Izrada mobilnih aplikacija korištenjem Flutter/Dart razvojnog okvira na primjeru aplikacije MovieLens

Kapulica, Mislav

SVEUČILIŠTE VERN'

Zagreb

Poslovna informatika

ZAVRŠNI RAD

Izrada mobilnih aplikacija korištenjem Flutter/Dart razvojnog okvira na primjeru aplikacije MovieLens

Mislav Kapulica

Zagreb, 2022.

SVEUČILIŠTE VERN'

Preddiplomski stručni studij

Poslovna informatika

ZAVRŠNI RAD

Izrada mobilnih aplikacija korištenjem Flutter/Dart razvojnog okvira na primjeru aplikacije MovieLens

Mentor: Dean Nižetić, struč. spec. inf. v. pred.

Student: Mislav Kapulica

Zagreb, rujan 2022.

SVEUČILIŠTE VERN'
Zagreb, Palmotićeva ulica 82/1
Poslovna informatika

Broj: 4325

ZADATAK ZAVRŠNOG RADA

Student/ica: Mislav Kapulica

Zadatak: Izrada mobilnih aplikacija korištenjem Flutter/Dart razvojnog okvira na primjeru aplikacije MovieLens

U radu je potrebno razraditi sljedeće:

- Razvojni okviri za izradu mobilnih aplikacija. Donijeti pregled postojećih okvira.
- Prednosti i mane korištenja *Fluttera* i *Darta* u odnosu na pristup izradi mobilnih bez korištenja okvira za razvoj aplikacija. Napraviti osvrt na budućnost korištenja razvojnih okvira i posebice *Fluttera*.
- Izraditi funkcionalnu specifikaciju za mobilnu aplikaciju *MovieLens.*
- Opisati proces izrade aplikacije korištenjem razvojnog okvira *Flutter* i *Dart*.
- Na osnovu naučenog u procesu izrade donijeti zaključke i preporuke za usvajanje i korištenje razvojnog okvira.

Napomena: *Pri izradi završnog rada kandidat ima obvezu pridržavati se i uvažavati primjedbe, sugestije i naputke mentora, koristiti i primjenjivati znanja i umijeća stečena tijekom studija, upotrebljavati informacije i podatke prikupljene vlastitim istraživanjem te spoznaje i činjenice iz odgovarajuće znanstvene i stručne literature uz ispravno navođenje korištenih izvora.*

Zadatak zadan: 9. 4. 2021.

Rok predaje: 7. 9. 2022.

Mentor
Dean Nižetić, struč. spec. inf, v. pred.

REPUBLIKA HRVATSKA 2 ZAGREB SVEUČILIŠTE VERN

MP

Pročelnica studija
Jadranka Musulin, dipl. oec., v. pred.

SADRŽAJ

SAŽETAK

Razvoj mobilnih aplikacija je nova paradigma razvoja softvera. Iako se udio mobilnih platformi na tržištu ustalio, s dvije glavne platforme, iOS-om i Androidom, i dalje postoji širok izbor razvojnih alata za te platforme. Jedna od glavnih prepreka u razvoju nativnih mobilnih aplikacija za više platformi je činjenica da je potrebno razviti jednu aplikaciju za svaku platformu koju se želi podržati. *Cross-platform* razvojni okviri su se u ovom slučaju predstavili kao rješenje, nudeći razvoj jedne aplikacije za više različitih platformi. Prednosti takvih rješenja su često umanjene slabijim performansama, problematičnim i nekonzistentnim komponentama grafičkog sučelja i lošijom podrškom. Flutter, kao jedan od najnovijih *cross-platform* razvojnih okvira, nastoji ukloniti većinu navedenih problema. U ovom radu analiziraju se metode razvoja mobilnih aplikacija, i kroz proces izrade mobilne aplikacije razvojnim okvirom Flutter, ispituju tvrdnje o njegovim prednostima.

Ključne riječi: mobilne aplikacije, razvojni okviri, *cross-platform*, *Flutter*

ABSTRACT

Developing mobile applications using Flutter/Dart framework on the example of the MovieLens application

Mobile application development is a new paradigm of software development. Although the market share of mobile platforms has stabilized, with the two main platforms, iOS and Android, there is still a wide variety of development tools for these platforms. One of the main obstacles in developing native mobile apps for multiple platforms is the fact that you need to develop one app for each platform you want to support. Cross-platform development frameworks presented themselves as a solution, offering the development of one application for several different platforms. The advantages of such solutions are often offset by lower performance, problematic and inconsistent graphical interface components, and worse support. Flutter, as one of the latest cross-platform development frameworks, tries to eliminate most of the listed problems. In this paper, the methods of mobile application development are analysed, and through the process of creating a mobile application using the Flutter development framework, claims about the advantages of Flutter are examined.

Keywords: Mobile Apps, Frameworks, Cross-platform, Flutter

1. UVOD

Mobilne aplikacije se prvi put pojavljuju 1984. na džepnom računalu Psion Organiser I[1]. U početku su to bile jednostavne aplikacije, poput sata i kalkulatora, zbog ograničenih mogućnosti prijenosne tehnologije tog doba (Pountain, 1984). 2002. godine, gotovo dva desetljeća kasnije, kanadska tvrtka Research in Motion predstavlja svoj prvi pametni telefon BlackBerry, koji nudi naprednije opcije poput slanja i primanja elektroničke pošte te posjećivanja mrežnih stranica (Leung, 2012). No, mobilne aplikacije koje danas poznajemo nisu se pojavile do 2008. godine, kada je Apple otvorio svoju trgovinu aplikacija App Store i time omogućio svim zainteresiranim stranama razvoj mobilnih aplikacija za njihov pametni telefon iPhone[2].

2008. godine Apple je za svoje pametne telefone iPhone, razvio iPhone SDK, set razvojnih alata za izradu mobilnih aplikacija, što je programerima omogućilo jednostavniji razvoj mobilnih aplikacija za tu specifičnu platformu[3]. Uz pojavu operativnog sustava Google Android i popratne trgovine aplikacija, došlo je do potrebe razvoja mobilnih aplikacija za više platformi u svrhu povećanja tržišnog dosega (Ornbo, 2019). Do tada su gotovo sve mobilne aplikacije bile nativne (eng. *native apps*), što znači da su se za njihov razvoj primjenjivali razvojni alati koji ciljaju samo jednu platformu, onu za koju su napravljeni. Posljedično, to je iziskivalo razvoj iste aplikacije više puta, da bi se ona mogla upotrebljavati na više platformi, čime su troškovi i vrijeme razvoja aplikacije bili puno veći.

Uzletom interneta dvijetisućitih godina i sveprisutnošću mrežnih preglednika, došla je ideja za razvoj mrežnih aplikacija koje se pokreću unutar mrežnih preglednika (Garret, 2005). Time se jednom razvijena aplikacija mogla rabiti na svim uređajima koji imaju mrežne preglednike i pristup internetu. Takve su aplikacije imale nekoliko nedostataka; bile su značajno sporije od nativnih rješenja, zahtijevale su dostupnost interneta i nisu

[1] A Brief History of Mobile Apps. Preuzeto s: https://www.captechu.edu/blog/brief-history-of-mobile-apps (5. 8. 2022.)

[2] iPhone App Store Downloads Top 10 Million in First Weekend. Apple. Preuzeto s: https://www.apple.com/newsroom/2008/07/14iPhone-App-Store-Downloads-Top-10-Million-in-First-Weekend/ (5. 8. 2022.)

[3] Apple Announces iPhone 2.0 Software Beta. Apple. Preuzeto s: https://www.apple.com/newsroom/2008/03/06Apple-Announces-iPhone-2-0-Software-Beta/ (5. 8. 2022.)

podržavale naprednije funkcionalnosti platformi na kojima su se upotrebljavale, jer su bile ograničene mogućnostima preglednika i tehnologija kojima su se koristile.

Krajem dvijetisućitih godina, kombinirane su ideje razvoja nativnih i mrežnih aplikacija u hibridni model razvoja mobilnih aplikacija (Wargo, 2012). Time se omogućio razvoj aplikacije za više platformi istovremeno, ali se, za razliku od mrežnih aplikacija, nisu upotrebljavale unutar mrežnog preglednika nego su bile zasebne aplikacije poput nativnih. Razvoj hibridnih aplikacija zahtijeva korištenje razvojnih okvira koji podržavaju više platformi. Takvi razvojni okviri apstrahiraju iste mogućnosti na različitim platformama, zbog čega se programeri ne moraju opterećivati specifičnošću implementacije iste značajke aplikacije na različitim uređajima (Wargo, 2012). U usporedbi s nativnim aplikacijama, hibridne su aplikacije najčešće bile osjetno sporije i nisu se mogle koristiti svim mogućnostima platforme na kojoj su se upotrebljavale, jer ovise o mogućnostima razvojnog okvira koji se primjenjivao za njihov razvoj (Windmill, 2020). No, ipak su bile brže i efikasnije od mrežnih aplikacija i mogle su iskoristiti daleko više mogućnosti platformi na kojima su se rabile.

Ta tri tipa aplikacija - nativni, mrežni i hibridni, i dalje su u primjeni. Uz toliko opcija za razvoj mobilnih aplikacija, prvo pitanje koje možemo postaviti jest kojom se od njih koristiti? Hibridne aplikacije u zadnje vrijeme privlače sve veću pozornost zbog svojih velikih prednosti i manje značajnih nedostataka. U ovom radu nastoji se prikazati trenutno stanje u području razvoja mobilnih aplikacija hibridnog tipa, i ponajprije usporediti razvojni okvir Flutter s ostalima i prikazati proces izrade mobilne aplikacije korištenjem razvojnog okvira Flutter. Također, na osnovi iznesenih činjenica prikupljenih u procesu istraživanja, za potrebe ovog rada i iskustva stečenog razvojem mobilne aplikacije MovieLens korištenjem Fluttera, iznijeti zaključke i preporuke za usvajanje i korištenje tog razvojnog okvira.

2. RAZVOJNI OKVIRI

Za razumijevanje razvojnih okvira za mobilne aplikacije potrebno je sagledati njihov povijesni kontekst, njihove mogućnosti i kako se oni međusobno uspoređuju. Također je potrebno uočiti što je zahtijevalo njihovu pojavu, odnosno koje probleme oni nastoje riješiti.

2.1. Povijest razvojnih okvira za mobilne aplikacije na više platforma

Nakon što je 2008. godine Apple predstavio svoju trgovinu aplikacija i nedugo nakon toga Google napravio isti potez, razvoj mobilnih aplikacija bio je u uzletu. U isto vrijeme počela su se pojavljivati inovativna rješenja koja su obećavala jednostavan i brz razvoj mobilnih aplikacija s podrškom za više platformi, tzv. *cross-platform* aplikacije. Jedno od prvih takvih rješenja bio je PhoneGap, predstavljen 2009. godine, a koji je prerastao u današnji Apache Cordova. PhoneGap se koristio mrežnim tehnologijama HTML5, CSS3 i JavaScript i omogućavao mrežnim *developerima* razvoj mobilnih aplikacija bez potrebe za učenjem novih tehnologija, koje bi inače bile potrebne za njihov razvoj (Wargo, 2012). Uz to, PhoneGap je omogućio razvoj aplikacije za više platformi istovremeno.

Uz PhoneGap pojavila su se i druga rješenja, kao što je RhoMobile, koji se koristio istim mrežnim tehnologijama i programskim jezikom za razvoj aplikacija Ruby, i godinu kasnije MoSync, koji se koristio C/C++ programskim jezicima i HTML5-om. Kasnijih godina predstavljeni su NSB/AppStudio, Sencha Touch, Kivy, Enyo, Xamarin i drugi. 2013. godine predstavljen je Ionic, 2014. NativeScript, 2015. godine Facebookov React Native, a 2018. Googleov razvojni okvir Flutter.

Iako je postojalo više razvojnih okvira za mobilne aplikacije, mali broj njih dosegao je dovoljnu popularnost da i dalje bude u aktivnom razvoju s podrškom za nove platforme i verzije operativnih sustava. Uslijed toga, danas su najčešće korišteni sljedeći razvojni okviri za mobilne aplikacije: Xamarin, Ionic, NativeScript, React Native i Flutter. Apache Cordova je do trenutka pisanja ovog rada i dalje bio korišten za razvoj aplikacija, ali i taj projekt počeo je gubiti podršku (Eom, 2021).

2.2. Obilježja razvojnih okvira

Razvojni okviri su programski alati za razvoj programa, tj. aplikacija i sličnih programskih rješenja. Oni čine skup gotovih funkcionalnosti koje se mogu implementirati u programskom rješenju, a čime se nastoji ubrzati i pojednostaviti njihov razvoj (Wolfgang, 1994). Razvojni okviri imaju dva glavna obilježja koja su ključna za donošenje odluke o njihovom korištenju prilikom razvoja programskih rješenja. To su tehnologije ili programski jezici kojima se koriste za razvoj programskih rješenja i platforme za koje omogućuju razvoj.

Razvojni okviri za mobilne aplikacije su oni koji podržavaju platforme mobilnih uređaja poput pametnih telefona i tableta. Uz mobilne uređaje, većina današnjih razvojnih okvira podržava i druge platforme, poput onih za stolna računala, pametne satove ili pak mrežnu platformu. Tablica 2.1. prikazuje trenutno najpopularnije razvojne okvire za mobilne aplikacije na više platformi, njihove tehnologije i ciljane platforme.

Tablica 2.1. Usporedba *cross-platform* razvojnih okvira

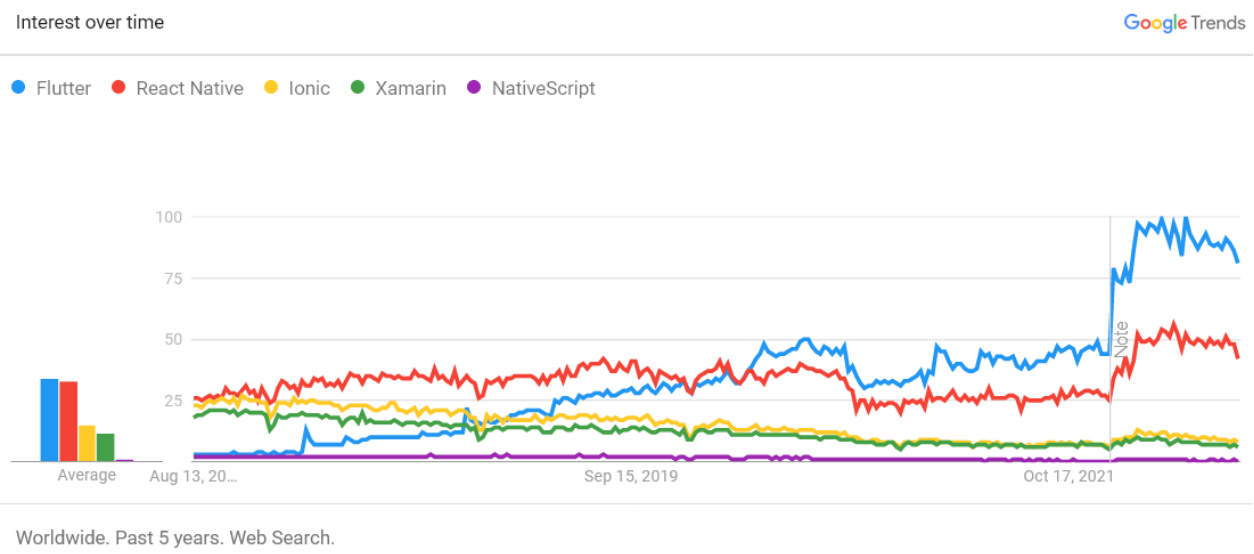
Razvojni okvir	**Tehnologije razvoja**	**Podržane platforme**
Xamarin	C#, XAML	iOS, Android, Windows, macOS
Ionic	JavaScript, CSS, HTML	iOS, Android, Windows, macOS, Linux, Web
NativeScript	JavaScript, CSS, XML	iOS, Android
React Native	JavaScript	iOS, Android, Windows, macOS, Linux, Web
Flutter	Dart	iOS, Android, Windows, macOS, Linux, Web

Izvor: autorovo djelo[4]

[4] Podatci tablice ispunjeni su prema informacijama sa službenih mrežnih stranica projekata navedenih razvojnih okvira.

Uz tehnologije razvoja i podržanih platformi, podobnost uporabe nekog razvojnog okvira može se odrediti i prema kvaliteti popratne dokumentacije, jednostavnosti korištenja razvojnog okvira, performansama gotovih aplikacija, veličini zajednice koja sudjeluje u razvoju te kvaliteti i mogućnostima gotovih predložaka grafičkog dizajna elemenata aplikacije koji su dio razvojnog okvira. Navedeni kriteriji su u većini slučajeva teško usporedivi, ali nude dodatan uvid u kvalitetu razvojnog okvira i bitan su čimbenik prilikom razmatranja odabira razvojnog okvira za razvoj programskog rješenja. Veličinu zajednice i popularnost razvojnog okvira možemo pokušati procijeniti pomoću broja pretraga na Google tražilici i provedenih anketa o korištenju različitih razvojnih okvira. Slika 2.1. prikazuje trendove u pretragama pet različitih *cross-platform* razvojnih okvira na Google tražilici kroz zadnjih pet godina[5], a slika 2.2. rezultate ankete programera o uporabi većine dostupnih *cross-platform* razvojnih okvira.

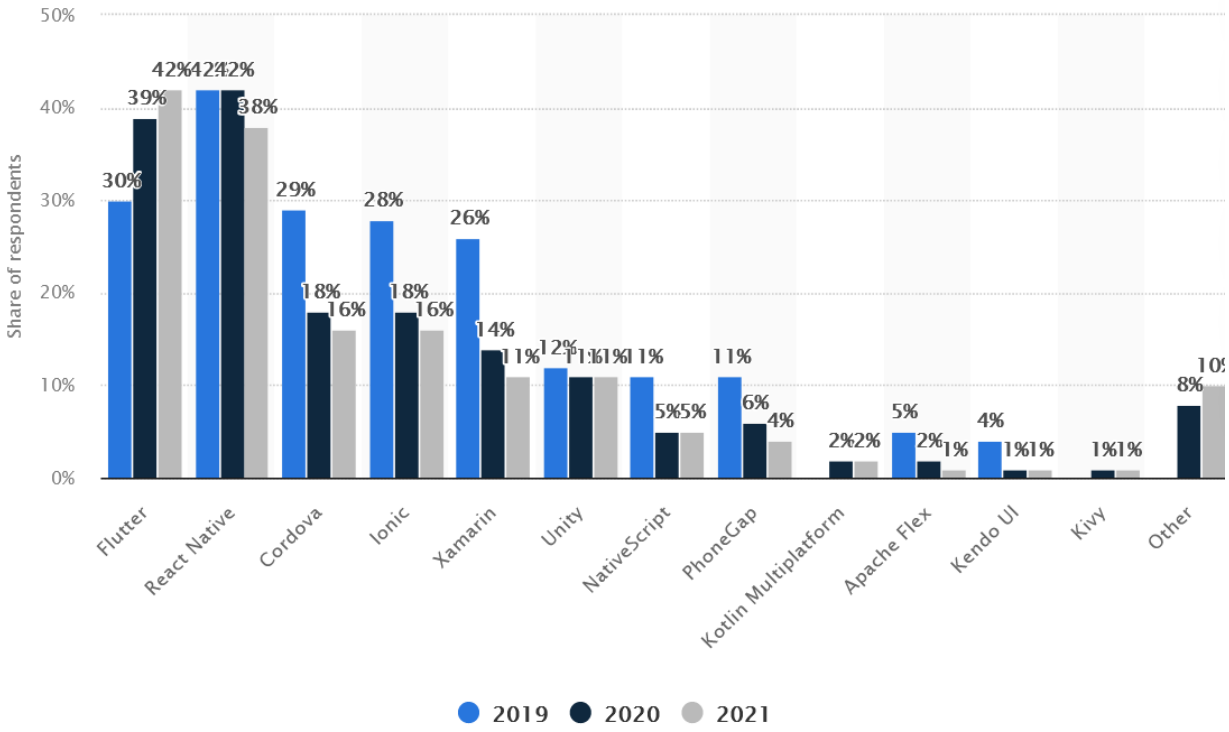
Slika 2.1. Popularnost razvojnih okvira prema pretragama na tražilici Google



Izvor: https://trends.google.com/trends/explore?date=today%205-y&q=%2Fg%2F11f03_rzbg,%2Fg%2F11h03gfxy9,%2Fg%2F1q6l_n0n0,Xamarin,%2Fg%2F11c57wr2yt (12. 8. 2022.)

[5] Gdje je moguće, za termine pretrage rabila se kategorija Software kako bi se uklonile nerelevantne pretrage (npr. *flutter* također znači 'lepršati').

Slika 2.2. Rezultati ankete o korištenju *cross-platform* razvojnih okvira



Izvor: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/ (12.8.2022.)

Prema navedenim grafičkim prikazima može se zaključiti da je razvojni okvir Flutter od svoje pojave (prije manje od pet godina) do danas postao najpopularniji i vjerojatno najčešće korišten razvojni okvir mobilnih aplikacija koji podržava više platformi.

3. FLUTTER I OSTALE METODE RAZVOJA MOBILNIH APLIKACIJA

Flutter je jedan od najnovijih razvojnih okvira za mobilne aplikacije. U ovom poglavlju će se uspoređivati s drugim načinima razvoja mobilnih aplikacija koji su trenutno u najvećoj primjeni.

3.1. O Flutteru

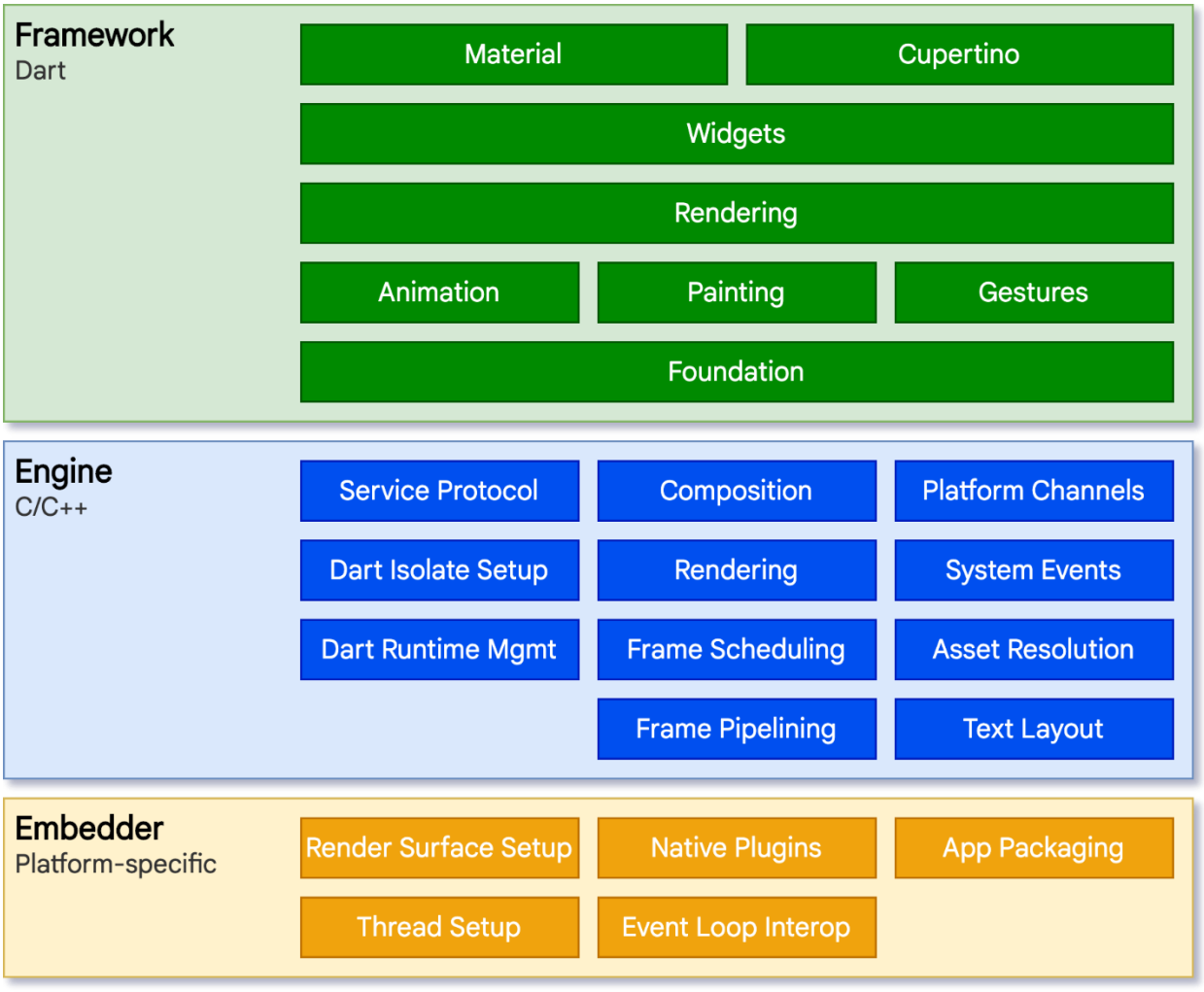
Flutter je predstavljen javnosti 2018. godine, kao najnovije rješenje za *cross-platform* razvoj (Ladd, 2018). Iako ga je razvio Google, on je ponuđen kao besplatan razvojni alat u obliku *open-source* projekta. U početku je bio zamišljen kao razvojni okvir koji se koristi prednostima programskog jezika Dart za razvoj mobilnih aplikacija za iOS i Android. Programski jezik Dart podržava dva načina izvršavanja, primjenjujući prevoditelje (eng. *compiler*) *Just-in-time* (JIT) i *Ahead-of-time* (AOT). Prevođenje JIT omogućuje korištenje značajke *hot-reload*, koja ubrzava vrijeme razvoja aplikacije, jer omogućuje gotovo trenutno vidljive promjene unutar aplikacije nakon izmjene programskog kôda, uz zadržavanje stanja aplikacije (Windmill, 2020). Prevođenje AOT se, zatim, primjenjuje kada je aplikacija spremna za uporabu, a kako bi bila brža i učinkovitija, jer se u tom slučaju kôd aplikacije ne prevodi prilikom izvršavanja nego prije pokretanja aplikacije, čime se skraćuje vrijeme njezinog izvođenja (Windmill, 2020).

Razvojni okvir Flutter sastoji se od nekoliko arhitekturnih slojeva, gdje se viši slojevi koriste funkcionalnostima nižeg sloja. Najniži sloj, *embedder* sloj, sadrži *low-level* funkcionalnosti (funkcionalnosti niske razine) specifičnih platformi, pisane u programskim jezicima platforme, što ih čini vrlo efikasnima. Sloj iznad njega, *engine* sloj, sadrži implementaciju *Skia render enginea*, I/O sučelje, *text layout*, sistemske događaje, *Dart runtime* i *compile toolchain*, i druge *low-level* funkcionalnosti koje su neovisne o platformi[6]. *Engine* sloj je pisan u C++ programskom jeziku, što također omogućuje efikasnost izvođenja, jer je C++ uglavnom namijenjen za razvoj programskih rješenja na uređajima ograničenih resursa. Najgornji sloj, *framework* sloj, onaj je kojim se služe programeri koji upotrebljavaju razvojni okvir Flutter. On se koristi programskim jezikom

[6] Flutter architectural overview. Preuzeto s: https://docs.flutter.dev/resources/architectural-overview (11. 8. 2022.)

Dart i između ostaloga, sadrži alate za izradu logike aplikacije, izgleda i rasporeda, animacija, kao i gotove dizajnirane grafičke elemente. Taj sloj se također sastoji od nekoliko slojeva, a viši slojevi koriste se nižima. Slika 3.1. prikazuje ilustraciju opisanih arhitekturnih slojeva razvojnog okvira Flutter.

Slika 3.1. Pregled slojeva arhitekture razvojnog okvira Flutter



Izvor: https://docs.flutter.dev/resources/architectural-overview (11. 8. 2022.)

Iz navedenog prikaza vidljiva je modularnost ovog pristupa razvoja arhitekture razvojnog okvira. To Flutteru omogućuje veće arhitekturne preinake s minimalnim promjenama *framework* sloja, koji sadrži sve komponente potrebne za razvoj aplikacija. Posljedica toga je mogućnost dodavanja novih podržanih platformi bez većih izmjena aplikacije, kao što je to bio slučaj za *web*, Windows, macOS i Linux platforme u posljednjih par godina.

Flutter, inspiriran principima dizajna razvojnog okvira React Native, koristi se hijerarhijskim stablima objekata grafičkih elemenata, koji se nazivaju *widgeti*[7]. *Widgeti* su zaduženi za sinkronizaciju stanja aplikacije i stanja grafičkog sučelja. To omogućuje manju razinu kompleksnosti u razvoju aplikacije, jer je sam *framework* zadužen za osvježavanje korisničkog sučelja. Svaki *widget* može primiti kontekst *widgeta* koji je hijerarhijski iznad njega. Nakon zaprimljenog događaja, poput interakcije korisnika s aplikacijom, *framework* ažurira korisničko sučelje nakon usporedbe starih i novih *widgeta*. *Widgeti* mogu biti *stateless* ili *stateful*. *Stateless widgeti* ne sadrže stanje, tj. ne mijenjaju se kroz vrijeme, dok *stateful widgeti* imaju stanje i mogu se mijenjati.

3.2. Usporedba Fluttera i razvojnih okvira za nativne aplikacije

Danas na tržištu pametnih telefona prevladavaju dvije platforme, iOS i Android, s ukupnim tržišnim udjelom od preko 99 %[8]. Za razvoj nativnih aplikacija za Appleov iOS, primjenjuju se programski jezici Objective-C ili Swift, a za razvoj nativnih aplikacija za Googleov Android, programski jezici Java ili Kotlin. Flutter se za razvoj aplikacija, za obje navedene platforme, koristi programskim jezikom Dart.

Prednost korištenja Fluttera u odnosu na nativni razvoj aplikacije, je ta što se Flutter aplikacije mogu pokretati na obje platforme, iOS-u i Androidu. Time se vrijeme razvoja aplikacije uvelike skraćuje. Uz to, održavanje Flutter aplikacija je jednostavnije od održavanja nativnih aplikacija za obje platforme, zbog manje količine programskog kôda koji je pisan u samo jednom programskom jeziku. Postoje i nedostaci razvoja aplikacija Flutterom u usporedbi s nativnim razvojem, a to su uglavnom slabije performanse aplikacije, ograničenost raspoloživih komponenti grafičkog sučelja i ograničen pristup značajkama platformi u odnosu na nativne aplikacije. Iako se potonja dva nedostatka mogu zaobići kreiranjem potrebnih komponenti korištenjem nativnog razvoja aplikacije, time se negira prednost korištenja samo jednog programskog jezika.

[7] Flutter architectural overview. Preuzeto s: https://docs.flutter.dev/resources/architectural-overview (11.8.2022.)

[8] Mobile operating systems' market share worldwide from January 2012 to January 2022. Preuzeto s: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (11. 8. 2022.)

S obzirom na to da je Flutter relativno nov razvojni okvir, do sada nije proveden veliki broj istraživanja koja uspoređuju razvoj mobilnih aplikacija u Flutteru s ostalim načinima razvoja, uključujući nativni. Dodatan problem takve usporedbe je iskustvo i poznavanje različitih načina razvoja mobilnih aplikacija, što se može odraziti na kvalitetu finalnih aplikacija koje se uspoređuju. Uz navedene probleme ovih usporedbi, mora se uzeti u obzir i vremenski faktor, jer s vremenom softverska rješenja za razvoj aplikacija sazrijevaju, nadograđuju se i često postaju efikasnija. Uzimajući u obzir navedena ograničenja, u sljedećim odlomcima predstavit će se rezultati takvih usporedbi.

Količina kôda aplikacije je jedna od mjera koja naznačuje njenu kompleksnost i ona se mjeri brojem linija kôda. Treba imati na umu da, ovisno o programskom jeziku i stilu programiranja, broj linija kôda iste aplikacije može varirati. Prema Gonsalves (2018), aplikacija razvijena za obje platforme, Flutterom i nativnim razvojem, sadržavala je više linija kôda kada je bila razvijena nativnim razvojem u usporedbi s razvojem korištenjem razvojnog okvira Flutter. Ti rezultati predočeni su u tablici 3.1 i svjedoče jednostavnosti razvoja u razvojnom okviru Flutter.

Tablica 3.1. Usporedba količine programskog kôda nativnih i Flutter aplikacija

Application	***LOC***	***Files***	***Dependencies***
Native Android	1.870	24	4
Native iOS	1.534	12	6
Flutter	1.267	16	7
Cordova/Ionic	1.123	27	36

Izvor: Gonsalves, M. (2018). *Evaluating the mobile development frameworks Apache Cordova and Flutter and their impact on the development process and application characteristics* (diplomski rad). California State University, Chico

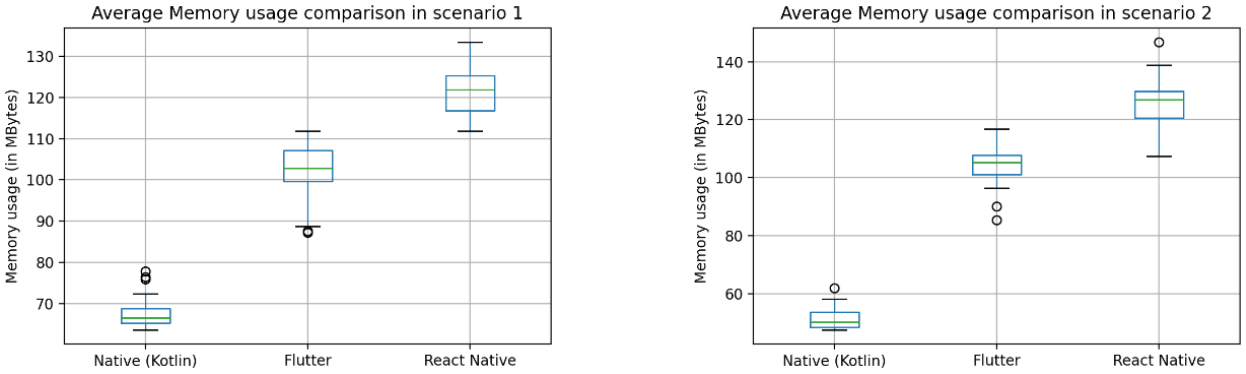
Što se tiče performansi aplikacija, nativne aplikacije ovdje imaju prednost zato što njihova implementacija nema komponentu međusloja, koji je zadužen za apstrakciju funkcionalnosti platforme, jer nativne aplikacije ne moraju podržavati više platformi. To rezultira mjerljivo boljim performansama tijekom njihovog pokretanja i izvođenja u usporedbi s aplikacijama razvijenima Flutterom (Mahendra i Anggorojati, 2020). No,

pokazalo se da su takve razlike u performansama u praksi gotovo nezamjetne (Haider, 2021).

Veličina aplikacije nakon instalacije je još jedna stavka usporedbe, i tu su rezultati testiranja također u skladu s očekivanjima. Flutter aplikacije zauzimaju puno više memorije zbog svog *engine* sloja, koji sadrži *Skia render engine*, i zbog *framework* sloja, koji sadrži sve UI-elemente u obliku *widgeta*. No, ne bi bilo ispravno zaključiti da porastom veličine aplikacije ta razlika u veličini nativnih i Flutter aplikacija linearno korelira, zato što su komponente Flutter aplikacije uglavnom fiksne veličine, čime bi razlika u veličini kompleksnijih aplikacija bila relativno manja (Haider, 2021).

Radna memorija je privremena memorija koju aplikacija zauzima za vrijeme svog izvođenja i tu, kao i drugdje, možemo uočiti razliku. Slika 3.2 prikazuje usporedbu prosječnog zauzeća privremene memorije nativne, Flutter i React Native aplikacije tijekom izvođenja.

Slika 3.2. Usporedba zauzeća privremene memorije nativne, Flutter i React Native Android aplikacije



Izvor: Mahendra, M., Anggorojati, B. (2020). *Evaluating the performance of Android based Cross-Platform App Development Frameworks*. 2020 the 6th International Conference on Communication and Information Processing, 2020, 32-37.

Flutter aplikacije tijekom izvođenja zauzimaju više radne memorije od nativnih aplikacija (Mahendra i Anggorojati, 2020). U budućnosti to možda neće biti problem, jer pametni telefoni već danas imaju količinu radne memorije usporedivu sa stolnim računalima, ali danas bi, za kompleksnije aplikacije na slabijim uređajima, to možda uzrokovalo osjetnu razliku kod pokretanja više različitih aplikacija istovremeno.

Iskorištenje procesora (CPU *utilization*) ukazuje na zahtjevnost izvođenja aplikacije i na njenu energetsku efikasnost. S obzirom na to da se mjeri postotak korištenja hardverske komponente uređaja, mora se uzeti u obzir da su na različitim uređajima moguća različita odstupanja. Mahendra i Anggorojati (2020) utvrdili su da nema značajne razlike u iskorištenju procesora između Flutter i nativne Android aplikacije.

3.3. Usporedba Fluttera i razvojnog okvira React Native

Budući da danas postoji više različitih *cross-platform* razvojnih okvira za mobilne aplikacije, od kojih je Flutter jedan od najnovijih, nije bilo puno istraživanja u kojima se s Flutterom uspoređuje više različitih razvojnih okvira. React Native je uz Flutter trenutno najpopularniji *cross-platform* razvojni okvir, pa je iz tog razloga većina istraživanja koja uključuju Flutter bila provedena uspoređujući ga s React Nativom.

Prednost korištenja razvojnog okvira Flutter u odnosu na React Native se najbolje može vidjeti u načinu izrade korisničkog sučelja. Flutter rabi *widgete*, UI-elemente, čiji dizajn prati smjernice dizajna platforme (*Material Design* za Android i *Cupertino* za iOS). Tim *widgetima* se jednostavno mogu kreirati elementi grafičkog sučelja, koji izgledom i funkcionalnošću imitiraju elemente specifične za pojedine platforme. Razvojni okvir React Native se koristi nativnim UI-elementima za Android i iOS, dok je za implementaciju kompleksnijih elemenata i elemenata s konzistentnim dizajnom na različitim platformama potrebno kreirati vlastite komponente ili se koristiti gotovim *third-party* rješenjima, a što iziskuje više vremena za implementaciju.

Aplikacije React Native upotrebljavaju takozvani *bridge* modul, koji se nalazi između Javascript kôda aplikacije i programskog sučelja platforme (API platforme). Njegova zadaća je prevoditi kôd aplikacije prevoditeljem JIT i pozivati funkcionalnosti platforme (Windmill, 2020). Za razliku od React Nativea, Flutter primjenjuje prevoditelj AOT, što čini komunikaciju između Flutter kôda i aplikativnog sučelja platforme puno bržom, jer nije potrebno prevoditi kôd aplikacije za vrijeme izvođenja. Ta razlika u komunikaciji s aplikativnim sučeljem platforme, animacije u Flutter aplikacijama čini osjetno glađima i aplikaciju efikasnijom. Potrebno je napomenuti da se u razvojnom okviru React Native trenutno uvode promjene koje će zamijeniti postojeći *bridge* s efikasnijim komponentama

koje ostvaruju funkcionalnost sličnu onoj u Flutteru, čime bi se ova prednost Fluttera umanjila.

Već spomenuti programski jezik JavaScript je danas u širokoj primjeni, uglavnom zbog toga što se upotrebljava za razvoj mrežnih stranica, koje su postale nezaobilazan dio naše svakodnevice. Iz tog razloga je programerima koji se bave razvojem mrežnih stranica jednostavnije započeti s razvojem React Native aplikacija, koje se koriste JavaScriptom, nego nativnih aplikacija, koje upotrebljavaju Objective-C, Swift, Javu ili Kotlin programske jezike. S druge strane, razvojni okvir Flutter koristi se programskim jezikom Dart i njegovim razvojnim okvirom. Dart je u usporedbi s JavaScriptom noviji i manje poznat programski jezik, koji do pojave Fluttera gotovo nije ni bio u uporabi. To predstavlja prepreku u korištenju razvojnog okvira Flutter, jer se potrebno upoznati sa sintaksom i funkcionalnošću programskog jezika Dart prije nego što se započne razvoj aplikacija u Flutteru.

Uzimajući u obzir već spomenuta ograničenja u usporedbi performansi aplikacija, rezultati usporedbe Flutter i React Native aplikacija pokazuju da Flutter aplikacije obično imaju bolje performanse, što je u skladu s očekivanjima (Mahendra i Anggorojati, 2020). Ipak, uglavnom nema većih razlika u performansama navedenih aplikacija, zbog čega bi se dalo zaključiti da to nije razlog pada popularnosti razvojnog okvira React Native u odnosu na Flutter.

4. IZRADA FUNKCIONALNE SPECIFIKACIJE MOBILNE APLIKACIJE MOVIELENS

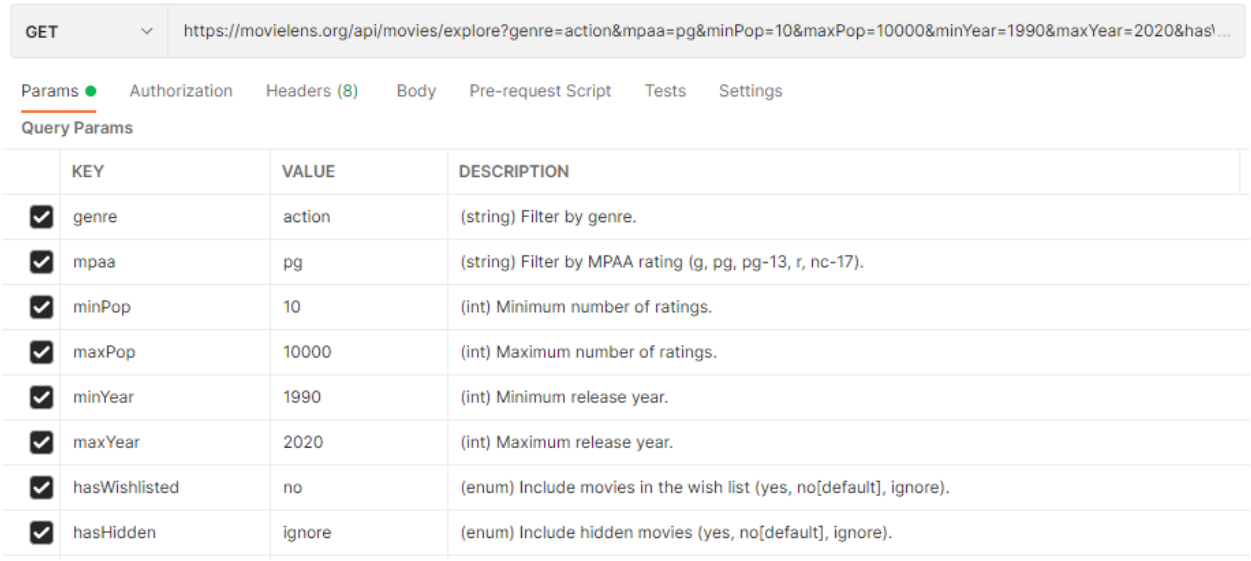
Funkcionalna specifikacija opisuje željenu funkcionalnost aplikacije, koju je zatim potrebno implementirati. U ovom poglavlju opisuje se proces izrade funkcionalne specifikacije za mobilnu aplikaciju MovieLens, koja se bazira na istoimenom istraživačkom projektu GroupLens Research grupe sa Sveučilišta u Minnesoti.

4.1. Istraživanje dostupnih funkcionalnosti mrežne stranice MovieLensa

Mrežna stranica projekta MovieLens služi za istraživanje metoda i algoritama za kreiranje personaliziranih preporuka filmova. Ova se mrežna stranica koristi programskim sučeljem aplikacije (API-jem) za dohvat i prikaz filmova, podataka vezanih za filmove, dohvat i upravljanje korisničkim postavkama, ocjenjivanje filmova i ostalim dostupnim funkcionalnostima. U ovom slučaju radi se o RESTful API-ju, koji upotrebljava *GET, POST, PUT* i *DELETE* metode *Hypertext Transfer* protokola (HTTP-a), a vraća objekte u JSON (*JavaScript Object Notation*) formatu. Ne postoji javno dostupna dokumentacija API-ja koji se primjenjuje na MovieLens stranici, pa je bilo potrebno ručno istražiti dostupne funkcionalnosti promatrajući mrežni promet između preglednika i poslužitelja stranice i potom ih dokumentirati.

Proces dokumentiranja API-ja uključivao je interakciju s mrežnom stranicom, tijekom koje su zabilježeni specifični XMLHttpRequest objekti, a koji služe za komunikaciju s poslužiteljem. Nakon toga je svaki od njih dokumentiran u aplikaciji za dokumentiranje i testiranje RESTful API-ja, Postmanu. Dokumentacija uključuje korištenu metodu HTTP-a, mrežnu adresu pristupne točke (API *endpointa*), zaglavlja HTTP-a, eventualni sadržaj API-poziva i moguće parametre s njihovim opisom i tipom podataka. Jednom dokumentirani objekt zatim je testiran, a odgovor poslužitelja zabilježen i spremljen uz dokumentaciju kako bi poslužio kao primjer primljenih podataka. Primjer dokumentacije API-ja nalazi se na slikama 4.1 i 4.2.

Slika 4.1. Primjer dokumentacije za poziv API-ja



GET https://movielens.org/api/movies/explore?genre=action&mpaa=pg&minPop=10&maxPop=10000&minYear=1990&maxYear=2020&has\...

Params ● Authorization Headers (8) Body Pre-request Script Tests Settings

Query Params

KEY	VALUE	DESCRIPTION
genre	action	(string) Filter by genre.
mpaa	pg	(string) Filter by MPAA rating (g, pg, pg-13, r, nc-17).
minPop	10	(int) Minimum number of ratings.
maxPop	10000	(int) Maximum number of ratings.
minYear	1990	(int) Minimum release year.
maxYear	2020	(int) Maximum release year.
hasWishlisted	no	(enum) Include movies in the wish list (yes, no[default], ignore).
hasHidden	ignore	(enum) Include hidden movies (yes, no[default], ignore).

Izvor: autorovo djelo

Slika 4.2. Primjer dokumentacije zaprimljenog odgovora nakon poziva API-ja


```
{
    "status": "success",
    "data": {
        "title": null,
        "description": null,
        "url": null,
        "searchResults": [
            {
                "movieId": 213111,
                "movie": {
                    "movieId": 213111,
                    "tmdbMovieId": 592867,
                    "imdbMovieId": "10127562",
                    "title": "Dragon Quest: Your Story",
                    "originalTitle": "ドラゴンクエスト ユア・ストーリー",
                    "mpaa": "PG",
                    "runtime": 102,
                    "releaseDate": "2019-08-02",
                    "dvdReleaseDate": null,
                    "genres": [
                        "Animation",
                        "Adventure",
                        "Comedy",
```


Izvor: autorovo djelo

Slika 4.1 sadrži metodu HTTP-a, uz adresu pristupne točke API-ja i listu njegovih parametara s opisima. Ostale stavke dokumentacije nalaze se u *Authorization*, *Headers*

i *Body* izbornicima ispod adrese pristupne točke. Slika 4.2 prikazuje isječak spremljenog odgovora API-ja u obliku objekta u JSON formatu (dalje u tekstu: JSON objekt).

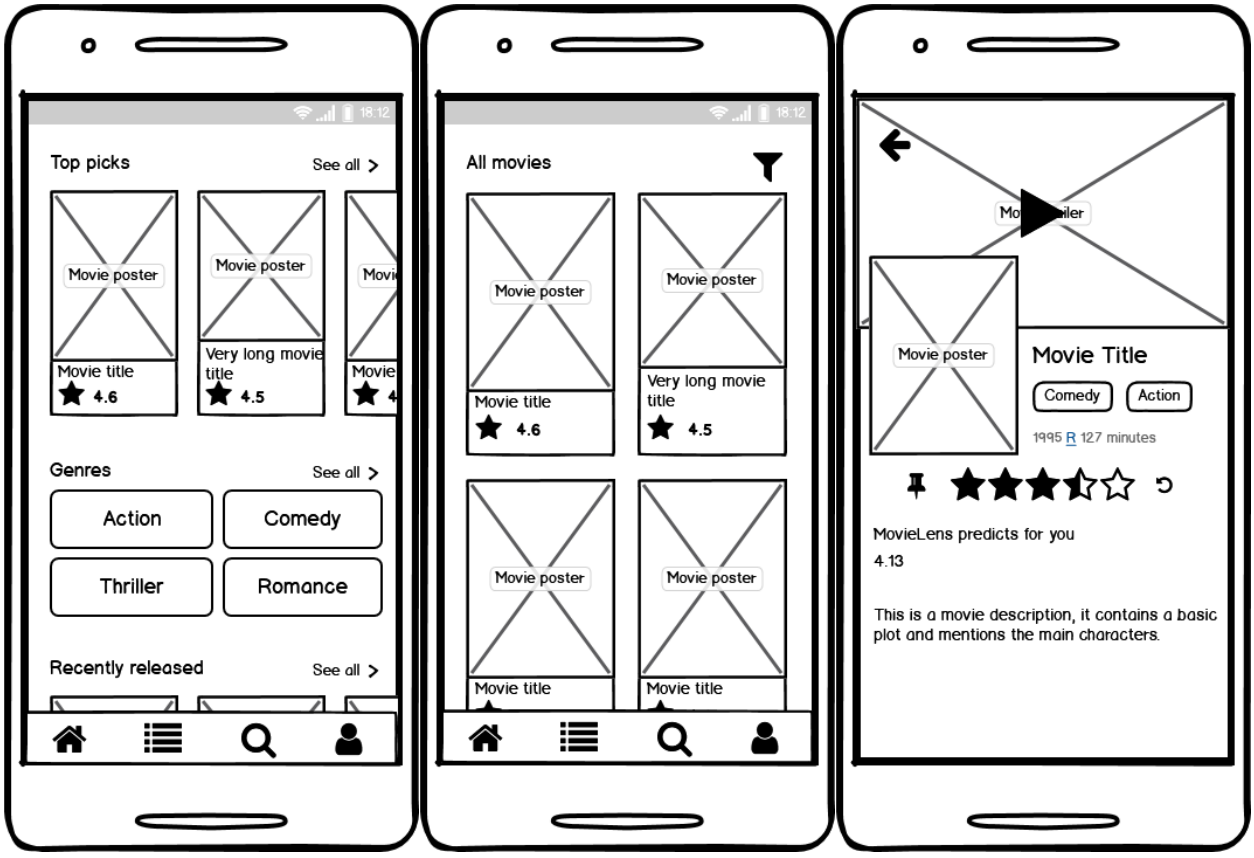
4.2. Izrada predloška grafičkog sučelja

Jednom kreirana, dokumentacija je pružila uvid u dostupne funkcionalnosti koje se mogu primjenjivati u mobilnoj aplikaciji. Dizajn grafičkog sučelja mobilne aplikacije MovieLens, uvelike je inspiriran postojećim dizajnom istoimene mrežne stranice, ali su pritom eliminirane funkcionalnosti koje su procijenjene kao manje bitne za ovaj projekt. Identificirano je nekoliko bitnih zaslona (eng. *screens*), koji bi bili potrebni za ostvarivanje željene funkcionalnosti aplikacije, pri čemu su se nastojale uočiti sličnosti zaslona kako bi se njihov broj mogao smanjiti korištenjem istog zaslona za više različitih prikaza informacija. Željena funkcionalnost uključuje prijavu i odjavu korisnika, pregled glavnih kategorija filmova, pregled filmova unutar kategorije (s mogućnošću promjene opcija filtriranja), pregled detalja filma, pretraživanje filmova, pregled i upravljanje postavkama korisničkog profila i aplikacije, ocjenjivanje filmova, dodavanje filmova u popis želja za gledanje, te skrivanje filmova. Popis zaslona s privremenim nazivima je kako slijedi:

- *SignInScreen* – sadrži polja za unos podataka za prijavu i gumb za prijavu
- *HomeScreen* – sadrži brzi pregled najkorištenijih kategorija filmova
- *MovieList* – sadrži pregled rezultata pretrage filmova i opciju filtriranja
- *MovieListFilters* – sadrži pregled opcija filtriranja filmova
- *MovieViewScreen* – sadrži pregled informacija o pojedinom filmu
- *SearchScreen* – zaslon za pretragu filmova korištenjem znakovnog niza
- *Profile&Settings* – sadrži pregled postavki i opcija aplikacije.

Potom je izrađen *wireframe* predložak dizajna grafičkog sučelja s navigacijskim mogućnostima aplikacije, čiji se primjer nalazi na slici 4.3.

Slika 4.3. Predlošci dizajna grafičkog sučelja



Izvor: autorovo djelo

Zasloni prikazani na slici su redom: *HomeScreen*, *MovieList* i *MovieViewScreen*. Prekriženi pravokutnici označavaju elemente koji sadržavaju grafičke prikaze poput slika. Izbornik u donjem dijelu prvih dvaju predložaka čini glavni izbornik aplikacije, a do trećeg zaslona dolazi se pritiskom na neku od kartica filmova prikazanih na prva dva predloška. Povratak na prethodni zaslon se u tom slučaju izvršava pritiskom na strelicu u gornjem lijevom kutu *MovieViewScreen* zaslona ili korištenjem navigacijskih mogućnosti platforme. Budući da se radi o predlošku dizajna, finalni dizajn grafičkog sučelja aplikacije može se u nekoj mjeri razlikovati od prikazanog. Svi predlošci dizajna grafičkog sučelja nalaze se u prilogu ovoga rada.

5. PROCES IZRADE APLIKACIJE MOVIELENS KORIŠTENJEM FLUTTERA

Izrada aplikacije korištenjem razvojnih okolina Flutter i Dart uključuje postavljanje razvojne okoline, odabir teme aplikacije i glavnih komponenti te izradu logike aplikacije i korisničkog sučelja. Jedan od preduvjeta razvoja aplikacije korištenjem bilo kojeg razvojnog okvira je potpuna i kvalitetna dokumentacija za korištenje razvojnog okvira, što je jedna od velikih prednosti Fluttera, jer njegova dokumentacija, osim tehničke specifikacije, sadrži i mnoštvo primjera i preporuka uz interaktivni sadržaj.

5.1. Postavljanje razvojne okoline i odabir glavnih komponenti aplikacije

Kako bi se izradila aplikacija korištenjem razvojnog okvira Flutter, potrebno je preuzeti Flutter SDK (*software development kit*) za platformu na kojoj će se razvijati aplikacija, i iz *command-line interfacea*, poput CMD-a na Windowsu, pokrenuti naredbu *flutter doctor* u direktoriju u kojem se nalazi Flutter SDK. Time se provjerava jesu li postavljeni svi softverski alati potrebni za razvoj Flutter aplikacija, pri čemu se stvara izvještaj koji navodi koje je softverske alate potrebno instalirati na računalo. U ovom slučaju preuzet je Flutter SDK za Windows operativni sustav i, s obzirom na to da se razvija Flutter aplikacija za Android platformu, prilikom pokretanja *flutter doctor* naredbe u izvještaju se nalazi Android Studio kao jedan od potrebnih softverskih alata koji nedostaju. Nakon instalacije svih potrebnih stavki izvještaja može se započeti s razvojem aplikacije u IDE-u (*integrated development environment*) koji podržava Flutter i programski jezik Dart, jedan od kojih je i Android Studio, koji je korišten u ovom slučaju. U IDE-u Android Studio potrebno je zatim instalirati dodatke (eng. *plugins*) za Flutter i Dart, nakon čega se može započeti s razvojem aplikacije.

Prilikom stvaranja novog Flutter projekta, kreira se početni projekt, koji demonstrira korištenje Fluttera za razvoj aplikacije jednostavnog brojača, čija je funkcionalnost objašnjena u službenoj dokumentaciji. Svaka Flutter aplikacija pokreće se iz *main.dart* datoteke, koja sadrži *main* funkciju. *Main* funkcija prima parametar naziva *child*, koji označuje polazišni *widget* aplikacije. Taj polazišni *widget* obično je onaj koji sadrži

komponente koje imitiraju nativne komponente određene platforme (*MaterialApp* ili *CupertinoApp widgeti*). Preporučuje se korištenje *MaterialApp widgeta* jer je fleksibilniji, pa se iz tog razloga upotrebljavao i u ovom projektu.

Prije nego što se započne s razvojem aplikacije u Flutteru, potrebno je donijeti odluku o tome koji će se alat primjenjivati za *state management*. *State management* je termin koji označuje upravljanje stanjem aplikacije, a stanje, prema definiciji Flutter dokumentacije, čine svi podatci potrebni za rekreiranje grafičkog sučelja u bilo kojem trenutku. Za razliku od većine razvojnih okvira za mobilne aplikacije, Flutter nema predefinirano rješenje za *state management*, ali njegova dokumentacija sadrži popis mogućih *state management* rješenja. Trenutna preporuka je korištenje paketa *Provider* za *state management*, zbog njegove jednostavnosti i značajki koje pokrivaju većinu željenih funkcionalnosti *state management* rješenja, čime se pokazao kao dobro rješenje za potrebe razvoja aplikacije MovieLens.

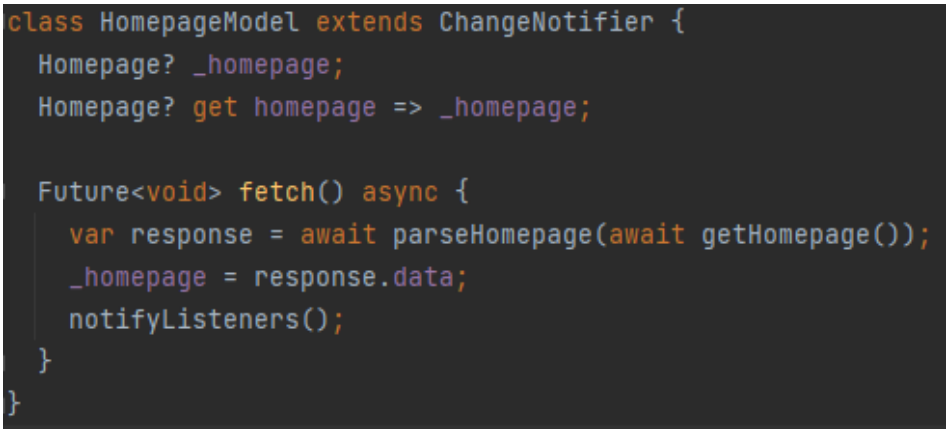
Flutter i Dart paketi su softverski paketi dodatnih funkcionalnosti za istoimene razvojne okvire. Svi navedeni paketi dostupni su na mrežnoj stranici *pub.dev*, koja omogućuje filtriranje paketa prema platformi, razvojnoj okolini i podršci za relativno novu *null safety* značajku programskog jezika Dart. Uz to, u nastojanju da se olakša prepoznatljivost kvalitetnih paketa, postoji sustav bodovanja paketa prema kriterijima poput kvalitete dokumentacije, praćenja konvencije Dart datoteka, automatske provjere ispravnosti kôda i podrške za najnovije verzije paketa o kojima taj paket ovisi. Flutter i Dart paketi sadrže i upute za njihovu instalaciju u projekt kako bi se potom mogli upotrebljavati. Svi paketi o kojima projekt ovisi nalaze se u *pubspec.yaml* datoteci projekta, pod atributom *dependencies*, s naznačenom verzijom paketa koja se rabi u projektu (npr. „http: ^0.13.5“). Pokretanjem naredbe *flutter pub upgrade*, paketi projekta se nadograđuju na najnoviju moguću verziju s obzirom na ovisnost o drugim paketima.

5.2. State management

Paket *Provider* sadrži *widgete* koji omogućuju upravljanje stanjem aplikacije. Klasa (eng. *class*) *ChangeNotifier* omogućuje slanje obavijesti o promjenama podataka unutar objekta klase i dio je Flutter SDK-a. *ChangeNotifier* se rabi tako da se funkcionalnost postojeće klase proširi ključnom riječju *extends*, a unutar funkcija (metoda) klase, gdje je

poželjno, pozove funkcija *notifyListeners*. Primjer uporabe navedene funkcionalnosti nalazi se na slici 5.1.

Slika 5.1. Primjer implementacije funkcionalnosti klase *ChangeNotifier*



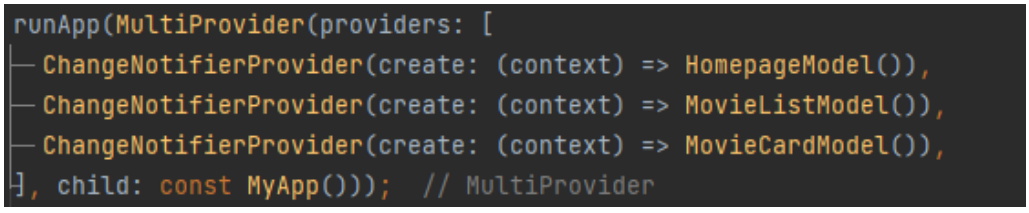
```
class HomepageModel extends ChangeNotifier {
  Homepage? _homepage;
  Homepage? get homepage => _homepage;

  Future<void> fetch() async {
    var response = await parseHomepage(await getHomepage());
    _homepage = response.data;
    notifyListeners();
  }
}
```

Izvor: autorovo djelo

U navedenom primjeru definirana je klasa *HomepageModel*, koja proširuje funkcionalnosti *ChangeNotifiera*. Ona sadrži varijablu *_homepage* tipa *Homepage?*, funkciju *get*, zaduženu za dohvat vrijednosti varijable *_homepage*, i funkciju *fetch.* U Dartu znak donje crte (_) ispred naziva varijable označava privatnu varijablu, koja je dostupna samo klasi u kojoj se nalazi. Unutar funkcije *fetch* definira se varijabla *response*, u koju se sprema vrijednost koju vraća funkcija *parseHomepage*. Nakon toga se u privatnu varijablu *_homepage* sprema *data* element varijable *response* i poziva funkcija *notifyListeners*, koja šalje obavijest o promjeni.

ChangeNotifierProvider widget, koji je dio *Providera*, objektima (*widgetima*) koji su hijerarhijski ispod njega pruža instancu *ChangeNotifiera*. Na taj se način tim *widgetima* omogućuje pregled i upravljanje stanjem klase koja ima funkcionalnost *ChangeNotifiera*. Ako je potrebno upotrebljavati više klasa, može se upotrijebiti *MultiProvider*, kao što je prikazano na slici 5.2.

Slika 5.2. Primjer korištenja *widgeta ChangeNotifierProvider*



```
runApp(MultiProvider(providers: [
  ChangeNotifierProvider(create: (context) => HomepageModel()),
  ChangeNotifierProvider(create: (context) => MovieListModel()),
  ChangeNotifierProvider(create: (context) => MovieCardModel()),
], child: const MyApp()));  // MultiProvider
```

Izvor: autorovo djelo

Na slici je vidljivo da *MultiProvider* prima parametar *providers*, koji je zapravo lista *ChangeNotifierProvider widgeta*. Svaki *ChangeNotifierProvider* pod parametrom *create* prima funkciju s parametrom konteksta (u ovom slučaju primjenjuje se sintaksa za stvaranje jednostavne funkcije bez naziva i tipa podatka), koja vraća objekt klase koja proširuje *ChangeNotifier*.

Da bi se obavijesti o promjeni stanja mogle obraditi i ažurirati, korisničko sučelje potrebno je funkciji *builder Consumer widgeta* paketa *Provider* predati kontekst i naziv instance klase, te parametar *child*, koji može poslužiti za optimizaciju. *Builder* je funkcija *widgeta* koja uvijek vraća *widget*, pa će ona u ovom slučaju vratiti *widget* koji koristi klasu s podacima koje je potrebno ažurirati. *Widgetu Consumer* je također potrebno definirati tip, odnosno klasu kojom će se koristiti. *Widget Consumer* se mora nalaziti hijerarhijski ispod *ChangeNotifierProvidera* iste klase, kako bi imao pristup obavijestima koje se šalju. Slika 5.3 prikazuje primjer korištenja *Consumer widgeta*.

Slika 5.3. Primjer korištenja *widgeta Consumer*

```
Consumer<MovieListModel>(builder: (context, movieList, child) {
  return MovieList(queryResults: movieList.movieQueryResults);
}),  // Consumer
```

Izvor: autorovo djelo

Naposljetku, ponekad je potrebno pristupiti podacima ili funkcijama objekta klase koja se koristi funkcionalnosti *ChangeNotifiera*, a da se pritom ne ažurira korisničko sučelje. To se može učiniti korištenjem funkcionalnosti *Providera - Provider.of*, gdje se navodi klasa, kontekst i parametar *listen* postavljen na *false* - kao što je prikazano na slici 5.4.

Slika 5.4. Primjer korištenja funkcionalnosti *Provider.of*

```
Provider.of<MovieListModel>(context, listen: false)
    .movieQuery
    .pageNum = queryResults!.pager.currentPage - 1;
```

Izvor: autorovo djelo

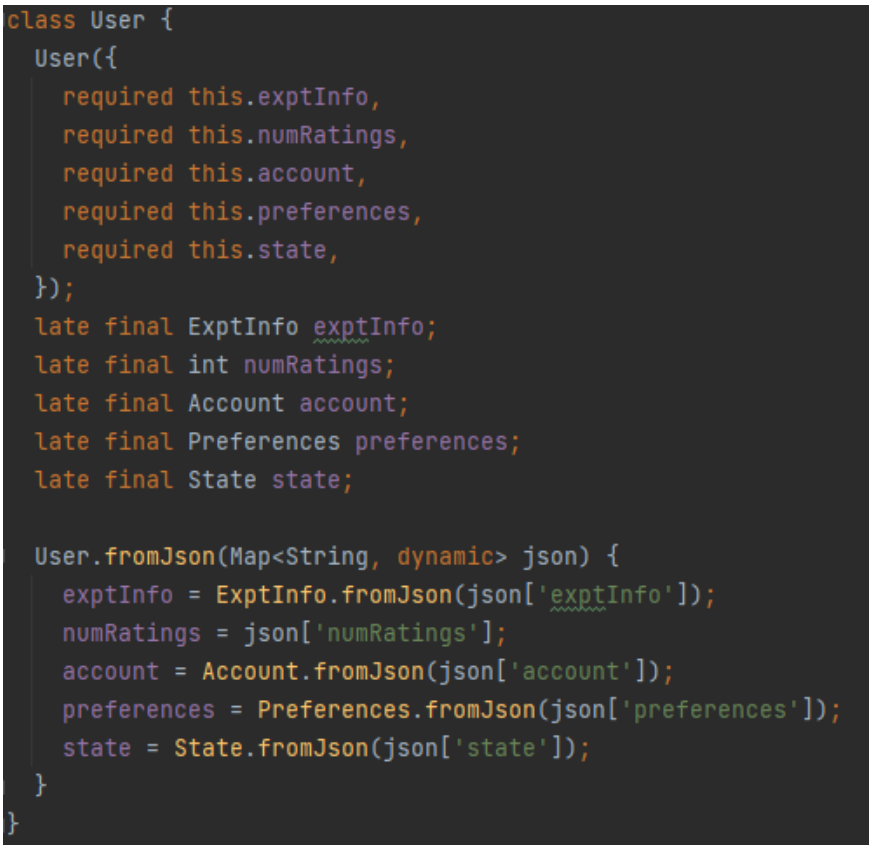
U tom primjeru, *Provider.of* prima tip *MovieListModel* i iz postojećeg objekta te klase dohvaća element *pageNum* elementa *movieQuery* i u njega sprema određenu vrijednost.

5.3. Izrada modela za komunikaciju s API-jem

U prethodnom poglavlju izvršen je proces dokumentiranja API-ja, za komunikaciju s poslužiteljem MovieLensa. API je sada potrebno implementirati u mobilnu aplikaciju MovieLens kako bi se mogla iskoristiti njegova funkcionalnost. Pritom je potrebno kreirati klase koje odražavaju podatke JSON objekata, koje API vraća. S obzirom na to da je Dart, za razliku od JavaScripta, *strongly typed* programski jezik, varijablama se mora predati podatak koji tipom odgovara tipu varijable, na primjer *String* tip varijable može primiti samo znakovni niz. To znači da je pri definiranju klase koja imitira JSON objekt potrebno definirati ispravne tipove podataka.

Flutter nudi nekoliko opcija za stvaranje klasa s mogućnošću serijalizacije JSON objekata, gdje serijalizacija znači kopiranje podataka iz JSON objekta u klasu. Za manje projekte moguće je ručno kreirati klase i funkcije za serijalizaciju, ali se za veće projekte preporučuje korištenje biblioteka (paketa) za generiranje kôda. Za potrebe ovog projekta korišten je *online* generator kôda, tako da se iz svakog API *endpointa* kopirao primjer JSON objekta, koji se dobio kao odgovor u generator kôda, nakon čega se generirani kôd modificirao da bi tipovima u potpunosti odgovarao JSON objektu. Prilikom generiranja potrebnih klasa uočilo se ponavljanje određenih dijelova pojedinih klasa, te su se ti dijelovi izvukli kako bi se stvorila klasa koja će biti sadržana unutar ostalih klasa, nakon čega su prilagođene funkcije za serijalizaciju JSON objekata, kako bi odgovarale napravljenim izmjenama. Primjer jedne takve klase nalazi se na slici 5.5.

Slika 5.5. Primjer Dart klase pripadajućeg JSON objekta



```
class User {
  User({
    required this.exptInfo,
    required this.numRatings,
    required this.account,
    required this.preferences,
    required this.state,
  });
  late final ExptInfo exptInfo;
  late final int numRatings;
  late final Account account;
  late final Preferences preferences;
  late final State state;

  User.fromJson(Map<String, dynamic> json) {
    exptInfo = ExptInfo.fromJson(json['exptInfo']);
    numRatings = json['numRatings'];
    account = Account.fromJson(json['account']);
    preferences = Preferences.fromJson(json['preferences']);
    state = State.fromJson(json['state']);
  }
}
```

Izvor: autorovo djelo

Klasa *User* iz primjera sadrži konstruktor klase čiji su parametri sve varijable klase i čiji je unos obavezan, što je naznačeno ključnom riječju *required*. Ključne riječi *late* i *final* označavaju kasniju inicijalizaciju varijable i da je, jednom kada joj je postavljena vrijednost, konstantna. Funkcija *fromJson* prima JSON objekt iz kojeg čita vrijednosti i sprema ih u varijable klase.

Jednom kada su sve potrebne klase kreirane, moguće je kopirati podatke zaprimljenih JSON objekata u objekte klasa programskog jezika Dart i primjenjivati ih u kôdu aplikacije. Da bi se JSON objekti mogli zaprimiti, potrebno je ostvariti komunikaciju s poslužiteljem pomoću odgovarajućih metoda HTTP-a. U tu svrhu kreirane su osnovne funkcije: *get*, *post*, *put* i *delete* koje upotrebljavaju istoimene funkcije paketa *http* razvojnog okvira Dart. U cilju osiguravanja modularnosti i jednostavnosti funkcionalnih komponenti aplikacije, kreirana je klasa *ApiResponse*, koja čini apstrakcijski sloj između odgovora poslužitelja i klasa u koje će se kopirati podaci. Na slici 5.6 nalazi se primjer funkcije *get*, koja se koristi

zaglavljem s kolačićem sesije spremljenim u varijabli *_headers* i koje se, u slučaju uspješne komunikacije s API-jem, ažurira funkcijom *updateCookie*. Uz to, kolačić sesije sprema se na trajnu memoriju uređaja pomoću Flutter paketa *Hive*, kako bi se mogao upotrebljavati i nakon ponovnog pokretanja aplikacije.

Slika 5.6. Primjer funkcije *get* za komunikaciju s API-jem

```
Future<ApiResponse> get(String path) async {
  ApiResponse apiResponse = ApiResponse();
  try {
    http.Response response =
        await http.get(Uri.parse('$_baseUrl$path'), headers: _headers);
    switch (response.statusCode) {
      case 200:
        updateCookie(response);
        apiResponse.data = utf8.decode(response.bodyBytes);
        break;
      default:
        apiResponse.apiError = ApiError.fromJson(json.decode(response.body));
        break;
    }
  } on SocketException {
    apiResponse.apiError = ApiError("Server error. Please retry");
  }
  return apiResponse;
}
```

Izvor: autorovo djelo

Iz priložene slike može se uočiti da je tip podataka koji funkcija vraća *Future<ApiResponse>*, što je zapravo klasa *Future* tipa *ApiResponse*. U programskom jeziku Dart, *Future* je klasa koja se rabi prilikom asinkronog izvršavanja i ona prima tip koji je definiran unutar < i > znakova. *Future* nije ništa drugo negoli „obećanje“ da će se u varijablu tog tipa, u jednom trenutku pohraniti vrijednost koja ima tip koji je definiran između < i >. Svaka funkcija tipa *Future*<T> (gdje T predstavlja bilo koji tip) mora upotrebljavati ključne riječi *async* i *await*, gdje *async* označava asinkronu funkciju, a *await* označava operaciju čije je izvršavanje asinkrono. Slijedom toga, svaka funkcija koja poziva asinkronu funkciju mora također biti asinkrona.

Nadalje, koristeći se novokreiranim *get*, *post*, *put* i *delete* funkcijama, potrebno je kreirati i funkcije za specifične pristupne točke API-ja. S obzirom na to da neki pozivi API-ja upotrebljavaju parametre, kreirana je klasa koja sadrži sve dokumentirane parametre takvih poziva, kao i *enum* tipove podataka za parametre, koji mogu poprimiti samo određene vrijednosti poput *yes*, *no* i *ignore*. Primjer jedne takve funkcije prikazan je na slici 5.7.

Slika 5.7. Primjer funkcije za komunikaciju sa specifičnom pristupnom točkom API-ja

```
Future<ApiResponse> getHomepage() async {
  return await session.get('/users/me/frontpage');
}
```

Izvor: autorovo djelo

Jednom zaprimljeni, odgovori potom moraju biti pretvoreni u objekt odgovarajuće klase. Budući da ta operacija može biti relativno dugotrajna, kako bi se izbjeglo blokiranje glavne dretve (koja je, između ostaloga, zadužena za prikaz grafičkog sučelja, zbog čega bi došlo do vizualnog usporenja aplikacije) potrebno se koristiti *compute* funkcijom razvojnog okvira Flutter. *Compute* kao parametre prima funkciju i parametar funkcije te za navedenu stvara novu izoliranu dretvu, koju pri završetku izvođenja funkcije gasi i vraća njezinu finalnu vrijednost. Takva funkcionalnost omogućuje vrlo jednostavnu implementaciju višedretvenosti u aplikacijama. Na slici 5.8 nalazi se primjer korištenja funkcije *compute*.

Slika 5.8. Primjer korištenja funkcije *compute*

```
Future<homepage.ApiResponseData> parseHomepage(
    ApiResponse homepageResponse) async {
  return compute(homepage.apiResponseDataFromJson, homepageResponse.data);
}
```

Izvor: autorovo djelo

Priloženi primjer sadrži funkciju *parseHomepage*, koja prima objekt *ApiResponse* klase u kojoj se, između ostaloga, nalazi zaprimljeni JSON objekt. Taj se objekt dalje prosljeđuje *compute* funkciji, uz funkciju koja je zadužena za njegovu serijalizaciju - u objekt klase *homepage.ApiResponseData*.

5.4. Izrada korisničkog sučelja

Kao što je ranije spomenuto, polazišna točka svake Flutter aplikacije je *main* funkcija, koja se obično nalazi u *main.dart* datoteci. U njoj će biti definirana glavna tema aplikacije, rute aplikacije koje će se kasnije primjenjivati za navigaciju, i još neke funkcionalnosti koje se moraju inicijalizirati prilikom pokretanja aplikacije. Početna ruta aplikacije bit će *Landing widget* definiran u datoteci *landing.dart*. *Landing* provjerava postoji li sesija i ovisno o tome, preusmjerava korisnika na početni zaslon, ili zaslon za prijavu, a za vrijeme obavljanja provjere prikazuje kružni indikator učitavanja kako bi korisniku signalizirao da se obavlja neka radnja.

U slučaju da ne postoji ranije stvorena sesija, korisnik je preusmjeren na zaslon za prijavu, koji sadrži formu s podacima za prijavu. Nakon unosa korisničkog imena i lozinke te pritiska na gumb za prijavu, poziva se funkcija koja provjerava jesu li uneseni podaci u polja za prijavu i ako nisu, upozorava korisnika. Ako je provjera uspješna, poziva se funkcija koja se za prijavu koristi API-jem, koji u slučaju uspješne prijave sprema kolačić sesije i korisnika preusmjerava na početni zaslon i šalje zahtjev za dohvatom sadržaja, a u slučaju neuspješne prijave, prikazuje poruku greške. Zaslon za prijavu definiran je *widgetom Login* koji je, budući da sadržava stanje, *stateful widget*.

Stateful widgeti su klase koje proširuju *widget StatefulWidget. Stateful widgetima* je potrebno definirati klasu koja proširuje klasu *State*<T> (gdje je T klasa *stateful widgeta*), u kojoj se nalaze funkcionalnosti *widgeta* i *build widget* u kojem je definirano grafičko sučelje. Uz to je moguće definirati i inicijalno stanje, odnosno funkcionalnosti koje se izvršavaju prilikom kreiranja *stateful widgeta* korištenjem funkcije *initState*. *Build widget* i funkciju *initState* je potrebno predznačiti ključnom riječi *@override*, kako bi se uhvatile moguće greške kod pokušaja premošćivanja funkcionalnosti *widgeta StatefulWidget*.

Stateless widgeti proširuju *widget StatelessWidget* i za razliku od *stateful widgeta*, ne sadrže stanje. Početni zaslon je u ovom slučaju definiran kao *stateless widget* naziva *Homepage*, jer samo prikazuje sadržaj početnog zaslona, a on se nalazi unutar *widgeta MyHomePage*, definiranog u datoteci *home.dart*. Programski kôd *Homepage widgeta* prikazan je na slici 5.9.

Slika 5.9. Isječak kôda za prikaz početne stranice *Home*

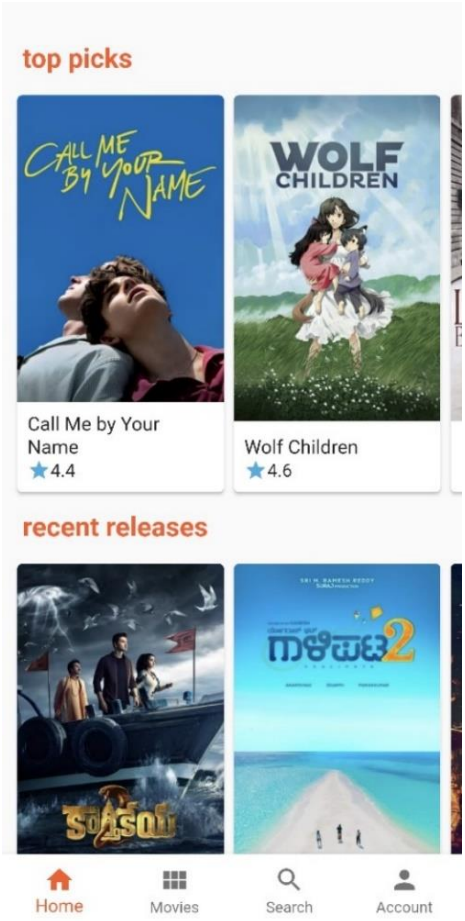
```
class Homepage extends StatelessWidget {
  const Homepage({Key? key, required this.homepage, required this.switchScreen})
      : super(key: key);

  final Function switchScreen;
  final home.Homepage? homepage;

  @override
  Widget build(BuildContext context) {
    return homepage == null
        ? const Center(child: CircularProgressIndicator())
        : RefreshIndicator(
            child: SingleChildScrollView(
                padding: const EdgeInsets.only(top: 60, left: 8),
                child: Column(
                    children: List.generate(
                        homepage!.listOfSearchResults.length,
                        (index) => Column(
                              crossAxisAlignment: CrossAxisAlignment.start,
                              children: [
                                TextButton(
                                    onPressed: () {
                                      // Load the whole category movie list
                                      Provider.of<MovieListModel>(context,
                                              listen: false)
                                          .fetchCategory(homepage!
                                              .listOfSearchResults[index]
                                              .title!);
                                      switchScreen(1);
                                    },
                                    child: Text(
                                      homepage!.listOfSearchResults[index].title
                                          .toString(),
                                      style: const TextStyle(
                                          fontWeight: FontWeight.bold,
                                          fontSize: 21),  // TextStyle
                                    )),  // Text, TextButton
                                HomepageList(
                                    movieList: homepage!
                                        .listOfSearchResults[index]
                                        .searchResults),  // HomepageList
                              ],
                            )  // Column
                    ))),  // List.generate, Column, SingleChildScrollView
            onRefresh: () =>
                Provider.of<HomepageModel>(context, listen: false).fetch());  // RefreshIndicator
  }
}
```

Izvor: autorovo djelo

Tijekom inicijalizacije *Homepage widgeta*, provjerava se postoji li objekt klase *Homepage* iz datoteke *homepage.dart*, koja se nalazi u direktoriju *models* i koja sadrži ranije dohvaćene podatke. Ako podatci ne postoje, prikazuje se kružni indikator učitavanja sve dok isti ne postanu dostupni. U tom se slučaju *widget Homepage* ponovno kreira.

Kada su podatci dostupni, na zaslonu se stvara složena struktura grafičkih elemenata definiranih u preostalom dijelu kôda *Homepage widgeta*. Početni zaslon ima više redova elemenata, pa je iz tog razloga korišten *widget SingleChildScrollView*, koji omogućuje *scroll* navigaciju sučelja. Unutar tog *widgeta* nalazi se stupac definiran *widgetom Column*, koji sadrži listu stupaca čiji su elementi *widgeti* - *TextButton*, s nazivom kategorije i *HomepageList*, kojim je definiran niz kartica filmova s određenim informacijama. Cijeli *SingleChildScrollView* nalazi se pod *widgetom RefreshIndicator*, koji povlačenjem zaslona prema dolje omogućuje osvježavanje njegovog sadržaja. Osvježava se pozivom funkcije *fetch*, *HomepageModel* modela *Providera* iz slike 5.1 definiranog u *provider_classes.dart* datoteci, koja potom poziva *notifyListeners* funkciju da obavijesti druge komponente sučelja o nastaloj promjeni. Također, pritiskom na gumb s nazivom kategorije, poziva se slična funkcionalnost, ali uz to se korisnika prebacuje na drugi zaslon pozivom funkcije *switchScreen*, koja je kao argument predana *Homepage widgetu*. Rezultat svega opisanog prikazan je na slici 5.10.

Slika 5.10. Izgled početne stranice *Home*



Izvor: autorovo djelo

Widget HomepageList sastoji se od *widgeta SingleChildScrollView*, u kojem se nalazi *widget Row*, koji prikazuje više elemenata u jednom redu. Ti su elementi generirani prema listi filmova definiranih modelom *Movie* iz datoteke *movie.dart* i, kao parametar, predani su *widgetu HomepageList*, a zatim proslijeđeni kao parametar *widgeta MovieCard*. *Widget MovieCard* se rabi na početnom zaslonu *Home* i na zaslonu *Movies*, koji prikazuje listu filmova. *MovieCard widget* se koristi *widgetom GestureDetector* kako bi, kod dugog dodira kartice filma, iskočio prozor *widgeta PopupDialog*, koji omogućuje brzo i jednostavno ocjenjivanje filmova.

Navigacija na glavnom zaslonu aplikacije koristi se *widgetom BottomNavigationBar*, s ikonama i tekstom koji opisuju glavne komponente aplikacije: *Home, Movies, Search* i *Account*. Taj se *widget* nalazi u *home.dart* datoteci unutar *MyHomePage widgeta* i omogućuje prebacivanje zaslona pritiskom na jednu od ikona, čime se definira lista

widgeta odgovarajućih prikaza čiji redoslijed odgovara redoslijedu ikona i zatim se, pritiskom na ikonu, *widget* prikaže na traženoj poziciji unutar liste.

Tema same aplikacije, osim što je definirana eksplicitnim oblikovanjem *widgeta*, definirana je i unutar datoteke *main.dart*, kao *MaterialApp widget* s dodatnim postavkama glavnih boja aplikacije. Današnje verzije platformi podržavaju i dvije sheme glavnih boja aplikacije, takozvanu svijetlu (eng. *light)* i tamnu (eng. *dark)* temu. Ako nije eksplicitno određeno, aplikacija će se koristiti *light* temom, čiji su grafički elementi svijetlih boja. *Dark* ili *light* teme mogu se odrediti i prema postavkama platforme, tako da kod promjene teme grafičkog sučelja platforme i aplikacija promijeni temu. U Flutteru je to vrlo jednostavno implementirati, pod parametrima *theme* i *darkTheme widgeta MaterialApp* teme se definiraju klasom *ThemeData*, a parametru *themeMode* predaje se vrijednost varijable *ThemeMode.system*, koja sadrži trenutnu temu platforme. Slika 5.11 sadrži primjer implementacije teme aplikacije.

Slika 5.11. Primjer implementacije teme aplikacije u Flutteru

```
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'MovieLens',
      routes: {
        '/': (context) => const Landing(),
        '/login': (context) => const Login(),
        '/home': (context) => const MyHomePage(
              title: 'MovieLens',
              key: null,
            ),  // MyHomePage
      },
      theme: ThemeData(
        primarySwatch: Colors.deepOrange,
      ),  // ThemeData
      darkTheme: ThemeData(
        brightness: Brightness.dark,
        primarySwatch: Colors.deepOrange,
      ),  // ThemeData
      themeMode: ThemeMode.system,
    );  // MaterialApp
  }
}
```

Izvor: autorovo djelo

Iz primjera je vidljivo korištenje ranije spomenutog *widgeta*, *MaterialApp* - u kojem su definirane glavne rute aplikacije. Rute su definirane kao *key-value* parovi, čije su vrijednosti *widgeti*. Parametar *PrimarySwatch* klase *ThemeData* predstavlja glavnu boju aplikacije koja se upotrebljava u gumbima i ikonama. Cijeli se *MaterialApp widget* nalazi unutar *MyApp widgeta*, kako bi se, jednom definiran, mogao primjenjivati u drugom dijelu kôda aplikacije.

6. ZAKLJUČCI I PREPORUKE

Razvojni okvir za mobilne aplikacije Flutter se u svojih nepunih pet godina postojanja istaknuo kao jedan od znatnijih iskoraka u razvoju *cross-platform* mobilnih aplikacija. Taj uspjeh se može objasniti inovativnim pristupom komunikacije s programskim sučeljem platforme koja omogućuje efikasnije izvođenje aplikacija, ali i jednostavnošću razvoja grafičkog sučelja *widgetima,* koji imitiraju izgled i funkcionalnost grafičkih komponenti iOS i Android platformi. Programski jezik Dart je jedna od prednosti Fluttera zbog JIT i AOT načina izvođenja, koji omogućuju jednostavno testiranje i efikasno izvođenje aplikacije, ali ujedno i nedostatak, jer je Dart jedan od manje poznatih programskih jezika, što bi se s vremenom moglo promijeniti uzevši u obzir popularnost Fluttera.

Razvoj nativnih aplikacija će i dalje ostati najbolja opcija za razvoj mobilnih aplikacija jedne platforme, ali potrebno vrijeme, znanje i iskustvo za razvoj nativnih aplikacija za više platformi, kao i problematičnost održavanja više različitih aplikacija, predstavlja ozbiljnu prepreku za razvoj aplikacija na više platformi. Zbog relativno neznatno lošijih performansi aplikacija i goleme uštede vremena za razvoj mobilnih aplikacija za više platformi korištenjem *cross-platform* razvojnih okvira, kao i uniformnosti izgleda grafičkog sučelja aplikacija na različitim platformama - razvojni okviri poput Fluttera pokazali su se kao bolje rješenje za razvoj manje složenih aplikacija za više platformi.

Koliko je jednostavno razviti mobilnu aplikaciju razvojnim okvirom Flutter, svjedoči i cijeli proces izrade mobilne aplikacije MovieLens, za potrebe ovog rada u kojem je, bez prethodnog iskustva u razvoju mobilnih aplikacija i poznavanja razvojnog okvira Flutter, u relativno kratkom razdoblju kreirana funkcionalna mobilna aplikacija. Preporuke za korištenje Flutter razvojnog okvira za izradu mobilnih aplikacija, uključuju upoznavanje dokumentacije razvojnih okvira Flutter i Dart, kroz praćenje primjera razvoja jednostavnih aplikacija, pretraživanje Flutter i Dart paketa s gotovim funkcionalnostima koje se žele implementirati u aplikaciju i kroz isprobavanje funkcionalnosti ponuđenih *widgeta* korištenjem *hot-reload* značajke, koja omogućuje gotovo trenutnu izmjenu aplikacije, uz zadržavanje stanja.

LITERATURA

Knjige

1. Wargo, J. M. (2012). Introduction to PhoneGap. U: B. Curtis (ur.), *PhoneGap Essentials: Building Cross-Platform Mobile Apps* (3-22). Crawfordsville, Indiana, Sjedinjene Američke Države: Pearson Education, Inc.
2. Windmill, E. (2020). *Flutter in Action*. Sjedinjene Američke Države, Manning Publications Co.

Članci

1. Mahendra, M., Anggorojati, B. (2020). Evaluating the performance of Android based Cross-Platform App Development Frameworks*. 2020 the 6th International Conference on Communication and Information Processing*, 2020, 32-37.
2. Wolfgang, P. (1994). Meta patterns — A means for capturing the essentials of reusable object-oriented design. *Object-Oriented Programming - ECOOP 1994.: Lecture Notes in Computer Science, vol 821.* 150-162.

Internetski izvori

1. Apple (6.3.2008.). Apple Announces iPhone 2.0 Software Beta. *Apple Newsroom*. Preuzeto s: https://www.apple.com/newsroom/2008/03/06Apple-Announces-iPhone-2-0-Software-Beta/ (5. 8. 2022.)
2. Apple (14. 7. 2008.). iPhone App Store Downloads Top 10 Million in First Weekend. *Apple Newsroom*. Preuzeto s: https://www.apple.com/newsroom/2008/07/14iPhone-App-Store-Downloads-Top-10-Million-in-First-Weekend/ (5. 8. 2022.)
3. Capitol Technology University (15.12.2021.). A Brief History of Mobile Apps. *Capitology Blog*. Preuzeto s: https://www.captechu.edu/blog/brief-history-of-mobile-apps (5. 8. 2022.)
4. Eom, J. (15. 10. 2021.). Announcing Apache Cordova Retirement in App Center. *Microsoft Developer Blogs*. Preuzeto s:

https://devblogs.microsoft.com/appcenter/announcing-apache-cordova-retirement/ (6. 8. 2022.)

5. Flutter architectural overview. *flutter.dev*. Preuzeto s: https://docs.flutter.dev/resources/architectural-overview (11. 8. 2022.)
6. Garret, J. J. (18. 2. 2005.). Ajax: A New Approach to Web Applications. a*daptivepath.com*. Preuzeto s: https://web.archive.org/web/20080702075113/http://www.adaptivepath.com/ideas/essays/archives/000385.php (5. 8. 2022.)
7. Google Trends. Preuzeto s: https://trends.google.com/trends/explore?date=today%205-y&q=%2Fg%2F11f03_rzbg,%2Fg%2F11h03gfxy9,%2Fg%2F1q6l_n0n0,Xamarin,%2Fg%2F11c57wr2yt (12. 8. 2022.)
8. Ladd, S. (27.2.2018.). Announcing Flutter beta 1: Build beautiful native apps. *Medium*. Preuzeto s: https://medium.com/flutter/announcing-flutter-beta-1-build-beautiful-native-apps-dc142aea74c0 (11. 8. 2022.)
9. Leung, I. (26.2.2012.). BlackBerry Limited. *The Canadian Encyclopedia*. Preuzeto s: https://www.thecanadianencyclopedia.ca/en/article/blackberry-limited (5. 8. 2022.)
10. Ornbo, G. (16. 11. 2019.). The End of Native Apps. *shapeshed.com*. Preuzeto s: https://shapeshed.com/the-end-of-native-applications/ (5. 8. 2022.)
11. Statista (2022). Cross-platform mobile frameworks used by software developers worldwide from 2019 to 2021. *statista.com*. Preuzeto s: https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/ (12. 8. 2022.)
12. Statista (2022). Mobile operating systems' market share worldwide from January 2012 to January 2022. *statista.com*. Preuzeto s: https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (11. 8. 2022.)

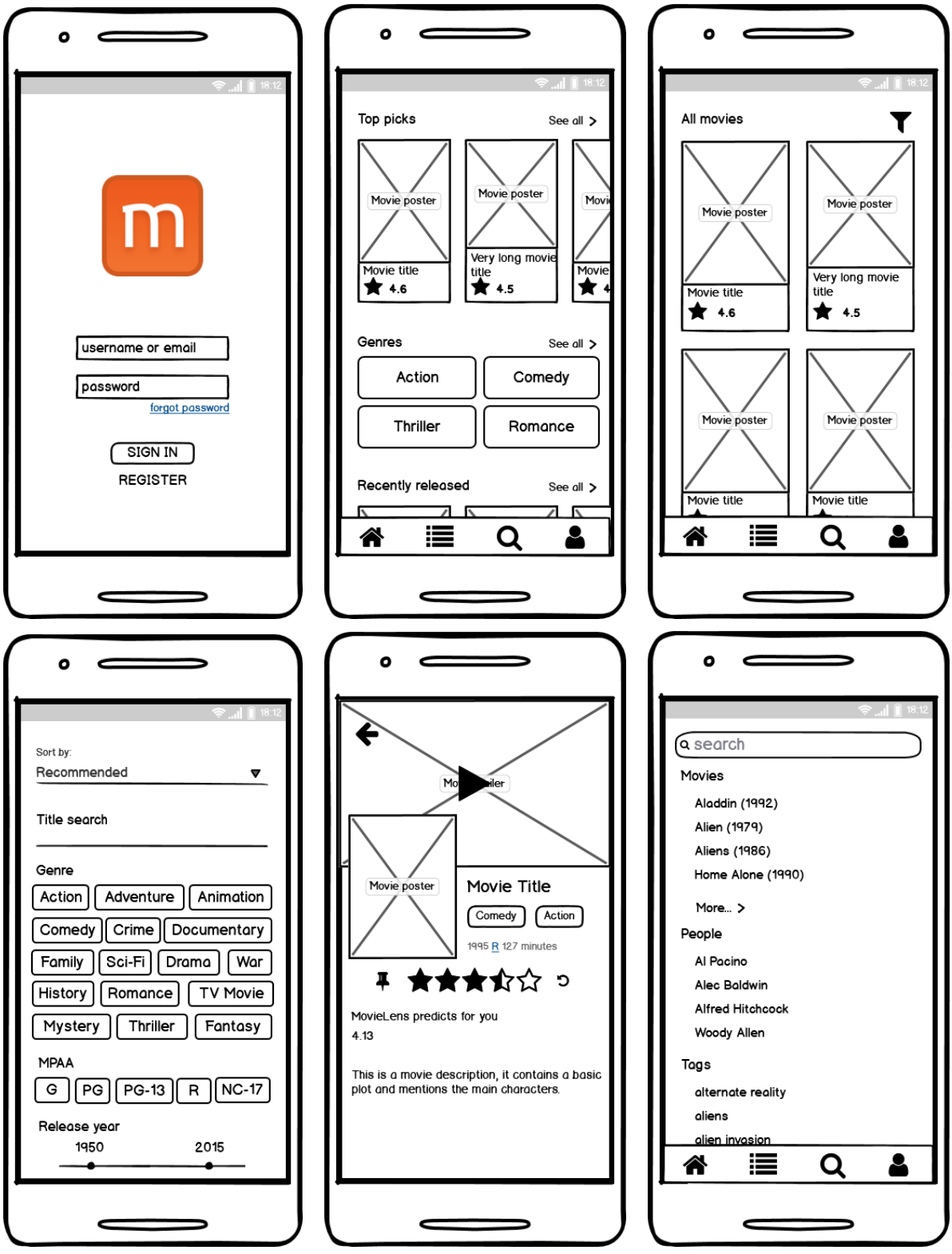
Ostalo

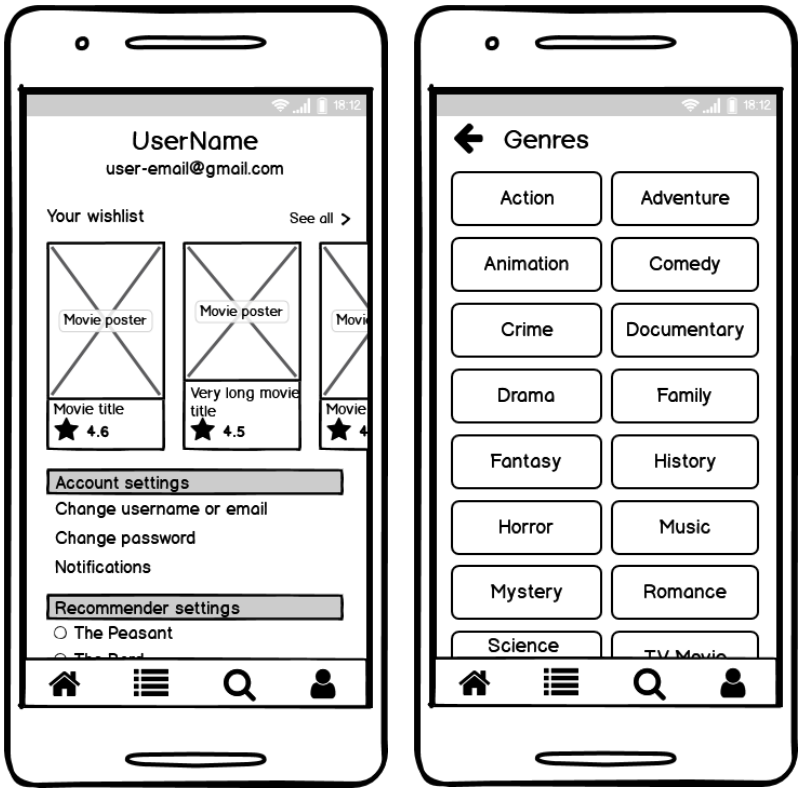
1. Gonsalves, M. (2018). *Evaluating the mobile development frameworks Apache Cordova and Flutter and their impact on the development process and application characteristics* (diplomski rad). California State University, Chico
2. Haider, A. (2021). *Evaluation of cross-platform technology Flutter from the user's perspective* (diplomski rad). KTH, School of Electrical Engineering and Computer Science, Stockholm
3. Pountain, D. (11.1984.). A Plethora of Portables: Apricots and the Organiser. Byte Magazine, Volume 9 Number 12 - New Chips, 413-420.

POPIS SLIKA I TABLICA

PRILOZI

Prilog 1: Predlošci dizajna grafičkog sučelja aplikacije




UserName
user-email@gmail.com
Your wishlist
See all >
Movie poster
Movie title
4.6
Movie poster
Very long movie title
4.5
Account settings
Change username or email
Change password
Notifications
Recommender settings
The Peasant
Genres
Action
Adventure
Animation
Comedy
Crime
Documentary
Drama
Family
Fantasy
History
Horror
Music
Mystery
Romance
Science

SVEUČILIŠTE VERN'

IZJAVA

kojom izjavljujem da sam završni rad pod naslovom

Izrada mobilnih aplikacija korištenjem Flutter/Dart razvojnog okvira na primjeru aplikacije MovieLens,

izradio/la samostalno. Svi dijelovi rada, nalazi ili ideje koje su u radu citirane ili se temelje na drugim izvorima, bilo da su u pitanju knjige, znanstveni ili stručni članci, internetske stranice, propisi i sl. u radu su jasno označeni kao takvi te adekvatno navedeni u popisu literature.

Zagreb, 13. 09. 2022.

MISLAV KAPULICA
(ime, prezime)

M. Kapulica

(potpis)