# Mobilna Android aplikacija za upravljanje performansama nogometnog kluba zasnovana na predlošku programske arhitekture MVVM

**Lukac, Ivan**

**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:200:498286





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

**Sveučilišni diplomski studij**

# MOBILNA ANDROID APLIKACIJA ZA UPRAVLJANJE PERFORMANSAMA NOGOMETNOG KLUBA ZASNOVANA NA PREDLOŠKU PROGRAMSKE ARHITEKTURE MVVM

**Diplomski rad**

**Ivan Lukac**

**Osijek, 2023.**

# SADRŽAJ

# 1. UVOD

Tehnologija je davno zakoračila u sport, no svakim danom sve više i više prodire u svaku poru sporta. Pomoću tehnologije sportaši ostvaruju sve bolje rezultate te se ljestvica izvrsnosti stalno podiže. Tehnologiju primarno koriste treneri sportaša kako bi uvidjeli u kojim dijelovima mogu poboljšati performanse igrača te samim time i rezultate koje ostvaruju, kroz praćenje fizičkih performansi igrača pri naporu, pregledavanje snimki suparničkog tima i proučavanja njihove statistike ili na neki drugi način. Kao što treneri koriste tehnologiju za postizanje boljih rezultata svog tima, tako i igrači koriste tehnologiju kako bi poboljšali svoje karakteristike, fizičke, psihičke ili taktičke, za boljitak svog razvoja.

Glavni zadatak ovog diplomskog rada je osmisliti i implementirati mobilnu Android aplikaciju za upravljanje nogometnim klubom. Tu aplikaciju bi mogli koristiti treneri kako bi poboljšali rezultate te performanse svojih igrača i cijelog tima. Budući da se u razvoju mobilnih aplikacija koristi arhitektura za strukturiran i uredan programski kod, u ovom radu će se na temelju arhitekture *Model-View-ViewModel* (MVVM) stvoriti mobilna aplikacija za upravljanje performansama nogometnog kluba. Prikazat će se kako se implementira MVVM arhitektura, razlozi njenog korištenja u razvoju Android aplikacija, koje su prednosti i nedostaci tog pristupa, te na kraju implementacija i ispitivanje rada aplikacije.

U drugom poglavlju rada definirane su performanse nogometnog kluba, izazovi vezani uz performanse nogometnog kluba te trenutna potpora računalnih tehnologija za upravljanje nogometnim klubom i performansama igrača. Prikazana su i postojeća slična rješenja na platformi Google Play. Trećim poglavljem definirano je rješenje mobilne aplikacije te su navedeni svi funkcionalni i nefunkcionalni zahtjevi na aplikaciju. Opisana je arhitektura MVVM koja će se koristiti u aplikaciji. U četvrtom poglavlju prikazana je programska implementacija aplikacije za upravljanje performansama nogometnog kluba, te tehnologije koje su korištene u razvoju mobilne aplikacije. Opisano je programsko rješenje s korisničke i poslužiteljske strane. Peto poglavlje prikazuje način korištenja aplikacije, te analizira rad aplikacije i testira funkcionalnosti aplikacije pri slučajevima korištenja aplikacije.

# 2. POTPORA RAČUNALNIH TEHNOLOGIJA UPRAVLJANJU PERFORMANSAMA NOGOMETNOG KLUBA I STANJE U PODRUČJU

Živimo u vremenu u kojemu se tehnologija pokušava primijeniti u sve grane života kako bi si olakšali djelovanje u toj grani. Sport nije iznimka, već se toliko novih tehnologija primijenilo u rad sa sportašima, praćenje njihovih nastupa na natjecanjima, pa čak i na suđenje natjecanja, kao što je na primjer video asistent poznatiji kao VAR (eng. *Video Assistant Referee*). Budući da svi igrači i treneri imaju mobilni uređaj uz sebe veliki dio dana, razvoj mobilne aplikacije kojima igrači i treneri mogu pratiti svoj napredak i svoju statistiku je nešto što bi moglo biti često korišteno i korisno svima koji koriste tu mobilnu aplikaciju. Razvojem mobilne aplikacije koja prati performanse igrača i tima te prikazuje eventualna poboljšanja u performansama, pridonosi se razvoju igrača i boljim rezultatima klubova u kojima igraju.

## 2.1. Performanse nogometnog kluba

Performanse nogometnog kluba su performanse njegovog trenerskog i igračkog kadra. Performanse kojima se opisuju igrači su njihove psihičke i fizičke vještine koje koriste na treningu ili nogometnoj utakmici kako bi pomogli svojoj ekipi da pobijedi. Prema [1], trenutno na svijetu postoji nekoliko tisuća profesionalnih nogometnih klubova te više od deset tisuća profesionalnih nogometaša. To znači da je konkurencija ogromna i da igrači žele biti što bolji. Igrači i treneri konstantno teže prema tome da poboljšaju svoje trenutne karakteristike i vještine. Svoje karakteristike i vještine mogu poboljšati treningom. Prema [2], trening je jedan od najbitnijih, a možda i najbitniji čimbenik u razvoju igrača. Svi problemi koje igrač ima, bili oni vezani uz tehniku ili igračeve fizičke ili psihičke karakteristike, rješavaju se pomoću treninga. Što se tiče fizičkih performansi igrača, one se poboljšavaju kondicijskom pripremom igrača. Igrač može napraviti kondicijsku pripremu individualnim treningom gdje je vođen od strane kondicijskog trenera koji je dubinskom analizom napravio trening koji će pogađati elemente fizičkog razvoja koje igrač mora popraviti. Taktičke performanse igrača se povećavaju grupnim treningom s ekipom gdje glavnu ulogu ima trener i njegovi pomoćnici koji igračima pokazuju svoje taktičke zamisli i postupke kako se ponašati na određenu postavku druge ekipe. Igrač također može raditi na svojim tehničkim sposobnostima uz pomoć trenera specijaliziranih za individualni rad, ali igrač se najčešće posvećuje poboljšanju svojih tehničkih vještina u pauzi između dvije sezone zato što tada ima najviše vremena i tijekom sezone kada veći prioritet ima taktički i kondicijski trening nad tehničkim treningom.

Kao što se performanse poboljšavaju treningom, tako se i manjkom istog mogu i pogoršati s vremenom. Igračima su njihove performanse jako bitne zato što utječu na rezultat njihovog tima

te utječu na njihovu plaću u klubu. Ukoliko igrač ima najveće performanse u klubu postoji velika mogućnost da će mu drugi klub dati veću plaću kako bi došao igrati kod njih sa većom konkurencijom i kako bi nastavio svoj igrački rast. Prema [3], vrijednost transfera može se odrediti na temelju trenutnih performansi igrača, ali i na temelju potencijalnog rasta performansi tog igrača. Klubu je isplativije platiti manje novaca za transfer mladog talentiranog igrača koji će s vremenom napredovati gdje će ga klub opet moći prodati nekom drugom za veće novce te zaraditi.

## 2.2. Izazovi upravljanja performansama nogometnog kluba, potpora računalnih tehnologija i stanje u području

U profesionalnom sportu, sportaši su u velikom broju slučajeva mladi ljudi koji su upoznati s tehnologijom te svi imaju pametni telefon kojim se vješto služe. Igračima je svaka nova tehnologija uvedena u klub novo osvježenje koje će oni lako savladati, a imat će velike koristi od nje. Dok su igrači većinom mladi ljudi u rasponu od 18-40 godina starosti, treneri, menadžeri i ljudi iz uprave kluba su najčešće iskusniji ljudi iznad 50 godina starosti. Ljudima u tim godinama se nije toliko lagano priviknuti na novu tehnologiju te im treba u velikom broju slučajeva puno više vremena za naučiti raditi s novom tehnologijom. Velika većina sportskih trenera sve podatke koje koriste spremaju u nepregledne tablice ili čak pišu u svoje bilježnice koje kasnije mogu izgubiti i svi podaci vezane za performanse igrača i tima su izgubljene. Analiza tih podataka je na taj način izuzetno teška, ali uz pomoć računalnih tehnologija ta analiza se može znatno skratiti i ljepše prikazati. Razvojem programskih rješenja koji bi podatke o performansama igrača, tima i kluba prikazao i vizualizirao na način gdje bi treneru sve bilo jasno vidljivo bez gledanja u razne tablice nego samo pogledom u svoj mobilni uređaj sve bi mu bilo jasno vezano za određenog igrača ili tim. Tako bi trener mogao nakon odrađenog treninga, uz par klikova na svom mobilnom uređaju pregledati stanje cijele ekipe i to obaviti u nekoliko minuta, a opet saznati sve potrebne informacije. Jedan oblik korištenja tehnologije u vrhunskom sportu koji se pokazao kao vrlo dobar je analiza videozapisa utakmice. Bila to utakmica protivničke ekipe ili svoje, može se saznati puno podataka o ekipi iz videozapisa i kvalitetno pripremiti za sljedeću utakmicu. Tehnologija značajno pomaže svim igračima i trenerima koji je koriste radi poboljšanja svojih performansi. Bez primjene tehnologije je iznimno teško ostati konkurentan u natjecanju ili ligi. Prema [4], performanse igrača se mogu predvidjeti i poboljšati pomoću strujnog učenja ili dubokog učenja na temelju prikupljenih podataka o igračima. Veliki broj klubova u svijetu koristi strojno učenje kako bi pokušali predvidjeti buduću vrijednost svojih igrača. Također, sportaši mogu nositi tehnologiju poput pametnih satova ili prsluka pomoću kojih se njihovi zdravstvene vrijednosti u naporu i odmoru bilježe te se prema njima oblikuje njihov trening.

## 2.3. Postojeća slična programska rješenja

U ovom dijelu rada će biti prikazana slična već postojeća rješenja na platformi za Android aplikacije Google Play. Google Play nije jedini način za instalirati mobilnu android aplikaciju, ali je najčešći. Prema [5], Google Play je nastao 2012. godine spajanjem Android Marketa, Google glazbe i Google trgovine e-knjigama. U svrhu pronalaženja sličnih aplikacija pretraživani su pojmovi vezani za upravljanje performansama sportskog kluba i karakteristikama igrača.

### 2.3.1. Stanje u području programskih rješenja

Pogledom na trgovinu aplikacijama Google Play trenutno stanje govori da postoji veliki broj aplikacija koje su napravljene kao igrice gdje korisnik upravlja svojim nogometnim klubom, trenira svoje igrače, igra utakmice, radi transfere, itd. Iako postoji veliki broj igrica koje oponašaju upravljanje nogometnim klubom i performansama nogometnog kluba, trenutno ne postoji na Google Play platformi aplikacija koja nudi sučelje gdje možete upisati potrebne podatke o svojim igračima i gdje će to biti prikladno vizualizirano. Kao što je već navedeno, to što takvih aplikacija baš i nema na Google Play platformi, ne znače da one ne postoje. Klubovi se mogu dogovoriti s agencijom da se samo za njih napravi takva aplikacija koju će onda oni koristiti u svome klubu. Također, moguće je da drugačije rješenje postoji i na *App Storeu*, koji je platforma za objavljivanje aplikacija za iPhone uređaje, odnosno uređaje s operacijskim sustavom iOS. Na javnim platformama ne postoji aplikacija ovog tipa, budući da klubovi traže ovu aplikaciju samo za sebe kako bi se odvojili od konkurencije. Tvorci sličnih aplikacija su najčešće agencije koje potpišu ugovor s klubom o izradi, zatim agencija napravi točno ono što klub zatraži da bude u aplikaciji što znači da je aplikacija napravljena točno za njih i njihove potrebe što joj daje dodatnu vrijednosti. Veliki broj pronađenih aplikacija podržava sustav ocjenjivanja igrača za pojedinu kategoriju, opciju transferiranja igrača i simulaciju igranja utakmica. U pronađenim aplikacijama se nudi opcija poboljšanja performansi igrača tako da korisnik svakodnevno ulazi u aplikaciju i odrađuje treninge s igračima. Tim postupkom se poboljšavaju performanse igrača. Također, u nekim aplikacijama nudi se opcija plaćanja dodatnim sredstvima kako bi se poboljšala kvaliteta korisnikovih igrača.

### 2.3.2. Postojeća slična rješenja

#### 2.3.2.1. Box-to-box: Nogometni trening

Box-to-box je mobilna Android aplikacija za nogometni trening koja služi igraču za poboljšanje nogometnih vještina bez potrebe rada s trenerom. Vježbe u Box-to-box aplikaciji je osmislio trener

specijaliziran za indvidualni rad. Također, u aplikaciji je moguće upisati svoje trenutne performanse u velikom broju kategorija te pratiti napredak s vremenom. Moguće je prilagoditi plan treninga svojoj poziciji te pratiti veliki broj statistika za vrijeme odrađenog treninga. Dijelovi aplikacije su prikazani na slikama 2.1 i 2.2.



**Slika 2.1** Prikaz početnog zaslona aplikacije

**Slika 2.2.** Prikaz zaslona s karakteristikama igrača

#### 2.3.2.2. Top Eleven Be Football Manager

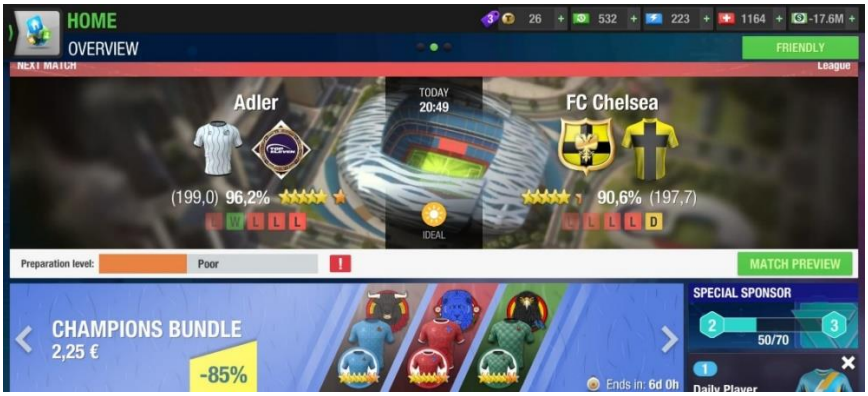
Top Eleven je mobilna aplikacija koja služi za simuliranje vođenja nogometnog kluba. Napravila ju je tvrtka Nordeus. Ova aplikacija omogućuje svojim korisnicima da upravljaju svojim timom i igračima u timu. Moguće je treningom povećavati performanse igrača kako bi im se povećala vrijednost te kako bi postizali bolje rezultate na utakmicama kluba. Top Eleven također omogućuje postavljanje taktike te korištenje različitih metoda za poboljšanje morala igrača. Ova vrlo popularna mobilna aplikacija ima više od 100 milijuna preuzimanja te ukupnu ocjenu 4.6. Izgled aplikacije je vidljiv na slikama 2.3 i 2.4.

#### 2.3.2.1. Fifa Football

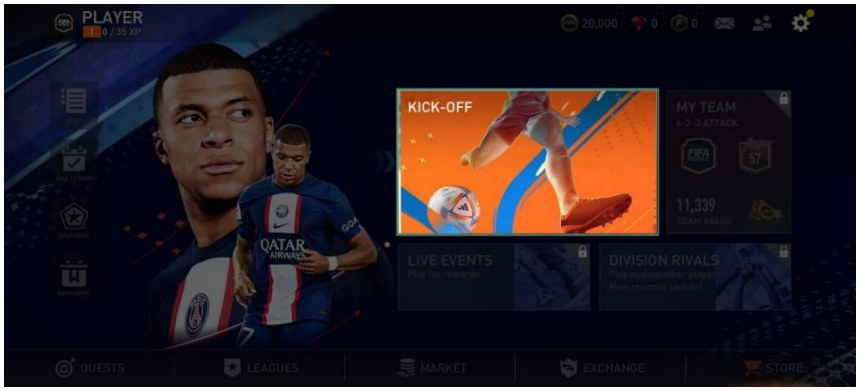
Fifa Football je računalna igrica dostupna za stolna računala i mobilne uređaje. U ovom dijelu će biti opisana mobilna inačica igre. Napravila ju je tvrtka Electronic Arts te svoju popularnost ističe u brojkama preuzimanja koje su veće od 100 milijuna preuzimanja te ukupnom ocjenom 4.3. Fifa

Football korisniku omogućava vođenje svog kluba i svojih igrača u svrhu osvajanja trofeja i postizanja dobrih rezultata. Moguće je kupovati i prodavati igrače, trenirati ih, igrati utakmice s njima te još veliki broj različith mogućnosti. Pregled mobilne igre vidljiv je na slikama 2.5 i 2.6.

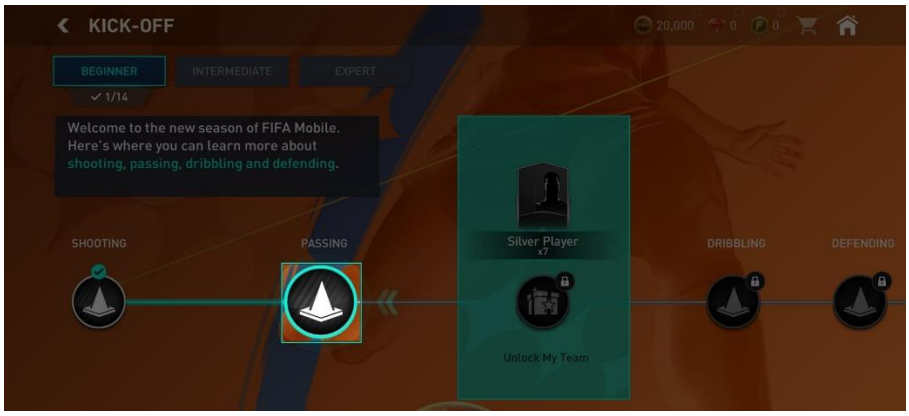

**Slika 2.3**. Prikaz taktičke formacije momčadi




**Slika 2.4.** Prikaz početnog zaslona aplikacije




**Slika 2.5.** Prikaz odabira odigravanja utakmice




**Slika 2.6.** Prikaz načina treniranja

U prethodnom dijelu poglavlja opisane su tri različite aplikacije. Zajedničko im je da su vrlo popularne, imaju veliki broj različitih značajki koji pridonose iskustvu korisnika u igri. U opisanim aplikacijama moguće je upravljati igračima i povećavati njihove performanse treningom. Također je moguće praćenje njihovih trenutnih performansi za niz određenih kategorija te još razne mogućnosti poput gradnje stadiona, transferiranje igrača, itd. Navedene aplikacije su s razlogom postigle veliki uspjeh, ali nisu primjenjive u stvarnom svijetu. Za razliku od njih, aplikacija u ovom radu će biti korištena u stvarnom svijetu za boljitak kluba koji je koristi.

# 3. MODEL I ARHITEKTURA APLIKACIJE ZA UPRAVLJANJE PERFORMANSAMA NOGOMETNOG KLUBA

U ovome dijelu rada će biti opisani funkcijski i nefunkcijski zahtjevi na mobilnu aplikaciju, arhitektura koja će biti implementirana u aplikaciji i detaljne funkcionalnosti i postupci unutar aplikacije. Funkcionalnosti će biti prikazane na temelju dizajna aplikacije koji je napravljen prije njih.

## 3.1. Funkcijski i nefunkcijski zahtjevi na mobilnu aplikaciju za upravljanje performansama nogometnog kluba

Kako bi se lakše shvatila bit i rad aplikacije u ovom poglavlju će se pojasniti funkcionalnosti aplikacije te uz prikaz dijagrama korištenja aplikacije kako bi čitatelju rad ove aplikacije bio potpuno jasan. Ova aplikacija je namijenjena za trenere nogometnih ekipa primarno, a mogu je koristiti i igrači. Kada korisnik pokrene aplikaciju prvo mu se prikazuje zaslon s anketom o posljednjem treningu gdje odgovara na pitanja o koncentraciji tima na treningu, trudu igrača, vrsti treninga te trener može dodati svoje bilješke ako želi. Na dnu prvog zaslona su prikazana dva gumba, jedan za spremanje treninga, drugi za odustajanje od spremanja. Ako korisnik pritisne gumb za spremanje treninga taj trening će biti spremljen u bazu podataka, a ako pritisne gumb za odustajanje, samo će prijeći na zaslon sa trenutno spremljenim treninzima. U slučaju da korisnik ne želi odmah spremiti trening u bazu podataka, on uvijek može naknadno ispuniti anketu pritiskom na plutajući gumb. Nakon što korisnik ispuni upitnik vidljiva su mu tri zaslona između kojih može vrlo lako navigirati zato što sva tri dijele „*bottom*“ navigaciju. Prvi zaslon je zaslon sa trenutno spremljenim treninzima u bazi podataka, drugi zaslon je prikaz svih igrača u klubu i njihovih karakteristika, treći zaslon je prikaz performansi i karakteristika tima i njihovo trenutno stanje. Na zaslonu s listom igrača moguće je kliknuti na pojedinog igrača gdje će se na novom zaslonu prikazati svi detalji o tom igraču te moguća poboljšanja za određene kategorije tog igrača. Na zaslonu s karakteristikama tima je također moguće pritiskom na gumb prikazati sva moguća poboljšanja u određenim kategorijama. Performanse igrača i tima su prikladno vizualizirane u odnosu na uspjeh igrača ili tima u toj kategoriji. Cijela aplikacija je ostvarena na MVVM arhitekturi gdje su prikladno odvojeni modeli podataka, logika aplikacije te prikaz korisničkog sučelja. Navigacija u aplikaciji je vrlo jednostavna i intuitivna gdje se vrlo lako mogu slati i podaci između zaslona.

Funkcijski zahtjevi na aplikaciju su:

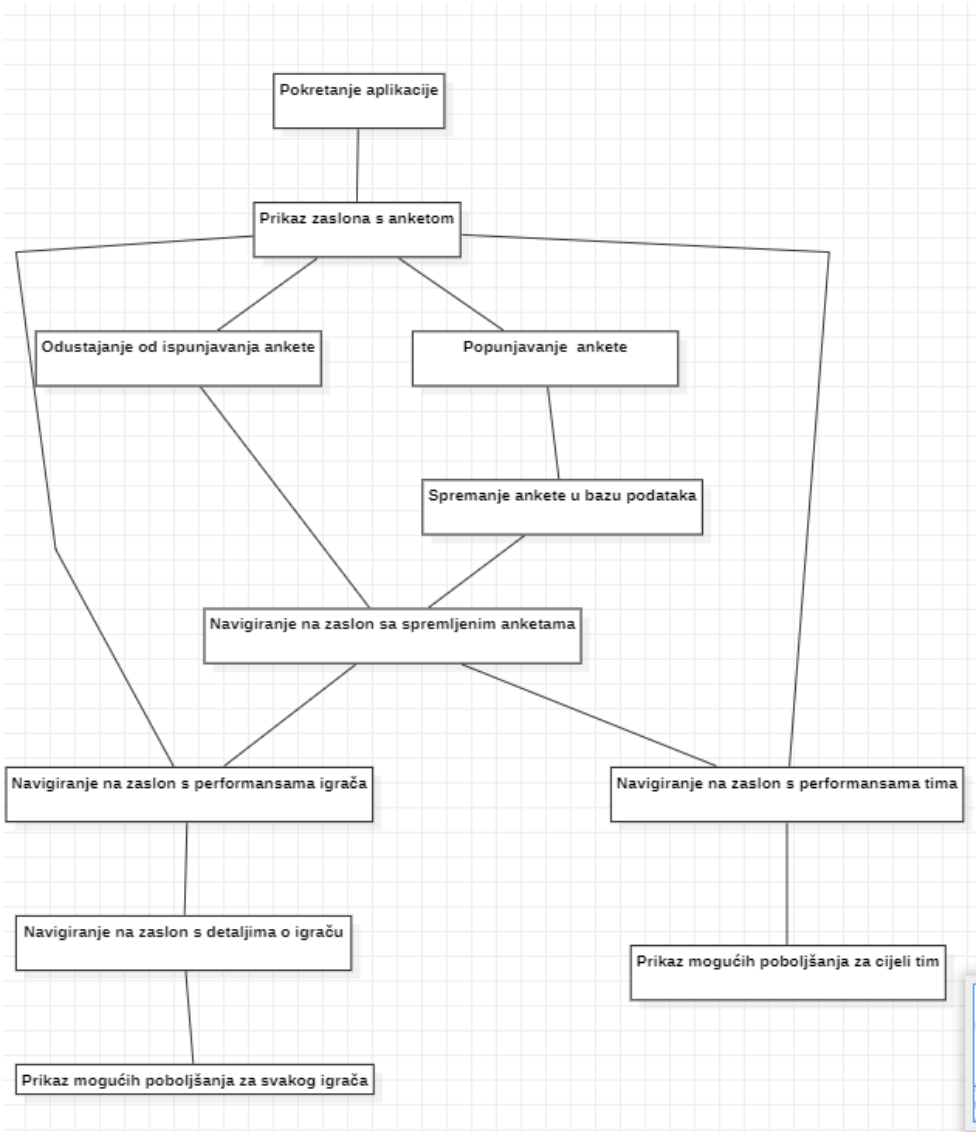
- Ulaskom u aplikaciju korisnik može birati hoće li ispuniti anketu o prethodno ispunjenom treningu
- Ispunjavanje ankete o treningu
- Spremanje treninga u bazu podataka
- Korisnik može vidjeti prethodno odrađene treninge koji su spremljeni u bazu podataka u obliku liste
- Korisnik može obrisati treninge iz baze podataka po želji
- Korisnik može vidjeti listu igrača svoje ekipe
- Korisnik može vidjeti posebne statistike i performanse svakog igrača u više kategorija
- Korisnik može vidjeti statistike i performanse cijelog tima u više kategorija
- Korisnik može vidjeti poboljšanja pojedinog igrača u različitim kategorijama
- Korisnik može vidjeti poboljšanja cijelog tima u različitim kategorijama

Kao što su navedeni funkcionalni zahtjevi, tako je potrebno navesti i nefunkcionalne zahtjeve. Prema [6], nefunkcionalni zahtjevi ili zahtjevi kvalitete usluge su zahtjevi koji su opisani sustavom ili prema sustavu. Za razliku od funkcionalnih zahtjeva koji opisuju što će programski proizvod raditi, nefunkcionalni zahtjevi opisuju kako programska podrška mora reagirati u određenim „rubnim“ situacijama kao što je na primjer veliki broj korisnika, koliko vremena će aplikacija bit nedostupna, brzina aplikacije, minimalne postavke operacijskog sustava za rad s aplikacijom te još mnogi. Nefunkcionalni zahtjevi mogu se odnositi na: performanse, sigurnost, skalabilnost, preciznost, promjenjivost i prenosivost. U sljedećem dijelu će biti navedeni neki od nefunkcionalnih zahtjeva za aplikaciju za ovaj rad.

Aplikacija *TrainingHelper*, odnosno aplikacija koja se izrađuje su sklopu ovog rada ima minimalni SDK 24, što znači da korisnici s operacijskim sustavom niže inačice od Androida 7 neće moći koristiti ovu aplikaciju prema [7]. Inačicu Androida 7 ili više ima 96,2% korisnika, što znači da će velika većina korisnika biti u mogućnosti koristiti ovu aplikaciju. Ciljana SDK inačica je 33, što odgovara inačici Android 13 operacijskog sustava, što je u ovom trenutku još uvijek najnovija inačica Android OS-a, budući da je Android 14 još uvijek u beta inačici za vrijeme izrade ovog rada. Vezano za brzinu aplikacije, aplikacija radi s dohvaćanjem podataka iz baze podataka te s vanjske usluge. Podaci iz baze podataka se dohvaćaju odmah, odnosno za manje od 0.5 sekundi. Dohvaćanje podataka s vanjske usluge uvelike ovisi o brzini interneta korisnika aplikacije, ali u slučaju testiranja je svaki put bilo manje od sekunde, učitavanje slike igrača može biti između 1

do 2 sekunde. Tako se zaključuje da aplikacija radi vrlo brzo i odgovara na sve zahtjeve u vrlo kratkom roku.

Aplikacija nema nikakvu autentifikaciju pri ulasku tako da ne ispunjava sigurnosne kriterije, ali budući da je predviđeno da se takva aplikacija daje izravno klubu na korištenje. Klub mora omogućiti da samo treneri i igrači imaju pristup toj aplikaciji te podacima unutar nje. Programski kod aplikacije je pisan tako da se u svakom trenutku može dodati nova funkcionalnost bez potrebe za mijenjanjem neke od prethodnih funkcionalnosti. Stoga se može reći kako je aplikacija vrlo proširiva i promjenjiva. Na slici 3.1 vidljiv je dijagram korištenja aplikacije kako bi korisnik mogao vidjeti koje su sve njegove mogućnosti pri korištenju ove aplikacije. Iz dijagrama je razvidno kako ova aplikacija nudi priliku svojim korisnicima koji su igrači ili treneri da vide svoje trenutne karakteristike te možda i još važnije svoje potencijalne napretke.




**Slika 3.1**. Dijagram toka rada aplikacije

## 3.2. Potrebni postupci i programski pristupi za ostvarenje aplikacije

U ovom dijelu bit će opisane sve funkcionalnosti u aplikaciji, način njihova rada, te razlozi zašto se baš te funkcionalnosti koriste. Implementiran je dizajn prema kojem će se dodavati ove funkcionalnosti u aplikaciju.

### 3.2.1. Postupak postavljanja početne ankete

Kako bi se počela raditi prva funkcionalnost ove aplikacije, potrebno je napraviti korisničko sučelje prema kojem će se vidjeti zaslon aplikacije kako ga bude sam korisnik vidio. Za izradu grafičkog sučelja u izradi Android aplikacija se koristi biblioteka *Jetpack Compose*. *Jetpack Compose* će biti u potpunosti opisan u daljnjim dijelova rada. *Compose* ima niz predefiniranih elemenata koji mogu biti korišteni za izradu početnog upitnika. Na primjer, *Jetpack Compose* ima svoj predefinirani *RadioButton* koji je onda iskorišten i modificiran prema potrebama ove aplikacije. Tako se na ovom zaslonu nalazi više elemenata kao što su *RadioButton, Slider, TextField*. Cilj početne ankete je dobiti trenerov dojam nakon odrađenog treninga, te onda kada ispuni anketu da na listi prethodno ispunjenih anketa vidi kakav je trend na njegovim treninzima. Je li zadovoljan napretkom igrača, njihovom koncentracijom, zalaganjem, itd. Pitanja koja su postavljena u anketi su glavne informacije o prethodnom treningu, a to je: koja vrsta treninga je bila. Mogući odgovori su ofenzivni, defenzivni te fizički. Ti odgovori su u obliku *radio buttona,* odnosno moguće je samo jedan od tri odgovora označiti. Drugo pitanje je koliko je trener zadovoljan s koncentracijom svojih igrača na treningu. Odgovor je na *slideru*, a šalje se vrijednost u postotku, odnosno u granicama od 0 do 100. Sljedeće pitanje je koliko je trener zadovoljan zalaganjem svojih igrača na treningu. Kao i na prethodnom pitanju, odgovor je na *slideru* kao postotak u granicama od 0 do 100. Četvrto i zadnje pitanje zapravo nije i pitanje nego mjesto za trenera da upiše svoje bilješke s posljednjeg treninga, ako želi, trener može ostaviti to polje praznim. Pitanja u ovoj anketi su postavljena treneru, kada ih on ispuni, ima dvije opcije za pritisnuti, jedna je „*Cancel*“ gdje odustaje od spremanja tog upitnika, a drugi je „*Confirm*“ pomoću kojeg trenutni trening sprema u bazu podataka, no to će biti objašnjeno u podpoglavlju 3.2.2.

### 3.2.2. Spremanje i prikazivanje prethodno ispunjene ankete

Kao što je već opisano u podpoglavlju 3.2.1, ulaskom u aplikaciju korisnik ispunjava anketu vezan uz posljednji trening. Nakon što ispuni anketu ima dvije opcije, spremiti trening u bazu podataka ili može odustati od spremanja treninga. Obje radnje će ga odvesti na zaslon koji prikazuje sve spremljene treninge iz baze podataka. Kada korisnik pritisne gumb za spremanje treninga u bazu podataka, taj trening je spremljen u bazu podataka i ako se prati elemente baze podataka, vidljivo

je da je taj novi element odmah dodan u bazu podataka. U ovom dijelu će biti opisano dohvaćanje svih elemenata baze podataka te prikaz istih u obliku liste. Sve elemente baze podataka se dobije tako da se pozove metoda iz DAO sučelja *getAllSurveys*() koja će vratiti sve elemente u obliku liste. Nakon što se dohvate svi elementi liste potrebno ih je mapirati kako bi bili spremni za svako moguće stanje ekrana. Može biti prazna lista, neuspješno dohvaćanje liste, učitavanje liste i uspješno učitana lista. Kada se ti podaci mapiraju, priprema se korisničko sučelje. Korisničko sučelje je u obliku liste, a pojedini element liste je kartica. Kartica je napravljena tako da pozadinska boja kartice može biti zelena, plava ili crvena, ovisno o vrsti treninga koji je napravljen. Ako je trening ofenzivan pozadinska boja je plava, ako je defenzivan onda je crvena, a ako je fizički onda je zelena. Osim pozadinske boje, na kartici je i vrijednost postotka koncentracije igrača na treningu, njihovo zalaganje također u obliku postotka te trenerove bilješke.

### 3.2.3. Model baze podataka

Kao što je vidljivo iz prethodnih poglavlja ovog rada, baza podataka je iznimno bitna za rad ove aplikacije. Baza podataka se koristi za spremanje upitnika o treningu koje ispunjava trener. Budući da se radi o Android aplikaciji za bazu podataka se koristi biblioteka *Jetpack Room Database*. *Room* baza podataka će biti opisana kasnije u radu. Podaci koji se spremaju u bazu podataka su vrsta treninga tipa *string* koji se dobije kao označeni odgovor *radio buttona*, koncentracija igrača kao decimalni broj koji se dobije iz *slidera*, zalaganje igrača na treningu kao decimalni broj koji se također dobije iz *slidera*, te trenerove bilješke koje se spremaju u obliku *stringa*. Model koji se sprema u ovu bazu podataka se sastoji od dva *stringa* te dva decimalna broja. Za spremanje elementa u bazu podataka koristi se funkcija *insert* iz DAO-a. Kao parametar funkciji *insert* se mora poslati element koji se treba spremiti u bazu podataka. Izvršavanjem metode insert, element odmah postaje vidljiv u bazi podataka. U ovoj aplikaciji još postoje dvije metode za manipuliranje bazom podataka osim inserta, a to su *delete* koja briše pojedinog člana baze podataka te *getAll* koja dohvaća sve elemente baze podataka u obliku liste.

### 3.2.4. Priprema parametara performansi igrača

Podatke o igračima kluba inače bi trebao voditi sam klub u evidenciji te onda te podatke predati agenciji koja radi aplikaciju kako bi ih oni pripremili za adekvatan prikaz u aplikaciji. U ovom slučaju, podaci za igrače su pripremljeni za ekipu Chelsea FC. Podaci su pripremljeni s internetskih stranica [8] i [9].

Podatke je filtrirao i pripremio kreator ovog rada. Podaci su pripremljeni za 15 igrača. Kategorije igrača koje su popunjene podacima su: ime, prezime, datum rođenja, pozicija, broj, brzina,

izdržljivost, snaga, kontrola lopte, koncentracija, kreativnost, timski rad, reakcija, snaga udarca, visina, težina, golovi, asistencije, igrači utakmice te slika igrača. Kategorije koje su karakteristike igrača vezane uz njegov stil igra su vrijednosti u granicama od 0 do 100, dok su igračeve fizičke karakteristike te njegova statistika na utakmicama brojevi koji nisu izraženi u postotku nego zapravo kakvi oni sami i jesu. Nakon što su podaci bili pripremljeni, bilo je potrebno ostvariti ih u nekom obliku poslužitelja kojeg će onda mobilna aplikacija koristiti. Za svrhe ove aplikacije bilo je potrebno samo čitanje podataka s vanjskog poslužitelja.

Prvo su podaci prebačeni iz tablice u JSON. Prema [10], JSON ili *JavaScript Object Notation* je lagani format za razmjenu podataka, koji je lako čitljiv ljudima, a strojevi ga lako analiziraju. JSON se temelji na programskom jeziku JavaScript. JSON se zasniva na strukturi ključ-vrijednost. Upravo zato što ga i ljudi i strojevi vrlo lako koriste, JSON je idealan alat za razmjenu podataka. Nakon što su podaci prebačeni u JSON format, bilo je potrebno pronaći alat koji će biti vanjski poslužitelj koji će prikazivati pripremljene podatke. U slučaju ove aplikacije korišten je github.io. Prema [11], *Github* dopušta posluživanje jedne web stranice po *Github* računu. Napravljena je stranica u koju je dodan JSON s podacima koji su potrebni za rad aplikacije. Nakon što je to napravljeno, podaci i vanjski poslužitelj su spremni za korištenje. U tablici 3.1 prikazani su podaci o igraču Reeceu Jamesu.

| Parametri igrača | Primjeri unesenih vrijednosti |
|---|---|
| Ime | Reece |
| Prezime | James |
| Datum rođenja | 8.12.1999. |
| Pozicija | RB |
| Broj dresa | 24 |
| Brzina | 81 |
| Izdržljivost | 77 |
| Snaga | 85 |
| Kontrola lopte | 82 |
| Koncentracija | 77 |
| Kreativnost | 84 |
| Timski rad | 89 |
| Reakcija | 83 |
| Snaga udarca | 89 |
| Visina | 182 |
| Težina | 82 |
| Golovi | 1 |
| Asistencije | 1 |
| Igrač utakmice | 0 |
| Fotografija | URL |

Tablica **3.1.** Prikaz statistika pojedinog igrača

### 3.2.5. Dohvaćanje i prikaz igrača i njihovih performansi

Budući da su u poglavlju 3.2.4 pripremljeni podaci o igračima i postavljeni na vanjski poslužitelj, sada je potrebno iskoristiti ih i prikazati u aplikaciji. Za dohvaćanje podataka s vanjskog poslužitelja u Android aplikaciji jedan od načina je korištenje biblioteke *Retrofit*. Prema [12], *Retrofit* je HTTP mrežna biblioteka korištena za razvoj Android aplikacija. Koristi se za mrežne pozive, nudi sve funkcionalnosti CRUD razvoja *(Create, Read, Update, Delete)*. Retrofit podržava Kotlin korutine te pomoću njih radi mrežne pozive u pozadini te ne blokira glavnu nit. Prema [13], korutine predstavljaju koncept asinkronog programiranja. Omogućuju izvršavanje blokirajućih operacija bez gubljenja stanja te bez blokiranja glavne niti tako što omogućuju izvršavanje na drugim nitima. Korutine omogućuju efikasno upravljanje resursima. Korutine se označavaju ključno riječi suspend. Prije dohvaćanja podataka potrebno je definirati podatkovnu klasu u koju će se serijalizirati JSON podaci s interneta. Podatkovna klasa ima sve elemente koje ima i JSON, ali to ne mora biti uvijek slučaj. Podatkovna klasa ima parametre: ime, prezime, datum rođenja, pozicija, broj, brzina, izdržljivost, snaga, kontrola lopte, koncentracija, kreativnost, timski rad, reakcija, snaga udarca, visina, težina, golovi, asistencije, igrač utakmice te fotografija igrača. Kada se podaci dohvate i serijaliziraju možemo manipulirati s listom objekata igrača. Te podatke je potrebno mapirati s obzirom na trenutno stanje korisničkog sučelja. Stanje korisničkog sučelja može biti: učitavanje, neuspješno dohvaćanje igrača, dohvaćanje prazne liste igrača te uspješno dohvaćanje liste igrača. Kada su podaci mapirani, šalje ih se prema korisničkom sučelju. Korisničko sučelje je u ovom slučaju pripremljeno kao *HorizontalPager* gdje je svaka stranica jedna kartica o igraču. Kartica o igraču se sastoji od njegovog imena i prezimena, datuma rođenja, broja na dresu, broja golova, pozicije, broja ostvarenih igrača utakmice te broja asistencija. Prijelazom po ekranu lijevo ili desno se mogu vidjeti sve kartice igrača, pomoću indikatora na dnu ekrana vidljivo je na kojoj poziciji se trenutno nalazite. Klikom na pojedinu karticu, prelazi se na novi zaslon koji prikazuje sve detalj o tom igraču. Na tom zaslonu je u gornjoj aplikacijskoj traci vidljivo ime, prezime i broj igrača. Na samom ekranu je vidljiva njegova slika te svi ostali detalji o igraču u obliku postotka na *slideru*. Podaci na *slideru* su u granicama od 0 do 100, te su s obzirom na vrijednost obojani različitom bojom.

### 3.2.6. Algoritam za poboljšavanje performansi igrača

Budući da je već opisan proces dohvaćanja podataka o igračima s vanjskog poslužitelja, potrebno je osmisliti algoritam koji će s obzirom na vrijednosti njihovih performansi prikazati moguća poboljšanja u određenim kategorijama te načine kako je moguće ostvariti bolje rezultate u tim kategorijama. Kategorije za koje su se prikazivala moguća poboljšanja performansi su brzina,

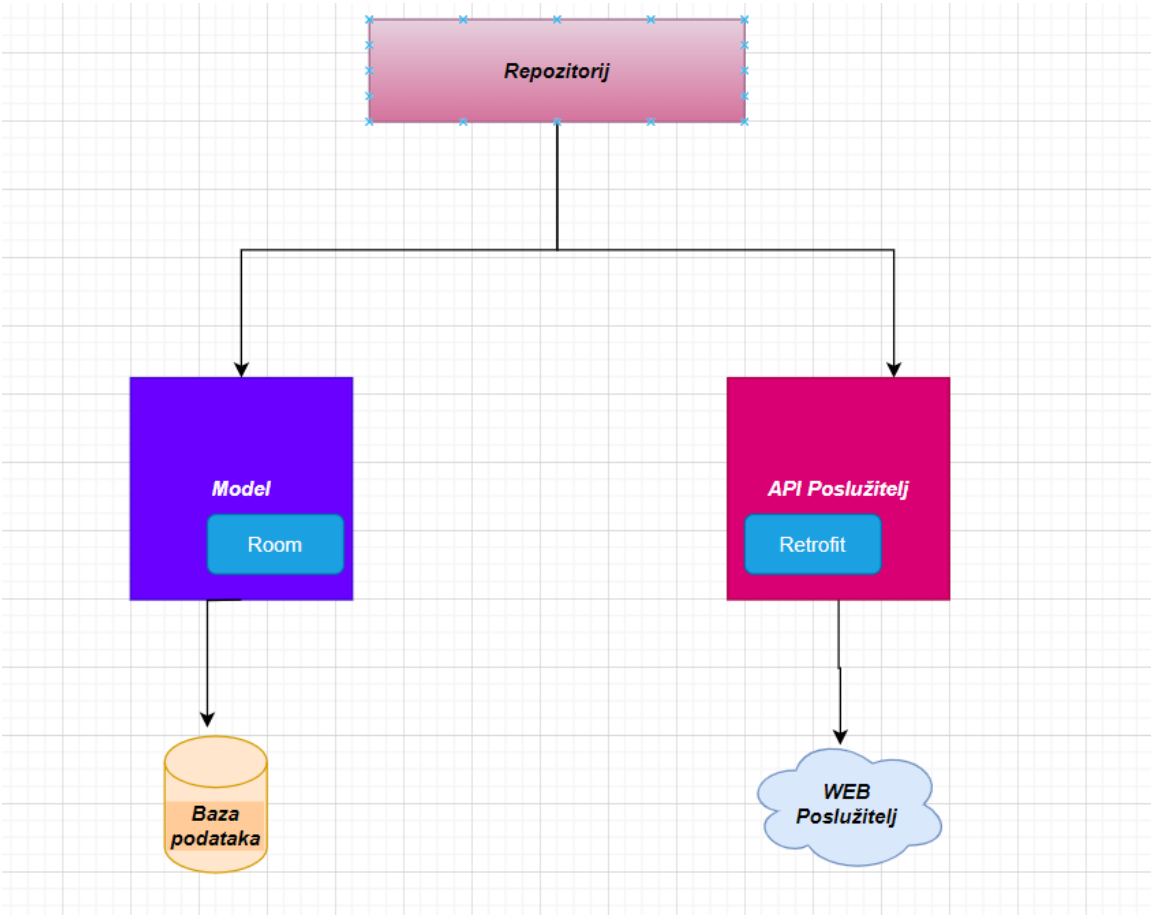
izdržljivost, snaga, koncentracija, kreativnost i timski rad. Budući da su performanse igrača bile ocijenjene u rasponu od 0 do 100, prema tim vrijednostima su ciljana i poboljšanja igrača. Ukoliko igrač ili tim ima vrijednost gore nabrojanih fizičkih atributa manju od 70, onda mu se preporučuje intenzivni rad s kondicijskim trenerom ili rad u klupskoj teretani kako bi poboljšao svoje performanse. Ukoliko su vrijednosti igračevih fizičkih atributa između 70 i 85 igraču se preporučuje da u svoje slobodno vrijeme poradi na elementu na kojem su mu vrijednosti u tom rasponu. Preporuke u aplikaciji su stvarane na temelju onoga što igraču nedostaje. Primjerice, ako je igrač sporiji od onoga što je preporučeno za njegovu poziciju, tom igraču se preporučuje više kondicijskih treninga te fizičkih treninga na kojima će raditi na agilnosti, brzini te sličnim karakteristikama. Ukoliko su igračevi problemi fizičke prirode, igraču će se kao put do poboljšanja nuditi dodatni rad s kondicijskim trenerom kako bi poboljšao svoje fizičke performanse. Slična stvar je i za psihičke atribute kao što su koncentracija, timski rad i kreativnost, samo što rješenje nije rad s kondicijskim trenerom nego druženje s ostalim igračima, razmišljanje izvan zamišljenih okvira te slična ostala rješenja. Ukoliko igrač ima vrijednost nekog od atributa iznad 85 prikazat će mu se poruka pohvale i uputa da samo nastavi u tom smjeru budući da je to zavidan rezultat. Ta poboljšanja će se prikazati tek kada ih korisnik zatraži, odnosno kada pritisne gumb za prikaz poboljšanja. Neke od preporuka su: dodatni rad s kondicijskim trenerom, samostalan odlazak u klupsku teretanu, odlazak na druženja s drugim igračima, pokušaj razmišljanja na drugačiji način i sl.

## 3.3. Arhitektura mobilne aplikacije za upravljanje performansama nogometnog kluba

Pri izradi ove aplikacije je korištena MVVM arhitektura. Prema [14], *Model-View-ViewModel* je arhitekturalni obrazac koji nudi veću odvojenost koda i može biti korištena među više platformi. MVVM arhitektura se prvi put spominje pojavom Googleove biblioteke *Data Binding* na događaju Google I/O 2015. godine. *Data Binding* je biblioteka koja se koristi u razvoju Android aplikacija pomoću opisnog jezika XML. Služi za povezivanje koda i komponenti grafičkog sučelja uz minimalnu uporabu programskog koda [15]. Pojavom biblioteke Jetpack Compose, korištenje data bindinga više nije potrebno, budući da Compose u pozadini rukuje s tim procesima. Glavna svrha MVVM arhitekture je odvajanje slojeva ili odgovornosti (*eng. separation of concerns*), odvajanje poslovne logike i modela od korisničkog sučelja. MVVM arhitektura je nastala iz arhitekture MVP *(model-view-presenter)* uz Googleov dodatak nove klase koja se naziva *ViewModel*. Prema [16], glavne prednosti korištenja MVVM arhitekture su: lakše testiranje koda, kod nije usko povezan, lakša navigacija kroz strukturu projekta, lakše održavanje aplikacije te lakše dodavanje novih

funkcionalnosti. Kao što ime nalaže, MVVM se sastoji od tri sloja. *Model*, *View* i *ViewModel*. Prema [17], neke od korisnih strana MVVM arhitekture su: lakše otklanjanje pogrešaka, lakše pisanje testova te lakše dodavanje novih funkcionalnosti.

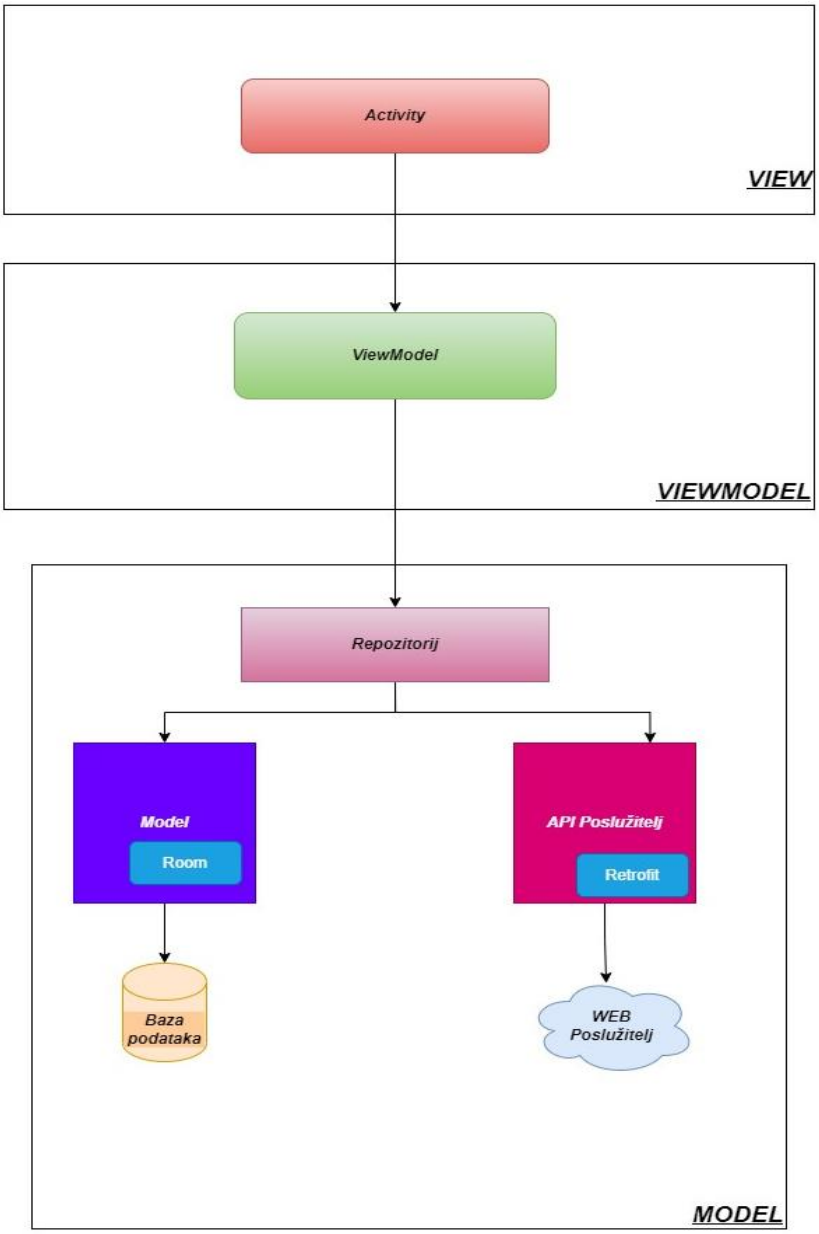
Prema [18], model je arhitekturalni sloj koji sadrži klase koje su zadužene za direktan pristup podacima s ciljem da se pojednostavi pristup tim podacima. Na primjeru ove aplikacije u modelu postoje podatkovne klase koje služe za serijalizaciju JSON podataka, podatkovna klasa koja služi kao entitet za spremanje u bazu podataka, instance baze i vanjskog poslužitelja, te repozitoriji u kojima se odvija logika prilagodbe podataka potrebama aplikacije. Poveznica između modela i *viewmodela* je repozitorij. Veza između repozitorija i *viewmodela* nije jednaka, repozitorij ne zna za *viewmodel*, dok *viewmodel* zna za repozitorij i poziva funkcije iz repozitorija. U repozitoriju se nalaze sve metode kojima se oblikuju podaci koji dolaze s vanjskog poslužitelja, baze podataka ili nekog drugog poslužitelja s poslužiteljske strane. Aplikacije mogu imati više izvora podataka, baš kao što i ova aplikacija ima. Zato repozitorij služi kao jedinstvena veza s više izvora podataka, kao što je vidljivo na slici 3.2.




**Slika 3.2.** Prikaz strukture obrasca *repository*

*Viewmodel* je arhitekturni sloj koji služi kao veza između *viewa* i *modela*. On je također klasa koja je odgovorna za pripremu podataka i mapiranje podataka za view. Prema [19], *Viewmodel* priprema podatke dobivene iz data modela te ih prilagođava Viewu koji te podatke prikazuje na korisničkom sučelju. *Viewmodel* ne zna za *view* kojem priprema podatke, ali zna za repozitorij od kojeg prima podatke s kojima rukuje. Također, *Viewmodel* preživljava konfiguracijske promjene kao što je, na primjer, rotacija zaslona. Ukoliko dođe do konfiguracijske promjene, a podaci se nalaze u *viewu*, ti podaci će nestati, ali ako su u *viewmodelu* oni će biti sačuvani i ponovno predani *viewu* da ih prikaže na zaslonu. Ukoliko aplikacija koristi Fragmente, *Viewmodel* može služiti za komunikaciju između fragmenata i može držati podatke koje će oba fragmenta koristiti [20]. Ono što je *presenter* u MVP arhitekturi, to je *viewmodel* u MVVM arhitekturi, razlika je što *presenter* ima referencu na *view* dok *viewmodel* nema. *Viewmodel* nema referencu na *view* zato što može proživjeti konfiguracijske promjene, a view ne može. Ukoliko bi imao referencu na *view*, i došlo bi do konfiguracijske promjene, dogodio bi se prestanak rada aplikacije. Svaki *viewmodel* mora naslijediti klasu *Viewmodel()* koja je predefinirana klasa razvijena od strane *Googlea* [21].

*View* je arhitekturni sloj koji je zadužen za prikaz korisničkog sučelja i prikazivanje podataka iz *viewmodel*a na zaslon korisnika. Budući da *view* ima referencu na *viewmodel*, moguće je promatrati promjene u *viewmodelu* i na svaku promjenu u *viewmodelu* reagirati promjenom *viewa*. Prema MVVM arhitekturi, view ne smije sadržavati poslovnu logiku. Glavna uloga *viewa* je da se pretplati na podatke koje će, prema ovoj arhitekturi, dobiti od *viewmodela* [21]. Activityji koji predstavljaju zaslon u razvoju Android aplikacija su zapravo dio viewa. U razvoju Android aplikacija, *view* se može prikazati na dva načina: pomoću opisnog jezika XML, te pomoću programskog jezika Kotlin te biblioteke *Jetpack Compose*. Na slici 3.3 vidljiva je struktura MVVM arhitekture.




**Slika 3.3.** Prikaz MVVM arhitekture

# 4. PROGRAMSKO RJEŠENJE MOBILNE APLIKACIJE ZA UPRAVLJANJE PERFORMANSAMA NOGOMETNOG KLUBA

U ovom dijelu rada opisane su korištene programske tehnologije i jezici te programska implementacija aplikacije za upravljanje performansama nogometnog kluba s korisničke i poslužiteljske strane.

## 4.1. Korištene programske tehnologije, jezici i razvojne okoline

### 4.1.1. Operacijski sustav Android

Prema [22], Android OS je otvoreni operacijski sustav za mobitele, tablete, televizore i automobile. Osnovne usluge Androida temelje se na Linux jezgri 2.5, kao što je sigurnost, upravljanje memorijom, procesima i modelom upravljačkih programa. Linux jezgra također predstavlja apstraktni sloj između hardverskog i programske slojeva. [23] U ovom trenutku je jedan od dva najpopularnija operacijska sustava za mobilne uređaje na svijetu uz Appleovu inačicu operacijskog sustava. Android OS je nastao 2003. godine kada su Andy Rubin, Rich Miner, Nick Sears i Chris White osnovali tvrtku *Android Inc*. kako bi razvijali aplikacije za pametne mobitele. Prvi uređaji s Android OS-om nisu bili mobiteli s zaslonom na dodir, već s tipkama. No kako je konkurencija počela stvarati uređaje s zaslonom na dodir, tako su i iz Androida reagirali. Značajke operacijskog sustava Android su jednostavno korisničkog sučelje kojim korisnik jasno može ostvariti zacrtane zadatke na uređaju. Bazira se na interakciji korisnika pomoću gesti na uređaju koje okidaju akcije u pozadini koje će obaviti neki zadatak. Android je poznat po svojoj dostupnosti. Svaki developer može manipulirati operacijskim sustavom i napraviti svoju inačicu iz temeljne inačice Android operacijskog sustava. Najviše se cijene inačice operacijskog sustava najsličnije originalnoj inačici Androida koji je *Google* napravio. *Android Inc*. je u poslovnoj suradnji s Googleom, ali ne i u njegovu vlasništvu. Služi kao prevoditelj između korisnika i uređaja, npr. kada korisnik želi fotografirati nešto, sustav mu pruža gumb koji će korisnik stisnuti i onda će sustav narediti uređaju kako da to napravi. U trenutku pisanja ovog rada, najnovija inačica Android OS je Android 13. Inačice Android OS-a su nazvane prema raznim desertima kao na primjer *Oreo, Lollipop, Nougat*, itd.

### 4.1.2. Programski jezik Kotlin

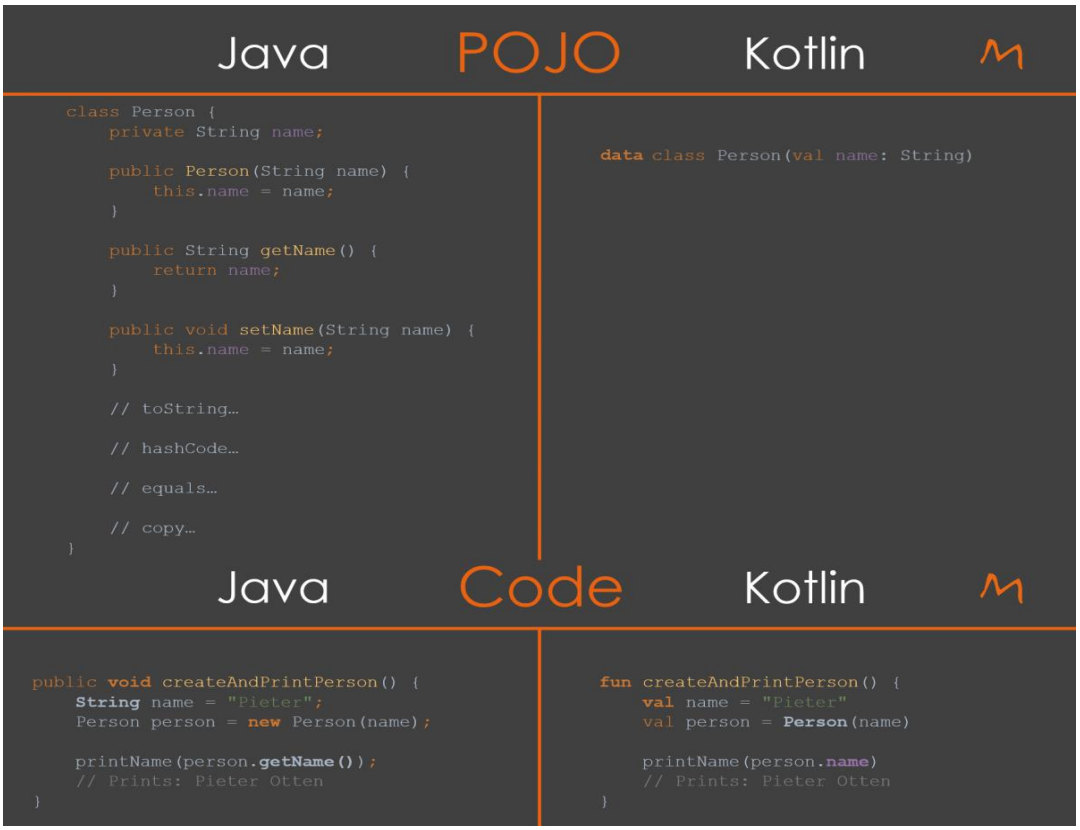
Prema [24], *Kotlin* je programski jezik nastao 2011. godine kao *„open source“* projekt osnovan od strane ruske kompanije *JetBrains*. Kao što je programski jezik Java dobio ime po indonezijskom otoku, tako je i *Kotlin* dobio svoje ime po otoku u blizini grada Sankt Petersburg. *Kotlin* je vrlo moćan programski jezik koji nudi mnoge alate kao što su naprimjer podrška za

asinkrono programiranje, *„null safety“,* ekstenzijske funkcije, lambda izraze te mnoge druge. Ti alati čine *Kotlin* vrlo elegantnim i konciznim jezikom za pisanje urednog i održivog koda. Prema [25], uporabom Kotlina u razvoju programske podrške značajno se poboljšala kvaliteta programskog koda. Najpoznatiji je po uporabi u razvoju Android aplikacija, gdje je zamijenio Javu zato što je jednostavniji za pisat uz manje programskog koda, te nudi sve funkcionalnosti koje ima Java te još dodatne kojima se podiže iznad *Jave*. *Kotlin* i *Java* su interoperabilni što znači da se projekti mogu postupno prebacivati s *Jave* na *Kotlin,* te se biblioteke napravljene za Javu mogu koristiti i s *Kotlinom*.

U veljači 2016. godine izlazi prva stabilna inačica *Kotlina*. U trenutku pisanja ovog rada najnovija inačica Kotlina je 1.8.22 te sa svakom inačicom dolaze dodatne funkcionalnosti koje olakšavaju razvoj aplikacija programeru. 2017. godine na Google I/O događaju, *Kotlin* je proglašen preferiranim programskim jezikom za razvoj Android aplikacija što mu je dodatno povećalo popularnost. Osim za razvoj mobilnih aplikacija, *Kotlin* se može koristiti i u drugim tehnologijama, kao što su na primjer razvoj poslužiteljske strane pomoću razvojnih okruženja *Ktor* ili *Spring Boot*, za razvoj web aplikacija te za podatkovne procese vezanih za strojno učenje. Jedan od razloga korištenja *Kotlina* je velika baza korisnika te je lakše otkloniti problem te pronaći rješenje na internetu, zato što se netko vrlo vjerojatno već susreo s takvim problemom i postavio rješenje na internetu. Na slici 4.1 prikazana je razlika *Jave* i *Kotlina* na primjeru podatkovne klase.

### 4.1.1. Biblioteka Jetpack Compose

Prema [26], *Jetpack Compose* je moderni alat za izgradnju nativnog korisničkog sučelja za Android aplikacije. U srpnju 2021. godine izašla je prva stabilna inačica *Jetpack Composea*. Pomoću *Jetpack Composea*, razvoj korisničkog sučelja se pojednostavljuje i ubrzava, koriste se intuitivni *Kotlin* API-i. *Jetpack Compose* koristi deklarativni način programiranja, za razliku od dosadašnjeg pristupa s XML-om gdje se koristio imperativni način programiranja. *Compose* se temelji na tome da se predefinirane komponente uređuju prema potrebama programera te slažu kao korisničko sučelje. *Compose* donosi brojne prednosti u odnosu na XML. Vrlo je jednostavan za učenje, prikaz komponenti je jednostavan i brz, može biti interoperabilan s imperativnim pristupom. Nudi intuitivnu i vrlo jednostavnu sintaksu, dok se njegove komponente vrlo lako koriste u više navrata što smanjuje potrebu za ponavljanjem koda.




**Slika 4.1.** Razlike programskog koda Jave i Kotlina [24]

Također, nudi opciju „*Preview*“ kojom programer odmah može vidjeti promjene koje je implementirao u programski kod. *Preview* se može pokrenuti u odvojenom prozoru ili na uređaju, bio on fizički ili emulator. Pojavom *Jetpack Composea* više ne postoji potreba za implementacijom *Data Bindinga* ili *View Bindinga* zato što *Compose* to sam rukuje. Jedna od najbitnijih značajki Composea je njegova reaktivnost. Ako dođe do promjene stanja podatka koji se prikazuje, doći će do rekompozicije. Rekompozicija je proces u kojemu pri promjeni stanja varijable se ponovno poziva *Compose* funkcija te se korisničko sučelje ponovno stvara [27]. Svaka funkcija kojom se želi nešto prikazati na zaslonu mora imati anotaciju *@Composable*. Što se tiče performansi, još uvijek blagu prednost ima XML pred *Composeom*, ali nema sumnje da će se u doglednoj budućnosti to promijeniti.

### 4.1.2. Razvojna okolina Android Studio

Android Studio je integrirano razvojno okruženje napravljeno za razvoj mobilnih aplikacija za operacijski sustav Android. Osim za razvoj mobilnih aplikacija, može se koristiti i za razvoj aplikacija za automobile, tablete te TV uređaje. Preuzimanje i instalacija Android Studija je potpuno besplatna te je također dostupan za operacijske sustave kao što su Windows, Linux i *macOS*. Aplikacije u Android Studiju je moguće pisati u programskim jezicima *Kotlin* i *Java,* ali *Kotlin* je preporučeni jezik za programiranje Android aplikacija. Android Studio dolazi s nizom

funkcionalnosti kao što su: uređivanje koda, stvaranje korisničkog sučelja, rad s emulatorom, *gradle* za građenje projekta, profiliranje rada aplikacije te integracija s Android SDK. Uređivanje koda u Android Studiju je vrlo jednostavno zato što sustav označava sintaksu te nudi pametno dovršavanje koda, također moguće je i reformatirati kod kako bi bio čitljiviji i uredniji. Pomoću Android Studija je lako napraviti korisničko sučelje aplikacije zato što sustav nudi opciju slaganja predefiniranih komponenti kako bi se stvorilo korisničko sučelje uz klasični način stvaranja sučelja što je pisanje programskog koda. Kako bi korisnik Android Studija vidio rad svoje aplikacije nudi mu se opcija pokretanja aplikacije na fizičkom uređaju te na emulatoru. Korištenje emulatora je dosta praktično kako korisnik ne bi stalno morao pokretati aplikaciju na fizičkom uređaju, te može imati i više emulatora kako bi vidio rad aplikacije na različitim uređajima. Uz provjeru rada aplikacije, korisnik može vidjeti performanse svoje aplikacije te eventualne probleme s memorijom ili brzinom izvođenja zadataka. Pomoću alata u Android Studiju moguće je pratiti takve probleme te ih popraviti te optimizirati rad aplikacije. Pri stvaranju svakog projekta u Android Studiju, automatski se stvore i *gradle* datoteke. *Gradle* se koristi za automatizaciju gradnje aplikacije te omogućava vrlo lagano dodavanje ovisnosti vanjskih biblioteka. Integracija s Android SDK omogućuje korisniku pristup raznim alatima te bibliotekama korisnim za razvoj Android aplikacija, koji će vrlo lako implementirati uz detaljne upute sa službene dokumentacije. Kao što su inačice Android OS-a dobile ime po desertima, tako su inačice Android Studija dobile ime po životinjama, kao što su *Dolphin*, *Chipmunk*, *Electric eel*. U trenutku pisanja ovog rada, najnovija stabilna inačica je *Flamingo*.

## 4.2. Programsko rješenje na strani korisnika

U ovome poglavlju će biti opisana implementacija programskog rješenja aplikacije za upravljanje performansama nogometnog kluba. Uz prikazane dijelove programskog koda aplikacije bit će i objašnjenja samog koda te razlozi zašto je kod tako napisan.

### 4.2.1. Zaslon s prikazanom anketom

Ovo je početni zaslon aplikacije koji će se otvoriti. Na njemu se nalaze elementi poput *Radio Buttona*, *Slidera* te *Outlined TextFielda*. Korisničko sučelje zaslona je napravljeno pomoću biblioteke *Jetpack Compose*. *Jetpack Compose* u sebi već ima ugrađene komponente kao što su *Button*, *TextField*, *Alert Dialog* te mnoge druge. Te komponente je moguće prilagoditi prema svojim potrebama te potrebama aplikacije u kojoj su korištene, kao što je i učinjeno i u ovom slučaju. Kao što je vidljivo na slici 4.3, napravljene su nove komponente koje se nazivaju *RadioButtonQuestion, SliderQuestion* te *TextFieldQuestion*. To su zapravo, funkcije u kojima su

pozvane predefinirane komponente koje su onda uređene prema potrebama aplikacije. Kako bi funkcija bila prikazana na zaslonu mora imati anotaciju *@Composable*.

```
@Composable
fun SurveyContent(
    answerOptions: List<String>,
    selectedOption: String,
    onOptionSelected: (String) -> Unit,
    firstSliderPosition: Float,
    onFirstSliderChange: (Float) -> Unit,
    secondSliderPosition: Float,
    onSecondSliderChange: (Float) -> Unit,
    text: TextFieldValue,
    onTextChange: (TextFieldValue) -> Unit,
    onClearTextClick: () -> Unit,
    onCancelClick: () -> Unit,
    onConfirmClick: () -> Unit
) {
    Scaffold(
        bottomBar = {
            MultipleButtonsRow(
                onCancelClick = onCancelClick,
                onConfirmClick = onConfirmClick
            )
        }
    ) { it: PaddingValues
        Column(
            modifier = Modifier
                .fillMaxSize()
                .background(MaterialTheme.colorScheme.background)
                .verticalScroll(rememberScrollState())
                .padding(
                    start = 16.dp,
                    end = 16.dp,
                    top = 16.dp,
                    bottom = it.calculateBottomPadding()
```

**Slika 4.2.** Prikaz prvog dijela funkcije za prikaz korisničkog sučelja ankete

```
    ) { this: ColumnScope
        RadioButtonQuestion(
            quizQuestion = "What type of training you had?",
            answerOptions = answerOptions,
            selectedOption = selectedOption,
            onOptionSelected = onOptionSelected
        )
        SliderQuestion(
            quizQuestion = "What was your teams level of concentration?",
            sliderPosition = firstSliderPosition,
            onSliderChange = onFirstSliderChange
        )
        SliderQuestion(
            quizQuestion = "How do you rate your teams effort?",
            sliderPosition = secondSliderPosition,
            onSliderChange = onSecondSliderChange
        )
        TextFieldQuestion(
            quizQuestion = "This is where you write your additional notes...",
            text = text,
            onTextChange = onTextChange,
            onExitClick = onClearTextClick
```

**Slika 4.3.** Prikaz drugog dijela funkcija za prikaz korisničkog sučelja ankete

Kao što je vidljivo iz prikazanih dijelova koda, u ovoj funkciji nema nikakvih poziva *viewmodela* niti navigacije nego je ovo čista funkcija s korisničkim sučeljem. Zato je napravljena dodatna *@Composable* funkcija koja poziva ovu funkciju te ima referencu na *viewmodel* koji obavlja funkcionalnosti aplikacije te navigacija na daljnje zaslone aplikacije. Prikaz te funkcije je na slici 4.4.

```
@RootNavGraph(start = true)
@Composable
@Destination
fun SurveyScreen(navController: DestinationsNavigator) {
    val quizViewModel = getViewModel<QuizViewModel>()
    val answerOptions = listOf("Offensive", "Defensive", "Physical")
    val (selectedOption, onOptionSelected) = remember { mutableStateOf(answerOptions[1]) }
    var firstSliderPosition by remember {
        mutableStateOf( value: 0.0F)
    }
    var secondSliderPosition by remember {
        mutableStateOf( value: 0.0F)
    }
    var text by remember {
        mutableStateOf(TextFieldValue( text: ""))
    }
    SurveyContent(
        answerOptions = answerOptions,
        selectedOption = selectedOption,
        onOptionSelected = onOptionSelected,
        firstSliderPosition = firstSliderPosition,
        onFirstSliderChange = { firstSliderPosition = it },
        secondSliderPosition = secondSliderPosition,
        onSecondSliderChange = { secondSliderPosition = it },
        text = text,
        onTextChange = { text = it },
        onClearTextClick = { text = TextFieldValue( text: "") },
        onCancelClick = {
            navController.navigate(NavBarContainerScreenDestination) { this: NavOptionsBuilder
                popUpTo(SurveyScreenDestination) { this: PopUpToBuilder
                    inclusive = true
                }
            }
```

**Slika 4.4.** Prikaz funkcije SurveyScreen koja smješta podatke na korisničko sučelje

Na slici 4.4 je vidljivo kako funkcija drži referencu na *QuizViewmodel* pomoću kojeg se sprema sadržaj ovog zaslona u bazu podataka. Prema anotaciji *@RootNavGraph(start=true)* vidljivo je da je ovo početni zaslon koji će se prikazati pri otvaranju aplikacije. Također inicijaliziran je niz varijabli koje imaju vrijednost promjenjivog stanja koje će pri svakoj promjeni stanja tih varijabli okinuti novi poziv *@Composable* funkcija i ekran će se osvježiti. Definirane varijable se šalju *SurveyContent* funkciji koja je prikazana na slikama 4.2 i 4.3.

Na slici 4.5 prikazan je *QuizViewmodel* koji, prema MVVM arhitekturi drži logiku za ovaj zaslon. Ovaj zaslon nema previše poslovne logike u sebi tako da je *viewmodel* u ovom slučaju relativno jednostavan.

```
navi1111 *
class QuizViewModel(private val surveyRepository: SurveyRepository) : ViewModel() {
    navi1111 *
    fun insertSurvey(survey: Survey) {
        viewModelScope.launch { this: CoroutineScope
            surveyRepository.insertSurvey(survey)
        }
    }
}
```

**Slika 4.5.** Prikaz programskog koda QuizViewModela

Kao što je vidljivo iz slike 4.5, *QuizViewModel* je klasa koja nasljeđuje ugrađenu klasu *Viewmodel()*, tek nakon što se naslijedi ta klasa, *QuizViewModel* zapravo postaje *viewmodel* zato što tek sad može preživjeti konfiguracijske promjene. Da nije bilo nasljeđivanja te ugrađene klase ovo bi bila samo obična klasa koja ne bi imala posebnu funkcionalnost. *QuizViewModel* ima parametar koji je referenca na *SurveyRepository*. U funkciji *insertSurvey* se odvija poziv metode iz repozitorija istog imena. Taj poziv je moguće ostvariti zato što postoji referenca na repozitorij te funkcija *insertSurvey* u repozitoriju je *public*. Poziv funkcije *insertSurvey* iz *viewmodel*a se odvija u funkciji *SurveyScreen* što je prikazano na slici 4.6.

```
        onConfirmClick = {
            quizViewModel.insertSurvey(
                Survey(
                    playerConcentration = firstSliderPosition,
                    playerEffort = secondSliderPosition,
                    additionalNotes = text.text,
                    trainingType = selectedOption
                )
            )
            navController.navigate(NavBarContainerScreenDestination) { this: NavOptionsBui
                popUpTo(SurveyScreenDestination) { this: PopUpToBuilder
                    inclusive = true
                }
            }
        }
    )
}
```

**Slika 4.6.** Prikaz poziva funkcija iz QuizViewModela

### 4.2.2. Zaslon s prethodno ispunjenim anketama

Na zaslonu s prethodno ispunjenim anketama su prikazani sve trenutno spremljene ankete koje je trener ispunio nakon treninga. Prikazani su pomoću ugrađene *Jetpack Compose* komponente koja

se naziva *Lazy Column*. *Lazy Column* je ekvivalent *Recycler View* komponenti u XML-u. *Lazy column* stvara toliko objekata koliko je vidljivo na zaslonu uz još nekoliko objekata van trenutnog opsega zaslona. Implementacija korisničkog sučelja ovog zaslona je prikazana na slici 4.7.

```
@Destination
@Composable
fun SurveyListScreen() {
    val surveyListViewModel = getViewModel<SurveyListViewModel>()
    val collectedState by surveyListViewModel.uiState.collectAsStateWithLifecycle()
    when (collectedState) {
        UIState.Loading -> LoadingScreen()
        UIState.Error -> ErrorScreen { surveyListViewModel.getAllSurveys() }
        UIState.EmptyListState -> EmptyListScreen()
        is UIState.Success<*> -> SurveyListContent(
            surveyList = (collectedState as UIState.Success<*>).data as List<Survey>,
            onCardClick = { surveyListViewModel.deleteSurvey(it) }
        )
    }
}

 navi1111
@Composable
private fun SurveyListContent(surveyList: List<Survey>, onCardClick: (Survey) -> Unit) {
    LazyColumn() { this: LazyListScope
        items(surveyList) { this: LazyItemScope survey ->
            survey.additionalNotes?.let { additionalNotes ->
                SurveySummaryCard(
                    survey.playerEffort,
                    survey.playerConcentration,
                    additionalNotes,
                    survey.trainingType,
                    survey,
                    onCardClick
                )
            }
        }
    }
}
```

**Slika 4.7.** Prikaz funkcija koje čine korisničko sučelje zaslona s prethodno ispunjenim anketama

Funkcija *SurveyListContent* kao parametre prima listu objekata *Survey*, te *callback* kada korisnik klikne na jedan od elemenata *Lazy Columna*. *Survey* je objekt koji se sprema u bazu podataka, a to je zapravo anketa koju trener ispunjava nakon svakog treninga, što je obrađeno u prethodnom poglavlju. *Lazy Column* funkcionira tako da mu se u ovom slučaju preda lista objekata *Survey* te onda u lambda funkciji se odlučuje kako će se prikazati svaki *Survey* iz liste.

Na slici 4.8 je prikazana *@Composable* funkcija koja prikazuje zasebnu karticu na kojoj su prikazani podaci pojedinačnog *Surveya*. Svi podaci su prikazani na ugrađenoj komponenti *OutlinedCard*. Podaci koji se prikazuju su trud igrača na treningu, njihova koncentracija, trenerove

dodatne bilješke, te tip treninga (napadački, obrambeni, kondicijski) koji je prikazan drugačijom bojom pozadine zasebne kartice.

```
navi1111
@Composable
fun SurveySummaryCard(
    playerEffort: Float,
    playerConcentration: Float,
    coachesNotes: String,
    trainingType: String,
    survey: Survey,
    onCardClick: (survey: Survey) -> Unit
) {
    OutlinedCard(
        colors = CardDefaults.surveyCardContainerColor(trainingType = trainingType),
        border = BorderStroke(1.dp, color = MaterialTheme.colorScheme.outlineVariant),
        modifier = Modifier
            .fillMaxWidth()
            .padding(12.dp)
    ) { this: ColumnScope
        Text(
            text = "Team effort: ${String.format("%.1f",playerEffort * 100)} %",
            modifier = Modifier.padding(start = 8.dp, end = 8.dp, top = 8.dp, bottom = 4.dp),
            fontWeight = FontWeight.Bold,
            color = MaterialTheme.colorScheme.onSurface
        )
        Text(
            text = "Team concentration: ${String.format("%.1f", playerConcentration * 100)}%",
            modifier = Modifier.padding(horizontal = 8.dp, vertical = 4.dp),
            fontWeight = FontWeight.Bold,
            color = MaterialTheme.colorScheme.onSurface
        )
        Text(
            text = stringResource(R.string.coaches_notes),
            modifier = Modifier.padding(start = 8.dp),
            fontWeight = FontWeight.Bold,
            color = MaterialTheme.colorScheme.onSurface
        )
        Text(
            text = coachesNotes,
            modifier = Modifier.padding(8.dp),
            fontWeight = FontWeight.SemiBold,
            color = MaterialTheme.colorScheme.onSurface
        )
        Row(modifier = Modifier.fillMaxWidth(), horizontalArrangement = Arrangement.End) { this: RowScope
            TextButton(onClick = { onCardClick.invoke(survey) }) { this: RowScope
                Text(text = stringResource("Delete"))
            }
        }
    }
}
```

**Slika 4.8.** Prikaz funkcije koja stvara karticu prethodno ispunjene ankete

Na slici 4.9 je prikazana ekstenzijska funkcija kojom se mijenja boja pozadine kartice ovisno o tipu treninga koji je odrađen.

```
@Composable
fun CardDefaults.surveyCardContainerColor(trainingType: String): CardColors {
    return cardColors(
        containerColor = when (trainingType) {
            "Offensive" -> Color.Blue.copy( alpha: 0.2F)
            "Defensive" -> Color.Red.copy( alpha: 0.2F)
            "Physical" -> Color.Green.copy( alpha: 0.2F)
            else -> MaterialTheme.colorScheme.surface
        }
    )
}
```

**Slika 4.9.** Programski kod ekstenzijske funkcije za promjenu pozadinske boje kartice

Za prikaz drugačije pozadinske boje kartice s obzirom na tip treninga, napravljena je ekstenzijska funkcija koja s obzirom na dobiveni tip treninga koji je tipa *String*, vraća objekt tipa *CardColors*. Ako je trening napadački, vratit će plavu boju s koeficijentom prozirnosti 0.2, ako je obrambeni onda crvenu s istim koeficijentom prozirnosti, te ako je kondicijski zelenu s istim koeficijentom.

*@Composable* funkcija *SurveyListScreen* služi za prikaz različitih zaslona s obzirom na stanje pristiglih podataka. Takvu funkciju je bitno napisati iz više razloga. Jedan od glavnih razloga je što ako korisnik nije spremio niti jednu anketu u bazu podataka, a ova funkcija nije napisana. Onda bi samo bio prikazan prazan zaslon bez ikakve poruke i korisnik ne bi mogao znati je li došlo do greške pri dohvaćanju podataka, učitavaju li se podaci ili nema podataka za prikaz. Ovako kada je napravljena funkcija  koja će rukovati sa stanjem pristiglih podataka, pokriveni su svi slučajevi koji se mogu dogoditi pri dohvaćanju podataka iz baze. Kako bi bilo moguće koristiti takav pristup, jedan od najčišćih načina je pravljenjem zapečaćenog sučelja koja je prikazana na slici 4.10.

```
navi1111
sealed interface UIState {
    navi1111
    object Loading : UIState
    navi1111
    data class Success<T>(val data: T) : UIState
    navi1111
    object Error : UIState
    navi1111
    object EmptyListState : UIState
}
```

**Slika 4.10.** Zapečaćeno sučelje koje drži stanja korisničkog sučelja

Ovo sučelje će biti korišteno u svakom *viewmodelu* koji radi sa podacima koji moraju doći iz baze podataka ili vanjskog poslužitelja. Pokriveni su slučajevi učitavanja podataka, greške pri dohvaćanju podataka, dohvaćanje prazne liste podataka te uspješno dohvaćanje podataka.

Stanja prikazana na slici 4.10 će biti dodijeljena u *viewmodelu* što će i biti prikazano na slici 4.11.

```
navi1111 *
class SurveyListViewModel(private val surveyRepository: SurveyRepository) : ViewModel() {
    private var _uiState = MutableStateFlow<UIState>(UIState.Loading)
    val uiState = _uiState.asStateFlow()

    navi1111 *
    init {
        getAllSurveys()
    }

    navi1111 *
    fun getAllSurveys() {
        viewModelScope.launch { this: CoroutineScope
            try {
                surveyRepository.getAllSurveys().collect { it: List<Survey>
                    if (it.isEmpty()) {
                        _uiState.value = UIState.EmptyListState
                    } else {
                        _uiState.value = UIState.Success(data = it)
                    }
                }
            } catch (e: Exception) {
                _uiState.value = UIState.Error
            }
        }
    }

    navi1111
    fun deleteSurvey(survey: Survey) {
        viewModelScope.launch { this: CoroutineScope
            surveyRepository.deleteSurvey(survey)
        }
    }
}
```

**Slika 4.11.** Programski kod SurveyListViewModela

Kao i prethodno opisani *viewmodel*, i ovaj ima referencu na repozitorij. Nova stvar koja se pojavljuje u ovom *viewmodelu* je *Flow*. Prema [28], Kotlin *Flow* služi za emitiranje više uzastopnih vrijednosti u korutinama, za razliku od *suspend* funkcija koje vraćaju samo jednu vrijednost. Npr., *Flow* može biti korišten za trenutno ažuriranje podataka iz baze podataka ili vanjskog poslužitelja. U ovom slučaju je napravljena varijabla *_uiState* koja je privatna i koja će mijenjati svoju vrijednost. Ona je *Flow* promjenjivog stanja tipa *UIState*. Budući da se *Flowu* promjenjivog stanja mora postaviti početna vrijednost postavljena je vrijednost *UIState.Loading*. U *init* bloku *viewmodel*a odvija se poziv metode *getAllSurveys* iz repozitorija. U slučaju uspješnog

dohvaćanja podataka stanje se mapira u *UIState.Success* gdje se kao parametar predaju ti podaci. U slučaju da je lista prazna stanje se mapira u *UIState.EmptyListState*, a u slučaju iznimke (*engl. Exception*) stanje se mapira u *UIState.Error*.

U *viewmodel*u se još nalaze dvije funkcije. Funkcija *getAllSurveys* je već opisana u gornjem dijelu teksta, a funkcija *deleteSurvey* poziva funkciju iz repozitorija *deleteSurvey*. Kao parametar prima objekt *Survey* koji će biti obrisan.

### 4.2.3. Zaslon s osnovnim podacima igrača

Na ovome zaslonu se prikazuju igrači koji se dohvaćaju s vanjskog poslužitelja. S vanjskog poslužitelja se dohvaća lista igrača, a na zaslonu se prikazuju pomoću komponente *HorizontalPager*. *Horizontal Pager* djeluje slično kao *Lazy Column*, tako da mu se preda lista elemenata koji se želi prikazati, te onda u lambda funkciji rukuje kako će se svaki zasebni element liste prikazati. Programski kod zaslona je prikazan na slici 4.12.

```
navi1111 *
@OptIn(ExperimentalFoundationApi::class)
@Composable
fun PlayerListContent(playerList: List<Player>, onCardClick: (Player) -> Unit) {
    val pagerState = rememberPagerState()
    Column(
        modifier = Modifier
            .fillMaxSize()
            .background(MaterialTheme.colorScheme.background),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) { this: ColumnScope
        HorizontalPager(pageCount = playerList.count(), state = pagerState) { page ->
            PlayerCard(player = playerList[page]) { it: Player
                onCardClick(it)
            }
        }
        HorizontalPagerIndicator(
            pagerState = pagerState,
            pageCount = playerList.count(),
            modifier = Modifier.padding(vertical = 32.dp, horizontal = 16.dp)
        )
    }
}
```

**Slika 4.12.** Prikaz programskog koda funkcije *PlayerListContent*

Kao što je vidljivo na slici, *@Composable* funkcija *PlayerListContent* prima dva parametra, listu objekata *Player*, te *callback* ukoliko korisnik klikne na pojedinog igrača. Svaki igrač je prikazan pojedinačnom karticom na kojoj su prikazani njegovi osnovni podaci.

Podaci koji se prikazuju na kartici su prikazani pomoću teksta ili ikonice i teksta. Ikonica i tekst se nalaze u funkciji *PlayerAtribute*. Podaci koji se prikazuju su igračevo ime i prezime, datum

rođenja, broj dresa, broj postignutih pogodaka, asistencija, igrača utakmice te njegova pozicija. Funkcija *PlayerCard* koja prikazuje karticu pojedinačnog igrača prikazana je na slici 4.13.

```
@Composable
fun PlayerCard(player: Player, onPlayerClick: (Player) -> Unit) {
    OutlinedCard(
        onClick = { onPlayerClick.invoke(player) },
        modifier = Modifier
            .fillMaxWidth()
            .padding(horizontal = 12.dp)
    ) { this: ColumnScope
        Column(
            modifier = Modifier.fillMaxWidth(),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ) { this: ColumnScope
            Spacer(modifier = Modifier.height(40.dp))
            Text(
                text = player.name,
                textAlign = TextAlign.Center,
                fontWeight = FontWeight.Bold,
                fontSize = 32.sp,
                modifier = Modifier.padding(horizontal = 6.dp)
            )
            Text(
                text = player.surname,
                textAlign = TextAlign.Center,
                fontWeight = FontWeight.Bold,
                fontSize = 32.sp,
                modifier = Modifier.padding(horizontal = 6.dp)
            )
            Spacer(modifier = Modifier.height(26.dp))
            Text(text = "DOB: ${player.dateOfBirth}", fontStyle = FontStyle.Italic)
            Spacer(modifier = Modifier.height(26.dp))
            Row(
                horizontalArrangement = Arrangement.SpaceEvenly,
                modifier = Modifier.fillMaxWidth()
            ) { this: RowScope
                PlayerAttribute(
                    painterResource = R.drawable.playershirt,
                    playerValue = player.number.toString()
                )
                PlayerAttribute(
                    painterResource = R.drawable.baseline_sports_volleyball_24,
                    playerValue = player.goals.toString()
                )
                PlayerAttribute(
                    painterResource = R.drawable.player_position,
                    playerValue = player.position
                )
                PlayerAttribute(
                    painterResource = R.drawable.baseline_workspace_premium_24,
                    playerValue = player.motm.toString()
                )
                PlayerAttribute(
                    painterResource = R.drawable.assist,
                    playerValue = player.assists.toString()
                )
            }
            Spacer(modifier = Modifier.height(40.dp))
```

**Slika 4.13.** Prikaz funkcije *PlayerCard* koja prikazuje karticu igrača na korisničkom sučelju

Funkcionalnosti vezane uz ovaj zaslon su dohvaćanje liste igrača te klik na karticu koja drži podatke vezane uz određenog igrača što će i biti prikazano na slici 4.14.

```
navi1111
class PlayerListViewModel(private val playerListRepository: PlayerListRepository) : ViewModel() {

    private var _uiState = MutableStateFlow<UIState>(UIState.Loading)
    val uiState = _uiState.asStateFlow()
    navi1111
    init {
        getPlayerList()
    }

    navi1111
    fun getPlayerList() {
        viewModelScope.launch { this: CoroutineScope
            val response = playerListRepository.getPlayerList()
            mapPlayerListToState(response = response)
        }
    }

    navi1111
    private fun mapPlayerListToState(response: Resource) {
        when (response) {
            is Resource.Success<*> -> _uiState.value = UIState.Success(response.data)
            is Resource.Error -> _uiState.value = UIState.Error
        }
    }
}
```

**Slika 4.14.** Programski kod *PlayerListViewModela*

*PlayerListViewModel* dosta podsjeća na prethodni *viewmodel* zato što se dohvaćaju podaci i mapiraju s obzirom na uspješno ili neuspješno dohvaćanje. Dohvaćanje igrača se odvija pomoću mrežnog poziva koji će kasnije biti detaljno objašnjen, no iz *viewmodela* se vidi kako se radi o pozivu funkcije *getPlayerList* iz repozitorija. Kao i u prethodnom slučaju početna vrijednost stanja je *UIState.Loading* te uspješnim dohvaćanjem podataka prelazi u *UIState.Sucess* ili neuspješnim u *UIState.Error*.

Kao što je vidljivo na slici 4.15, pomoću bloka *when* se postavlja određeni zaslon s obzirom na stanje pristiglih podataka. *When* blok je sličan kao *switch case* u programskom jeziku Java, samo što je *when* blok jednostavniji za pisanje te će prikazivati grešku ako se ne pokriju svi slučajevi čime se programer osigurava od potencijalne greške.

```
navi1111
@OptIn(ExperimentalAnimationApi::class)
@Composable
@Destination
fun PlayerListScreen(navigator: DestinationsNavigator) {
    val playerListViewModel = koinViewModel<PlayerListViewModel>()
    val uiState by playerListViewModel.uiState.collectAsStateWithLifecycle()
    AnimatedContent(targetState = uiState) { this: AnimatedVisibilityScope  targetState ->
        when (targetState) {
            UIState.Error -> ErrorScreen { playerListViewModel.getPlayerList() }
            UIState.EmptyListState -> EmptyListScreen()
            UIState.Loading -> LoadingScreen()
            is UIState.Success<*> -> PlayerListContent(playerList = targetState.data as List<Player>) { it: Player
                navigator.navigate(
                    PlayerDetailsScreenDestination(it)
                )
            }
        }
    }
}
```

**Slika 4.15.** Prikaz funkcije *PlayerListScreen* koja pridružuje podatke zaslonu s karticama igrača

Funkcionalnost klika na pojedinu karticu igrača se ne odvija u *viewmodelu* nego u korisničkom sučelju zato što se radi o navigaciji na sljedeći zaslon gdje se kao parametar predaje objekt *Player* čija je kartica pritisnuta.

### 4.2.4. Zaslon s performansama igrača i mogućim poboljšanjima

Na ovaj zaslon moguće je doći tako da se pritisne na karticu na kojoj se nalaze osnovni podaci o igračima. Pritiskom na karticu pojedinog igrača navigacija vodi na sljedeći zaslon koji drži veliki broj podataka vezan za tog igrača. Korisničko sučelje koje prikazuje taj zaslon je prikazano slikom 4.16. Na zaslonu se nalazi slika igrača učitana pomoću biblioteke *Coil* koja služi za učitavanje poveznica slike u pravu sliku. U *Jetpack Composeu* se dobije korištenjem funkcije *AsyncImage* koja kao parametar prima model koji je zapravo poveznica fotografije koju je potrebno prikazati te *content description* koji je pomoć osobama koji imaju problema s vidom. Funkcija *PlayerDetailsContent* kao parametar prima objekt tipa Player pomoću kojeg će prikazati sve podatke, *paddingValues* što su vrijednosti koje je potrebno postaviti kao margine da se dio korisničkog sučelja ne bi preklapao, te *onImprovementsClick* koji je *callback* koji će se okinuti kada korisnik pritisne na gumb *„Improvements“.* Do objekta *Player* se došlo tako da je predan kao parametar u navigaciji među zaslonima.

```
@Composable
fun PlayerDetailsContent(player: Player, paddingValues: PaddingValues, onImprovementsClick: (Player) -> Unit) {
    Column(
        modifier = Modifier
            .fillMaxSize()
            .background(MaterialTheme.colorScheme.background)
            .padding(start = 16.dp, end = 16.dp, top = paddingValues.calculateTopPadding())
            .verticalScroll(
                rememberScrollState()
            ),
        horizontalAlignment = Alignment.CenterHorizontally
    ) { this: ColumnScope
        AsyncImage(model = player.image, contentDescription = null)
        Text(
            text = stringResource(id = "These are %1$s %2$s current stats, if you want to see p...", player.name, player.surname),
            fontWeight = FontWeight.SemiBold,
            fontStyle = FontStyle.Italic,
            fontSize = 16.sp
        )
        Row(modifier = Modifier.fillMaxWidth(), horizontalArrangement = Arrangement.End) { this: RowScope
            Button(onClick = { onImprovementsClick.invoke(player) }) { this: RowScope
                Text(text = stringResource("Improvements"))
            }
        }
        CategoryProgress(category = stringResource("Speed"), categoryProgress = player.speed.toFloat())
        CategoryProgress(category = stringResource("Stamina"), categoryProgress = player.stamina.toFloat())
        CategoryProgress(category = stringResource("Strength"), categoryProgress = player.strength.toFloat())
        CategoryProgress(category = stringResource("Ball control"), categoryProgress = player.ballControl.toFloat())
        CategoryProgress(
            category = stringResource("Concentration"),
            categoryProgress = player.concentration.toFloat()
        )
        CategoryProgress(category = stringResource("Creativity"), categoryProgress = player.creativity.toFloat())
        CategoryProgress(category = stringResource("Teamwork"), categoryProgress = player.teamwork.toFloat())
        CategoryProgress(category = stringResource("Shot Power"), categoryProgress = player.shotPower.toFloat())
```

**Slika 4.16.** Prikaz funkcije *PlayerDetailsContent* koja prikazuje detaljne podatke o igračima

Kao što je vidljivo iz slike 4.16, postoji veliki broj poziva funkcije *CategoryProgress*. *CategoryProgress* je *@Composable* funkcija koja služi za prikaz kvalitete igrača u određenoj igračkoj kategoriji. Napravljena je za prikaz kategorija u postotku od 0 do 100%. Temelj *CategoryProgress* funkcije je ugrađena komponenta *LinearProgressIndicator* koja je uređena prema potrebama ove aplikacije te za ljepši prikaz postotka kvalitete igrača u određenoj kategoriji.

```
navi1111 *
@Composable
fun CategoryProgress(category: String, categoryProgress: Float) {
    var progressPercentage = categoryProgress / 100
    Text(
        text = category,
        fontWeight = FontWeight.Bold,
        fontSize = 20.sp,
        modifier = Modifier.padding(vertical = 16.dp)
    )
    LinearProgressIndicator(
        modifier = Modifier
            .fillMaxWidth()
            .height(12.dp)
            .clip(RoundedCornerShape(6.dp))
            .animateContentSize(),
        color = progressBarColorPicker(categoryProgress),
        progress = progressPercentage
    )
    Text(
        text = "$categoryProgress%",
        textAlign = TextAlign.Right,
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 8.dp),
        fontWeight = FontWeight.Bold,
        fontSize = 18.sp
    )
    Divider(thickness = 1.dp, color = MaterialTheme.colorScheme.onBackground)
}
```

**Slika 4.17.** Prikaz funkcije *CategoryProgress*

*PlayerDetailsViewModel* koji je vezan uz ovaj zaslon je vrlo jednostavan s jednom funkcionalnošću, a to je dohvaćanje mogućih poboljšanja igrača koji se računaju u repozitoriju te će u daljnjem dijelu rada biti detaljnije obrađeni. Pritiskom na gumb *„Improvements“* doći će do poziva funkcije iz *viewmodel*a te će prikazati skočni prozor s tekstualnim objašnjenjem o mogućim poboljšanjima igrača. *PlayerDetailsViewModel* je vidljiv na slici 4.18.

```
navi1111
class PlayerDetailsViewModel(private val playerListRepository: PlayerListRepository) : ViewModel() {
    private var _improvements = MutableStateFlow( value: "")
    val improvements = _improvements.asStateFlow()

    navi1111
    fun getPlayerImprovements(player: Player) {
        viewModelScope.launch { this: CoroutineScope
            _improvements.value = playerListRepository.getPlayerImprovements(player)
        }
    }
}
```

**Slika 4.18.** Programski kod *PlayerDetailsViewModela*

Kao što je vidljivo iz slike 4.18, *PlayerDetailsViewModel* kao parametar prima *playerListRepository* te pomoću njega poziva funkciju *getPlayerImprovements* gdje se kao parametar u toj funkciji predaje objekt tipa *Player* čija poboljšanja u određenim kategorijama je potrebno prikazati.

### 4.2.5. Zaslon s performansama tima i mogućim poboljšanjima

Zaslon s performansama tima i mogućim poboljšanjima je treći zaslon koji se nalazi u *bottom* navigaciji. Na zaslonu su prikazane statističke kategorija tima te moguća poboljšanja. Slično kao detaljni prikaz igrača, no na ovom zaslonu se na strani poslužitelja događa dosta više računanja nego u slučaju s detaljima pojedinog igrača. Na slici 4.19 će biti prikazano korisničko sučelje kojim je prikazan ovaj zaslon.

```
navi1111 *
@Composable
private fun TeamStatsContent(playerStats: PlayerStats, onImprovementsClick: (PlayerStats) -> Unit) {
    Column(
        modifier = Modifier
            .fillMaxSize()
            .background(MaterialTheme.colorScheme.background)
            .verticalScroll(rememberScrollState())
            .padding(16.dp)
    ) { this: ColumnScope
        Text(
            text = stringResource("These are your teams currents stats, if you want to see..."),
            fontWeight = FontWeight.SemiBold,
            fontStyle = FontStyle.Italic,
            fontSize = 16.sp
        )
        Row(modifier = Modifier.fillMaxWidth(), horizontalArrangement = Arrangement.End) { this: RowScope
            Button(onClick = { onImprovementsClick(playerStats) }) { this: RowScope
                Text(text = stringResource(id = "Improvements"))
            }
        }
        CategoryProgress(category = stringResource(id = "Speed"), categoryProgress = playerStats.speed.toFloat())
        CategoryProgress(category = stringResource(id = "Stamina"), categoryProgress = playerStats.stamina.toFloat())
        CategoryProgress(category = stringResource(id = "Strength"), categoryProgress = playerStats.strength.toFloat())
        CategoryProgress(category = stringResource(id = "Ball control"), categoryProgress = playerStats.ballControl.toFloat())
        CategoryProgress(category = stringResource(id = "Concentration"), categoryProgress = playerStats.concentration.toFloat())
        CategoryProgress(category = stringResource(id = "Creativity"), categoryProgress = playerStats.creativity.toFloat())
        CategoryProgress(category = stringResource(id = "Teamwork"), categoryProgress = playerStats.teamwork.toFloat())
        CategoryProgress(category = stringResource(id = "Shot Power"), categoryProgress = playerStats.shotPower.toFloat())
```

**Slika 4.19.** Prikaz funkcije *TeamStatsContent* za prikaz timskih statistika na korisničkom sučelju

Kao što je vidljivo iz slike 4.19, sadržaj je sličan zaslonu s detaljnim informacijama o pojedinom igraču. Funkcija *CategoryProgress* se poziva više puta kako bi se prikazao ukupni učinak tima u određenoj kategoriji. Također, pritiskom na gumb „*Improvements*“ prikazat će se moguća poboljšanja te na kojim stvarima je potrebno raditi kako bi se tim poboljšao u određenoj kategoriji.


**Slika 4.20.** Prikaz funkcije *TeamStatsScreem* koja pridružuje podatke korisničkom sučelju

Kao i na prethodnim zaslonima, također je napravljeno potrebno osiguranje kako, u slučaju greške pri pristizanju podataka ne bi došlo do prestanka rada aplikacije. Na slikama 4.21, 4.22 i 4.23 će biti prikazani zasloni u slučaju greške pri dohvaćanju podataka, dohvaćanju prazne liste ili pri učitavanju podataka.


**Slika 4.21.** Prikaz funkcije *ErrorScreen* za prikazivanje greške na korisničkom sučelju

```
@Composable
fun EmptyListScreen() {
    Column(
        modifier = Modifier
            .fillMaxSize()
            .background(MaterialTheme.colorScheme.background),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ) { this: ColumnScope
        Icon(
            painter = painterResource(id = R.drawable.baseline_block_24),
            contentDescription = null,
            modifier = Modifier.size(128.dp)
        )
        Text(
            text = stringResource(R.string.there_aren_t_any_saved_training_sessions),
            textAlign = TextAlign.Center,
            fontWeight = FontWeight.Bold,
            color = MaterialTheme.colorScheme.onBackground,
            fontSize = 24.sp
        )
    }
```

**Slika 4.22.** Prikaz funkcije *EmptyListScreen* za prikaz prazne liste na korisničkom sučelju

```
@Composable
fun LoadingScreen() {
    Column(
        verticalArrangement = Arrangement.Center,
        modifier = Modifier
            .fillMaxSize()
            .background(MaterialTheme.colorScheme.background),
        horizontalAlignment = Alignment.CenterHorizontally
    ) { this: ColumnScope
        CircularProgressIndicator(
            color = MaterialTheme.colorScheme.onBackground
        )
    }
}
```

**Slika 4.23.** Prikaz funkcije LoadingScreen za prikaz dok se podaci učitavaju

### 4.2.6. Navigacija unutar aplikacije

U ovome potpoglavlju rada opisat će se navigacija unutar aplikacije te između zaslona aplikacije. Za navigaciju unutar aplikacije je korištena biblioteka *Compose Destinations*. *Compose Destinations* je biblioteka koja pomoću anotacija u kodu u pozadini generira kod koji koristi službenu *Jetpack Compose* Navigaciju. Zahvaljujući ovoj biblioteci, nema više potrebe za pisanjem zamarajućeg koda koji se mora napisati kako bi službena *Compose* navigacija funkcionirala. Ovako, ova biblioteka to odradi i moguće je fokusirati se na ostale izazove u aplikaciji. Navigacija je vrlo jednostavna pomoću ove biblioteke te vrlo razumljiva. Biblioteku razvija Rafael Costa. Jedna od prednosti ove biblioteke je to što je *type safe*, što znači da kada se neki objekt želi poslati kao argument navigacije s ovom bibliotekom je to moguće bez dodatnog pisanja koda.

Kako bi bilo moguće koristiti ovu biblioteku potrebno je prvo dodati ovisnosti u *gradle* aplikacije. Gradle ovisnosti će biti dodane na slici 4.24.

```
//destinations compose
implementation "io.github.raamcosta.compose-destinations:core:$destinations_compose_version"
ksp "io.github.raamcosta.compose-destinations:ksp:$destinations_compose_version"
```

**Slika 4.24.** Prikaz dodanih ovisnosti za Compose Destinations biblioteku

Nakon toga potrebno je definirati *@Composable* funkcije koje će biti dio navigacije tako da im se na vrhu doda anotacija *@Destination* i jedna od funkcija se mora postaviti kao startna destinacija, što je i prikazano na slici 4.25.

```
@RootNavGraph(start = true)
@Composable
@Destination
fun SurveyScreen(navController: DestinationsNavigator) {
```

**Slika 4.25.** Prikaz anotacije za startnu rutu navigacijskog grafa

Ovako se zna da kada se postavi navigacijski graf u aplikaciji da će *SurveyScreen* biti početna destinacija te prvi zaslon koji će se otvoriti pri pozivu navigacijskog grafa. Kada se pokrene građenje projekta ove klase će biti stvorene. Nakon toga potrebno je definirati navigacijski graf koji će držati zaslone koji će biti dio navigacije. Na slici 4.26 je vidljivo da će *MainActivity* držati navigacijski graf koji će onda držati sve ostale zaslone aplikacije.

```
navi1111
class MainActivity : ComponentActivity() {
    navi1111
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            TrainingHelperTheme {
                DestinationsNavHost(navGraph = NavGraphs.root)
            }
        }
    }
}
```

**Slika 4.26.** Prikaz poziva DestinationsNavHost

Programski kod prikazan na slici 4.27 predstavlja zaslon koji drži donju navigaciju aplikacije. Donja navigacija (*eng. Bottom navigation*) ima svoj zasebni navigacijski kontroler koji služi za navigaciju među zaslonima. Zasloni koji se nalaze u donjoj navigaciji su *SurveyListScreen*, *PlayerListScreen* te *TeamStatsScreen*. Ukoliko iz jednog od tri nabrojana zaslona želite izaći iz donje navigacije, potrebno je imati pristup destinacijskom navigatoru koji je navigacijski kontroler osnovne navigacije.

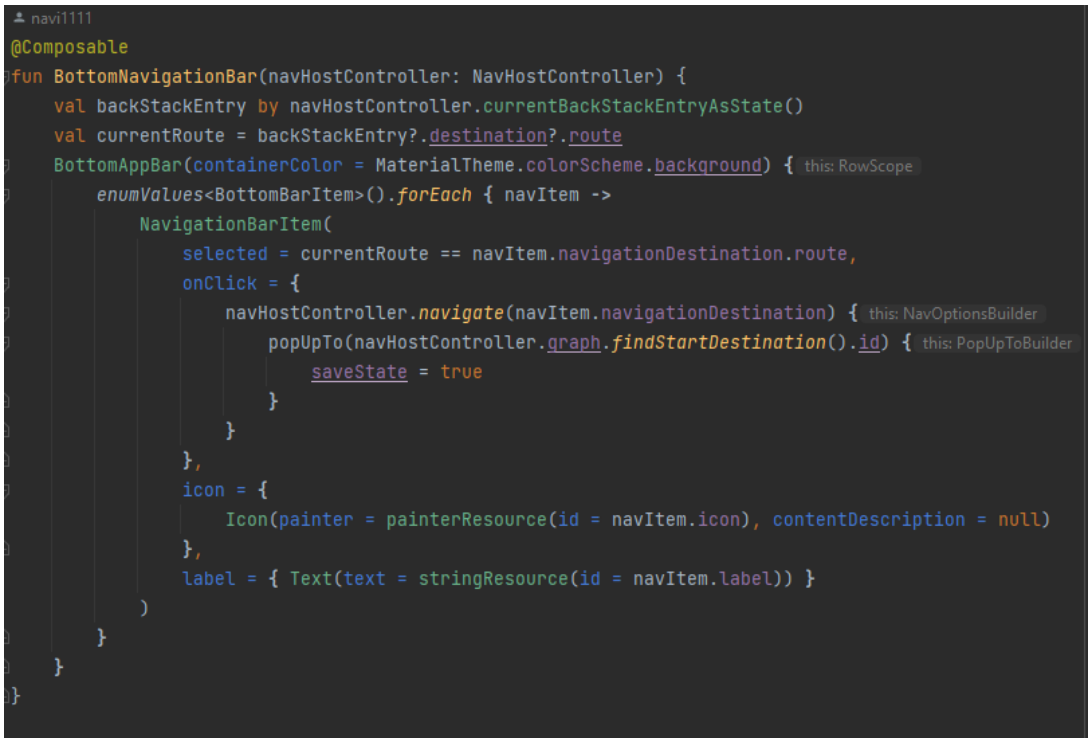
```
@Destination
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun NavBarContainerScreen(destinationsNavigator: DestinationsNavigator) {
    val navController: NavHostController = rememberNavController()
    Scaffold(
        bottomBar = { BottomNavigationBar(navHostController = navController) },
        floatingActionButton = {
            FloatingActionButton(onClick = { destinationsNavigator.navigate(SurveyScreenDestination) }) {
                Icon(painter = painterResource(id = R.drawable.outline_add_24), contentDescription = null)
            }
        },
        floatingActionButtonPosition = FabPosition.End
    ) { it: PaddingValues
        BottomNavBarGraph(navHostController = navController, it, destinationsNavigator)
    }
}

navi1111
@Composable
fun BottomNavBarGraph(navHostController: NavHostController, paddingValues: PaddingValues, destinationsNavigator: DestinationsNavigator) {
    DestinationsNavHost(
        navGraph = NavGraphs.root,
        startRoute = SurveyListScreenDestination,
        navController = navHostController,
        modifier = Modifier.padding(bottom = paddingValues.calculateBottomPadding())
    ) { this: ManualComposableCallsBuilder
        composable(SurveyListScreenDestination) { this: DestinationScope<Unit>
            SurveyListScreen()
        }
        composable(PlayerListScreenDestination) { this: DestinationScope<Unit>
            PlayerListScreen(destinationsNavigator)
        }
        composable(TeamStatsScreenDestination) { this: DestinationScope<Unit>
            TeamStatsScreen()
```

**Slika 4.27.** Prikaz funkcija za stvaranje *bottom* navigacije i njenog navigacijskog grafa

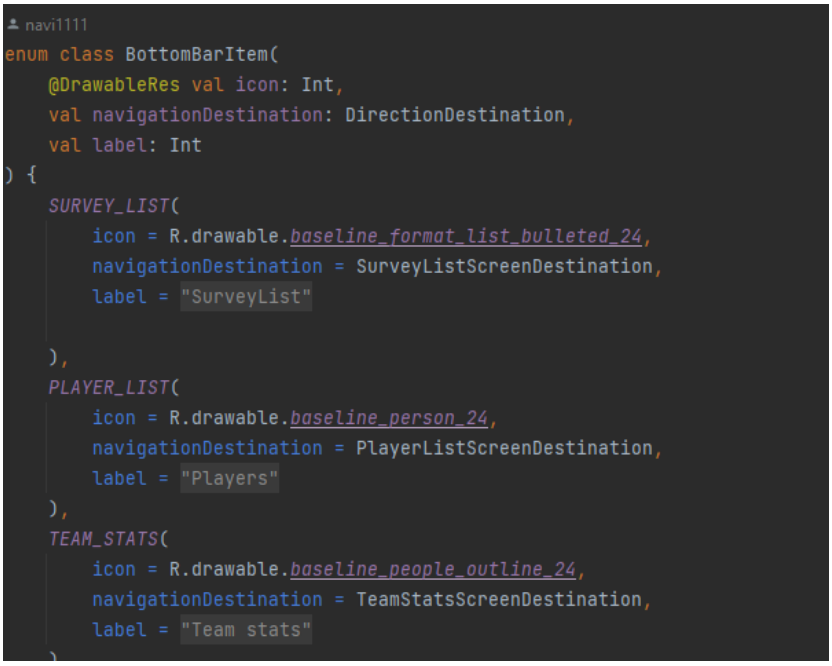
Na slici 4.28 prikazana je *@Composable* funkcija koja stvara donju navigaciju. Iz slike je vidljivo kako se poziva ugrađena komponenta *BottomAppBar* u kojoj se postavljaju elementi donje navigacije. Ako se obrati pažnja na *onClick* vidljivo je kako se odvija prijelaz između zaslona u

donjoj navigaciji, klikom na element donje navigacije prijelazi se na zaslon koji taj element predstavlja. Također, vidljivo je kako donja navigacija ima svoj zasebni navigacijski kontroler.




**Slika 4.28.** Prikaz dodjeljivanja destinacija, ikona i naslova bottom navigaciji

No, pitanje je kako donja navigacija zna koje vrijednosti treba postaviti te ikonice i naslove. Zato je napravljena *enum* klasa *BottomBarItem* koja u sebi drži konstante koje imaju svoju ikonu, naslov te navigacijsku destinaciju kao što je i prikazano na slici 4.29.




**Slika 4.29.** Prikaz elemenata *bottom* navigacije

## 4.3. Programsko rješenje na strani poslužitelja

U ovome poglavlju rada bit će opisana postignuta programska rješenja na strani poslužitelja pomoću kojih su pripremljeni podaci kojima je omogućen rad ove mobilne aplikacije za upravljanje performansama nogometnog kluba.

### 4.3.1. Baza podataka Room

Prema [29], baza podataka *Room* je biblioteka koja dio Android Jetpacka, predstavlja apstrakciju na *SQLite* bazom podataka. Služi za lokalno spremanje podataka u aplikaciji. *Room* baza podataka radi pomoću anotacija. Anotacije služe za generiranje programskog koda „ispod haube“ odnosno programskog koda koji nije vidljiv programeru. Pomoću anotacija se stvaraju upiti (*eng. Query*) za manipulaciju nad podacima u bazi podataka. Kako bi se koristila Room baza podataka potrebno je dodati ovisnosti u *gradle* aplikacije. Na slici 4.30 prikazane su dodane ovisnosti u gradle datoteku aplikacije.

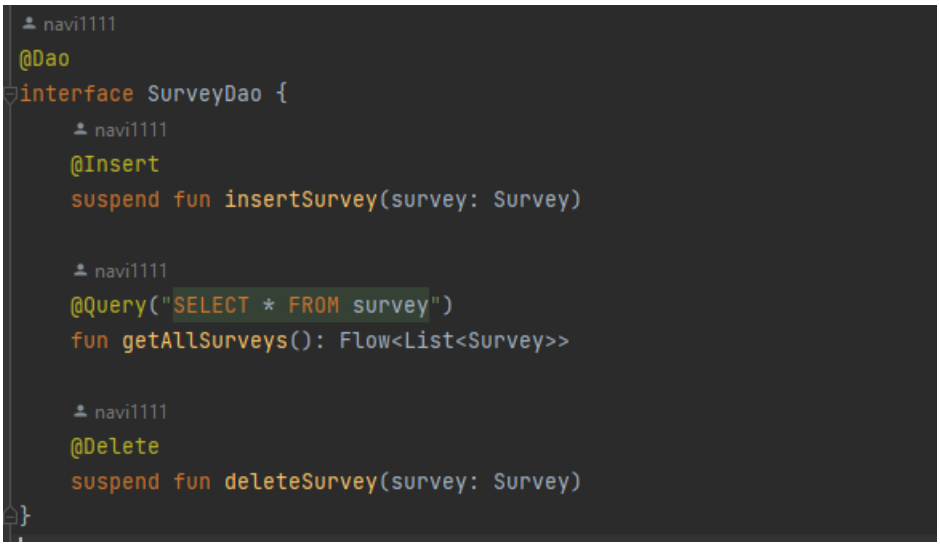
```
// room database
implementation "androidx.room:room-runtime:$room_version"
ksp "androidx.room:room-compiler:$room_version"
implementation "androidx.room:room-ktx:$room_version"
```

**Slika 4.30.** Prikaz dodanih ovisnosti za biblioteku *Room*

Nakon dodanih ovisnosti potrebno je napraviti apstraktnu klasu koja će predstavljati bazu podataka te sučelje DAO (eng. *Data Access Object*) koje sadrži metode pomoću kojih se manipulira podacima u bazi podataka. Iz slika 4.31 i 4.32 je vidljiva definicija baze podataka te sučelja DAO te ranije navedenih anotacija kako bi program mogao prepoznati njihove uloge te definirati programski kod.

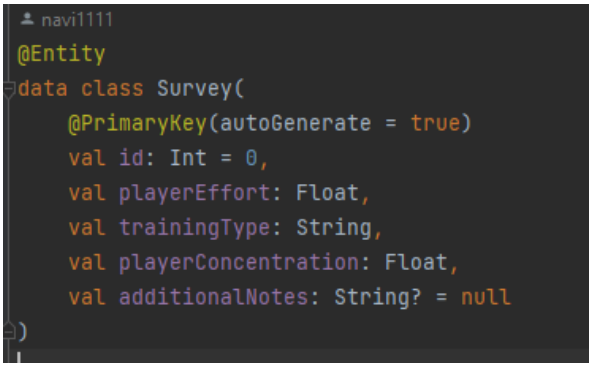
```
navi1111
@Database(entities = [Survey::class], version = 1, exportSchema = false)
abstract class AppDatabase : RoomDatabase() {
    navi1111
    abstract fun surveyDao(): SurveyDao
}
```

**Slika 4.31.** Prikaz apstraktne klase *AppDatabase*




**Slika 4.32.** Prikaz sučelja *SurveyDao*

Također, potrebno je definirati i model po kojem će se spremati podaci u bazu podataka. Ako se koristi *Room* baza podataka, taj model se naziva entitet i također je anotiran, što je vidljivo iz slike 4.33.




**Slika 4.33.** Prikaz podatkovne klase i entiteta *Survey*

Ako je klasa anotirana s *@Entity* potrebno je imati člana koji će biti anotiran s anotacijom *@PrimaryKey*. Primarni ključ je poseban stupac u bazi podataka koji je jedinstven i ne može sadržavati vrijednosti koje mogu biti *null*. Podatkovna klasa *Survey* služi kao model po kojem će se spremati podaci u bazu podataka. Kao što je vidljivo iz metoda u DAO sučelju, metoda koja ima anotaciju *insert* mora kao parametar predati objekt tipa *Survey* koji se želi spremiti u bazu podataka. Tako je i za *delete*, kao parametar se mora predati objekt tipa *Survey* koji se želi obrisati iz baze podataka. Još jedan razlog čestog korištenja room baze podataka u razvoju Android aplikacija je integracija s korutinama te flowom i live data-om što je vidljivo iz DAO sučelja te ključnih riječi *suspend* i *Flow*.

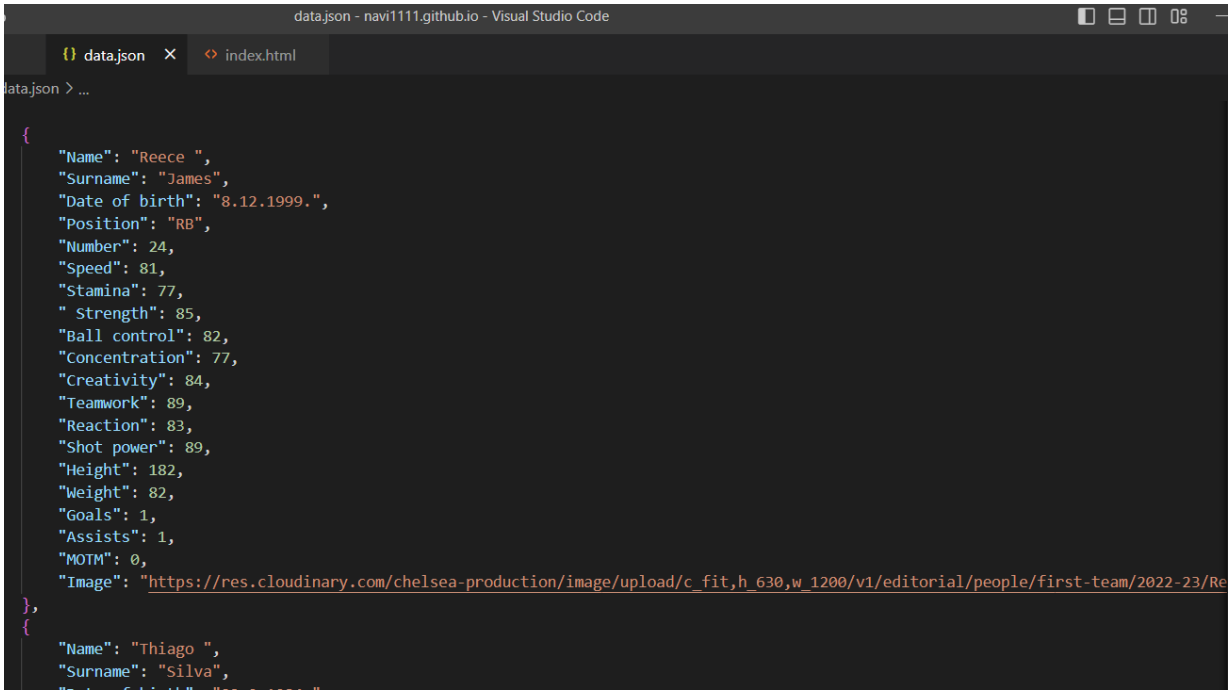
Naravno, DAO je samo sučelje, što znači da je metode iz DAO-a potrebno negdje pozvati. Prema MVVM arhitekturi i *repository* obrascu to će biti napravljeno u repozitoriju.


**Slika 4.34.** Prikaz klase SurveyRepository

*SurveyRepository* kao parametar prima *surveyDao* na kojemu zove metode za spremanje elementa u bazu podataka, brisanje elementa iz baze podataka te dohvaćanje svih elemenata baze podataka. Ove funkcije iz repozitorija se kasnije pozivaju u *viewmodelu* što je prikazano u prethodnim poglavljima.

### 4.3.2. Postavljanje podataka na vanjsku uslugu

Budući da za izradu ove aplikacije su bili potrebni podaci o nogometašima i njihovim karakteristikama, na internetu nije bilo moguće pronaći gotov API koji će zadovoljiti potrebe ove aplikacije. Stoga su podaci skupljeni s različitih stranica te je vanjski usluga napravljen od temelja. Kao što je i opisano u potpoglavlju 3.2.4 podaci su pronađeni na više web stranica te pripremljeni za ovu aplikaciju. Podaci su pripremljeni u obliku JSON filea. JSON za ovu aplikaciju prikazuje polje objekata *Player*. Za ovu aplikaciju je iskorištena opcija na *Githubu* koji nudi usluge poslužitelja za jednu stranicu po napravljenom *Github* profilu. Stoga je pomoću okruženja *Visual Studio Code* bilo potrebno napraviti stranicu s JSON podacima. Na slici 4.35 je prikazan JSON s podacima o igračima koji su korišteni u aplikaciji.




**Slika 4.35.** Prikaz JSON podataka u okruženju *Visual Studio Code*

Kada se ovaj projekt iz *Visual Studio Codea* postavi na *Github* pod imenom username.github.io ta stranica postaje svima dostupna. Stoga kako bi se pristupilo ovim podacima potrebno je pristupiti poveznici https://navi1111.github.io/data/data.json.

### 4.3.3. Programska implementacija dohvaćanja podataka s vanjskog poslužitelja

Nakon što su podaci podignuti na vanjsku uslugu, potrebno ih je dohvatiti, prilagoditi svojim potrebama te prikazati u aplikaciji. U ovoj aplikaciji dohvaćanje podataka s vanjskog poslužitelja će biti izvršeno pomoću biblioteke *Retrofit.* Kako bi bilo moguće koristiti biblioteku *Retrofit*, potrebno je dodati ovisnosti u *gradle* aplikacije što je prikazano na slici 4.36.

```
//kotlinx serialization
implementation "org.jetbrains.kotlinx:kotlinx-serialization-json:$serialization_version"

//retrofit
implementation "com.squareup.retrofit2:retrofit:2.9.0"

//retrofit serialization converter
implementation "com.jakewharton.retrofit:retrofit2-kotlinx-serialization-converter:0.8.0"
```

**Slika 4.36**. Prikaz dodanih ovisnosti za biblioteku Retrofit

Nakon dodavanja ovisnosti potrebno je promotriti podatke koji su na vanjskoj usluzi te s obzirom na te podatke, napraviti podatkovnu klasu pomoću koje će se podaci s interneta serijalizirati u objekte.

```kotlin
@Serializable
data class Player(
    @SerialName("Name")
    val name: String,
    @SerialName("Surname")
    val surname: String,
    @SerialName("Date of birth")
    val dateOfBirth: String,
    @SerialName("Position")
    val position: String,
    @SerialName("Number")
    val number: Int,
    @SerialName("Speed")
    val speed: Int,
    @SerialName("Stamina")
    val stamina: Int,
    @SerialName(" Strength")
    val strength: Int,
    @SerialName("Ball control")
    val ballControl: Int,
    @SerialName("Concentration")
    val concentration: Int,
    @SerialName("Creativity")
    val creativity: Int,
    @SerialName("Teamwork")
    val teamwork: Int,
    @SerialName("Reaction")
    val reaction: Int,
    @SerialName("Shot power")
    val shotPower: Int,
    @SerialName("Height")
    val height: Int,
    @SerialName("Weight")
    val weight: Int,
    @SerialName("Goals")
    val goals: Int,
    @SerialName("Assists")
    val assists: Int,
    @SerialName("MOTM")
    val motm: Int,
    @SerialName("Image")
    val image: String
```

**Slika 4.37.** Prikaz podatkovne klase Player

Budući da se radi s Kotlin serijalizacijskim alatom, potrebno je klasu označiti s anotacijom *@Serializable*. Podaci u ovoj klasi će kasnije biti prikazani na zaslonu. Nakon što je napravljena podatkovna klasa, potrebno je kreirati sučelje koje će djelovati kao API poslužitelj. Kao što je vidljivo iz slike 4.38, *Retrofit* ima integraciju s korutinama.

```kotlin
navi1111
interface ApiService {
    navi1111
    @GET("/data/data.json")
    suspend fun getPlayerList(): List<Player>
}
```

**Slika 4.38.** Prikaz sučelja ApiService za rad s vanjskom uslugom

To je vidljivo iz ključne riječi suspend. Funkcija getPlayerList će vratiti listu objekata Player. Također, vidljivo je da *Retrofit*, kao i Room baza podataka, koristi anotacije. Anotacije označuju metode s *HTTP* akcijama kao što su: *GET, POST, DELETE, PUT*. U ovom primjeru *GET* je

dovoljan zato što se podaci samo dohvaćaju, nema potrebe za uređivanjem podataka na vanjskom poslužitelju. *GET* kao parametar prima dio putanje nakon osnovne poveznice s kojeg se dohvaćaju podaci.

Kada je kreirano sučelje API poslužitelja, potrebno je kreirati instancu *Retrofita*. To će biti napravljeno pomoću ubrizgavanja ovisnosti pomoću biblioteke *Koin* koja će biti u daljnjem dijelu rada detaljno objašnjena. Na slici 4.39 prikazano je kreiranje instance *Retrofita* pomoću *Koina*.

```
@OptIn(ExperimentalSerializationApi::class)
val networkModule = module { this: Module
    single { get<Retrofit>().create(ApiService::class.java) }
    single { Json { ignoreUnknownKeys = true } }
    single { this: Scope  it: ParametersHolder
        Retrofit
            .Builder()
            .baseUrl(ApiRoutes.BASE_URL)
            .addConverterFactory(get<Json>().asConverterFactory("application/json".toMediaType()))
            .build()
```

**Slika 4.39.** Prikaz obrasca Graditelj za stvaranje instance Retrofita

Instanca *Retrofita* se kreira pomoću obrasca Graditelj (eng. *Builder*) prema kojemu se funkcije pozivaju lančano. Potrebno je dodati osnovnu poveznicu, tvornicu koja će pretvoriti podatke u objekte te kreirati instancu pomoću API poslužitelja.

Na slici 4.40 prikazana je implementacija repozitorija u kojemu je implementirana funkcija za dohvaćanje podataka s vanjskog poslužitelja.

```
class PlayerListRepository(private val apiService: ApiService) {

    navi1111 *
    suspend fun getPlayerList(): Resource {
        return try {
            withContext(Dispatchers.IO) { this: CoroutineScope
                val response = apiService.getPlayerList()
                if(response.isEmpty()){
                    Resource.EmptyData ^withContext
                }else{
                    Resource.Success(response) ^withContext
                }
            }
        } catch (e: Exception) {
            Resource.Error
        }
    }
```

**Slika 4.40.** Prikaz klase PlayerListRepository te funkcije za dohvaćanje igrača s vanjskog poslužitelja

Klasa *PlayerListRepository* kao parametar prima *apiService*, zato što će se kasnije pozivati funkcija iz *apiServicea*. Upravo to se događa u metodi *getPlayerList* gdje se vidi da se poziva metoda api usluga *getPlayerList*. I u ovom slučaju pomoću klase *Resource* aplikacija je sigurna od potencijalnog neuspješnog dohvaćanja podataka te prestanka rada aplikacije tako da ukoliko dođe do neuspjelog dohvaćanja podataka, prikazat će se zaslon s prikladnom porukom. Izgled klase *Resource* će biti prikazan na slici 4.41.

```
navi1111 *
sealed class Resource {
    navi1111
    data class Success<T>(val data: T) : Resource()
    navi1111
    object Error : Resource()
    new *
    object EmptyData : Resource()
}
```

**Slika 4.41.** Prikaz klase Resource koja drži stanja pri dohvaćanju podataka

### 4.3.4. Programska implementacija ubrizgavanja ovisnosti

Ubrizgavanje ovisnosti je tehnika u programiranju koja služi za polaganje temelja dobre arhitekture programskog koda. Implementiranje ubrizgavanja ovisnosti omogućava ponovnu uporabu programskog koda, lakše refaktoriranje programskog koda te lakše testiranje programskog koda [30]. U ovoj aplikaciji korišten je alat za ubrizgavanje ovisnosti koji se zove *Koin*. *Koin* je razvojni alat za ubrizgavanje ovisnosti u aplikacijama koje koriste Kotlin kao programski jezik. To mogu biti Android aplikacije, *Kotlin Multiplatform* aplikacije te *Ktor* poslužiteljske aplikacije. *Koin* je razvila tvrtka *Kotzilla* te open-source programeri [31].

Kako bi bilo moguće koristit *Koin* potrebno je dodati ovisnosti u *gradle* aplikacije te kreirati aplikacijsku klasu i tamo ga pokrenuti. Na slikama 4.42 i 4.43 je prikazano dodavanje ovisnosti u gradle datoteku projekta i pokretanje Koina u aplikacijskoj klasi ove aplikacije.

```
//koin
implementation "io.insert-koin:koin-android:$koin_android_version"
implementation "io.insert-koin:koin-androidx-compose:$koin_android_version"
```

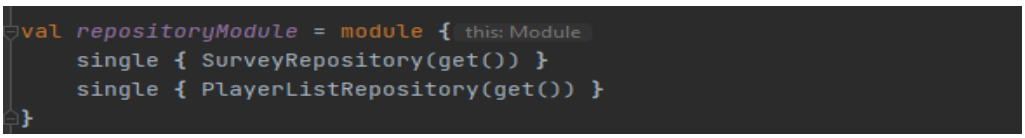
**Slika 4.42.** Prikaz dodanih ovisnosti biblioteke Koin


**Slika 4.43.** Prikaz glavne aplikacijske klase u kojoj se pokreće Koin

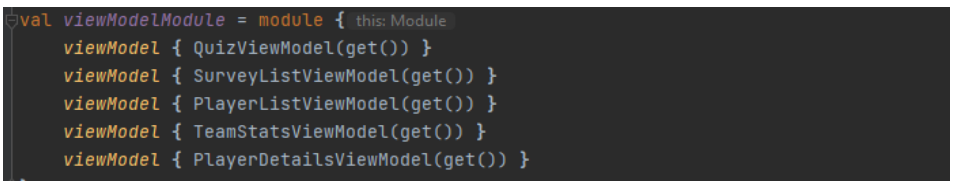
Za pokretanje *Koina* potrebno je definirati kontekst aplikacije te module koji će biti pokrenuti s *Koinom*. Te module je potrebno kreirati. Na slikama 4.44, 4.45, 4.46 i 4.47 će biti prikazani svi moduli potrebni za rad ove aplikacije.


**Slika 4.44.** Prikaz modula baze podataka


**Slika 4.45.** Prikaz modula vanjskog poslužitelja




**Slika 4.46.** Prikaz modula repozitorija




**Slika 4.47.** Prikaz modula ViewModela

U *Koinu* pisanjem ključne riječi *get(), Koin* će sam znati što mora predati kao parametar, budući da je to već definirano kao parametar konstruktora u svim klasama repozitorija i *viewmodela*.

### 4.3.5. Programska implementacija algoritma za procjenu poboljšanja igrača i tima

U ovom dijelu će biti prikazana obrada podataka pristiglih s vanjskog poslužitelja te prilagođena prema potrebama ove aplikacije. Prikazat će se računanje timskih statistika te algoritam za procjenu poboljšanja igrača. Algoritam za procjenu poboljšanja igrača i timskih statistika opisan je potpoglavljem 3.2.6. Na slici 4.48 prikazana je funkcija *calculateAverageStats* kojom se računaju statistike tima tako da se napravi prosjek statistika igrača. Zbroje se sve statistike igrača i podjele s brojem igrača i tako se dobiju timske statistike.

```
private suspend fun calculateAverageStats(playerList: List<Player>): PlayerStats {
    var speed = 0
    var stamina = 0
    var strength = 0
    var ballControl = 0
    var concentration = 0
    var creativity = 0
    var teamwork = 0
    var reaction = 0
    var shotPower = 0
    withContext(Dispatchers.Default) { this: CoroutineScope
        for (player in playerList) {
            speed += player.speed
            stamina += player.stamina
            strength += player.strength
            ballControl += player.ballControl
            concentration += player.concentration
            creativity += player.creativity
            teamwork += player.teamwork
            reaction += player.reaction
            shotPower += player.shotPower
        }
    }
    speed /= playerList.count()
    stamina /= playerList.count()
    strength /= playerList.count()
    ballControl /= playerList.count()
    concentration /= playerList.count()
    creativity /= playerList.count()
    teamwork /= playerList.count()
    reaction /= playerList.count()
    shotPower /= playerList.count()

    return PlayerStats(
        speed,
        stamina,
        strength,
        ballControl,
        concentration,
        creativity,
        teamwork,
        reaction,
        shotPower
```

**Slika 4.48.** Prikaz funkcije za računanje timskih statistika

Prikazana funkcija *getImprovements* će vratiti poruku tipa *String* s mogućim poboljšanjima tima. Funkcija će vratiti određenu poruku s obzirom na vrijednosti statističkih parametara igrača ili tima. Na primjer ako je ukupna brzina tima manja od 70, funkcija će vratiti poruku da je potrebno raditi na brzini tima tako da se poveća broj kondicijskih treninga. Moguća poboljšanja su postavljena za šest parametara: brzina, izdržljivost, snaga, koncentracija, kreativnost i timski rad. Za svaku kategoriju je preporučena i vrsta treninga kojom bi se vrijednosti mogle poboljšati. Ova funkcija se nadalje poziva u *viewmodelu* koji onda delegira tu poruku na korisničko sučelje. Vrlo slična funkcija *getPlayerImprovements* radi istu stvar za statističke parametre igrača, ali s nešto drugačijim porukama. Moguća poboljšanja su vidljiva na zaslonima s detaljnim podacima o igraču te na zaslonu koji prikazuje timske statističke parametre. Na slici 4.49 je prikazan dio programskog koda funkcije *getImprovements*.

```
suspend fun getImprovements(playerStats: PlayerStats): String {
    val currentResponse = StringBuilder()
    withContext(Dispatchers.Default) {
        when (playerStats.speed) {
            in 0..69 -> currentResponse.append("You need to improve your team's speed as much as you can. You can do it with extensive physi
            in 70..85 -> currentResponse.append("Try improving your team's speed a bit more. You should have more condition training.\n")
            else -> currentResponse.append("Your speed is on high level, keep doing what you do.\n")
        }
        when (playerStats.stamina) {
            in 0..69 -> currentResponse.append("You need to improve your team's stamina level as much as you can. You can do it with long ph
            in 70..85 -> currentResponse.append("Try improving your stamina a bit more. Work with your strength and conditioning coach.\n")
            else -> currentResponse.append("Your stamina is on high level, keep doing what you do.\n")
        }
        when (playerStats.strength) {
            in 0..69 -> currentResponse.append("You need to improve your strength as much as you can. You should go to clubs gym and work wi
            in 70..85 -> currentResponse.append("Try improving your strength a bit more. Visit our clubs gym.\n")
            else -> currentResponse.append("Your strength is on high level, keep doing what you do.\n")
        }
        when (playerStats.concentration) {
            in 0..69 -> currentResponse.append("You should really work on your concentration level. Your team lacks focus\n")
            in 70..85 -> currentResponse.append("More focus would win you more games.\n")
            else -> currentResponse.append("Your concentration is on high level, keep doing what you do.\n")
        }
        when (playerStats.creativity) {
            in 0..69 -> currentResponse.append("You team needs more imagination in your game.\n")
            in 70..85 -> currentResponse.append("Try thinking outside of the box.\n")
            else -> currentResponse.append("Your creativity is on high level, keep doing what you do.\n")
        }
```

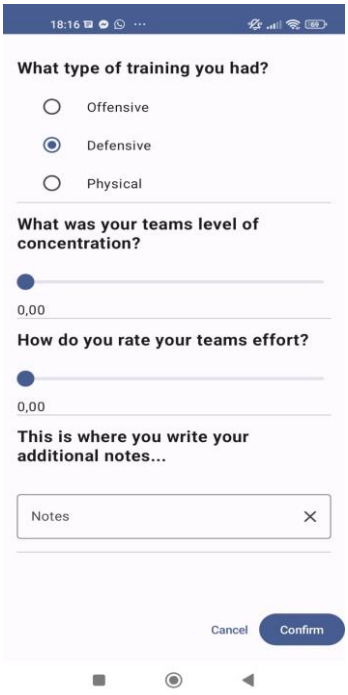
**Slika 4.49.** Prikaz funkcije koja računa moguća poboljšanja s obzirom na predani parametar funkcije

# 5. PRIKAZ NAČINA RADA APLIKACIJE, ISPITIVANJE I ANALIZA REZULTATA

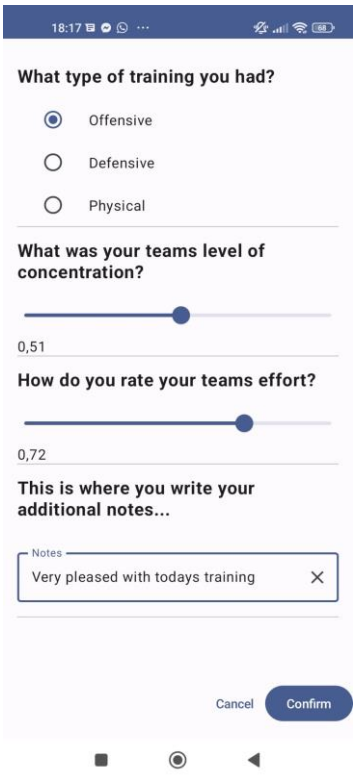
Ovim poglavljem rada će biti prikazan rad mobilne aplikacije za upravljanje performansama nogometnog kluba te upute za korištenje same aplikacije. Prvo će biti prikazan tok aplikacije kako bi korisnik znao koje sve značajke može koristiti unutar ove aplikacije. Nakon prikazivanja toka aplikacije, prikazat će se sve funkcionalnosti aplikacije te rubni slučajevi pri korištenju ove aplikacije.

## 5.1. Upute za korištenje aplikacije za upravljanje performansama nogometnog kluba

Ulaskom u aplikaciju prvo se prikazuje zaslon za ispunjavanje ankete nakon odrađenog treninga. Kao i sama aplikacija, anketa je namijenjena treneru za ispunjavanje. Trener može, ali i ne mora odmah ispuniti anketu. Ako je ispuni anketa će se prikazati na sljedećem zaslonu na listi prethodno ispunjenih anketa. Na slici 5.1 je prikazana ne ispunjena anketa.

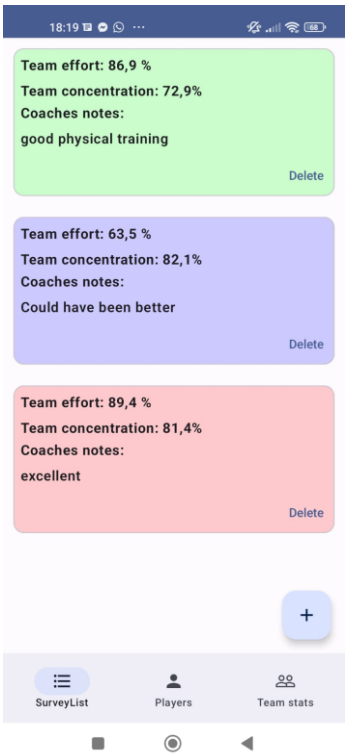



Slika 5.1. Prikaz zaslona s anketom za trenera




Slika 5.2. Prikaz ispunjene ankete za trenera

Nakon ispunjavanja ankete i pritiska na gumb „*Confirm*“ prijelazi se na sljedeći zaslon, a to je zaslon s prethodno ispunjenim anketama koji je prikazan na slici 5.3.



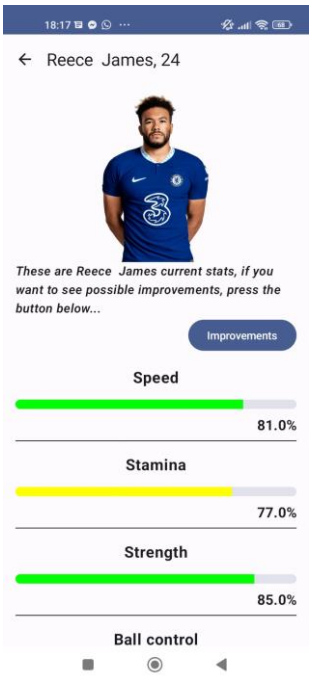

**Slika 5.3.** Prikaz liste prethodno ispunjenih anketa

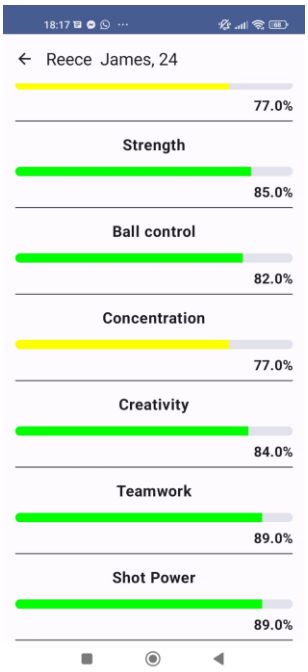
Na zaslonu s prethodno ispunjenim anketama se vidi da svaku anketu predstavlja jedna kartica, pozadina kartice ovisi o vrsti treninga koji je napravljen te sve ostale informacije o treningu su vidljive na kartici. Također moguće je pritiskom na tekstualni gumb „*Delete*“ obrisati anketu iz baze podataka.

Kao što se vidi iz slike 5.3, zaslon s prethodno ispunjenim anketama je jedan od tri zaslona koji su dio „*bottom*“ navigacije. Ostala dva zaslona su zaslon s podacima o igračima te zaslon s timskim statističkim parametrima. Nakon što je pregledan zaslon s prethodno ispunjenim anketama, prijelazi se na zaslon s podacima o igračima. Na ovome zaslonu je moguće prijelaziti prstom lijevo ili desno kako bi se vidjela kartica s podacima o sljedećem igraču. Također, vidljivo je kazalo koje govori na kojoj stranici se trenutno nalazite. Na slici 5.4 su prikazane osnovne informacije o igraču te njegovi statistički uspjesi u prethodnoj sezoni.


**Slika 5.4.** Prikaz zaslona s karticama igrača

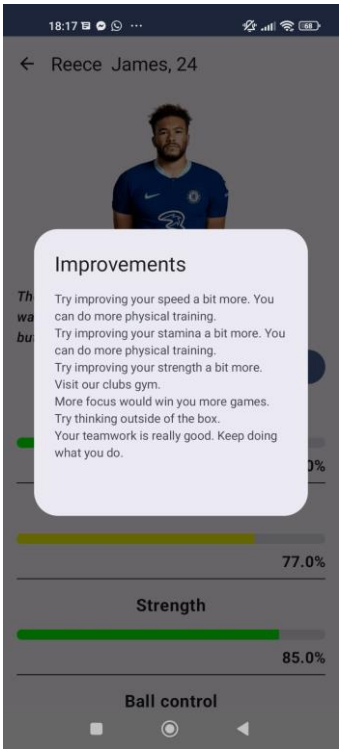
Pritiskom na karticu, prijelazi se na novi zaslon s detaljima o pojedinom igraču, koji su vidljivi na slikama 5.5 i 5.6.



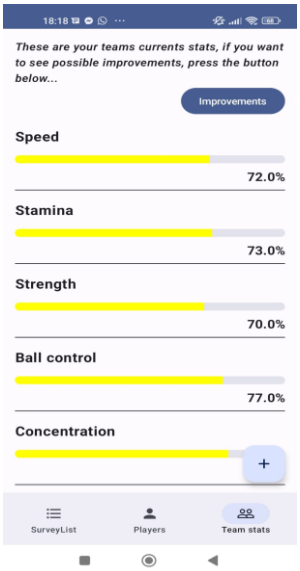

**Slika 5.5.** Prvi dio prikaza detalja o igraču **Slika 5.6.** Drugi dio prikaza detalja o igraču

Na ovome zaslonu su vidljivi statistički podaci o igraču te su prikazani različitim bojama ovisno o vrijednosti koju postižu. Uz to prikazana je i fotografija igrača, te njegovi osnovni podaci na gornjem dijelu zaslona. Pritiskom na gumb „*Improvements*“ prikazat će se prozor s mogućim poboljšanjima igrača u određenim kategorijama, što se vidi iz priložene slike 5.7.



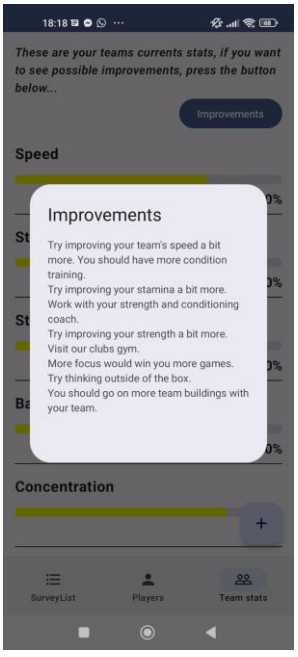

**Slika 5.7.** Prikaz savjeta poboljšanja karakteristika igrača

Nakon što je korisnik pročitao moguća poboljšanja, pritiskom na bilo koji dio zaslona će maknuti ovaj prozor s mogućim poboljšanjima, te prijelazi na zaslon s timskim statistikama. Zaslon s timskim statistikama je vidljiv na slici 5.8.




**Slika 5.8.** Prikaz timskih statistika

Ovaj zaslon je sličan zaslonu s detaljima o igraču, prikazuje timske statističke parametre s različitim bojama s obzirom na postignutu vrijednost parametra. Također, pritiskom na gumb „*Improvements*“ je moguće vidjeti moguća poboljšanja tima te savjete kako ih postići što je vidljivo iz slike 5.9.




**Slika 5.9.** Prikaz savjeta poboljšanja timskih statistika

Također, vidljivo je iz svih zaslona koji se nalaze u „*bottom*“ navigaciji kako je vidljiv i gumb sa znakom plus. Taj gumb ponovno vodi na zaslon o anketi nakon treninga. Stoga je moguće u svakom trenutku ispuniti anketu o prethodnom treningu.

## 5.2. Uvjeti i korisnički slučajevi ispitivanja rada aplikacije

U ovome dijelu rada prikazat će se rad aplikacije u posebnim uvjetima koji nisu idealni za aplikaciju, npr. nedostatak interneta ili dohvaćanje prazne liste podataka.

### 5.2.1. Definiranje uvjeta ispitivanja aplikacije

Za korištenje ove aplikacije ne postoji veliki niz uvjeta pod kojim se ona mora koristiti. Najvažniji uvjet je da korisnik ima pametni mobitel te pristup internetu. Iako, u slučaju da korisnik nema pristup internetu, još uvijek će moći koristiti ovu aplikaciju ali neće imati potpuni doživljaj pri korištenju aplikacije. Ova aplikacija je stvorena za trenere u nogometnim klubovima. Osoblje kluba umjesto trenutnih podataka o igračima može unijeti podatke za sve svoje igrače te promatrati njihov napredak, kako na terenu tako i van njega.

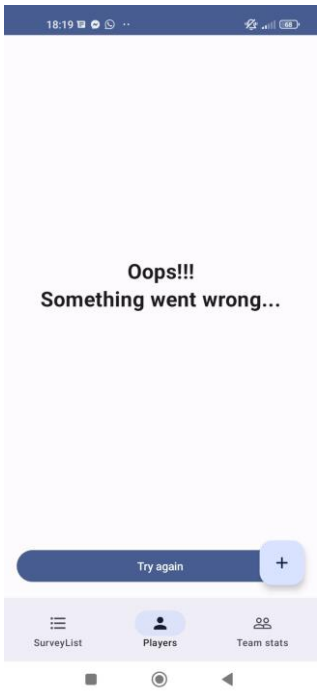
U ovom dijelu će se testirati nekoliko funkcionalnosti aplikacije. Prvi test će biti dohvaćanje podataka o igračima s vanjskog poslužitelja u različitim okruženjima. Drugi test će biti ispravan rad baze podataka gdje će se testirati spremaju li se ispravno elementi u bazu podataka, prikazuju li se ispravno te je li moguće brisati elemente baze podataka. Treći test će biti provjera prikazivanja poboljšanja za igrača ili tim kada se pritisne gumb „*Improvements*“.

### 5.2.2. Slučajevi korištenja aplikacije

Prvi slučaj koji se testira je dohvaćanje podataka s vanjskog poslužitelja u različitim okruženjima. Prvo okruženje je kada je Internet dostupan. Očekivani rezultat je prikaz kartica o igraču i njegovim statistikama. Ukoliko Internet nije dostupan, očekivani rezultat je prikaz zaslona s porukom. Na slici 5.10 prikazan je slučaj dohvaćanja podataka s vanjskog poslužitelja kada je Internet dostupan.


**Slika 5.10.** Slučaj korištenja aplikacije s dostupnim internetom

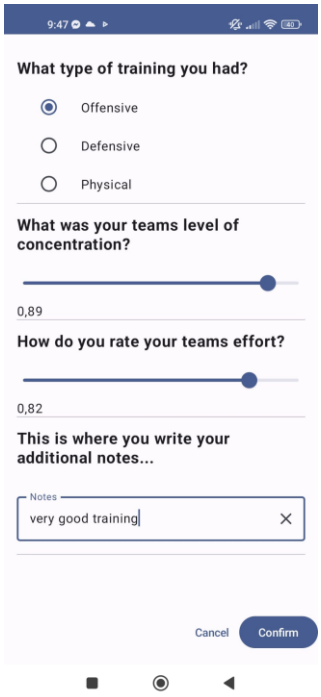
Ako se pogleda slika 5.10, vidi se da je Internet dostupan, i u tom slučaju se podaci dohvaćaju uspješno, čime je prvi dio prvog testa uspješno prošao. Na slici 5.11 je prikazan slučaj kada korisnik nema pristup internetu.




**Slika 5.11.** Slučaj korištenja aplikacije kada nema interneta

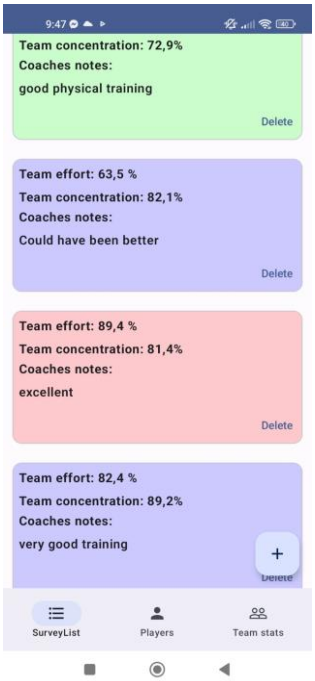
Iz slike 5.11 je vidljivo kako bez internetskog pristupa se prikazuje zaslon s prikladnom porukom. To znači da drugi dio prvog testa je uspješno prošao čime je dokazana prva funkcionalnost uspješnog i neuspješnog dohvaćanja podataka s vanjskog poslužitelja.

Drugi testni slučaj je rad s bazom podataka, testirat će se funkcije spremanja i brisanja iz baze podataka, te dohvaćanje podataka iz baze podataka. Na slici 5.12, prikazano je ispunjavanje ankete o treningu.




**Slika 5.12.** Uspješno ispunjavanje ankete

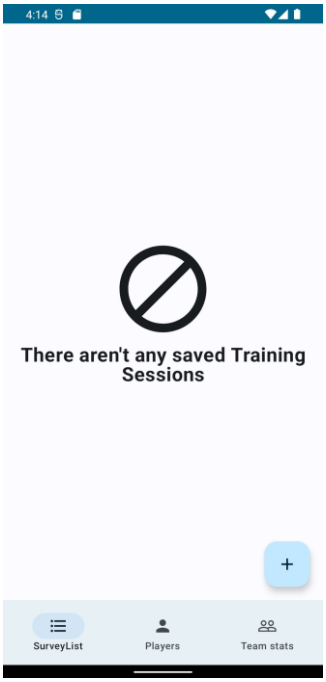
Kada korisnik ispuni anketu i pritisne na gumb „*Confirm*“, ta anketa o treningu se sprema u bazu podataka što je i vidljivo na slici 5.13.




**Slika 5.13.** Uspješno dohvaćanje prethodno spremljenih anketa

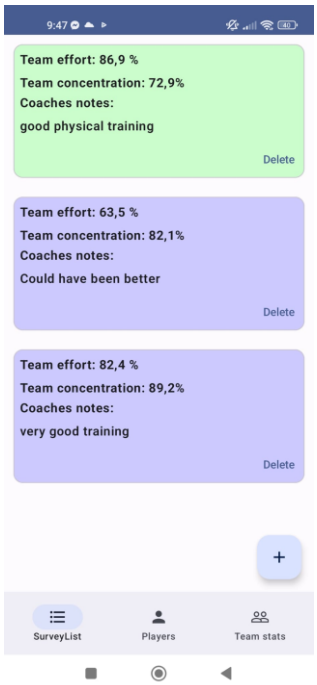
Na slici 5.13 je vidljivo kako se anketa o treningu s prethodne slike uspješno spremila u bazu podatka, i vidljivo je kako su svi elementi baze podataka uspješno prikazani.

Ukoliko je baza podataka prazna te niti jedan element ne postoji za prikaz, umjesto praznog zaslona prikazat će se zaslon s prikladnom porukom, kao što je vidljivo na slici 5.14.




**Slika 5.14.** Slučaj kada nema spremljenih anketa u bazu podataka

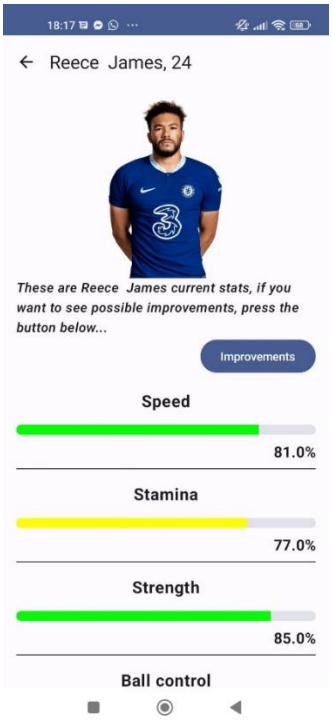
Ovime je pokriveno dvije trećine drugog testa, spremanje u bazu podataka te dohvaćanje elemenata iz baze podataka. Sada je potrebno provjeriti funkcionalnost brisanja elemenata iz baze podataka. Ako se pažljivo pogleda slika 5.14, vidljivo je da na svakoj kartici ankete o treningu postoji gumb *„Delete“*. Kada se taj gumb pritisne, element baze podataka čiji gumb je pritisnut se briše iz baze podataka te se zaslon automatski osvježava bez te ankete o treningu kao što se vidi na slici 5.15.

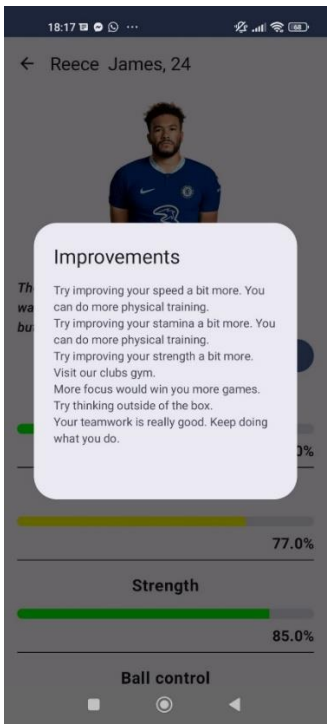



**Slika 5.15.** Prikaz liste anketa bez obrisanog elementa

Slikom 5.15 je potvrđena izjava u tekstu iznad ove slike, a time je i dokazano da brisanje elemenata iz baze podataka uspješno radi u ovoj aplikaciji. Tako je cjelokupno pokriven i drugi test funkcionalnosti ove aplikacije, a to je rad s bazom podataka.

Trećim testom se provjerava hoće li se pritiskom na gumb „*Improvements*“ prikazati moguća poboljšanja za pojedinog igrača ili cijeli tim. Taj test je prikazan na slikama 5.16 i 5.17.




**Slika 5.16.** Zaslon prije prikaza poboljšanja **Slika 5.17.** Zaslon poslije prikaza poboljšanja

Kada se otvori zaslon s timskim statistikama ili sa detaljnim statistikama pojedinog igrača, jedan od prikazanih elemenata je i gumb „*Improvements*“ pritiskom na taj gumb, kao što se vidi iz slike 5.17, prikazuju potencijalna poboljšanja za igrača te načine na koji je moguće ostvariti ta poboljšanja. Ovime je i treći test funkcionalnosti uspješno prošao čime su dokazane funkcionalnosti ove aplikacije te potencijalni slučajevi u kojima se može naći korisnik aplikacije.

## 5.3. Analiza rada aplikacije

Ova aplikacija za upravljanje performansama nogometnog kluba je pravljena s namjerom da bude jednostavna za korištenje, intuitivna, a da ispunjava svoju svrhu i funkcionalnosti. Najbitnije je da nije korisniku teška za koristiti, te da olakšava posao koji je do sada bio zamarajući. Ispunjava funkcionalnosti koje su potrebne treneru kako bi vidio sve potrebne karakteristike svog tima ili pojedinog igrača. Također može pratiti kako su igrači odradili prethodne treninge te je li i on sam zadovoljan s tim treninzima. Aplikacija je vrlo brza te dosadašnje korisničko iskustvo nije pokazalo nikakve neočekivane zastoje. Aplikacija ima vrlo jednostavno sučelje iz kojeg su jasno vidljive sve prikazane informacije te je prilagođena svim korisnicima svih dobnih skupina. Vrlo je lagano upotrebljiva i intuitivna za korištenje. Dohvaćanje podataka s vanjskog poslužitelja funkcionira relativno brzo i sa sporijim internetom. Zasloni aplikacije nisu pretrpani informacijama te su sve stavke na zaslonima pregledne. Algoritam za preporuku mogućih poboljšanja igračkih ili timskih karakteristika ispunjava svoju svrhu zato što pruža točne i potrebne informacije i postupke koje igrač ili ekipa mogu napraviti kako bi poboljšali svoje performanse i rezultate na utakmicama. U svrhu testiranja rada aplikacije, provedena su tri testa. Test dohvaćanja podataka s vanjskog poslužitelja, rad s bazom podataka te funkcionalnost prikazivanja mogućih poboljšanja za pojedine igrače ili cjelokupni tim. Rezultati testiranja svih funkcionalnosti su bili uspješni. Pozitivni rezultat u svim testovima funkcionalnosti pokazuje kvalitetu izrade ove aplikacije.

# 6. ZAKLJUČAK

U trenutku pisanja ovog rada, računalne tehnologije su već uvelike prisutne svim sportovima i razvoju sportaša. One koriste svim sportašima i ljudima koji brinu o razvoju sportaša kako bi napredovali i postizali što bolje rezultate. Aplikacija za upravljanje performansama nogometnog kluba je pokušaj doprinosa ovog rada sportu. Radi na principu postavljanja početne ankete treneru kako bi on mogao vidjeti koliko je zadovoljan treningom te u usporedbi s ostalim treninzima trend rasta ili pada kvalitete treninga te zalaganja igrača na treningu. Aplikacija nudi preporuke za poboljšanje performansi igrača ili tima temeljeno na trenutnim vrijednostima performansi. Moguće je pogledati statistike igrača u svojem timu te na temelju tih statistika pomoću algoritma za poboljšanje performansi igrača prikazati vrste treninga kojima igrač može poboljšati svoje performanse u određenim kategorijama. Kao što je moguće prikazati performanse igrača i njihova poboljšanja, tako je moguće i prikazati za cijelu ekipu i moguća poboljšanja timskih performansi te načine pomoću kojih ih mogu poboljšati. Također, kroz razvoje ove mobilne aplikacije prikazano je korištenje predloška programske arhitekture MVVM (*Model-View-ViewModel*) na kojoj je ova aplikacije zasnovana.

Nakon programskog implementiranja mobilne aplikacije, provedeno je ispitivanje njenih definiranih funkcionalnosti, a sva tri provedena testa pokazala su uspješnost u njenom ostvarenju. Ispitivanja koja su provedena su test dohvaćanja podataka s vanjskog poslužitelja, rad s bazom podataka te funkcionalnost prikazivanja mogućih poboljšanja za pojedine igrače ili za cjelokupni tim. Također, može se zaključiti da se korištenjem predloška programske arhitekture MVVM postižu pogodnosti poput slojevitosti koda, pravilna povezanost koda, načela čistog koda te odvajanje odgovornosti u kodu.

# LITERATURA


[1] R. Stanojević i L. Gyarmati, „Towards data-driven football player assessment“ u *2016 IEEE 16th International Conference on Data Mining Workshops,* 2016, pp. 167-172.

[2] D. Škarić, „90plus blog“ [Mrežno]. Available: https://90plus.blog/nogometni-trening/kako-bi-trebao-izgledati-nogometni-trening. [Pokušaj pristupa lipanj 2023].

[3] V. B. Jishnu, H. Narayanan, S. Aanand i P. T. Joy, „Football Player Transfer Value Prediction Using Advanced Statistics and FIfa 22 Data“ u *2022 IEEE 19th India Council International Conference (INDICON),* 2022, pp. 1-6.

[4] S. Manish, V. Bhagat i R. Pramila, „Prediction of Football Players Performance using Machine Learning and Deep Learning Algorithms“ u *2021 2nd International Conference for Emerging Technology (INCET),* Belgaum, 2021, pp. 1-5.

[5] B. Petrovan, „BOOM! Android Market Becomes Google Play Store, along with Google Music and Google Ebookstore [Updated]“ Android Authority, [Mrežno]. Available: https://www.androidauthority.com/google-play-android-market-google-play-store-google-play-music-google-play-movies-60425. [Pokušaj pristupa lipanj 2023].

[6] J. Dalbey, „Nonfunctional Requirements“ [Mrežno]. Available: http://users.csc.calpoly.edu/~jdalbey/SWE/QA/nonfunctional.html. [Pokušaj pristupa lipanj 2023].

[7] „Android Api Levels“ [Mrežno]. Available: https://apilevels.com. [Pokušaj pristupa 15 lipanj 2023].

[8] „FIFACM“ [Mrežno]. Available: https://www.fifacm.com. [Pokušaj pristupa svibanj 2023].

[9] „WhoScored“ [Mrežno]. Available: https://www.whoscored.com/Teams/15/Show/England-Chelsea. [Pokušaj pristupa svibanj 2023].

[10] „Introducing JSON“ [Mrežno]. Available: https://www.json.org/json-en.html. [Pokušaj pristupa lipanj 2023].

[11] „Github Pages“ [Mrežno]. Available: https://pages.github.com/?(null. [Pokušaj pristupa svibanj 2023].

[12] B. Nzivu, „Section,SImple GET request using Retrofit in Android“ [Mrežno]. Available: https://www.section.io/engineering-education/making-api-requests-using-retrofit-android. [Pokušaj pristupa srpanj 2023].

[13] R. Elizarov, M. Belyaev, M. Akhin i I. Usmanov, „Kotlin Coroutines: Design and Implementation“ u *Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software,* Chicago, 2021,pp. 68-84.

[14] P. Capek, E. Kral i R. Senkerik, „ViewModel visualization tool“ u *2016 International Conference on Computational Science and Computational Intelligence,* Zlin, 2016, pp.1426-1427.

[15] J. Audu, F. Dahunsi, O. Obe i O. Sarumi, „Android Architectures for a Robust Mobile Application Development“ *Nigerian Research Journal of Engineering and Environmental Sciences,* pp. 607-617, 12 prosinac 2021.

[16] A. Tyagi, „Better Android Apps Using MVVM With Clean Architecture“ Toptal, [Mrežno]. Available: https://www.toptal.com/android/android-apps-mvvm-with-clean-architecture. [Pokušaj pristupa lipanj 2023].

[17] T. Lou, *A comparison of Android Native App Architecture, Diplomski rad* Eindhoven: Eindhoven University of Technology, 2016.

[18] „Uvod u MVVM arhitekturu“ Web Programiranje, [Mrežno]. Available: https://www.webprogramiranje.org/uvod-u-mvvm-arhitekturu. [Pokušaj pristupa lipanj 2023].

[19] C. Chekhaba, H. Rebatchi, N. Moha, G. ElBoussaidi i S. Kpodjedo, „Coach: Classification-based Architectural Patterns Detection in Android Apps“ u *The 36th ACM/SIGAPP Symposium on Applied Computing*, Sacramento, 2021, pp. 1429-1438.

[20] „ViewModel“ Android Developers, [Mrežno]. Available: https://developer.android.com/reference/androidx/lifecycle/ViewModel. [Pokušaj pristupa lipanj 2023].

[21] T. Balint, *Android-arhitektura u Kotlinu,* Varaždin: Fakultet organizacije i informatike, 2018.

[22] E. Mixon, „AndroidOS“ TechTarget Mobile Computing, [Mrežno]. Available: https://www.techtarget.com/searchmobilecomputing/definition/Android-OS. [Pokušaj pristupa srpanj 2023].

[23] „Application Development Research Based on Android Platform“ u *2014 7th International Conference on Intelligent Computation Technology and Automation*, Beijing, 2014, pp. 579-582.

[24] „Kotlin vs Java“ Mediaan, [Mrežno]. Available: https://mediaan.com/mediaan-blog/kotlin-vs-java. [Pokušaj pristupa srpanj 2023].

[25] A. Esakia, „Transitioning to Teaching Android With Kotlin and Jetpack“ u *SIGCSE '20, March 11–14, 2020, Portland, OR, USA, (SIGCSE)*, Portland, 2020, pp. 1302-1302.

[26] „Jetpack Compose Tutorial“ [Mrežno]. Available: https://www.jetpackcompose.net/jetpack-compose-introduction. [Pokušaj pristupa srpanj 2023].

[27] „Thinking in Compose“ Android Developers, [Mrežno]. Available: https://developer.android.com/jetpack/compose/mental-model. [Pokušaj pristupa srpanj 2023].

[28] „Kotlin flows on Android“ Android Developers, [Mrežno]. Available: https://developer.android.com/kotlin/flow. [Pokušaj pristupa srpanj 2023].

[29] „Save data in a local database using Room“ Android Developers, [Mrežno]. Available: https://developer.android.com/training/data-storage/room. [Pokušaj pristupa srpanj 2023].

[30] „Dependency injection in Android“ Android Developers, [Mrežno]. Available: https://developer.android.com/training/dependency-injection. [Pokušaj pristupa srpanj 2023].

[31] „Koin“ [Mrežno]. Available: https://insert-koin.io. [Pokušaj pristupa srpanj 2023].

# ŽIVOTOPIS

Ivan Lukac rođen je 4.11.1999. u Osijeku. Nakon pohađanja Osnovne škole Ivan Goran Kovačić u Đakovu. 2014. godine upisuje opću gimnaziju Antun Gustav Matoš u Đakovu te 2018. završava gimnaziju te upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija u Osijeku, smjer računarstvo. 2019. godine polaže ispit za turističkog vodiča za četiri lokalne županije. Završava Preddiplomski studij Računarstva 2019. godine te upisuje Diplomski studij Računarstva, smjer Programsko inženjerstvo. 2022. godine stječe diplomu Sofascore Android akademije te nakon toga odrađuje stručnu praksu u tvrtki DICE gdje radi na razvoju Android mobilnih aplikacija. Govori dva strana jezika, engleski i njemački, te poznaje mnoge programske jezike kao što su C, C++, Java  i C#. U slobodno vrijeme rekreativno se bavi košarkom.

_____

Potpis autora

# SAŽETAK

U ovom diplomskom radu zamišljena je i programski implementirana mobilna Android aplikacija za upravljanje performansama nogometnog kluba. Ova aplikacija je napravljena za trenere nogometnih klubova kako bi nakon treninga mogli evidentirati svoje utiske s treninga, imati uvid u napredak svojih igrača ili cjelokupnog tima i kako bi sve navedeno mogli poboljšati. Aplikacija je programski ostvarena u razvojnom okruženju Android Studio te je pisana u programskom jeziku Kotlin. Korisničko sučelje aplikacije također je ostvareno u Kotlinu pomoću biblioteke Jetpack Compose. Za lokalno spremanje podataka korištena je baza podataka Room, a za dohvaćanje podataka s vanjskog poslužitelja korištena je biblioteka Retrofit. Aplikacija je zasnovana na arhitekturi Model-View-ViewModel (MVVM) koja je trenutni standard za razvoj Android aplikacija. Svrha ove aplikacije je olakšati posao treneru i pružiti mu sustavni pregled treninga i igračkih te timskih karakteristika na temelju kojih je moguće poboljšati njihove performanse. Provedeno je ispitivanje definiranih funkcionalnosti aplikacije, a sva tri provedena testa pokazala su uspješnost u njenom ostvarenju. Provedena ispitivanja obuhvaćaju dohvaćanje podataka s vanjskog poslužitelja, rad s bazom podataka te funkcionalnost prikazivanja mogućih poboljšanja za pojedine igrače ili cjelokupni tim. Korištenjem predloška programske arhitekture MVVM postižu se pogodnosti poput slojevitosti koda, pravilna povezanost koda, načela čistog koda te odvajanje odgovornosti u kodu.

**Ključne riječi:** mobilna Android aplikacija, predložak programske arhitekture MVVM, stvaranje preporuka, upravljanje performansama nogometnog kluba.

## ABSTRACT

In this master thesis, a mobile Android application for managing the performance of a football club was conceived and implemented programmatically. This application was made for coaches of football clubs so that after training they could record their impressions from training, have an insight into the progress of their players or the entire team and so that they could improve all of the above. The application was created programmatically in the Android Studio development environment and was written in the Kotlin programming language. The user interface of the application is also realized in Kotlin using the Jetpack Compose library. The Room database was used to store data locally, and the Retrofit library was used to retrieve data from an external server. The application is based on the Model-View-ViewModel (MVVM) architecture, which is the current standard for Android application development. The purpose of this application is to make the coach's job easier and provide him with a systematic overview of training and player and team characteristics on the basis of which it is possible to improve their performance. A test of the application's defined functionalities was conducted, and all three conducted tests showed success in its implementation. The tests carried out include retrieving data from an external server, working with a database, and the functionality of displaying possible improvements for individual players or the entire team. By using the MVVM programming architecture template, benefits such as code layering, proper code connectivity, clean code principles, and separation of concerns in code are achieved.



**Keywords:** Android mobile application, MVVM architecture, performance enhancement recommendation, managing football club performance.

# PRILOZI

Prilog 1. Završni rad u formatu .docx.

Prilog 2. Završni rad u formatu .pdf.

Prilog 3. Programsko rješenje izrađene mobilne aplikacije.