenje evidencije prisutnosti u obrazovnim institucijama. Studenti mogu bilježiti svoju prisutnost putem prepoznavanja lica u unaprijed definiranim intervalima, čime se uklanja potreba za ručnim ili papirnatim metodama. Profesori imaju intuitivno sučelje za upravljanje učenicima, predmetima i rasporedima te generiranje detaljnih izvještaja i statistika o prisutnosti kako bi pratili angažiranost učenika. Sustav osigurava točnost, smanjuje administrativni teret i pruža uvid u trendove prisutnosti. Razvijen s naglaskom na jednostavnost i učinkovitost, ovaj sustav integrira suvremenu tehnologiju kako bi unaprijedio iskustvo obrazovanja za studente i profesore.

Postoje 2 tipa korisnika; običan student koji unutar aplikacije može samo prijaviti prisutnost skeniranjem lica i profesor koji ima dodatne ovlasti koje će biti detaljnije razrađene u idućim poglavljima.

Omogućena je registracija profesora u sustav pomoću emaila i passworda. Nakon registracije, profesor može postaviti predmet koji se trenutno izvodi sa predviđenim datumom i vremenskim intervalom. Zatim, implementirane su funkcionalnosti dodavanja novih studenata u sustav pomoću ručnog uploada slika, ili pomoću izvlačenja frameova iz live snimke, ovisno o potrebi. Nadalje, omogućeno je automatizirano slanje obavijesti putem maila za svaku uspješnu prijavu prisustva korištenjem SendGrid API-ja.

Kod svake prijave prisustva također se provjerava se je li pred kamerom prava osoba ili slika pomoću jednostavne anti-spoofing mehanike. (postoje i neki napredniji API-ji za liveness check, ali su poprilično skupi).

Nakon postavljanja predmeta i dodavanja studenata, moguće je prijaviti prisutnost tako što aplikacija najprije provjeri koje je trenutno vrijeme i pripada li ono nekom od intervala za postavljene predmete. Zatim, kada student stane pred kameru, kreće složeni proces prepoznavanja koristeći OPENAI-jev CLIP model i Facebook-ov AI Similarity Search.

Što se tiče implementacije same aplikacije, korišten je Pythonov web-development framework Flask za backend, HTML za frontend te Sqlite za bazu podataka.

2. Uvod i motivacija

Ova aplikacija nastala je zbog želje za povezivanjem više različitih područja poput baza podataka, web-developmenta, računalnogvida i umjetne inteligencije u kompletan "proizvod" i zbog toga što me navedena područja izrazito zanimaju i definitivno će se njima (bar se nadam) baviti u budućnosti. Odabran je Python kao jezik izrade aplikacije zato što se iz mog dosadašnjeg iskustva čini kao najbolji izbor za bilo kakve zadatke vezane uz AI. Također, posebno je motivirajuća činjenica da je ova aplikacija prilično inovativna. (Naravno, nije prva aplikacija koja implementira computer vision za prepoznavanje ljudi i evidenciju prisutnosti, ali je prva koja koristi kombinaciju CLIP-a i FAISS-a).

Nadalje, ono što je zanimljivo, su svakako prilično dobri rezultati koje ona postiže. Na iznimno dobrim referentnim slikama, postiže se točnost od oko 100%. Korištenjem nešto manje kvalitetnih slika, rezultati opadaju (~73%), no možemo tvrditi da je aplikacija pri primarnom use-caseu gotovo savršena. Ako bismo uzeli 30-ak ljudi (ekvivalent jednom školskom razredu) i par njihovih slika za feed-anje u model, aplikacija bi ih identificirala u 100% slučajeva.

Naravno, nije sve bilo savršeno iz prvog pokušaja i bilo je potrebno puno eksperimentiranja, istraživanja, pokušaja i pogrešaka, te u trenutku pisanja dokumentacije još uvijek tražimo idealnu kombinaciju parametara koja će dati maksimalne rezultate i za lošiji dataset slika.

2.1. SWOT analiza

Strengths:

- **Inovativna tehnologija:** Korištenje computer visiona i modela poput CLIP-a donosi modernu i naprednu tehnologiju u obrazovni sektor i ostale ustanove gdje je evidencija prisutnosti od ključne važnosti
- **Automatizacija procesa:** Eliminira ručno vođenje evidencije o prisutnosti, čime štedi vrijeme nastavnicima i administraciji
- **Preciznost:** Modeli mogu prepoznati učenike čak i u velikim grupama, uz minimalne greške
- **Jednostavna integracija:** Može se lako prilagoditi različitim školskim sustavima
- **Smanjenje varanja:** Prepoznavanje lica smanjuje mogućnost lažnog prijavljivanja prisutnosti

Weaknesses

- **Ovisnost o kvaliteti dataseta:** Loši podaci smanjuju preciznost modela, što može dovesti do misklasifikacija i frustrirati korisnike
- **Privatnost i sigurnost:** Mogući problemi s GDPR-om i zaštitom osobnih podataka učenika, usprkos činjenici da se podaci strogo čuvaju i teoretski ne bi trebali izlaziti izvan granica sustava
- **Infrastrukturni zahtjevi:** Potreba za kamerama visoke kvalitete i stabilnom mrežom u ustanovama što u nekim slučajevima može predstavljati velik problem
- **Ograničenja u nepovoljnim uvjetima:** Loše osvjetljenje, pokretni subjekti i slabe kamere mogu utjecati na točnost

- **Troškovi:** Ustanove s ograničenim budžetom mogu imati poteškoća s implementacijom sustava

Opportunities

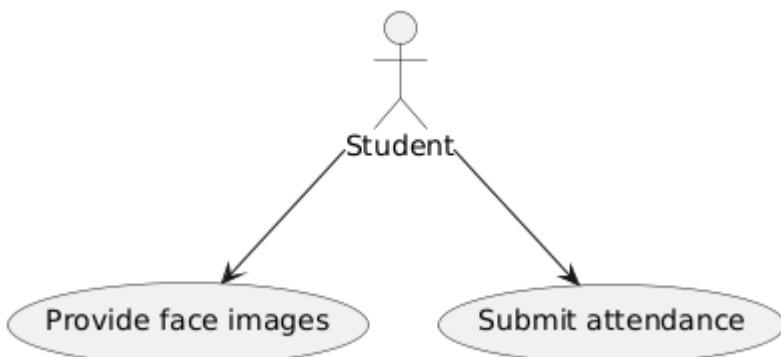
- **Širenje na druga tržišta:** Fakulteti, korporacije (praćenje zaposlenika), industrijske primjene (sigurnost i autorizacija)
- **Povećanje funkcionalnosti:** Integracija s dnevnicima, sustavima ocjenjivanja ili aplikacijama za roditelje
- **Prilagodba za druge namjene:** Npr. sigurnosne provjere, kontrola ulaza u objekte ili praćenje emocionalnog stanja učenika
- **Suradnja s vladinim institucijama:** Kao dio digitalizacije obrazovnog sustava
- **Razvoj freemium modela:** Osnovne funkcije besplatne, dok napredne zahtijevaju pretplatu

Threats

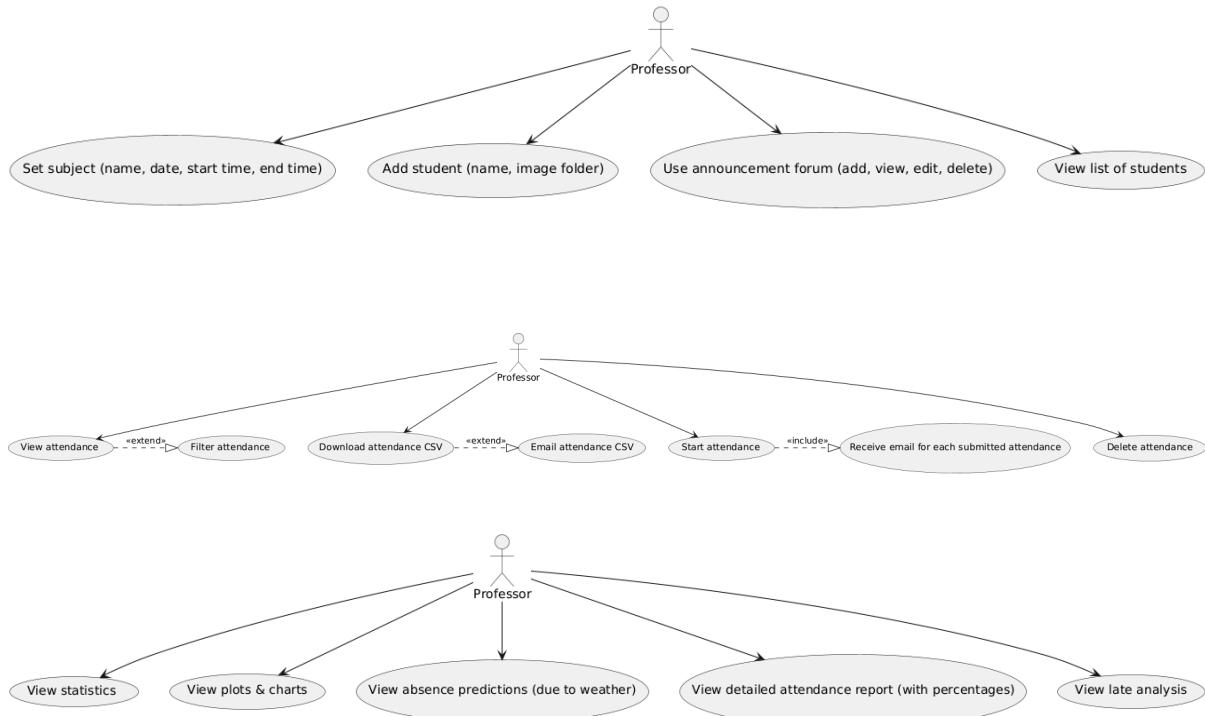
- **Regulacije i zakoni:** Stroge regulative o privatnosti podataka mogu ograničiti implementaciju
- **Ovisnost o tehnologiji:** Tehnički kvarovi, zastarijevanje modela ili sigurnosne rupe mogu imati loš utjecaj na prihvatanje aplikacije
- **Otpor korisnika:** Strah ili skepticizam prema tehnologiji prepoznavanja lica, posebno kod roditelja
- **Troškovi održavanja:** Aplikacija zahtijeva stalna ažuriranja i održavanje kako bi bila konkurentna

3. Use Case i Class dijagrami sustava

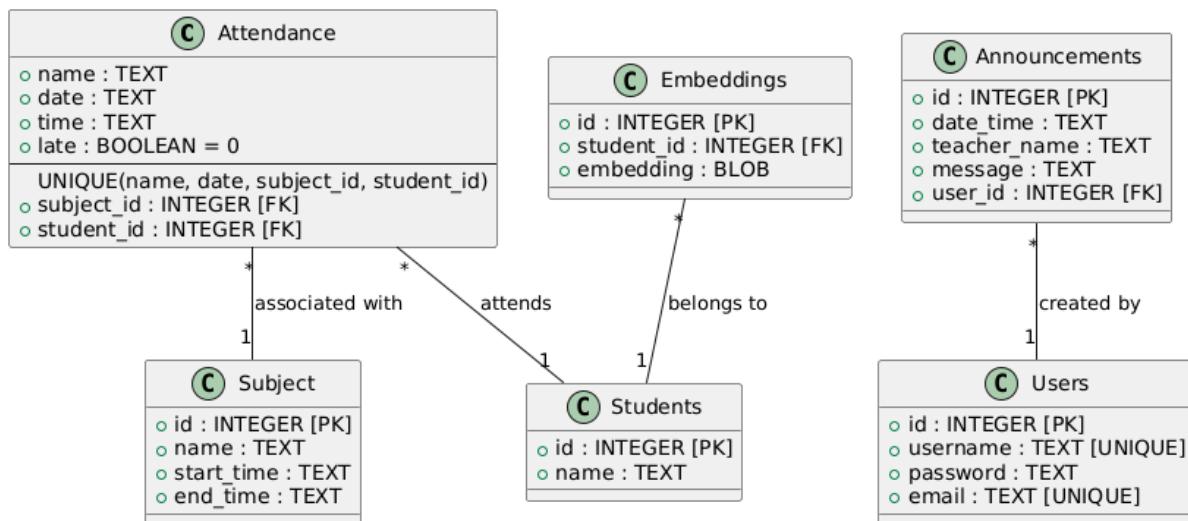
Sustav ima 2 tipa korisnika: studenta koji nema praktički nikakve ovlasti unutar aplikacije osim prijavljivanja prisutnosti i profesora koji ima pune administratorske ovlasti, što je detaljnije razrađeno na sljedećim dijigramima (iako su i student i profesor dio istog sustava, razdvojeni su u 2 dijagrama radi preglednosti te je dodatno dijagram za profesora razdvojen u 3 dijela).



Use case za studenta - prilaže slike za evidenciju u sustav i prijavljuje prisutnost



Use case za profesora - nakon prijave u sustav može postaviti trenutni predmet, njegov start time i end time, dodavati nove studente u sustav, uređivati objave na forumu, pregledavati popis svih studenata, pregledavati podatke o prisustvu s opcijom brisanja i filtriranja, preuzeti izvješće o prisustvima u csv formatu i dijeliti ga e-mailom, započeti interval bilježenja prisutnosti i zaprimiti email s potvrdom svake prijave i po potrebi obrisati neki od zapisa o prisutnosti. Može i pregledavati razne statističke analize, predikcije, izvještaje i grafove vezane uz podatke o prisutnostima.



Class dijagram sustava - aplikacija se sastoji od 6 glavnih klasa.

Announcements klasa pripada korisniku na način da svaki announcement mora imati samo jednog korisnika koji ga je kreirao, a 1 korisnik može kreirati više announcementsa.

Embeddings klasa pripada studentu na način da svaki embedding pripada samo jednom studentu, dok 1 student može imati više embeddinga.

Students klasa predstavlja 1 studenta.

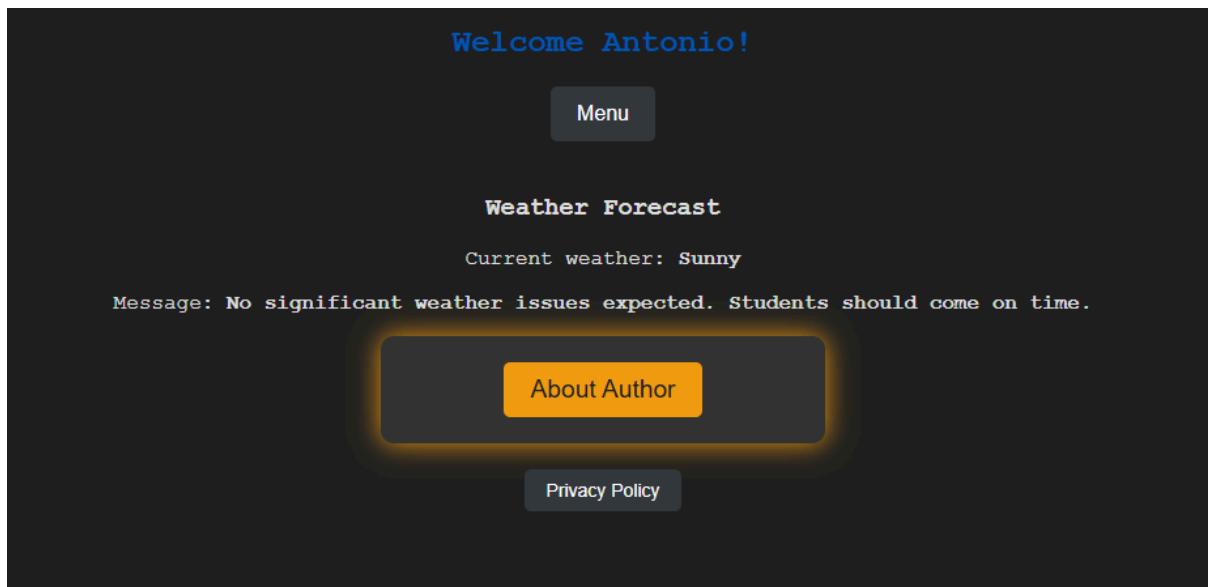
Subjects klasa predstavlja 1 kolegij.

Attendance klasa predstavlja 1 zapis o prisutnosti. Svaki attendance povezan je s točno jednim subjectom (dok 1 subject može biti povezan s više attendance-a) i točno jednim studentom (dok 1 student može biti povezan s više attendance-a).

4. Razrada funkcionalnosti

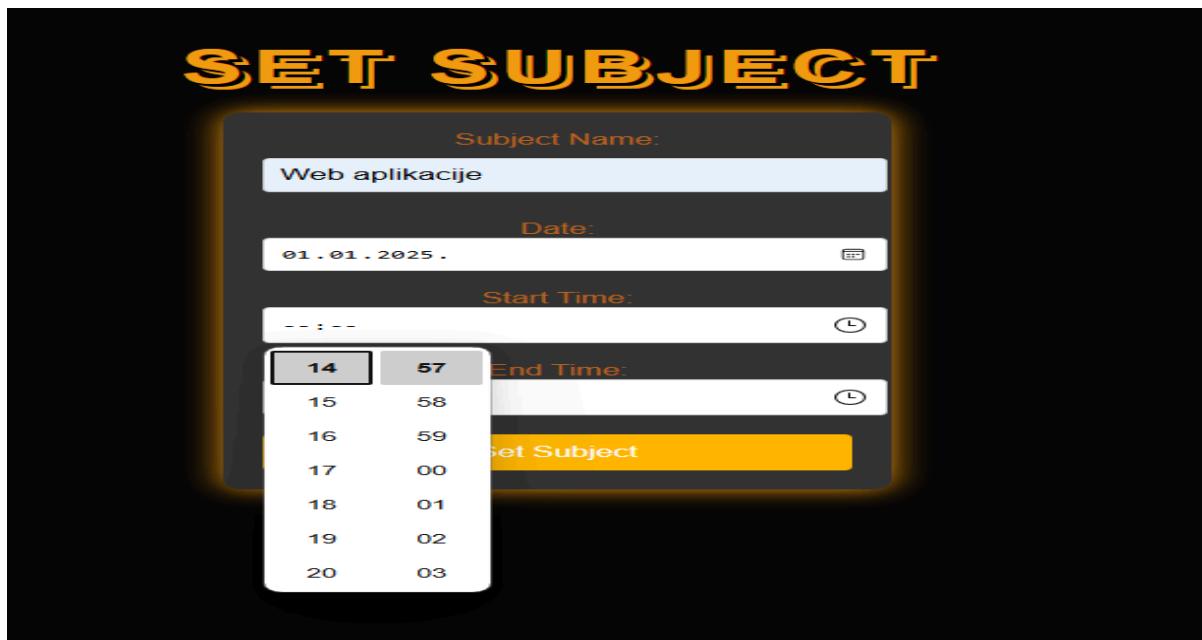
Aplikacija se sastoji od 2 dijela - frontenda napisanog u HTML-u i backenda napisanog u Flasku uz dodatak SQL baze podataka te CLIP modela i FAISS-a. U nastavku će biti detaljnije opisani svi dijelovi aplikacije uz napomenu da sposobnosti dizajna definitivno nisu nešto čime se mogu pohvaliti. Neki, jednostavniji i manje-više nebitni dijelovi bit će preskočeni (error redirectovi i slično).

4.1. Landing page



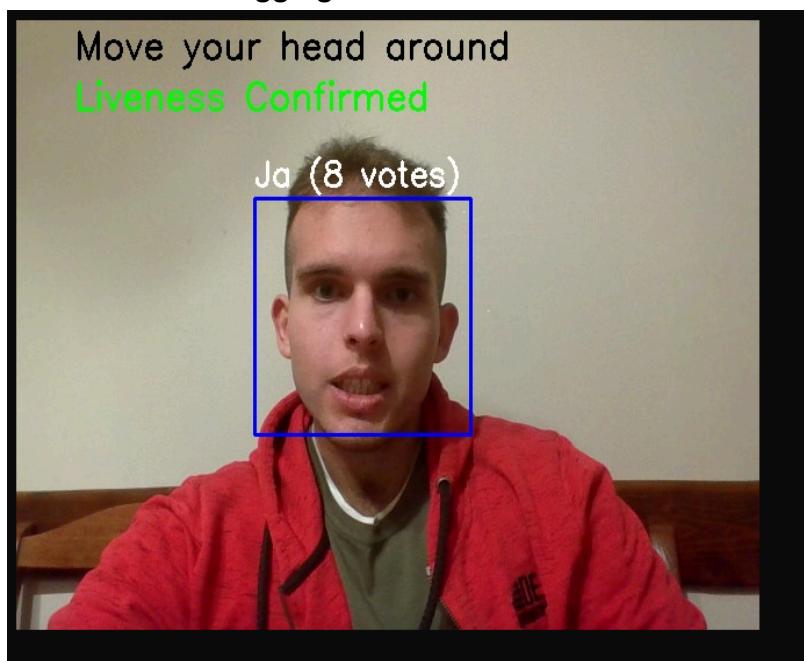
Nakon što se korisnik uspješno prijavi u sustav, dočekat će ga landing page koji sadrži welcome poruku s dinamički dohvaćenim korisničkim imenom, tipkom za dropdown menu koja se aktivira pri hoveru miša, tipkom za rutiranje na about stranicu i tipkom za pregled privacy policy-ja koji objašnjava kako referentne slike korištene za prepoznavanje unutar aplikacije ni u kojem slučaju neće izaći izvan sustava. Također, prikazuje se i trenutna vremenska prognoza za Pulu dohvaćena putem vanjskog API-ja.

4.2. Set subject



Komponenta za postavljanje trenutnog predmeta je točka od koje kreće svaka evidencija prisutnosti. U tom dijelu aplikacije profesor definira za koji se predmet trenutno bilježi prisutnost. Nadalje, definira se datum evidentiranja prisutnosti koji se automatski postavlja na današnji datum, no moguće ga je promijeniti ukoliko je to potrebno. Na kraju, još je potrebno definirati start time i end time za evidenciju kako bi se prijava prisutnosti ograničila na definirani vremenski interval.

4.3. Attendance logging



Najsloženiji dio aplikacije svakako se odnosi na modul za live prijavu prisustva. Pomoću cv2 biblioteke, otvara se kamera koja najprije traži lica u dohvaćenim frameovima koristeći haarcascade. Za svako prepozнато lice, kreira se bounding box. Nakon toga, provodi se jednostavna liveness detekcija koja provjerava kreće li se lice i registrira treptaje (kao što je već spomenuto, postoje bolja rješenja za liveness provjeru, ali naplaćuju se). Ukoliko lice

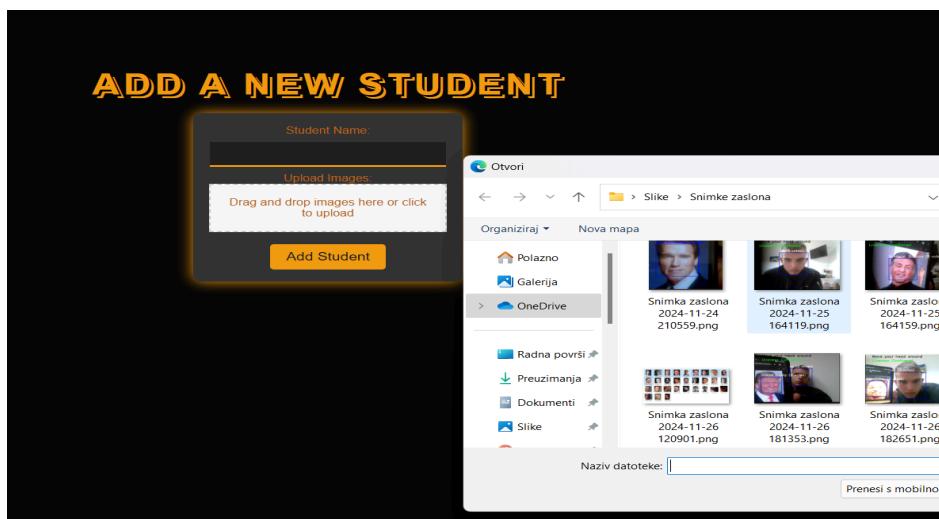
prođe liveness check, kreće proces konverzije slike u embedding koristeći clip. Ono što se događa u pozadini je zapravo pretvorba featuresa slike u tensor. Nakon što se tensor dodatno normalizira, kreće usporedba dobivenog tensora sa tensorima svih poznatih lica koji su pohranjeni u faiss indexu. Faiss index omogućava indeksiranu pretragu na temelju sličnosti. Takva je potraga implementirana zato što su lica najprije bila pohranjivana u običan embedding space, no to se nije pokazalo funkcionalnim za veći broj poznatih lica jer je došlo do preklapanja embeddinga i misklasifikacije. Faiss je omogućio dohvati top-k rezultata, odnosno uzimanje inputa tensora sa kamere i dohvati k slika koje najviše sliče inputu. Nakon dohvata k slika, provodi se majority voting koji provjerava s kojom se klasom input najviše podudara. Radi dodatne optimizacije, koriste se k1 i k2 koji omogućuju da se ne pretražuju sva lica, već se na samom početku iz razmatranja izbace ona lica koja su previše različita. Također, postoji i parametar thresholda koji definira koliko % inputa mora sličiti nekom od lica kako bi se oni smatrali validnim parom.

4.4. Attendance overview

The screenshot shows a web-based application titled "ATTENDANCE RECORDS". At the top, it displays the date "Date: 2025-01-01" and the subject "Subject: Web aplikacije (1 students)". Below this is a search bar containing fields for "Name:" and "Subject:", both with blacked-out placeholder text. Underneath these fields is a "Date" section with dropdown menus for "dd.mm.yyyy.", "Weekday", "Any", "Month", "Any", "Year", and "Late". At the bottom of the search bar are three buttons: "Any", "Filter", and "Clear Filters". A "Download Attendance CSV" link is located just above the search bar. The background of the entire interface is dark.

Nakon uspješne evidencije prisutnosti, u bazu podataka se pohranjuje zapis o prisutnosti koji se sastoji od trenutnog predmeta za koji bilježimo prisutnost, vremena evidentiranja, datuma, imena i prezimena studenta i late flaga. Late flag označava kasni li student ili je došao na vrijeme. Taj se dio evidentira provjeravajući koje je definirano početno vrijeme za postavljeni predmet i koje je vrijeme evidentiranja prisutnosti. Ukoliko je prisutnost evidentirana 15 ili više minuta kasnije od početnog vremena, taj se zapis definira kao kašnjenje. (Postoji akademski četvrt). Ime i prezime se dohvata preko naziva klase u koju je skenirano lice svrstano. Dodatno, postoji mogućnost brisanja zapisa i filtriranje po imenu studenta, predmetu, datumu, danu u tjednu, mjesecu, godini i late flagu. Na samom kraju, radi lakše obrade podataka omogućeno je preuzimanje zapisa o prisutnosti u csv formatu.

4.5. Add student manually



Moguće je dodati novog studenta u bazu poznatih lica ručno, koristeći slike iz lokalne pohrane. Potrebno je unijeti ime i odabrati slike iz računala ili ih drag&droppati na predviđeno mjesto. Minimalni unos je 1 slika, dok gornje granice nema. No, kako takav pristup nije naročito praktičan, omogućeno je i live dodavanje pomoću kamere.

4.6. Add student using live feed

Add New Student via Live Feed

Student Name: [Start Capturing](#)

Dodavanje pomoću kamere funkcioniра na prilično jednostavan način. Potrebno je unijeti ime i prezime studenta i samo pritisnuti na tipku. Tada student treba gledati u kameru nekoliko sekundi, kako bi se ulovilo 25 frameova i pohranilo ih se u novi folder s njegovim imenom. Ova mogućnost olakšava prikupljanje slika za proširivanje baze poznatih lica kako ne bi bilo potrebno za svakog studenta uploadati slike ručno i značajno ubrzava proces.

4.7. Students overview



Pri prvotnoj implementaciji spremanja prisutnosti došlo je do malih poteškoća u provjeri poznatih studenata, pa je tako nastala ova, inicijalno, debug ruta koja dohvata sve studente iz svih zapisa o prisutnosti, neovisno o predmetu i datumu. Provodi se sql upit koji dohvata sve unique studente koji su ikada spremljeni u bazu, odnosno evidentirani kao prisutni.

4.8. Announcement forum

The screenshot shows a dark-themed web application interface. At the top center is the word "ANNOUNCEMENTS" in yellow capital letters. Below it is a text input field with placeholder text "Write your announcement here...". A "Post Announcement" button is located below the input field. To the right of the input field, there is a timestamp "2025-01-02 11:34:35 - Antonio: Hello everyone" followed by "Edit" and "Delete" buttons. Below these buttons is a link "All Announcements" and a "Back to Home" link at the bottom.

Announcement forum zamišljen je kao svojevrsni “interni chat” za korisnike aplikacije. Za svaku objavu vidljiv je korisnik koji ju je napisao te datum i vrijeme objave. Nakon objavljivanja, korisnik može obrisati i urediti svoju objavu. Također, implementiran je NLP model:

“badmatr11x/distilroberta-base-offensive-hateful-speech-text-multiclassification” koji pri pokušaju objavljivanja provjerava sadrži li objava neprimjerene riječi tako što najprije izvrši tokenizaciju objave, pretvorи je u tensor i izračuna “offensiveness-score”. Ukoliko je score veći od 30%, objava se flagira kao neprimjerena i onemogućuje se njezina objava uz prikladno upozorenje. Ista se provjera primjenjuje i pri editiranju objave.

The screenshot shows the same dark-themed interface as the first one. An orange horizontal bar at the top contains the text "Your announcement contains inappropriate language and was not posted". Below this, the "ANNOUNCEMENTS" section is visible with its input field, "Post Announcement" button, and timestamped announcement. The "All Announcements" and "Back to Home" links are also present.

4.9. Late entry analysis

The screenshot shows a section titled "Analysis:" with the following text:
 The most common hour for late entries is: 16:00 with 1 late entries.
 The most common weekday for late entries is: Wednesday with 1 late entries.

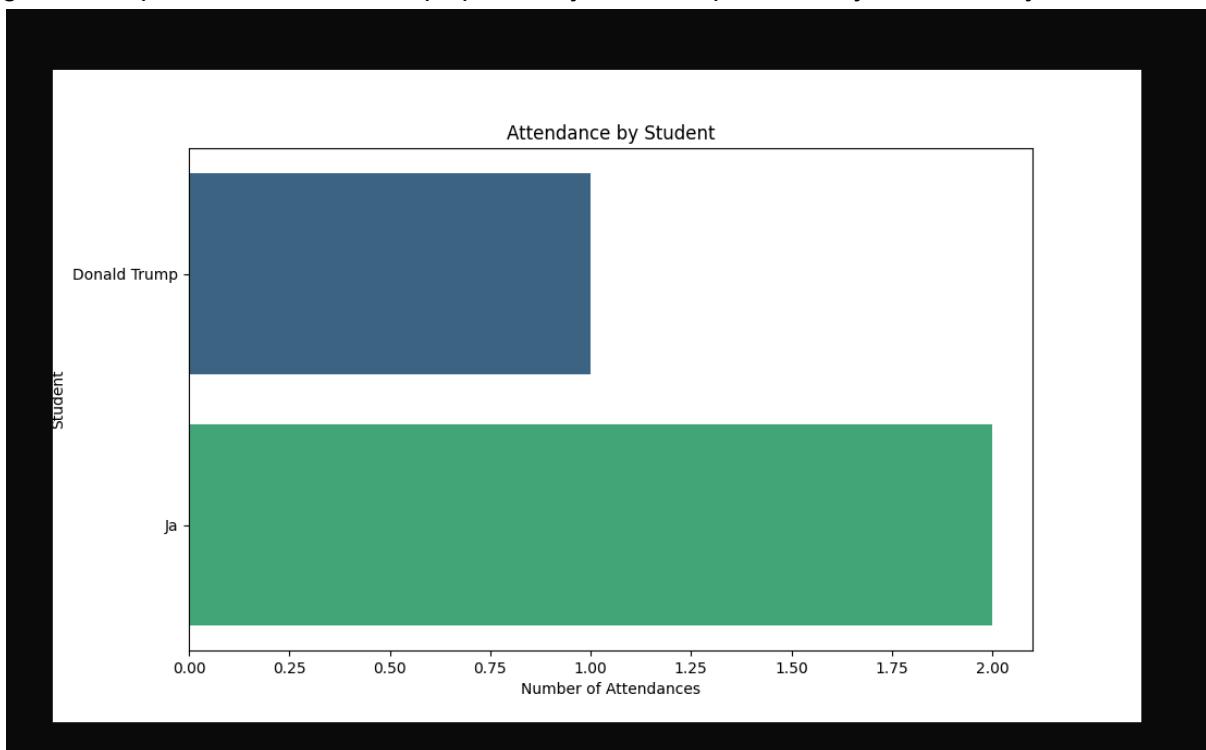
Below this is a chart titled "Late Entries by Hour" showing the number of late entries per hour of the day. The x-axis is labeled "Hour of the Day" from 0 to 23, and the y-axis is labeled "Number of Late Entries" from 0.0 to 1.0. A single blue bar is shown at hour 16 with a value of 1.0.

Name	Date	Time In	Subject
Ja	2025-01-01	16:39:38	Web aplikacije

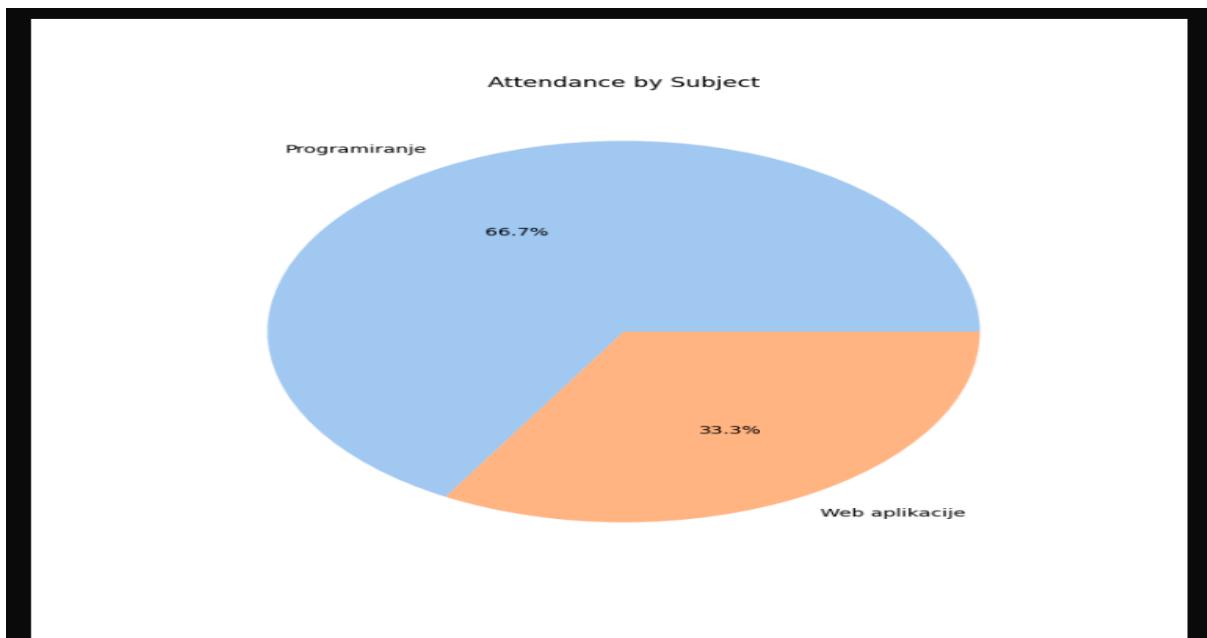
Za svako predavanje prati se vrijeme kada studenti dolaze i kasne li. Kašnjenje se evidentira ako je student prijavio prisutnost 15 ili više minuta nakon početka predavanja. Kašnjenja se dohvaćaju iz baze podataka pomoću late flaga, odnosno samo se select-aju sve zapisi gdje je late flag = True. Ti se podaci vizualiziraju pomoću grafikona koji pokazuje koliko kasnih ulazaka ima za pojedini sat u danu. Na taj se način može utvrditi koji sati u danu su posebno problematični i potencijalno se može optimizirati raspored na način da se minimiziraju kašnjenja.

4.10. Plot router & data visualization

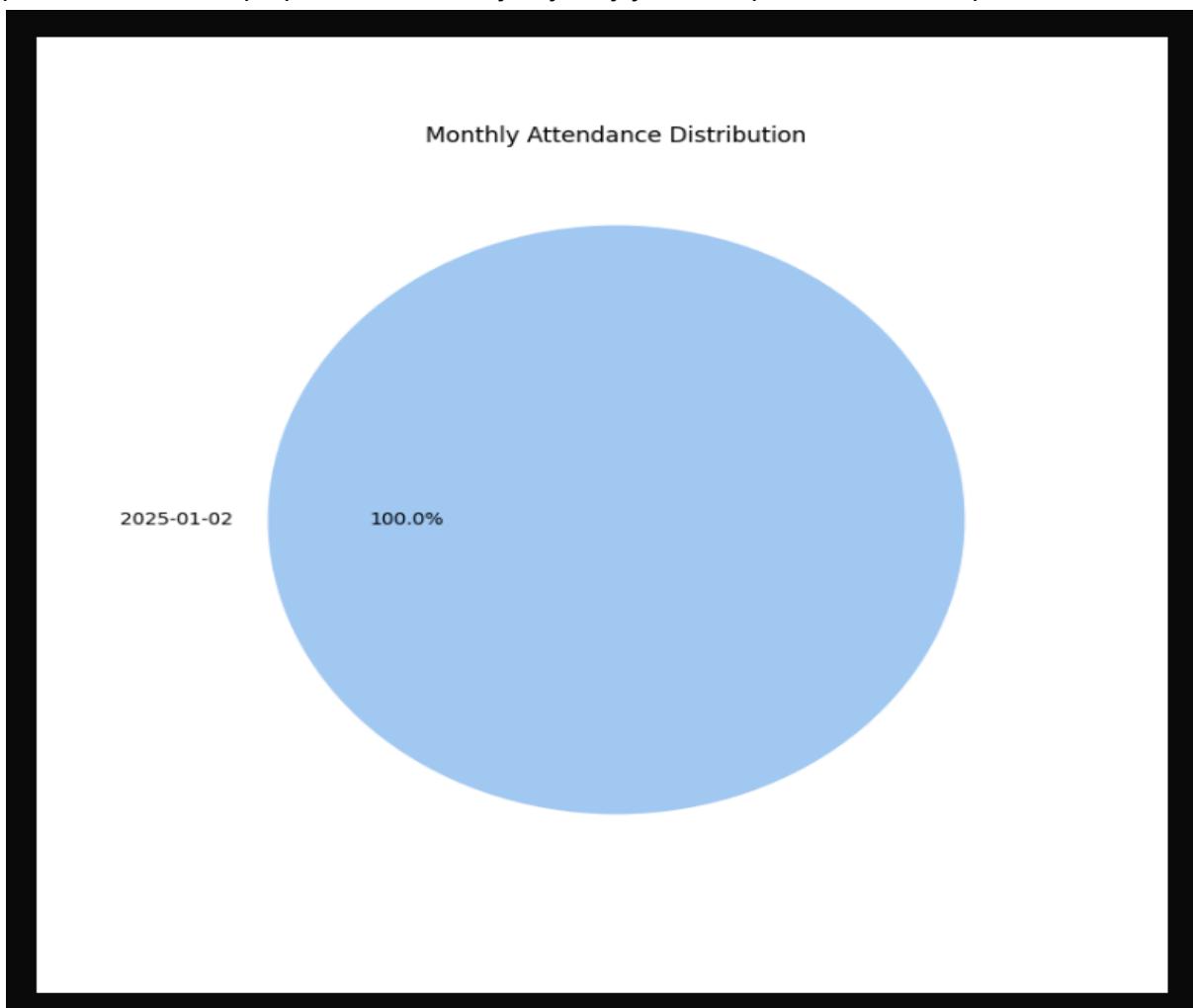
Definirana su 3 grafikona kojima možemo pristupiti pomoću posebnog rutera. Svaki od grafikona "puni se" određenim sql upitom koji dohvaća podatke koji nas zanimaju.



plot 1. - Prisutnost po studentu => vidljivo je na koliko je predavanja svaki pojedini student bio prisutan



plot 2. - Prisutnost po predmetu => vidljivo je koji je odnos prisutnosti među predmetima



plot 3. - Mjesečna prisutnost => vidljivo je na koji datum je zabilježen koji udio prisutnosti, grupirano po mjesecu

4.11. Statistics

Attendance Statistics		
Student Attendance Counts		
Name	Subject	Attendance Count
Donald Trump	Matematika	1
Ja	Matematika	1
Ja	Web aplikacije	1

Subject Attendance Counts	
Subject	Attendance Count
Matematika	2
Web aplikacije	1

Omogućena je i brza statistička analiza koja pomoći sql upita provjerava koliko je puta pojedini student prisustvovao nekom predmetu i koliko je ukupno unique studenata prisustvovalo pojedinom predmetu.

4.12. Profile



Antonio Labinjan

Bio: Student on FIPU Interested in: app development, databases, computer vision I love Python

Followers: 3

Repositories: 149

[Go to Home](#)

Profile, odnosno stranica o autoru (meni) dinamički dohvata podatke s mojeg github accounta koristeći BeautifulSoup html web-parser za webscraping. Konkretni podaci koji su učitani s githuba su: profilna slika, username, bio, broj followera i broj **javnih** repozitorija na dan 2. siječnja 2025. Također, koristi se flaskova posebna funkcija render template string koja omogućava da izrenderiramo html template direktno iz Python koda bez da moramo pisati poseban template file.

4.13. Calendar

Non-Working Days 2024/2025

- 7.10. Početak nastavne godine u 1.11. Blagdan Svi sveti 18.12. Redovita sjednica Senata zimskom semestru i 18.11. Dan sjećanja na Žrtve 25.12. Božić
- 24.10. Smotra Sveučilišta Vukovara i Škabrnje 23.-31. Zimski odmor
- 1.-3. Zimski odmor 3.2. Početak zimskog ispitnog 3.3. Početak nastavne godine
- 1.1. Nova Godina roka u ljetnom semestru
- 6.1. Sveta tri kralja 26.2. Redovita sjednica Senata 26.3. Redovita sjednica Senata
- 16.4. Dan Sveučilišta - nenastavni dan 1.5. Praznik rada 13.6. Završetak nastavne godine
- 20.4. Uskrs 28.5. Redovita sjednica Senata u ljetnom semestru
- 21.4. Uskršnji ponedjeljak 30.5. Dan državnosti 16.6. Početak ljetnog ispitnog roka
- 23.4. Redovita sjednica Senata 19.6. Tijelovo
- 11.7. Završetak ljetnog ispitnog 1.-29.8. Ljetni odmor 1.9. Početak jesenskog ispitnog roka
- roka 5.8. Dan domovinske zahvalnosti 24.9. Redovita sjednica Senata
- 16.7. Redovita sjednica Senata 15.8. Blagdan Velike Gospe 26.9. Završetak jesenskog ispitnog
- 14.-31.7. Ljetni odmor roka - Završetak akademске

Pomoću pdf scrapera dohvaćaju se neradni dani iz službenog unipu kalendarja za akademsku godinu 2024/25 i prikazani su u html templateu koji se kreira pomoću već spomenute render template string funkcije.

4.14. Ostalo

Implementirane su još i komponente za autentifikaciju koristeći flaskove ugrađene funkcije iz biblioteke flask-login uz dodatnu validaciju passworda, success stranice za postavljanje predmeta i dodavanje studenta i pop-up komponenta za notifikaciju nakon svake zabilježene prisutnosti. Kako se radi o prilično jednostavnim komponentama, nema smisla da ih opširno objašnjavam, pošto se nema što pretjerano objasniti.

5. Moduli aplikacije

Zbog specifičnog načina pisanja Flask aplikacija, teoretski je bilo moguće kompletну aplikaciju (bar backend) napisati u jednom jedinom .py fajlu, što sam i radio na početku. No, kako je u jednom trenutku postalo nemoguće scrollati kroz više stotina linija koda i snalaziti se u njima, odvojio sam glavni app.py file na više modula radi lakšeg snalaženja i efikasnijeg pisanja koda.

Ti su moduli:

- imports.py
- model_loader.py
- load_post_check.py
- db_startup.py
- app.py
- također, postoji i cijeli niz helper datoteka koje se ne koriste u direktnom kontekstu aplikacije, već su kreirane s ciljem ubrzavanja određenih sub-taskova koje je bilo potrebno odraditi pri pisanju koda i testiranju

Imports.py

Imports.py je datoteka koja služi samo za importanje svih potrebnih biblioteka koje koristimo unutar aplikacije. Funkcionira na način da u njoj navedemo sve importove koji nam trebaju i onda u glavnom kodu samo pozovemo `from imports import *`.

U prijevodu, u glavni file uvozimo sve stvari navedene u imports.py.

Model_loader.py

Pošto se, teoretski, za isti use-case može koristiti više različitih AI modela, učitavanje modela je također odvojeno u poseban modul. Također, specifično je to što se model fine-tunira na trenutni dataset pri prvom pokretanju aplikacije i kreira nove weightove za taj dataset. Ukoliko bismo htjeli u nekom trenutku promijeniti model, samo bismo morali malo izmijeniti ovaj file. Konkretno, ovaj file učitava [openai/clip-vit-base-patch32](#).

Load_post_check.py

Na sličan način funkcionira i ovaj modul. Učitava model za klasifikaciju teksta:

[badmatr11x/distilroberta-base-offensive-hateful-speech-text-multiplication](#).

Db_startup.py

I kreiranje baze podataka izdvojeno je u poseban modul. Koristi se sqlite3 baza.

```
conn = sqlite3.connect('attendance.db')
c = conn.cursor()
```

Najprije kreiramo konekciju na bazu i onda pozivamo kurzor za izvođenje upita te ga spremamo u varijablu koju možemo kasnije pozivati.

```
c.execute('''DROP TABLE IF EXISTS attendance''')
c.execute('''CREATE TABLE IF NOT EXISTS attendance
            (name TEXT, date TEXT, time TEXT, subject TEXT, late
BOOLEAN DEFAULT 0, UNIQUE(name, date, subject))''')

c.execute('''DROP TABLE IF EXISTS announcements''')
c.execute('''CREATE TABLE IF NOT EXISTS announcements
            (id INTEGER PRIMARY KEY AUTOINCREMENT,
             date_time TEXT,
             teacher_name TEXT,
             message TEXT)''')

c.execute('''CREATE TABLE IF NOT EXISTS users
            (id INTEGER PRIMARY KEY AUTOINCREMENT,
             username TEXT UNIQUE,
             password TEXT,
             email TEXT UNIQUE)''')

c.execute('''
CREATE TABLE IF NOT EXISTS students (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL
)
'''')
```

```

c.execute('''
CREATE TABLE IF NOT EXISTS embeddings (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    student_id INTEGER,
    embedding BLOB,
    FOREIGN KEY (student_id) REFERENCES students (id)
)
''')

conn.commit()
conn.close()

```

Nakon toga, koristeći kurzor kreiramo tablice koje se koriste u aplikaciji. Attendance za spremanje zapisa o prisutnosti, announcements za objave, users za korisnike (profesore), students za studente i embeddings za spremanje embeddinga svakog studenta. Nakon toga, sve što je kurzor odradio se commita i konekcija se zatvara kako u aplikaciji ne bi nastao kaos zbog slučajne interakcije s bazom podataka.

App.py

U app.py modulu odvija se glavni rad aplikacije. Najprije importamo sve što nam treba iz imports.py

```
from imports import *
```

Onda importamo funkcije iz ostalih datoteka.

```

from db_startup import create_db
create_db()

from model_loader import load_clip_model
from load_post_check import load_model_and_tokenizer

```

Učitavamo dotenv modul kako bismo dohvatali osjetljive podatke iz .env filea koje nije poželjno hardkodirati; poput secret-keya, api ključeva i sendgrid konfiguracije.

```
load_dotenv()
```

Incijaliziramo flask aplikaciju i dohvaćamo secret key iz .env-a. Key ima nekoliko funkcija, ali najvažnija je svakako omogućavanje session managementa.

```

app = Flask(__name__)
app.secret_key = os.getenv('SECRET_KEY')

```

Konfiguriramo slanje e-mailova koristeći sendgrid. Također, radi sigurnosti, svi credentialsi se dohvaćaju iz .env. Dodatan razlog tome je i što sendgrid ne voli da se njegovi podaci commitaju na public github repozitorije i odmah suspendira account ukoliko slučajno commitamo api ključ ili nešto slično.

```

app.config['MAIL_SERVER'] = os.getenv('MAIL_SERVER')
app.config['MAIL_PORT'] = int(os.getenv('MAIL_PORT'))

```

```

app.config['MAIL_USE_TLS'] = os.getenv('MAIL_USE_TLS') == 'True' #
dohvaća bool varijablu iz .env i uspoređuje je s True. Ako je u .env
varijabla True, ovaj check će tornat True, i pičimo dalje
app.config['MAIL_USERNAME'] = os.getenv('MAIL_USERNAME')
app.config['MAIL_PASSWORD'] = os.getenv('MAIL_PASSWORD')

# SendGrid API Key
sendgrid_api_key = os.getenv('SENDGRID_API_KEY')

```

Konfiguriramo flask login paket kako bismo omogućili autentifikaciju korisnika. Koristimo LoginManagera i definiramo da će view, odnosno ruta na kojoj se login obavlja biti ‘login’. Ukoliko bi se ruta zvala ‘prijava’, samo bismo ovdje morali redefinirati vrijednost login_view varijable.

```

login_manager = LoginManager()
login_manager.init_app(app)
login_manager.login_view = 'login'

```

Definiramo klasu User za kreiranje korisnika. Svaka instanca klase (a.k.a. svaki user) ima atribute id, username, password i email. Self dio klase služi kako bismo se referencirali na neki određeni objekt, a ne na klasu generalno. Koristeći self, Python zna da kreiramo novi objekt. Propertyji nisu bitni u ovoj aplikaciji, ali bez njih ne možemo instancirati korisnika. Postoje propertyji koji provjeravaju je li korisnik aktivan, autenticiran ili anoniman. Za naše potrebe, definirano je da je svaki korisnik po defaultu aktivan i autenticiran i nije anoniman. Također, postoji metoda koja vraća userov id kao string.

```

class User:
    # konstruktor
    def __init__(self, id, username, password, email):
        self.id = id
        self.username = username
        self.password = password
        self.email = email

    # Inače se moru dodat i složenije provjere, ali meni to ni toliko
    # bitno. Pretpostavljamo da je svaki user aktivan, autenticiran i da ni
    # anoniman (to su default properties čim se ulogira)
    # Flask-Login needs these properties to work correctly
    @property
    def is_active(self):
        # Return True if the user is active. You can modify this based
        # on user status.
        return True

```

```

@property
def is_authenticated(self):
    # Return True if the user is authenticated
    return True

@property
def is_anonymous(self):
    # Return False because this is not an anonymous user
    return False

def get_id(self):
    # Return the user's ID as a string
    return str(self.id)

```

Zadnji dio autentifikacije na backendu je učitavanje korisnika preko id-ja. Korisnik unosi svoje credentialse, provjerava se koji je id korisnika s tim credentialsima i ako je takav korisnik nađen, dozvolit će se ulazak u aplikaciju. Ako nema takvog korisnika, vratit će se None i onemogućiti ulazak u aplikaciju.

```

# prosljedimo id u funkciju, spajamo se na bazu, cursor izvršava query
na bazu tako da dohvaća onega korisnika koji ima odgovarajući id
@login_manager.user_loader
def load_user(user_id):
    conn = sqlite3.connect('attendance.db')
    c = conn.cursor()
    c.execute("SELECT * FROM users WHERE id = ?", (user_id,))
    # kurzor dohvaća prvega usera s odgovarajućim id-jem pomoću
fetchone (doduše, ne more imati 2 usera, isti id, ali čisto preventivno)
    user = c.fetchone()
    # zatvaramo konekciju da se sve ne zbresira
    conn.close()

    # ako smo našli korisnika s tim id-jem, dohvaćamo attribute
indeksirano po redu kako smo definirali u klasi
    if user:
        return User(id=user[0], username=user[1], password=user[2],
email=user[3])
    return None

```

Computer vision dio aplikacije počinje s učitavanjem poznatih lica u aplikaciju. Sama "baza" lica strukturirana je koristeći known_faces folder koji se sastoji od n subfoldera (n označava

broj poznatih osoba u aplikaciji). U svakom subfolderu nalaze se slike na kojima je model, glupo rečeno, istreniran. Funkcija dodaje nova lica tako da prolazi kroz sve putanje do svih slika, provjerava što se tamo nalazi, učitava slike i prebacuje ih u RGB format. . Zatim se te slike prosljeđuju u clip processor koji ih pretvara u tensore koji će se kasnije koristiti kao input za detekciju/prepoznavanje unutar modela. Zatim se iz tih tensora izvlače featuresi, odnosno embeddingsi koji pomažu u “identifikaciji” pojedine klase (svaka klasa ima manje ili više unikatne embeddinge). Na samom kraju se ti embeddingsi pretvore u numpy array i normaliziraju koristeći linalg norm funkciju i dodaju u listu koja će se kasnije učitati.

```
# Function to add a known face.

# Dodaje sliku po slicu. Prosljeđuje putanju do slike i ime
klase(osobe)

def add_known_face(image_path, name):
    # učitavamo sliku iz putanje koju smo proslijedili
    image = cv2.imread(image_path)
    # ako je ni (None=> nema je; nismo je našli) dajemo value error
    if image is None:
        raise ValueError(f"Image at path {image_path} not found.")
    # ako je ima, konvertiramo je u drugi format da clip more delat s
    # njom
    image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    # processor uzima rgb slike i pretvara ih u tensore koji su input
    # za clip model
    inputs = processor(images=image_rgb, return_tensors="pt")
    with torch.no_grad():
        # uzimamo tensore i izvlačimo featurse iz njih tj. embeddinge
        outputs = model.get_image_features(**inputs)
        # pretvorimo u numpy array
        embedding = outputs.cpu().numpy().flatten()
        # normaliziranje za lakšu usporedbu
        known_face_encodings.append(embedding / np.linalg.norm(embedding))
        # appendamo u listu
        known_face_names.append(name)
    print(f"Added face for {name} from {image_path}")
```

Učitavamo poznata lica u aplikaciju zako što iteriramo kroz folder known_faces i sve njegove subfoldere. Definiramo o kojoj se klasi radi pomoću naziva subfoldera. U trenutnoj implementaciji postoje train i val subfolderi unutar svake klase, no to se može izbaciti i koristiti samo 1 set slika. Na kraju se samo ispisuje koliko se tensora i klasa ukupno učitalo.

```
# Load all known faces from the 'known_faces' directory

def load_known_faces():
    # Loop through each class (subfolder) in the 'known_faces' folder
    for student_name in os.listdir('known_faces'):
```

```
# Path to the train subfolder for the current class =>
učitavamo samo train
train_dir = os.path.join('known_faces', student_name, 'train')

# Check if the train subfolder exists
if os.path.isdir(train_dir):
    # Loop through all images in the train subfolder
    for filename in os.listdir(train_dir):
        image_path = os.path.join(train_dir, filename)
        try:
            # Add the known face from the image
            add_known_face(image_path, student_name)
        except ValueError as e:
            print(e)

# Debugging
print(f"Loaded {len(known_face_encodings)} known face encodings")
print(f"Known face names: {known_face_names}")
```

Unutar aplikacije se koristi i Facebook AI Similarity Search (FAISS). Služi za brzo pretraživanje i usporedbu embeddinga lica u bazi. Svaka klasa imao svoje embeddinge, koji predstavljaju jedinstvene karakteristike tog lica. Kada aplikacija prepozna lice, generira se embedding koji se zatim uspoređuje s postojećim embeddingsima u bazi podataka. FAISS omogućava brzo i efikasno pretraživanje tih embeddinga, čineći proces identifikacije lica bržim. Pogotovo u odnosu na prvotnu implementaciju gdje je korišteno linearno pretraživanje koje je postajalo sve sporije sa svakim novim dodanim licem. Osim toga, FAISS pomaže u grupiranju sličnih lica, što je korisno za organiziranje i analizu podataka. Uzimamo clip embeddinge, pretvorimo ih u array i spremamo to u FAISS indeks koji kasnije koristimo za pretragu.

```
# Build an index for Facebook AI Similarity Search (FAISS) using known
face encodings
def build_index(known_face_encodings):
    # Pretvaramo listu known_face_encodings u numpy array za rad s
    # FAISS-om
    known_face_encodings = np.array(known_face_encodings)

    # Dobivamo dimenziju svakega embedding vektora (broj feturesa po
    # vektoru)
    dimension = known_face_encodings.shape[1]

    # Kreiramo FAISS index koji koristi L2 (Euclidean distance) =>
    # korijen od kvadrirane sume razlika podudarnih elemenata u 2 vektora
    faiss_index = faiss.IndexFlatL2(dimension)

    # Dodajemo sve poznate face encodings u FAISS index
    faiss_index.add(known_face_encodings)

    # Vraćamo izgrađeni FAISS index
    return faiss_index
```

Postavljamo basic setup za rad s kamerom koristeći biblioteku cv2. Nakon toga, koristimo haarcascade za osnovnu detekciju lica na kameri i crtanje bounding boxeva oko njih. Svako lice koje detektiramo na kameri, dobiva svoj bounding box.

```
# Initialize the webcam. Stavimo nulu za koristit defaultnu kameru (ako
# smo toliko luksuzni da imamo više kamere, moremo stavit 1,2,3 itd. ;)
video_capture = cv2.VideoCapture(0)

# Face detection using Haar Cascade
# ovo ubacimo da detektira lica i crta bounding boxeve oko njih
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
'haarcascade_frontalface_default.xml')
```

Svaki put kada započinjemo novo bilježenje prisutnosti, moramo definirati koji se predmet trenutno sluša. Trebaju nam atributi current_subject, attendance_date, start_time i end_time. Defaultna vrijednost im je None zato što varijabla treba biti inicijalizirana, a ne želimo postaviti neku smislenu vrijednost kako ne bi došlo do problema u kasnijem radu.

```
# Inicijaliziranje atributa za trenutnu prisutnost
current_subject = None
attendance_date = None
start_time = None
end_time = None
```

Implementirana je validacija password-a koja zahtjeva da ima bar 8 znakova, bar 1 malo i 1 veliko slovo, bar 1 broj i bar 1 poseban znak. Ukoliko neki od uvjeta nije ispunjen, funkcija će vratiti prikladni error message, a ako je sve OK, neće se vratiti ništa i password će biti odobren.

```
def validate_password(password):
    if len(password) < 8:
        return "Password must be at least 8 characters long."
    if not re.search(r"[A-Z]", password):
        return "Password must contain at least one uppercase letter."
    if not re.search(r"[a-z]", password):
        return "Password must contain at least one lowercase letter."
    if not re.search(r"\d", password):
        return "Password must contain at least one number."
    if not re.search(r"[@#$%^&*(),.?':{}|<>/]", password):
        return "Password must contain at least one special character."
    return None
```

Ruta za signup koristi i GET i POST iz razloga što pomoću GET metode dohvaćamo formular za signup, dok pomoću POST-a šaljemo kreirane podatke o korisniku koji se dohvaćaju pomoću request_form (dohvat podataka koji su uneseni u html formular). Dodatno još provjerimo podudaraju li se prvi password i ponovljeni password i jesu li username i email unikatni. Ako je sve u redu, kreiramo novog korisnika.

```
@app.route('/signup', methods=['GET', 'POST'])
def signup():
    # get je jer dohaćamo formular, post je šaljemo zahtjev za signup
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']
        repeat_password = request.form['repeat_password']
        email = request.form['email']

        # standard validation
        if password != repeat_password:
            flash("Passwords do not match. Please try again.", "error")
```

```

        return redirect(url_for('signup'))

    password_error = validate_password(password)
    if password_error:
        flash(password_error, "error")
        return redirect(url_for('signup'))

    try:
        conn = sqlite3.connect('attendance.db')
        c = conn.cursor()

        c.execute("SELECT * FROM users WHERE username = ? OR email = ?",
                  (username, email))
        if c.fetchone() is not None:
            flash("Username or email already taken. Please choose a different one.", "error")
            conn.close()
            return redirect(url_for('signup'))

        # Hashiranje
        hashed_password = generate_password_hash(password,
method='pbkdf2:sha256')

        # Save to db
        c.execute("INSERT INTO users (username, password, email) VALUES (?, ?, ?)",
                  (username, hashed_password, email))
        conn.commit()
        conn.close()

        flash("Signup successful! Please log in.", "success")
        return redirect(url_for('login'))

    except sqlite3.IntegrityError:
        flash("An error occurred during signup. Please try again.", "error")
        return redirect(url_for('signup'))

    return render_template('signup.html')

```

Iz istog razloga kao kod signupa i kod logina koriste se GET i POST metode. Pomoću GET dohvativamo template prijavu, a ako je metoda pristupa ruti POST, onda iz requesta dohvativamo poslani username i password. Spajamo se na bazu iinstanciramo kurzor koji pomoću SQL upita traži korisnika u bazi preko username-a. Ako nađe korisnika, onda prelazi na provjeru passworda. Ukoliko se hash passworda iz baze podudara sa hashem unesenog passworda (password je isti, pa za istu hash funkciju daje identičan hash), poziva se ugrađena funkcija login_user koja u trenutni session sprema usera koji je ulogiran.

```
@app.route('/login', methods=['GET', 'POST'])

def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        conn = sqlite3.connect('attendance.db')
        c = conn.cursor()
        c.execute("SELECT * FROM users WHERE username = ?",
        (username,))
        user = c.fetchone() # Fetches user row (dohvati 1 usera)
        conn.close()

        if user:
            # User is found
            user_id, db_username, db_password, db_email = user
            if check_password_hash(db_password, password):
                # Successful login
                login_user(User(id=user_id, username=db_username,
password=db_password, email=db_email))
                return redirect(url_for('index'))

            flash("Invalid username or password")
            return redirect(url_for('login'))

    return render_template('login.html')
```

Logout ruta je iznimno jednostavna, samo poziva logout_user ugrađenu funkciju i vraća nas na login rutu. (kad se korisnik izlogira, biti će preusmjeren na login formular)

```
@app.route('/logout')
@login_required
def logout():
    # samo call-a ugrađenu funkciju za logout
    logout_user()
    return redirect(url_for('login'))
```

Set subject služi za postavljanje trenutnog predmeta za kojeg se prijavljuje prisutnost. Ruta koristi login_required dekorator koji ograničava pristup ruti te dopušta da ju samo ulogirani korisnici vide. Definira se funkcija koja postavlja predmet tako što definira globalne varijable za predmet, datum, početak i kraj. Te varijable moraju biti globalne kako bi im bilo moguće pristupiti i izvan ove funkcije (koristimo predmet za prijavu prisutnosti u za to predviđenoj funkciji). Ako ruti pristupimo sa post zahtjevom, onda iz forma dohvaćamo podatke, postavljamo predmet i dinamički ispunimo set_subject_success template sa podacima koji su proslijeđeni kroz post zahtjev.

```
@app.route('/set_subject', methods=['GET', 'POST'])
@login_required # moramo bit ulogirani za pristupit ruti
def set_subject():
    # ove varijable su globalne jer želimo da ih se mora čitat/dohvaćat
    # i van funkcije (tribat će nam kad budemo logali attendance)
    global current_subject, attendance_date, start_time, end_time
    # ako šaljemo post request, spremamo u varijable ovo ča smo
    # prosljedili kroz req. form
    if request.method == 'POST':
        current_subject = request.form['subject']
        attendance_date = request.form['date']
        start_time = request.form['start_time']
        end_time = request.form['end_time']
        # renderiramo template s dodanima podacima
        return render_template('set_subject_success.html',
                               subject=current_subject,
                               date=attendance_date,
                               start_time=start_time,
                               end_time=end_time)
    return render_template('set_subject.html')
```

Pomoćna funkcija is_within_time_interval() služi kako bismo vidjeli koje je trenutno vrijeme i koji je trenutni datum. To dohvaćamo pomoću funkcije datetime.now() iz koje dodatno extractamo bitne djelove. Funkcija zatim provjeri podudaranju li se ti djelovi s odgovarajućim dijelovima iz onoga što smo definirali kroz set_subject. Vraća True ili False ovisno o podudaranju.

```
# provjera dali je trenutno vrijeme unutar intervala koji smo
# definirali za predavanje
def is_within_time_interval():
    # u now spremamo trenutni datum i vreme
    now = datetime.now()
    # iz now extractamo posebno datum i posebno vreme
    current_time = now.strftime("%H:%M")
    current_date = now.strftime("%Y-%m-%d")
    # vraća true ako smo unutar intervala, odnosno false ako nismo
```

```

        return (current_date == attendance_date and
                start_time <= current_time <= end_time)

```

Add_student ruta služi za dodavanje novog poznatog studenta u aplikaciju. Ako šaljemo get zahtjev na rutu, samo se izrenderira template za dodavanje. Šaljemo li post zahtjev na rutu, onda kreće kreiranje novog studenta. Ima se dohvaća pomoću request_form (dohvati se ime koje smo unijeli u form field), a slike se dohvaćaju kao lista koja će se kasnije pretvoriti u embeddinge pomoću add_known_face. Taj se proces odradi za svaku sliku u listi i tako se doda novi poznati student. Minimum je dodavanje 1 slike, ali može ih biti i više. Ako se sve odradi i uspije, prikaže se success template.

```

# ruta za dodavanje studenta
@app.route('/add_student', methods=['GET', 'POST'])
# moramo bit ulogirani
@login_required
def add_student():
    # dohvaćamo ime iz requesta i izlistavamo sve slike koje smo
    # poslali kroz request (1 je minimum, al more i više)
    if request.method == 'POST':
        name = request.form['name']
        images = request.files.getlist('images')

        # Novi subfolder za nove studente (each student se dela
        # folder)
        student_dir = os.path.join('known_faces', name)
        os.makedirs(student_dir, exist_ok=True)
        for image in images:
            # Dodajemo svaku sliku u studentov folder
            image_path = os.path.join(student_dir, image.filename)
            image.save(image_path)
            # pozivamo add_known_face za svaku sliku
            add_known_face(image_path, name)

        # dodajemo dinamički parametar name u template za success
        return render_template('add_student_success.html', name=name)

    return render_template('add_student.html')

# Route to confirm success => samo izrenderiramo stranicu ako uspije
@app.route('/add_student_success')
def add_student_success():
    return render_template('add_student_success.html')

```

Na vrlo sličan način, implementirana je i funkcija za live dodavanje gdje se ne zahtjeva manualni upload slika, već osoba samo treba gledati u kameru . Dohvaćaju se slike sa camera capture-a i pretvaraju se u embeddinge, normaliziraju i spremaju.

```
# Route to capture live feed images and add a new student
# Uglavnom, ovo je isti vrag kao ono static dodavanje, samo lovimo iz kamere

def add_known_face_from_frame(image_frame, name):
    # koristimo one iste encodinge i nameove od prije
    global known_face_encodings, known_face_names

    # Convert frame to RGB and process it
    image_rgb = cv2.cvtColor(image_frame, cv2.COLOR_BGR2RGB)
    inputs = processor(images=image_rgb, return_tensors="pt")

    # Generate the image feature embedding
    with torch.no_grad():
        outputs = model.get_image_features(**inputs)

    # Normalize the embedding
    embedding = outputs.cpu().numpy().flatten()
    normalized_embedding = embedding / np.linalg.norm(embedding)

    # If `known_face_encodings` is a list, append directly
    if isinstance(known_face_encodings, list):
        known_face_encodings.append(normalized_embedding)
    else:
        # If it's a numpy array, use numpy.vstack to add the new embedding
        known_face_encodings = np.vstack([known_face_encodings,
                                         normalized_embedding])

    # Append the name
    known_face_names.append(name)
    print(f"Added face for {name} from live capture")
```

Ruta za live dodavanje ponovno koristi get i post metode, ali u ovome slučaju se pri post zahtjevu otvara video capture. Definira se varijabla frame_count koja se inicijalizira na 0. Zatim se definira koliko frameova će biti dohvaćeno i započinje snimanje. Snimanje traje sve dok nije dohvaćeno i spremljeno dovoljno frameova ili ga korisnik prekine s q. Radi lakšeg snalaženja slike se nazivaju u formatu "name_timestamp_number".

```

@app.route('/add_student_live', methods=['GET', 'POST'])
def add_student_live():
    if request.method == 'POST':
        name = request.form['name']
        student_dir = os.path.join(dataset_path, name)
        os.makedirs(student_dir, exist_ok=True)

        # Start webcam capture
        cap = cv2.VideoCapture(0)
        frame_count = 0
        required_frames = 25 # Number of frames to capture => 25 po
klasi/osobi (more više, more manje)

        # otpri camera capture i lovi frameove sve dok ih više nema
(ugasimo kameru ili ima dovoljno frejmova); onda samo udri break
        while cap.isOpened():
            ret, frame = cap.read()
            if not ret:
                break

            # Display live video feed
            cv2.imshow("Capture Face - Press 'q' to Quit", frame)

            # Capture every frame => sve dok ih je manje od 25
(required)
            if frame_count < required_frames:
                # Save frame in student's directory
                timestamp = datetime.now().strftime("%Y%m%d_%H%M%S") #
=> dohvati datum i vreme za timestamp za naziv slike
                frame_path = os.path.join(student_dir,
f"{name}_{timestamp}_{frame_count}.jpg") # nazovi sliku
ime_timestamp_redni_broj_framea
                cv2.imwrite(frame_path, frame)

                # Process and save the embedding => spremi tensor i
povećaj frame count za 1
                add_known_face_from_frame(frame, name)
                frame_count += 1

            # Killaj petlju ako ima dovoljno slika ili smo quitali s q
            if cv2.waitKey(1) & 0xFF == ord('q') or frame_count >=
required_frames:

```

```

        break

    # osloboodi kameru, zapri detection window
    cap.release()
    cv2.destroyAllWindows()

    # isto ko manualno dodavanje
    return redirect(url_for('add_student_success', name=name))

return render_template('add_student_live.html')

```

Za svaku uspješno prijavljenu prisutnost, šalje se email notifikacija koristeći sendgrid. Dohvaćaju se podaci o trenutnom predmetu koje smo postavili sa set_subject. Definira se sender_email i proizvoljan broj receiver_emailova. Kreira se mail sa definiranim sadržajem i šalje na odabrane adrese. Printa se response, kako bismo bili sigurni da sve radi dobro. API key dohvata se iz .env datoteke.

```

def send_attendance_notification(name, date, time, subject): # => ovi
podaci nas zanimaju (postavili smo ih kroz set subject i ubacit ćemo ih
u mail)

    message = Mail(
        # izgled maila
        from_email='attendance.logged@gmail.com',
        to_emails='alabinjan6@gmail.com', # napravit neki official mail
za ovaj app da ne koristim svoj mail
        subject=f'Attendance Logged for {name}',
        plain_text_content=f'Attendance for {name} in {subject} on
{date} at {time} was successfully logged.'
    )

try:
    print("Attempting to send email...")
    #sendgrid_api_key = os.getenv('SENDGRID_API_KEY')
    sg = SendGridAPIClient(os.getenv('SENDGRID_API_KEY'))
    response = sg.send(message)
    # debug ako sendgrid ne dela
    print(f"Email sent: {response.status_code}")
    print(f"Response body: {response.body}")
except Exception as e:
    print(f"Error sending email: {str(e)}")

```

Detektirana su lica u frameovima koristeći Haar Cascade. Interno se prebacuju boja frameova u sivu i skalira se veličinu frameova kako bi svi bili cca. jednaki. Zatim se definira varijabla minNeighbors koja označava u koliko se frameova lice mora uzastopno pojaviti kako bi se priznalo kao detekcija i koja mora biti minimalna veličina skeniranog lica. Na kraju se kreira bounding box oko svakog lica pomoću koordinata koje su dohvaćene u prethodnom koraku.

```
def detect_face(frame):
    """Detect faces in the frame using Haar cascades."""
    # Convert the frame (which is in BGR color) to grayscale
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    # Haar Cascade classifier detektira lica
    # `scaleFactor` skalira veličinu da svako detektirano lice bude cca isto
    # `minNeighbors` osigurava da svako lice mora biti u bar 5 frameova da bi ga priznalo
    # `minSize` osigurava minimalnu veličinu
    faces = face_cascade.detectMultiScale(gray, scaleFactor=1.1,
minNeighbors=5, minSize=(30, 30))
    # vraća koordinate svakega lica pomoću bounding boxa
    return faces
```

Implementirana je jednostavna liveness check mehanika koristeći kombinaciju detekcije očiju, treptanja, i pokreta glave. Cilj je osigurati da sustav prepoznae stvarne ljude, a ne fotografije ili druge oblike prijevare. Sastoje se od sljedećih funkcija:

detect_eye_movement(frame, face)

Ova funkcija provjerava detekciju očiju unutar prepoznatog lica. Ako se unutar područja lica detektiraju dva oka, pretpostavlja se da se radi o živom licu jer su oba oka otvorena.

Povratna vrijednost je True ako su oči detektirane, inače False.

detect_eye_blink(face, gray_frame)

Funkcija provjerava treptanje analizom prisutnosti očiju unutar grayscale slike lica. Ako nisu detektirana dva oka, smatra se da su oči zatvorene, što ukazuje na treptanje. Povratna vrijednost je True ako su oči zatvorene, a False ako su otvorene.

detect_head_movement(old_gray, new_gray, faces)

Provjerava pomicanje glave uspoređivanjem razlika između dvije uzastopne slike lica. Izračunava apsolutnu razliku između regija interesa (ROI) lica na staroj i novoj slici te broji značajne promjene piksela. Ako promjene prelaze određeni prag, zaključuje se da je glava pomaknuta. Povratna vrijednost je True ako je detektirano kretanje, inače False.

check_liveness_over_time(frame, faces, old_gray, new_gray)

Ova funkcija integrira provjere treptanja i pokreta glave za svako detektirano lice u nizu uzastopnih sličica (frameova). Ako se detektira i da oči nisu zatvorene i da je glava pomaknuta, povećava se brojač liveness_frame_count. Kada brojač dosegne zadani threshold, zaključuje se da je osoba živa. Povratna vrijednost je True ako je liveness potvrđen, inače False.

check_liveness(frame, faces)

Asinkrona funkcija koja za svako lice brzo provjerava liveness putem detekcije očiju

(koristeći detect_eye_movement). Namjena joj je ubrzati osnovnu provjeru živosti. Povratna vrijednost je True ako je detektiran živi subjekt, a False u suprotnom.

```
def detect_eye_movement(frame, face):
    # lovimo koordinate lica i oči. Ako ulovi 2 oka, prepostavlja da smo čovik
    (x, y, w, h) = face
    roi_gray = frame[y:y + h, x:x + w]
    eyes = eye_cascade.detectMultiScale(roi_gray)

    # Check if two eyes are detected for liveness
    return len(eyes) >= 2 # Live face detected (eyes are open)

# provjerava dali trepćemo
def detect_eye_blink(face, gray_frame):
    (x, y, w, h) = face
    roi_gray = gray_frame[y:y+h, x:x+w]
    eyes = eye_cascade.detectMultiScale(roi_gray)
    return len(eyes) < 2 # Eyes are closed if fewer than 2 eyes are detected

# provjerava dali mičemo glavu (dali ima neke razlike u koordinatama između frameova)
def detect_head_movement(old_gray, new_gray, faces):
    for face in faces:
        (x, y, w, h) = face
        roi_old = old_gray[y:y+h, x:x+w]
        roi_new = new_gray[y:y+h, x:x+w]
        difference = cv2.absdiff(roi_old, roi_new)
        non_zero_count = np.count_nonzero(difference)
        if non_zero_count > 50: # Threshold for movement
            return True
    return False

liveness_frame_count = 0 # init varijable
LIVENESS_FRAMES_THRESHOLD = 5 # Number of consecutive frames for liveness confirmation

# za svako lice provjeravamo dali je pomicalo oči i glavu; ako je, povećamo liveness frame count
def check_liveness_over_time(frame, faces, old_gray, new_gray):
    global liveness_frame_count
    # delaj provjere za sva detektirana lica
    for face in faces:
```

```

# variabile za treptanje i micanje glave (more bit true/false)
eyes_closed = detect_eye_blink(face, new_gray)
head_moved = detect_head_movement(old_gray, new_gray, faces)

if not eyes_closed and head_moved: # oba uvjeta moraju bit zadovoljena da se frame prizna ko live frame
    liveness_frame_count += 1
else:
    liveness_frame_count = 0

if liveness_frame_count >= LIVENESS_FRAMES_THRESHOLD:
    return True # ako liveness pasa u dovoljno frameova, smatramo osobu živom
return False

async def check_liveness(frame, faces):
    # za svako detektirano lice provjeravamo liveness. Stavljen je u async s ciljem da dela bar malo brže
    for face in faces:
        if detect_eye_movement(frame, face):
            return True # Live face
    return False # Spoof

```

Prisutnost se bilježi koristeći funkciju log_attendance. Kad se uspješno definira koja osoba se nalazi u frameovima, taj se podatak proslijedi u funkciju. Dohvaćaju se varijable current_subject, attendance_date, start_time i end_time koje su definirane koristeći set_subject. Zatim provjeravamo je li predmet definiran. Dohvaćamo trenutni datum i vrijeme koristeći datetime objekt i iz njega zasebno izvlačimo datum i vrijeme. Nakon toga kreiramo start_time objekt dohvaćanjem iz set_subject funkcije i definiramo toleranciju dolaska od 14 minuta kasnije od definiranog vremena u objektu koristeći timedelta funkciju. Ako se osoba pokušava prijaviti nakon što je prošlo više od 14 minuta, na frameovima će se nadodati upozorenje "late_entry". Spajamo se na bazu i provjeravamo postoji li već na definirani datum, za definirani predmet i prepoznatu osobu neki zapis o prisutnosti. Ukoliko postoji, obavještavamo korisnika da se već prijavio. Ako ne postoji, spremamo zapis u bazu sa dodanim late flagom koji je po defaultu False, ali se prebací na True ako je ispunjen definirani uvjet za kašnjenje. Po uspješnom zapisivanju prisutnosti još se poziva funkcija send_attendance_notification koja obavještava o prijavi prisutnosti putem maila.

```

# e ovo je zabavno
def log_attendance(name, frame):
    # ovo je globalna varijable jer smo je postavili u drugoj funkciji (set subject i dohvaćamo je tu). Svako loganje attendancea će bit za točno taj subject ako je postavljen
    global current_subject, attendance_date, start_time, end_time

```

```

# ako nismo postavili predmet ili ako nismo unutar intervala
(starttime > x && x < endtime)
    if current_subject is None or not is_within_time_interval():
        print("Subject is not set or current time is outside of allowed
interval. Attendance not logged.")
        return frame

# lovimo trenutni datum i vrijeme
now = datetime.now()
date = now.strftime("%Y-%m-%d")
time = now.strftime("%H:%M:%S")

# Create datetime object for start time => kad počinje loganje
attendancea
start_time_obj = datetime.strptime(f"{attendance_date}"
{start_time}", "%Y-%m-%d %H:%M")

# Toleriramo do 15 minuta kašnjenja (starttime + interval od 14
minuta)
late_time_obj = start_time_obj + timedelta(minutes=14)

# Check if the current time is late => dali je trenutno vrijeme
veće od starttimea za više od late_time_obj(a.k.a 14 minuta)
if now > late_time_obj:
    # za pisat po camera captureu; niš bistro
    cv2.putText(frame, f"Late Entry: {name}", (50, 50),
cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)

# takamo se na bazu i inicializiramo kurzor
conn = sqlite3.connect('attendance.db')
c = conn.cursor()

# provjeravamo dali već postoji u tablici attendance zapis za to
ime, taj datum i taj predmet i fetchamo prvi zakov zapis
c.execute("SELECT * FROM attendance WHERE name = ? AND date = ? AND
subject = ?", (name, date, current_subject))
existing_entry = c.fetchone()

# ako postoji, vraćamo alert da je prisutnost već loggana
if existing_entry:
    print(f"Attendance for {name} on {date} for subject
{current_subject} is already logged.")
    return frame

```

```

# ako ne postoji , pozivamo cursor insert i dodamo zapis
# Late is 1 if we are late, if not then 0 => bool flag za provjeru
c.execute("INSERT INTO attendance (name, date, time, subject, late)
VALUES (?, ?, ?, ?, ?)",
          (name, date, time, current_subject, 1 if now >
late_time_obj else 0))
conn.commit()
conn.close()

print(f"Logged attendance for {name} on {date} at {time} for
subject {current_subject}.")
```

poziva se funkcija za slanje maila

```
send_attendance_notification(name, date, time, current_subject)

return frame
```

Perform_liveness_check pomoću kamere otvara capture i detektira lica na onome što vidi. Ako detektira lica, sprema ih u listu. Zatim kreće iteracija kroz listu i za svako lice se provodi liveness check i crta se bounding box. Kod svakog boxa kreira se pripadna labela i printa se zapis o livenessu u konzoli. Ukoliko je lice klasificirano kao živo, ide se u daljnju provjeru, a ako nije, neće se spremiti nigdje.

```

def perform_liveness_check(frame):
    """Capture video from the camera and perform liveness detection."""
    cap = cv2.VideoCapture(0) # Use the primary camera
    live_face_detected = False # varijable je po defaultu false, pa je
hitimo na true ako detektiramo živost

    # opet ona fora da detecta, sve dok više ne detecta
    ret, frame = cap.read()
    if not ret:
        cap.release()
        return False

    # Detect faces in the current frame
    faces = detect_face(frame)

    # If faces are detected, check for liveness
    for face in faces:
        is_alive = check_liveness(frame, [face])
        (x, y, w, h) = face
```

```

        # Draw a bounding box around the detected face
        cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0), 2) # Blue box

        # dajemo labelu i print
        if is_alive:
            live_face_detected = True
            print("OK, real")
            cv2.putText(frame, "Live Face Detected", (x, y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 255, 0), 2)
        else:
            print("ERROR, fake")
            cv2.putText(frame, "Spoof Detected", (x, y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.8, (0, 0, 255), 2)

        cap.release()

        # daje true/false i o temu ovisi dal ćemo ga spremi ili ne
    return live_face_detected

```

Classify_face funkcija klasificira embeddinge lica koristeći FAISS index i listu poznatih lica koja odgovaraju index-u. K1 parametar definira primarni broj najbližih matcheva za voting, dok K2 parametar definira broj matcheva koji se uzima u razmatranje ukoliko klasifikacija nije uspjela koristeći K1. Threshold parametar definira koji je minimalni prag sličnosti da bi se 2 lica smatrала matchevima. Traži se max(k1, k2) najsličnijih unosa u Faiss indeksu za dani face_embedding. **Face_embedding[np.newaxis, :]** dodaje novu dimenziju kako bi ulaz bio oblikovan kao 2D niz (potrebno za Faiss).

zip(I[0], D[0]): Kombinira indekse (**I[0]**) i udaljenosti (**D[0]**) u parove, tako da svaki indeks iz Faiss-a odgovara udaljenosti tog rezultata od ulaznog embeddinga. (vidimo koliko je daleko ulazni embedding od embeddinga u indeksu i to ponovimo za svaki embedding u indeksu). **idx:** Predstavlja indeks rezultata u Faiss indeksu. Ako je **idx == -1**, to znači da nije pronađen valjni rezultat (Faiss vraća -1 za prazne pozicije).

dist: Udaljenost između ulaznog embeddinga i embeddinga iz Faiss-a. Ako je udaljenost veća od **thresholda**, rezultat se ignorira jer se smatra previše različitim.

votes: Za svaki valjni rezultat, label pridružen indeksu (**known_face_names[idx]**) povećava svoj broj glasova u rječniku **votes**. Na kraju, **majority_class** je label s najviše glasova, dok je "Unknown" povratna vrijednost kad nema glasova

```

def classify_face(face_embedding, faiss_index, known_face_names, k1=3,
k2=5, threshold=0.6):
    """
    Classifies a face embedding using majority voting logic.

    Args:
        face_embedding: The embedding of the face to classify.
        faiss_index: FAISS index for known faces.
    """

```

```

        known_face_names: List of known face names corresponding to
FAISS index.

    k1: Number of nearest neighbors for majority voting.
    k2: Number of fallback neighbors.
    threshold: Similarity threshold for classification.

Returns:
    majority_class: Predicted class label.
    class_counts: Vote counts for each class.

"""
D, I = faiss_index.search(face_embedding[np.newaxis, :], max(k1,
k2))
votes = {}

for idx, dist in zip(I[0], D[0]):
    if idx == -1 or dist > threshold:
        continue
    label = known_face_names[idx]
    votes[label] = votes.get(label, 0) + 1

if votes:
    majority_class = max(votes, key=votes.get)
else:
    majority_class = "Unknown"

return majority_class, votes

```

Generate frames funkcija najprije otvara kameru koristeći OpenCV (cv2.VideoCapture(0)). Argument 0 označava default kameru uređaja. (Kad bismo ih imali više, mogli bismo koristiti 1,2,3 itd.). Provjerava radi li kamera (cap.isOpened()). Ako ne može otvoriti kameru, ispisuje grešku i prekida funkciju. Inicijalizira old_gray za kasniju usporedbu frameova tijekom provjere livenessa. Sve dok je kamera otvorena, snima se trenutni frame pomoću cap.read(), a ako dođe do greške i frame ne uspije biti pročitan, pokušava se ponovno koristeći Continue. Pretvara se trenutni frame u grayscale, jer Haarcascade detektor lica radi bolje na takvim frameovima. Koristi se haarcascade_frontalface_default.xml za detekciju lica u frameu. Dobivaju se koordinate (x, y, w, h) svih detektiranih lica. Ukoliko smo trenutno na prvom frameu (tek smo počeli i old_gray još nije inicijaliziran), kopira se trenutni grayscale frame u old_gray kako bi se proces mogao nastaviti. Na frame se zatim pomoću putText funkcije dodaje tekst koji korisniku daje upute da pomiče glavu kako bi liveness detekcija mogla biti uspješna). Zatim se poziva funkcija za liveness detekciju i ukoliko vrati True, ispisuje se “liveness detected” zelenom bojom. Iteriramo kroz sva pronađena lica u frameovima i kreiramo “slike” za svako lice. Dobivamo niz slika za svako lice i to pretvaramo u embeddinge koje normaliziramo radi usporedbe. Ako postoje poznati embeddings-i, koristi se FAISS za brzu pretragu i pronalazak najbližeg matcha. Zatim se prikazuje naziv klase(ime osobe) s najviše glasova i broj glasova i bounding box oko lica. A ukoliko liveness nije

uspješno detektiran, vidi se samo obavijest o tome, i ništa od gore objašnjenog koda se ne izvodi. Ažurira se frame i prelazi se na idući, kodira se u .jpeg format i priprema za slanje putem web-streama. Na kraju se kamera oslobađa kad se petlja završi.

```
def generate_frames(k1=1, k2=2, threshold=0.5): # placeholder => ne smi  
bit prazno, ali ne služi ničemu  
    # Open the camera  
    cap = cv2.VideoCapture(0)  
    if not cap.isOpened():  
        print("Error: Camera could not be opened.")  
        return  
  
    old_gray = None # Initialize for liveness detection  
  
    while True:  
        ret, frame = cap.read()  
        if not ret:  
            print("Warning: Couldn't grab frame. Retrying...")  
            continue  
  
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)  
        faces = cv2.CascadeClassifier(cv2.data.haarcascades +  
'haarcascade_frontalface_default.xml').detectMultiScale(gray, 1.1, 4)  
  
        if old_gray is None:  
            old_gray = gray.copy()  
  
        # Display liveness instructions  
        cv2.putText(frame, "Move your head around", (50, 30),  
cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 0), 2)  
  
        if check_liveness_over_time(frame, faces, old_gray, gray):  
            cv2.putText(frame, "Liveness Confirmed", (50, 70),  
cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2)  
  
        for (x, y, w, h) in faces:  
            face_image = frame[y:y+h, x:x+w]  
            face_rgb = cv2.cvtColor(face_image, cv2.COLOR_BGR2RGB)  
            inputs = processor(images=face_rgb,  
return_tensors="pt")  
  
        with torch.no_grad():
```

```

        outputs = model.get_image_features(**inputs)
        face_embedding = outputs.cpu().numpy().flatten()
        face_embedding /= np.linalg.norm(face_embedding)  #

Normalize embedding

        if len(known_face_encodings) > 0:
            # Perform majority vote classification
            majority_class, class_counts = classify_face(
                face_embedding, faiss_index, known_face_names,
                k1, k2, threshold
            )

            # Format vote results for display
            match_text = f"{majority_class} ({class_counts[majority_class]} votes)"
            if majority_class != "Unknown":
                frame = log_attendance(majority_class, frame)
            else:
                majority_class = "Unknown"
                match_text = "Unknown (0 votes)"

            # Draw rectangle and label on the frame
            cv2.rectangle(frame, (x, y), (x + w, y + h), (255, 0, 0), 2)
            cv2.putText(frame, match_text, (x, y - 10),
cv2.FONT_HERSHEY_SIMPLEX, 0.9, (255, 255, 255), 2)
            else:
                cv2.putText(frame, "Liveness Failed: No Movement Detected",
(50, 70), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)

        old_gray = gray.copy()

        # Encode and yield the frame
        ret, buffer = cv2.imencode('.jpg', frame)
        frame = buffer.tobytes()
        yield (b'--frame\r\n'
               b'Content-Type: image/jpeg\r\n\r\n' + frame + b'\r\n')

    cap.release()

```

Na video_feed ruti se samo poziva generate_frames funkcija na način da se svaki frame vrati kao poseban response.

```
# na ruti za video feed se poziva generate frames funkcija
```

```

@app.route('/video_feed')
def video_feed():
    return Response(generate_frames(),
mimetype='multipart/x-mixed-replace; boundary=frame') # http live
streaming di je svaki frame zasebni response

```

Na / ruti se prikazuje početna stranica aplikacije. Najprije se iz .env učitava API ključ za dohvati vremenske prognoze i postavlja se lokacija za koju nas prognoza zanima. Poziva se funkcija get_weather_forecast koja prima api_key i željenu lokaciju te vraća odgovor API-ja s prognozom. Taj se odgovor prosljedi u funkciju predict_absence_due_to_weather koja na temelju određenih ključnih riječi u prognozi klasificira prognozu kao "lošu" ili "dobru". Ako je "loša", prikazuje se poruka o mogućim kašnjenjima zbog lošeg vremena. Nadalje, provjerava se je li korisnik u trenutnoj sesiji video privacy_policy popup. Ako ga je već viđao, neće se prikazati, ali ako prvi put pristupa aplikaciji, vidjet će privacy_policy popup.

```

# / ruta. Početna stranica
@app.route('/')
def index():
    # dohvativamo api ključ iz env
    api_key = os.getenv('WEATHER_API_KEY')
    location = "Pula" # Ili nešto drugo
    # pozivamo funkciju za get_weather_forecast
    weather_condition = get_weather_forecast(api_key, location)

    # Logika za vremensku prognozu => ako dohvaćena prognoza sadrži
    # neki bad keyword, predictamo da će bit loše u suprotnem, će bit ok
    if predict_absence_due_to_weather(weather_condition):
        message = "Bad weather predicted, late entries due to traffic
problems are possible."
    else:
        message = "No significant weather issues expected. Students
should come on time."

    # Provjera je li korisnik već viđao popup => kad se prvi prvi put ulogiramo, dobit ćemo popup o privacy-ju
    if 'seen_privacy_policy' not in session:
        show_popup = True
        session['seen_privacy_policy'] = False # Po defaultu je
false, ako ga nismo vidili (logično :D)
    else:
        show_popup = False

```

```

        return render_template('index.html',
weather_condition=weather_condition, message=message,
show_popup=show_popup)

```

Na ruti /attendance, šalje se GET zahtjev koji dohvata zapise o prisutnosti iz baze. Korisnik mora biti ulogiran da bi joj pristupio. Omogućeno je 7 vrsta filtera koji se dohvataju iz request parametara; filteri po: imenu, predmetu, datumu, danu tjedna, mjesecu, godini i kašnjenju. Spajamo se na bazu i instanciramo cursor te dinamički gradimo upit koji preko cursora šaljemo na bazu. Svaki dodani filter se dinamički appenda u upit koji se na kraju izvodi. Pomoću fetchall() dohvataćemo sve zapise koji odgovaraju zadanim uvjetima i grupiramo ih po datumu tako da svaki datum ima svoj dictionary gdje su predmeti keyevi, a vrijednosti su svi studenti koji su bili prisutni. Na kraju se ti zapisi dinamički izrenderiraju u html stranici.

```

# GET ruta za dohvata zapisa o prisutnosti
@app.route('/attendance', methods=['GET'])
@login_required
def attendance():
    # Lovimo filter parametre iz request argumenta => ako ga nema u
    # request argumentu, onda je taj parametar prazan
    name_filter = request.args.get('name')
    subject_filter = request.args.get('subject')
    date_filter = request.args.get('date')
    weekday_filter = request.args.get('weekday')
    month_filter = request.args.get('month')
    year_filter = request.args.get('year')
    late_filter = request.args.get('late')

    # connectamo se na bazu i upalimo cursor
    conn = sqlite3.connect('attendance.db')
    c = conn.cursor()

    # Dynamic build of query
    query = "SELECT rowid, subject, name, date, time, late FROM
    attendance WHERE 1=1"
    params = []
    # ako smo dali određeni argument, onda appendamo taj argument u
    # tekst querija i u parametre
    if name_filter:
        query += " AND name = ?"
        params.append(name_filter)

    if subject_filter:
        query += " AND subject = ?"
        params.append(subject_filter)

```

```

if date_filter:
    query += " AND date = ?"
    params.append(date_filter)

if weekday_filter:
    query += " AND strftime('%w', date) = ?"
    params.append(weekday_filter)

if month_filter:
    query += " AND strftime('%m', date) = ?"
    params.append(f"{int(month_filter):02d}")

if year_filter:
    query += " AND strftime('%Y', date) = ?"
    params.append(year_filter)

if late_filter:
    query += " AND late = ?"
    params.append(late_filter)

query += " ORDER BY date, time"
c.execute(query, params)

# dohvatimo sve rezultate koji odgovaraju filteru
records = c.fetchall()
conn.close()

# Group records by date and subject => dictionary za recordse po
# datumu. Svaki datum ima predmete za keyeve a values si svi studenti
# koji su bili tamo
grouped_records = {}
for rowid, subject, name, date, time, late in records:
    if date not in grouped_records:
        grouped_records[date] = {}
    if subject not in grouped_records[date]:
        grouped_records[date][subject] = []
    grouped_records[date][subject].append((rowid, name, time,
late))

return render_template('attendance.html',
grouped_records=grouped_records)

```

Definirane su 2 gotovo identične rute: /download i /download_and_email. Jedina razlika je to što druga dodatno još šalje preuzeti csv zapis na email, pa će objasniti samo nju, pošto je onda direktna nadogradnja /download rute. Spajamo se na bazu, instancira se kurzor i pomoću njega dohvataju se svi zapisi iz baze. Sortiraju se po predmetu, datumu i vremenu. Podaci se zapisuju u CSV datoteku koristeći StringIO. Svaki predmet se grupira, a broj studenata po predmetu dodaje se na kraju grupe. Zaglavje CSV datoteke sadrži stupce Subject, Name, Date, Time i Number of students. Na kraju se dohvata e-mail korisnika iz URL parametara pomoću request.args.get('email'). CSV podaci i korisnički e-mail prosleđuju se u HTML template download.html. Korisnik može preuzeti generiranu CSV datoteku putem preglednika.

```
@app.route('/download_and_email')
def download_and_email_attendance():
    conn = sqlite3.connect('attendance.db')
    c = conn.cursor()
    c.execute("SELECT subject, name, date, time FROM attendance ORDER
    BY subject, date, time")
    records = c.fetchall()
    conn.close()

    output = io.StringIO()
    writer = csv.writer(output)

    previous_subject = None
    student_count = 0

    writer.writerow(['Subject', 'Name', 'Date', 'Time', 'Number of
    students'])

    for record in records:
        subject, name, date, time = record

        if subject != previous_subject:
            if previous_subject is not None:
                writer.writerow(['', '', '', '', student_count])
                writer.writerow([])
            writer.writerow([subject])
            previous_subject = subject
            student_count = 0

        writer.writerow(['', name, date, time])
        student_count += 1

    if previous_subject is not None:
        writer.writerow(['', '', '', '', student_count])
```

```

        output.seek(0)

        csv_data = output.getvalue()

        # dohvatimo request argumente
        user_email = request.args.get('email')

        return render_template('download.html', csv_data=csv_data,
user_email=user_email)

```

Svaki je zapis o prisutnosti moguće obrisati spajanjem na bazu i instanciranjem kurzora koji izvodi sql upit koji briše zapis po id-ju na način da se taj id mora podudarati s id-jem iz zahtjeva. Iako, bi bilo logičnije koristiti delete metodu za ovaj zadatak, bilo je lakše implementirati post request zbog lakšeg rada s browserom. Praktički se samo šalje upit koji će nešto obrisati.

```

# brisanje po id-ju => spojimo se na bazu, pošaljemo query
@app.route('/delete_attendance/<int:id>', methods=['POST'])
def delete_attendance(id):
    conn = sqlite3.connect('attendance.db')
    c = conn.cursor()
    c.execute("DELETE FROM attendance WHERE rowid = ?", (id,))
    conn.commit()
    conn.close()
    return redirect(url_for('attendance'))

```

Moguć je statistički pregled podataka spajanjem na bazu i izvođenjem specifičnih sql upita za dohvat pojedinih informacija. Trenutno je implementiran pregled prisutnosti po studentu i po predmetu, no moguća je vrlo jednostavna i brza implementacija dodatnih statistika.

```

@app.route('/statistics')
@login_required
def statistics():
    conn = sqlite3.connect('attendance.db')
    c = conn.cursor()

    c.execute("SELECT name, subject, COUNT(*) FROM attendance GROUP BY
name, subject")
    student_attendance = c.fetchall()

    c.execute("SELECT subject, COUNT(*) FROM attendance GROUP BY
subject")
    subject_attendance = c.fetchall()

    conn.close()

```

```

    return render_template('statistics.html',
student_attendance=student_attendance,
subject_attendance=subject_attendance)

```

Pregledavanje svih studenata koji su ikada zabilježili prisutnost je inicijalno bila debug ruta zbog problema s prikazom studenata u zapisima o prisutnosti. No, ostavljena je u aplikaciji i nakon rješavanja tih poteškoća. Ruta funkcioniра tako što se spaja na bazu i dohvaca sva distinct imena studenata koji su ikad prijavljivali prisutnost. (Svaki student se dohvati samo jednom, neovisno o tome koliko puta se prijavljivao i koliko predmeta sluša)

```

@app.route('/students')
@login_required
def students():
    conn = sqlite3.connect('attendance.db')
    c = conn.cursor()
    c.execute("SELECT DISTINCT name FROM attendance ORDER BY name")
    students = c.fetchall()
    conn.close()

    return render_template('students.html', students=students)

```

Ove rute omogućuju generiranje i prikaz različitih grafikona temeljenih na podacima prisutnosti pohranjenim u SQLite bazi podataka. Svaka ruta koristi matplotlib i seaborn za generiranje vizualizacija koje se zatim vraćaju korisnicima kao PNG slike pomoću BytesIO objekta koji omogućuje slanje putem HTTP-a. Važno je napomenuti kako nije dobra ideja kreirati grafikone ukoliko nema podataka u bazi, jer to generira error.

/plot/student_attendance: Ova ruta prikazuje grafikon koji prikazuje broj prisutnosti po studentu. Podaci se dohvate grupiranjem prisutnosti po imenu studenta, a grafikon je prikazan kao horizontalni bar plot s brojem prisutnosti na X-osi i imenom studenta na Y-osi.

/plot/subject_attendance: Ova ruta generira pie-chart koji prikazuje udjel prisutnosti po predmetu. Podaci se dohvataju grupiranjem prisutnosti prema predmetu, a rezultat je prikazan u obliku pie chart-a s udjelima za svaki predmet.

/plot/monthly_attendance: Ova ruta također generira pie-chart, ali ovdje su podaci grupirani prema datumima (mjesecima) prisutnosti. Grafikon prikazuje raspodjelu prisutnosti kroz različite datume.

```

# Ove rute su sve za grafikone => fetchaš podatke iz baze, malo ih
obradiš i plotaš s matplotlibom
# BITNO!!! NE NANKA POKUŠAVAT OPIRAT PLOTOVE AKO JOŠ NEMA ZABILJEŽENIH
PRISUTNOSTI, JER ĆE SE ZBREJKAT
@app.route('/plot/student_attendance')
def plot_student_attendance():
    conn = sqlite3.connect('attendance.db')

```

```

c = conn.cursor()

c.execute("SELECT name, COUNT(*) as count FROM attendance GROUP BY
name")
data = c.fetchall()
conn.close()

# Prepare data for plotting
names, counts = zip(*data)

# Create the plot
plt.figure(figsize=(10, 6))
sns.barplot(x=counts, y=names, palette='viridis')
plt.title('Attendance by Student')
plt.xlabel('Number of Attendances')
plt.ylabel('Student')

# Save the plot to a BytesIO object and return it as a response
img = io.BytesIO()
plt.savefig(img, format='png')
img.seek(0)
plt.close()
return send_file(img, mimetype='image/png')

@app.route('/plot/subject_attendance')
def plot_subject_attendance():
    conn = sqlite3.connect('attendance.db')
    c = conn.cursor()

    c.execute("SELECT subject, COUNT(*) as count FROM attendance GROUP
BY subject")
    data = c.fetchall()
    conn.close()

    # Prepare data for plotting
    subjects, counts = zip(*data)

    # Create the plot
    plt.figure(figsize=(8, 8))
    plt.pie(counts, labels=subjects, autopct='%.1f%%',
    colors=sns.color_palette('pastel'))
    plt.title('Attendance by Subject')

```

```

# Save the plot to a BytesIO object and return it as a response
img = io.BytesIO()
plt.savefig(img, format='png')
img.seek(0)
plt.close()
return send_file(img, mimetype='image/png')


@app.route('/plot/monthly_attendance')
def plot_monthly_attendance():
    conn = sqlite3.connect('attendance.db')
    c = conn.cursor()

    c.execute("SELECT date, COUNT(*) as count FROM attendance GROUP BY date")
    data = c.fetchall()
    conn.close()

    # Prepare data for plotting
    dates, counts = zip(*data)

    # Create the plot
    plt.figure(figsize=(8, 8))
    plt.pie(counts, labels=dates, autopct='%1.1f%%',
    colors=sns.color_palette('pastel'))
    plt.title('Monthly Attendance Distribution')

    # Save the plot to a BytesIO object and return it as a response
    img = io.BytesIO()
    plt.savefig(img, format='png')
    img.seek(0)
    plt.close()
    return send_file(img, mimetype='image/png')

```

Svakom od grafikona može se pristupiti preko plot-routera koji prikazuje korisniku tipke za kreiranje pojedinog grafikona. Najprije provjerava ima li odgovarajućih zapisa u bazi. Ukoliko nema, prikazat će upozorenje putem Flaskove flash funkcije.

```

@app.route('/plots')
@login_required
def plots():
    conn = sqlite3.connect('attendance.db')
    c = conn.cursor()

```

```

c.execute('SELECT COUNT(*) FROM attendance')
attendance_count = c.fetchone()[0] # Get the count from the result
conn.close()

# Provjera dali ima podataka; ako nema, ne otvarat !!
if attendance_count > 0:
    return render_template('plot_router.html')
else:
    flash("No attendance records found. Please add some attendance
data before viewing the plots, because everything will break if you try
to plot non-existing data <3", "error")
    return render_template('flash_redirect.html') # Render a new
template for displaying the message

```

Funkcija `get_weather_forecast` koristi API za vremensku prognozu i šalje zahtjev za određenu lokaciju koja je odabrana. Response se dohvata pomoću `requests.get` i zatim se pretvara u JSON kako bi se iz njega mogli dohvatiti traženi podaci i vratiti kao rezultat funkcije.

```

def get_weather_forecast(api_key, location="your_city"):
    url =
f"http://api.weatherapi.com/v1/forecast.json?key={api_key}&q={location}
&days=1"
    response = requests.get(url)
    data = response.json()
    return
data["forecast"]["forecastday"][0]["day"]["condition"]["text"]

```

Funkcija `predict_absence_due_to_weather` uzima dohvaćenu prognozu, provjerava sve riječi unutar prognoze te vraća True ukoliko je neka od tih riječi jednaka keywordsim koje smo definirali kao `bad_weather`.

```

def predict_absence_due_to_weather(weather_condition):
    bad_weather_keywords = ["rain", "storm", "snow", "fog",
" hurricane"]
    for keyword in bad_weather_keywords:
        if keyword in weather_condition.lower():
            return True
    return False

```

Rut `predict_absence` koristi `get` metodu za dohvat odgovora od API-ja koristeći funkciju `get_weather_forecast`. Kad dohvati prognozu, sprema je u varijablu `weather_condition` koju prosljeđuje u funkciju `predict_absence_due_to_weather`. Ako ta funkcija vrati True, ispisuje se poruka o lošem vremenu i mogućim kašnjenjima. U suprotnom, ispisuje se poruka o tome da je sve OK.

```
@app.route('/predict_absence', methods=['GET'])
```

```

def predict_absence():
    api_key = os.getenv("WEATHER_API_KEY")
    location = "London"
    weather_condition = get_weather_forecast(api_key, location)

    if predict_absence_due_to_weather(weather_condition):
        message = "Bad weather predicted, late entries due to traffic problems are possible."
    else:
        message = "No significant weather issues expected. Students should come on time"

    return jsonify({
        "weather_condition": weather_condition,
        "message": message
    })

```

Koristi se NLP model za rastavljanje forumskih poruka na tokene i pretvorbu u tensore. Oni se zatim proslijede u model koji provjerava kolika je vjerojatnost da je poruka potencijalno uvredljiva. Ako je vjerojatnost veća od 30%, poruka je flagirana kao uvredljiva i neće se prikazati.

```

# učitavanje NLP modela za filtriranje neprimjerjenih riječi
tokenizer, model2 = load_model_and_tokenizer()
# prima message, rastavlja je na tokene i pretvori u tensor (threshold označava koliko je osjetljiv)
# vraća koji je probability da je poruka offensive. Ako je offensiveness veći od 30%, flag-a poruku i ne prikaže je
def is_inappropriate_content(message, threshold=0.30):
    inputs = tokenizer(message, return_tensors="pt")
    outputs = model2(**inputs)
    probs = torch.nn.functional.softmax(outputs.logits, dim=-1)
    offensive_score = probs[0][1].item() #label index 1 corresponds to "offensive" class
    return offensive_score > threshold

```

Ruta announcements služi za dohvati svih objavljenih poruka. Spaja se na bazu, dohvaća sve objave i sortira ih po datumu silazno.

```

# OVO JE KAO NEKI PROFESORSKI FORUM/CHAT ILI ČA JA ZNAN KAKO BI SE TO ZVALO
# Niš pametno; običan sql upit
@app.route('/announcements', methods=['GET'])
@login_required
def announcements():

```

```

conn = sqlite3.connect('attendance.db')
c = conn.cursor()
c.execute("SELECT * FROM announcements ORDER BY date_time ASC")
announcements = c.fetchall()
conn.close()
return render_template('announcements.html',
announcements=announcements)

```

Pri objavi poruke, sadržaj se dohvati iz request.form. Ako je taj form prazan, objava se onemogući jer se prazna poruka ne može objaviti. Ukoliko ima teksta, onda se on proslijedi u is_inappropriate_content() za provjeru je li taj tekst primjeren. Ako nije, prikazat će se pripadna poruka. Ako je sve ok, onda se dohvaća trenutno vrijeme i trenutni korisnik kako bi se u bazu spremili svi podaci o poruki.

```

@app.route('/post_announcement', methods=['POST'])
# samo šaljemo post s tekstom poruke, ubacimo ga u model i vidimo dali
# se more objavit. Ako da, spremi se u bazu i prikaže. Autor je trenutno
# ulogirani korisnik
@login_required
def post_announcement():
    message = request.form['message']
    if not message:
        return jsonify({"error": "Message cannot be empty."}), 400
    if is_inappropriate_content(message):
        return redirect(url_for('announcements', warning="Your
announcement contains inappropriate language and was not posted."))
    date_time = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    teacher_name = current_user.username

    conn = sqlite3.connect('attendance.db')
    c = conn.cursor()
    c.execute("INSERT INTO announcements (date_time, teacher_name,
message) VALUES (?, ?, ?)",
              (date_time, teacher_name, message))
    conn.commit()
    conn.close()
    return redirect(url_for('announcements'))

```

Ruta za brisanje poruke poprilično je jednostavna. Traži poruku preko id-ja i šalje sql upit koji briše poruku s tim id-jem.

```

@app.route('/delete_announcement/', methods=['POST'])
@login_required
def delete_announcement(id):
    conn = sqlite3.connect('attendance.db')

```

```

c = conn.cursor()
c.execute("DELETE FROM announcements WHERE id = ?", (id,))
conn.commit()
conn.close()
return redirect(url_for('announcements'))

```

Ruta za uređivanje poruke je, pak, prilično dugačka. Podržava i get i post metode. Get dio omogućava dohvati pojedine poruke po id-ju, ali samo ako je autor poruke jednak trenutno ulogiranom korisniku. Post dio dohvata poruku u json formatu, izmjenjuje tekst sa novim tekstrom koji je korisnik napisao, dohvata trenutni datum i vrijeme kao vrijeme uređivanja poruke i trenutnog korisnika kao autora poruke. Zatim provjerava je li izmjenjena poruka neprimjerena na isti način kao i kod kreiranja poruke. Ako je poruka primjerena, izvodi se update upit na bazi i sprema se uređena poruka.

```

@app.route('/announcements/<int:id>', methods=['GET', 'POST'])
@login_required
def edit_announcement(id):

    if request.method == 'POST':
        data = request.get_json() # Expecting JSON data for POST
        message = data.get('message')
        date_time = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        teacher_name = current_user.username

        if not message:
            return jsonify({"error": "Message cannot be empty."}), 400

        # Check for inappropriate content
        if is_inappropriate_content(message, threshold=0.5):
            return jsonify({"error": "Announcement contains
inappropriate content."}), 400

    try:
        with sqlite3.connect('attendance.db') as conn:
            c = conn.cursor()
            c.execute("""
                UPDATE announcements
                SET date_time = ?, message = ?
                WHERE id = ? AND teacher_name = ?
            """, (date_time, message, id, teacher_name))
            if c.rowcount == 0: # If no rows are updated
                return jsonify({"error": "Announcement not found or
not authorized."}), 404

```

```

        return jsonify({"success": "Announcement updated successfully."}), 200
    except sqlite3.Error as e:
        return jsonify({"error": f"Database error: {str(e)}"}), 500

    elif request.method == 'GET':
        try:
            with sqlite3.connect('attendance.db') as conn:
                c = conn.cursor()
                c.execute("SELECT * FROM announcements WHERE id = ?",
                          (id,))
                announcement = c.fetchone()

            if announcement is None or announcement[2] != current_user.username:
                return redirect(url_for('announcements'))

            return render_template('edit_announcement.html',
                                  announcement=announcement)
        except sqlite3.Error as e:
            flash(f"Error retrieving announcement: {str(e)}", "error")
            return redirect(url_for('announcements'))

```

Ruta /report generira izvještaj o prisutnosti studenata za svaki predmet tako što iz baze podataka izračunava ukupan broj održanih predavanja svakog predmeta. Zatim za svakog studenta koji je slušao pojedini predmet računa broj odslušanih predavanja. Kreira se lista students_with_status u koju će se dodati svi studenti s njihovim postocima i statusima. Postotak za svakog studenta po predmetu se izračuna dijeljenjem broja odslušanih predavanja i broja ukupnih predavanja za taj predmet i množenjem sa 100. Podaci se grupiraju po imenu studenta kako bi se dobio jedan zapis po studentu za svaki predmet. Nakon što svaki student na predmetu dobije svoj postotak, provjerava se je li taj postotak veći od 50%. Ako jest, prikazuje se kvačica koja govori da je student zadovoljio zahtjev prisutnosti, a ako nije, prikazuje se crveni x. Izračunava se i prosječna prisutnost za pojedini predmet i na kraju se sve to izrenderira u html template.

```

#generira izvještaj o prisutnosti studenata za svaki predmet.
# Iz baze podataka izračunava ukupan broj održanih predavanja, postotak
prisutnosti svakog studenta, te prosječnu prisutnost za predmet.
# Studenti se također označavaju kao oni koji ispunjavaju ili ne
ispunjavaju prag od 50% prisutnosti.
# Generirani izvještaj prosljeđuje se predlošku attendance_report.html
za prikaz.
@app.route('/report')
@login_required

```

```

def report():
    conn = sqlite3.connect('attendance.db')
    conn.row_factory = sqlite3.Row
    cur = conn.cursor()

    cur.execute('SELECT DISTINCT subject FROM attendance')
    subjects = cur.fetchall()

    report = []

    for subject in subjects:
        subject_name = subject['subject']

        cur.execute('SELECT COUNT(DISTINCT date) as total_lectures FROM attendance WHERE subject = ?', (subject_name,))
        total_lectures = cur.fetchone()['total_lectures']

        cur.execute('''SELECT name, COUNT(*) as attended_lectures,
                           (COUNT(*) * 100.0 / ?) as attendance_percentage
                      FROM attendance
                     WHERE subject = ?
                    GROUP BY name
                    ORDER BY attendance_percentage DESC''',
                   (total_lectures, subject_name))
        students_attendance = cur.fetchall()

        students_with_status = []
        for student in students_attendance:
            meets_requirement = student['attendance_percentage'] >= 50
            students_with_status.append({
                'name': student['name'],
                'attended_lectures': student['attended_lectures'],
                'attendance_percentage':
                    student['attendance_percentage'],
                'meets_requirement': meets_requirement
            })

        cur.execute('''SELECT AVG(attendance_percentage) as avg_attendance
                           FROM (SELECT COUNT(*) * 100.0 / ? as
                                 attendance_percentage
                                FROM attendance
                               WHERE subject = ?)
            ''')

```

```

        GROUP BY name) ''',
        (total_lectures, subject_name))
avg_attendance = cur.fetchone()['avg_attendance']

report.append({
    'subject': subject_name,
    'total_lectures': total_lectures,
    'average_attendance': avg_attendance,
    'students': students_with_status
})

conn.close()

return render_template('attendance_report.html', report=report)

```

Analiziraju se kasni dolasci na predavanje s ciljem optimizacije rasporeda i minimizacije kašnjenja. Za to se koristi counter iz biblioteke collections. Spajanjem na bazu, dohvaćaju se svi zapisi gdje je late flag postavljen na 1. I spremaju u listu. Konekcija se zatvara i inicijaliziraju se 2 counter-a. Jedan za sate i drugi za dane u tjednu. Iz svakog zapisa u listi late_entries dohvaća se datum i vrijeme ulaska koji se zatim konvertiraju u datetime objekte (1 H:M:S zapis i 1 Y:M:D zapis). Nakon toga se prebrojavaju pojavljivanja pojedinih dana i pojedinih sati u zapisima o kašnjenju i dohvaćaju se najčešći dan i najčešći sat kada dolazi do kašnjenja. Zatim se kreira grafikon s brojevima 0-23 na x-osi koji predstavljaju sate u danu i s brojevima kasnih dolazaka na y-osi što omogućava uvid u broj kašnjenja za svaki pojedini sat u danu.

```

# Ruta koja će dohvatiti sva kašnjenja, i prikazati grafikon da se vidi
# kad najviše kasne
# Koristimo collections/counter za brojanje najčešćih sati i dana kad
# ljudi kasne
@app.route('/late_analysis', methods=["GET", "POST"])
@login_required
def late_entries():
    conn = sqlite3.connect('attendance.db')
    c = conn.cursor()

    # Fetch all late entries
    c.execute("SELECT * FROM attendance WHERE late = 1")
    late_entries = c.fetchall()
    conn.close()

    # Init counters
    hour_counter = Counter()
    weekday_counter = Counter()

```

```

for entry in late_entries:
    time_in = entry[2]
    date = entry[1]

    # Convert time and date
    time_obj = datetime.strptime(time_in, "%H:%M:%S")
    date_obj = datetime.strptime(date, "%Y-%m-%d")
    # Count the hour
    hour_counter[time_obj.hour] += 1

    # Count the weekday (0=Monday, 6=Sunday)
    weekday_counter[date_obj.weekday()] += 1

# Convert results to lists => rendering
most_common_hour = hour_counter.most_common(1)[0] if hour_counter
else None
most_common_weekday = weekday_counter.most_common(1)[0] if
weekday_counter else None

# Show all hours from 00 to 23
hours = list(range(24))
hour_counts = [hour_counter.get(hour, 0) for hour in hours] # Get
count or 0 if not in the counter

# Visualization
if hour_counter:
    plt.bar(hours, hour_counts)
    plt.xticks(hours) # Ensure all hours are labeled on the x-axis
    plt.xlabel('Hour of the Day')
    plt.ylabel('Number of Late Entries')
    plt.title('Late Entries by Hour')

    img = io.BytesIO()
    plt.savefig(img, format='png')
    img.seek(0)
    plot_url = base64.b64encode(img.getvalue()).decode()
    plt.close()
else:
    plot_url = None

return render_template('late_entries.html',
                      late_entries=late_entries,
                      most_common_hour=most_common_hour,

```

```
most_common_weekday=most_common_weekday,
plot_url=plot_url)
```

Funkcija `scrape_github_profile` služi za web-scraping podataka sa Githuba. Dohvaća se cijela stranica u `response`-u i zatim se parsira kao tekst html filea. Traže se samo određeni dijelovi stranice za prikaz, konkretno to su: username, bio, profilna slika, broj followera i broj javnih rezpositorija. To se vraća kao objekt koji će se prikazati u HTML-u.

```
def scrape_github_profile(url):
    try:
        # http req na github profil (moj url)
        response = requests.get(url)
        response.raise_for_status()

        # parsiramo cijelu stranicu u html-u
        soup = BeautifulSoup(response.text, 'html.parser')

        # tražimo dio s usernameom
        name = soup.find('span', class_='p-name').text.strip()

        # tražimo dio s biography
        bio = soup.find('div', class_='p-note user-profile-bio mb-3 js-user-profile-bio f4').text.strip() if soup.find('div', class_='p-note user-profile-bio mb-3 js-user-profile-bio f4') else 'No bio available'

        # tražimo profile picture
        profile_picture = soup.find('img', class_='avatar-user')['src']
        if soup.find('img', class_='avatar-user') else None

        # tražimo broj followera
        followers = soup.find('span', class_='text-bold').text.strip()

        # tražimo broj public rezpositorija
        repositories_element = soup.find('span', class_='Counter')
        repositories = repositories_element.text.strip() if repositories_element else '0' # => hendlanje slučaja ako ih ima 0

        # vraćanje scrapearnih podataka
        return {
            'name': name,
            'bio': bio,
            'profile_picture': profile_picture,
            'followers': followers,
```

```

        'repositories': repositories,
    }

except Exception as e:
    print(f"Error scraping the website: {e}")
    return None

```

Podatke koji su dohvaćeni s Githuba, dinamički se renderira koristeći render_template_string_funkciju koja omogućava da se definira izgled HTML stranice unutar same rute, bez potrebe da se u templates direktoriju kreira poseban novi file.

```

@app.route('/scrape_github', methods=['GET'])
def github_profile():
    # koji url?
    url = 'https://github.com/AntonioLabinjan'

    if not url:
        return jsonify({"error": "Please provide a GitHub profile URL"}), 400

    profile_info = scrape_github_profile(url)

    if profile_info:
        # ako smo nešto našli, izrenderirat ćemo stranicu s ovim contentom
        html_content = f"""
        <!DOCTYPE html>
        <html lang="en">
        <head>
            <meta charset="UTF-8">
            <meta name="viewport" content="width=device-width,
initial-scale=1.0">
            <title>GitHub Profile</title>
            <style>
                body {{{
                    font-family: Arial, sans-serif;
                    background-color: #111;
                    color: #f9f9f9;
                    margin: 0;
                    padding: 0;
                }}}
                .container {{{
                    max-width: 800px;
                    margin: 50px auto;
                    padding: 20px;
                }}}
            </style>
        </head>
        <body>
            <div class="container">
                <h1>GitHub Profile</h1>
                <p>Name: Antonio Labinjan</p>
                <p>Profile URL: https://github.com/AntonioLabinjan</p>
                <p>Bio: A software engineer with a passion for Python and machine learning. I have experience in web development, data analysis, and AI research. I am currently working on a project to build a AI-powered code generator for developers.
                </div>
            </body>
        </html>
    """

```

```

        background-color: #222;
        border-radius: 8px;
        box-shadow: 0 4px 8px rgba(0, 0, 0, 0.3);
        text-align: center;
    } }
h1 { {
    color: #ff6600;
} }
p { {
    font-size: 18px;
    line-height: 1.6;
} }
.followers, .repositories { {
    font-weight: bold;
    color: #ff6600;
    font-size: 20px;
} }
.profile-picture { {
    width: 150px;
    height: 150px;
    border-radius: 50%;
    border: 2px solid #ff6600;
    margin-bottom: 20px;
} }
.back-button { {
    margin-top: 20px;
    padding: 10px 20px;
    font-size: 18px;
    color: #fff;
    background-color: #ff6600;
    border: none;
    border-radius: 5px;
    cursor: pointer;
} }
.back-button:hover { {
    background-color: #cc5200;
} }
</style>
</head>
<body>
<div class="container">
    

```

```

        <h1>{profile_info['name']}</h1>
        <p><strong>Bio:</strong> {profile_info['bio']}

```

Prikaz akademskog kalendarisa izveden je pomoću scrapinga PDF kalendarisa sa službenih UNIPU stranica. Dohvaća se pomoću url-a i otvara se ukoliko je odgovor uspješan. Otvara se dokument i extracta se sav tekst iz PDF-a. Zanimljivo je to što za dohvaćanje PDF-a s UNIPU stranica treba isključiti SSL.

```

def extract_pdf_text(pdf_url):
    # dajemo request za dohvati pdf-a , ugasimo ssl
    response = requests.get(pdf_url, verify=False)
    # ako je response ok, skinemo fajl
    if response.status_code == 200:
        with open("calendar.pdf", "wb") as f:
            f.write(response.content)

        # Extract texta iz fajla
        text_content = ""
        with pdfplumber.open("calendar.pdf") as pdf:
            for page in pdf.pages:
                text_content += page.extract_text() + "\n"
        return text_content
    else:
        return "Failed to retrieve PDF."

```

Nakon dohvaćanja teksta iz PDF-a, extractaju se svi neradni dani pomoću keywordsa. I spremaju se u listu gdje se svaki novi zapis odvaja sa /n. Na kraju se svi join-aju u 1 popis koji se prikazuje na frontendu koristeći unoredered listu s bullet pointsima i dinamičko renderiranje koristeći render_template_string.

```

def get_non_working_days(text):
    # Define keywords

```

```

keywords = [
    "Blagdan", "Praznik", "nenastavni", "odmor", "Božić", "Nova
Godina",
    "Tijelovo", "Dan sjećanja", "Uskrs", "Svi sveti", "Sveta tri
kralja",
    "Dan državnosti", "Velike Gospe", "Dan domovinske zahvalnosti"
]

# tražimo datume s tima keywordsima
non_working_days = []
# dohvatimo sve dane koji sadrže neki keyword, splitamo svaki sa /n
for line in text.split("\n"):
    if any(keyword in line for keyword in keywords):
        non_working_days.append(line.strip())
# onda ih izjoinamo
return "\n".join(non_working_days)

@app.route("/calendar")
def show_calendar():
    pdf_url =
"https://www.unipu.hr/_download/repository/Sveu%C4%8Dili%C5%A1ni%20kale
ndar%20za%202024._2025..pdf"
    calendar_text = extract_pdf_text(pdf_url)
    non_working_days = get_non_working_days(calendar_text)

    # Splitamo extractane dane
    non_working_days_list = non_working_days.split("\n")

    # izrenderiramo ih s bullet pointsima
    html_content = f"""
<html>
    <head>
        <title>Non-Working Days 2024/2025</title>
        <style>
            body {{
                font-family: Arial, sans-serif;
                margin: 40px;
                background-color: #f0f0f0;
                color: #333;
            }}
            h1 {{
                text-align: center;
                color: #0056b3;
            }}
        </style>
    </head>
    <body>
        <h1>Non-Working Days 2024/2025</h1>
        <ul style="list-style-type: none; padding-left: 0;">
{non_working_days_list}
        </ul>
    </body>
</html>
"""

    return Response(html_content, mimetype='text/html')

```

```

        ul {{
            background-color: #ffffff;
            padding: 20px;
            border-radius: 8px;
            border: 1px solid #ddd;
            line-height: 1.6;
            font-size: 14px;
            list-style-type: disc;
        }}
        ul li {{
            margin-bottom: 10px;
        }}
.container {{
    max-width: 900px;
    margin: 0 auto;
    padding: 20px;
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
    background-color: #ffffff;
    border-radius: 8px;
}}

```

</style>

</head>

<body>

```

<div class="container">
    <h1>Non-Working Days 2024/2025</h1>
    <ul>
        { ''.join(f"<li>{day}</li>" for day in
non_working_days_list if day.strip()) }
    </ul>
</div>

```

</body>

</html>

"""

```

return render_template_string(html_content)

```

Na samom kraju backenda definiran je osnovni Flask running setup. `if __name__ == "__main__"` označava da se ovdje radi o glavnom file-u aplikacije (u pomoćnim .py fileovima nema ovog dijela). `App.run` je funkcija za pokretanje aplikacije s parametrima `host`, `port`, `debug` i `use_reloader`. `Host="0.0.0.0"` definira da aplikaciju mogu pokretati sva računala koja su spojena na istu mrežu što je učinjeno s ciljem da se omogući prijava prisustva unutar cijele učionice na svim uređajima (uz pretpostavku da je cijela učionica na istom internetu npr. eduroam). `Port=5145` označava port na kojem se aplikacija vrti. Inače je Flaskov defaultni port 5000, no na mom računalu se paralelno vrti sve i svašta, pa sam namjerno uzeo neki malo veći random port. `Debug = True` omogućava prikaz informacija o greškama

na frontendu što značajno olakšava debugging. (Treba postaviti na False u produkciji). Use_reloader ponovno učitava aplikaciju nakon svake promjene, bez potrebe da se aplikacija pokreće i zaustavlja (nešto kao Nodemon). Ta je funkcionalnost ovdje isključena zato što se pri ponovnom pokretanju aplikacije treba čekati ~1 minutu dok se učitaju sva lica što je prilično iritantno prilikom razvoja.

```
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5145, debug=True, use_reloader =
False)
```

6. Sažeta specifikacija backenda

Specification for /set_subject Route

- **Supported Methods:** GET , POST
- **Functions Used:**
 - request.form : Retrieves form data (subject , date , start_time , end_time) from the POST request.
 - render_template() : Renders the HTML templates set_subject.html or set_subject_success.html .
- **Templates Rendered:**
 - set_subject.html : Rendered when the route is accessed via a GET request, presenting the form to the user.
 - set_subject_success.html : Rendered upon a successful POST request, confirming the subject and timings have been set.
- **Route Usage:** This route allows users to set the current subject for logging, including the date and time, and displays a success page upon submission.

Specification for /add_student Route

- Supported Methods: GET, POST
- Functions Used:
 - request.form : Retrieves the student's name from the form data.
 - request.files.getlist('images') : Retrieves the list of images uploaded by the user.
 - os.path.join() : Constructs file paths for saving images and creating directories.
 - os.makedirs() : Creates a new directory for the student within the known_faces folder, if it doesn't already exist.
 - image.save(image_path) : Saves each uploaded image to the designated folder.
 - add_known_face(image_path, name) : Processes the saved images for face recognition and associates them with the student's name.
 - render_template() : Renders the HTML templates add_student.html or add_student_success.html .
- Templates Rendered:
 - add_student.html : Rendered when the route is accessed via a GET request, presenting the form to add a new student.
 - add_student_success.html : Rendered upon a successful POST request, confirming the student has been added and their images have been saved.
- Route Usage: This route allows users to add a new student to the database, save their images for face recognition in a designated folder, and display a success page upon completion.

Specification for /add_student_success Route

- Supported Methods: GET
- Functions Used:
 - render_template() : Renders the HTML template add_student_success.html .
- Templates Rendered:
 - add_student_success.html : Displayed to confirm the successful addition of a student.
- Route Usage: This route renders a success page confirming that a new student has been successfully added to the database.

Specification for /video_feed Route

- **Supported Methods:** GET
- **Functions Used:**
 - generate_frames() : A function that captures video frames, processes them for face recognition, and adds bounding boxes and labels to detected faces.
 - Response() : Streams the video frames with the specified mimetype to be displayed in real-time on the client side.
- **Templates Rendered:** None (this route directly streams video data).
- **Route Usage:** This route streams video frames captured from the camera, with face recognition applied, displaying the frames with bounding boxes and labels for recognized faces.

Specification for / (Main) Route

- **Supported Methods:** GET
- **Functions Used:**
 - render_template() : Renders the HTML template index.html .
- **Templates Rendered:**
 - index.html : The main homepage of the application.
- **Route Usage:** This is the main route that renders the homepage of the application when accessed.

Specification for /attendance Route

- **Supported Methods:** GET
- **Functions Used:**
 - sqlite3.connect() : Connects to the SQLite database attendance.db .
 - c.execute() : Executes a SQL query to retrieve attendance records from the database, including the subject, student name, date, time, and whether the student was late.
 - c.fetchall() : Fetches all the resulting records from the executed query.
 - conn.close() : Closes the database connection after fetching the records.
 - render_template() : Renders the attendance.html template with the grouped attendance data.
- **Templates Rendered:**
 - attendance.html : Displays the attendance records grouped by date and subject.
- **Route Usage:** This route retrieves all attendance records from the database, organizes them by date and subject, and renders them in a structured format for easy viewing and analysis.

Specification for /download Route

- **Supported Methods:** GET

- **Functions Used:**
 - `sqlite3.connect()` : Connects to the SQLite database `attendance.db`.
 - `c.execute()` : Executes a SQL query to retrieve attendance records, sorted by subject, date, and time.
 - `c.fetchall()` : Fetches all the resulting records from the executed query.
 - `conn.close()` : Closes the database connection after fetching the records.
 - `io.StringIO()` : Creates an in-memory string buffer to hold the CSV data.
 - `csv.writer()` : Writes data to the CSV format in the string buffer.
 - `writer.writerow()` : Writes individual rows of data to the CSV file, including headers, subject sections, and attendance records.
 - `output.seek(0)` : Resets the buffer's position to the beginning before sending the response.
 - `Response()` : Sends the generated CSV file as a downloadable response with the appropriate MIME type (`text/csv`) and headers (`Content-Disposition`).
- **Templates Rendered:** None (this route directly generates and sends a CSV file).
- **Route Usage:** This route fetches all attendance records from the database, organizes them by subject, and exports them as a CSV file that users can download.

Specification for `/delete_attendance/<int:id>` Route

- **Supported Methods:** `POST`
- **Functions Used:**
 - `sqlite3.connect()` : Connects to the SQLite database `attendance.db`.
 - `c.execute()` : Executes a SQL `DELETE` command to remove the attendance record that matches the provided `rowid`.
 - `conn.commit()` : Commits the changes to the database, making the deletion permanent.
 - `conn.close()` : Closes the database connection after the deletion.
 - `redirect()` : Redirects the user to the `/attendance` page after the deletion is completed.
 - `url_for()` : Generates the URL for the `/attendance` route.
- **Templates Rendered:** None (this route performs a redirection after deletion).
- **Route Usage:** This route handles the deletion of a specific attendance record from the database using the record's `rowid`, and then redirects the user back to the attendance overview page to reflect the update.

Specification for /statistics Route

- **Supported Methods:** GET
- **Functions Used:**
 - `sqlite3.connect()` : Connects to the SQLite database `attendance.db`.
 - `c.execute()` : Executes SQL queries to retrieve attendance statistics:
 - The first query counts the number of attendance records per student for each subject.
 - The second query counts the total number of attendance records for each subject.
 - `c.fetchall()` : Fetches all the resulting records from the executed queries.
- `conn.close()` : Closes the database connection after retrieving the data.
- `render_template()` : Renders the `statistics.html` template with the retrieved attendance statistics data.
- **Templates Rendered:**
 - `statistics.html` : Displays the attendance statistics, including the count of records per student for each subject and the total count per subject.
- **Route Usage:** This route retrieves and displays attendance statistics from the database, showing the number of records grouped by student and subject, as well as overall subject attendance, presented in a structured format on the `statistics` page.

Specification for /students Route

- **Supported Methods:** GET
- **Functions Used:**
 - `sqlite3.connect()` : Connects to the SQLite database `attendance.db`.
 - `c.execute()` : Executes a SQL query to select distinct student names from the `attendance` table, ordered alphabetically.
 - `c.fetchall()` : Fetches all unique student names from the query result.
 - `conn.close()` : Closes the database connection after retrieving the data.
 - `render_template()` : Renders the `students.html` template, passing the list of student names.
- **Templates Rendered:**
 - `students.html` : Displays the sorted list of unique student names.
- **Route Usage:** This route retrieves a list of unique student names from the database, sorted alphabetically, and displays them on the `students.html` page.

Specification for /plot/student_attendance Route

- **Supported Methods:** GET
 - **Functions Used:**
 - `sqlite3.connect()` : Connects to the SQLite database `attendance.db` .
 - `c.execute()` : Executes a SQL query to retrieve the count of attendances per student from the database.
 - `c.fetchall()` : Fetches all the resulting records from the executed query.
 - `conn.close()` : Closes the database connection after fetching the records.
 - `plt.figure()` : Creates a new figure for the plot.
 - `sns.barplot()` : Creates a bar plot using Seaborn, displaying the count of attendances for each student.
 - `plt.title()`, `plt.xlabel()`, `plt.ylabel()` : Set the title and axis labels for the plot.
 - `io.BytesIO()` : Creates an in-memory byte stream to hold the plot image.
 - `plt.savefig()` : Saves the plot image to the byte stream.
 - `img.seek(0)` : Resets the byte stream's position to the beginning before sending the response.
 - `plt.close()` : Closes the plot to free up resources.
 - `send_file()` : Sends the plot image as a response with MIME type `image/png` .
 - **Templates Rendered:** None (this route directly returns a plot image).
 - **Route Usage:** This route generates and serves a bar plot showing the number of attendances per student.
-

Specification for /plot/subject_attendance Route

- **Supported Methods:** GET
- **Functions Used:**
 - `sqlite3.connect()` : Connects to the SQLite database `attendance.db` .
 - `c.execute()` : Executes a SQL query to retrieve the count of attendances per subject from the database.
 - `c.fetchall()` : Fetches all the resulting records from the executed query.
 - `conn.close()` : Closes the database connection after fetching the records.
 - `plt.figure()` : Creates a new figure for the plot.
 - `plt.pie()` : Creates a pie chart using Matplotlib, displaying the proportion of attendances for each subject.
 - `plt.title()` : Sets the title of the pie chart.
 - `io.BytesIO()` : Creates an in-memory byte stream to hold the plot image.
 - `plt.savefig()` : Saves the plot image to the byte stream.
 - `img.seek(0)` : Resets the byte stream's position to the beginning before sending the response.
 - `plt.close()` : Closes the plot to free up resources.
 - `send_file()` : Sends the plot image as a response with MIME type `image/png` .
- **Templates Rendered:** None (this route directly returns a plot image).
- **Route Usage:** This route generates and serves a pie chart showing the distribution of attendances across different subjects.

Specification for /plot/monthly_attendance Route

- **Supported Methods:** GET
- **Functions Used:**
 - `sqlite3.connect()` : Connects to the SQLite database `attendance.db`.
 - `c.execute()` : Executes a SQL query to retrieve the count of attendances per month from the database.
 - `c.fetchall()` : Fetches all the resulting records from the executed query.
 - `conn.close()` : Closes the database connection after fetching the records.
 - `plt.figure()` : Creates a new figure for the plot.
 - `plt.pie()` : Creates a pie chart using Matplotlib, displaying the proportion of attendances for each month.
 - `plt.title()` : Sets the title of the pie chart.
 - `io.BytesIO()` : Creates an in-memory byte stream to hold the plot image.
 - `plt.savefig()` : Saves the plot image to the byte stream.
 - `img.seek(0)` : Resets the byte stream's position to the beginning before sending the response.
 - `plt.close()` : Closes the plot to free up resources.
 - `send_file()` : Sends the plot image as a response with MIME type `image/png`.
- **Templates Rendered:** None (this route directly returns a plot image).
- **Route Usage:** This route generates and serves a pie chart showing the distribution of attendances across different months.

Specification for /plots Route

- Supported Methods: GET
 - Functions Used:
 - render_template(): Renders the HTML template plot_router.html .
 - Templates Rendered:
 - plot_router.html : Provides links to the different plot routes available.
 - Route Usage: This route renders a page with links to the various plot routes, allowing users to view different visualizations.
-

Specification for /predict_absence Route

Supported Methods: GET

Functions Used:

- get_weather_forecast(api_key, location="your_city"): Calls an external weather API to get the weather forecast for a specific location.
- predict_absence_due_to_weather(weather_condition): Determines if the weather condition is likely to cause absences based on predefined bad weather keywords.
- requests.get(): Sends a GET request to the weather API.
- jsonify(): Converts the response data to a JSON format for easy consumption by clients.

Templates Rendered: None

Route Usage: This route fetches the current weather forecast using a weather API and predicts if there will be possible late arrivals or absences due to bad weather. It returns a JSON response containing the weather condition and a message advising on potential attendance issues.

Specification for /announcements Route

Supported Methods: GET

Functions Used:

- sqlite3.connect(): Connects to the SQLite database attendance.db .
- c.execute(): Executes a SQL query to select all announcements ordered by date and time in ascending order.
- c.fetchall(): Fetches all rows from the announcements table as a list of tuples.
- conn.close(): Closes the database connection after retrieving the data.
- render_template(): Renders the announcements.html template, passing the list of announcements.

Templates Rendered:

- announcements.html: Displays all announcements in chronological order.

Route Usage: This route retrieves all announcements from the database, ordered by date and time, and displays them on the announcements.html page.

Specification for /post_announcement Route

Supported Methods: POST

Functions Used:

- `request.form['message']`: Retrieves the message content from the submitted form data.
- `datetime.now().strftime("%Y-%m-%d %H:%M:%S")`: Gets the current date and time, formatted as a string.
- `current_user.username`: Retrieves the username of the currently logged-in user.
- `sqlite3.connect()`: Connects to the SQLite database `attendance.db`.
- `c.execute()`: Executes a SQL query to insert a new announcement into the announcements table.
- `conn.commit()`: Commits the transaction to save the new announcement.
- `conn.close()`: Closes the database connection after saving the data.
- `redirect(url_for('announcements'))`: Redirects the user back to the announcements page after posting.

Templates Rendered: None

Route Usage: This route allows authenticated users to post a new announcement. The announcement is saved to the database with the current date, time, and the teacher's name, then the user is redirected to the `announcements` page to see the updated list.

Specification for /delete_announcement/<int:id> Route

Supported Methods: POST

Functions Used:

- `sqlite3.connect()`: Connects to the SQLite database `attendance.db`.
- `c.execute()`: Executes a SQL query to delete an announcement by its ID.
- `conn.commit()`: Commits the transaction to remove the announcement from the database.
- `conn.close()`: Closes the database connection after deleting the data.
- `redirect(url_for('announcements'))`: Redirects the user back to the announcements page after deletion.

Templates Rendered: None

Route Usage: This route allows authenticated users to delete an announcement by its ID. After deletion, the user is redirected to the `announcements` page.

Specification for /edit_announcement/<int:id> Route

Supported Methods: GET, POST

Functions Used:

- `sqlite3.connect()`: Connects to the SQLite database `attendance.db`.
- `request.method == 'POST'`: Checks if the request method is POST, indicating that the form was submitted.
- `request.form['message']`: Retrieves the updated message content from the submitted form data.
- `datetime.now().strftime("%Y-%m-%d %H:%M:%S")`: Gets the current date and time, formatted as a string.
- `current_user.username`: Retrieves the username of the currently logged-in user.

- `c.execute()`: Executes a SQL query to update the announcement if the user is the owner.
- `conn.commit()`: Commits the transaction to save the updated announcement.
- `conn.close()`: Closes the database connection after saving the data.
- `redirect(url_for('announcements'))`: Redirects the user back to the announcements page after editing.
- `render_template('edit_announcement.html', announcement=announcement)`: Renders the `edit_announcement.html` template, passing the current announcement data for editing.

Templates Rendered:

- `edit_announcement.html`: Displays the form for editing the announcement's message.

Route Usage: This route allows authenticated users to edit an announcement they posted. It first checks if the user owns the announcement, then updates it. If the method is GET, the current announcement is displayed in an edit form. If POST, the changes are saved and the user is redirected to the announcements page.

Specification for /report Route

Supported Methods: GET

Functions Used:

- `sqlite3.connect()`: Connects to the SQLite database `attendance.db`.
- `cur.execute()`: Executes a SQL query to select distinct subjects and count lectures and student attendance.
- `cur.fetchall()`: Fetches all results of the executed queries.
- `conn.close()`: Closes the database connection after retrieving the data.
- `render_template('attendance_report.html', report=report)`: Renders the `attendance_report.html` template, passing the report data.

Templates Rendered:

- `attendance_report.html`: Displays the attendance report for each subject, including student attendance percentages and a status indicating if they meet the attendance requirement.

Route Usage: This route generates a detailed attendance report by subject. It counts the total lectures for each subject and the number of lectures each student attended, calculates attendance percentages, and checks if the student meets the 50% attendance requirement. The report is displayed on the `attendance_report.html` page.

Specification for /late_analysis Route

- **Supported Methods:** GET , POST

- **Functions Used:**

- `sqlite3.connect('attendance.db')` : Connects to the SQLite database `attendance.db`.
- `c.execute()` : Executes SQL queries to fetch all entries marked as "late" from the `attendance` table.
- `c.fetchall()` : Fetches all rows resulting from the executed query as a list of tuples.

- `Counter()` : Utilizes Python's `collections.Counter` to count the frequency of late entries by hour and weekday.
- `datetime.strptime()` : Converts string representations of date and time into `datetime` objects for further analysis.
- `plt.bar()` : Creates a bar chart using Matplotlib to visualize the distribution of late entries by hour.
- `plt.xticks()` : Ensures all hours from 0 to 23 are labeled on the x-axis of the plot.
- `plt.xlabel()`, `plt.ylabel()`, `plt.title()` : Sets the labels and title for the plot.
- `io.BytesIO()` : Creates an in-memory binary stream to store the generated plot image.
- `plt.savefig()` : Saves the generated plot as a PNG image in the binary stream.
- `base64.b64encode()` : Encodes the binary image data as a base64 string for easy embedding in HTML.
- `render_template()` : Renders the `late_entries.html` template, passing relevant data for display.
- **Templates Rendered:**
 - `late_entries.html` : Displays the list of late entries, the most common hour and weekday for late arrivals, and a bar chart visualizing the frequency of late entries by hour.
- **Route Usage:**
 - **Data Collection:**
 - Retrieves all entries from the `attendance` database where the `late` flag is set to 1.
 - Analyzes the `time` and `date` fields to determine the most common hour and weekday for late arrivals.
 - **Data Visualization:**
 - Creates a bar chart showing the distribution of late entries by hour of the day.
 - Encodes the plot image as a base64 string for embedding directly into the rendered HTML.
 - **Rendering:**
 - Renders the `late_entries.html` template, displaying:
 - The list of late entries.
 - The most common hour and weekday for late arrivals.
 - The bar chart of late entries by hour.
 - **Edge Cases:**
 - If no late entries are found, the route handles the absence of data gracefully, ensuring the page still loads without errors.

Specification for `/download_and_email` Route

Supported Methods:

- GET

Functions Used:

- `sqlite3.connect()` : Establishes a connection to the `attendance.db` SQLite database.
- `c.execute()` : Executes a SQL query to retrieve attendance records, ordered by subject, date, and time.
- `c.fetchall()` : Fetches all rows from the executed SQL query.
- `io.StringIO()` : Creates an in-memory string buffer to store CSV data.
- `csv.writer()` : Writes CSV data to the string buffer.
- `request.args.get('email')` : Retrieves the user's email address from the query parameters.
- `render_template()` : Renders the HTML template `download.html` with the generated CSV data and user's email.

Templates Rendered:

- `download.html` : Renders a page that allows the user to download the generated CSV file and includes a button to open Gmail with the user's email pre-filled as the sender.

Route Usage:

- This route allows users to download attendance data in CSV format and provides an option to open Gmail in a new tab with the user's email pre-filled. The attendance data is grouped by subject and ordered by date and time. The user's email is passed through the query parameters or session, and the generated CSV data is embedded in the HTML template for download.

Functions Used:

- `request.args.get('email')` : Retrieves the email address passed as a query parameter in the GET request.
- `EmailMessage()` : Creates a new email message object.
- `msg.set_content()` : Sets the plain text content of the email.
- `msg.add_attachment()` : Attaches the generated CSV file to the email.
- `smtplib.SMTP()` : Connects to an SMTP server for sending the email.
- `smtplib.SMTP_SSL()` : Optionally, connects to the SMTP server using SSL encryption.
- `smtp.send_message()` : Sends the email message.
- `smtp.quit()` : Closes the connection to the SMTP server.
- `Response()` : Sends a JSON response back to the client indicating the success or failure of the email operation.

Templates Rendered:

- None (this route directly generates and sends a CSV file and an email).

Route Usage:

- **Data Generation:**
 - Retrieves attendance records from the database and writes them into a CSV format.
 - Stores the CSV data in an in-memory string buffer.
- **Email Preparation:**
 - Prepares an email with the attendance CSV file attached.
 - Uses the email address provided as a query parameter to send the email.
- **Response Handling:**
 - Sends a JSON response back to the client, indicating whether the email was sent successfully or if there was an error.
- **Edge Cases:**
 - Handles cases where no email is provided or if the email sending fails, ensuring appropriate error messages are returned in the JSON response.

- Specification for `/scrape-github-profile` Route

Route Overview:

- **Route Name:** `/scrape-github-profile`
- **HTTP Method:** `GET`
- **Function Name:** `scrape_github_profile_route()`
- **Description:** This route is responsible for scraping profile information from a specific GitHub profile page and rendering that data on a predefined HTML template or returning an error message if scraping fails.

Specifications:

1. **Request Method:** `GET`

- This route accepts HTTP GET requests and does not require any query parameters or payload.

2. **GitHub Profile URL:**

- The GitHub profile being scraped is hardcoded as `https://github.com/AntonioLabinjan`.
- In future extensions, the URL could be passed as a parameter to make the route more dynamic.

3. **Scraping Logic:**

- The `scrape_github_profile(url)` function is called to scrape the GitHub profile information.
- The function should return a dictionary with the following keys:
 - `name` : GitHub user's full name.
 - `bio` : GitHub user's bio.
 - `followers` : GitHub user's follower count.
- If the scraping operation fails (returns `None` or an empty dictionary), a JSON response with an error message will be returned.

4. **Response - Successful Scraping:**

- If the scraping is successful, the following information is passed to the HTML template (`profile.html`):
 - `name` : The GitHub user's full name.
 - `bio` : The GitHub user's bio.
 - `followers` : The GitHub user's follower count.
- The HTML template will render the scraped data for display.

5. **Response - Failed Scraping:**

- If scraping fails, the route will return a JSON response with an error message and a `500` HTTP status code:

```
{  
    "error": "Failed to scrape the data"  
}
```

6. HTML Template (profile.html):

- The HTML template will dynamically display the scraped data using the following variables:
 - {{ name }} : GitHub user's name.
 - {{ bio }} : GitHub user's bio.
 - {{ followers }} : Number of followers.

Example Response Flow:

1. Successful Scraping:

- User sends a GET request to /scrape-github-profile .
- The server successfully scrapes the GitHub profile and renders the following in profile.html :

```
<h1>{{ name }}</h1>  
<p>{{ bio }}</p>  
<p>Followers: {{ followers }}</p>
```

2. Failed Scraping:

- User sends a GET request to /scrape-github-profile .
- The server fails to scrape the profile and returns a JSON error:

```
{  
    "error": "Failed to scrape the data"  
}
```

- Response HTTP status code: 500 .

Route: /calendar

HTTP Method: GET

Description:

- This route fetches a PDF from a specified URL (University calendar), extracts its text, filters non-working days based on specific keywords, and then displays these days in an HTML template. The days are displayed as an unordered list in a styled web page.

Input:

- URL: /calendar
- No user input required.

Processes:

1. PDF Fetching:

- The route retrieves the PDF from the given URL using `requests.get()` with SSL verification disabled (`verify=False`).
- The PDF is saved locally (`calendar.pdf`) for temporary extraction.

2. PDF Text Extraction:

- Using `pdfplumber`, the PDF text is extracted page by page. The entire content of the PDF is stored as a single string.

3. Keyword Filtering:

- The extracted text is scanned for specific non-working day keywords (such as public holidays and university off-days).
- Lines containing any of these keywords are filtered and stored as the list of non-working days.

4. HTML Rendering:

- The filtered non-working days are passed into an HTML template and displayed as an unordered list (``).
- The HTML template is inline with simple styling for better visual presentation.

Output:

- A rendered HTML page listing the non-working days for the 2024/2025 academic year in a bullet-point format.
-

Functions Used:

1. `extract_pdf_text(pdf_url)`

- **Description:** Downloads the PDF, saves it locally, and extracts its textual content using `pdfplumber`.
- **Input:**
 - `pdf_url`: The URL of the PDF file to be fetched.
- **Output:** The extracted text as a single string.

2. `get_non_working_days(text)`

- **Description:** Filters the extracted text and identifies lines containing keywords that signify non-working days.
 - **Input:**
 - `text`: The full text extracted from the PDF.
 - **Output:** A string containing non-working day descriptions, separated by newlines.
-

Key Elements in the HTML Output:

- **Page Title:** "Non-Working Days 2024/2025"
 - **Styled Container:** A center-aligned container with box-shadow and padding for the list of non-working days.
 - **Unordered List:** The list of non-working days is displayed as `` elements under an unordered list (``).
-

Example Output:

```
GET /calendar
A webpage with a header "Non-Working Days 2024/2025" and a list like:
- Božić (25.12.2024)
- Nova Godina (01.01.2025)
- Dan državnosti (25.06.2025)
- Uskrs (06.04.2025)
```

7. Popis tehnologija

Pri izradi aplikacije korištene su sljedeće tehnologije:

- Python + Flask za backend
- HTML + Jinja templating engine + JS za frontend
- SQLITE baza podataka
- ChatGPT
- Openai CLIP model
- FAISS
- SendGrid api
- Open CV HaarCascade
- WeatherAPI
- Distilroberta-base-offensive-hateful-speech-text-multiclassification model



8. Upute za korištenje

Ova je aplikacija poprilično jednostavna za korištenje i za rad s njom nije potrebno manje-više nikakvo posebno znanje osim osnova rada s računalom. No, korištenje se ipak može rastaviti u nekoliko osnovnih neophodnih koraka i preduvjeta.

1. Za rad je potrebno imati funkcionalnu i stabilnu internetsku vezu
2. Novi korisnici koji žele imati "admin" ovlasti, odnosno imati ulogu "profesora" trebaju kreirati račun koristeći validnu email adresu
3. Za dodavanje novih poznatih lica u aplikaciju, potrebno je, uz odobrenje osobe, dodati bar 1 sliku u sustav učitavanjem iz lokalne pohrane ili putem live dodavanja
4. Slike ne smiju (ili bar ne bi smjele) izlaziti izvan sustava radi zaštite privatnosti
5. Opasno je otvarati sekciju za vizualizaciju podataka ukoliko nema podataka (don't do that)
6. Pri prvom otvaranju aplikacije u browseru je potrebno omogućiti pristup kamери (u suprotnom se ne može skenirati lice)
7. Nastavno na prethodnu točku, potrebno je imati kamерu na uređaju na kojem se aplikacija koristi
8. Iako je moguće istovremeno prepoznati više ljudi, idealna uporaba predviđa prepoznavanje studenata one-by-one (to je idealni use-case koji minimizira mogućnost grešaka)
9. U funkciji send_attendance_notification moguće je dodati proizvoljni mail za slanje obavijesti, kao i više proizvoljnih mailova za zaprimanje obavijesti
10. Ako aplikaciji treba određeno vrijeme da se učita pri prvom pokretanju, to je zato jer se učitavaju poznata lica. Please be patient; taj se proces nažalost ne može ubrzati

Ako su svi preduvjeti ispunjeni, onda je samo potrebno postaviti uređaj na adekvatno mjesto kako bi svaka osoba koja uđe mogla pogledati u kamenu i tako zabilježiti svoj dolazak.