Mobilna aplikacija za anotaciju slika

Dragić, Domagoj

Master's thesis / Diplomski rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:329075

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-05-15**

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni studij

MOBILNA APLIKACIJA ZA ANOTACIJU SLIKA

Diplomski rad

Domagoj Dragić

Osijek, 2023.

SADRŽAJ

1. UVOD ... 1
1.1. Zadatak diplomskog rada ... 2
2. PREGLED PODRUČJA TEME RADA ... 3
2.1. Segmentacija slika u svrhu strojnog učenja ... 3
2.2. Konvolucijske neuronske mreže ... 5
2.2.1. U-Net arhitektura dubokog učenja ... 6
2.3. Anotiranje slika za potrebe strojnog učenja ... 7
2.4. Postojeća programska rješenja ... 9
2.4.1. Annotate mobilna aplikacija ... 9
2.4.2. iMarkup mobilna aplikacija ... 10
2.4.3. V7 web alat za anotiranje ... 11
2.4.4. LabelMe web alat za anotiranje ... 12
3. TEHNOLOGIJE I ALATI KORIŠTENI U IZRADI RADA ... 13
3.1. Android operacijski sustav ... 13
3.2. Android Studio razvojno okruženje ... 15
3.3. Kotlin programski jezik ... 17
3.4. GitHub servis za verzioniranje koda ... 18
4. ARHITEKTURA PROGRAMSKOG RJEŠENJA ... 20
4.1. Funkcionalni zahtjevi na aplikaciju ... 20
4.2. Dijagram tijeka aplikacije ... 21
4.3. Firebase razvojni alati ... 22
4.3.1. Firebase usluga za upravljanje korisnicima ... 23
4.3.2. Firebase usluga spremnika u oblaku ... 24
4.3.3. Firebase usluga baze podataka ... 25
4.4. MVVM arhitektura mobilne aplikacije ... 26
5. IMPLEMENTACIJA PROGRAMSKOG RJEŠENJA ... 29
5.1. Upravljanje korisnicima u sustavu ... 29
5.2. Generiranje skupa slika za anotiranje ... 31
5.3. Postavljanje trenutne slike za anotiranje ... 33
5.4. Alati za anotiranje slika ... 36
5.5. Dijalog za dodatne informacije o slici ... 40
5.6. Spremanje rezultata anotiranja u bazu podataka ... 42
6. ZAKLJUČAK ... 44
LITERATURA ... 46
SAŽETAK ... 50
ABSTRACT ... 51

ŽIVOTOPIS 52

1. UVOD

Požar uglavnom podsjeća na nekakvu nesreću i veliku ekološku i ekonomsku štetu. Često je požarom ugrožen i ljudski život [1]. Upravo iz tog razloga provode se brojna istraživanja koja u prvi plan stavljaju rano otkrivanje požara te prevenciju istog. Neke od poznatijih metoda prepoznavanja požara temelje se na očitavanju temperature, vlažnosti ili prozirnosti zraka. Analize navedenih pojava najčešće obrađuje detektor koji promatra parametre temperature, vlažnosti ili prozirnosti zraka te, ako neki od njih prekorači gornju granicu normalne vrijednosti, okida alarm koji signalizira opasnost od mogućeg požara. Jedan od važnijih problema ovih detektora je to što oni zapravo ne pružaju neke dodatne informacije o požaru, kao što je primjerice mjesto ili veličina požara. Također, budući da detektori rade na principu otkrivanja abnormalnosti u parametrima koje promatraju, možemo reći kako nisu uvijek pouzdani jer se povišene temperature ili dim mogu proizvesti na razne načine koji ne dovode nužno do požara. Uzmimo za primjer kuhanje na povišenoj temperaturi. U nekim slučajevima detektor će taj dim prepoznati kao opasan i pretpostaviti kako se radi o potencijalnom požaru te će okinuti alarm. Kako bi se smanjio broj lažnih alarma i općenito poboljšao opseg informacija koji se dobiva o požaru, sve zanimljiviji postaje vizualni pristup detekcije požara. Ovaj rad usredotočit će se upravo na taj vizualni pristup otkrivanja požara te metode koje su potrebne za implementiranje jednog takvog sustava. U praktičnom dijelu rada bit će napravljena Android mobilna aplikacija za anotaciju vatre i dima na slikama koje će kasnije služiti za treniranje modela za strojno učenje kako bi uspješno prepoznavao požar u stvarnim situacijama.

Tijekom posljednjih nekoliko desetljeća strojno učenje je postalo jedno od glavnih grana proučavanja u informacijskim tehnologijama. Već danas možemo vidjeti ogroman utjecaj strojnog učenja u raznim područjima od prepoznavanja slika do zdravstvene zaštite i financijskih tržišta. Strojno učenje najlakše je opisati kao postupak kojim računala uče raditi stvari koje su ljudima prirodna i normalna. Strojno učenje može se klasificirati u 2 segmenta: nadzirano i ne nadzirano učenje. Nadzirano učenje možemo objasniti kao učenje iz primjera. Računalo radi s 2 skupa podataka, onima za učenje i onima za testiranje. Nadzirano učenje primjenu je pronašlo u područjima kao što su bioinformatika, prepoznavanje rukopisa te prepoznavanje objekata u računalnom vidu, što je upravo i glavni dio ovoga rada. Ne nadzirano učenje je puno subjektivnije od nadziranog učenja zbog činjenice da nema jednostavnog cilja analize kao što je primjerice predviđanje odgovora. Ne nadzirano učenje se općenito povezuje s idejom korištenja zbirke opažanja. U praktičnom dijelu ovog rada cilj je omogućiti korisniku anotaciju slike tako da odvoji

vatru i dim od ostalih segmenata slike. Tako anotirane slike služit će kao skup podataka za treniranje modela strojnog učenja.

1.1. Zadatak diplomskog rada

U teorijskom dijelu diplomskog rada potrebno je opisati načine segmentacije slika za potrebe treniranje modela strojnog učenja. Nadalje, potrebno je opisati i proces anotiranja slika, slične aplikacije, MVVM arhitekturu koja će se koristiti prilikom izrade mobilne aplikacije te *Firebase* bazu podataka koja će se koristiti za spremanje slika i anotacija. U praktičnom dijelu rada potrebno je korištenjem predložene arhitekture i baze podataka razviti Android mobilnu aplikaciju za anotiranje slika za potrebe učenja modela strojnog učenja.

2. PREGLED PODRUČJA TEME RADA

U ovom poglavlju bit će prikazano trenutno stanje u usko vezanim područjima spomenutim u ovom diplomskom radu. Prvo će biti objašnjena segmentacija i metode segmentacije slike za potrebe treniranja modela strojnog učenja. Zatim će biti opisano strojno učenje korišteno za prepoznavanje vatre i dima na temelju podataka dobivenih anotacijom slika i treniranjem spomenutog modela. Za kraj će biti prikazana postojeća dva mobilna i dva web rješenja uz detaljan opis prednosti i mana istih.

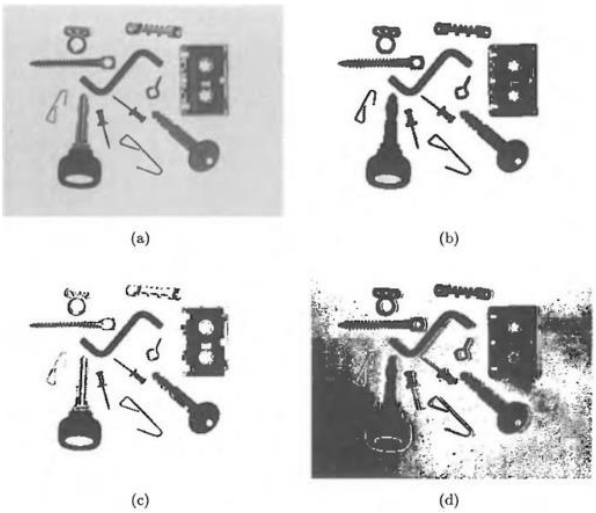
2.1. Segmentacija slika u svrhu strojnog učenja

Za segmentaciju slike možemo reći kako je to proces dijeljenja slike u logički povezana područja ili dijelove temeljene na njezinom sadržaju. Rezultat segmentacije slike su jedinstvena područja koja su predstavljena zasebnim objektima ili strukturama na slici. Jednostavnije rečeno, cilj segmentacije slike je prepoznati i grupirati piksele koji su dio istog objekta na slici i tako ih odvojiti od drugih objekata na slici. Segmentacija slike pronalazi široku upotrebu u mnogim područjima računalnog vida: prepoznavanje i praćenje objekata, medicinsko snimanje, video nadzoru itd.

Za segmentiranje slika mogu se koristiti razne tehnike i metode kao što su:

- Metoda praga – pikseli objekata se izdvajaju od pozadine na temelju definiranog praga boja
- Metoda detekcije rubova – radi na temelju otkrivanja rubova, odnosno granica, između područja kojima se odvaja objekt od okoline
- Metoda detekcije regija – radi na temelju prepoznavanja regija sličnih boja, tekstura ili drugih karakteristika
- Metoda grupiranja – algoritam grupacije kojim se formiraju grupe piksela na temelju zajedničkih značajki

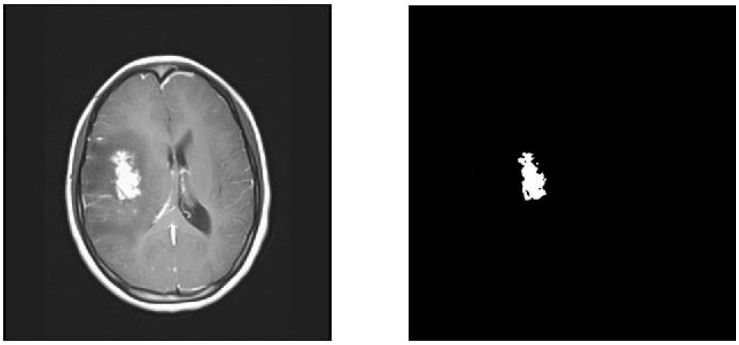
Na slici 2.1. prikazan je postupak segmentacija slike metodom praga i različiti rezultati iste. A) slika prikazuje originalnu sliku raznih kućnih predmeta, b) slika prikazuje izgled slike kada na nju primijenimo metodu praga s normalnim parametrima praga. C) slika prikazuje istu metodu prilikom podešavanja praga prenisko dok d) slika prikazuje previsoki prag.



Sl. 2.1. Prikaz segmentacije slike metodom praga [2]

Izbor konkretne metode segmentacije ovisi o njezinoj primjeni, željenoj točnosti i učinkovitosti. U nastavku će biti prikazani praktični primjeri svake od navedenih metoda segmentacije.

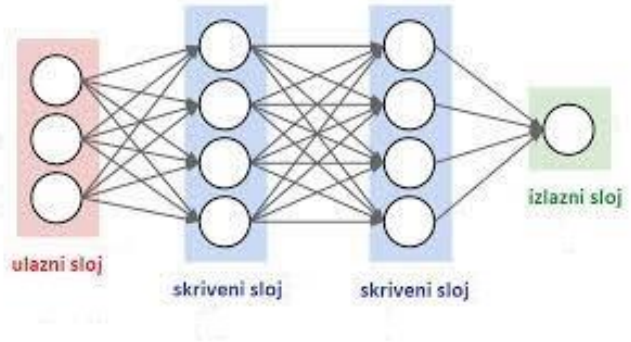
Ako je potrebno obraditi neki dokument tako da se izvuče tekst iz pozadine koristili bi metodu praga na način da postavimo ispravan intenzitet praga. Shodno tome, tekst se segmentira te je moguće vršiti daljnju obradu nad njim. Metoda detekcije rubova često se koristi u sustavima autonomne vožnje. Cilj je detektirati rubove na slici te tako segmentirati objekte kao što su vozila, prometni znakovi, rubovi ceste i pješaci u prometu. Kontinuiranim segmentiranjem navedenih objekata omogućuje se olakšano praćenje objekata te se tako pridonosi sigurnijoj autonomnoj vožnji. Metoda detekcije regija najčešće je korištena u medicini, konkretno u radiologiji i onkologiji, za detekciju tumora. Nakon segmentacije slike u regije moguće je vrlo jednostavno primijetiti abnormalnosti na organima zbog različitih intenziteta ili tekstura regija. Na ovaj način moguće je vrlo brzo odrediti točnu dijagnozu za pacijenta. Slika 2.2. a) prikazuje tumor na mozgu uslikan magnetskom rezonancom dok je na b) slici taj tumor segmentiran metodom regije. Metoda grupiranja koristi se pri analiziranju satelitskih snimaka kako bi se segmentirali različite vegetacije na zemlji. Ako grupiramo piksele na temelju njihovih spektralnih vrijednosti lako možemo grupirati objekte prema njihovim karakteristikama. Takvim postupkom segmentiramo šume, poljoprivredno zemljište, vodene površine itd.



Sl. 2.2. Tumor mozga uslikan magnetskom rezonancom [3]

2.2. Konvolucijske neuronske mreže

Prema [4], konvolucijske neuronske mreže su jedna od najkorištenijih i najučinkovitijih metoda dubokog učenja za detekciju požara. Konvolucijske neuronske mreže možemo zamisliti kao nadograđene obične neuronske mreže budući da se i one sastoje od jednog ulaznog, jednog ili više skrivenih, te jednog izlaznog sloja. Specifičnost kod konvolucijskih neuronskih mreža su upravo konvolucijski slojevi koji ih i razlikuju od običnih neuronskih mreža. Na slici 2.3. vidljiva je struktura konvolucijskih neuronskih mreža s njezinim slojevima.



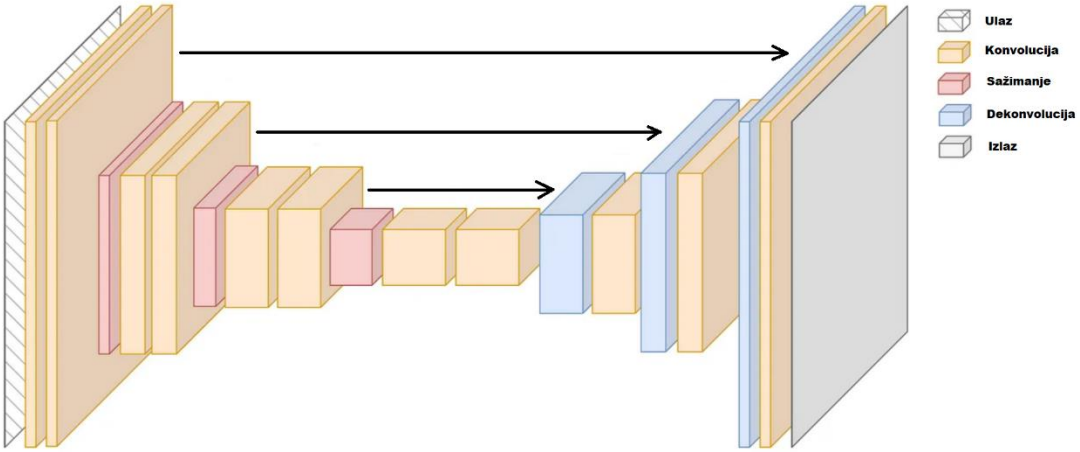
Sl. 2.3. Prikaz konvolucijske neuronske mreže [5]

U konvolucijskom sloju dolazi do podešavanja slike s ulaznog sloja. Ulazna slika se konvoluira s određenim parametrom jezgre i kao takva izlazi iz prvog skrivenog sloja te ulazi dalje u drugi. U svakom narednom sloju izlučuju se novi oblici sve dok se ne obrade sve do kraja. Konvolucijsku jezgru možemo zamisliti kao svojevrstan filter kroz koji slika prolazi više puta sve dok se ne postigne željeni rezultat. Bitno je napomenuti da se parametar konvolucijske jezgre konstantno mijenja sve dok se ne postigne optimalna vrijednost, a to se postiže procesom učenja koji i jest jedna od glavnih značajki neuronskih mreža. Još jedna razlika konvolucijskih mreža u odnosu na obične neuronske mreže jesu i slojevi sažimanja, koji dolaze nakon konvolucijskih slojeva. U njima se odvijaju operacije sažimanja, odnosno dio mape značajki se postavlja na maksimalnu

vrijednost. Na ovaj način se postiže veća otpornost na rotacije ili translacije budući da, nakon sažimanja, čak i da promatramo blago rotirani objekt, mreža će ga uspjeti prepoznati jer na izlazu imamo jednaku vrijednost. Prema [6], ovakva arhitektura konvolucijskih mreža istaknula se kao jedna od najboljih u obradi slike i prepoznavanju objekata sa slike. Glavni problem konvolucijskih neuronskih mreža je potreba za velikim setom podataka za treniranje modela, ovaj problem riješen je uvođenjem U-Net arhitekture koja će biti opisana u nastavku.

2.2.1. U-Net arhitektura dubokog učenja

U-Net [7] je izrazito popularna arhitektura korištena za treniranje modela dubokog učenja segmentacijom slika. Glavna prednost nad ostalim modelima konvolucijskih neuronskih mreža je brzina učenja U-Net modela koji treba znatno manji broj testnih podataka kako bi bio učinkovit. Ključni element U-Net arhitekture je preskakanje veza [8] (engl. *Skip connections*) te upravo zbog toga ova arhitektura ima bolje performanse u odnosu na ostale. Osim preskakanja veza bitno je shvatiti cijelu U-Net arhitekturu koja se još sastoji i od: kodera, dekodera i završnog sloja. Koder je komponenta koja je dio svih konvolucijskih arhitektura i služi za razdvajanje značajki na slici. To je prvi sloj kroz koji ulazna slika prolazi i njegova zadaća je stvoriti kompaktni prikaz te ulazne slike. U konvolucijskom sloju se odvija mapiranje slike kroz filter jezgre dok se u sloju sažimanja smanjuje dimenzija slike. Kombinacijom više konvolucijskih slojeva i slojeva sažimanja izvlačimo detalje iz slike, od najnižih oblika kao što su rubovi i boje do onih najviših kao što su konkretni detalji slike. Dekoder je komponenta koja rekonstruira sliku. Prije ulaska u dekoder slika je smanjenih dimenzija zbog prolaska kroz prijašnje slojeve, dekoder ima zadaću vratiti prvobitnu dimenziju slike uz zadržane značajke reducirane slike. Na ovaj način dekoder može prenijeti važne značajke koderu, ali i dalje ostaje problem lokacije tih značajki. Kada bi arhitektura ostala na ovome i dalje bi nam trebao ogroman broj testnih podataka kako bismo istrenirali model da točno rekonstruira slike iz reduciranih prikaza. Primjenom elementa preskakanja veza moguće je znatno smanjiti ovaj zahtjev. Preskakanje veza koristi se na ranijim slojevima ove arhitekture za prijenos informacija o položaju značajki na dekonvolucijske slojeve i samim time govori mreži odakle na slici dolaze te značajke. Kombiniranjem svih ovih slojeva na kraju dolazimo do boljih performansi i potrebom za znatnom manjim skupom testnih podataka. Vizualni prikaz U-Net arhitekture može se vidjeti na slici 2.4.



Sl. 2.4. U-Net arhitektura [8]

2.3. Anotiranje slika za potrebe strojnog učenja

Za potrebe treniranja modela strojnog učenja ključno je imati dobar skup testnih podataka. U slučaju ovog rada, to će biti razne slike vatre, požara i dima u zatvorenom ili otvorenom prostoru. Bitno je znati da će upravo o tim slikama i njihovim označavanjima ovisiti uspjeh prepoznavanja modela za strojno učenje. Prema [9], svaka slika testnog skupa mora biti točno anotirana kako bi se model strojnog učenja ispravno podučio prepoznavanju objekata na slici. Također, potrebno je pripaziti na samu kvalitetu slika korištenih za treniranje. Što je bolja kvaliteta slike i mogućnost izdvajanja objekata s nje, to će modelu biti lakše naučiti osnovna pravila raspoznavanja objekata. Prema [10], postoje 4 vrste anotiranja slike koje će biti objašnjene na primjeru slike 2.5. koja prikazuje vizualnu reprezentaciju ovih vrsta anotacije:

- Klasifikacija – otkriva nam samo postoji li objekt, u ovom slučaju banana, na slici ili ne
- Detekcija objekta – otkriva prisutnost, lokaciju i broj objekata na slici, odnosno govori nam da u ovom slučaju postoje 4 banane na slici
- Semantička segmentacija – otkriva prisutnost i lokaciju objekta, a nekada i veličinu i oblik samog objekta, na primjeru sa slike ovom segmentacijom moguće je primijetiti da postoje banane unutar označenih piksela
- Segmentacija instance – otkriva sve podatke o objektu na slici i ovo je najdetaljnija anotacija, iz primjera se slike možemo zaključiti da na slici postoje 4 banane te kakvog su oblika i veličine

Sl. 2.5. Vrste anotacija slike [10]

U svrhu ovog rada, bit će korišteno 50 slika koje će predstavljati testni skup podataka koji je potrebno anotirati za treniranje modela. Budući da je potrebno imati što više različitih slučajeva, kako bi se poboljšala preciznost modela, izabrane slike bit će podijeljene u 5 kategorija: slike na kojima prevladava vatra, slike na kojima je prisutna kombinacija vatre i dima, slike na kojima prevladava dim, slike vatre u zatvorenom prostoru te slike vatre u otvorenom prostoru. U stvarnosti, potreban je znatno veći broj testnih podataka za ispravno treniranje modela, no ovo je dovoljno za potrebe praktičnog rada, koji se usredotočuje na proces anotiranja slika. Također, ako i dođe do potrebe za većim skupom testnih podataka, vrlo lako bi ih se samo dodalo u postojeću bazu podataka. Anotiranje slike općenito možemo zamisliti kao proces označavanja granica nekog objekta te pridodavanja atributa uz taj objekt na slici. Tako testni podaci dobivaju novu dimenziju u vidu meta-podataka koji su najbitniji dio u procesu treniranja modela. Ova metoda pripada strojnom učenju pod nadzorom te je vrlo bitno shvatiti ozbiljnost procesa anotiranja i kontrolu anotiranih slika kako bismo dobili što bolji testni skup. Na slici 2.6. vidljiv je prikaz dobro anotiranog objekta i njegovih značajki.

Sl. 2.6. Anotacija objekata na slici

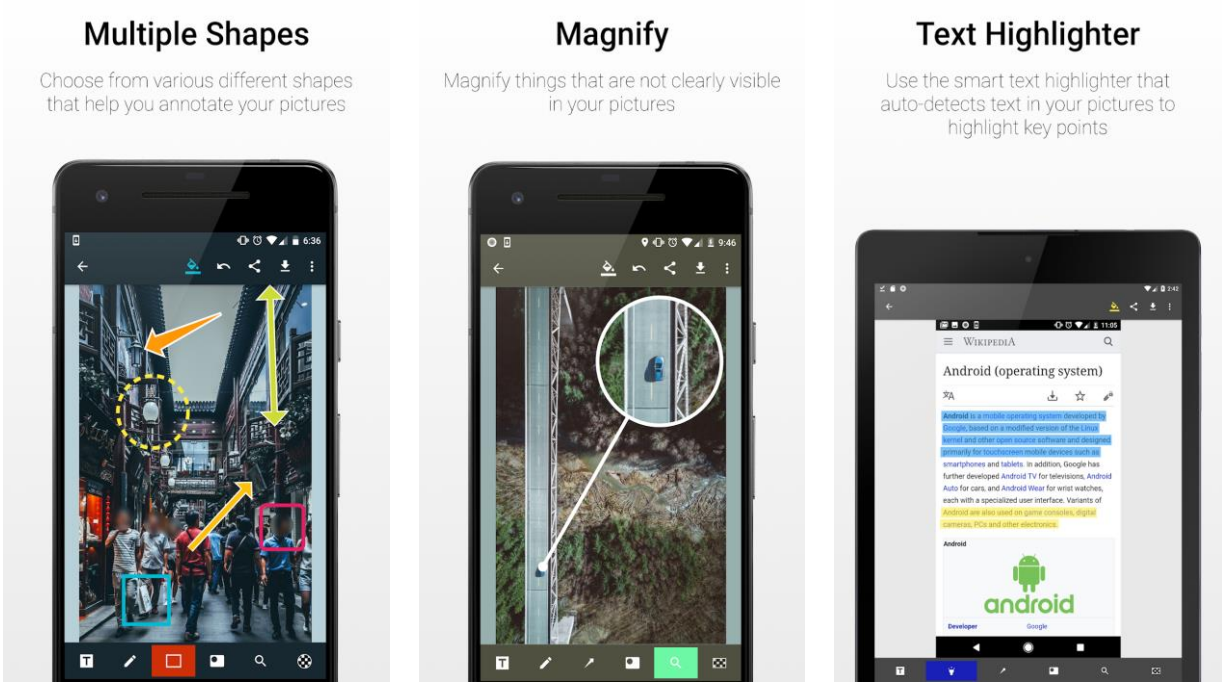
2.4. Postojeća programska rješenja

2.4.1. Annotate mobilna aplikacija

Annotate [11] je mobilna Android aplikacija koja korisnicima omogućuje brzu anotaciju slika s vlastitog mobilnog uređaja bez potrebe za autorizacijom ili posebnim dopuštenjima. Aplikaciju su razvili programeri iz tvrtke *RHS Apps*. Aplikacija je objavljena 8. srpnja 2018. godine, a posljednje ažurirana 21. kolovoza 2021. godine. U svojoj kategoriji anotacije slika je jedna od prvih izbora što dokazuje više od 50 tisuća preuzimanja. Na *Google Play store*-u ima ocjenu 3.3/5. Glavne značajke ove aplikaciju su:

- Odabir slike s mobilnog uređaja i brzi početak anotacije
- Izbor između više različitih oblika za anotiranje (pravokutnik, krug, nepravilni oblik, itd.)
- Obrezivanje, zumiranje ili zakretanje odabrane slike
- Izbor između više različitih boja
- Mogućnost brzog dijeljenja anotirane slike putem društvenih mreža
- Podrška za engleski, japanski i korejski jezik

Na slikama 2.7. a), b) i c) prikazana je mobilna aplikacija Annotate i njezine glave mogućnosti



a) **b)** **c)**

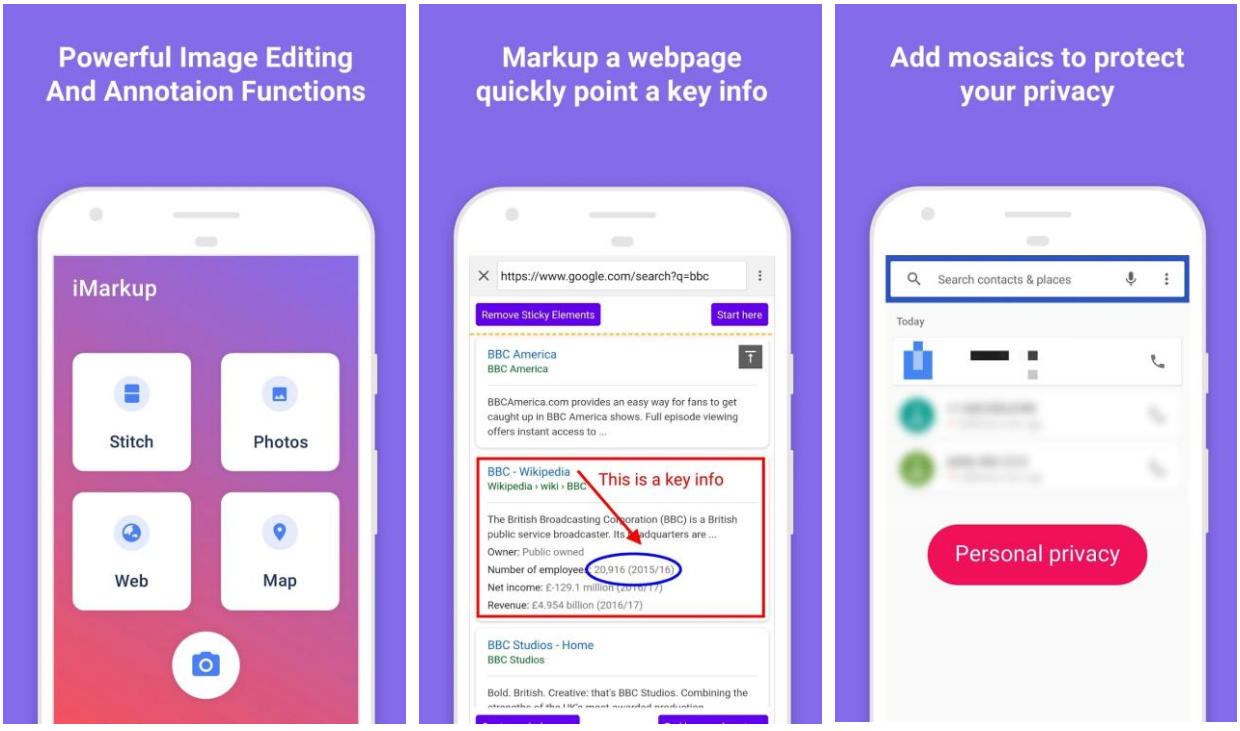
Sl. 2.7. Aplikacija Annotate [11]

2.4.2. iMarkup mobilna aplikacija

Mobilna Android aplikacija iMarkup [12] je još jedan moćan alat koji korisnicima omogućuje označavanje slika. Pruža i dodatne mogućnosti kao što su dodavanje teksta, pikseliranje slike, obrezivanje slike i još mnogo toga. Kao glavne prednosti ističu se mala veličina aplikacije koja iznosi manje od 10 megabajta, besplatno preuzimanje, visoka kvaliteta označene slike te podržavanje PNG formata slike. Aplikaciju su razvili programeri iz *Winteroso Team*-a, a objavljena je 1. siječnja 2019. godine. iMarkup je najpopularnija aplikacija za anotiranje slika s više od 1 milijun preuzimanja te vrlo visokom prosječnom ocjenom na *Google Play store*-u 4.7/5. Neke od glavnih značajki ove aplikacije su:

- Obrezivanje i rotiranje slike – mogućnost izrezivanja u razne oblike (pravokutne, zvjezdaste, trokutaste itd.)
- Povećavanje i zamagljivanje slike – pikseliziranje dijela slike kako bi se zamutila područja koja se ne žele prikazati
- Obilježavanje slike raznim oblicima kao što su strelice, pravokutnici, krugovi itd.
- Spajanje više slika u jednu panoramsku sliku vodoravno ili okomito
- Spremanje označene slike u galeriju te dijeljenje s ostalim korisnicima

Slike 2.8. a), b) i c) prikazuju aplikaciju iMarkup



a) **b)** **c)**

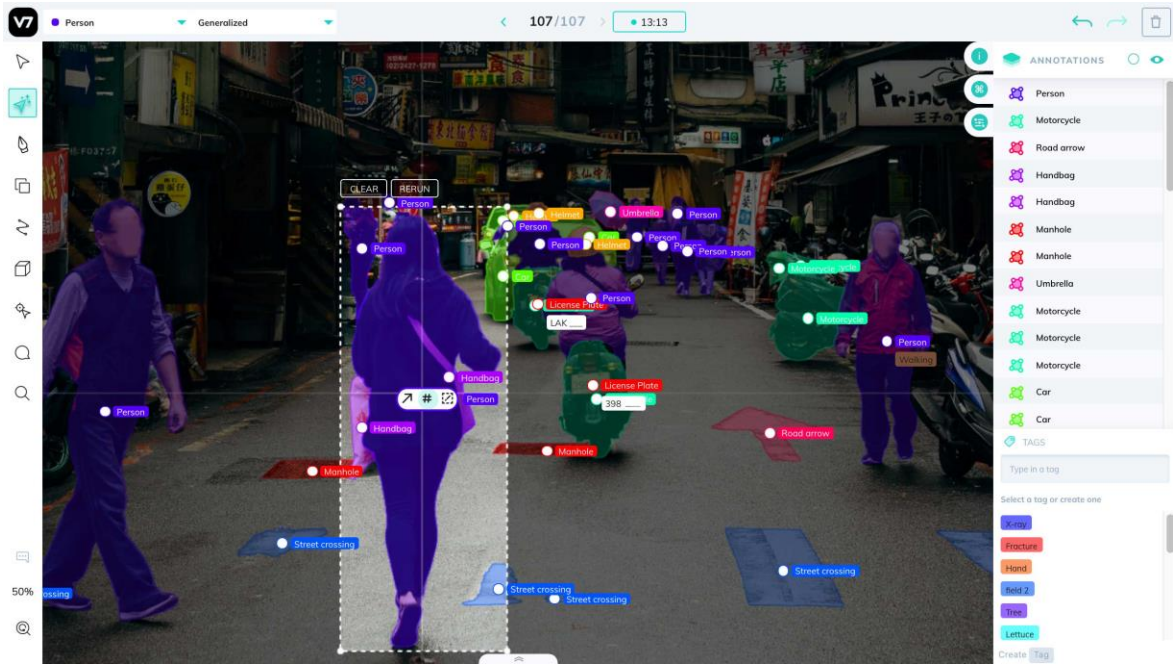
Sl. 2.8. Aplikacija iMarkup [12]

2.4.3. V7 web alat za anotiranje

V7 [13] je iznimno snažna platforma koja je nastala 2018. godine, a služi za treniranje modela strojnog učenja koja omogućuje anotaciju slika, videozapise, mikroskopske slike i još mnogo toga. Prema informacijama iz 2022. godine čak 60 ljudi je zaposleno na održavanju i unaprjeđenju ove platforme. Poznato je da se ljudski vidni korteks sastoji od 6 područja, V1 do V6, upravo iz toga proizlazi ime V7 kojim se želi naglasiti kako ova platforma cilja svim računalima pružiti vid te na tako proširiti ljudski vidni korteks. To je automatizirana platforma koja pruža najbrži način dobivanja visokokvalitetno označenih podataka koje je moguće koristiti za treniranje osobnog modela računalnog vida. V7 osim navedenih stvari pruža i usluge automatskog anotiranja slika, treniranje modela u oblaku te unajmljivanje profesionalnih anotatora ako je korisniku potrebna pomoć. Prema [14], V7 alat može smanjiti vrijeme označavanja podataka za čak 90%. Neke od glavnih značajki V7 alata za anotiranje slika su:

- Piksel-savršeno automatsko anotiranje slika
- Vrlo jednostavno upravljanje testnim skupom podataka
- Treniranje modela jednim klikom
- Upravljanje skupom podataka koje zadržava robusnost i u velikim razmjerima
- Skeniranje teksta i izvlačenje bitnih informacija pomoću već istreniranog modela

Ova platforma pruža korisnicima uvoz i izvoz anotacija u najpoznatijim formatima kao što su JSON, COCO, CVAT i ostali. Na slici 2.9 prikazan je izgled V7 sučelja za anotaciju slika.



Sl. 2.9. Sučelje V7 alata za anotiranje slike [15]

2.4.4. LabelMe web alat za anotiranje

LabelMe [16] je besplatni web alat otvorenog koda za anotiranje slika koje je moguće koristiti za treniranje modela strojnog učenja. To je projekt stvoren na MIT-u, a prvi put se spominje u članku iz 2008. godine gdje opisano na koji način funkcionira te kako je napravljen LabelMe. Sam program napisan je u Pythonu, a za grafičko sučelje koristi Qt. Ovaj alat je vrlo intuitivan i lagan za korištenje pa je upravo zbog toga i činjenice da je besplatan, vrlo popularan izbor za anotiranje slika. LabelMe ima mnoštvo korisnih značajki no isto tako dolazi i s nekoliko ograničenja. U nastavku su prikazane prednosti i mane LabelMe alata.

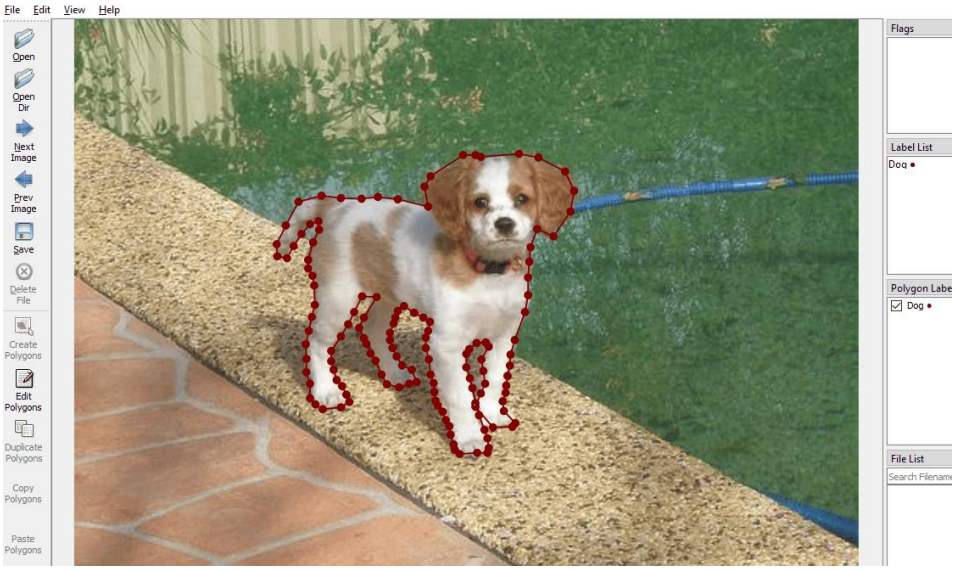
Prednosti:

- Pruža mnogo različitih oblika za anotiranje (pravokutnik, poligon, krug, linija itd.)
- Omogućuje obradu više slika kao skupine – moguće je otvoriti mapu u kojoj se nalaze sve slike te se prolaskom kroz njih može označiti svaka slika
- Vrlo lagana aplikacija s laganom instalacijom za sve vrste sustave

Mane:

- Nije skalabilan – ne omogućuje laganu suradnju ako projekt koji je u pitanju uključuje veći broj ljudi
- Ograničeni formati izvoza – LabelMe ne podržava izvoz u neke od popularnijih formata kao što su YOLO i CreateML
- Ne omogućuje alate za sigurno upravljanje skupovima podataka što ga čini gotovo neupotrebljivim pri anotaciji velike količine podataka

Slika 2.10. prikazuje sučelje LabelMe web alata za anotiranje slika.



Sl. 2.10. Sučelje LabelMe web alata [17]

3. TEHNOLOGIJE I ALATI KORIŠTENI U IZRADI RADA

U ovom poglavlju bit će objašnjene glavne tehnologije i alati bez kojih izrada ovog diplomskog rada ne bi bila moguća. Za početak će biti prikazan pregled Android operacijskog sustava kroz povijest do danas. Zatim će biti opisano Android Studio razvojno okruženje sa svojim glavnim značajkama i konačno programski jezik Kotlin kojim će biti pisan programski kod aplikacije izrađene kao praktični dio ovoga rada.

3.1. Android operacijski sustav

Iako je tvrtka *Google* kupila *start-up* tvrtku Android Inc. još 2005. godine, projekt Android operacijskog sustava zaživio je tek 2008. godine objavljivanjem Android 1.0. verzije [18]. Jedan od glavnih razloga ovome je *Google*-ova odluka da nastavi razvijati i nadograđivati platformu u tajnosti. *Google*-ovi planovi su se promijenili kada je 2007. godine tvrtka *Apple* na tržištu odlučila predstaviti svoj novitet *iPhone* koji je radio na iOS platformi i donio mnoge nove značajke i funkcionalnosti u svijetu mobilnih uređaja. Ovaj *Apple*-ov potez je na neki način prisilio *Google* da ubrzano završi razvijanje svog Android operacijskog sustava te svojim korisnicima pruži podjednako dobre funkcionalnosti kako bi zadržali konkurentnost. Kao što je već navedeno, *Google* 2008. godine izbacuje Android 1.0. koja je odmah uključivala kolekciju *Google*-ovih aplikacija kao što su *Gmail*, Mape, Kalendar itd. Osim toga, Android 1.0 je došao i s podrškom za zaslon na dodir i virtualnom tipkovnicom koja je zamijenila dotadašnje fizičke tipkovnice ugrađene na samom uređaju. Slika 3.1. prikazuje uređaj na kojem je pokrenut Android 1.0 operacijski sustav te neke od aplikacija koje su tada bile dostupne.

Kroz godine, izgled i mogućnosti Android operacijskog sustava su se razvijali i usavršavali, pa tako danas, svaki uređaj ima skup zajedničkih funkcionalnosti koje dijele svi uređaji koji rade na ovom sustavu. Te funkcionalnosti mogu se podijeliti u 3 različite kategorije:

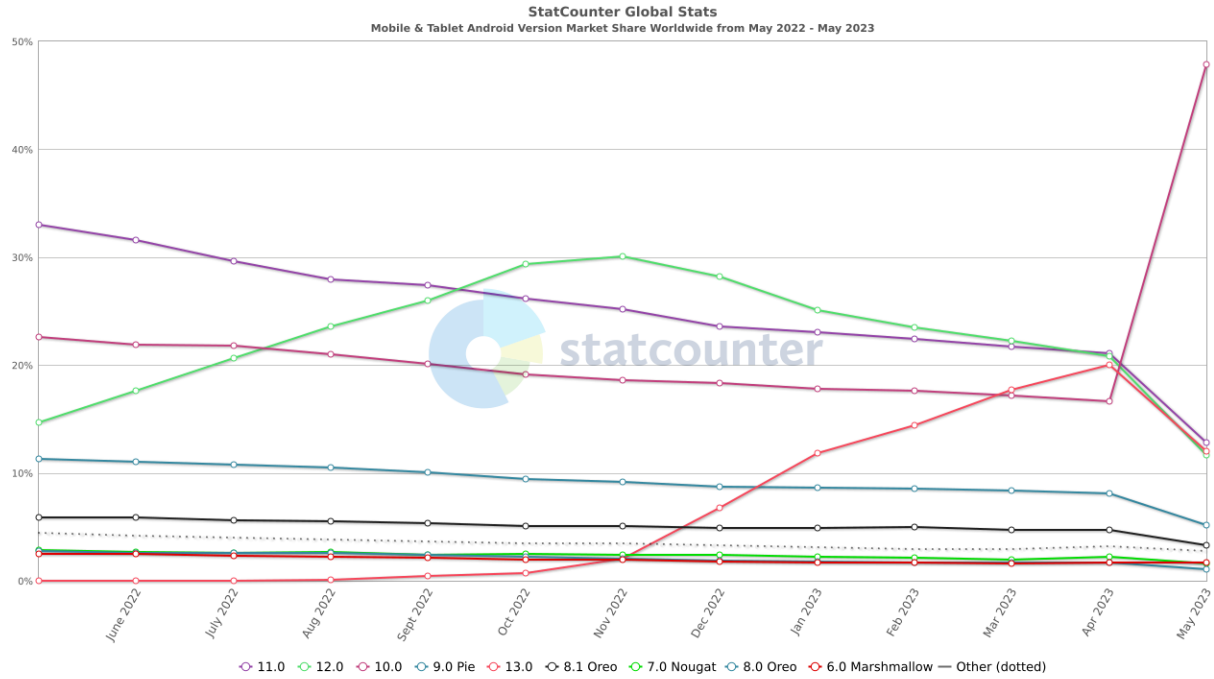
1. Korisničko sučelje – Android operacijski sustav pruža korisničko sučelje koje se sastoji od početnog zaslona, područja za obavijesti, navigacijskog područja te ladice u kojoj su pohranjene instalirane aplikacije
2. Povezanost i integracija – Android operacijski sustav pruža različite tehnologije bežičnog povezivanja (*Wi-Fi, NFC, Bluetooth* itd.), integraciju s *Google*-ovim servisima u oblaku (*Cloud, Drive, Photos* itd.) te razne senzore i hardverske komponente koji olakšavaju interakciju s okolinom

3. Privatnost i sigurnost – glavna značajka sigurnosnog sustava kod Android operacijskih sustava je sustav dopuštenja. Korisniku se šalje zahtjev za dopuštenje neke aplikacije koji on može prihvatiti ili odbiti. Tako korisnik sam ima kontrolu nad dopuštenjima bilo koje aplikacije sustava

Osim ovih glavnih funkcionalnosti, jedna od prednosti Android operacijskih sustava je i mogućnost prilagođavanja uređaja svakom od korisnika. Korisnik tako može uređivati svoju temu sustava, prilagođavati geste koje sustav prepoznaje, kreirati različite prečace u sustavu i još mnogo toga.

U 2009. godini *Google* je uveo prepoznatljivo imenovanje verzija Android operacijskog sustava po različitim desertima. Prva tako imenovana verzija je 1.5 pod nazivom Kolačić (engl. *Cupcake*), a posljednja stabilna verzija je Android 13.0 pod nazivom Tiramisu. Na slici 3.2. može se vidjeti raspodjela Android uređaja po trenutnoj verziji operacijskog sustava koja se koristi. Sa slike 3.2. također je moguće primijetiti kako je do travnja 2023. godine uporaba verzija 10.0, 11.0, 12.0 te 13.0 bila poprilično izjednačena dok u svibnju 2023. godine dolazi do znatnog pada u korištenju verzija 11.0, 12.0 i 13.0 te značajnog skoka u korištenju verzije 10.0 operacijskog sustava. Prema [19], u trenutku pisanja ovog rada globalna raspodjela zastupljenosti najpopularnijih verzija Android operacijskih sustava redom glasi:

1. Verzija 10.0 – 47.25% korisnika
2. Verzija 11.0 – 13% korisnika
3. Verzija 13.0 – 12.16% korisnika
4. Verzija 12.0 – 11.79% korisnika



Sl. 3.2. Globalna raspodjela korisnika Android operacijskog sustava [19]

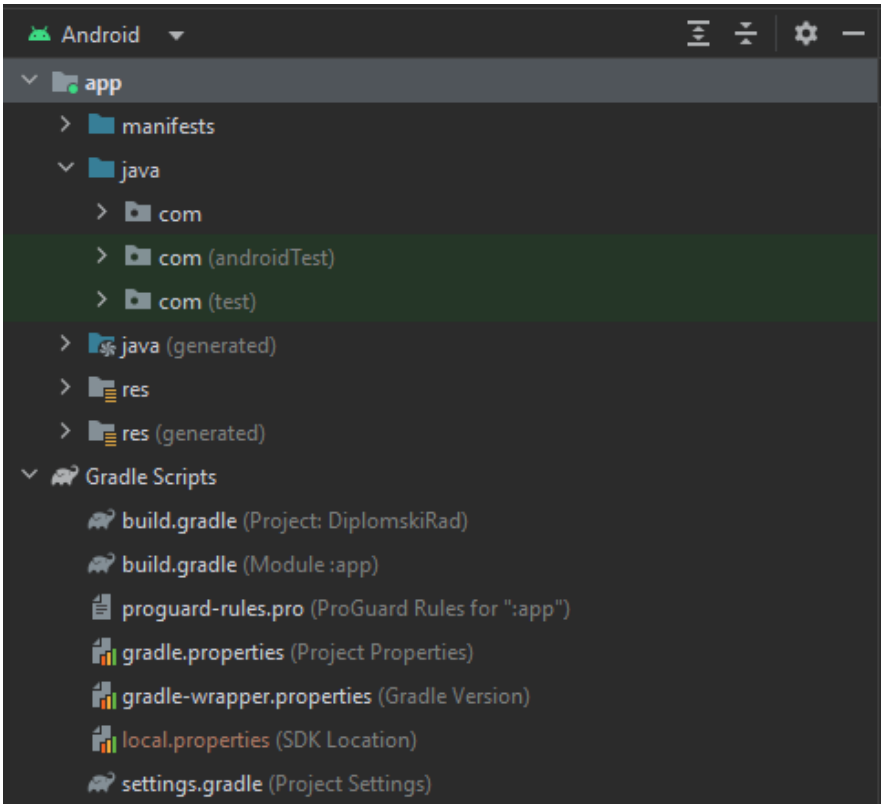
3.2. Android Studio razvojno okruženje

Budući da će u praktičnom dijelu rada biti razvijena Android mobilna aplikacija potrebno je i razvojno okruženje u kojemu je moguće razviti takvu aplikaciju. Iako postoje mnoga popularna razvojna okruženja, kao što su *Eclipse* ili *Visual Studio*, koja nude različite mogućnosti za programere, jedno se ipak ističe među njima. Riječ je Android Studio integriranom razvojnom okruženju koje je prvi izbor pri izradi Android mobilnih aplikacija. Ovo razvojno okruženje službeno potvrđuje i preporučuje sam *Google*, a to svakako nije mala stvar [20]. Jedan od glavnih razloga zašto *Google* preporučuje korištenje Android Studio razvojno okruženje je činjenica da je ono bazirano na *IntelliJ IDEA* [21] okruženju iza kojeg stoji moćna tvrtka *JetBrains*, a koja je, u više navrata surađivala upravo s *Google*-om. Ovo je iznimno bitan podatak jer, kada iza nekog razvojnog okruženja stoje dvije snažne i velike tvrtke, sa sigurnošću se može reći kako će se to razvojno okruženje nastaviti održavati i poboljšavati. Razvojno okruženje Android Studio programerima nudi razne mogućnosti kod razvoja Android aplikacija. Neke od tih mogućnosti su:

- Snažan i prilagodljiv sustav za razvoj aplikacija utemeljen na *Gradle*-u
- Emulatore s velikim brojem značajki i funkcionalnosti koji služe za virtualnu simulaciju fizičkih Android uređaja
- Unikatno okruženje koje nudi razvoj aplikacija za sve vrste uređaja s Android operacijskim sustavom

- Podršku za *Jetpack Compose* u vidu ažuriranja *Composable* komponenata u stvarnom vremenu na emulatoru
- Integracija s *GitHub* sustavom za verzioniranje koda
- Intuitivni i snažni alati za testiranje napisanog koda

Kao što je vidljivo na slici 3.3. projekt kreiran u Android Studio razvojnom okruženju sastavljen je od 2 modula: *app* i *Gradle Scripts*. U aplikacijskom modulu nalazi se više različitih direktorija kao što su: *manifests*, *java* i *res*. Unutar *manifests* direktorija nalazi se datoteka AndroidManifest.xml u kojoj su deklarirane sve komponente aplikacije kao i njihove aktivnosti i ponašanja, dopuštenja koja aplikacija smije koristiti u sustavu, naziv aplikacije i mnoge druge stvari. *Java* direktorij sadrži sve klase koje se koriste u projektu zajedno s izvornim kodom koji je napisan u njima. Također, u ovom direktoriju se nalazi i *JUnit* kod za testiranje. U *res* direktoriju nalaze se svi resursi korišteni u projektu (dimenzije, stilovi, teme, slike, boje itd.). U *Gradle Scripts* modulu nalaze se *Gradle* skripte koje su nužne za pokretanje aplikacije. Svaki projekt ima dvije *Gradle* datoteke, jednu na razini cijelog projekta i jednu na razini svakog modula u projektu. U ovom modulu još se nalaze i razne postavke i konfiguracije vezane uz projekt.



Sl. 3.3. Moduli unutar Android Studio projekta

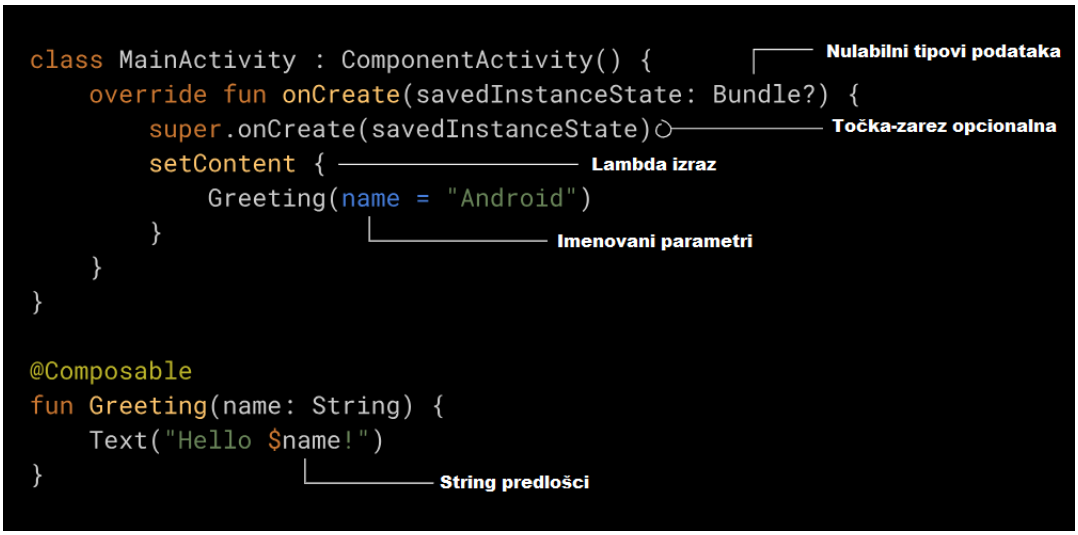
3.3. Kotlin programski jezik

Kotlin [22] programski jezik prvi put se pojavio 2011. godine kada ga je predstavila tvrtka *JetBrains*, a od 2019. godine je službeno podržan od *Google*-a kao preferirani programski jezik za razvoj Android aplikacija. Kotlin programski jezik može se opisati kao sažet i izražajan programski jezik što znači da prilikom kompajliranja i izgradnje koda svaki izraz mora biti poznat kompajleru (engl. *Compiler*). Tako se vrlo rano prepoznavanje grešaka u kodu te čak i sam kompajler može predložiti rješenje nekih grešaka. Zbog svega navedenog, Kotlin pripada skupini statički pisanih programskih jezika zajedno s jezicima kao što su: Java, C++, Swift, Go itd. Druga skupina programskih jezika su dinamički pisani programski jezici koji, za razliku od prethodne skupine, provjeru koda provode prilikom izvođenja samog koda. Neki od programskih jezika koji pripadaju ovoj skupini su: JavaScript, Python, Ruby, PHP itd.

Prema [23], Kotlin programski jezik vrlo je cijenjen kod razvojnih programera diljem svijeta zbog svojih svestranih funkcionalnosti. Neke od njih su:

- Sažetost i izražajnost – Kotlin pruža jednostavnu i razumljivu sintaksu koja omogućava lako izražavanje zamišljenih ideja i smanjuje količinu potrebnog koda za pisanje nekih funkcionalnosti.
- Povećana sigurnost koda – Kotlin pruža bogat izbor različitih značajki koje uvelike pridonose izbjegavanju nekih učestalih grešaka u pisanju koda. Jedna od najpoznatijih takvih grešaka je zasigurno iznimka nultog pokazivača koja je jako lijepo eliminirana u Kotlinu.
- Usklađenost s Java programskim jezikom – Java programski jezik je dugo bio najpopularniji izbor za izradu Android mobilnih aplikacija. Iako je to u današnje vrijeme Kotlin, još uvijek se pruža potpuna interoperabilnost s Javom što je iznimno bitno u starijim projektima.
- Paralelnost – Kotlin korutine (engl. *Coroutines*) su vrlo moćna funkcionalnost koja omogućuje izvođenje asinkronog koda. Korutine također znatno olakšavaju upravljanje svim vrstama pozadinskih zadataka.

Na slici 3.4. prikazan je dio programskog koda napisan u Kotlinu, a u nastavku će biti opisana sintaksa i prednosti Kotlina



Sl. 3.4. Programski kod u Kotlinu [24]

Kao što je već navedeno ranije u ovom poglavlju, Kotlin pruža sigurnost u radu s nulabilnim tipovima podataka tako da ih kompajler prepoznaje i provjerava napisani kod te javlja programeru ako postoji neki problem. Na ovaj način osiguravaju se ispadi aplikacije uzrokovani *null* vrijednošću. Jedna od vrlo pozitivno prihvaćenih promjena koje donosi Kotlin je opcionalno pisanje točke zareza u kodu. Kompajler sam raspoznaje kraj linije koda i zbog toga se pisanje točke zareza u Kotlinu smatra redundantnim. Lambda izrazi su još jedna novost u Kotlinu, a to su zapravo pojednostavljeni prikazi funkcije. U primjeru sa slike 3.4. prikazana je jednostavnost predavanja koda u funkciju kao parametar, a to je moguće upravo zbog lambda izraza. Imenovani parametri u Kotlinu poboljšavaju čitljivost koda i omogućavaju olakšano razumijevanje kompleksnog koda. *String* predlošci omogućavaju lakše ulančavanje kada se radi s tipom podataka *String*. Ovo su samo neke od prednosti koje Kotlin programski jezik donosi u odnosu na druge programske jezike. Upravo zbog svih razloga i prednosti nabrojanih u ovom poglavlju, Kotlin programski jezik je izabran za izradu Android aplikacije koja služi kao praktični dio ovoga rada.

3.4. GitHub servis za verzioniranje koda

U današnje vrijeme je nezamislivo da jedan programer nije bar čuo ili radio s GitHub servisom za verzioniranje koda. Prema [25], GitHub je iznimno popularan servis koji omogućuje ugošćivanje (engl. *hosting*) Git repozitorija u oblaku. Ova platforma nastala je 2008. godine, a danas broji više od 73 milijuna razvojnih inženjera koji koriste GitHub kao jedan od glavnih alata za verzioniranje koda i više od toga. GitHub je originalno započeo kao mjesto gdje su programeri mogli međusobno surađivati, a danas je vodeća platforma na kojoj ljudi mogu pohraniti svoje projekte i radove te dobiti pregled razvoja istih. GitHub je lakše za shvatiti ako se podijeli na 2 dijela: Git i središte

(engl. *Hub*). Git dio najlakše je opisati kao sustav koji omogućuje kontrolu verzija programskog koda te upravljanje promjenama koje se događaju na kodu. Spomenuta kontrola koda je neophodan proces u stvaranju softverskog projekta i uvelike olakšava rad velikog broja programera na istom projektu. Na ovaj način programerima je omogućeno da na jednostavan način surađuju na projektima, dijele ideje i prototipe te uređuju i kontroliraju svoj kod. *Hub* dio je nešto jednostavniji za shvatiti. To je zapravo mjesto koje okuplja istomišljenike i pruža im potporu u zajedničkom radu. Opći stav zajednice je pomoći drugima neovisno o razini tehničkog znanja koje pojedinac posjeduje. Glavne prednosti koje GitHub donosi su [26]:

- *Markdown* – iznimno lagani jezik kojim je moguće stvoriti formatirani tekst korištenjem najobičnijeg uređivača teksta. Služi za komentare korisnika, praćenje problema ili GitHub wikije i još mnogo toga.
- Dokumentacija – GitHub ima jednu od najboljih dokumentacija koje je moguće pronaći. Pokrivena su sva područja koja imaju veze s Git-om kroz razne članke, vodiče i odjeljke.
- *Gists* – značajka GitHub-a koja omogućuje pretvaranje datoteka u Git spremište. Na ovaj način još je dodatno pojednostavljeno praćenje izmjena ili dijeljenje napravljenih datoteka, skripti ili koda.
- Suradnja – GitHub je jedan od najmoćnijih alata za kolaboraciju u kodiranju. Iznimno lako se nosi s grananjem i spajanjem što direktno omogućava da više programera radi na istom dijelu koda.
- Sigurnosna kopija – GitHub pruža mogućnost jednostavnog skladištenja projekta i projektnog koda u repozitoriju na mreži.

4. ARHITEKTURA PROGRAMSKOG RJEŠENJA

U prvom dijelu ovog poglavlja bit će opisani svi funkcionalni zahtjevi koje aplikacija mora zadovoljiti kako bi bila valjana te dijagram tijeka aplikacije. Zatim će biti opisana web platforma *Firebase* i njezini dijelovi koji su korišteni pri izradi aplikacije. Na kraju će biti opisana i konkretna arhitektura koja će biti korištena za ostvarenje aplikacije.

4.1. Funkcionalni zahtjevi na aplikaciju

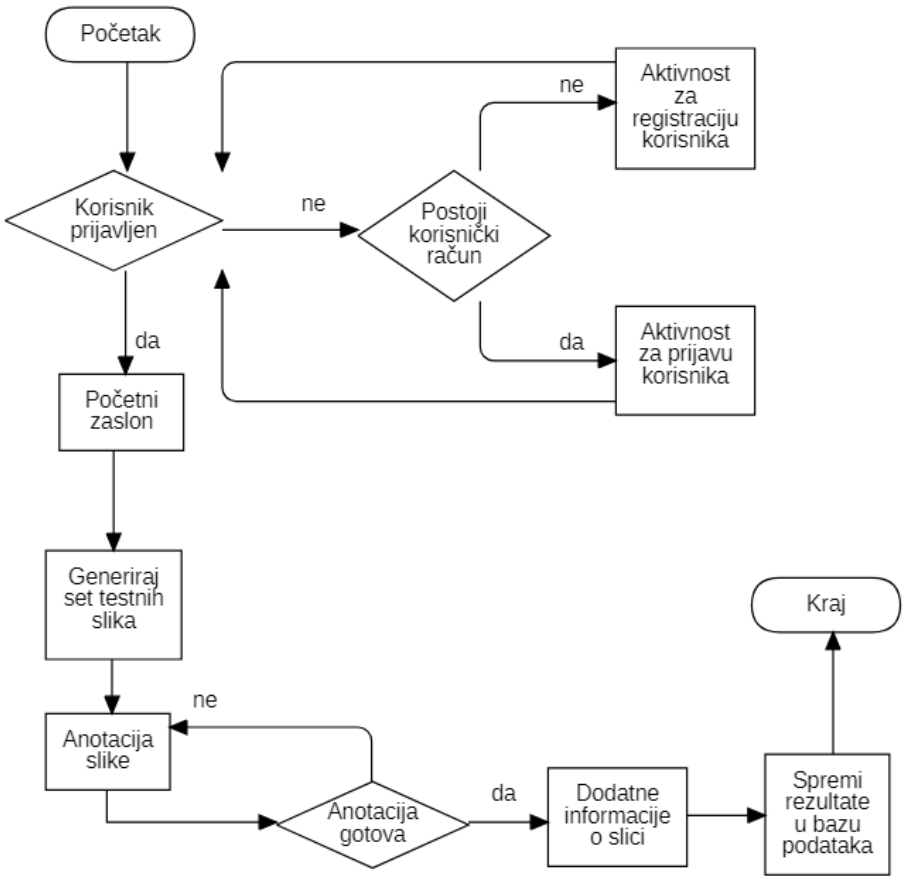
U praktičnom dijelu ovog rada bit će razvijena mobilna Android aplikacija koja će korisnicima omogućiti anotiranje slika vatre i dima koje će se kasnije koristiti u svrhu treniranja modela strojnog učenja. Osnovni funkcionalni zahtjevi na aplikaciju pokrivaju sve slučajeve koje aplikacija mora zadovoljavati te olakšavaju rukovanje samom aplikacijom. U slučaju ove aplikacije, funkcionalni zahtjevi su:

- Omogućiti korisniku kreiranje novog računa pomoću adrese elektroničke pošte i zaporke
- Omogućiti korisniku prijavu u aplikaciju putem svog računa
- Popuniti bazu podataka slikama za anotaciju
- Generirati set od 10 slika spremnih za anotaciju
- Omogućiti korisniku anotiranje slike pomoću postavljanja točaka po dijelovima slike formirajući nepravilan oblik
- Omogućiti korisniku funkciju brisanja posljednje dodane točke ukoliko dođe do greške
- Omogućiti spremanje anotiranog oblika i prelazak na drugi objekt
- Omogućiti korisniku različite klase anotiranja (vatra/dim)
- Zatražiti od korisnika dodatne informacije vezane uz konkretnu sliku koja se anotira (tip prostora, doba dana, veličinu plamena)
- Nakon uspješnog anotiranja slike i popunjenih dodatnih informacija o slici spremiti rezultate u bazu podataka

Nužno je osigurati da svaki navedeni zahtjev bude ispravno implementiran kako bi aplikacija korisniku pružila potpuno iskustvo korištenja. Također, nakon što je aplikacija razvijena i spremna za rad ovi funkcionalni zahtjevi poslužit će za identifikaciju i provjeru rada svih funkcionalnosti aplikacije. Koraci implementacije i najbitniji dijelovi koda bit će objašnjeni u sljedećem poglavlju.

4.2. Dijagram tijeka aplikacije

U svrhu lakšeg shvaćanja svih aktivnosti aplikacije te olakšanog kretanja kroz aplikaciju napravljen je dijagram tijeka koji se može vidjeti na slici 4.1. Prilikom ulaska u aplikaciju prvo se provjerava je li korisnik već autoriziran u aplikaciji ili ne. Ako je odvodi ga se na početni zaslon aplikacije, a ako nije odvodi ga se na aktivnost za prijavu, odnosno registraciju, u ovisnosti o tome postoji li već račun ili ga je potrebno kreirati. Nakon uspješne prijave u sustav korisnik na početnom zaslonu aplikacije može izabrati generiranje testnog seta slika te ga se odvodi na aktivnost u kojoj odrađuje anotaciju. Na svakoj slici korisnik prvo odabire klasu objekta koji želi anotirati (vatra ili dim) te započinje anotaciju tako da po granicama objekta postavlja točke koje na kraju čine nepravilan oblik oko tog objekta. Kada je korisnik označio sve objekte na slici postavljaju mu se dodatna pitanja na kojima označava tip prostora, doba dana i veličinu plamena na slici. Nakon što potvrdi sve potrebne podatke, rezultati se šalju u bazu podataka gdje se spremaju i čuvaju. Rezultati anotiranja kasnije će se koristiti u svrhu treniranja modela strojnog učenja. Za kreiranje dijagrama tijeka korišten je programski alat StarUML [27].



Sl. 4.1. Dijagram tijeka aplikacije

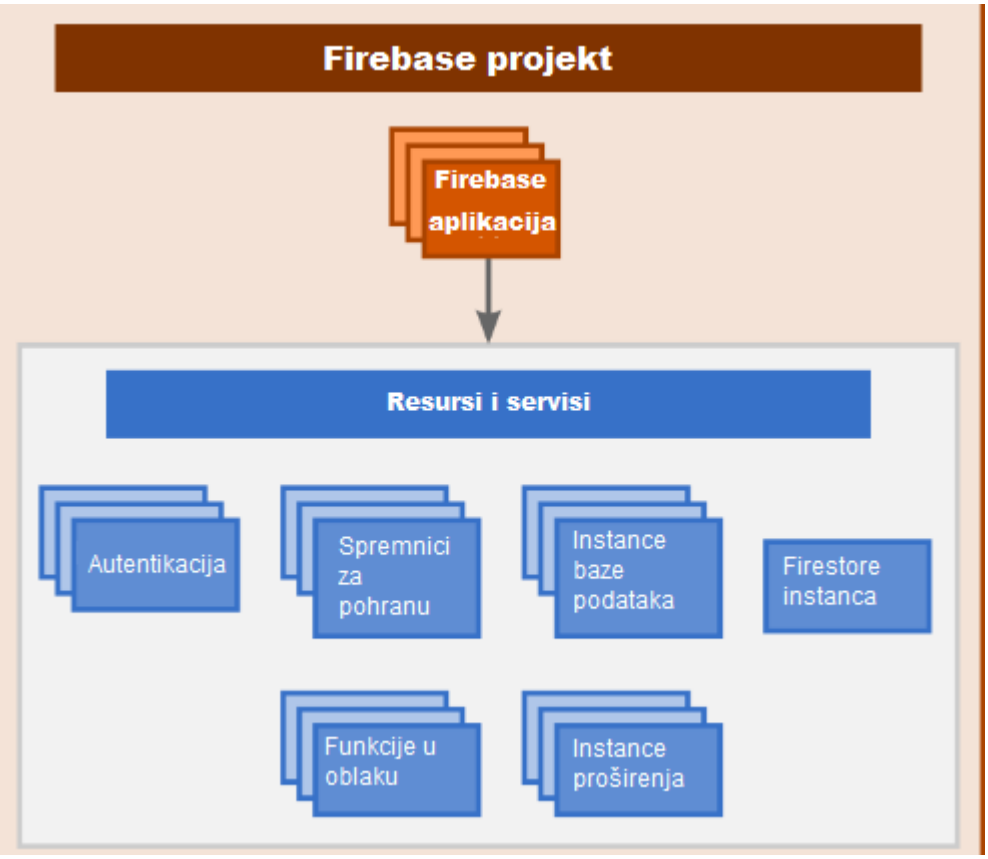
4.3. Firebase razvojni alati

Prema [28], Firebase je *Google*-ova platforma nastala 2012. godine kojoj je glavni cilj olakšanje razvoja, izgradnje i skaliranja aplikacija. To je zapravo skupina razvojnih alata utemeljena u oblaku te je kao takva izvrstan primjer BaaS (engl. *Backend as a Service*) usluge. Prema [29], BaaS uslugama razvojni programeri mogu drastično smanjiti vrijeme potrebno za izgradnju i kreiranje mobilne aplikacije korištenjem predefiniranih programskih alata i API-ja. Ova usluga tako pruža mogućnost jednostavnog upravljanja bazom podataka u oblaku. Glavne značajke koje pruža Firebase su:

- Autentifikacija – na vrlo jednostavan i siguran način moguće je upravljati korisnicima u sustavu. Budući da će se ova funkcionalnost koristiti i u ovom radu, ona će biti još detaljnije opisana u sljedećem odjeljku
- Baza podataka u stvarnom vremenu – usluge NoSQL baze podataka koja pruža mogućnost pohrane i sinkronizacije bitnih podataka i to sve u stvarnom vremenu. Ova funkcionalnost je posebno bitna u aplikacijama koje imaju čestu promjenu podatka te im je ažurnost najbitnija karakteristika
- Slanje poruka u oblaku (engl. *Firebase Cloud Messaging*) – usluga kojom razvojni inženjeri ili vlasnici aplikacije mogu na jednostavan način poslati poruku ili obavijest svim korisnicima aplikacije.
- *Crashlytics* – moćan alat Firebase-a koji omogućava praćenje rada i ispada u aplikacijama. Nudi iznimno detaljan izvještaj ako dođe do pada aplikacije i tako razvojnim programerima olakšava pronalaženje uzroka problema.
- Sustav za praćenje performansi – Firebase pruža lijepo dizajnirano sučelje za praćenje performansi aplikacije. Na ovaj način moguće je pratiti potencijalne probleme kao što su preveliko korištenje procesorske snage, memorije ili mrežnog prometa.
- Ispitni laboratorij – važna usluga koja programerima pruža mogućnost testiranja razvijene aplikacije na velikom izboru različitih uređaja ili konfiguracija. Korištenjem ove usluge moguće je utvrditi ispravan rad na raznim uređajima odnosno prepoznati probleme u radu na nekom od testiranih uređaja.

Slika 4.2. prikazuje detaljnu hijerarhiju Firebase projekata i njihove odnose. Prva komponenta je Firebase projekt kojega se moguće zamisliti kao spremnik u kojemu se nalaze sve aplikacije, resursi i usluge koje je potrebno koristiti. U jednom projektu moguće je registrirati više od jedne aplikacije (npr. Android i iOS verziju iste aplikacije) dok im se osigurava pristup istim uslugama

i resursima koji se koriste na razini projekta. Dobar primjer dijeljenja istih usluga je autentifikacija korisnika. Aplikacija na obje registrirane platforme koristi istu uslugu za autentifikaciju te je moguća prijava s istim korisničkim podacima neovisno na kojoj platformi se trenutno koristi. Još jedan primjer je usluga analitika gdje su sve registrirane aplikacije na spojene na isti entitet za analitike ali svaka aplikacija zasebno ima uvid u svoje podatke.

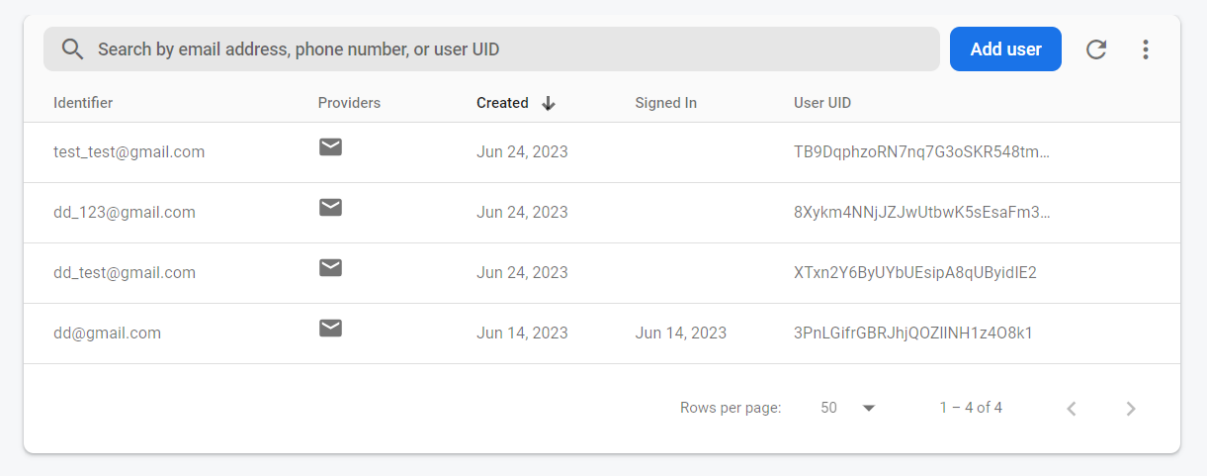


Sl. 4.2. Hijerarhija Firebase projekta

4.3.1. Firebase usluga za upravljanje korisnicima

U današnje vrijeme vrlo je bitno imati nekakav sustav autorizacije u aplikaciji. Tako se vodi evidencija o korisnicima te se pruža dodatan sloj zaštite korisničkom računu. Upravo iz tog razloga, u ovoj aplikaciji koristit će se Firebase sustav za autentifikaciju korisnika. Prema [30], Firebase sustav za autentifikaciju korisnika pruža gotova biblioteke i alate za upravljanje korisnicima u aplikaciji. Nudi rješenja prijave i registracije korisnika putem elektroničke pošte i lozinke, telefonskog broja ili čak povezivanjem postojećih računa s Google-a, Facebook-a ili sličnih mreža. Nakon prijave korisnika omogućava se korištenje dodatnih značajki kao što su: verifikacija elektroničke pošte, verifikacija broja mobilnog uređaja, ponovno postavljenje lozinke, autentifikacija s više faktora i još mnogo toga. Moguće je i nadograditi na plaćenu verziju

autentifikacije koja omogućava još dodatnih značajki kao što su blokiranje korisnika, praćenje korisničke aktivnosti, korisničke kvote bez ograničenja i druge. Za potrebe ove aplikacije ipak je dovoljan obični besplatni plan ove usluge. Na slici 4.3. prikazano je sučelje Firebase autentifikacijske usluge nakon dodanih par korisnika u sustav.



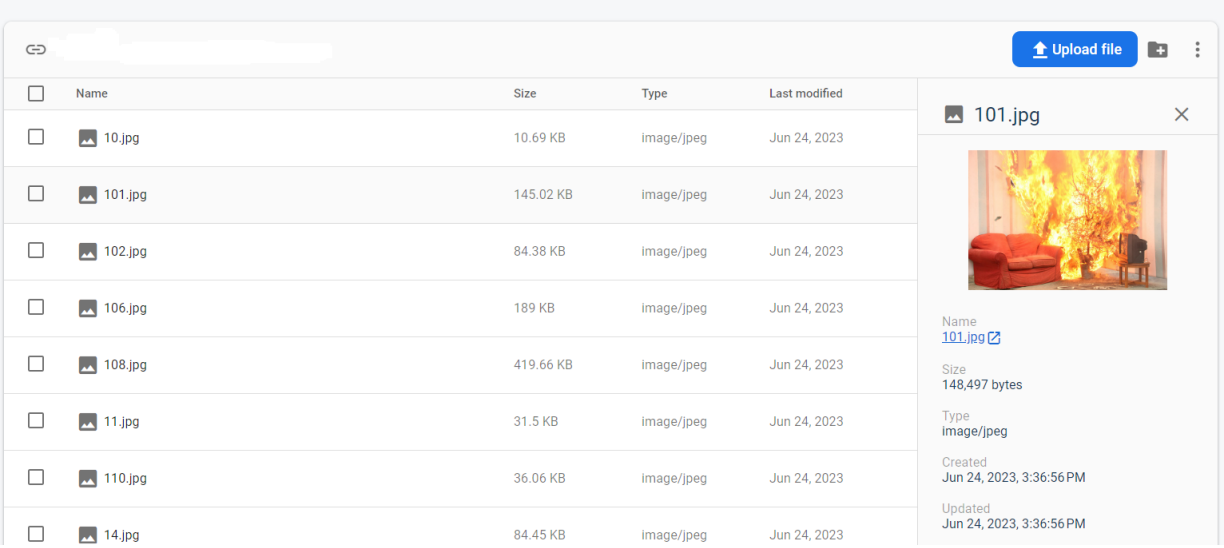
Sl. 4.3. Sučelje Firebase usluge za autentifikaciju korisnika

4.3.2. Firebase usluga spremnika u oblaku

Još jedna Firebase usluga koja će se koristiti u ovom radu je usluga pohrane u oblaku. Prema [31], Firebase usluga spremnika u oblaku (engl. *Cloud Storage*) iznimno je moćna i jednostavna usluga koja se koristi za spremanje i dohvaćanje objekata kao što su: slike, videozapisi, audio zapisi te razni drugi sadržaji. Ova usluga bit će korištena za spremanje slika vatre koje će se kasnije dohvaćati u aplikaciji kako bi ih korisnik mogao anotirati. Glavne značajke spremnika u oblaku su:

- Robusne operacije – neovisno o kvaliteti mreži Firebase-ov SDK dopušta preuzimanja i prijenos podataka. Robusnost u ovom slučaju znači da se u slučaju pucanja mreže ili nekog neplaniranog događaja preuzimanje ili prijenos podatka nastavlja tamo gdje je stalo te tako štedi vrijeme i resurse
- Visoka sigurnost – Firebase-ov SDK koristi integraciju sa sustavom za autentifikaciju te tako također pruža i vrlo intuitivnu verifikaciju za razvojne programere. Također pruža uslugu korištenja Firebase-ovog deklarativnog sigurnosnog modela za pristup željenom sadržaju
- Visoka skalabilnost – budući da pohrana u oblaku podržava iznimno velike količine prostora za skladištenje nema potrebe za mijenjanjem usluge ili infrastrukture u slučaju da aplikacija postane prepoznata na globalnoj razini.

Slika 4.4. prikazuje izgled spremnika popunjenog slikama vatre koje se koriste u aplikaciji.



Sl. 4.4. Sučelje Firebase spremnika u oblaku

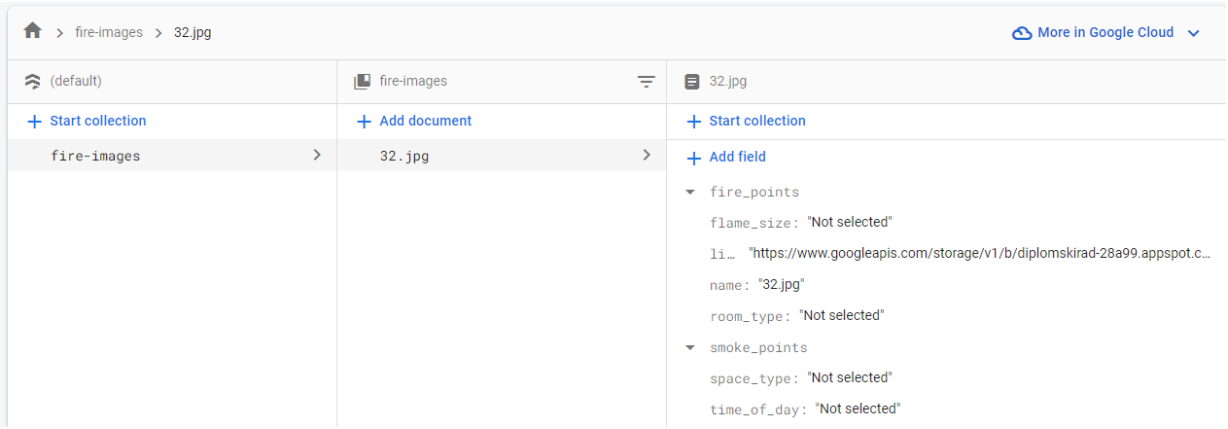
4.3.3. Firebase usluga baze podataka

Prema [32], Firebase usluga baze podatka u oblaku (engl. *Firestore*) pruža usluge fleksibilnog i skalabilnog rješenja za pohranu raznih podataka koja po obilježjima pripada u NoSQL grupu baza podataka. Za potrebe ovoga rada, koristit će se ova usluga kako bi se spremali rezultati anotiranja pojedinih slika. Ovakav način spremanja podataka je vrlo učinkovit jer se tim rezultatima lako može naknadno pristupiti te ih koristiti u svrhu treniranja modela strojnog učenja. Jedna od glavnih prednosti ovi usluge je mogućnost sinkronizacije podataka u svim aplikacijama registriranim na ovu uslugu i to putem sustava koji radi u stvarnom vremenu. Također je vrlo bitno napomenuti kako ova usluga pruža i podršku izvan mreže što znatno pridonosi responzivnosti aplikacija koje koriste ovu uslugu jer ne moraju voditi brigu o povezanosti na mrežu. U nastavku su prikazane glavne značajke i prednosti *Firestore* usluge:

- Fleksibilnost – *Firestore* usluga radi s modelom podataka koji podržava hijerarhijske strukture podataka. Upravo zbog toga moguće je podatke spremati u dokumente koji mogu sadržavati razne objekte i strukture.
- Ekspresivni upiti – *Firestore* usluga također nudi i mogućnost dohvaćanja specifičnih dokumenata iz baze podataka. To je moguće korištenjem ekspresivnih upita i podešavanja parametara upita.
- Ažurnost – kao što je već ranije navedeno, ova usluga pruža ažuriranje podataka u stvarnom vremenu na bilo kojem od uređaja.

- Podrška za rad izvan mreže – podaci koji se aktivno koriste se spremaju te tako omogućuju pisanje i čitanje podataka čak i ako uređaj nema mreže.

Slika 4.5. prikazuje sučelje usluge *Firestore* u koju će biti spremni rezultati anotiranja izvršenog kroz razvijenu aplikaciju.



Sl. 4.5. Sučelje *Firestore* usluge

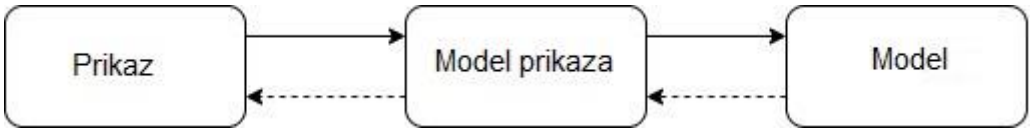
4.4. MVVM arhitektura mobilne aplikacije

U današnje vrijeme gotovo je nezamislivo započeti neki veliki projekt bez prijašnjeg definiranja arhitekture koja će se koristiti. Ovo tvrdnja vrijedi za sva područja razvoja aplikacija, ali ovaj rad usredotočuje se na arhitekture rješenja u mobilnom razvoju. Dodavanjem novih datoteka u projekt on posljedično raste u veličini te nakon određenog vremena postaje poprilično komplicirano pretraživati datoteke u jednom takvom projektu. Kako bi se izbjegle ovakve situacije kao i olakšalo snalaženje u velikom projektu, praksa svakog dobrog programera je organiziranje projekta po raznim paketima i direktorijima. Upravo u ovome pomažu pravila arhitekture koja zapravo pomažu u organizaciji koda predlaganjem strukture projekta. Na ovaj način znatno se olakšava rad velikog broja programera koji rade na istom projektu jer se sada znaju jasna pravila gdje i kako pronaći određenu datoteku u projektu.

Iako je odnedavno Google preporučio korištenje njihove čiste (engl. *Clean*) arhitekture u izradi mobilnih Android aplikacija za potrebe ovoga rada ipak će se koristiti MVVM arhitektura zbog jednostavnije primjene na manjim projektima kao što je ovaj. Glavna razlika između čiste i MVVM arhitekture je postojanje dodatne dimenzije slojeva kod čiste arhitekture. Čista arhitektura podrazumijeva 3 sloja po kojima se dijeli kod a to su prezentacijski sloj, domenski sloj i podatkovni sloj. Jedna od glavnih mana čiste arhitekture je zapravo njezina složenost i veliki broj paketa što ju čini izvrsnom za velike projekte pa je u ovom slučaju MVVM arhitektura je puno bolji izbor.

Prilikom objašnjavanja čiste arhitekture često se može čuti usporedba MVVM i čiste arhitekture. Prezentacijski sloj čiste arhitekture je zapravo i utemeljen na MVVM principima te koristi model prikaza iz te arhitekture za rad s podacima koji se kasnije prikazuju na zaslonu.

Prema [33], MVVM arhitektura u razvoju mobilnih Android aplikacija zapravo označava principe prema kojima se odvaja poslovna logika koda od njezinog prikaza na zaslonu aplikacije. Sastoji se od 3 dijela: modela (engl. *Model*), prikaza (engl. *View*) te modela prikaza (engl. *ViewModel*). Slika 4.6. prikazuje komponente MVVM arhitekture i njihove odnose.

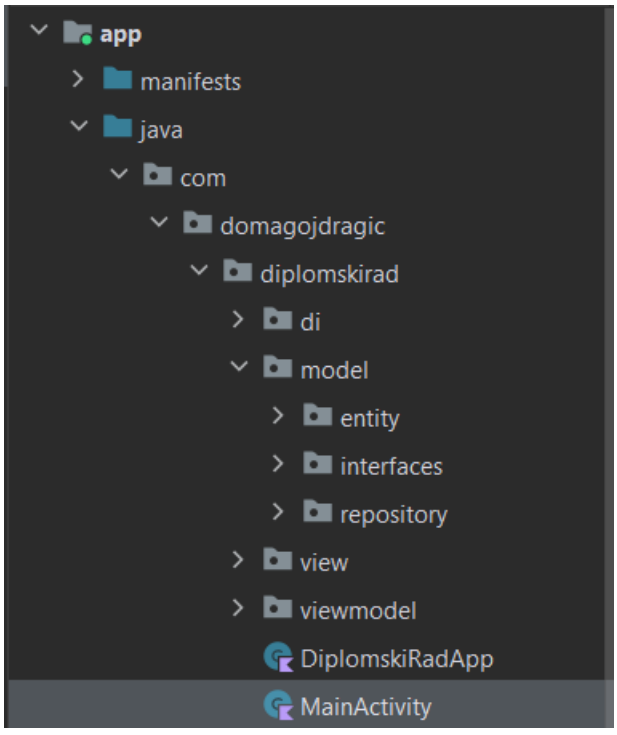


Sl. 4.6. Komponente MVVM arhitekture

Model komponenta MVVM arhitekture sadrži podatkovni model sustava odnosno sve podatkovne klase koje se koriste unutar aplikacije te poslovnu logiku sustava. Model ostvaruje komunikaciju s modelom prikaza dok za sam prikaz uopće ne zna a to je vrlo bitno zbog ostvarivanja značajke razdvajanja poslovne logike od prikaza. Komponenta prikaza je zapravo reprezentacija korisničkog sučelja sustava te sadrži samo nužne informacije koje joj služe za prikaz podataka na zaslonu. Bitno je napomenuti kako ova komponenta ne manipulira nikakvim podacima već sve bitne informacije dobiva preko modela prikaza s kojim direktno i komunicira. Kao što model ne zna za prikaz tako isto i prikaz ne zna za model u implementiranom sustavu. Posljednja komponenta MVVM arhitekture je model prikaza i on je zapravo poveznica između ostale dvije komponente. Unutar ove komponente nalazi se poslovna logika koda te se vrši manipulacija nad podacima koji se pripremaju za prikaz na zaslonu aplikacije. Model prikaza dobiva sve podatke koji mu trebaju od komponente modela, obrađuje ih i priprema za komponentu prikaza koja onda te podatke prikazuje na korisničkom sučelju.

Kako bi se dobro shvatio način rada MVVM arhitekture bitno je prvo dobro shvatiti svaku komponentu sustava i njezinu zadaću te način na koji pojedina komponenta komunicira s drugom. Ako krenemo od modela prikaza možemo reći kako je on zadužen za detektiranje korisnikove interakcije s korisničkim sučeljem (dodiri zaslona, unos na tipkovnici itd.). Kada model prikaza detektira događaj koji se dogodio na zaslonu on šalje signal na model prikaza da treba osvježiti neke podatke ili pak prikazati neke druge podatke u ovisnosti o korisnikovoj akciji. Model prikaza sadrži razna svojstva i naredbe na koje se sam prikaz može vezati i to je upravo njihova veza putem koje se vrši komunikacija između ove dvije komponente. Prikaz se tako pretplati na događaje za koje model prikaza vodi računa i kada je potrebno osvježi podatke te se oni prikažu na zaslonu.

Potrebno je osigurati da se ti podaci mogu pravovremeno i ažurno prikazati u sustavu. Ta funkcionalnost postiže se korištenjem mehanizama kao što su tok podataka (engl. *Flow*) [34] i korutine (engl. *Coroutines*) [35]. Tok podataka moguće je zamisliti kao otvorenu pipu kroz koju kontinuirano prolaze podaci. Kako je već navedeno komponenta prikaza pretplaćena je na tu pipu te direktno može te podatke prikazati na zaslonu. Korutine u ovom slučaju služe kako bi raspodijelile ovaj posao na pozadinske niti kako ne bi došlo do zastoja na glavnoj niti koja služi za iscrtavanje zaslona aplikacije. Posljednja komponenta koju je potrebno objasniti u ovoj komunikaciji je sam model. Model dohvaća sve podatke koji su potrebni za ispravan rad aplikacije s nekog vanjskog ili unutarnjeg spremnika. Model je zadužen da pravovremeno proslijedi te podatke modelu prikaza kako bi se tamo dodatno pripremili podaci za prikaz na zaslonu. Na ovaj način završena je cjelokupna komunikacija u MVVM arhitekturi i prikazan je njen način rada. Slika 4.7. prikazuje organizaciju koda aplikacije prema MVVM arhitekturi kao što je prethodno opisano.



Sl. 4.7. Organizacija koda u MVVM arhitekturi

5. IMPLEMENTACIJA PROGRAMSKOG RJEŠENJA

U ovom poglavlju bit će opisana mobilna Android aplikacija koja je razvijena kao praktični dio ovoga rada. Također će biti predstavljeni i opisani glavni dijelovi programskog koda te popratne snimke zaslona iz aplikacije. Aplikacija treba služiti kao alat za anotiranje slika vatre i dima na testnim slikama a u svrhu treniranja modela strojnog učenja. Aplikacija korisniku treba generirati skup slika spremnih za anotaciju. Nakon uspješnog anotiranja slike, aplikacija treba omogućiti spremanje rezultata u vanjsku bazu podataka. Također, aplikacija mora omogućiti i upravljanje korisnicima, prijavu i registraciju te omogućiti korištenje na raznim uređajima.

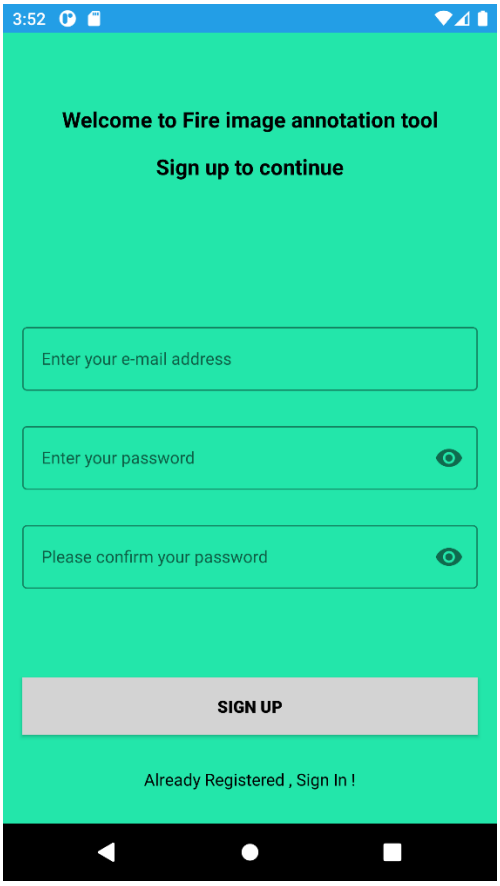
5.1. Upravljanje korisnicima u sustavu

Prilikom prvog pokretanja aplikacije korisnika se odvodi na aktivnost za prijavu u sustav. Korisniku unosi svoju elektroničku poštu i zaporku od postojećeg korisničkog računa. Ako korisnik nema kreiran važeći račun, može ga napraviti na aktivnosti za registraciju. Programski kod 5.1. prikazuje implementaciju registracije novog korisničkog računa preko Firebase usluge za autentifikaciju korisnika. Varijabla *firebaseAuth* inicijalizira se tako da se u nju postavlja instanca usluge za autentifikaciju korisnika. Zatim se implementira događaj pritiska gumba za registraciju. Prilikom pritiska gumba za registraciju, u varijable *email*, *password* i *confirmPassword* zapisuju se vrijednosti s ekrana koje je korisnik unio, a zatim ide provjera jesu li one prazne ili postoje nekakve vrijednosti u njima. Nakon što je potvrđena prisutnost vrijednosti u ovim varijablama sljedeće što se provjerava jest jednakost varijabli *password* i *confirmPassword*. Nakon što su ovi uvjeti zadovoljeni, na instanci *firebaseAuth* se poziva metoda koja kreira novog korisnika i zapisuje ga na udaljenom serveru u oblaku te ako je ta metoda uspješno odrađena korisnika se odvodi na početnu aktivnost aplikacije. Slika 5.1. prikazuje snimku zaslon aktivnosti za registraciju.

```
val email = binding.emailEt.text.toString()
val password = binding.passEt.text.toString()
val confirmPassword = binding.confirmPassEt.text.toString()

if (email.isNotEmpty() && password.isNotEmpty() &&
confirmPassword.isNotEmpty()) {
    if (password == confirmPassword) {
        firebaseAuth.createUserWithEmailAndPassword(email, password)
            .addOnCompleteListener {
                if (it.isSuccessful) {
                    val intent = Intent(this, SignInActivity::class.java)
                    startActivity(intent)
                } else {
                    Toast.makeText(this, it.exception.toString(),
```

```
Toast.LENGTH_SHORT)
                        .show()
                }
            }
```

Programski kod 5.1. Registracija korisnika u sustav



Sl. 5.1. Snimka zaslona za registraciju korisnika

Korisnik se također može odjaviti iz sustava tako da na početnoj aktivnosti proširi meni u gornjem desnom kutu i pritisne na opciju za odjavu. Zatim mu se prikazuje dijalog koji ga obavještava da je pritisnut gumb za odjavu iz sustava te još jednom traži potvrdu za izlazak. Implementacija opisanog je prikazana u programskom kodu 5.2.

```
binding.overflowToolbar.setOnMenuItemClickListener {
    when (it.itemId) {
        R.id.signOut -> {
            MaterialAlertDialogBuilder(this)
                .setMessage(getString(R.string.supporting_text))
                .setPositiveButton(getString(R.string.accept)) { _, _ ->
                    firebaseAuth.signOut()
                    val intent = Intent(this, SignInActivity::class.java)
                    startActivity(intent)
                }
                .setNegativeButton(getString(R.string.decline)) { dialog, _ -
>
                    dialog.dismiss()
                }
                .show()
            true
```

```
        }
        else -> false
    }
}
```

Programski kod 5.2. Odjava korisnika iz sustava

5.2. Generiranje skupa slika za anotiranje

Korisnik na početnom zaslonu još može pritisnuti gumb za generiranje skupa slika spremnih za anotaciju. Nakon što to učini odvodi ga se na aktivnost za anotaciju. Prilikom kreiranja ove aktivnosti inicijalizira se i odgovarajući model prikaza putem usluge za ubrizgavanje ovisnosti *Koin* [36]. *Koin* preuzima zadaću pravovremenog kreiranja mrežnog klijenta, repozitorija, modela prikaza i drugih modula koji su nužni za ispravan rad aplikacije. Programski kod 5.3. prikazuje primjer definiranja jednog *Koin* modula i primjer kreiranja varijable korištenjem delegata i Koin metode unutar navede aktivnosti.

```
val viewModelModule = module {

    viewModel { AnnotationViewModel(get(), get(named(BG_DISPATCHER))) }
}
private val annotationViewModel: AnnotationViewModel by viewModel()
```

Programski kod 5.3. Definiranje Koin modula i njegova primjena

Prilikom inicijalizacije modela prikaza okida se metoda koja obavještava repozitorij da se dogodio događaj na zaslonu aplikacije te je potrebno dohvatiti slike za anotiranje. Taj događaj dalje putuje od repozitorija do mrežnog klijenta koji obavlja API zahtjev te dohvaća zatražene slike koje prvo dolaze do repozitorija gdje se odvija transformacija iz *Response* klase podataka u *Entity* klasu podataka. Razlog tome je što *Response* klasa služi samo za dohvaćanje podataka s API-ja pa samim time može biti i nestabilna u daljnjem radu. Nakon što se odrade sve potrebne transformacije podataka oni dalje idu prema modelu prikaza koji ih je i zatražio. Unutar modela prikaza ti podaci se, ako je potrebno, dodatno obrađuju i pripremaju za prikaz na zaslonu. Nakon što je sve pripremljeno podaci se prikazuju na zaslonu i tako je upotpunjen tok podataka. Bitno je napomenuti da se na ovaj način poštuje dobra programerska praksa o jednosmjernosti podataka gdje se na zaslonu događa nekakva akcija koja redom poziva metode iz nižih slojeva i zatim se dobiveni podaci vraćaju obrnutim redoslijedom prema zaslonu. Programski kodovi 5.4., 5.5. i 5.6. prikazuju opisani postupak dohvaćanja podataka.

```
 override suspend fun fetchFireImages(): ImageListResponse =
client.get(API_URL).body()
@Serializable
data class ImageListResponse(
    @SerialName("items")
```

```
    val images: List<ImageResponse>
)
@Serializable
data class ImageResponse(
    @SerialName("name")
    val imageName: String,
    @SerialName("selfLink")
    val selfLink: String,
    @SerialName("contentType")
    val contentType: String,
)
```

Programski kod 5.4. Metoda koja dohvaća podatke s API-ja i *Response* klase

Metoda *fetchFireImages*() putem API poziva dohvaća sve podatke sadržane u Firebase-ovom spremniku u oblaku i prihvaća ih u obliku podatkovne klase *ImageListRepsonse* unutar koje je sadržana lista *ImageResponse* podatkovne klase koja u sebi sadrži ime slike, njezinu poveznicu u spremniku te njezin tip podatka. Anotacije *Serializable* i *SerialName* se koriste zbog serijalizacije JSON odziva koji se dobije prilikom API poziva u Kotlin sigurne tipove podataka.

```
private val storageImages = MutableSharedFlow<List<ImageEntity>>()

override fun getStorageImages(): Flow<List<ImageEntity>> {
    return storageImages
}

override suspend fun fetchStorageImages() =
    storageImages.emit(
        storageApi.fetchFireImages().images
            .filter { imageResponse ->
                imageResponse.contentType == CONTENT_TYPE_IMAGE
            }
            .shuffled()
            .subList(0, 10)
            .map { filteredImageResponse ->
                filteredImageResponse.toImageEntity()
            }
    )
```

Programski kod 5.5. Implementacija metoda unutar repozitorija

FetchStorageImages() metoda koja se prva poziva u modelu prikaza emitira dohvaćene i transformirane podatke koje dobiva s instance mrežnog klijenta *storageApi* na varijablu *storageImages* tipa *MutableSharedFlow*. Operacije koje se izvode na dobivenim podacima su redom:

- *.filter()* – operacija koja prima listu podataka tipa *ImageResponse* i filtrira ju tako što zadržava samo one elemente koji su tipa slike
- *.shuffled()* – elemente filtrirane liste vraća nasumičnim poretkom
- *.subList()* – uzima prethodnu listu i zadržava prvih deset elemenata
- *.map()* – svaki element liste mapira iz tipa *ImageRepository* u *ImageEntity* putem funkcije proširenja *toImageEntity()*

Metoda *getStorageImages()* vraća taj emitirani tok podataka koji će se kasnije koristiti u modelu prikaza.

```
private val _imagesState = MutableStateFlow(emptyList<ImageEntity>())
val imagesState = _imagesState.asStateFlow()

init {
    viewModelScope.launch(bgDispatcher) {
        imageRepository.fetchStorageImages()
    }
    viewModelScope.launch(bgDispatcher) {
        imageRepository.getStorageImages()
            .collect() { images ->
                _imagesState.value = images
            }
    }
}
```

Programski kod 5.6. Init blok modela prikaza

Pozivanjem metoda u *init* bloku osigurava se njihovo izvršavanje odmah prilikom kreiranja objekta te klase. Na ovom primjeru prikazan je poziv dvije metode unutar korutinskog opsega modela prikaza. Vrlo je bitno ovakve operacije postaviti na pozadinsku nit kako ne bi došlo do blokiranja glavne niti i samim time zamrzavanja zaslona aplikacije. Obje metode se pozivaju na varijabli *imageRepository* koja predstavlja instancu repozitorija. Prvo se poziva metoda *fetchStorageImages()* koja emitira dohvaćene podatke na tok, a zatim metoda *getStorageImages()* dohvati taj tok podataka, prikupi ga te vrijednost varijable *_imagesState* postavi na vrijednost prikupljenog toka podataka. Na ovom primjeru vidljiva je još jedna dobra programerska praksa , a to je jedna privatna promjenjiva varijabla kojoj je moguće mijenjati vrijednost unutar klase, te jedna nepromjenjiva javna varijabla koja samo prikazuje vrijednost promjenjive varijable. Na ovaj način osigurava se integritet podataka koji dođe do modela prikaza.

5.3. Postavljanje trenutne slike za anotiranje

Nakon što su dohvaćene slike koje čine generirani set slika spremnih za anotiranje potrebno ih je prikazati jednu po jednu kako bi ih korisnik mogao ispravno anotirati. Za to je zadužena metoda *setImage()* unutar aktivnosti za anotiranje. Nakon što je postavljen prilagođeni korutinski opseg za ovu aktivnost sigurno se može pristupiti ranije opisanoj vrijednosti varijable *imageState* iz modela prikaza i prikupiti njezin tok podataka. Nakon što je potvrđeno da je prikupljena nekakva vrijednost s navedenog toka u varijablu *imageName* se postavlja ime trenutne slike za anotiranje putem metode *getImageName*() iz modela prikaza. Ako se umjesto imena slike vrati vrijednost *null* to znači da je korisnik stigao do kraja generiranog seta te ga se vraća na početnu aktivnost aplikacije. Ako varijabla *imageName* sadrži vrijednost tada se dohvaća referenca te slike s

Firebase-ovog spremnika u oblaku, a sama slika se dohvaća u obliku privremenog dokumenta pozivom metode *getFile()* na njezinoj referenci. Ako se zadatak uspješno izvrši, odnosno dobijemo privremeni dokument u kojem su sadržani podaci slike, tada se podaci dekodiraju te se postavlja dobivena slika na zaslon kroz metodu *setImageBitmap()*. Implementacija metode *setImage()* prikazana je u programskom kodu 5.7.

```
private fun setImage() {
    mainScope.launch {
        annotationViewModel.imagesState.collect { images ->
            if (images.isNotEmpty()) {
                binding.annotationCanvasView.clearCanvas()
                val imageName = annotationViewModel.getImageName(images)
                if (imageName == null) {
                    Toast.makeText(binding.root.context,
getString(R.string.end_of_generated_set_message), Toast.LENGTH_SHORT).show()
                    val intent = Intent(binding.root.context,
MainActivity::class.java)
                    startActivity(intent)
                } else {
                    currentImage =
annotationViewModel.getCurrentImage(images)
                    val imageRef = storageRef.child(imageName)
                    val localFile = File.createTempFile(TEMP_IMAGE_PREFIX,
TEMP_IMAGE_SUFFIX)
                    imageRef.getFile(localFile).addOnSuccessListener {
                        val bitmap =
BitmapFactory.decodeFile(localFile.absolutePath)
                        binding.annotationImage.setImageBitmap(bitmap)
                    }.addOnFailureListener {
                        throw Exception("${it.message}")
                    }
                }
            }
        }
    }
}
```

Programski kod 5.7. Implementacija metode *setImage()*

Također, osim slike za anotiranje, potrebno je omogućiti i dvije različite klase anotiranja: vatru i dim. Ova funkcionalnost postignuta je postavljanjem dvije ikonice koje predstavljaju navedene klase na donju traku zaslona. Na programskom kodu 5.8. moguće je vidjeti implementaciju elemenata donje trake i rukovanje pritiscima na njih. U početku je postavljena klasa vatre a kasnije se ta vrijednost može mijenjati na dim odgovarajućim pritiskom.

```
private fun setBottomBar() {
    binding.bottomBar.selectedItemId = R.id.fireClass
    binding.bottomBar.setOnItemSelectedListener {
        val selectedObjectType: AnnotationCanvas.AnnotationObjectType
        when (it.itemId) {
            R.id.fireClass -> {
                selectedObjectType =
AnnotationCanvas.AnnotationObjectType.FIRE

binding.annotationCanvasView.updateObjectType(selectedObjectType)
```

```
                true
            }
            R.id.smokeClass -> {
                selectedObjectType =
AnnotationCanvas.AnnotationObjectType.SMOKE

binding.annotationCanvasView.updateObjectType(selectedObjectType)
                true
            }
            else -> false
        }
    }
}
```

Programski kod 5.8. Implementacija metode *setBottomBar()*

Na zaslonu ove aplikacije još se nalaze i gumbi za pomoć korisniku prilikom anotiranja kao što su: gumb za spremanje anotiranog oblika, gumb za poništavanje posljednje akcije, gumb za čišćenje platna za anotiranje te gumb koji, kada je korisnik gotov s anotiranjem, vodi do dijaloga gdje je potrebno pružiti dodatne informacije o slici. Konačni izgled aktivnosti za anotaciju na zaslonu može se vidjeti na slici 5.2.

Sl. 5.2. Snimka zaslona za anotiranje slike

5.4. Alati za anotiranje slika

Glavni dio praktičnog rada upravo je fokusiran na anotiranje slike. U ovom dijelu bit će opisani svi alati i funkcije koje su korištene kako bi korisniku bilo omogućeno ne smetano anotiranje. Za početak, kreiran je prilagođeni prikaz koji predstavlja platno po kojem je moguće postavljati točke po granicama objekta koji se označava. To je postignuto tako da klasa prilagođenog prikaza nasljeđuje od klase *View* kako bi dobila sve funkcionalnosti prikaza. Programski kod 5.9. prikazuje definiranje klase prilagođenog prikaza.

```
class AnnotationCanvas @JvmOverloads constructor(
    context: Context,
    attrs: AttributeSet? = null,
    defStyle: Int = 0,
) : View(context, attrs, defStyle)
```

Programski kod 5.9. Klasa *AnnotationCanvas*

Nakon kreiranja klase koja predstavlja platno za anotiranje, potrebno je taj prikaz postaviti preko slike za anotiranje pazeći da oba elementa budu na istom mjestu i istih dimenzija. To je postignuto korištenjem *FrameLayout* komponente unutar *xml* dokumenta aktivnosti te postavljanjem slike pa zatim prilagođeni prikaz unutar te komponente. Programski kod 5.10. prikazuje opisane komponente unutar *xml* datoteke.

```
<FrameLayout
    android:id="@+id/canvas_container"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:background="@color/white"
    app:layout_constraintBottom_toTopOf="@id/bottom_bar"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/save_shape">

    <ImageView
        android:id="@+id/annotation_image"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:contentDescription="@string/image_for_annotation"
        android:scaleType="fitXY"
        android:src="@android:color/transparent" />

    <com.domagojdragic.diplomskirad.view.customview.AnnotationCanvas
        android:id="@+id/annotation_canvas_view"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:background="@android:color/transparent" />

</FrameLayout>
```

Programski kod 5.10. Slika za anotiranje i prilagođeni prikaz unutar *xml* datoteke

Nakon što je prikaz uspješno postavljen preko slike za anotiranje potrebno je dati vlastite implementacije nekih metoda naslijeđenih od komponente prikaza. Prva takva metoda je

onTouchEvent() koja registrira korisnikov dodir na zaslon. Prilikom dodira zaslona moguće je dohvatiti x i y koordinate točke na zaslonu koja je pritisnuta te ju spremiti. Programski kod 5.11. prikazuje implementaciju opisane metode.

```
override fun onTouchEvent(event: MotionEvent?): Boolean {
    if (event != null) {
        when (event.action) {
            MotionEvent.ACTION_DOWN -> {
                if (annotationObjectType == AnnotationObjectType.FIRE) {
                    annotationPointsFire.add(PointF(event.x, event.y))
                }
                if (annotationObjectType == AnnotationObjectType.SMOKE) {
                    annotationPointsSmoke.add(PointF(event.x, event.y))
                }
                invalidate()
            }
        }
    }
    return true
}
```

Programski kod 5.11. Metoda *onTouchEvent()*

Kao što je moguće vidjeti na slici 5.11. prilikom detekcije dodira prvo se provjerava izabrana klasa anotiranja, vatra ili dim, te se zatim pritisnuta točka dodaje na odgovarajuću listu koja čuva sve točke pojedine klase. Nakon dodavanja nove točke poziva se metoda *invalidate()* koja obavještava prikaz da je došlo do nekakve promjene te da treba pozvati *onDraw()* metodu. U *onDraw()* metodi potrebno je implementirati sva crtanja po zaslonu koja se događaju. Programski kod 5.12. prikazuje implementaciju metode *onDraw()*, a zbog njezine kompleksnosti bit će podijeljena u više manjih metoda koje će biti objašnjene pojedinačno.

```
override fun onDraw(canvas: Canvas?) {
    super.onDraw(canvas)
    if (canvas == null) return

    drawFirePoints(canvas, firePaint)
    drawSmokePoints(canvas, smokePaint)
    drawFireLines(canvas, firePaint)
    drawSmokeLines(canvas, smokePaint)
    drawFireShape(canvas, firePaint)
    drawSmokeShape(canvas, smokePaint)

    if (saveShape) {
        when (annotationObjectType) {
            AnnotationObjectType.FIRE -> saveFireShape()
            AnnotationObjectType.SMOKE -> saveSmokeShape()
        }
    }
}
```

Programski kod 5.12. Metoda *onDraw()*

Prilikom poziva metode *onDraw()* redom se prvo iscrtavaju točke, zatim se te točke povezuju linija i konačno nakon spremanja oblika i prelaska na drugi iscrtava se spremljeni oblik. Također, ako

je potrebno spremiti neki oblik unutar ove metode se poziva odgovarajuća metoda. Svaka klasa anotiranja ima svoje metode za iscrtavanje, ali u principu, one rade na identičan način pa će biti objašnjeni primjeri metoda za iscrtavanje klase vatre. Prva metoda koja će biti objašnjena je *drawFirePoints()* a njezinu implementaciju moguće je vidjeti u programskom kodu 5.13. Kao parametre prima vrijednosti *canvas*, koji predstavlja platno po kojemu će se iscrtavati točke i *paint*, koji predstavlja boju i njezine atribute kojom se iscrtava. Za svaku točku s liste *annotationPointsFire* nad *canvas*-om se poziva metoda *drawPoint()* s odgovarajućim parametrima.

```
private fun drawFirePoints(canvas: Canvas, paint: Paint) {
    for (point in annotationPointsFire) {
        canvas.drawPoint(point.x, point.y, paint.apply { strokeWidth = 15f })
    }
}
```

Programski kod 5.13. Metoda *drawFirePoints()*

Nakon što se iscrta pojedina točka, na red dolazi metoda *drawFireLines()* koja prima iste parametre kao i prethodna metoda, a razlika je u tome što ova metoda prolazi kroz listu točaka, uzima dvije točke, prethodnu i sljedeću, te na temelju početnih i krajnjih koordinata iscrtava liniju između njih. Implementaciju ove metode moguće je vidjeti u programskom kodu 5.14.

```
private fun drawFireLines(canvas: Canvas, paint: Paint) {
    for (i in 1 until annotationPointsFire.size) {
        val startPoint = annotationPointsFire[i - 1]
        val endPoint = annotationPointsFire[i]
        canvas.drawLine(startPoint.x, startPoint.y, endPoint.x, endPoint.y,
paint.apply { strokeWidth = 8f })
    }
}
```

Programski kod 5.14. Metoda *drawFireLines()*

Metoda *drawFireShape()* također prima iste parametre kao i prethodne dvije metode i iako naizgled radi isti posao kao i metoda *drawFireLines()*, ova metoda je bitna zato što iscrtava spremljeni oblik i omogućava prelazak na anotiranje idućeg oblika na slici bez gubitka prethodno označenog oblika. Implementacija metode *drawFireShape()* prikazana je u programskom kodu 5.15.

```
private fun drawFireShape(canvas: Canvas, paint: Paint) {
    annotationFireShapes.forEach { points ->
        for (i in 1 until points.size) {
            val startPoint = points[i - 1]
            val endPoint = points[i]
            canvas.drawLine(startPoint.x, startPoint.y, endPoint.x,
endPoint.y, paint.apply { strokeWidth = 8f })
        }
    }
}
```

Programski kod 5.15. Metoda *drawFireLines()*

Posljednja metoda unutar *onDraw()* metode je *saveFireShape()*. Ova metoda poziva se kada korisnik završi anotiranje jednog od objekata na slici i želi nastaviti na novi objekt. Prvo se oblik s liste *annotationPointsFire* zatvara tako da se u listu dodaje kranja točka koja ima vrijednosti kao i početna točka. Zatim se ta lista sprema u *annotationFireShapes* i oslobađa. Na kraju metode se zastavica *saveShape* postavlja na vrijednost *false* i poziva se metoda *invalidate()*. Metoda *saveFireShape()* prikazana je u programskom kodu 5.16.

```
private fun saveFireShape() {
    annotationPointsFire.add(annotationPointsFire.first())
    annotationFireShapes.add(annotationPointsFire.toList())
    annotationPointsFire.clear()
    saveShape = false
    invalidate()
}
```

Programski kod 5.16. Metoda *saveFireShape()*

Osim metoda *onTouchEvent()* i *onDraw()* unutar ove klase postoje još neke metode koje korisniku olakšavaju proces anotiranja, a to su *undo()* i *clearCanvas()* koje se pozivaju pritiskom na odgovarajući gumb na zaslonu. Metoda *undo()* služi za poništavanje posljednje iscrtane točke dok metoda *clearCanvas()* potpuno briše sve iscrtane točke na zaslonu. Njihove implementacije prikazane su u programskom kodu 5.17.

```
fun clearCanvas() {
    annotationPointsFire.clear()
    annotationPointsSmoke.clear()
    annotationFireShapes.clear()
    annotationSmokeShapes.clear()
    invalidate()
}

fun undo() {
    if (annotationObjectType == AnnotationObjectType.FIRE) {
        if (annotationPointsFire.isEmpty()) return
        annotationPointsFire.removeLast()
    } else {
        if (annotationPointsSmoke.isEmpty()) return
        annotationPointsSmoke.removeLast()
    }
    invalidate()
}
```

Programski kod 5.17. Metode *clearCanvas()* i *undo()*

Nakon implementacije svih ovih metoda, aplikacija je dobila većinu potrebnih funkcionalnosti za ispravan rad. Slika 5.3. prikazuje snimku zaslona s nekoliko anotiranih objekata.

Sl. 5.3. Snimka zaslona sa anotiranim objektima vatre

5.5. Dijalog za dodatne informacije o slici

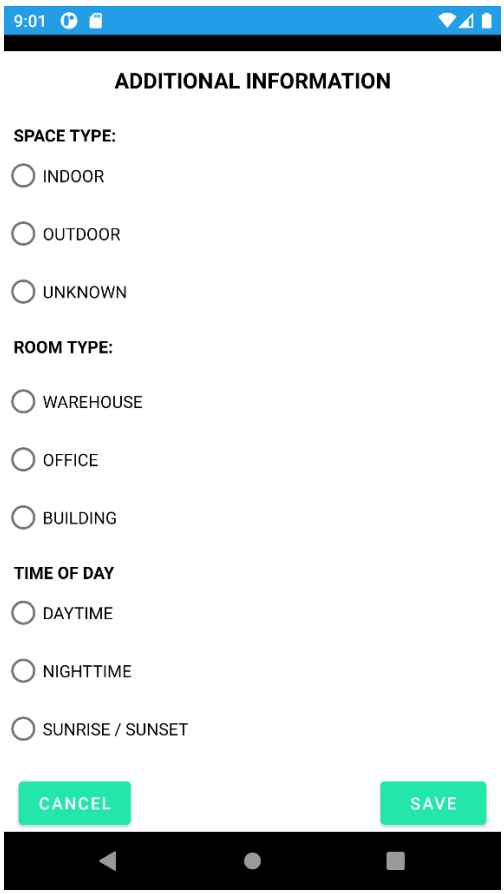
Nakon uspješnog anotiranja objekata na slici, korisnik treba pritisnuti gumb *Done* koji zatim prikazuje dijalog s različitim informacijama o slici koja je upravo anotirana. Dijalog se sastoji od više gumba za odabir koje je potrebno točno označiti u ovisnosti o tvrdnjama koje vrijede za konkretnu sliku. Primjer implementacije jedne grupe gumba za odabir prikazan je u programskom kodu 5.18.

```
<RadioGroup
    android:id="@+id/group_space_type"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@id/space_type">

    <RadioButton
        android:id="@+id/btn_indoor"
        android:layout_width="wrap_content"
```

```
        android:layout_height="wrap_content"
        android:text="@string/indoor"
        android:textSize="14sp" />

    <RadioButton
        android:id="@+id/btn_outdoor"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/outdoor"
        android:textSize="14sp" />

    <RadioButton
        android:id="@+id/btn_unknown_space_type"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/unknown"
        android:textSize="14sp" />

</RadioGroup>
```

Programski kod 5.18. Grupa gumba za odabir

Klasa koja predstavlja dijalog za dodatne informacije kreirana je tako što se nasljeđuje apstraktna klasa *BaseDialogView* kojoj u konstruktor predajemo vrijednost *ViewBindinga* od dijaloga. Ta apstraktna klasa pruža metodu *show()* koju je potrebno pozvati kada se želi prikazati dijalog. Sama klasa od dijaloga u sebi ima kontekst od aktivnosti u kojoj je potrebno prikazati dijalog i metodu *render()* kojom se pripremaju podaci za prikaz dijaloga. Dijalog je moguće vidjeti na slici 5.4.



Sl. 5.4. Snimka zaslona dijaloga s dodatnim informacijama

Dohvaćanje vrijednosti s ovih gumba za odabir radi se kroz pozivanje odgovarajuće metode pa tako na primjer za tip prostora poziva se metoda *getSpaceType()* s parametrom koji dohvaća ID gumba koji je odabran. Vrijednosti ovih gumba bit će potrebne prilikom spremanja informacija o slici s anotiranim objektima. Implementacija ove metode vidljiva je u programskom kodu 5.19.

```
private fun getSpaceType(buttonId: Int): String {
    return when (buttonId) {
        R.id.btn_indoor -> context.getString(R.string.indoor_save)
        R.id.btn_outdoor -> context.getString(R.string.outdoor_save)
        R.id.btn_unknown_space_type ->
context.getString(R.string.unknown_save)
        else -> context.getString(R.string.not_selected)
    }
}
```

Programski kod 5.19. Implementacija metode *getSpaceType()*

5.6. Spremanje rezultata anotiranja u bazu podataka

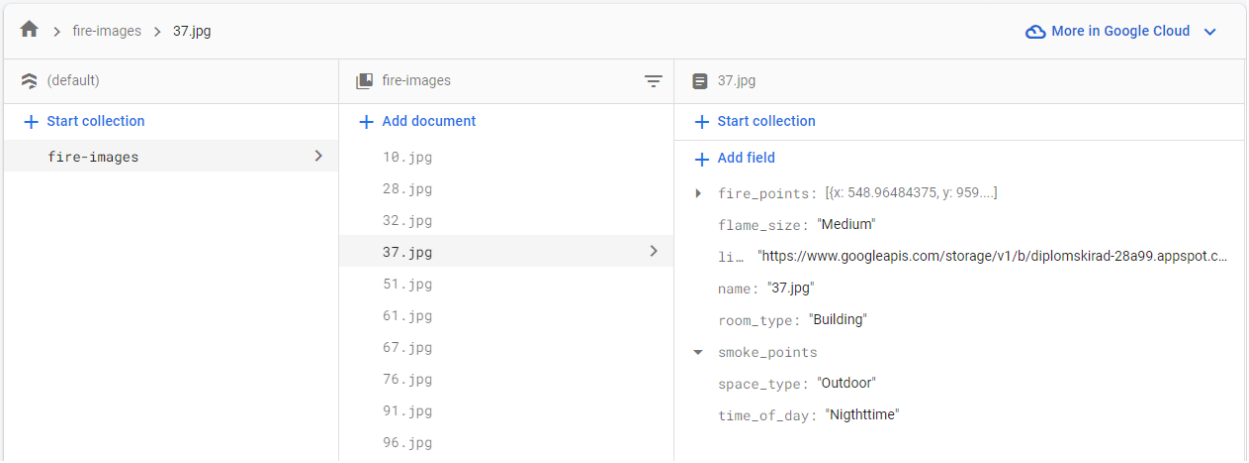
Nakon što su prikupljeni podaci o poziciji objekata vatre i dima na slici te dodatne informacije o slici koja se anotira potrebno je još pripremiti te podatke za spremanje u bazu podataka *Firestore*. Kao i svaka Firebase usluga za početak je potrebno dohvatiti referencu baze podataka kao i mjesto gdje će se pojedina anotacija spremiti. Nakon dohvaćanja instance baze na njoj se poziva metoda *document()* unutar koje se predaje ime kolekcije unutar koje se sprema poseban dokument za svaku sliku koja se anotira. Ime tog dokumenta također se definira unutar ove metode. Budući da se podaci spremaju u dokumente potrebno je pripremiti sve vrijednosti koje se žele spremiti u *HashMap* kolekciju. Svaki podatak spremamo u kolekciju tako da prvo definiramo ključ koji će označavati tu vrijednost a zatim i objekt koji se sprema. Kada su svi podaci pripremljeni i unutar *HashMap* kolekcije poziva se metoda set() nad instancom *Firestore* baze podataka u koju se predaje navedena kolekcija. Nakon što je spremanje u bazu uspješno odrađeno ostalo je još samo promijeniti zastavicu *isComplete* od trenutne slike na vrijednost *true*, pozvati akciju onSave() koja će zapravo ponovno pozvati metodu *setImage*() koja će postaviti novu sliku za anotiranje te zatvoriti dijalog. Sve ovo opisano moguće je vidjeti u programskom kodu 5.20. koji prikazuje implementaciju događaja prilikom pritiska na gumb *Save*.

```
val firestoreDB =
FirebaseFirestore.getInstance().document("$DOCUMENT_NAME/$imageName")
val dataToSave = HashMap<String, Any>()

dataToSave[NAME] = imageName
dataToSave[LINK] = currentImage.selfLink
dataToSave[FIRE_POINTS] = fireShapes.flatten()
dataToSave[SMOKE_POINTS] = smokeShapes.flatten()
dataToSave[SPACE_TYPE] = spaceType
dataToSave[ROOM_TYPE] = roomType
```

```
dataToSave[TIME_OF_DAY] = timeOfDay
dataToSave[FLAME_SIZE] = flameSize

firestoreDB.set(dataToSave).addOnSuccessListener {
    viewModel.updateIsComplete(currentImage)
    onSave()
    popupWindow.dismiss()
    Toast.makeText(context,
context.getString(R.string.firestore_save_success_message),
Toast.LENGTH_SHORT).show()
}.addOnFailureListener {
    popupWindow.dismiss()
    Toast.makeText(context,
context.getString(R.string.firestore_save_failure_message),
Toast.LENGTH_SHORT).show()
}
```

Programski kod 5.19. Implementacija metode *setOnClickListener* gumba *Save*

Konačno, nakon anotiranja cijelog generiranog seta od 10 slika i spremanja istih u bazu podataka korisnika se odvodi na početnu aktivnost gdje može odabrati generiranje seta od novih 10 slika. Nakon anotiranja, izgled popunjenog sučelja baze podataka može se vidjeti na slici 5.5.



Sl. 5.5. Popunjeno sučelje baze podataka *Firestore*

6. ZAKLJUČAK

U posljednje vrijeme sve se više radi na razvijanju pametnih rješenja za rano otkrivanje i prevenciju požara. Ako se na vrijeme ne spriječi, požar može uzrokovati katastrofalne ekonomske i ekološke posljedice pa čak i ugroziti ljudski život. Jedno od mogućih rješenja ovog problema je razvijanje pametnog sustava koji na temelju računalnog vida prepoznaje vatru i dim te sam pristupa sprečavanju potencijalnog požara. Kako bi jedan takav sustav mogao ispravno prepoznavati vatru i dim potrebno je iznimno dobro istrenirati model strojnog učenja na primjeru anotiranih slika vatre i požara. Upravo iz tog razloga cilj ovog diplomskog rada bio je proučiti probleme s kojima se jedan takav pametni sustav može susresti te predložiti rješenje za anotiranje slika u obliku mobilne Android aplikacije.

Proučeno je teorijsko područje vezano uz strojno učenje, segmentaciju slika, načine treniranja modela strojnog učenja te sam proces anotiranja slika. Nakon provedenih usporedbi prednosti i nedostataka navedenih područja odlučeno je da će se proces anotiranja slika provoditi na način segmentacije instanci te da će se za potrebe treniranja modela strojnog učenja koristiti model konvolucijskih neuronskih mreža, odnosno U-Net arhitektura modela konkretno.

Detaljnom analizom ponuda na tržištu i postojećih programskih rješenja osmišljena je aplikacija koja rješava navedeni problem i omogućuje korisniku anotiranje slika koje će se kasnije koristiti u svrhe treniranja modela strojnog učenja da prepoznaje požar i shodno tome može reagirati. Prethodno implementiranju programskog rješenja postavljeni su funkcionalni zahtjevi na aplikaciju kao i dijagram tijeka aplikacije koji će naknadno služiti za verifikaciju svih komponenata aplikacije te njezinog ispravnog rada.

Za razvoj aplikacije izabrano je programsko okruženje Android Studio te programski jezik Kotlin, a za verzioniranje koda izabran je GitHub servis. Proučavanjem teorije vezane uz ovo područje rada izabrane su programske tehnologije i alati koji će pomoći u realizaciji same aplikacije. Za potrebe upravljanja korisnicima korištena je Firebase usluga za autentifikaciju korisnika, skup slika na kojima se nalaze vatra i dim sačuvan je na Firebase *Storage* usluzi dok se rezultati anotiranja zapisuju u NoSQL bazu podataka u oblaku Firebase *Firestore*. Cijela aplikacija razvijena je prateći MVVM arhitekturu izrade mobilnih aplikacija.

Rezultat praktičnog dijela ovoga rada je mobilna Android aplikacija koja korisniku nakon prijave u sustav generira set od deset slika na kojima se nalaze vatra i dim. Korisnik tada započinje proces anotiranja svake slike i nakon što je zadovoljan anotiranim pruža dodatne informacije o slici kao

što su tip prostora koji je prikazan na slici, doba dana u kojem je slika nastala te o veličini plamena koji je prikazan na slici. Rezultati korisnikovog anotiranja zatim se spremaju u vanjsku bazu podataka. Potencijalno proširenje ove aplikacije obuhvaćalo bi dodavanje većeg broja novih slika vatre i dima u postojeći spremnik. Ovo proširenje vrlo je jednostavno za implementirati budući da Firebase Storage usluga dopušta naknadno dodavanje novih podataka u spremnik. Drugo moguće proširenje aplikacije bilo bi u vidu dodavanja novih i boljih alata za anotiranje. U ovom slučaju trebalo bi zamijeniti postojeće alate novima u kodu.

LITERATURA

[1] Thou-Ho Chen, Cheng-Liang Kao, Sju-Ma Chang, „An intelligent real-time fire-detection method based on video processing “, *IEEE 37th Annual 2003 International Carnahan Conference onSecurity Technology, 2003. Proceedings.*, Taipei, Taiwan: IEEE, 2003, str. 104–111. doi: 10.1109/CCST.2003.1297544.

[2] Petar Ćurković, dipl. ing. Tomislav Stipančić, dipl. ing., Segmentacija slike, dostupno na: https://slideplayer.gr/slide/14907603/91/images/7/Tresholding-odre%C4%91ivanje+praga.jpg [19.6.2023.]

[3] Wankai Deng, Wei Xiao, He Deng, Jianguo Liu, MRI brain tumor segmentation with region growing method based on the gradients and variances along and inside of the boundary curve, 2010, dostupno na: https://www.semanticscholar.org/paper/MRI-brain-tumor-segmentation-with-region-growing-on-Deng-Xiao/879fc413ebc138db1121fa5ab93b230205a4c1a6/figure/1 [19.6.2023.]

[4] K. Muhammad, J. Ahmad, Z. Lv, P. Bellavista, P. Yang, S. W. Baik, „Efficient Deep CNN-Based Fire Detection and Localization in Video Surveillance Applications “, *IEEE Trans. Syst. Man Cybern, Syst.*, sv. 49, izd. 7, str. 1419–1434, srp. 2019, doi: 10.1109/TSMC.2018.2830099.

[5] K. Džomba, "Konvolucijske neuronske mreže", Diplomski rad, Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet, Zagreb, 2018. Dostupno na: https://urn.nsk.hr/urn:nbn:hr:217:020017

[6] Y. Lecun, L. Bottou, Y. Bengio, P. Haffner, „Gradient-based learning applied to document recognition “, *Proc. IEEE*, sv. 86, izd. 11, str. 2278–2324, stu. 1998, doi: 10.1109/5.726791.

[7] O. Ronneberger, P. Fischer, T. Brox, „U-Net: Convolutional Networks for Biomedical Image Segmentation “, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, N. Navab, J. Hornegger, W. M. Wells, A. F. Frangi, Ur., u Lecture Notes in Computer Science, vol. 9351. Cham: Springer International Publishing, 2015, str. 234–241. doi: 10.1007/978-3-319-24574-4_28.

[8] C. O’Sullivan, „U-Net Explained: Understanding its Image Segmentation Architecture “, Towards Data Science, ožu. 2023., dostupno na: https://towardsdatascience.com/u-net-explained-understanding-its-image-segmentation-architecture-56e4842e313a [19.6.2023.]

[9] J. Solawetz, „How to Train a Custom Object Detection Model with YOLO v5 “, Towards Data Science, lip. 2023., dostupno na: https://towardsdatascience.com/how-to-train-a-custom-object-detection-model-with-yolo-v5-917e9ce13208 [20.6.2023.]

[10] Image Annotation for Computer Vision, Cloud Factory, dostupno na: https://www.cloudfactory.com/image-annotation-guide [20.6.2023.]

[11] RHS Apps, Annotate - Image Annotation To, dostupno na: https://play.google.com/store/apps/details?id=com.annotate.image [20.6.2023.]

[12] Winteroso Team, iMarkup: Text, Draw on photos, dostupno na: https://play.google.com/store/apps/details?id=com.winterso.markup.annotable [20.6.2023.]

[13] A. Rizzoli, 13 Best Image Annotation Tools of 2023 [Reviewed], V7, velj. 2023., dostupno na: https://www.v7labs.com/blog/best-image-annotation-tools [20.6.2023.]

[14] G2, V7 Reviews & Product Details, dostupno na: https://www.g2.com/products/v7/reviews [20.6.2023.]

[15] Wikipedia, V7-Darwin, dostupno na: https://upload.wikimedia.org/wikipedia/en/4/46/V7-darwin-annotation-interface.jpg [20.6.2023.]

[16] A. Torralba, B. C. Russell, J. Yuen, „LabelMe: Online Image Annotation and Applications “, *Proc. IEEE*, sv. 98, izd. 8, str. 1467–1484, kol. 2010, doi: 10.1109/JPROC.2010.2050290.

[17] H. Bandyopadhyay, Labeling with LabelMe: Step-by-step Guide, dostupno na: https://www.v7labs.com/blog/labelme-guide [20.6.2023.]

[18] E. Mixon, Android OS, velj. 2023., dostupno na: https://www.techtarget.com/searchmobilecomputing/definition/Android-OS [21.6.2023.]

[19] StatCounter, Mobile & Tablet Android Version Market Share Worldwide , dostupno na: https://gs.statcounter.com/android-version-market-share/mobile-tablet/worldwide/#monthly-202205-202305 [21.6.2023.]

[20] Android developers, Meet Android Studio, dostupno na: https://developer.android.com/studio/intro [21.6.2023.]

[21] IntelliJ IDEA, Why IntelliJ IDEA, dostupno na: https://www.jetbrains.com/idea/ [21.6.2023.]

[22] Get started with Kotlin, lip. 2023., dostupno na: https://kotlinlang.org/docs/getting-started.html#install-kotlin [22.6.2023.]

[23] Android developers, Android's Kotlin-first approach, dostupno na: https://developer.android.com/kotlin/first [22.6.2023.]

[24] Android developers, Develop Android apps with Kotlin, dostupno na: https://developer.android.com/kotlin [28.6.2023.]

[25] N. Mc Cool, What is GitHub?, dostupno na: https://codeinstitute.net/global/blog/github-might-benefit-using/ [22.6.2023.]

[26] CodeClouds, The Advantages and Disadvantages of Using GitHub, stu. 2021., dostupno na: https://www.codeclouds.com/blog/advantages-disadvantages-using-github/ [22.6.2023.]

[27] StarUML programski paket, dostupno na: https://staruml.io/ [22.6.2023.]

[28] K. T. Hanna, Google Firebase, svi. 2023., dostupno na: https://www.techtarget.com/searchmobilecomputing/definition/Google-Firebase [23.6.2023.]

[29] H. Fatima, Why is Firebase the best Mobile Backend-as-a-Service, pro. 2017., dostupno na: https://blog.resellerclub.com/why-is-firebase-the-best-mobile-backend-as-a-service/ [23.6.2023]

[30] Firebase documentation, Firebase Authentication, lip. 2023., dostupno na: https://firebase.google.com/docs/auth [24.6.2023.]

[31] Firebase documentation, Cloud Storage for Firebase, lip. 2023., dostupno na: https://firebase.google.com/docs/storage [24.6.2023.]

[32] Firebase documentation, Cloud Firestore, lip. 2023., dostupno na: https://firebase.google.com/docs/firestore [25.6.2023.]

[33] E. Garcia Gallardo, What is MVVM Architecture?, sij. 2023., dostupno na: https://builtin.com/software-engineering-perspectives/mvvm-architecture [26.6.2023.]

[34] Android developers, Kotlin flows on Android, ožu. 2023., dostupno na: https://developer.android.com/kotlin/flow [26.6.2023.]

[35] Android developers, Kotlin coroutines on Android, ožu. 2023., dostupno na: https://developer.android.com/kotlin/coroutines [26.6.2023.]

[36] What is Koin?, 2023., dostupno na: https://insert-koin.io/docs/reference/introduction [21.8.2023.]

SAŽETAK

Razvojem napredne tehnologije sve češća je pojava pametnih sustava za ranu detekciju požara. Kako bi takvi sustavi mogli ispravno obavljati svoj posao potrebno ih je prethodno naučiti kako razlikovati vatru i dim od ostatke okoline te kako prepoznati rani požar. Cilj ovog rada bio je upoznati se s metodama strojnog učenja koja omogućavaju rješavanje ovog problema te razviti aplikaciju koja će poslužiti kao alat za anotiranje testnih slika koje će se kasnije koristiti upravo u svrhe treniranje tog modela. Korištenjem predložene arhitekture i pomoćnih alata razvijena je mobilna aplikacija za Android operacijski sustav. Nakon razvoja aplikacije verificirane su sve njezine komponente i ispravnost istih usporedbom s funkcionalnim zahtjevima koji su ranije raspisani.

Ključne riječi: Android, anotiranje, strojno učenje, vatra

ABSTRACT

MOBILE APPLICATION FOR IMAGE ANNOTATION

With the development of advanced technology, the appearance of smart systems for early fire detection is becoming more and more common. In order for such systems to be able to perform their work correctly, they must first be taught how to distinguish fire and smoke from the rest of the environment and how to recognize an early fire. The goal of this paper was to learn about machine learning methods that enable solving this problem and to develop an application that will serve as a tool for annotating test images that will later be used for the purpose of training that model. Using the proposed architecture and utility tools, a mobile application for the Android operating system was developed. After the development of the application, all of its components and their integrity were verified by comparing them with the functional requirements that were written out earlier.

Keywords: Android, annotation, fire, machine learning

ŽIVOTOPIS

Domagoj Dragić rođen je 4.12.1999. godine u Đakovu. Završio je Osnovnu školu Vladimir Nazor u Đakovu i opću gimnaziju Antun Gustav Matoš u Đakovu. Po završetku srednje škole upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija u Osijeku, smjer Računarstvo na preddiplomskom sveučilišnom studiju, koji završava tri godine kasnije, 2021. godine. Odmah nakon završetka preddiplomskog studija, na istom fakultetu upisuje diplomski studij, smjer Programsko inženjerstvo, koji trenutno pohađa. Poznaje engleski i njemački jezik, a od programskih jezika najbolje poznaje Kotlin. Najviše voli izrađivati mobilne Android aplikacije.

Domagoj Dragić