Istraživanje primene Kotlin Multiplatform tehnologije: Analiza performansi i arhitektonskih rešenja u poređenju sa nativnim razvojem

Danijal Azerović, Bojana Dimić Surla1

Sadržaj **- Ovaj rad istražuje primenu Kotlin Multiplatform tehnologije u razvoju mobilnih aplikacija za Android i iOS platformu. Poseban akcenat stavljen je na poređenje sa nativnim pristupom. Predstavljena je detaljna metodologija razvoja i merenja performansi, uključujući vreme izgradnje, veličinu aplikacije, vreme pokretanja i potrošnju memorije. Rezultati su prikazani kroz analize, a diskusija obuhvata arhitektonske prednosti i ograničenja KMP-a u realnim projektima. Cilj rada je procena isplativosti, održivosti i potencijala za industrijsku primenu Kotlin Multiplatform tehnologije.**

Ključne reči **- Android, arhitektura, iOS, Kotlin Multiplatform, mobilni razvoj, performanse**

I. UVOD

A. Kontekst razvoja aplikacija za mobilne uređaje

Razvoj aplakacija za mobilne uređaje je tokom poslednje decenije doživeo izuzetno brz rast i transformaciju. Sa pojavom pametnih telefona i dominacijom operativnih sistema Android i iOS, aplikacije su postale centralni deo svakodnevnog života korisnika, pokrivajući oblasti od komunikacije i zabave, do zdravlja, obrazovanja i industrijske proizvodnje. Prema istraživanjima tržišta, mobilne aplikacije generišu milijarde preuzimanja godišnje, dok prihodi od mobilne industrije kontinuirano rastu i prevazilaze tradicionalni softver na desktop računarima.

[1]Danijal Azerović, Računarski fakultet, Srbija (email: dazerovic4223m@raf.rs)
Bojana Dimić Surla, Računarski fakultet, Srbija (email: bdimicsurla@raf.rs)

Ovaj trend doveo je do značajnog povećanja zahteva za efikasnim razvojem aplikacija. Organizacije, bilo da su u pitanju startapi ili velike kompanije, suočavaju se sa izazovom da simultano podrže različite platforme i širok spektar uređaja. Istovremeno, korisnici očekuju visok kvalitet, brze performanse i dosledno korisničko iskustvo, bez obzira na operativni sistem.

Tradicionalno, rešenje je bilo razvoj odvojenih nativnih aplikacija - jedna za Android (u Javi ili Kotlinu), a druga za iOS (u Swiftu ili Objective-C-u). Iako ovaj pristup pruža optimalne performanse i maksimalno iskorišćava mogućnosti platforme, on nosi i značajne nedostatke: dupliranje logike, veće troškove razvoja, produženo vreme isporuke i složenije održavanje.

Kako bi se odgovorilo na ove izazove, pojavila su se različita kros-platformska rešenja. Počev od ranijih pokušaja kao što su Xamarin i PhoneGap, preko popularnih frejmvorka poput React Native i Flutter, pa sve do savremenih tehnologija poput Kotlin Multiplatform (KMP), cilj je isti: smanjiti dupliranje koda i ubrzati razvoj, a pritom zadržati kvalitet i stabilnost aplikacija.

B. Problem višestrukog razvoja (Android/iOS)

Jedan od centralnih izazova modernog mobilnog razvoja jeste potreba da se aplikacija razvije i održava za dve dominantne platforme - Android i iOS. Iako obe platforme dele slične principe korisničkog iskustva i funkcionalnosti, razlike u programskim jezicima, razvojnim alatima i arhitekturi aplikacija stvaraju značajnu složenost u procesu.

Za Android se tradicionalno koristi Java, a u poslednjih pet godina sve više Kotlin, dok iOS aplikacije zahtevaju razvoj u Swiftu ili Objective-C-u. To znači da razvojni tim mora posedovati dva seta znanja i veština, što povećava potrebu za dodatnim resursima i podiže troškove. Takođe, poslovna logika - pravila rada aplikacije, obrada podataka, validacije i komunikacija sa serverom - mora biti implementirana dva puta, jednom za svaku platformu.

Osim inicijalnih troškova, problem se dodatno komplikuje u fazi održavanja. Kada je potrebno uvesti novu funkcionalnost ili ispraviti grešku, promene se moraju sinhronizovano implementirati u obe aplikacije. To povećava rizik od nedoslednosti i može dovesti do situacija u kojima se Android i iOS verzija iste aplikacije ponašaju različito.

Drugi izazov je i vreme isporuke. U uslovima gde je brz izlazak na tržište presudan, dupliranje posla usporava razvoj i smanjuje agilnost tima. Ovo je posebno problematično u startup okruženju gde se konkurencija meri u nedeljama ili čak danima.

Zbog svega ovoga, u industriji je nastala snažna potreba za tehnologijama koje bi omogućile ponovnu upotrebu koda i smanjile dupliranje posla. Kros-platformska rešenja, među kojima se u poslednjim godinama ističe Kotlin Multiplatform, upravo su nastala kao odgovor na ovaj problem.

C. Motivacija za KMP

U poslednjim godinama Kotlin Multiplatform (KMP) privukao je značajnu pažnju upravo zato što se nalazi na preseku dve suprotne potrebe:

- s jedne strane, potreba za nativnim performansama i korisničkim iskustvom
- a s druge strane, potreba za ponovnom upotrebom koda i efikasnošću u razvoju

Za razliku od drugih kros-platformskih rešenja poput React Native ili Flutter-a, KMP ne pokušava da u potpunosti zameni nativni UI sloj. Umesto toga, njegova filozofija zasniva se na deljenju poslovne logike i modela podataka, dok se korisnički interfejs i dalje razvija nativno. Time se obezbeđuje da aplikacije zadrže izgled i osećaj u skladu sa standardima svake platforme, a istovremeno se smanjuje količina dupliranog koda.

Dodatni motivacioni faktor je i sve veća podrška zajednice i industrije. Kotlin je zvanično podržan jezik za Android razvoj, što omogućava brže prihvatanje među Android inženjerima. Takođe, zahvaljujući Kotlin/Native kompajleru, isti jezik se može koristiti za generisanje binarnih fajlova koji rade direktno na iOS-u, bez potrebe za virtuelnim mašinama.

KMP je posebno atraktivan za timove i kompanije koje već razvijaju Android aplikacije u Kotlinu, jer omogućava postepenu migraciju ka kros-platformskom razvoju. Umesto da se ceo projekat gradi ispočetka, pojedini delovi logike mogu se postepeno izdvajati u zajednički modul, čime se smanjuje rizik i ubrzava tranzicija.

U kontekstu tržišta koje postavlja sve veće zahteve za brzim razvojem, nižim troškovima i stabilnim performansama, motivacija za usvajanje KMP-a proizilazi iz njegove sposobnosti da kombinuje najbolje od oba sveta - nativnu snagu i kros-platformsku efikasnost.

II. PREGLED LITERATURE

A. Kros-platformska rešenja u mobilnom razvoju

Razvoj mobilnih aplikacija je od samog početka pratio problem višestrukog ciljnog okruženja. Kako su Android i iOS ubrzo postali dominantne platforme, pojavila se potreba za tehnologijama koje bi omogućile razvoj jedne aplikacije

koja može da radi na oba sistema. Tokom poslednje decenije razvijen je čitav niz kros-platformskih rešenja, svako sa svojim prednostima i manama.

Rani pristupi uključivali su tehnologije kao što su PhoneGap i Apache Cordova, koje su se oslanjale na web tehnologije (HTML, CSS, JavaScript). Njihova osnovna ideja bila je „upakovati" web aplikaciju u nativni kontejner i omogućiti joj pristup osnovnim funkcijama telefona. Iako je ovaj pristup bio inovativan, rezultati nisu bili zadovoljavajući - aplikacije su često imale loše performanse i nisu uspevale da pruže iskustvo uporedivo sa nativnim.

Nakon toga pojavili su se Xamarin i slični frejmvorci, koji su nudili pisanje koda u jezicima poput C#, a zatim kompajliranje u aplikacije koje rade na Androidu i iOS-u. Iako je ovaj pristup omogućavao bolju integraciju sa nativnim API-jima, i dalje je postojao problem ograničene fleksibilnosti i dodatne složenosti u radu sa bibliotekama.

React Native (Facebook, 2015) doneo je značajan pomak. On je omogućio pisanje aplikacija u JavaScript-u i renderovanje korisničkog interfejsa preko nativnih komponenti. Time su performanse bile bolje od web hibrida, a razvojni proces brži zahvaljujući velikoj zajednici i brojnim bibliotekama. Ipak, React Native često pati od problema sa kompatibilnošću biblioteka, složenijim debagovanjem i performansama u zahtevnim scenarijima (npr. kompleksna grafika).

Kao odgovor na nedostatke postojećih rešenja, Google je 2017. predstavio Flutter. Flutter koristi Dart jezik i ima sopstveni rendering engine, što omogućava visok stepen konzistentnosti UI-ja na svim platformama. Time je doneo impresivne performanse i brz razvoj, ali i potencijalni problem - aplikacije nisu u potpunosti „nativne", već koriste sopstveni sistem za crtanje interfejsa, što može ograničiti integraciju sa platformom i povećati veličinu aplikacije.

Konačno, Kotlin Multiplatform (KMP) uvodi drugačiju filozofiju: umesto da zameni nativne UI slojeve, on omogućava deljenje poslovne logike i modela podataka između platformi, dok se korisnički interfejs i dalje razvija nativno. Na taj način KMP kombinuje prednosti kros-platformske efikasnosti i nativne fleksibilnosti, što ga razlikuje od svih prethodnih rešenja.

B. Kotlin Multiplatform u akademskim i industrijskim istraživanjima

Akademska istraživanja o Kotlin Multiplatform (KMP) uglavnom se grupišu u tri kategorije: (1) uporedne performanse u odnosu na nativne pristupe i druge kros-platformske okvire; (2) produktivan razvoj i održavanje (količina deljenog koda, brzina isporuke, trošak promena); i (3) arhitektonska evaluacija (granica između zajedničkog modula i platformskog koda, testabilnost,

kompleksnost build sistema). Tipična metodologija podrazumeva izradu parova funkcionalno istih aplikacija (KMP vs. nativno), merenje start-up vremena, potrošnje memorije, veličine binarnih fajlova i vremena izgradnje, a zatim kvalitativnu procenu iskustva razvoja i održavanja. Nalazi su u celini konzistentni: KMP zadržava nativno korisničko iskustvo uz smanjenje dupliranog koda, pri čemu su performansni kompromisi najvidljiviji na iOS-u u domenu memorije i inicijalizacije zajedničkog modula, dok su u realnim use-case scenarijima razlike najčešće marginalne.

Industrijska praksa potvrđuje akademske uvide, ali naglašava evolutivnu adopciju. Umesto "big-bang" migracija, timovi prvo izdvajaju jasno određene domene (npr. networking, domain/model sloj, validacije, formatiranje) u zajednički KMP modul, a zatim postepeno šire granicu deljenog koda. U tom procesu ključnu ulogu imaju biblioteke koje ublažavaju platformske razlike (npr. za HTTP/serialization, lokalnu bazu, i18n), kao i strateška odluka da UI ostane nativan. Praksa pokazuje da se najveće dobitke vidi u konsolidaciji poslovne logike: jednom implementirana pravila, transformacije i integracije sa backend-om postaju izvor istine za obe platforme, što smanjuje regresije i "drift" između Android i iOS verzije.

Značajan deo literature bavi se ograničenjima i uslovima pod kojima KMP isporučuje najviše vrednosti. Izdvajaju se: (a) zrelost toolchain-a i stabilnost interop-a sa platformskim bibliotekama; (b) kompleksnost build-a i CI/CD-a (posebno kada se uvode multiplatformske meta-konfiguracije, varijante i targeti); (c) upravljanje memorijom na iOS-u i efekti na kratke "cold start" tokove. Novije studije beleže poboljšanja u runtime-u i alatima, ali naglašavaju da performanse nisu jedina metrika odluke: ukupan trošak i brzina razvoja novih funkcionalnosti često nadvladaju male performanske razlike, posebno u aplikacijama koje nisu grafički ekstremno zahtevne.

C. Performanse kros-platformskih rešenja

Performanse su jedan od najčešćih kriterijuma na osnovu kojih se vrednuju kros-platformska rešenja. Akademska i industrijska istraživanja naglašavaju da je upravo ovaj aspekt bio razlog neuspeha mnogih ranijih frejmvorka. Web-hibridni pristupi poput PhoneGap-a ili Cordova-e nisu mogli da obezbede zadovoljavajuću brzinu izvršavanja i odziva aplikacija. Xamarin je postigao određeni napredak, ali su i dalje postojala ograničenja u domenu optimizacije i integracije sa nativnim API-jima.

Kod modernijih tehnologija, poput React Native-a i Flutter-a, performanse su znatno bolje zahvaljujući upotrebi nativnih komponenti (React Native) ili posebnog rendering engine-a (Flutter). Ipak, obe tehnologije nose određene kompromise. React Native često pokazuje probleme u kompleksnim

aplikacijama koje zahtevaju intenzivnu grafiku, dok Flutter aplikacije mogu imati veću veličinu i zahtevniji memory footprint.

U poređenju sa njima, Kotlin Multiplatform nudi poseban pristup. Budući da ne pokušava da zameni UI sloj već se fokusira na deljenje poslovne logike, performanse korisničkog interfejsa ostaju potpuno nativne. To znači da aplikacije izrađene u KMP-u mogu iskoristiti maksimalne mogućnosti platforme kada je reč o renderovanju i korisničkom iskustvu. Glavna razlika u performansama uočava se u oblasti inicijalizacije zajedničkog modula i upravljanja memorijom na iOS platformi, gde se beleži blago povećano vreme pokretanja i potrošnja RAM-a.

Brojna merenja u literaturi pokazuju da su razlike u performansama između KMP-a i nativnog pristupa uglavnom zanemarljive za krajnjeg korisnika. Startup time može biti duži za nekoliko stotina milisekundi, dok je u regularnom radu razlika često manja od 5%. Potrošnja memorije je veća, ali najčešće ne prelazi prag koji bi ozbiljnije uticao na iskustvo. Zaključak većine istraživanja je da je KMP dovoljno performantno rešenje za većinu poslovnih aplikacija, dok bi kod aplikacija sa ekstremnim zahtevima (npr. 3D igre ili aplikacije za obradu videa) nativni razvoj i dalje bio neophodan.

D. Produktivnost i održavanje

Pored performansi, jedan od ključnih faktora za izbor razvojne tehnologije jeste produktivnost tima i jednostavnost održavanja aplikacija na duži rok. Dok nativni pristup omogućava punu kontrolu nad svakom platformom, on istovremeno zahteva dupliranje velikog dela posla, što povećava troškove i produžava rokove isporuke.

Kotlin Multiplatform direktno adresira ovaj problem. Deljenjem poslovne logike i modela podataka između Android i iOS aplikacija, količina koda koji se mora pisati i održavati dvaput značajno se smanjuje. Istraživanja pokazuju da se u tipičnim aplikacijama može deliti od 50% do 80% koda, u zavisnosti od složenosti korisničkog interfejsa i integracija. Ova ponovna upotreba koda dovodi do kraćih ciklusa razvoja, manjeg rizika od grešaka i bržeg uvođenja novih funkcionalnosti.

U industrijskoj praksi zabeleženi su brojni slučajevi gde je prelazak na KMP omogućio timovima da sinhronizuju razvojne tokove i izbegnu situacije u kojima jedna platforma kasni za drugom. Posebno je značajno što se bugfix-evi i izmene poslovnih pravila implementiraju na jednom mestu i automatski prenose na obe platforme. Time se smanjuje verovatnoća nedoslednosti između verzija i poboljšava ukupni kvalitet proizvoda.

Naravno, postoji i određeni trošak obuke i prilagođavanja. Timovi koji nisu upoznati sa multiplatformskim alatima moraju uložiti dodatni trud da savladaju specifičnosti build sistema, interoperabilnost i konfiguracije. Takođe, alati i okruženje još uvek nisu na nivou stabilnosti i udobnosti kakvu nude isključivo nativni SDK-ovi. Ipak, većina istraživanja zaključuje da su dobici u produktivnosti i održavanju dugoročno veći od inicijalnih izazova, posebno za projekte sa bogatom i kompleksnom poslovnom logikom.

E. Arhitektonska razmatranja

Jedna od ključnih tema u vezi sa Kotlin Multiplatform tehnologijom odnosi se na arhitektonsku organizaciju koda i način na koji se poslovna logika razdvaja od korisničkog interfejsa. Za razliku od okvira poput Flutter-a, koji renderuje UI kroz sopstveni engine, KMP se fokusira na ponovnu upotrebu logike, dok se korisnički interfejs i dalje razvija nativno na Android-u i iOS-u.

Osnovni arhitektonski princip KMP-a jeste upotreba deljenog modula, koji sadrži poslovnu logiku, modele podataka i komunikaciju sa spoljnim servisima. Ovaj modul se kompajlira i distribuira ka Android i iOS projektima, pri čemu svaka platforma zadržava sopstveni sloj za prezentaciju i interakciju sa korisnikom. Time se postiže jasan razdvojeni sloj odgovornosti: zajednički kod rešava *šta* aplikacija radi, dok nativni deo određuje *kako* se to prikazuje korisniku.

KMP se dobro uklapa u poznate arhitektonske obrasce kao što su MVVM (Model-View-ViewModel) i MVI (Model-View-Intent). Na primer, poslovna logika i ViewModel sloj mogu se implementirati u zajedničkom modulu, dok se View sloj razvija nativno za svaku platformu. Ovakav pristup omogućava da ključne funkcionalnosti (validacija podataka, transformacije, upravljanje stanjima) budu jedinstvene i konzistentne, dok UI ostaje u skladu sa pravilima i standardima svake platforme.

Prednost ovakvog razdvajanja je što timovi dobijaju modularnu i skalabilnu arhitekturu, pogodnu za velike projekte. Svaka promena u poslovnoj logici automatski se reflektuje na obe platforme, što smanjuje rizik od nedoslednosti. Takođe, testiranje postaje jednostavnije jer se većina testova može izvršavati nad zajedničkim modulom.

S druge strane, postoje i izazovi. Na primer, potrebno je rešiti interop sloj između zajedničkog koda i nativnih API-ja, što može zahtevati dodatna prilagođavanja. Takođe, build sistem i CI/CD konfiguracija postaju kompleksniji jer je potrebno obezbediti pravilnu sinhronizaciju između multiplatformskog i nativnih projekata.

U nativnim aplikacijama poslovna logika i UI često su implementirani unutar istog koda baze, što može ubrzati razvoj manjih projekata, ali otežava održavanje kada aplikacija raste. KMP ovde uvodi čvršće razgraničenje slojeva i podstiče bolju organizaciju koda. U poređenju sa drugim kros-platformskim rešenjima, najveća prednost KMP arhitekture jeste što ne kompromituje nativni izgled i osećaj aplikacije, već ga zadržava kao deo ukupnog dizajna sistema.

F. Zaključak pregleda literature

Pregled literature pokazuje da se razvoj kros-platformskih rešenja kretao od jednostavnih hibridnih tehnologija, zasnovanih na web komponentama, ka sofisticiranim okvirima koji nastoje da objedine prednosti nativnog i zajedničkog koda. Rani pokušaji, poput PhoneGap-a i Xamarina, ponudili su uštede u vremenu i resursima, ali su često žrtvovali performanse i korisničko iskustvo. Modernija rešenja, kao što su React Native i Flutter, značajno su unapredila konzistentnost i odzivnost aplikacija, iako i dalje nose određene kompromise.

Kotlin Multiplatform se izdvaja drugačijim pristupom. Umesto da pokušava da kreira univerzalni UI sloj, KMP se fokusira na deljenje poslovne logike i modela podataka, ostavljajući prostor nativnim tehnologijama da oblikuju korisnički interfejs. Literatura ukazuje da ovaj pristup omogućava značajne uštede u održavanju, skraćivanje ciklusa razvoja i očuvanje nativnog korisničkog iskustva.

Istovremeno, istraživanja beleže određena ograničenja: povećanu potrošnju memorije na iOS-u, složeniji build sistem i nedovoljnu zrelost pojedinih biblioteka. Ipak, većina autora saglasna je da su ovi izazovi privremeni i rešivi, te da prednosti u produktivnosti i arhitektonskoj jasnoći čine KMP vrednim izborom, posebno u projektima sa složenom poslovnom logikom i potrebom za paralelnim razvojem na više platformi.

Zaključno, literatura postavlja osnovu na kojoj se ovaj rad nadovezuje: kvantitativnom analizom performansi i kvalitativnim razmatranjem arhitektonskih aspekata KMP-a, u poređenju sa nativnim razvojem. Time se doprinosi boljem razumevanju praktičnih mogućnosti i granica primene ove tehnologije.

III. METODOLOGIJA

A. Razvojno okruženje

Za potrebe istraživanja korišćen je niz razvojnih okruženja, kako bi se obezbedila što realističnija simulacija uslova u kojima se tipično razvijaju

mobilne aplikacije. Cilj je bio da se obuhvate svi relevantni segmenti procesa - od pisanja i organizacije koda, preko izgradnje aplikacija, pa do testiranja i merenja performansi.

Razvoj KMP modula realizovan je u Android Studio okruženju, koje obezbeđuje punu podršku za Kotlin Multiplatform projekte. Za nativnu Android aplikaciju korišćen je Android Studio, dok je nativna iOS aplikacija razvijana u Xcode-u. Izbor ovih alata omogućio je maksimalnu usklađenost sa industrijskim standardima i realnim okruženjem u kojem rade profesionalni razvojni timovi.

Svi projekti su građeni korišćenjem Gradle sistema (za Android i KMP deo) i Xcode build sistema (za iOS deo). Verzije Gradle plugina, Kotlin kompajlera i Xcode okruženja precizno su dokumentovane, jer su se pokazale kao značajan faktor koji može uticati na performanse izgradnje i kompatibilnost biblioteka.

Za implementaciju poslovne logike i zajedničkog modula korišćen je Kotlin, dok je Android deo takođe razvijan u Kotlinu, a iOS aplikacija u Swiftu.

U zajedničkom KMP modulu korišćene su biblioteke:

- Ktor za mrežnu komunikaciju i rad sa REST API-jem,
- Room za upravljanje lokalnom bazom podataka,
- Koin za dependency injection,
- Moko-resources za internacionalizaciju i deljenje resursa između platformi.

Ove biblioteke izabrane su jer obezbeđuju multiplatformsku podršku i omogućavaju da veći deo poslovne logike ostane unutar zajedničkog modula.

Verzije alata i konfiguracija:

- Kotlin: 2.1.20
- Swift: 5
- Gradle: 8.7.2

Na ovaj način obezbeđena je metodološka konzistentnost i mogućnost poređenja rezultata između različitih platformi.

B. Specifikacija uređaja i softverskog okruženja

Sva testiranja, kako nativnih tako i Kotlin Multiplatform rešenja, sprovedena su na istim fizičkim uređajima i pod identičnim softverskim uslovima. Na taj način obezbeđena je metodološka konzistentnost i izbegnuta mogućnost da hardverske ili verzijske razlike utiču na rezultate. Testovi su

izvođeni na realnim uređajima umesto emulatora, kako bi se dobili pouzdaniji podaci o performansama u stvarnom okruženju.

C. Arhitektura aplikacija

U istraživanju su razvijene tri aplikacije koje implementiraju potpuno istu funkcionalnost, ali različitim pristupima:

1. Nativna Android aplikacija
2. Nativna iOS aplikacija
3. Kotlin Multiplatform (KMP) aplikacija sa nativnim korisničkim interfejsima.

KMP rešenje zasniva se na principu razdvajanja zajedničke logike od platformskog koda. Zajednički modul obuhvata poslovnu logiku, modele podataka, komunikaciju sa BLE uređajem, pristup lokalnoj bazi i mrežnu komunikaciju. Platformski slojevi (Android/iOS) služe isključivo za implementaciju korisničkog interfejsa i povezivanje sa specifičnim API-jima. Na Android strani UI je razvijen u Jetpack Compose-u, dok je na iOS strani korišćen SwiftUI. Ovim pristupom postignuto je da se većina logike deli, a da se UI i dalje ponaša u skladu sa standardima svake platforme.

Za potrebe poređenja razvijene su i dve nativne aplikacije - jedna za Android i jedna za iOS. U njima je poslovna logika implementirana odvojeno, korišćenjem standardnih alata i biblioteka (Android Bluetooth API i Room za Android, CoreBluetooth i CoreData za iOS). Time je obezbeđena maksimalna kontrola nad platformama, ali i dvostruki razvojni napor, jer je isti skup funkcionalnosti morao biti implementiran dva puta.

D. Funkcionalnosti aplikacija

Sve tri aplikacije - nativna Android, nativna iOS i Kotlin Multiplatform verzija - razvijene su tako da imaju identičan skup funkcionalnosti. Na ovaj način obezbeđena je potpuna uporedivost rešenja i uklonjeni su faktori koji bi mogli narušiti objektivnost merenja.

1) Pretraga i povezivanje sa BLE uređajem

Aplikacije omogućavaju otkrivanje i povezivanje sa BLE (Bluetooth Low Energy) senzorima. Proces skeniranja implementiran je kroz odgovarajuće nativne API-je, dok je u KMP rešenju logika skeniranja i obrade rezultata bila deo zajedničkog modula.

2) Prikupljanje i obrada podataka

Nakon uspostavljanja veze sa senzorom, aplikacije prikupljaju podatke o zdravstvenim parametrima (npr. puls, saturacija kiseonika). Poslovna logika obezbeđuje validaciju podataka, njihovu interpretaciju i pripremu za dalje skladištenje.

3) Skladištenje podataka u lokalnoj bazi

Prikupljeni podaci se čuvaju u lokalnoj bazi. U nativnim aplikacijama korišćeni su standardni mehanizmi (Room za Android, CoreData za iOS), dok je u KMP rešenju korišćen Room za KMP, čime je obezbeđeno da kod za upravljanje podacima bude zajednički za obe platforme.

4) Dobavljanje podataka sa backend servera

Korišćenjem Ktor biblioteke, KMP verzija je omogućila da logika dobavljanja podataka bude zajednička, dok su nativne aplikacije imale odvojene UI implementacije.

5) Pregled istorije i vizualizacija podataka

Korisnici imaju mogućnost da pregledaju istoriju prikupljenih podataka. Prikazani sadržaj i format podataka oslanjaju na zajedničku logiku.

E. Definisanje metrika performansi

Da bi se uporedno vrednovala nativna i Kotlin Multiplatform rešenja, definisan je set jasno merljivih performansnih metrika. Svaka metrika izabrana je tako da obuhvati ključne aspekte kvaliteta aplikacija: brzinu, efikasnost i stabilnost.

1) Vreme pokretanja (Startup time)

Mereno je koliko vremena je potrebno od trenutka pokretanja aplikacije do prikaza prvog ekrana spremnog za interakciju. Analizirane su dve situacije:

- Cold start - aplikacija se pokreće prvi put nakon instalacije ili nakon što je potpuno uklonjena iz memorije
- Warm start - aplikacija se ponovo pokreće dok su podaci već delimično keširani u memoriji

2) Veličina aplikacije

Upoređivana je veličina instalacionih paketa (APK za Android i IPA za iOS). Ova metrika ukazuje na uticaj multiplatformskih biblioteka i runtime okruženja na finalni proizvod.

3) Potrošnja memorije (RAM usage)

Praćena je prosečna i maksimalna potrošnja RAM-a u scenarijima tipičnog korišćenja: pokretanje aplikacije, skeniranje BLE uređaja, prikupljanje i prikaz podataka. Ova metrika je ključna za procenu efikasnosti aplikacije, posebno na uređajima sa ograničenim resursima.

4) Vreme izgradnje (Build time)

Mereno je vreme potrebno da se aplikacija kompletno izgradi iz izvornog koda. Ova metrika je relevantna za procenu produktivnosti tima i brzine iteracija tokom razvoja.

5) Broj fajlova u projektu

Brojanjem fajlova u projektima dobijen je uvid u složenost i preglednost koda. KMP rešenje trebalo bi da ima manje fajlova zbog centralizovane logike, dok se kod nativnog razvoja logika duplira na obe platforme.

F. Alati i procedure za merenje

Kako bi se obezbedila preciznost i uporedivost rezultata, korišćen je kombinovani pristup u kojem su uključeni standardni razvojni alati i prilagođene procedure za testiranje.

Alati za merenje:

1. Android Profiler**:** korišćen je za merenje potrošnje memorije i CPU aktivnosti. Omogućio je detaljnu vizualizaciju resursnih zahteva tokom različitih scenarija korišćenja.
2. Xcode Instruments**:** upotrebljen je na iOS platformi za analizu performansi. Korišćeni su moduli za praćenje potrošnje memorije i CPU aktivnosti.
3. Sistemski logovi**:** Logovi su korišćeni na obe platforme za merenje vremena izvršavanja inicijalizacije aplikacije ili pri izvršavanju specifičnih operacija.

Procedura merenja

1. Ponavljanje testova - svaki scenario meren je najmanje 10 puta, kako bi se smanjio uticaj slučajnih oscilacija u radu uređaja.
2. Izračunavanje prosečnih vrednosti - rezultati su prikazani kroz prosečne vrednosti, a u slučajevima velikih odstupanja razmatran je i median.

3. Eliminacija outliera - ekstremne vrednosti koje značajno odstupaju od ostalih rezultata analizirane su i, ukoliko su posledica očiglednog spoljnog faktora (npr. pozadinska notifikacija), izostavljene iz konačnih izveštaja.
4. Kontrolisano okruženje - tokom testiranja uređaji su korišćeni isključivo za potrebe merenja; sve pozadinske aplikacije i servisi koji nisu bili deo eksperimenta bili su isključeni.
5. Dokumentovanje uslova - zabeleženi su svi parametri testnog okruženja (verzija alata, OS), kako bi se obezbedila reproduktivnost rezultata.

Na ovaj način obezbeđena je visoka validnost merenja i minimiziran rizik da eksterni faktori utiču na dobijene nalaze.

IV. REZULTATI I ANALIZA

A. Vreme izgradnje (Build time)

Vreme izgradnje predstavlja jedan od ključnih pokazatelja produktivnosti tokom razvoja mobilnih aplikacija. U ovom istraživanju analizirano je trajanje procesa build-a za nativna rešenja i za Kotlin Multiplatform (KMP), na Android i iOS platformi.

Rezultati su pokazali da KMP značajno produžava hladnu izgradnju (cold build) u odnosu na nativno rešenje. Konkretno, prosečno vreme cold build-a kod KMP aplikacije bilo je oko 78,78 sekundi, dok je kod nativne aplikacije iznosilo 39,96 sekundi. Dakle, izgradnja KMP projekta zahtevala je približno duplo više vremena.

Kod inkrementalnih izgradnji (incremental build), gde se kompajliraju samo izmene u kodu, razlika je bila znatno manja. KMP je i dalje imao nešto duže vreme u poređenju sa nativnim razvojem, ali ta razlika je bila u granicama koje ne utiču značajno na svakodnevni rad.

Na iOS platformi rezultati su pokazali sličan trend. KMP projekat je u proseku imao duže vreme cold build-a, dok su razlike kod incremental build-a bile znatno manje izražene. Iako je build proces u Xcode-u sam po sebi optimizovan za inkrementalne promene, dodavanje multiplatformskog sloja uvodi dodatnu složenost i produžava ukupno trajanje kompilacije.

Dobijeni rezultati potvrđuju da KMP uvodi određeni overhead u procesu build-a, naročito kod cold build-a, gde se vreme izgradnje gotovo udvostručuje u odnosu na nativni razvoj. Ipak, u svakodnevnoj praksi programeri se mnogo češće oslanjaju na inkrementalne build-ove, gde razlike nisu kritične. Zbog

toga se može zaključiti da, iako KMP donosi sporije inicijalne build-ove, to u praksi nije presudna prepreka za njegovu upotrebu, posebno u timovima gde su dobici u ponovnoj upotrebi koda važniji od samog vremena kompilacije.

B. Veličina aplikacije

Veličina aplikacije je važna metrika jer direktno utiče na vreme preuzimanja i instalacije, kao i na količinu memorije koju zauzima na uređaju krajnjeg korisnika. U okviru istraživanja izvršeno je poređenje KMP i nativnih aplikacija na Android i iOS platformama.

Rezultati pokazuju da je KMP Android aplikacija znatno veća od nativne.

- Veličina APK fajla za KMP aplikaciju iznosila je 12,8 MB, dok je nativna aplikacija bila svega 3,2 MB.
- Kada se aplikacija instalira na uređaj, razlika je i dalje primetna: 13,22 MB kod KMP rešenja naspram 7,42 MB kod nativne aplikacije.

Na iOS strani, razlike su još izraženije:

- Veličina IPA fajla za KMP verziju bila je 18,8 MB, dok je nativna aplikacija imala samo 1,6 MB.
- Na samom telefonu KMP aplikacija zauzima 18,9 MB, dok nativna zauzima svega 1,7 MB.

Rezultati jasno ukazuju na prednost nativnih aplikacija u pogledu manje veličine paketa i zauzeća memorije. Veća veličina KMP aplikacija rezultat je dodatnih biblioteka i runtime okruženja potrebnih za multiplatformsku podršku. Iako to ne predstavlja veliki problem na modernim uređajima sa značajnim memorijskim kapacitetom, može biti faktor u situacijama kada su resursi ograničeni ili kada korisnici preuzimaju aplikacije preko sporijih internet konekcija.

C. Vreme pokretanja aplikacije

Vreme pokretanja aplikacije i brzina odziva direktno utiču na kvalitet korisničkog iskustva. U ovom istraživanju mereno je vreme do prvog prikaza sadržaja (Time to First Frame) i vreme do potpune upotrebljivosti aplikacije (Time to Usable State).

Android platforma:

- Time to First Frame: KMP aplikacija je imala prosečno 676 ms, dok je nativna aplikacija postigla 382 ms.
- Time to Usable State: Oba rešenja pokazala su identičan rezultat od 1,02 sekunde.

Ovi rezultati ukazuju da KMP aplikacija ima nešto sporiji inicijalni prikaz sadržaja na Android-u, ali kada je reč o spremnosti za interakciju, razlike praktično ne postoje.

iOS platforma:

- Time to First Frame: KMP aplikacija je postigla 103 ms, dok je nativna bila vrlo blizu sa 96 ms.
- Time to Usable State: KMP verzija imala je 1,019 s, dok je nativna imala 1,084 s.

Na iOS-u razlike su minimalne i pokazuju da KMP može da pruži iskustvo gotovo identično nativnom rešenju.

Na osnovu rezultata može se zaključiti da KMP ima blagi zaostatak na Android platformi u pogledu inicijalnog prikaza sadržaja, dok su razlike u pogledu pune upotrebljivosti zanemarljive. Na iOS platformi razlike su gotovo neprimetne, što potvrđuje da je KMP dovoljno efikasno rešenje za aplikacije koje zahtevaju brzo pokretanje i odziv.

D. Performanse izvršavanja deljenog koda

Ovaj deo analize fokusira se na upoređivanje performansi zajedničkog koda implementiranog u KMP rešenju sa odgovarajućim nativnim implementacijama. Posebna pažnja posvećena je Bluetooth komunikaciji i skladištenju podataka u lokalnoj memoriji, jer su ove operacije radjenje u deljenom kodu uz pristup nativnim bibliotekama.

Android platforma:

- Bluetooth - vreme do skeniranja uređaja: KMP aplikacija prosečno je beležila 749 ms, dok je nativna verzija imala značajno brže vreme od 294 ms.
- Bluetooth - vreme za prikupljanje podataka: KMP je ostvario 1,588 s, dok je nativno rešenje postiglo nešto bolje vreme od 1,242 s.
- Skladištenje podataka u lokalnu memoriju: Zanimljivo je da je KMP aplikacija imala blagu prednost (46 ms) u odnosu na nativnu aplikaciju (52 ms) .

iOS platforma:

- Bluetooth - vreme do skeniranja uređaja: KMP aplikacija imala je 404 ms, dok je nativna bila brža sa 290 ms.

- Bluetooth - vreme za prikupljanje podataka: rezultati su ovde bili vrlo bliski - KMP je imao 1,812 s, dok je nativna aplikacija postigla 1,897 s.
- Skladištenje podataka u lokalnu memoriju: KMP je bio sporiji (29 ms) u poređenju sa nativnom verzijom (12 ms) .

Rezultati pokazuju da KMP rešenja imaju manje kašnjenje u odnosu na nativna kada je reč o inicijalnom Bluetooth skeniranju i prikupljanju podataka, posebno na Android platformi. Na iOS strani razlike su manje izražene, a u pojedinim slučajevima KMP pokazuje čak i uporediva ili bolja vremena od nativnog. Kod skladištenja podataka, situacija je mešovita - na Android-u KMP ima blagu prednost, dok je na iOS-u sporiji.

Opšti zaključak je da performanse zajedničkog koda u KMP-u ostaju u prihvatljivim granicama, pri čemu su najveće razlike primećene kod Bluetooth operacija na Android platformi. Međutim, i te razlike su takve da ne narušavaju upotrebljivost aplikacije u realnim scenarijima.

E. Potrošnja memorije tokom rada aplikacije

Potrošnja memorije predstavlja ključnu metriku jer direktno utiče na stabilnost aplikacije, posebno na uređajima sa ograničenim resursima. U ovom istraživanju upoređena je potrošnja memorije između KMP i nativnih aplikacija na obe platforme.

Android platforma:

Rezultati su pokazali da KMP aplikacije koriste više memorije u svim scenarijima:

- Idle stanje: KMP aplikacija prosečno je koristila 225 MB, dok je nativna koristila 175 MB.
- Peak usage (tokom intenzivnog rada sa BLE uređajem): KMP je dosegao 385 MB, dok je nativna aplikacija imala 350 MB.
- Nakon peak opterećenja: KMP je zadržao 340 MB, dok se nativna aplikacija vratila na 240 MB.

Ovi rezultati pokazuju da KMP aplikacija ima tendenciju da zadrži više zauzete memorije i nakon intenzivnog rada, što može biti značajan faktor kod uređaja sa manjom količinom RAM-a.

iOS platforma:

Na iOS-u su razlike manje izražene, ali i dalje u korist nativnog rešenja:

- Idle stanje: KMP aplikacija koristila je 27 MB, dok je nativna koristila 16,5 MB.
- Peak usage: KMP je imao 48 MB, dok je nativna aplikacija zabeležila 44 MB.
- Nakon peak opterećenja: KMP se vratio na 47 MB, dok je nativna aplikacija imala 43 MB.

Generalno, KMP aplikacije koriste više memorije od nativnih, kako u mirovanju, tako i tokom vršne upotrebe. Najveće razlike primećene su na Android platformi, gde KMP pokazuje veće zadržavanje memorije nakon peak opterećenja. Na iOS-u su odstupanja manja, ali i dalje prisutna. Ovaj nalaz ukazuje da bi optimizacija upravljanja memorijom trebalo da bude prioritet za KMP projekte, naročito kada se ciljaju uređaji sa ograničenim hardverskim resursima.

F. Broj korišćenih fajlova

Jedan od pokazatelja složenosti projekta i lakoće njegovog održavanja jeste broj fajlova potrebnih za implementaciju funkcionalnosti. U ovom istraživanju izvršeno je poređenje između nativnog pristupa i Kotlin Multiplatform (KMP) rešenja.

Rezultati poređenja

- Nativni pristup:
  - Android aplikacija: 62 fajla
  - iOS aplikacija: 57 fajlova
  - Ukupno: 119 fajlova
- KMP pristup:
  - Shared modul: 68 fajlova
  - Android specifični deo: 28 fajlova
  - iOS specifični deo: 24 fajla
  - Ukupno: 120 fajlova

Na prvi pogled, rezultati pokazuju da je KMP projekat imao jedan fajl više nego kombinacija obe nativne aplikacije. Međutim, ovakav odnos je specifičan za ovaj eksperiment i ne odražava realnu prednost KMP-a u većim projektima. Kako se povećava obim deljene logike i broj funkcionalnosti, KMP omogućava da se ista logika implementira jednom u zajedničkom modulu, dok bi u nativnim rešenjima ona morala biti duplirana.

Dakle, iako u ovom primeru nema uštede, kod složenijih projekata očekuje se da će KMP značajno smanjiti ukupan broj fajlova i time olakšati održavanje, jer se izbegava dupliranje logike i smanjuje potreba za višestrukim izmenama istih funkcionalnosti.

Broj fajlova u ovom istraživanju nije pokazao prednost KMP-a, ali metodološki je važno istaći da u realnim, većim projektima prednosti dolaze do izražaja. Upravo ovakvi rezultati ukazuju da KMP donosi dugoročne benefite u održavanju i razvoju, čak i ako inicijalni broj fajlova ne pokazuje jasnu prednost.

V. DISKUSIJA

Dobijeni rezultati pokazuju da Kotlin Multiplatform može da obezbedi gotovo identične performanse kao nativna rešenja u većini metrika. Startup vreme i odziv aplikacija na iOS platformi praktično su jednaki, dok na Androidu postoje blaga odstupanja u korist nativnih aplikacija.

Najveće prednosti KMP-a ogledaju se u ponovnoj upotrebi poslovne logike i smanjenju dupliranog koda. To dugoročno donosi značajnu uštedu u održavanju i brži razvoj novih funkcionalnosti. Arhitektura zasnovana na zajedničkom modulu takođe doprinosi boljoj konzistentnosti i lakšem testiranju.

Rezultati su pokazali da KMP uvodi određeni overhead u potrošnji memorije i vreme izgradnje, naročito kod cold build-a i na Android platformi. Takođe, veća veličina instalacionih paketa može biti ograničavajući faktor u specifičnim slučajevima.

Za aplikacije koje imaju značajan deo zajedničke logike i ne zavise od ekstremnih performansi (npr. kompleksne grafike ili real-time aplikacija), KMP predstavlja održivo i isplativo rešenje. S druge strane, projekti sa vrlo specifičnim zahtevima performansi i optimizacije mogu i dalje imati prednost u potpunom nativnom razvoju.

Na osnovu sprovedenog istraživanja, može se preporučiti KMP kao pogodno rešenje za većinu poslovnih aplikacija, uz oprez kada su u pitanju zahtevne grafičke operacije ili striktna ograničenja resursa. Dalja unapređenja alatnog lanca i biblioteka verovatno će dodatno smanjiti postojeća ograničenja.

VI. ZAKLJUČAK

Istraživanje je pokazalo da Kotlin Multiplatform predstavlja zrelo i fleksibilno rešenje za razvoj mobilnih aplikacija koje ciljaju više platformi. Iako postoje izazovi u pogledu performansi, posebno na iOS-u, prednosti u vidu smanjenja dupliranog koda, ubrzanja razvoja i lakšeg održavanja čine ga

atraktivnim izborom za mnoge projekte. Budući rad treba usmeriti ka optimizaciji alata, poboljšanju kompatibilnosti sa bibliotekama i unapređenju procesa izgradnje.

LITERATURA

[1] S. Scantz, "Performance comparison of Kotlin Multiplatform and native mobile apps," Journal of Mobile Computing, 2023.

[2] M. Gušpiel, "Kotlin Multiplatform in patient tracking systems," Software Engineering Review, 2024.

[3] A. Evert, "Evaluating cross-platform mobile frameworks," Proceedings of Mobile DevConf, 2019.

[4] D. Punja et al., "Productivity benefits of Kotlin Multiplatform," International Journal of Software Development, 2024.

[5] B. Stanić, M. Ćirković, "Architectural aspects of Kotlin Multiplatform," IT Horizons, 2024.

ABSTRACT

This paper presents research on the use of Kotlin Multiplatform technology for developing mobile applications targeting both Android and iOS platforms. The study includes performance measurements, architectural analysis, and comparative evaluation with native approaches to assess the cost-effectiveness and sustainability of this solution. Three functionally identical applications were developed: one using Kotlin Multiplatform, one natively for Android, and one natively for iOS. The findings show that while Kotlin Multiplatform may have slight drawbacks in areas such as memory usage, it offers significant benefits in development speed, maintainability, and code reusability, making it a compelling choice for projects with substantial cross-platform logic sharing.

Research on the Application of Kotlin Multiplatform Technology: Performance and Architectural Analysis Compared to Native Development

Danijal Azerović, Bojana Dimić Surla