Arhitekture mobilnih aplikacija razvijanih okvirom React Native

Potočki, Helena

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Helena Potočki

ARHITEKTURE MOBILNIH APLIKACIJA RAZVIJANIH OKVIROM REACT NATIVE

DIPLOMSKI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Helena Potočki

Matični broj: 0016129570

Studij: Organizacija poslovnih sustava

ARHITEKTURE MOBILNIH APLIKACIJA RAZVIJANIH OKVIROM REACT NATIVE

DIPLOMSKI RAD

Mentor :

Izv. prof. dr. sc. Zlatko Stapić

Varaždin, rujan 2024.

Helena Potočki

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.

Autorica potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U ovom diplomskom radu razrađuje se tematika arhitektura mobilnih aplikacija pisanih u razvojnom okviru React Native. Iznosi se definicija i važnost arhitekture sustava kao izrazito važnog dijela razvoja bilo kojeg sustava, a tako i mobilne aplikacije. Polazi se od činjenice da je za postizanje kvalitete aplikacije važno osigurati da je aplikacija skalabilna, izdržljiva i lako održiva kako bi postigla svoj potencijal u korištenju i kako bi se smanjili troškovi izrade same aplikacije. U sklopu teorijskog definiranja arhitekture pojašnjene su i neke metode i principi u razvoju aplikacije koje predstavljaju provjerene najbolje prakse i često su korištene u razvoju programskih proizvoda. Između ostalih, navedeni su SOLID principi razvoja aplikacija te su u sklopu obrade navedenih principa izrađeni kratki primjeri korištenja u programskom jeziku JavaScript. U nastavku su navedene najčešće korištene vrste arhitektura u razvoju aplikacija React Native-om. Svaka od arhitektura detaljno je pojašnjena te je njihova struktura prikazana na dijagramima. U sklopu prikazivanja obrađenih i istraženih arhitektura, izrađena je mobilna aplikacija na kojoj se može vidjeti primjena triju odabranih arhitektura. Također, pojašnjene su prednosti i nedostaci u primjeni arhitektura te česta pojava kombiniranja više vrsta arhitekture. Ovim radom pokazalo se kako je kvalitetna arhitektura ključna u postizanju kvalitete aplikacije te se pokazalo za koje su situacije pojedine vrste arhitektura pogodne, a za koje ima prikladnijih rješenja.

Ključne riječi: React Native; arhitektura; mobilne aplikacije; SOLID principi; križno platformne aplikacije; Redux; MobX; MVC

Sadržaj

1. Uvod

U ovom radu obrađuje se tematika vezana za mobilne aplikacije, njihov razvoj i korištenje različitih arhitektura u sklopu njihovog razvoja. Neke od arhitektura koje će biti odabrane za teoretsku obradu i usporedbu biti će prikazane i na praktičnom primjeru na aplikaciji razvijenoj okruženjem React Native. U svrhu pisanja rada izrađena je i aplikacija na kojoj se obrađuje nekoliko spomenutih arhitektura. Arhitektura mobilne aplikacije je vrlo bitna stavka kod razvoja jer olakšava planiranje razvoja, programiranje aplikacije, a na kraju krajeva i performanse te kvalitetu izrađene aplikacije.

Svakodnevno se može primijetiti kako se današnji svijet uvelike oslanja na tehnologiju te ljudi imaju sve veće potrebe za raznoraznim alatima u obliku aplikacija. Postalo je normalno da većina ljudi konstantno drži mobilne uređaje u svojoj blizini te je stoga porasla i potražnja za mobilnim aplikacijama u širokom spektru područja. Ono što se nekada pisalo na papir ili u rokovnike danas je digitalizirano, a ono što je zahtijevalo pomno planiranje, analiziranje i promišljanje danas je također moguće digitalizirati. Sve većim napretkom tehnologije dolazi i do sve više mogućnosti digitalizacije i automatizacije određenih procesa, a samim time dolazi i do sve većih zahtjeva korisnika. Iz toga proizlazi činjenica da sve više raste potreba za skalabilnim, pouzdanim i održivim sustavima jer softveri postaju sve složeniji i zahtijevaju učinkovite arhitekture radi sve kompleksnijih operacija. Jedni od glavnih izazova u razvoju modernih aplikacija su upravljanje nezasitnim zahtjevima korisnika koji se mijenjaju iz dana u dan i čestim promjenama i novitetima u tehnologiji te iz tog razloga često dolazi i do potrebe za potrebama modifikacije sustava.

Planiranje i razvoj arhitekture softvera nije samo proces izrade, već temeljni korak u razvoju koji osigurava dugoročnu uspješnost aplikacije. Kvalitetna arhitektura osigurava bolju organizaciju koda, lakše održavanje, veću čitljivost i lakše proširivanje aplikacije. Jedan od glavnih ciljeva arhitekture je omogućiti aplikaciji da ostane fleksibilna, omogućujući jednostavne nadogradnje i promjene bez utjecaja na osnovne funkcionalnosti. Planiranje i razvoj arhitekture su nužni za postizanje brzine, efikasnosti i lake održivosti aplikacije kao i brojnih drugih prednosti. Iz tih razloga proizlazi i moja motivacija za odabir ove teme, uključujući i osobni interes za razvoj mobilnih aplikacija i implementaciju najboljih praksi u razvoju aplikacije, a samim time i razvoju odnosno planiranju arhitekture sustava.

Cilj ovog rada je istražiti i naučiti principe odabranih vrsta arhitektura, a samim time i pokazati važnost arhitekture u aplikacijama te prednosti i nedostatke nekih od odabranih arhitektura. U teoretskom dijelu ovog rada obradit će se uvod u mobilne aplikacije i njihova podjela po vrstama pristupu razvoju. Vrste pristupa razvoju važne su u kontekstu razvoja aplikacija okruženjem React Native jer su danas najzastupljenija vrsta aplikacija, a to su križno platformne, razvijane upravo navedenim okruženjem. Razlog zašto su križno platformne aplikacije najpopularnija vrsta aplikacija je to što su neovisne o platformi te samim time zauzimaju šire tržište, a da pritom ne zahtijevaju znatno više vremena za proces razvoja. U nastavku će se obraditi cjelokupni koncept arhitekture sustava, njena definicija i podjela na osnovne slojeve te važnost arhitekture i vrste korištene arhitekture u razvoju, a navesti će se i najvažniji faktori

odluke kod odabira vrste arhitekture koja će se koristiti u razvoju softvera. Nadalje, u kontekstu korištenja najboljih praksi u razvoju mobilnih i ostalih aplikacija navedeni su neki često korišteni principi i metode u razvoju. Principi koji se najviše povezuju sa okruženjem React Native su SOLID principi koji će također biti pobliže obrađeni i prikazani na primjerima u programskom kodu, točnije u Javascript-u. Nakon definiranja SOLID principa, navesti će se vrste arhitektura aplikacija razvijanih u okruženju React Native koje su odabrane za obradu i implementaciju na primjeru. U svrhu izrade teorijskog dijela, izrađen je i praktični dio na primjeru aplikacije za planiranje putovanja. Na navedenoj aplikaciji primijenit će se tri odabrane vrste arhitektura kako bi se usporedilo njihova svojstva i performanse.

2. Metode i tehnike rada

Za potrebe pisanja ovog diplomskog rada i izrade praktičnog dijela korišteni su razni alati i razni izvori literature. U sklopu izrade teorijskog dijela rada korišteni su znanstveni članci, knjige, blogovi softverskih inženjera, internet stranice i službena dokumentacija React Native okruženja i ostalih korištenih i obrađenih tehnologija i vrsta arhitektura. Kao što se spomenulo u uvodu, praktični dio rada odnosno aplikacija izrađena je korištenjem React Native razvojnog okruženja, dok je sama izrada bila popraćena alatima poput MS Visual Studio Code, Git/GitHub sustava za verzioniranje, Sourcetree sučelja za Git i SQLite baze podataka te sustava za upravljanje bazom podataka u obliku ekstenzije na okruženju MS VS Code. Teoretski dio ovog rada uglavnom je izrađen prema informacijama iz izvora literature, dok je za potrebe izrade praktičnog dijela izvor informacija pronađen u raznim tutorijalima za izradu aplikacija i korištenje određenih arhitektura na platformama YouTube i Udemy. U nastavku će se opisati najvažniji alati korišteni u radu.

2.1. Git i GITHub platforma

Git je distribuirani sustav za kontrolu verzija koji omogućava praćenje promjena u kodu tijekom vremena razvoja. Glavna svrha korištenja Git-a u programiranju je da olakša upravljanje verzijama projekata te da omogući pohranu povijesti promjena u projektima i time olakša otklanjanje pogrešaka. Prednost Git-a nad ostalim sustavima za verzioniranje je jednostavnost i intuitivnost sustava i njegovog rada te velika podrška zajednice što rezultira velikom količinom dostupnih materijala za pomoć. Sukladno sustavu Git, razvijena je i platforma GitHub koja služi za pružanje usluga pohrane Git repozitorija na oblaku (eng. hosting).

2.2. React Native

React Native je razvojni okvir otvorenog izvornog koda razvijen od strane Facebook-a za izradu mobilnih aplikacija pomoću programskog jezika JavaScript i razvojnog okvira React. Omogućava korištenje jedinstvenog koda za razvoj aplikacija za više platformi odnosno križno platformnih aplikacija. React Native prilično je jednostavan za naučiti s obzirom na to da koristi postojeće programske jezike koji su pokriveni mnoštvom materijala za učenje, posebice programskog jezika JavaScript, označnog jezika (eng. markup language) HTML-a te stilizacijskog jezika (eng. style sheet language) CSS-a.

2.3. Expo Go

Expo je platforma i set alata za brzi razvoj aplikacija razvijanih u okviru React Native. Omogućava jednostavno kreiranje, testiranje i postavljanje mobilnih aplikacija te eliminira potrebu za kompliciranim nativnim okruženjima poput XCode-a ili Android Studija. Vrlo je jednostavan za instalaciju i postavljanje te omogućava testiranje aplikacija u stvarnom vremenu na

fizičkim uređajima putem skeniranja QR koda.

2.4. SQLite baza podataka

SQLite je jednostavna baza podataka koja ne zahtijeva poslužitelje za rad što ju čini idealnom bazom za mobilne aplikacije. Za razliku od većih baza podataka poput MySQL-a ili PostgreSQL-a, SQLite je baza smještena izravno unutar aplikacije kao jednostavna datoteka što omogućuje jednostavnu implementaciju i korištenje lokalne pohrane. Smanjuje potrebu za mrežnim povezivanjem što ju čini neovisnom o vanjskim poslužiteljima te prigodnom bazom podataka za izvanmrežni (eng. offline) način rada. SQLite podržava transakcije i time slijedi principe atomičnosti, konzistentnosti, izolacije i trajnosti (eng. ACID - Atomicity, Consistency, Isolation, Durability) zbog čega čini pouzdan izbor za rad s osjetljivim podacima. SQLite čini vrlo popularnu bazu podataka za razvoj križno platformnih aplikacija zbog svoje pouzdanosti i efikasnosti te je zbog toga odabrana za korištenje u pratkičnom dijelu ovog rada.

2.5. Sourcetree

Sourcetree je grafičko korisničko sučelje za upravljanje Git repozitorijima koje olakšava korištenje sustava za kontrolu verzija. Eliminira potrebu za upotrebom naredbenog retka i time ubrzava verzioniranje te pruža vizualizaciju upravljanja repozitorijima, praćenja promjena, kreiranja grana i upravljanje promjenama unutar projekata.

Svi ovi alati su važni za moderan razvoj mobilnih aplikacija, a Git, GitHub i Expo omogućuju jednostavno verzioniranje i testiranje aplikacija. SQLite je ključan za jednostavnu i efikasnu lokalnu pohranu podataka, dok razvojni okvir React Native i integrirani razvojni okvir VS Code omogućuju brz i efikasan razvoj aplikacija. Implementirane arhitekture u praktičnom dijelu mogu se vidjeti na aplikaciji dostupnoj na sljedećoj poveznici:

https://github.com/HelenaPotocki/TravelPlannerApp.git

3. Mobilne aplikacije

Mobilne aplikacije su vrste programskih proizvoda koje su razvijane isključivo za korištenje na manjim uređajima kao što su pametni telefoni i tableti. U današnje doba mobilne aplikacije su dio ljudske svakodnevnice te se svaki korisnik pametnog telefona oslanja na različite mobilne aplikacije koje mu olakšavaju učestale radnje u aspektu osobnog života, komunikacije, socijalizacije, financija, zdravlja, poslovnog života te zabave.

Izrada mobilnih aplikacija svodi se na korištenje raznih programskih jezika, primjerice Java, Kotlin, Dart i Swift. U razvoju mobilnih aplikacija koriste se i tzv. razvojna okruženja, primjerice Android Studio, React Native, Flutter te Xcode. Programski jezici i razvojno okruženje koje će programer odabrati za izradu mobilnih aplikacija ovisi o potrebama aplikacije odnosno složenosti funkcionalnosti, svrsi aplikacije te o operacijskom sustavu uređaja na kojem će aplikacija biti instalirana.

Bitna stavka kod razvoja mobilnih aplikacija je vrsta operacijskog sustava na kojem će aplikacija biti instalirana. Vrsta operacijskog sustava povlači vrstu mobilne aplikacije iz razloga što pojedine vrste aplikacija mogu raditi na samo jednom operacijskom sustavu, dok druge vrste rade na svakom operacijskom sustavu ili više njih.

U nastavku poglavlja navesti će se vrste operacijskih sustava i pristupi razvoju mobilnih aplikacija.

3.1. Operacijski sustavi za mobilne uređaje

Operacijski sustav na mobilnom uređaju je ključan za rad mobilne aplikacije iz razloga što operacijski sustav pruža platformu na kojoj aplikacija radi i na kojoj se operacije aplikacije mogu izvršavati. Kako bi aplikacija koju programer razvija radila na određenom operacijskom sustavu, programer je dužan koristiti one tehnologije koje ciljani operacijski sustav podržava. U nastavku navest će se neki od najpopularnijih operacijskih sustava za mobilne uređaje.

3.1.1. Operacijski sustav Android

Prema Ali (2023.) Android je najviše korišten operacijski sustav za mobilne uređaje koji je trenutno na tržištu, a razvija ga Google. Koristi se na različitim vrstama mobilnih uređaja kao što su pametni telefoni, pametni satovi, televizori, tableti i automobili. Specifičnost operacijskog sustava Android je visoka fleksibilnost i prilagodljivost za programere te integracija sa Google-ovim uslugama, primjerice sa uslugom GoogleMaps. Aplikacije za Android uređaje uglavnom se razvijaju u programskom jeziku poput Jave ili Kotlina te se koristi Android SDK (eng. Software Development Kit) za pristup nekim funkcionalnostima uređaja poput Android API-ja (eng. Application Programming Interface - API).

3.1.2. Operacijski sustav iOS

Ali (2023.) navodi da je iOS operacijski sustav za mobilne uređaje izrađen isključivo za uređaje proizvođača Apple, dakle iPhone, iPad i iPod. Zatvorenog je koda te pruža izuzetno visoku sigurnost i stabilnost. Prilikom razvoja aplikacija za operacijski sustav iOS važno je pripaziti na korištene tehnologije u razvoju jer iOS ne podržava aplikacije razvijane u programskom jeziku Java/Kotlin, već isključivo programskim jezicima Swift i ObjectiveC. Uz to, podržava aplikacije razvijane križno platformnim razvojnim okvirima poput React Native-a, Flutter-a, Xamarin-a i još nekoliko drugih manje popularnih tehnologija, a za pristup iOS API-ju koristi se iOS SDK.

3.1.3. Ostali operacijski sustavi za mobilne uređaje

Postoji još mnogo operacijskih sustava koje bismo mogli spomenuti, kao što su Windows Mobile i BlackBerry OS. Posebnosti kod operacijskog sustava Windows Mobile su integracija s Microsoft-ovim uslugama te malo drugačije korisničko sučelje, dok kod BlackBerry OS-a postoji jedna specifična karakteristika, a to je fizička tipkovnica na većini modela mobilnih uređaja. Aplikacije za Windows Mobile obično se razvijaju u programskom jeziku C, a za Blackberry OS u Javi. Dakle, kod planiranja projekata razvoja mobilnih aplikacija važno je uzeti u obzir operacijski sustav uređaja za koje će aplikacija biti predviđena. U nastavku će se spomenuti različite vrste pristupa razvoju mobilnih aplikacija sukladno klasifikaciji mobilnih aplikacija.

3.2. Vrste pristupa razvoju mobilnih aplikacija

Sukladno operacijskom sustavu ili više njih za koje se predviđa aplikacija postoje i vrste pristupa razvoju mobilnim aplikacijama. Prema Oladele (2023.) pristupi razvoju mobilnih aplikacija su nativni ili izvorni pristup, križno platformni pristup, hibridni pristup i brzi pristup razvoju odnosno RMAD (eng. Rapid Mobile App Development). Postoji još vrsta pristupa razvoju mobilnih aplikacija, no navedene su najviše korištene te će biti opisane u nastavku.

3.2.1. Nativni pristup razvoju

Oladele (2023.) smatra da je nativni pristup razvoju korištenje programskih jezika specifičnih za platformu te softverskih razvojnih kompleta (eng. Software Development Kit – SDK) i razvojnih okruženja koje nude pružatelji operativnih sustava. Ukoliko se razvija aplikacija za operacijske sustave iOS i Android, nativnim pristupom razvija se aplikacija zasebno za svaku platformu koristeći potpuno različite tehnologije. Primjerice, za razvoj aplikacije za Android mogu se koristiti programski jezici Java, Kotlin ili neki drugi, a za razvoj aplikacije za operacijski sustav iOS mogu se koristiti programski jezici Swift ili Objective-C. Nativni pristup ima određene prednosti zbog kojih ga programeri nekad preferiraju, a to su sljedeće:

- Nativne aplikacije podržavaju sve dostupne značajke platforme i kompatibilnih uređaja

- Podržano je stvaranje jedinstvenog korisničkog iskustva s nativnim alatima
- Pruža bolje performanse i responzivnost
- Pruža potpuni pristup svim jedinicama hardvera
- Pruža veću sigurnost i pouzdanost.

Dakle, iako nativni pristup zahtijeva da se za svaki operacijski sustav razvije zasebna aplikacija iz čega proizlaze veći troškovi i veća količina potrebnog znanja programera, ponekad će u odluci odabira pristupa ovaj pristup biti preferiran radi korisničkih i aplikacijskih zahtjeva.

3.2.2. Križno platformni pristup razvoju

Prema Oladele (2023.), križno platformni pristup razvoju mobilnih aplikacija predstavlja razvoj aplikacija koje rade na različitim platformama. Smatra se vrlo dobrom alternativom nativnom pristupu jer rješava problem stvaranja zasebne aplikacije za svaku mobilnu platformu, odnosno omogućava isporuku aplikacije za više platformi istovremeno te se razvija korištenjem jezika i alata različitih od nativnih setova alata koje nude primjerice Google i Apple. U razvoju programeri koriste okvire i alate bazirane na JavaScriptu ili .NET/C#. Zbog jednostavnosti izvedbe projekta razvoja križno platformnih mobilnih aplikacija također se smanjuju i troškovi izrade, kao i potrebna znanja programera jer u ovom slučaju nema potrebe za više timova od kojih će svaki razvijati aplikaciju za drugu platformu, već je dovoljan jedan tim programera koji će raditi na jednoj tehnologiji odnosno na jednom skupu tehnologija. Prednosti u križno platformnom pristupu razvoju su sljedeće:

- Uniformnost na svim platformama
- Efikasnost u financijskom kontekstu
- Jednostavna implementacija
- Visoka demografska pokrivenost

Može se reći da je u današnje doba najviše korišten pristup upravo križno platformni iz razloga što je primjenjiv na sve vrste mobilnih uređaja neovisno o operacijskom sustavu, a s druge strane ne zahtijeva prevelike financijske investicije kao što je to slučaj kod nativnog pristupa.

3.2.3. Hibridni pristup razvoju

Hibridni pristup razvoju mobilnih aplikacija klasificira se kao oblik razvoja za više platformi, kao i križno platformni pristup. Oladele (2023.) navodi da se u hibridnom pristupu jezgra aplikacije razvija korištenjem standardnih tehnologija za razvoj web-a i alata kao što su JavaScript, CSS i HTML5, a zatim se taj kod izvršava unutar nativne ljuske. Konačne aplikacije

razvijane ovim pristupom imaju brzinu standardne aplikacije za web, a korisničko iskustvo slično bilo kojoj vrsti nativne mobilne aplikacije.

Kao i kod križno platformnog pristupa, smanjuju se troškovi izrade zbog potrebe za samo jednim timom jer nema potrebe za razvojem zasebnih aplikacija ovisno o platformi. No, iako oba pristupa razvoju rezultiraju aplikacijom namijenjenoj za više platformi, postoji jedna osnovna razlika. Križno platformni pristup razvoju zagovara korištenje alata za razvoj mobilnih aplikacija predviđenim za rad na više operativnih sustava, dok hibridni pristup razvoju koristi aplikacije za razvoj web-a poput HTML-a, CSS-a i JavaScripta te se obično razvijaju unutar razvojnih okvira kao što je Apache Cordova i Ionic. Sukladno tome, hibridne aplikacije se izvršavaju unutar komponente WebView što znači da aplikacija radi u web pregledniku.

Prednosti hibridnog pristupa razvoju mobilnih aplikacija su sljedeće:

- Smanjeni troškovi izrade
- Sposobnost korištenja hardverskih komponenti
- Jednaka korisnička iskustva nativnim mobilnim aplikacijama
- Mogućnost mrežnog rada (eng. online) i izvanmrežnog (eng. offline)

Finalni proizvod kod razvoja aplikacija hibridnim pristupom je aplikacija koja se pokreće u pregledniku, no kompatibilna je sa svim operacijskim sustavima pa pruža svojevrsnu konkurenciju križno platformnom pristupu. No, na kraju krajeva, ukoliko su zahtjevi takvi da je potrebno izraditi mobilnu aplikaciju neovisno o pregledniku, prednost ima križno platformni pristup.

3.2.4. Pristup brzog razvoja mobilnih aplikacija

Pristup brzog razvoja mobilnih aplikacija (eng. Rapid Mobile App Development, u daljnjem tekstu RMAD) uglavnom se koristi za razvoj aplikacija za više platformi u kratkom vremenskom razdoblju. Oladele (2023.) navodi da ovaj pristup uključuje korištenje specifičnih alata za razvoj bez koda ili sa smanjenom količinom koda (eng. Low Code Tools) za programiranje jednostavnih aplikacija za različita poslovna rješenja. Pristup RMAD je vrlo sličan metodologiji brzog razvoja koja se temelji na minimalnom planiranju, početnom prototipiranju, reciklirajućim softverskim komponentama i korištenju adaptivnog procesa. Princip RMAD pristupa je takav da se funkcionalnosti i značajke aplikacija deklariraju na front-end strani, dok back-end strana prevodi specifikacije u kod. Takvim načinom front-end koristi metapodatke da bi mogao funkcionirati i to na način da sažme glavne informacije o funkcijama aplikacije, primjerice upravitelji sredstava ili elementi korisničkog sučelja u bazi podataka. Time se eliminira potreba za kodiranjem baze podataka.

Iako se ovakav pristup koristi u specifičnim situacijama te nije preferiran pristup razvoju, ima i nekoliko prednosti, kao što su sljedeće:

- Veći povrat ulaganja

- Niska složenost u procesu razvoja
- Fleksibilnost u kontekstu vrsta projekata
- Ne zahtijeva puno znanja i iskustva programera u različitim kompleksnim tehnologijama
- Kod koji se može ponovno koristiti

RMAD pristup zahtijeva puno manje troškove izrade i brže iteracije, no iz razloga što je manje efikasan kod korištenja komponenti uređaja nije preferiran pristup razvoju.

S obzirom na to da se u ovom radu opisuju arhitekture mobilnih aplikacija te je odabrano okruženje React Native, pristup koji će se koristiti kod razvoja mobilne aplikacije je križno platformni te će se u skladu s tim pristupom i odabranom tehnologijom pojasniti arhitektura programskih proizvoda kao pojam te navesti moguće vrste arhitekture kod razvoja križno platformnih aplikacija u React Native-u.

4. Arhitektura programskog proizvoda

Postoji mnogo definicija arhitekture programskog proizvoda te će svaki programer reći nešto drugačije o tome što bi se trebalo ubrajati u arhitekturu i kako se naposljetku razvija arhitektura nekog proizvoda. Jedan od razloga zašto je to tako jest činjenica da se općenita arhitektura vrlo brzo mijenja zbog čestih promjena u zahtjevima sustava, sigurnosnim mjerama, dolasku novih naprednijih tehnologija itd. U zadnjih nekoliko godina pojam arhitekture uvelike se promijenio, a naročito dolaskom mikroservisa koji je kompletni pristup arhitekturi sveo na nešto sasvim drugačije od prethodnog. S jedne strane olakšao je naknadne promjene na sustavu i povećao izdržljivost sustava, a s druge strane unio je visok stupanj kompleksnosti upravljanja samom arhitekturom.

Kao i svaki složeni sustav, programski proizvod može se podijeliti na manje jedinice gdje svaka od njih ima određenu funkcionalnost tj. zadaću. Neke od njih skladište podatke, neke izvršavaju transakcije nad podacima, a neke brinu o tome da se korisniku sustava podaci prezentiraju na željeni način. Zadaća arhitekta programskog proizvoda je da rasporedi navedene funkcije na adekvatan način te da poveže jedinice kako bi one ispravno funkcionirale u jednoj složenoj cjelini.

Kako bi se laičko objašnjenje potkrijepilo stručnim definiranjem arhitekture, citira se Brown S. (2014.) koji navodi da je arhitektura zapravo struktura nastala dekompozicijom proizvoda u kolekciju manjih blokova i njihovih interakcija odnosno veza između tih blokova. Ta struktura mora uzeti u obzir cjelinu proizvoda na način da uključi temelje infrastrukturnih servisa koji rješavaju generalne probleme, primjerice sigurnost, konfiguracija, upravljanje greškama i sl. S druge strane, Bass, Clements i Kazman (2012.) navode arhitekturu kao strukturu svih struktura sustava koja objedinjuje sve programske elemente i njihova eksterno vidljiva svojstva te poveznice između njih.

4.1. Arhitektura mobilnih aplikacija

Prema Mistry P. (2024.) arhitektura mobilnih aplikacija opisuje kako je sustav složen od strukturnih komponenti i različitih setova sučelja, a također uključujući okruženje aplikacije, elemente i korisničko sučelje. Odabir adekvatne arhitekture mobilne aplikacije uključuje definiranje organizacije modula, način na koji se podaci spremaju i dohvaćaju, način na koji se kreira korisničko iskustvo (eng. user experience - UX) i kako su ti dijelovi međusobno povezani. Ističe se da važnost adekvatne arhitekture mobilne aplikacije osigurava skalabilnost, održivost i dobru izvedbu te poboljšava korisničko iskustvo. U kontekstu organizacija koje se bave razvojem mobilnih aplikacija, visoka kvaliteta arhitektura aplikacija osigurava smanjene troškove razvoja i održavanja te poboljšava upravljanje rizicima. Benefiti takve arhitekture u kontekstu same aplikacije su veća brzina i kvaliteta, kompatibilnost sa željenim sustavima i okruženjima, prilagodljivost i skalabilnost. S druge strane, loša arhitektura može dovesti do visokog izlaganja greškama, niske čitljivosti koda te izazova u razvoju i održavanju same aplikacije.

Vrlo važna stavka kod razvoja mobilnih aplikacija je razumijevanje da arhitektura mora

biti podijeljena na slojeve. Slojevita struktura arhitekture omogućava programerima brzo i jednostavno uklanjanje problema i grešaka bez modificiranja cijele aplikacije. Mistry (2024.) dijeli arhitekturu mobilnih aplikacija na tri sloja, a to su prezentacijski sloj, poslovni sloj i sloj podataka.

4.1.1. Prezentacijski sloj

Prema Mistry P. (2024.) glavni cilj prezentacijskog sloja je istražiti kako prezentirati odnosno dostaviti aplikaciju korisniku. Pritom je važna odgovornost programera da tijekom kreiranja prezentacijskog sloja odrede koji je tip klijenta prikladan za potrebnu infrastrukturu. Također je važno uzeti u obzir ograničenja u implementaciji klijenta kako i korištenje striktnih tehnika provjere podataka što je vrlo bitno kako bi se izbjegle nepotrebne greške kod unosa, primjerice unos valjane adrese elektroničke pošte (eng. e-mail) u adekvatnom formatu koji je potrebno poštivati.

4.1.2. Poslovni sloj

Mistry P. (2024.) navodi da poslovni sloj obuhvaća radne tokove, poslovne komponente i entitete ispod slojeva modela domene i usluge. Glavne odgovornosti poslovnog sloja uključuju upravljanje iznimkama, kreiranje predmemorije (eng. cache) podataka, logiranje odnosno vođenje evidencije aktivnosti i validaciju podataka. Dakle, poslovni sloj se fokusira na poslovnu logiku aplikacije i osigurava mogućnost instalacije aplikacije na određenu platformu za koju je aplikacija na kraju krajeva i razvijena. Iako od tri sloja arhitekture svaki ima svoju funkciju i aplikacija jednostavno ne može raditi bez jednog od njih, možemo reći da je poslovni sloj zapravo ključni dio arhitekture u aspektu integracije poslovne logike i efikasnog funkcioniranja te skaliranja aplikacije.

4.1.3. Podatkovni sloj

Prema Mistry P. (2024.), u podatkovni sloj spadaju svi nužni alati za manipulaciju podacima, agenti usluga (eng. service agents) i komponente pristupa koje su potrebne za rad podatkovnih transakcija. Na programerima je da razmotre načine održavanja podataka s ciljem da aplikacija bude prilagodljiva promjenama zahtjeva i potreba krajnjih korisnika. Ovaj sloj sastoji se od različitih alata, agenata usluga i komponenti koji su jedinstveni za podatke. Nadalje, postoje dvije kategorije u koje se ovaj sloj može svrstati, a to su mrežni sloj i sloj perzistencije. Mrežni sloj integrira komunikaciju putem mreže, izvještavanje o greškama i usmjeravanje, dok sloj perzistencije koristi aplikacijska programska sučelja za dohvaćanje podataka iz izvora podataka.

4.2. Faktori odluke kod odabira vrste arhitekture

Kako bi se prilikom razvoja mobilne aplikacije kreirala adekvatna arhitektura sustava potrebno je pripaziti na nekoliko faktora. Prema Mistry-ju (2024.) ti faktori su sljedeći:

- Analiza tipa uređaja na kojem će se aplikacija koristiti
- Internetski pristup i propusnost
- Korisničko sučelje

Kao što je spomenuto ranije, bitno je znati platformu na kojoj će se koristiti aplikacija, odnosno operacijski sustav, a isto tako je bitna i rezolucija ekrana odnosno responzivnost, količina dostupne memorije na uređaju te na kraju i okruženje u kojem će aplikacija biti implementirana. Ukoliko je za potrebe rada aplikacije potreban pristup internetu, valja imati na umu slučaj da se aplikacija može naći u situaciji slabe ili nikakve internetske veze te je stoga preporučljivo obratiti pažnju na propusnost prilikom planiranja arhitekture. U ovom slučaju smatra se da je vrlo korisna stvar pohranjivanje podataka u predmemoriju kako bi se smanjila potreba za ponovnim preuzimanjem podataka koji se preuzimaju svaki puta kada se aplikacija ili neka njena funkcionalnost pokrenu. Ono što je obično najbitnije krajnjem korisniku, osim ispravnog funkcioniranja aplikacije, jest izgled korisničkog sučelja. U to ubrajamo bilo kakav aspekt estetike aplikacije, poput gumba, izbornika, rasporeda svih komponenti i krajnji dizajn. Također, u sklopu tog dizajna najbitnija je stavka korisnička pristupačnost sučelja (eng. user-friendliness).

4.3. Načela i principi planiranja arhitekture

Kako bi programeri i arhitekti izradili dobar plan arhitekture moraju se držati nekih načela i principa koji osiguravaju ispunjenje ciljeva samog planiranja arhitekture, a to su primjerice povećanje kvalitete aplikacije, olakšano održavanje, skalabilnost, lakšu integraciju i testiranje, itd. Postoji mnogo načela koja prate najbolje prakse u razvoju aplikacija, a Kugell (2018.) navodi sljedeća načela i principe:

- SOLID principi
- DRY načelo (eng. Don't Repeat Yourself)
- YAGNI načelo (eng. You aren't gonna need it)
- KISS načelo (eng. Keep it short and simple)

SOLID principi biti će pojašnjeni u jednom od sljedećih poglavlja. DRY načelo naglašava da se informacije i logika ne smiju duplicirati te samim time podržava modularnost odnosno stvaranje funkcija ili modula koji se mogu iskoristiti na više mjesta unutar istog sustava. YAGNI načelo naglašava da bi se funkcionalnosti trebale dodavati u trenutku kada su zaista potrebne, a ne kada programer pretpostavi da će mu to trebati. Ovo načelo proizlazi iz nekih principa

agilne metodologije razvoja programskih proizvoda te potiče česte objave novih verzija temeljenih na promjenama zahtjeva korisnika, a krajnji cilj nije samo brza isporuka već i isporuka zadovoljstva. KISS načelo govori da bi programeri trebali održavati jednostavnost svake manje funkcionalnosti te izbjegavati nepotrebne kompleksnosti kako bi programski proizvod radio na najbolji i najbrži mogući način.

Postoji još često korištenih načela koja se koriste u razvoju aplikacija kako bi se izbjegle česte pogreške s neželjenim posljedicama, no daleko najprihvatljiviji i najpoznatiji principi su SOLID principi koji će biti pojašnjeni u sljedećem poglavlju.

5. SOLID principi u React Native okruženju

Kao što je spomenuto u prethodnom poglavlju, prilikom razvoja softvera, održavanja kvalitete, održivosti i skalabilnosti programeri se uvelike oslanjaju na neke standardne smjernice i principe dizajna. SOLID principi su najviše korištene smjernice te zauzimaju posebno mjesto u metodologijama razvoja aplikacija zbog sposobnosti da unaprijede strukturu i održavanje programskih proizvoda.

SOLID principi su grupa od pet smjernica za planiranje objektno orijentiranog programskog proizvoda te su osmišljeni kako bi olakšali razvoj i održavanje aplikacija. SOLID principi prvi puta su spomenuti od strane Roberta C. Martina također poznatog pod nadimkom „Uncle Bob“, a jedan je od najutjecajnijih softverskih inženjera i autora u području razvoja softvera. S obzirom na to da je posvetio karijeru promociji najboljih praksi u dizajnu softvera, razvio je smjernice kojima programeri mogu izbjeći uočene ponavljajuće obrasce problema. Spomenute smjernice prvi puta je objavio u knjizi „*Agile Software Development, Principles, Patterns, and Practices*“ .

Martin (2003.) navodi da primjena SOLID principa u razvoju aplikacija osigurava izgradnju fleksibilnog i skalabilnog programskog proizvoda sa smanjenom stopom grešaka, poboljšanom čitljivosti koda, lakšim testiranjem i bržim razvojem same aplikacije. U nastavku poglavlja navesti će se i detaljno objasniti svaki od 5 SOLID principa.

5.1. Princip samo jedne odgovornosti

Martin (2003.) navodi da princip samo jedne odgovornosti govori o tome kako bi jedna klasa morala imati isključivo jedan razlog za promjenu iz čega proizlazi da bi ta klasa morala imati isključivo jednu odgovornost. Ukoliko klasa ima više odgovornosti, primjerice zadatak izračunavanja površine nekog geometrijskog lika te crtanje istog, ta se klasa mora razdvojiti na dvije klase sa pripadajućim metodama za izvršenje zadatka. Razlog takvog načina programiranja klasa leži u činjenici da ukoliko dolazi do promjene u nekoj od odgovornosti klasa, odnosno u načinu rada nekih metoda, vrlo je visoka mogućnost da će promjenom nad jednom odgovornošću omesti izvršenje druge odgovornosti te će sustav vraćati grešku i odgovornost se neće moći izvršavati. U nastavku slijedi isječak programskog koda na kojem je primijenjen princip samo jedne odgovornosti.

```
const DetaljiVozila = ({ vozilo }) => {
  return (
    <View>
      <Text>{vozilo.naziv}</Text>
      <Text>{vozilo.tip}</Text>
      <Text>{vozilo.snagaMotora} kW</Text>
    </View>
  );
};
```

```
const dohvatiVozila = () => {
  // logika za dohvacanje podataka o svim vozilima ili filtriranim
};

const ListaVozila = () => {
  const [vozila, setVozila] = useState([]);

  useEffect(() => {
    dohvatiVozila().then(data => {
      setVozila(data);
    }).catch(error => {
      console.error(error);
    });
  }, []);

  return (
    <FlatList
      data={vozila}
      keyExtractor={(item) => item.id.toString()}
      renderItem={({ item }) => <DetaljiVozila vozilo={item} />}
    />
  );
};
```

Programski kod 1: Primjer korištenja principa samo jedne odgovornosti (vlastita izrada)

U navedenom primjeru može se vidjeti ispravna implementacija principa samo jedne odgovornosti. Dakle, postoje dvije funkcionalne komponente naziva *DetaljiVozila* i *ListaVozila* te postoji funkcija *dohvatiVozila*. *DetaljiVozila* služe za prikaz jednog zapisa vozila dok *ListaVozila* služi za poziv metode za dohvaćanje vozila te prikaz dohvaćene liste vozila što je implementirano objektom naziva *Flatlist* koji za prikaz elemenata liste koristi funkcionalnu komponentu *DetaljiVozila* te prosljeđuje jedan po jedan zapis u listi vozila. Funkcija *dohvatiVozila* sadrži logiku za dohvaćanje liste vozila iz baze podataka. Neispravan kod koji krši princip samo jedne odgovornosti bio bi da se u jednoj funkcionalnoj komponenti istovremeno implementira funkcija za dohvaćanje liste vozila iz baze podataka, prikaz liste vozila i komponenta za prikaz jednog zapisa vozila.

5.2. Princip otvorenosti i zatvorenosti

S obzirom na to da jedna promjena u kodu može rezultirati cijelim vodopadom promjena u konkretnom modulu, savjetuje se da se sustav refaktorira na način da bilo kakva promjena ne rezultira potrebom za modifikacijom drugih dijelova sustava. Prema Martinu (2003.) princip otvorenosti i zatvorenosti rješava takav problem sa svoja dva podnačela, a to su otvorenost za ekstenzije odnosno proširenja i zatvorenost za modifikacije. Primjenom tog principa bilo kakve modifikacije se vrše na način da se dodaje novi kod umjesto da se mijenja stari kod koji već ispravno radi.

U kontekstu otvorenosti prema ekstenzijama odnosno proširenjima mijenja se ponaša-

nje modula. Ukoliko je došlo do promjena zahtjeva aplikacije, moduli se mogu proširiti novim ponašanjima koja zadovoljavaju promjene u zahtjevima. S druge strane, zatvorenost za promjene znači da spomenuto proširenje modula nikako ne rezultira promjenama u izvornom ili binarnom kodu modula te ti dijelovi ostaju netaknuti.

Način na koji se postiže princip otvorenosti i zatvorenosti je kroz upotrebu apstrakcije kako bi se omogućilo proširivanje funkcionalnosti modula bez izmjene izvornog koda. Primjerice, u objektno orijentiranom programiranju u React Native-u, apstrakcija se implementira korištenjem sučelja ili apstraktnih klasa gdje se definiraju setovi metoda ili svojstava koje će svaka komponenta morati implementirati. U nastavku slijedi isječak programskog koda na kojem je primijenjen princip otvorenosti i zatvorenosti.

```
class BazaVozila {
  constructor(db) {
    this.db = db;
  }

  dodajVozilo(vozilo) {
    this.db.transaction(tx => {
      tx.executeSql('INSERT␣INTO␣vozila␣(ime,␣tip)␣VALUES␣(?,␣?)', [vozilo.ime,
          vozilo.tip]);
    });
  }
}

class ProsirenaBazaVozila extends BazaVozila {
  ukloniVozilo(idVozila) {
    this.db.transaction(tx => {
      tx.executeSql('DELETE␣FROM␣vozila␣WHERE␣id␣=␣?', [idVozila]);
    });
  }
}
```

Programski kod 2: Primjer korištenja principa otvorenosti i zatvorenosti (vlastita izrada)

U prethodnom primjeru koda prikazana je implementacija principa otvorenosti i zatvorenosti. Postoji klasa *BazaVozila* koja sadrži funkciju *dodajVozilo* za dodavanje vozila te proširenje te klase pod nazivom *ProsirenaBazaVozila*. Ukoliko želimo proširiti klasu *BazaVozila* te joj dodati funkcionalnost brisanja iz baze vozila, potrebno je kreirati ekstenziju odnosno proširenje klase te se u tom proširenju mogu navesti nove klase koje mijenjaju inicijalno ponašanje modula. Prema tome se u klasu *ProsirenaBazaVozila* dodala nova metoda *ukloniVozilo* koja služi za uklanjanje vozila iz baze podataka. Neispravan kod koji ne prati princip otvorenosti i zatvorenosti bio bi ubacivanje funkcije *ukloniVozilo* u klasu *BazaVozila* što bi kršilo podnačela "otvorenost za proširenja, zatvorenost za modifikacije".

5.3. Liskovin princip zamjene

Kako se ranije spomenulo, primarni mehanizmi kod principa otvorenosti i zatvorenosti su apstrakcija i polimorfizam koji se u većini programskih jezika implementiraju nasljeđivanjem. Međutim, prilikom implementacije nasljeđivanja dolazi se do pitanja koja su pravila u toj implementaciji, koje su karakteristike učinkovite hijerarhije nasljeđivanja i kako izbjeći nesukladnost te hijerarhije sa principom otvorenosti i zatvorenosti. To je problematika kojom se bavi Liskovin princip zamjene.

Martin (2003.) navodi da Liskovin princip govori o tome da podklase moraju biti zamjenjive za svoje nadklase. Drugim riječima, ako postoji klasa A koja je nadklasa klase B, instanca klase B mora moći zamijeniti instancu klase A bez narušavanja ispravnosti programa. U nastavku prikazan je isječak koda na kojem je primijenjen Liskovin princip zamjene, nadovezujući se na princip otvorenosti i zatvorenosti odnosno primjer korištenja istog.

```
const izvrsiOperacije = (bazaVozila) => {
  bazaVozila.dodajVozila({ naziv: 'Harley Davidson', tip: 'Motocikl' });
};

const db = SQLite.openDatabase('vozila.db');
const bazaVozila = new BazaVozila(db);
const prosirenaBazaVozila = new ProsirenaBazaVozila(db);

izvrsiOperacije(bazaVozila);
izvrsiOperacije(prosirenaBazaVozila);
```

Programski kod 3: Primjer korištenja Liskovinog principa zamjene (vlastita izrada)

U navedenom primjeru prikazuje se implementiran Liskovin princip zamjene. Isječak koda u ovom primjeru nadovezuje se na primjer naveden za prethodni princip odnosno princip otvorenosti i zatvorenosti u kojem se klasa *BazaVozila* proširila klasom *ProsirenaBazaVozila*. Kako je i spomenuto u pojašnjenju Liskovinog principa zamjene, može ga se shvatiti kao nadovezivanje na princip otvorenosti i zatvorenosti jer, ako se već implementiralo proširenje neke postojeće klase, onda bi ta implementacija trebala rezultirati sukladnošću klase i njezine apstrakcije. U ovom slučaju, ako postoji instanca klase *BazaVozila* i instanca klase *ProsirenaBazaVozila*, funkcija *izvršiOperacije* mora moći primiti obje navedene instance kao ulazni argument i izvesti operaciju iz one klase koja je u ovom slučaju klasa roditelj, a to je klasa *BazaVozila*.

5.4. Princip razdvajanja sučelja

Kako Martin (2003.) navodi, princip razdvajanja sučelja temelji se na nedostacima „debelih“ sučelja, odnosno sučelja koja imaju nisku razinu kohezivnosti. U ovom kontekstu, kohezivnost se odnosi na stupanj u kojem elementi unutar sučelja rade zajedno kako bi izvršili krajnji zadatak. Sučelja klase bi se prema ovom principu trebala podijeliti na skupine metoda gdje svaka skupina služi različitim vrstama klijenata (korisnika klase). U praksi postoje objekti koji zahtijevaju nekohezivna sučelja, međutim princip razdvajanja sučelja sugerira da klijenti

ne bi trebali znati za njih kao jednu klasu već za apstraktne bazne klase koje imaju kohezivna sučelja. Prednosti kohezivnosti su lakše održavanje koda, povećana ponovna iskoristivost i smanjena kompleksnost što kod čini preglednijim i jednostavnijim za čitanje. U nastavku slijedi isječak programskog koda s implementiranim principom razdvajanja sučelja.

```
class IBazaVozila {
  dodajVozilo(vozilo) {}
}

class IIzvjestajVozila {
  generirajIzvjestaj() {}
}

class BazaVozila extends IBazaVozila {
  constructor(db) {
    super();
    this.db = db;
  }

  dodajVozilo(vozilo) {
    this.db.transaction(tx => {
      tx.executeSql('INSERT␣INTO␣vozila␣(naziv,␣tip)␣VALUES␣(?,␣?)', [vozilo.naziv,
          vozilo.tip]);
    });
  }
}

class IzvjestajVozila extends IIzvjestajVozila {
  generirajIzvjestaj(vozila) {
    // Logika za generiranje izvjestaja
  }
}

const db = SQLite.openDatabase('vozila.db');
const bazaVozila = new BazaVozila(db);
```

Programski kod 4: Primjer korištenja principa razdvajanja sučelja (vlastita izrada)

U navedenom primjeru može se vidjeti implementacija principa razdvajanja sučelja gdje svaka klasa implementira samo one metode koje su relevantne za njihove uloge. Postoji klasa *BazaVozila* koja implementira sučelje *IBazaVozila* i sadrži metodu za dodavanje vozila te time ispunjava svoju ulogu, a to je upravljanje podacima o vozilima u bazi podataka. S druge strane, postoji klasa *IzvjestajVozila* koja implementira sučelje *IIzvještajVozila* te sadrži metodu za generiranje izvještaja i ispunjava svoju ulogu upravljanja izvještajima vezanim za podatke o vozilima. Kršenje principa razdvajanja sučelja bila bi implementacija jednog sučelja koje definira metodu dodavanja vozila i metodu generiranja izvještaja.

5.5. Princip inverzije ovisnosti

Martin (2003.) navodi da princip inverzije ovisnosti govori o tome kako moduli više razine ne smiju ovisiti o modulima niže razine, već bi moduli općenito trebali ovisiti o apstrakcijama. Apstrakcije također ne bi smjele ovisiti o detaljima, već bi detalji trebali ovisiti o apstrakcijama. Rezultat implementacije ovog principa su lakše zamjene i proširenje komponenti unutar sustava bez potrebe za promjenama modula više razine i apstrakcijama. U nastavku slijedi isječak koda sa implementacijom principa inverzije ovisnosti.

```
class IBazaVozila {
  dodajVozilo(vozilo) {}
}

class Vozilo {
  constructor(bazaVozila) {
    this.bazaVozila = bazaVozila;
  }

  dodajVozilo(vozilo) {
    this.bazaVozila.dodajVozilo(vozilo);
  }
}

class SqliteBazaVozila extends IBazaVozila {
  constructor(db) {
    super();
    this.db = db;
  }

  dodajVozilo(vozilo) {
    this.db.transaction(tx => {
      tx.executeSql('INSERT␣INTO␣vozila␣(naziv,␣tip)␣VALUES␣(?,␣?)', [vozilo.naziv,
          vozilo.tip]);
    });
  }
}

//koristenje:
const db = SQLite.openDatabase('vozila.db');
const sqliteBazaVozila = new SqliteBazaVozila(db);
const vozilo = new Vozilo(sqliteBazaVozila);

vozilo.dodajVozilo({ naziv: 'Harley␣Davidson', tip: 'Motocikl' });
```

Programski kod 5: Primjer korištenja principa inverzije ovisnosti (vlastita izrada)

U navedenom isječku koda može se vidjeti implementacija principa inverzije ovisnosti. Klasa *Vozilo* ovisi o apstrakciji *IBazaVozila*, a ne o specifičnoj implementaciji odnosno klasi *SqliteBazaVozila* koja implementira sučelje *IBazaVozila*. U korištenju može se primijetiti kako se u inicijalizaciji instance klase *Vozilo* prosljeđuje instanca klase *SqliteBazaVozila* što znači da

tada klasa *Vozilo* u svojoj metodi *dodajVozilo* poziva metodu *dodajVozilo* iz specifične implementacije *SqliteBazaPodataka*.

Kako bi razvoj arhitekture i razvoj same aplikacije prošao čim efikasnije te kako bi finalni proizvod bio lako održiv, važno je slijediti navedene SOLID principe. Njihovom primjenom dolazi se do izdržljive, održive i efikasne aplikacije što je na kraju krajeva cilj svakog projekta razvoja mobilne aplikacije, a i ostalih programskih proizvoda. Najvažniji cilj koji SOLID principi postižu jest modularnost pojedinih dijelova što je najvažnija karakteristika kvalitetnog softvera. Neke od arhitektura koje će se kasnije navesti i pojasniti već u svojoj strukturi primjenjuju neke od SOLID principa, a u praktičnom dijelu ovog rada izradit će se aplikacija slijedeći sve navedene SOLID principe.

6. Arhitekture React Native aplikacija

Razvoj mobilnih aplikacija doživio je značajan napredak s pojavom React Native okruženja koje je razvio Facebook. React Native omogućava programerima korištenje JavaScript jezika i React-a za izgradnju križno platformnih aplikacija te samim time nudi prednost korištenja jednog koda za više platformi. Ovakav pristup ubrzava razvoj aplikacija te olakšava njihovo održavanje jer se promjene u zahtjevima aplikacije mogu ažurirati odjednom za sve platforme. Da bi se moglo razumjeti kako React Native zapravo funkcionira glede križno platformnog pristupa razvoju, važno je razumjeti podjelu tog okruženja na različite slojeve. Pritom govorimo o tri sloja koja prema Mehta (2018.) čine arhitekturu React Native-a, a to su sljedeći:

- Sloj JavaScript
- Sloj most (eng. Bridge)
- Izvorni sloj (eng. Native)

Sloj JavaScript je jezgra React Native-a gdje programeri pišu svoj kod prilikom razvoja aplikacija koristeći mogućnosti React-a za stvaranje deklarativnog korisničkog sučelja baziranog na komponentama. Ovaj sloj upravlja logikom aplikacije, stanjem i interakcijama, a također se bavi problematikom ažuriranja korisničkog sučelja. Za učinkovito ažuriranje korisničkog sučelja React Native koristi virtualni DOM (eng. Document Object Model) kao što React koristi za web aplikacije, no u kontekstu React Native-a koristi se „shadow tree“ koji je vrlo sličan virtualnom DOM-u. "Shadow tree" označava strukturu koja predstavlja raspored korisničkog sučelja u hijerarhiji prikaza, a radi na način da izračunava pozicije i veličine elemenata prije nego ih prikaže na zaslonu. Taj koncept uvelike utječe na performanse i brzinu aplikacije jer prilikom interakcije korisnika s aplikacijom nije potrebno provoditi niz promjena u korisničkom sučelju, već se provode samo one promjene koje su zaista potrebne te proizlaze iz konkretne akcije korisnika.

Sloj most djeluje kao komunikacijski centar između sloja JavaScript i izvornog sloja. Omogućava besprijekornu interakciju JavaScript koda s nativnim modulima i omogućava programerima korištenje temeljne platforme. Ovaj sloj je najbitniji faktor u postizanju performansi te pristupu specifičnim funkcionalnostima uređaja, a da ujedno zadržava fleksibilnost i jednostavnost razvoja u JavaScriptu.

Nativni sloj ili izvorni sloj je mjesto gdje se nalazi kod specifičan za platformu. Primjerice, za operacijski sustav iOS pisan je u programskom jeziku Swift ili Objective-C, a za operacijski sustav Android pisan je u Javi ili Kotlinu. React Native ne apstrahira izvorni kod već ga prihvaća takvog kakav je. U prijevodu, to bi značilo da programeri prilikom razvoja aplikacije po potrebi mogu izravno pristupiti izvornom kodu i funkcionalnostima specifičnim za platformu, bez da se taj isti kod skriva ili pojednostavljuje do te mjere da je generalno neprepoznatljiv. Ovaj sloj dakle osigurava da aplikacija na određenom uređaju izgleda i „osjeća“ se izvorno korištenjem komponenti i ponašanja specifičnih za platformu na kojoj se nalaze. Također, radi spomenutog izbjegavanja apstrakcije izvornog koda, programerima je omogućena integracija izvornih

modula u svoju aplikaciju omogućujući time pristup funkcionalnostima specifičnim za uređaj i konkretnu platformu.

Kao što se spomenulo u prethodnim poglavljima ovog rada, odabir adekvatne arhitekture sustava kod razvoja mobilnih aplikacija je nužan za osiguranje kvalitete, izdržljivosti, skalabilnosti i niskih troškova održavanja aplikacije. Arhitektura aplikacija razvijanih u React Native-u ima ključnu ulogu u kvaliteti aplikacije s obzirom na to da različite arhitekture nude različite pristupe upravljanju stanjem, rukovanjem podacima i komunikacijom između komponenti.

Prema Mehta (2018.) najpoznatiji i najviše korišteni arhitekturalni uzorci u React Native-u koji slijede prethodno navedene principe su sljedeći:

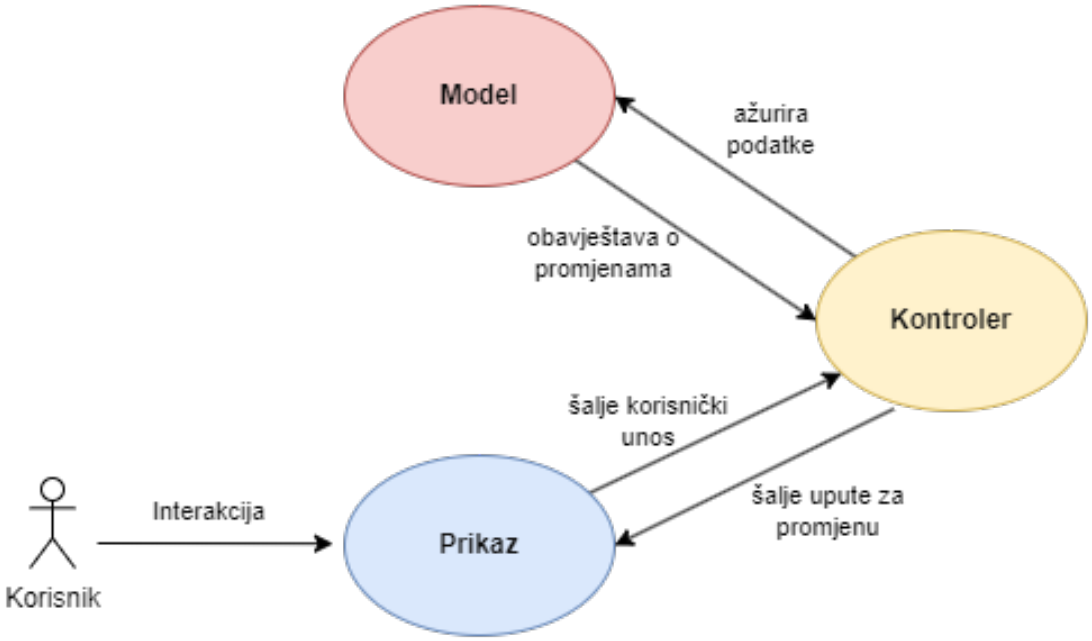
- MVC – (Model – View – Controller)
- MVVM (Model – View – ViewModel)
- Flux
- Redux
- MobX

U nastavku će se opisati i predstaviti svaki od arhitekturalnih uzoraka te će se navesti njihove prednosti naspram ostalih.

6.1. MVC (Model - View - Controller)

Kako navodi Mehta (2018.), uzorak arhitekture Model – View – Controller temelji se na odvajanju podatkovnog sloja odnosno modela, (eng. Model), prezentacijskog sloja odnosno prikaza (eng. View) i sloja logike odnosno kontrolera (eng. Controller). Ova vrsta arhitekture pogodna je za veće aplikacije gdje se može vidjeti njena učinkovitost kod razdvajanja odgovornosti, dok za manje i srednje aplikacije MVC arhitektura predstavlja preveliku kompleksnost iz razloga što odvajanje odgovornosti može biti preopterećujuće za manje sustave. Kumar (2024.) navodi da se za podatkovni sloj mogu integrirati stanja Redux, komponente React Native-a za prezentacijski sloj i funkcije za upravljanje korisničkim unosima i ažuriranju podatkovnog sloja kod implementacije sloja logike.

Na slici 1. prikazan je dijagram koji opisuje koje komponente postoje u MVC modelu i kako one međusobno komuniciraju. Model obavještava kontrolera o promjenama koje su se dogodile u podacima, a zatim kontroler šalje prikazu upute za promjenu podataka kako bi se ispravni podaci prikazali korisniku. Ukoliko korisnik napravi neku interakciju sa prikazom, primjerice klik na gumb, tada prikaz šalje kontroleru korisnički unos te kontroler zatim ažurira podatke u modelu, odnosno pozove funkciju u modelu i proslijedi podatke koji će se mijenjati te nove vrijednosti.



Slika 1: Dijagram MVC arhitekture (Izvor: **Eggenspieler**, **2022.**)

Prednosti korištenja vrste arhitekture MVC u React Native-u ("*MVC*", bez dat.):

- Razdvajanjem na komponente povećava se fleksibilnost, održivost i skalabilnost aplikacije
- Mogućnost testiranja slojeva odvojeno jedan od drugog zbog njihove jasne razdvojenosti
- Modeli mogu imati više pogleda
- Mogućnost konfiguracije različitih razina sigurnosti za različite slojeve.

Iako je MVC široko prihvaćena vrsta arhitekture i jedna je od najpopularnijih vrsta arhitektura zahvaljujući svojim prednostima, no ima i određene nedostatke koji ju čine neprikladnom za implementaciju na određenim sustavima. Primjeri nedostataka arhitekture MVC su sljedeći:

- Nije pogodna za male aplikacije iz razloga što manje sustave čini kompleksnijima nego što bi trebali biti
- Zbog slojevite arhitekture može biti sporiji u aplikacijama koje zahtijevaju visoke performanse što rezultira kašnjenjem u radu aplikacije

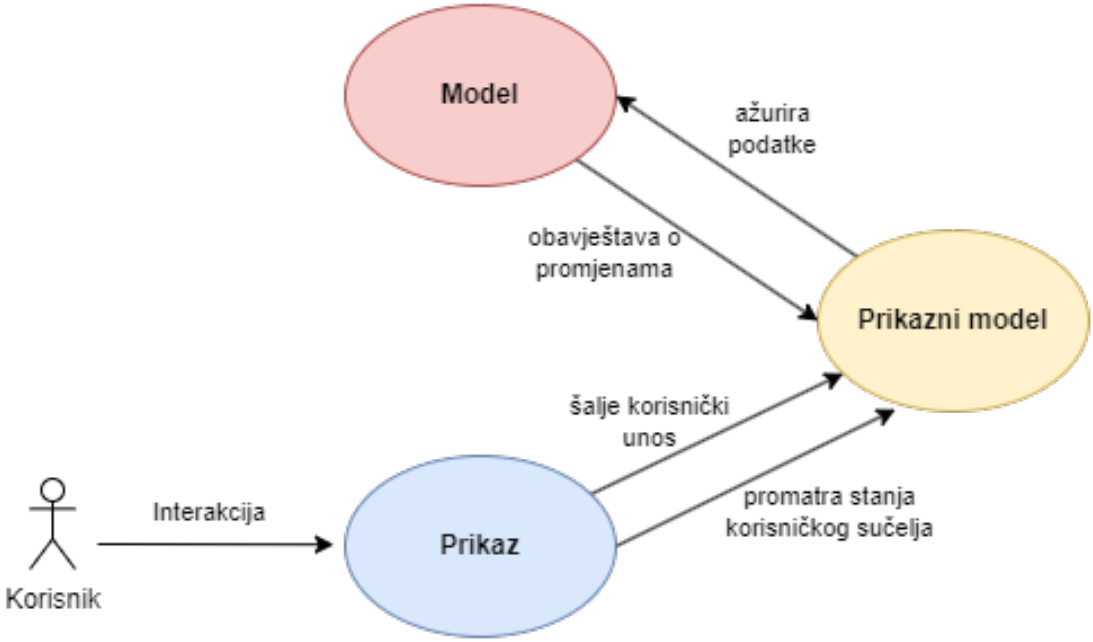
Vrsta arhitekture MVC donosi mnoge koristi, ali nije univerzalno rješenje za sve vrste aplikacija. Prikladnost upotrebe ovisi o specifičnim zahtjevima projekta, veličini aplikacije i potrebama za skalabilnošću ili performansama.

6.2. MVVM (Model - View - ViewModel)

Prema Mehta (2018.), uzorak arhitekture Model – View – ViewModel (MVVM) je svojevrsna nadogradnja na MVC uzorak, no MVVM koristi sloj ViewModel kao posrednika između View

i Model slojeva. Dodatni sloj ViewModel omogućava olakšano testiranje jer razdvaja poslovnu logiku od korisničkog sučelja što omogućava izolirano testiranje poslovne logike bez potrebe za prikazom korisničkog sučelja. Također, MVVM model pojednostavljuje povezivanje podataka automatiziranjem sinkronizacije između stanja aplikacije i korisničkog sučelja. U okruženju React Native postoje biblioteke koje koriste MVVM principe, primjerice biblioteka naziva React Native Paper.

Na slici 2. prikazan je dijagram arhitekture MVVM gdje možemo vidjeti kako komponente zapravo rade. Može se primijetiti kako je struktura više manje jednaka MVC strukturi, no s jednom promjenom, a to je veza između prikaza i prikaznog modela. Prikazni model koji u MVVM arhitekturi služi kao posrednik između modela i prikaza, više ne mora prikazu slati upute o promjeni podataka odnosno ne mora ga ručno obavještavati o tome, već prikaz prati promjene u prikaznom modelu. Praćenje promjena u prikaznom modelu najčešće se postiže pretplatom na događaje.



Slika 2: Dijagram MVVM arhitekture (Izvor: **Eggenspieler**, **2022.**)

Prema Garcia Gallardo (2023.) prednosti MVVM arhitekture su sljedeći:

- Lakši razvoj sustava zbog razdvajanja pogleda od poslovne logike što je bitno u timskoj podjeli rada na razvoju sustava.
- Lakše testiranje iz razloga što su model-pogled i model potpuno neovisni o pogledu, odnosno svaki od njih se može testirati posebno. Iz toga također proizlazi i lakše otklanjanje pogrešaka.
- Lakše održavanje koje proizlazi iz jasne razdvojenosti između različitih slojeva što čini programski kod čišćim i jednostavnijim za razumijevanje i implementaciju promjena.

Nedostaci vrste arhitekture MVVM koji dolaze do izražaja u određenim situacijama su sljedeći:

- U manjim aplikacijama, kao i MVC vrsta arhitekture, može rezultirati prevelikom kompleksnošću
- U velikim sustavima može rezultirati vrlo kompliciranom implementacijom iz razloga što je dizajniranje sloja model-pogled sa pravom razinom generalizacije može biti izazovno.
- Teško otklanjanje pogrešaka zbog deklarativne prirode kod vezivanja podataka.

MVVM vrsta arhitekture na neki način poboljšava MVC arhitekturu prebacivanjem dijela odgovornosti s modela na sloj model-pogled, no ipak sama složenost arhitekture MVVM može izazvati probleme i poteškoće u razvoju i održavanju sustava u većim, a također i manjim projektima.

6.3. Arhitektura Flux

Prema Weerakoon-u (2023.) Flux je uzorak arhitekture koji je prvi puta predstavljen 2014.godine od strane Facebook-a za upravljanje podatkovnim tokovima u kompleksnim aplikacijama za web. Bazirana je na ideji jednosmjernog toka podataka što rezultira čistom aplikacijskom strukturom vrlo jednostavnom za održavanje i ponajprije s lakšim otklanjanjem pogrešaka. Arhitektura Flux je osmišljena kako bi riješila neke od problema koji se javljaju u arhitekturi MVC, primjerice uske povezanosti i složenog upravljanja stanjem. U kontekstu uske povezanosti misli se na čvrstu povezanost između dijelova sustava odnosno komponenata što rezultira i većom ovisnošću komponenata jedna o drugoj. Problem se dakle javlja radi teške održivosti koda u takvim aplikacijama jer eventualne promjene u kodu moraju biti pažljivo usklađene između različitih dijelova aplikacija što zahtijeva mnogo vremena i resursa, a arhitektura Flux pojednostavljuje taj specifičan problem i pruža programerima lakše razumijevanje i bolju čitljivost koda.

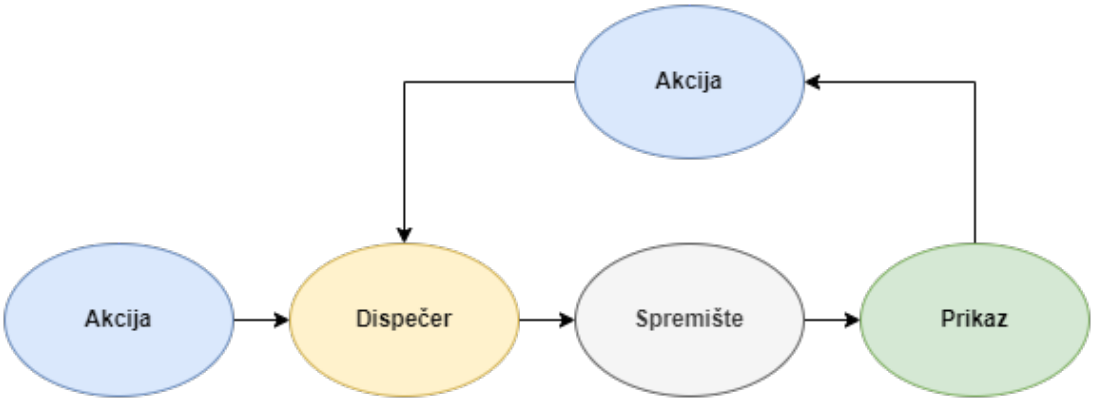
Uzorak arhitekture Flux bazira se na jednostavnoj strukturi od 4 glavne vrste komponenti:

- Akcija (eng. Action)
- Dispečer (eng. Dispatcher)
- Spremište (eng. Store)
- Pogled (eng. View)

Na slici 3. može se vidjeti kako izgleda struktura arhitekture Flux. Akcije su objekti koji predstavljaju događaje koji se izvrše u aplikaciji te ti objekti sadržavaju podatke koji se trebaju proslijediti iz pogleda u spremište. Primjerice, korisnik klikne na gumb te se šalje akcija kako bi

se ažurirali podaci u spremištu. Dispečer (eng. dispatcher) je središnje mjesto koje prima akcije iz pogleda i prosljeđuje ih u spremišta. Dispečer djeluje kao posrednik poruka i osigurava da se prave akcije i podaci šalju u odgovarajuća spremišta. Pogledi su komponente koje prikazuju korisničko sučelje te primaju stanje iz spremišta te prosljeđuju korisničke interakcije nazad u spremište koristeći akcije.

Važno je spomenuti da za razliku od ranije spomenutih arhitektura MVC i MVVM, arhitektura Flux više pažnje posvećuje upravljanju stanjem aplikacije što je u mobilnim aplikacijama vrlo bitno jer upravljanje stanjem ima veliki utjecaj na performanse aplikacije, korisničko iskustvo i lakoću održavanja. S druge strane, manje pažnje posvećuje podatkovnom sloju, odnosno nema nikakva striktna pravila o tome kako bi se trebalo komunicirati s bazom podataka ili serverom.



Slika 3: Dijagram Flux arhitekture (Izvor: **Weerakoon**, **2023.**)

Prednosti korištenja Flux uzorka arhitekture su sljedeće:

- Jednosmjerni protok podataka koji olakšava razumijevanje toka podataka u kodu i pojednostavljuje otklanjanje pogrešaka
- Bolje upravljanje stanjem zbog mogućnosti centralizacije spremišta što osigurava konzistentno i predvidljivo ponašanje aplikacije i također olakšava ispravljanje pogrešaka i održavanje aplikacije
- Poboljšana skalabilnost aplikacije zbog dijeljenja aplikacije na manje i lakše upravljive dijelove
- Bolja organizacija koda iz razloga što arhitektura Flux pruža vrlo jasnu razliku između akcija, spremišta i pogleda što olakšava organizaciju koda na komponente i lakše razumijevanje što svaka od njih radi
- Bolje i lakše testiranje odnosno olakšano pisanje automatiziranih testova što je također rezultat korištenja jednosmjernog protoka podataka i centraliziranog upravljanja stanjem.

Nedostaci korištenja arhitekture FLUX u aplikacijama razvijanim okruženjem React Native su sljedeći:

- Kompleksnost u implementaciji koja rezultira sporijim tempom učenja i razumijevanja sustava te većoj složenosti koda
- Pojava *Boilerplate* koda za postavljanje i upravljanje svim komponentama arhitekture
- Naspram drugih arhitektura kao što su MobX i Redux, arhitektura FLUX ima vrlo slabu pokrivenost s pomoćnim alatima i bibliotekama za upravljanje stanjem te iz tog razloga zahtijeva dodatni napor za integraciju i održavanje sustava
- Teže otklanjanje pogrešaka i teže testiranje zbog složenosti toka podataka koji korištenjem drugih arhitektura može biti jednostavniji
- Smanjena fleksibilnost, posebno kod složenijih i dinamičnih zahtjeva.

Iako Flux arhitektura pruža strukturiran i predvidljiv pristup upravljanju stanjem u aplikacijama te omogućava jasne tokove podataka, može doći do nepotrebne složenosti u velikim i kompleksnijim aplikacijama. Iako Flux nudi kvalitetnu osnovu za upravljanje stanjem, za specifične situacije postoje bolje opcije te je važno razmotriti nedostatke ove arhitekture kako bi njena eventualna implementacija ispunila svoj potencijal.

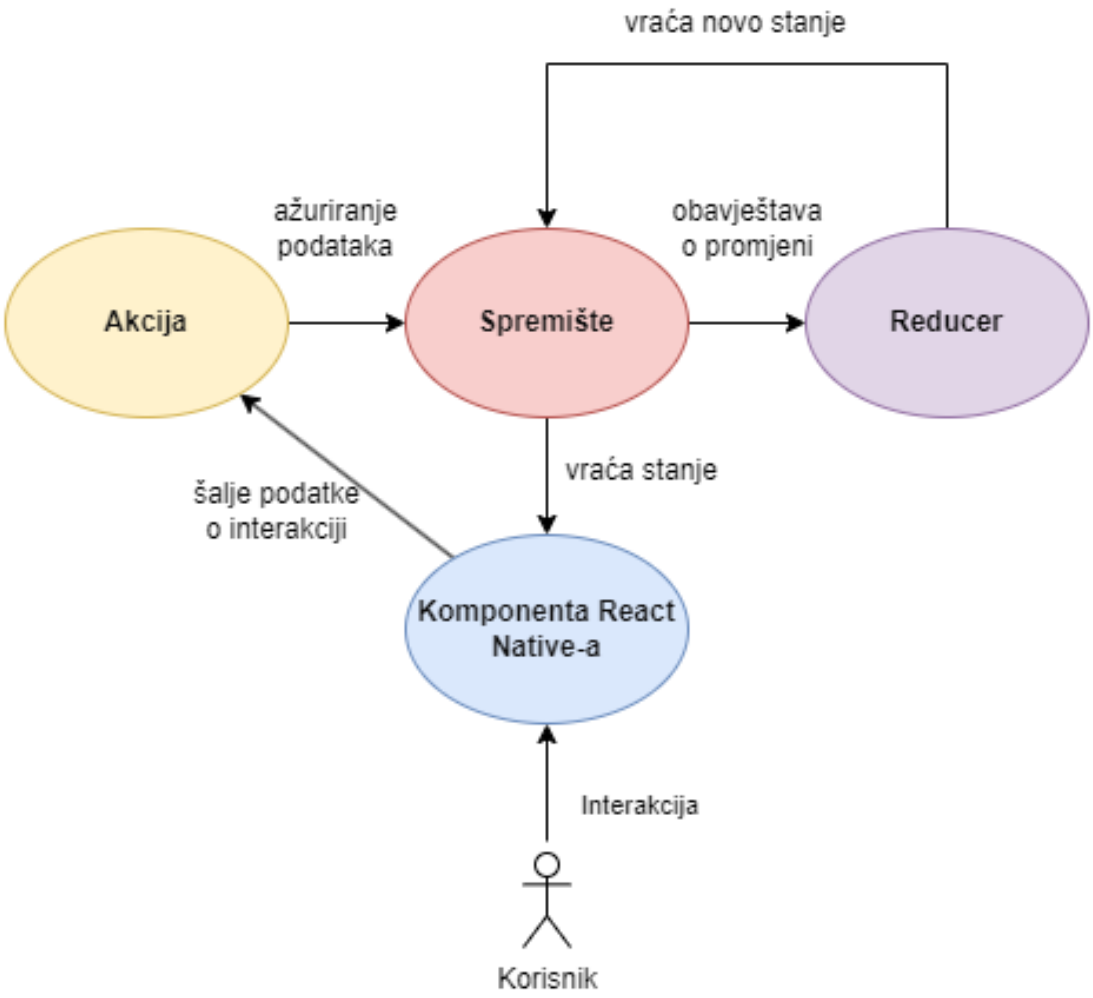
6.4. Arhitektura Redux

Prema Kim-u (2016.) Redux je uzorak dizajna te u skladu s korištenjem tog uzorka postoji i istoimena biblioteka koja podržava Redux arhitekturu. Tradicionalno razvijane aplikacije u React Native-u građene su podjelom na komponente te u takvim složenijim aplikacijama postoji mnogo stanja što uzrokuje teško praćenje stanja, nepredvidljivost ponašanja aplikacije, komplikacije u upravljanju stanjem te tzv. „prop drillin“ što označava pojavu kada se stanje aplikacije prosljeđuje kroz veliki broj slojeva komponenti koje uopće ne koriste to stanje. Arhitektura Redux rješava te probleme nekim svojim obilježjima, kao što su centralizirano stanje aplikacije i jednosmjernim tokom podataka kao u arhitekturi Flux. No, razlika između arhitektura Flux i Redux je to što Redux zastupa centralizaciju skladišta stanja, dok arhitektura Flux ima tu mogućnost, no nema striktno pravilo oko centralizacije.

Redux arhitektura se temelji na sljedećim objektima:

- Akcije ili kreatori
- Spremište (eng. Store)
- Reduceri (eng. Reducers)
- React komponente.

Najvažniji koncept arhitekture Redux je centralizirano spremište koje sadrži sva stanja aplikacije te se bilo koja komponenta može pretplatiti na praćenje tih stanja. Način na koji se stanje u spremištu mijenja je putem akcije odnosno objekta iz JavaScript-a koji sadrži tip događaja i ostale opcionalne podatke. Akcija se šalje iz komponente u kojoj je došlo do potrebe za promjenom stanja, a nakon toga reducer obrađuje akciju, odnosno prima trenutno stanje i akciju te vraća novo stanje na temelju tipa akcije. Na slici 4. prikazana je pojašnjena struktura arhitekture Redux.



Slika 4: Dijagram Redux arhitekture (Izvor: Vlastita izrada)

Prema Ehile (2023.) prednosti u korištenju arhitekture Redux u aplikacijama razvijanim okruženjem React Native su sljedeće:

- Centralizirano upravljanje stanjem što znači da sve komponente mogu pristupiti istom stanju te se time olakšava održavanje aplikacije sinkronizirane sa svim praćenim podacima i prikaz ažurnih podataka korisniku
- Predvidljiv protok podataka koji olakšava razumijevanje kako podaci teku. Na taj način olakšava se otklanjanje pogrešaka kod složenijih sustava i veću razumljivost i jasnoću koda

- Modularniji i lakše održiv kod
- Striktna pravila kod implementacije Redux stanja što pomaže u održavanju konzistentnosti i povećava predvidljivost aplikacije kao primjerice nepromjenjivost stanja
- Velika podrška zajednice i mnoštvo dostupnih resursa za učenje, a i implementaciju arhitekture Redux. Prvenstveno se radi o bibliotekama i paketima s gotovim funkcijama za kreiranje slojeva i pristup praćenim podacima.

Iako je vrlo prihvaćen od strane programera, posebno na složenijim aplikacijama, Redux ima i nekoliko nedostataka:

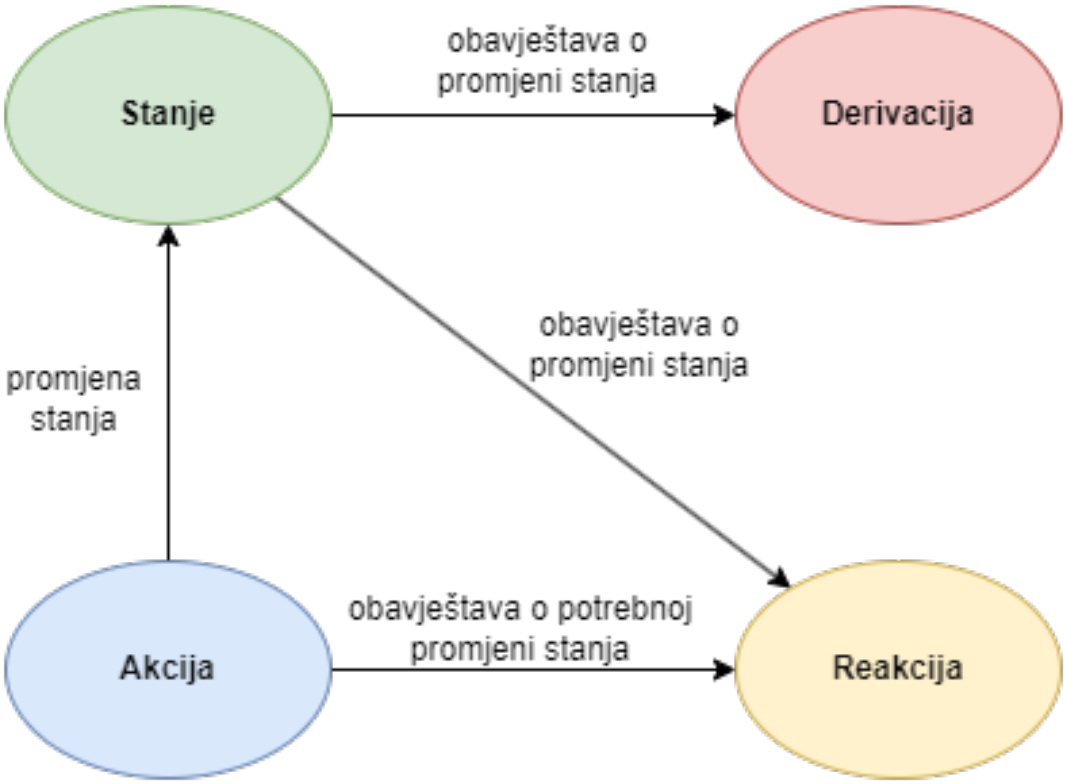
- Kompleksnost pri razumijevanju i učenju korištenja arhitekture Redux kod primjene na razvojno okruženje React Native
- Pojava tzv. *Boilerplate* koda koji Redux zahtijeva prilikom postavljanja skladišta, pisanja akcija i reducera. Taj dio razvoja može biti vremenski zahtjevan i repetitivan
- Ukoliko aplikacija radi s velikom količinom podataka, Redux može dodati određeno opterećenje performansama aplikacije
- Nepotrebna kompleksnost kod manjih i jednostavnijih aplikacija. Iako je moguće implementirati Redux na jednostavnim sustavima, postoje mnogo bolja rješenja.

Redux ponajviše iskače iz ostalih vrsta arhitektura zbog centralizacije stanja i zastupanja jednosmjernog toka podataka, a također i zbog odlične podrške u obliku biblioteka i paketa. Iako se smatra odličnom arhitekturom u složenijim sustavima, može biti nepotrebno kompleksan u vrlo jednostavnim tokovima podataka.

6.5. MobX arhitektura

Prema MobX dokumentaciji (bez dat.) MobX je u principu biblioteka za upravljanje stanjem koja koristi princip reaktivnog programiranja i smatra se uzorkom arhitekture zbog svojih koncepata koji se primjenjuju prilikom razvoja aplikacija. MobX je jednostavno, skalabilno i provjereno rješenje koje čini upravljanje stanjem jednostavnim rješavajući problem proizvodnje nekonzistentnog stanja. Nekonzistentno stanje može dovesti do neodrživih aplikacija punih grešaka, a stanje aplikacije je svojevrsna jezgra svake aplikacije.

Struktura arhitekture MobX može se vidjeti na dijagramu na slici 5.



Slika 5: Dijagram MobX arhitekture (Izvor: Vlastita izrada)

MobX se temelji na jednostavnoj strukturi gdje postoji stanje, akcija, reakcija i derivacija. Pod aplikacijskim stanjem smatraju se svi grafovi objekata, nizovi, liste, reference i sl. koje formiraju model aplikacije te se ti podaci smatraju podatkovnim jedinicama aplikacije. Derivacije označavaju funkcije koje izračunavaju bilo koju vrijednost koja može biti izračunata automatski iz stanja aplikacije. Te derivacije, odnosno izračunate vrijednosti, mogu varirati od jednostavnih vrijednosti kao što su brojevi pa sve do složenih vrijednosti kao što su prikazi u HTML-u. Reakcije su slične derivacijama, no te funkcije ne proizvode vrijednost već se pokreću kako bi izvršile neki zadatak, obično u kontekstu ulaznih i izlaznih vrijednosti. Također, reakcije osiguravaju ažuriranje DOM-a te da se mrežni zahtjevi automatski izvršavaju u pravo vrijeme. Akcije su objekti koji mijenjaju stanje. MobX osigurava da sve promjene stanja aplikacije uzrokovane akcijama budu automatski obrađene od strane svih derivacija i akcija.

Prema Gizzi i Mayes (2023.), prednosti MobX arhitekture su sljedeće:

- Dosljedno i reaktivno ažuriranje stanja aplikacija što rezultira nemogućnošću dovođenja aplikacije u nekonzistentno stanje
- Jednostavno upravljanje stanjem zahvaljujući konceptu promatranih podataka koji automatski obavještavaju komponente React Native-a o nastalim promjena
- Fleksibilnost u stvaranju skladišta zbog mogućnosti stvaranja više različitih skladišta. Time se omogućuje modularna organizacija stanja što također na neki način pomaže u organizaciji složenih aplikacija.

MobX arhitektura u implementaciji je vrlo jednostavna i laka za razumijevanje, no postoji

nekoliko ozbiljnijih nedostataka:

- Moguća arhitekturalna kompleksnost u velikim aplikacijama
- Moguća nepredvidivost zbog manjka striktnih pravila kao što je to slučaj u arhitekturi Redux. Iako je fleksibilnost u jednostavnijim sustavima prednost, u većim sustavima može doći do težeg razumijevanja i održavanja koda.

7. Implementacija odabranih arhitektura na aplikaciji

U ovom poglavlju prikazat će se primjena nekoliko arhitektura na stvarnoj aplikaciji. Aplikacija je razvijana korištenjem okruženja React Native. React Native je vrlo popularan razvojni okvir u kojem je danas izrađena većina mobilnih aplikacija na tržištu, a kreiran je od strane Facebook-a. S obzirom na to da je React Native često korišteno okruženje, postoji mnogo različitih paketa s već gotovim funkcijama koje uvelike pomažu kod razvoja aplikacija s ciljem poboljšanja performansi kreiranih aplikacija te olakšavanja primjene određenih koncepata kao što su to SOLID principi, a također i praćenja određenih arhitektura sustava. Između ostalog, React Native okruženje ima pakete za primjenu Flux, Redux i MobX arhitektura koje je potrebno instalirati u React Native projekt prije početka razvoja aplikacije. U nastavku će se spomenuti koji su to paketi i kako pomažu u primjeni spomenutih arhitektura.

U sklopu ovog projekta izrađena je aplikacija za planiranje putovanja. Aplikacija je osmišljena kao svojevrsni planer gdje korisnik na svom mobilnom uređaju može lokalno spremati planirane izlete. Prilikom otvaranja aplikacije korisnik može vidjeti listu putovanja te dodati novo putovanje ili izbrisati postojeće. Svako putovanje ima početno i završno vrijeme te se za svaki dan između tih datuma dodaje po jedan zapis dnevnog plana. Dnevni plan može imati aktivnosti i prijevoze. Za svaku aktivnost dodaje se lokacija koja se može vidjeti na karti.

Na opisanoj aplikaciji primijenjene su tri od pet spomenutih vrsta arhitektura, a to su MVC arhitektura, Redux i MobX. S obzirom na to da aplikacija nema funkcije visoke razine kompleksnosti, odnosno kompleksan model podataka, nema potrebe za arhitekturom poput Flux-a koja je osmišljena za velike i složene aplikacije s mnogo entiteta i složenih relacija i ograničenja. S druge strane, MVVM arhitektura je suviše slična MVC arhitekturi te u većini slučajeva nema prevelike razlike u performansama, osim što se dijelovi arhitekture i njihove zadaće razlikuju. Također, MVVM arhitektura je osmišljena za složenije sustave i sama po sebi je teža za razumijevanje prilikom otklanjanja pogrešaka.

Razlog primjene konkretnih arhitektura je to što su te arhitekture prikladne za prikaz na osmišljenoj aplikaciji. Iako su MVC i Redux arhitekture osmišljene kako bi pojednostavile programski kod, olakšale otklanjanje pogrešaka i zapravo cjelokupni razvoj aplikacije, one se mogu primijeniti i na manjim aplikacijama i poželjno ih je koristiti u aplikacijama neovisno o složenosti. S druge strane, MobX arhitektura je najprikladnija za primjenu na aplikacijama manje razine složenosti iz razloga što ne dodaje nepotrebnu dodatnu kompleksnost već zaista pojednostavljuje kod i razdvaja ga na jasne slojeve.

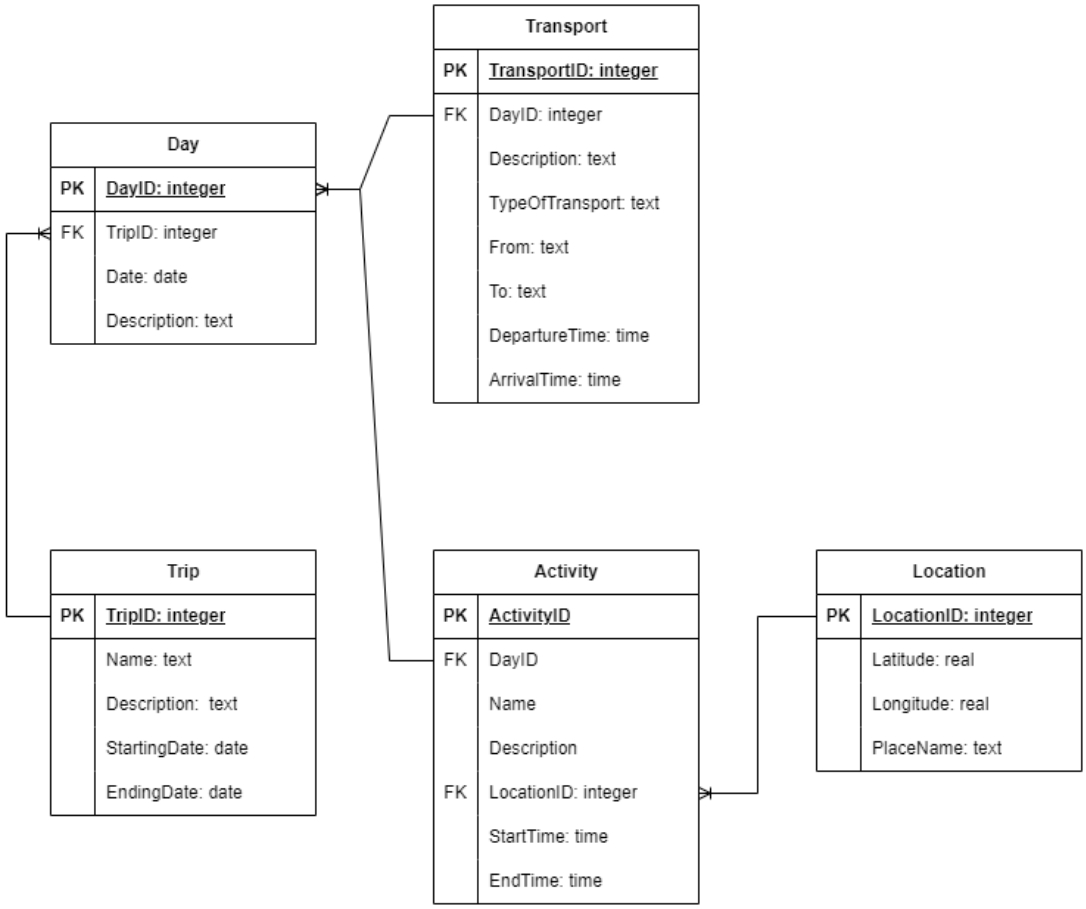
7.1. Podatkovni model aplikacije i baza podataka

Za bolje razumijevanje strukture i svrhe aplikacije, pojasnit će se podatkovni model aplikacije. Podatkovni model sastoji se od sljedećih entiteta:

- Trip
- Day

- Activity
- Transport
- Location.

Entitet *Trip* služi za spremanje zapisa putovanja. Entitet *Day* služi kao spojnica između zapisa putovanja i korespondirajućih aktivnosti i transporta. Iako se aktivnosti i transporti mogu direktno vezati za putovanje, ovakav je koncept osmišljen kako bi korisnik mogao unijeti opis za konkretni dan i lakše organizirati planove prema danima. Entitet *Location* služi za spremanje lokacija koje se koriste u aktivnostima. Lokacije se spremaju na ovaj način kako ne bi došlo do redundancije spremljenih podataka u kontekstu većeg broja aktivnosti na istoj lokaciji. U tom slučaju, svaka aktivnost bila bi vezana za jedan jedini zapis u bazi podataka. Entitet *Activity* predstavlja zapis aktivnosti koji se sastoji od vanjskog ključa na pripadajući dan, imena, opisa, početnog i završnog vremena te vanjskog ključa na lokaciju na koju je vezana aktivnost. Entitet *Transport* također je dio dana pa sadrži vanjski ključ na dan, opis, tip transporta, polazište, odredište, vrijeme polaska i vrijeme dolaska. Na slici 6. prikazan je model podataka na dijagramu entiteta i veza (eng. Entity Relationship Diagram).



Slika 6: Dijagram entiteta i veza (Izvor: Vlastita izrada)

Baza podataka izrađena je pomoću SQLite sustava za upravljanjem bazom podataka. Razlog odabira SQLite-a je jednostavnost spremanja podataka. Za ovakvu vrstu aplikacije

gdje nema smisla implementirati server ili spremati podatke u pohrani na oblaku, najpogodnije je koristiti lokalnu bazu podataka i spremati podatke na uređaju na kojem se aplikacija koristi. SQLite sprema podatke na način da ih zapisuje u jednu datoteku te su sve tablice i zapisi u tablicama centralizirani. Iako je to pogodno za jednostavne aplikacije sa jednostavnim modelima podataka, za potrebe spremanja veće količine podataka i za kompleksnije modele postoje bolje opcije.

Prilikom prvog pokretanja aplikacije, nakon instalacije, poziva se sljedeća naredba:

```
const db = SQLite.openDatabase('TravelPlannerDatabase.db');
```

Programski kod 6: Naredba za otvaranje baze podataka (vlastita izrada)

Naredba *openDatabase* inicijalno služi za otvaranje baze podataka i spremanje reference na otvorenu bazu u varijablu *db*. Također, prije otvaranja baze podataka provjerava postoji li lokalno spremljena datoteka danog imena, a ako ne postoji, kreira novu datoteku. Nakon kreiranja i otvaranja baze, pomoću reference *db* šalje se transakcija sa SQL klauzulama za primjerice kreiranje tablica i ograničenja te operacije nad podacima kao što su unos, čitanje, modifikacija i brisanje.

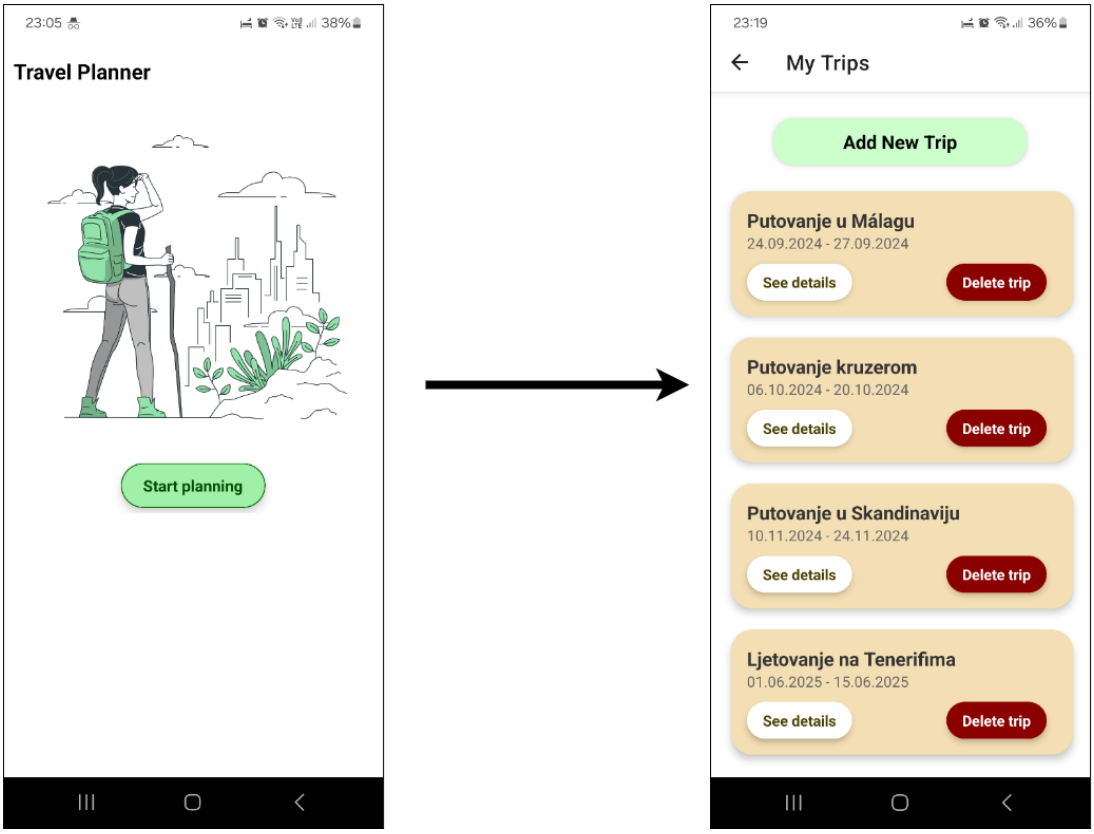
```
db.transaction(tx => {
    tx.executeSql(
      `CREATE TABLE IF NOT EXISTS Trip (
    TripId INTEGER PRIMARY KEY AUTOINCREMENT,
    Name TEXT,
    Description TEXT,
    StartDate DATE,
    EndDate DATE
    );`
    );
```

Programski kod 7: Primjer kreiranja tablice u SQLite bazi podataka (vlastita izrada)

U prethodnom isječku koda prikazan je primjer kreiranja tablice. Dakle, referenci na bazu podataka *db* šalje se transakcija sa naredbom u SQL-u. Na taj način se komunicira s bazom podataka, u ovom slučaju za kreiranje tablice sa konkretnim atributima.

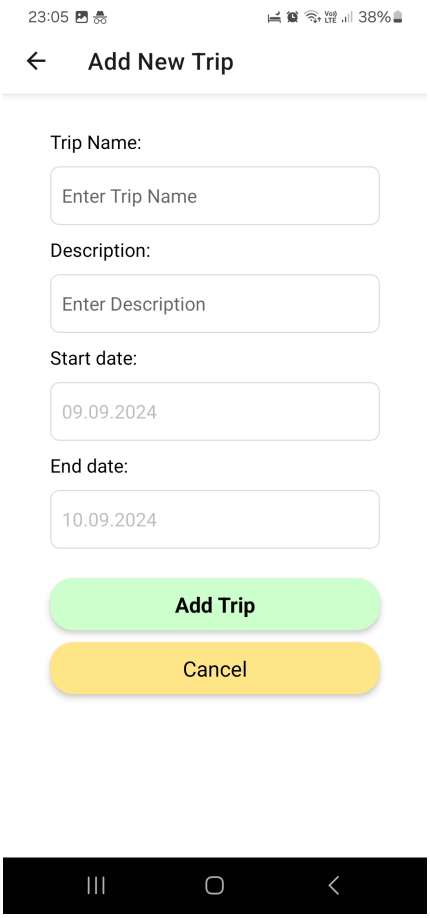
7.2. Svrha i funkcionalnosti aplikacije

Kako se ranije spomenulo, aplikacija je osmišljena kao alat za pomoć korisniku u planiranju putovanja. Za korištenje aplikacije nije potrebna nikakva prijava korisnika s obzirom na to da je aplikacija lokalnog dosega i podaci se spremaju lokalno. Ulaskom u aplikaciju korisniku se prikaže zaslon sa listom putovanja i gumbom za otvaranje forme za kreiranje putovanja. Na sljedećoj slici može se vidjeti početni zaslon i lista putovanja.



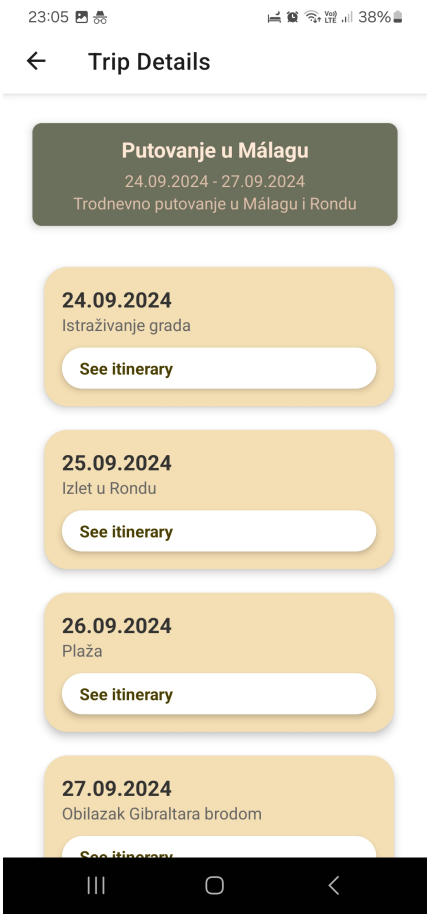
Slika 7: Početni zaslon i lista putovanja (Izvor: Vlastita izrada)

Klikom na gumb *Start Planning* otvara se zaslon za prikaz liste putovanja. Lista putovanja kreirana je kao komponenta *Flatlist* koja prima pomoćne komponente odnosno kartice *TripCard* u kojima su prikazani podaci jednog putovanja. Klikom na *Add New Trip* na desnom zaslonu na slici 7. korisniku se otvara forma za unos novog putovanja koja je prikazana na sljedećoj slici.



Slika 8: Prikaz forme za dodavanje novog putovanja (Izvor: Vlastita izrada)

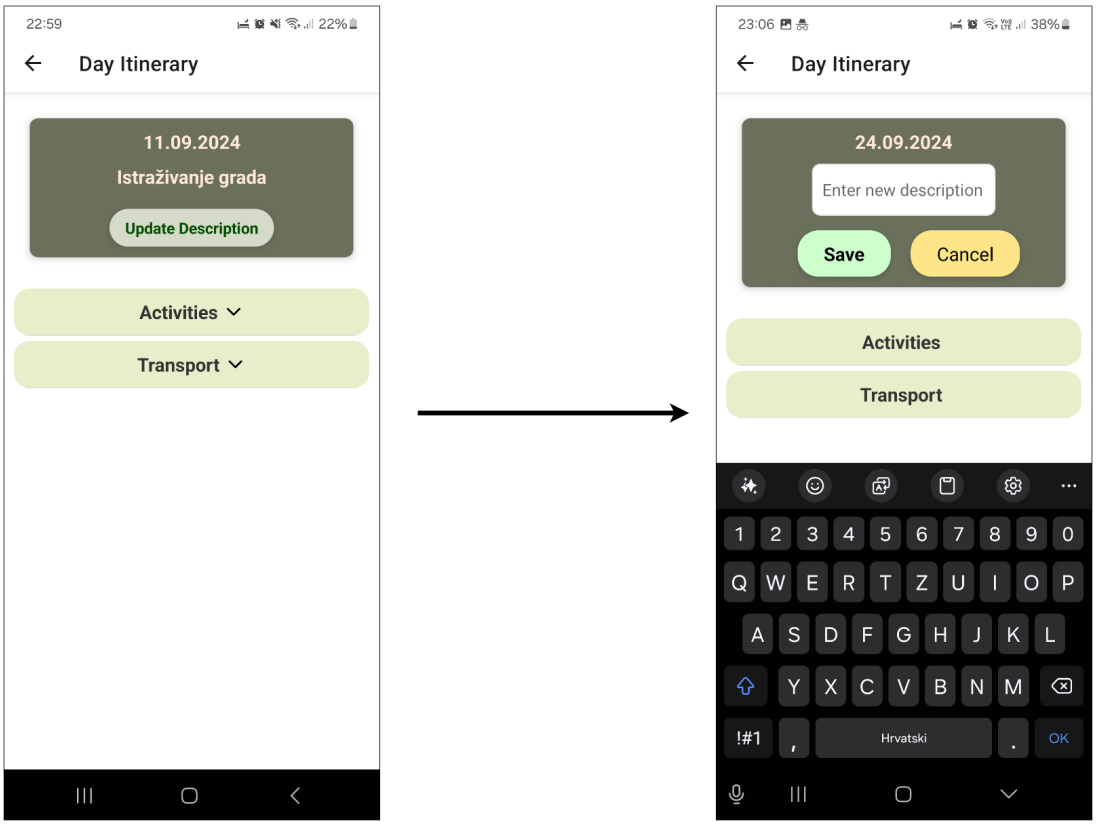
Na slici 8. prikazana je forma za dodavanje putovanja, a putovanje ima naziv, opis, početni datum i završni datum. Ograničenje na ovoj formi je takvo da završni datum uvijek mora biti jednak ili kasniji od početnog datuma. Klikom na gumb Save podaci se spremaju i novi zapis putovanja je unesen u bazu. Nakon dodavanja odnosno nakon klika na gumb i spremanja putovanja ažuriraju se podaci na listi putovanja te je odmah vidljivo novododano putovanje.



Slika 9: Zaslon za prikaz detalja putovanja i liste dana (Izvor: Vlastita izrada)

Važno je spomenuti da se prilikom kreiranja novog zapisa putovanja automatski u bazi kreira po jedan zapis za svaki dan putovanja gdje je raspon izračunat pomoću početnog i završnog datuma. Klikom na gumb *See Details* na kartici jednog putovanja na listi putovanja prikazanoj na prethodnim slikama otvaraju se detalji putovanja, odnosno otvara se lista dana putovanja. Zaslon s detaljima prikazan je na slici 9.

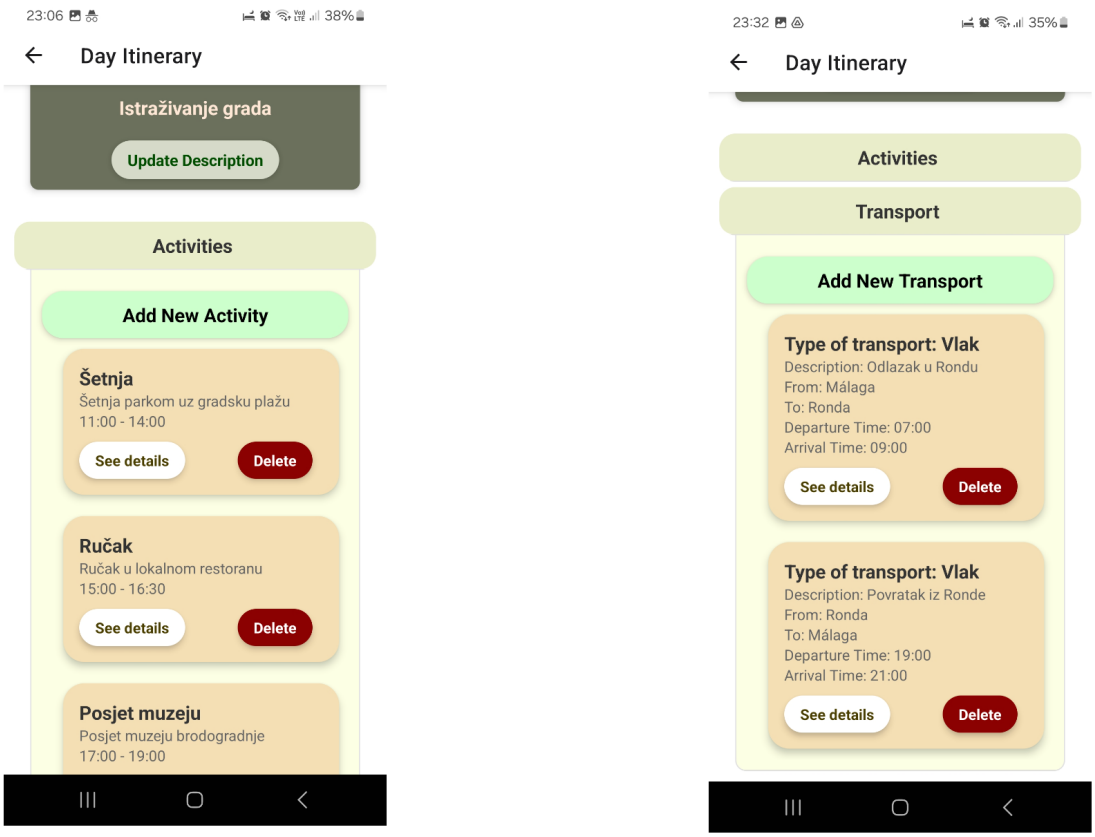
Svaki dan može imati i svoj opis koji se modificira na detaljima dana te se nakon toga prikazuje na ovom zaslonu u kartici pojedinog dana. Na sljedećoj slici prikazan je zaslon liste dana za odabrano putovanje. Klikom na gumb *See itinerary* otvaraju se detalji jednog odabranog dana. Inicijalno, dan nema nikakav opis već je to prepušteno korisniku. Prilikom kreiranja dana, opisi dana su prazni.



Slika 10: Zaslon za prikaz plana dana i modifikaciju opisa dana (Izvor: Vlastita izrada)

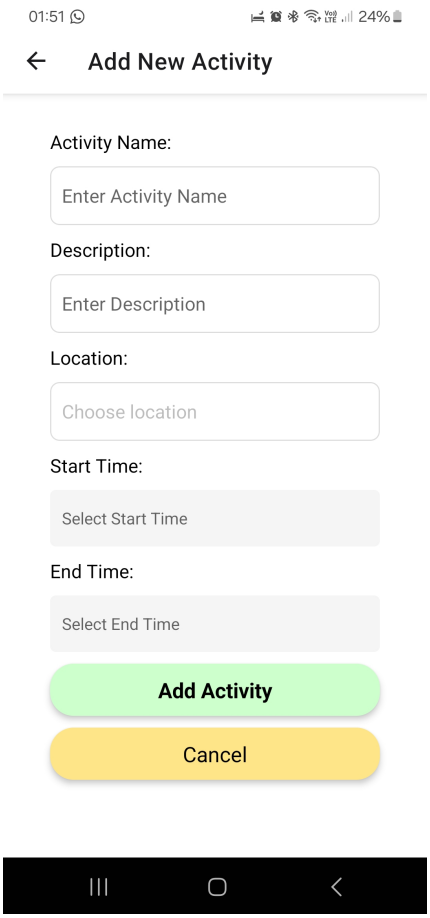
Na slici 10. prikazane su varijante zaslona za prikaz detalja dana. Na lijevom zaslonu postoji gumb *Update Description* te se klikom na taj gumb otvara polje za unos novog opisa što je vidljivo na desnom zaslonu. Ukoliko opis već postoji jer ga je korisnik prethodno unio, no želi ga promijeniti, prilikom otvaranja polja u polju se nalazi trenutni opis te ukoliko je opis jednak staroj vrijednosti, ne događa se ništa. Klikom na gumb *Save* korisnik sprema opis, a klikom na gumb *Cancel* korisnik zatvara varijantu zaslona za modifikaciju opisa dana.

Na stranici detalja dana također se nalaze dvije komponente, *Activities* i *Transports*. Nakon što se klikne na jednu od komponenata, one se proširuju te se prikazuje lista aktivnosti ili lista prijevoza, sukladno komponenti na koju je korisnik kliknuo.



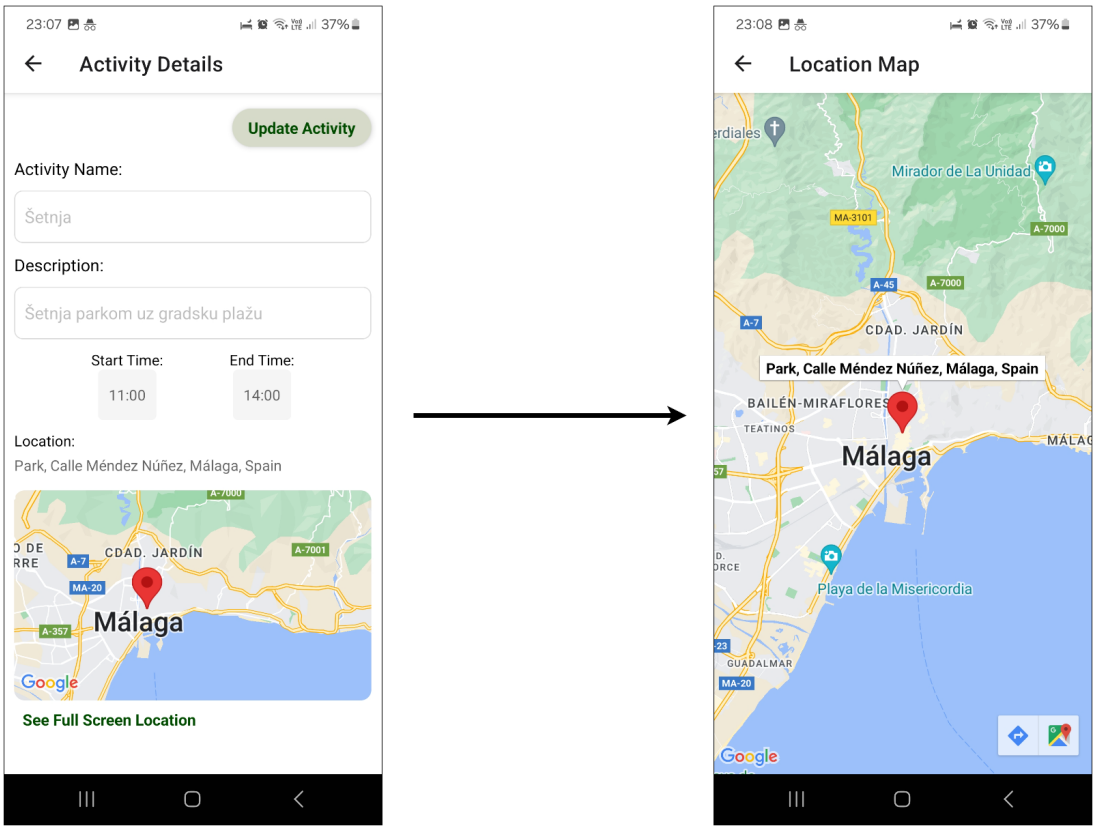
Slika 11: Zaslon detalja dana s otvorenim listama aktivnosti i prijevoza (Izvor: Vlastita izrada)

Na slici 11. prikazane su otvorene liste aktivnosti i transporta. Komponente koje sadrže liste aktivnosti i prijevoza su komponente tipa *Collapsible* iz skupine komponenata React Native i uglavnom se koriste za ugradnju lista u zaslone, no mogu sadržavati i druge komponente. Aktivnosti su prikazane na karticama koje sadrže naziv aktivnosti, opis (ukoliko postoji), vrijeme početka i vrijeme kraja. Kartice za prijevoz sadrže podatke o konkretnom prijevozu, kao što je opis, polazište, odredište, vrijeme polaska i dolaska na destinaciju. Također, klikom na gumb *Delete* na svakoj kartici aktivnosti i prijevoza zapisi se mogu obrisati, no prije toga iskače zaslon za potvrdu brisanja. Klikom na gumb *See Details* mogu se vidjeti svi podaci o aktivnosti ili o prijevozu. Iznad kartica aktivnosti nalazi se gumb *Add New Activity* te se klikom na taj gumb otvara forma za unos aktivnosti prikazana na slici 12.



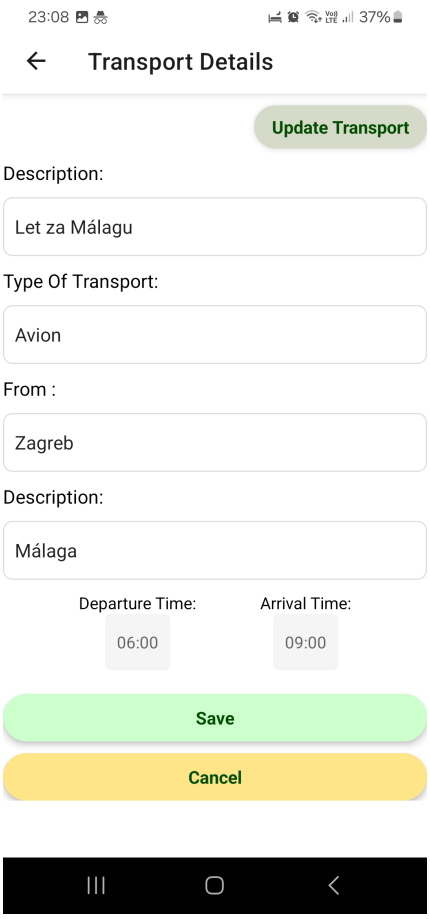
Slika 12: Zaslon za prikaz forme za dodavanje nove aktivnosti (Izvor: Vlastita izrada)

Prilikom unosa aktivnosti, može se odabrati i lokacija. Odabir lokacije integriran je sa sustavom Google Maps preko Google API-ja tako da se odabire realno mjesto te se koordinate tih lokacija spremaju u bazu. Važno je spomenuti da prilikom kreiranja lokacije sustav provjerava postoji li takav zapis u bazi podataka te aktivnost spaja sa tom lokacijom, a ako ne postoji ta lokacija u bazi, tada se automatski kreira nova i spaja na novu aktivnost prilikom kreiranja iste. Implementacija je vezana za REDUX arhitekturu te će biti pobliže objašnjena u kasnijem poglavlju.



Slika 13: Zaslon za prikaz detalja aktivnosti i zaslon lokacije na punom zaslonu (Izvor: Vlastita izrada)

Prilikom klika na gumb *See Details* na kartici aktivnosti otvara se zaslon s detaljima o aktivnosti prikazan na slici 13. Osim već prikazanih podataka o aktivnosti ovdje se prikazuje i lokacija vezana za aktivnost. Lokacija je prikazana na komponenti ugrađenoj u stranicu o detaljima, a klikom na *See Full Screen Location* otvara se posebna stranica na kojoj je lokacija u punom zaslonu prikazana na desnom zaslonu na slici 13. Na zaslonu detalja aktivnosti u desnom gornjem kutu nalazi se gumb *Update Activity* na čiji klik polja s detaljima postaju polja za unos, odnosno njihovo svojstvo *editable* podešava se na istinitu vrijednost.



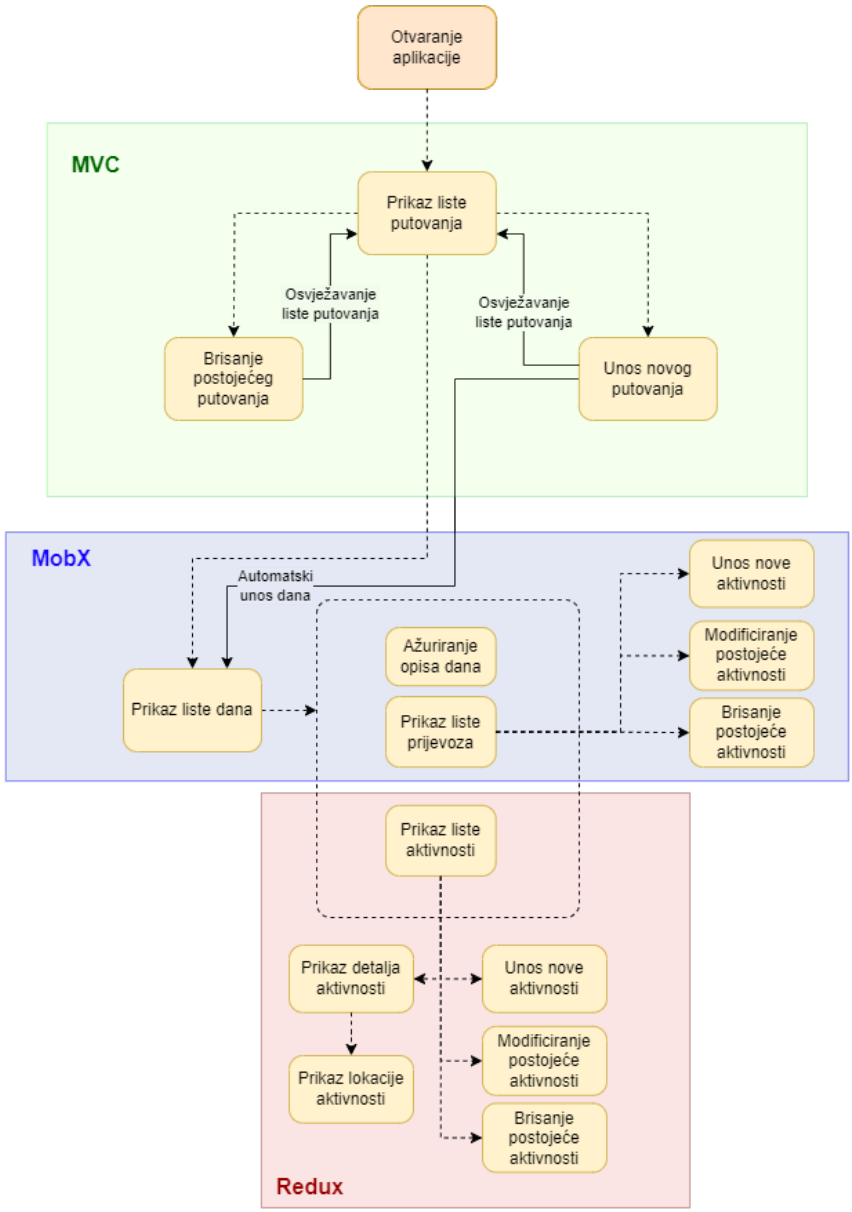
Slika 14: Zaslon za prikaz detalja o danu i modifikaciju opisa (Izvor: Vlastita izrada)

Na *Collapsible* komponenti koja sadrži listu prijevoza nalaze se kartice s detaljima o prijevozu i gumbima *See Details* i *Delete*. Klikom na *Delete* otvara se zaslon za potvrdu brisanja te se u slučaju potvrde zapis briše, a lista prijevoza se automatski ažurira. Klikom na gumb *See Details* otvara se zaslon prikazan na slici 14. gdje su navedeni svi detalji zapisa prijevoza. U desnom gornjem kutu nalazi se gumb *Update Transport* na čiji klik se polja s podacima pretvaraju u polja za unos novih vrijednosti. Klikom na gumb *Save* zapis se sprema u bazu podataka, lista prijevoza se ponovno renderira, a povratkom na prethodnu stranicu prikazuju se ažurirani podaci.

7.3. Podjela aplikacije po implementiranim arhitekturama

U sklopu izrade aplikacije izabrane su tri vrste spomenutih arhitektura, a to su MVC odnosno Model - Pogled - Kontroler te arhitekture Redux i MobX. S obzirom na to da su arhitekture MVC i MVVM prilično slične, nije potrebno prikazati obje već je odabrana arhitektura MVC. Vrsta arhitekture MVC je prilično klasična te za nju nema nikakvih potrebnih dodatnih paketa odnosno biblioteka s funkcijama za implementaciju. Razlog tome je što je arhitektura MVC, kao i MVVM fokusirana na raspodjelu zadaća po slojevima, a svaki sloj u implementaciji arhitekture u aplikaciji razvijanoj u React Native-u se nalazi u zasebnoj funkciji, klasi ili konstanti. Vrste arhitektura Flux, Redux i MobX zahtijevaju korištenje paketa s funkcijama za lakšu

implementaciju istih, a u sljedećim poglavljima biti će pojašnjeno koji su to paketi i zašto se koriste. U ostatku aplikacije koristi se Redux i MobX, dok Flux nije implementiran iz razloga što je Flux osmišljen za aplikacije visoke kompleksnosti te nije prigodan za korištenje na samo jednom dijelu izrađene aplikacije, a s druge strane suviše je sličan arhitekturi Redux, osim što ne zastupa strogu centralizaciju skladišta stanja. MVC i Redux arhitekture su također prikladne za kompleksne sustave, no njihovu svrhu i način rada je moguće prikazati i na aplikaciji niže kompleksnosti te se danas preporuča korištenje istih na bilo kojim razinama kompleksnosti. Redux i MobX arhitekture su za razliku od MVC arhitekture više usmjerene na upravljanje stanjem aplikacije te nemaju striktna pravila za komunikaciju s bazom podataka kao što to ima MVC arhitektura koja posjeduje sloj Model čija je zadaća komunikacija s bazom. Na slici 15. prikazan je dijagram funkcionalnosti te koje su arhitekture primijenjene na koju funkcionalnost aplikacije.



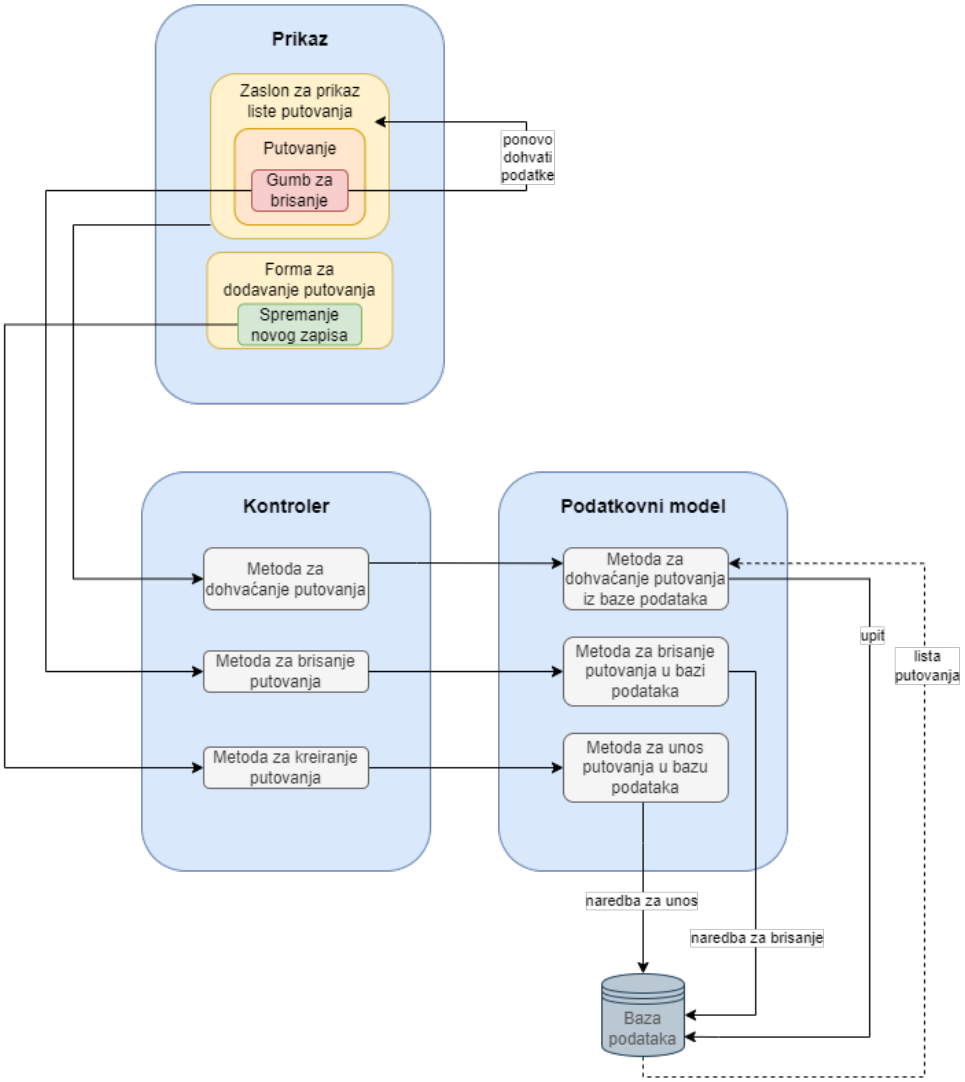
Slika 15: Dijagram raspodjele arhitektura po funkcionalnostima aplikacije (Izvor: Vlastita izrada)

MVC arhitektura je implementirana na dijelu aplikacije koji se dotiče liste putovanja, dodavanja novog zapisa putovanja i brisanja postojećeg putovanja. Prikazano je kako u MVC arhitekturi funkcionira ažuriranje podataka odnosno osvježavanje liste što je jedna od karakteristika arhitektura koja se uvelike razlikuje u implementacijama različitih arhitektura. Strelica sa labelom *Osvježavanje liste putovanja* označava ručno izrađenu funkciju koja ponovno dohvaća listu putovanja te na taj način osvježava stranicu s prikazom liste putovanja. Takav način ažuriranja podataka nije potreban u arhitekturama Redux i MobX što je prilično velika mana arhitekture MVC te jedan od glavnih razloga zašto se u praksi arhitektura MVC kombinira s drugim arhitekturama koje imaju bolje rješenje problema ažuriranja podataka na prikazu. MobX arhitektura je korištena za dodavanje zapisa dana u bazu podataka nakon što se u dijelu aplikacije s primijenjenom MVC arhitekturom dodao novi zapis putovanja. Također, MobX je implementiran na dio aplikacije za prikaz liste prijevoza, dodavanje novih prijevoza, modifikaciju zapisa postojećih prijevoza i brisanje prijevoza. Redux je implementiran na dijelu aplikacije za prikaz aktivnosti i upravljanje stanjem lokacije i prikaza lokacije na komponenti geografske karte.

8. Implementacija Model - Pogled - Kontroler (MVC) arhitekture

Kao što je spomenuto u prethodnom podpoglavlju, MVC arhitektura je implementirana na funkcionalnosti prikaza, brisanja i dodavanja putovanja. MVC arhitektura je prilično jednostavna i svaki od slojeva arhitekture ima jasnu podjelu odgovornosti. MVC arhitektura se dakle sastoji od tri sloja, a to su model, pogled i kontroler.

MVC arhitektura u implementaciji ne zahtijeva nikakve dodatne biblioteke ili pakete te se više odnosi na raspored metoda za upravljanje tokom podataka i generalnom raspodjelom slojeva aplikacije. Za razliku od arhitektura koje uključuju upravljanje stanjem aplikacije, ovaj model sprema stanja na razini metode odnosno u lokalnim stanjima te različiti dijelovi dijele podatke pomoću prosljeđivanja u parametrima metoda. U nastavku je prikazan dijagram implementacije arhitekture MVC na primjeru aplikacije gdje se može vidjeti tok podataka i poziva metoda iz različitih slojeva.



Slika 16: Dijagram implementirane arhitekture MVC (Izvor: Vlastita izrada)

Na slici 16. prikazan je dijagram implementirane arhitekture Model - Pogled - Kontroler. Prikazan je svaki od slojeva i sadržaj svakog sloja. Može se primijetiti kako sloj za prikaz sadrži komponente React Nativa, odnosno zaslon za prikaz liste putovanja, formu za dodavanje i ostalo. Prilikom bilo kakve interakcije korisnika u korisničkom sučelju na prikazu, komponenta iz prikaza zove metodu u kontroleru koja poziva metodu u podatkovnom modelu. Metode u podatkovnom modelu sadrže naredbe u SQL-u za komunikaciju s bazom podataka. U slučaju potrebe za ažuriranjem podataka na prikazu, komponenta sadrži metodu za ponovno dohvaćanje podataka te se na taj način korisniku prikazuju ažurirani podaci. U nastavku će svaki od slojeva biti detaljnije objašnjen te će biti prikazani isječci programskog koda u svrhu demonstracije implementacije na primjeru.

8.1. Implementacija modela

Model služi za upravljanje podacima i poslovnom logikom aplikacije. U ovom slučaju, model je komponenta koja sadrži metode za kreiranje novog putovanja, dohvaćanje liste putovanja i brisanje putovanja.

```
import db from '../../database/database';
import {
  insertTripQuery,
  getAllTripsQuery,
  getTripByIdQuery,
  updateTripQuery,
  deleteTripQuery
} from '../../database/queries';

const TripModel = {
    createTrip: async (name, description, startDate, endDate) => {
        return new Promise((resolve, reject) => {
        db.transaction(tx => {
          tx.executeSql(
            insertTripQuery(name, description, startDate, endDate),
            [],
            (_, result) => {
              const tripId = result.insertId;
              resolve(result);},
            (_, error) => reject(error)
          );
        });
  });
    },

    getAllTrips: async () => {
        return new Promise((resolve, reject) => {
    db.transaction(tx => {
      tx.executeSql(
        getAllTripsQuery(),
        [],
```

```
        (_, { rows: { _array } }) => resolve(_array),
        (_, error) => reject(error)
      );
    });
  });
    },

    deleteTrip: async (tripId) => {
        return new Promise((resolve, reject) => {
    db.transaction(tx => {
      tx.executeSql(
        deleteTripQuery(tripId),
        [],
        (_, result) => resolve(result),
        (_, error) => reject(error)
      );
    });
  });
    }
};

export default TripModel;
```

Programski kod 8: Implementacija modela u arhitekturi MVC (vlastita izrada)

Na prikazanom isječku koda može se vidjeti implementacija modela. Model sadrži metode za direktnu komunikaciju s bazom podataka gdje pomoću reference na bazu podataka *db* šalje bazi podataka naredbe za operacije odnosno upite koji se nalaze u datoteci *queries*. Upiti se mogu pisati i direktno u ove metode, no u ovom slučaju su sadržani u zasebnim varijablama kako bi se lakše otklonile pogreške te povećala jasnoća koda.

8.2. Implementacija pogleda

Pogled je sloj koji služi za prikaz podataka i komunikaciju s korisnikom te time predstavlja sučelje za interakciju. To uključuje prikaz odnosno prezentaciju ažurnih podataka, primjerice prikaz liste putovanja te prikaz ažurirane liste putovanja nakon što je korisnik unio novo putovanje ili obrisao postojeće. Druga zadaća pogleda je primanje ulaznih podataka od korisnika kao što su podaci novog putovanja koje korisnik želi unijeti, a može biti i zahtjev za nekom akcijom, primjerice brisanje. Pogled prikazuje podatke koji su generirani od strane modela, dakle on prati promjene modela i ažurira prikaz u skladu s nastalim promjenama.

```
export default function TripList({navigation}) {
    const [trips, setTrips] = useState([]);
    const isFocused = useIsFocused();

    const fetchTrips = async () => {
      try {
            const allTrips = await TripController.getAllTrips();
            setTrips(allTrips);
```

```
      } catch (error) {
          console.error('Error fetching trips:', error);
        }
    };

    useEffect(() => {
        if (isFocused) {
            fetchTrips();
        }
    }, [isFocused]);

    const onRefresh = useCallback(() => {
      fetchTrips();
    }, []);

    return (
    <ScreenWrapper>
        <View style={styles.addButtonContainer}>
          <TouchableOpacity
          style={styles.addButton}
          onPress={() => navigation.navigate('AddTrip')}
          >
          <Text style={styles.addButtonText}>Add New Trip</Text>
        </TouchableOpacity>
        </View>
        <FlatList
          data={trips}
          keyExtractor={(item) => item.TripId}
          renderItem={({ item }) => <TripCard trip={item} navigation={navigation}
              onRefresh={onRefresh}/>}
        />
    </ScreenWrapper>
  )
}
```

Programski kod 9: Primjer komponente pogleda za prikaz liste putovanja (vlastita izrada)

Na prikazanom isječku koda nalazi se jedan od pogleda u implementiranoj MVC arhitekturi. Dakle, dijelovi sloja pogleda u ovom slučaju su stranica za prikaz svih putovanja, stranica za dodavanje novog putovanja i jedna zasebna komponenta *TripCard* koja služi kao kartica za prikaz jednog zapisa putovanja u listi putovanja. Način na koji pogled dolazi do podataka o listi putovanja je putem metode *fetchTrips* koja poziva metodu u kontroleru za dohvaćanje liste putovanja. Važno je istaknuti način na koji se osvježava stranica prilikom brisanja zapisa. Naime, MVC sam po sebi nema nikakav sustav praćenja stanja te stoga ažuriranje pogleda mora biti ručno. Poziva se tzv. *callback* metoda koja ponovno dohvaća listu putovanja te zatim prikazuje ispravnu listu. Iz tog razloga MVC arhitektura se u praksi često kombinira sa arhitekturom Redux koja zastupa praćenje stanja aplikacije i ponovno renderiranje komponenata te samim time unosi dodatnu vrijednost aplikacije, a to je ažurnost podataka na prikazu bez korištenja dodatnih metoda za osvježavanje stranica. U nastavku će se pojasniti i prikazati uloga kontrolera te kako se preko njega dolazi do podataka iz baze.

8.3. Implementacija kontrolera

Pogled ne komunicira direktno sa modelom, već koristi kontroler kao posrednika što znači da je jedina zadaća kontrolera da čeka zahtjev pogleda da se izvrši neka operacija, odnosno metoda u kontroleru. Kontroler sadrži metode koje pozivaju metode iz modela te prosljeđuje potrebne podatke modelu ili vraća potrebne podatke pogledu.

```
import TripModel from '../Models/TripModel';

const TripController = {
    createTrip: async (name, description, startDate, endDate) => {
        try {
             const newTripId = await TripModel.createTrip(name, description,
                 startDate, endDate);

             return newTripId;
        } catch (error) {
            console.error('Failed to create trip:', error);
        }
    },

    getAllTrips: async () => {
        try {
            const trips = await TripModel.getAllTrips();
            return trips;
        } catch (error) {
            console.error('Failed to get trips:', error);
            return [];
        }
    },

    deleteTrip: async (tripId) => {
        try {
            await TripModel.deleteTrip(tripId);
        } catch (error) {
            console.error('Failed to delete trip:', error);
        }
    }
};

export default TripController;
```

Programski kod 10: Implementacija kontrolera u arhitekturi MVC (vlastita izrada)

U prethodnom isječku koda prikazan je kontroler koji sadrži metode za komunikaciju s modelom. U suštini to je samo primanje korisničkih zahtjeva i obrada podataka. Primjerice, ukoliko korisnik klikne na gumb za brisanje nekog putovanja, pogled će pozvati metodu iz kontrolera pod nazivom *deleteTrip* te će mu prilikom poziva proslijediti id vrijednost putovanja koje želi obrisati. Tada kontroler poziva metodu u modelu za brisanje iz baze podataka te, po potrebi, nakon izvršenja obavještava o rezultatu. Ukoliko postoji kompleksnijih ograničenja u vezi

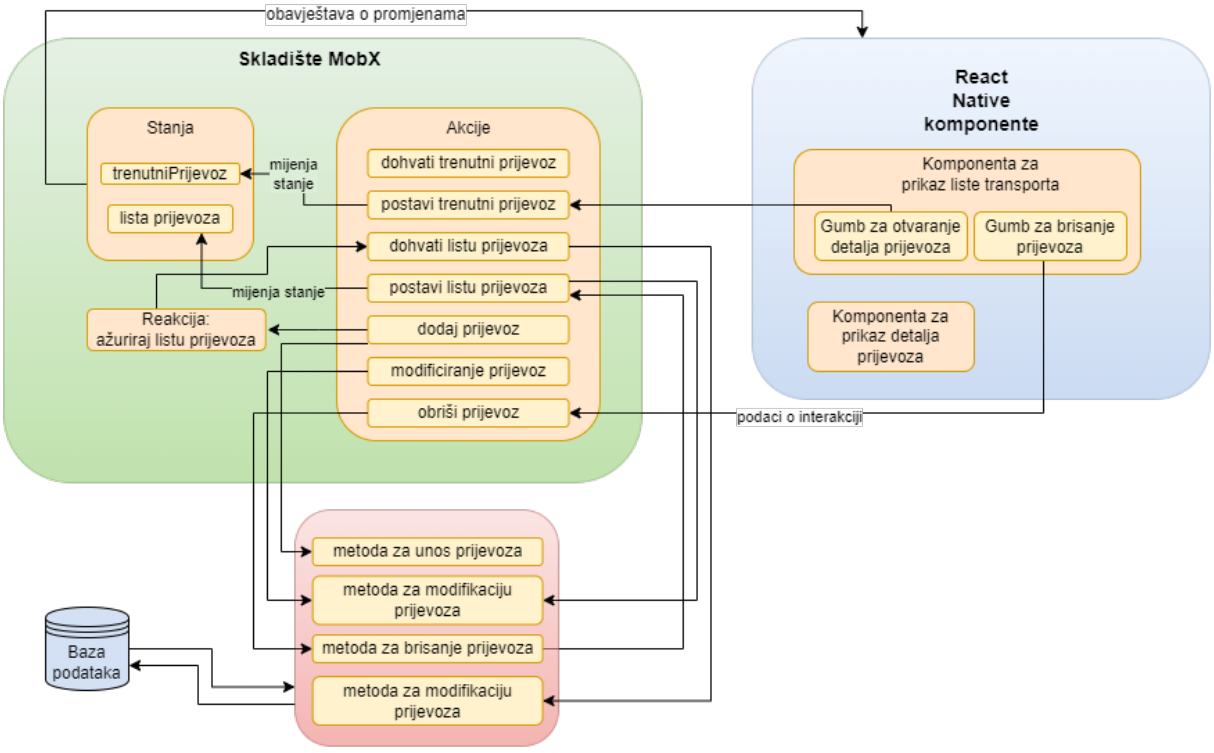
podataka ili potrebnih dodatnih operacija nad podacima prije prikaza korisniku, a koje nemaju veze sa modelom, također se mogu izvršavati u kontroleru tako da kontroler preda prikazu podatke koji su u potpunosti spremni za prikaz.

Implementacija MVC modela pokazala se vrlo jednostavnom i jasnom. Također, pokazalo se da je ovaj model prikladan za jednostavnije sustave s niskom razinom kompleksnosti logike. U suprotnom, ova arhitektura rezultirala bi otežanim nadograđivanjem, održavanjem i otklanjanjem pogrešaka.

Arhitektura MVC prikladnija je od MVVM za mnoge projekte jer je lakša za razumijevanje te jednostavnija i brža za implementaciju. MVC jasno razdvaja podatke i podatkovne transakcije, korisničko sučelje i poslovnu logiku čineći ga pristupačnijim za projekte veće složenosti. Budući da MVC ima manju krivulju učenja i koristi jednostavniji pristup, često je bolji izbor za projekte gdje složene funkcionalnosti nisu potrebne. Međutim, ni MVC ni MVVM nisu idealne za male sustave ili jednostavne aplikacije. Obje arhitekture uvode dodatnu kompleksnost razdvajanjem odgovornosti u više slojeva, što može biti prekomjerno za aplikacije koje nemaju mnogo stanja ili kompleksnije poslovne logike. U takvim slučajevima preporuča se pristup korištenja drugih arhitekturalnih modela ili arhitekture koje se više posvećuju upravljanju stanjem nego čistoj arhitekturalnoj slojevitosti. Na taj način postiže se veća efikasnost i olakšano održavanje.

9. Implementacija MobX vrste arhitekture

MobX vrsta arhitekture na primjeru aplikacije implementirana je na funkcionalnosti unosa zapisa dana, ažuriranja dana i dostupnosti selektiranog dana u kontekstu stanja aplikacije. MobX arhitektura sastoji se od nekoliko slojeva, a to su promatrani podaci (eng. Observables) odnosno stanja, akcija, reakcija i izračunatih vrijednosti odnosno derivacija (ukoliko su potrebne). S obzirom na to da je temeljni princip MobX arhitekture reaktivno programiranje, MobX se najviše fokusira na upravljanje stanjem aplikacije te je glavni cilj otkloniti neke nepotrebne operacije kao što je ručno osvježavanje prikaza, odnosno ažuriranje podataka koji se prikazuju korisniku nakon izvršenja određenih radnji. U primjeru na aplikaciji arhitektura MobX zadužena je za upravljanje stanjem trenutnog dana tako da prilikom klika na određeni dan stanje se podešava na taj dan te se toj vrijednosti može pristupiti u bilo kojoj komponenti koja se navede kao promatrač (eng. observer), odnosno komponenti koja sluša promjene tog konkretnog stanja. U nastavku je prikazan dijagram implementirane arhitekture na primjeru.



Slika 17: Dijagram implementirane arhitekture MobX (Izvor: Vlastita izrada)

Na slici 17. prikazan je dijagram implementirane arhitekture MobX na aplikaciji za planiranje putovanja. Prikazuje se dio vezan za funkcionalnosti prijevoza, točnije prikaz liste svih prijevoza, prikaz detalja o prijevozu, unos novog zapisa te brisanje i modifikacija postojećeg zapisa prijevoza. U skladištu MobX-a nalaze se sva stanja, akcije i reakcije. U nekim slučajevima kada u aplikaciji postoje posebne funkcije preračunavanja ili izračuna vrijednosti za prikaz, kao što bi bilo izračunavanje ukupne cijene na računu prema količini i jediničnoj cijeni, takva bi se funkcija zvala derivacija. U ovom slučaju nema takvih funkcija pa nema ni potrebe za derivacijama. Na dijagramu se može vidjeti i kako podaci teku kroz slojeve, primjerice prilikom unosa novog prijevoza poziva se akcija koja poziva metodu za unos u bazu podataka te se sukladno

tome okida reakcija koja osvježava listu prijevoza u stanju. Nakon toga, sve komponente React Native-a koje slušaju stanje liste prijevoza su obaviještene o promjeni te se ponovno renderiraju kako bi korisniku prikazale ažuriranu listu prijevoza na prikazu liste prijevoza u aplikaciji. U nastavku će biti detaljnije opisano kako je arhitektura MobX implementirana na spomenutoj aplikaciji.

9.1. Postavljanje okruženja

MobX je jedna od arhitektura koje su prilično često korištene u današnjem razvoju mobilnih aplikacija u React Native okruženju. S obzirom na to, razvijeni su određeni paketi koji pomažu u korištenju i implementaciji MobX arhitekture. Prije početka rada korištenjem MobX arhitekture, potrebno je instalirati pakete navedene u sljedećoj naredbi:

```
npm install mobx-react
```

Programski kod 11: Naredba za instalaciju paketa za korištenje MobX-a (vlastita izrada)

Prema službenoj dokumentaciji MobX-a, paket *mobx-react* omogućuje vrlo jednostavno upravljanje stanjem aplikacije korištenjem promatranih objekata i automatski ažurira sučelje na način da spaja MobX promatranja stanja s React komponentama što rezultira automatskim osvježavanjem komponenata u slučajevima promjena stanja u MobX promatranim stanjima. Također, MobX je vrlo efikasan i rezultira boljim performansama i bržem radu aplikacije jer osvježava isključivo one komponente koje sadrže modificirana stanja, dok ostale komponente ostaju netaknute.

9.2. Definiranje promatranih svojstava i definiranje skladišta

Promatrani podaci (eng. Observables) su objekti koji prate promjene u svom stanju te kada označimo neko svojstvo kao observable, njihove promjene se automatski detektiraju od strane MobX-a te se sukladno tim promjenama na njih reagira. To omogućava da se svaki puta kada se podaci osvježe, ažuriraju se i podaci na prikazu odnosno u komponentama koje ta promatrana stanja slušaju.

U sljedećem isječku koda prikazana je implementacija promatranih objekata i njihov konstruktor.

```
class DayStore {

  selectedDay = null;
  days = [];
  loading = false;

  constructor() {
    makeAutoObservable(this);
```

```
  }
  // ostale metode (akcije i reakcije)
}

const dayStore = new DayStore();
export default dayStore;
```

Programski kod 12: Implementacija promatranih objekata u skladištu DayStore (vlastita izrada)

Klasa *DayStore* sadrži svojstva koje u ovom kontekstu zovemo promatrani podaci. U konstruktoru klase potrebno je pozvati metodu *makeAutoObservable()* iz *mobx-react* paketa koja pretvara sva svojstva klase u reaktivne elemente, iako se to može postići i deklariranjem svojstava kao *observable*. Na taj način postiže se reaktivnost stanja bez ručnog definiranja i eksplicitnog korištenja drugih tzv. dekoratora iz MobX-a. Nakon definiranja klase i njenih metoda, potrebno je instancirati objekt *DayStore*, odnosno stvoriti novu instancu klase *DayStore* i spremiti je u konstantu *dayStore* kako bi je bilo moguće izvesti kao zadani izvoz modula i koristiti u drugim dijelovima aplikacije. Tu instancu objekta nazivamo skladištem koje sadrži promatrana svojstva, odnosno vrijednosti trenutnog stanja. Ovakav pristup osigurava da je instanca klase dostupna kao globalna varijabla, no samo u onim dijelovima u kojima se poziva njezino korištenje putem funkcije *import*.

MobX također podržava kreiranje više skladišta za jedan sustav. Iz tog razloga MobX arhitekturu karakteriziramo kao fleksibilnu. U sklopu demonstracije korištenja više skladišta i njihove međusobne povezanosti izrađeno je skladište za prijevoz odnosno *transport*.

U nastavku je prikazano drugo izrađeno skladište koje sadrži promatrana svojstva za potrebe komponenata vezanih za zapise prijevoza.

```
import { makeAutoObservable, runInAction } from "mobx";
import {
    createTransport,
    getAllTransportsPerDay,
    updateTransport,
    deleteTransport
} from '../../database/models/Transport';
import dayStore from './DayStore';

class TransportStore {

  selectedTransport = null;
  transports = [];
  loading = false;

  constructor() {
    makeAutoObservable(this);
  }
  //ostale metode (akcije i reakcije)
}
const transportStore = new TransportStore();
```

```
export default transportStore;
```

Programski kod 13: Implementacija skladišta MobX-a za prijevoze (vlastita izrada)

Na prethodnom isječku koda može se vidjeti skladište MobX-a za prijevoze. U skladištu su definirana promatrana stanja *selectedTransport,* lista *transports* i varijabla *loading* koja služi kao indikator mijenjaju li se ili učitavaju podaci u danom trenutku. Na ovom primjeru može se vidjeti slučaj gdje jedno skladište u svojim akcijama koristi promatrane podatke drugog skladišta.

9.3. Implementacija akcija

U slučaju korištenja više skladišta gdje jedno skladište koristi podatke iz drugog skladišta ili poziva metodu iz drugog skladišta vrlo je važno osigurati da između dvaju skladišta ne postoji dvostruka veza jer u tom slučaju postoji cirkularna ovisnost između dva skladišta. U ovom slučaju, u skladištu *TransportStore* koriste se podaci iz skladišta *DayStore* što je implementirano uvozom instance *dayStore*. U isječku koda u nastavku biti će prikazana metoda u kojoj se koristi podatak iz skladišta dana.

```
async addTransport(description, type, from, to, departureTime, arrivalTime) {
    this.loading = true;
    const day = dayStore.getSelectedDay();
    try {
      await createTransport(day.DayID, description, type, from, to, departureTime,
          arrivalTime);
      runInAction(() => {
        this.setTransports(day.DayID);
        this.loading = false;
      });
    } catch (error) {
      console.error("Failed to a transport in database:", error);
      runInAction(() => {
        this.loading = false;
      });
    }
  }
```

Programski kod 14: Metoda za dodavanje prijevoza (vlastita izrada)

U isječku koda prikazana je asinkrona funkcija, odnosno akcija *addTransport* koja prima podatke o prijevozu kao ulazne argumente te prije kreiranja novog zapisa transporta u bazi podataka dohvaća trenutno stanje varijable dana u skladištu podataka te preko funkcije *getSelectedDay()* pristupa trenutnom stanju. Nakon dohvaćanja trenutnog dana skladištu, metoda poziva metodu *createTransport* koja se nalazi u modelu i služi za slanje transakcije bazi podataka koja izvodi SQL naredbu za unos novog retka.

```
  async updateTransportInDatabase(description, type, from, to, departureTime,
      arrivalTime) {
```

```
    this.loading = true;
    try {
      await updateTransport(this.selectedTransport.TransportID, description, type,
          from, to, departureTime, arrivalTime);
      runInAction(() => {
        this.selectedTransport = { ...this.selectedTransport, description, type,
            from, to, departureTime, arrivalTime };
        this.setTransports(this.selectedTransport.DayID);
        this.loading = false;
      });
    } catch (error) {
      console.error("Failed to update transport in database:", error);
      runInAction(() => {
        this.loading = false;
      });
    }
  }
```

Programski kod 15: Metoda za modifikaciju prijevoza u bazi podataka (vlastita izrada)

Kako je prikazano na prethodnom isječku koda, u skladištu se također nalaze metode za modifikaciju i brisanje skladišta iz baze podataka kao i funkcija za dohvaćanje trenutno odabranog prijevoza i postavljanje istog. To stanje je potrebno u aplikaciji prilikom modificiranja jer otvaranjem forme za prikaz detalja u skladište se sprema odabrani prijevoz te prilikom klika na gumb za modifikaciju skladišta i za spremanje promjena, skladište već zna koji zapis u bazi podataka treba promijeniti pa se takvim načinom smanjuje broj argumenata koji se prosljeđuju putem ulaznih ili izlaznih argumenata akcije. Važno je primijetiti kako nakon uspješnog modificiranja podataka u bazi, akcija mijenja stanje liste transporta. S obzirom na to da komponenta za prikaz prijevoza tu listu promatra odnosno sluša njezine promjene, ona se automatski ponovno renderira i kao rezultat korisniku prikazuje ažuriranu listu prijevoza.

U klasi gdje su navedena svojstva koja predstavljaju promatrana svojstva navode se i metode pripadajuće klase. Te metode zovemo akcije i one služe za sve operacije koje se događaju nad podacima spremljenim u promatranim svojstvima. Akcije osiguravaju siguran i kontroliran način promjena stanja unutar prethodno definiranih granica kako bi MobX mogao pravilno pratiti promjene i reagirati u skladu s njima. U akcije je također potrebno uključiti sve kompleksnije operacije kako bi se smanjio broj potrebnih ažuriranja, no poželjno je i pripaziti na to da svaka metoda i dalje ima jednu i samo jednu zadaću. Akcije su slične reakcijama te mogu čak i uključivati pozivanje reakcija, iako u složenim sustavima reakcije imaju tzv. dekoracije, odnosno svojstva koja im se pridodaju kako bi mehanizam MobX-a prepoznao da se radi o reakciji.

Primjerice, u akcije spadaju *get* i *set* metode, kao i metode za dodavanje, brisanje i ažuriranje podataka u bazi. Uz akcije, postoje i objekti koji se zovu reakcije, a to su one metode koje djeluju kao reakcije na neke određene akcije koje se pozovu i izvrše. U sljedećem isječku koda prikazana je metoda odnosno akcija koja se nalazi u prethodno prikazanoj klasi.

```
async updateDayInDatabase(day) {
    this.loading = true;
```

```
    try {
      const result = await updateDayDescription(day.DayID, day.Description);
      runInAction(() => {
        this.selectedDay = { ...this.selectedDay, Description: day.Description };
        this.setDays(this.selectedDay.TripID);
        this.loading = false;
      });
    } catch (error) {
      console.error("Failed to update day in database:", error);
      runInAction(() => {
        this.loading = false;
      });
    }
  }
```

Programski kod 16: Metoda za modifikaciju zapisa dana u bazi podataka (vlastita izrada)

Prikazana metoda *updateDayInDatabase(day)* prima kao argument objekt *day* koji sadrži id vrijednost dana koji se modificira i novu vrijednost opisa dana. U kontekstu MobX objekata, navedena metoda predstavlja akciju, a unutar nje nalazi se svojevrsna reakcija. Unutar navedene akcije nalazi se poziv na MobX funkciju *runInAction()* koja omogućuje definiranje reakcija. Reakcije su promjene stanja unutar jednog reaktivnog okvira koje se događaju kao efekt neke akcije. Ako postoji više ishoda neke akcije, potrebno je navesti reakciju za svaki mogući ishod. U ovom slučaju, prilikom uspješnog ažuriranja baze podataka, stanje *loading* mijenja se u negativno i selektirani dan odnosno vrijednost u stanju *selectedDay* mijenja se na ono stanje koje je trenutno spremljeno u bazi odnosno ažurirano. No, ono što čini reakciju u ovom slučaju je promjena u promatranim podacima koja se dogodila kao rezultat promjene nekog drugog promatranog podatka, a u ovom slučaju to je pozivanje metode *setDays* koja nakon modifikacije jednog zapisa dana ažurira listu dana kako bi korisnik prilikom povratka na listu dana mogao vidjeti ažurirane opise dana. Način na koji se to postiže je osiguravanje reaktivnosti komponenata koje koriste promatrana stanja.

9.4. Pristup skladištu i akcijama u React Native komponentama

U sljedećem isječku koda prikazano je korištenje stanja upravljanih MobX-om u komponentama React Native okruženja. Konkretno, radi se o kartici za prikaz jednog zapisa putovanja na listi putovanja, odnosno o komponenti *TripCard*.

```
//...
import { inject, observer } from 'mobx-react';

const TripCard = inject('dayStore')(observer((
    { dayStore,
      trip,
      navigation,
      onRefresh
```

```
    }) => {

  const handleSeeDetails = async () => {
    await dayStore.setDays(trip.TripId);
    navigation.navigate('TripDetails', { trip });
  };

//...

  }));

export default TripCard;
```

Programski kod 17: Primjer korištenja stanja na kartici putovanja (vlastita izrada)

Kako bi bilo moguće pristupiti promatranim stanjima u komponentama potrebno je definirati MobX funkcije *inject* i *observe* iz *mobx-react* paketa koji služe za integraciju stanja i komponenata omogućujući reaktivnost aplikacije. Funkcija *observer* pretvara React komponentu u reaktivnu komponentu što znači da će se ta komponenta automatski renderirati kada se promijeni bilo koji promatrani podatak odnosno stanje koje ta komponenta koristi. Funkcija *inject* omogućuje injekciju MobX store komponente, odnosno podatkovnog skladišta kao svojstva komponenente što olakšava pristup podacima iz skladišta bez potrebe za eksplicitnim prosljeđivanjem, kao što je to slučaj u MVC arhitekturi.

Komponenta *TripCard* sadrži gumb *See Details* na čiji klik se otvaraju pojedinosti putovanja, a u sklopu toga i lista dana koji su dodani u bazi podataka nakon dodavanja jednog zapisa putovanja. Klikom na gumb poziva se akcija *setDays(tripId)* iz *dayStore* klase koja prima id putovanja koje je odabrano. Akcija je prikazana u isječku koda nastavku.

```
// u klasi dayStore:

//...
async setDays(tripId) {
    try {
      const fetchedDays = await getAllDaysPerTrip(tripId);
      this.days = fetchedDays;
    } catch (error) {
      console.error('Failed to fetch days:', error);
    }
  }
//...
```

Programski kod 18: Primjer akcije za podešavanje liste dana u skladištu dana (vlastita izrada)

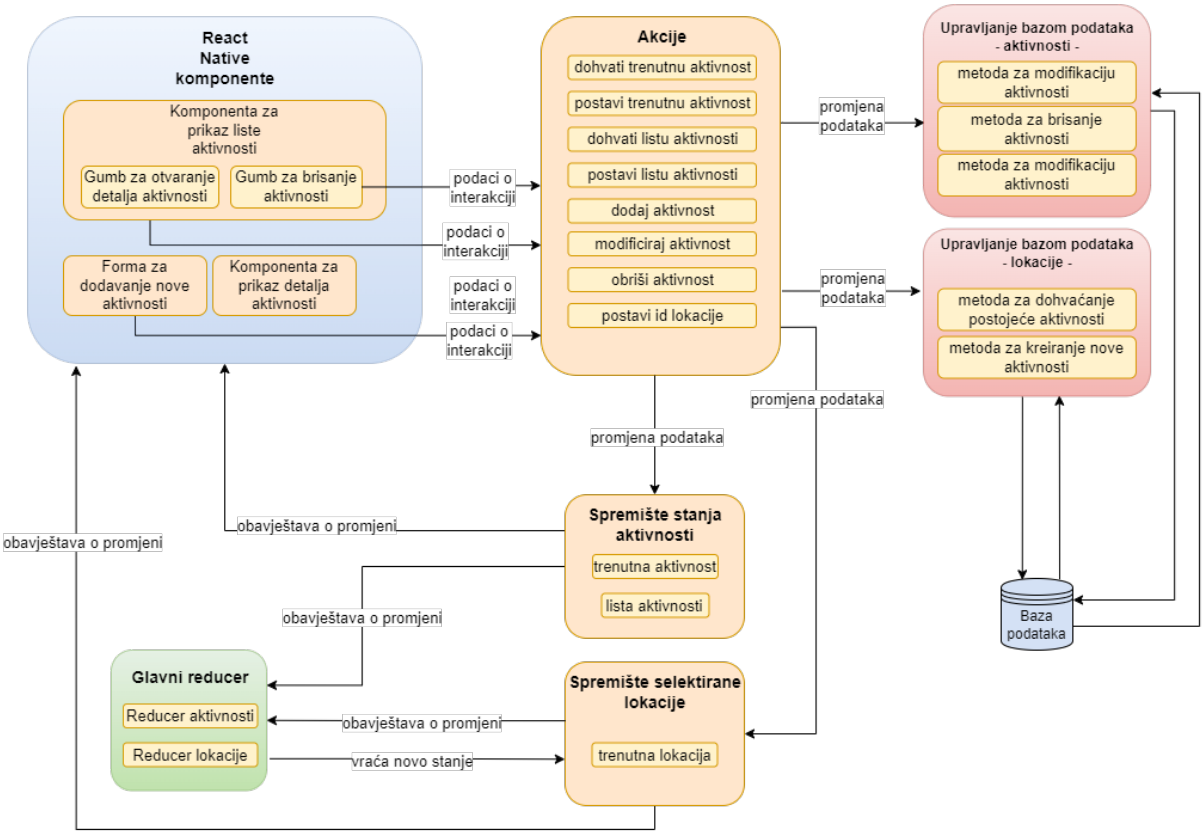
Navedena akcija *setDays(tripId)* poziva metodu *getAllDaysPerTrip(tripId)* koja u sebi sadrži poziv na bazu podataka, odnosno transakciju s upitom kojim dohvaća sve dane povezane s putovanjem proslijeđene id vrijednosti. Ta metoda nalazi se u posebnoj komponenti koja služi kao svojevrsni model s definiranim upitima na bazu, no može se navesti i u metodi *setDays(tripId)*. Nakon poziva metode i spremanja rezultata u konstantu *fetchedDays* akcija sprema rezultat u promatrano stanje *days*, odnosno listu dana. Promjenom te liste dana ren-

derira se komponenta koja sluša promjene, a to je komponenta (ili više njih) koja dohvaća konkretnu listu, te se ovim načinom definira reaktivnost komponente. U ovom slučaju komponenta se ne renderira ponovno, već se samo dohvaća lista svih dana koja će nakon klika na gumb *See Details* na kartici *TripCard* podesiti listu dana te će sljedeća stranica, *TripDetails*, imati gotovu listu koju će samo pročitati iz promatranog objekta *days* u skladištu koje je s komponentom također povezano pomoću ranije spomenutih funkcija *inject* i *observe*.

Arhitektura MobX pokazala se vrlo jednostavnom, fleksibilnom i intuitivnom. Posjeduje odličnu metodiku upravljanja stanjem čime pridonosi reaktivnosti aplikacije. Pristup temeljen na promatranim podacima i automatskoj sinkronizaciji stanja s korisničkim sučeljem čini je idealnom za aplikacije koje trebaju brz i efikasan prikaz podataka. Arhitektura MobX je posebno korisna za aplikacije manje i srednje veličine gdje je važna brzina i reaktivnost.

10. Implementacija Redux vrste arhitekture

Redux vrsta arhitekture na primjeru aplikacije implementirana je na funkcionalnosti koje se dotiču aktivnosti, a odnosi se na dohvaćanje liste aktivnosti, unos nove aktivnosti i brisanje. Također, obuhvaća i upravljanje stanjem lokacije prilikom odabira lokacije na formi za unos nove aktivnosti kao i na formi za modifikaciju aktivnosti i izmjene trenutno odabrane lokacije te prikaza iste na komponenti karte i na karti u punom zaslonu. Kao što je ranije spomenuto, arhitektura Redux u okruženju React Native definira 4 različita sloja, a to su akcije, reduceri, spremište i komponenta React Native. U nastavku biti će prikazana implementacija svakog od slojeva na aplikaciji.



Slika 18: Dijagram implementirane arhitekture Redux (Izvor: Vlastita izrada)

Na slici 18. prikazan je dijagram implementirane arhitekture Redux gdje možemo vidjeti kako teku podaci i informacije kroz različite slojeve arhitekture. Nakon interakcije korisnika sa sučeljem odnosno komponentom React Native-a pozivaju se metode u akciji vezane uz promjenu podataka ili dohvaćanje istih. Komponenta sluša promjene stanja u spremištu i na taj način korisniku prikazuje ažurne podatke. Akcije su metode koje primaju podatke od komponente, primjerice podatke o novom zapisu aktivnosti ili informaciju da se iz baze treba obrisati neki zapis. Akcije pozivaju metode za promjenu podataka u bazi te mijenjaju podatke u spremištu. Spremište obavještava reducere o promjeni te reducer na temelju toga vraća novo stanje. Kako je spomenuto ranije, Redux nema striktna pravila u vezi komunikacije s bazom, nego se više fokusira na upravljanje stanjem u spremištu i konzistentnosti podataka koji se prikazuju

korisniku. Iz tog razloga, u praksi se često kombinira više vrsta arhitektura, a Redux se najčešće kombinira s MVC arhitekturom. U ovom primjeru može se vidjeti da ova arhitektura ima neka obilježja slična onima u arhitekturi MVC, gdje bi React Native komponente bile ekvivalentne pogledu, metode za upravljanje bazom podataka modelu, a akcije, spremište i reduceri kontroleru.

10.1. Postavljanje okruženja

Kao što je to slučaj kod arhitekture MobX, za korištenje arhitekture Redux također su izrađeni paketi pomoćnih funkcija za korištenje te arhitekture. Za početak razvoja aplikacije korištenjem arhitekture Redux i okruženja React Native potrebno je instalirati pakete koji služe kao pomoć kod kreiranja potrebnih slojeva i povezivanje istih u skladnu funkcionalnu i reaktivnu cjelinu. Paketi se instaliraju u aplikaciju pomoću naredbe navedene u nastavku.

```
npm install redux react-redux
npm install redux-thunk
```

Programski kod 19: Naredbe za instalaciju paketa za korištenje Redux-a (vlastita izrada)

U paketima *redux, react-redux* i *redux-thunk* nalaze se pomoćne funkcije koje olakšavaju rad sa Redux skladištima i stanjima. Paket *redux-thunk* je specifičan paket koji služi kao posrednik za pisanje asinkronih akcija. Asinkrone funkcije su obično one koje služe za komunikaciju s bazom, kao primjerice dohvaćanje aktivnosti po odabranom danu. Posrednik za asinkrone akcije omogućuje da se odgodi izvršenje nekih sinkronih funkcija prije nego što je neka asinkrona funkcija izvršila svoju zadaću.

10.2. Implementacija tipova akcija i reducera

Za početak korištenja arhitekture Redux potrebno je definirati tipove akcija. Tipovi akcija definiraju vrstu promjene koja će se izvršiti nad podacima u Redux skladištu. Sami po sebi tipovi akcija nemaju neku veliku funkciju osim što olakšavaju prepoznavanje i klasifikaciju promjena stanja u skladištu. U principu, Redux arhitektura može se implementirati i bez tipova akcije, no u tom slučaju programski kod postaje nejasan, naročito kod kompleksnijih sustava. U nastavku prikazan je primjer definiranja tipova akcija.

```
export const SET_LOCATION = 'SET_LOCATION';
export const CREATE_ACTIVITY = 'CREATE_ACTIVITY';
export const CREATE_ACTIVITY_SUCCESS = 'CREATE_ACTIVITY_SUCCESS';
export const CREATE_ACTIVITY_FAILURE = 'CREATE_ACTIVITY_FAILURE';
export const SET_CURRENT_ACTIVITY = 'SET_CURRENT_ACTIVITY';
export const CLEAR_CURRENT_ACTIVITY = 'CLEAR_CURRENT_ACTIVITY';
export const FETCH_ACTIVITIES = 'FETCH_ACTIVITIES';
export const DELETE_ACTIVITY = 'DELETE_ACTIVITY';
export const UPDATE_ACTIVITY = 'UPDATE_ACTIVITY';
```

Programski kod 20: Definicija tipova akcija u arhitekturi Redux (vlastita izrada)

Dakle, za svaku vrstu akcije koja će se u sustavu dogoditi potrebno je definirati tip akcije kojim će se u reduceru definirati i izvršiti neka operacija. U sljedećem isječku koda prikazan je reducer za aktivnosti u kojem se koriste i spomenuti tipovi akcija.

```
import {
    CREATE_ACTIVITY,
    CREATE_ACTIVITY_SUCCESS,
    CREATE_ACTIVITY_FAILURE,
    SET_CURRENT_ACTIVITY,
    CLEAR_CURRENT_ACTIVITY,
    FETCH_ACTIVITIES
} from './actions';

const initialState = {
  activities: [],
  currentActivity: null,
  loading: false,
  error: null,
};

const activityReducer = (state = initialState, action) => {
  switch (action.type) {
    case CREATE_ACTIVITY:
      return {
        ...state,
        loading: true,
        error: null,
      };
    case CREATE_ACTIVITY_SUCCESS:
      return {
        ...state,
        loading: false,
        activities: [...state.activities, action.payload],
      };
    case CREATE_ACTIVITY_FAILURE:
      return {
        ...state,
        loading: false,
        error: action.payload,
      };
      case SET_CURRENT_ACTIVITY:
        return {
          ...state,
          currentActivity: action.payload,
        };
      case CLEAR_CURRENT_ACTIVITY:
        return {
          ...state,
          currentActivity: null,
        };
        case FETCH_ACTIVITIES:
      return {
```

```
        ...state,
        activities: action.payload,
      };
    default:
      return state;
  }
};

export default activityReducer;
```

Programski kod 21: Primjer reducera s korištenjem tipova akcija (vlastita izrada)

U prethodnom isječku koda možemo vidjeti reducer za aktivnosti te uvoz spomenutih tipova. Kao što je spomenuto u prethodnim poglavljima, reducer je funkcija koja uzima trenutno stanje i akciju koju je potrebno provesti te izvršava akciju nad stanjima. Potrebno je također definirati stanja te njihove inicijalne vrijednosti. Primjerice, u ovom slučaju postoji definirana lista aktivnosti i vrijednost trenutne aktivnosti koja se popunjava prilikom zahtjeva korisnika za otvaranjem detalja neke aktivnosti. U nastavku koda postoji grananje *switch* koje, sukladno potrebnom tipu akcije, izvršava neku operaciju nad spremljenim stanjima. Princip korištenja arhitekture Redux je takav da za svaku funkcionalnost ili modul mora postojati zaseban reducer kako bi se osigurala kvalitetnija strukturiranost, a samim time i skalabilnost te čitljivost koda. Iako Redux zastupa zasebne reducere, skladište podataka je centralizirano. Kako bi se omogućilo centralizirano skladište sa više reducera, postoje metode kojima se reduceri spajaju te se kreira jedan glavni reducer, tzv. *root reducer*. U nastavku će biti prikazan primjer glavnog reducera koji u slučaju ove aplikacije spaja tri različita reducera, a to su reducer za lokaciju, reducer za selektiranu lokaciju i reducer za aktivnosti.

```
import { combineReducers } from 'redux';
import locationReducer from './locationReducer';
import activityReducer from './activityReducer';
import selectedLocationReducer from './selectedLocationReducer';

const rootReducer = combineReducers({
  location: locationReducer,
  activityState: activityReducer,
  selectedLocation: selectedLocationReducer,
});

export default rootReducer;
```

Programski kod 22: Primjer glavnog reducera (vlastita izrada)

U isječku koda može se vidjeti implementacija centraliziranog skladišta koristeći funkciju *combineReducers()* iz Redux paketa. Funkcija služi za kombiniranje odnosno spajanje više manjih reducera u jedan glavni reducer. Kao argumente prima dijelove stanja i kao vrijednosti tih stanja navode se reduceri koji upravljaju konkretnim dijelom stanja. U ovom slučaju to su dijelovi *location, activityState* i *selectedLocation* te sukladno njima njihovi reduceri. Glavni reducer je izvedba koja ima i alternativu, a to je korištenje jednog reducera za sva stanja. No, to je praksa koja se uglavnom izbjegava iz razloga što u aplikacijama srednje i veće veličine

nije praktično koristiti jedno skladište s obzirom na to da je u pitanju velika količina podataka. Takvim se pristupom dolazi do nepregledne, nejasne i nepotrebno velike funkcije s niskom održivosti.

Nakon definiranja reducera za potrebe aplikacija te kreiranja glavnog reducera dolazi se do koraka kreiranja skladišta (eng. store). Kao što je spomenuto u prethodnim poglavljima, skladište u Redux-u služi kao centralno mjesto gdje se pohranjuju stanja aplikacije te se u arhitekturi Redux smatra jedinim izvorom istine odnosno ispravnih podataka. Zadaće skladišta su omogućiti svim komponentama u aplikaciji pristup skladištenim stanjima poput lokacije i podataka o aktivnosti, primanjem i slanjem akcija unosa, dohvaćanja, brisanja i modificiranja pomoću funkcije *dispatch*. U nastavku prikazana je implementacija skladišta.

```
import { legacy_createStore as createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import rootReducer from './rootReducer';


const store = createStore(
  rootReducer,
  applyMiddleware(thunk)
);

export default store;
```

Programski kod 23: Primjer implementacije skladišta u arhitekturi Redux (vlastita izrada)

U isječku koda možemo vidjeti implementaciju skladišta koje obuhvaća glavni reducer, odnosno *rootReducer*. Funkcija *createStore* kreira skladište te uključuje u njega glavni reducer te funkciju *applyMiddleware* koja omogućava dodavanje ranije spomenutog posrednika u Redux skladište za upravljanje asinkronim operacijama. Ovakvom implementacijom kreiralo se centralizirano skladište na kojem se zapravo i temelji cijela Redux arhitektura.

10.3. Implementacija Redux slojeva na komponentama React Native-a

Efikasnosti u arhitekturi Redux uvelike doprinosi način na koji se dohvaćaju stanja. Naime, stanjima se pristupa isključivo u onim komponentama u kojima se ona koriste. Druge komponente nemaju potrebe imati pristup stanjima niti slušati njihove promjene. Kao i kod MobX arhitekture, Redux arhitektura zahtijeva uključivanje komponente pružatelja (eng. Provider) u aplikaciju na onom mjestu gdje će se koristiti stanja iz skladišta. Način primjene pružatelja je sličan primjeni navigacijskih komponenata u React Native aplikacijama, a to je omatanje cijele aplikacije u *Provider* komponentu. Iako je pružatelj osmišljen da omota onaj dio aplikacije koji će koristiti stanja iz skladišta, u manjim aplikacijama to nije važno jer ne doprinosi performansama niti djeluje na čitljivost i jasnoću koda. No, s obzirom na to da je Redux osmišljen kao arhitektura sa centraliziranim skladištem gdje se svi podaci nalaze u jednom skladištu, ta činjenica povlači i potrebu za omatanjem cijele aplikacije u komponentu *Provider*. U nastavku je

prikazan isječak koda gdje se vidi omatanje aplikacije u *Provider*.

```
import { Provider as ReduxProvider } from 'react-redux';
import store from './src/redux_location/store';

export default function App() {
  return (
    <MobXProvider dayStore={dayStore} transportStore={transportStore}>
        <ReduxProvider store={store}>
          <AppNavigation />
        </ReduxProvider>
    </MobXProvider>
  );
}
```

Programski kod 24: Primjer omatanja aplikacije u komponentu pružatelja (vlastita izrada)

U prethodnom isječku koda prikazana je implementacija komponente Provider u aplikaciji definirana ključnom riječi *ReduxProvider* te je također definirano i uvezeno skladište kojem *Provider* daje pristup. U nastavku biti će prikazano na koji način komponente u aplikaciji pristupaju stanjima u skladištu te kako se pokreću određene akcije.

```
const AddActivity = inject('dayStore')(observer(({ dayStore, navigation }) => {

  const dispatch = useDispatch();
  const location = useSelector((state) => state.location.location);
  const day = dayStore.getSelectedDay();

  //...ostale lokalne metode i lokalna stanja

  const handleAddActivity = async () => {
    try {
       if (!location) {
        throw new Error('Please␣select␣a␣location');
      }
      dispatch(createActivityAction(day.DayID, name, description, startTime, endTime
          , location));
      Alert.alert('Success', `Activity "${name}" has been added successfully!`);
      navigation.goBack();
    } catch (error) {
      console.error('Error␣creating␣activity:', error);
      Alert.alert('Error', `Failed to create activity: ${error.message}`);
    }
  };
return (

        // ...
        <Text style={styles.label}>Location:</Text>
        <TouchableOpacity onPress={() => navigation.navigate('LocationSearchScreen')
            }>
          <TextInput
```

```
            style={styles.input}
            value={location ? location.name : 'Choose location'}
            editable={false}
            placeholder="Location"
          />
        </TouchableOpacity>

        <TouchableOpacity style={styles.addButton} onPress={handleAddActivity}>
          <Text style={styles.addButtonText}>Add Activity</Text>
        </TouchableOpacity>
      </View>
    </ScreenWrapper>
  );

}));

export default AddActivity;
```

Programski kod 25: Primjer pristupanja stanju u skladištu Redux-a (vlastita izrada)

U isječku koda prikazan je selektiran dio komponente *AddActivity* koja služi kao forma za dodavanje nove aktivnosti. S obzirom na to da ova komponenta pristupa nekim stanjima iz MobX dijela arhitekture, kod definicije sadrži i funkcije za pristup MobX skladištu, no to je irelevantno za Redux dio komponente. Za početak rada sa Redux akcijama potrebno je definirati lokalnu konstantu *dispatch* koja poziva funkciju *useDispatch()*. Funkcija spada u specifičnu vrstu funkcija, tzv. React Redux *hooks*. Funkcija *useDispatch()* koristi se za dohvaćanje reference na dispatch funckiju iz Redux skladišta te se sprema u lokalnu konstantu kako bi se pojednostavnilo slanje akcija.

Koristi se također i *useSelector* hook koji služi za dohvaćanje stanja iz Redux skladišta. U ovom slučaju potrebno je definirati lokalno stanje *location* koje prilikom definiranja treba spojiti na skladište kako bi lokalno stanje pratilo promjene u skladištu te kako bi u nastavku koda na komponenti za unos lokacije bila prikazana odabrana lokacija. Stanje lokacije u skladištu mijenja se u komponenti *LocationSearchScreen* kada korisnik odabere željenu lokaciju iz te komponente, a prilikom automatskog povratka na formu za dodavanje aktivnosti Redux automatski ažurira tu vrijednost te se zaslon ponovo renderira i uklanja se potreba za ručnim osvježavanjem podataka na tom zaslonu. Nakon klika na gumb *Add Activity* poziva se funkcija definirana u reduceru za kreiranje nove aktivnosti s prethodno unesenim podacima.

Kako je spomenuto u prethodnim poglavljima, Redux arhitektura ne definira konkretno komunikaciju s bazom podataka, osim što bilo kakve operacije u bazi moraju prolaziti kroz definirane akcije. U praksi, to znači da komunikacija s bazom, kao što su pozivi na API (eng. Application Programming Interface) metode ili slanje SQL transakcija, može biti implementirana upravo u tim akcijama ili pak u zasebnim metodama koje se pozivaju iz tih akcija. U slučajevima korištenja Redux arhitektura najčešća praksa je definirati zasebne metode koje u sebi sadrže kod za komunikaciju s bazom podataka iz razloga što se na taj način postiže modularnost i skalabilnost te se olakšava revizija koda i održavanje same aplikacije. U ovoj aplikaciji korišten je pristup zasebnih metoda, odnosno posebno definiranje modela s kojim komuniciraju akcije.

U nastavku prikazana je akcija *createActivityAction()* s pripadajućim ulaznim parametrima koja se poziva prilikom klika na gumb Add Activity u prethodno prikazanom isječku koda.

```
export const createActivityAction =
    (dayId, name, description, startTime, endTime, location) => async (dispatch) =>
       {
      dispatch({ type: CREATE_ACTIVITY });
      try {
        const locationId = await dispatch(getOrCreateLocation(location));
        const result = await createActivity(dayId, name, description, startTime,
            endTime, locationId);

        dispatch({ type: CREATE_ACTIVITY_SUCCESS, payload: result });
        dispatch(fetchActivities(dayId));

      } catch (error) {
        dispatch({ type: CREATE_ACTIVITY_FAILURE, payload: error.message });
      }
};
```

Programski kod 26: Primjer akcije u Redux-u (vlastita izrada)

U prethodnom isječku koda prikazana je definirana akcija za unos nove aktivnosti. Može se vidjeti način na koji se implementira akcija koja koristeći dispečer da bi obavijestila reducer o zahtjevu za promjenu stanja. Nakon toga poziva se funkcija za promjenom podataka u bazi, što također obavlja akcija. U toj se akciji može navesti cijela metoda za komunikaciju s bazom, kako je objašnjeno i ranije, no u ovom slučaju poziva se eksterna metoda koja je zadužena za komunikaciju s bazom odnosno za prosljeđivanje podataka o novoj aktivnosti. U nastavku je prikazan programski kod metode za navedenu operaciju te će biti pojašnjena metoda koja vraća id vrijednost nove ili postojeće lokacije da bi se mogla vezati za novu aktivnost. Također može se vidjeti da nakon unosa nove lokacije, u slučaju uspjeha, dispečer poziva akciju *fetchActivities(dayId)* za osvježavanje liste aktivnosti kako bi se korisniku odmah nakon unosa nove aktivnosti prikazala ažurirana lista aktivnosti.

```
export const createActivity = (dayId, name, description, startTime, endTime,
    locationId) => {
  return new Promise((resolve, reject) => {
    db.transaction(tx => {
      tx.executeSql(
        insertActivityQuery(dayId, name, description, startTime, endTime, locationId
            ),
        [],
        (_, result) => resolve(result),
        (_, error) => reject(error)
      );
    });
  });
};
```

Programski kod 27: Metoda za kreiranje novog zapisa aktivnosti u bazi podataka (vlastita izrada)

U prethodnom isječku koda prikazana je metoda koja putem reference na bazu podataka šalje transakciju koja sadržava SQL naredbu za kreiranje novog zapisa aktivnosti. Upit je sadržan u posebnoj konstanti naziva *insertActivityQuery* da bi programski kod bio jasniji i pregledniji.

S obzirom na to da je jedan od aplikacijskih zahtjeva potreba za odabirom lokacije u posebnoj komponenti, izrađena je akcija *getOrCreateLocation(location)* koja poziva metodu *createLocation*.

```
export const getOrCreateLocation = (location) => async (dispatch) => {
    dispatch({ type: CREATE_LOCATION_REQUEST });
    try {
        const locationId = await createLocation(location.latitude, location.
            longitude, location.name);
        await dispatch({
            type: CREATE_LOCATION_SUCCESS,
            payload: locationId,
        });
        return locationId;
    } catch (error) {
        console.error('Error inserting location:', error);
        dispatch({
            type: CREATE_LOCATION_FAILURE,
            payload: error.message,
        });
        throw error;
    }
};
```

Programski kod 28: Primjer akcije za dohvaćanje id vrijednosti lokacije (vlastita izrada)

U prethodnom isječku prikazana je navedena metoda koja je zadužena za pozivanje eksterne metode *createLocation()* za pretraživanje baze za odabranom lokacijom te ukoliko ta lokacija već postoji, vraća se njezin id. Ukoliko lokacija ne postoji u bazi podataka, metoda kreira novu lokaciju te vraća id vrijednost novonastalog zapisa aktivnosti u bazi podataka. U nastavku je prikazana implementacija spomenute metode.

```
export const createLocation = (latitude, longitude, placeName) => {
  return new Promise((resolve, reject) => {
    db.transaction(tx => {
      tx.executeSql(
        getLocationId(),
        [latitude, longitude],
        (_, result) => {
          if (result.rows.length > 0) {
            resolve(result.rows.item(0).LocationID);
          } else {
            tx.executeSql(
              insertLocationQuery(latitude, longitude, placeName),
              [latitude, longitude, placeName],
              (_, insertResult) => resolve(insertResult.insertId),
              (_, insertError) => reject(insertError)
```

```
            );
          }
        },
        (_, error) => reject(error)
      );
    });
  });
};
```

Programski kod 29: Primjer metode za dohvaćanje lokacije ili kreiranje nove (vlastita izrada)

Kako je spomenuto ranije, Redux vrsta arhitekture više se fokusira na arhitekturu stanja aplikacije i nema striktna pravila vezana uz implementaciju modela podataka kao što to imaju arhitekture MVC i MVVM. U Redux arhitekturi veći je naglasak na upravljanje stanjem aplikacije. No, ipak ima nekih sličnosti u implementaciji jer se sva poslovna logika implementira na akcijama te se time dolazi do čitljivijeg koda u komponentama koje su uglavnom zadužene samo za prikaz ažurnih i relevantnih podataka korisniku.

Redux arhitektura u implementaciji pokazala se vrlo jasnom i jednostavnom, iako bi sa porastom kompleksnosti aplikacije bilo primjerenije koristiti Flux arhitekturu koja ne zastupa striktno centralizaciju skladišta, već po potrebi ima mogućnost korištenja više skladišta. U velikim sustavima gdje postoji velik broj komponenti koje u suštini nisu povezane ni na koji način, a naročito u kontekstu podataka koje koriste ili prikazuju, nije prikladno korištenje centraliziranog skladišta. U slučaju implementacije takvog skladišta stanja, programski kod aplikacije postao bi previše složen, težak za razumijevanje, a također bi održavanje i nadogradnja takve aplikacije bilo otežano.

11. Zaključak

U usporedbi arhitektura MVC, MVVM, Redux, Flux i MobX može se zaključiti da svaka od njih donosi svoje prednosti u implementaciji za specifičnim sustavima. Svaka od njih zastupa različite pristupe upravljanju stanjem, kao i strukturiranju aplikacija i odgovornosti unutar aplikacije. Može se primijetiti kako arhitektura MVC pruža prilično jednostavan i intuitivan način razdvajanja odgovornosti između slojeva, no upravo to ju čini manje prikladnim za manje aplikacije jer iz nepotrebno komplicira. Arhitektura MVVM je slična arhitekturi MVC, no razlika je u tome što prebacuje dio odgovornosti s modela na sloj model pogled i time još dodatno olakšava vezivanje podataka između modela i prikaza. Preostale tri arhitekture više se fokusiraju na najvažniju zadaću u mobilnim aplikacijama, a to su stanja aplikacija. Može se reći da su Redux, MobX i Flux arhitekture upravljanja stanjem, no one definiraju ponašanje svih dijelova sustava i toka podataka, stoga ih smatramo arhitekturama cjelokupne mobilne aplikacije. Redux i Flux za razliku od MobX-a zastupaju strukturu s jednosmjernim tokom podataka što pomaže u održavanju predvidljivosti stanja u velikim aplikacijama, no za manje aplikacije unose preveliku razinu kompleksnosti te se ne smatraju najboljim rješenjem za manje aplikacije. MobX se svojom fleksibilnošću gotovo savršeno uklapa u kontekst manjih aplikacija, no u velikim sustavima ta fleksibilnost postaje nedostatak. Za potrebe ove aplikacije najboljom se pokazala arhitektura MobX zbog jednostavnosti, lakog shvaćanja toka podataka, izvrsnog praćenja stanja i konzistentnim ažuriranjem podataka. Redux arhitektura pak se s druge strane pokazala upravo onakvom kakvom se spominjala u kontekstu njenih nedostataka, a to je da vrlo jednostavne zahtjeve nepotrebno komplicira.

Česta pojava u razvoju mobilnih aplikacija je kombiniranje više arhitektura, primjerice arhitektura Model - Pogled - Kontroler (MVC) sa arhitekturom Redux. Razlog spajanja tih dviju arhitektura je to što MVC zagovara jasnu i strogu raspodjelu na slojeve, dok Redux ima odličan sustav upravljanja stanjima te takvom hibridnom arhitekturom prednosti obje vrste arhitektura dolaze do izražaja te čine aplikaciju skalabilnom, jednostavnijom za održavanje i jasnijom u smislu programskog koda koji sadrži.

Važno je prilikom odabira arhitekture znati koje prednosti se mogu dobiti od strane koje vrste arhitekture kako bi se odabrala najprikladnija, a odabir ovisi o kompleksnosti zahtjeva, modelu podataka, potrebama i ciljevima projekta.

Popis literature

Alex Kugell. (2021.). Best Software Design Principles for Successful Engineering. https://trio.dev/software-design-principles/

Avishek Kumar. (2024.). Medium. Choosing the Right Architecture for Your React Native App: MVVM vs. MVC vs. MVP. https://blog.stackademic.com/choosing-the-right-architecture-for-your-react-native-app-mvvm-vs-mvc-vs-mvp-57c29d2260aa

Ayo Oladele. (2023.). Velvetech. 5 Key Mobile Development Approaches. https://www.velvetech.com/blog/5-key-mobile-development-approaches/

Eric Kim. (2016). Medium. How to use Redux in React? https://medium.com/@bosung90/using-redux-in-react-749c5295c542

Estefanía García Gallardo. (2023). What is MVVM architecture? https://builtin.com/software-engineering-perspectives/mvvm-architecture

Godwin Ehile. (2023.). The Pros and Cons of Using Redux with React. Hyperskill. (2024.). MVC. https://hyperskill.org/learn/step/17197

Len Bass (first), Paul Clements, Rick Kazman. (2012). Software Architecture in Practice. Addison-Wesley.

Pasindu Weerakoon. (2023.). Medium. An introduction to Flux architecture: Explain the high-level overview, benefits, and how it works. https://medium.com/@wmpasindu/an-introduction-to-flux-architecture-explain-the-high-level-overview-benefits-and-how-it-works-fa7eac915784

Pranjal Mehta. (2024.). Zealous Systems. React Native Mobile App Architecture Guide: Layers, Patterns, Principles. https://www.zealousys.com/blog/react-native-mobile-app-architecture/

Pratik Mistry. (2024.). Medium. A Comprehensive Guide to Mobile App Architecture. https://pratikmistry.medium.com/a-comprehensive-guide-to-mobile-app-architecture-7ff0f864a9e1

Rodrigo Rizzi, Andres Mayes. (bez dat.). Redux vs. MobX: A Comparative Look at State Management in Software Development. https://www.linkedin.com/pulse/redux-vs-mobx-comparative-look-state-management-software-development

Ray Ali. (2023). What are the different mobile operating systems? https://www.uswitch.com/mobiles/guides/mobile-operating-systems/?msockid=278443c57ffb649104f1573e7e7665c9

Robert C. Martin. (2003). Agile Software Development, Principles, Patterns, and Practices.

Sebastien Eggenspieler. (2022.). DashLane. Android UI architecture migration to MVVM. https://www.dashlane.com/blog/android-ui-architecture-mvvm

Simon Brown. (bez dat.-a). Software Architecture for Developers. http://static.codingthearchitecture.com/sddconf2014-software-architecture-for-developers-extract.pdf

Ten Minute Introduction to MobX and React. (bez dat.). Preuzeto 2024., od https://mobx.js.org/getting-started

Popis slika

Popis programskih kodova

Popis priloga

[1] Poveznica na Git repozitorij praktičnog dijela rada:

https://github.com/HelenaPotocki/TravelPlannerApp.git