

**SVEUČILIŠTE U SPLITU
FAKULTET ELEKTROTEHNIKE, STROJARSTVA I
BRODOGRADNJE**

DIPLOMSKI RAD

**IMPLEMENTACIJA MOBILNOG SUSTAVA
ZA PRAĆENJE I ANALIZU OSOBNIH
FINANCIJA**

Nina Rađa

Split, rujan 2025.



Sveučilišni diplomski studij: **Računarstvo**

Smjer/Usmjerenje: /

Oznaka programa: 250

Akademска godina: 2024./2025.

Ime i prezime: **Nina Rađa**

JMBAG: 0036522969

ZADATAK DIPLOMSKOG RADA

Naslov: **IMPLEMENTACIJA MOBILNOG SUSTAVA ZA PRAĆENJE I ANALIZU OSOBNIH FINANCIJA**

Zadatak: Razviti mobilnu aplikaciju za praćenje osobnih financija koristeći React Native. Aplikacija će korisnicima omogućiti unos i kategorizaciju troškova, praćenje potrošnje te skeniranje računa radi automatskog unosa podataka. Osim toga, sustav će omogućiti analizu potrošnje i generiranje izvještaja kako bi korisnici lakše upravljali svojim financijama. Fokus rada bit će na intuitivnom korisničkom sučelju i integraciji različitih tehnologija za obradu podataka. Ovim radom student će stići iskustvo u razvoju mobilnih aplikacija i implementaciji funkcionalnosti za učinkovito praćenje financija.

Rad predan:

Predsjednik
Odbora za diplomski rad:

prof. dr. sc. Linda Vicković

Mentor:

prof. dr. sc. Mario Čagalj

IZJAVA

Ovom izjavom potvrđujem da sam diplomski rad s naslovom „Implementacija mobilnog sustava za praćenje i analizu osobnih financija“ pod mentorstvom prof. dr. sc. Maria Čaglja pisala samostalno, primjenivši znanja i vještine stečene tijekom studiranja na Fakultetu elektrotehnike, strojarstva i brodogradnje, kao i metodologiju znanstveno-istraživačkog rada, te uz korištenje literature koja je navedena u radu. Spoznaje, stavove, zaključke, teorije i zakonitosti drugih autora koje sam izravno ili parafrazirajući navela u diplomskom radu citirala sam i povezla s korištenim bibliografskim jedinicama.

Studentica

Nina Rađa

SADRŽAJ

1.	UVOD	1
2.	OPIS RADA APLIKACIJE	2
2.1.	Prijava korisnika.....	2
2.2.	Početni zaslon i navigacija	3
2.3.	Skeniranje i ručni unos računa	4
2.4.	Kategorizacija potrošnje.....	6
2.5.	Analiza potrošnje.....	8
2.6.	Pregled prethodno unesenih računa.....	9
2.7.	Korisnički profil	10
3.	KORIŠTENE TEHNOLOGIJE.....	12
3.1.	Klijentski sloj	12
3.1.1.	React Native	12
3.1.2.	Expo Framework	13
3.1.3.	Povezivanje s poslužiteljem (Axios)	14
3.1.4.	Korisničko sučelje i vizualno oblikovanje	14
3.2.	Poslužiteljski sloj.....	15
3.2.1.	Node.js i Express.js	15
3.2.2.	MongoDB.....	15
3.2.3.	Autentifikacija i autorizacija	16
3.2.4.	Google Document AI	17
3.2.5.	Ostale pomoćne biblioteke	17
4.	ZAHTJEVI SUSTAVA.....	19
4.1.	Funkcionalni zahtjevi	19
4.2.	Nefunkcionalni zahtjevi	20
4.3.	Ograničenja	22
5.	ARHITEKTURA SUSTAVA	23
5.1.	Pregled arhitekture	23
5.2.	Tokovi podataka.....	24
6.	IMPLEMENTACIJA KLIJENTSKOG SLOJA	27
6.1.	Struktura projekta	27
6.2.	Autentifikacija i sesija korisnika	29
6.2.1.	Registracija.....	29
6.2.2.	Prijava postojećeg korisnika.....	31
6.2.3.	Korisnički profil	32

6.3.	Unos i pregled računa.....	35
6.3.1.	Skeniranje računa kamerom	35
6.3.2.	Uvoz fotografije iz galerije i ručni unos.....	37
6.3.3.	Uređivanje i pretpregled.....	37
6.3.4.	Pregled računa	40
6.4.	Kategorije	43
6.4.1.	Pregled svih kategorija	43
6.4.2.	Pregled pojedine kategorije	45
6.4.3.	Omiljene kategorije	47
6.5.	Analitika	48
6.5.1.	Mjesečni sažetak potrošnje.....	48
6.5.2.	Trend potrošnje u posljednja četiri mjeseca	49
7.	IMPLEMENTACIJA POSLUŽITELJSKOG SLOJA	54
7.1.	Struktura projekta	54
7.2.	Modeli podataka	55
7.3.	Google Cloud Document AI	57
7.4.	Posrednički sloj	59
7.5.	Organizacija API ruta	61
7.5.1.	Rute korisničkih funkcionalnosti	61
7.5.2.	Rute za upravljanje računima	62
7.6.	Poslužiteljska poslovna logika API-ja.....	62
7.6.1.	Modul za upravljanje korisnicima.....	62
7.6.2.	Modul za upravljanje računima	68
8.	ZAKLJUČAK	72
	LITERATURA.....	73
	PRILOZI.....	75
	Kazalo slika i tablica	75
	Kazalo slika	75
	Kazalo tablica	78
	Popis oznaka i kratica.....	78
	SAŽETAK.....	80
	KLJUČNE RIJEČI	80
	SUMMARY	81
	KEYWORDS	81

1. UVOD

Upravljanje osobnim financijama predstavlja važan aspekt svakodnevnog života, budući da finansijske odluke izravno utječu na kvalitetu života, sigurnost i dugoročnu stabilnost pojedinca. Precizno praćenje prihoda i rashoda temelj je finansijske pismenosti, a ujedno i preduvjet za donošenje odgovornih odluka u potrošnji i planiranju budućih ulaganja. Međutim, iskustvo pokazuje da većina ljudi nema razvijene navike sustavnog praćenja svojih troškova, a tradicionalne metode, poput vođenja bilježnica ili tablica, često se pokazuju nepraktičnima, vremenski zahtjevnima i podložnima pogreškama.

Razvojem mobilnih tehnologija otvorene su nove mogućnosti za automatizaciju i pojednostavljenje procesa praćenja osobnih financija. Mobilne aplikacije postale su pogodan alat jer korisnicima nude stalnu dostupnost, intuitivno sučelje i integraciju s naprednim tehnologijama za obradu i analizu podataka. Na taj način omogućuju brži unos podataka, bolji uvid u obrasce potrošnje i donošenje kvalitetnijih odluka o upravljanju osobnim budžetom.

Cilj ovog diplomskog rada je razvoj mobilnog sustava za praćenje i analizu osobnih financija. Sustav omogućuje unos i kategorizaciju troškova, pregled povijesti transakcija te vizualizaciju podataka. Posebna funkcionalnost sustava je automatsko prepoznavanje i ekstrakcija podataka s računa korištenjem usluge Google Document AI, čime se smanjuje potreba za ručnim unosom i povećava točnost podataka. Klijentski dio aplikacije implementiran je u React Native i Expo okruženju, dok je poslužiteljski dio izrađen u Node.js i Express.js tehnologijama te povezan s MongoDB bazom podataka. Za autentifikaciju i autorizaciju korisnika koristi se JSON Web Token (JWT), čime se osigurava sigurna komunikacija i zaštita podataka.

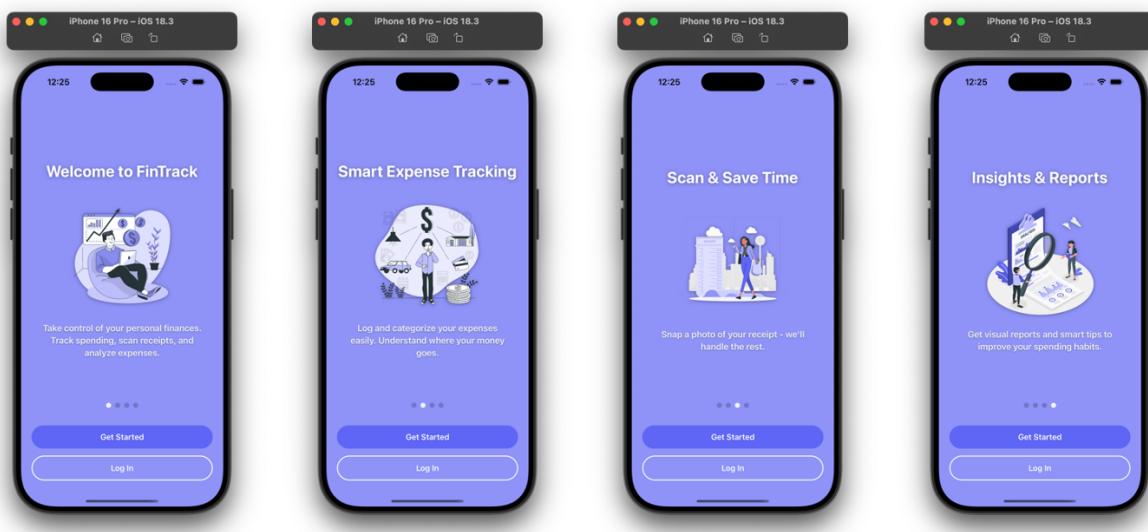
Ovaj diplomska rad podijeljen je na osam poglavlja. U drugom poglavlju prikazan je rad aplikacije s naglaskom na ključne funkcionalnosti: prijavu korisnika, upravljanje računima (unos i pregled) te alate za kategorizaciju i analizu potrošnje. Treće poglavlje donosi opis korištenih tehnologija za klijentski i poslužiteljski sloj sustava. U četvrtom poglavlju definirani su funkcionalni i nefunkcionalni zahtjevi aplikacije, dok je u petom opisana arhitektura sustava i protok podataka. Šesto poglavlje detaljno prikazuje implementaciju klijentskog dijela aplikacije, a sedmo poglavlje implementaciju poslužiteljskog dijela s naglaskom na modele podataka, API-je i poslovnu logiku. Posljednje poglavlje sadrži zaključak čitavog rada.

2. OPIS RADA APLIKACIJE

U ovom poglavlju aplikacija se prikazuje kroz niz zaslona i tipičnih koraka uporabe iz perspektive korisnika. Naglasak je na doživljaju sučelja, tj. što korisnik vidi, koje su mu mogućnosti dostupne na pojedinom ekranu i kojim se redoslijedom kreće od prijave u sustav do pregleda i analize potrošnje. Svaki odjeljak popraćen je relevantnim snimkama zaslona koje ilustriraju ključne interakcije (kao što su unos računa, kategorizacija, povijest, analitika, profil).

2.1. Prijava korisnika

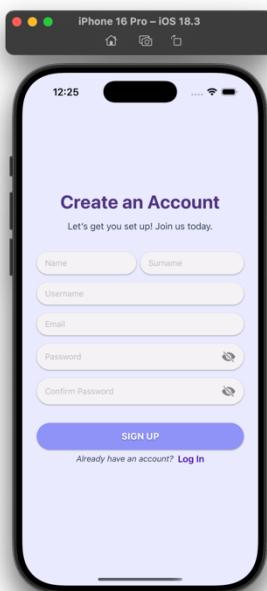
Na samom pokretanju aplikacije prikazuje se sekvenca od četiri *onboarding* zaslona: *Welcome to FinTrack*, *Smart Expense Tracking*, *Scan & Save Time* i *Insights & Reports* (slika 2.1). Njihova svrha je upoznati korisnika s ključnim značajkama aplikacije prije prve prijave. Svi zasloni dijele konzistentnu vizualnu temu: pozadinu u nijansi svjetlo ljubičaste boje, središnje postavljeni ilustrirani motiv te tipografiju u bijeloj i tamnoljubičastoj boji koja osigurava dobar kontrast i čitljivost.



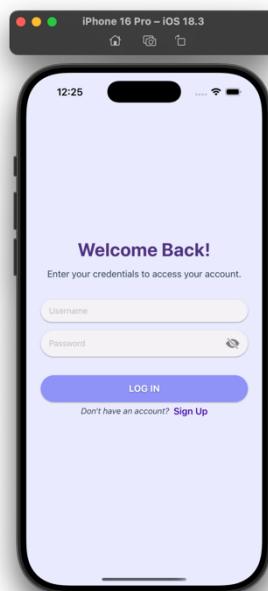
Slika 2.1 Prikaz onboarding zaslona

Ispod ilustracije nalaze se indikatori stranica (paginacijske točke) koji korisniku pokazuju trenutačnu poziciju u sekvenci. Na dnu svakoga zaslona postavljene su dvije akcijske tipke: *Get Started* (primarna, ispunjena) koja vodi izravno na registraciju te *Log In* (sekundarna, obrubljena) za postojeće korisnike. Ovakva vizualna hijerarhija jasno usmjerava novog korisnika da najprije prođe proces registracije, a istovremeno omogućuje brzi pristup prijavi za postojeće korisnike.

Zaslon *Create an Account* (slika 2.2) pruža formu za unos osnovnih korisničkih podataka: ime (eng. *name*), prezime (eng. *surname*), korisničko ime (eng. *username*), email, lozinka (eng. *password*) i potvrda lozinke (eng. *confirm password*). Sva tekstualna polja smještena su unutar zaobljenih okvira u svijetloj nijansi, čime se postiže vizualna konzistentnost s ostatkom sučelja. Za polja lozinke predviđena je ikona prekriženog oka (eng. *eye-slash*) kojom se omogućuje prikaz ili skrivanje unesene lozinke radi lakše provjere. Primarna tipka *Sign Up* postavljena je neposredno ispod forme, dok je poveznica *Log In* smještena u podnožje, čime se korisniku nudi alternativna radnja bez narušavanja toka registracije.



Slika 2.2 Zaslon za kreiranje novog korisničkog profila



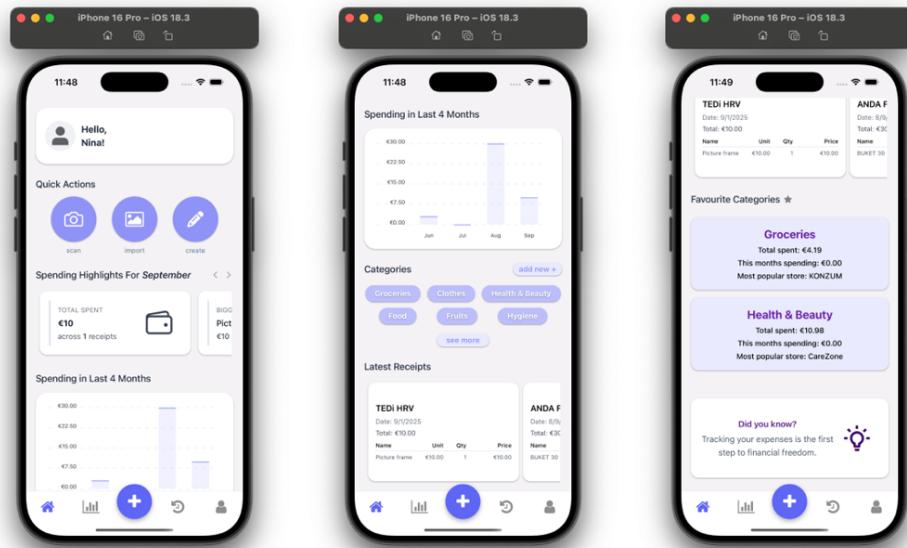
Slika 2.3 Zaslon za prijavu postojećeg korisnika

Zaslon *Welcome Back!* (slika 2.3) namijenjen je autentifikaciji postojećih korisnika. Forma sadrži dva polja za korisničko ime i lozinku, pri čemu polje lozinke također sadrži ikonu za prikaz/sakrivanje unosa. Primarna tipka *Log In* jasno je istaknuta, a ispod nje se nalazi kontekstualna poveznica *Sign Up* za korisnike koji još nemaju račun. Smještaj i stil elemenata prati istu vizualnu logiku kao i registracijski zaslon, što pridonosi konzistentnosti korisničkog iskustva.

2.2. Početni zaslon i navigacija

Slika 2.4 prikazuje početni zaslon koji se otvara po prijavi. Ispod personaliziranog pozdrava nalaze se *Quick Actions* opcije tj. tri kružne tipke (*scan*, *import*, *create*) koje omogućuju brzi pristup skeniranju računa, učitavanju slike iz galerije ili ručnom unosu troška. Slijedi odjeljak

Spending Highlights s kliznim karticama koje prikazuju ključne metrike tekućega mjeseca (ukupna potrošnja, najveća kupnja, najskuplja kategorija).



Slika 2.4 Prikaz početnog zaslona

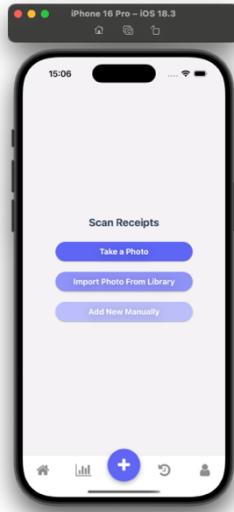
Grafikon *Spending in Last 4 Months* daje vizualni pregled troškova zadnja četiri mjeseca, dok sekcija *Categories* prikazuje najčešće kategorije s mogućnošću dodavanja nove (eng. *add new +*) ili pregleda proširenog popisa (eng. *see more*). Pri dnu se nalazi vodoravni prikaz posljednjih skeniranih računa pod nazivom *Latest Receipts* te dio s omiljenim kategorijama *Favourite Categories* i motivacijski savjet *Did you know?*.

Navigacija je organizirana kroz donju traku s pet stavki: *Home* kao početni zaslon, *Analytics* za grafičke prikaze potrošnje, središnju plutajuću akcijsku tipku (eng. *floating action*) „+“ za brzi unos novog računa, *History* za pregled svih ranijih računa te *Profile* za korisničke postavke. Aktivna ikona označena je ljubičastom bojom, dok ostale ostaju sive, što korisniku jasno naznačuje trenutačni kontekst. Ovakva kombinacija *tab* navigacije i plutajuće akcijske tipke omogućuje intuitivan pristup svim glavnim funkcijama u samo jednom ili dva dodira.

2.3. Skeniranje i ručni unos računa

Pritiskom na središnju „+“ tipku u donjoj navigaciji otvara se zaslon za unos računa (slika 2.5). Korisniku su ponuđene tri jasne akcije: snimanje fotografije računa, uvoz fotografije iz galerije ili ručni unos. Time se jednim zaslonom obuhvaća cijeli tok unosa, bez obzira na izvor podataka, uz minimalističko sučelje i velike, pristupačne gume. Nakon odabira akcije sa zaslona unosa, *Take a Photo* otvara kameru uređaja, dok *Import Photo From Library* pokreće sustavni odabir slike iz galerije. Nakon potvrde fotografije prikazuje se međuekran s ikonom

učitavanja *Extracting receipt data...* (slika 2.6), tijekom kojeg se iz slike parsiraju ključni podaci.

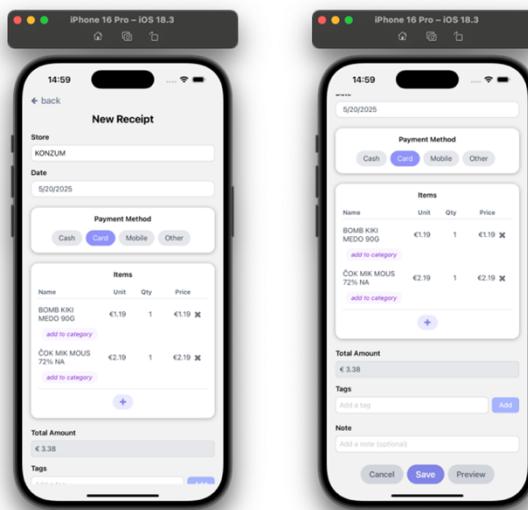


Slika 2.5 Zaslon za unos računa



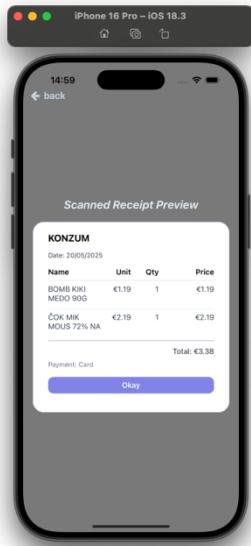
Slika 2.6 Zaslon čekanja podataka

Po završetku ekstrakcije automatski se otvara forma *New Receipt* (slika 2.7) sa unaprijed popunjena dobivenim vrijednostima radi brže provjere i eventualnih dorada. U gornjem dijelu prikupljaju se osnovni metapodaci: trgovina, datum i način plaćanja. Središnji dio čini tablica *Items* s pregledom stavki (naziv, jedinična cijena, količina, iznos) te brzim akcijama za dodavanje nove stavke i dodjelu kategorije pojedinoj stavci. Svaka promjena odmah se odražava u izračunu iznosa. Donji dio forme sadrži *Total Amount* za ukupan iznos, opcionalna polja *Tags* za tagove i *Note* za bilješke te primarne akcije *Save* i *Preview*.

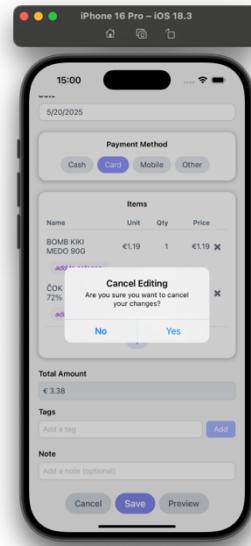


Slika 2.7 Forma s podacima novog računa

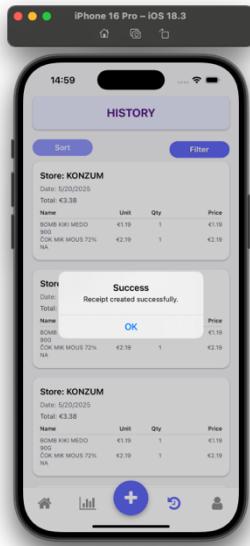
Opcija *Preview* otvara pregled skeniranog/unesenog računa u čitkoj, kompaktnoj prezentaciji (slika 2.8). Pregled služi kao završna provjera točnosti prije potvrde, nakon čega se korisnik vraća u tijek i može spremiti račun. Ako korisnik pokuša napustiti formu bez spremanja, prikazuje se potvrdni dijalog (slika 2.9) s naslovom *Cancel Editing* i izborom *Yes/No*, čime se sprječava nenamjerni gubitak unesenih izmjena. Nakon odabira *Save* zapis se trajno spremi, korisnik se vraća na zaslon *History*, a sustav prikazuje potvrdu uspjeha (slika 2.10) što pruža nedvosmislen završetak toka.



Slika 2.8 Pregled unesenog računa



Slika 2.9 Odustanak od unosa

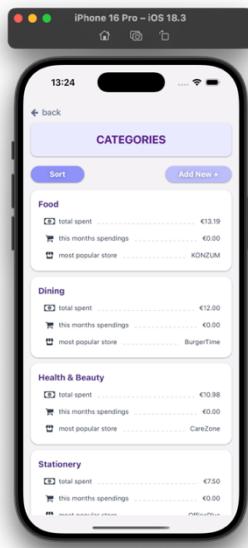


Slika 2.10 Potvrda uspjeha

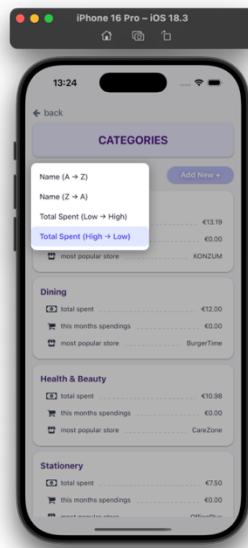
Odabirom opcije *Add New Manually*“ otvara se prazna forma „*New Receipt*“ namijenjena potpunom ručnom unosu. Korisnik unosi osnovne metapodatke (trgovina, datum i način plaćanja), dodaje stavke te po potrebi pridružuje oznake (eng. *tags*) i bilješku (eng. *note*). Nakon popunjavanja, račun je moguće pregledati (eng. *preview*) ili spremiti (eng. *save*) kao što je prethodno opisano.

2.4. Kategorizacija potrošnje

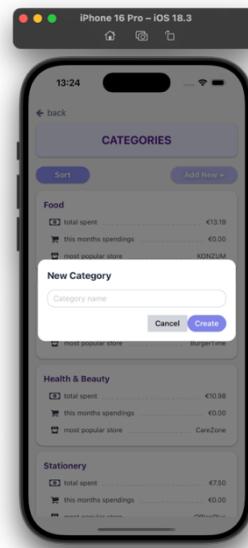
Klikom na poveznicu *see more* na početnom zaslonu otvara se pregled kategorija *Categories* (slika 2.11) koji korisniku omogućuje detaljno upravljanje kategorijama troškova. Gornja traka nudi tipku *Sort* kojom se popis može rasporediti po nazivu ili ukupno potrošenom iznosu (slika 2.12) te tipku *Add New +* za dodavanje nove kategorije putem modalnog obrasca (slika 2.13). Svaka kartica kategorije prikazuje tri ključne metrike: ukupan iznos potrošnje (eng. *total spent*), potrošnju u tekućem mjesecu (eng. *this month's spending*) i najčešće korištenu trgovinu (eng. *most popular store*).



Slika 2.11 Prikaz svih kategorija

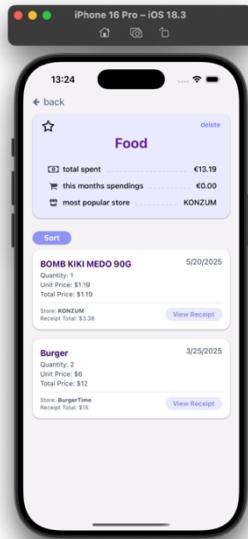


Slika 2.12 Prikaz opcije sortiranja

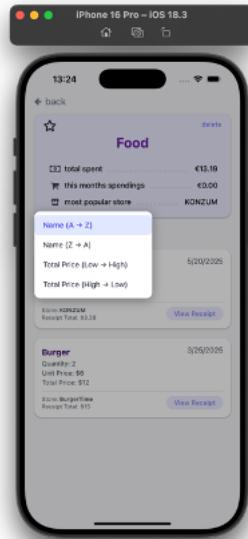


Slika 2.13 Kreiranje nove kategorije

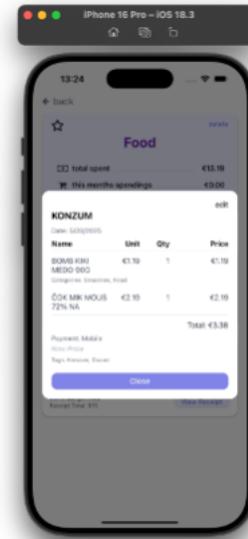
Dodirom na karticu otvara se zaslon s detaljima (slika 2.14) na kojem su prikazane sve stavke unutar odabrane kategorije, uz mogućnost sortiranja (slika 2.15) te pregled cijelog računa na kojem se stavka nalazi (slika 2.16).



Slika 2.14 Prikaz određene kategorije



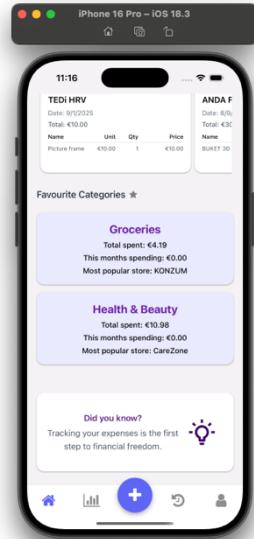
Slika 2.15 Prikaz sortiranja artikala u kategoriji



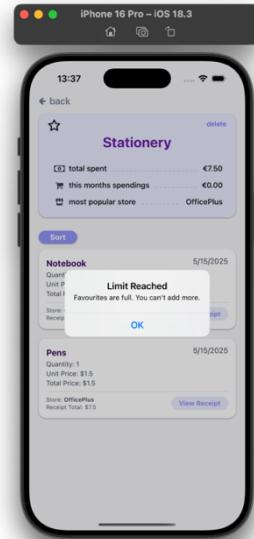
Slika 2.16 Prikaz računa odabranog artikla

Klikom na ikonu zvjezdice u gornjem lijevom kutu korisnik može dodati kategoriju na popis omiljenih *Favourite Categories* (slika 2.17) koji se nalazi na početnom zaslonu te je moguće dodati najviše četiri takve kategorije. Nakon što korisnik pokuša dodati petu omiljenu kategoriju, aplikacija prikazuje informativni modalni prozor (slika 2.18) koji ga obavještava da

je dosegnuto ograničenje pomoću modala s naslovom *Limit Reached* te je tako spriječeno daljnje dodavanje.



Slika 2.17 Prikaz omiljenih kategorija na početnom zaslonu

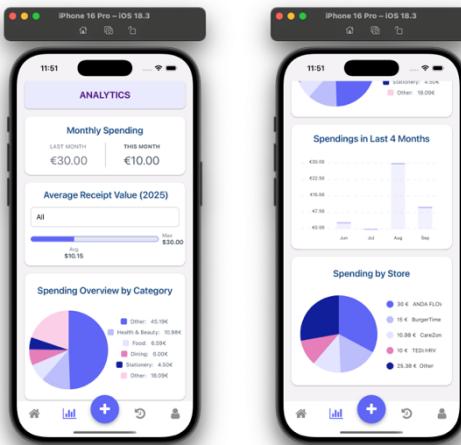


Slika 2.18 Modalni prozor za obaveštanje o dosegnutom ograničenju

Ovakva struktura omogućuje korisniku da, uz samo nekoliko dodira, doda novu kategoriju, filtrira postojeće prema željenom kriteriju ili analizira konkretne troškove unutar odabrane kategorije čime se postiže brza i intuitivna kontrola potrošnje.

2.5. Analiza potrošnje

Na zaslonu *Analytics* (slika 2.19) korisniku se na jednome mjestu prikazuju sažeti uvidi o potrošnji koji su organizirani u kartice i grafikone čime se omogućuje da korisnik u nekoliko pomaka prsta dobije odgovor na tri tipična pitanja: koliko trošim, na što trošim i gdje trošim.



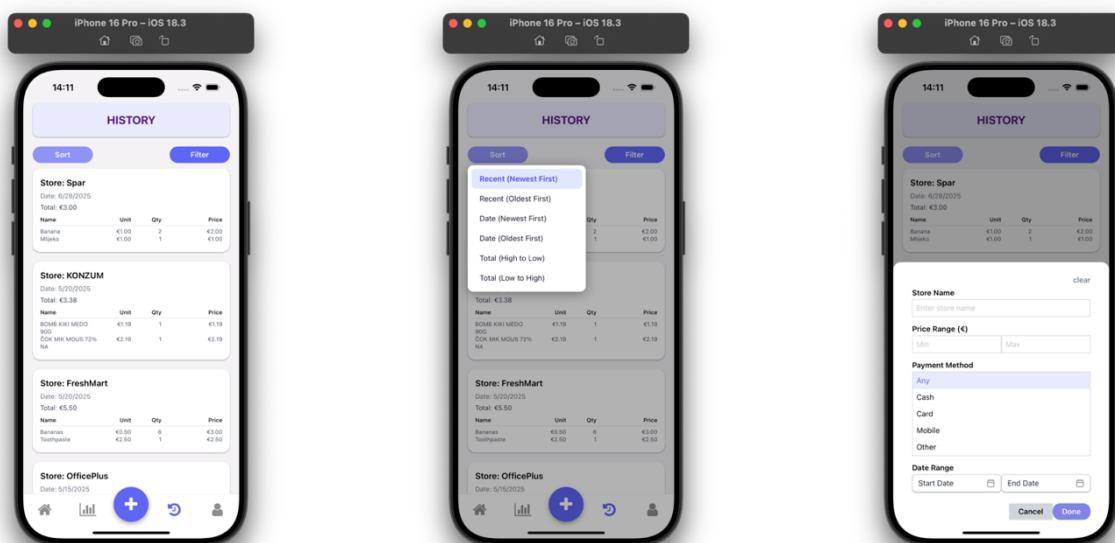
Slika 2.19 Prikaz zaslona za analizu potrošnje

Kartica *Monthly Spending* uspoređuje ukupnu potrošnju u prethodnom i tekućem mjesecu i daje korisniku brzu procjenu trenutačnog trenda. Slijedi kartica *Average Receipt Value* koja prikazuje prosječan iznos računa unutar odabranog razdoblja, a željeno vremensko razdoblje korisnik može odabrati u modalnom prozoru koji se otvara dodirom na padajući izbornik.

Komponenta *Spending Overview by Category* koristi kružni grafikon za distribuiranje troškova po kategorijama, a legenda uz grafikon prikazuje iznose u eurima. Stupčasti graf *Spendings in Last 4 Months* zbraja potrošnju po mjesecima prethodna četiri mjeseca, dok zadnja kartica *Spending by Store* prikazuje udjele potrošnje po trgovinama u obliku kružnog grafikona, čime korisniku olakšava prepoznavanje najčešćih mjesta kupnje.

2.6. Pregled prethodno unesenih računa

Zaslon *History* (slika 2.20) prikazuje kronološki popis svih ranije unesenih računa. Svaki zapis sadrži osnovne metapodatke: naziv trgovine, datum kupnje, ukupan iznos te tablični prikaz artikala s količinama i jediničnim cijenama, što korisniku omogućuje brzu procjenu troška. Na vrhu se nalaze dvije funkcijeske tipke *Sort* i *Filter*. *Sort* otvara padajući izbornik (slika 2.21) s više kriterija razvrstavanja, kao što su *Recent (Newest First)*, *Date (Oldest First)* ili *Total (High to Low)*. Filter poziva modalni prozor (slika 2.22) u kojem korisnik može suziti prikaz prema nazivu trgovine, rasponu iznosa, načinu plaćanja ili vremenskom intervalu uz kalendarsko odabiranje startnog i završnog datuma.

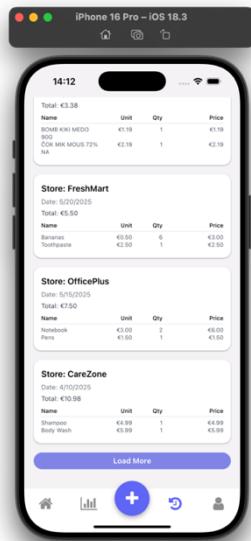


Slika 2.20 Prikaz „History“ zaslona

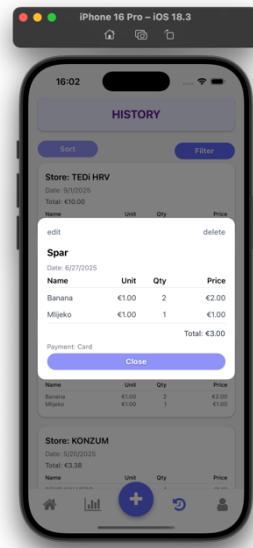
Slika 2.21 Prikaz izbornika za sortiranje

Slika 2.22 Prikaz izbornika za filtriranje

Radi bolje preglednosti i izvedbe, početno se učitava prvih pet računa. Tipka *Load More* (slika 2.23) koja se nalazi ispod popisa, dohvaća sljedećih pet računa što omogućuje beskonačno listanje bez preopterećenja zaslona. Dodirom na stavku iz popisa otvara se modalni prikaz pojedinačnog računa (slika 2.24) sa svim artiklima, iznosima i dodatnim informacijama. Ovaj modal podržava i opciju za uređivanje *edit*, čime korisnik može naknadno ispraviti ili dopuniti podatke o računu, te opciju za brisanje računa *delete*.



Slika 2.23 Prikaz tipke „Load More“

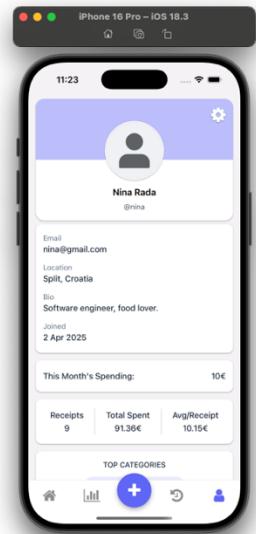


Slika 2.24 Modalni prikaz pojedinačnog računa

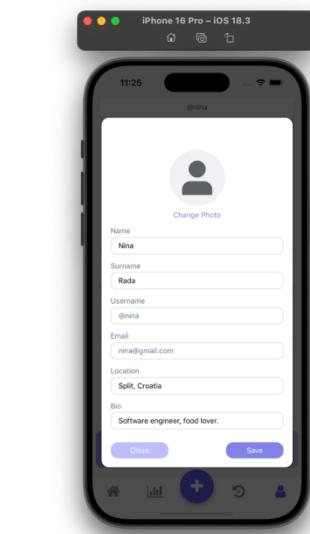
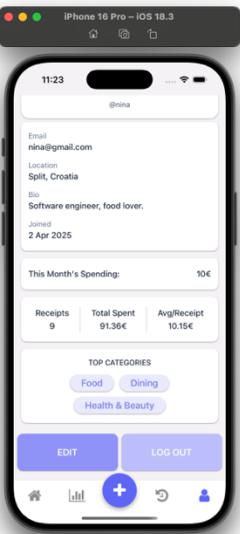
Opisana kombinacija sortiranja, filtriranja i postupnog učitavanja osigurava korisniku jednostavno pretraživanje i učinkovito upravljanje povijesti transakcija.

2.7. Korisnički profil

Zaslon *Profile* (slika 2.25) donosi pregled osobnih podataka i osnovnih statistika korisnika. Gornji segment prikazuje korisničku sliku, puno ime i korisničko ime. Ispod su prikazani statični podaci e-pošta, lokacija, opis i datum pridruživanja te su organizirani u karticu sa zaobljenim rubovima za bolju čitljivost. Slijedi kartica *This Month's Spending* koja prikazuje kumulativni iznos potrošnje ostvaren u tekućem mjesecu. Odmah nakon slijedi trodijelna statistika koja prikazuje ukupan broj unesenih računa (eng. *number of receipts*), kumulativni iznos potrošnje (eng. *total spent*) te prosječnu vrijednost pojedinog računa (eng. *average value per receipt*). Odjeljak *Top Categories* na dnu zaslona grafički ističe tri najčešće kategorije troškova, prikazane u obliku ljubičastih čipova.



Slika 2.25 Prikaz zaslona korisničkog profila



Slika 2.26 Modal za uređivanje profila

Pri dnu zaslona nalaze se dvije akcijske tipke *Edit* koja otvara modalni prozor (slika 2.26) u kojem korisnik može promijeniti profilnu sliku i osobne podatke te *Log Out* koji odjavljuje korisnika iz aplikacije i briše JSON Web Token iz lokalne pohrane. Ovakav raspored omogućuje brz pristup ključnim informacijama i jednostavno održavanje osobnih postavki, a pritom zadržava vizualnu konzistentnost s ostalim zaslonima aplikacije.

3. KORIŠTENE TEHNOLOGIJE

Ovo poglavlje prikazuje cjeloviti tehnološki sklop na kojem je izrađena aplikacija, a koji obuhvaća mobilnog klijenta, poslužiteljsku stranu i usluge u oblaku. Poseban naglasak stavljen je na razloge odabira svake tehnologije, način integracije i doprinos funkcionalnostima poput autentifikacije, obrade računa, vizualizacije i trajne pohrane.

3.1. Klijentski sloj

3.1.1. React Native

React Native je razvojni okvir otvorenog koda (eng. *open-source framework*) koji omogućuje razvoj mobilnih aplikacija za platforme iOS i Android koristeći JavaScript i React biblioteke. Razvio ga je Meta Platforms (prijašnji Facebook Inc.) 2015. godine, a od tada je postala jedna od najpopularnijih tehnologija za razvoj višestruko-platformskih (eng. *cross-platform*) aplikacija. [1]

Umjesto pisanja zasebnog koda za svaku platformu, React Native *framework* omogućava korištenje zajedničkog JavaScript koda koji se izvršava na obje platforme, a pritom koristi izvorne (eng. *native*) komponente operativnog sustava za prikaz sučelja. To znači da aplikacije izradene u React Native-u imaju performanse i izgled usporedive s izvornim aplikacijama, a istovremeno zadržavaju prednosti bržeg razvoja i lakšeg održavanja koje pruža JavaScript. Razvojni tok oslanja se na *Metro bundler* i *hot reload*, pa su iteracije nad korisničkim sučeljem brze i bez potpunog ponovnog pokretanja aplikacije. [1]

React Native posjeduje bogat i aktivan ekosustav dodatnih biblioteka koje proširuju njegovu osnovnu funkcionalnost. Njegova modularna struktura omogućuje jednostavnu integraciju alata za navigaciju, mrežnu komunikaciju, pristup *native* značajkama uređaja te je moguće integrirati vlastiti izvorni kod. [1]

U sklopu ovog rada React Native služi kao temelj klijentske aplikacije. Njegove prednosti u brzini razvoja, lakoći održavanja i pristupu *native* funkcijama omogućile su izradu responzivne i funkcionalne aplikacije dostupne na različitim mobilnim platformama. Za vizualizaciju podataka o potrošnji korištene su *react-native-chart-kit* i *react-native-svg*. Biblioteka *react-native-chart-kit* nudi gotove, prilagodljive komponente grafikona (npr. stupčaste i kružne), dok *react-native-svg* osigurava SVG podršku i kontrolu nad iscrtavanjem te animacijama. [1]

Za lokalno upravljanje stanjem koristi se AsyncStorage koji predstavlja jednostavno, asinkrono trajno spremište u obliku ključ-vrijednost. Tokom autentifikacije JWT token se po uspješnoj prijavi pohranjuje lokalno kako bi se omogućila trajna sesija i pristup zaštićenim API rutama bez ponovnog unosa vjerodajnika.

3.1.2. Expo Framework

Expo je besplatni razvojni okvir otvorenog koda koji pojednostavljuje razvoj React Native aplikacija koje se mogu pokretati na Androidu, iOS-u i web preglednicima. Omogućuje brži i lakši razvoj mobilnih aplikacija pružajući unaprijed konfiguirirane komponente i servise, čime smanjuje potrebu za složenim podešavanjima i integracijama. Expo i React Native često se koriste zajedno kako bi se maksimalno iskoristile njihove prednosti u bržem i jednostavnijem razvoju *cross-platform* mobilnih aplikacija. [2]

Glavna prednost Expo-a jest mogućnost razvoja, testiranja i distribucije aplikacija na jednostavan i efikasan način, bez potrebe za izravnim upravljanjem izvornim kodom. Platforma uključuje razvojno okruženje koje obuhvaća aplikaciju Expo Go, putem koje se aplikacije mogu odmah testirati na stvarnim uređajima ili emulatorima bez potrebe za dugotrajnim procesom instalacije. Također, Expo pruža pristup brojnim *native* API-jima, poput kamere, senzora, lokacije, obavijesti i drugih, što programerima omogućuje jednostavan pristup ključnim funkcionalnostima uređaja. [2]

U okviru ovog diplomskog rada korištene su dvije važne biblioteke iz Expo ekosustava: *expo-camera* i *expo-image-picker*. Biblioteka *expo-camera* omogućuje pristup kamери uređaja za snimanje fotografija i skeniranje računa direktno unutar aplikacije, dok *expo-image-picker* omogućuje korisnicima da odaberu slike iz galerije. Obje biblioteke su jednostavne za integraciju i potpuno kompatibilne s Expo okvirom, čime se dodatno pojednostavljuje rad s medijskim sadržajem na mobilnim uređajima.

Navigacijska arhitektura zasnovana je na Expo Router-u, koji uvodi usmjeravanje temeljeno na datotekama (eng. *file-based routing*) u direktoriju *app/* čija struktura izravno definira rute i tokove. Navigacija i prosljeđivanje parametara provode se programskim pozivima kroz *useRouter* i čitanjem parametara putem *useLocalSearchParams*. Takav pristup donosi dosljedno ponašanje na iOS-u i Androidu, pojednostavljuje dubinsko povezivanje (eng. *deep linking*) i smanjuje količinu posredničkog koda što olakšava održavanje i razvoj.

3.1.3. Povezivanje s poslužiteljem (Axios)

Za komunikaciju s poslužiteljem i mrežne operacije, aplikacija koristi Axios tj. HTTP klijent baziran na obećanjima (eng. *promises*) koji je pogodan za preglednike i Node.js okruženja. [3] Axios pojednostavljuje rad s REST API-jima u React Native aplikacijama putem metoda poput *axios.get(url, config)* i *axios.post(url, data, config)*, a podržava sve osnovne CRUD operacije (GET, POST, PUT, DELETE i PATCH). [4] Instalacija Axiosa je jednostavna pomoću npm ili yarn naredbi, a uključuje i ugrađene TypeScript tipove za bolju razvojnu podršku. [4] Axios podržava automatsku transformaciju JSON podataka, korištenje *interceptora* za obradu zahtjeva i odgovora, te jednostavno rukovanje greškama putem *promises*, čime se osigurava pouzdanost i čitljivost koda. [3]

U sklopu aplikacije, Axios se koristi za slanje unesenih troškova ili podataka skeniranih računa s klijenta na poslužitelj, kao i za dohvat rezultata analize i izvještaja. Osim toga, Axios se koristi i za prijavu korisnika, upravljanje korisničkim sesijama, dohvat korisničkih profila te brojne druge funkcionalnosti.

3.1.4. Korisničko sučelje i vizualno oblikovanje

Za stilizaciju korisničkog sučelja u projektu se koristi NativeWind biblioteka koja omogućuje primjenu pristupa temeljenog na pomoćnim klasama (eng. *utility-first*) poznatog iz Tailwind CSS-a unutar React Native aplikacija. NativeWind koristi Tailwind klasu kao visoko-razinski jezik za definiranje dizajna omogućujući dijeljenje stilskih komponenti između različitih platformi pri čemu koristi optimalan stilizacijski mehanizam za svaku od njih. [5]

NativeWind omogućuje punu integraciju s datotekom *tailwind.config.js* što znači da su dostupne funkcionalnosti poput prilagođavanja tema, korištenja dodataka i stilizacije putem pseudo-klasa. Osim toga, NativeWind koristi i Babel *plugin* koji automatski pretvara *className* atribut u *native* stilove i omogućuje podršku za Tailwind IntelliSense u editorima koda. [5]

U sklopu ovog diplomskog rada, datoteka *tailwind.config.js* je prilagođena kako bi definirala vizualni identitet aplikacije. Konfiguracija sadrži proširene palete boja za različite slučajeve upotrebe kao što su *primary*, *success*, *warning* i *error*, kao i nijanse za tekst i pozadinu, a stilovi se automatski primjenjuju na sve komponente unutar direktorija *app/* što omogućuje dosljedno tematsko iskustvo kroz cijelu aplikaciju.

3.2. Poslužiteljski sloj

3.2.1. Node.js i Express.js

Node.js je *open-source*, višestruko platformsko JavaScript izvršno okruženje koje omogućuje pokretanje JavaScript koda izvan web preglednika. Izgrađen je na V8 JavaScript mehanizmu, koji prevodi JavaScript u izvorni strojni kod, omogućujući brzo i učinkovito izvršavanje. Node.js koristi neblokirajuću arhitekturu temeljenu na događajima, što ga čini osobito prikladnim za izgradnju skalabilnih i visokoučinkovitih mrežnih aplikacija. [8]

U kontekstu ovog diplomskog rada, Node.js služi kao temelj za poslužiteljsku stranu aplikacije, omogućujući korištenje JavaScripta i na klijentskom i na poslužiteljskom dijelu, čime se pojednostavljuje razvoj i smanjuje potreba za prebacivanjem između različitih tehnologija. Uz ugrađene module, Node.js pruža osnovne funkcionalnosti potrebne za obradu logike na poslužitelju, rad s datotekama te upravljanje asinkronim događajima.

Kako bi se dodatno pojednostavilo upravljanje poslužiteljem i rutama, projekt koristi Express.js, minimalistički i fleksibilan web *framework* za Node.js. Express pruža bogat skup značajki za izradu API-ja, uključujući podršku za *routing*, posredničke slojeve (eng. *middlewares*), HTTP metode i rad s predlošcima. On uklanja potrebu za pisanjem velikog dijela ponavlјajućeg koda koji bi inače bio potreban pri radu s izvornim Node.js poslužiteljem, čime omogućuje pregledniju i modularniju strukturu aplikacije. [9]

Node.js i Express.js predstavljaju temeljne tehnologije na kojima se zasniva poslužiteljski dio sustava, pružajući brzo, fleksibilno i razvojno pogodno okruženje za izgradnju REST API-ja i obradu HTTP zahtjeva na jasan i održiv način.

3.2.2. MongoDB

MongoDB široko je korištena, *open-source* NoSQL baza podataka, osmišljena za pohranu, dohvati i obradu podataka korištenjem fleksibilnog modela podataka orijentiranog prema dokumentima. Za razliku od tradicionalnih relacijskih baza podataka koje se temelje na tablicama, retcima i unaprijed definiranim shemama, MongoDB podatke pohranjuje u dokumentima nalik JSON formatu te ih organizira unutar kolekcija. Ovakav pristup omogućuje prirodnije modeliranje podataka koje je u skladu s objektima korištenima u aplikacijskom kodu. [10]

MongoDB je dizajniran za horizontalno skaliranje i visoku dostupnost. Funkcionalnosti poput *shardinga* omogućuju distribuciju podataka na više poslužitelja, dok replikacijski setovi

pružaju otpornost na greške i automatski oporavak u slučaju prekida rada, čime se povećava pouzdanost sustava. Uz to, MongoDB podržava napredno indeksiranje, fleksibilno pretraživanje te agregacijski okvir, čime se omogućuje učinkovita obrada i analiza podataka. [10]

Radi dodatnog pojednostavljenja rada s bazom podataka unutar Node.js aplikacije, u projektu je integrirana biblioteka Mongoose, odnosno popularna ODM (*Object Data Modeling*) biblioteka za MongoDB koja omogućuje definiranje shema podataka na razini aplikacije. Također olakšava izvođenje operacija poput dohvaćanja, izmjene i brisanja zapisa, te podržava dodatne funkcionalnosti poput virtualnih polja, povezivanja dokumenata i *middleware* funkcija, tj. funkcija posredničkog sloja. [10]

U sklopu ovog diplomskog rada, MongoDB se koristi za pohranu strukturiranih podataka, uključujući korisničke račune i skenirane račune. Mongoose se koristi za definiranje modela nad tim podacima, zajedno s pravilima za validaciju i zadanim vrijednostima, čime se postiže dosljednost i sigurnost podataka unutar aplikacijske logike. Kombinacija fleksibilnosti MongoDB-a i strukturne kontrole koju omogućuje Mongoose pruža uravnoteženo rješenje koje olakšava upravljanje podatkovnim slojem i održavanje aplikacije kroz dulje vremensko razdoblje.

3.2.3. Autentifikacija i autorizacija

U svrhu osiguravanja pristupa aplikaciji i zaštite korisničkih podataka, poslužitelj koristi kombinaciju tehnologija JWT (*JSON Web Token*) i *bcrypt*. Ove tehnologije omogućuju implementaciju pouzdanog sustava autentifikacije i autorizacije u skladu s aktualnim sigurnosnim praksama u razvoju web aplikacija.

JSON Web Token predstavlja standardizirani format za prijenos podataka između dviju strana u obliku digitalno potpisanih tokena. [11] U ovoj aplikaciji koristi se za upravljanje korisničkim sesijama nakon prijave. Kada korisnik uspješno unese pristupne podatke, poslužitelj generira JWT koji sadrži osnovne informacije o korisniku potpisane tajnim ključem. Taj se token zatim šalje klijentskoj aplikaciji, gdje se pohranjuje lokalno te se prilaže u zaglavlju uz svaki naredni zahtjev prema poslužitelju. Na taj način omogućena je sigurna autorizacija pristupa zaštićenim API rutama. [12]

Za potrebe sigurnog pohranjivanja lozinki koristi se biblioteka *bcrypt*. Prilikom registracije korisnika, njegova lozinka se prije spremanja u bazu podataka kriptografski *hashira* pomoću

bcrypt algoritma. *Bcrypt* primjenjuje *salt* mehanizam i višestruko kriptiranje, čime se znatno otežava pokušaj reverzibilne rekonstrukcije originalne lozinke čak i u slučaju neovlaštenog pristupa bazi. [13]

Korištenjem ovih tehnologija ostvarena je sigurna i skalabilna autentifikacija korisnika, koja omogućuje jednostavno proširivanje sustava na više korisničkih razina, vremensko ograničenje pristupa, te integraciju dodatnih mehanizama. Ovakav pristup sigurnosti usklađen je s modernim preporukama u razvoju web aplikacija te značajno pridonosi zaštiti privatnosti i povjerljivosti korisničkih podataka.

3.2.4. Google Document AI

Google Cloud Document AI predstavlja uslugu u oblaku (eng. *cloud service*) temeljenu na strojnom učenju za automatsku obradu i razumijevanje dokumenata. U sklopu ovog rada koristi se specijalizirani model *Expense Parser*, čija je uloga strukturirano izdvajanje podataka s računa kao što su datum, naziv trgovine, lokacija, način plaćanja, ukupni iznos te stavke (naziv, količina, jedinična i ukupna cijena). Servis vraća rezultat u JSON formatu uz pripadajuće pokazatelje pouzdanosti, čime se postiže standardiziran i strojno obradiv ulaz za daljnju pohranu i analitiku. Prednost primjene Document AI-a jest da uklanja potrebu za izradom i održavanjem vlastitih OCR (*Optical Character Recognition*) modela, osigurava skalabilnost i visoku dostupnost, te se jednostavno integrira putem REST API-ja i kontrolira kroz IAM (*Identity and Access Management*) putem servisnih računa, uloga i ključevima. Time se značajno smanjuje broj pogrešaka pri ručnom unosu, ubrzava tok podataka i podiže kvaliteta izveštavanja.

3.2.5. Ostale pomoćne biblioteke

Uz temeljne tehnologije poslužiteljskog sloja sustava, u projektu se koriste i brojne pomoćne biblioteke koje pojednostavljaju konfiguraciju, obradu podataka i komunikaciju između klijenta i poslužitelja. Iako ne sudjeluju izravno u poslovnoj logici aplikacije, ove biblioteke imaju ključnu ulogu u stvaranju stabilnog, sigurnog i funkcionalnog okruženja za rad API-ja.

Biblioteka *dotenv* koristi se za učitavanje varijabli okruženja iz *.env* datoteke čime se omogućuje sigurno upravljanje osjetljivim podacima kao što su API ključevi, lozinke za baze podataka i privatni tokeni. Na taj se način povećava sigurnost aplikacije i omogućuje jednostavno prilagođavanje konfiguracije u različitim okruženjima poput lokalnog razvoja, testiranja ili produkcije. [15]

Middleware biblioteka *cors* (*Cross-Origin Resource Sharing*) omogućuje poslužitelju prihvaćanje zahtjeva iz klijentskog dijela aplikacije koja se izvršava na drugoj domeni ili portu. Budući da web preglednici prema zadanim postavkama blokiraju zahtjeve između različitih izvora iz sigurnosnih razloga, *cors* omogućuje definiranje pravila pristupa čime se osigurava ispravna komunikacija između klijenta i poslužitelja u razvojnim i produkcijskim okruženjima. [16]

Multer je *middleware* za obradu prijenosa datoteka, osobito onih koje se šalju putem *multipart/form-data* zahtjeva. Multer podržava različite načine pohrane, filtriranje tipova datoteka te postavljanje ograničenja veličine. [17]

Korištenjem navedenih biblioteka značajno se pojednostavljuje razvoj i održavanje sustava, uz povećanu sigurnost i bolju strukturu aplikacijskog koda.

4. ZAHTJEVI SUSTAVA

Ovo poglavlje definira zahtjeve sustava na kojima se temelji razvoj mobilne aplikacije za praćenje i analizu osobnih financija. Funkcionalni zahtjevi opisuju što sustav mora omogućiti da bi ispunio svoje ciljeve, dok nefunkcionalni zahtjevi propisuju kako te funkcionalnosti trebaju biti ostvarene u pogledu sigurnosti, performansi, upotrebljivosti i drugih kvalitativnih aspekata. Jasna i nedvosmislena specifikacija zahtjeva pruža jasan plan za dizajn i implementaciju te služi kao objektivno mjerilo pri verifikaciji i validaciji završnog proizvoda.

4.1. Funkcionalni zahtjevi

Funkcionalni zahtjevi određuju konkretnе radnje koje sustav mora omogućiti kako bi ispunio svoju svrhu. Ovi su zahtjevi temelj za projektiranje arhitekture sustava i implementaciju. U nastavku su navedeni funkcionalni zahtjevi označeni šiframa FZ-x radi lakšeg pozivanja u kasnijim poglavlјima.

Tablica 4.1 Funkcionalni zahtjevi

Oznaka	Naziv zahtjeva	Kratki opis
FZ-1	Registracija korisnika	Sustav mora omogućiti stvaranje korisničkoga računa unosom imena, prezimena, e-pošte, korisničkoga imena i lozinke.
FZ-2	Prijava korisnika	Korisnik se mora moći prijaviti pomoću registriranih vjerodajnica.
FZ-3	Sigurnosni token	Nakon uspješne prijave sustav generira JWT kojim se ovjeravaju naknadni zahtjevi prema poslužitelju.
FZ-4	Profil korisnika	Korisnik mora moći pregledati i ažurirati osobne podatke.
FZ-5	Odjava korisnika	Korisnik se mora moći sigurno odjaviti iz aplikacije.
FZ-6	Promjena lozinke	Sustav mora omogućiti korisniku sigurno ažuriranje postojeće lozinke unosom stare i nove lozinke.

FZ-7	Brisanje osobnih podataka	Na zahtjev korisnika svi njegovi osobni podaci i povezane transakcije moraju biti trajno uklonjeni iz sustava.
FZ-8	Ručni unos troška	Korisnik mora moći ručno unijeti trošak.
FZ-9	Skeniranje računa	Sustav mora omogućiti fotografiranje ili učitavanje slike računa te automatsku ekstrakciju stavki pomoću OCR-a.
FZ-10	Uređivanje unesenih računa	Korisnik mora moći urediti svaki prethodno uneseni račun.
FZ-11	Sigurno brisanje unesenih računa	Korisnik mora moći trajno obrisati pojedini račun pri čemu se povezani podaci uklanjuju iz baze.
FZ-12	Kategorizacija troškova	Svaki trošak mora se moći dodijeliti unaprijed stvorenoj kategoriji.
FZ-13	Upravljanje kategorijama	Korisnik može stvarati, brisati i označiti do četiri omiljene kategorije prikazane na početnom zaslonu.
FZ-14	Vizualizacija podataka	Sustav mora generirati grafikone koji prikazuju potrošnju po vremenu, kategoriji i trgovcu.
FZ-15	Povijest transakcija	Sustav prikazuje popis prethodno unesenih računa s paginacijom i prikazom detalja svakog računa.
FZ-16	Filtriranje i sortiranje	Korisnik mora moći filtrirati i/ili sortirati transakcije prema datumu, iznosu, kategoriji i trgovcu.

4.2. Nefunkcionalni zahtjevi

Nefunkcionalni zahtjevi, označeni šifrom NFZ-x, usmjereni su na kvalitativna svojstva sustava. Za razliku od funkcionalnih zahtjeva, koji definiraju što sustav mora omogućiti, nefunkcionalni zahtjevi specificiraju kako se te funkcionalnosti trebaju ostvariti i pod kojim uvjetima moraju

djelovati. Drugim riječima, oni služe kao mjerila kvalitete koja određuju standarde ponašanja sustava, ali ne zadiru u njegovu poslovnu logiku.

Tablica 4.2 Nefunkcionalni zahtjevi

Oznaka	Naziv zahtjeva	Kratki opis
NFZ-1	Sigurnost prijenosa podataka	Sav promet između klijenta i poslužitelja mora biti zaštićen HTTPS protokolom, a lozinke pohranjene <i>bcrypt</i> algoritmom.
NFZ-2	Autorizacija API-ja	Svaki zahtjev prema zaštićenim rutama mora sadržavati valjan JSON Web Token, a pri isteku tokena pristup se odbija.
NFZ-3	Performanse učitavanja	Ključni zasloni i rezultati obrade moraju se prikazivati bez vidljivog odgađanja za krajnjega korisnika.
NFZ-4	Skalabilnost	Sustav mora održavati stabilne performanse i responzivnost i pri povećanom broju istodobnih korisnika.
NFZ-5	Višeplatformska podrška	Aplikacija mora pružati istu funkcionalnost i iskustvo na podržanim verzijama Android-a i iOS-a.
NFZ-6	Upotrebljivost	Osnovni korisnički tijek (prijava, skeniranje računa, prikaz rezultata) mora biti izvediv bez dodatnih uputa, uz sažetu <i>onboarding</i> sekvencu.
NFZ-7	Pouzdanost servisa	Poslužitelj mora održavati visoku dostupnost, uz evidentiranje i dojavu svake kritične greške.
NFZ-8	Održavanje	Izvorni kod mora biti modularno organiziran sa ažurnom API specifikacijom.

4.3. Ograničenja

Glavna ograničenja sustava proizlaze iz ovisnosti o vanjskim resursima i specifičnostima mobilne okoline. Funkcionalnost izvan mreže je ograničena ili nedostupna jer aplikacija zahtjeva stalnu internetsku vezu za prijavu, sinkronizaciju podataka i pozivanje usluge Google Document AI. Točnost i dostupnost ekstrakcije teksta u potpunosti ovise o navedenoj OCR usluzi, pri čemu svaki zastoj ili degradacija rada izravno utječe na korisničko iskustvo. Točnost prepoznavanja računa dodatno ovisi o kvaliteti kamere uređaja, uključujući rezoluciju, fokus i osvjetljenje pri snimanju. Svi se korisnički podaci trenutno pohranjuju u lokalnoj instanci baze MongoDB na razvojnem računalu, čime se ograničava pristup aplikaciji na jedno okruženje.

U ovoj verziji aplikacije namjerno je uvedeno i nekoliko ograničenja korisničkog sučelja tako da korisnik može označiti najviše četiri omiljene kategorije, a popis računa učitava se u stranicama od po pet zapisa. Ta ograničenja dizajnerska su odluka kojom se održava preglednost sučelja te smanjuje opterećenje memorije i mreže pri dohvaćanju podataka. Vrijednosti su softverski definirane i po potrebi ih je moguće prilagoditi budućim zahtjevima ili većem opsegu podataka.

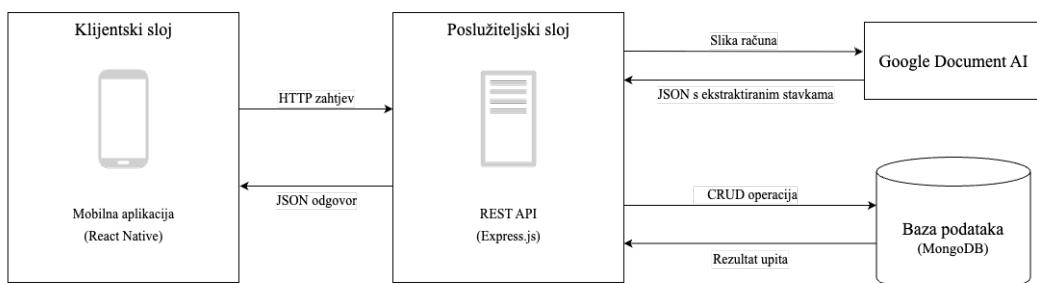
5. ARHITEKTURA SUSTAVA

U ovom se poglavlju razrađuje arhitektura informacijskog sustava za mobilno praćenje i analizu osobnih financija s naglaskom na prevođenje funkcionalnih i nefunkcionalnih zahtjeva u konkretno tehničko rješenje. Prikazuje se trodijelni slojeviti model (klijent, poslužitelj, baza podataka uz Google Document AI), standardizirana komunikacija te ključni tokovi podataka od unosa računa do pohrane i analitike.

5.1. Pregled arhitekture

Aplikacijski sustav organiziran je u tri jasno odijeljena sloja. Klijentski sloj čini mobilna aplikacija izgrađena u React Native-u i on upravlja korisničkim sučeljem, lokalnim stanjem i validacijom unosa te prema potrebi pohranjuje privremene podatke u AsyncStorage. Poslužiteljski sloj implementiran je u Express.js i izložen kao REST API. Taj sloj provodi poslovnu logiku, autorizaciju putem JWT-a, validaciju zahtjeva i orkestrira pristup ostalim resursima. Treći sloj čine baza podataka koja obuhvaća lokalnu instancu baze MongoDB u kojoj se čuvaju korisnički profili i uneseni računi, te vanjska usluga Google Document AI koja se koristi za OCR i strukturiranje skeniranih računa.

Slika 5.1 prikazuje dijagram kojim je predviđen uobičajeni protokol podataka. Mobilna aplikacija, tj. klijent koji pruža korisničko sučelje, šalje HTTP zahtjev s JWT-om prema REST API-ju. Poslužitelj (eng. *server*) obrađuje te zahtjeve, po potrebi dohvaća ili zapisuje podatke u MongoDB bazu te vraća klijentu odgovor u JSON formatu. Kada korisnik skenira ili učita sliku računa, klijent je šalje poslužitelju kao *multipart/form-data* te je poslužitelj proslijeđuje servisu Google Document AI gdje se izdvajaju relevantni entiteti. Document AI vraća strukturirani JSON s ekstrahiranim podacima koje poslužitelj normalizira, po potrebi pohranjuje u MongoDB i vraća klijentu. Klijent rezultate prikazuje kao grafove, kartice ili ažurirane popise, čime se zatvara cijelovit tok komunikacije od korisničkog sučelja do baze podataka i vanjske usluge te natrag.



Slika 5.1 Arhitektura sustava

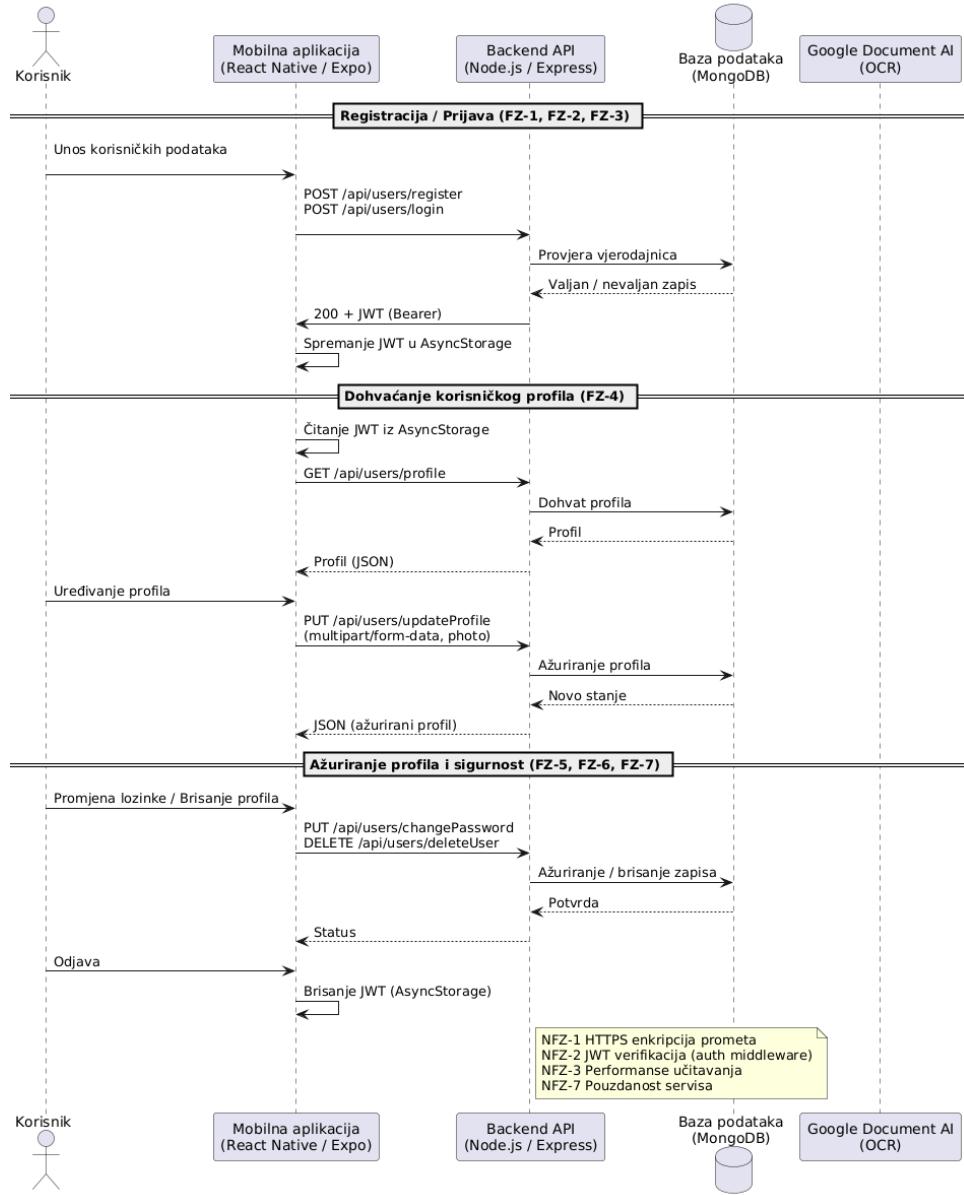
5.2. Tokovi podataka

Prikaz ključnih tokovi podataka obuhvaća interakcije između mobilne aplikacije, poslužiteljskog API-ja, baze podataka i vanjske usluge Google Document AI. Budući da cjelina obuhvaća više zasebnih korisničkih scenarija, zbog lakše čitljivosti dijagram je podijeljen u dva logička dijela: prvi obuhvaća životni ciklus korisnika i sigurnost), a drugi rad s računima, kategorijama i analitikom. Pritom su ucrtani glavni tokovi koji operacionaliziraju funkcionalne zahtjeve (FZ-1 do FZ-16) te su naznačene točke na kojima se provode nefunkcionalni zahtjevi (NFZ-1 do NFZ-8) iz prethodnog poglavlja (4), poput zaštite prometa, autorizacije pristupa, performansi učitavanja i pouzdanosti servisa.

Prvi dijagram (slika 5.2) opisuje autentifikacijski tok koji počinje unosom vjerodajnica nakon čega aplikacija poziva `/api/users/register` ili `/api/users/login`, ovisno o tome da li korisnik stvara novi račun ili se postojeći korisnik prijavljuje u sustav. Poslužitelj provjerava podatke u bazi i u slučaju uspjeha vraća JWT (FZ-1, FZ-2, FZ-3). Token se lokalno sprema u AsyncStorage te se potom koristi pri dohvaćanju profila preko rute `/api/users/profile` (FZ-4).

Ažuriranje profila prikazano je kao zahtjev PUT `/api/users/updateProfile` s podrškom za prijenos fotografije u *multipart/form-data*, pri čemu se novo stanje sprema u bazu i vraća klijentu. U nastavku su objedinjeni tokovi za promjenu lozinke i brisanje korisničkog računa (FZ-6, FZ-7), kao i sigurna odjava (FZ-5) koja lokalno uklanja token.

Sve zaštićene rute prolaze kroz posrednički sloj za provjeru JWT-a, a komunikacija je osigurana HTTPS-om, čime se ispunjavaju nefunkcionalni zahtjevi NFZ-1 (sigurnost prometa) i NFZ-2 (autorizacija API-ja). Performanse i pouzdanost (NFZ-3, NFZ-7) adresiraju se dosljednim korištenjem laganih JSON odgovora i jasnim rukovanjem greškama na poslužitelju.

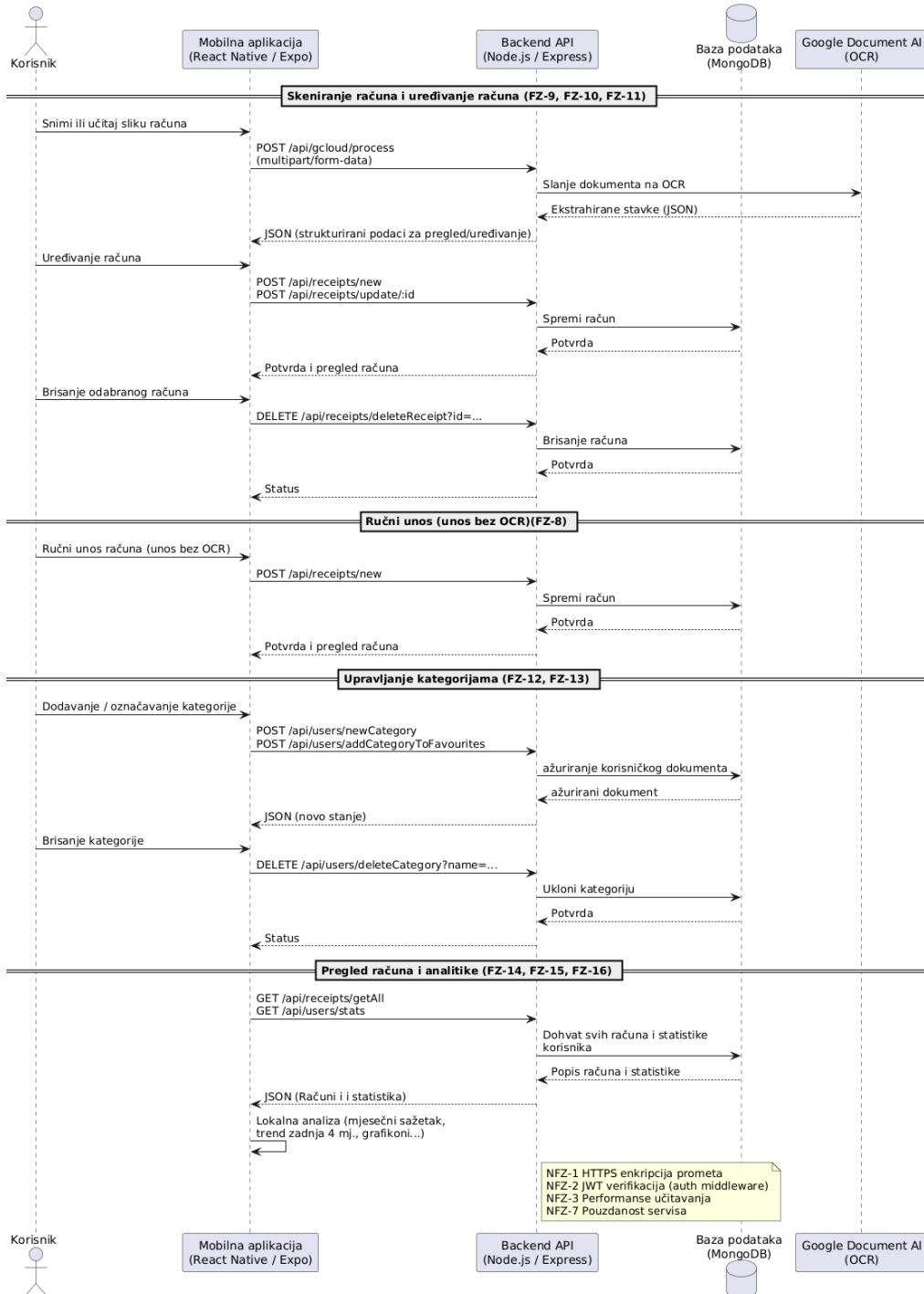


Slika 5.2 Tokovi podataka

Drugi dijagram (slika 5.3) fokusira se na tokove vezane uz račune i analitiku. Kada korisnik snimi račun ili učita sliku iz galerije, aplikacija šalje *multipart/form-data* na */api/gcloud/process*. API prosljeđuje dokument Google Document AI-ju te nakon obrade vraća strukturirane stavke spremne za korisničko uređivanje (FZ-9, FZ-10). Potvrđeni račun se sprema putem */api/receipts/new* ili ažurira preko */api/receipts/update/:id*, dok je brisanje modelirano rutom */api/receipts/deleteReceipt* (FZ-11). Predviđen je i ručni unos bez OCR-a (FZ-8), kojim se izravno šalje sadržaj računa na trajno pohranjivanje.

Upravljanje kategorijama (FZ-12, FZ-13) uključuje stvaranje i označavanje omiljenih kategorija te njihovo brisanje preko rute */api/users/deleteCategory*, pri čemu se promjene pohranjuju u korisničkom dokumentu i vraćaju klijentu kao ažurno stanje.

Pregled povijesti i analitike (FZ-14, FZ-15, FZ-16) započinje dohvatom svih računa i sažetih statistika (`/api/receipts/getAll`, `/api/users/stats`), nakon čega aplikacija lokalno provodi agregacije za mjesečni sažetak, trendove kroz posljednja četiri mjeseca i raspodjelu po kategorijama ili trgovcima. Takva raspodjela posla smanjuje broj mrežnih poziva, poboljšava responzivnost sučelja i pridonosi ispunjenju zahtjeva NFZ-3 (performanse učitavanja), uz istovremeno zadržavanje sigurnosnih jamstava (NFZ-1, NFZ-2).



Slika 5.3 Tokovi podataka

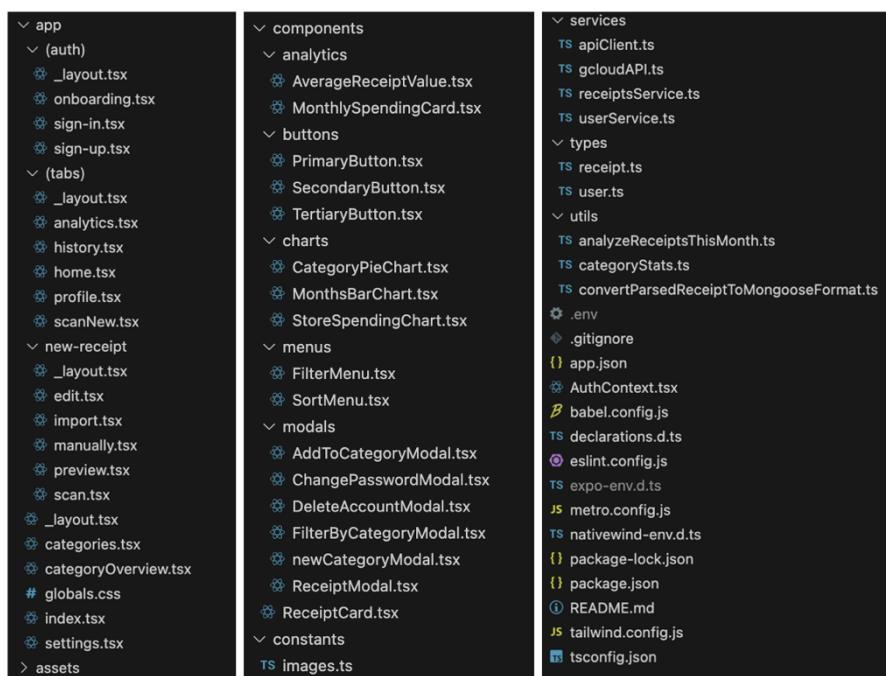
6. IMPLEMENTACIJA KLIJENTSKOG SLOJA

U ovom poglavlju opisana je implementacija klijentskog dijela sustava s naglaskom na arhitekturu, organizaciju koda i komunikaciju s poslužiteljem. Prikazani su ključni korisnički tokovi autentifikacije, unosa i pregleda računa, rad s kategorijama i pregled analitike, a izlaganje je popraćeno odabranim isjećcima koda.

6.1. Struktura projekta

Klijentski dio je izgrađen u okruženju React Native i Expo uz TypeScript, a navigacija je u potpunosti temeljena na *expo-routeru*, odnosno na navigaciji prema strukturi datoteka. Ulazna točka aplikacije *app/index.tsx* provjerava postoji li valjana korisnička sesija, tj. JWT u AsyncStorage-u, te u skladu s time preusmjerava korisnika na odgovarajuću skupinu ruta. Globalni raspored i zajedničke postavke deklarirani su u *app/_layout.tsx*, uz korištenje *safe area* komponenti radi dosljednog prikaza na različitim uređajima i platformama.

Arhitektura ruta organizirana je u tri logične cjeline (slika 6.1). Direktorij *app/* i datoteke koje služe kao ulazne točke i globalni rasporedi. Skupina ruta (*auth/*) s vlastitim rasporedom (*_layout.tsx*) i ekranima *onboarding.tsx*, *sign-in.tsx* i *sign-up.tsx* čini samostalan autentifikacijski tok koji novim korisnicima nudi *onboarding* uvod u aplikaciju i registraciju, dok postojeći korisnici prolaze kroz proces prijave prije ulaska u ostatak aplikacije.



Slika 6.1 Struktura klijentskog sloja

Slijedi skupina (*tabs*)/koja označava glavnu radnu površinu aplikacije nakon prijave s ekranima *home.tsx*, *analytics.tsx*, *history.tsx*, *profile.tsx* i *scanNew.tsx*. Ovi ekrani tvore donju kartičnu (*tab*) navigaciju. *Home* služi kao početni pregled, *analytics* za grafičke prikaze i analizu potrošnje, *history* za pregled ranije unesenih računa, *profile* za osobne podatke i statistike, dok *scanNew* predstavlja ulaznu točku u tok snimanja/uvoza računa.

Poseban tok unosa računa nalazi se u mapi *new-receipt/* te u njoj ekrani *scan.tsx* za fotografiranje računa, *import.tsx* za uvoz iz galerije, *manually.tsx* za ručni unos, *preview.tsx* za pregled unesenog računa te *edit.tsx* za uređivanje računa. Ova izolacija pojednostavljuje održavanje i omogućuje da se cijeli proces unosa i provjere računa razvija i testira neovisno o ostatku aplikacije.

Osim navedenih mapa, u korijenu *app/* nalaze se i dodatni ekrani *categories.tsx* za pregled svih dostupnih kategorija, *categoryOverview.tsx* za pregled određene kategorije, *settings.tsx* za upravljanje postavkama aplikacije te *globals.css* s globalnim stilovima, koji u kombinaciji s Tailwind/NativeWind klasama, osiguravaju dosljednu tipografiju, razmake i boje kroz cijeli projekt.

Mapa *components/* organizirana po podmapama: *analytics/* (kartice i prikazi metrika), *buttons/* (primarni/sekundarni/tercijarni gumbi), *charts/* (kružni graf, stupčasti graf po mjesecima, graf potrošnje po trgovini), *menus/* (izbornici za filtriranje i sortiranje) te *modals/* (modalni prozori za dodavanje kategorije, promjenu lozinke, brisanje računa, pregled računa itd.).

U mapi *services/* nalazi se centralizirani sloj za pristup API-ju *apiClient.ts* koji definira Axios instancu s baznim URL-om i zajedničkim postavkama (slika 6.2), na koju se oslanjaju moduli *gcloudAPI.ts* (pozivi prema Google Cloud Document AI-u), *receiptsService.ts* (CRUD nad računima) i *userService.ts* (autentifikacija i korisnički profil).

```
const apiClient = axios.create({
  baseURL: `http://${process.env.EXPO_PUBLIC_IP}:5001/`,
  headers: { "Content-Type": "application/json" },
});

apiClient.interceptors.request.use(
  async (config) => {
    const token = await AsyncStorage.getItem("token");
    if (token) {
      config.headers.Authorization = `Bearer ${token}`;
    }
    return config;
  },
  (error) => Promise.reject(error)
);

export default apiClient;
```

Slika 6.2 Konfiguracija Axios klijenta

Osim definiranja zajedničke Axios instance s baznim URL-om uveden je i *request interceptor* koji se izvršava prije svakog HTTP poziva i asinkrono dohvata JWT iz AsyncStorage te ga, ako je prisutan, dodaje u zaglavlj u *Authorization: Bearer*. Na taj se način autentifikacija centralizira na jednom mjestu, izbjegava duplicitanje koda po servisima, a pristup zaštićenim rutama postaje transparentan za sve klijentske pozive.

Mapa *types/* koja sadrži tipove domena *receipt.ts* i *user.ts* te *utils/* s pomoćnim funkcijama za izračune i transformacije. Ispod su glavne konfiguracijske i *build* datoteke Expo/React Native okruženja (*app.json*, *metro.config.js*, *babel.config.js*, *tailwind.config.js*, *eslint.config.js*, *tsconfig.json*), kao i *.env* za variabile okruženja.

Opisana arhitektura je slojovita, modularna i usklađena s *expo-router* paradigmom. Skupine ruta jasno odvajaju autentifikaciju, glavne funkcionalnosti i tok unosa računa, dok servisi posreduju u komunikaciji s poslužiteljem, a stilovi i globalne postavke osiguravaju konzistentan vizualni identitet. Navedena organizacija olakšava održavanje, testiranje i postupno proširivanje funkcionalnosti, uz dobru čitljivost koda i predvidljiv tok navigacije.

6.2. Autentifikacija i sesija korisnika

Ovo potpoglavlje obuhvaća cijeli tok autentifikacije i upravljanja sesijom, od prvog pokretanja aplikacije, preko registracije/prijave, do pohrane JWT-a i učitavanja korisničkog profila. Na samom početku korisnika dočekuje kratki *onboarding* zaslon koji informira o funkcionalnostima i nudi dvije akcije: *Get Started* za registraciju novih korisnika i *Log In* za prijavu postojećih korisnika. Budući da *onboarding* sadrži isključivo prikazne komponente bez domenskih funkcija, nije izdvojeno u zasebno potpoglavlje.

U nastavku se fokus prebacuje na implementaciju autentifikacije, izdavanje i pohranu JWT tokena u AsyncStorage te dohvati i sinkronizaciju podataka profila.

6.2.1. Registracija

Komponenta *SignUp* (*sign-up.tsx*) implementira cijeli tok registracije tako da se prvo provodi validacija forme nakon čega slijedi poziv servisa i inicijalizacije korisničke sesije. Funkcija *handleRegistration* provodi slojevitu provjeru svih polja prije slanja zahtjeva poslužitelju (slika 6.3). Ime i prezime moraju imati od 2 do 15 znakova te sadržavati samo slova, razmake, apostrofe i crtice. Korisničko ime je ograničeno na 3 do 20 znakova, dopušta slova, znamenke i donju crtu, pri čemu se spriječavaju nepravilni obrasci (npr. uzastopne donje crte). E-pošta se validira uobičajenim obrascem *user@domain.tld*, dok lozinka mora imati najmanje 6 znakova.

te se dodatno uspoređuju polja password i password2. Neuspješne provjere prekidaju tok s odgovarajućom porukom putem *Alert.alert*, čime se izbjegavaju nepotrebni mrežni pozivi.

```

if (!name || name.trim().length === 0 || name.length < 2 || name.length > 15 || !/^[a-zA-Z\s'-]+$/ .test(name.trim()) ) {
    Alert.alert('Error', `Please enter valid name between 2 and 15 characters. Name can only contain letters, spaces, apostrophes, and hyphens. `);
    return;
}
if (!surname || surname.trim().length === 0 || surname.length < 2 || surname.length > 15 || !/^[a-zA-Z\s'-]+$/ .test(surname.trim()) ) {
    Alert.alert('Error', `Please enter valid surname between 2 and 15 characters. surname can only contain letters, spaces, apostrophes, and hyphens.`);
    return;
}
if (!username || username.trim().length === 0 || username.length < 3 || username.length > 20 || !/[a-zA-Z0-9_]+$/ .test(username) || username.includes('__')) {
    Alert.alert( 'Error', `Please enter valid username. Username must be between 3 and 20 characters, only containing letters, numbers, and underscores, and without consecutive underscores.`);
    return;
}
if (!email || email.trim().length === 0 || !/^\w+@\w+\.\w+$/ .test(email.trim())) {
    Alert.alert('Error', 'Please enter valid email.');
    return;
}
if (!password || !password2 || password.length < 6) {
    Alert.alert('Error', 'Please enter valid password. Password must be at least 6 characters long.');
    return;
}
if (password !== password2) {
    Alert.alert('Error', 'Passwords do not match.');
    return;
}

```

Slika 6.3 Validacija unosa podataka

Ako sve provjere prođu uspješno, poziva se servis *registerUser* te se vraćeni JWT spremi u trajnu pohranu pomoću asinkrone funkcije *AsyncStorage.setItem* za pohranu tokena (slika 6.4). Time se inicijalizira korisnička sesija, a daljnji pozivi prema API-ju automatski nose *Bearer* token kroz globalni Axios *interceptor*. Nakon uspješne registracije korisnik se preusmjerava na početni zaslon, dok eventualne pogreške rezultiraju informativnom porukom preko *Alert.alert*.

```

try {
    const response = await registerUser(username, email, password, name, surname);
    await AsyncStorage.setItem('token', response.token);
    router.replace('/');
} catch (error) {
    Alert.alert('Registration Failed', 'Something went wrong. Please try again.');
}

```

Slika 6.4 Pohrana tokena i preusmjeravanje

Slika 6.5 prikazuje servisni poziv koji realizira registraciju korisnika prema poslužitelju putem POST zahtjeva na ruti */api/users/register*. Funkcija *registerUser* koristi zajednički HTTP klijent (*ApiClient*) te predaje strukturirano tijelo zahtjeva s unesenim poljima korisničko ime (eng. *username*), email, lozinka (eng. *password*), ime (eng. *name*) i prezime (eng. *surname*). Uspješan odgovor poslužitelja vraća se klijentu sa priloženim JWT koji se potom pohranjuje na klijentu.

```

export const registerUser = async (username:string, email:string, password:string, name:string, surname:string): Promise<User> => {
  try {
    const response = await apiClient.post('/api/users/register', { username, email, password, name, surname, });
    return response.data;
  } catch (error: any) {
    if (error.response && error.response.data && error.response.data.message) {
      throw new Error(error.response.data.message);
    } else {
      throw new Error("Error registering. Please try again.");
    }
  }
};

```

Slika 6.5 Servisna funkcija za registriranje korisnika

6.2.2. Prijava postojećeg korisnika

U komponenti za prijavu funkcija *handleLogin* (slika 6.6) upravlja korisničkim tokom prijave tako da provodi osnovnu provjeru prisutnosti korisničkog imena i lozinke, a u slučaju praznih polja prikazuje informativnu poruku *Alert*. Nakon validacije poziva servis *loginUser*, preuzima izdani JWT i spremi ga u AsyncStorage radi pohranjivanja sesije te izvršava navigaciju *router.replace('/')* kako bi korisnika preusmjerio u autentificirani kontekst. Izuzeci se hvataju u *try/catch* bloku i komuniciraju korisniku putem *Alert* obavijesti, čime se osigurava robusnost rukovanja pogreškama.

```

const handleLogin = async () => {
  if (!username || !password) {
    Alert.alert('Error', 'Please enter both username and password.');
    return;
  }
  try {
    const response = await loginUser(username, password);
    await AsyncStorage.setItem('token', response.token);
    router.replace('/');
  } catch (error) {
    Alert.alert('Login Failed', 'Something went wrong. Please try again.');
  }
};

```

Slika 6.6 Funkcija za prijavu korisnika

Funkcija *loginUser* (slika 6.7) realizira pozadinski HTTP poziv prema */api/users/login* korištenjem *apiClient.post*, pri čemu poslužitelju proslijeđuje korisničko ime i lozinka. Uspješan odgovor vraća tijelo s JWT-om, dok se kod pogreške preferira poruka vraćena sa servera (*error.response.data.message*), a ako nije dostupna generira se opća poruka.

```

export const loginUser = async (username: string, password: string): Promise<User> => {
  try {
    const response = await apiClient.post('/api/users/login', { username, password, });
    return response.data;
  } catch (error: any) {
    if (error.response && error.response.data && error.response.data.message) {
      throw new Error(error.response.data.message);
    } else {
      throw new Error("Error logging in. Please try again.");
    }
  }
};

```

Slika 6.7 Servisna funkcija za prijavu korisnika

6.2.3. Korisnički profil

Zaslon profila koristi aktivnu sesiju i dohvaća podatke korisnika putem zaštićenih API ruta kako bi omogućio personalizirano iskustvo (npr. prikaz slike profila, osnovnih podataka i pregled statistika potrošnje).

Slika 6.8 prikazuje funkciju *fetchUserProfile* koja dohvaća osnovne podatke o korisniku (*getMyProfile*) i prateće statistike potrošnje (*getUserStats*). Dobiveni profil se postavlja u stanje (*setUser*, *setEditUser*), a ako korisnik ima vlastitu sliku priprema se URI za prikaz i uređivanje (*setUserPhoto*, *setEditUserPhoto*). Statistički podaci se pohranjuju u *setStats*, dok se eventualne greške komuniciraju korisniku putem *Alert* poruke, čime se osigurava robusno rukovanje pogreškama u prikazu profila.

```
const fetchUserProfile = useCallback(async () => {
  try {
    const profile = await getMyProfile();
    const statsData = await getUserStats();
    setUser(profile);
    if (profile.photo !== "/public/images/profile-picture.png") {
      setUserPhoto({ uri: profile.photo });
      setEditUserPhoto({ uri: profile.photo });
    }
    setEditUser(profile);
    setStats(statsData);
  } catch (error) {
    Alert.alert("Error", "Failed to load profile data.");
  }
}, [ ]);
```

Slika 6.8 Dohvat korisničkih podataka i statistika

Slika 6.9 prikazuje servisnu funkciju *getMyProfile* koja šalje autorizirani GET zahtjev na */api/users/profile*. Uspješan odgovor vraća se klijentu i koristi se u komponenti profila za popunjavanje osnovnih korisničkih podataka, dok funkcija *getUserStats* (slika 6.10) dohvaća agregirane statistike korisnika slanjem autoriziranog GET zahtjeva na */api/users/stats*. Vraćeni podaci koriste se za prikaz sažetaka potrošnje (npr. broj računa, potrošnja po periodu).

```
export const getMyProfile = async () => {
  try {
    const response = await apiClient.get('/api/users/profile');
    return response.data;
  } catch (error) {
    throw new Error("Error loading user profile.");
  }
}
```

Slika 6.9 Servisna funkcija za dohvaćanje profila

```
export const getUserStats = async () => {
  try {
    const response = await apiClient.get('/api/users/stats');
    return response.data;
  } catch (error) {
    throw new Error('Error loading user stats.');
  }
}
```

Slika 6.10 Dohvaćanje statistike korisnika

Evidentiranje izmjena profila pokreće se u komponenti kroz funkciju *handleSaveEdit* (slika 6.11). Nakon osnovne provjere da su obavezna polja popunjena, funkcija poziva servisnu operaciju za ažuriranje profila s trenutno uređivanim vrijednostima (*editUser*). Uspješan odgovor rezultira zatvaranjem načina uređivanja (*setEditMode(false)*) te osvježavanjem prikaza lokalnim stanjem *setUser(editUser)*, čime korisnik odmah vidi primijenjene izmjene.

```

const handleSaveEdit = async () => {
  try {
    if(editUser.name && editUser.surname) {
      const updated = await updateProfile({
        name: editUser.name,
        surname: editUser.surname,
        location: editUser.location,
        bio: editUser.bio,
        photo: editUser.photo,
      });
      const newUrl = updated?.photo || user?.photo;
      const freshUrl = newUrl ? `${newUrl}?t=${Date.now()}` : undefined;
      setUser((prev: any) => ({ ...prev, ...updated, photo: newUrl }));
      setUserData(freshUrl ? { uri: freshUrl } : images.profile_picture);
      setEditingMode(false);
    }
  } catch (error) {
    Alert.alert("Error", "Failed to update profile.");
  }
}

```

Slika 6.11 Funkcija za spremanje izmjena profila

Pozadinska obrada izmjena odvija se u servisnoj funkciji *updateProfile* (slika 6.12). Podaci se prikupljaju u *FormData* objekt (samo prisutna polja), a za opcionalnu fotografiju profilne slike dodaje se zapis s ispravnim MIME tipom i imenom datoteke. Zahtjev se šalje kao PUT prema rutu */api/users/updateProfile*, uz zaglavje *Content-Type: multipart/form-data*. Funkcija vraća tijelo odgovora ili, u slučaju pogreške, baca iznimku koja se obrađuje na razini komponente.

```

export const updateProfile = async (updatedData: UpdatedProfileData) => {
  try {
    const formData = new FormData();
    if (updatedData.name) formData.append("name", updatedData.name);
    if (updatedData.surname) formData.append("surname", updatedData.surname);
    if (updatedData.location) formData.append("location", updatedData.location);
    if (updatedData.bio) formData.append("bio", updatedData.bio);
    if (updatedData.photo?.uri) {
      const fileUri = updatedData.photo.uri;
      const fileType = mime.getType(fileUri) || "image/jpeg";
      const fileName = fileUri.split("/").pop();
      formData.append("photo", { uri: fileUri, name: fileName, type: fileType, } as any);
    }
    const response = await apiClient.put('/api/users/updateProfile', formData, {headers: { "Content-Type": "multipart/form-data", }});
    return response.data;
  } catch (error) {
    throw new Error('Error updating profile.');
  }
}

```

Slika 6.12 Servisna funkcija za spremanje izmjena profila

Funkcija *handleLogout* (slika 6.13) provodi zatvaranje sesije tako da u okviru *try/catch* bloka trajno uklanja JWT iz AsyncStoragea, lokalno se stanje čisti postavljanjem korisnika na *null* i brisanjem tokena, a zatim se korisnik preusmjerava na početni (*onboarding*) ekran. U slučaju iznimke prikazuje se poruka putem *Alert*, čime se osigurava kontrolirani prekid sesije i povratak u neautentificirano stanje.

```

const handleLogout = async () => {
  try {
    await AsyncStorage.removeItem("token");
    setUser(null);
    setToken('');
    router.replace("/onboarding");
  } catch (error) {
    Alert.alert("Error", "Failed to log out.");
  }
};

```

Slika 6.13 Čišćenje sesije i povratak na onboarding

Promjena lozinke odvija se u zasebnom modalnom prozoru *ChangePasswordModal*, koji se otvara iz zaslona postavki i sadrži dva ulazna polja (trenutna i nova lozinka), akcije *Cancel* za odustajanje i *Save* za spremanje te indikator učitavanja preko poluprozirnog *overlaya*. Modal lokalno upravlja stanjem i validacijom te u sklopu njega funkcija *handleSave* (slika 6.14) provodi tok promjene lozinke. Najprije se provodi klijentska validacija, odnosno oba polja moraju biti popunjena, a nova lozinka mora imati najmanje šest znakova, a u suprotnom se postupak prekida uz prikaz poruke korisniku. U *try* bloku se postavlja indikator učitavanja, poziva se servis *changePassword*, a uspjeh se potvrđuje putem *Alert* poruke, nakon čega se lokalno stanje resetira (prazne se polja) i modal zatvara pozivom *onClose()*.

```

const handleSave = async () => {
  if (!currentPassword || !newPassword) {
    Alert.alert("Error", "Both fields are required");
    return;
  }
  if (newPassword.length < 6) {
    Alert.alert("Error", "New password must be at least 6 characters");
    return;
  }
  try {
    setLoading(true);
    await changePassword(currentPassword, newPassword);
    Alert.alert("Success", "Password changed successfully");
    setCurrentPassword("");
    setNewPassword("");
    onClose();
  } catch (err: any) {
    Alert.alert("Error", err.message);
  } finally {
    setLoading(false);
  }
};

```

Slika 6.14 Promjena lozinke

Servisna funkcija *changePassword* (slika 6.15) izvodi mrežni poziv za promjenu lozinke. Implementirana je kao PUT zahtjev prema ruti */api/users/changePassword*, s tijelom koje sadrži trenutnu i novu lozinku.

```

export const changePassword = async (currentPassword: string, newPassword: string): Promise<User> => {
  try {
    const response = await apiClient.put('/api/users/changePassword', { currentPassword, newPassword, });
    return response.data;
  } catch (error: any) {
    const message = error?.response?.data?.message || "Error changing password.";
    throw new Error(message);
  }
};

```

Slika 6.15 Servisna funkcija za promjenu lozinke

Brisanje korisničkog profila ostvaraju se *DeleteAccountModal* funkcijom, koja se poziva iz odjeljka *Settings* i korisniku eksplicitno naglašava nepovratnost operacije (tekstualno upozorenje u modalu te destruktivan akcijski gumb). U obradi događaja *handleDelete* (slika 6.16) provodi se validacija (obavezan unos lozinke), nakon čega se postavlja indikator učitavanja i poziva servis *deleteUser*. Uspješan odgovor rezultira obavijesti o uspjehu, čišćenjem lokalnog stanja i zatvaranjem modala, a pogreške se korisniku prikazuju preko *Alert* dok se u *finally* grani uvijek resetira indikator učitavanja.

```
const handleDelete = async () => {
  if (!password) {
    Alert.alert("Error", "Password is required");
    return;
  }
  try {
    setLoading(true);
    await deleteUser(password);
    Alert.alert("Success", "Password changed successfully");
    setPassword("");
    onClose();
  } catch (err: any) {
    Alert.alert("Error", err.message);
  } finally {
    setLoading(false);
  }
};
```

Slika 6.16 Brisanje profila

Slika 6.17 prikazuje servisnu funkciju *deleteUser* koja briše profil putem poziva poslužitelju kao DELETE zahtjev prema ruti */api/users/deleteUser* s tijelom koje sadrži lozinku korisnika.

```
export const deleteUser = async (password: string): Promise<User> => {
  try {
    const response = await apiClient.delete('/api/users/deleteUser', { params: { password } });
    return response.data;
  } catch (error: any) {
    const message = error?.response?.data?.message || "Error deleting account.";
    throw new Error(message);
  }
};
```

Slika 6.17 Servisna funkcija za brisanje profila

6.3. Unos i pregled računa

Tok unosa računa u mobilnoj aplikaciji obuhvaća tri ulazne točke: snimanje kamerom (*scan.tsx*), uvoz postojeće fotografije iz galerije (*import.tsx*) i ručni unos (*manually.tsx*). Bez obzira na ulaz, cilj je dobiti strukturirani objekt koji se potom pregledava, po potrebi uređuje te spremi preko REST API-ja.

6.3.1. Skeniranje računa kamerom

Slika 6.18 prikazuje funkciju *openCamera* iz *scan.tsx* koja najprije traži dopuštenje za kameru (*requestCameraPermissionsAsync*), a zatim otvara kameru preko *launchCameraAsync* s postavkama za slike, uređivanje i *Base64* enkodiranje. U slučaju odustajanja vraća se na prethodni ekran, dok se pri uspješnom snimanju u stanje spremaju URI fotografije i njezin

Base64 sadržaj. Slika 6.19 prikazuje funkciju *detectTextFromImage* koja odabranu sliku šalje poslužitelju putem servisa *processReceiptImage*, prima rezultat Document AI-ja te ga odmah prevodi u domenski model pomoću *convertParsedReceiptToMongooseFormat*. Tijekom poziva upravlja se indikatorom učitavanja, a u slučaju pogreške stanje se vraća na inicijalni račun, dok se nakon uspješne obrade postavlja zastavica za nastavak procesa i prelazi na uređivanje.

```
const openCamera = async () => {
  const { status } = await ImagePicker.requestCameraPermissionsAsync();
  if (status !== 'granted') {
    alert('Camera permission is required');
    return;
  }
  const result = await ImagePicker.launchCameraAsync({
    mediaTypes: ImagePicker.MediaTypeOptions.Images,
    allowsEditing: true,
    quality: 1,
    base64: true,
  });
  if (result.canceled) {
    router.back();
    return;
  }
  if (result.assets[0].base64) {
    setImage(result.assets[0].uri);
    setImageBase64(result.assets[0].base64);
  }
};
openCamera();
```

Slika 6.18 Otvaranje kamere i dohvata fotografije

```
const detectTextFromImage = async () => {
  if (!image) return;
  setLoading(true);
  setReceipt(initialReceipt);
  try {
    const result = await processReceiptImage({
      uri: image,
      type: 'image/jpeg',
      name: 'receipt.jpg',
    });
    setReceipt(convertParsedReceiptToMongooseFormat(result));
    setProceed(true);
  } catch (error) {
    console.error(error);
    setReceipt(initialReceipt);
  } finally {
    setLoading(false);
  }
};
```

Slika 6.19 Poziv obrade i mapiranje rezultata

Slanje fotografije računa poslužitelju odvija se putem servisnog poziva *processReceiptImage* iz datoteke *gcloudAPI.ts* (slika 6.20). Funkcija formira *FormData* i u polje *file* dodaje objekt s URI, MIME tipom i nazivom datoteke, a zatim preko *axios.post* upućuje zahtjev poslužitelju rutom */api/gcloud/process* uz zaglavljem *Content-Type: multipart/form-data*. Povratna vrijednost je normalizirani JSON s poslužitelja, dok se u slučaju pogreške detalji zapisuju u konzolu i prosljeđuje se iznimka s porukom „Receipt processing failed“.

```
import axios from 'axios';

export async function processReceiptImage(photo: { uri: string; type: string; name: string }) {
  const formData = new FormData();
  formData.append('file', {
    uri: photo.uri,
    type: photo.type || 'image/jpeg',
    name: photo.name || 'receipt.jpg',
  } as any);

  try {
    const response = await axios.post(`http://${process.env.EXPO_PUBLIC_IP}:5001/api/gcloud/process`, formData, {
      headers: {
        'Content-Type': 'multipart/form-data',
      },
    });
    return response.data;
  } catch (error: any) {
    console.error('Error processing receipt:', error.response?.data || error.message);
    throw new Error('Receipt processing failed.');
  }
}
```

Slika 6.20 Slanje slike na poslužitelj

6.3.2. Uvoz fotografije iz galerije i ručni unos

Proces uvoza postojeće slike računa iz galerije prikazan je kao slika 6.21. Funkcija *pickImage* iz *import.tsx* otvara galeriju putem *ImagePicker.launchImageLibraryAsync* s filtriranjem na slike, mogućnošću uređivanja i opcijom *Base64* postavljenom na *boolean* vrijednost *true*. Ako korisnik odustane, aplikacija se vraća na prethodni ekran, a u suprotnom se u stanje spremaju URI i Base64 sadržaj odabrane fotografije.

```
const pickImage = async () => {
  const result = await ImagePicker.launchImageLibraryAsync({
    mediaTypes: ImagePicker.MediaTypeOptions.Images,
    allowsEditing: true,
    base64: true,
  });
  if (result.canceled) {
    router.back();
    return;
  }
  if (result.assets && result.assets[0].base64) {
    setImage(result.assets[0].uri);
    setImageBase64(result.assets[0].base64);
  }
};
```

Slika 6.21 Uvoz računa iz galerije

Daljnja obrada u potpunosti slijedi isti tok kao kod snimanja kamerom odnosno poziva se već ranije opisana *detectTextFromImage* (slika 6.19), koja preko servisa *processReceiptImage* (slika 6.20) šalje sliku poslužitelju (*multipart/form-data* na rutu */api/gcloud/process*) i preuzima strukturirani odgovor Document AI-a za popunjavanje podataka računa.

Za ručni unos predviđen je ekran *manually.tsx*, koji ne sadrži vlastiti formular, nego odmah preusmjerava korisnika na zajednički editor *edit.tsx*. Time se potpuno zaobilazi OCR tok (Document AI), a korisnik unosi stavke, iznose, trgovinu i ostale metapodatke izravno u istom sučelju koje se koristi i za naknadna uređivanja. Takav *redirect* pristup smanjuje dupliciranje koda jer se koristi jedan editor za sve slučajeve.

6.3.3. Uređivanje i pretpregled

Komponenta *EditReceipt* predstavlja jedinstveni editor računa koji se koristi i pri kreiranju novog zapisa i pri doradi postojećeg. Po dolasku s prethodnih ekrana (skenera, uvoza ili ručnog unosa) editor preuzima inicijalne vrijednosti kroz *useLocalSearchParams* i inicijalizira lokalno stanje (naziv trgovine, datum, stavke, metoda plaćanja, tagovi i bilješka). Slika 6.22 demonstrira tipizirani ulaz i početno punjenje forme iz prethodnih koraka toka.

```

const paymentMethods: PaymentMethod[] = ["Cash", "Card", "Mobile", "Other"];
const isPaymentMethod = (v: unknown): v is PaymentMethod => paymentMethods.includes(v as PaymentMethod);

const EditReceipt = () => [
  const router = useRouter();
  const { data, mode, receiptId } = useLocalSearchParams();
  const parsedData = JSON.parse(data as string);

  const [store, setStore] = useState(parsedData.store);
  const [date, setDate] = useState(parsedData.date ? new Date(parsedData.date) : undefined);
  const [createdAt, setCreatedAt] = useState(mode === 'existing' ? new Date(parsedData.createdAt) : undefined);
  const [updatedAt, setUpdatedAt] = useState(mode === 'existing' ? new Date(parsedData.updatedAt) : undefined);
  const [totalAmount, setTotalAmount] = useState(parsedData.totalAmount);
  const [items, setItems] = useState(parsedData.items);
  const [paymentMethod, setPaymentMethod] = useState<PaymentMethod>(
    isPaymentMethod(parsedData.paymentMethod) ? parsedData.paymentMethod : "Other"
  );
  const [note, setNote] = useState(parsedData.note);
  const [tags, setTags] = useState(parsedData.tags);
]

```

Slika 6.22 Inicijalizacija i preuzimanje parametara

Nakon što korisnik uneše naziv trgovine i datum, doda ili uredi stavke te po potrebi odabere način plaćanja, tagove i bilješku, obrazac se validira i priprema za spremanje. U metodi *handleSave* provodi se provjera obveznih polja (naziv trgovine, datum, barem jedna stavka), semantička validacija svake stavke (naziv, pozitivna jedinična cijena i količina), provjera ukupnog iznosa i ispravnosti metode plaćanja. Nakon toga slijedi *try/catch* blok funkcije *handleSave* (slika 6.23) koji ovisno o *mode* varijabli poziva *updateReceipt* za ažuriranje postojećeg računa ili *createReceipt* za kreiranje novog računa. Uspješan ishod potvrđuje se *Alert* porukom i navigacijom na */history* (uz prosljeđivanje *receiptId*), dok se u *catch* grani prikazuje poruka o pogrešci i bilježi detalj u konzoli.

```

try {
  if(mode === "existing") {
    if (!receiptId) {
      Alert.alert("Error", "Missing receipt ID for update.");
      return;
    }
    await updateReceipt(receiptId as string, receiptData);
    Alert.alert("Success", "Receipt updated successfully.");
    router.push({
      pathname: "/history",
      params: { receiptId },
    });
  } else {
    await createReceipt(receiptData);
    Alert.alert("Success", "Receipt created successfully.");
    router.push({
      pathname: "/history",
      params: { receiptId },
    });
  }
} catch (error) {
  Alert.alert("Error", "Failed to update receipt.");
  console.error(error);
}

```

Slika 6.23 Try/catch blok funkcije *handleSave*

U sloju servisa implementirane su funkcije za stvaranje i ažuriranje računa koje kapsuliraju pozive prema REST API-ju. Funkcija *createReceipt* (slika 6.24) koristi zajedničku Axios instancu te šalje podatke na rutu */api/receipts/new*. Povratna vrijednost je sadržaj odgovora, dok se pogreške pretvaraju u iznimke koje obrađuju viši sloj sučelja.

```

export const createReceipt = async (data: CreateReceipt) => {
  try {
    const response = await apiClient.post('/api/receipts/new', { ...data, });
    return response.data;
  } catch (error) {
    throw new Error("Error creating receipt.");
  }
}

```

Slika 6.24 Servisni poziv *createReceipt*

Slično tome, funkcija *updateReceipt* (slika 6.25) izvršava operaciju PUT prema */api/receipts/update/{receiptId}*, pri čemu je ključna razlika *path* parametar *receiptId* koji služi za pronađak postojećeg računa u bazi i njegovo ažuriranje.

```

export const updateReceipt = async (receiptId: string, data: CreateReceipt) => {
  try {
    const response = await apiClient.put(`/api/receipts/update/${receiptId}`, { ...data, });
    return response.data;
  } catch (error) {
    throw new Error("Error updating receipt.");
  }
}

```

Slika 6.25 Servisni poziv *updateReceipt*

Uz opciju *Save*, editor nudi i akcije *Cancel* i *Preview*. Slika 6.26 prikazuje funkciju *handleCancel* koja sprječava slučajni gubitak izmjena i prikazuje *Alert* s dvama gumbima (*No* i *Yes*), pri čemu potvrda poziva *router.back()* i vraća korisnika na prethodni ekran. Slika 6.27 prikazuje funkciju *handlePreview* koja iz trenutačnih vrijednosti forme formira objekt zahtjevate putem *expo-router-a* na rutu */new-receipt/preview* proslijeđuje podatke kao *JSON.stringify(receiptData)*.

```

const handleCancel = () => {
  Alert.alert(
    "Cancel Editing",
    "Are you sure you want to cancel your changes?",
    [
      { text: "No", style: "cancel" },
      {
        text: "Yes",
        onPress: () => {
          router.back();
        }
      }
    ]
  );
}

```

Slika 6.26 Potvrda odustajanja od uređivanja

```

const handlePreview = () => {
  const receiptData: CreateReceipt = {
    store: store.trim(),
    date: date ? date.toISOString() : undefined,
    items: items ? items : [],
    totalAmount: totalAmount,
    paymentMethod: paymentMethod,
    note: note ? note : "",
    tags: tags ? tags : []
  }
  router.push({
    pathname: "/new-receipt/preview",
    params: {
      data: JSON.stringify(receiptData),
      mode: mode,
    }
  });
}

```

Slika 6.27 Priprema podataka i navigacija na pregled

Dobiveni podatci iz editora (*EditReceipt*) predaju se komponenti *PreviewReceipt* kao serijalizirani JSON kroz parametar rute data. U komponenti se taj parametar dohvata preko *useLocalSearchParams*, deserijalizira i normalizira u objekt *receipt* sa zadanim vrijednostima. Datum se pritom odmah formatira u lokalizirani niz pomoću *toLocaleDateString("en-GB")*, čime se osigurava konzistentan prikaz bez dodatne obrade u prikaznom sloju. Ovaj korak

pripreme podataka prikazan je kao slika 6.28 te osigurava da su podaci standardizirani i odmah spremni za prikaz u pretpregledu.

```
const PreviewReceipt = () => {
  const router = useRouter();
  const { data, mode } = useLocalSearchParams();
  const parsedData = JSON.parse(data as string);
  const dateISO = parsedData.date ?? undefined;
  const dateDisplay = parsedData.date ? new Date(parsedData.date).toLocaleDateString("en-GB") : "";

  const receipt = {
    store: parsedData.store || "",
    date: dateISO,
    items: parsedData.items || [],
    totalAmount: parsedData.totalAmount,
    paymentMethod: parsedData.paymentMethod || "Other",
    note: parsedData.note || "",
    tags: parsedData.tags || []
  }
}
```

Slika 6.28 Pretvorba parametra rute u objekt pretpregleda

6.3.4. Pregled računa

Zaslon *History* služi za pregled svih računa prijavljenog korisnika. Dohvaća sve korisnikove račune, omogućuje filtriranje po trgovini, rasponu cijene, metodi plaćanja, vremenskom intervalu, sortiranje po datumu unosa, datumu računa ili ukupnom iznosu uz smjer *asc/desc*, te paginaciju za učitavanje određenog broja računa. Detalji računa otvaraju se u modalu *ReceiptModal*, a filtri se unose kroz komponentu *FilterMenu*.

Pri povratku fokusa na zaslon, lista se deterministički osvježava pomoću *useFocusEffect*. Ona poziva memoriranu asinkronu funkciju *fetchReceipts* (slika 6.29) definiranu preko *useCallback* React *hooka*. Time su omogućeni zadržavanje stabilne reference i izbjegavanje nepotrebnih ponovnih učitavanja. Unutar nje se poziva servis *getMyReceipts()*, dohvata račune te rezultat spremi u lokalno stanje *receipts*, uz resetiranje broja vidljivih stavki na početnu vrijednost.

```
const fetchReceipts = useCallback(async () => {
  const data = await getMyReceipts();
  setReceipts(data);
  setVisibleCount(PAGE_SIZE);
}, []);

useFocusEffect(useCallback(() => { fetchReceipts(); }, [fetchReceipts]));
```

Slika 6.29 Osježavanje liste na fokus zaslona

Funkcija *getMyReceipts* (slika 6.30), u sloju *services* izvršava GET zahtjev na rutu */api/receipts/getAll* preko globalnog Axios *interceptora* koji automatski dodaje *Bearer token* zaglavlje.

```
export const getMyReceipts = async () => {
  try {
    const response = await apiClient.get('/api/receipts/getAll');
    return response.data;
  } catch (error) {
    throw new Error("Error loading receipts.");
  }
}
```

Slika 6.30 Servisni poziv za dohvati svih računa

Metoda `getFilteredReceipts` (slika 6.31) slijedno primjenjuje sve odabране kriterije kao što je djelomično podudaranje naziva trgovine, točna metoda plaćanja, numeričke granice ukupnog iznosa (`minPrice/maxPrice`), sigurna konverzija i `null`-toleranciju te vremenski raspon na temelju datuma. Rezultat je niz računa koji zadovoljavaju sve uvjete i predstavlja ulaz u fazu sortiranja.

```
const getFilteredReceipts = () => {
  return receipts.filter((receipt) => {
    if (
      filterOptions.storeName &&
      !receipt.store?.toLowerCase().includes(filterOptions.storeName.toLowerCase())
    ) return false;

    if (
      filterOptions.paymentMethod &&
      receipt.paymentMethod !== filterOptions.paymentMethod
    ) return false;

    const min = parseFloat(filterOptions.minPrice);
    const max = parseFloat(filterOptions.maxPrice);
    if (!isNaN(min) && (receipt.totalAmount ?? 0) < min) return false;
    if (!isNaN(max) && (receipt.totalAmount ?? 0) > max) return false;

    const receiptDate = new Date(receipt.date);
    if (filterOptions.startDate && receiptDate < filterOptions.startDate) return false;
    if (filterOptions.endDate && receiptDate > filterOptions.endDate) return false;

    return true;
  });
};
```

Slika 6.31 Filtriranje računa

Nad filtriranim skupom primjenjuje se sortiranje prema `sortOption` (slika 6.32) tako da `recent` (datum unosa) koristi `createdAt`, `date` koristi datum računa, a `total` ukupni iznos `totalAmount`. Smjer određuje `sortDirection` (uzlazno/silazno) te se dobiveni niz potom dijeli na `visibleCount` elemente za *Load more* paginaciju, čime se postižu responzivan prikaz i stabilne performanse i na većim skupovima podataka.

```
const getSortedReceipts = () => {
  const filtered = getFilteredReceipts();
  const sorted = [...filtered];
  sorted.sort((a, b) => {
    let valA: number = 0;
    let valB: number = 0;

    switch (sortOption) {
      case "recent":
        valA = new Date(a.createdAt!).getTime();
        valB = new Date(b.createdAt!).getTime();
        break;
      case "date":
        valA = new Date(a.date).getTime();
        valB = new Date(b.date).getTime();
        break;
      case "total":
        valA = a.totalAmount ?? 0;
        valB = b.totalAmount ?? 0;
        break;
    }
    return sortDirection === "asc" ? valA - valB : valB - valA;
  });
  return sorted;
};
```

Slika 6.32 Sortiranje računa

Nakon što se popis računa osvježi i prikaže, dodir na bilo koji zapis otvara modalni prikaz detalja `ReceiptModal` (slika 6.33). `ReceiptModal` prima odabrani račun preko svojstva `receipt`, a u suprotnom odmah izlazi. Također, definira funkciju `handleReceiptEditPress` koja najprije

zatvara modal, a zatim otvara ranije opisani uređivač računa na rutu `/new-receipt/edit` u načinu rada *existing*, uz proslijedjeni identifikator računa (`receiptId`) i sa serijaliziranim podacima računa (`data`). Ovim pristupom korisnik iz pregleda povijesti jednim korakom otvara detalje, a potom po potrebi nastavlja na uređivanje istog zapisa bez gubitka konteksta.

```
const ReceiptModal: React.FC<ReceiptModalProps> = ({ receipt, onClose }) => {
  if (receipt === null) return;

  const router = useRouter();

  const handleReceiptEditPress = (receiptId: string) => {
    onClose();
    router.push({
      pathname: "/new-receipt/edit",
      params: {
        mode: 'existing',
        receiptId,
        data: JSON.stringify(receipt),
      },
    });
  };
}
```

Slika 6.33 Pregled odabranog računa

Uz uređivanje, u istom modalu postoji i mogućnost trajnog brisanja računa. Odabirom opcije *Delete* prikazuje se sustavna potvrda (*Alert*) kako bi se spriječilo slučajno brisanje te se tek nakon korisničke potvrde izvršava funkcija `handleReceiptDeletePress` (slika 6.34), koja asinkrono poziva servisnu funkciju `deleteReceipt(receiptId)` (slika 6.35). Servis šalje HTTP DELETE zahtjev na rutu `/api/receipts/deleteReceipt` s identifikatorom računa u `query` parametru, a nakon uspješnog odgovora modal se zatvara, prikazuje se obavijest o uspjehu i poziva `router.replace('/history')` radi osvježavanja liste.

```
const handleReceiptDeletePress = (receiptId: string) => {
  Alert.alert(
    "Delete Receipt",
    `Are you sure you want to delete this receipt?`,
    [
      { text: "Cancel", style: "cancel" },
      {
        text: "Delete",
        style: "destructive",
        onPress: async () => {
          try {
            await deleteReceipt(receiptId);
            onClose();
            router.replace('/history');
            Alert.alert("Deleted", `Receipt was deleted successfully.`);
          } catch (error: any) {
            Alert.alert("Error", error.message || "Failed to delete receipt.");
          }
        },
      },
    ],
  );
}
```

Slika 6.34 Funkcija za potvrdu i brisanje računa

```
export const deleteReceipt = async (selectedId: string) => {
  try {
    const response = await apiClient.delete('/api/receipts/deleteReceipt', {params: { selectedId }});
    return response.data;
  } catch (error) {
    throw new Error("Error deleting receipt.");
  }
}
```

Slika 6.35 Servisna funkcija za brisanje odabranog računa

6.4. Kategorije

6.4.1. Pregled svih kategorija

Zaslon *AllCategories* (*categories.tsx*) prikazuje korisnikove kategorije uz osnovne aggregate potrošnje (ukupna potrošnja, potrošnja u tekućem mjesecu, najčešća trgovina) te omogućuje sortiranje i brzo kreiranje novih kategorija. Komponenta dohvaća korisnikov profil i za svaku kategoriju računa statistiku. Rezultat se prikazuje kao lista kartica s mogućnošću ulaza u detaljni pregled kategorije.

Prvo se putem prethodno opisane servisne funkcije *getMyProfile* preuzima korisnikov profil s popisom kategorija, a zatim se statistike za svaku kategoriju računaju paralelnim pozivima *fetchCategoryStatsByName(category)* (slika 6.36) objedinjene kroz *Promise.all*. Dobiveni rezultati pohranjuju se u lokalno stanje *categoryStats*, dok se pogreške obrađuju kroz *try/catch/finally* blok. Kako bi prikaz ostao usklađen s poslužiteljem, učitavanje se pri povratku na ekran automatski ponavlja korištenjem *useFocusEffect* i *useCallback*,

```
const fetchCategoriesAndStats = async () => {
  try {
    const profile = await getMyProfile();
    const categories = profile.categories || [];
    const statsPromises = categories.map((category: string) => fetchCategoryStatsByName(category));
    const statsResults = await Promise.all(statsPromises);
    setCategoryStats(statsResults);
  } catch (error) {
    Alert.alert('Error', 'Failed to load categories and stats.');
  } finally {
    setLoading(false);
  }
};

useFocusEffect(useCallback(() => {fetchCategoriesAndStats();}, [ ]));
```

Slika 6.36 Dohvat kategorija i statistike

Odabirom akcije *Add New* otvara se modalni dijalog za kreiranje nove kategorije. Nadređena komponenta upravlja prikazom preko stanja *newCategoryModalVisible* te u renderu instancira *NewCategoryModal* sa svojstvima *visible*, *onClose* i *onSuccess*. Slika 6.37 prikazuje poziv modala za kreiranje nove kategorije iz nadređene komponente *AllCategories*.

```
<NewCategoryModal
  visible={newCategoryModalVisible}
  onClose={() => setNewCategoryModalVisible(false)}
  onSuccess={handleSuccessfullyCreatedCategory}
/>
```

Slika 6.37 Poziv modala za kreiranje nove kategorije

U samom modalu poziva se funkcija za stvaranje kategorije i provodi se osnovna validacija unosa, normalizacija naziva i poziv servisne funkcije *createCategory* (slika 6.38). Po uspjehu se aktivira funkcija *onSuccess* kako bi se osvježili lokalni popis i statistike te se prikaže opcija

navigacije na detaljni pregled novonastale kategorije (*/categoryOverview*). Greške se obrađuju kroz *try/catch* uz prikaz informativne poruke.

```
interface ModalProps {
  visible: boolean;
  onClose: () => void;
  onSuccess: (categoryName: string) => void;
}

const NewCategoryModal: React.FC<ModalProps> = ({ visible, onClose, onSuccess }) => {
  const router = useRouter();
  const [categoryName, set categoryName] = useState('');

  const handleCreate = async (nameInput: string) => {
    if (!nameInput) {
      Alert.alert("Error", "Category name is required.");
      return;
    }
    const name = nameInput.trim();
    try {
      await createCategory(name);
      onSuccess(name);
      Alert.alert('Success', 'Category created successfully.', [
        {text: 'Close', style: 'cancel'},
        {text: 'See Category', onPress: () => router.push({ pathname: '/categoryOverview', params: { name } }), style: 'default'},
      ],
      {cancelable: true});
    } catch (error: any) {
      Alert.alert('Error', error?.message || 'Failed to create category.');
    }
  }
}
```

Slika 6.38 Modal za stvaranje nove kategorije

Slika 6.39 prikazuje implementaciju servisne funkcije *createCategory* koja prema poslužitelju šalje POST zahtjev na rutu */api/users/newCategory* koristeći zajednički *apiClient* i u tijelu zahtjeva predaje se samo naziv kategorije

```
export const createCategory = async (name: string) => {
  try {
    const response = await apiClient.post('/api/users/newCategory', {name});
    return response.data;
  } catch (error: any) {
    const message = error?.response?.data?.message || "Error creating category.";
    throw new Error(message);
  }
}
```

Slika 6.39 Servisna funkcija *createCategory*

U komponenti *AllCategories* ažuriranje prikaza nakon uspješnog stvaranja nove kategorije provodi se funkcijom *handleSuccessfullyCreatedCategory* (slika 6.40). Funkcija prvo asinkrono dohvata agregirane podatke za upravo kreiranu kategoriju pozivom *fetchCategoryStatsByName*. Zatim se lokalno stanje ažurira kroz izradu nove kopije niza i dodavanje dobivenog objekta na postojeće podatke (*setCategoryStats*) te se promjena odmah vidi u korisničkom sučelju bez ponovnog učitavanja cijele liste. Na kraju se modal zatvara pozivom *setNewCategoryModalVisible* s boolean vrijednošću *false*, čime se proces dodavanja završava.

```

const handleSuccessfullyCreatedCategory = async (categoryName: string) => {
  const newCategoryStats = await fetchCategoryStatsByName(categoryName);
  setCategoryStats(prev => [...prev, newCategoryStats]);
  setNewCategoryModalVisible(false);
}

```

Slika 6.40 Ažuriranje liste nakon kreiranja nove kategorije

6.4.2. Pregled pojedine kategorije

Pogled *CategoryOverview* prikazuje detalje jedne odabrane kategorije, odnosno listu artikala koji joj pripadaju te sažetke potrošnje (ukupna potrošnja, potrošnja za ovaj mjesec, najčešća trgovina). Komponenta dohvata naziv kategorije iz rute sa funkcijom *useLocalSearchParams* i korisniku nudi sortiranje rezultata, označavanje kategorije kao omiljene te brisanje kategorije kao i prikaz računa kojem pripada svaki artikl unutar te kategorije.

U inicijalnom učitavanju komponenta pokreće asinkronu funkciju *fetchCategoryItems* (slika 6.41) koja dohvata articlje iz odabrane kategorije pozivom servisne funkcije *getCategoryItems* i sprema ih u lokalno stanje. Nakon toga izračunava statistiku odabrane kategorije pozivom funkcije *fetchCategoryStatsByName* te ih sprema u varijablu *categoryStats*. Slijedi dohvatanje profila korisnika (*getMyProfile*) kako bi se odredilo nalazi li se kategorija na popisu omiljenih kategorija te koliko je omiljenih kategorija trenutno postavljeno. Obrada pogrešaka provedena je kroz *try/catch* bloku uz korisničku notifikaciju putem *Alert*.

```

const fetchCategoryItems = useCallback(async () => {
  try {
    const data = await getCategoryItems(name);
    setCategoryItems(data);
    const dataStats = await fetchCategoryStatsByName(name);
    setCategoryStats([dataStats]);
    const userData = await getMyProfile();
    const isFavourite = userData.favouriteCategories.includes(name);
    setIsFavourited(isFavourite);
    const count = userData.favouriteCategories.length;
    setFavouritesCount(count);
  } catch (error) {
    Alert.alert("Error", "Failed to load profile data.");
  }
}, [ ]);

```

Slika 6.41 Dohvat podataka za pregled kategorije

Pozadinski pristup podacima ostvaruje se funkcijom *getCategoryItems* (slika 6.42), koja šalje GET zahtjev na REST rutu */api/receipts/getCategoryItems* s *query* parametrom koji nosi ime kategorije. Odgovor vraća normaliziran skup stavki (naziv, količina, jedinična i ukupna cijena) zajedno s metapodacima računa čime se klijentu omogućuje prikaz artikala u punom kontekstu bez dodatnih mrežnih poziva.

```

export const getCategoryItems = async (category: string) => {
  try {
    const response = await apiClient.get('/api/receipts/getCategoryItems', {params: {category: category}});
    return response.data;
  } catch (error) {
    throw new Error("Error loading receipts.");
  }
}

```

Slika 6.42 Servisna funkcija za dohvaćanje stavki kategorije

Pomoćna funkcija *fetchCategoryStatsByName* (slika 6.43) provodi agregacije na klijentskoj strani. Prvo iz skupa svih korisnikovih računa izdvaja one čije stavke sadrže zadanu kategoriju, zatim zbraja *totalPrice* kako bi dobila ukupnu potrošnju te dodatno izračunava potrošnju u tekućem mjesecu filtriranjem po datumu računa. Paralelno broji pojave naziva trgovina kako bi identificirala najčešću trgovinu. Rezultat se vraća kao strukturirani objekt *CategoryStats* (*name*, *totalSpent*, *thisMonthsSpendings*, *mostPopularStore*), koji se koristi za prikaz sažetka u zaglavlju ekrana.

```

export const fetchCategoryStatsByName = async(categoryName:string):Promise<CategoryStats> => {
  const receipts = await getMyReceipts();
  const now = new Date();
  const currentMonth = now.getMonth();
  const currentYear = now.getFullYear();
  const filteredReceipts = receipts.filter((receipt: Receipt) => receipt.items.some(item => item.categories?.includes(categoryName)));
  let totalSpent = 0;
  let thisMonthsSpendings = 0;
  const storeCounts: Record<string, number> = {};

  filteredReceipts.forEach((receipt: Receipt) => {
    const matchingItems = receipt.items.filter(item => item.categories?.includes(categoryName));
    const matchingTotal = matchingItems.reduce((sum, item) => sum + (item.totalPrice || 0), 0);
    totalSpent += matchingTotal;
    const receiptDate = new Date(receipt.date);
    if (receiptDate.getMonth() === currentMonth && receiptDate.getFullYear() === currentYear) {
      thisMonthsSpendings += matchingTotal;
    }
    const store = receipt.store || 'Unknown';
    storeCounts[store] = (storeCounts[store] || 0) + 1;
  });

  let mostPopularStore: string | null = null;
  let maxCount = 0;
  for (const [store, count] of Object.entries(storeCounts)) {
    if (count > maxCount) {
      mostPopularStore = store;
      maxCount = count;
    }
  }
  return { name: categoryName, totalSpent, mostPopularStore, thisMonthsSpendings, };
};

```

Slika 6.43 Izračun sažetaka za kategoriju

U komponenti *CategoryOverview* brisanje je dostupno u zaglavlju sažetka za trenutno odabranu kategoriju. Funkcija *handleDelete* (slika 6.44) otvara potvrđni dijalog s opcijama *Cancel* za odustanak i *Delete* za brisanje. Ako korisnik potvrdi, asinkrono se poziva servisna funkcija *deleteCategory*, a uspjeh se potvrđuje uspješnom porukom i navigacijom na listu kategorija, dok se u *catch* grani prikazuje jasno objašnjenje greške.

```

const handleDelete = async () => {
  Alert.alert(
    "Delete Category",
    `Are you sure you want to delete the category "${name}"? This will remove it from all receipts.`,
    [
      { text: "Cancel", style: "cancel" },
      {
        text: "Delete",
        style: "destructive",
        onPress: async () => {
          try {
            await deleteCategory(name);
            Alert.alert("Deleted", `Category "${name}" was deleted successfully.`);
          } catch (error: any) {
            Alert.alert("Error", error.message || "Failed to delete category.");
          }
        },
      },
    ],
  );
};

```

Slika 6.44 Brisanje kategorije

Funkcija *deleteCategory* (slika 6.45) implementira servisni sloj za uklanjanje kategorije. Prema puti */api/users/deleteCategory* šalje se DELETE zahtjev s nazivom kategorije u *query* parametru. Metoda vraća *response.data*, dok u slučaju pogreške propagira poruku sa servera kako bi prikaz bio konzistentan na klijentu.

```

export const deleteCategory = async (name: string) => {
  try {
    const response = await apiClient.delete('/api/users/deleteCategory', {params: { name }});
    return response.data;
  } catch (error: any) {
    const message = error?.response?.data?.message || "Error deleting category.";
    throw new Error(message);
  }
}

```

Slika 6.45 Pozadinski poziv za brisanje kategorije

6.4.3. Omiljene kategorije

U prikazu *CategoryOverview* korisnik može označiti kategoriju kao omiljenu ili ukloniti oznaku. Interakcija započinje na razini sučelja tako što asinkrona funkcija *handleAddToFavourites* (slika 6.46) prema odluci korisnika izračuna željeno stanje (dodavanje ili uklanjanje iz popisa omiljenih kategorija) i izvrši poziv prema poslužitelju. Nakon uspjeha, iz vraćenog polja *favouriteCategories* derivira novi status te ga odmah upisuje u lokalno stanje preko *setIsFavourited*. U slučaju pogreške, funkcija razlikuje dosegnuti limit omiljenih kategorija (*favourites limit reached*) od ostalih scenarija te prikazuje odgovarajuću obavijest.

```

const handleAddToFavourites = async() => {
  try {
    const response = await addCategoryToFavourites( name, !isFavourited);
    const isNowFavourited = response.favouriteCategories.includes(name);
    setIsFavourited(isNowFavourited);
  } catch (error: any) {
    const message = error?.message || "Failed to update favourites.";
    if (message.toLowerCase().includes("favourites limit reached")) {
      Alert.alert("Limit Reached", "Favourites are full. You can't add more.");
    } else {
      Alert.alert("Error", message);
    }
  }
};

```

Slika 6.46 Dodavanje kategorije u popis omiljenih

Servisna funkcija `addCategoryToFavourites` (slika 6.47) šalje POST zahtjev prema ruti `/api/users/addCategoryToFavourites` s tijelom u kojem su uključeni ime kategorije i zastavica `add` koja označava dodaje li se kategorija u omiljene ili uklanja. Poziv vraća ažurirani popis `favoriteCategories`, čime se osigurava jedinstveni izvor istine na klijentu.

```
export const addCategoryToFavourites = async (categoryName: string, add: boolean) => {
  try {
    const response = await apiClient.post('/api/users/addCategoryToFavourites', { categoryName, add });
    return response.data;
  } catch (error: any) {
    const message = error?.response?.data?.message || "Error adding favourite category.";
    throw new Error(message);
  }
};
```

Slika 6.47 Servisna funkcija dodavanje kategorije u omiljene

Za razliku od zaslona `CategoryOverview` koji obrađuje jednu kategoriju, početni zaslon `Home` prikazuje sažet pregled svih korisnikovih omiljenih kategorija. Nakon učitavanja korisničkog profila, nad popisom omiljenih kategorija provodi se paralelno dohvaćanje sažetaka za svaku omiljenu kategoriju pozivom `fetchCategoryStatsByName` uz uporabu `Promise.all`. Dobiveni niz rezultata reducira se preslikavanjem u `{[category]:stats}`, koje se pohranjuje u lokalno spremište `favStats`. Takva struktura omogućuje dohvati pripadajuće statistike u konstantnom vremenu pri renderiranju sučelja (slika 6.48).

```
if (profile.favoriteCategories?.length > 0) {
  const statsArray = await Promise.all(
    profile.favoriteCategories.map(async (category: string) => {
      const stats = await fetchCategoryStatsByName(category);
      return { category, stats };
    })
  );
  const statsObject = statsArray.reduce((acc, { category, stats }) => {
    acc[category] = stats;
    return acc;
  }, {});
  setFavStats(statsObject);
}
```

Slika 6.48 Prikaz omiljenih kategorija

6.5. Analitika

6.5.1. Mjesečni sažetak potrošnje

Na početnom zaslonu mjesečni sažetak nastaje lokalno pozivom funkcije `analyzeReceiptsThisMonth` (slika 6.49), čiji se rezultat sprema u varijablu `thisMonthData` i potom prikazuje u tri informacijske kartice u horizontalnom `ScrollViewu`. Funkcija najprije određuje granice tekućeg mjeseca (`startOfMonth`, `endOfMonth`), filtrira račune u tom intervalu te akumulira ukupnu potrošnju (`totalSpent`) i broj računa (`receiptCount`). Paralelno pronalazi najskuplji artikl (usporedbom `item.totalPrice` i pamćenjem datuma računa) te zbraja iznose po kategorijama koristeći prvu oznaku artikla. Nakon obrade podataka, određuje se kategorija s

najvećom potrošnjom i formira se objekt s podacima *totalSpentThisMonth* (ukupna potrošnja tekućeg mjeseca), *receiptCountThisMonth* (broj unesenih računa tekućeg mjeseca), *mostExpensiveItemThisMonth* (najskuplji proizvod tekućeg mjeseca), *mostSpendingCategoryThisMonth* (kategorija s najvećom potrošnjom tekućeg mjeseca), *mostSpendingCategoryAmountThisMonth* (iznos potrošnje prethodno navedene kategorije) i *currentMonthName* (naziv tekućeg mjeseca).

```
export function analyzeReceiptsThisMonth(receipts: Receipt[]): AnalysisResult {
  const now = new Date();
  const startOfMonth = new Date(now.getFullYear(), now.getMonth(), 1);
  const endOfMonth = new Date(now.getFullYear(), now.getMonth() + 1, 0, 23, 59, 59);
  let totalSpent = 0;
  let receiptCount = 0;
  let mostExpensiveItem: { item: Receipt['items'][0]; date: Date } | null = null;
  const categoryTotals: Record<string, number> = {};

  receipts.forEach((receipt) => {
    const receiptDate = new Date(receipt.date);
    if (receiptDate >= startOfMonth && receiptDate <= endOfMonth) {
      receiptCount++;
      totalSpent += receipt.totalAmount || 0;
      receipt.items.forEach((item) => {
        if (!mostExpensiveItem || item.totalPrice > mostExpensiveItem.item.totalPrice) {
          mostExpensiveItem = { item, date: receiptDate };
        }
        const categories = item.categories && item.categories.length > 0 ? item.categories : ['Other'];
        const mainCategory = categories[0];
        categoryTotals[mainCategory] = (categoryTotals[mainCategory] || 0) + item.totalPrice;
      });
    }
  });

  let maxCategory: string | null = null;
  let maxCategoryAmount = 0;
  for (const [cat, amount] of Object.entries(categoryTotals)) {
    if (amount > maxCategoryAmount) {
      maxCategoryAmount = amount;
      maxCategory = cat;
    }
  }
}
```

Slika 6.49 Funkcija za analizu potrošnje tekućeg mjeseca

Zatim se iz tog objekta pune kartice *Total Spent*, *Biggest Purchase* i *Top Spending Category*, čime se korisniku pruža sažet, vizualno istaknut pregled ključnih mjesečnih pokazatelja bez dodatnih mrežnih poziva.

6.5.2. Trend potrošnje u posljednja četiri mjeseca

U bloku *Spending in Last 4 Months* agregacija i prikaz podataka odvijaju se u dva koraka. Prvo pomoćna funkcija *getMonthlySpendingLast4Months* (slika 6.50) u jednoj prolaznoj petlji nad svim računima računa mjesečnu potrošnju za posljednja četiri mjeseca. To ostvaruje tako što prolazi kroz sve račune, za svaki računa razliku u mjesecima u odnosu na „dan” (*diffMonth*) te, ako datum računa spada u prethodna četiri mjeseca, iznos računa akumulira u prethodno alocirani niz *monthlyTotals*. Potom generira rezultat s točnim redoslijedom mjeseci (od najstarijeg prema najnovijem), uz lokalizirane kratke nazive mjeseca i iznose zaokružene na dvije decimale.

```

function getMonthlySpendingLast4Months(receipts: Receipt[]): MonthlySpending[] {
  const now = new Date();
  const monthlyTotals = Array(4).fill(0);
  receipts.forEach((receipt) => {
    const date = new Date(receipt.date);
    const diffMonth = (now.getFullYear() - date.getFullYear()) * 12 + (now.getMonth() - date.getMonth());
    if (diffMonth >= 0 && diffMonth < 4) {
      monthlyTotals[3 - diffMonth] += receipt.totalAmount;
    }
  });
  const result: MonthlySpending[] = [];
  for (let i = 3; i >= 0; i--) {
    const d = new Date(now.getFullYear(), now.getMonth() - i, 1);
    const label = d.toLocaleString('default', { month: 'short' });
    result.push({
      month: label,
      total: parseFloat(monthlyTotals[3 - i].toFixed(2)),
    });
  }
  return result;
}

```

Slika 6.50 Računanje potrošnje za zadnja četiri mjeseca

Dobiveni niz se predaje prezentacijskoj komponenti *MonthsBarChart* (slika 6.51), koja priprema podatke za *react-native-chart-kit BarChart*, dodaje osi s oznakom valute (prefiks „€“), te obrađuje rubni slučaj bez podataka prikazom poruke umjesto grafa. Radi boljeg korisničkog doživljaja, ulazni prikaz grafa animira se paralelnom promjenom neprozirnosti i vertikalne skale (*Animated.parallel*), a širina se prilagođava širini zaslona.

```

const MonthsBarChart: React.FC<Props> = ({ receipts }) => {
  const spending = getMonthlySpendingLast4Months(receipts);
  const totalSum = spending.reduce((acc, m) => acc + m.total, 0);
  const fadeAnim = useRef(new Animated.Value(0)).current;
  const scaleAnim = useRef(new Animated.Value(0.8)).current;

  useEffect(() => {
    Animated.parallel([
      Animated.timing(fadeAnim, { toValue: 1, duration: 800, useNativeDriver: true, }),
      Animated.timing(scaleAnim, { toValue: 1, duration: 800, useNativeDriver: true, }),
    ]).start();
  }, [fadeAnim, scaleAnim]);

  if (totalSum === 0) {
    return <Text className="text-gray-500 mt-10 text-center">No data available for the last 4 months.</Text>;
  }

  const chartData = {
    labels: spending.map(s => s.month),
    datasets: [ { data: spending.map(s => s.total), }, ],
  };
}

```

Slika 6.51 Graf za prikaz potrošnje u posljednja četiri mjeseca

Na zaslonu *Analytics* komponenta *MonthlySpendingCard* (slika 6.52) zadužena je za brzu mjesečnu agregaciju i usporedbu potrošnje. Prima niz računa te lokalno formira oznake tekućeg i prethodnog mjeseca u formatu YYYY-MM. Slijedi pomoćna funkcija *getMonthlySpending* koja jednom prolazi kroz sve račune, iz datuma svakog računa formira ključ mjeseca te, ako je jednak ciljanom mjesecu, pribraja *totalAmount* u akumulator. Na kraju se funkcija poziva dvaput, za *thisMonth* i *lastMonth*, čime se dobivaju agregirane vrijednosti potrošnje za prikaz usporedbe „ovaj mjesec“ naspram „prošli mjesec“.

```

const MonthlySpendingCard = ({ receipts }: Props) => {
  const now = new Date();
  const thisMonth = `${now.getFullYear()}-${String(now.getMonth()) + 1}.padStart(2, '0')`;
  const lastMonthDate = new Date(now.getFullYear(), now.getMonth() - 1, 1);
  const lastMonth = `${lastMonthDate.getFullYear()}-${String(lastMonthDate.getMonth() + 1).padStart(2, '0')}`;

  const getMonthlySpending = (receipts: Receipt[], targetMonth: string): number => {
    let total = 0;
    receipts.forEach((receipt) => {
      const date = new Date(receipt.date);
      const receiptMonth = `${date.getFullYear()}-${String(date.getMonth()) + 1}.padStart(2, '0')`;
      if (receiptMonth === targetMonth) {
        total += receipt.totalAmount;
      }
    });
    return total;
  };

  const thisMonthSpendings = getMonthlySpending(receipts, thisMonth);
  const lastMonthSpendings = getMonthlySpending(receipts, lastMonth);
}

```

Slika 6.52 Funkcija za računanje potrošnje u prethodnom i tekućem mjesecu

Komponenta *AverageReceiptValue* (slika 6.53) prikazuje prosječnu vrijednost računa i uspoređuje je s maksimalnom vrijednošću u odabranom razdoblju. Na ulazu prima niz računa, a pomoću *useMemo* generira listu posljednjih šest mjeseci te filtrira račune prema odabranom mjesecu (ili *All* za sve račune). Nad filtriranim skupom u jednoj agregaciji računa sumu i maksimum, nakon čega računa napredak (*progressPercent=average/max*) koji se vizualizira pomoću trake napretka iz biblioteke *react-native-progress*. Odabir mjeseca rješava se modalnim dijalogom, a širina trake prilagođava se širini ekrana preko *Dimensions*. U slučaju da nema podataka za selekciju, metričke vrijednosti padaju na nulu, čime se izbjegavaju greške i održava konzistentan prikaz.

```

const AverageReceiptValue: React.FC<AverageReceiptValueProps> = ({ receipts }) => [
  const [selectedValue, setSelectedValue] = useState<string | number>("all");
  const [modalVisible, setModalVisible] = useState(false);
  const today = new Date();
  const currentMonth = today.getMonth();
  const currentYear = today.getFullYear();
  const lastSixMonths = useMemo(() => {
    const months = [];
    for (let i = 0; i < 6; i++) {
      let monthIndex = currentMonth - i;
      if (monthIndex < 0) monthIndex += 12;
      months.push({ label: fullMonths[monthIndex], value: monthIndex });
    }
    return months;
  }, [currentMonth]);
  const filteredReceipts = useMemo(() => {
    if (selectedValue === "all") return receipts;
    return receipts.filter((r) => new Date(r.date).getMonth() === selectedValue);
  }, [receipts, selectedValue]);
  const { average, max } = useMemo(() => {
    if (filteredReceipts.length === 0) return { average: 0, max: 0 };
    let sum = 0;
    let maxValue = 0;
    filteredReceipts.forEach((r) => {
      sum += r.totalAmount;
      if (r.totalAmount > maxValue) maxValue = r.totalAmount;
    });
    return { average: sum / filteredReceipts.length, max: maxValue };
  }, [filteredReceipts]);
]

```

Slika 6.53 Funkcija za računanje prosječne vrijednosti računa

U sklopu zaslona *Analytics* implementiran je prilagođeni tortni grafikon (eng. *pie chart*) za raspodjelu potrošnje po kategorijama. Prvo se definiraju pomoćne geometrijske funkcije *polarToCartesian* i *describeArc* (slika 6.54) koje, na temelju središta i radijusa, pretvaraju polarne koordinate u kartezijeve te generiraju SVG *path* za kružni isječak. Time se omogućuje

crtanje segmenata kolača korištenjem *react-native-svg* bez oslanjanja na dodatne biblioteke za grafove.

```

function polarToCartesian(cx: number, cy: number, r: number, angleInDegrees: number) {
  const angleInRadians = ((angleInDegrees - 90) * Math.PI) / 180.0;
  return {
    x: cx + r * Math.cos(angleInRadians),
    y: cy + r * Math.sin(angleInRadians),
  };
}

function describeArc(cx: number, cy: number, r: number, startAngle: number, endAngle: number) {
  const start = polarToCartesian(cx, cy, r, endAngle);
  const end = polarToCartesian(cx, cy, r, startAngle);
  const largeArcFlag = endAngle - startAngle <= 180 ? "0" : "1";

  return [
    `M ${cx} ${cy}`,
    `L ${start.x} ${start.y}`,
    `A ${r} ${r} 0 ${largeArcFlag} 0 ${end.x} ${end.y}`,
    "Z",
  ].join(" ");
}

```

Slika 6.54 Funkcije za crtanje „pie“ grafa

Sama komponenta *CategorySpendingPieChart* (slika 6.55) najprije agregira potrošnju po kategorijama nad svim stavkama računa. Ako je stavka označena s više kategorija, njezin se iznos proporcionalno dijeli na sve pripadajuće kategorije, čime se izbjegava dvostruko brojanje. Rezultat se sortira i uzima se prvih 5, dok se ostatak spaja u grupu *Other*. Za svaki segment računa se udio u ukupnoj potrošnji i mapira se na boju iz palete. Prilikom prikaza koristi se Animated API kako bi se pri ulasku na zaslon izvršila animacija svakog isječka (akumulativni kut se povećava do ciljne vrijednosti). Lijevo se crta SVG kolač segment po segment, dok se desno prikazuje legenda s nazivom kategorije i iznosom za brzu usporedbu.

```

const CategorySpendingPieChart: React.FC<ValueProps> = ({ receipts }) => {
  const size = 180;
  const radius = size / 2;
  const [animatedProgress, setAnimatedProgress] = useState(0);
  const isFocused = useIsFocused();
  const categoryMap: Record<string, number> = {};

  receipts.forEach((receipt: { items: any[]; }) => {
    receipt.items.forEach((item) => {
      const categories = item.categories?.length ? item.categories : ["Other"];
      const splitAmount = item.totalPrice / categories.length;
      categories.forEach((category: string | number) => {
        categoryMap[category] = (categoryMap[category] || 0) + splitAmount;
      });
    });
  });

  const sortedEntries = Object.entries(categoryMap).sort((a, b) => b[1] - a[1]);
  const top5 = sortedEntries.slice(0, 5);
  const others = sortedEntries.slice(5);
  const othersTotal = others.reduce((sum, [_, amount]) => sum + amount, 0);
  const pieColors = ["#5F66F5", "#BCBEFB", "#e3e4ff", "#e67eba", "#111F9A", "#FBCFE8"];
  const chartData = [...top5, ["Other", othersTotal]]
    .filter(([_, amount]) => Number(amount) > 0)
    .map(([category, amount], i) => ({
      name: category,
      value: parseFloat(Number(amount).toFixed(2)),
      color: pieColors[i % pieColors.length],
    }));
  const total = chartData.reduce((sum, d) => sum + d.value, 0);
}

```

Slika 6.55 Funkcija *CategorySpendingPieChart*

Za prikaz udjela potrošnje po trgovinama koristi se komponenta *StoreSpendingChart* (slika 6.56). Ulagani niz računa se memorizira i transformira u mapu, nakon čega se parovi sortiraju

silazno te izdvajaju *Top 4* trgovine, a preostali zbroj se grupira u stavku *Other*. Svakom segmentu dodjeljuje se naziv u formatu „€ (ime trgovine)“ i deterministička boja iz unaprijed definirane palete, što osigurava dosljedan izgled pri svakom iscrtavanju. Za crtanje se koristi *PieChart* iz paketa *react-native-chart-kit* s responzivnom širinom dobivenom iz *Dimensions*. U slučaju praznog skupa podataka prikazuje se informativna poruka.

```
const StoreSpendingChart: React.FC<StoreSpendingChartProps> = ({ receipts }) => {
  const pieColors = ["#5F66F5", "#BCBEB8", "#e3e4ff", "#e67eba", "#111F9A", "#FBCFE8"];
  let colorIndex = 0;

  const storeData = useMemo(() => {
    const totals: Record<string, number> = {};
    receipts.forEach(({ store, totalAmount }) => {
      if (!store) return;
      totals[store] = (totals[store] || 0) + totalAmount;
    });
    const sortedStores = Object.entries(totals).sort(([a], [b]) => b - a);
    const topStores = sortedStores.slice(0, 4);
    const otherSum = sortedStores.slice(4).reduce(([sum, ...total]) => sum + total, 0);
    const data = topStores.map(([store, total]) => ({
      name: `€ ${store}`,
      amount: Number(total.toFixed(2)),
      color: getRandomColor(),
      legendFontColor: "#333",
      legendFontSize: 12,
    }));
    if (otherSum > 0) {
      data.push({
        name: "€ Other",
        amount: Number(otherSum.toFixed(2)),
        color: getRandomColor(),
        legendFontColor: "#333",
        legendFontSize: 12,
      });
    }
    return data;
  }, [receipts]);
```

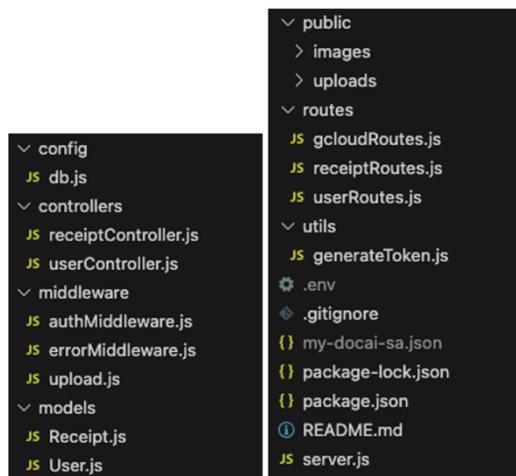
Slika 6.56 Prikaz udjela potrošnje po trgovinama

7. IMPLEMENTACIJA POSLUŽITELJSKOG SLOJA

U ovom poglavlju prikazuje se realizacija poslužiteljske strane sustava, oblikovana kao REST usluga na Node.js/Express platformi. Opisuje se struktura modula i ruta, modeliranje podataka te persistencija u bazi, zajedno s validacijom ulaza, obradom pogrešaka i *middleware* mehanizmima. Posebna pozornost posvećena je autentifikaciji i autorizaciji temeljnoj na JWT tokenima, kao i integraciji vanjskog servisa Google Document AI za ekstrakciju podataka iz računa.

7.1. Struktura projekta

Struktura poslužiteljskog dijela organizirana je slojevito, s jasnim razdvajanjem konfiguracije, domenskih modela, poslovne logike, ruta i međuslojeva (slika 7.1). Ulazna točka sustava je datoteka *server.js* koja inicijalizira Express aplikaciju, učitava globalne među slojeve te registrira rute.



Slika 7.1 Struktura poslužiteljskog sloja

Povezivanje s bazom centralizirano je u *config/db.js*, čime se pojednostavljuje inicijalizacija i nadzor nad eventualnim pogreškama pri spajanju (slika 7.2).

```
const mongoose = require("mongoose");

const connectDB = async () => {
  try {
    const conn = await mongoose.connect(process.env.MONGO_URI);
    console.log(`MongoDB connected: ${conn.connection.host}`);
  } catch (error) {
    console.error("MongoDB Connection Failed:", error);
    process.exit(1);
  }
}

module.exports = connectDB;
```

Slika 7.2 Povezivanje s bazom podataka

Domenski sloj definiran je u mapi *models*, gdje se nalaze sheme i modeli za korisnike i račune. Iznad njega smješten je sloj kontrolera (*controllers*) koji inkapsulira poslovnu logiku, od operacija nad korisnicima i računima do integracije s uslugom Google Document AI za ekstrakciju podataka s računa. Rute su izdvojene u zasebnu mapu (*routes*) i predstavljaju sloj koji mapira HTTP krajnje točke na odgovarajuće metode kontrolera, čime se osigurava pregledan i proširiv REST sučelni sloj.

Posrednički slojevi (*middleware*) obavljaju transverzalne funkcije kao što su autentifikacija putem JWT-a, obrada *multipart/form-data* prilikom učitavanja slika te centralizirano hvatanje i standardizacija pogrešaka. Statički resursi i privitci pohranjuju se unutar *public* hijerarhije, dok se pomoćne funkcije (poput izdavanja tokena) nalaze u *utils*. Konfiguracijskim parametrima, poput tajne za potpis tokena, URL baze i vjerodajnice servisnog računa, upravlja se kroz *.env* i vanjske JSON vjerodajnice, pri čemu je predviđeno razdvajanje razvojnih i produkcijskih postavki.

Ovakva podjela odgovornosti omogućuje visoku koheziju unutar slojeva i slabu spregu među njima, olakšava jedinično testiranje (eng. *unit testing*) i smanjuje trošak održavanja. Istodobno, arhitektura ostaje dovoljno fleksibilna za buduća proširenja bez narušavanja cjelokupne strukture projekta.

7.2. Modeli podataka

Sustav koristi dokumentno orijentiranu bazu MongoDB, pri čemu se pristup i validacija shema ostvaruju putem Mongoosea. Definirana su dva temeljna modela: *User* i *Receipt*, s ugniježđenom podshemom *Item*.

User shema (slika 7.3) čuva osnovne podatke o korisniku: korisničko ime (*username*), email, lozinku (*password*), ime (*name*), prezime (*surname*), opis (*bio*), lokaciju (*location*) i sliku (*photo*). Polje *categories* inicijalno se popunjava zadanim skupom kategorija, dok *favouriteCategories* predstavlja popis omiljenih kategorija. Ograničenje „najviše četiri omiljene kategorije“ provodi se na razini aplikacijske logike (kontroler), čime se zadržava fleksibilnost sheme.

Model sadrži metodu *matchPassword* za usporedbu unesene lozinke s pohranjenim *hashom* te *pre-save hook* koji pomoću *bcrypt* funkcije hešira lozinku prije njezina spremanja. Stvorna lozinka nigdje se ne sprema u izvornom obliku.

```

const userSchema = mongoose.Schema({
  username: { type: String, required: true, unique: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
  name: { type: String, required: false },
  surname: { type: String, required: false },
  bio: { type: String, required: false },
  location: { type: String, required: false },
  photo: { type: String, default: '/public/images/profile-picture.png' },
  categories: {
    type: [String],
    default: [
      "Groceries",
      "Clothes",
      "Electronics",
      "Health & Beauty",
      "Restaurants",
      "Pharmacy",
      "Toys & Games",
      "Fitness",
      "Other"
    ],
  },
  favouriteCategories: { type: [String], default: [] },
}, {
  timestamps: true,
});

userSchema.methods.matchPassword = async function (enteredPassword) {
  return await bcrypt.compare(enteredPassword, this.password);
};

userSchema.pre('save', async function (next) {
  if (!this.isModified('password')) {
    next();
  }
  const salt = await bcrypt.genSalt(10);
  this.password = await bcrypt.hash(this.password, salt);
});

```

Slika 7.3 Model User

Item shema (slika 7.4) opisuje jednu stavku računa i modeliran je kao ugniježđeni dokument unutar računa. Sadrži nužna polja ime (*name*), količina (*quantity*), jedinična cijena (*unitPrice*), ukupna cijena (*totalPrice*) te listu kategorija (*categories*) koja je opcionalna. Ugradnja stavki izravno u *Receipt* omogućuje atomske izmjene jednog računa i jednostavnije agregacije po stavkama.

```

const itemSchema = new mongoose.Schema({
  name: { type: String, required: true },
  quantity: { type: Number, required: true },
  unitPrice: { type: Number, required: true },
  totalPrice: { type: Number, required: true },
  categories: [{ type: String, required: false }],
});

```

Slika 7.4 Model stavki računa

Receipt shema (slika 7.5) predstavlja pojedini račun te referencira vlasnika preko *user* polja. Ostala polja uključuju: datum (*date*), stavke (*items*), ukupna cijena (*totalAmount*), bilješka (*note*), način plaćanja (*paymentMethod*), tagovi (*tags*) i trgovina (*store*). Uključena je i opcija *timestamps: true*, čime se automatski održavaju polja *createdAt* i *updatedAt*. Prije spremanja dokumenta aktivira se *pre-save middleware* koji izračunava ukupni iznos računa zbrajanjem ukupnih cijena svih stavki, čime se sprječava nedosljednost između suma i pojedinačnih vrijednosti.

```

const receiptSchema = new mongoose.Schema({
  user: { type: mongoose.Schema.Types.ObjectId, ref: 'User', required: true },
  date: { type: Date, default: Date.now },
  items: [itemSchema],
  totalAmount: { type: Number, required: false },
  note: { type: String, required: false },
  paymentMethod: {
    type: String,
    enum: ['Cash', 'Card', 'Mobile', 'Other'],
    required: false,
  },
  tags: [{ type: String, required: false }],
  store: { type: String, required: false },
}, { timestamps: true });

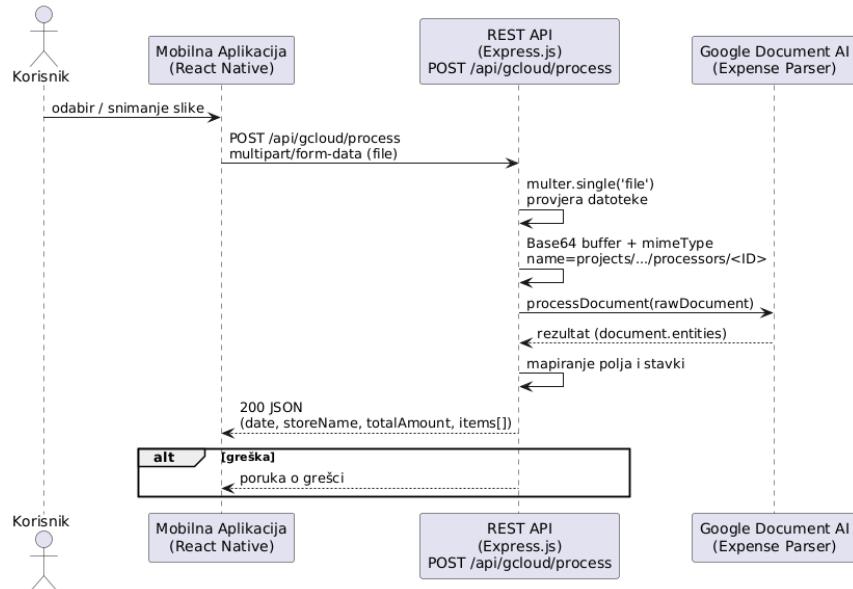
receiptSchema.pre('save', function (next) {
  this.totalAmount = this.items.reduce((sum, item) => sum + item.totalPrice, 0);
  next();
});

```

Slika 7.5 Model računa

7.3. Google Cloud Document AI

U okviru sustava koristi se Google Cloud Document AI za izdvajanje strukturiranih podataka iz slike računa. Poziv prema Document AI-ju odvija se tako da klijent šalje fotografiju poslužitelju na ruti `/api/gcloud/process` u formatu `multipart/form-data`, a on uz pomoć `multer middlewarea` preuzima datoteku, enkodira sadržaj te prosljeđuje dokument Document AI-u koji obrađuje podatke te generira strukturirani izlaz u JSON formatu. Poslužitelj zatim provodi filtriranje, normalizira polja i mapira ih u dogovoren model. Tako oblikovani podaci vraćaju se klijentskom sloju kao standardizirani JSON odgovor spremjan za daljnju validaciju, uređivanje i pohranu. Slika 7.6 prikazuje UML dijagram koji vizualizira opisani proces obrade računa.



Slika 7.6 OCR obrada računa

Konfiguracija za obradu računa preko Google Cloud Document AI definirana je kroz varijable okruženja i poslužiteljske postavke. U `.env` dokumentu su navedene varijable

GCLOUD_PROJECT_ID, *GCLOUD_REGION*, *GCLOUD_PROCESSOR_ID* i *GOOGLE_APPLICATION_CREDENTIALS* (apsolutna putanja do JSON ključa servisnog računa), a učitavaju se pri pokretanju poslužitelja putem *dotenv* modula.

Na platformi Google Cloud najprije je kreiran projekt, a u *APIs & Services* omogućen je Document AI API. U Document AI kreiran je procesor s definiranim entitetima relevantnim za račune (*date*, *totalAmount*, skup stavki *itemName*, *itemQuantity*, *itemUnitPrice* i *itemTotalPrice*, te neobvezni entiteti *storeName*, *location* i *paymentMethod*). Procesor je zatim treniran i testiran na uzorku računa, nakon čega je izrađen *service account* s ulogom Document AI API *User* i generiranim JSON ključem koji poslužiteljska aplikacija učitava putem varijable *GOOGLE_APPLICATION_CREDENTIALS*.

Ruta POST */process* uz pomoć *multer middlewarea* preuzima sliku računa, provjerava prisutnost datoteke te konstruira identifikator procesora iz varijabli okruženja. Datoteka se enkodira u *Base64* i šalje metodi *processDocument* klijenta *DocumentProcessorServiceClient*. Nakon obrade dohvaća se polje *result.document.entities*, koje se u sljedećim koracima prevodi u standardizirani JSON model, dok se pogreške bilježe i vraća se odgovarajuća HTTP poruka. (slika 7.7)

```
router.post('/process', upload.single('file'), async (req, res) => {
  const file = req.file;
  if (!file) return res.status(400).send('No file uploaded.');
  const name = `projects/${process.env.GCLOUD_PROJECT_ID}/locations/${process.env.GCLOUD_REGION}/processors/${process.env.GCLOUD_PROCESSOR_ID}`;

  try {
    const [result] = await client.processDocument({ name, rawDocument: {
      content: file.buffer.toString('base64'),
      mimeType: file.mimetype,
    },});
    const entities = result.document.entities;
```

Slika 7.7 Poslužiteljska ruta */process* i poziv Document AI procesora

Slika 7.8 prikazuje petlju koja prolazi kroz sve prepoznate entitete te se, uz prag pouzdanosti, mapiraju ključni atributi računa: datum (*date*), lokacija (*location*), način plaćanja (*paymentMethod*), naziv trgovine (*storeName*) i ukupni iznos (*totalAmount*).

```

for (const entity of entities) {
  const { type, mentionText, confidence } = entity;
  if (confidence < 0.7) continue;
  switch (type) {
    case 'date':
      responseData.date = mentionText;
      break;
    case 'location':
      responseData.location = mentionText;
      break;
    case 'paymentMethod':
      responseData.paymentMethod = mentionText;
      break;
    case 'storeName':
      responseData.storeName = mentionText;
      break;
    case 'totalAmount':
      responseData.totalAmount = mentionText;
      break;
    case 'item':
      break;
  }
}

```

Slika 7.8 Filtriranje entiteta i mapiranje globalnih atributa računa

Entitet tipa *item* obrađuje se zasebno u mehanizmu agregacije stavki. Definira se mapa polja *itemFieldMap* te dinamički agregiraju entiteti koje je vratio Document AI (*itemName*, *itemQuantity*, *itemUnitPrice*, *itemTotalPrice*) u jedinstvene objekte stavki. Za svako detektirano polje pronalazi se, ili stvara, trenutna stavka bez popunjenoog odgovarajućeg atributa te se vrijednost upisuje u taj objekt. Na taj se način iz niza nevezanih entiteta konstruira konzistentan niz *responseData.items* bez duplikata i s ispravno sparenim vrijednostima po retku računa (slika 7.9).

```

const itemFieldMap = {
  itemName: 'itemName',
  itemTotalPrice: 'itemTotalPrice',
  itemQuantity: 'itemQuantity',
  itemUnitPrice: 'itemUnitPrice',
};

if (type in itemFieldMap) {
  let currentItem = responseData.items.find(item => !item[type]);
  if (!currentItem || currentItem[type]) {
    currentItem = {};
    responseData.items.push(currentItem);
  }
  currentItem[itemFieldMap[type]] = mentionText;
}

```

Slika 7.9 Mapiranje entiteta stavki u strukturu items

7.4. Posrednički sloj

U poslužiteljskom dijelu sustava ključne nedovršene poslove obavlja posrednički sloj (*middleware*), tj. modularne funkcije koje se ulančavaju u Express *pipeline* i centralizirano rješavaju autentifikaciju, rukovanje pogreškama i prijenos datoteka. Time se pojednostavljuju kontroleri, smanjuje dupliciranje koda i osigurava dosljedno ponašanje API-ja.

Posrednički sloj *protect* očekuje JSON Web Token u zaglavljku *Authorization: Bearer <token>*. Token se verificira tajnom vrijednošću (*JWT_SECRET*), a u slučaju valjanosti u *req.user* se učitava korisnik iz baze (bez polja lozinke). Ako token nedostaje ili je neispravan, zahtjev se završava s 401 i porukom *Not authorized*. Ovaj se sloj postavlja ispred svih privatnih ruta čime se osigurava da kontroleri rade nad već provjerjenim identitetom (slika 7.10).

```
const protect = async (req, res, next) => {
  let token;
  if (req.headers.authorization && req.headers.authorization.startsWith('Bearer')) {
    try {
      token = req.headers.authorization.split(' ')[1];
      const decoded = jwt.verify(token, process.env.JWT_SECRET);
      req.user = await User.findById(decoded.id).select('-password');
      next();
    } catch (error) {
      res.status(401);
      next(new Error('Not authorized, token failed'));
    }
  }
  if (!token) {
    res.status(401);
    next(new Error('Not authorized, no token'));
  }
};
```

Slika 7.10 Posrednički sloj *protect*

Završni posrednički sloj presreće iznimke i neuspjele zahtjeve te vraća standardiziran JSON odgovor s poljem *message*. HTTP status preuzima već postavljeni *res.statusCode* (ili pada na 500 ako je ostao 200), a *stack* se uključuje samo izvan produksijskog okruženja (*NODE_ENV !== 'production'*) (slika 7.11).

```
const errorHandler = (err, req, res, next) => {
  const statusCode = res.statusCode === 200 ? 500 : res.statusCode;
  res.status(statusCode).json({
    message: err.message,
    stack: process.env.NODE_ENV === 'production' ? null : err.stack,
  });
};
```

Slika 7.11 Funkcija za obradu grešaka u sustavu

Za trajne korisničke datoteke (npr. fotografija profila) koristi se *multer* s *diskStorage* strategijom: datoteke se spremaju u *public/uploads/*, a nazivu se dodaje vremenska oznaka i originalna ekstenzija. *fileFilter* ograničava MIME tipove na *image/jpeg* i *image/png*, čime se smanjuje rizik od neprihvatljivog sadržaja (slika 7.12). Ovaj posrednički sloj se primjenjuje na rute koje očekuju *multipart/form-data* (npr. ažuriranje profila). Za OCR tok obrade računa koristi se zasebna, memoriju varijanta *multer-a* unutar same rute, jer se slika ne pohranjuje trajno, nego se odmah enkodira i proslijeđuje servisu Document AI.

```

const storage = multer.diskStorage({
  destination: (req, file, cb) => { cb(null, "public/uploads/"); },
  filename: (req, file, cb) => { cb(null, Date.now() + path.extname(file.originalname)); },
});

const fileFilter = (req, file, cb) => {
  const allowedTypes = ["image/jpeg", "image/png"];
  if (allowedTypes.includes(file.mimetype)) {
    cb(null, true);
  } else {
    cb(new Error("Only .jpg and .png files are allowed"), false);
  }
};

const upload = multer({ storage, fileFilter });

```

Slika 7.12 Konfiguracija multer paketa za prijenos datoteka

7.5. Organizacija API ruta

Na poslužiteljskoj strani, REST API dostupan je pod prefiksom `/api`. Komunikacija se odvija u JSON formatu preko HTTPS-a, a pristup privatnim resursima dopušten je samo uz token u zaglavlju `Authorization`, čiju valjanost provjerava `protect` posrednički sloj prije izvršavanja kontrolera.

7.5.1. Rute korisničkih funkcionalnosti

U skupini korisničkih funkcionalnosti rute su izložene pod `/api/users`. Javno dostupne su registracija (POST `/api/users/register`) i prijava (POST `/api/users/login`). Za prijavljene korisnike dostupni su dohvati profila (GET `/api/users/profile`) i agregiranih statistika za profil (GET `/api/users/stats`). Ažuriranje profila na ruti PUT `/api/users/updateProfile` podržava slanje podatkovnog obrasca s poljem `photo` za sliku profila. Promjena lozinke izvodi se pozivom PUT `/api/users/changePassword`, gdje klijent šalje trenutnu i novu lozinku, a nakon validacije vjerodajnica, poslužitelj ažurira `hash` lozinke. Trajno brisanje korisničkog računa dostupno je preko rute DELETE `/api/users/deleteUser` koja zahtijeva potvrdu lozinke. Upravljačke operacije nad kategorijama uključuju stvaranje nove korisničke kategorije (POST `/api/users/newCategory`), brisanje postojeće (DELETE `/api/users/deleteCategory?name=...`, gdje se naziv predaje kao upitni parametar) te upravljanje omiljenima (POST `/api/users/addCategoryToFavourites`). Slika 7.13 prikazuje rute korisničkih funkcionalnosti.

```

router.post('/register', registerUser);           // POST /api/users/register - Register a new user
router.post('/login', authUser);                  // POST /api/users/login - Login user

router.get('/profile', protect, getMyProfile);    // GET /api/users/profile - Get user profile
router.get('/stats', protect, getUserStats);       // GET /api/users/stats - Get user stats (for profile)
router.put('/updateProfile', protect, upload.single("photo"), updateMyProfile); // PUT /api/users/updateProfile
router.put('/changePassword', protect, changePassword); // PUT /api/users/changePassword
router.post('/newCategory', protect, createCategory); // POST /api/users/newCategory - Create a new category
router.post('/addCategoryToFavourites', protect, addCategoryToFavourites); // POST /api/users/addCategoryToFavourites
router.delete('/deleteCategory', protect, deleteCategory); // DELETE /api/users/deleteCategory?name=Groceries - Delete selected category
router.delete('/deleteUser', protect, deleteUser); // DELETE /api/users/deleteUser

```

Slika 7.13 Rute korisničkih funkcionalnosti

7.5.2. Rute za upravljanje računima

Rute za račune nalaze se pod `/api/receipts`. Kreiranje novog računa (POST `/api/receipts/new`) prima popis stavki ugniježđenih u tijelu zahtjeva, dok se ažuriranje postojećeg računa provodi putem PUT `/api/receipts/update/:id`, uz eksplicitnu provjeru da traženi dokument pripada prijavljenom korisniku. Brisanje unosa omogućeno je rutom DELETE `/api/receipts/deleteReceipt` koja prima identifikator računa. Za dohvat podataka izložene su rute GET `/api/receipts/getAll` koja vraća sve korisnikove račune, sortirano prema vremenu nastanka, GET `/api/receipts/:id` za dohvat pojedinačnog računa te GET `/api/receipts/getCategoryItems?category=...` koja agregacijom vraća popis stavki za zadanu kategoriju zajedno s osnovnim metapodacima računa. Slika 7.14 prikazuje sve navedene rute vezane za upravljanje računima.

```
router.post('/new', protect, createReceipt);           // POST /api/receipts/new - Create a new receipt
router.put('/update/:id', protect, updateReceipt);     // PUT /api/receipts/update/:id
router.delete('/deleteReceipt', protect, deleteReceipt); // DELETE /api/receipts/deleteReceipt
router.get('/getAll', protect, getReceipts);           // GET /api/receipts/getAll - Get all receipts for the logged-in user
router.get('/getCategoryItems', protect, getCategoryItems); // GET /api/receipts/getCategory - Get all items for the selected category
router.get('/:id', protect, getReceiptById);           // GET /api/receipts/:id
```

Slika 7.14 Rute za upravljanje računima

7.6. Poslužiteljska poslovna logika API-ja

Kontroleri predstavljaju središnji sloj poslužiteljske logike, tj. preuzimaju validirane zahtjeve s ruta, koordiniraju rad s Mongoose modelima te vraćaju JSON odgovore uz dosljedne HTTP statuse. Autentifikacija se provodi prethodno s *middleware-om* `protect`, dok se centralizirano rukovanje pogreškama delegira završnom `errorHandleru`. Kontroleri su fokusirani na poslovna pravila, validaciju ulaza i oblikovanje odgovora. U sklopu projekta razlikujemo dva kontrolera `userController` i `receiptController`.

7.6.1. Modul za upravljanje korisnicima

U kontroleru `userController` okupljene su sve operacije nad korisnikom, odnosno registracija, prijava, dohvati i ažuriranje profila, statistike te upravljanje kategorijama i omiljenima. Implementira validaciju ulaza, koristi JWT za zaštitu ruta i vraća konzistentne HTTP odgovore.

Funkcija `registerUser`, (slika 7.15), provodi niz validacija prije kreiranja korisnika: provjerava postoji li e-pošta i korisničko ime, popunjenoš obavezni polja te minimalnu duljinu lozinke. Ako su uvjeti zadovoljeni, kreira korisnika i vraća odgovor *201 Created* s osnovnim podacima i JWT tokenom. U suprotnom vraća odgovarajuće *400 Bad Request* poruke.

```

const registerUser = async (req, res) => {
  const { username, email, password, name, surname } = req.body;
  const userExists = await User.findOne({ email });
  if (userExists) {
    res.status(400);
    throw new Error('Email already in use.');
  }
  if (!email || !username || !password) { return res.status(400).json({ message: "All fields are required." }); }
  if (password.length < 6) { return res.status(400).json({ message: "Password must be at least 6 characters long." }); }
  const existingUsername = await User.findOne({ username });
  if (existingUsername) { return res.status(400).json({ message: "Username already in use." }); }

  const photo = `${req.protocol}://${req.get("host")}/uploads/123.png`;
  const user = await User.create({ username, email, password, name, surname, photo });

  if (user) {
    res.status(201).json({
      _id: user._id,
      username: user.username,
      email: user.email,
      name: user.name,
      surname: user.surname,
      token: generateToken(user._id),
      favouriteCategories: user.favouriteCategories,
    });
  } else {
    res.status(400);
    throw new Error('Invalid user data');
  }
};

```

Slika 7.15 Funkcija za registriranje korisnika

Funkcija *authUser* (slika 7.16) obavlja prijavu korisnika tako da počinje provjerom da je poslano ispravno korisničko ime i lozinka, nakon čega se korisnik dohvata iz baze i lozinka uspoređuje metodom *matchPassword*.

```

const authUser = async (req, res, next) => {
  const { username, password } = req.body;
  if (!username || !password) { return res.status(400).json({ message: "All fields are required." }); }
  try {
    const user = await User.findOne({ username });
    if (user && (await user.matchPassword(password))) {
      res.json({
        _id: user._id,
        username: user.username,
        email: user.email,
        token: generateToken(user._id),
        favouriteCategories: user.favouriteCategories,
      });
    } else {
      const error = new Error('Invalid username or password');
      error.status = 401;
      next(error);
    }
  } catch (error) {
    next(error);
  }
};

```

Slika 7.16 Funkcija za prijavu postojećih korisnika

Funkcija *getMyProfile* (slika 7.17) na temelju korisničkog identifikatora *req.user._id* dohvata dokument korisnika. Odgovor vraća s ključnim poljima profila: korisničko ime, e-pošta, foto URL, ime i prezime, lokacija, opis, vrijeme kreiranja, kategorije i omiljene kategorije. Ako korisnik ne postoji, vraća se *404 Not Found*.

```

const getMyProfile = async (req, res) => {
  const user = await User.findById(req.user._id);
  if (user) {
    res.json({
      _id: user._id,
      username: user.username,
      email: user.email,
      photo: user.photo,
      name: user.name,
      surname: user.surname,
      location: user.location,
      bio: user.bio,
      joined: user.createdAt,
      categories: user.categories,
      favouriteCategories: user.favouriteCategories,
    });
  } else {
    res.status(404);
    throw new Error('User not found');
  }
};

```

Slika 7.17 Funkcija za dohvaćanje profila korisnika

Za ažuriranje profila koristi se funkcija *updateMyProfile* (slika 7.18) koja omogućuje ažuriranje tekstualnih polja profila (ime, prezime, lokacija, opis) te učitavanje fotografije. Ako je slika učitana, gradi se absolutni URL koristeći *req.protocol* i *req.get("host")*. Nakon uspješnog spremanja promjena putem *save()* funkcije vraća se ažurirani profil, a u slučaju pogreške šalje se status 500 s porukom.

```

const updateMyProfile = async (req, res) => {
  const user = await User.findById(req.user._id);
  if (user) {
    const { name, surname, location, bio } = req.body;
    user.name = name || user.name;
    user.surname = surname || user.surname;
    user.location = location || user.location;
    user.bio = bio || user.bio;
    if (req.file) { user.photo = `${req.protocol}://${req.get("host")}/uploads/${req.file.filename}`; }
    try {
      const updatedUser = await user.save();
      res.json({
        _id: updatedUser._id,
        username: updatedUser.username,
        email: updatedUser.email,
        photo: updatedUser.photo,
        name: updatedUser.name,
        surname: updatedUser.surname,
        location: updatedUser.location,
        bio: updatedUser.bio,
        joined: updatedUser.createdAt,
        categories: updatedUser.categories,
        favouriteCategories: updatedUser.favouriteCategories,
      });
    } catch (error) {
      res.status(500).json({ message: "Failed to update profile." });
    }
  } else {
    res.status(404);
    throw new Error("User not found");
  }
};

```

Slika 7.18 Funkcija za ažuriranje korisničkih podataka

Slika 7.19 prikazuje funkciju *getUserStats* koja dohvaća statistiku za određenog korisnika. Nad kolekcijom *Receipt* računaju se agregati: ukupan broj računa, ukupna potrošnja, potrošnja u tekućem mjesecu te prosječna vrijednost računa. Dodatno se akumulira potrošnja po kategorijama stavki te se rezultat sortira silazno i izdvajaju tri kategorije s najvećim iznosima. Vrijednosti se zaokružuju na dvije decimale i vraćaju u JSON odgovoru zajedno s ostalim statističkim pokazateljima.

```

const getUserStats = async (req, res) => {
  if (!req.user || !req.user._id) { return res.status(401).json({ message: "Unauthorized" }); }
  try {
    const receipts = await Receipt.find({ user: req.user._id });
    const totalReceipts = receipts.length;
    let totalSpent = 0;
    let currentMonthSpent = 0;
    const categoryTotals = {};
    const now = new Date();
    const currentMonth = now.getMonth();
    const currentYear = now.getFullYear();

    receipts.forEach((receipt) => {
      const date = new Date(receipt.date);
      const isCurrentMonth = date.getMonth() === currentMonth && date.getFullYear() === currentYear;

      if (isCurrentMonth) { currentMonthSpent += receipt.totalAmount || 0; }

      totalSpent += receipt.totalAmount || 0;
      receipt.items.forEach((item) => {
        if (Array.isArray(item.categories)) {
          item.categories.forEach((category) => {
            if (!categoryTotals[category]) { categoryTotals[category] = 0; }
            categoryTotals[category] += item.totalPrice;
          });
        }
      });
    });

    const avgPerReceipt = totalReceipts > 0 ? totalSpent / totalReceipts : 0;
    const topCategories = Object.entries(categoryTotals).sort((a, b) => b[1] - a[1]).slice(0, 3)
      .map(([category, total]) => ({ category, total: Number(total.toFixed(2)) }));
    res.status(200).json({
      totalReceipts,
      totalSpent: Number(totalSpent.toFixed(2)),
      avgPerReceipt: Number(avgPerReceipt.toFixed(2)),
      currentMonthSpent: Number(currentMonthSpent.toFixed(2)),
      topCategories,
    });
  }
};

```

Slika 7.19 Funkcija za dohvaćanje statistike korisnika

Za stvaranje nove kategorije koristi se funkcija *createCategory* (slika 7.20). Nakon uspješne autentifikacije, provjerava se ime kategorije te postojanje duplikata. Ako su zadovoljeni svi uvjeti, nova kategorija se uspješno sprema u bazu podataka.

```

const createCategory = async (req, res) => {
  const { name } = req.body;

  if (!req.user || !req.user._id) { return res.status(401).json({ message: "Unauthorized" }); }

  if (!name || typeof name !== 'string' || name.trim() === '') { return res.status(400).json({ message: "Category name is required" }); }

  try {
    const user = await User.findById(req.user._id);

    if (!user) { return res.status(404).json({ message: "User not found" }); }

    const trimmedName = name.trim();
    const categoryExists = user.categories.some(cat => cat.toLowerCase() === trimmedName.toLowerCase());

    if (categoryExists) { return res.status(400).json({ message: "Category with that name already exists." }); }

    user.categories.push(trimmedName);
    await user.save();

    res.status(201).json({ message: "Category created successfully", categories: user.categories });
  } catch (error) {
    console.error("Error creating category:", error);
    res.status(500).json({ message: "Server error while creating category." });
  }
};

```

Slika 7.20 Funkcija za stvaranje nove kategorije

Za razliku od kreiranja kategorije, brisanje kategorije odvija se na dva mesta: iz korisnikovih popisa (*categories* i *favouriteCategories*) te iz svih korisnikovih računa putem funkcije *updateMany*. Odgovor uključuje broj modificiranih računa i preostale korisničke kategorije. Slika 7.21 prikazuje opisani postupak brisanja kategorije koji se provodi funkcijom *deleteCategory*.

```

const deleteCategory = async (req, res) => {
  const name = req.query.name;
  if (!req.user || !req.user._id) { return res.status(401).json({ message: "Unauthorized" }); }
  if (!name || typeof name !== 'string' || name.trim() === '') { return res.status(400).json({ message: "Category name is required" }); }

  try {
    const trimmedName = name.trim();
    const user = await User.findById(req.user._id);
    if (!user) return res.status(404).json({ message: "User not found" });
    const initialLength = user.categories.length;
    user.categories = user.categories.filter(cat => cat.toLowerCase() !== trimmedName.toLowerCase());

    if (user.categories.length === initialLength) { return res.status(404).json({ message: "Category not found in user profile" }); }

    user.favouriteCategories = user.favouriteCategories.filter(favCat => favCat.toLowerCase() !== trimmedName.toLowerCase());
    await user.save();

    const updateResult = await Receipt.updateMany(
      {
        user: req.user._id,
        "items.categories": trimmedName
      },
      {
        $pull: { "items.$[].categories": trimmedName }
      }
    );

    res.status(200).json({
      message: `Category '${trimmedName}' deleted successfully`,
      updatedReceipts: updateResult.modifiedCount,
      remainingCategories: user.categories,
    });
  } catch (error) {
    console.error("Error deleting category:", error);
    res.status(500).json({ message: "Server error while deleting category." });
  }
}

```

Slika 7.21 Funkcija za brisanje kategorije

Funkcija *addCategoryToFavourites* (slika 7.22) zadužena je za dodavanje ili uklanjanje kategorije iz liste omiljenih kategorija. Nakon početnih validacija (autentifikacija, neprazan *categoryName*, logički *add*), u bloku *try* dohvata se korisnik, naziv se normalizira i provjerava postoji li tražena kategorija u korisničkom popisu. Ako je *add* zastavica postavljena na vrijednost true, funkcija sprječava duplike te nameće ograničenje od najviše četiri omiljene. Nakon *user.save()* vraća se poruka o dodavanju/uklanjanju, ažurirana lista *favouriteCategories* i brojač.

```

const addCategoryToFavourites = async (req, res) => {
  const { categoryName, add } = req.body;
  if (!req.user || !req.user._id) { -}
  if (!categoryName || typeof categoryName !== 'string' || categoryName.trim() === '') { -}
  if (typeof add !== 'boolean') { -}
  try {
    const trimmedName = categoryName.trim();
    const user = await User.findById(req.user._id);
    if (!user) return res.status(404).json({ message: "User not found" });
    const categoryExists = user.categories.some(cat => cat.toLowerCase() === trimmedName.toLowerCase());
    if (!categoryExists) { return res.status(400).json({ message: "Category does not exist in user's categories" }); }
    if (add) {
      if (user.favouriteCategories.some(favCat => favCat.toLowerCase() === trimmedName.toLowerCase())) {
        return res.status(400).json({ message: "Category is already a favourite" });
      }
      if (user.favouriteCategories.length >= 4) {
        return res.status(400).json({ message: "Favourites limit reached. You can't add more." });
      }
      user.favouriteCategories.push(trimmedName);
    } else {
      user.favouriteCategories = user.favouriteCategories.filter(favCat => favCat.toLowerCase() !== trimmedName.toLowerCase());
    }
    await user.save();
    res.status(200).json({
      message: add
        ? `Category '${trimmedName}' added to favourites`
        : `Category '${trimmedName}' removed from favourites`,
      favouriteCategories: user.favouriteCategories,
      favouritesCount: user.favouriteCategories.length,
    });
  } catch (error) {

```

Slika 7.22 Funkcija za dodavanje kategorije u popis omiljenih

Funkcija *changePassword* (slika 7.23) provodi cjelovit, sigurnosno vođen tok izmjene lozinke. Prvo se provjerava autentičnost zahtjeva (postojanje *req.user* i njegovog identifikatora), a potom valjanost ulaznih podataka (potrebne su trenutačna i nova lozinka, pri čemu nova lozinka ne smije biti kraća od 6 znakova). Nakon toga dohvaća se korisnik iz baze uz eksplicitno uključivanje polja *password* te se pozivom metode *user.matchPassword(currentPassword)* provjerava točnost postojeće lozinke. Ako je provjera uspješna, atribut *user.password* se ažurira novom vrijednošću. Ažuriranje lozinke aktivira hashiranje kroz Mongoose *hook*, čime se nezaštićena lozinka nikad ne pohranjuje. Funkcija vraća status 200 i poruku o uspjehu, a u svim ostalim slučajevima vraćaju se odgovarajući HTTP statusi s jasnim opisom pogreške.

```
const changePassword = async (req, res) => {
  const { currentPassword, newPassword } = req.body;

  if (!req.user || !req.user._id) { return res.status(401).json({ message: "Unauthorized" }); }
  if (!currentPassword || !newPassword) { return res.status(400).json({ message: "Both current and new passwords are required" }); }
  if (newPassword.length < 6) { return res.status(400).json({ message: "New password must be at least 6 characters long" }); }

  try {
    const user = await User.findById(req.user._id).select("+password");
    if (!user) { return res.status(404).json({ message: "User not found" }); }
    const isMatch = await user.matchPassword(currentPassword);
    if (!isMatch) { return res.status(401).json({ message: "Current password is incorrect" }); }

    user.password = newPassword;
    await user.save();
    res.status(200).json({ message: "Password changed successfully" });
  } catch (error) {
    console.error("Error changing password:", error);
    res.status(500).json({ message: "Server error while changing password" });
  }
};
```

Slika 7.23 Funkcija za promjenu lozinke

Funkcija za brisanje računa zahtijeva autentificiran zahtjev i dodatnu potvrdu lozinkom čime se onemogućavaju slučajna ili neovlaštena brisanja. Nakon provjere autentičnosti, dohvaća se korisnik te se provjerava unesena lozinka metodom *user.matchPassword*. U slučaju uspjeha, najprije se brišu svi povezani zapisi o računima, a zatim i sam korisnički dokument, čime se osigurava konzistentnost i uklanjanje osobnih podataka (slika 7.24).

```
const deleteUser = async (req, res) => {
  const { password } = req.query;
  if (!req.user || !req.user._id) { return res.status(401).json({ message: "Unauthorized" });}

  try {
    const user = await User.findById(req.user._id);
    if (!user) { return res.status(404).json({ message: "User not found" }); }
    const isMatch = await user.matchPassword(password);
    if (!isMatch) { return res.status(401).json({ message: "Incorrect password" });}

    await Receipt.deleteMany({ user: user._id });
    await user.deleteOne();

    res.status(200).json({ message: "User account and related receipts deleted successfully" });
  } catch (error) {
    console.error("Error deleting user:", error);
    res.status(500).json({ message: "Server error while deleting user" });
  }
};
```

Slika 7.24 Funkcija za brisanje korisničkog profila

7.6.2. Modul za upravljanje računima

Kontroler *receiptController* objedinjuje sve operacije nad računima kao što su kreiranje i ažuriranje računa, dohvati svih i pojedinačnih zapisa računa te agregirani dohvati stavki računa po kategoriji. Kontroler validira ulaz, veže zapise uz prijavljenog korisnika i prije izmjena provjerava vlasništvo, dok je pristup rutama zaštićen JWT-om.

Funkcija *createReceipt* (slika 7.25) prima podatke o računu i najprije provjerava valjanog JWT-a i ulaznog tijela tako da mora postojati barem jedna stavka na računu, a svaka stavka mora sadržavati *name*, *quantity*, *unitPrice* i *totalPrice*. Uspješnim prolaskom provjera kreira se račun vezan uz trenutno prijavljenog korisnika (*req.user._id*).

```
const createReceipt = async (req, res) => {
  const { items, note, paymentMethod, tags, store, date } = req.body;

  if (!req.user || !req.user._id) { return res.status(401).json({ message: "Unauthorized" }); }
  if (!items || !Array.isArray(items) || items.length === 0) { return res.status(400).json({ message: "Receipt must include at least one item." }); }
  for (const item of items) {
    if (!item.name || !item.quantity || !item.unitPrice || !item.totalPrice) {
      return res.status(400).json({ message: "Each item must have name, quantity, unitPrice, and totalPrice." });
    }
  }
  try {
    const receipt = await Receipt.create({
      user: req.user._id,
      items,
      note,
      paymentMethod,
      tags,
      store,
      date,
    });
    res.status(201).json(receipt);
  } catch (error) {
    console.error("Error creating receipt:", error);
    res.status(500).json({ message: "Server error while saving receipt." });
  }
};
```

Slika 7.25 Funkcija za stvaranje unesenog računa

Slika 7.26 prikazuje funkciju *updateReceipt* koja proširuje prethodnu funkciju izmjenom postojećeg dokumenta tj. računa. Nakon uspješne provjere, dohvaća se dokument s određenim identifikatorom te se provodi kontrola vlasništva. Korisnik ima pristup dokumentu jedino ako je njegov korisnički identifikator jednak *req.user._id* te se u tom slučaju dokument ažurira i sprema.

```

const updateReceipt = async (req, res) => {
  const receiptId = req.params.id;
  const { items, note, paymentMethod, tags, store, date } = req.body;

  if (!req.user || !req.user._id) { return res.status(401).json({ message: "Unauthorized" }); }
  if (!items || !Array.isArray(items) || items.length === 0) { return res.status(400).json({ message: "Receipt must include at least one item." }); }
  for (const item of items) {
    if (!item.name || !item.quantity || !item.unitPrice || !item.totalPrice) {
      return res.status(400).json({ message: "Each item must have name, quantity, unitPrice, and totalPrice." });
    }
  }
  try {
    const receipt = await Receipt.findById(receiptId);
    if (!receipt) { return res.status(404).json({ message: "Receipt not found." }); }
    if (receipt.user.toString() !== req.user._id.toString()) { return res.status(403).json({ message: "Forbidden: You don't have permission to update this receipt." }); }

    receipt.items = items;
    receipt.note = note;
    receipt.paymentMethod = paymentMethod;
    receipt.tags = tags;
    receipt.store = store;
    receipt.date = date;

    const updatedReceipt = await receipt.save();
    res.status(200).json(updatedReceipt);
  } catch (error) {

```

Slika 7.26 Funkcija za ažuriranje računa

Funkcija `getReceiptById` (slika 7.27) omogućuje dohvati pojedinačnog računa tako što se u bazi podataka traži dokument s kombinacijom traženog ID dokumenta i ID prijavljenog korisnika, što znači da korisnik može vidjeti samo vlastite račune. Uspješnom pretragom, funkcija vraća traženi dokument.

```

const getReceiptById = async (req, res) => {
  const userId = req.user?._id;
  const receiptId = req.params.id;

  if (!userId) { return res.status(401).json({ message: "Unauthorized" }); }
  try {
    const receipt = await Receipt.findOne({ _id: receiptId, user: userId });
    if (!receipt) { return res.status(404).json({ message: "Receipt not found" }); }
    res.json(receipt);
  } catch (error) {
    console.error(error);
    res.status(500).json({ message: "Server error" });
  }
};

```

Slika 7.27 Funkcija za dohvaćanje određenog računa

Za razliku od prethodne funkcije, funkcija `getReceipts` (slika 7.28) vraća sve račune prijavljenog korisnika, a ne samo jedan određeni. Nakon provjere JWT-a dohvaća se kolekcija filtrirana po jedinstvenoj korisničkoj oznaci (korisnički ID) i sortira se silazno po vremenu nastanka (`createdAt: -1`) kako bi noviji zapisi bili prvi.

```

const getReceipts = async (req, res) => {
  if (!req.user || !req.user._id) { return res.status(401).json({ message: "Unauthorized" }); }
  try {
    const receipts = await Receipt.find({ user: req.user._id }).sort({ createdAt: -1 });
    res.status(200).json(receipts);
  } catch (error) {
    console.error("Error fetching receipts:", error);
    res.status(500).json({ message: "Server error while fetching receipts." });
  }
};

```

Slika 7.28 Funkcija za dohvaćanje svih računa

Funkcija `getCategoryItems` (slika 7.29) provodi serversku agregaciju za prikaz stavki po kategoriji. Cjevovod (eng. *pipeline*) najprije filtrira račune prijavljenog korisnika (`$match`), zatim projekcijom (`$project`) izdvaja samo potrebna polja, rastavlja polje `items` (`$unwind`), filtrira stavke na temelju naziva kategorije uz „točan“ *case-insensitive regex* (`^category$`), te kroz `$addFields` svakoj stavci pridružuje osnovne metapodatke računa (datum, trgovina, ukupni iznos, ID računa). Na kraju, `replaceRoot` vraća svaku stavku kao zaseban dokument i šalje se poruka *200 OK* s listom stavki.

```
const getCategoryItems = async (req, res) => {
  if (!req.user || !req.user._id) { return res.status(401).json({ message: "Unauthorized" }); }
  const category = req.query.category;
  if (!category) { return res.status(400).json({ message: "Category query parameter is required." }); }
  try {
    const filteredItems = await Receipt.aggregate([
      { $match: { user: req.user._id } },
      {
        $project: {
          items: 1,
          receiptDate: "$date",
          receiptStore: "$store",
          receiptTotal: "$totalAmount"
        }
      },
      { $unwind: "$items" },
      {
        $match: {
          "items.categories": { $in: [new RegExp(`^${category}$`, "i")] }
        }
      },
      {
        $addFields: {
          "items.receiptDate": "$receiptDate",
          "items.receiptStore": "$receiptStore",
          "items.receiptTotal": "$receiptTotal",
          "items.receiptId": "$_id"
        }
      },
      {
        $replaceRoot: { newRoot: "$items" }
      }
    ]);
    res.status(200).json(filteredItems);
  } catch (error) {
    console.error("Error fetching category items:", error);
    res.status(500).json({ message: "Server error while fetching category items." });
  }
}
```

Slika 7.29 Funkcija za dohvatanje svih stavki određene kategorije

Kontrolorska funkcija `deleteReceipt` (slika 7.30) provodi sigurno brisanje pojedinačnog računa uz provjeru vlasništva. Identifikator računa čita se iz upitnog parametra `selectedId`, a zahtjev se odmah odbija s kodom 401 ako nije autentificiran. Zatim se računu pristupa isključivo kroz upit `Receipt.findOne({ _id: selectedId, user: req.user._id })`, čime se onemogućava brisanje tuđih zapisa. Ako je račun pronađen, poziva se `receipt.deleteOne()` te se klijentu vraća status 200 s porukom *Receipt deleted*, a u slučaju da dokument ne postoji, vraća se odgovor *404 Receipt not found*.

```
const deleteReceipt = async (req, res) => {
  const selectedId = req.query.selectedId;

  if (!req.user || !req.user._id) {
    return res.status(401).json({ message: "Unauthorized" });
  }

  try {
    const receipt = await Receipt.findOne({ _id: selectedId, user: req.user._id });
    if (!receipt) {
      return res.status(404).json({ message: 'Receipt not found.' });
    }
    await receipt.deleteOne();
    return res.status(200).json({ message: 'Receipt deleted.' });
  } catch (error) {
    console.error("Error deleting receipt:", error);
    res.status(500).json({ message: "Server error while deleting receipt." });
  }
}
```

Slika 7.30 Funkcija za brisanje određenog računa

Zaključno, kontroleri *userController* i *receiptController* provode ključnu domensku logiku sustava uz dosljedan REST pristup tako da se ulaz validira, pristup se štiti JWT-om, vlasništvo nad resursima se provjerava, a odgovori su uredno mapirani u JSON s jasnim HTTP statusima. Ovakva separacija odgovornosti (rute, *middleware*, kontroler, baza podataka) pojednostavljuje održavanje i testiranje.

8. ZAKLJUČAK

Kontrola osobnih financija danas je gotovo nezamisliva bez korištenja digitalnih alata koji omogućuju jednostavan i brz pregled potrošnje. Mobilne aplikacije posebno su pogodne za tu svrhu jer korisnicima nude stalnu dostupnost, pregledno sučelje i mogućnost unosa podataka u stvarnom vremenu. Dodatna vrijednost postiže se automatizacijom obrade računa, čime se smanjuje potreba za ručnim unosom i ostvaruje značajna vremenska ušteda.

Za izradu mobilnog sustava za praćenje i analizu osobnih financija bilo je potrebno kombinirati različite tehnologije. Klijentski dio razvijen je u React Native i Expo okruženju, što je omogućilo izradu prilagodljive i responzivne aplikacije. Poslužiteljski dio izrađen je u Node.js i Express.js tehnologijama, dok je za pohranu podataka korištena MongoDB baza podataka. Sigurnost sustava ostvarena je primjenom JSON Web Token tehnologije, koja osigurava siguran prijenos podataka i pouzdanu autentifikaciju korisnika. Automatizirano prepoznavanje i obrada podataka s računa ostvareni su pomoću Google Document AI servisa, čime je ostvaren značajan iskorak u jednostavnosti korištenja aplikacije.

Razvijena aplikacija korisnicima omogućuje upravljanje profilom, unos i kategorizaciju računa, pregled povijesti transakcija, sortiranje i filtriranje podataka te vizualizaciju potrošnje. Pored osnovnih funkcionalnosti, korisnicima je osigurana mogućnost pretraživanja i grupiranja podataka prema različitim kriterijima, što olakšava detaljnu analizu strukture troškova. Vizualni prikazi potrošnje dodatno pridonose razumijevanju finansijskih navika i omogućuju brže prepoznavanje kategorija u kojima dolazi do najveće potrošnje. Na taj način aplikacija ne služi samo kao alat za bilježenje transakcija, već i kao podrška u finansijskom planiranju i donošenju odluka, pružajući korisnicima pregledan uvid u osobne financije te veću kontrolu nad troškovima.

LITERATURA

- [1] React Native: Introduction, s Interneta, <https://reactnative.dev/docs/getting-started>, zadnji pristup: 2.9.2025
- [2] Expo: Documentation, s Interneta, <https://expo.dev/>, zadnji pristup: 2.9.2025
- [3] npm: Axios, s Interneta, <https://www.npmjs.com/package/axios>, zadnji pristup: 2.9.2025
- [4] GeeksforGeeks: Axios in React Native, s Interneta, <https://www.geeksforgeeks.org/react-native/axios-in-react-native/>, zadnji pristup: 2.9.2025
- [5] NativeWind: Documentation, s Interneta, <https://www.nativewind.dev/>, zadnji pristup: 2.9.2025
- [6] npm: react-native-chart-kit, s Interneta, <https://www.npmjs.com/package/react-native-chart-kit>, zadnji pristup: 2.9.2025
- [7] npm: react-native-svg, s Interneta, <https://www.npmjs.com/package/react-native-svg>, zadnji pristup: 2.9.2025
- [8] Node.js: Introduction, s Interneta, <https://nodejs.org/en>, zadnji pristup: 2.9.2025
- [9] Express: Documentation, s Interneta, <https://expressjs.com/>, zadnji pristup: 2.9.2025
- [10] MongoDB: Introduction, s Interneta, <https://www.mongodb.com/>, zadnji pristup: 2.9.2025
- [11] JWT: Introduction, s Interneta, <https://www.jwt.io/>, zadnji pristup: 2.9.2025
- [12] npm: jsonwebtoken, s Interneta, <https://www.npmjs.com/package/jsonwebtoken>, zadnji pristup: 2.9.2025
- [13] npm: bcryptjs, s Interneta, <https://www.npmjs.com/package/bcryptjs>, zadnji pristup: 2.9.2025
- [14] Google Cloud: Document AI Overview, s Interneta, <https://cloud.google.com/document-ai/docs/overview>, zadnji pristup: 2.9.2025

[15] npm: dotenv, s Interneta, <https://www.npmjs.com/package/dotenv>, zadnji pristup:
2.9.2025

[16] npm: cors, s Interneta, <https://www.npmjs.com/package/cors>, zadnji pristup:
2.9.2025

[17] npm: multer, s Interneta, <https://www.npmjs.com/package/multer>, zadnji pristup:
2.9.2025

PRILOZI

Kazalo slika i tablica

Kazalo slika

Slika 2.1 Prikaz onboarding zaslona	2
Slika 2.2 Zaslon za kreiranje novog korisničkog profila	3
Slika 2.3 Zaslon za prijavu postojećeg korisnika	3
Slika 2.4 Prikaz početnog zaslona.....	4
Slika 2.5 Zaslon za unos računa	5
Slika 2.6 Zaslon čekanja podataka	5
Slika 2.7 Forma s podacima novog računa	5
Slika 2.8 Pregled unesenog računa.....	6
Slika 2.9 Odustanak od unosa	6
Slika 2.10 Potvrda uspjeha	6
Slika 2.11 Prikaz svih kategorija.....	7
Slika 2.12 Prikaz opcije sortiranja.....	7
Slika 2.13 Kreiranje nove kategorije.....	7
Slika 2.14 Prikaz određene kategorije	7
Slika 2.15 Prikaz sortiranja artikala u kategoriji	7
Slika 2.16 Prikaz računa odabranog artikla.....	7
Slika 2.17 Prikaz omiljenih kategorija na početnom zaslonu	8
Slika 2.18 Modalni prozor za obavještavanje o dosegnutom ograničenju	8
Slika 2.19 Prikaz zaslona za analizu potrošnje	8
Slika 2.20 Prikaz „History“ zaslona.....	9
Slika 2.21 Prikaz izbornika za sortiranje.....	9
Slika 2.22 Prikaz izbornika za filtriranje.....	9
Slika 2.23 Prikaz tipke „Load More“	10
Slika 2.24 Modalni prikaz pojedinačnog računa	10
Slika 2.25 Prikaz zaslona korisničkog profila	11
Slika 2.26 Modal za uređivanje profila	11
Slika 5.1 Arhitektura sustava	23
Slika 5.2 Tokovi podataka.....	25
Slika 5.3 Tokovi podataka.....	26

Slika 6.1 Struktura klijentskog sloja	27
Slika 6.2 Konfiguracija Axios klijenta	28
Slika 6.3 Validacija unosa podataka	30
Slika 6.4 Pohrana tokena i preusmjeravanje	30
Slika 6.5 Servisna funkcija za registriranje korisnika	31
Slika 6.6 Funkcija za prijavu korisnika.....	31
Slika 6.7 Servisna funkcija za prijavu korisnika	31
Slika 6.8 Dohvat korisničkih podataka i statistika	32
Slika 6.9 Servisna funkcija za dohvaćanje profila	32
Slika 6.10 Dohvaćanje statistike korisnika	32
Slika 6.11 Funkcija za spremanje izmjena profila	33
Slika 6.12 Servisna funkcija za spremanje izmjena profila	33
Slika 6.13 Čišćenje sesije i povratak na onboardingu	34
Slika 6.14 Promjena lozinke.....	34
Slika 6.15 Servisna funkcija za promjenu lozinke	34
Slika 6.16 Brisanje profila.....	35
Slika 6.17 Servisna funkcija za brisanje profila	35
Slika 6.18 Otvaranje kamere i dohvat fotografije	36
Slika 6.19 Poziv obrade i mapiranje rezultata.....	36
Slika 6.20 Slanje slike na poslužitelj.....	36
Slika 6.21 Uvoz računa iz galerije	37
Slika 6.22 Inicijalizacija i preuzimanje parametara	38
Slika 6.23 Try/catch blok funkcije handleSave.....	38
Slika 6.24 Servisni poziv createReceipt.....	39
Slika 6.25 Servisni poziv updateReceipt.....	39
Slika 6.26 Potvrda odustajanja od uređivanja	39
Slika 6.27 Priprema podataka i navigacija na pretpregled	39
Slika 6.28 Pretvorba parametra rute u objekt pretpregleda	40
Slika 6.29 Osvježavanje liste na fokus zaslona.....	40
Slika 6.30 Servisni poziv za dohvat svih računa.....	40
Slika 6.31 Filtriranje računa	41
Slika 6.32 Sortiranje računa	41
Slika 6.33 Pregled odabranog računa.....	42
Slika 6.34 Funkcija za potvrdu i brisanje računa	42

Slika 6.35 Servisna funkcija za brisanje odabranog računa	42
Slika 6.36 Dohvat kategorija i statistike.....	43
Slika 6.37 Poziv modala za kreiranje nove kategorije	43
Slika 6.38 Modal za stvaranje nove kategorije	44
Slika 6.39 Servisna funkcija createCategory.....	44
Slika 6.40 Ažuriranje liste nakon kreiranja nove kategorije	45
Slika 6.41 Dohvat podataka za pregled kategorije	45
Slika 6.42 Servisna funkcija za dohvaćanje stavki kategorije	46
Slika 6.43 Izračun sažetaka za kategoriju	46
Slika 6.44 Brisanje kategorije	47
Slika 6.45 Pozadinski poziv za brisanje kategorije	47
Slika 6.46 Dodavanje kategorije u popis omiljenih	47
Slika 6.47 Servisna funkcija dodavanje kategorije u omiljene	48
Slika 6.48 Prikaz omiljenih kategorija	48
Slika 6.49 Funkcija za analizu potrošnje tekućeg mjeseca	49
Slika 6.50 Računanje potrošnje za zadnja četiri mjeseca	50
Slika 6.51 Graf za prikaz potrošnje u posljednja četiri mjeseca	50
Slika 6.52 Funkcija za računanje potrošnje u prethodnom i tekućem mjesecu.....	51
Slika 6.53 Funkcija za računanje prosječne vrijednosti računa	51
Slika 6.54 Funkcije za crtanje „pie“ grafa.....	52
Slika 6.55 Funkcija CategorySpendingPieChart	52
Slika 6.56 Prikaz udjela potrošnje po trgovinama.....	53
Slika 7.1 Struktura poslužiteljskog sloja	54
Slika 7.2 Povezivanje s bazom podataka	54
Slika 7.3 Model User.....	56
Slika 7.4 Model stavki računa	56
Slika 7.5 Model računa.....	57
Slika 7.6 OCR obrada računa	57
Slika 7.7 Poslužiteljska ruta /process i poziv Document AI procesora.....	58
Slika 7.8 Filtriranje entiteta i mapiranje globalnih atributa računa.....	59
Slika 7.9 Mapiranje entiteta stavki u strukturu items.....	59
Slika 7.10 Posrednički sloj protect.....	60
Slika 7.11 Funkcija za obradu grešaka u sustavu.....	60
Slika 7.12 Konfiguracija multer paketa za prijenos datoteka.....	61

Slika 7.13 Rute korisničkih funkcionalnosti	61
Slika 7.14 Rute za upravljanje računima.....	62
Slika 7.15 Funkcija za registriranje korisnika.....	63
Slika 7.16 Funkcija za prijavu postojećih korisnika	63
Slika 7.17 Funkcija za dohvaćanje profila korisnika	64
Slika 7.18 Funkcija za ažuriranje korisničkih podataka.....	64
Slika 7.19 Funkcija za dohvaćanje statistike korisnika.....	65
Slika 7.20 Funkcija za stvaranje nove kategorije	65
Slika 7.21 Funkcija za brisanje kategorije	66
Slika 7.22 Funkcija za dodavanje kategorije u popis omiljenih.....	66
Slika 7.23 Funkcija za promjenu lozinke	67
Slika 7.24 Funkcija za brisanje korisničkog profila	67
Slika 7.25 Funkcija za stvaranje unesenog računa.....	68
Slika 7.26 Funkcija za ažuriranje računa	69
Slika 7.27 Funkcija za dohvaćanje određenog računa	69
Slika 7.28 Funkcija za dohvaćanje svih računa.....	69
Slika 7.29 Funkcija za dohvaćanje svih stavki određene kategorije	70
Slika 7.30 Funkcija za brisanje određenog računa	71

Kazalo tablica

Tablica 4.1 Funkcionalni zahtjevi	19
Tablica 4.2 Nefunkcionalni zahtjevi	21

Popis oznaka i kratica

API	Application Programming Interface
AI	Artificial Intelligence
JSON	JavaScript Object Notation
JWT	JSON Web Token
SVG	Scalable Vector Graphics
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure

REST	Representational State Transfer
CRUD	Create, Read, Update, Delete
CSS	Cascading Style Sheets
SQL	Structured Query Language
NoSQL	Not Only SQL
ODM	Object Document Mapper
OCR	Optical Character Recognition
IAM	Identity and Access Management
CORS	Cross-Origin Resource Sharing
FZ	Funkcionalni zahtjev
NFZ	Nefunkcionalni zahtjev
URL	Uniform Resource Locator
URI	Uniform Resource Identifier
MIME	Multipurpose Internet Mail Extensions
UI	User Interface
UML	Unified Modeling Language

SAŽETAK

U okviru ovog diplomskog rada razvijen je mobilni sustav za praćenje i analizu osobnih financija koji korisnicima na jednostavan i intuitivan način omogućuje unos, pregled i analizu troškova. Klijentski dio aplikacije implementiran je korištenjem React Native i Expo razvojnih okvira, dok je poslužiteljski dio izrađen u Node.js i Express.js okruženju te povezan s bazom podataka MongoDB. Sustav dodatno koristi uslugu Google Document AI za automatsko prepoznavanje i ekstrakciju podataka s računa, čime se značajno ubrzava i olakšava unos finansijskih informacija. Korisnicima je omogućeno upravljanje profilom, unos i kategorizacija računa, pregled povijesti transakcija te vizualizacija potrošnje. Sustav koristi JSON Web Token za sigurnu autentifikaciju i komunikaciju, a dodatno nudi filtriranje i sortiranje podataka te pregled najčešćih kategorija i trgovina, čime se korisnicima olakšava upravljanje osobnim financijama i kontrola troškova.

KLJUČNE RIJEČI

React Native, Node.js, MongoDB, Google Document AI, mobilna aplikacija, osobne financije

SUMMARY

Title: Implementation of a Mobile System for Personal Finance Tracking and Analysis

As part of this master's thesis, a mobile system for tracking and analyzing personal finances was developed, providing users with a simple and intuitive way to enter, review, and analyze expenses. The client-side application was implemented using React Native and Expo frameworks, while the server-side was built in Node.js and Express.js and connected to a MongoDB database. The system additionally uses Google Document AI for automatic recognition and extraction of receipt data, significantly accelerating and simplifying the entry of financial information. Users can manage their profiles, enter and categorize receipts, review transaction history, and visualize expenses. The system employs JSON Web Token for secure authentication and communication and further enables filtering and sorting of data as well as viewing the most common categories and stores, thereby simplifying personal finance management and expense control.

KEYWORDS

React Native, Node.js, MongoDB, Google Document AI, mobile application, personal finance