УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ


Милица Гаљак

# АУТОМАТСКО ТЕСТИРАЊЕ МОБИЛНИХ АПЛИКАЦИЈА

мастер рад

Београд, 2023.

**Ментор**:

др Милена Вујошевић Јаничић, ванредни професор
Универзитет у Београду, Математички факултет

**Чланови комисије**:

др Весна Маринковић, доцент
Универзитет у Београду, Математички факултет

др Иван Чукић, доцент
Универзитет у Београду, Математички факултет

**Датум одбране**: септембар 2023.

*Захваљујем се менторки на подршци, разумевању и саветима, Филипу на мотивацији и породици која је увек била уз мене.*

*Овај рад посвећујем сестрићу Александру Заварку.*

**Наслов мастер рада**: Аутоматско тестирање мобилних апликација

**Резиме**: Рад сагледава процес развоја и аутоматског тестирања мобилних апликација, показује како аутоматски тестови убрзавају развојни циклус, побољшавају квалитет кода и осигуравају стабилност апликација. У оквиру рада имплементирана је апликација за помоћ при организацији догађаја и мониторинг званица, *Duma*, помоћу развојног оквира *Flutter*, где се илуструје писање тестова и примена библиотеке *Mockito*, као и коришћење услуга за складиштење података и аутентификацију платформе *Firebase*. Овај рад наглашава да развојни оквир *Flutter* пружа окружење за брз развој и тестирање, а да библиотека *Mockito* олакшава изолацију и проверу јединица кода.

**Кључне речи**: мобилна апликација, развојни оквир *Flutter*, аутоматско тестирање, тестирање јединица, библиотека *Mockito*, платформа *Firebase*, покривеност кода

# Садржај

# Глава 1

# Увод

У данашњем дигиталном добу, мобилне апликације су постале кључни фактор наше свакодневице, пружајући нам разноврсне услуге и информације. Како би се осигурало да ове апликације задовољавају висок ниво квалитета и функционалности, тестирање постаје неизоставан корак у процесу њиховог развоја. Аутоматско тестирање се све више намеће као кључна пракса која омогућава развојним тимовима да брже идентификују грешке, осигурају стабилност и побољшају корисничко искуство.

Кроз овај мастер рад је приказан концепт аутоматског тестирања мобилних апликација, истражујући примену технологије *Flutter*, библиотеке *Mockito* и платформе *Firebase*. Анализом различитих аспеката, доприноси се разумевању како ове технологије играју кључну улогу у изградњи и осигуравању висококвалитетних мобилних апликација. Поред практичних примера и конкретне примене технологија за развој мобилних апликација, обрађени су теоријски делови који описују различите технологије за развој мобилних апликација, врсте, нивое и технике тестирања софтвера као и специфичности за тестирање мобилних апликација.

У наставку је дат приказ тезе по поглављима. Поглавље 2 описује разноврсност мобилних апликација и технологије које се користе за њихов развој. Нативне и хибридне апликације имају своје предности и недостатке, а технологије за развој хибридних апликација као што је *Flutter* постају све популарније као средство за развој висококвалитетних апликација за више платформи где се постиже ефекат брзог и квалитетног развоја софтвера. Поглавље 3 продубљује разумевање тестирања софтвера. У фокусу су статичка и динамичка верификација софтвера, као и карактеристике и изазови при

тестирању мобилних апликација. Поглавље 4 пружа опис апликације *Duma*, анализира се структура апликације, случајеви употребе и технологије које су коришћене током њеног развоја. Апликација *Duma* помаже у ораганизацији догађаја и има функционалности за штампање акредитација званица догађаја, као и све неопходне податке везане за протекле, текуће и будуће догађаје. Поред манипулације података за догађаје, могуће је видети актуелну статистику догађаја и податке пријављеног налога. Поглавље 5 се бави тестирањем апликације *Duma* кроз призму аутоматских тестова. Користећи тестове јединица, приказано је како библиотека *Mockito* олакшава процес тестирања, омогућавајући прецизно испитивање функционалности апликације и идентификацију потенцијалних проблема. У поглављу 6 је изведен закључак рада и могући кораци за даље унапређење тестирања апликације *Duma* као и саме апликације.

# Глава 2

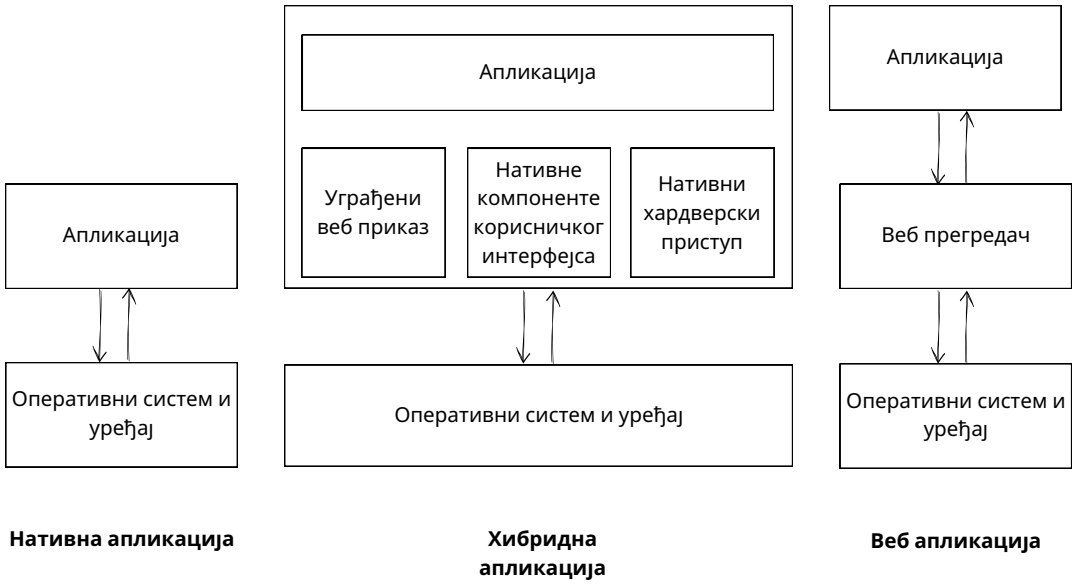
# Технологије за развој мобилних апликација

Мобилна апликација је програм који је дизајниран да ради на мобилном уређају као што је мобилни телефон, таблет или паметни сат. За креирање мобилних апликација користе се разне библиотеке, развојни оквири, алати и компоненте. Експанзија употребе мобилних уређаја, а посебно паметних телефона, довела је до повећане потражње и потребе за развојем софтвера за мобилне уређаје. Како се повећавала потражња за мобилним уређајима, тако се повећавао број различитих врста и модела мобилних уређаја на тржишту. Различитост уређаја је приморала програмере да приликом развоја апликација посматрају различите аспекте и карактеристике уређаја као што су: једноставност апликације, ограничења уређаја (трајање батерије, величина екрана, различите резолуције, брзина процесора, количина меморије итд.), могућност мрежне инфраструктуре, а уз то и политику сигурности и приватности података корисника... Тежња ка томе да се омогући што већем броју корисника да неометано користе апликацију довела је до тога да се развој апликација окрене ка специфичностима самих уређаја и платформи.

## 2.1 Врсте мобилних апликација

Мобилне апликације можемо поделити на **веб апликације** (енг. *web applications*), **нативне** (енг. *native mobile applications*) и **хибридне** (енг. *cross platform applications*) [8, 11]. На слици 2.1 је илустрована укратко објашњена архитектура различитих типова апликација, где се може видети на који на-

чин оперативни систем уређаја комуницира са мобилном апликацијом, да ли директно (код нативних апликација), преко веб технологија за кориснички интерфејс (код хибридних апликација) или путем веб прегледача (код веб апликација). У наставку текста су детаљно објашњене предности и мане сваког од ових приступа.




Слика 2.1: Архитектура нативне, хибридне и веб апликације [8]

## Веб апликације

Веб апликације су развијене помоћу веб технологија као што су *HTML*, *HTML5*, *JavaScript* и *CSS*. За апликације овог типа није потребна инсталација на уређају, већ корисник добија приступ апликацији кроз веб претраживач уређаја користећи *URL* апликације. Предности традиционалних веб апликација су:

- једноставан и брз развој аликације коришћењем веб технологија,

- логика апликације је смештена на серверу, док се мобилном уређају шаље само кориснички интерфејс за рендеровање,
- одржавање апликације је једноставно јер се измене врше само на серверу,
- развијена је јединствена апликација која може да се извршава на различитим платформама.

Неке од мана оваквих апликација су:

- непостојање апликације у продавници апликација (енг. *App store*) на корисниковом уређају,
- интернет је неопходан за извршавање апликације,
- апликација није у могућности да приступи софтверу и хардверу уређаја,
- лоше перформансе због технологија које се интерпретирају и парсирају кроз веб претраживач,
- не прате изглед и коришћење апликације карактеристичне за специфичну платформу мобилног уређаја.

## Нативне апликације

Нативне апликације су развијене помоћу алата и програмских језика који су намењени за употребу на одређеној мобилној платформи. Апликација може да се извршава само на платформи за коју је намењена. Уколико је нативна апликација објављена у продавници апликација на уређају, апликација се одатле може преузети и инсталирати. Главне предности овог типа апликација су:

- могућност приступа софтверу и хардверу уређаја као што су камера, *GPS*, сензори, приступ интернету, меморији уређаја, *SMS*, *email*...,
- могућност да се максимално искористи хардвер и пруже најбоље перформансе софтвера,
- кориснички интерфејс који је у складу са принципима карактеристичним за платформу.

Главна мана оваквог приступа јесте отежан развој апликације због потребе за развојем различитих апликација за различите платформе што изискује додатно време, знање и новац.

### Хибридне апликације

Хибридне апликације су намењене за извршавање на различитим оперативним системима и уређајима и представљају комбинацију веб и нативних апликација. Иако имају неке од карактеристика нативних апликација, као што је приступ софтверу и хардверу уређаја, хибридне апликације се у основи извршавају помоћу веб технологија, као што је *Webkit*, који представља додатни слој између апликације и корисника [20]. Предности хибридних апликација су:

- заузимају мало меморијског простора, ако су подаци смештени на серверу,
- апликација може да приступи софтверу и хардверу уређаја,
- развија се јединствена апликација за различите платформе,
- време развоја је краће него код нативних апликација, а самим тим је и развој јефтинији.

Неке од мана хибридних апликација су:

- постојање додатног слоја који се налази између апликације и оперативног система узрокује лошије перформансе од нативних апликација које немају додатни слој,
- кориснички интерфејс треба да се прилагођава стилу различитих платформи, што често доводи до редудантности у коду.

## 2.2 Технологије за развој нативних и хибридних мобилних апликација

Најпопуларнији оперативни системи мобилних уређаја су *Android* [4] и *iOS* [5].

Најпопуларнији програмски језици за развој **нативних** апликација за оперативни систем *Android* су *Java* [1] и *Kotlin* [16], док се за развој **нативних** апликација за оперативни систем *iOS* највише користе *Swift* [26] и *Objective-C* [22]. Најкоришћеније технологије за развој **хибридних** апликација за мобилне уређаје јесу *React Native* [24], *Flutter* [12] и *Xamarin* [32]. У наставку следе најистакнутије карактеристике за сваки од споменутих језика и одговарајућих технологија.

## *Java* и *Kotlin*

*Java* је широко коришћен објектно-орјентисани програмски језик који своју синтаксу и принципе темељи на програмским језицима *C* и *C++*. Основна предност овог програмског језика је портабилност, односно извршавање програма на било којој машини која подржава *Java* виртуелну машину (енг. *Java virtual machine*), интерпретер, који служи за учитавање, верификацију и извршавање кода написаног у програмском језику *Java*. Особине које су такође допринеле да програмски језик *Java* буде годинама уназад један од најпопуларнијих програмских језика, и то не само у развоју мобилних апликација, јесу безбедност, робусност, дистрибуираност, високе перформансе при извршавању и вишенитност [15].

Цео скуп функционалности оперативног система *Android* је доступан преко интерфејса који је направљен помоћу програмског језика *Java*. Интерфејсе чине градивни елементи који се користе за прављење *Android* апликација тако да поједностављују поновну употребу основних, модуларних системских компоненти и сервиса, који укључују следеће:

- Богат и проширив скуп елемената за приказ на екрану који се користе за развој корисничког интерфејса, као што су листе, оквири за текст, дугмад, веб прегледачи који се могу уградити у апликацију итд.
- Менаџер ресурса који пружа приступ ресурсима који немају директне везе са програмским језиком *Java*, као што су локализовани стрингови и фајлови за организацију и графику апликације.
- Менаџер обавештења који омогућава свим апликацијама да прикажу прилагођена обавештења у статусном бару мобилног уређаја.

- Менаџер активности који управља животним циклусом апликација и обезбеђује уобичајену навигацију ка претходном екрану у апликацији (енг. *back stack*).
- Приступ подацима из других апликација као што су *Contacts*, или да дељење сопствених података.
- Потпуни приступ истом развојном оквиру са интерфејсима који користе *Android* системске апликације [2].

*Kotlin* је статички типизиран програмски језик који подржава и објектно-орјентисано и функционално програмирање. *Kotlin* је настао као замена за програмски језик *Java* и извршава се на *Java* виртуелној машини. Стога, могуће је додавање *Java* библиотека у код, односно комбиновање оба програмска језика у истом пројекту. Будући да је *Kotlin* релативно нов програмски језик, објављен 2016. године, нема толико велику заједницу и подршку као програмски језик *Java*. С друге стране, *Kotlin* је за кратко време постао изузетно популаран због добрих перформанси, стабилности, модерног корисничког интерфејса и једноставније синтаксе програмског језика што омогућава бржи развој и лакше одржавање кода [6, 18]. Сада је беспрекорно интегрисан у *Android studio* и активно се користи од стране многих компанија за развој *Android* апликација. Коришћење *Kotlin*-а за развој *Android* мобилних апликација доноси многе бенефите у развоју система. *Kotlin* компајлер детектује велики број пропуста, чинећи га сигурним. *Kotlin* има одличну подршку и много доприноса од стране заједнице која расте широм света, па је због тога, и због једноставне синтаксе, једноставан за учење, посебно за *Java* програмере. *Kotlin* се пре свега користи за развој *Android* апликација, али поседује и подршку за развој вишеплатформских апликација где се *Kotlin* може користити не само за развој *Android*, већ и *iOS*, *backend* и веб апликација. Екстензије *KTX* додају програмском језику *Kotlin* језичке функционалности, као што су екстензионе функције, ламбда изразе и именоване параметре, постојећим *Android* библиотекама [17].

### *Swift* и *Objective-C*

*Objective-C* је програмски језик који се првенствено користи за развој апликација које се извршавају на платформи *iOS*. Пре објављивања програмског

језика *Swift*, био је годинама стандард за развој мобилних апликација за *iOS* уређаје. *Objective-C* је надградња програмског језика *C* и да додатно пружа објектно-орјентисани концепт и динамичко извршавање кода [3].

*Swift* је општенаменски, компајлирани програмски језик који је развијен од стране компаније *Apple Inc.* Први пут је објављен 2014. године, као замена за програмски језик *Objective-C*, јер је *Objective-C* остао непромењен од раних 1980-их и недостајале су му модерне језичке функционалности. *Swift* користи *Cocoa* и *Cocoa Touch* развојне оквире, а кључни аспект дизајна овог програмског језика је могућност интероперабилности са огромним бројем постојећег *Objective-C* кода који је развијен за *Apple* производе током претходних деценија. На *Apple* платформама се користи *Objective-C* библиотека за извршавање кода која омогућава извршавање кода написаног помоћу програмских језика *C*, *Objective-C*, *C++* и *Swift* унутар једног програма [25, 28].

### *React Native* и *Xamarin*

*React Native* је развојни оквир који је креирала компанија *Meta Platforms* и користи се за развој апликација за платформе *Android*, *Android TV*, *iOS*, *macOS*, *tvOS*, *Windows* и *UWP*, омогућавајући програмерима да користе *React Native* развојни оквир заједно са могућностима одређене платформе. Користи се за развој *Android* и *iOS* апликација у компанијама *Facebook*, *Microsoft* и *Shopify*. Такође се користи за развој апликација за виртуелну стварност у компанији *Oculus* [24, 29].

*Xamarin* је развојни оквир за развој модерних апликација са високим перформансама за платформе *iOS*, *Android* и *Windows*. *Xamarin* користи програмски језик *C#* и интегрише се са развојним окружењем *Visual Studio*, *Microsoft*-овим интегрисаним развојним окружењем за *.NET* платформу које омогућава развој за *iOS* и *Windows*. *Xamarin* је такође објавио продавницу компоненти (енг. *component store*) која омогућава интеграцију библиотека и контролера корисничког интерфејса директно у мобилне апликације. Неке познате компаније, као што су *3M*, *AT&T* и *HP*, су користиле ову платформу за израду својих апликација [32, 30].

## *Flutter* и *Dart*

*Flutter* је развојни оквир, који користи програмски језик *Dart* за развој мобилних апликација високих перформанси. *Dart* је строго типизиран, објектно-орјентисани програмски језик који је направила компанија *Google*. *Dart* може да се транспилира у програмски језик *JavaScript*, подржава *JIT* (енг. *Just-in-Time*) компилацију као и компилацију у машински кôд архитектура *ARM* і *x86-64*. Синтакса програмског језика *Dart* је развијана по узору на програмски језик *C*.

Једна од основних предности *Flutter*-а је могућност компилације програмског језика *Dart*, што му даје боље перформансе од осталих технологија које се користе за хибридни приступ развоја мобилних апликација. Такође, додатни разлог због којег се *Flutter* истиче у хибридним технологијама је постојање брзог поновног учитавања промена (енг. *hot reload*), што омогућава брз и олакшан развој [12, 31].

Развојни оквир *Flutter* пружа разне функционалности, а неке од њих су: изградња корисничког интерфејса, интеграција са различитим интерфејсима и веб сервисима, приступ хардверским функционалностима попут камере, сензора покрета, *GPS*-а, микрофона и сл. *Flutter* омогућава приступ платформским сервисима као што су локација уређаја, календар, обавештења, база података. Користе се пакети попут *geolocator* за приступ локацији или *firebase_messaging* за управљање обавештењима. *Flutter* пружа алате за једноставно писање и извођење тестова како би се проверила исправност апликације. Пружа могућност израде тестова јединица, интеграционих тестова и тестова за кориснички интерфејс. У апликацији *Duma* (поглавље 4) коришћене су функционалности за изградњу корисничког интерфејса, приступ платформским сервисима и тестирање.

Програмски језик *Dart* има уграђену подршку за сигурност од недостајућих вредности (енг. *sound null safety*). То значи да вредности променљивих не могу бити недостајуће осим ако се изричито наведе да могу бити. Захваљујући подршци за сигурност од недостајућих вредности *Dart* има функционалност да путем статичке анализе кода спречи настајање изузетака (енг. *exceptions*) због недостајућих вредности променљивих током извршавања кода. Без обзира на то на којем се оперативном систему извршава програм или на који начин се компајлира код, извршавање кода захтева извршну платформу *Dart* (енг. *Dart runtime*). Ова платформа је одговорна за следеће кључне задатке:

- управљање меморијом — *Dart* користи модел за управљање меморијом где неискоришћена меморија бива ослобођена помоћу сакупљача отпадака (енг. *garbage collector*),
- динамичке провере — спроводи динамичке провере типова помоћу оператора за проверу типа (енг. *is*) и оператора за кастовање (енг. *as*),
- управљање *изолатима* (енг. *isolates*) који су независни токови слични нитима, али немају заједничку меморију, већ комуницирају међусобно само путем порука — Извршна платформа Dart контролише главни изолат (енг. *the main isolate*), где се код извршава, и било који други изолат који креира апликација [9].

Програмски језик *Dart* подржава богат скуп библиотека које се могу лако увести у пројекат, а неке од њих су:

- *dart:core* — уграђени типови, колекције и друге основне функције за сваки програм написан помоћу програмског језика *Dart*,
- *dart:collection* — богатији типови колекција као што су редови, повезане листе, хеш мапе и бинарна стабла,
- *dart:convert* — кодери и декодери за конверзију између различитих приказа података, укључујући и формате *JSON* и *UTF-8*,
- *dart:math* — математичке константе и функције, и генерисање случајних бројева,
- *dart:io* — подршка за управљање фајловима, сокетима и *HTTP* захтевима,
- *dart:async* — подршка за асинхроно програмирање, са класама као што су *Future* и *Stream*,
- *dart:typed_data* — листе које ефикасно манипулишу подацима фиксне величине (на пример, *unsigned int*) и нумеричким типовима *SIMD* (енг. *Single Instruction and Multiple Data Stream*),
- *dart:ffi* — интерфејси функција за које не постоји приступ коду за интероперабилност са другим кодом који представља интерфејс у стилу програмског језика *C*,

- *dart:isolate* — конкурентно програмирање користећи изолате,
- *dart:html* — елементи *HTML*-а и други ресурси за веб апликације које треба да остваре интеракцију са веб прегледачем и *DOM* (енг. *Document Object Model*) структуром стабла за хијерархијски приказ веб странице [10].

# Глава 3

# Тестирање софтвера

Тестирање софтвера је процес евалуације софтвера који проверава да ли се софтвер понаша у складу са планираним захтевима. Тестирање обухвата проверу различитих аспеката софтвера укључујући и безбедност, поузданост и тачност софтвера. Софтвер је квалитетнији уколико садржи што је мање могуће пропуста, а тестирање омогућава да се открију пропусти и самим тим унапреди софтвер. Стога, тестирање представља једну од најважнијих фаза у развојном циклусу софтвера.

У оквиру тестирања, од великог значаја за побољшање квалитета софтвера (енг. *software quality*) су верификација и валидација софтвера. Верификација је процес који брине о томе да се софтвер развија исправно, а валидација је процес који брине о томе да се развија софтвер са захтеваним функционалностима. Верификација је процес који се састоји од активности које осигуравају тачност специфичних функционалности софтвера. Обично је обавља развојни тим, док валидација обухвата и екстерне процесе који укључују активности које гарантују да је софтвер у складу са клијентовим захтевима. Верификација обично почиње пре валидације, а затим се паралелно одвијају до пуштања софтвера у продукцију. Постоје различите фазе тестирања, описане у наставку, за које важи правило да је откривање и исправка пропуста у ранијим фазама тестирања увек јефтинија него у каснијим [27].

Постоје два главна приступа у верификацији софтвера: **статички** и **динамички**. **Статички приступ** у верификацији се одвија без извршавања програма, док се **динамички приступ** обавља током извршавања програма са реалним улазним подацима.

## 3.1 Статички приступ

Статички приступ у верификацији софтвера захтева приступ изворном коду и бави се анализом кода, обрадом изузетака, провером захтеваних функционалности и стандардима у писању кода. Најпознатије технике тестирања статичким приступом јесу **формални** и **неформални** прегледи кода [34] .

**Формални прегледи** (енг. *formal inspections*) обухватају групне састанке на којима се дискутује о коду и раде прегледи. Овакав вид тестирања је веома скуп и временски захтеван, све мање се користи јер иако се на овај начин може пронаћи највећи број дефеката у коду, ово захтева превише времена и ангажовања, а већина фирми то не може да приушти [34] .

Основне **врсте неформалног** тестирања су [34] :

- преглед преко рамена,
- преглед преко мејла,
- преглед преко алата за преглед кода,
- програмирање у пару.

**Преглед преко рамена** [34] је најчешћи и најнеформалнији вид прегледа и обавља се тако што програмер објашњава прегледачу шта је написано у коду и због чега. Мана овакве технике је што није могуће испратити шта је прегледано, а шта није, могуће је пропустити неку измену иако је примећено да треба да се уради, и иако се констатују неки дефекти и ако се спроведе акција да се ти дефекти исправе, могуће је да се то уради погрешно или да се уведу нови дефекти.

**Преглед преко мејла** [34] се обавља тако што измена кода стигне мејлом прегледачу, углавном пре него што код уђе у репозиторијум или се аутоматски пошаље након што код уђе у репозиторијум. Оваква техника је превазиђена алатима који решавају проблеме који су настајали техником прегледа преко мејла.

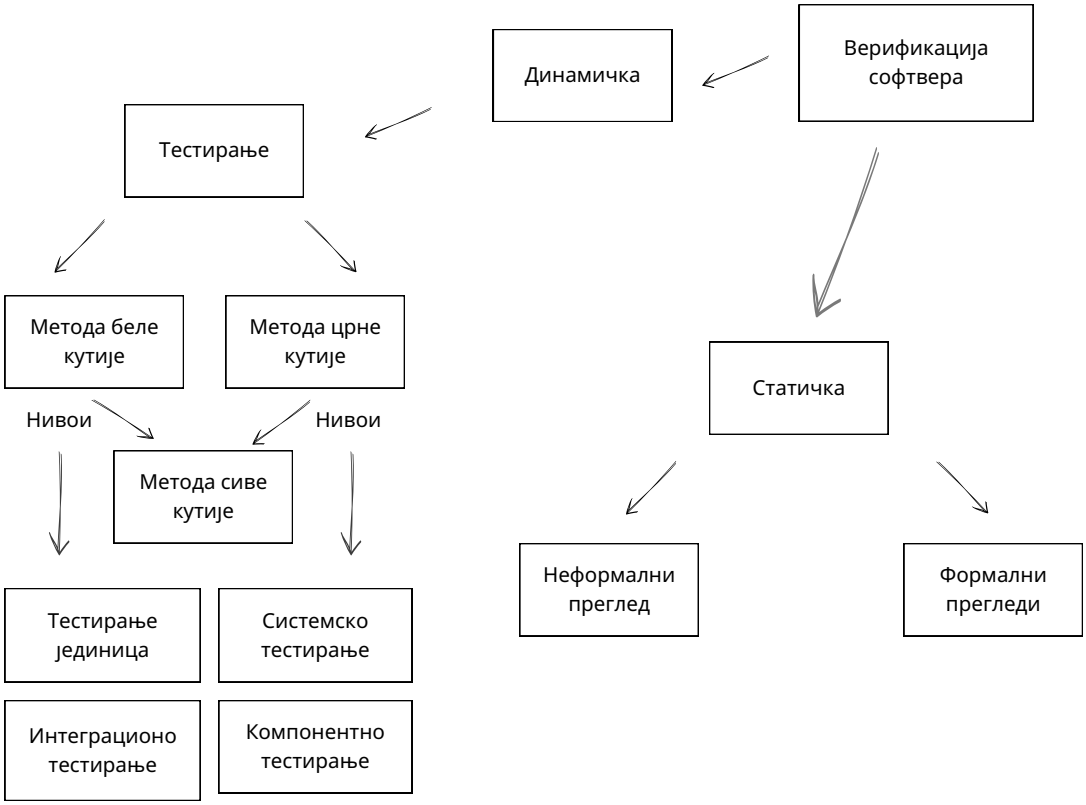
**Преглед преко алата за преглед кода** [34] се обавља помоћу алата као што су *Phabricator* [23], *Gerrit* [13], *GitLab* [14] и обично се обавља од стране једног или више искуснијих програмера пре него што код уђе у репозиторијум.

**Програмирање у пару** [34] је техника која води ка квалитетнијем коду, али некада програмери који раде заједно имају исти принцип размишљања и

самим тим заједно превиђају грешке па су због тога екстерни прегледи ипак неопходни.

## 3.2 Динамички приступ

За динамичку верификацију софтвера постоје три главне технике тестирања: методе **црне кутије** (енг. *black-box*), методе **беле кутије** (енг. *white-box testing*) и методе **сиве кутије** (енг. *grey-box testing*). Све три технике се изводе у различитим нивоима тестирања и све три обухватају неколико типова тестирања. Постоје четири нивоа тестирања: **тестирање јединица** (енг. *unit testing*), **компонентно** тестирање (енг. *component testing*), **интеграционо** тестирање (енг. *integration testing*) и **системско** тестирање (енг. *system testing*). За сваки ниво постоје различити начини тестирања [27, 33]. На слици 3.1 је приказана шема која илуструје врсте верификације софтвера, нивое и технике тестирања.




Слика 3.1: Врсте верификације софтвера, нивои и технике тестирања [27, 33]

### Нивои тестирања

У складу са поделом према нивоима тестирања, где се могу тестирати појединачни модули, групе модула (везаних наменом, употребом, понашањем или структуром) или цео систем, разликујемо тестове јединица, компонентне тестове, интеграционе тестове и системске тестове [33].

**Тестирањем јединица** кода (енг. *unit testing*) [33] се проверава функционисање подпрограма, класа, мањих или већих целина које могу да се тестирају независно. На овом нивоу тестирања се детаљно проверавају и најмањи делови система, а циљ тестирања јединица јесте да утврди да ли тестиране целине имају предвиђену функционалност. Уколико тестирана јединица комуницира са мрежом, базом података, фајл системом или другим класама и модулима у оквиру система, такве компоненте које су ван јединице се апстрахују, јер је током тестирања јединица дозвољена само директна комуникација са меморијом. Како је на овом нивоу тестирања потребно изузетно познавање начина функционисања дела система који се тестира, тестове јединица најчешће пише програмер.

**Компонентно тестирање** (енг. *component testing*) [33] проверава исправност компоненте која представља скуп повезаних јединица кода које имају заједнички интерфејс према осталим компонентама. Компонентно тестирање се обавља након тестирања јединица, а поред провере функционалности компоненти се проверава и исправност међусобне комуникације компоненти. Током тестирања исправности компоненте, компонента се изолује од остатка система, а само тестирање се обавља након креирања саме компоненте.

**Интеграционо тестирање** (енг. *integration testing*) [33] проверава комуникацију између компоненти које представљају једну целину система. Проверава се да ли је њихова комуникација дефинисана и реализована по спецификацији пројекта. Током интеграционог тестирања се проналазе пропусти везани за међусобну комуникацију компоненти и проверава се исправност компоненти током заједничког функционисања. Овај ниво тестирања је сличан компонентном тестирању, с тим што код компонентног тестирања компонента представља мању целину система него код интеграционог тестирања.

**Системско тестирање** (енг. *system testing*) [33] проверава функционисање система као целине, односно да ли је систем у складу са спецификацијом. Приступ бази и хардверским деловима система у овом нивоу тестирања је неопходан. Код системског тестирања се проверава и функционални и не-

функционални аспект, где се из функционалног аспекта проверава исправност апликације у односу на захтеве из спецификације клијента, а из нефункционалног аспекта технички квалитет апликације. У системско тестирање се понекад убрајају и истраживачко тестирање (енг. *exploratory testing*) и тестирање прихватљивости (енг. *acceptance testing*), а некада се ове две врсте тестирања издвајају независно.

### Методе црне, беле и сиве кутије

**Метода црне кутије** (енг. *black box testing*) [33] је скуп техника тестирања код којих се тестови генеришу искључиво на основу спецификације софтвера. Други називи су и фукционално тестирање (енг. *functional testing*), тестирање понашања (енг. *behavioural testing*), тестирање вођено подацима (енг. *data driven testing*). Познавање интерне структуре кода није неопходно, стога овакво тестирање углавном обављају тестери. Задатак тестера је да систему пружи улазе, а затим да провери излазе у односу на дату спецификацију. Акценат ових техника јесте да се софтвер посматра из корисниковог угла и да се таквим приступом уочи што више направљених пропуста. Неке од техника црне кутије су класе еквиваленције (енг. *equivalence class testing*), метода граничних случајева (енг. *boundary value testing*), табеле одлучивања (енг. *decision table*), дијаграми стања (енг. *state-transition diagram*), табеле стања (енг. *state transition tables*) и погађање грешака (енг. *error guessing*).

**Метода беле кутије** (енг. *white box testing*) је скуп техника тестирања код којих се генеришу тест примери познајући интерну структуру кода и начин функционисања система. Углавном су програмери задужени за писање оваквих тестова. Други називи су и структурно тестирање (енг. *structural testing*), тестирање вођено логиком (енг. *logic driven testing*). Тестирањем методама беле кутије испитују се различите путање кроз програм. Тестирање беле кутије се најчешће користи за писање тестова јединица, али се такође користи за интеграционо и системско тестирање. Ова врста тестирања је скупа и спроводи се обично за системе код којих су грешке скупе.

**Метода сиве кутије** (енг. *gray box testing*) [33] је скуп техника тестирања са мешовитом стратегијом. Како се код ових техника софтвер посматра и из корисниковог угла и из угла програмера, постоји увид у унутрашњу структуру система, али не у тој мери као код техника беле кутије. Користи се код компонентног и интеграционог тестирања. Ово су технике коју користе

и програмери и тестери.

**О покривености кода**

Покривеност кода представља проценат линија кода или функционалности која је извршена током извођења тестова. Висока покривеност није само доказ да су функционалности тестиране, већ и основни индикатор квалитета софтвера. Ниска покривеност оставља простор за потенцијалне грешке које нису откривене тестовима. Тежња ка високој покривености кода узрокује да програмери боље разумеју интеракције између различитих делова кода и да примењују конзистентан начин писања кода прилагођен тестирању, што доводи до побољшане читљивости и одржавања кода.

Како се тестирање јединице кода фокусира на изоловано тестирање појединачних јединица софтвера, као што су функције, класе или модули, овакви тестови су брзи, специфични и откривају грешке на нивоу јединице, олакшавајући исправљање проблема и одржавање. Висока покривеност тестовима јединица осигурава да већина функционалности сваке јединице буде адекватно тестирана. Висока покривеност кода доноси низ позитивних утицаја на развој софтвера. Неке од њих су наведене у наставку секције.

**Поузданост** — Висока покривеност значи да су веће шансе да ће потенцијалне грешке бити откривене пре него што стигну до корисника.

**Лакше одржавање** — Ако се промене уводе у код, висока покривеност осигурава да тестови открију евентуалне негативне последице тих промена на остатак апликације.

**Бржи развој** — Испитивање засебних јединица помаже у откривању грешака раније, чиме се смањује време потребно за дебаговање комплексних проблема.

**Побољшана документација** — Тестови јединица служе као документација која илуструје како се очекује да компоненте функционишу.

## 3.3 О тестирању мобилних апликација

У контексту техника тестирања мобилних апликација разликујемо **структурне**, које су засноване на структури кода апликације, и **фукционалне**,

које су засноване на моделу апликације. Код структурних техника тестирања се узимају у обзир специфичности програмског језика за функционалности као што су управљање локацијом, осетљивост екрана на додир и потрошња енергије, па се на основу тих разматрања праве контролни графови, графови токова података, као и њихови критеријуми покривености. Код функционалних техника тестирања је неопходно да се прецизира сама апликација и окружење у коме апликација функционише. Посматрају се различита стања апликације као што је понашање апликације током укљученог авионског режима, током трајања позива или када је батерија празна. Моделирају се опсези вредности променљивих и посматра се утицај различитих вредности на апликацију која се тестира.

Код мобилних апликација је од суштинске важности **тестирање јединица**, док је интеграционо тестирање присутно али не у толикој мери. У оквиру технологија за развој нативних и хибридних апликација постоје алати и библиотеке које подржавају тестирање јединица. Поред аутоматизованих тестова јединица и интеграционих тестова, код мобилних апликација се примењују разни тестови који доприносе квалитету софтвера. У наставку су описани неки од њих [21].

**Тестови за проверу перформанси и поузданости** (енг. *performance and reliability testing*) — у великој мери зависе од квалитета самог уређаја и протока интернета у тренутку тестирања. Тестови се спроводе ради провере перформанси и понашања апликације под одређеним условима као што су слаба батерија, лоша покривеност мрежом, мала доступна меморија, истовремени приступ серверу апликације од стране више корисника итд.

**Анализа потрошње енергије и меморије** (енг. *memory and energy testing*) — утврђује се који део апликације највише оптерећује уређај, а након тога и побољшава коришћење енергије и простора на уређају при коришћењу апликације. За овакве анализе се користе профајлери које подржавају технологије за развој нативних и хибридних мобилних апликација.

**Тестирање сигурности** (енг. *security testing*) — од изузетне је важности за тестирање мобилних апликација. Прикупљају се подаци о кориснику

као што је локација уређаја, време коришћења апликације и подаци о мрежи на којој је накачен уређај.

**Тестирање корисничког интерфејса** (енг. *GUI testing*) — најбитнија питања која се постављају код ове врсте тестирања јесу да ли различити уређаји адекватно рендерују податке и да ли се нативне апликације исправно приказују на различитим уређајима. Праве се аутоматски тестови који симулирају интеракцију са екраном уређаја чиме се скраћује време тестирања, а самим тим и утрошен новац на исто.

**Тестирање линије производа** (енг. *product line testing*) — производ се тестира на што већем броју различитих уређаја. Како постоји велики број различитих уређаја, овакво тестирање је скупо па се неретко пушта бета верзија производа у продукцију како би се испратило да ли апликација има непредвиђених грешака [21].

# Глава 4

# Апликација *Duma*

У овом поглављу ће бити детаљно анализирана апликација *Duma* која помаже у организацији догађаја и кључне технологије које су коришћене током њеног развоја. Кроз ову анализу ће бити представљени случајеви употребе и намена апликације *Duma*, где су описане функционалности апликације.

## 4.1 Технологије коришћене за развој

Развојни оквир за креирање корисничког интерфејса *Flutter*, омогућава брз и ефикасан развој препознатљивог корисничког искуства. Платформа *Firebase* пружа стабилно и скалибилно окружење за аутентификацију и базу података, док библиотека *Mockito* олакшава изолацију и тестирање компоненти.

### Развијање интерфејса у развојном оквиру *Flutter*

Обично се у материјалима за учење изградње корисничког интерфејса помоћу развојног оквира *Flutter* пропагира реченица „Све је виџет." (енг. *„Everything is a widget."*). Виџети су класе написане у програмском језику *Dart* које описују кориснички интерфејс апликације. Виџети дефинишу кориснички интерфејс из различитих перспектива, било да се посматра организациони, структурни или стилски део корисничког интерфејса. Неки виџети, као што је *Row*, дефинишу организацију и распоред елемената на корисничком интерфејсу. Неки виџети су мање апстрактни и дефинишу структурне елементе, као што су *Button* и *TextField*. Свака класа која представља виџет

мора да садржи методу *build* која има повратну вредност типа *Widget*. Неке од намена и најчешће коришћених виџета у развоју апликација су:

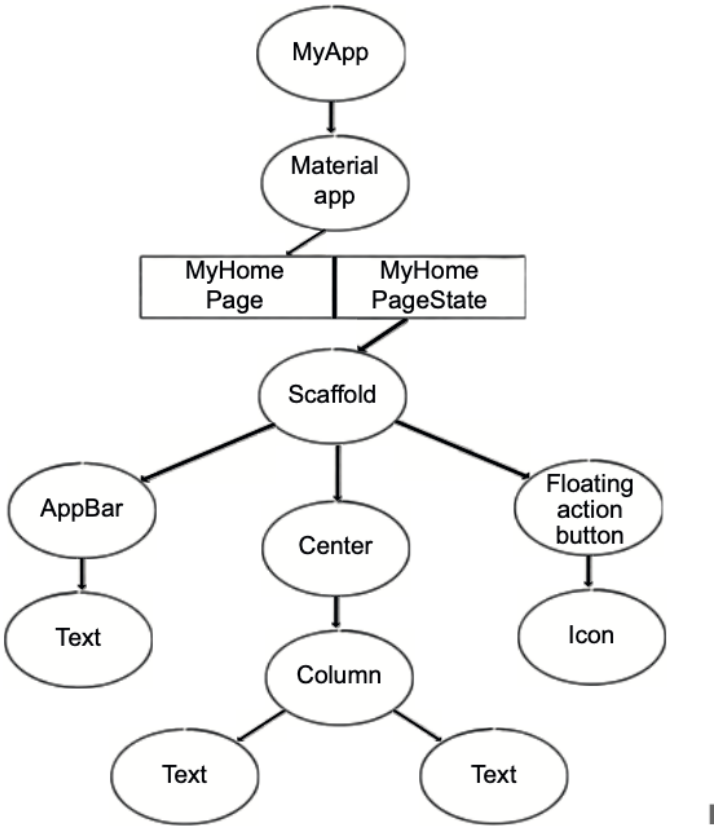
- организација (енг. *layout*) — *Row, Column, Scaffold, Stack*,
- структура (енг. *structure*) — *Button, Toast, MenuDrawer*,
- анимирање (енг. *animation*) — *FadeInPhoto, Transformations*,
- стил (енг. *style*) — *TextStyle, Color, Padding*,
- позиција и поравнање (енг. *positioning and alignment*) — *Center, Align*.

Општи циљ при развоју корисничког интерфејса помоћу *Flutter*-а је да се конструктивно комбинују многи виџети како би се изградило стабло виџета. *Flutter* апликација је представљена помоћу стабла виџета, слично као што је *DOM* (енг. *Document Object Model*) структура стабла за хијерархијски приказ веб странице. Стабло виџета је структура података у коду коју током развоја апликације аутоматски изграђује *Flutter*, али је такође корисна када говоримо о структури *Flutter* апликације. Стабло је скуп чворова, где је сваки чвор један виџет. Сваки пут када додамо виџет и његову методу *build*, додамо нови чвор у стабло. Чворови су повезани односом родитељ-дете. Слика 4.1 приказује стабло виџета на једноставном примеру. Виџет *MaterialApp* представља базни виџет на који се надовезује *MyHomePage* са својим стањем *MyHomePageState* које је задужено за поновно исцртавање екрана у случају да се подаци који се исцртавају промене. Затим се надовезује виџет *Scaffold* чија су деца виџети *AppBar*, *Center* и *FloatingActionButton*. Виџет *AppBar* заузима горњи део екрана, *Center* средишњи, а *FloatingActionButton* представља дугме са иконицом у центру. На слици 4.2 је приказано шта визуелно представљају виџети у стаблу из примера 4.1.

Процес комбиновања виџета у стабло се врши тако што се у структури виџета прослеђује виџет који одређује његово дете. Једноставан пример је стилизовање текста чији је код дат у листингу 4.1:

```
return Container(
    child: Padding(
        padding: EdgeInsets.all(8.0),
        child: Text("Padded Text")
    ),
);
```

Пример 4.1: Комбинација виџета [31]



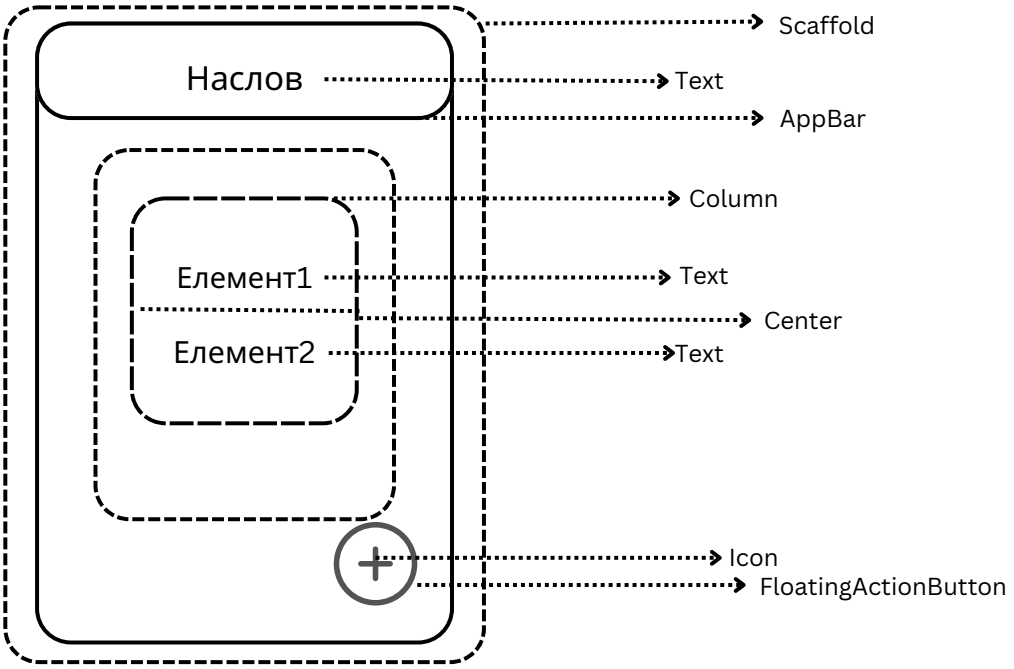

Слика 4.1: Стабло виџета на једноставном примеру

Виџет *Container* у конструктору има параметар који се зове *child* и који је типа *Widget*, у овом случају је то виџет *Padding*. Виџет *Padding* такође има параметар који се зове *child* и који је типа *Widget*. У стаблу виџета, *Container* је родитељ од *Padding*-а, који је родитељ од виџета *Text*.

### Виџети *StatelessWidget* и *StatefulWidget*

У развојном оквиру *Flutter* постоји много доступних виџета за коришћење при имплементацији. Скоро сви су направљени од две различите врсте виџета: *StatelessWidget* и *StatefulWidget*. Разлика између *StatefulWidget*-а и *StatelessWidget*-а је што *StatefulWidget* прати своје унутрашње стање, док *StatelessWidget* нема унутрашње стање које се мења током постојања инстанце виџета (енг. *lifecycle widget*).

За *StatelessWidget* није битно каква је конфигурација или које податке приказује. Може му бити прослеђена конфигурација од родитеља, или кон-




Слика 4.2: Приказ вицета на једноставном примеру

фигурација може бити дефинисана унутар самог вицета, али сам вицет не може мењати своју конфигурацију. Вицет *StatelessWidget* је непроменљив. Пример 4.2 приказује вицет дугмета без конфигурације.

```
class SubmitButton extends StatelessWidget {
    Widget build(context) {
    return Button(
        child: Text('Potvrdi'),
    );
    }
}
```

Пример 4.2: Вицет дугмета без конфигурације [31]

У овако имплементираном вицету не постоји грешка, али можда постоји потреба да дугме има текстуално поље *„Potvrdi"* у неким случајевима, а *„Ažuriraj"* у другима. Како би класа дугме била прилагодљива, потребно је направити вицет дугме са конфигурацијом, где *Flutter* рендерује дугме на основу његове конфигурације и података, као што је приказано у примеру 4.3. *Flutter* има функционалност да рендерује дугме са конфигурацијом сваки пут када се прослеђена променљива разликује.

```
class SubmitButton extends StatelessWidget {
    final String buttonText;
    SubmitButton(this.buttonText);

    Widget build(context) {
    return Button(
        child: Text(buttonText),
    );
    }
}
```

Пример 4.3: Виџет дугмета са конфигурацијом [31]

Међутим, овакав виџет је статичан јер не може да се ажурира сам и нема функционалност да разликује значење дугмета. Његова конфигурација се ослања на родитељске виџете. Нема функционалност да затражи поновно исцртавање виџета, за разлику од *StatefulWidget*-а. *StatelessWidget* може имати методе и својства као било која друга класа, али се *StatelessWidget* у потпуности брише када га *Flutter* уклони из стабла виџета. Стога, *StatelessWidget* не би требало да буде одговоран за било какве податке који су релевантни за чување.

*StatefulWidget* има унутрашње стање и може да управља тим стањем. Сви виџети типа *StatefulWidget* имају одговарајуће објекте стања, тј. за сваку инстанцу типа *StatefulWidget* је везано одговарајуће стање инстанце. Пример 4.4 приказује листинг виџета *StatefulWidget*. Класа *MyHomePage* је повезана са стањем *_MyHomePageState* преко методе *createState()* коју мора имати сваки виџет типа *StatefulWidget* и која враћа објекат типа *State*. Виџети типа *StatefulWidget* су засебно гледано непроменљиви, али њихови припадајући објекти стања су паметни, променљиви и могу да задрже своје стање чак и када *Flutter* поново рендерује виџет.

```
class MyHomePage extends StatefulWidget {
    @override
    _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
    @override
    Widget build(BuildContext context) {
    // ..
    }
}
```

Пример 4.4: Виџет типа *StatefulWidget* [31]

Претпоставимо да виџет *MyHomePage* управља стањем бројача који има могућност да се инкрементира. Када притиснемо дугме за инкрементацију, позива се метода *_incrementCounter* која је дата у примеру 4.5.

```
void _incrementCounter() {
    setState(() {
        _counter++;
    });
}
```

Пример 4.5: Метода која увећава бројач [31]

Метода *setState* је метода која је од суштинске важности за објекат стања. У датом примеру има функционалност да изврши код у склопу тела методе, тј. да увећа вредност бројача за један, а затим да обавести да је потребно поновно исцртавање виџета које се ослањају на променљиву *_counter*. Оно што је битно да се напомене за методу *setState* јесте да ова метода не може извршавати асинхрони код. Све асинхроне радње треба обавити пре позива методе *setState*, јер је погрешно да се екран исцртава пре него што подаци који треба да се прикажу буду спремни за приказ. На пример, ако желимо да прикажемо слику позивајући одређени линк са интернета, не треба да се позове метода *setState* пре него што слика буде спремна за приказ [31].

## О платформи *Firebase*

Платформа *Firebase* представља софтверско решење компаније *Google* које пружа услуге сервиса намењених за развој алпикација и њихово одржавање. У наставку су описани неки од сервиса које нуди платформа *Firebase*:

**Аналитика (сервис *Analytics*)** — Омогућава приказ и праћење статистике коришћења *iOS* и *Android* апликација на основу којих се побољшава квалитет и корисност апликације.

**Аутентификација (сервис *Authentication*)** — Садржи подршку за регистровање и пријављивање корисника. Нуди подршку за пријављивање помоћу е-мејл адресе и лозинке, аутентификацију помоћу броја телефона, као и пријављивање помоћу *Google*, *Facebook*, *Twitter* и *GitHub* налога.

**Систем за слање порука (сервис *Firebase Cloud Messaging*)** — Омогућава уградњу система за бесплатно слање порука. Оваква врста услуге

се широко користи код апликација за дописивање, као и код било којих других где је доступно слање порука између корисника путем апликације.

**Складиштење података (сервис *Real-Time Database*)** — Омогућава складиштење података помоћу *NoSQL* базе података смештене на *cloud*-у и нуди синхронизоване податке свим корисницима.

**Статистика грешака (сервис *Crashlytics*)** — Омогућава преглед података који служе програмерима да препознају проблеме при коришћењу апликације и на тај начин помаже да се проблем брже открије, а самим тим и отклони.

**Сервис за праћење перформанси (сервис *Performance Monitoring*)** — Помаже програмерима да распознају где могу да побољшају перформансе апликације и да испрате побољшања која су уведена.

**Сервиси за тестирање (сервис *Test Lab*)** — Инфраструктура за тестирање смештена на *cloud*-у која посебно помаже при тестирању мобилних апликација нудивши широк спектар различитих модела мобилних уређаја [7].

## О концепту моковања и библиотеци *Mockito*

Тешко је тестирати све могуће сценарије успеха и неуспеха користећи стварне сервисе или базе података [19]. **Моковање** (енг. *mocking*) је концепт који се користи у тестовима јединица како би се избегла зависност од стварних сервиса или базе података. Мокови омогућавају емулирање сервиса и података и враћање фиксираних резултата у складу са потребама. Овиме се и убрзава процес тестирања јер позивање стварних сервиса или базе података значајно успорава извршавање тестова. Такође, уклања се и могућност да тест који је раније пролазио постане неуспешан (ако стварни сервис или база података врате неочекиване резултате). Овакви тестови се називају „нестабилни тестови" (енг. *flaky tests*).

Можемо моковати зависности тако што ћемо креирати алтернативну имплементацију неке класе. Можемо ручно написати ове алтернативне имплементације или користити библиотеку *Mockito* која нам олакшава овај процес.

У наставку је дат пример коришћења библиотеке *Mockito* над функцијом која дохвата податке о догађајима где је приказано креирање тестова јединица

за проверу захтева ка платформи *Firebase*. У примеру 4.6, за који се креира тест јединице, се позива *URL* који враћа у одговору низ догађаја и њихове детаље.

```
Future<Album> fetchEvents(http.Client client) async {
  final response = await client
      .get(Uri.parse('https://test-5a00e-default-rtdb.europe-west1.
    firebasedatabase.app/events.json?'));

  if (response.statusCode == 200) {
    return Album.fromJson(jsonDecode(response.body));
  } else {
    throw Exception('Failed to load events');
  }
}
```

Пример 4.6: Функција за дохватање података о догађајима

Пример 4.7 генерише мок објекат *http* библиотеке где је потребно да се креира нова инстанца овог објекта за сваки тест јединице у коме се користи. Након генерисања објекта потребно је да се за сваки различит одговор направи нови тест.

```
import 'package:http/http.dart' as http;
import 'package:mocking/main.dart';
import 'package:mockito/annotations.dart';

@GenerateMocks([http.Client])
void main() {
}
```

Пример 4.7: Генерисање мок објекта библиотеке *http*

Тестови који се односе на исту функцију се групишу и тестирају појединачно или на нивоу групе. Пример 4.8 приказује креирање групе и тестова у оквиру те групе, где се у оквиру једног теста помоћу функција *when* и *thenAnswer* креира одговор при позивању методе *get* над креираним мок објектом *client* за конкретни тест који се извршава. Затим се користи функција *expect* која проверава да ли функција *fetchEvents* враћа одговарајућу вредност. У првом тесту на линији 21 у листингу у примеру 4.8 се проверава да ли је повратна вредност функције *fetchEvents* типа *Events*, док се у другом тесту на линији 31 у листингу у примеру 4.8 проверава да ли функција враћа изузетак.

```
import 'package:flutter_test/flutter_test.dart';
import 'package:http/http.dart' as http;
import 'package:mocking/main.dart';
import 'package:mockito/annotations.dart';
```

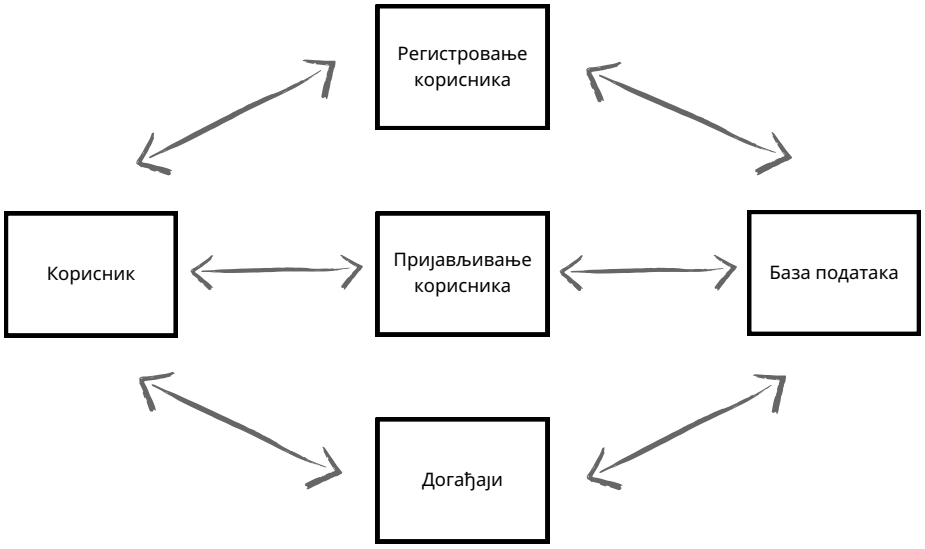
```dart
import 'package:mockito/mockito.dart';

import 'fetch_events_test.mocks.dart';


@GenerateMocks([http.Client])
void main() {
  group('fetchEvents', () {
    test('returns an Events if the http call completes successfully', () async {
      final client = MockClient();

      when(client
              .get(Uri.parse('https://test-5a00e-default-rtdb.europe-west1.firebasedatabase.app/events.json?')))
          .thenAnswer((_) async =>
              http.Response('{"eventId": 1, "id": 2, "title": "mock"}', 200));

      expect(await fetchEvents(client), isA<Events>());
    });

    test('throws an exception if the http call completes with an error', () {
      final client = MockClient();

      when(client
              .get(Uri.parse('https://test-5a00e-default-rtdb.europe-west1.firebasedatabase.app/events.json?')))
          .thenAnswer((_) async => http.Response('Not Found', 404));

      expect(fetchEvents(client), throwsException);
    });
  });
}
```

Пример 4.8: Генерисање групе за тестирање и генерисање тестова

## 4.2 Намена апликације *Duma* и случајеви употребе

Апликација ***Duma*** је мобилна апликација која је развијена у склопу овог рада и њена намена је да помаже фирмама у организацији великих догађаја тиме што евидентира госте и врши статистику броја заказаних догађаја по месецима. Код апликације је јавно доступан `https://github.com/milicagaljak/DumaEventApp`.

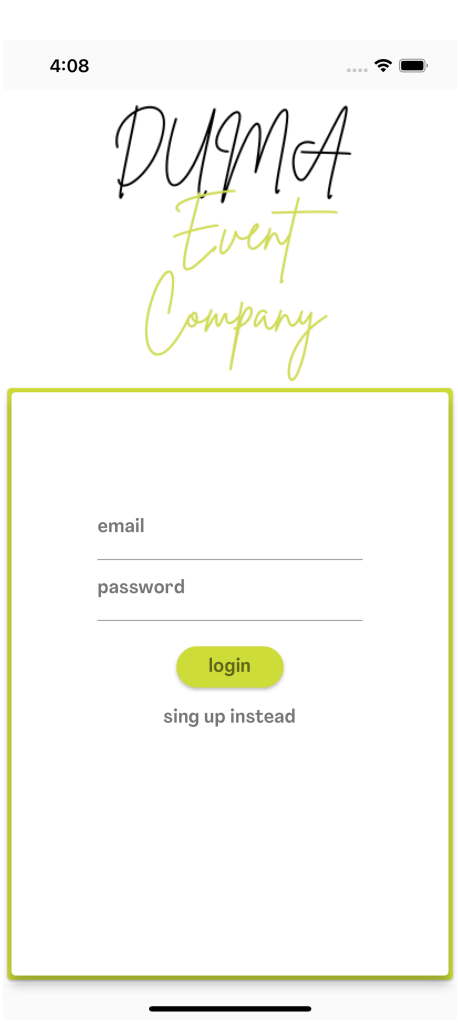
Слика 4.3 приказује главне ентитете апликације и ток података у апликацији. Регистровање и пријављивање корисника, као и манипулација података о догађајима су издвојени ентитети на којима се темељи апликација. Апликација дохвата и мења тражене податке од стране корисника помоћу функција које комуницирају са базом *Firebase*. У наставку су описане функционалности апликације кроз главне случајеве употребе.



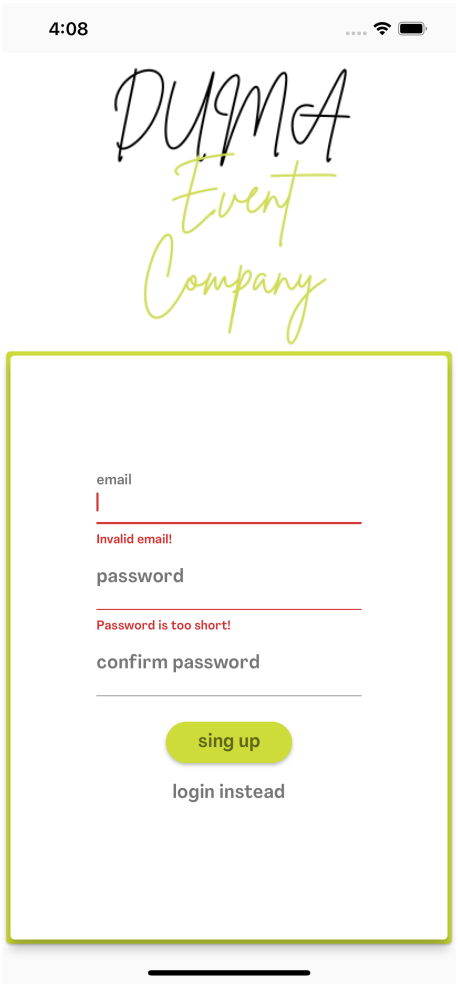

Слика 4.3: Ентитети и ток података

## Регистровање, пријављивање и одјава корисника

Уколико корисник жели да приступи функционалностима апликације, потребно је да корисник буде регистрован или пријављен помоћу мејл адресе и лозинке. За овај случај употребе не постоји предуслов за корисника. Слика 4.4a приказује екран за регистровање и пријављивање корисника. Уколико корисник унесе погрешну лозинку или мејл адресу, на екрану се исписује порука о неисправно унетом податку. Такође, уколико нису унети обавезни подаци, приказује се порука који обавезни подаци недостају. Пример поруке је приказан на слици 4.4b. Уколико корисник жели да се одјави и да се при-




(a) Екран за пријављање и регистровање



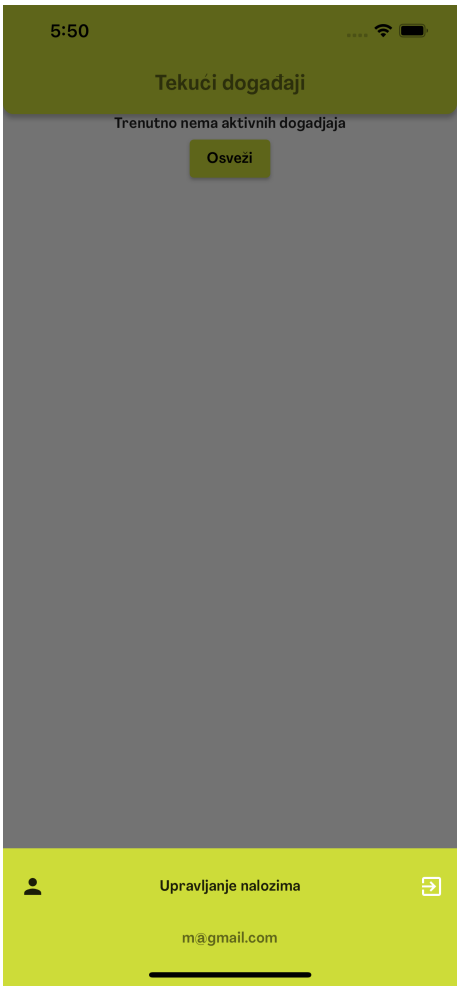

(b) Екран за пријављање и регистровање када нису унети валидни подаци

Слика 4.4: Пријава и регистровање

јави са другим налогом или да направи нови, потребно је да то уради кликом на иконицу за одјављивање из апликације. Слика 4.5 приказује секцију за управљање налогом у којој се налази иконица за одјављивање из апликације. Подаци који су унети приликом регистровања су сачувани у бази и могу да се користе за пријављивање на апликацији.

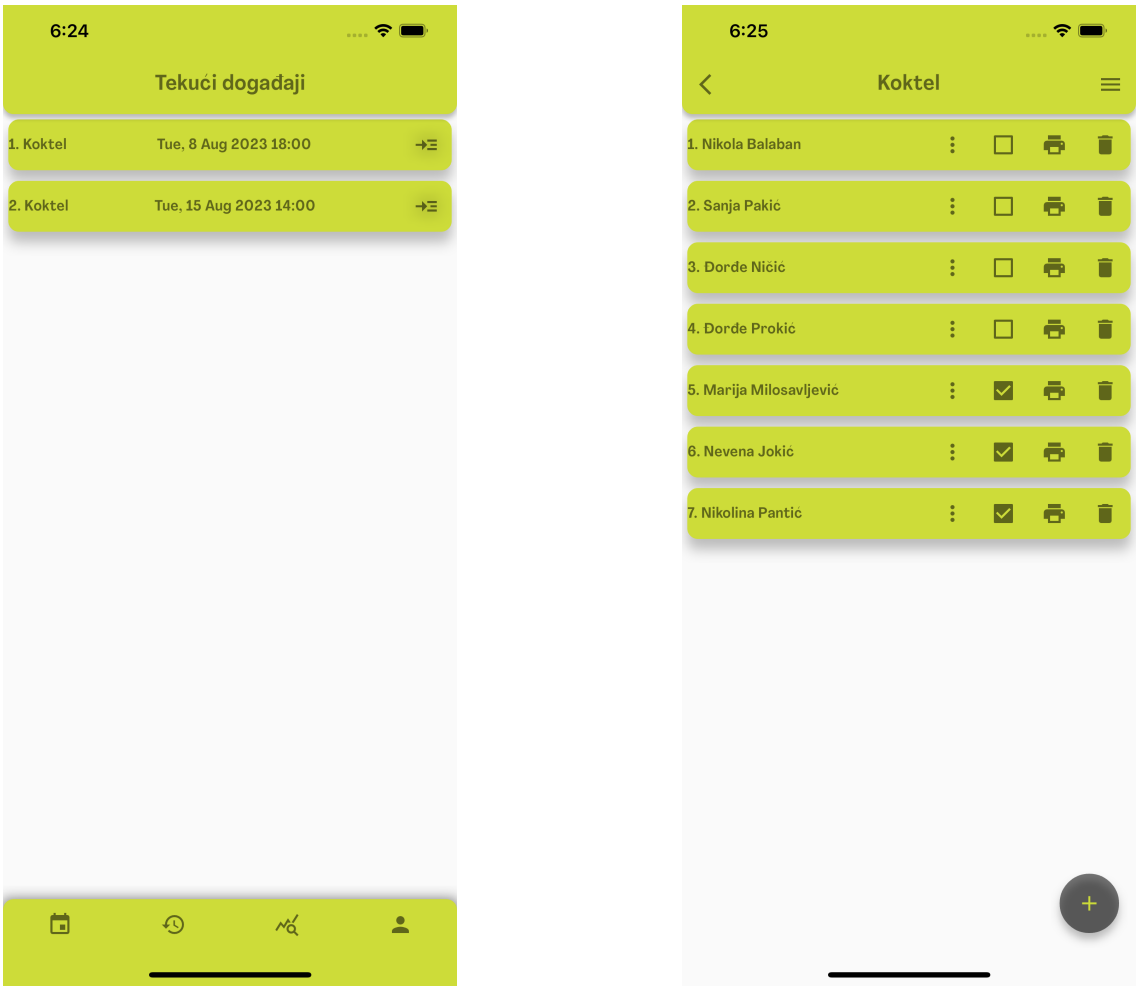
## Текући догађаји

Предуслов овог случаја употребе јесте да је корисник пријављен на апликацији. Након пријаве или регистровања приказује се екран за текуће догађаје




Слика 4.5: Екран за одјављивање

који је илустрован на слици 4.6a. Приказује се низ текућих догађаја и на дну екрана се приказују иконице за приказ листе будућих и протеклих догађаја, секције за статистику догађаја и за управљање налогом редом. Кликом на изабрани текући догађај приказује се екран где се излиставају званице изабраног догађаја. Описани екран је приказан на слици 4.6b. Једна званица садржи податке као што су име и презиме званице, позиција у фирми, фирма коју представља, ознаку да ли је присуствовао догађају и функционалност за брисање званице и штампање акредитације која садржи податке о званици. Поред листе званица, на екрану је приказано дугме за додавање нове званице на дну екрана, као и дугме за приказ детаља догађаја у горњем десном углу екрана. Кликом на дугме за додавање нове званице приказује се форма која је илустрована на слици 4.7a, где се попуњавају подаци о новој званици.

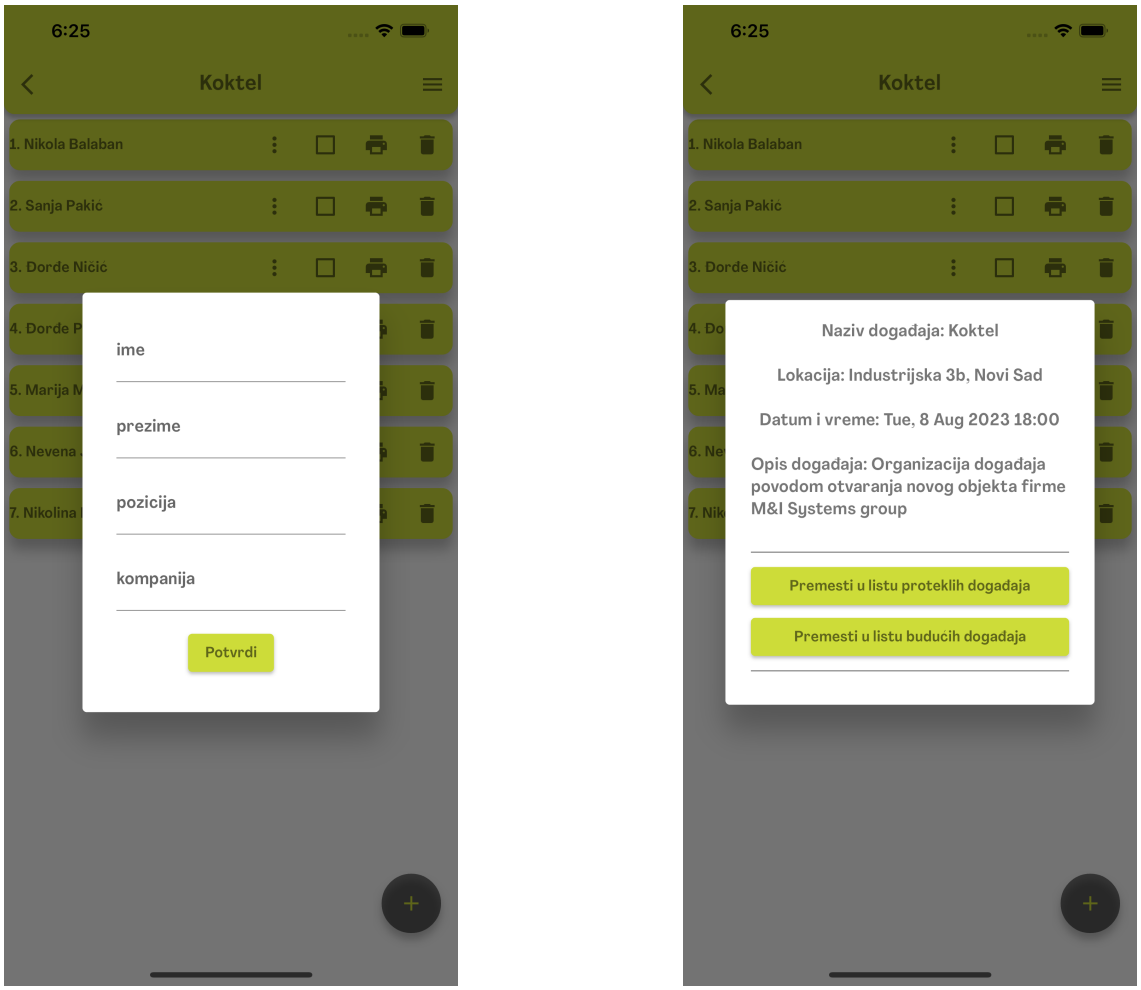



(a) Екран за приказ свих текућих догађаја

(b) Екран за приказ свих званица изабраног догађаја

Слика 4.6: Текући догађаји и званице

Кликом на дугме за приказ детаља догађаја, приказује се екран илустрован на слици 4.7b где су наведени детаљи догађаја као што су назив, локација догађаја, датум и време одржавања догађаја и опис и функционалност за премештање догађаја у листу протеклих или листу будућих догађаја. Кликом на иконицу за штампање акредитације званице приказује се акредитација која ће се одштампати. Могуће је изменити параметре као што су величина акредитације на папиру, оријентација, број копија акредитације итд. Овај екран илустрован је на слици 4.8.



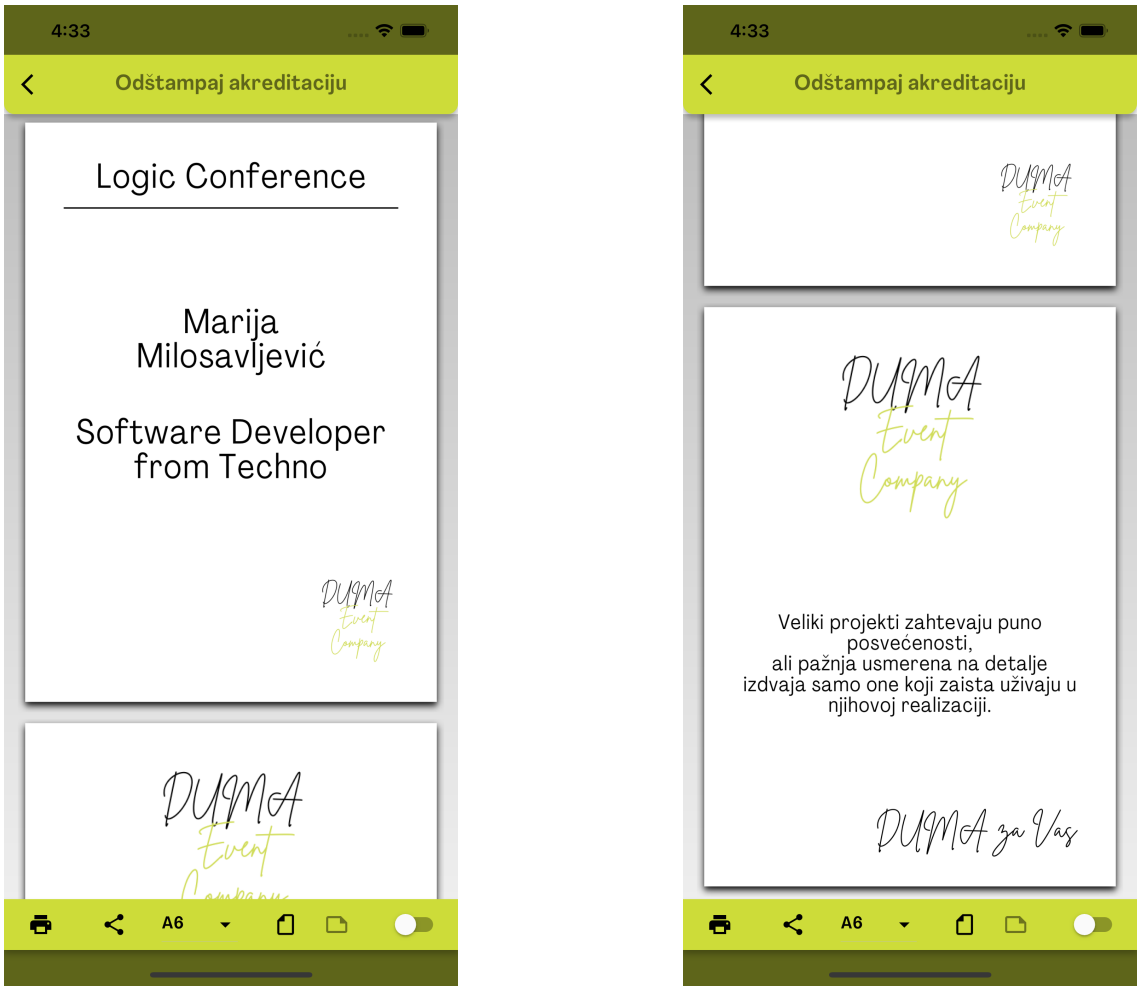

(a) Форма за креирање нове званице изабраног догађаја

(b) Екран за приказ детаља изабраног догађаја

Слика 4.7: Нова званица и детаљи догађаја

## Будући догађаји

Предуслов овог случаја употребе јесте да је корисник пријављен на апликацији и да је на екрану за приказ текућих догађаја кликнуо на иконицу која води до екрана за приказ будућих догађаја који је приказан на слици 4.9a. Поред листе будућих догађаја на дну екрана је приказано дугме које има функционалност за додавање новог догађаја. Кликом на дугме појављује се форма за попуњавање података о новом догађају која је приказана на слици 4.9b. Подаци који се попуњавају су име догађаја, датум догађаја, време догађаја, локација и опис. Графички прикази за одабир датума и сатнице новог догађаја илустровани су на сликама 4.10a и 4.10b.



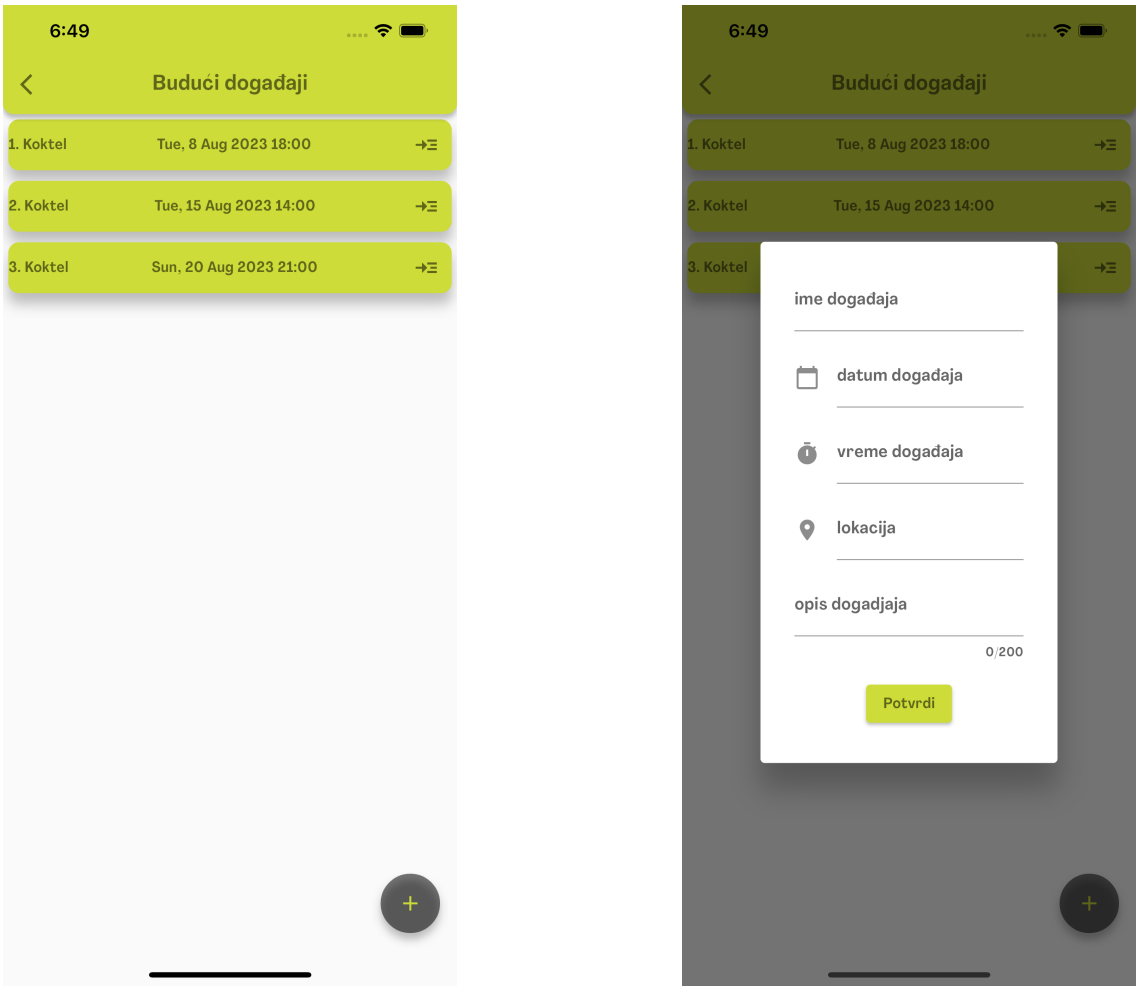

(a) Прва страна за штампање акредитације званице

(b) Друга страна за штампање акредитације званице

Слика 4.8: Екран за штампање акредитације званице

## Протекли догађаји и статистика

Предуслов овог случаја употребе јесте да је корисник пријављен на апликацији и да је на екрану за приказ текућих догађаја кликнуо на иконицу која води до екрана за приказ протеклих догађаја који је приказан на слици 4.11a или да је кликнуо на иконицу која води до екрана за приказ статистике догађаја који је приказан на слици 4.11b. На екрану протеклих догађаја може се видети листа догађаја коју је корисник означио да су протекли, док се на екрану за статистику догађаја може видети графички приказ статистике броја догађаја по месецима у текућој години.




(a) Екран за приказ будућих догађаја

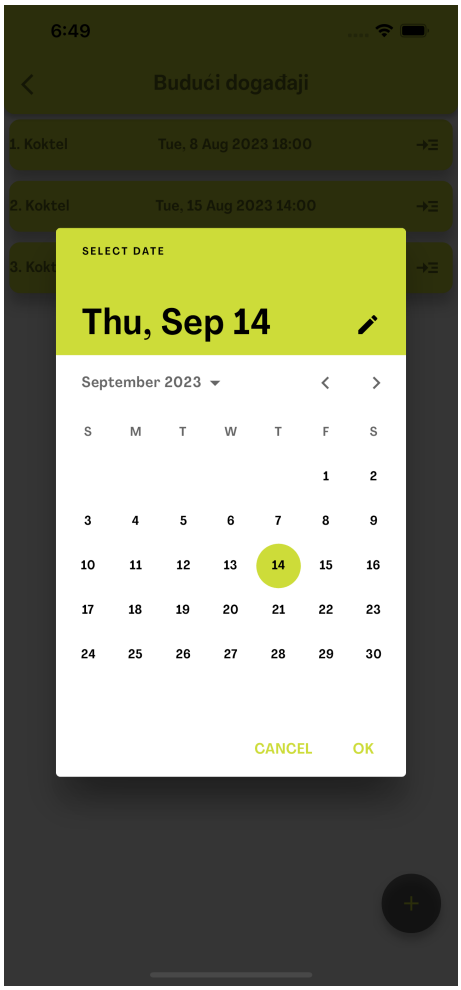
(b) Форма за креирање новог догађаја

Слика 4.9: Будући догађаји

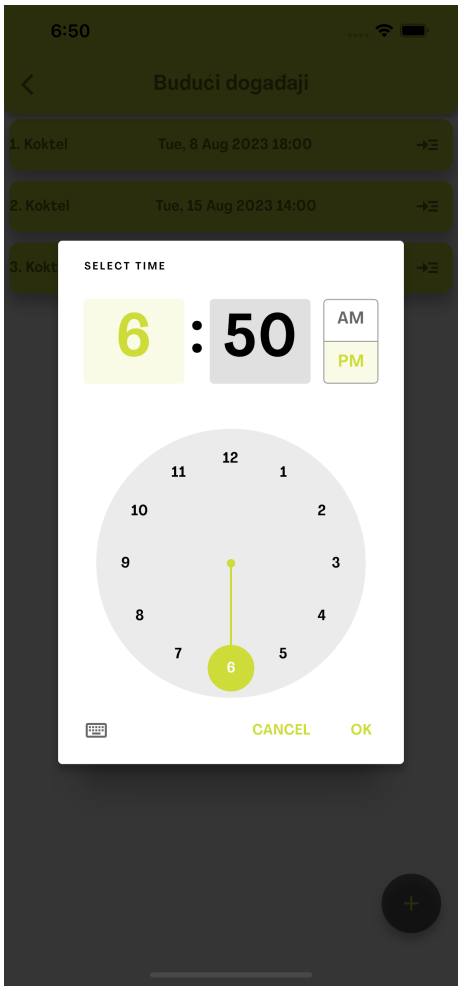
## 4.3 Организација пројекта

У раду се издвајају две целине, имплементација апликације *Duma* и тестирање апликације *Duma*. У складу са тим, у пројекту се издвајају два директоријума *lib* и *test* која су приказана на слици 4.12.

На слици 4.12a је приказана организација директоријума *lib*. У директоријуму *features* се налазе имплементирани виџети који су груписани по директоријумима на основу екрана на којима се исцртавају. У директоријуму *models* се налазе модели за мапирање података који се дохватају из базе података. У директоријуму *services* се налазе фајлови који садрже класе са методама за комуникацију са базом података.




(a) Екран за одабир датума новог догађаја



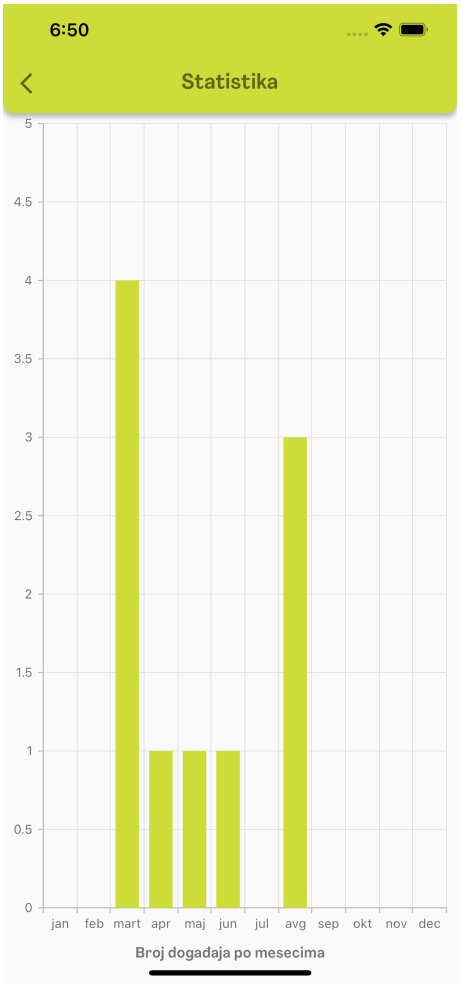

(b) Екран за одабир сатнице новог догађаја

Слика 4.10: Одабир датума и сатнице новог догађаја

На слици 4.12b је приказана организација директоријума *test*. Директоријум садржи тестиране делове кода из директоријума *lib*. Називи директоријума су исти као у директоријуму *lib*, док је за фајлове са екстензијом *.dart* на постојећи назив фајла из директоријума *lib* додат наставак *_test*. Изузетак је директоријум *widgets* у коме се налазе тестови за тестирање корисничког интерфејса који за сваки екран садржи по један фајл са екстензијом *.dart*.

На слици 4.13 је приказан дијаграм класа апликације *Duma*. Средишња класа је класа која одговара догађају и која је повезна са свим осталим класама. За сваку класу су наведени атрибути и њихови типови, операције те класе и односи са другим класама. Дијаграм класа помаже у разумевању

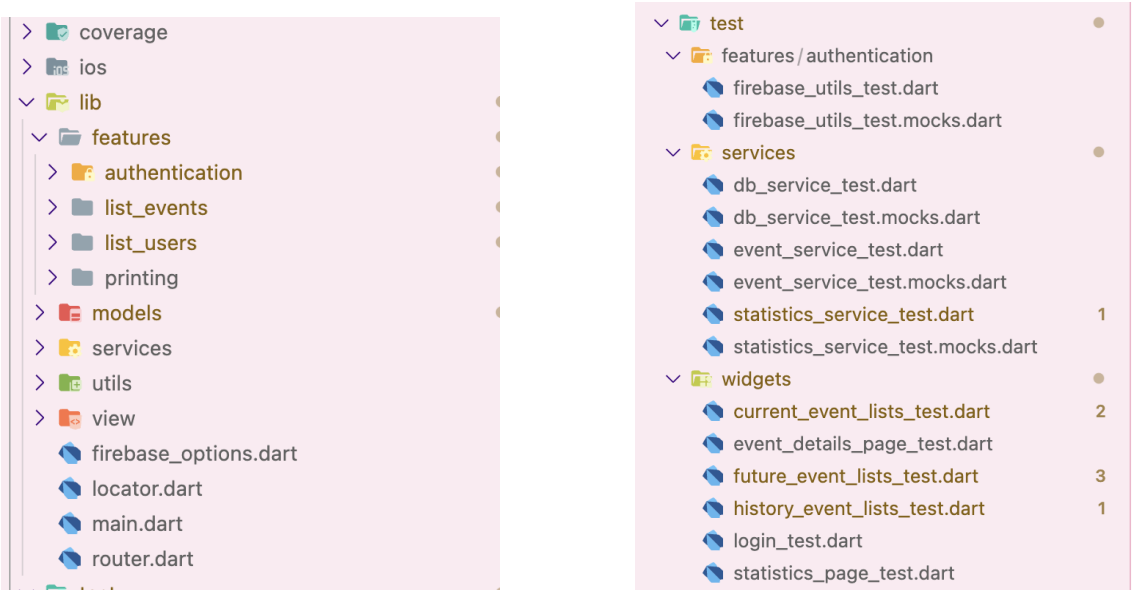

(a) Екран за приказ протеклих догађаја




(b) Екран за приказ статистике

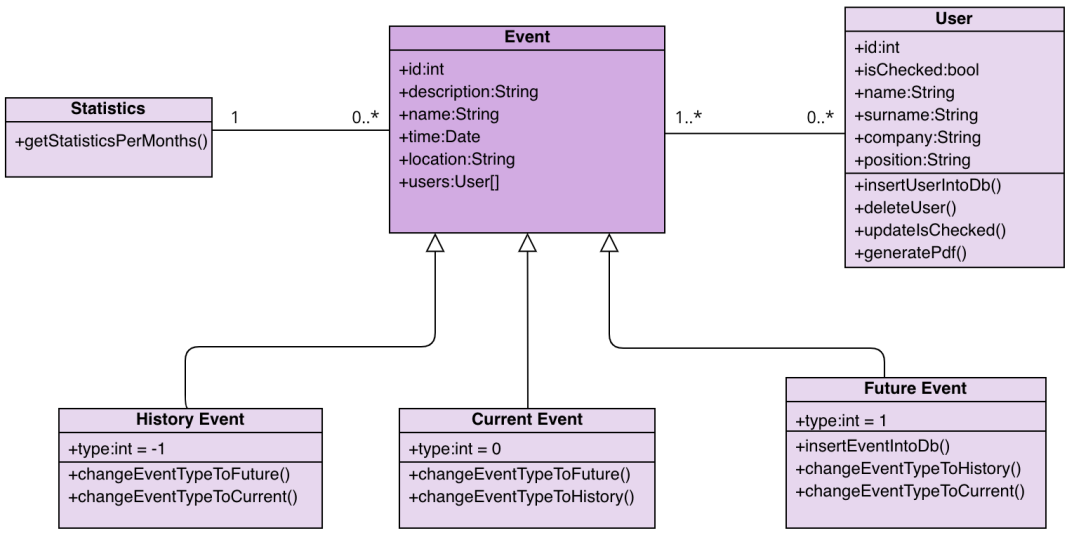
Слика 4.11: Протекли догађаји и статистика

структуре пројекта и међусобним односима класа.




(a) Организација директоријума *lib* (b) Организација директоријума *test*

Слика 4.12: Организација директоријума апликације *Duma*




Слика 4.13: Дијаграм класа апликације *Duma*

# Глава 5

# Тестирање апликације *Duma*

Потребна имплементација система за тестирање апликације *Duma* обухвата више нивоа тестирања, и то тестирање на јединичном нивоу, тестирање вицета и интеграционо тестирање. Тестирање на јединичном нивоу подразумева изолацију сваке функционалне јединице кода и прављење тестова који обухватају различите улазне вредности изоловане јединице. Кроз тестирање вицета се помоћу симулације додира екрана проверава тачност исцртавања вицета на екрану. Интеграционо тестирање проверава понашање апликације на различитим уређајима. У току тестирања апликације потребно је мануелно проверити да ли је целокупни рад апликације у складу са различитим случајевима употребе.

У наставку је описан процес тестирања јединица кода апликације *Duma*, као и резултати покривености кода која је постигнута приликом тестирања.

## 5.1 Тестирање јединица кода

За успешно тестирање јединица кода неопходно је дефинисати које целине кода ће бити тестиране и на који начин ће те целине бити изоловане од остатка кода. Свака метода и функција у апликацији *Duma* се може посматрати као изолована јединица кода где је потребно одстранити зависност од екстерних променљивих које утичу на извршавање јединице. Библиотека *Mockito* има функционалност да направи мок објекте како би се изоловала јединица кода која се тестира као и могућност груписања тестова зарад прегледа целина које се тестирају.

Сваки тест се састоји од геренисања нових мок објеката, уколико јединица

користи екстерне сервисе и податке ван јединице кода, и провере да ли се приликом извршавања кода при задатим условима добија очекивана излазна вредност.

Тестови су груписани на основу различитих функционалности апликације, а унутар групе су дефинисани различити тестови који проверавају исту функционалност али са различитим улазним и излазним вредностима. Групе су описане на слици 5.1, где је наведен назив групе, опис групе, број тестова и време извршавања групе тестова. Написано је укупно 54 различита теста, који су подељени у 10 група. Време извршавања при паралелном покретању свих тестова је приближно једна секунда.

У наставку су описани неки од имплементираних тестова за које се креирају мок објекти ради изолације јединице кода.

| Назив групе тестова | Опис групе | Број тестова | Време извршавања |
|---|---|---|---|
| *create new user* | Регистровање корисника | 5 | 25 милисекунди |
| *sing in* | Пријављивање корисника | 5 | 40 милисекунди |
| *get user* | Дохватање пријављеног корисника | 1 | 21 милисекунда |
| *sing out* | Одјављивање корисника | 2 | 19 милисекунди |
| *database actions* | Додавање и мењање података о догађајима у бази | 14 | 42 милисекунде |
| *get all events* | Дохватање свих догађаја | 4 | 33 милисекунде |
| *get current events* | Дохватање података о текућим догађајима | 4 | 33 милисекунде |
| *get history events* | Дохватање података о протеклим догађајима | 4 | 36 милисекунди |
| *get future events* | Дохватање података о будућим догађајима | 4 | 33 милисекунде |
| *get the number of events* | Дохватање укупног броја догађаја | 5 | 29 милисекунди |
| *get users of the event* | Дохватање званица догађаја | 4 | 25 милисекунди |
| *get statistics* | Дохватање података о статистици | 2 | 12 милисекунди |

Слика 5.1: Групе тестова

У листингу 5.1 је дата функција за креирање корисника за аутентификацију апликације *Duma* на платформи *Firebase*. Објекат *firebaseAuth* чија је метода позвана на линији 3 у примеру 5.1, је објекат класе *FirebaseAuth* из

библиотеке *firebase_auth* која пружа функционалности за аутентификацију корисника на платформи *Firebase*.

```
1  Future<int> createNewUser(String emailAddress, String password) async {
2      try {
3        await firebaseAuth.createUserWithEmailAndPassword(
4          email: emailAddress,
5          password: password,
6        );
7        return 1;
8      } on FirebaseAuthException catch (e) {
9        if (e.code == 'weak-password') {
10         print('The password provided is too weak.');
11         return -1;
12       } else if (e.code == 'email-already-in-use') {
13         print('The account already exists for that email.');
14         return -2;
15       }
16     } catch (e) {
17     }
18     return 0;
19   }
```

Пример 5.1: Функција за регистровање корисника

Како је то екстерни објекат који се користи у функцији, потребно је креирати мок објекат чија класа имплементира методе класе *FirebaseAuth*. У листингу 5.2 је дефинисана класа која наслеђује методе класе *FirebaseAuth*, затим је наглашено на линији 1 у листингу 5.3 које мок класе се креирају. Након тога покренута је команда *flutter pub run build_runner build* која креира мок класу *MockFirebaseAuthTest*, а затим је на линији 3 у листингу 5.3 инстанциран мок објекат који ће се даље користити у тестирању. При креирању објекта класе *FirebaseUtils* која је имплементирана у апликацији *Duma* и која садржи методе за аутентификацију ка платформи *Firebase*, прослеђен је мок објекат класе *FirebaseAuth* који ће надаље користити објекат класе *FirebaseUtils*. У групи тестова функције за креирање корисника која је приказана у листингу 5.4 креирани су тестови за све могуће повратне вредности методе *createUserWithEmailAndPassword* која за прослеђен *email* и лозинку креира корисника на платформи *Firebase*. У тестовима се проверава повратна вредност интерне функције за креирање корисника у зависности од повратне вредности коју врати платформа *Firebase*. Повратна вредност методе *createNewUser* може бити 1, која одговара успешном одговору методе *createUserWithEmailAndPassword*, -1 која одговара случају када је бачен изу-

зетак *FirebaseAuthException(code: „weak - password ")*, -2 која одговара случају када је бачен изузетак *FirebaseAuthException(code: „email -already -in -use")* и 0 која одговара случају када је бачен изузетак *FirebaseAuthException(code: „invalid - email")* или *FirebaseAuthException(„wrong - password")*.

```
class FirebaseAuthTest extends Mock implements FirebaseAuth {
```

Пример 5.2: Креирање мок класе која имплементира методе класе *FirebaseAuth*

```
@GenerateMocks([FirebaseAuthTest, UserTest, UserCredentialTest])
void main() {
  MockFirebaseAuthTest mockFirebaseAuthTest = MockFirebaseAuthTest();
  FirebaseUtils firebaseUtils =
      FirebaseUtils(firebaseAuth: mockFirebaseAuthTest);
```

Пример 5.3: Генерисање мок објекта класе *FirebaseAuth*

```
group("create new user", () {
    test("test create new user", () async {
      MockUserCredentialTest mockCredential = MockUserCredentialTest();

      when(mockFirebaseAuthTest.createUserWithEmailAndPassword(
              email: "milica@gmail.com", password: "pass1234"))
          .thenAnswer((realInvocation) async {
        return mockCredential;
      });

      expect(
          await firebaseUtils.createNewUser("milica@gmail.com", "pass1234"),
    1);
    });

    test("test create new user email-already-in-use", () async {
      when(mockFirebaseAuthTest.createUserWithEmailAndPassword(
              email: "milica@gmail.com", password: "pass1234"))
          .thenThrow(FirebaseAuthException(code: "email-already-in-use"));

      expect(
          await firebaseUtils.createNewUser("milica@gmail.com", "pass1234"),
    -2);
    });

    test("test create new user invalid-email", () async {
      when(mockFirebaseAuthTest.createUserWithEmailAndPassword(
              email: "milica@gmail.com", password: "pass1234"))
          .thenThrow(FirebaseAuthException(code: "invalid-email"));

      expect(
```

```
30          await firebaseUtils.createNewUser("milica@gmail.com", "pass1234"),
    0);
31    });
32
33    test("test create new user operation-not-allowed", () async {
34      when(mockFirebaseAuthTest.createUserWithEmailAndPassword(
35              email: "milica@gmail.com", password: "pass1234"))
36          .thenThrow(FirebaseAuthException(code: "wrong-password"));
37
38      expect(
39          await firebaseUtils.createNewUser("milica@gmail.com", "pass1234"),
    0);
40    });
41
42    test("test create new user weak-password", () async {
43      when(mockFirebaseAuthTest.createUserWithEmailAndPassword(
44              email: "milica@gmail.com", password: "pass1234"))
45          .thenThrow(FirebaseAuthException(code: "weak-password"));
46
47      expect(await firebaseUtils.createNewUser("milica@gmail.com", "pass1234")
    ,
48          -1);
49    });
50  });
```

Пример 5.4: Група тестова јединица за проверу функције за регистровање корисника

У листингу у примеру 5.5 приказана је метода за креирање догађаја у бази на платформи *Firebase*. Потребни објекти за моковање јесу објекат *database* класе *FirebaseDatabase* и објекат *ref* класе *DatabaseReference* из библиотеке *firebase_ database*. Тест који враћа успешан одговор методе за креирање догађаја је приказан у листингу у примеру 5.6.

```
1  Future<int> insertEventIntoDb(int eventId, String eventName, String eventDate,
2        String eventLocation, String description) async {
3      DatabaseReference ref = database.ref("events/${eventId}");
4
5      return await ref.set({
6        "id": eventId,
7        "name": eventName,
8        "time": eventDate,
9        "location": eventLocation,
10       "description": description,
11       "type": 1,
12     }).then((_) {
13       // Data saved successfully!
14       return 1;
15     }).catchError((error) {
16       // The write failed...
```

```
      return 0;
    });
  }
```

Пример 5.5: Метода за креирање догађаја у бази

```
test("insert event into db", () async {
      MockDatabaseReferenceTest databaseReferenceTest =
          MockDatabaseReferenceTest();
      MockFirebaseDatabaseTest firebaseDatabaseTest =
          MockFirebaseDatabaseTest();
      DbService dbService = DbService(database: firebaseDatabaseTest);

      when(firebaseDatabaseTest.ref("events/20")).thenAnswer((_) {
        return databaseReferenceTest;
      });

      when(await databaseReferenceTest.set({
        "id": 20,
        "name": "test",
        "time": "2023-08-31 00:00:00",
        "location": "eventLocation",
        "description": "description",
        "type": 1,
      }))
          .thenAnswer((_) async {
        return;
      });

      expect(
          await dbService.insertEventIntoDb(20, "test", "2023-08-31 00:00:00",
              "eventLocation", "description"),
          1);
    });
```

Пример 5.6: Тест за проверу креирања догађаја у бази

У листингу у примеру 5.7 приказана је метода за дохватање текућих догађаја из базе. У линији 3 поменутог листинга се дохватају сви догађаји, а затим се узимају само они којима је параметар *type* једнак нули. У наставку је дат пример 5.8 теста описане методе за случај када не постоје текући догађаји у коме је потребно да се направи мок објекат класе *Client* из библиотеке *http*.

```
Future<List<Event>?> getEvents(http.Client client) async {
    try {
      final response = await client.get(Uri.parse(
          "https://test-5a00e-default-rtdb.europe-west1.firebasedatabase.app/
    events.json?"));

```

```
6       List<dynamic> result = json
7           .decode(response.body)
8           .toList()
9           .where((i) => i != null)
10          .map((i) => Event.fromJson(i))
11          .toList();
12
13      return result.isNotEmpty
14          ? List<Event>.from(result)
15              .where((element) => element.type == 0)
16              .toList()
17          : [];
18    } catch (e) {
19      return [];
20    }
21  }
```

Пример 5.7: Метод за дохватање текућих догађаја

```
1  test("get current events empty list", () async {
2       MockClient client = MockClient();
3
4       when(client.get(Uri.parse(
5               'https://test-5a00e-default-rtdb.europe-west1.firebasedatabase.
    app/events.json?')))
6           .thenAnswer((_) async => http.Response([].toString(), 200, headers:
    {
7                 'charset': 'utf-8',
8                 'content-type': 'application/json'
9               }));
10
11      expect(await EventService().getEvents(client), []);
12    });
```

Пример 5.8: Тест за методу за дохватање тренутних догађаја у случају да нема тренутних догађаја

## 5.2 О покривености кода апликације *Duma*

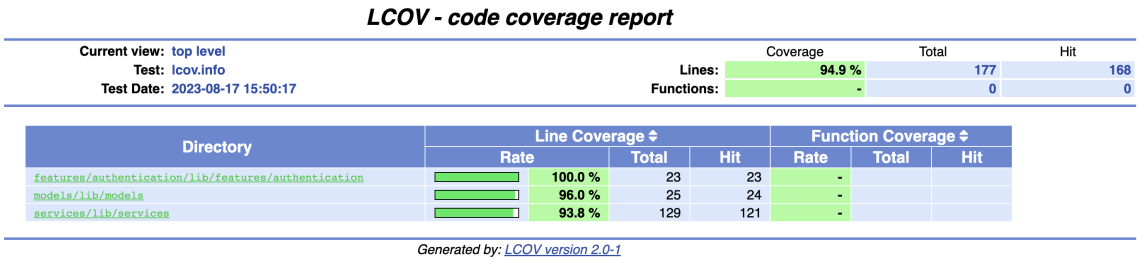
Апликација *Duma* демонстрира како употреба библиотеке *Mockito* може помоћи у постизању високе покривености кода путем тестова јединица. Постигнута је покривеност од 94.9% што указује на то да су готово све кључне компоненте апликације биле укључене у тестирање јединица. Конкретно, функционалности које су укључене у тестирање јесу јединице кода везане за манипулацију догађајима, статистику догађаја, пријављивање и регистрацију корисника.

Развојни оквир *Flutter* пружа функционалност за генерасање извештаја о покривености кода, који садржи потребне информације за постизање високопроцентне покривености кода. Указује на линије кода које нису тестиране и оне су означене црвеном бојом, док оне које су покривене тестовима су означене зеленом бојом. Команде за манипулацију извештаја о покривености кода су приказане у листингу 5.9. Прва команда генерише извештај, друга команда претвара извештај *lcov.info* у читљив документ *index.html* и трећа команда отвара генерисани документ у веб прегледачу.

На слици 5.2 је приказан извештај о покривености кода апликације *Duma*. Функционалности за пријављивање и регистровање корисника на платформу *Firebase* су максимално покривене и оне се налазе у директоријуму *authentication*, док су директоријуми за моделе, где се налазе модели за догађаје и кориснике, и сервисе, где се налазе функције за манипулацију подацима у бази и рачунање статистике, покривени са више од 90%.

```
flutter test --coverage
genhtml coverage/lcov.info -o coverage/html
open coverage/html/index.html
```

Пример 5.9: Команде за креирање извештаја о покривености кода и отварање извештаја помоћу веб прегледача




Слика 5.2: Извештај о покривености кода

## 5.3 Тестирање корисничког интерфејса апликације *Duma*

Тестирање корисничког интерфејса проверава да ли се компоненте корисничког интерфејса правилно приказују на екрану. За прављење тестова је коришћена библиотека *flutter_test*. Направљено је 29 тестова који су подељени у групе у зависности од екрана који се тестира. Тестовима је утврђено

да се тестирани вицети правилно приказују на екрану. На слици 5.3 су приказани називи група заједно са описом и бројем тестова у групи.

| Назив групе тестова | Опис групе | Број тестова |
|---|---|---|
| *Login screen* | Екран за регистровање корисника | 4 |
| *Event details screen* | Екран за приказ детаља догађаја | 3 |
| *Future events list screen* | Екран за приказ будућих догађаја | 5 |
| *History events list screen* | Екран за приказ протеклих догађаја | 4 |
| *Current events list screen* | Екран за приказ текућих догађаја | 9 |
| *Statistics screen* | Екран за приказ статистике | 4 |

Слика 5.3: Групе тестова за тестирање корисничког интерфејса

У наставку су описани неки од тестова који илуструју коришћење библиотеке *flutter_ test*.

У листингу у примеру 5.10 је приказана имплементација теста за замену текста на дугмету на екрану за регистровање корисника. Класа *WidgetTester* која је коришћена у примеру има доступне методе за симулацију додира екрана и унос текста, као и методе за поновно исцртацање екрана. На линији 4 је позвана метода за креирање стабла вицета за вицет *LoginScreen*. Затим је на линијама 6 и 8 проверено да ли се у креираном стаблу налазе вицети *ElevatedButton* са текстом *login* и *TextButton* са текстом *sing up instead*. Уколико корисник жели да се региструје потребно је да притисне текстуално дугме са текстом *sing up instead* и тиме ће се променити текст вицета *ElevatedButton* у *sing up*. За симулацију додира екрана је искоришћена метода *tap* која је позвана на линији 11 у примеру 5.10. Помоћу методе *pump* на ли-

нији 12 позвано је поновно исцртавање екрана. На крају листинга у примеру 5.10 је проверено да ли виџет *ElevatedButton* садржи текст *sing up*.

```
testWidgets(
        'Click on "sing up instead" TextButton and show ElevatedButton as "
    sing up"',
        (WidgetTester tester) async {
      await tester.pumpWidget(MaterialApp(home: LoginScreen()));

      expect(find.widgetWithText(ElevatedButton, 'login'), findsOneWidget);

      expect(
          find.widgetWithText(TextButton, 'sing up instead'), findsOneWidget);

      await tester.tap(find.byType(TextButton));
      await tester.pump();

      expect(find.widgetWithText(ElevatedButton, 'sing up'), findsOneWidget);
    });
```

Пример 5.10: Тест за замену текста дугмета на страници за регистровање

У примеру 5.11 је приказан тест који проверава да ли се кликом на иконицу за додавање догађаја исцртава форма за креирање новог догађаја. На линији 3 у листингу 5.11 је коришћена библиотека *Mockito* за прављење мок објекта *MockClient* који се прослеђује у конструктору при креирању објекта класе *FutureEventList*. Након клика на иконицу за додавање догађаја на линији 18 у примеру 5.11 се проверава да ли је приказан виџет *AlertDialog* који у оквиру класе *FutureEventList* представља форму за креирање догађаја.

```
testWidgets('Add event - click on add icon and show AlertDialog',
      (WidgetTester tester) async {
    MockClient client = MockClient();
    FutureEventList futureEventList = FutureEventList(
      client: client,
    );

    locator.registerLazySingleton(() => NavigationService());

    await tester.pumpWidget(MaterialApp(home: futureEventList));

    expect(find.byIcon(Icons.add), findsOneWidget);

    await tester.tap(find.byIcon(Icons.add));

    await tester.pump();

    expect(find.byType(AlertDialog), findsOneWidget);
```

```
19  });
```

Пример 5.11: Тест за приказ форме за креирање новог догађаја

У примеру 5.12 је приказан тест који проверава да ли се на екрану за приказ текућих догађаја након клика на иконицу за профил исцртава вицет за управљање налозима. Провера се врши након клика на иконицу за профил на линији 15 у примеру 5.12, где се проверава да ли се на екрану налази текст „*Upravljanje nalozima*”.

```
1  testWidgets(
2          'Build current events list and press on profile icon and find "
       Upravljanje nalozima"',
3          (WidgetTester tester) async {
4        MockClient client = MockClient();
5
6        CurrentEventList currentEventList = CurrentEventList(
7          client: client,
8        );
9
10       await tester.pumpWidget(MaterialApp(home: currentEventList));
11       expect(find.byIcon(Icons.person), findsOneWidget);
12       await tester.tap(find.byIcon(Icons.person));
13       await tester.pump();
14
15       expect(find.text("Upravljanje nalozima"), findsOneWidget);
16     });
```

Пример 5.12: Тест за приказ вицета за управљање налозима

# Глава 6

# Закључак

Овај рад истиче улогу аутоматског тестирања у развоју мобилних апликација кроз примену развојног оквира *Flutter* и библиотеке *Mockito*.

Кроз анализу тестирања апликације *Duma* помоћу *Mockito* библиотеке, показано је да је овакав приступ кључан за ефикасно тестирање компоненти апликације, олакшавајући изолацију и проверу функционалности у контролисаном окружењу. У току фазе имплементације апликације и фазе тестирања од изузетне помоћи при раду је била детаљно и јасно написана документација развојног оквира *Flutter*. Иако представља технологију за развој хибридних апликација, развојни оквир *Flutter*, има посебне конфигурације за различите оперативне системе за специфичне функционалности које се уграђују у апликацију, као што је конфигурација за коришћење платформе *Firebase*. Такође, апликација која је развијена није прилагођена уобичајеном корисничком интерфејсу за оперативни систем *Android*, већ само за *iOS*. Уколико би се кориснички интерфејс апликације прилагодио зарад корисничког искуства и стандарда корисничког интерфејса код оперативног система *Android*, дошло би до мењања структуре кода пројекта како не би долазило до редудантности писања кода. Свакако, и поред тога што постоје засебне целине које се уводе за различите оперативне системе, развој и тестирање апликације је лакше и ефикасније уз помоћ развојног оквира *Flutter*.

Могућа унапређења на развоју апликације *Duma* укључују додавање функционалности као што је увођење *Google Maps* за локацију догађаја, додавање података о пријављеном кориснику на страници профила, као и пријављивање на апликацију путем броја телефона. Додатно, могла би да се допуни страница за статистику са потребним статистичким подацима. Апликација би

могла да се прошири са нивоима приступа апликацији и функционалностима за запослене чланове, менаџмент и клијенте. Ради унапређења тестирања апликације могу се користити сервиси за интеграционо тестирање, као што је сервис *TestLab*, који проверава рад апликације на различитим врстама и моделима уређаја. Увођење бета тестирања апликације, где би група корисника користила апликацију на догађајима, би помогло за прикупљање информација о корисничком искуству и сходно томе унапређењу саме апликације.

# Библиографија


[1] Android documentation. on-line at: https://developer.android.com/codelabs/build-your-first-android-app#0.

[2] Platform architecture. on-line at: https://developer.android.com/guide/platform#:~:text=The%20Android%20platform%20provides%20Java,3D%20graphics%20in%20your%20app.

[3] About Objective-C. https://developer.apple.com/library/archive/documentation/Cocoa/Con

[4] Android. on-line at: https://www.britannica.com/technology/Android-operating-system.

[5] iOS. on-line at: https://www.britannica.com/topic/iOS.

[6] Patrick Böllhoff. Kotlin vs. Java. on-line at: https://kruschecompany.com/kotlin-vs-java/#:~:text=Last%20but%20not%20least%20is,have%20the%20status%20as%20Kotlin.

[7] Pankaj Chougale, Vaibhav Yadav, and Anil Dr. Gaikwad. FIREBASE - OVERVIEW AND USAGE, 2021. on-line at: https://www.researchgate.net/publication/362539877_FIREBASE_-OVERVIEW_AND_USAGEs.

[8] Jan Christoph, Daniel Rösch, Thomas Schuster, and Lukas Waidelich. Current Progress in Cross-Platform Application Development, 2019. on-line at: https://www.thinkmind.org/articles/soft_v12_n12_2019_3.pdf.

[9] The official Dart site. on-line at: https://dart.dev/language/concurrency.

[10] The official Dart site. on-line at: https://dart.dev/overview#platform.

[11] Wafaa S. El-Kassas, Bassem A. Abdullah, Ahmed H. Yousef, and Ayman M. Wahba. Taxonomy of Cross-Platform Mobile Applications Development Approaches, 2014. on-line at: https://www.sciencedirect.com/science/article/pii/S2090447915001276.

[12] The official Flutter site. on-line at: https://flutter.dev/.

[13] The official Gerrit site. on-line at: https://www.gerritcodereview.com/.

[14] The official Gitlab site. on-line at: https://docs.gitlab.com/ee/development/code_review.html.

[15] What is Java? on-line at: https://www.ibm.com/topics/java.

[16] Kotlin documentation. on-line at: https://developer.android.com/kotlin.

[17] Kotlin Language Documentation 1.8.20. on-line at: https://kotlinlang.org/docs/kotlin-reference.pdf.

[18] Kotlin overview. on-line at: https://developer.android.com/kotlin/overview#:~:text=Kotlin%20is%20an%20open%2Dsource,and%20Scala%2C%20among%20many%20others.

[19] The official Flutter site. on-line at: https://docs.flutter.dev/cookbook/testing/unit/mocking.

[20] LACHGAR Mohamed and ABDALI Abdelmounaïm. Decision Framework for Mobile Development Methods, 2017. on-line at: https://www.researchgate.net/publication/314165913_Decision_Framework_for_Mobile_Development_Methods.

[21] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito. Software Testing of Mobile Applications: Challenges and Future Research Directions, 2012. on-line at: https://www.researchgate.net/publication/254041958_Software_testing_of_mobile_applications_Challenges_and_future_research_directions.

[22] Objective-C documentation. on-line at: https://livebook.manning.com/book/objective-c-fundamentals/chapter-1/.

[23] The official Phabricator site. on-line at: https://www.phacility.com/phabricator/.

[24] React Native documentation. on-line at: https://reactnative.dev/.

[25] About Swift. https://www.swift.org/about/.

[26] Swift documentation. on-line at: https://www.swift.com/.

[27] Mubarak Albarka Umar. Comprehensive study of software testing: Categories, levels, techniques, and types, 2020. on-line at: https://www.researchgate.net/publication/342538504_Comprehensive_study_of_software_testing_Categories_levels_techniques_and_types.

[28] About Swift. https://en.wikipedia.org/wiki/Swift_(programming_language).

[29] React Native. on-line at: https://en.wikipedia.org/wiki/React_Native#See_also.

[30] Xamarin Wikipedia. on-line at: https://en.wikipedia.org/wiki/Xamarin#Xamarin_platform.

[31] Eric Windmill. Flutter in action, 2020. on-line at: https://dokumen.pub/flutter-in-action-1nbsped-1617296147-978-1617296147.html.

[32] Xamarin documantation. on-line at: https://learn.microsoft.com/en-us/xamarin/?WT.mc_id=dotnet-35129-website.

[33] Милена Вујошевић Јаничић. Верификација софтвера, 2023. on-line at: http://www.verifikacijasoftvera.matf.bg.ac.rs/vs//verifikacija_softvera.pdf.

[34] Милена Вујошевић Јаничић. Статичка анализа, 2023. on-line at: http://www.verifikacijasoftvera.matf.bg.ac.rs/vs//predavanja/04_staticka_analiza_pregledi/04_staticka_analiza.pdf.

# Биографија аутора

**Милица Гаљак** рођена је 11.07.1996. у Краљеву. Са одличним успехом је завршила основну школу „Вук Караџић" на Берановцу и Гимназију у Краљеву. Године 2015. уписује основне студије на смеру Математика на модулу Рачунарство и информатика на Математичком факултету у Београду. У септембру 2020. године завршава студије и исте године уписује мастер студије на истој катедри.

У октобру 2020. године запошљава се у компанији *Procescom* где ради и данас. Током рада у *Procescom*-у сусрела се са различитим проблемима и решењима у области телекомуникација. Тренутно ради на развијању и тестирању микросервисних апликација написаним у програмском језику *Java*.