Dizajn i implementacija modularnosti mobilnih aplikacija pisanih u Flutteru

Kunštek, Antonio

Master's thesis / Diplomski rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:871778

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Antonio Kunštek

DIZAJN I IMPLEMENTACIJA MODULARNOSTI MOBILNIH APLIKACIJA PISANIH U FLUTTERU

DIPLOMSKI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Antonio Kunštek

JMBAG: 0016133325

Studij: Organizacija poslovnih sustava

DIZAJN I IMPLEMENTACIJA MODULARNOSTI MOBILNIH APLIKACIJA PISANIH U FLUTTERU

DIPLOMSKI RAD

Mentor:

Izv. prof. dr. sc. Zlatko Stapić

Varaždin, rujan 2023.

Antonio Kunštek

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Rad “Dizajn i implementacija modularnosti mobilnih aplikacija pisanih u Flutteru” predstavlja istraživanje najčešćih arhitektura korištenih u izradi Flutter mobilnih aplikacija. Arhitektura programskog proizvoda bavi se spajanjem različitih komponenti korisničkog sučelja i poslovne logike s ciljem poboljšanja kvalitete programskog proizvoda te njegovog budućeg održavanja. Postoje različite vrste arhitektura mobilnih aplikacija, a sam odabir ovisi o tehnologiji i zadanom problemu. U ovome radu teorijski su obrađene vrste mobilnih aplikacija te su objašnjeni pojmovi arhitektura i modularnost mobilnih aplikacija. Prikazane su različite vrste arhitekture Flutter mobilnih aplikacija s pratećim UML dijagramima. Na praktičnom primjeru prikazan je način izrade modularne Flutter aplikacije koristeći BLoC (Business Logic Components) arhitekturu, odvojen na tri zasebna modula na primjeru aplikacije za praćenje vlastitog budžeta. Cilj ovog rada je prikazati dizajn i implementaciju jedne od arhitektura mobilnih aplikacija koristeći Flutter razvojni okvir.

Ključne riječi: Flutter, više-platformske aplikacije, arhitektura mobilnih aplikacija, modularnost

SADRŽAJ:

1. UVOD

Mobilne aplikacije postale su neizostavan dio svakodnevnog života čovjeka. Koriste se za razne stvari, od naručivanja hrane do kupovine različitih uređaja. Povećanjem broja aplikacija dostupnih korisniku, pojavila se potreba za korištenjem novijih tehnologija i pristupa za razvoj programskih proizvoda. Osim klasičnih alata za razvoj izvornih aplikacija, na tržištu su se pojavili programski alati za razvoj hibridnih i više platformskih aplikacija. Flutter je relativno nova platforma koja se koristi za razvoj mobilnih, web i desktop aplikacija. Platforma je stekla popularnost zbog svoje jednostavnosti, brzine i fleksibilnosti razvoja i održavanja aplikacija. Prethodno, ukoliko biste željeli pokriti cijelo tržište mobilnih aplikacija, morali biste razviti posebnu aplikaciju za svaki operacijski sustav. Ne samo da biste morali kopirati programski kod na drugu platformu, nego također prepisati i prilagoditi aplikaciju jedinstvenim nijansama kompleta za razvoj aplikacija (eng. Software development kit, kraće SDK). Korištenjem Flutter razvojnog okvira može se uz pomoć jedinstvene baze koda (eng. Single codebase) razviti aplikacija za više operacijskih sustava istovremeno, čime smanjujemo vrijeme i trošak potreban za razvoj programskog proizvoda.

Cilj ovog rada je prikazati prednosti modularnog pristupa u razvoju Flutter mobilnih aplikacija, fokusirajući se na to kako modularnost poboljšava organizaciju programskog koda, promiče ponovnu iskoristivost programskog koda te pojednostavljuje održavanje i učinkovitost razvoja programskog proizvoda.

Na početku rada objašnjene su razlike između vrsta mobilnih aplikacija te zašto je za izradu aplikacije odabran Flutter razvojni okvir. Zatim slijedi objašnjenje različitih pristupa arhitekturi programskih proizvoda te zašto je izuzetno bitna čista arhitektura. Objašnjena su i, na jednostavnim primjerima, pokazana SOLID načela dizajna razvoja programskih proizvoda te osnovnih uzoraka dizajna pisanih u Dart programskom jeziku. Nakon svih načela slijedi objašnjenje i prikaz najkorištenijih arhitektura prilikom razvoja Flutter aplikacija. Na kraju diplomskog rada nalazi se praktični primjer gdje je prikazana implementacija Business Logic Components (kraće BLoC) arhitekture prilikom razvoja aplikacije za upravljanje vlastitim budžetom.

2. METODE I TEHNIKE RADA

Prilikom izrade diplomskog rada korišteni su razni alati koji su olakšali razvoj mobilne aplikacije, planiranje i implementaciju baze podataka, izradu raznih dijagram i slično. Kako je navedeno u prethodnom poglavlju, za izradu mobilne aplikacije korišten je Flutter razvojni okvir. Flutter je razvojni okvir koji se na jednostavan način može povezati se raznim integriranim razvojnim okruženjima. Iako većina Flutter programera preferira Android Studio, za izradu aplikacije korišten je Microsoft Visual Studio Code uređivač teksta. Programski kod verzioniran je uz pomoć Git-a, GitHub-a i SourceTree platforme. Korisničko sučelje (eng. User Interface, kraće UI) i korisničko iskustvo (eng. User Experience, kraće UX) planirani su uz pomoć Figme. Baza podataka implementirana je uz pomoć DBeaver sustava za upravljanje bazama podataka (eng. Database Managment System, kraće DBMS) nad PostgreSQL relacijskom bazom podataka. Aplikacija je bazirana na mikro servisnoj arhitekturi te se za razvoj mikro servisa koristi Spring Framework pisan u Java programskom jeziku. Testiranje i simulacija REST API poziva omogućena je pomoću Postman platforme. Kod izrade diplomskog rada korištene su razne knjige, Internet članci te službena Flutter dokumentacija. Metodom analize i sinteze napravljen je pregled različitih arhitektura Flutter aplikacija, a deskriptivnom metodom opisani su svi teorijski pojmovi objašnjeni u ovom radu. Programski kod nalazi se na GitHub repozitoriju te mu se može pristupiti putem poveznice.

Popis korištene literature, vanjskih biblioteka te multimedijskog sadržaja može se pronaći na dnu rada. U nastavku slijedi kratak opis korištenih alata prilikom izrade praktičnog dijela diplomskog rada.

2.1. Microsoft Visual Studio Code

Visual Studio Code je moćan uređivač izvornog koda. Popularan je za izradu programskih proizvoda pisanih u različitim programskim jezicima. Alat je dostupan na Windows, macOS i Linux operacijskim sustavima. Postoji i inačica programa dostupna putem Interneta. Program dolazi s ugrađenom podrškom za JavaScript, TypeScript i Node.js. Bogati ekosustav nudi proširenja za druge programske jezike i okruženja kao što su C++, C#, Java, Python, PHP, GO, .NET i slično. Zbog bogatog ekosustava i laganog pristupa dodatnim ekstenzijama, alat je popularan među programerima. (Microsoft Visual Studio, 2023.)

2.2. Git, GitHub i SourceTree

Git je besplatan sustav za kontrolu verzija programskog koda, a služi za lakše praćenje razvoja programskog proizvoda. Dizajniran je za brzu i učinkovitu obradu od manjih do velikih projekata. Omogućuje stvaranje i odvajanje dijelova programskog koda na više neovisnih grana. Stvaranje, spajanje i brisanje grana traje nekoliko sekundi. (Git, 2023.)

GitHub je web stranica te servis temeljen na oblaku koji služi za pohranjivanje, upravljanje, praćenje i kontrolu programskog koda. Servis je besplatan te je pogodan za spremanje projekata otvorenog koda. (Kinsta, bez dat.)

SourceTree je besplatan Git klijent koji pojednostavljuje komunikaciju između lokalnih i mrežnih repozitorija. Omogućava klijentu lakšu vizualizaciju programskog koda te smanjuje mogućnost pogreške. (SourceTree, 2023.)

2.3. Figma

Figma je web aplikacija za dizajn korisničkog sučelja i korisničkog iskustva s naglaskom na suradnju u stvarnom vremenu. Unutar Figme postoje dva načina pregleda kod izrade dizajna korisničkog sučelja. Način kodiranja daje pristup kodu koji je u osnovi izgled i dojam projekta, a drugi prototip način omogućuje dizajnerima izradu, prezentaciju i modifikaciju digitalnog proizvoda. Figma je dosta popularan alat među dizajnerima diljem svijeta. (Gonzalez, R. , 2017.)

2.4. DBeaver i PostgreSQL

DBeaver je besplatan sustav za upravljanje relacijskim bazama podataka. Pogodan je alat za programere, administratore baze podataka, analitičare te sve korisnike koji na neki način moraju raditi s relacijskim bazama podataka. Besplatna verzija aplikacije podržava sve popularne relacijske baze podataka kao što su PostgreSQL, MySQL, Oracle, DB2 i slično. Plaćena verzija podržava i dodatne vrste baze podataka kao što su objektno orijentirane baze podataka i NoSQL baze podataka. (DBeaver, 2023.)

PostgreSQL je besplatan relacijski sustav baze podataka. Sustav je otvorenog koda sa snažnom zajednicom koja aktivno poboljšava pouzdanost, robusnost značajki i performansi. Postoje razne ekstenzije koje olakšavaju rad sa naprednijim značajkama relacijskih baza podataka. (PostgreSQL, 2023.)

DBeaver i PostgreSQL su alati koji su olakšali razvoj i implementaciju baze podataka tijekom izrade praktičnog dijela projekta.

2.5. Spring razvojni okvir i Postman

Spring razvojni okvir (eng. Spring Framework) čini programiranje u Javi bržim, lakšim i sigurnijim. Fokusom na brzinu, jednostavnost i produktivnost postao je najpopularniji Java razvojni okvir na svijetu. Pogodan je za izradu mikro servisa i Internet aplikacija. (Spring, 2023.)

Postman služi kao platforma za izradu i testiranje REST API web servisa. Pojednostavljuje svaki korak kod razvoja web servisa jer je fokusiran na suradnju tako da više ljudi u timu može stvarati, testirati i dokumentirati web servis. Omogućuje jednostavnu pohranu i upravljanje specifikacijama web servisa te se kolekcija mogućih metoda web servisa može jednostavno podijeliti s drugima. (Postman, 2023.)

3. VRSTE MOBILNIH APLIKACIJA

Mobilne aplikacije (u daljnjem tekstu, aplikacije) su programski proizvodi namijenjeni za rad na mobilnim i tablet uređajima. Postale su važan dio ljudskog života te su promijenile način na koji koristimo tehnologiju. Revolucionirale su način na koji komuniciramo, radimo, provodimo slobodno vrijeme. Svake godine izlazi sve više aplikacija na tržište, a dolaskom novih tehnologija kao što su umjetna inteligencija, lanac blokova i slično, čini se kako taj trend neće stati.

Rast potražnje aplikacija na tržištu doveo je do razvoja novih tehnologija te načina izrade aplikacija. Trenutno postoje tri vrste aplikacija, a to su: izvorne aplikacije, hibridne aplikacije i aplikacije za više platformi. Odabir između načina izrade aplikacija jedna je od najvažnijih odluka prilikom planiranja projekta razvoja aplikacije. Ova odluka ima ogromne implikacije na odabir dizajna, arhitekture, tehnologije te naposljetku broj korisnika koji mogu pristupiti aplikaciji.

U nastavku poglavlja slijedi detaljno objašnjenje svih vrsta mobilnih aplikacija klasificiranih po načinu izrade te njihove prednosti i nedostaci.

3.1. Izvorne aplikacije

Izvorne aplikacije namijenjene su za rad na određenim platformama kao što su Android i iOS. Programeri se oslanjaju na točno određeni programski jezik s kojim dostižu najoptimiziranije iskustvo prilikom korištenja aplikacije. Izvornim razvojem aplikacije omogućuje se pristup primarnim hardverskim elementima pametnog uređaja kao što su razni senzori, mikrofon, GPS i tako dalje.

Dva najpopularnija operacijska sustava mobilnih uređaja su Android i iOS. Kod razvoja izvornih aplikacija za Android najviše se koristi programski jezik Java, ali sve više programskih inženjera okreće se novijem programskom jeziku Kotlin. Kotlin je programski jezik koji smanjuje uobičajene greške te se lako integrira u postojeće aplikacije. Mnogi članci preporučuju Kotlin za izradu Android mobilnih aplikacija. Što se tiče iOS operacijskog sustava najviše se koristi Objective-C ili Swift.

Prema Singh S. (2023.) u sljedećoj tablici (vidi tablicu 1) nalazi se popis prednosti i nedostataka izvornih aplikacija:

Tablica 1: Prednosti i nedostaci izvornih aplikacija (Izvor: Singh S., 2023.)

PREDNOSTI I NEDOSTACI IZVORNIH APLIKACIJA	
PREDNOSTI	**NEDOSTACI**
• Brzina aplikacije • Izvan mrežna funkcionalnost • Intuitivnost i interaktivnost • Smanjen broj potencijalnih grešaka • Najbolje performanse • Povećana sigurnost	• Nemogućnost ponovnog korištenja programskog koda • Skupo održavanje • Specifičan programski jezik

Budući da izvorne aplikacije ne dolaze s kompleksnim programskim kodom osjeti se razlika u brzini izvršenja pojedine funkcionalnosti aplikacije. Zbog brzine razvoja i malih troškova razvoja, izvorne aplikacije su preporučeni izbor za mala poduzeća. Izvan mrežna funkcionalnost je jedna od najvećih prednosti mobilnih aplikacija. Pogodna je za korisnike koji imaju ograničen pristup internetu ili imaju ograničen podatkovni plan. Kako se izvorne aplikacije izrađuju za specifičan operativni sustav, u većini slučajeva programeri za implementaciju određene funkcionalnosti koriste neke geste koje su korisniku uređaja već poznate, te se time olakšava upotreba. Fokusom na određeni operacijski sustav smanjuje se broj potencijalnih grešaka te se ostvaruju najbolje performanse tijekom rada aplikacije. Najveći nedostatak izvornih aplikacija je nemogućnost ponovnog korištenja programskog koda. Kako bi se aplikacija plasirala na svim operacijskim sustavima, potrebno je za svaki operacijski sustav kreirati novi projekt što dovodi do povećanja troškova i trajanja vremena razvoja. (Singh S., 2023.)

3.2. Hibridne aplikacije

Kombinacijom izvornih i web aplikacija dobivamo hibridne aplikacije. U ovome slučaju nužno je poznavanje tehnologija za izradu web aplikacija. Programski kod pisan je u HTML, CSS i JavaScript/TypeScript te se uz pomoć alata kao što je Apache Cordova omogućuje pristup izvornim funkcionalnostima.

Hibridne aplikacije sastoje se od dvije komponente, pozadinski programski kod i izvorni preglednik za prikaz akcija i pozadine. Programski kod se piše samo jednom, a može se koristiti na više operacijskih sustava.

Prema Singh S. (2023.), prednosti i nedostaci hibridnih aplikacija nalaze se u idućoj tablici (vidi tablicu 2)

Tablica 2: Prednosti i nedostaci hibridnih aplikacija (Singh S., 2023.)

PREDNOSTI I NEDOSTACI HIBRIDNIH APLIKACIJA	
PREDNOSTI	**NEDOSTACI**
• Brzina razvoja • Jednostavno održavanje • Smanjeni troškovi • Bolje korisničko iskustvo	• Nemogućnost izvan mrežnog pristupa • Nekonzistentnost operacijskog sustava

Budući se aplikacija razvija za više platformi istovremeno, smanjuju se vrijeme i troškovi potrebni za plasiranje aplikacije na tržište. Olakšava se održavanje aplikacije jer se zahtjeva promjena samo na jednom mjestu programskog koda. Nisu potrebna dva tima kako bi se osigurala ispravnost aplikacije na svim platformama. Najveći nedostatak hibridnih aplikacija je nedostupnost u izvan mrežnom načinu rada. Korisnik aplikacije mora imati stabilnu internetsku vezu kako bi mogao pristupiti svim funkcionalnostima aplikacije. Također, neke funkcionalnosti mogu biti specifične za određeni operacijski sustav te bi se moglo proizvesti greške ukoliko se one pokrenu na drugom operacijskom sustavu. (Singh S., 2023.)

3.3. Aplikacije za više platformi

Mobilni razvoj aplikacija za više platformi omogućuje izradu jedne aplikacije koja glatko radi na nekoliko operativnih sustava. Ovakav razvoj omogućuje da se programski kod može dijeliti na više dijelova čime se pojedini dijelovi mogu prilagoditi funkcionalnostima konkretnog operacijskog sustava, dok se ostali dijelovi mogu iskoristiti za sve platforme. Time se povećava modularnost programskog koda čime se štedi na vremenu i troškovima razvoja. (Jetbrains, 2023.)

Najpopularniji razvojni okviri za razvoj aplikacija više platformi su:

- React Native
- Xamarian
- Flutter

Prema Singh S. (2023.), u sljedećoj tablici (vidi tablicu 3) nalaze se prednosti i nedostaci aplikacija razvijenih istovremeno za više platformi.

Tablica 3: Prednosti i nedostaci aplikacija za više platformi (Singh S., 2023.)

PREDNOSTI I NEDOSTACI APLIKACIJA ZA VIŠE PLATFORMI	
PREDNOSTI	**NEDOSTACI**
• Brzina razvoja • Jednostavno održavanje • Smanjeni troškovi • Modularnost programskog koda	• Kompliciran životni ciklus razvoja aplikacije • Komplicirana integracija

Mogućnost ponovnog korištenja programskog koda uz prilagođavanje pojedinih dijelova za konkretni operacijski sustav jedna je od najvećih prednosti aplikacija više platformi. Modularnost programskog koda olakšava testiranje, ispravljanje i nadogradnju programskog koda čime se osigurava vrhunska kvaliteta mobilnih aplikacija. Spajanjem zajedničkih funkcionalnosti smanjuje se vrijeme i trošak potreban za razvoj. Otežan je razvoj pojedinog dijela funkcionalnosti za konkretan operacijski sustav. Programer mora dobro poznavati razlike između operacijskih sustava kako bi osigurao besprijekoran rad aplikacije na svim platformama. (Singh S., 2023.)

Izradom izvorne aplikacije, programeri se automatski limitiraju na određeni operacijski sustav i određeni skup korisnika aplikacije. Ukoliko bi željeli pokriti cijelo tržište potrebno je za istu aplikaciju raspisati više projekata čime dolazi do repetitivnih zadataka i ponavljanja programskog koda. Nadalje, svaka promjena poslovne domene, također će zahtijevati određenu promjenu na svim projektima. Programeri se također susreću sa zadatkom prilagodbe arhitekture različitim operacijskim sustavima, veličinama zaslona i mogućnostima uređaja. Izradom aplikacija za više platformi postiže se određena razina modularnosti jer se programski kod za sve operacijske sustave nalazi unutar jednog projekta te olakšava daljnji razvoj i održavanje programskog proizvoda. Flutter je jedan od najpopularnijih razvojnih okvira za izradu takvih aplikacija. U nastavku diplomskog rada objašnjen je pojam arhitekture mobilnih aplikacije.

4. ARHITEKTURA MOBILNIH APLIKACIJA

Arhitektura mobilnih aplikacija obuhvaća načela dizajna i obrasce koji vode prema razvoju i organizaciji mobilnih aplikacija. Može se reći da je arhitektura zapravo kostur programa, a cjelokupni rad mobilne aplikacije određen je njezinom kvalitetom. Propuštanjem važnog elementa kao što je planiranje arhitekture mobilne aplikacije ugrožava se uspjeh projekta. Dobro osmišljena arhitektura osigurava nesmetan rad, poboljšava učinkovitost, olakšava održavanje te ubrzava daljnji razvoj aplikacije.

Vujević T. (2023.) uspoređuje arhitekture mobilnih aplikacija sa ljudskim tijelom. Navodi kako su svi dijelovi mobilne aplikacije raspoređeni na način koji omogućuje nesmetan rad pri čemu bi bilo koja promjena u strukturi nepovratno utjecala na njeno izvođenje. Također, navodi kako je jedna od glavnih prednosti arhitekture aplikacije modularnost čime se osigurava neovisno razvijanje, ažuriranje i testiranje komponenata.

Kako bi zadovoljili gore navedene potrebe, Android Developer (2023.) navodi sljedećih nekoliko specifičnih načela koje bi svaka arhitektura trebala slijediti.

Načelo razdvajanja odgovornosti (eng. Seperation of concerns) navodi kako bi klase trebale biti razdvojene na zasebne jedinice s minimalnim preklapanjem između ostalih jedinica. Razdvajanje odgovornosti postiže se korištenjem enkapsulacije i rasporeda u slojevima programskog proizvoda. Jedinice međusobno komuniciraju pomoću određenih sučelja. Zahvaljujući ovom principu, programski proizvod je podijeljen na više komponenata kojima se lakše može upravljati.

Načelo odvajanja podatkovnih modela od korisničkog sučelja (eng. Drive UI from data models) navodi kako je korisničko sučelje potrebno pokretati iz podatkovnih modela. Podatkovni modeli su zasebne jedinice aplikacije te nisu ovisni o elementima korisničkog sučelja, što znači da nisu vezani uz životni ciklus aplikacije, ali će i dalje biti uništeni kada operacijski sustav odluči ukloniti proces iz memorije.

Načelo jednog izvora podataka (eng. Single source of truth) navodi kako je za svaku vrstu podataka potrebno odrediti točno jedan izvor koji će biti vlasnik tih podataka. Izvoru je omogućeno modificiranje i brisanje podataka. Kako bi se to postiglo, izvor izlaže podatke pomoću nepromjenjivog tipa, a za izmjenu podataka izlaže funkcije ili prima događaje koje drugi izvori mogu pozvati. Ovo načelo osigurava centralizaciju podataka na jedno mjesto pri čemu osigurava sljedljivost promjene podataka.

Posljednje načelo, načelo jednosmjernog protoka podataka (eng. Unidirectional Data Flow) koristi se zajedno uz načelo razdvajanja odgovornosti, a navodi kako stanje podataka

putuje u jednom smjeru, a događaji koji uzrokuju promjenu podataka u suprotnom smjeru. Primjer, podaci aplikacije obično teku od izvora podataka do korisničkog sučelja, dok korisnički događaji, kao što su pritisci gumba, teku iz korisničkog sučelja do jedinstvenog izvora informacija gdje se podaci modificiraju i izlažu u nepromjenjivom tipu.

Načela navedena u ovome poglavlju izvedena su iz pojma čiste arhitekture koju je u svojoj knjizi „Clean Architecture: A Craftsman's Guide to Software Structure and Design“ objasnio autor Robert C. Martin poznatiji pod nadimkom „Ujak Bob“. U nastavku slijedi objašnjenje čiste arhitekture te kako ona može doprinijeti boljoj kvaliteti programskog proizvoda.

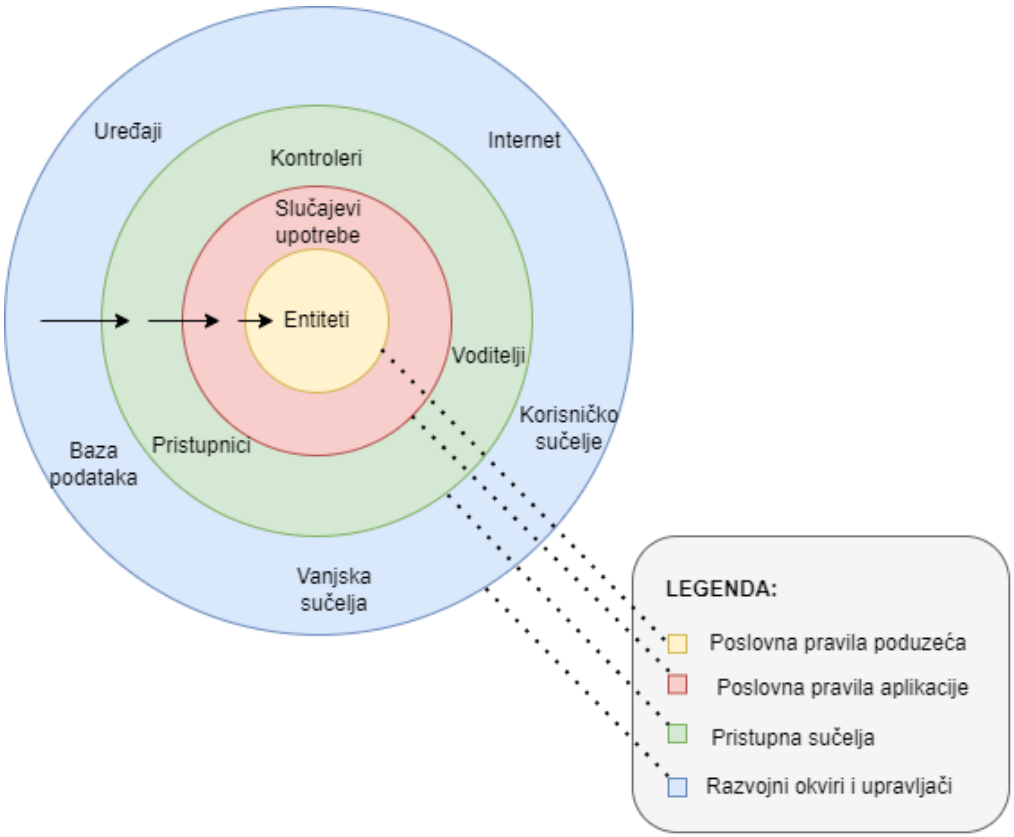
4.1. Čista arhitektura

Svaka tehnologija preporuča svoju najbolju arhitekturu sustava. Budući je svaka arhitektura samo preporuka, programer ili tim programera odlučuje o arhitekturi sustava koju će slijediti prilikom razvoja programskog proizvoda. Iako se svaka od preporučenih arhitektura razlikuje, one sve teže zajedničkom cilju, a to je odvajanje odgovornosti na određene slojeve.

U svojoj knjizi Martin R. C. (2018.) navodi kako bi svaka arhitektura trebala imati dva sloja. Jedan sloj će sadržavati poslovnu logiku, dok drugi sloj služi za definiranje korisničkih i sustavnih sučelja pri čemu oni tvore sustav koji zadovoljava iduća obilježja:

- Neovisna o razvojnom okviru (eng. Independent of frameworks) – arhitektura sustava ne smije ovisiti o raznim bibliotekama razvojnog okvira pri čemu razvojni okvir koristimo kao alat prilikom razvoja
- Provjerljiva (eng. Testable) – poslovna logika mora se moći testirati bez prisutnosti korisničkog sučelja, baze podataka i vanjskog servisa
- Neovisna o korisničkom sučelju (eng. Independent of the user interface) – korisničko sučelje mora se moći zamijeniti bez dodatnih promjena poslovne logike.
- Neovisna o bazi podataka (eng. Independent of the database) – poslovna logika ne smije ovisiti o odabranoj vrsti baze podataka
- Neovisna o vanjskim sustavima (eng. Independent of any external agency) – poslovna logika treba biti neovisna o vanjskim bibliotekama

Na sljedećoj slici (vidi sliku 1) nalazi se prikaz slojeva programskog proizvoda koje integrira gornjih 5 navedenih obilježja:



Slika 1: Čista arhitektura (Martin R. C. 2018.)

Pojam čiste arhitekture prema Martin R. C. (2018.) prikazuje 4 sloja programskog proizvoda, a to su: poslovna pravila poduzeća (eng. Enterprise Business Rules), poslovna pravila aplikacije (eng. Application Business Rules), pristupna sučelja (eng. Interface Adapters) te razvojni okviri i upravljači (eng. Frameworks and Drivers). Slojevi su prikazani na prethodnoj slici koncentričnim kružnicama. Autor navodi pravilo ovisnosti (eng. Dependency Rule) pri kojemu unutrašnje kružnice (unutrašnji slojevi) ne bi trebale znati o postojanju, a samim time niti ovisiti o načinu implementacije vanjskih kružnica (vanjskih slojeva).

Sloj poslovnih pravila poduzeća sadrži entitete. U ovome sloju dodaje se općenita logika koja je zajednička poslovnoj domeni. Na primjer, ovdje se može nalaziti programski kod koji služi za validaciju osobnog identifikacijskog broja jer se on može iskoristiti kod više različitih aplikacija.

Sloj poslovnih pravila aplikacije sadrži implementacije slučajeva upotrebe (eng. Use cases) sustava. Slučajevi upotrebe orkestriraju tok podataka prema entitetima. Promjene u ovome sloju ne bi trebale uzrokovati promjenu entiteta niti izgled korisničkog sučelja.

Sloj pristupnih sučelja sastoji se od broja transformatora koji konvertiraju podatak iz jednog formata, pogodnog za prikaz podataka, u format pogodan za slučajeve upotrebe i entitete.

Sloj razvojnih okvira i upravljača pruža podatke korisničkom sučelju ili bazi podataka. Ovaj sloj trebao bi sadržavati najmanje linija programskog koda.

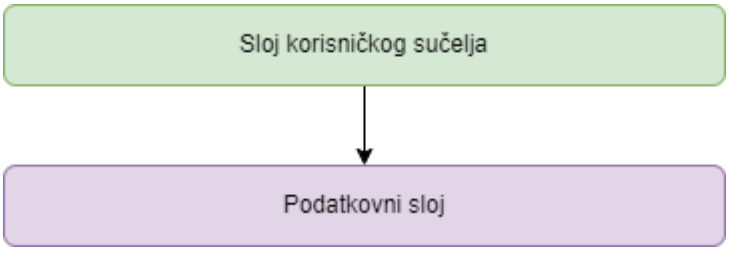
Nije nužno imati samo 4 sloja kao što je prikazano na slici 1. Jedino je bitno pratiti pravilo ovisnosti pri čemu su unutrašnji dijelovi više apstraktni, dok se vanjski dijelovi sastoje od konkretne implementacije.

Prateći navedena obilježja te odvajanjem programskog proizvoda u slojeve dobiva se sustav koji je jednostavan za testiranje i izmjenjiv. Svaki sloj može se zasebno testirati, a ukoliko dođe potreba za promjenom vanjskih elemenata kao što je baza podataka, nema potrebe za promjenom poslovne logike aplikacije jer ona ne ovisi o načinu implementacije baze podataka. Android Developer (2023.) je implementacijom navedenih načela kreirao generičku arhitekturu mobilnih aplikacija o kojoj se može više pročitati u sljedećem poglavlju.

4.2. Općenita arhitektura mobilnih aplikacija

Android Developer (2023.) navodi kako se ova arhitektura mobilnih aplikacija može primijeniti na širok spektar aplikacija i tehnologija kako bi se omogućilo skaliranje, poboljšala kvaliteta i robusnost te olakšalo testiranje. Međutim, ova arhitektura treba se koristiti kao smjernica te ju je nužno prilagoditi vlastitim zahtjevima.

Svaka aplikacija trebala bi sadržavati najmanje dva sloja, sloj korisničkog sučelja (eng. User Interface Layer) te podatkovni sloj (eng. Data Layer). Na slici 2 može se vidjeti generička arhitektura mobilnih aplikacija.



Slika 2: Općenita arhitektura mobilnih aplikacija (Android Developer, 2023.)

Uloga sloja korisničkog sučelja je prikazati podatke aplikacije na ekranskom sučelju. Ukoliko dođe do promjene podataka, bilo zbog interakcije korisnika ili vanjskog unosa, korisničko sučelje bi se trebalo ažurirati te prikazati nastale promjene. Ovaj sloj sastoji se od dva elemenata, elementi korisničkog sučelja (eng. User Interface Elements) i držači stanja (eng. State holders). Elementi korisničkog sučelja prikazuju podatke na zaslonu, a držači stanja su klase koje drže podatke te ih izlažu korisničkom sučelju. Međutim, podaci aplikacije koji se dobiju iz podatkovnog sloja su često u drugačijem formatu od informacija koje treba prikazati. Sloj korisničkog sučelja prima sve podatke potrebne za prikaz, pretvara ih u oblik pogodan za prikaz te ih izlaže korisniku.

Podatkovni sloj sadrži poslovnu logiku aplikacije. Poslovna logika sastoji se od pravila koja određuju kako aplikacija stvara, pohranjuje i ažurira podatke. Ovaj sloj sastoji se od dva elemenata, spremište podataka (eng. Repositories) te proizvoljan broj izvora podataka (eng. Dana Sources). Za svaku različitu vrstu podataka potrebno je kreirati novo spremište. Podatkovni sloj zadužen je za izlaganje podataka ostatku aplikacije te odvajanje izvora podataka od ostatka aplikacije.

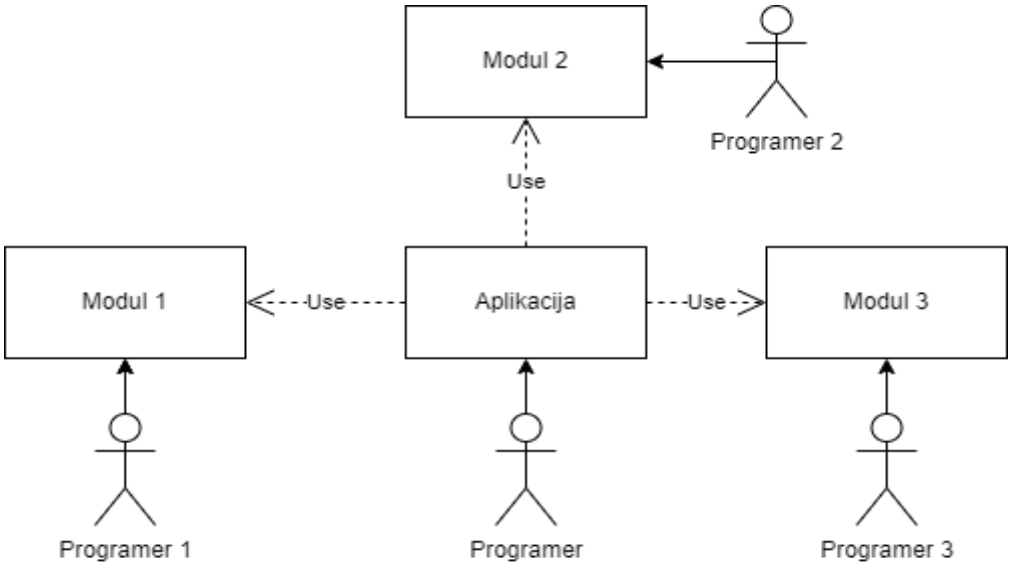
Programski proizvod se neprestano razvija te je izrada složenih aplikacija postala norma. Kako veličina i opseg projekta nastavlja rasti tijekom korištenja, tako raste i potreba za izgradnjom softverskih rješenja koja se mogu održavati i lagano izmijeniti. Kako bi se promjene mogle učiniti bez prevelike muke, potrebno je programski kod izdvojiti u manje, neovisne i kohezivne jedinice poznatije kao moduli. U nastavku je objašnjen pojam modularnosti.

4.3. Modularnost

Prema MacDonald M. (2023.), modularnost je opći koncept programiranja gdje programeri razdvajaju programske funkcije u neovisne dijelove. Svaki dio postaje građevni blok programskog proizvoda pri čemu svaki blok sadrži sve potrebne dijelove za izvođenje jednog aspekta funkcionalnosti. Pojedini građevni blok je zaseban te se lagano može testirati. Na kraju se blokovi spajaju u jedinstvenu cjelinu koja naposljetku tvori aplikaciju.

Najveća prednost modularnog programiranja je što rastavljanjem velikog softverskog programa na manje dijelove dovodi do programskog koda kojeg je naposljetku lakše razvijati i održavati. Olakšava čitanje programskog koda jer je svaki dio odvojen u zasebnu funkciju. Također, imamo manji rizik od zastoja aplikacije ukoliko dođe do greške u jednom od modula, potrebno je ispraviti samo zaseban modul jer je modul za sebe te ne ovisi o drugim modulima.

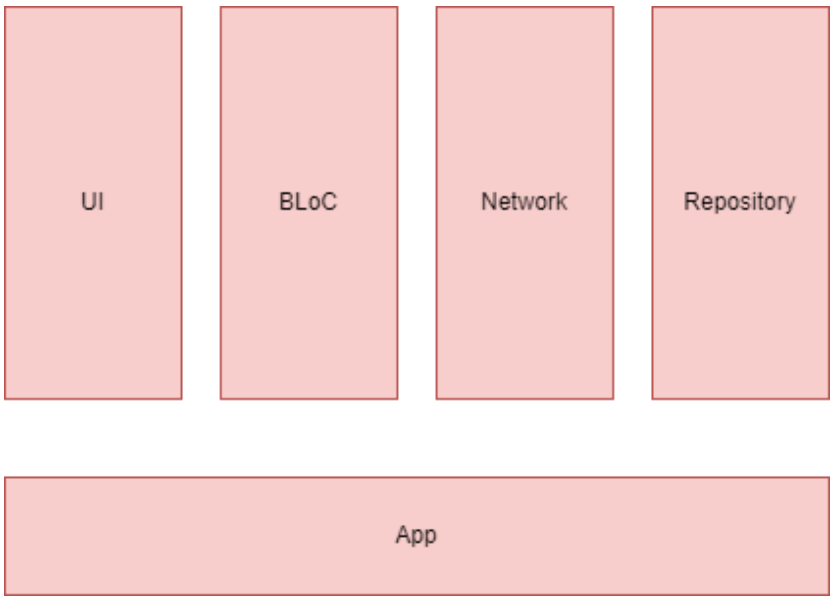
Svaki od modula može biti izdvojen u vlastiti repozitorij, što dovodi do lakšeg praćenja, ispravka i održavanja pojedinog modula. Na kraju se modul zapakira te se koristi unutar aplikacije. Na slici 3 možemo vidjeti dijagram modularnog pristupa programskog proizvoda pri čemu je svaki programer odgovoran za zaseban modul. Programer koji je zadužen za aplikaciju u ovom je slučaju odgovoran za integraciju svih pojedinih modula u aplikaciju.



Slika 3: Modularnost programskog proizvoda (vlastita izrada)

Prema Komari Fauzi Rifa R., (2020.), postoje različiti načini izvedbe modularnosti programskih proizvoda, izvedba modularnosti prema slojevima, izvedba modularnosti prema funkcionalnosti te izvedba modularnosti prema kombinaciji slojeva i funkcionalnosti.

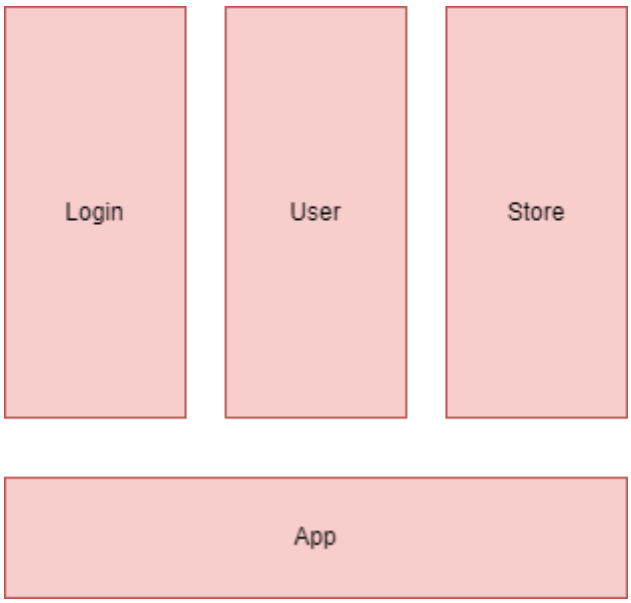
Izvedba modularnosti prema slojevima izdvaja programski kod prema sloju za koji je zadužen. Na slici 4 može se vidjeti ovakva izvedba modularnosti.



Slika 4: Izvedba modularnosti odvajanjem na slojeve (Komari Fauzi Rifa R., 2020.)

Ovdje imamo definiran poseban modul za svaki sloj aplikacije. Može se definirati zaseban modul koji sadrži sve poglede koji će se nalaziti na korisničkom sučelju.

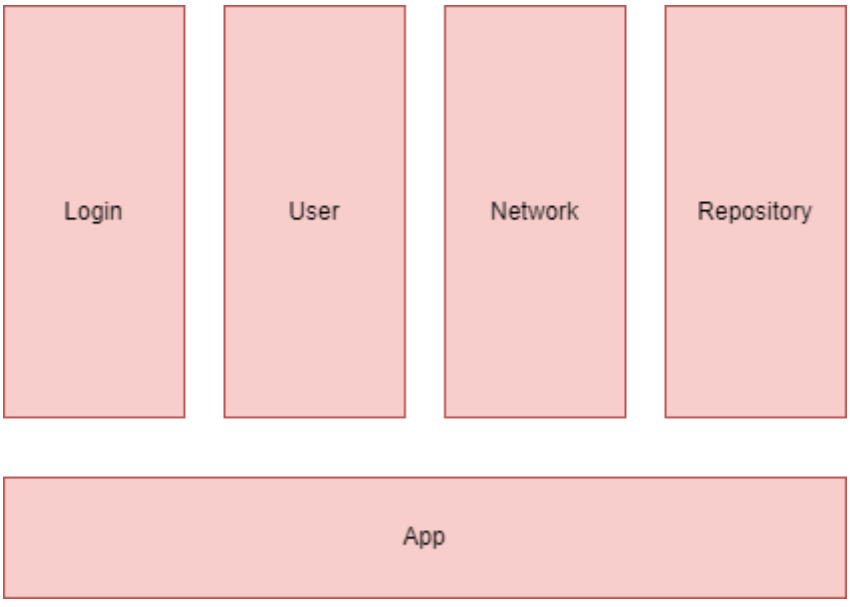
Drugi način izvedbe modularnosti je odvajanje funkcionalnosti na zasebne module (vidi sliku 5).



Slika 5: Izvedba modularnosti odvajanjem na funkcionalnosti (Komari Fauzi Rifa R., 2020.)

U ovom slučaju svaki modul predstavlja točno određenu funkcionalnost aplikacije. Primjer, Login modul sadrži sve potrebno za uspješnu prijavu korisnika unutar aplikacije. Ovakav način izvedbe korišten je u praktičnom primjeru.

Posljednja izvedba modularnosti je kombinacija odvajanja na slojeve te odvajanja na funkcionalnosti (vidi sliku 6).



Slika 6: Izvedba modularnosti kombinacijom slojeva i funkcionalnosti (Komari Fauzi Rifa R., 2020.)

U ovoj izvedbi postoje zasebni moduli na temelju slojeva te zasebni moduli na temelju funkcionalnosti. Umjesto da User modul ima sve vezano za upravljanje sa korisnicima aplikacije, on svoje REST API pozive ima definirane u posebnom modulu Network.

Dodatno, modularnost se postiže korištenjem načela inverzije ovisnosti. Načelo inverzije ovisnosti, objašnjeno u sklopu idućeg poglavlja, jedno je od temeljnih načela objektno orijentiranog programiranja s ciljem postizanja modularnosti promicanjem odvojene arhitekture te smanjenjem izravnih ovisnosti između različitih komponenti. U sklopu ovog načela komponente su izgrađene kao realizacije apstrakcije, a ne kao konkretne implementacije što rezultira fleksibilnijim i proširivim programskim proizvodom.

Kako bi se arhitektura sustava mogla primijeniti, nužno je da svi elementi pojedinog sloja budu ispravni. U nastavku rada prikazani su SOLID principi opisani u eseju „Design Principles and Design Patterns“ autora Robert C. Martin. Principi su namijenjeni za lakše razumijevanje, održavanje i proširenje programskog koda. Ukratko, praćenje ovih načela olakšava programerima izbjegavanje problema te izgradnju prilagodljivog, učinkovitog i agilnog programskog proizvoda.

5. SOLID NAČELA U DART PROGRAMSKOM JEZIKU

Kako bi uspješno implementirali željenu arhitekturu aplikacije, potrebno je dobro postaviti njezine temeljne dijelove. Glavni cilj SOLID principa je smanjiti ovisnost jednog područja programskog proizvoda o drugome. Smanjenje ovisnosti doprinosi izgradnji prilagodljivog, učinkovitog i agilnog programskog proizvoda. Uvođenjem ovih principa ne rješavaju se svi problemi dizajna arhitekture aplikacija, ali njihovo pravilno korištenje dovodi do održivog i lako čitljivog programskog koda.

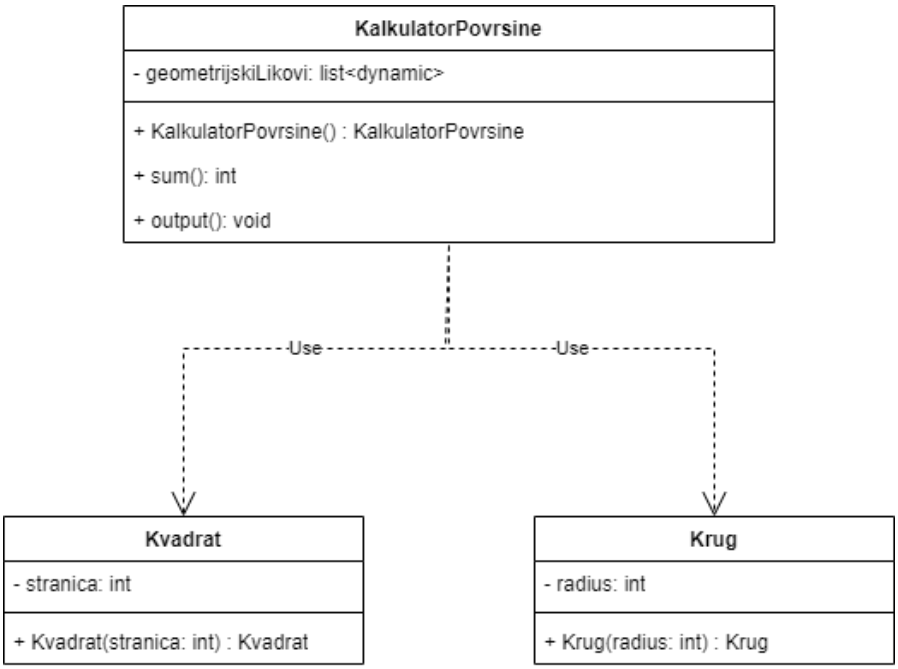
SOLID je akronim za načela dizajna programskog koda koje je potrebno slijediti, što znači da ukupno ima 5 načela dizajna. Prema Oloruntoba S. (2021), akronim se sastoji od:

- S – Načelo jedinstvene odgovornosti (eng. Singe-Responsibility Principle)
- O – Princip otvorenosti i zatvorenosti (eng. Open-Closed Principle)
- L – Načelo Liskove substitucije (eng. Liskov Substitution Principle)
- I – Načelo razdvajanja sučelja (eng. Interface Segregation Principle)
- D – Načelo inverzije ovisnosti (eng. Dependency Inversion Principle)

Budući bi svaki inženjer izrade programskog proizvoda trebao znati i koristiti ova načela, u nastavku slijedi implementacija SOLID načela na temelju primjera navedenog u članku Oloruntoba S. (2021.) u programskom jeziku Dart.

5.1. S – Načelo jedinstvene odgovornosti

Princip jedinstvene odgovornosti govori da klasa treba imati jedan i samo jedan razlog za promjenu, što znači da klasa treba obavljati samo jedan zadatak. Na sljedećem primjeru prikazano je nepoštivanje načela jedinstvene odgovornosti. Kreirane su klase prema UML dijagramu klasa iz slike 7.



Slika 7: UML dijagram kršenja načela jedinstvene odgovornosti (vlastita izrada)

UML dijagram se sastoji od tri klase: *Kvadrat*, *Krug* i *KalkulatorPovrsine*. Klasa *Kvadrat* se sastoji od atributa *stranica* cjelobrojnog tipa i konstruktora. Implementacija klase *Kvadrat* može se vidjeti na isječku programskog koda 1.

```
class Kvadrat {
  final int stranica;
  Kvadrat(this.stranica) {}
}
```

Isječak programskog koda 1: Implementacija modela Kvadrat (vlastita izrada)

Klasa *Krug* sastoji se od atributa *radius* cjelobrojnog tipa i konstruktora. Implementacija klase *Krug* može se vidjeti na isječku programskog koda 2.

```
class Krug {
  int radius;

  Krug(this.radius) {}
}
```

Isječak programskog koda 2: Implementacija modela Krug (vlastita izrada)

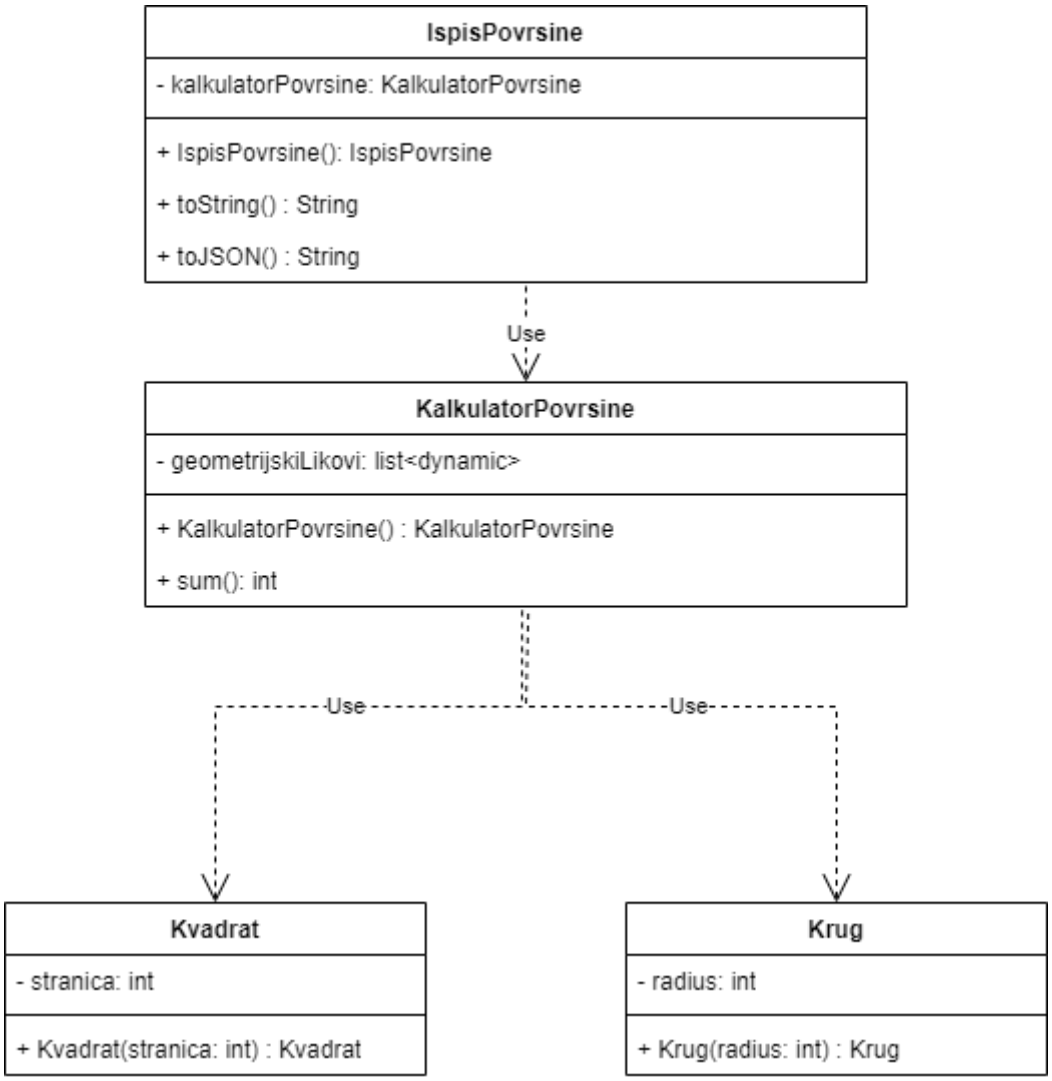
Posljednja klasa *KalkulatorPovrsine* sastoji se od atributa *geometrijskiLikovi* koji je lista sa dinamičnim tipovima podataka, dvije metode *sum()* i *output()* te konstruktorom. Metoda

sum() vraća zbroj svih površina geometrijskih likova koji se nalaze unutar liste, a metoda *output()* ispisuje sumu na konzoli. Implementacija klase *KalkulatorPovrsine* nalazi se na isječku programskog koda 3.

```
import "../models/Krug.dart";
import "../models/Kvadrat.dart";

class KalkulatorPovrsine {
  late List<dynamic> geometrijskiLikovi;

  KalkulatorPovrsine() {
    Krug krug1 = new Krug(2);
    Kvadrat kvadrat1 = new Kvadrat(3);
    Krug krug2 = new Krug(3);

    geometrijskiLikovi = List.from([krug1, kvadrat1, krug2]);
  }

  int sum() {
    int sum = 0;
    geometrijskiLikovi.forEach((element) {
      if (element is Krug) {
        sum = sum + (element.radius * 3);
      } else if (element is Kvadrat) {
        sum = sum + (element.stranica * element.stranica);
      } else {
        throw new Exception("Unrecognized type of class");
      }
    });

    return sum;
  }

  void output() {
    print("Izračun površina svih likova iznosi: " + sum().toString());
  }
}
```

Isječak programskog koda 3: Implementacija klase KalkulatorPovrsine (vlastita izrada)

U klasi *KalkulatorPovrsine* se krši princip jedinstvene odgovornosti. Osim što je klasa zadužena za izračun ukupne površine, zadužena je i za ispis sume. Ukoliko bi se pokazala potreba za dodatnim načinima ispisa, morali bi mijenjati klasu *KalkulatorPovrsine* sa dodatnim metodama. Kako bi se ispravila ova greška, potrebno je kreirati novu klasu koja će obavljati ispis sume na željeni način. Na sljedećoj slici (vidi sliku 8) nalazi se dijagram klasa koji sadrži ispravak greške te poštuje načelo jedinstvene odgovornosti.



Slika 8: UML dijagram korištenja načela jedinstvene odgovornosti (vlastita izrada)

Iz klase *KalkulatorPovrsine* maknuta je metoda *output()* koja je služila za ispis sume. Ova metoda je prebačena u novu klasu *IspisPovrsine* koja sadrži dvije metode*: toString()* i *toJSON().* Obje metode služe za ispis sume u željenom formatu. Implementacija novokreirane klase može se vidjeti na isječku programskog koda 4.

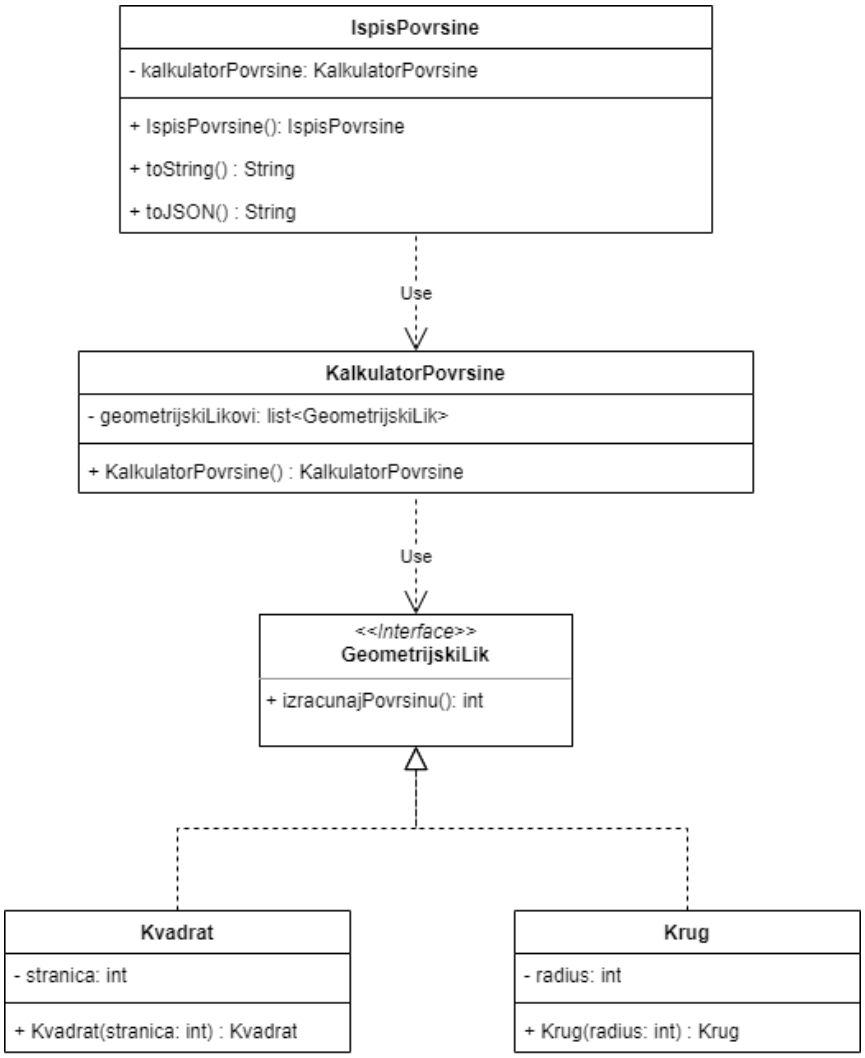
```dart
import '../utils/KalkulatorPovrsine.dart';

class IspisPovrsine {
  late KalkulatorPovrsine kalkulatorPovrsine;

  IspisPovrsine() {
    kalkulatorPovrsine = new KalkulatorPovrsine();
  }

  String toString() {
    return "String vrijednost: " + kalkulatorPovrsine.sum().toString();
  }

  String toJSON() {
    return "{ povrsina: ${kalkulatorPovrsine.sum()} }";
  }
}
```

Isječak programskog koda 4: Implementacija klase IspisPovrsine (vlastita izrada)

Ovim načinom izdvojena je odgovornost od ispisa sa klase *KalkulatorPovrsine* na zasebnu klasu *IspisPovrsine* s čime se više ne krši princip jedinstvene odgovornosti,

5.2. O – Princip otvorenosti i zatvorenosti

Princip otvorenosti i zatvorenosti navodi kako bi svaka klasa trebala biti lako proširiva bez modifikacije same klase. Ovaj princip također kršimo u prethodnom poglavlju unutar klase *KalkulatorPovrsine*. Unutar metode *sum()* provjerava se tip podatka te na temelju vraćenog tipa izračunava površina geometrijskog lika. Ukoliko bi se pokazala potreba za dodavanjem novog geometrijskog lika, bilo bi potrebno promijeniti implementaciju metode *sum()*. Na slici 9 nalazi se dijagram klasa koji poštuje princip otvorenosti i zatvorenosti.



Slika 9: UML dijagram principa otvorenosti i zatvorenosti (vlastita izrada)

Unutar aplikacije dodano je novo sučelje *GeometrijskiLik*. Svaka klasa koja će implementirati sučelje će morati sadržavati metodu *izracunajPovrsinu()* koja će vraćati cjelobrojni tip podatka. U Dart programskom jeziku ne postoji klasično sučelje, nego je potrebno kreirati apstraktnu klasu koja će služiti kao sučelje. Implementacija apstraktne klase *GeometrijskiLik* može se vidjeti na isječku programskog koda 5.

```
abstract class GeometrijskiLik {
  int izracunajPovrsinu() {
    return 0;
  }
}
```

Isječak programskog koda 5: Implementacija sučelja GeometrijskiLik (vlastita izrada)

Potrebno je modificirati klase *Krug* i *Kvadrat*. Ključnom riječi implements označujemo koja sve sučelja konkretna klasa mora implementirati. Klase koje implementiraju sučelje obvezne su pregaziti metode definirane unutar sučelja. Na isječku programskog koda 6, nalazi se modificirana klasa *Krug* koja će sada implementirati novokreirano sučelje.

```
import 'GeometrijskiLik.dart';

class Krug implements GeometrijskiLik {
  int radius;

  Krug(this.radius) {}

  @override
  int izracunajPovrsinu() {
    return this.radius * 3;
  }
}
```

Isječak programskog koda 6: Implementacija klase Krug (vlastita izrada)

Posljednje što je preostalo je modificirati klasu *KalkulatorPovrsine*. Lista *geometrijskiLikovi* više neće biti dinamičkog tipa, već će biti tipa *GeometrijskiLik*. Ovo nam osigurava da će svi podaci unutar liste imati metodu *izracunajPovrsinu()*. Sada se može zamijeniti metoda *sum()* prema isječku programskog koda 7.

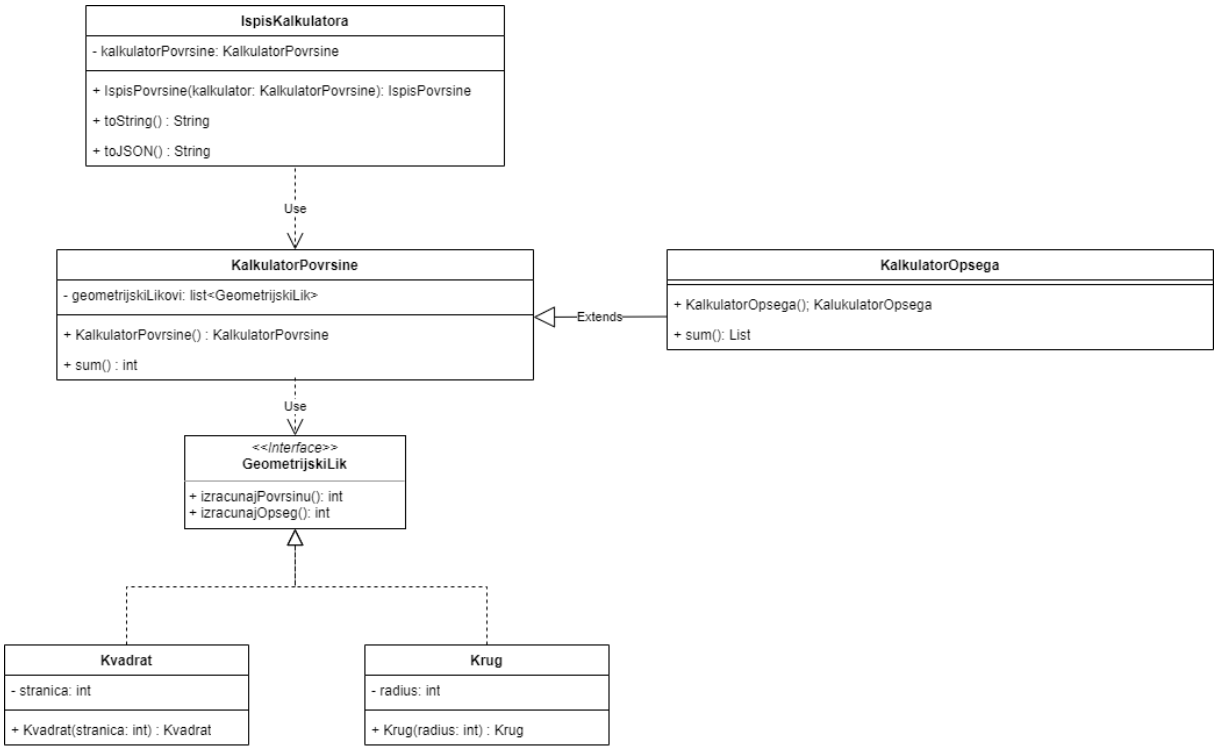
```dart
import '../models/GeometrijskiLik.dart';
import '../models/Krug.dart';
import '../models/Kvadrat.dart';

class KalkulatorPovrsine {
  late List<GeometrijskiLik> geometrijskiLikovi;

  KalkulatorPovrsine() {
    Krug krug1 = new Krug(2);
    Kvadrat kvadrat1 = new Kvadrat(3);
    Krug krug2 = new Krug(3);

    geometrijskiLikovi = List.from([krug1, kvadrat1, krug2]);
  }

  int sum() {
    int sum = 0;

    geometrijskiLikovi.forEach((element) {
      sum += element.izracunajPovrsinu();
    });

    return sum;
  }
}
```

Isječak programskog koda 7: Implementacija klase KalkulatorPovrsine (vlastita izrada)

Ovom promjenom, unutar aplikacije može se dodati novi geometrijski lik koji će implementirati sučelje te budući da *KalkulatorPovrsine* sada ovisi o sučelju neće biti potrebno mijenjati programski kod za izračun sume.

5.3. L – Načelo Liskove substitucije

Princip Liskove supstitucije navodi kako bi svaka podklasa trebala biti zamjenjiva za svoju roditeljsku klasu. Može se reći da je ovo očekivano ponašanje, jer kada se koristi nasljeđivanje pretpostavlja se da podklasa nasljeđuje sve što ima roditeljska klasa. Podklasa proširuje ponašanje roditeljske klase, ali ga nikad ne sužava. Na sljedećoj slici (vidi sliku 10) prikazan je dijagram klasa koji će služiti kao primjer za objašnjenje ovog načela.



Slika 10: UML dijagram načela Liskove substitucije (vlastita izrada)

Unutar aplikacije, dodana je nova klasa *KalkulatorOpsega* koja proširuje klasu *KalkulatorPovrsine* te nadjačava metodu *sum()*. Implementacija klase koja proširuje drugu klasu nalazi se na isječku programskog koda 8.

```
import './KalkulatorPovrsine.dart';

class KalkulatorOpsega extends KalkulatorPovrsine {
  KalkulatorOpsega() {

  }
  @override
  int sum() {
    int sum = 0;

    geometrijskiLikovi.forEach((element) {
      sum += element.izracunajOpseg();
    });

    return sum;
  }
}
```

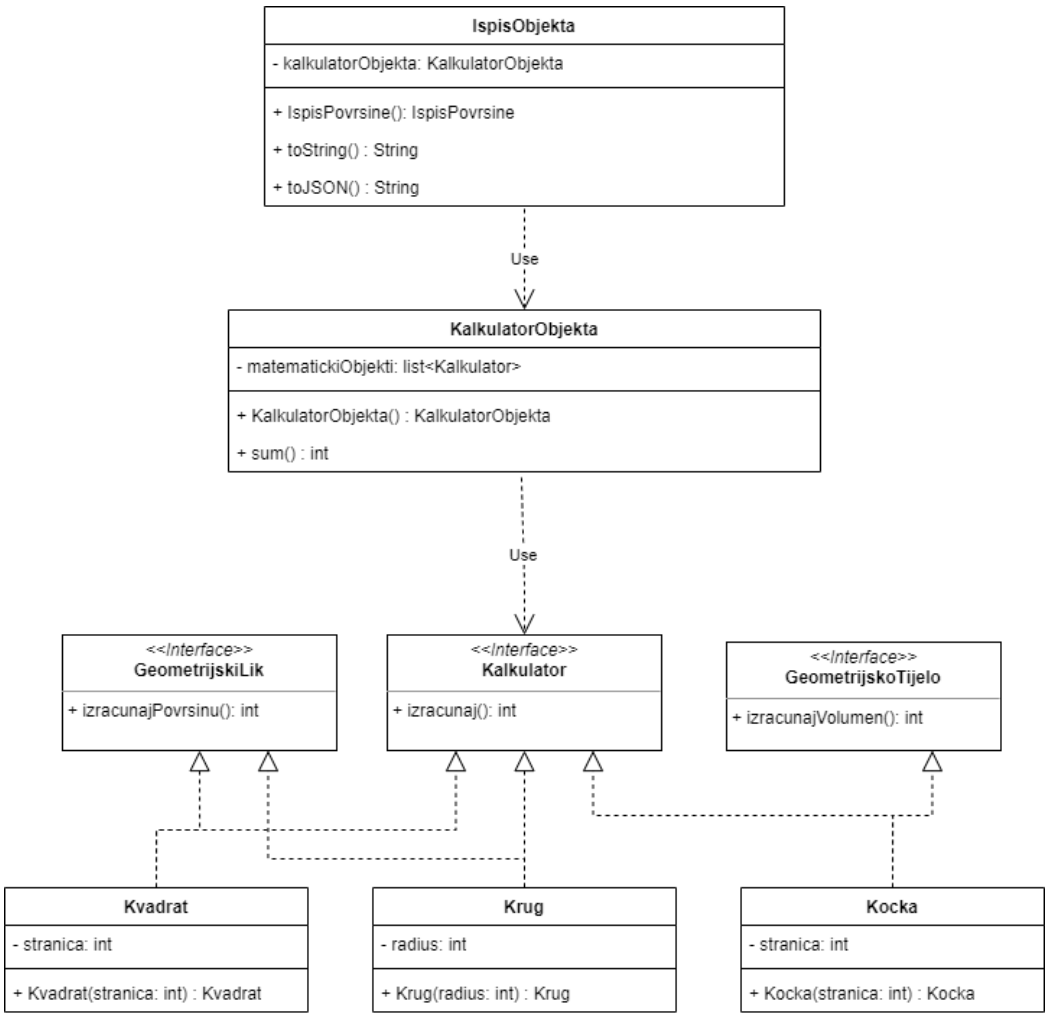
Isječak programskog koda 8: Implementacija klase KalkulatorOpsega (vlastita izrada)

Budući da proširena klasa sadrži sve atribute i metode sužene klase, kao *KalkulatorPovrsine* se može proslijediti *KalkulatorOpsega* te će sve raditi na isti način bez dodatnih promjena.

5.4. I – Načelo razdvajanja sučelja

Četvrti princip, princip odvajanja sučelja navodi kako niti jedna klasa ne bi trebala implementirati sučelje koje ne koristi ili ne bi trebali biti prisiljeni ovisiti o metodama koje ne koriste.

Ukoliko se doda geometrijsko tijelo, iz osnovne škole matematike zna se kako se ne može izračunati opseg geometrijskog tijela. Stoga nema smisla da novokreirana klasa koja predstavlja određeno geometrijsko tijelo implementira sučelje *GeometrijskiLik*. Implementacija načela razdvajanja sučelja napravljena je prema dijagramu klasa koji se nalazi na idućoj slici (vidi sliku 11)



Slika 11: UML dijagram načela segregacije sučelja (vlastita izrada)

Kreirana su dva nova sučelja. Sučelja *GeometrijskoTijelo* i *Kalkulator* sadrže jednu metodu. Implementacija novokreiranih sučelja nalazi se na isječku programskog koda 9.

```
abstract class GeometrijskoTijelo {
  int volumen() {
    return 0;
  }
}
abstract class Kalkulator {
  int izracunaj() {
    return 0;
  }
}
```

Isječak programskog koda 9: Implementacija sučelja GeometrijskoTijelo i Kalkulator (vlastita izrada)

Dodana je nova klasa *Kocka* koja implementira dva sučelja. Kako je bilo prikazano na prethodnim primjerima, klasa mora pregaziti sve metode koje su definirane unutar sučelja. Na isječku programskog koda 10 nalazi se implementacija novokreirane klase.

```
import 'interfaces/GeometrijskoTijelo.dart';
import 'interfaces/Kalkulator.dart';

class Kocka implements GeometrijskoTijelo, Kalkulator {
  int stranica;

  Kocka(this.stranica);

  @override
  int volumen() {
    return this.stranica * this.stranica * this.stranica;
  }

  @override
  int izracunaj() {
    return volumen();
  }
}
```

Isječak programskog koda 10: Implementacija klase Kocka (vlastita izrada)

Potrebno je izmijeniti već postojeće klase *Krug* i *Kvadrat*. Na obje klase dodana je implementacija sučelja *Kalkulator*. Nadjačana metoda sučelja *Kalkulator* vraćat će izračunatu

vrijednost dobivenu pozivom metode *izracunajPovrsinu()*. Na sljedećem isječku programskog koda (vidi isječak programskog koda 11) nalazi se implementacija klase *Krug*. Na isti način je modificirana klasa *Kvadrat*.

```
import 'interfaces/GeometrijskiLik.dart';
import 'interfaces/Kalkulator.dart';

class Krug implements GeometrijskiLik, Kalkulator {
  int radius;

  Krug(this.radius) {}

  @override
  int izracunajPovrsinu() {
    return this.radius * 3;
  }

  @override
  int izracunaj() {
    return izracunajPovrsinu();
  }
}
```

Isječak programskog koda 11: Dodano sučelje Kalkulator unutar klase Krug (vlastita izrada)

Posljednje što je potrebno promijeniti je klasa *KalkulatorPovrsine*. Umjesto da list sada prima klase koje implementiraju sučelje *GeometrijskiLik*, ona sada prima klase koje implementiraju sučelje *Kalkulator*. Time se osigurava da sve kreirane klase koje se nalaze u listi imaju implementiranu metodu *izracunaj()*. Budući da klasa više ne računa samo površinu, klasa *KalkulatorPovrsine* preimenovana je u *KalkulatorObjekta* te se implementacija nalazi na sljedećem isječku programskog koda.

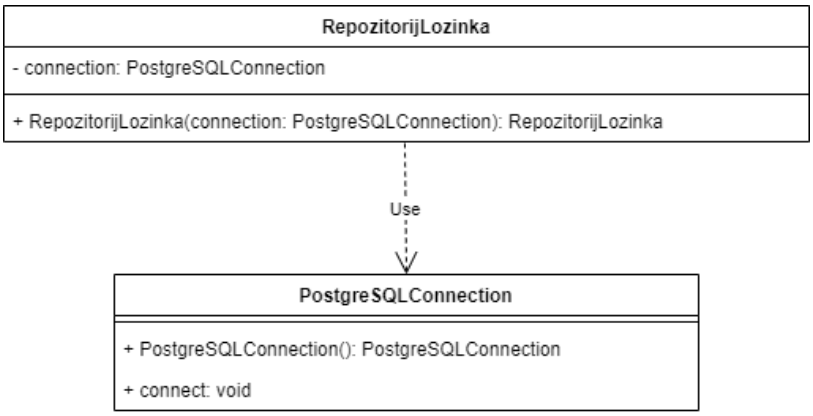
```dart
import '../models/Kocka.dart';
import '../models/Krug.dart';
import '../models/Kvadrat.dart';
import '../models/interfaces/Kalkulator.dart';

class KalkulatorObjekta {
  late List<Kalkulator> matematickiObjekti;

  KalkulatorObjekta() {
    Krug krug1 = new Krug(2);
    Kvadrat kvadrat1 = new Kvadrat(3);
    Krug krug2 = new Krug(3);
    Kocka kocka = new Kocka(2);

    matematickiObjekti = List.from([krug1, kvadrat1, krug2, kocka]);
  }

  int sum() {
    int sum = 0;

    matematickiObjekti.forEach((element) {
      sum += element.izracunaj();
    });

    return sum;
  }
}
```

Isječak programskog koda 12: Promijenjena klasa KalkulatorPovrsine u KalkulatorObjekta (vlastita izrada)

Ovisno o tome radi li se o geometrijskom tijelu ili geometrijskom liku, metoda *izračunaj()* vraća površinu ili volumen kao izračun.

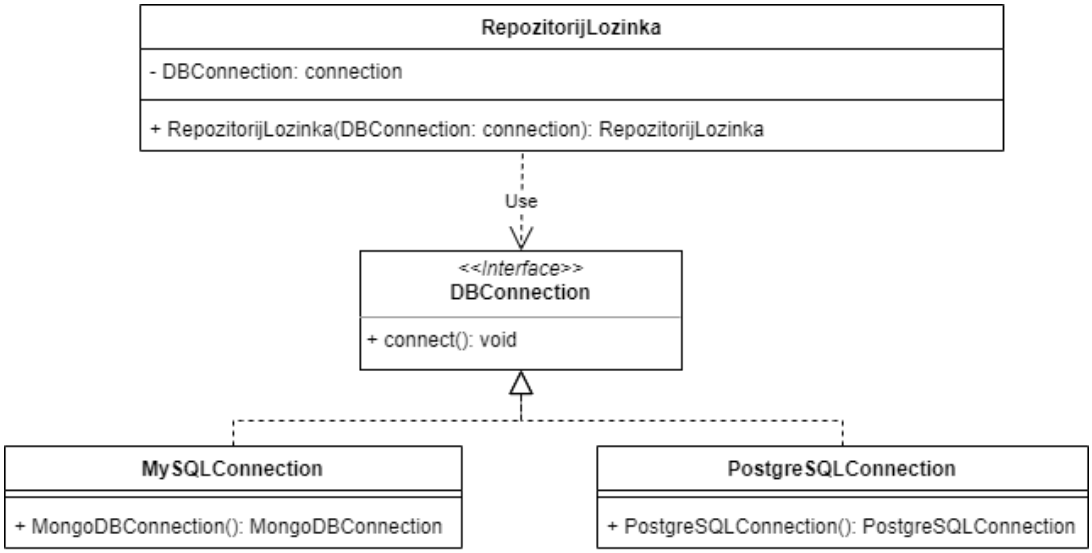
5.5. D – Načelo inverzije ovisnosti

Posljednje načelo navodi kako objekti moraju ovisiti o apstrakcijama, a ne o konkretnim klasama. Navodi da modul visoke razine ne smije ovisiti o modulu niske razine, već oni trebaju ovisiti o apstrakcijama. Na sljedećoj slici (vidi sliku 12) nalazi se dijagram klasa koji krši posljednje načelo.



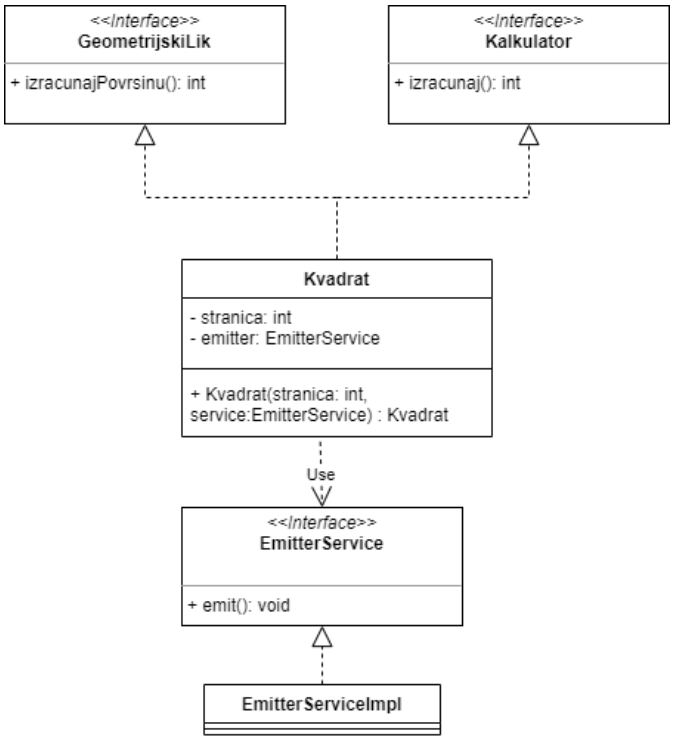
Slika 12: Dijagram klasa koji krši načelo inverzije ovisnosti (vlastita izrada)

Na prethodnom dijagramu klasa krši se ovo načelo, jer klasa *RepozitorijLozinka* ovisi o konkretnoj klasi *PostgreSQLConnection*. Kako bi se ovo izbjeglo, potrebno je kreirati novo sučelje koje će sadržavati sve metode kao i *PostgreSQLConnection* klasa te izmijeniti tip podataka *connection* iz konkretne klase u apstraktnu klasu. Dijagram klasa koje poštuje posljednje načelo nalazi se na slici 13.



Slika 13: Dijagram klasa koje poštuje načelo inverzije ovisnosti (vlastita izrada)

Koristeći primjer definiran u prethodnim načelima. Možemo kreirati novu klasu koja će služiti kao odašiljač vrijednosti. Kreirane su klase, prema UML dijagramu, prikazane na slici 14.



Slika 14: Dijagram klasa koji poštuje načelo inverzije ovisnosti prethodnog primjera (vlastita izrada)

Klasa *Kvadrat* u svojem konstruktoru prima dva parametra, a to su duljina stranice i klasu koja implementira sučelje *EmitterService*. Na ovaj način, klasa ne ovisi o konkretnoj klasi već njezinoj apstrakciji. Budući da svaka klasa koja implementira sučelje *EmitterService* mora imati metodu *emit().* Način izvedbe možemo odrediti kreiranjem novih konkretnih klasa koje implementiraju sučelje te ih proslijediti konstruktoru prilikom kreiranja *Kvadrat* objekta.

SOLID principi daju skup smjernica koje promiču razvoj robusnih, održivih i skalabilnih programskih proizvoda. Pridržavajući se ovih načela, razvijači programskih proizvoda grade temelje nužne za izradu modularne, fleksibilne, proširive i lako razumljive arhitekture mobilnih aplikacija. Implementacija ovih načela u početku se može činiti napornim i nepotrebnim utroškom vremena, ali redoviti rad s njima dovodi do prepoznavanja razlika između koda pisanog u skladu s načelima i koda koji nije u skladu s njima. Implementacija svih ovih načela koristiti će se kod prikaza najčešćih arhitektura mobilnih aplikacija pisanih u Flutter razvojnom okviru.

6. ARHITEKTURA FLUTTER APLIKACIJA

Kao što je bilo navedeno u prethodnim poglavljima, jedna od ključnih odluka prilikom razvoja mobilnih aplikacija je odabir pravilnog uzorka arhitekture. Uzorak arhitekture određuje organizaciju programskog koda te interakciju između pojedinih komponenata aplikacije. Odabir prave arhitekture olakšava izradu, testiranje i održavanje aplikacije. U nastavku ovog poglavlja ukratko je objašnjen Flutter razvojni okvir te su prikazani najčešći uzorci arhitekture mobilnih aplikacija razvijenih u Flutter razvojnom okviru.

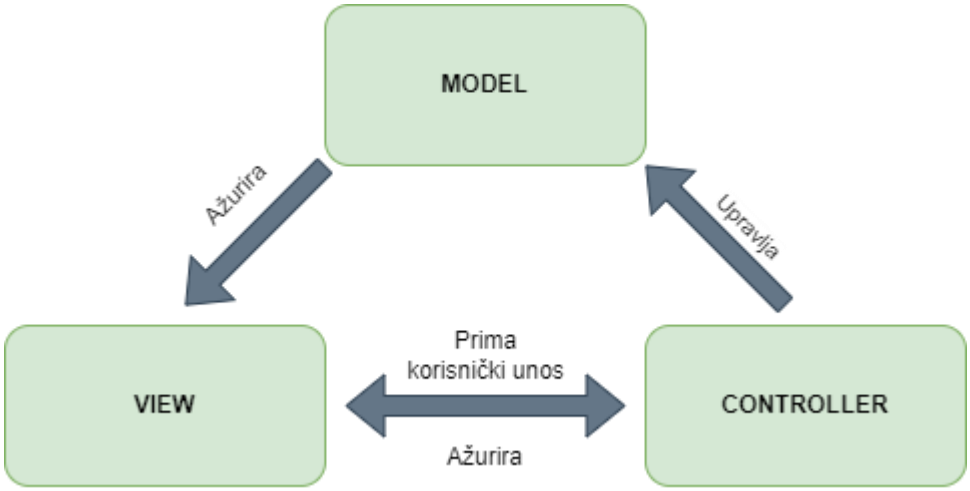
Razvoj mobilnih aplikacija putem Flutter razvojnog okvira razlikuje se od klasičnih razvojnih okvira kao što su Android SDK i iOS UIKit. Na primjer, Flutter je dovoljno brz da se umjesto mijenjanja, slobodno mogu rekonstruirati dijelovi korisničkog sučelja. U svojem članku „Start thinking declaritively“, na službenoj dokumentaciji Flutter razvojnog okvira (2023.), navodi kako je Flutter deklarativan razvojni okvir što znači da on konstruira korisničko sučelje na temelju trenutnog stanja aplikacije. Stanje aplikacije predstavlja podatke spremljene unutar pogleda koje se može promijeniti ovisno o funkciji. Ukoliko se stanje aplikacije promjeni, na primjer, korisnik okrene prekidač na zaslonu postavki, on zapravo mijenja stanje te pokreće rekonstruiranje korisničkog sučelja. Ovakav stil programiranja ima mnoge prednosti, kao što je jedinstveni programski kod koji opisuje kako bi korisničko sučelje trebalo izgledati za bilo koje stanje. Razdvajanjem odgovornosti programskog koda i odabirom tehnike upravljanja stanja određujemo arhitekturu Flutter mobilnih aplikacija. Postoje različite arhitekture Flutter aplikacija, a među najpopularnijima su: Model-View-Controller (kraće, MVC.), Business Logic Components (kraće, BLoC), Provider i Redux. Cijelu listu možete pronaći na službenim stranicama Flutter dokumentacije. U nastavku su objašnjeni najpopularniji uzorci arhitekture Flutter aplikacija.

6.1. MVC - Model – View – Controller

Model – View – Controller (kraće MVC), jedan je od najpopularnijih obrazaca arhitekture mobilnih aplikacija. Inicijalna ideja arhitekture je odvajanje programskog koda u odjeljke pri čemu svaki odjeljak ispunjava određenu svrhu. Jedan dio programskog koda sadrži podatke aplikacije, drugi dio upravlja načinom rada aplikacije, a treći dio izgrađuje korisničko sučelje te prikazuje podatke aplikacije krajnjem korisniku.

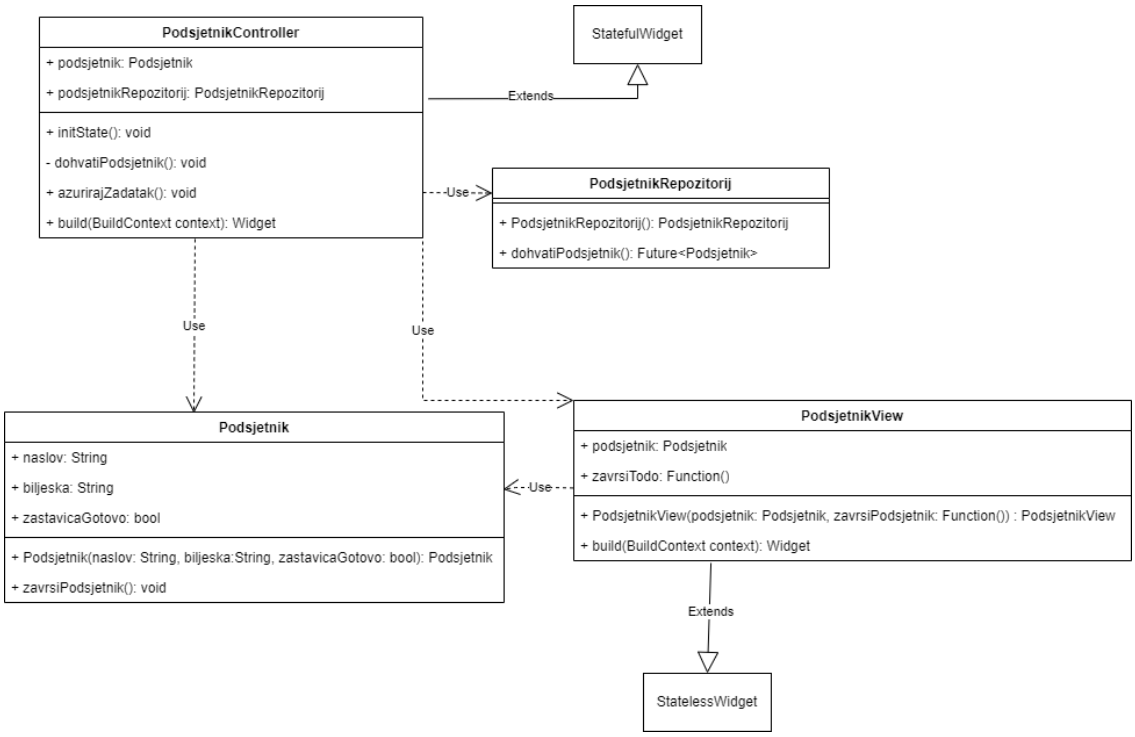
Prema Codecademy Team (bez datuma), MVC sastoji se od tri dijela (vidi sliku 15), a to su:

- Model – programski kod modela sastoji se od neobrađenih podataka, odnosno entiteta iz stvarnog svijeta
- View – programski kod koji se sastoji od svih funkcija koje izravno komuniciraju s korisnikom. Ovdje se definira izgled aplikacije te način kako krajnji korisnik vidi i komunicira s aplikacijom
- Kontroler – programski kod koji djeluje kao veza između Model-a i View-a. Kontroler prima korisnički unos i odlučuje što učiniti s njim



Slika 15: MVC model (Codeacademy, bez datuma)

Implementacija MVC arhitekture ostvarena je putem dijagrama klasa prikazanog na idućoj slici (slika 16). Aplikacija se sastoji od 4 klase, a to su: *Podsjetnik*, *PodsjetnikController*, *PodsjetnikView* i *PodsjetnikRepozitorij*. Na dijagramu klasa može se vidjeti kako *PodsjetnikController* komunicira sa klasom *Podsjetnik*, koja zapravo predstavlja model unutar ove arhitekture, te klasom *PodsjetnikView* koja prima korisnički unos te konstruira korisničko sučelje.



Slika 16: Dijagram klasa MVC arhitekture (vlastita izrada)

Klasa *Podsjetnik* se sastoji od tri atributa, jednog konstruktora i jedne metode. Implementaciju klase *Podsjetnik* može se vidjeti na idućem isječku programskog koda.

```
class Podsjetnik {
  String naslov;
  String biljeska;
  bool zastavicaGotovo;

  Podsjetnik({required this.naslov, required this.biljeska, required this.zastavicaGotovo});

  void zavrsiPodsjetnik() {
    zastavicaGotovo = !zastavicaGotovo;
  }
}
```

Isječak programskog koda 13: Implementacija klase Podsjetnik kod MVC arhitekture (vlastita izrada)

Klasa *PodsjetnikView* sastoji se od dva atributa, jednog konstruktora i jedne metode. Budući da klasa *PodsjetnikView* nasljeđuje sve atribute klase *StatelessWidget*, mora implementirati metodu *build()* koja vraća *Widget*. Prema Flutter službenoj dokumentaciji (bez datuma), *Widget* je temeljni konstrukt aplikacije koji opisuje izgled korisničkog sučelja s obzirom na njihovu trenutnu konfiguraciju i stanje. Kada se stanje promjeni, *Widget* ponovno gradi korisničko sučelje. Izgled korisničkog sučelja unutar *PodsjetnikView build()* metode napravljen je kombinacijom *Widgeta* koji dolazi sa Flutter razvojnim okvirom. *Widgeti Text, SizedBox, Column, Row* su prethodno definirani *Widgeti* koji se nalaze unutar material.dart biblioteke. Na službenoj dokumentaciji može se pronaći popis svih prethodno definiranih *Widgeta*. Implementacija *PodsjetnikView* klase nalazi se idućem isječku programskog koda.

```
import 'package:flutter/material.dart';
import 'package:mvc/models/Podsjetnik.dart';

class PodsjetnikView extends StatelessWidget {
  final Podsjetnik? todo;
  final Function() zavrsiTodo;

  const PodsjetnikView({super.key, required this.todo, required this.zavrsiTodo});

  @override
  Widget build(BuildContext context) {
    return Card(
      child: Padding(
        padding: EdgeInsets.all(8.0),
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: todo != null ? [
           Text(todo!.zastavicaGotovo ? "Riješeno" : "U tijeku", style: TextStyle(fontSize: 12.0),),
            SizedBox(height: 8.0),
                Text(todo!.naslov, style: TextStyle(fontSize: 18.0, fontWeight: FontWeight.bold),),
            SizedBox(height: 8.0),
            Text(todo!.biljeska, style: TextStyle(fontSize: 14.0)),
            SizedBox(height: 8.0,),
              TextButton(onPressed: zavrsiTodo, child: Text("Azuriraj ToDo"))
          ] : [
            CircularProgressIndicator()
          ]
        ),
      ),
    );
  }
}
```

Isječak programskog koda 14: Implementacija PodsjetnikView klase unutar MVC arhitekture (vlastita izrada)

PodsjetnikRepozitorij klasa sastoji se od jedne metode *fetchTodo()*. Ova metoda će simulirati poziv resursa za dohvat podataka o *Podsjetniku*. Metoda će nakon 5 sekundi vratiti *Podsjetnik* sa inicijalnim vrijednostima.

```
import '../models/Podsjetnik.dart';

class PodsjetnikRepository {

  Future<Podsjetnik> fetchTodo() async {
    await Future.delayed(Duration(seconds: 5));
      return Future.value(Podsjetnik(naslov: "Naslov 1", biljeska: "Biljeska 1", zastavicaGotovo: false));
  }
}
```

Isječak programskog koda 15: Implementacija PodsjetnikRepozitorij klase unutar MVC arhitekture (vlastita izrada)

Posljednja klasa *PodsjetnikController* povezuje model i pogled. Kada korisnik pokrene metodu *azurirajZadatak(),* ažurira se atribut unutar modela te pokreće rekonstrukciju korisničkog sučelja, odnosno *PodsjetnikView Widgeta*.

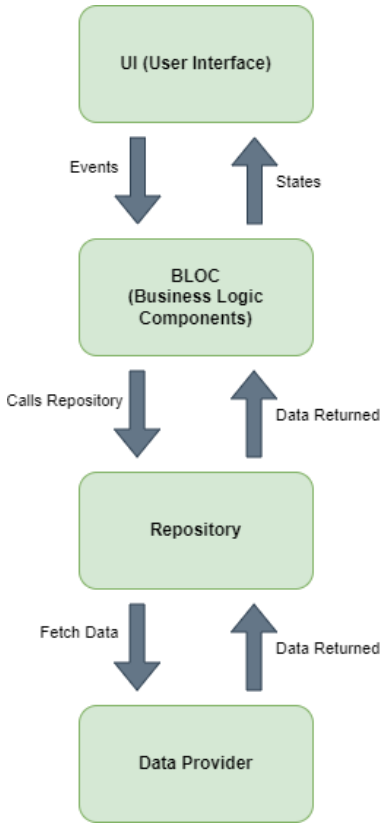
```
import 'package:flutter/material.dart';
import 'package:mvc/models/Podsjetnik.dart';
import 'package:mvc/repositories/PodsjetnikRepository.dart';

import '../views/PodsjetnikView.dart';

class PodsjetnikController extends StatefulWidget{
  PodsjetnikController({
    super.key
  });

  @override
  _PodsjetnikState createState() => _PodsjetnikState();
}

class _PodsjetnikState extends State<PodsjetnikController> {
  Podsjetnik? podsjetnik;
  PodsjetnikRepository toDoRepository = PodsjetnikRepository();

  @override
  void initState() {
    _fetchTodo();
  }

  void _fetchTodo() async {
    Podsjetnik noviPodsjetnik = await toDoRepository.fetchTodo();
    setState(() {
      podsjetnik = noviPodsjetnik;
    });
  }

  void azurirajZadatak() {
    setState(() {
      podsjetnik?.zavrsiPodsjetnik();
    });
  }

  @override
  Widget build(BuildContext context) {
        return PodsjetnikView(todo: podsjetnik, zavrsiTodo:
azurirajZadatak);
```

Isječak programskog koda 16: Implementacija klase PodsjetnikController unutar MVC arhitekture (vlastita izrada)

Implementacijom MVC arhitekture dobiva se dobro definirana struktura dizajna programskog proizvoda. Odvajanjem odgovornosti između upravljanja podacima, korisničkog sučelja i programske logike, promiče modularnost, mogućnost održavanja, prilagodbu korisničkog sučelja i slično. Jedna je od najpopularnijih obrazaca arhitekture prilikom izrade jednostavnih aplikacija koje nemaju puno složenosti. Međutim kada aplikacija raste u veličini i složenosti, održavanje i ažuriranje ovakvih aplikacije postaje veoma izazovno.

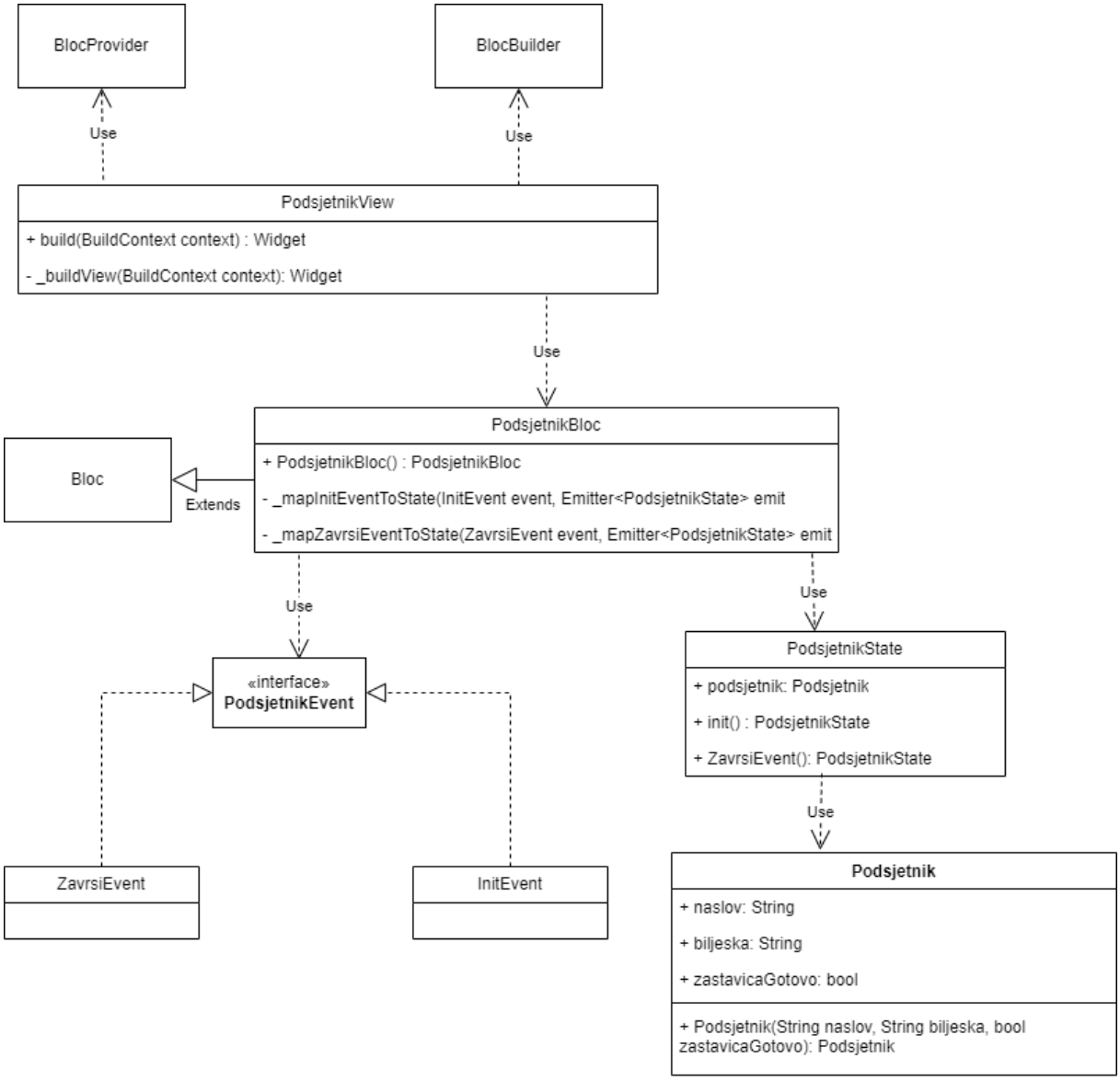
6.2. BLoC – Business Logic Components

Business Logic Components (kraće, BLoC) je jedan od najpopularnijih arhitektonskih obrazaca koji se koristi u razvoju Flutter mobilnih aplikacija. Ovaj obrazac arhitekture je najkorišteniji te ga Flutter programerska zajednica najviše preporučuje prilikom odabira arhitekture. Varijacija je MVC arhitekture s naglaskom na reaktivno programiranje. Prema Nnamdi C. (bez datuma), BLoC koristi koncept tokova događaja, odnosno kada se stvori novi tok događaja, pretplatnici se pretplaćuju na događaj te se obavještavaju o novim podacima kada se emitiraju u tok događaja. Tok događaja kreira se unutar BLoC-a, pogledi (*Widgeti*) se pretplaćuju na događaje kreirane od BLoC te se obavještavaju ukoliko dođe do promjene podatka. Arhitektura BLoC obrazca može se vidjeti na slici 17.



Slika 17: BLoC model (prema Suri S., 2018.)

Primjer implementacije BLoC arhitekture prikazan je uz pomoć dijagrama klasa prikazanog na slici 18.



Slika 18: UML dijagram klasa BLoC arhitekture (vlastita izrada)

Implementacija će koristiti „bloc“ biblioteku dostupnu na službenoj stranici javno dostupnih biblioteka. Dokumentacija „bloc“ biblioteke dostupna je putem poveznice. Dijagram klasa sastoji se od 6 klasa i jednog sučelja. Prvo je kreiran *Podsjetnik* model koji se sastoji od tri atributa i jednog konstruktora. Implementacija modela *Podsjetnik* vidljiva je na idućem isječku programskog koda.

```
class Podsjetnik {
  String naslov;
  String biljeska;
  bool zastavicaGotovo;

   Podsjetnik({
    required this.naslov,
    required this.biljeska,
    required this.zastavicaGotovo
  });
}
```

Isječak programskog koda 17: Implementacija modela Podsjetnik unutar BLoC arhitekture (vlastita izrada)

Klasa *PodsjetnikState* sadrži sva moguća stanja u kojem se model *Podsjetnik* može naći. U ovom slučaju model sadrži inicijalno stanje implementirano pomoću *init()* metode te dodatno moguće stanje koje invertira *zasatavicaGotovo* na novo stanje. Obje metode vraćaju *PodsjetnikState* kao tip podatka.

```
import 'package:redux/model/Podsjetnik.dart';

class PodsjetnikState {
  late Podsjetnik podsjetnik;

  PodsjetnikState init() {
    return PodsjetnikState()..podsjetnik = Podsjetnik(naslov: "Naslov
1", biljeska: "Biljeska 1", zastavicaGotovo: false);
  }

  PodsjetnikState ZavrsiEvent() {
    podsjetnik.zastavicaGotovo = !podsjetnik.zastavicaGotovo;
    return PodsjetnikState()..podsjetnik = podsjetnik;
  }
}
```

Isječak programskog koda 18: Implementacija klase PodsjetnikState unutar BLoC arhitekture (vlastita izrada)

Sljedeće je potrebno definirati događaje koji će uzrokovati promjenu stanja. Potrebno je definirati sučelje *PodsjetnikEvent* koje će svi događaji implementirati. Za potrebe prikaza ove arhitekture kreirana su dva događaja *InitEvent* i *ZavrsiEvent*. Ovdje se definiraju samo događaji koji se mogu inicirati za vrijeme trajanja rada aplikacije, odnosno ovdje se ne definira implementacija događaja. Radi lakšeg prikaza sučelje *PodsjetnikEvent* te klase *InitEvent* i *ZavrsiEvent* su spojeni u jedan isječak programskog koda.

```
abstract class PodsjetnikEvent {}

class InitEvent extends PodsjetnikEvent {}

class ZavrsiEvent extends PodsjetnikEvent {}
```

Isječak programskog koda 19: Implementacija događaja unutar BLoC arhitekture (vlastita izrada)

Unutar *PodsjetnikBloc* klase spajaju se događaji sa potrebnim stanjem. Prilikom iniciranja *InitEvent* definiranog na prethodnom isječku programskog koda, emitira se stanje definirano unutar *PodsjetnikState.init()* metode. Prilikom iniciranja *ZavrsiEvent*, emitira se novo stanje definirano metodom *ZavrsiEvent()*. Implementacija *PodsjetnikBloc* klase prikazana je na idućem isječku programskog koda.

```
import 'dart:async';

import 'package:flutter_bloc/flutter_bloc.dart';

import 'PodsjetnikEvent.dart';
import 'PodsjetnikState.dart';

class PodsjetnikBloc extends Bloc<PodsjetnikEvent, PodsjetnikState> {
  PodsjetnikBloc() : super(PodsjetnikState().init()) {
    on<InitEvent>(_mapInitEventToState);
    on<ZavrsiEvent>(_mapZavrsiEventToState);
  }

  void _mapInitEventToState(InitEvent event, Emitter<PodsjetnikState>
emit) async {
      emit(state);
  }

      void _mapZavrsiEventToState(ZavrsiEvent event,
Emitter<PodsjetnikState> emit) async {
    emit(state.ZavrsiEvent());
  }
}
```

Isječak programskog koda 20: Implementacija klase PodsjetnikBloc unutar BLoC arhitekture (vlastita izrada)

Posljednja implementirana klasa je *PodsjetnikView*. Ova klasa sadrži definiciju korisničkog sučelja te na temelju promjena stanja definiranog pomoću bloc varijable unutar *build(BuildContext context)* metode mijenja izgled korisničkog sučelja. *Widget* koji koristi definirani bloc potrebno je omotati sa *BlocProvider* widgetom koji omogućuje pristup trenutnom stanju *Podsjetnik* modela. Implementacija pogleda *PodsjetnikView* dostupna je na idućem isječku programskog koda.

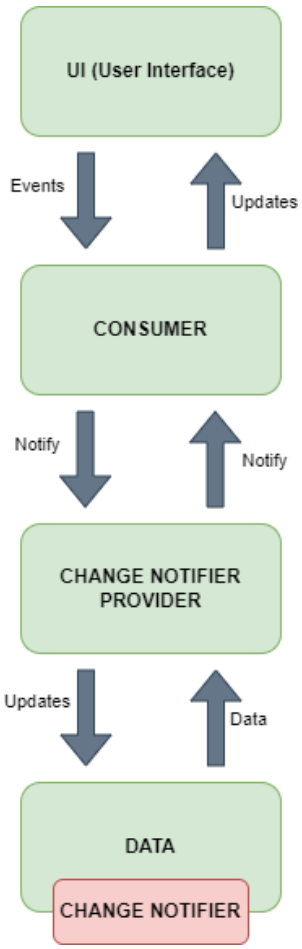
```
import 'package:flutter/material.dart';
import 'package:flutter_bloc/flutter_bloc.dart';
import 'package:redux/bloc/PodsjetnikState.dart';

import '../bloc/PodsjetnikBloc.dart';
import '../bloc/PodsjetnikEvent.dart';

class PodsjetnikView extends StatelessWidget {
  const PodsjetnikView({super.key});

  @override
  Widget build(BuildContext context) {
    return BlocProvider(
                        create:      (BuildContext      context)      =>
PodsjetnikBloc()..add(InitEvent()),
      child: Builder(builder: (context) => _buildView(context)),
    );
  }

  Widget _buildView(BuildContext context) {
    final bloc = BlocProvider.of<PodsjetnikBloc>(context);

    return Scaffold(
      appBar: AppBar(title: Text("Bloc arhitektura")),
      body: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        crossAxisAlignment: CrossAxisAlignment.center,
        children: [
          BlocBuilder<PodsjetnikBloc, PodsjetnikState>(
            builder: (context, state) {
                return Text(state.podsjetnik.zastavicaGotovo ? "T" :
"F");
            }),

          TextButton(child: Text("Press"), onPressed: () {
            bloc.add(ZavrsiEvent());
          },)
        ],
      ),
    );
  }
```

Isječak programskog koda 21: Implementacija klase PodsjetnikView unutar BLoC arhitekture (vlastita izrada)

BLoC obrazac popularan je arhitektonski obrazac u razvoju reaktivnih i skalabilnih Flutter aplikacija. Pruža jasno odvajanje odgovornosti te olakšava upravljanje stanjem aplikacije, olakšavajući stvaranje složenih baza kodova koje je moguće održavati. Njegova popularnost i široka prihvaćenost unutar Flutter zajednice osigurava njegovu učinkovitost u izgradnji modernih mobilnih i web aplikacija bogatih značajkama.

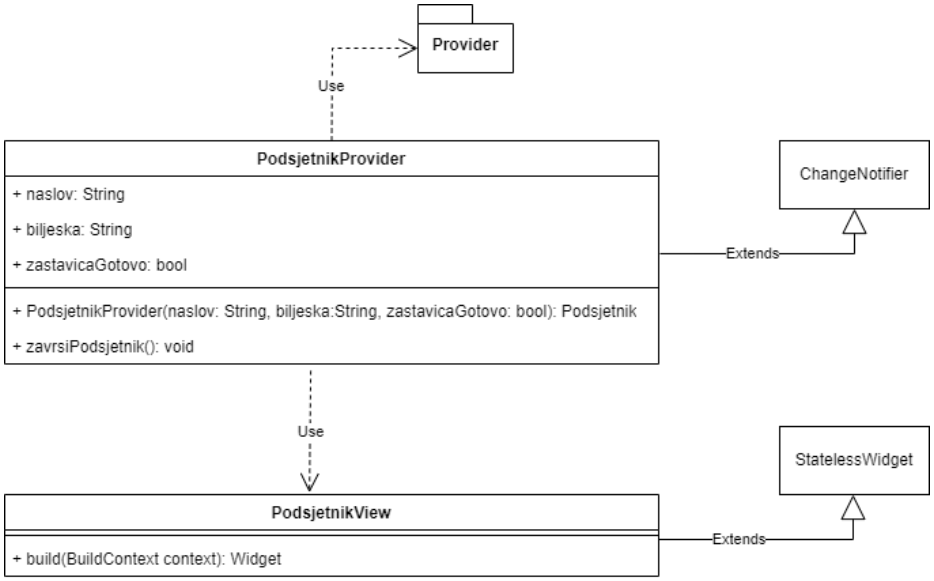
6.3. Provider

Prema Flutter Guru (2022.), Provider je biblioteka kreirana od Remi Rousselet sa ciljem što čišćeg rukovanja stanjem. Pogledi prate promjene u stanju te ažuriraju korisničko sučelje po primitku obavijesti. Ovim načinom se umjesto ponovnog rekonstruiranja cijelog korisničkog sučelja, rekonstruira samo određeni *Widget*, čime se smanjuje količina posla te se omogućuje brži rad aplikacije.



Slika 19: Provider model (Lapp T. J., 2019.)

Primjer Provider arhitekture Flutter aplikacija prikazati će se uz pomoć dijagrama klasa prikazanog na slici 20. Kako bi se implementirala Provider arhitektura, potrebno je preuzeti javno dostupnu biblioteku kojoj se može pristupiti pomoću poveznice.



Slika 20: Dijagram klasa Provider arhitekture (vlastita izrada)

Provider arhitektura implementirana je uz pomoć dvije klase, *PodsjetnikProvider* i *PodsjetnikView*. *PodsjetnikProvider* nasljeđuje klasu *ChangeNotifier* te sadrži jednu metodu *zavrsiPodsjetnik()* koja u svojoj definiciji sadrži metodu *notifyListeners()*. Svi *Widgeti* koji će sadržavati *Provider* tipa *PodsjetnikProvider* će prilikom poziva funkcije *zavrsiPodsjetnik()* ažurirati svoje stanje te rekonstruirati pogled, odnosno korisničko sučelje. Implementaciju klase *PodsjetnikProvider* može se vidjeti na idućem isječku programskog koda.

```
import 'package:flutter/material.dart';

class PodsjetnikProvider extends ChangeNotifier {
  String naslov = "";
  String biljeska = "";
  bool zastavicaGotovo = false;

    PodsjetnikProvider(String naslov, String biljeska, bool
zastavicaGotovo){
    this.naslov = naslov;
    this.biljeska = biljeska;
    this.zastavicaGotovo = zastavicaGotovo;
  }

  void zavrsiPodsjetnik() {
    this.zastavicaGotovo = !this.zastavicaGotovo;
    notifyListeners();
  }
}
```

Isječak programskog koda 22: Implementacija PodsjetnikProvider unutar Provider arhitekture (vlastita izrada)

Klasa *PodsjetnikView* pretplaćuje se na *PodsjetnikProvider* uz pomoć ključne riječi *Provider.of()*. Implementacija klase *PodsjetnikView* može se vidjeti na idućem isječku programskog koda.

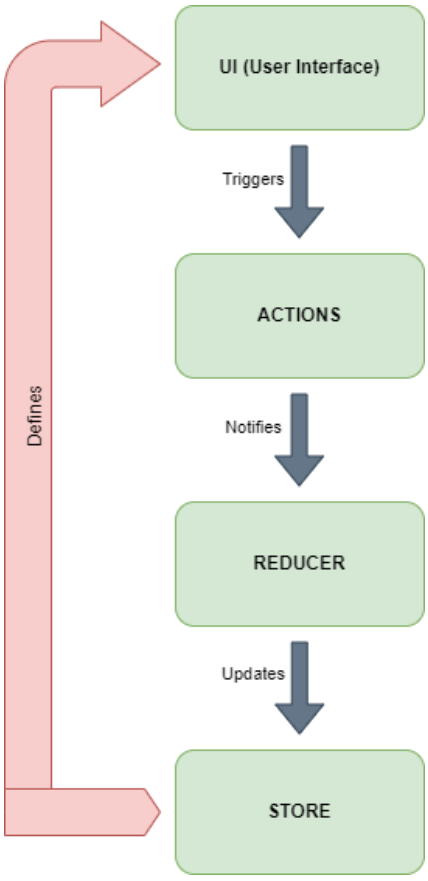
```
import 'package:flutter/material.dart';
import 'package:mvvm/model/Podsjetnik.dart';
import 'package:provider/provider.dart';

class PodsjetnikView extends StatelessWidget {
  const PodsjetnikView({super.key});

  @override
  Widget build(BuildContext context) {
    final todo = Provider.of<PodsjetnikProvider>(context);
    return Card(
      child: Padding(
        padding: EdgeInsets.all(8.0),
        child: Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: [
                Text(todo.zastavicaGotovo ? "Riješeno" : "U tijeku",
style: const TextStyle(fontSize: 12.0),),
              SizedBox(height: 8.0),
                 Text(todo.naslov, style: TextStyle(fontSize: 18.0,
fontWeight: FontWeight.bold, color: Colors.cyan)),
              SizedBox(height: 8.0),
              Text(todo.biljeska, style:  TextStyle(fontSize: 14.0)),
              SizedBox(height: 8.0,),
              TextButton(onPressed: () {
                todo.zavrsiPodsjetnik();
                }, child: Text("Azuriraj podsjetnik"))
            ]
        ),
      ),
    );
  }
}
```

Isječak programskog koda 23: Implementacija PodsjetnikView unutar Provider arhitekture (vlastita izrada)

Provider je jednostavna biblioteka za razumijevanje te ne koristi mnogo programskog koda. Koristi koncepte koji su primjenjivi u svakom drugom pristupu. Postao je jedan od glavnih rješenja za upravljanje stanjem Flutter aplikacija nudeći jednostavan, ali snažan pristup upravljanju stanjem aplikacije. Biblioteka Provider pruža programerima pouzdano i učinkovito rješenje za izgradnju proširivih i održivih Flutter aplikacija.

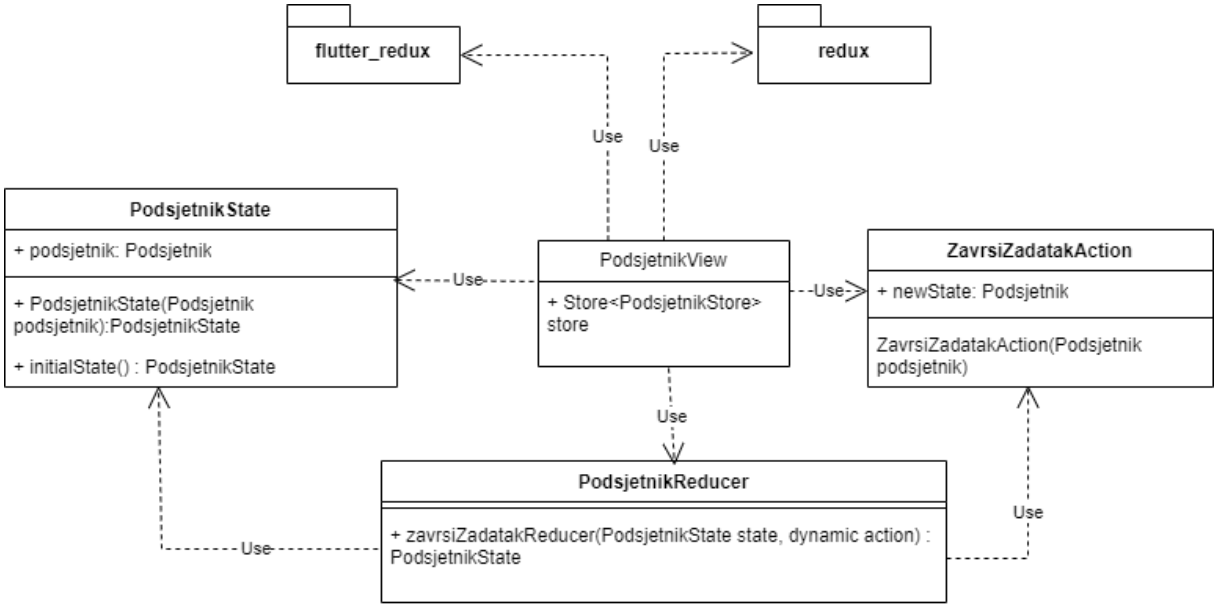
6.4. Redux

Prema Joleyemi D. (2021.), Redux je biblioteka za kreiranje arhitekture Flutter aplikacije koja uspješno upravlja stanjem te distribuira podatke kroz widgete. Umjesto da se stanje aplikacije prosljeđuje s widgeta na widget, s Reduxom se stanja aplikacija strukturiraju u centralizirano spremište naziva „Store“. Podacima u centraliziranoj lokaciji mogu pristupiti svi widgeti koji zahtijevaju određeni podatak. Arhitektura Redux aplikacija može se vidjeti na slici 21. Reducer predstavljaju funkcije koje primaju akciju i prethodno stanje podataka te ovisno o akciji vraćaju ažurirani podatak (time mijenjaju stanje aplikacije). Akcije su funkcije koje čitaju ili ažuriraju stanje aplikacije.



Slika 21: Redux model (vlastita izrada)

Implementacija Redux arhitekture ostvarena je pomoću dijagrama klasa (vidi sliku 22) i javno dostupne biblioteke kojoj se može pristupiti pomoću poveznice.



Slika 22: UML dijagram klasa Redux arhitekture (vlastita izrada)

Redux arhitektura sastoji se od najmanje 4 klase, *PodsjetnikState*, *PodsjetnikView*, *ZavrsiZadatakAction* i *PodsjetnikReducer*. Prvo je potrebno definirati model koji se izrađuje na sličan način kao u prethodnim primjerima. Implementacija modela *Podsjetnik* može se vidjeti na sljedećem isječku programskog koda.

```
class Podsjetnik {
  String naslov;
  String biljeska;
  bool zastavicaGotovo;

  Podsjetnik.initialState() : naslov = "Naslov1", biljeska="Biljeska1",
zastavicaGotovo=false;
}
```

Isječak programskog koda 24: Implementacija klase Podsjetnik unutar Redux arhitekture (vlastita izrada)

Na sličan način kao i kod BLoC arhitekture potrebno je kreirati stanje modela. *PodsjetnikState* sadržavat će trenutno stanje aplikacije. Implementacija klase *PodsjetnikState* vidljiva je na idućem isječku programskog koda.

```
import '../model/Podsjetnik.dart';

class PodsjetnikState {
  late Podsjetnik podsjetnik;

  PodsjetnikState(Podsjetnik podsjetnik){
    this.podsjetnik = podsjetnik;
  }

  PodsjetnikState.initialState() {
    this.podsjetnik = Podsjetnik.initialState();
  }
}
```

Isječak programskog koda 25: Implementacija klase PodsjetnikState unutar Redux arhitekture (vlastita izrada)

Sljedeće je potrebno definirati akcije. Akcije su zapravo metode sa kojima vraćamo novo stanje modela u obliku nove instance. Implementacija klase *ZavrsiZadatakAction* dostupna je na idućem isječku programskog koda.

```
import 'package:bloc/model/Podsjetnik.dart';

class ZavrsiZadatakAction {
  final Podsjetnik newState;

  ZavrsiZadatakAction(this.newState);
}
```

Isječak programskog koda 26: Implementacija klase ZavrsiZadatakAction unutar Redux arhitekture (vlastita izrada)

Može se vidjeti kako akcije ne ažuriraju trenutno stanje modela. Kako bismo provjerili vrstu akcije te pravilno ažurirali stanje potrebno je kreirati reducere. Reduceri na temelju pozvanih akcija vraćaju novo stanje modela. Implementaciju *PodsjetnikReducer* može se vidjeti na idućem isječku programskog koda.

```
import 'package:bloc/actions/PodsjetnikActions.dart';
import 'package:bloc/states/AppState.dart';

PodsjetnikState zavrsiZadatakReducer(PodsjetnikState state, dynamic
action) {
  if (action is ZavrsiZadatakAction) {
   action.newState.zastavicaGotovo = !action.newState.zastavicaGotovo;
    return PodsjetnikState(action.newState);
  } else {
    return state;
  }
}
```

Isječak programskog koda 27: Implementacija Reducer unutar Redux arhitekture (vlastita izrada)

Posljednje je potrebno kreirati globalno spremište. Globalno spremište sadrži trenutno stanje modela unutar aplikacije. Spremište se definira na početnom *Widgetu* aplikacije te uz pomoć *StoreProvider* pruža stanja svim *Widgetima* definiranih poslije početnog *Widgeta*. Implementacija klase *PodsjetnikView* dostupna je na idućem isječku programskog koda.

```
import 'package:bloc/actions/PodsjetnikActions.dart';
import 'package:flutter/material.dart';
import 'package:flutter_redux/flutter_redux.dart';
import 'package:redux/redux.dart';

import '../model/Podsjetnik.dart';
import '../reducers/PodsjetnikReducer.dart';
import '../states/AppState.dart';

class PodsjetnikView extends StatelessWidget {
  const PodsjetnikView({super.key});

  @override
  Widget build(BuildContext context) {
    final Store<PodsjetnikState> _store = Store<PodsjetnikState>(
    zavrsiZadatakReducer,
    initialState: PodsjetnikState.initialState()
  );

    return StoreProvider(store: _store,
      child: Scaffold(
        body: StoreConnector<PodsjetnikState, Podsjetnik>(
          converter: (store) => store.state.podsjetnik,
          builder: (context, Podsjetnik state) =>
          Column(
            mainAxisAlignment: MainAxisAlignment.center,
            crossAxisAlignment: CrossAxisAlignment.center,
            children: [
              Text(state.zastavicaGotovo ? "T" : "F"),
              TextButton(onPressed: () {
                StoreProvider.of<PodsjetnikState>(context).dispatch(Z
avrsiZadatakAction(state));
              }, child: Text("Azuriraj stanje"))
            ],
          ),
        ),
      ),
    );
  }
}
```

Isječak programskog koda 28: Implementacija klase PodsjetnikView unutar Redux arhitekture (vlastita izrada)

Redux je snažno rješenje za upravljanje stanjem kod Flutter aplikacija. Dobro definirana arhitektura te jednosmjeran protok podataka osiguravaju robusno i proširivo rješenje za rukovanje složenim interakcijama stanja unutar Flutter aplikacija.

Svaka od prikazanih arhitektura Flutter aplikacija postiže modularnost na način odvajanja zadatka aplikacija u različite komponente. Svaka komponenta odgovorna je za specifične zadatke. Odvajanjem zadataka na zasebne komponente omogućuje ponovnu upotrebu koda, mogućnost održavanja, skalabilnost i kolaborativni razvoj.

Odabir pravilne arhitekture igra ključnu ulogu kod razvoja modularnih aplikacija. Obrasci navedeni u ovom poglavlju pružaju strukturirani pristup organiziranju programskog koda, upravljanja stanjem te rješavanjem složenosti razvoja Flutter aplikacija. Svaka od prikazanih arhitektura u prethodnim primjerima promiče razdvajanje odgovornosti, pisanje modularnog programskog koda te povećanje iskoristivosti programskog rješenja. Odvajanjem poslovne logike od korisničkog sučelja olakšava suradnju između članova tima, jer različiti programeri mogu raditi na različitim dijelovima aplikacije neovisno. Uz pisanje čistog koda te odabir pravilne arhitekture Flutter aplikacije, dodatno se pojedini dijelovi aplikacije mogu razdvojiti na module. Pojedini modul može sadržavati vlastiti git repozitorij čime se olakšava razvoj i održavanje aplikacije. U nastavku je prikazano planiranje i implementacija modularne arhitekture koristeći BLoC uzorak na primjeru aplikacije za upravljanje osobnog proračuna pisane u Flutter razvojnom okviru.

7. IMPLEMENTACIJA MODULARNOSTI FLUTTER APLIKACIJA

U teorijskom dijelu diplomskog rada bile su opisane različite arhitekture kod izrade Flutter aplikacija. Budući da je tema ovog rada dizajn i implementacija modularne arhitekture mobilnih aplikacija pisanih u Flutteru, u praktičnom dijelu biti će prikazana aplikacija za upravljanje vlastitim budžetom odvojena na tri zasebna modula koristeći BLoC arhitekturu.

Svrha ovog praktičnog dijela je prikazati način izrade jedinstvene aplikacije odvojene na više zasebnih modula prateći jednu od arhitektura navedenih u prethodnim poglavljima. Cilj ove aplikacije je unošenje različitih vrsta transakcija s ciljem praćenja potrošnje korisnika. Primjerice, svaki registrirani korisnik može kreirati više računa (novčanik) po kojima želi bilježiti određene vrste transakcija (prihode i troškove).

7.1. Izgled korisničkog sučelja

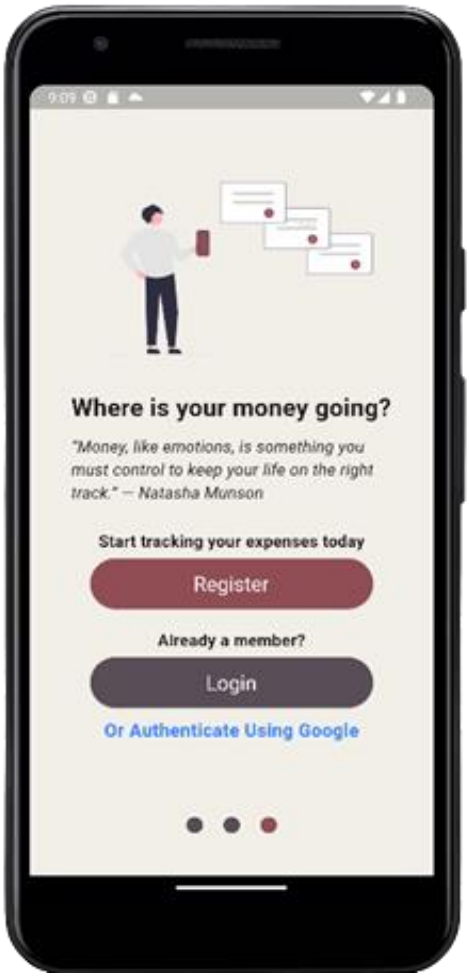
Aplikacija se sastoji od 10 različitih ekrana. Kod pokretanja aplikacije, otvara se uvodni ekran koji je zadužen za provjeru postojanja korisnika u lokalnoj bazi podataka. Ekran koji provjerava lokalnu bazu podataka prikazan je na idućoj slici.

Slika 23: Početni zaslon aplikacije (vlastita izrada)

Na ovom ekranu provjerava se *SharedPreferences* spremište. Ovo spremište je ključ-vrijednost baze podataka te provjerava vrijednost ključa *„user“*. Ukoliko je vrijednost ključa *null,* znači da korisnik nije prijavljen ili registriran te je nužan proći proces autorizacije, u slučaju kada je vrijednost ključa različita od *null,* znači korisnik je već prijavljen u aplikaciju te ga se automatski preusmjerava na zaslon koji sadrži popis njegovih računa.

U slučaju kada korisnik ne postoji u spremištu, otvara mu se ekran s motivacijskim porukama o štednji i važnosti praćenja proračuna. Ovaj dio nema nikakav utjecaj na rad aplikacije, već služi boljem korisničkom iskustvu.

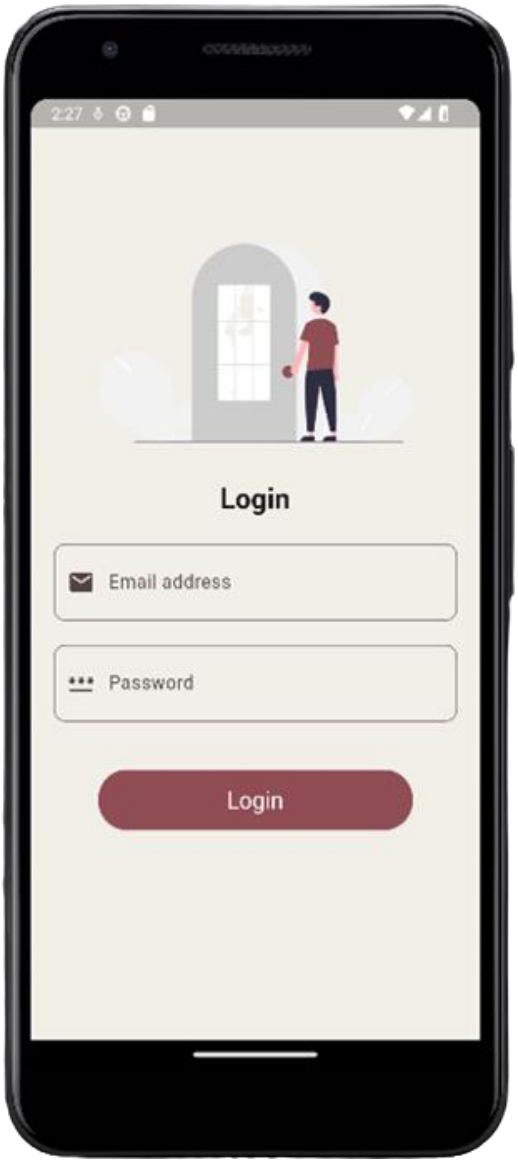
Slika 24: Ekran motivacijske poruke (vlastita izrada)



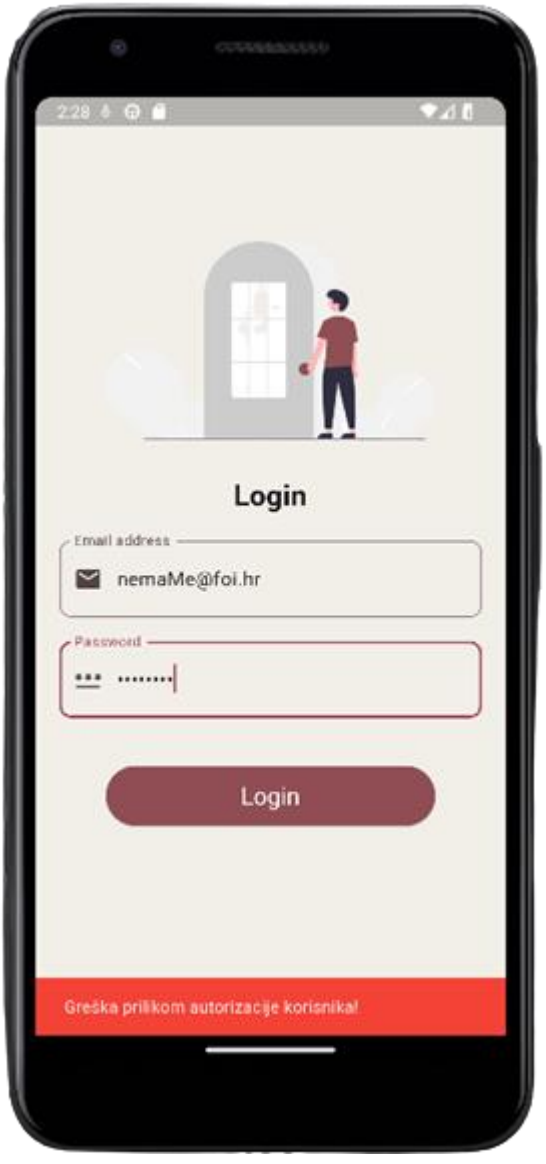
Slika 25: Ekran s mogućnošću odabira (vlastita izrada)

Posljednji ekran (vidi sliku 25) sadrži mogućnost prijave ili registracije korisnika. Ovisno o odabiru korisnika otvara se drugačija forma. Također, korisnik se može u aplikaciju registrirati putem vlastitog Google računa.

Korisniku se omogućuje odabir između registracije ili prijave u aplikaciju. Klikom na odgovarajući izbor otvara se drugačija forma za unos podataka. Ukoliko korisnik klikne na *Login* otvara mu se ekran prikazan na idućoj slici.



Slika 26: Forma za prijavu korisnika u aplikaciju (vlastita izrada)

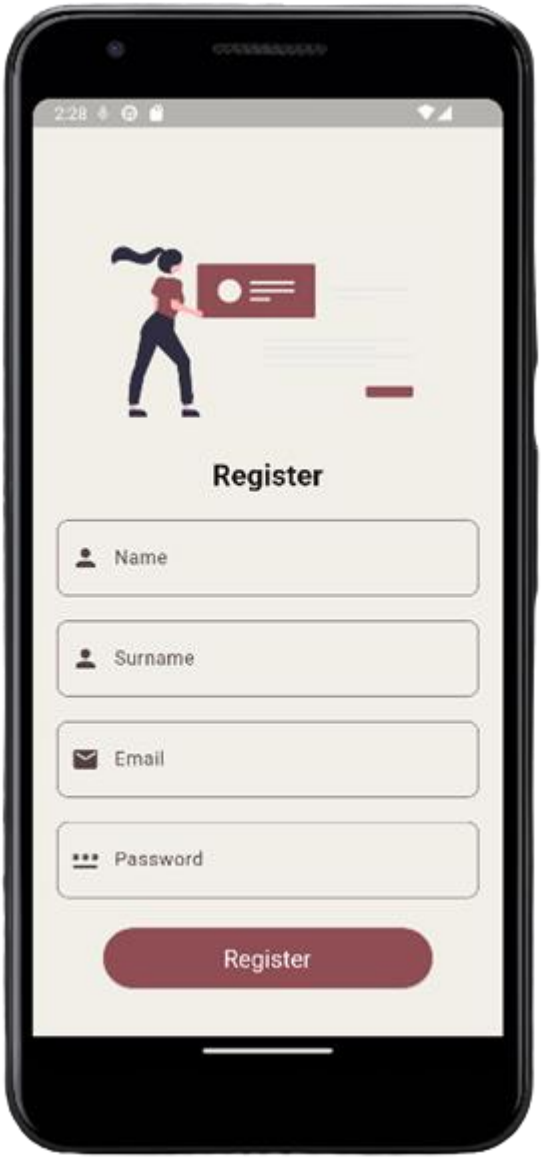


Slika 27: Neispravni korisnički podaci (vlastita izrada)

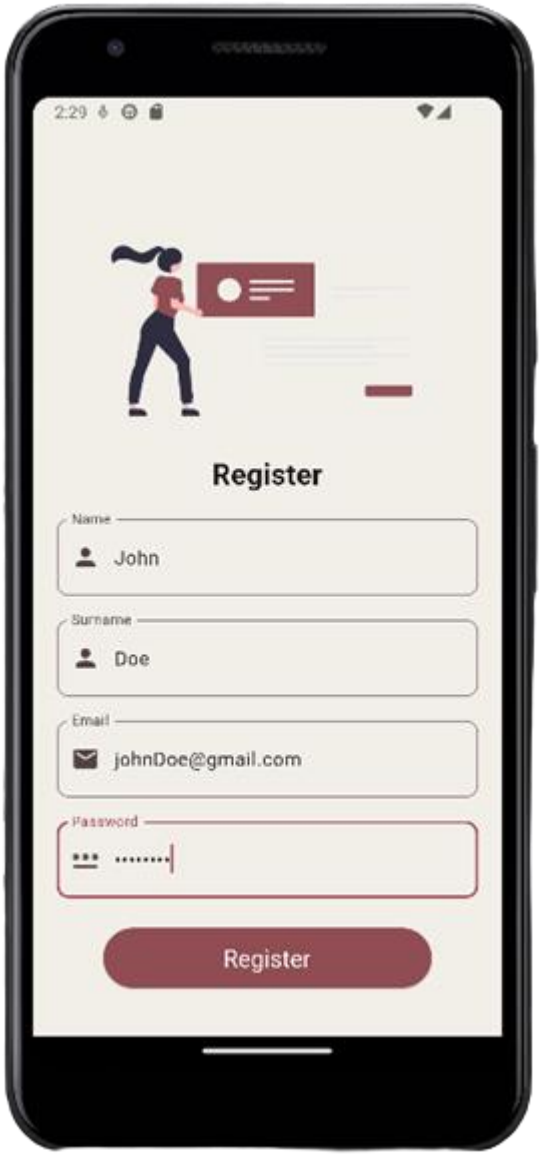
Na ovom ekranu korisnik unosi podatke za prijave, a to su e-mail adresa i lozinka. Nakon što korisnik klikne na *Login* pokreće se postupak provjere ispravnosti podataka. Ukoliko su korisnički podaci neispravni, javlja se greška prikazana na idućoj slici.

Ista poruka greške javlja se korisniku ukoliko ne postoji u bazi podataka te ukoliko je došlo do pogreške kod poziva web servisa.

Ukoliko korisnik ne postoji u bazi podataka, potreban je registrirati se. Registrirati se može odabirom *Register* na ekranskom sučelju prikazanom na slici 25, a registracijska forma prikazana je na idućoj slici.



Slika 28: Registracijsko sučelje (vlastita izrada)

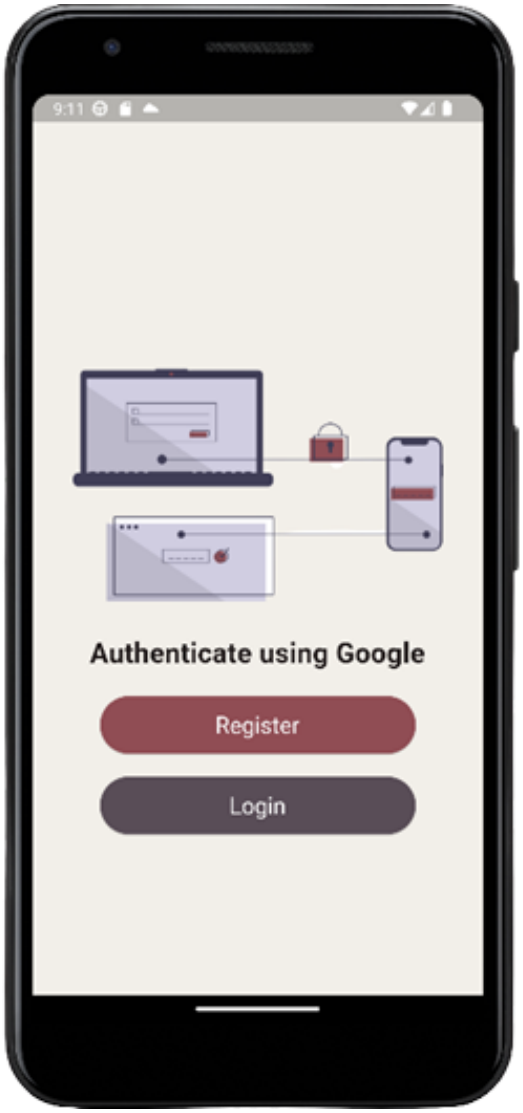


Slika 29: Registracijska forma sa unesenim podacima (vlastita izrada)

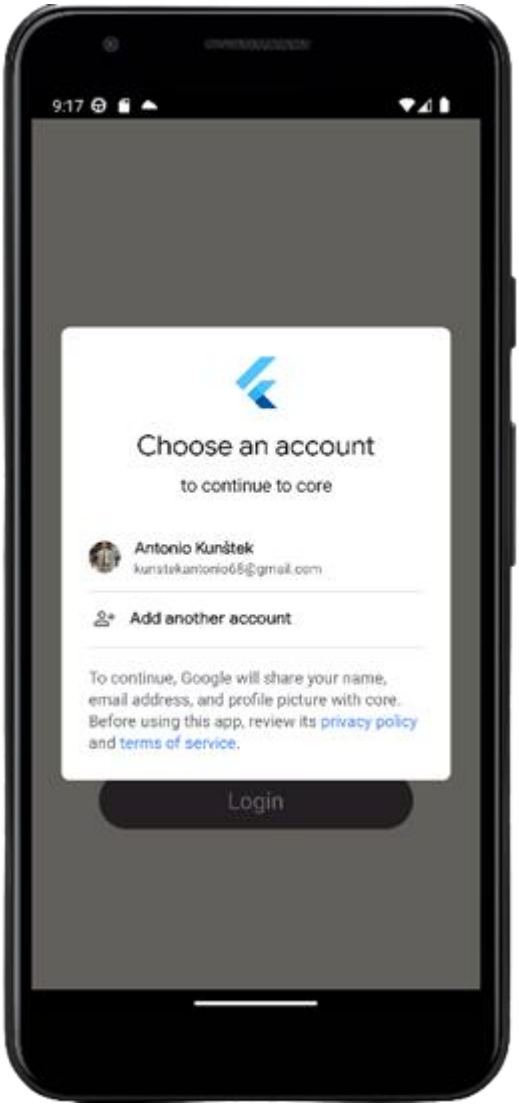
Ovdje korisnik unosi nužne podatke kako bi se uspješno registrirao unutar aplikacije. Korisnik je nužan unijeti ime, prezime, e-mail adresu i lozinku. E-mail adresa i lozinka služiti će za daljnju prijavu u aplikaciju. Izgled forme sa unesenim parametrima može se vidjeti na idućoj slici.

Nakon unesenih svih parametara te klikom na *Register*, korisnika se zapisuje u bazu podataka te mu se otvara ekran sa prikazom svih računa. Prilikom uspješne registracije u bazi podataka se aktivira okidač koji automatski dodaje račun „GOTOVINA“ za novo registriranog korisnika.

Na sličan način radi i autentifikacije putem Google računa, ukoliko na ekranu za odabir način autentifikacije korisnik odabire opciju Google, otvara mu se prozor unutar kojeg unosi podatke vezane uz vlastiti Google račun. Korisničko sučelje vezano za prijavu putem Google može se vidjeti na sljedećoj preslici zaslona (vidi sliku 30.).



Slika 30: Autentifikacija putem Google računa (vlastita izrada)



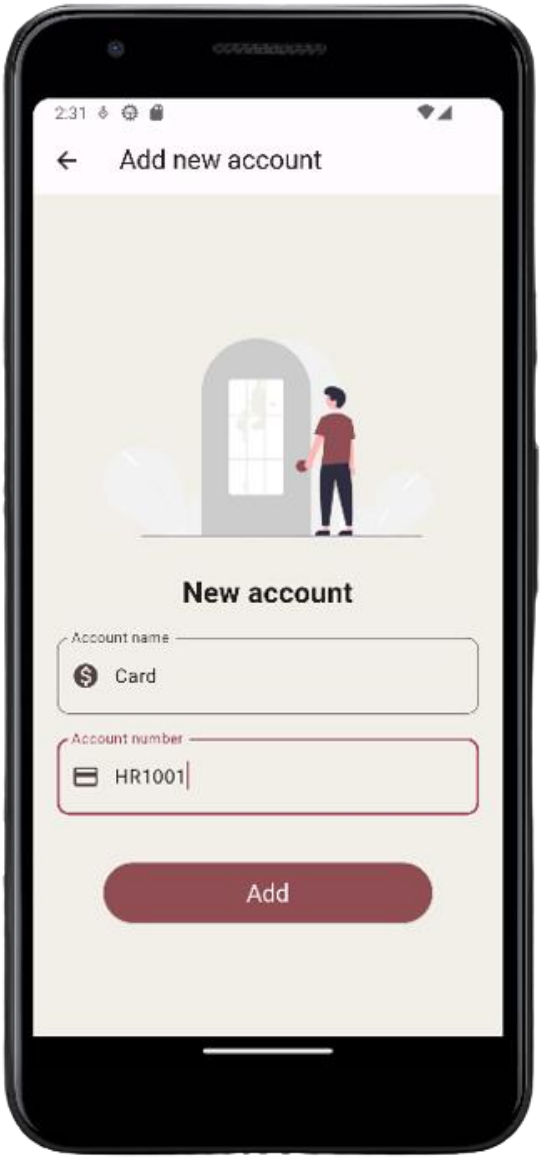
Slika 31: Odabir Google računa (vlastita izrada)

Kako bi se korisnik mogao prijaviti, najprije je potrebno izvršiti registraciju kako bi se uspješno kreirao JWT token vezan za autentifikaciju korisnika. Klikom na gumb „Register“ ili „Login“ otvara se isto korisničko sučelje vezano za odabir Google računa. Korisničko sučelje za odabir Google računa može se vidjeti na sljedećoj slici (vidi sliku 31).

Nakon odabira Google računa, vrši se ista programska logika za autentifikaciju korisnika te ga se prosljeđuje na ekran prikazan na slici 32.

Slika 32: Prikaz svih računa prijavljenog korisnika (vlastita izrada)

Na ovom ekranu korisnik može vidjeti sve račune dodane unutar sustava, te njihovo stanje. Može se vidjeti kako novokreirani račun „GOTOVINA“ ima 0.00€ raspoloživih sredstava. Klikom na donji desni gumb korisniku se otvara forma za unos novog računa (vidi sliku 33).



Slika 33: Forma za dodavanje novog računa (vlastita izrada)

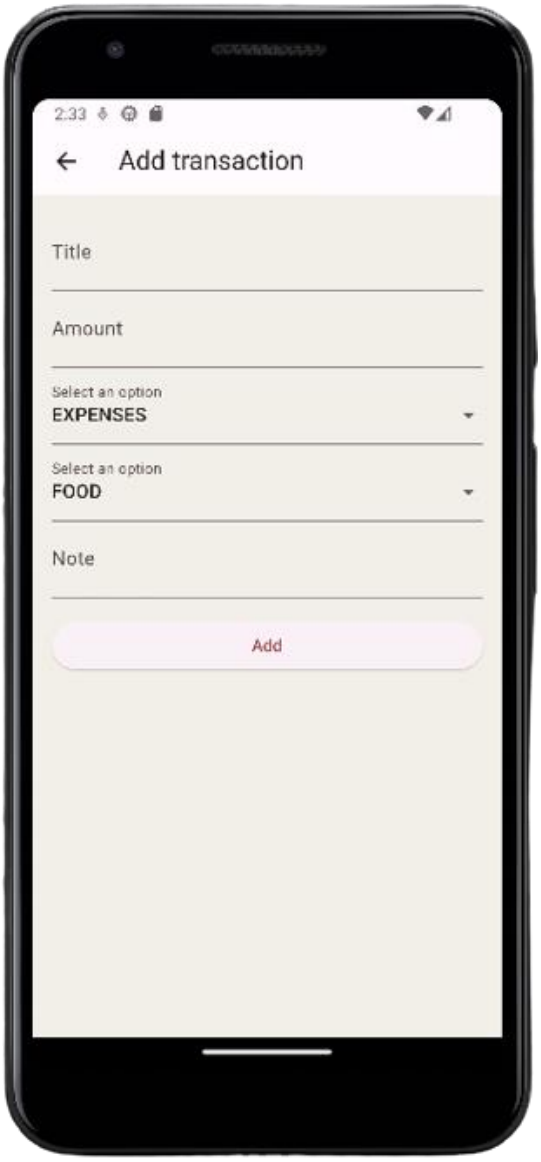
Slika 34: Prikaz svih računa nakon uspješnog dodavanja (vlastita izrada)

Ovdje korisnik unosi podatke vezane za izradu novog računa. Nakon što korisnik unese sve podatke i klikne na *Add* otvara mu se ekran sa svim računima koje ima spremljene u bazi podataka.

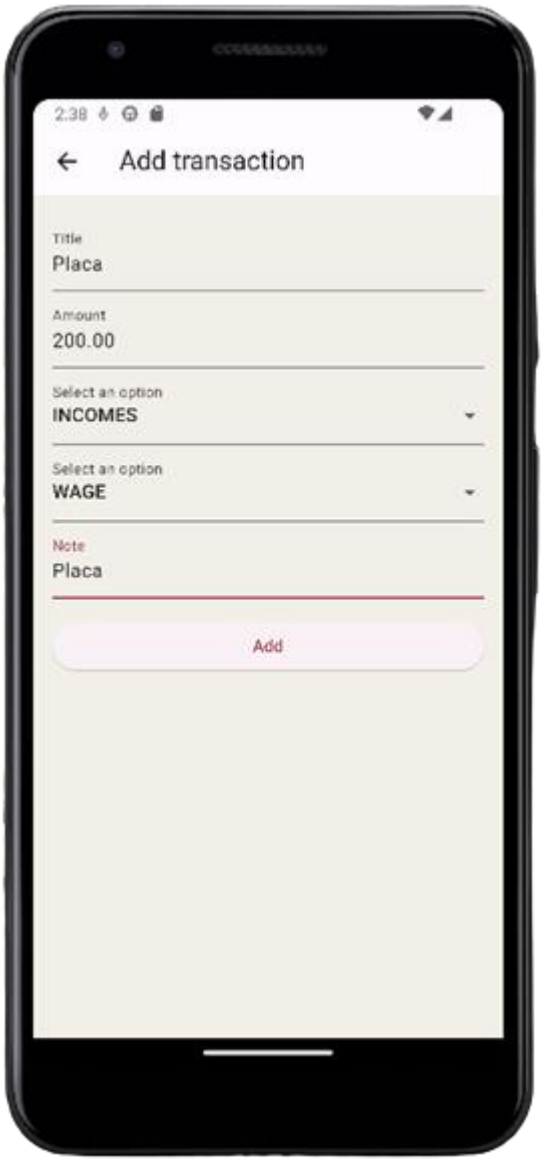
Može se vidjeti kako je račun predstavljen na slici 33 uspješno dodan unutar aplikacije. Klikom na bilo koji od računa korisnik ima mogućnost pregleda svih transakcija provedenih po tom računu (vidi sliku 35).

Slika 35: Prikaz svih transakcija po računu "Card" (vlastita izrada)

Budući da novo dodani račun još nema nikakvih transakcija, korisniku se otvara prazan prozor. Klikom na gumb koji se nalazi na donjem dijelu ekrana, korisniku se otvara forma za unos transakcije (vidi sliku 36).



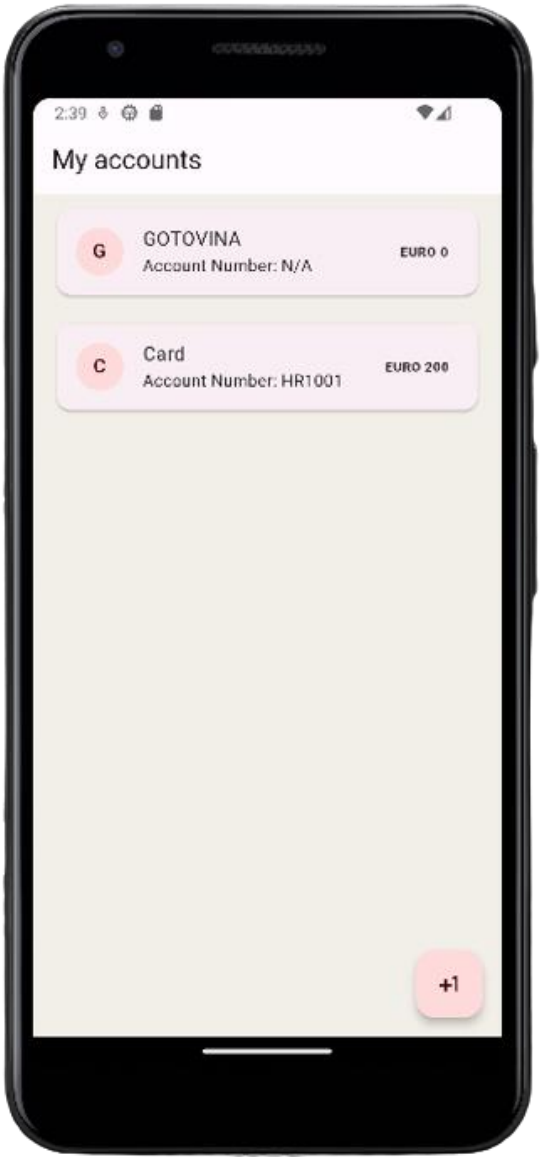
Slika 36: Forma za unos nove transakcije (vlastita izrada)



Slika 37: Dodavanje transakcije vrste prihod (vlastita izrada)

Na ovom ekranskom sučelju, korisnik unosi sve potrebne podatke vezane za transakciju. Ovisno o vrsti transakcije (prihod ili trošak), korisniku se pružaju različite kategorije transakcije. Budući da trenutno nema sredstva na računu *Card,* za početak dodati ćemo jednu transakciju vrste prihod (eng. Income).

Pritiskom na gumb *Add* transakcija se dodaje u sustav te se korisnika preusmjerava na početni ekran sa popisom svih računa.

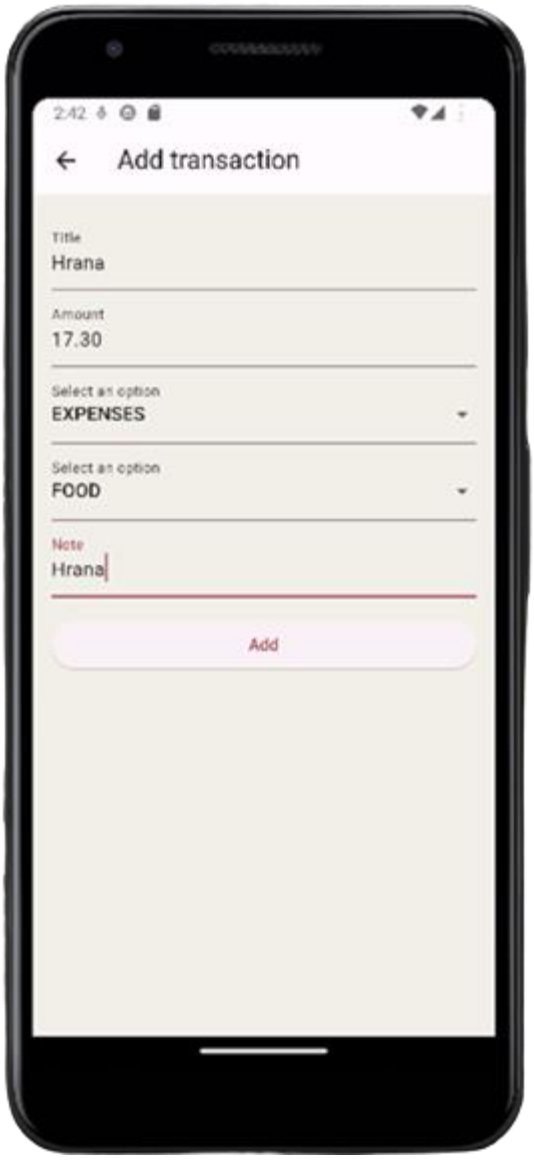


Slika 38: Popis svih računa nakon unošenja transakcije (vlastita izrada)

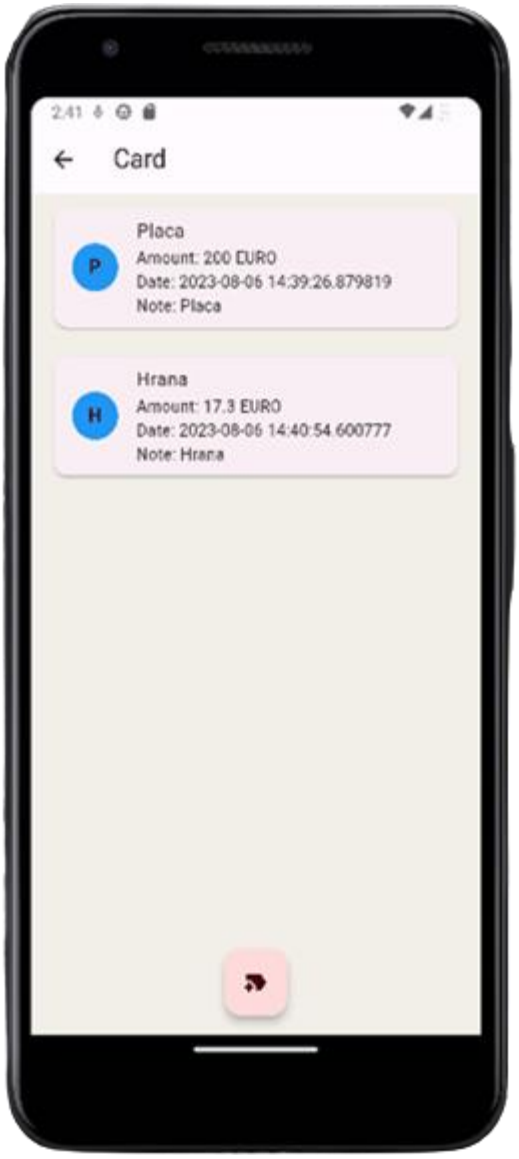
Na slici 38 može se vidjeti kako po računu *Card* korisnik sada ima položena sredstva. Pritiskom na račun *Card* korisniku se otvara ekran sa svim transakcijama učinjenim po odabranom računu.

Slika 39: Transakcije učinjene po računu Card (vlastita izrada)

Na slici 39 može se vidjeti prethodno dodana transakcija. Za potrebe kontrole izračuna ukupnog iznosa po računu dodana je još jedna transakcija u nastavku (vidi sliku 40).



Slika 40: Dodana transakcije vrste trošak (vlastita izrada)



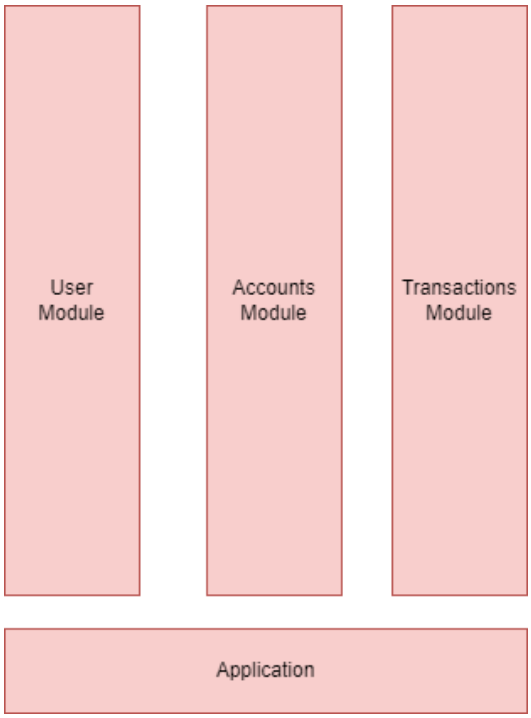
Slika 41: Prikaz svih unesenih transakcija po računu Card (vlastita izrada)

Na slici 40 dodana je transakcija vrste trošak. Korisniku se ponovno otvara ekran sa prikazom svih računa, a klikom na odabrani račun otvaraju se sve transakcije te se može vidjeti da su obje transakcije uspješno dodane.

U ovom poglavlju bili su prikazani svi korisnički ekrani dostupni unutar aplikacije. Sljedeće poglavlje sadrži implementaciju pojedinih modula navedenih na početku ovog poglavlja.

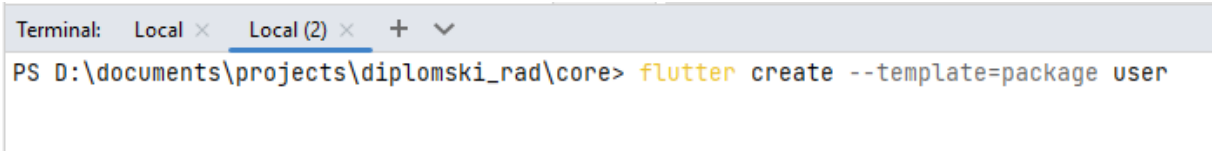
7.2. Implementacija modularnosti i BLoC arhitekture

Modularnost praktičnog dijela ostvarena je pomoću tri modula od kojih je svaki zadužen za točno jednu funkcionalnost (vidi sliku 42). *UserModule* je modul zadužen za registraciju i prijavu korisnika unutar aplikacije, *AccountsModule* zadužen je za kreiranje i dohvat svih računa kreiranih od zadanog korisnika, a *TransactionModule* zadužen je za kreiranje i dohvat svih transakcija po računu. Svaki od modula pisan je koristeći BLoC arhitekturu te je za svaki modul kreiran vlastiti repozitorij.



Slika 42: Modularnost aplikacija prema funkcionalnosti (vlastita izrada)

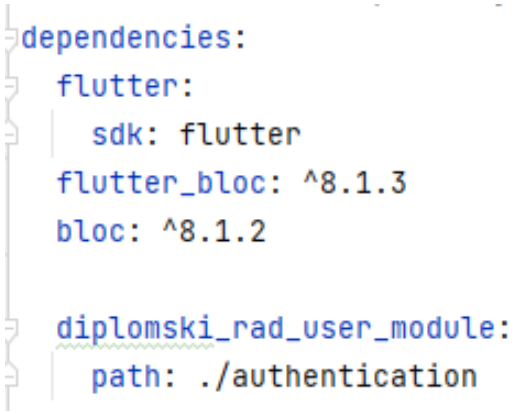
Novi modul kreira se na sljedeći način: putem terminala pozicioniramo se u direktorij u kojemu se nalazi projekt te izvršimo naredbu prikazanu na slici 43.



Slika 43: Izrada novog modula (vlastita izrada)

Izvršenjem ove komande, automatski će se kreirati sve potrebne datoteke za implementaciju vlastitog modula. Kako bi se novo kreirani modul mogao koristiti unutar glavne

aplikacije, potrebno je dodati referencu na modul unutar pubspec.yaml datoteke. Dodavanje modula može se vidjeti na slici 44.



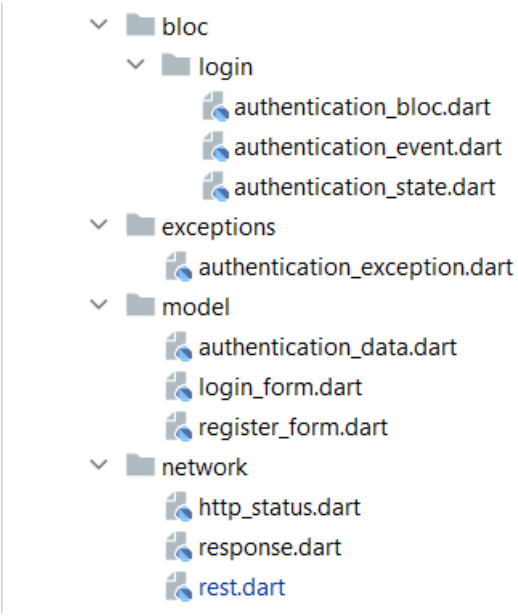
Slika 44: Dodavanje novog modula unutar aplikacije (vlastita izrada)

Nakon što se doda referenca na modul, aplikacija ima pristup svim klasama definiranim unutar modula. U nastavku se nalaze programski kodovi sa kojima su implementirani *UserModule*, *AccountModule* i *TransactionModule*.

7.2.1. UserModule

Kao što je bilo navedeno, *UserModul* je modul pisan BLoC arhitekturom te mu je glavna zadaća prijava i registracija korisnika unutar aplikacije. Programski kod nalazi se na github repozitoriju: https://github.com/antoniokunstek/diplomski_rad_user_module/tree/dev

Zaseban modul kreira se tako da se unutar terminala unese naredba: *flutter create --template=package „naziv-paketa“* . Flutter nam automatski generira potrebne datoteke za rad na novokreiranom modulu. Unutar lib direktorija kreirane su nove datoteke zbog lakšeg snalaženja u programskom kodu (vidi sliku 45).



Slika 45: Kreirane datoteke (vlastita izrada)

Bloc datoteka sadrži sve datoteke s kojima je definiran *UserBloc*, stanja u kojima se *UserBloc* može naći te događaji koji se mogu aktivirati, *exceptions* datoteka sadrži vlastitu iznimku kreiranu unutar ovog modula. Datoteka model sadrži sve modele potrebne za rad ovog modula. Naposljetku se nalazi network datoteka u kojoj se nalaze podaci vezani za rad sa REST API servisom.

```
class AuthenticationData {
  final String jwtToken;
  final String timestamp;

  const AuthenticationData({
    required this.jwtToken,
    required this.timestamp
  });

  factory AuthenticationData.fromJson(Map<String, dynamic> json) {
    return AuthenticationData(jwtToken: json['token'], timestamp:
json['timestamp']);
  }

  static Map<String, String> toJson(AuthenticationData data) {
    return {
      'jwtToken': data.jwtToken,
      'timestamp': data.timestamp
    };
  }
}
```

Isječak programskog koda 29: Izrada modela AuthenticationData (vlastita izrada)

Na početku kreiran je *AuthenticationData* model koji sadrži dvije varijable. Budući da je za pristup REST API servisu potreban jwt token, podaci o prijavljenom korisniku će biti spremljeni unutar tokena. Model sadrži dvije metode koje služe za pretvorbu objekta u json notaciju te za pretvorbu json notacije u model.

```
class LoginFormModel {
  String email;
  String password;

    LoginFormModel({
      required this.email,
      required this.password
  });

  static Map<String, String> toJson(LoginFormModel model) {
    return {
      'email': model.email,
      'password': model.password
    };
  }
}
```

Isječak programskog koda 30: Izrada modela LoginFormModel (vlastita izrada)

LoginFormModel predstavlja model unutar kojeg će korisnik spremati podatke unesene putem *Login* korisničkog sučelja.

```
class RegisterFormModel {
  String name;
  String surname;
  String email;
  String password;

  RegisterFormModel({
    required this.name,
    required this.surname,
    required this.email,
    required this.password
  });

  static Map<String, dynamic> toJson(RegisterFormModel model) {
    return {
      'firstName': model.name,
      'lastName': model.surname,
      'email': model.email,
      'password': model.password,
      'phone': null,
      'dateOfBirth': null,
      'gender': null,
      'residentialCountry': null
    };
  }
}
```

Isječak programskog koda 31: Izrada modela RegisterFormModel (vlastita izrada)

RegisterFormModel predstavlja model unutar kojeg će korisnik spremati podatke unesene putem *Register* korisničkog sučelja.

```
class AuthenticationException implements Exception {
  String message;
  AuthenticationException({required this.message});
}
```

Isječak programskog koda 32: Izrada AuthenticationException iznimke (vlastita izrada)

AuthenticationException predstavlja vlastitu kreiranu grešku. Pomoću nje će se emitirati drugačija vrijednost stanja aplikacije, primjerice: emitirati će se stanje koje će uzrokovati prikaz poruke o grešci.

```
import 'dart:convert';

import
'package:diplomski_rad_user_module/exceptions/authentication_exception.dart
';
import 'package:diplomski_rad_user_module/model/authentication_data.dart';
import 'package:diplomski_rad_user_module/model/login_form.dart';
import 'package:diplomski_rad_user_module/model/register_form.dart';
import 'package:diplomski_rad_user_module/network/response.dart';
import 'package:diplomski_rad_user_module/network/http_status.dart';
import 'package:http/http.dart' as http;

const String httpUrl = 'http://10.0.2.2:9999';

Future<AuthenticationData> fetchUser(LoginFormModel model) async {
  http.Response res = await
http.post(Uri.parse('$httpUrl/api/v1/auth/authenticate'),
          headers: <String, String> {
            'Content-type': 'application/json; charset=UTF-8',
      },
      body: json.encode(LoginFormModel.toJson(model))
    );

  if(res.statusCode == HttpStatus.OK.status) {
      Response res2 = Response.fromJson(jsonDecode(res.body));
      if(res2.successful) {
        return AuthenticationData.fromJson(res2.data);
      } else {
        throw AuthenticationException(message: 'Auth exception');
      }
    } else {
      throw AuthenticationException(message: 'Authentication exception');
    }
}

Future<AuthenticationData> registerUser(RegisterFormModel model) async {
  http.Response res = await
http.post(Uri.parse('$httpUrl/api/v1/auth/register'),
      headers: <String, String> {
        'Content-type': 'application/json; charset=UTF-8',
      },
      body: json.encode(RegisterFormModel.toJson(model))
  );

  if(res.statusCode == HttpStatus.OK.status) {
    Response res2 = Response.fromJson(jsonDecode(res.body));
    if(res2.successful) {
      return AuthenticationData.fromJson(res2.data);
    } else {
      throw AuthenticationException(message: 'Auth exception');
    }
  } else {
    throw AuthenticationException(message: 'Authentication exception');
  }
}
```

Isječak programskog koda 33: Izrada funkcija za rad s REST API servisom (vlastita izrada)

U datoteci *rest.dart* definirane su funkcije putem kojih se izvršavaju različiti pozivi REST servisa. Na primjer: *fetchUser()* metoda izvršava post metodu na putanji */api/v1/auth/authenticate* koja ovisno o uspješnosti poziva vraća grešku ili objekt tipa *AuthenticationData*.

```
import 'package:diplomski_rad_user_module/model/authentication_data.dart';

abstract class AuthenticationState {
  const AuthenticationState();
}

class AuthenticationInitial extends AuthenticationState {}

class AuthenticationProcessRequest extends AuthenticationState {}

class AuthenticationSuccess extends AuthenticationState {
  final AuthenticationData data;

  AuthenticationSuccess({required this.data});
}

class AuthenticationFailure extends AuthenticationState {}
```

Isječak programskog koda 34: Izrada mogućih AuthenticationState (vlastita izrada)

Unutar datoteke *authentication_state.dart* nalaze se sva stanja u kojima se *AuthenticationData* objekt može naći. Primjerice, prilikom kreiranja novog *AuthenticationState* emitira se *AuthenticationInitial* stanje. Događaji koji mogu promijeniti stanja definiraju se unutar *authentication_event.dart* datoteke.

```
import 'package:diplomski_rad_user_module/model/login_form.dart';
import 'package:diplomski_rad_user_module/model/register_form.dart';

abstract class AuthenticationEvent {

}

class OnLoginButtonPressed extends AuthenticationEvent {
  final LoginFormModel formModel;

  OnLoginButtonPressed({
    required this.formModel
  });
}

class OnRegisterButtonPressed extends AuthenticationEvent {
  final RegisterFormModel registerModel;

  OnRegisterButtonPressed({
    required this.registerModel
  });
}
```

Isječak programskog koda 35: Izrada mogućih AuthenticationEvent (vlastita izrada)

Kreirana su dva događaja koji uzrokuju promjenu stanja, a to su *OnLoginButtonPressed* i *OnRegisterButtonPressed*. Posljednje što je potrebno definirati je *authentication_bloc.dart* koji povezuje određeni događaj sa stanjem koji će se emitirati. Kako bi se authentication_bloc.dart mogao lako izmijeniti potrebno je kreirati sučelje koje će svaka konkretna implementacija *AuthenticationBloc* morati implementirati. U tu svrhu kreirano je sučelje *IAuthenticationBloc.* Kao primjer modularnosti koristeći inverziju ovisnosti kreirane su dva načina autentifikacije korisnika, a to su putem Google te putem vlastito implementiranog REST API serivsa. Implementaciju sučelja IAuthenticationBloc nalazi se na sljedećem isječku programskog koda.

```
abstract class IAuthenticationBloc implements StateStreamableSource<Authen-
ticationState>  {
  Future<void> onLoginButtonPressed(OnLoginButtonPressed event, Emitter<Au-
thenticationState> emit);
  Future<void> onRegisterButtonPressed(OnRegisterButtonPressed event, Emit-
ter<AuthenticationState> emit);
}


class GoogleAuthenticationBloc extends Bloc<AuthenticationEvent, Authenti-
cationState> implements IAuthenticationBloc {
  GoogleSignIn google = GoogleSignIn();
  GoogleAuthenticationBloc(): super(AuthenticationInitial()) {
    on<OnLoginButtonPressed>(onLoginButtonPressed);
    on<OnRegisterButtonPressed>(onRegisterButtonPressed);
  }

  @override
  Future<void> onLoginButtonPressed(OnLoginButtonPressed event, Emitter<Au-
thenticationState> emit) async {
    try {
      GoogleSignInAccount? account = await google.signIn();
      if(account == null) {
        throw AuthenticationException(message: "Google auth failed");
      }
      print(account.email);
      LoginFormModel formModel = LoginFormModel(email: account.email, pass-
word: "password");
      AuthenticationData authData = await fetchUser(formModel);
      emit(AuthenticationSuccess(data: authData));
    } catch (e) {
      print(e);
      emit(AuthenticationFailure());
    }
  }

  @override
  Future<void> onRegisterButtonPressed(OnRegisterButtonPressed event, Emit-
ter<AuthenticationState> emit) async {
   try {
     GoogleSignInAccount? account = await google.signIn();
     if(account == null) {
       throw new AuthenticationException(message: "Google auth failed");
     }
       RegisterFormModel formModel = RegisterFormModel(name: "Google", sur-
name: "User", email: account.email, password: "password");
       AuthenticationData authData = await registerUser(formModel);
       emit(AuthenticationSuccess(data: authData));
   } catch (e) {
     print(e);
     emit(AuthenticationFailure());
   }
  }
}

class RestAPIAuthentication extends Bloc<AuthenticationEvent, Authentica-
tionState> implements IAuthenticationBloc {
  RestAPIAuthentication(): super(AuthenticationInitial()) {
    on<OnLoginButtonPressed>(onLoginButtonPressed);
    on<OnRegisterButtonPressed>(onRegisterButtonPressed);
  }
```

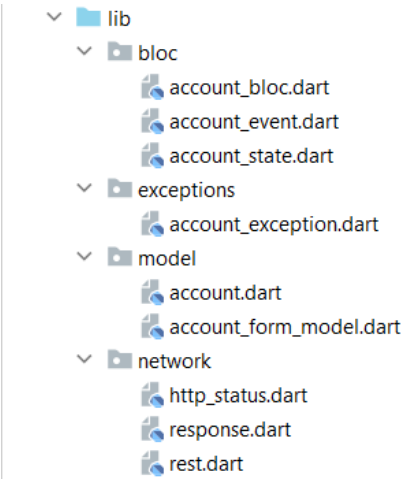
Isječak programskog koda 36: Izrada AuthenticationBloc (vlastita izrada)

Svaka implementacija *IAuthenticationBloc* klase prima vrstu događaja te vrstu stanja koje će se emitirati. Prilikom kreiranja nove instance objekta potrebno je definirati inicijalno stanje aplikacije sa događajima i funkcijama koje uzrokuju promjenu stanja. Primjerice, kada se aktivira događaj *OnLoginButtonPressed*, pozvati će se funkcija *_onLoginButtonPressed* koja emitira stanje *AuthenticationProcessRequest*, zatim stanje *AuthenticationSuccess* ili *AuthenticationFailure* ovisno o uspješnosti poziva web servisa.

7.2.2. AccountsModule

Kao što je prethodno navedeno *AccountsModul* je modul pisan BLoC arhitekturom te mu je glavna zadaća dohvaćanje i kreiranje novih računa prijavljenog korisnika. Programski kod nalazi se na github repozitoriju na lokaciji: https://github.com/antoniokunstek/diplomski-rad-account-module

. Unutar „*lib*“ direktorija kreirane su nove datoteke potrebne za izradu *AccountsModul* modula.



Slika 46: AccountsModul datoteke (vlastita izrada)

Budući se radi o BLoC arhitekturi sadržaj datoteke veoma je sličan *UserModule*.

```dart
import 'package:diplomski_rad_accounts_module/model/account.dart';

abstract class AccountState {
  const AccountState();
}

class AccountInitial extends AccountState {}

class AccountLoading extends AccountState {}

class AccountsLoaded extends AccountState {
  final List<Account> accountList;

  AccountsLoaded({
    required this.accountList
  });
}

class AccountCreated extends AccountState {}

class AccountsFailure extends AccountState {}
```

Isječak programskog koda 37: Izrada mogućih AccountState (vlastita izrada)

AccountState sadrži sva stanja u kojem se može naći račun korisnika.

```dart
import
'package:diplomski_rad_accounts_module/model/account_form_model.dart';

abstract class AccountEvent {

}

class OnWidgetInit extends AccountEvent {
  final String? authJwtToken;

    OnWidgetInit({
      required this.authJwtToken
  });
}

class OnCreateButtonPressed extends AccountEvent {
  final AccountFormModel model;
  final String? authJwtToken;

    OnCreateButtonPressed({
      required this.model,
      required this.authJwtToken
  });
}
```

Isječak programskog koda 38: Izrada AccountEvent (vlastita izrada)

Na isti način kao i sa *UserState*, potrebno je definirati sve događaje koji uzrokuju promjenu stanja *AccountState*.

```
import 'package:bloc/bloc.dart';

import '../model/account.dart';
import '../network/rest.dart';
import 'account_event.dart';
import 'account_state.dart';

class AccountsBloc extends Bloc<AccountEvent, AccountState> {
  AccountsBloc(): super(AccountLoading()) {
    on<OnWidgetInit>(_loadAccounts);
    on<OnCreateButtonPressed>(_createAccount);
  }

  Future<void> _loadAccounts(OnWidgetInit event, Emitter<AccountState>
state) async {
    try {
      List<Account> listOfAccounts = await
fetchAllAccounts(event.authJwtToken);
      emit(AccountsLoaded(accountList: listOfAccounts));
    } catch (e) {
      print(e);
      emit(AccountsFailure());
    }
  }

  Future<void> _createAccount(OnCreateButtonPressed event,
Emitter<AccountState> state) async {
    emit(AccountInitial());
    try {
      bool isCreated = await createAccount(event.authJwtToken,
event.model);
      if(isCreated) {
        emit(AccountCreated());
        emit(AccountLoading());
      } else {
        emit(AccountsFailure());
      }
    } catch (e) {
      emit(AccountsFailure());
    }
  }
}
```

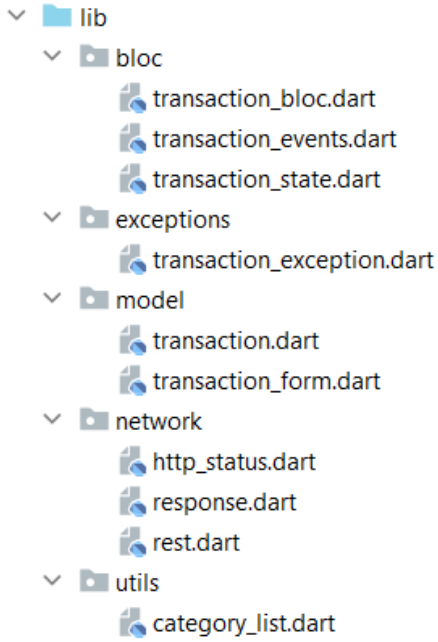
Isječak programskog koda 39: Izrada AccountBloc (vlastita izrada)

Na kraju je kreiran *AccountsBloc* koji povezuje događaje te emitira stanje ovisno o aktiviranom događaju. Svako stanje služiti će za prikaz drugačijeg izgleda korisničkog sučelja. Primjerice, dok se emitira stanje *AccountLoading*, na korisničkom sučelju prikazati će se animacija za učitavanje.

7.2.3. TransactionsModule

Kao što je bilo navedeno *TransactionsModule* je modul pisan BLoC arhitekturom te mu je glavna zadaća kreiranje i prikaz transakcija po odabranom računu unutar aplikacije. Programski kod sa implementacijom modula nalazi se na git repozitoriju: https://github.com/antoniokunstek/diplomski-rad-transaction-module.

Kao i u prethodnim modulima kreirana je struktura pogodna za razvoj BLoC arhitekture.



Slika 47: TransactionsModul kreirane datoteke (vlastita izrada)

Također potrebno je kreirati stanja, događaje te BLoC za navedeni modul. Prvobitno kreiramo stanja koje BLoC može emitirati.

```
import '../model/transaction.dart';

abstract class TransactionState {
  const TransactionState();
}

class TransactionInitial extends TransactionState {}
class TransactionLoading extends TransactionState {}
class TransactionLoaded extends TransactionState {
  final List<Transaction> transaction;

  TransactionLoaded({
    required this.transaction
  });
}

class TransactionCreated extends TransactionState {}
class TransactionFailure extends TransactionState {}
```

Isječak programskog koda 40: Izrada mogućih TransactionState (vlastita izrada)

Zatim je potrebno kreirati događaje koji se mogu aktivirati kako bi se emitiralo određeno stanje aplikacije.

```
import
'package:diplomski_rad_transactions_module/model/transaction_form.dart';

abstract class TransactionEvent {

}

class GetAllTransactions extends TransactionEvent {
  final String? accountId;
  final String? authJwtToken;

  GetAllTransactions({
    required this.accountId,
    required this.authJwtToken
  });
}

class GetAllIncomes extends TransactionEvent {
  final String? accountId;
  final String? authJwtToken;

  GetAllIncomes({
    required this.accountId,
    required this.authJwtToken
  });
}

class GetAllExpenses extends TransactionEvent {
  final String? accountId;
  final String? authJwtToken;

  GetAllExpenses({
    required this.accountId,
    required this.authJwtToken
  });
}

class AddTransaction extends TransactionEvent {
  final String? authJwtToken;
  final TransactionFormModel model;

  AddTransaction({
    required this.authJwtToken,
    required this.model
  });
}
```

Isječak programskog koda 41: Izrada TransacitonEvent (vlastita izrada)

Posljednje je potrebno spojiti kreirana stanja sa kreiranim događajima unutar *TransactionBloc* datoteke.

```
class TransactionBloc extends Bloc<TransactionEvent, TransactionState> {
  TransactionBloc(): super(TransactionInitial()) {
    on<GetAllTransactions>(_getAllTransactions);
    on<GetAllIncomes>(_getAllIncomes);
    on<GetAllExpenses>(_getAllExpenses);
    on<AddTransaction>(_addTransaction);
  }

  Future<void> _getAllTransactions(GetAllTransactions event,
Emitter<TransactionState> state) async {
    emit(TransactionLoading());
    try {
      List<Transaction> listOfTransaction = await
getAllTransactions(event.authJwtToken, event.accountId);
      emit(TransactionLoaded(transaction: listOfTransaction));
    } catch (e) {
      emit(TransactionFailure());
    }
  }

  Future<void> _getAllIncomes(GetAllIncomes event,
Emitter<TransactionState> state) async {
    emit(TransactionLoading());
    try {
      List<Transaction> listOfTransaction = await
getAllIncomes(event.authJwtToken, event.accountId);
      emit(TransactionLoaded(transaction: listOfTransaction));
    } catch (e) {
      emit(TransactionFailure());
    }
  }

  Future<void> _getAllExpenses(GetAllExpenses event,
Emitter<TransactionState> state) async {
    emit(TransactionLoading());
    try {
      List<Transaction> listOfTransaction = await
getAllExpenses(event.authJwtToken, event.accountId);
      emit(TransactionLoaded(transaction: listOfTransaction));
    } catch (e) {
      emit(TransactionFailure());
    }
  }

  Future<void> _addTransaction(AddTransaction event,
Emitter<TransactionState> state) async {
    emit(TransactionLoading());
    try {
      bool isCreated = await addTransaction(event.model,
event.authJwtToken);
      emit(TransactionCreated());
    } catch (e) {
      emit(TransactionFailure());
    }
  }
}
```

Isječak programskog koda 42: Izrada TransactionBloc (vlastita izrada)

Na isti način kao *UserModul* i *AccountsModul* potrebno je povezati određene događaje sa stanjima koje će događaj emitirati.

Nakon što su kreirani svi moduli, potrebno ih je unutar glavne aplikacije integrirati te spojiti sa određenim pogledima. U sljedećem poglavlju prikazan je način na koji je izvršena integracija kreiranih modula s aplikacijom.

7.3. Integracija modula unutar aplikacije

U ovom poglavlju prikazana je integracija zasebnih modula u jedinstvenu aplikaciju. Kako bi se kreirani moduli mogli koristiti unutar aplikacije potrebno ih je dodati u *pubspec.yml*. Možemo dodati nove sekcije unutar vanjskih biblioteka te za svaki modul odrediti u kojem repozitoriju se nalazi. Na ovaj način će se prilikom kreiranja nove verzije aplikacije povući posljednja verzija programskog koda koja se nalazi na git repozitoriju.

```
diplomski_rad_user_module:
  git:
    url: https://github.com/antoniokunstek/diplomski_rad_user_module.git
    ref: dev

diplomski_rad_accounts_module:
  git:
    url: https://github.com/antoniokunstek/diplomski-rad-account-module.git
    ref: master

diplomski_rad_transactions_module:
  git:
    url: https://github.com/antoniokunstek/diplomski-rad-transaction-
module.git
    ref: main
```

Isječak programskog koda 43: Preuzimanje vlastitih kreiranih modula (vlastita izrada)

Kako bi se kreirani BLoC obrasci mogli koristiti unutar aplikacije, potrebno je u *main()* metodi kreirati nove instance BLoC objekata. Prilikom kreiranja novih instanci, aplikacije će imitirati ona stanja koja smo zadali kao inicijalna stanja unutar svakog modula.

```
void main() {
  runApp(MultiBlocProvider(providers: [
    BlocProvider(create: (context) => AuthenticationBloc()),
    BlocProvider(create: (context) => AccountsBloc()),
    BlocProvider(create: (context) => TransactionBloc())
  ], child: BudgetPlannerApp()));
}
```

Isječak programskog koda 44: Kreiranje novih instanci Bloc objekata (vlastita izrada)

Spajanje određenog pogleda sa stanjem aplikacije vrši se pomoću *BlocListener* i *BlocBuilder* widgeta. *BlocListener* služi za „slušanje“ promjena stanja definiranog unutar

BlocListener-a. Ukoliko se promjeni stanje, prvo će se izvršiti *BlocListener*, ovdje se može izvršiti neki dio programskog koda koji ne utječe direktno na izgled korisničkog sučelja. Može se pokrenuti neka metoda u pozadini, primjerice brisanje određenog zapisa ili slično, a *BlocBuilder* je widget koji na temelju stanja rekonstruira korisničko sučelje.

```
class _WalletPageState extends State<WalletPage> {

  @override
  void initState() {
    _onWidgetInit(context);
  }

  void _onWidgetInit(BuildContext context) async {
    BlocProvider.of<AccountsBloc>(context).add(
        OnWidgetInit(authJwtToken: await
LocalStorage.getUserFromLocalStorage()));
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(automaticallyImplyLeading: false,title: const
Text("My accounts")),
        resizeToAvoidBottomInset: false,
        backgroundColor: AppColors.backgroundColor,
        body: BlocListener<AccountsBloc, AccountState>(
            listener: (context, state) {
                if (state is AccountsFailure) {
                ScaffoldMessenger.of(context).showSnackBar(
                  const SnackBar(
                    content: Text("Greška prilikom otvaranja računa
korisnika!"),
                    backgroundColor: Colors.red,
                  ),
                );
              }
            },
            child: BlocBuilder<AccountsBloc, AccountState>(
              builder: (context, state) {
                if(state is AccountLoading) {
                  _onWidgetInit(context);
                  return const Center(
                    child: CircularProgressIndicator(),
                  );
                }
                else if(state is AccountsLoaded) {
                  return ListView.builder(itemCount:
state.accountList.length,
                      itemBuilder: (context, index) {
                    return AccountCard(state.accountList[index],
context);
                  });
                } else {
                  return const Text("Error");
                }
              },
            ),
        ),
        floatingActionButton: FloatingActionButton(
          child: const Icon(Icons.plus_one),
          onPressed: () async => _pushToCreateAccountPage(context, await
LocalStorage.getUserFromLocalStorage()),
        ),
    );
  }
```

Isječak programskog koda 45: Kreiranje WalletPage pogleda koristeći Bloc arhitekturu (vlastita izrada)

Možemo vidjeti pogled *WalletPage* koji na temelju određenih stanja mijenja izgled korisničkog sučelja. Kada aplikacija emitira stanje *AccountsFailure* na dnu se prikaže poruka „Greška prilikom otvaranja računa korisnika!“, kada se emitira stanje *AccountsLoaded* onda se prikaže lista svih računa kreiranih od prijavljenog korisnika. Naposljetku, kada se emitira stanje *AccountLoading* onda se prikazuje animacija učitavanja podataka.

Kako bi se mogla koristiti različita vrsta autentifikacije na primjeru načela inverzije ovisnosti, potrebno je widgetu koji ovisi o određenu BLoC arhitekturi proslijediti konkretnu implementaciju sučelja koje ovisi o apstrakciji.

Odabirom različite vrste autentifikacije, korisniku se otvara drugačiji prikaz forme sa različitom implementacijom *IAuthenticationBloc*. Na idućem isječku programskog koda prikazana je inicijalizacija različitog prikaza ovisno o korisničkom odabiru.

```
const SizedBox(height: 5),
  ElevatedButton(onPressed: () {
    Navigator.push(context,
      MaterialPageRoute(builder: (context) =>
(LoginForm(authenticationBloc: RestAPIAuthentication())));
  },
      style: ElevatedButton.styleFrom(
          backgroundColor: AppColors.secondaryAppColor,
          fixedSize: const Size(275, 50),
          textStyle: const TextStyle(fontSize: 20)
      ),
      child: Text("Login", style: TextStyle(color: AppColors.buttonText))),
const SizedBox(height: 10),
GestureDetector(
  child: const Text("Or Authenticate Using Google",
          style: TextStyle(fontSize: 18, fontWeight: FontWeight.w800,
color: Colors.blueAccent),),
  onTap: () {
    Navigator.push(context,
        MaterialPageRoute(builder: (context) =>
(GoogleLoginForm(authenticationBloc: GoogleAuthenticationBloc())));
  },
)
```

Isječak programskog koda 46: Inicijalizacije drugačijeg IAuthenticationBloc (vlastita izrada)

Ukoliko korisnik klikne na gumb „Login“ otvara mu se forma sa implementacijom *IAuthenticationBloc* koja pristupa vlastitom REST API servisu, dok se pritiskom na tipku „*Or Authenticate Using Google*“ otvara drugačije korisničko sučelje sa pristupom Google autentifikacijskom servisu.

Na sljedećem isječku programskog koda prikazana je inverzija ovisnosti unutar određenog prikaza.

```
class GoogleLoginForm extends StatefulWidget {
  IAuthenticationBloc authenticationBloc;

  GoogleLoginForm({
    super.key,
    required this.authenticationBloc
  });

  @override
  State<GoogleLoginForm> createState() => _GoogleLoginFormState();
}
```

Isječak programskog koda 47: Inverzija ovisnosti autentifikacije (vlastita izrada)

Na prethodnom isječku može se vidjeti kako *GoogleLoginForm* sadrži sučelje *IAuthenticationBloc*, a budući da se apstrakcija ne može inicijalizirati, potrebno je prilikom inicijalizacije *GoogleLoginForm,* unutar konstruktora, proslijediti konkretnu implementaciju sučelja, u ovome slučaju *GoogleAuthenticationBloc*.

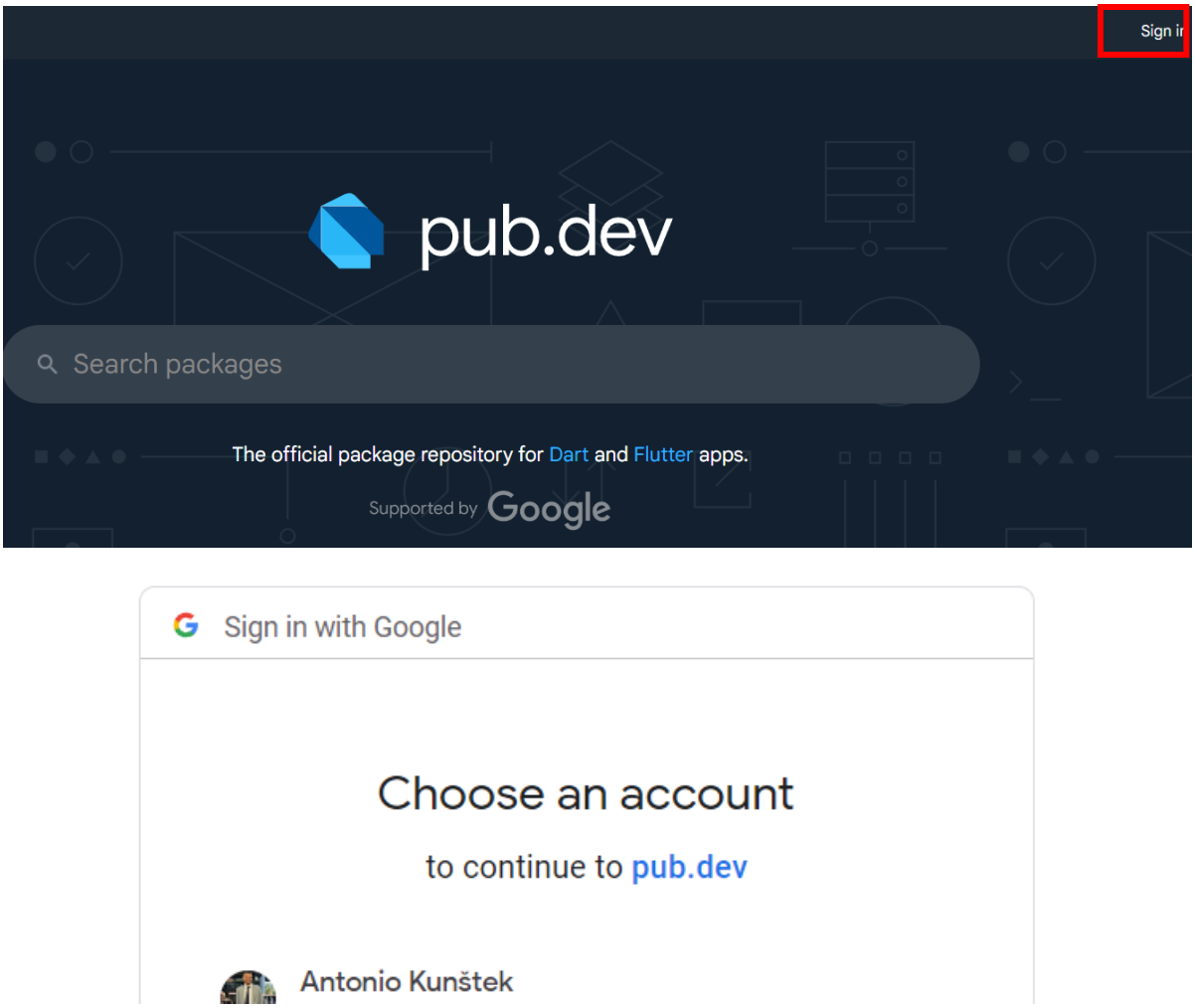
Korištenjem načela inverzije ovisnosti osigurali smo drugačiju implementaciju BLoC arhitekture ovisno o poslovnom događaju. Tako se mogu nadodati drugačije vrste autentifikacije, primjerice autentifikacija putem GitHub računa, te se umjesto *GoogleAuthenticationBloc* u konstruktoru proslijedi *GitHubAuthenticationBloc*.

U sljedećem poglavlju opisane su smjernice za objavljivanje vlastito kreiranog modula na javno dostupne repozitorije.

7.4. Objava vlastitog paketa na javno dostupan repozitorij

Ponekad se vlastito rješenje želi podijeliti sa ostalim programerima. Budući da je Flutter pisan u Dart programskom jeziku, vlastite pakete korisnici mogu objaviti na pub.dart javnom repozitoriju. Kako bi korisnik mogao objaviti paket javno, potrebno je pravilno dokumentirati, verzionirati i u konačnici strukturirati programski kod. Prvi korak je naravno kreirati novi paket (modul) prema primjeru koji smo naveli u ranijim poglavljima pri čemu je potrebno ažurirati pubspec.yaml datoteku. Ova datoteka sadrži meta podatke o paketu, njegovim vanjskim bibliotekama i slično. Nužno je ažurirati sljedeća polja: ime, opis, verzija i autor. Sljedeći korak je pisanje programskog koda u skladu s Dartovim smjernicama za strukturu paketa. Veoma je važno ažurirati dokumentaciju pisano u Markdown jeziku kako bi ostali korisnici mogli što jednostavnije implementirati paket u svojim projektima. Službena dokumentacija Dart programskog jezika, savjetuje prikaz korištenja paketa na više od jednog primjera. Prije objavljivanja paketa na službene stranice, potrebno je verzionirati paket slijedeći pravila

semantičke verzije. Za objavljivanje paketa, potrebno je kreirati novi Dart račun pri čemu će ostali programeri moći provjeriti informacije o autoru paketa. Novi račun kreira se na službenoj stranici pub.dev. Budući da se radi o Google biblioteci za razvoj aplikacija, potrebno je imati kreirani Google račun.



Slika 48: Prijava na pub.dev (vlastita izrada)

Nakon što se odabere vlastiti google račun, programer je u mogućnosti objaviti vlastiti paket na javno dostupan repozitorij. Prije objave, programer je dužan provjeriti sve datoteke koje će učitati sa svojim paketom. Srećom, postoji posebna komanda u Flutter koji automatski obrađuje validaciju paketa te korisnika upućuje na potencijalna upozorenja i greške.

```
PS D:\documents\projects\diplomski_rad\antonio_package_test> flutter pub publish --dry-run
```

Slika 49: Validacija paketa prije objave (vlastita izrada)

Prilikom pokretanja ove komande, Flutter provjerava README.md, CHANGELOG.md i pubspec.yaml datoteke. Ukoliko nešto nedostaje, Flutter u terminalu vrati poruku greške ili upozorenja.

```
Validating package... (4.1s)
Package validation found the following potential issue:
* It's strongly recommended to include a "homepage" or "repository" field in your pubspec.yaml

Package has 1 warning.
PS D:\documents\projects\diplomski_rad\antonio_package_test>
```

Slika 50: Povratne poruke Flutter validacije (vlastita izrada)

Na prethodnoj slici može se vidjeti kako prilikom objave paketa unutar pubspec.yaml datoteke nedostaje link na javno dostupan repozitorij. Budući da će ostali programeri koristiti biblioteku u vlastitim projektima, poželjno je da mogu vidjeti izvorni programski kod.

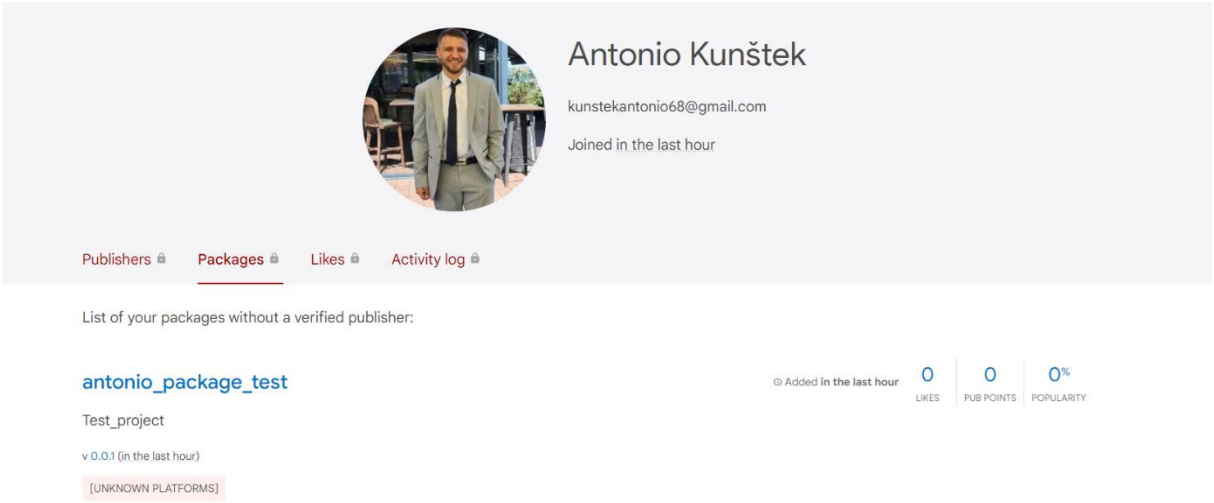
Nakon što se isprave sve greška i upozorenja upućenih od Flutter validacije, slijedi objava paketa na javno dostupan repozitorij pomoću naredbe flutter pub publish.

```
PS D:\documents\projects\diplomski_rad\antonio_package_test> flutter pub publish
```

Slika 51: Objava Flutter paketa na pub.dev (vlastita izrada)

Nakon uspješne objave, korisnici mogu uz pomoć *pub get ime_paketa* naredbe dohvatiti vanjsku biblioteku te ju koristiti u svojim projektima prema smjernicama navedenih u Markdown datoteci. Budući da će ostali programeri koristiti implementirano rješenje, potrebno je održavati i pravilno ažurirati paket te ukoliko se je odustalo od održavanja, obavijestiti korisnike u njegovom statusu. (Dart, 2023.)

Vlastito objavljeni javni paketi nalaze se na službenoj stranici prethodno kreiranog profila za objavljivanje paketa što se može vidjeti na sljedećoj slici.



Slika 52: Profil javno dostupnih paketa (vlastita izrada)

U poglavlju 7 je na jednostavnom primjeru aplikacije za praćenje stanja budžeta prikazana je modularna arhitektura koristeći BLoC obrazac. Usvajanjem ovog obrasca, odvaja

se poslovna logika od komponenti korisničkog sučelja te se kreirani BLoC-ovi mogu koristiti u drugim aplikacijama, a ukoliko želimo vlastita rješenja koristiti na više projekata, potrebno je objaviti paket na jednom od repozitorija.

8. ZAKLJUČAK

U ovom diplomskom radu obrađen je pojam modularnosti te su prikazane najčešće korištene arhitekture Flutter aplikacija. Korištenjem odgovarajuće arhitekture aplikacija dobiva izvanredan napredak u organizaciji koda, mogućnosti ponovne upotrebe, održavanju i skalabilnosti. Jasno razdvajanje problema s različitim modulima omogućuje povećanu suradnju između programera. Svaki programer može samostalno raditi na svojim modulima što povećava paralelni razvoj te povećava produktivnost. Kroz ovaj rad predstavljeni su različiti pristupi izrade modularne arhitekture. Svaki od obrazaca arhitekture ima fokus na razdvajanje odgovornosti čime se savršeno usklađuju s ciljevima modularnosti.

Na praktičnom primjeru korištena je BLoC arhitektura, koja se smatra jednom od najpopularnijih arhitektura među Flutter programerima. Jednosmjerni tok podataka BLoC arhitekture osigurava jednostavan način upravljanja podacima aplikacije te smanjuje rizik od nedosljednosti podataka i potencijalnih grešaka. Nadalje, modularna arhitektura uvelike je pojednostavila proces integracije novih značajki te izmjenu već postojećih funkcija. Svaki modul može se ažurirati bez utjecaja na druge dijelove aplikacije, čime se smanjuje rizik od neželjenih nuspojava te se ubrzava razvojni ciklus. Znanje i uvidi dobiveni ovim istraživanjem različitih vrsta arhitektura mogu poslužiti kao vrijedan izvor za programere koji žele stvoriti kvalitetne, modularne mobilne aplikacije pisane u Flutter razvojnom okviru.

POPIS LITERATURE

[1.] Microsoft Visual Studio (2023.) Preuzeto 16.4.2023. s

[2.] Git (2023.) Preuzeto 16.4.2023. s https://git-scm.com

[3.] Kinsta (bez dat.) What is Github? Preuzeto 16.4.2023. s https://kinsta.com/knowledgebase/what-is-github

[4.] Sourcetree (2023.) Preuzeto 16.4.2023. s https://www.sourcetreeapp.com

[5.] Gonzalez R. (2017). Figma Wants Designers to Collaborate Google-Docs Style. Preuzeto 17.4.2023. s https://www.wired.com/story/figma-updates

[6.] DBeaver (2023.) Preuzeto 17.4.2023. s https://dbeaver.io/

[7.] PostgreSQL (2023.) Preuzeto 17.4.2023. s https://www.postgresql.org

[8.] Spring (2023.) Preuzeto 18.4.2023. s https://spring.io/

[9.] Postman (2023.) Preuzeto 18.4.2023. s https://www.postman.com/

[10.] Singh S. (2023). Native vs Hybrid vs Cross Platform – What to Choose in 2023? Preuzeto 20.4.2023. s https://www.netsolutions.com/insights/native-vs-hybrid-vs-cross-platform/

[11.] Jetbrains (2023). What is cross-platform mobile development? Preuzeto 22.4.2023 s https://kotlinlang.org/docs/cross-platform-mobile-development.html

[12.] Android Developer (2023). Guide to app architecture. Preuzeto 28.5.2023. s https://developer.android.com/topic/architecture

[13.] Vujević T. (2023). The mobile app architecture guide. Preuzeto 28.5.2023. s https://decode.agency/article/mobile-app-architecture/

[14.] Martin, R. C. (2018). Clean Architecture: A Craftsman's Guide to Software Structure and Design, USA: Pearson Education, Inc.

[15.] Watts S. (2020). The Importance of SOLID Design Principles. Preuzeto 3.6.2023. s https://www.bmc.com/blogs/solid-design-principles/

[16.] Oloruntoba S. (2021). SOLID: The First 5 Principles of Object Oriented Design. Preuzeto 4.6.2023. s https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#open-closed-principle

[17.] Flutter (2023.). Start thinking declaratively. Preuzeto 5.6.2023. s https://docs.flutter.dev/data-and-backend/state-mgmt/declarative

[18.] Flutter (2023.). Introduction to widgets. Preuzeto 5.6.2023. s https://docs.flutter.dev/ui/widgets-intro

[19.] Codecademy Team (bez datuma). MVC: Model, View, Controller. Preuzeto 5.6.2023. s https://www.codecademy.com/article/mvc

[20.] Pub.dev (bez datuma). Službena stranica javno dostupnih paketa. Preuzeto 6.6.2023. s https://pub.dev/

[21.] Nnamdi C. (bez datuma). What is Flutter BLoC?. Preuzeto 7.6.2023. s https://www.educative.io/answers/what-is-flutter-bloc

[22.] Suri S. (2018.) Architect your Flutter project using BLOC pattern. Preuzeto 7.6.2023. s https://medium.com/codechai/architecting-your-flutter-project-bd04e144a8f1

[23.] Flutter Guru (2022.) Flutter State Management with Provider. Preuzeto 9.6.2023. s https://blog.devgenius.io/flutter-state-management-with-provider-5a57eca108f1

[24.] Lapp T. J. (2019.) Understanding Provider in Diagrams – Part 3: Architecture. Preuzeto 9.6.2023. s https://medium.com/flutter-community/understanding-provider-in-diagrams-part-3-architecture-a145e4fbbde1

[25.] Joleyemi D. (2021.) Flutter Redux: Complete tutorial with examples. Preuzeto 10.6.2023. s https://blog.logrocket.com/flutter-redux-complete-tutorial-with-examples

[26.] MacDonald M. (2023.) Modular programming: beyond the spaghetti mess. Preuzeto 14.7.2023.s https://www.tiny.cloud/blog/modular-programming-principle

[27.] Komara Fauzi Rifa R. (2020.) Flutter: Mastering Modularization – In Several Ways. Preuzeto 15.7.2023. s https://medium.com/flutter-community/mastering-flutter-modularization-in-several-ways-f5bced19101a

[28.] Dart službena dokumentacija (2023.) Publishing packages. Preuzeto 23.8.2023. s https://dart.dev/tools/pub/publishing

POPIS KORIŠTENIH BIBLIOTEKA

[1.] bloc 8.1.2. Preuzeta 7.6.2023. s https://pub.dev/packages/bloc

[2.] Provider 6.0.5. Preuzeta 9.6.2023. s https://pub.dev/packages/provider

[3.] redux 5.0.0. Preuzeta 10.6.2023. s https://pub.dev/packages/redux

[4.] flutter_redux 0.10.0. Preuzeta 10.6.2023. s https://pub.dev/packages/flutter_redux

[5.] google_sign_in 6.1.4 Preuzeta 22.8.2023 s https://pub.dev/packages/google_sign_in

POPIS SLIKA

POPIS TABLICA

POPIS ISJEČAKA PROGRAMSKOG KODA

POPIS POVEZNICA NA REPOZITORIJE

[1.] Aplikacija: https://github.com/antoniokunstek/diplomski-rad

[2.] UserModule: https://github.com/antoniokunstek/diplomski_rad_user_module

[3.] TransactionModule: https://github.com/antoniokunstek/diplomski-rad-transaction-module

[4.] AccountModule: https://github.com/antoniokunstek/diplomski-rad-account-module