Testiranje Flutter aplikacija u kontekstu procesa DevOps-a

Papac, Iva

Master's thesis / Diplomski rad

2022

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

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:603639

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Iva Papac

Testiranje Flutter aplikacija u kontekstu procesa DevOps-a

DIPLOMSKI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Iva Papac

Matični broj: 0016129565

Studij: Informacijsko i programsko inženjerstvo

Testiranje Flutter aplikacija u kontekstu procesa DevOps-a

DIPLOMSKI RAD

Mentor:

Izv. prof. dr. sc. Zlatko Stapić

Varaždin, rujan 2022.

Iva Papac

Izjava o izvornosti

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

Autorica potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

_____

Sažetak

Tema ovog rada je testiranje Flutter aplikacija u kontekstu procesa DevOps-a, što je i prepoznato kao važna aktivnost u sklopu procesa razvoja programskih proizvoda te dio je prakse DevOps-a. Rad započinje kratkim pregledom specifikacija programskog okvira Flutter i programskog jezika Dart, a prikazani su i jednostavni primjeri programskog koda koji se najčešće koriste u praksi. Slijedi opis osnovnih značajki DevOps-a te analiza utjecaja DevOps praksi na razvoj programskih proizvoda. U središtu rada opisane su metode testiranje te testiranje u kontekstu procesa DevOps-a. U praktičnom dijelu kreirana je jednostavna Flutter aplikacija te su kreirani, automatizirani i provedeni testovi. Za kraj su, dakle, prikazani najvažniji dijelovi implementacije i testiranja Flutter aplikacije te kratak opis i analiza navedenih procesa.

Ključne riječi: razvoj programskih proizvoda, jedinično testiranje, Flutter, automatizirano testiranje, integracijsko testiranje, DevOps

Sadržaj

1. Uvod

Tema ovog diplomskog rada je „Testiranje Flutter aplikacija u kontekstu procesa DevOps-a“ te ju možemo razlučiti na tri središnja dijela. To su razvoj programskih proizvoda pomoću programskog okvira Flutter i programskog jezika Dart, testiranje programskih proizvoda i osnove DevOps procesa. Dakle, cilj ovog rada je opisati programski okvir Flutter i razviti jedan programski proizvod koristeći navedenu tehnologiju. Sljedeći cilj je opisati testiranje programskih proizvoda, a posebice jedinično i integracijsko testiranje te testiranje elemenata korisničkog sučelja. Navedena testiranja provest će se na razvijenom programskog proizvodu u sklopu praktičnog dijela ovog rada. Na kraju, jedan od ciljeva je opisati osnovne koncepte DevOps procesa i u tom kontekstu spomenuti kontinuirano testiranje, integraciju i isporuku programskih proizvoda.

Važnost ove teme leži u testiranju programskih proizvoda što se često izbjegava i ne pridaje velika važnost u pogledu jedne od faza procesa razvoja programskog proizvoda. Molin [1] navodi da postoje pet razloga koji pridaju važnost testiranju. Kontinuirano testiranje omogućuje kontinuirano poboljšanje kvalitete programskog proizvoda s obzirom na to da je kvalitetu moguće mjeriti brojem grešaka, a greške je jedino moguće uočiti i otkloniti testiranjem. Drugi razlog leži u poslovici *„Tko radi – taj i griješi“* odnosno svaki razvojni programer radi greške prilikom pisanja programskog koda, stoga je bitno redovito testirati programski proizvod kako bi on bio u skladu s definiranim funkcionalnim zahtjevima. Nadalje, kontinuirano testiranje smanjuje rizik od pojavljivanja nenadanih grešaka u programskog proizvodu koji je već u produkciji. Učestale greške u radu programskog proizvoda utječu na zadovoljstvo korisnika i na njegov stav prema tom proizvodu. Osim kontinuiranog testiranje, bitno je i rano testiranje koje smanjuje rizik od produljenja roka završetka projekta. Sve navedeno vodi nas do zadovoljnog korisnika koji se može pouzdati u rad i kvalitetu programskog proizvoda [1].

U zadnjih nekoliko godina programski okvir Flutter postao je značajno popularan među razvojnim programerima mobilnih aplikacija, a Pandey [2] navodi da je tomu tako jer im omogućuje jedan programski kod za iOS, Android i web platforme. Najmanje promjene u programskog kodu automatski su vidljive na zaslonu mobilne aplikacije zahvaljujući takozvanoj *Hot Reload* funkcionalnosti, odnosno brzinskom ponovnom učitavanju komponenata korisničkog sučelja. Proces razvoja korisničkog sučelja je jednostavniji s obzirom na to da nam Google nudi gotove komponente kroz sustav *Material Design* [2].

Prije svega za odabir ove teme diplomskog rada motivirala me popularnost programskog okvira Flutter i potražnja razvojnih programera u toj niši. Diplomski rad sam željela iskoristiti za učenje novih tehnologija koje će mi olakšati karijerni put. Osim toga, testiranje programskih proizvoda je proces o kojem sam tek površno naučila tijekom studija i

često zanemarivala, iako je to vještina koja je često poželjna uz poznavanje određenih tehnologija. Vjerujem da je većina mojih starijih kolega tu vještinu savladala tek nakon stjecanja iskustva u samom programiranju. Moj cilj je da pomoću rada na ovom diplomskog radu što prije usvojim znanja iz područja programiranja i testiranja Flutter aplikacija kako bih imala poželjne vještine za buduće poslodavce.

Rad sadrži tri poglavlja u kojima se odabrana tema teorijski obrađuje, a jedno poglavlje sadrži opis implementacije i testiranja programskog proizvoda. U poglavlju 3 obrađuju se osnovne značajke i specifičnosti programskog okvira Flutter i programskog jezika Dart, a ono čini temelj ovog rada s obzirom na to da je cilj na primjeru navedenog programskog jezika i okvira opisati i prikazati testiranje programskih proizvoda. Drugo teorijsko poglavlje obuhvaća najznačajnije principe i prakse DevOps-a, najčešće korištene alate te uloge i odgovornosti DevOps tima. U posljednjem teorijskom poglavlju opisane su tri metode testiranja, jedinično i integracijsko testiranje te testiranje elemenata korisničkog sučelja. Također, obrađeno je testiranje u kontekstu procesa DevOps-a, što je ujedno i središte ovog rada. Navedene metoda korištene su u praktičnom dijelu rada, a proces implementacije istih je olakšan zbog obrade praktičnih primjera u poglavlju 5. Razvoj i testiranje programskog proizvoda u Flutteru dio su praktičnog dijela rada te u posljednjem poglavlju prikazani su najvažniji dijelovi programskog koda, dok je ostatak priložen uz rad. Cilj praktičnog dijela rada je primijeniti naučene metode testiranje na programskom proizvodu pisanom u Flutteru.

2. Metode i tehnike rada

Prije razrade sadržaja i pisanjem rada, istražena je okvirna literatura koja će se koristiti. Okosnica literature su nekoliko knjiga koje obuhvaćaju Flutter, testiranje programskih proizvoda i DevOps. Budući da se radi o novim tehnologijama, veliki dio literature bit će i izvori s interneta kao što su članci i službena dokumentacija. Za lakše upravljanje literaturom koristit će se alat Mendeley. Pisanje teorijskog dijela rada provodit će se paralelno s radom na praktičnom dijelu odnosno na razvoju i testiranju Flutter aplikacije. Za kreiranje i razradu sadržaja rada korištene su sljedeće metode:

- **Metoda analize** koristila se za rastavljanje složenih pojmova i definicija na manje dijelove s ciljem lakšeg razumijevanja sadržaja rada [3].
- **Metoda sinteze** koristila se za objedinjavanje različitih dijelova rada u cjelinu. Ti dijelovi su najčešće predstavljali sadržaj iz više različitih izvora literature [3].
- **Metoda generalizacije** je postupak kreiranja općih zaključaka izvedenih iz posebnih pojmova [3].
- **Metoda klasifikacije** korištena je za raspoređivanje pojmova, definicija i drugog sadržaja u odgovarajuće posebne skupine [3].
- **Metoda deskripcije** se koristi za pregled i opis činjenica bez znanstvenog objašnjavanja [3].
- **Metoda komplikacije** koristi se prilikom preuzimanja tuđih rezultata i stavova pri usporedbi programskog okvira Flutter s drugim programskim okvirima i jezicima [3].

Aplikacija iz praktičnog dijela rada razvijena je u sklopu Flutter tečaja koji je organizirala tvrtka Cinnamon Agency iz Zagreba. Proces razvoja programskog proizvoda krenuo je od faze razvoja s obzirom na to da su prototip aplikacije i dizajn sustava već bili pripremljeni. Za razvoj i testiranje Flutter aplikacije korišteni su sljedeći alati i tehnologije:

- **Android Studio** – integrirano razvojno okruženje *(eng. integrated development environment, IDE)* za razvoj mobilnih aplikacija.
- **Firebase** – platforma sa skupom različitih alata za razvoj *web* i mobilnih aplikacija. Sljedeći alati su korišteni u ovom radu:
  - **Authentication** – autentifikacija korisnika u mobilnu aplikaciju,
  - **Firestore Database** – baza podataka,
  - **Storage** – pohrana datoteka u .jpg i .pdf formatima, i
  - **Functions** – JavaScript funkcije koje se pozivaju na određeni korisnički događaj, na primjer registracija i validacija adrese e-pošte.
- **GitHub** – alat za verzioniranje izvornog programskog koda.
- **Codemagic** – alat za kontinuiranu integraciju i kontinuiranu isporuku

3. Flutter

Flutter je programski okvir za razvoj korisničkog sučelja *(eng. User Interface, UI)* te omogućuje razvoj mobilnih, *desktop* i *web* aplikacija s jednim programskim kodom i izvornim kompiliranjem *(eng. native compiling)* [4]. Prvi put je predstavljen javnosti 2015. godine na konferenciji *Dart Developer Summit* kao eksperiment pod nazivom *Dart on mobile*, a danas je već u širokoj upotrebi u svijetu informacijske tehnologije *(eng. Information Technology, IT)* [5]. Flutter je besplatan alat otvorenog koda kojemu svakodnevno doprinose Googleovi razvojni programeri i ostali istaknuti članovi IT zajednice [5], [6]. Flutter koristi programski jezik Dart za razvoj višeplatformskih programskih proizvoda. Riječ je o objektno orijentiranom jeziku koji se kompilira u izvorni ARM *(eng. Advanced RISC Machines)* kod [5].

U nastavku poglavlja detaljnije će biti opisana arhitektura i osnovni koncepti programskog okvira Flutter te koji se koncepti najčešće koriste u razvoju višeplatformskih aplikacija. Osim toga, u nastavku će biti opisani osnovni koncepti programskog jezika Dart.

3.1. Osnovne značajke

Za kreiranje Flutter projekta, prije svega, potreban je određeni set alata. Flutter SDK *(eng. software development kit)* je paket za razvoj programa i sadrži kompilator, upravitelj uređaja, dijagnostičke alate i alate za testiranje. Za integrirano razvojno okruženje *(eng. Integrated development environment, IDE)* razvojni programeri imaju više opcija – Microsoft Visual Studio Code, Android Studio i IntelliJ [6]. Za sada je Android Studio Googleov prvi izbor za razvoj Flutter aplikacija [4]. Posljednji potrebni alati su emulatori, odnosno virtualni uređaji na kojima se pokreću aplikacije, a radi se o iOS simulatoru i Android emulatoru. Za pokretanje iOS simulatora potrebno je Mac računalo, dok se Android emulator može pokrenuti na bilo kojem računalu [6].

Najvažnija datoteka Flutter projekta je `pubspec.yaml` pomoću koje se upravlja paketima i resursima korištenima u izvornom kodu. Kratica YAML označava *YAML Ain't Markup Language*, a radi se o jeziku za serijalizaciju podataka i koristi se u konfiguracijskim datotekama. Ovaj jezik koristi uvlake i prijelome redaka kao separatore, dakle nema znakova poput točke, točke-zareza i zareza [4].

Svaki Flutter projekt ima istu strukturu direktorija, a ovisno o razvojnom okruženju u strukturi se mogu naći još neki specifični direktoriji. Tablica 1 prikazuje strukturu direktorija i osnovnih datoteka Flutter projekata te njihov opis. Neki direktoriji i datoteke nisu spomenuti jer se ne mijenjaju tijekom procesa razvoja programskog proizvoda i njima upravlja razvojno okruženje [4].

Tablica 1: Struktura direktorija i datoteke Flutter projekta [4], [6]

Direktorij	Opis
android, ios	Ovi direktoriji sadrže programski kod vezan uz Android i iOS platformu, a automatski se kreiraju i prilikom kreiranja Android projekta u Android Studiu ili iOS projekta u XCodeu. U većini vremena u njima ne dolazi do promjena.
lib	Ovaj direktorij je srž Flutter projekta jer se u njemu nalazi sav izvorni kod pisan u Dartu.
test	U ovom direktoriju se pohranjuju jedinični i integracijski testovi te testovi korisničkog sučelja.
pubspec.yaml	Ovo je osnovna datoteka svakog projekta u kojoj se nalazi popis Dart paketa i ovisnosti, ime i opis projekta.
README.md, .gitignore	Ove datoteke vezane su uz verzioniranje izvornog koda. Datoteka README.md obično sadrži opis projekta i instalacije aplikacije. U datoteci .gitignore se nalazi popis svih direktorija i datoteka koje ne želimo postaviti u udaljeni repozitorij.
.metadata, .packages	Ovo su važne konfiguracijske datoteke potrebne za normalno funkcioniranje Flutter projekta.

U odnosu na druge programske jezike i okvire prvenstveno namijenjene razvoju mobilnih aplikacija, Flutter uvodi novi koncept pod nazivom *Hot Reload.* Ova funkcionalnost omogućuje brzinsko osvježavanje korisničkog sučelja aplikacije čuvajući posljednje stanje i podatke [6]. Ova funkcionalnost ne radi uvijek, već je aplikaciju potrebno ponovno pokrenuti prilikom promjena u metodi `initState()`, pri promjeni definicije klase i statičkih članova klase te prilikom promjena u glavnoj metodi `main()` [4].

3.1.1. Programski jezik Dart

Već je spomenuto da Flutter koristi programski jezik Dart pa nije moguće opisati osnovne koncepte Fluttera bez da se opišu osnovni koncepti ovog programskog jezika. Dart je objektno orijentiran, proceduralni programski jezik te ima sličnu sintaksu kao i drugi programski jezici poput Jave, C# i C++. Ova karakteristika omogućuje razvojnim programerima da brzo savladaju Dart [6].

Programski jezik Dart podržava tipove podataka poznate iz drugih jezika kao što su cijeli i decimalni brojevi *(int, double),* liste i rječnici *(List, Set, Map),* simboli, logički tipovi (*bool*) i drugi. Varijable je moguće kreirati i inicijalizirati na više načina, odnosno koristeći različite ključne riječi [7]. Tablica 2 sadrži ključne riječi za kreiranje varijabli i kratko objašnjenje. Moć programskog jezika Dart leži u tome da varijablu možemo kreirati uz pomoć ključne riječi *var* i

on će sam prepoznati o kojem je tipu podatka riječ na temelju inicijalne vrijednosti te varijable [6].

Tablica 2: Ključne riječi za kreiranje varijable [7]

Ključna riječ	Opis
var name = 'Iva'	Pohranjuje referencu na String objekt s vrijednošću 'Iva'.
String name = 'Iva'	Eksplicitno navođenje tipa podatka. Varijabla je ograničena na navedeni tip odnosno String.
late String name	Odgađanje inicijalizacije varijable koja ne može pohranjivati null vrijednost.
dynamic name = 'Iva'	Ključna riječ *dynamic* omogućuje da varijabla mijenja tip podatka u bilo koje vrijeme.

Dart, kao i svaki drugi objektno orijentirani jezik, podržava naredbe kontrole toka *(eng. control flow statements)* koje podrazumijevaju *if-else* i *switch-case* naredbe te *for, while* i *do-while* petlje. Jezik podržava klase, apstraktne klase i nasljeđivanje, no nije moguće eksplicitno definirati sučelje s ključnom riječi `interface`. Svaka klasa implicitno definira sučelje i druge klase ju mogu implementirati bez nasljeđivanje implementacije koristeći ključnu riječ `implements`. Za nasljeđivanje implementacije druge klase koristi se ključna riječ `extends` [7]. Novitet u klasama su skraćeni konstruktori, odnosno više nije potrebno pisati tijelo konstruktora u kojem vrijednosti pridružujemo varijablama klase [6]. Isječak kôda 1 prikazuje dva načina implementacije konstruktora. Ova dva konstruktora imaju istu svrhu, no u prvom konstruktoru programski jezik Dart za nas odrađuje posao pridruživanje vrijednosti varijablama klase. Razlika je, dakle, u broju linija koda i formatu metode.

Isječak kôda 1: Konstruktori u Dartu

```
class Student {
  String jmbag;
  String ime;
  String prezime;

  Person(this.jmbag, this.ime, this.prezime)
}

class Student {
  String jmbag;
  String ime;
  String prezime;

  Person(String jmbag, String ime, String prezime) {
    this.jmbag = jmbag;
```

```
    this.ime = ime;
    this.prezime = prezime;
  }
}
```

Nadalje, u programskom jeziku Dart ne postoje ključne riječi za definiranje vidljivosti članova klase, već su svi članovi prema zadanim postavkama javni *(eng. public).* Ako želimo da član klase bude privatni, tada ispred njegovog naziva stavljamo podvlaku *(eng. underscore)* [6].

Najvažnija promjena u odnosu na ostale objektno orijentirane jezike su funkcije, koje su u Dartu objekti i njihov tip podatka je `Function`. Ovo omogućuje da se funkcije dodijele varijablama ili da se proslijede drugim funkcijama u obliku parametra [6], [7]. Funkcijama je moguće definirati pozicijske parametre *(eng. positional parameters)*, što je uobičajeno u svim programskim jezicima, no moguće je definirati i imenovane parametre *(eng. named parameters)*. Imenovani parametri omogućuju da proslijedimo vrijednosti funkciji ne poštujući redoslijed, ali uz navođenje naziva parametara ispred tih vrijednosti [6]. Isječak kôda 2 prikazuje poziv funkcije s imenovanim parametrima.

Isječak kôda 2: Funkcija s imenovanim parametrima

```
upisiStudenta(jmbag: '0016129565', ime: 'Iva', prezime: 'Papac');
```

U konačnici, programski jezik Dart vrlo je sličan ostalim objektno orijentiranim jezicima, a u predstojećim poglavljima su navedene neke karakteristike po kojima se Dart razlikuje. Navedene promjene pospješuju čistoću programskog koda i omogućuju programerima čitljiviji kod.

3.1.2. Flutter i React Native

React Native je JavaScript programski okvir koji služi za razvoj mobilnih, *desktop* i *web* aplikacija. Temelji se na JavaScript biblioteci React koju je razvio Facebook. Za razvoj korisničkog sučelja koristi se JSX *(eng. JavaScript XML)* jezik označavanja, a riječ je o kombinaciji jezika XML *(eng. EXtensible Markup Language)* i JavaScript. React Native koristi „most“ koji komunicira s izvornim API-jima *(eng. Application Programming Interface)* s ciljem prikazivanja mobilnog korisničkog sučelja na izvornim platformama. Dakle, komponente korisničkog sučelja pretvaraju se u izvorne iOS ili Android komponente, a ne u *web* temeljene komponente [8].

U Wuovoj studiji slučaja [9] napravljena je usporedba iste aplikacije razvijene koristeći React Native i Flutter. Wu je usporedio navigaciju, kreiranje izgleda zaslona i oblikovanja komponenti korisničkog sučelja te performanse obje tehnologije. Unutar okvira React Native

ne postoji navigacija ili usmjeravanje, već je potrebno koristiti biblioteku *React Native Navigation.* S druge strane, Flutter ima ugrađenu navigaciju odnosno klase *Navigator* i *Router.* Što se tiče izrade korisničkog sučelja, obje tehnologije su podjednako dobre i podržavaju modularnost i višekratnu upotrebu komponenti. Za usporedbu performansi korišteni su sljedeći koncepti: broj sličica u sekundi *(eng. Frame per seconds, FPS),* vertikalno i horizontalno pomicanje zaslona *(eng. Scroll),* i brzina operacija pisanja i čitanja *(eng. DISK I/O)*. Što se tiče pomicanja zaslona, za obje tehnologije prosječni FPS iznosi 60. U nekim scenarijima FPS za React Native značajno opada jer se komponente korisničkog sučelja učitavaju u JavaScript dretvi, dok je FPS u Flutteru stabilan [9]. Nadalje, brzina operacija pisanja i čitanja podrazumijeva komunikaciju aplikacije i uređaja za dohvaćanje i pohranjivanje podataka. U ovoj kategoriji pobijedio je React Native imajući manje prosječno potrebno vrijeme za pohranu podataka od Fluttera [9].

Flutter i React Native proteklih godina uvelike su pridonijeli važnosti tehnologija za višeplatformski razvoj s obzirom na to da ubrzavaju proces postavljanja gotovog programskog proizvoda na tržište. Takav ubrzani proces razvoja programskih proizvoda je negativno utjecao na njihove performanse uspoređujući ih s performansama nativnih aplikacija [9].

3.1.3. Flutter i nativni razvoj mobilnih aplikacija

Olsson [10] je provela studiju slučaja s ciljem istraživanja razlika u razvoju mobilnih aplikacija koristeći Flutter, Kotlin i Swift. Nativni razvoj mobilnih aplikacija podrazumijeva razvoj mobilnih aplikacija koje su razvijene za određeni operativni sustav. Za Android platformu koristi se programski jezik Kotlin, dok se za iOS platformu koristi programski jezik Swift [10].

U navedenoj studiji slučaja uspoređivala se potrošnja procesora *(engl. central processing unit – CPU)* i broj linija koda potrebnih za kreiranje odgovarajućeg korisničkog sučelja. Kreirane su četiri mobilne aplikacije, a složenost i duljina programskog koda mjerila se prilikom revizije koda. Potrošnja procesora mjerila se ručno tri puta tako da su se zabilježile najmanje, srednje i najveće vrijednosti. Gledajući složenost i duljinu programskog koda, Flutter je u prednosti u odnosu na ostale programske jezike. Autorica mobilnih aplikacija također navodi da joj je bilo potrebno 12 sati za razvoj koristeći Kotlin, osam sati koristeći Swift i šest sati koristeći Flutter, a pritom nije imala iskustva u području razvoja mobilnih aplikacija. Što se tiče testiranje potrošnje procesora, ona nisu pokazala značajne razlike između Flutter, Android i iOS aplikacija [10].

Ova studija slučaja provedena je u suradnji s tvrtkom Consid iz Švedske, odnosno tvrtka je željela dobiti uvid o razlikama višeplatformskog i nativnog razvoja mobilnih aplikacija. U vrijeme provedbe studije slučajeva nisu postojali kvalitetni akademski radovi o programskom okviru Flutter, stoga se dovodi u pitanje valjanost postavljenih pitanja i hipoteza te odabranih metrika za usporedbu s obzirom na to da postoji manjak literature. Osim toga, autorica je

samostalno izradila aplikacije za studiju slučajeva bez prethodnog iskustva u području razvoja mobilnih aplikacija, što bi potencijalno moglo uzrokovati pogrešnu procjenu složenosti programskog koda i vremena potrebnog za izradu [10]. Rezultati bi bili objektivniji da ja autorica odabrala tri iskusna razvojna programera bez prethodnog znanja jednog od programskih jezika. Tako bi se izbjegla opasnost pogrešne procjene vremena potrebnog za izradu.

3.2. Elementi korisničkog sučelja

Elementi korisničkog sučelja u programskom okviru Flutter nazivaju se *widgets* te su temelj svakog korisničkog sučelja [6]. Pri razvoju korisničkog sučelja navedeni elementi slažu se u stablo pomoću kojega dobivamo raspored elemenata na zaslonu uređaja [5]. Prema Payneu [6], elementi korisničkog sučelja mogu se podijeliti na četiri kategorije – elementi za pohranu vrijednosti *(eng. value widgets)*, elementi za raspored *(eng. layout widgets)*, elementi za navigaciju *(eng. navigation widgets)* i ostali elementi korisničkog sučelja *(eng. other widgets)*. U tablici dolje izdvojeni su neki od najpoznatijih elemenata korisničkog sučelja po spomenutim kategorijama.

Tablica 3: Popis elemenata korisničkog sučelja prema kategoriji [6]

Kategorija	Elementi korisničkog sučelja
Elementi za pohranu vrijednosti	Form, FormField, Text, TextField, Slider, Switch, Checkbox, Radio, RichText, Tooltip, Image, FlatButton
Elementi za raspored	Card, Center, Column, Row, PageView, Padding, Scaffold, ListView, Expanded, Container, SizedBox, Stack, SnackBar
Elementi za navigaciju	MaterialApp, BottomNavigationaBar, AlertDialog, Navigator, Drawer, TabBar, TabBarView
Ostali elementi	GestureDetector, Theme, Cupertino, Transitions, Transforms

Prema Payneovoj procjeni, službena Flutter dokumentacija sadrži oko 160 elemenata korisničkog sučelja, stoga nisu svi navedeni u tablici iznad [6]. U nastavku poglavlja bit će opisani oni koji se najčešće koriste.

3.2.1. Elementi korisničkog sučelja sa stanjem i bez stanja

Sve elemente korisničkog sučelja je moguće kreirati bez stanja (eng. *stateless widgets)* ili sa stanjem *(eng. stateful widgets).* Kao što im i samo ime govori, elementi bez stanja ne održavaju vlastito stanje*,* dok elementi sa stanjem održavaju. Pod stanje razumijevamo sve podatke koji su kreirani i mijenjaju se tijekom životnog ciklusa određenog elementa korisničkog sučelja [6]. Životni ciklus započinje pozivanjem metode `initState()` i moguće ju je nadjačati ako se želi izvršiti određena akcija pri inicijalizaciji elementa korisničkog sučelja. Osim toga, postoji i metoda `dispose()` koja se poziva kada životni ciklus elementa završava. Tu metodu, također, je moguće nadjačati ako je potrebno zatvoriti resurse, kako ne bi došlo do curenja memorije [11].

3.2.2. Osnovni elementi korisničkog sučelja

Iako njegova svojstva nisu vidljiva na zaslonu uređaja, `MaterialApp` je najvažniji element korisničkog sučelja jer on omogućuje da definiramo ime aplikacije, temu uključujući paletu boja, font i veličinu teksta, rute i navigaciju te početni zaslon [6]. Sljedeći u nizu je `Scaffold` pomoću kojeg kreiramo osnovni raspored elemenata na zaslonu, a on se sastoji od alatne trake, tijela zaslona i navigacijske trake na lijevom ili donjem dijelu zaslona [5], [6]. Tijelo navedenog elementa je korisno omotati s drugim elementom pod nazivom `SafeArea`, koji automatski prilagođava svoj sadržaj svojstvima uređaja, s obzirom na to da većina današnjih uređaja na vrhu zaslona ima urez *(eng. notch)* [5], [6].

Uobičajeno je da na svakom zaslonu postoje više elemenata korisničkog sučelja raspoređenih jedan ispod ili pokraj drugog, što je moguće postići koristeći elemente `Row` ili `Column` [6]. Samo ime im govori da `Row` raspoređuje svoj sadržaj po horizontalnoj osi, dok `Column` raspoređuje po vertikalnoj osi uz odgovarajući razmak. Razmak i pozicija definiraju se uz pomoć parametara `CrossAxisAlignment` i `MainAxisAlignment`. Uz ova dva elementa korisničkog sučelja usko vežemo `ListView` i `PageView`. To su elementi istog formata kao `Column` i `Row`, jedino što oni još nude mogućnost vertikalnog ili horizontalnog pomicanja. Obično se koriste kada postoji dinamička lista elemenata za koju znamo da će biti veća od visine ili širine zaslona uređaja [4].

Vrlo se često element `Container` koristi za stilizaciju podređenog elementa, odnosno za definiranje boje, visine i širine, margina, rubova, poravnanja i drugih atributa. Ponekad se koristi i za dodavanje praznog prostora između dva elementa korisničkog sučelja [5]. Posljednji element je `GestureDetector`. On također nije vidljiv na korisničkom sučelju, a služi za detektiranje pokreta i upravljanje događajima. Pokreta ima mnogo, ali najčešće su to dodir, dvostruki dodir, dugi pritisak, vertikalno ili horizontalno pomicanje [6].

3.3. Upravljanje stanjem

Stanje odražava promjene nastale pri interakciji korisnika s aplikacijom [12]. Postoje dvije vrsta stanja – kratkotrajno *(eng. ephemeral)* stanje i stanje aplikacije. Kratkotrajno stanje, koje se često naziva UI ili lokalno stanje, označava podatke vezane uz jedan specifičan element korisničkog sučelja. Na primjer, odabrana kartica u navigacijskoj traci. Uobičajeno je da se za takve promjene ne koriste kompleksne tehnike upravljanja stanjem, već jednostavna ugrađena metoda `setState()`. S druge strane, stanje aplikacije nije kratkotrajno i pristupaju mu više komponenata i zaslona korisničkog sučelja, pa se često naziva i dijeljeno stanje. Na primjer, u dijeljeno stanje moguće je pohranjivati informacije o korisničkim preferencijama, sesiji, obavijestima i sl. Za upravljanje ovim stanjem potrebna je jedna od tehnika upravljanja, a nekoliko njih su objašnjene u nastavku poglavlja [11].

Upravljanje stanjem *(eng. state management)* služi za rukovanje interakcije korisnika s aplikacijom tako da osvježi korisničko sučelje, ažurira bazu podataka ili pošalje zahtjev prema odgovarajućem udaljenom servisu sukladno radnjama korisnika [12]. Vrlo je važno implementirati jednu od tehnika upravljanja stanjem u bilo koju Flutter aplikaciju jer time se postiže odvajanje sloja poslovne logike od korisničkog sučelja. To za posljedicu ima da se svaka funkcionalnost ispravno i precizno izvršava te da se poboljšava kvaliteta i čitljivost programskog koda [12]. Ispravna implementacija upravljanja stanjem podrazumijeva korištenje uzoraka dizajna, a najpopularniji među njima su MVVM *(eng. Model-View-ViewModel)*, MVC *(eng. Model-View-Controller)* i BLoC (*eng. Business Logic Component)*. Navedenim uzorcima dizajna želi se postići isti cilj kao i sa spomenutim tehnikama upravljanja stanjem, a to je odvajanje poslovne logike od korisničkog sučelja.

U uzorku dizajna MVVM poslovna logika odvaja se u logičke komponente posrednik podataka *(eng. ViewModel)* i model *(eng. Model)*, dok korisničko sučelje *(eng. View)* služi za prikazivanje podataka i upravljanje korisničkom interakcijom s aplikacijom. Model predstavlja „jedinstveni izvor istine“ *(eng. Single Source of Truth)* odnosno prosljeđuje dohvaćene podatke iz baze podataka. Posrednik podataka je posrednik između komponenti korisničkog sučelja i modela tako da prima korisničke događaje od komponente korisničkog sučelja i na temelju njih zahtijeva podatke od modela [13].

Drugi uzorak dizajna, MVC, razdvaja aplikaciju na tri logičke komponente – model *(eng. Model),* korisničko sučelje *(eng. View)* i kontroler *(eng. Controller).* Prve dvije logičke komponente model i korisničko sučelje imaju istu svrhu kao i u uzorku dizajna MVVM, dok kontroler služi za dohvaćanje podataka s udaljenih servisa i ažuriranje stanja aplikacije [14]. Gürel [14] predlaže kreiranje još jedne logičke komponente servisi *(eng. Services)* koja služi za dohvaćanje podataka s udaljenih servisa ili baza podataka, dok je kontroler onda zadužen samo za internu logiku aplikacije.

Treći uzorak dizajna, BLoC, detaljno je opisan u potpoglavlju 3.3.2. Iako je riječ o Flutter biblioteci, ona se temelji na spomenutom uzorku dizajna. Ključan pojam u ovom uzorku dizajna je tok *(eng. Stream)* koji prima, obrađuje i isporučuje podatke kao posljedicu interakcije korisnika s aplikacijom. Komponenta poslovne logike *(eng. Business Logic Component, BLoC)* prima i obrađuje korisničke događaje dobivene od komponenata korisničkog sučelja, ažurira stanje aplikacije i obavještava odgovarajuće komponente korisničkog sučelja o promjenama [15].

U službenoj Flutter dokumentaciji navedeno je desetak tehnika upravljanja stanjem, a neke od njih su *Provider, Riverpod, Redux, GetX* i *BLoC* [11]. Nijedna tehnika nije pogrešan odabir i svaki autor preporučuje drugu tehniku, stoga su u nastavku odabrane i opisane samo dvije – *Flutter BLoC* i *Provider*.

3.3.1. Provider

Payne [6] navodi da je paket *Provider,* koji je napisao Rémi Rousselet, robustan i jednostavan za korištenje. Ovaj paket koristi već ugrađenu Flutter klasu `ChangeNotifier` za pohranu stanja i obavještavanje korisničkog sučelja pri promjeni tog stanja [4]. Preporučena praksa za implementaciju upravljanja stanjem pomoću paketa *Provider* je sljedeća:

1. Prvi korak je kreiranje klase modela koja proširuje klasu `ChangeNotifier` i poziva metodu `notifyListeners()` pri svakoj promjeni stanja [4].
2. Drugi korak je omotavanje elementa korisničkog sučelja, koji osluškiva promjene stanja, s objektom klase `ChangeNotifierProvider` [4].
3. Treći korak je kreiranje instance klase modela korištenjem metode `Provider.of<T>` unutar metode `build`, koja je temeljna za kreiranje svakog elementa korisničkog sučelja. Slovo T u navedenoj metodi predstavlja klasu modela [4].

Miola [4] pruža uvid u jednostavan primjer korištenjem paketa *Provider* i navedenih triju koraka. Isječak kôda 3 prikazuje primjer klase modela proširene s klasom `ChangeNotifier`. Klasa sadrži privatnu varijablu `counter` i dvije metode `increment()` i `decrement()` koje služe se inkrementaciju i dekrementaciju varijable `counter`. Osim toga, navedene metode pozivaju metodu `notifyListeners()` [4].

Isječak kôda 3: Implementacija klase modela s paketom Provider [4]

```
class CounterModel with ChangeNotifier {
  int _counter = 0;
  void increment() {
    _counter++;
    notifyListeners();
  }

  void decrement() {
```

```
    _counter--;
    notifyListeners();
  }

  int get currentCount => _counter;
}
```

Nadalje, u korijenskom elementu, koji se poziva u metodi `main`, potrebno je kreirati instancu klase `ChangeNotifierProvider`. Ta instanca omogućuje da element korisničkog sučelja osluškuje promjene stanja odgovarajućeg klase modela [4]. Implementaciju je moguće vidjeti u primjeru Isječak kôda 4.

Isječak kôda 4: Implementacija metode main s paketom Provider [4]

```
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp();
  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (context) => CounterModel(),
      child: const DemoPage(),
    );
  }
}
```

Na kraju se kreira jedan element korisničkog sučelja koji koristi instancu klase modela, što je prikazano u primjeru Isječak kôda 5. Ako se pritisne gumb „+1“, tada će se pozvati metoda `increment()`, povećati vrijednost varijable `counter` i pozvati metoda `notifyListeners()` te će se na kraju ažurirati tekst koji prikazuje vrijednost. Isto vrijedi i za gumb „-1“, osim što će se pozvati metoda `decrement()` i vrijednost varijable `counter` će se smanjiti [4].

Isječak kôda 5: Implementacija elementa korisničkog sučelja s paketom Provider [4]

```
class DemoPage extends StatelessWidget {
  const DemoPage();
  @override
  Widget build(BuildContext context) {
    final counter = Provider.of<CounterModel>(context);

    return Scaffold(
        body: Center(
            child: Row(
      children: [
        FlatButton(
          child: const Text(
            "+1",
            style: TextStyle(color: Colors.green, fontSize: 25),
          ),
          onPressed: () => counter.increment(),
```

```
        ),
        Text(
          "${counter.currentCount}",
          style: const TextStyle(
            fontSize: 30,
          ),
        ),
        FlatButton(
          child: const Text(
            "-1",
            style: TextStyle(color: Colors.red, fontSize: 25),
          ),
          onPressed: () => counter.decrement(),
        )
      ],
    )));
  }
}
```

Koraci prikazani u primjerima iznad središnji su dio za svaku implementaciju upravljanja stanjem pomoću paketa *Provider*. Ovaj paket također podržava rad s asinkronim funkcijama i tipovima podataka [4].

3.3.2. Flutter BLoC

Drugi često korišteni paket za upravljanje stanjem je *The Flutter Bloc*, kojeg je razvio Felix Angelov. Kratica BLoC u nazivu označava komponentu poslovne logike *(eng. Business Logic Component, BLoC).* Svrha ovog paketa je da korisničke akcije i druge događaje pretvori u stanje. Taj proces se sastoji od tri osnovna koraka: element korisničkog sučelja obavještava *BLoC* klasu o događaju, zatim klasa obrađuje događaj i elementu vraća novo stanje, a element osluškiva promjenu stanja i osvježava se po potrebi [4].

Za prikaz navedenih koraka u praksi, koristit će se isti primjer s inkrementacijom i dekrementacijom vrijednosti varijable. Prvo se kreira *BLoC* klasa u kojoj su mogući događaji predstavljeni enumeracijom, a oni se prosljeđuju klasi na temelju čega se ažurira stanje. Primjer spomenute enumeracije i klase je moguće vidjeti u primjeru Isječak kôda 6.

Isječak kôda 6: Implementacija BLoC klasa [4]

```
enum CounterEvent { increment, decrement }

class CounterBloc extends Bloc<CounterEvent, int> {
  CounterBloc() : super(0);

  @override
  Stream<int> mapEventToState(CounterEvent event) async {
    switch (event) {
      case CounterEvent.increment:
        yield ++state;
        break;
      case CounterEvent.decrement:
        yield --state;
```

```
        break;
    }
  }
}
```

U korijenskom elementu korisničkog sučelja, koji se poziva u metodi *main*, potrebno je kreirati instancu klase `BlocProvider`. Ta instanca omogućuje da određeni element korisničkog sučelja osluškuje promjene stanja. Implementacija metode *main* prikazana je u primjeru Isječak kôda 7.

Isječak kôda 7: Implementacije metode main s paketom Flutter BLoC [4]

```
void main() => runApp(const MyApp());

class MyApp extends StatelessWidget {
  const MyApp();


  @override
  Widget build(BuildContext context) {
    return BlocProvider<CounterBloc>(
      create: (context) => CounterBloc(),
      child: const DemoPage();
    );
  }
}
```

Na kraju se kreira jedna instanca *BLoC* klase unutar odgovarajućeg elementa korisničkog sučelja. Ako se pritisne gumb „+1“, tada će se *BLoC* klasi proslijediti događaj `increment()`, povećati vrijednost varijable `counter` te će se na kraju ažurirati tekst koji prikazuje vrijednost. Isto vrijedi i za gumb „-1“, osim što će se proslijediti događaj `decrement()` i vrijednost varijable `counter` će se smanjiti. Navedeni koraci implementirani su u primjeru Isječak kôda 8.

Isječak kôda 8: Implementacija elementa korisničkog sučelja s paketom Flutter BLoC [4]

```
class DemoPage extends StatelessWidget {
  const DemoPage();

  @override
  Widget build(BuildContext context) {
    final counterBloc = context.bloc<CounterBloc>();

    return Scaffold(
      body: Center(
        child: Row(
          mainAxisAlignment: MainAxisAlignment.spaceAround,
          children: [
            FlatButton(
              child: const Text("+1",
                style: TextStyle(color: Colors.green, fontSize: 25)),
              onPressed: () => counterBloc.add(CounterEvent.increment),
            ),
            BlocBuilder<CounterEvent, int>(
```

```
              builder: (context, count) => Text("$count",
                style: const TextStyle(fontSize: 30)),
            ),
              FlatButton(
              child: const Text("-1",
                style: TextStyle(color: Colors.red, fontSize: 25)),
              onPressed: () => counterBloc.add(CounterEvent.decrement),
            ),
          ],
        )
      )
    );
  }
}
```

Ovaj jednostavan primjer daje pregled svih koraka potrebnih za implementaciju upravljanja stanjem koristeći paket *The Flutter BLoC*. Uz klasu `BlocBuilder` klasu moguće je koristiti `BlocListener` klasu koja omogućava da element korisničkog sučelja osluškuje promjenu stanja kako bi se okinuo odgovarajući događaj [4].

3.4. Navigacija i usmjeravanje

U službenoj Flutter dokumentaciji [11] navodi se da postoje dva ugrađena mehanizma za usmjeravanje, a to su elementi `Navigator` i `Router`. Za manje aplikacije `Navigator` je sasvim dovoljan, dok za složenije aplikacije se preporučuje kombinacija oba elementa [11]. Element `Navigator` služi za upravljanje stogom ruta kako bi se postiglo usmjeravanje i prijenos podataka s jednog na drugi zaslon. Najvažnije metode ovog elementa su: `push`, `pushNamed` i `pop` te je njima potrebno proslijediti argumente tipa `BuildContext` i/ili `Route`. Napoli nam daje i primjer programskog kod za poziv funkcije `push`, a prikazan je u primjeru Isječak kôda 9 [5]. Vidljivo je da je prvi parametar tipa `BuildContext`, a drugi tipa `MaterialPageRoute`. U objektu klase `Route` definiran je odredišni zaslon i vrsta zaslona pomoću atributa `fullscreenDialog`.

Isječak kôda 9: Implementacija funkcije *push* [5]

```
Navigator.push(
      context,
      MaterialPageRoute(
            fullscreenDialog: true,
            builder: (context) => About(),
      ),
);
```

Drugi način usmjeravanja moguće je postići korištenjem naziva rute zaslona i metode `pushedNamed`. Nazive ruta za pojedine zaslone moguće je definirati u elementu korisničkog sučelja `MaterialApp` pod atributom `routes`. Taj atribut prima mapu u kojoj je ključ naziv rute, a vrijednost element koji predstavlja zaslon sa svim svojim elementima. Tako je metodi `pushedNamed` potrebno proslijediti parametar tipa `BuildContext` i naziv rute [5].

U Flutteru postoje tri vrste navigacije, a to su navigacija stogom *(eng. stack navigation)*, navigacija bočnim izbornikom *(eng. drawer navigation)* i navigacija karticama *(eng. tab navigation)* [6]. Navigacija stogom objašnjena je u prethodnom odlomku.

Navigacija bočnim izbornikom pogodna je za aplikacije s brojnim navigacijskim rutama i dubokim stablom navigacije. Osim toga, ovime se štedi prostor zaslona odnosno korisničkog sučelja. Za kreiranje bočnog izbornika potrebno je koristiti element korisničkog sučelja `Drawer` kojeg je moguće dodati unutar elementa `Scaffold`. Payne nudi jednostavan primjer korištenja bočnog izbornika prikazanog u primjeru Isječak kôda 10 [6].

Isječak kôda 10: Implementacija bočnog izbornika [6]

```
Widget build(BuildContext context) {
      return Scaffold(
            appBar: AppBar(
                  title: const Text('Drawer Navigation'),
            ),
            body: const Text('Drawer Navigation'),
            drawer: Drawer(
                  child: ListView(
                        children: <Widget>[
                              Text('Option 1'),
                              Text('Option 2'),
                              Text('Option 3')],
                  )));
}
```

U primjeru je vidljivo da atribut `drawer` prima samo jedan element, stoga je potrebno proslijediti `Column` ili `ListView` kako bi bilo moguće definirati višestruke navigacijske rute. Prema izboru moguće je dodati i element `DrawerHeader` koji obično sadrži logo i podatke o aplikaciji [6].

Treći način je navigacija karticama koje se prikazuje na vrhu ili dnu zaslona uređaja, a za to su potrebni sljedeći elementi: `TabBar`, `TabBarView` i `TabController`. Svi elementi korisničkog sučelja, odnosno pojedini zasloni definiraju se u elementu `TabBarView` te se njihov prikaz izmjenjuje ovisno koju karticu je korisnik odabrao. Pomoću elemenata `TabBar` i `Tab` definiraju se kartice u traci te služe za preusmjeravanje korisnika na odabrani zaslon. Kako bi se sve navedeno ispravno prikazivalo na zaslonu, potrebno je definirati i upravljačku klasu koja rukuje korisničkim odabirom i usmjeravanjem. Tu klasu je moguće definirati pomoću atributa `controller` ili korijenski element `Scaffold` omotati s elementom `DefaultTabController` [5], [6]. Implementacija navigacije s trakom kartica prikazana je u primjeru Isječak kôda 11.

Isječak kôda 11: Implementacija trake kartica [6]

```
Widget build(BuildContext context) {
      return DefaultTabController(
            length: 3,
            child: Scaffold(
                  appBar: AppBar(
                  bottom: TabBar(
                        tabs: const <Widget>[
                              Tab(
                                    icon: Icon(Icons.looks_one),
                                    child:Text('Show A'),
                              ),
                              Tab(
                                    icon: Icon(Icons.looks_two),
                                    child:Text('Show B'),
                              ),
                              Tab(
                                    icon: Icon(Icons.looks_3),
                                    child: Text('Show C'),
                              ),
                        ]),
                  body: TabBarView(
                        children: <Widget>[
                              WidgetA(),
                              WidgetB(),
                              WidgetC(),
                        ]));
}
```

U primjerima iznad prikazane su moguće metode navigacije unutar aplikacija – navigacija stogom, navigacija bočnim izbornikom i navigacija karticama. Navedene metode mogu se međusobno kombinirati, no treba biti oprezan kako bi korisnik aplikacije uvijek imao mogućnost vratiti se na prethodni zaslon u stogu ili otići do željenog zaslona bočnim izbornikom [6]. Payne [6] preporučuje korištenje maksimalno dvije metode navigacije istovremeno, a najčešće su to navigacija stogom i karticama.

3.5. Rad s podacima

Rad s podacima, što često obuhvaća preuzimanje podataka s različitih udaljenih servisa, odnosno slanje prema različitim udaljenim servisima, zahtjeva rad s različitim formatima zapisa tih podataka. Najčešće korišteni formati su JSON *(eng. JavaScript Object Notation)* i XML *(eng. Extensible Markup Language).*

3.5.1. JSON

Najbolji pristup rada s JSON objektima je da se oni pretvore u instance odgovarajuće Dart klase modela, što je moguće učiniti ručno ili automatizirano koristeći određene Flutter pakete. Ručno pretvaranje podrazumijeva da varijablu koja sadrži JSON objekt pretvorimo u

mapu ključeva i vrijednosti pomoću funkcije `jsonDecode` iz paketa `dart:convert` [4]. Isječak kôda 12 prikazuje primjer korištenja spomenute funkcije.

Isječak kôda 12: Pretvaranje JSON objekta

```
import "dart:convert";

void main () {
  final json = '{ "jmbag": "0016129565", "ime": "Iva", "prezime":
"Papac"}';

  Map<String, dynamic> student = jsonDecode(json);
}
```

Sljedeći korak je pretvaranje dobivene mape u odgovarajuću klasu modela, a to je u ovom slučaju klasa `Student` prikazana u primjeru ispod. Dobivena mapa se pretvara u instancu klase `Student` tako da se pozove metoda `Student.fromJson(student)` koja je definirana u tijelu klase. Ako želimo slati podatke o studentu na neki udaljeni servis, uz pomoć metode `Student.toJson()`, također definirane u tijelu klase `Student`, instancu klase pretvaramo u mapu ključeva i vrijednosti, odnosno u JSON objekt [4]. Navedeni korak prikazan je u primjeru Isječak kôda 13.

Isječak kôda 13: Klasa s JSON metodama

```
class Student {
      final String jmbag;
      final String ime;
      final String prezime;

      Student._({
            required this.jmbag,
            required this.ime,
            required this.prezime
      });

      factory Student.fromJson(Map<String, dynamic> json) =>
            Student._(
                  jmbag: json["jmbag"],
                  ime: json["ime"],
                  prezime: json["prezime"],
            );

      Map<String, dynamic> toJson() => {
            "jmbag": jmbag,
            "ime": ime,
            "prezime": prezime,
      };
}
```

S druge strane, ovaj proces je moguće automatizirati, pogotovo ako je riječ o kompleksnim i ugniježđenim podacima. Prvo je potrebno dodati dva paketa: `json_annotation` i `json_serializable`. Navedeni paketi omogućuju da se metode `fromJson()` i `toJson()` automatski generiraju iz naredbenog retka, no potrebno je kreirati klasu modela s odgovarajućim anotacijama [4]. U primjeru Isječak kôda 14 vidljivo je da je potrebno klasu anotirati s `@JsonSerializable()` i definirati metode koje je potrebno generirati. Te metode će se kreirati u datoteci `student.g.dart` koju smo definirali iznad tijela klase [4].

Isječak kôda 14: Automatizacija implementacije klase s JSON metodama

```
import 'package:json_annotation/json_annotation.dart';
part 'student.g.dart';

@JsonSerializable()
class Student {
   final String jmbag;
   final String ime;
         final String prezime;

         Student(this.jmbag, this.ime, this.prezime);

         factory Student.fromJson(Map<String, dynamic> json) =>
_$StudentFromJson(json);

         Map<String, dynamic> toJson() => _$StudentToJson(this);
}
```

Nakon definiranja klase modela, potrebno je u naredbenom retku pokrenuti naredbu `$ flutter pub run build_runner build`. Ova naredba će pokrenuti generiranje datoteke `student.g.dart` koja sadrži implementacije metoda `fromJson()` i `toJson()` [4].

3.5.2. XML

Jezik za označavanje XML i dalje je među često korištenim jezicima za razmjenu podataka. Za Flutter aplikacije dostupan je paket `xml` kojeg je razvio Lukas Renggli, a funkcija koja se najčešće koristi je `XmlDocument.parse()`. Ova funkcija služi za pretvaranje String objekta u objekt klase `XmlDocument`. Ako navedeni String objekt nije u ispravnom XML formatu, tada navedena funkcija vraća iznimku [4]. Isječak kôda 15 sadrži primjer pretvorbe String objekta u objekt klase `XmlDocument`.

Isječak kôda 15: Pretvorba String objekta u XML objekt

```
void main() {
      var xmlString = "<studenti>
            <student>
                  <jmbag>0016129565</jmbag>
                  <ime>Iva</ime>
                  <prezime>Papac</prezime>
```

```
            </student>
      </studenti>";

      try {
            final xmlDoc = XmlDocument.parse(xmlString);
      } on XmlParserException catch (e){
            print(e);
      }
}
```

Za kreiranje String objekta u XML formatu potrebno je koristiti klasu `XmlBuilder` koja omogućuje ugnježđivanje podataka. Preporuka je da se za svaki element unutar korijenskog elementa kreira ponovno upotrebljiva funkcija radi bolje čitljivosti programskog koda [4]. U programskom kodu dolje nalazi se primjer kreiranja String objekta u XML formatu.

Isječak kôda 16: Implementacija String objekta u XML formatu

```
void dodajStudenta(XmlDocument.XmlBuilder builder, {
        required String jmbag,
  required String ime,
        required String prezime
}) {
        builder.element('student', nest: () {
        builder.element('jmbag', nest: jmbag);
        builder.element('ime', nest: ime);
        builder.element('prezime', nest: prezime);
       });
}

final builder = XmlBuilder();

builder.processing('xml', 'version="1.0"');
      builder.element('studenti', nest: () {
            dodajStudenta(builder,
                  jmbag: "0016129565“,
                  ime: „Iva“,
                  prezime: „Papac“);
            dodajStudenta(builder,
                  jmbag: „0123456789“,
                  ime: „Pero“,
                  prezime: „Perić“);
});

Final studentiXml = builder.build();
```

Klasa `XmlDocument` sadrži nekoliko metoda koje omogućuju iteraciju po instanci navedene klase. Prva metoda `findAllElements(String name)` vraća objekt tipa `Iterable` koji sadrži elemente s oznakom proslijeđenom u parametru navedene metode. Druga metoda `findElements()` vraća elemente podređene trenutno odabranom čvoru unutar hijerarhije XML objekta. Za odabir čvora moguće je koristiti ključne riječi `single`, `first` i `last` [4].

3.5.3. Korisničke preferencije

Korisničke preferencije podrazumijevaju razne osobne podatke koje želimo pohraniti u lokalnu memoriju. Na iOS uređajima lokacija tih podataka je `NSUserDefaults`, a na Android uređajima `SharedPreferences` [6]. U primjeru Isječak kôda 17 moguće je vidjeti kako koristiti i spremati korisničke preferencije u Flutter aplikacijama.

Isječak kôda 17: Korištenje korisničkih preferencija

```
SharedPreferences prefs = await SharedPreferences.getInstance();

prefs
 ..setString('imePrezime', 'Iva Papac')
 ..setBool('isStudent', true);

String imePrezime = prefs.getString('imePrezime');
bool isStudent = prefs.getBool('isStudent');
```

Za Flutter projekte postoji paket `shared_preferences` te on omogućuje čitanje i pisanje tih vrijednosti na različitim platformama. Navedeni paket koristi se tako da se prvo instancira objekt klase `SharedPreferences`. Za spremanje podataka koriste se `set` metode, dok se za čitanje podataka koriste `get` metode [5].

3.6. Rad s REST servisima

Važan segment u razvoju mobilnih aplikacija je komunikacija s REST *(eng. Representational State Transfer)* servisima koji održavaju vezu na bazu podataka ili bilo koji drugi oblik izvora podataka. Razvojni programeri šalju HTTP *(eng. Hypertext Transfer Protocol)* zahtjeve prema tim servisima s ciljem dohvaćanja podataka ili slanja podataka kako bi se pohranili u bazu. Obično su podaci zapisani u JSON obliku [4], [6]. Pri razvoju aplikacija najčešće se koriste ovih pet HTTP metoda za dohvaćanje i slanje podataka:

- GET – Metoda koja se koristi za dohvaćanje i čitanje podataka [6].
- DELETE – Metoda koja se koristi za brisanje podataka tako da se servisu proslijedi jedinstvena oznaka podatka koji se želi obrisati [6].
- POST – Metoda koja se koristi za pohranu novog podataka te kada se servisu šalje velika količina osjetljivih podataka [6].
- PUT – Metoda koja se koristi za zamjenu postojećih podataka s novim proslijeđenim podacima [6].
- PATCH – Metoda koja se koristi za ažuriranje postojećeg seta podataka [6].

Kako bi bilo moguće kreirati navedene zahtjeve, u Flutter projekt potrebno je dodati paket `http` kojeg je razvio službeni Dart tim [6].

3.6.1. GET zahtjevi

U poglavlju je već spomenuto da *GET* metoda služi za dohvaćanje podataka, ne zahtijeva definiranje tijela i najjednostavnija je. Na isti način funkcionira i *DELETE* metoda, no obično ne vraća nikakvu povratnu vrijednost [6]. Za potrebe sljedećeg primjera koristit će se testni podaci sa sljedeće poveznice: https://jsonplaceholder.typicode.com/. Na ovoj *web* stranici postoje 100 JSON zapisa koji će predstavljati REST servis [4].

Podaci se dohvaćaju uz pomoć metode `get()` koja vraća objekt `Future` odnosno riječ je o asinkronoj metodi, pa je potrebno pričekati rezultat koristeći ključnu riječ `await` [6]. U primjeru Isječak kôda 18 vidljivo je da se čeka rezultat servisa te se provjerava status zahtjeva. Ako je status zahtjeva 200, tada će se dobiveni podaci pretvoriti u mapu ključeva i vrijednosti te ista će se ispisati.

Isječak kôda 18: Implementacija GET zahtjeva

```
String url = 'https://jsonplaceholder.typicode.com/posts/1';
Response response = await get(url);

if(response.statusCode == 200) {
  Map<String, dynamic> post = json.decode(response.body);

        print(post['userId']);
  print(post['id']);
        print(post['title']);
        print(post['body']);
}
```

U primjeru također vidimo da se koriste ključne riječi `statusCode` i `body`. Prva služi za dobivanje HTTP statusa, a neki od njih mogu biti 200, 404 ili 500. Druga riječ služi za dohvaćanje sadržaja kojeg je vratio REST servis. Osim toga, moguće je dohvatiti duljinu tog sadržaja koristeći ključnu riječ `contentLength` te zaglavlje zahtjeva koje je poslao servis uz pomoć ključne riječi `headers` [4].

3.6.2. POST zahtjevi

Korištenje *POST*, *PUT* ili *PATCH* metoda pri slanju zahtjeva vrlo je slično onome što je prikazano u prethodnom primjeru. Razlika je u tome da tijelo zahtjeva mora sadržavati podatke, obično u JSON formatu, koje želimo poslati servisu [6].

Zahtjev se šalje tako da se pozove metoda `post()` kojoj se prosljeđuje poveznica, tijelo *(eng. body)* i zaglavlje *(eng. headers)*. Navedena metoda je također asinkrona, stoga je potrebno pričekati rezultat koristeći ključnu riječ `await` [4], [6].

Isječak kôda 19: Implementacija POST zahtjeva

```
Map<String, String> headers = {'Content-Type': 'application/json'};
String payload = '{"userId": 1, "id": 101, "title": "New post", "body":
"New post body"}';

Response = await post(url, headers: headers, body: payload);
```

U primjeru Isječak kôda 19 prvo je definirano zaglavlje u obliku mape ključeva i vrijednosti, a nakon toga i sadržaj u obliku JSON objekta. Sve navedeno se prosljeđuje u obliku parametara metodi `post()`.

4. DevOps

Termin DevOps prvi puta se javlja 2009. godine na konferenciji *DevOps Days* održanoj u Belgiji, a populariziran je nakon objave knjige *The Phoenix Project* 2013. godine. Autori knjige su Gene Kim, Kevin Behr i George Stafford, a opisali su principe DevOps-a u obliku izmišljenih likova [16], [17]. Ne postoji jedinstvena definicija pojma DevOps i različiti autori na različite načine ga definiraju. Bass, Weber i Zhu [18] navode da je DevOps skup praksi kojima se nastoji smanjiti vrijeme između razvoja i isporuke programskog proizvoda osiguravajući visoku kvalitetu. Vidljivo je da definicija ne govori o DevOps praksama i alatima jer autori navedene definicije smatraju da bilo koje prakse koje smanjuju vrijeme između razvoja i isporuke programskog proizvoda su DevOps prakse. Naglasak je i na visokoj kvaliteti razvoja i isporuke programskih proizvoda koju je moguće osigurati korištenjem automatiziranih testova, testiranjem u produkcijskoj okolini u kontroliranim uvjetima i nadziranjem novouvedenih funkcionalnosti. Autori navode da se DevOps prakse ne sastoje samo od testiranja i isporuke programskih proizvoda, već da je bitno osigurati kvalitetu od početka životnog ciklusa sustava počevši od planiranja [18].

Davis i Daniels [17] navode da je DevOps kulturološki pokret koji nastoji poboljšati proces razvoja programskog proizvoda i rad ljudi uključenih u taj proces. Pojam je nastao kombinacijom riječi razvoj *(eng. development)* i operacije *(eng. operations)*, no DevOps uključuje mnogo više od samih razvojnih i operacijskih timova. Navedeni autori pojam DevOps objašnjavaju pomoću čestih zabluda vezanih uz njega, odnosno nabrajajući sve ono što DevOps nije [17]. U nastavku je detaljnije opisan DevOps:

- U DevOps nisu uključeni samo razvojni programeri i sistemski inženjeri, već je potrebno uključiti sve osobe i timove koje pridonose poboljšanju učinkovitosti u procesima planiranja, razvoja i isporuke programskih proizvoda [17].
- Kreiranje DevOps tima nije dovoljno za kreiranje DevOps kulture, no može poslužiti kao kratkoročna strategija za uvođenje promjena. S vremenom, članovi DevOps tima vraćaju se u stare, tradicionalne timove s novim ulogama [17].
- DevOps nije radna pozicija, iako postoje programeri sa znanjem u području sistemskog inženjerstva i sistemski inženjeri koji znaju programirati. Jedan zaposlenik ne smije obavljati posao i sistemskog inženjera i programera jer s vremenom se ne postiže učinkovitost i visoka kvaliteta rada. Kako bi bio učinkovit, potrebno je da svi u organizaciji koriste prakse DevOps-a [17].
- Implementacija DevOps-a ne traje određeni broj dana, tjedana ili mjeseci, već je to stanje u tijeku. Dakle, to stanje nije lako odredivo ili mjerljivo jer je teško odrediti

koliko će određene kulturološke promjene trajati i koliko dugo je zaposlenicima potrebno da te promjene prihvate [17].

- DevOps ne podrazumijeva automatizaciju svakog posla ili zadatka u procesu razvoja programskog proizvoda. Ako automatizacija određenih poslova pomaže zaposlenicima da rade učinkovitije, tada se automatizacija preporučuje. S druge strane, nema potrebe za automatizacijom ako se više vremena potroši na pokušaj automatizacije određenog posla, nego ušteda vremena koja se dobije automatizacijom [17].

U nastavku poglavlja dodatno će se pojasniti pojam DevOps te njegove najvažniji principi i prakse. Osim toga, navedene su DevOps prakse i njihova uloga u procesu razvoja programskog proizvoda.

4.1. Osnovne značajke

U ovom potpoglavlju opisani su temeljni principi DevOps procesa koja se temelje na tri puta – načelo toka od razvoja do operacija, načelo povratnih informacija i načelo kontinuiranog učenja i eksperimentiranja [19]. U posebnom potpoglavlju je opisana i struktura DevOps tima.

4.1.1. Temeljni principi

Prvi put, načelo toka od razvoja do operacija, podrazumijeva unaprjeđenje toka rada od razvoja do operacija kako bi klijentima programski proizvod bio isporučen što prije. Taj tok roda vizualno je prikazan na slici dolje, a uključuje prakse procesa kontinuirane integracije, testiranja i isporuke programskih proizvoda [19].

Slika 1: Načelo toka od razvoja do operacija [19]

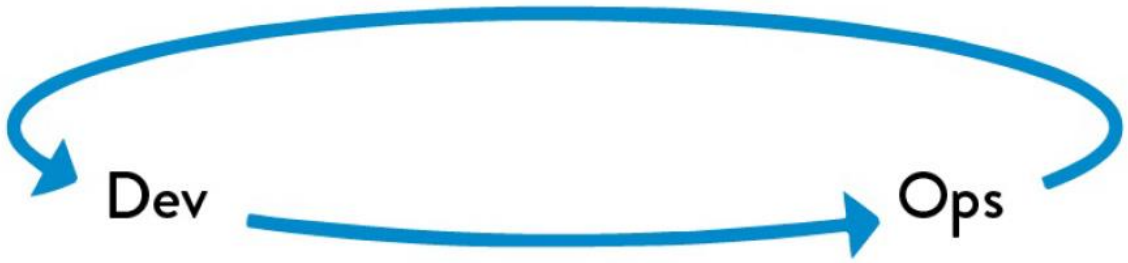
Cilj tih praksi je smanjiti vrijeme potrebno za razvoj novih funkcionalnosti te povećati pouzdanost i kvalitetu sustava [19]. Ovaj put se sastoji od šest principa:

- Prvi princip „Napravi posao vidljivim“ *(eng. Make our work visible)* podrazumijeva vizualno prikazivanje zadataka u procesu razvoja i isporuke programskog proizvoda koristeći kanban ploče ili ploče za planiranje sprinta. One obično sadrže

tri kolone s popisom zadataka koje je potrebno riješiti, koji se trenutno izvode ili koji su riješeni [19].

- Drugi princip „Ograniči količinu zadataka u izvođenju“ *(eng. Limit work in process)* odnosi se na ograničavanje zadataka u izvođenju kako bi se izbjegao paralelni, višestruki rad na projektima. Za ograničavanje rada, također, se mogu koristiti vizualne ploče tako da se odredi koliko zadataka najviše smije biti u koloni s popisom zadataka koji se trenutno izvode. Osim toga, ovo omogućuje brzo prepoznavanje problema u slučaju nemogućnosti završetka određenih zadataka [19].
- Treći princip „Smanji veličinu isporuke na produkciju“ *(eng. Reduce batch sizes)* podrazumijeva kontinuiranu isporuku manjih dijelova programskog koda na produkciju s ciljem povećanja kvalitete i smanjenja vremena isporuke programskog proizvoda. Što je veća promjena, teže će biti otkriti i otkloniti grešku u produkciji i na kraju povećat će se vrijeme ponovne isporuke [19].
- Četvrti princip „Smanji prosljeđivanje zadataka između timova“ *(eng. Reduce the number of handoffs)* odnosi se na zadatke koji omogućuju isporuku programskog koda s lokalnog repozitoriju u produkcijsku okolinu. Vrijeme tog procesa odnosno prosljeđivanje tih zadataka između različitih timova je moguće smanjiti pomoću automatizacije određenih aktivnosti ili pomoću reorganizacije timova [19].
- Peti princip „Kontinuirano identificiraj ograničenja sustava“ *(eng. Continually identify and elevate our constraints)* podrazumijeva identifikaciju procesa koji ograničavaju bržu isporuku klijentima. U DevOps procesu to su često ograničenja vezana uz kreiranje okoline, isporuke programskog koda, postavljanje i izvršavanje testova te uz nisku autonomiju razvojnih programera pri objavljivanju promjena [19].
- Šesti princip „Eliminiraj poteškoće i višak u vrijednosnom toku“ *(eng. Eliminate hardships and waste in the value stream)* odnosi se na otklanjanje svega što uzrokuje sporu isporuku programskog proizvoda klijentu. Najčešće poteškoće koje se javljaju su nedovršeni zadaci i njihova česta promjena, nepotrebni procesi i funkcionalnosti, čekanje, greške u programskom kodu i sl. [19].

Drugi put, načelo povratne informacije, podrazumijeva brz protok povratnih informacija od klijenata do operacija, razvoja i na kraju do cjelokupne organizacije omogućavajući stvaranje sigurnog i kvalitetnog sustava. Time se stvara petlja brzih i kontinuiranih povratnih informacija od razvoja do operacija kao što je vidljivo na slici dolje [19].



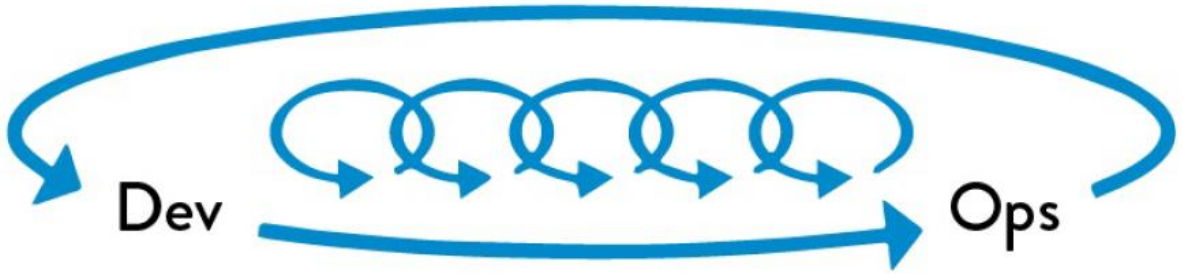
Slika 2: Načelo povratne informacije [19]

Cilj je stvoriti kvalitetni sustav povratnih informacija na početku razvojnog ciklusa te takvim pristupom omogućiti brzo uočavanje i otklanjanje manjih grešaka prije nastanka katastrofe [19]. Ovaj put se sastoji od pet principa:

- Prvi princip „Siguran rad u složenim sustavima“ *(eng. Working safely within complex systems)* podrazumijeva zadovoljavanje kriterija koji će omogućiti da se greške prepoznaju i otklone na vrijeme jer u složenim sustavima one su neizbježne. Kriteriji su upravljanje složenim poslovima, prepoznavanje i rješavanje problema, korištenje stečenog znanje u cjelokupnoj organizaciji te kontinuirano obučavanje novih voditelja [19].
- Drugi princip „Prepoznavanje grešaka na vrijeme“ *(eng. See problems as they occur)* zahtijeva od svih sudionika DevOps procesa da kontinuirano traže pogreške na svim mogućim područjima. Cilj takvog pristupa je omogućiti brži i bolji protok informacija na temelju kojih se mogu prepoznati i riješiti problemi te povećati otpornost i agilnost sustava [19].
- Treći princip „Rješavanje problema“ *(eng. Swarm and solve problems to build new knowledge)* podrazumijeva da se problemi moraju pokušati riješiti neposredno nakon što su isti prepoznati. Cilj ovakvog pristupa je riješiti problem prije nego se poveća i spriječiti nastavak rada što bi uzrokovalo nove, složenije probleme [19].
- Četvrti princip „Približi kontrolu kvalitete izvoru“ *(eng. Keep pushing quality closer to the source)* nam govori da briga o kvaliteti mora biti odgovornost svih, a ne samo jednog odjela. Tako kontrola kvalitete provodi se gdje se posao i izvodi, a donesene odluke su kvalitetnije što su bliže izvoru [19].

- Peti princip „Olakšaj drugima budući rad“ *(eng. Enable optimizing for downstream work centers)* podrazumijeva da prilikom dizajna programskog proizvoda uvažimo i zahtjeve zaposlenika, a ne samo klijenata. Takvim pristupom olakšava se rad i povećava kvaliteta u sljedećim fazama procesa razvoja programskog proizvoda [19].

Treći put, načelo kontinuiranog učenja i eksperimentiranja, bavi se kreiranjem organizacijske kulture koja potiče kontinuirano učenje i eksperimentiranje. Omogućuje stvaranje individualnog znanja koje se kasnije može pretvoriti u znanje tima ili organizacije. Slika 3 vizualno prikazuje kakav tok informacija i znanja se želi implementirati od razvoja do operacija. Dakle, cilj principa trećeg puta je kontinuirano kratiti i poboljšavati petlje povratnih informacija kako bi se stvorio siguran i niskorizičan sustav rada [19].



Slika 3: Načelo kontinuiranog učenja i eksperimentiranja [19]

Takva kultura potiče svakodnevno poboljšavanje posla, uvođenje stresa u svrhu individualnog napretka te simuliranje grešaka s ciljem poboljšanja otpornosti [19]. Ovaj put se sastoji od pet principa:

- Prvi princip „Kreiraj organizacijsku kulturu koja potiče učenje i daje osjećaj sigurnosti“ *(eng. Enabling organizational learning and a safety culture)* podrazumijeva kreiranje sigurne okoline u kojoj zaposlenici ne osjećaju strah, prijavljuju pogreške u radu i na njima uče. Takva kultura se naziva generativna, a karakterizira ju to da u slučaju ljudske pogreške ne traži se krivac, već se traži način kako spriječiti da se greška ponovi. Najčešće je riječ o analizama pomoću kojih se želi doći do mjera kako poboljšati sustav i brže otkriti greške [19].
- Drugi princip „Svakodnevno poboljšavaj posao“ *(eng. Institutionalize the improvement of daily work)* govori da svaki zaposlenik mora izdvojiti vrijeme unutar radnog dana kako bi riješio probleme unutar svoje domene. Rezultat takve prakse je rješavanje problema na vrijeme i mogućnost rješavanje manje očitih problema čime se poboljšava kvaliteta i sigurnost [19].

- Treći princip „Pretvori lokalno znanje u globalno“ *(eng. Transform local discoveries into global improvements)* podrazumijeva stvaranje mehanizama za pretvaranje znanja jednog tima ili eksperata u znanje organizacije. Neki od primjera takvih mehanizama su mogućnost pretraživanja izvještaja o problemima s kojima su se drugi susreli ili zajednički repozitorij programskog koda koji predstavlja najbolje što organizacija ima. Dakle, cilj je svim zaposlenicima omogućiti pristup individualnim znanjima unutar organizacije [19].
- Četvrti princip „Povećaj otpornost svakodnevnog posla“ *(eng. Inject resilience patterns into our daily work)* podrazumijeva uvođenje stresora u svakodnevne zadatke s ciljem smanjivanja vremena implementacije, povećanja pokrivenosti programskog koda s testovima i smanjivanje vremena izvršavanja testova. Rezultat takvih praksi je povećanje produktivnosti razvojnih programera i pouzdanosti sustava [19].
- Peti princip „Voditelji potiču kulturu učenja“ *(eng. Leaders reinforce a learning culture)* govori o odgovornostima voditelja u DevOps procesu, a najvažnija je kreiranje uvjeta vlastitome tima kako bi uspješno izvršavali dnevne zadatke, rješavali probleme i ostvarili ciljeve. Ovaj pristup od voditelja zahtjeva da svoje članove tima uči kako razmišljati i rješavati probleme [19].

Opisana tri načela daju uvid u DevOps principe te svaki od njih je važan za kvalitetnu implementaciju DevOps kulture u organizaciju. Principi načela toka od razvoja do operacija ključni su za postizanje DevOps rezultata, dok su principi načela povratne informacije važni za postizanje visoke pouzdanosti, kvalitete i sigurnosti. Principi načela kontinuiranog učenja i eksperimentiranja naglašavaju važnost organizacijskog učenja prihvaćajući pojavu problema u složenim sustavima i otvoreno razgovarajući o njima [19].

4.1.2. DevOps tim

U uvodu ovog poglavlja spomenuto je da kreiranje DevOps tima nije dovoljno za kreiranje DevOps kulture, već je potrebno uključiti sve osobe koje pridonose u procesima planiranja, razvoja i isporuke programskih proizvoda [17]. Sukladno tome, pod koncept DevOps tima razumijeva se razvojni tim s DevOps odgovornostima [18].

Jedno od uobičajenih pravila je, bez obzira na korištenu metodologiju, da tim ne smije biti velik. Prednost manjih timova je da na dnevnim sastancima svi članovi tima mogu iznijeti vlastito mišljenje o određenom problemu u kratkom vremenu, a posljedično voditelj tima može brzo donositi odluke. Osim toga, manji timovi su koherentnije jedinice unutar kojeg svi razumiju i podržavaju iste ciljeve. S druge strane, manji timovi rade na manjim dijelovima programskog koda i ne mogu se nositi s velikim zadacima. Takav način rada zahtijeva da se zadaci podijele

na manje zadatke i rasporede na više timova, a to također zahtijeva suradnju između timova [18].

Bass, Weber i Zhu [18] navode da DevOps tim sadrži dvije uloge agilnih timova, voditelj tima i članovi tima te dodatne uloga koje su bitne za DevOps, vlasnik usluge *(eng. Service owner)*, inženjer pouzdanosti *(eng. Reliability engineer)*, vratar *(eng. Gatekeeper)* i DevOps inženjer *(eng. DevOps engineer).* U tablici dolje opisane su spomenute uloge.

Tablica 4: Uloge u DevOps timu [18]

Uloga	Opis
Voditelj tima	Zadatak voditelja je olakšati rad članovima tima tako da pribavlja resurse, štiti ih od problema i u slučaju pojave ih rješava. Voditelj mora imati meke vještine *(eng. soft skills)* i vještine upravljanja projektom *(eng. project management skills)*.
Član tima	Članovi tima ili razvojni programeri zaduženi su za razvoj i isporuku programskih proizvoda, uključujući aktivnosti modeliranja, programiranja, testiranja i isporuke.
Vlasnik usluge	Vlasnik usluge sudjeluje aktivnostima vezanima uz cjelokupni životni ciklus sustava, određuje prioritet zadataka te je zadužen za komunikaciju s timom i klijentima.
Inženjer pouzdanosti	Inženjer pouzdanosti ima nekoliko zaduženja, a prva kreće neposredno prije isporuke programskog proizvoda. Nakon isporuke predstavlja kontakt za prijavu problema u produkciji. Ako postoji problem, provodi analizu kako bi dijagnosticirao i riješio problem uz pomoć ostatka tima i automatiziranih alata.
Vratar	Odgovornost vratara je koordinacija isporuke programskog proizvoda, odnosno odlučuje smije li određena verzija programskog koda nastaviti u sljedeći korak procesa isporuke. Neki od tih koraka su verzioniranje koda u glavnom repozitoriju, jedinično testiranje, integracijsko testiranje, isporuka na produkcijsku okolinu i sl.
DevOps inženjer	DevOps inženjer zadužen ja za alate koji se koriste u DevOps procesu. Drugim riječima, ova osoba zadužena je za odabir i prilagodbu alata potrebama tima. Ova uloga je ključna pri uvođenju automatizacije u procese implementacije i isporuke programskih proizvoda.

Pojedinci u timu mogu imati više uloga, dok se jedna uloga ne može podijeliti na više pojedinaca. Pri odabiru uloga potrebno je razmotriti vještine i radno opterećenje članova tima te vještine i količinu rada za svaku od uloga [18].

4.2. DevOps u procesu razvoja programskog proizvoda

U prethodnom potpoglavlju opisani su temeljni principi DevOps-a, a u ovom će biti opisana njihova primjena u praksi, odnosno primjena DevOps praksi u procesu razvoja programskog proizvoda. Cilj je stvoriti uvjete za održavanje brzog toka rada od razvoja do

operacija bez izazivanja problema u produkcijskoj okolini [19]. Te uvjete je moguće kreirati uz pomoću implementacije seta praksi pod nazivom kontinuirana isporuka *(eng. continuous delivery),* a prakse su sljedeće:

- kreiranje temelja za razvojni proces,
- brzo i pouzdano automatizirano testiranje,
- kontinuirana integracija, i
- automatizacija i arhitektura za niskorizičnu isporuku [19].

Kontinuirana isporuka temelji se na konceptu da se kvalitetan programski proizvod može brzo dizajnirati, izraditi, testirati i isporučiti, a uključuje kreiranje skripte za automatizaciju isporuke u produkcijsku okolinu, kontinuirano testiranje, dnevno verzioniranje programskog koda i projektiranje produkcijske okoline za niskorizičnu isporuku [19], [20]. Implementacija spomenutih praksi u proces razvoja programskog proizvoda omogućuje brzu isporuku u produkcijsku okolinu, kontinuirano testiranje, brze povratne informacije o kvaliteti rada te dnevnu isporuku manjih dijelova programskog koda u produkcijsku okolinu [19].

4.2.1. Kreiranje temelja za razvojni proces

Temelji u ovom slučaju podrazumijevaju automatizirano kreiranje razvojnih, testnih i drugih okolina sličnima produkcijskoj okolini. Prije DevOps praksi razvojni programeri bi tek isporukom u produkciju postali svjesni kako se zaista programski proizvod ponaša, a tada je prekasno za otklanjanje velikih grešaka. Potrebno je kreirati mehanizme koji će kreirati potrebne okoline na zahtjev, proširiti upotrebu verzioniranja programskog koda i osigurati da razvojni programeri u bilo kojem trenutku razvojnog procesa mogu testirati programski kod u okolini sličnoj produkciji. Okolinu na zahtjev moguće je implementirati kreiranjem zajedničkog mehanizma koji kreira razvojnu, testnu i produkcijsku okolinu, ali te okoline moraju biti stabilne, sigurne i niskorizične. Taj proces je moguće i automatizirati na nekoliko načina – kopiranjem virtualne okoline, upotrebom alata za upravljanje konfiguracijom ili alata za automatizirano upravljanje operativnim sustavom, kreiranjem okoline uz pomoć seta kontejnera i kreiranjem okoline u javnom ili privatnom oblaku [19].

Jedan od temelja je i postavljanje cjelokupnog sustava, uključujući programski kod i okolinu, u repozitorij kao dio verzioniranja. Takva praksa omogućuje da se redovito i pouzdano rekreira cjelokupni sustav i njegovi artefakti uključujući programski kod te produkcijsku i druge okoline. Osim toga, verzioniranje predstavlja jednu vrstu komunikacije između razvojnih i operacijskih timova. Nadalje, rekreiranje cjelokupnog sustava je omogućeno putem kreiranja okolina na zahtjev. Time se izbjegava dugotrajno popravljanje sustava usred nastanka problema. S druge strane, sve okoline moraju biti međusobno usklađene i konzistentne, što je

ostvarivo uz pomoć automatiziranih alata za upravljanje konfiguracijom kao što su Puppet i Chef [19].

Sve spomenute prakse ne bi imale smisla bez da u potpunosti provjerimo valjanost programskog proizvoda prije isporuke u produkciju. Pod potpunu provjeru valjanosti razumijevamo da je programski kod uspješno testiran, izgrađen i isporučen te se ponaša u skladu s očekivanjima u okolini sličnoj produkciji. Dakle, posao razvojnog programera se ne smatra gotovim ako spomenuti uvjeti nisu zadovoljeni [19].

4.2.2. Brzo i pouzdano automatizirano testiranje

Brzo i pouzdano automatizirano testiranje temelj je za implementaciju kontinuirane integracije i kontinuiranog testiranja. Cilj ove prakse je implementirati automatizirano testiranje u svakodnevni rad razvojnih programera te tako rano otkriti i otkloniti greške, i osigurati kvalitetu proizvoda. Automatizirano testiranje postiže se implementacijom cjevovodi za isporuku *(eng. deployment pipeline)* koja je zadužena da se automatski izgradi i testira novouvedeni programski kod u okolini sličnoj produkciji. Skripta se pokreće nakon što razvojni programer isporuči programski kod. Automatski pokreće jedinične testove, provodi statičku analizu koda te analizu redundantnosti i pokrivenosti testom. Ako su rezultati testova i analiza pozitivni, tada se programski kod automatski isporučuje na okolinu sličnoj produkciji i provode se testovi prihvatljivosti [19].

Drugi korak ove prakse je implementirati brze automatizirane testove prilikom svake promjene programskog koda u razvojnoj i testnoj okolini. Najčešće se automatiziraju jedinični i integracijski testovi te testovi prihvatljivosti. Cilj je što ranije otkriti greške u razvojnom procesu, što je jedan od razloga zašto se manji, brži i automatizirani testovi izvode prije ručnih testiranja. Najidealniji slučaj je otkrivanje većine grešaka pomoću jediničnih testova jer tako razvojni programeri najbrže dolaze do povratne informacije. Osim toga, greške otkrivene tijekom testiranja prihvatljivosti i integracijskih testiranja teško je reproducirati i samim time dugotrajno za otkloniti. Neke od preporuka za brže testiranje su implementacija praksi razvoja vođenim testom, paralelno izvođenje testova i automatizacija što više ručnih testiranja [19].

Treći korak postavlja pitanje što učiniti ako testovi nisu uspješni, a odgovor je zaustaviti razvojni proces sve dok se ne otklone otkriveni problemi. Ovaj potez se čini ekstremnim, no sprječava nastanak većih problema kasnije u razvojnom procesu i sprječava da bilo koja od okolina postane nepouzdana. Preporučuje se povećati vidljivost neuspjeha automatiziranih testova kako bi timovi znali u kojim slučajevima određena verzija programa ili provedeni testovi nisu uspješni [19].

4.2.3. Kontinuirana integracija

Prethodne dvije prakse naučile su nas da je potrebno dodijeliti više odgovornosti razvojnim programerima u razvojnom procesu, no njihov rad ne smije biti predugo izoliran jer tada je teže međusobno integrirati velike promjene u programskom kodu. Loša integracija uzrokuje ručno spajanje svih promjena u programskom kodu i prekida prirodan tok automatiziranih i ručnih testova. Takvim pristupom sprječava se kontinuirana isporuka manjih dijelova programskog koda na glavnu granu, a to također povlači izbjegavanje testiranja sve do kraja razvojnog procesa. Ove probleme rješava kontinuirana integracija, jedna od ključnih praksi koja omogućuje brz tok rada u ciklusu razvoja programskog proizvoda. Implementacija kontinuirane integracije postiže se prihvaćanjem sljedećih praksi:

- smanji obujam razvoja programskog koda,
- povećaj učestalost isporuke programskog koda, i
- razvoj temeljen na glavnoj grani *(eng. Trunk-based development).*

Voditelji timova moraju inzistirati na redovitoj isporuci i spajanju programskog koda te agresivnom refaktoriranju programskog koda kako bi on bio održiv i kako bi se kasnije jednostavnije implementirale nove funkcionalnosti [19].

Razvoj temeljen na glavnoj grani podrazumijeva da razvojni programeri barem jednom dnevnom isporuče promjene u programskom kodu na glavnu granu *(eng. master, main branch),* a ne da se te promjene gomilaju na njihovoj izoliranoj razvojnoj grani *(eng. dev branch)*. Redovitom isporukom promjena na glavnu granu postiže se kontinuirana integracija, odnosno kontinuirano spajanje manjih dijelova programskog koda [19].

4.2.4. Automatizacija i arhitektura za niskorizičnu isporuku

Za ostvarenje niskorizične isporuke programskog koda u produkciju potrebno je slijediti prakse kontinuirane integracije, a proces isporuke mora biti automatiziran, ponovljiv i predvidiv. Za automatizaciju procesa isporuke potrebno je automatizirati sljedeće aktivnosti procesa: pakiranje programskog koda pogodnog za isporuku, kreiranje unaprijed konfiguriranih kontejnera, isporuka i konfiguracija međuprograma *(eng. middleware)*, kopiranje paketa, biblioteka ili drugih datoteka u produkcijsku okolinu, ponovno pokretanje poslužitelja, aplikacija ili servisa, generiranje konfiguracijskih datoteka prema predlošcima, pokretanje testova koji provjeravaju ispravnost sustava i njegove konfiguracije, pokretanje testova, i migracija baze podataka. Automatizacija ovih koraka pozitivno utječe na procese izgradnje, testiranja i isporuke programskog proizvoda [19].

Nakon automatizacije procesa isporuke programskog koda, slijedi integracija navedenog procesa u proces isporuke na produkcijsku okolinu. Prije automatizacije procesa isporuke na produkcijsku okolinu, potrebno je zadovoljiti sljedeće uvjete, odnosno

implementirati sljedeće prakse: paketi kreirani tijekom procesa kontinuirane integracije moraju biti pogodni za isporuku u produkciju, produkcijska okolina mora biti spremna, automatska isporuka prikladnog programskog koda na produkciju, pohrana pokrenutih naredba i njihov rezultat, osoba koje su ih pokrenule i kontejnera na kojima su pokrenute, testovi koji provjeravaju ispravnost sustava i ispravnost postavljenih konfiguracija, i brza povratna informacija o rezultatu isporuke na produkciju [19].

Za bržu, niskorizičnu isporuku važno je odvojiti proces isporuke od procesa isporuke programskog proizvoda korisnicima. Proces isporuke ne mora biti nužno povezan s isporukom programskog proizvoda korisnicima, već to podrazumijeva instalaciju određene verzije programskog proizvoda na testnu, produkcijsku ili drugu proizvoljnu okolinu. S druge strane, isporuka programskog proizvoda korisnicima, tzv. izdanje nove verzije, podrazumijeva postavljanje seta novih funkcionalnosti javnima i dostupnima za korištenje. Ovdje se pojavljuje problem nemogućnosti izdavanja novog seta funkcionalnosti na vrijeme, no rizike od kašnjenja moguće je umanjiti primjenom jednog od uzorka izdavanja:

- Uzorci izdavanja temeljeni na okolini podrazumijevaju isporuku na dvije produkcijske okoline, ali samo jednoj od tih okolina imaju pristup korisnici. Takvim pristupom smanjuje se rizik od kašnjenja isporuke korisnicima jer sada ona ne ovisi o isporuci programskog proizvoda na produkcijsku okolinu.
- Uzorci izdavanja temeljeni na aplikaciji podrazumijeva prilagodbu aplikacije tako da se selektivno objavljuju odabrane funkcionalnosti aplikacije. Takav pristup omogućuje tehniku „mračno pokretanje“ *(eng. dark launching)* koja podrazumijeva izdavanje i testiranje nove funkcionalnosti s opterećenjem pravih korisnika, no sama funkcionalnost nije vidljiva korisnicima i izdavanje nije službeno [19].

Na kraju, odabrana arhitektura programskog proizvoda ima gotovo najveći utjecaj na način kako timovi testiraju i isporučuju programski kod. Osim toga, arhitektura je najbolji pokazatelj produktivnosti razvojnih programera te pokazatelj mogu li se promjene uvesti brzo i sigurno. Trenutno najviše problema izazivaju čvrsto povezane arhitekture *(eng. tightly-coupled architectures)* ili monolitne arhitekture *(eng. monolithic architectures)*. Problemi se javljaju prilikom pokušaja postavljanja programskog koda na glavnu granu jer razvojni programeri riskiraju kreiranje problema za druge razvojne probleme ili timove. Nadalje, da bi se izbjeglo kreiranje problema, potrebna je ogromna količina komunikacije i koordinacije između timova, a to može trajati danima. S druge strane, labavo povezana arhitektura *(eng. loosely-coupled architecture)* s dobro definiranim sučeljima, koja definiraju odnos i komunikaciju različitih modula, omogućuje implementaciju malih promjena na siguran način bez ugrožavanja povezanih dijelova programskog koda [19].

Monolitna arhitektura je tradicionalni model programskog proizvoda koji je izgrađen kao jedinstvena jedinica neovisna o drugim aplikacijama. Ova arhitektura omogućuje brzi razvoj

aplikacije, jednostavno testiranje i otklanjanje pogrešaka, i jednostavnu isporuku na produkciju s obzirom na to da postoji samo jedna baza programskog koda. S druge strane, vrijeme razvoja aplikacije je raste, što je aplikacija veća i složenija. Jedna baza programskog koda onemogućuje isporuku jedne promjene, već je potrebno cijelu aplikaciju ponovno isporučiti na produkciju. Drugi arhitektonski arhetip su mikroservisi. Arhitektura mikroservisa je model arhitekture koji se oslanja na niz neovisnih servisa, a svaki od njih ima vlastitu poslovnu logiku i bazu podataka. Drugim riječima, ažuriranje, testiranje, isporuka i skaliranje odvijaju se unutar svakog servisa. Ova arhitektura omogućuje agilni način rada u malim timovima te kontinuiranu i visoko pouzdanu isporuku neovisnih jedinica bez rizika da se naruši rad cjelokupne aplikacije. Budući da su servisi neovisni, timovi imaju slobodu eksperimentirati s odabirom alata i s uvođenjem novih promjena u programski kod. S druge strane, arhitektura mikroservisa uzrokuje širenje razvoja na više timova, što stvara za posljedicu dodatne troškove u pogledu međusobne komunikacije i suradnje. Kreiranjem novih servisa povećavaju se i infrastrukturni troškovi s obzirom na to da timovi neovisno biraju alate, programske jezike i okvire [21].

4.3. Alati za DevOps

Prilikom uvođenja DevOps praksi unutar organizacije, najbrže je te promjene uvesti kroz alate, no sami alati nisu dovoljni za unapređenjem isporuke programskih proizvoda klijentima. Ako se organizacija fokusira samo na alate, onda to često rezultira izbjegavanjem problema u komunikaciji unutar tima i između timova na razini cjelokupne organizacije. Osim toga, treba biti oprezan pri odabiru alata jer njihova snaga proizlazi iz toga koliko odgovaraju potrebama ljudi ili timova koji ih koriste. Njihovo korištenje nameće nova ponašanja zaposlenika, a ta ponašanja se onda odražavaju na organizacijsku kulturu [17]. U nastavku poglavlja opisani su najčešće korišteni alati prema ciklusima DevOps procesa. Različite literature definiraju različite cikluse DevOps procesa, a u ovom poglavlju je odabran životni ciklus koji definira tvrtka Atlassian.

4.3.1. Planiranje

Pod ciklusom planiranja važno je koristiti alate koji će nam omogućiti planiranje budućih zadataka na projektu, praćenje problema tijekom tog rada i suradnju svih članova tima [22]. Alati koji to mogu omogućiti su Jira Software, Confluence i Slack (Slika 4).

Slika 4: Logotipovi alata u ciklusu planiranja [22]

Jira Software je proizvod tvrtke Atlassian, a osmišljen je da raznovrsnim timovima olakša upravljanje poslom. Agilnim timovima omogućava praćenje zadataka, procjenu i zapisivanje vremena rada, izvještavanje o napretku i korištenje vizualnih ploča, dok razvojnim timovima nudi planiranje radnih ciklusa *(eng. sprint planning)*, planiranje isporuke i upravljanje zadacima. Jira Software kompatibilna je s drugim alatima koji se koriste u DevOps procesu kao što su alati za verzioniranje programskog koda, alati za upravljanje dokumentacijom te alati za nadgledanje [23]. Confluence je, također, proizvod tvrtke Atlassian i služi za suradnju timova te sistematizaciju i dijeljenje znanja. Za sistematizaciju znanja koriste se virtualni prostori i stranice u kojima svaki član tima može podijeliti znanje, problem, ideju ili bilo koju drugu informaciju u sklopu rada na određenom projektu [24]. Ova dva proizvoda integrirana su s aplikacijom za razmjenu poruka, Slack. Takva integracija omogućuje timovima da brzo komuniciraju i rješavaju probleme, a jedna od funkcionalnosti je primanje obavijesti o bilo kakvim promjenama dokumentacije, planova i ostalog unutar prethodna dva proizvoda [25].

4.3.2. Izrada

Alate, koji se koriste u ciklusu izrade, moguće je sistematizirati u tri kategorije – produkcijska okruženja za razvoj *(eng. Production-identical environments for development)*, infrastruktura kao kod *(eng. Infrastructure as code)* i verzioniranje programskog koda [22].

U prvu kategoriju moguće je smjestiti platformu Kubernetes, dok Docker je dostojan predstavnik i prve i druge kategorije. Docker je platforma koja, uz pomoć kontejnera, razvojnim programerima olakšava razvoj, isporuku i pokretanje aplikacija (Slika 5). Kontejneri su upakirani programski proizvodi kreirani pisanjem Docker datoteke *(eng. Dockerfile)*, a pokreću se na Docker poslužitelju koristeći naredbe za izvršavanje u naredbenom retku. Upravljanje i pokretanje kontejnera težak je proces, no ovdje Kubernetes ulazi u priču.

Slika 5: Logotip alata Docker [22]

Kubernetes je platforma otvorenog koda koju je razvio Google, a služi za upravljanje sustavima za izvršavanje kontejnera (Slika 6). Cilj ove platforme je smanjiti opterećenje mreže i povećati učinkovitost korištenja resursa, a to se postiže grupiranjem kontejnera kojima se onda upravlja s istog poslužitelja. Kubernetes usklađuje poslužitelje na temelju slobodnih resursa te upravlja dodjelom resursa, izolacijom, skaliranjem, otkrivanjem usluga i uravnoteženjem opterećenja. Dakle, Docker služi za kreiranja kontejnera, a Kubernetes je platforma za upravljanje i pokretanje tih kontejnera [26].

Slika 6: Logotip alata Kubernetes [22]

Treća kategorija podrazumijeva sustave za verzioniranje, koji bilježe promjene u obliku programskog koda koje naprave razvojni programeri. Oni mogu biti lokalni, centralizirani ili distribuirani ovisno o procesu pohrane spomenutih promjena. Lokalno verzioniranje programskog koda podrazumijeva pohranu promjena gdje je izvorni kod i pohranjen, dok se centralizirano odnosi na pohranu promjena na udaljenom poslužitelju. Distribuirano verzioniranje bilježi promjene na svim repliciranim repozitorijima na različitim granama [17]. Jedan od alata koji je moguće koristiti za verzioniranje programskog koda je GitHub (Slika 7).

Slika 7: Logotip alata GitHub [22]

GitHub je *web*-bazirana platforma za verzioniranje programskog koda i kolaboraciju razvojnih programera. Temelji se na Gitu, sustavu otvorenog koda za upravljanje programskim

kodom, kojeg je osmislio Linus Torvalds 2005. godine. Služi za pohranu izvornog programskog koda i bilježi povijest svih promjena, stoga sama platforma nudi razvojnim programerima ili timovima kreiranje repozitorija, koji sadrži sve datoteke projekta i povijest svih promjena, te paralelni rad, preuzimanje i postavljanje novih promjena [27].

4.3.3. Kontinuirana integracija i isporuka

Ciklus kontinuirane integracije i isporuke podrazumijeva nekoliko koncepata, a to su kontinuirana integracija *(eng. continuous integration)*, kontinuirana isporuka *(eng. continuous delivery)*, kontinuirano postavljanje na produkciju *(eng. continuous deployment)* te automatizirano testiranje. Kontinuirana integracija je proces integriranja novog programskog koda na glavnu *(eng. master, main)* granu u zajedničkom repozitoriju. Cilj ovog procesa je poticati male i česte promjene koje se integriraju u ostatak programskog proizvoda. Alati za kontinuiranu integraciju automatski pokreću testove kojima se provjerava integracija novih promjena s postojećim programskim kodom. Takvim pristupom jednostavnije je prepoznati, izolirati i otkloniti greške u programskom kodu [17].

Kontinuirana isporuka i kontinuirano postavljanje na produkciju su dva pojma koja se često pomiješaju. Kontinuirano postavljanje na produkciju je proces koji obuhvaća kontinuiranu integraciju i kontinuiranu isporuku te isporuke programskog proizvoda krajnjim korisnicima. Programski proizvod se isporučuje krajnjim korisnicima tek kad su uspješno izvršeni verifikacijski testovi. S druge strane, kontinuirana isporuka je proces česte implementacije promjena i isporuke nove verzije programskog proizvoda uz pomoć kontinuirane integracije i automatiziranog testiranja. Isporuka nove verzije podrazumijeva isporuku na druge okoline kao što su razvojna okolina ili okolina za testiranje [17].

Većina DevOps alata obuhvaća sva tri navedena procesa, a jedan od najpoznatijih takvih je Jenkins (Slika 8). Riječ je o alatu koji nudi postavljanje okoline za kontinuiranu integraciju, kontinuiranu isporuku i kontinuirano postavljanje na produkciju bez obzira na odabrane programske jezike i sustave za verzioniranje programskog koda. Jenkins je prvotno osmišljen kao alat za kontinuiranu isporuku programskih proizvoda pisanih u Javi, a autor ideje je nekadašnji razvojni programer Kohsuke Kawaguchi.

Slika 8: Logotip alata Jenkins [22]

Danas, uz pomoć Jenkinsa, razvojni programeri mogu automatizirati bilo koju aktivnost u procesu razvoja, testiranja ili isporuke programskog proizvoda jer podržava više od 1600 alata trećih strana u području korisničkog sučelja, administracije, upravljanja izgradnjom i verzioniranja programskog koda. Najvažniji aspekt korištenja ovakvih alata je kreiranje skripti za automatizaciju, tzv. cjevovodi *(eng. pipelines)*. Jenkins omogućuje kreiranje skripti pomoću sučelja, ali najboljom praksom se pokazalo samostalno kreiranje skripte poznate pod nazivom *Jenkinsfile* [28].

Automatizirano testiranje važna je komponenta ovog ciklusa, a od alata se očekuje da podržavaju automatizirano testiranje, upravljanje i usklađivanje testova. Dugoročno gledano automatizacija testova ubrzava kontinuirane cikluse razvoja i testiranja programskog proizvoda te smanjuje rizik od grešaka i povećava kvalitetu proizvoda. Većina alata za kontinuiranu integracija obuhvaća automatiziranu izgradnju i testiranje programskog proizvoda, no razmotrit ćemo alat pod nazivom Appium [22].

Appium je besplatan alat otvorenog koda specijaliziran za automatizaciju testiranja mobilnih i *web* aplikacija. Ovaj alat je razvila tvrtka Sauce Labs (Slika 9) i trenutno ga održava, a prvotno je bio namijenjen samo testiranju mobilnih aplikacija.

Slika 9: Logotip platforme Sauce Labs [22]

Prednosti ovog alatu su da podržava testiranje izvornih i višeplatformskih mobilnih aplikacija bez potrebe uvida u izvorni programski kod aplikacije. Arhitektura alata Appium sastoji se od poslužitelja i klijenta koji komuniciraju s virtualnim ili pravim mobilnim uređajima. Poslužitelj šalje zahtjeve klijentu, a klijent ih prosljeđuje mobilnom uređaju na kojem se izvršavaju skripte za automatizaciju testova [29].

4.3.4. Nadgledanje

Nadgledanje (eng. monitoring) je proces koji prati trenutno stanje sustava i okoline prikupljajući metrike poput informacije o radu poslužitelja, iskorištenju memorije, broju korisničkih zahtjeva, broju stavki u redu čekanja ili broju upita na bazu podataka. Navedene informacije se uspoređuju sa željenim stanjem sustava i provjerava se da li su uvjeti zadovoljeni. Nadzor novih sustava često se automatski uspostavlja s alatima za upravljanje konfiguracijom jednom kada korisnik odredi što sve želi nadgledati [17].

AppDynamics je rješenje za upravljanje performansama aplikacije kojeg je osmislila istoimena tvrtka 2008. godine, a od 2017. godine je u vlasništvu tvrtke Cisco (Slika 10).

Slika 10: Logotip alata AppDynamics [22]

Ovaj alat koristi umjetnu inteligenciju za prepoznavanje i rješavanje problema te prikuplja metrike ključne za optimizaciju programskih proizvoda. Neke od metrika su performanse aplikacije, korištenje procesora, stope pojave grašaka, stope zahtjeva i vrijeme odgovora na te zahtjeve. Sve dobivene informacije se nalaze na vizualnoj ploči koja prikazuje odnose između komponenti u cilju lakšeg otkrivanja uzroka problema [30].

4.3.5. Kontinuirana povratna informacija

Za kontinuirano prikupljanje povratnih informacija potrebna je organizacijska kultura, procesi i alati koji se bave redovitim prikupljanjem i analizom podataka. Procesi za prikupljanje povratnih informacija uključuju prikupljanje izvještaja o zadovoljstvu korisnika *(eng. Net Promoter Score, NPS)*, izradu anketa o napuštanju korisnika, prijavu problema i objave na društvenim mrežama. Preporuka je iskoristiti alate koji su integrirani s aplikacijama za razmjenu poruka i društvenim mrežama jer tako organizacija dobiva povratne informacije u stvarnom vremenu [22].

Na primjer, Pendo podržava integraciju s aplikacijama kao što su Slack i Jira Software. Pendo je skup alata za analitiku, kreiranje uputa unutar aplikacije i upravljanje povratnim informacijama korisnika (Slika 11).

Slika 11: Logotip alata Pendo [22]

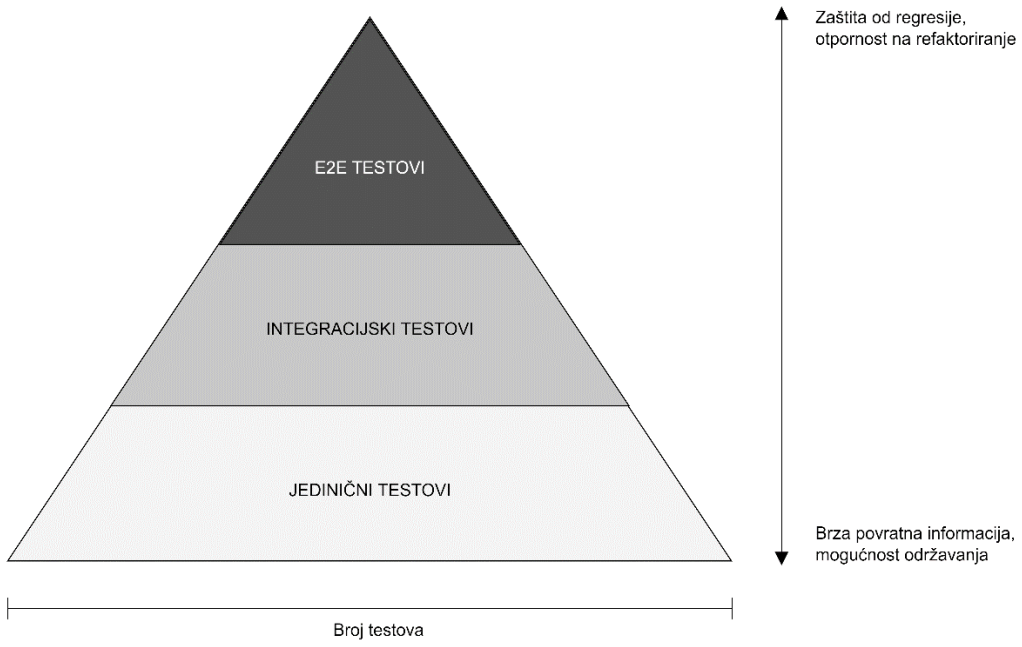
Ovaj set alata može koristiti bilo tko od zaposlenika jer ne zahtijeva programerske vještine, ali preporuka je da timovi surađuju s ciljem boljeg uvida u korisničko iskustvo i kreiranja boljih programskih rješenja. Platforma obuhvaća alate pomoću kojih timovi nadgledaju i analiziraju

interakciju korisnika s proizvodom, kreiraju ankete unutar proizvoda i bilježe korisničko iskustvo te kreiraju korisničke upute za korištenje. Osim toga, Pendo omogućava dvosmjernu komunikaciju između korisnika i proizvođača programskog proizvoda tako da korisnici mogu proslijediti prijedloge za poboljšanje, a odgovorne osobe u organizacije ih obavijestiti o statusu njihovog prijedloga [31].

5. Testiranja programskih proizvoda

Važnost testiranja programskih proizvoda kratko je opisana u uvodu rada, a razlog leži u nepredvidivim greškama u programsku kodu koje najčešće nastaju uvođenjem novih funkcionalnosti i integracija istih s postojećim programskim modulima. Iako se redovitim testiranje unaprjeđuje kvaliteta programskog proizvoda i dugoročno pozitivno utječu na troškove, ovaj proces se često zanemaruje [1].

Ovo poglavlje sadrži pregled metoda testiranja programskih proizvoda, a odnos tih metoda moguće je vidjeti na slici (Slika 12), a radi se o piramidi jediničnih, integracijskih i *end-to-end* testova. Jedinični i integracijski testovi su opisani u nadolazećim poglavljima. *End-to-end* testovi se neće obraditi u ovom radu, no riječ je o metodi testiranja programskih proizvoda pomoću koje provjeravamo sve dijelove programskog proizvoda, uključujući integrirane vanjske sustave, kao cjelinu. Svrha takvih testova je provjeriti ispravnost integracije i komunikacije programskog proizvoda s vanjskim bibliotekama, sučeljima, bazom podataka i drugim sustavima [32].



Slika 12: Piramida testova [33]

Svrha ove piramide je slikovito objasniti omjer navedenih testova u projektima. Taj omjer ili oblik piramide varira ovisno o složenosti projekta, pa će tako kod manjih projekata oblik biti pravokutnik s jediničnim i integracijskim testovima. Preporuka je, dakle, provjeriti što više poslovne logike i rubnih slučajeva koristeći jedinične testove, a pomoću integracijskih testova

provjeriti specifične scenarije i preostale rubne slučajeve. Implementacijom većeg broja jediničnih testova postiže se jednostavnije održavanje sustava [33].

5.1. Jedinična testiranja

Jedinična testiranja ili testiranje modula programskog proizvoda je metoda koja obuhvaća proces testiranja manjih, izoliranih dijelova programskog koda. Ti dijelovi mogu biti pojedinačni potprogrami, rutine, klase ili procedure [34]. Svaki jedinični test provjerava manji dio programskog koda u izoliranim uvjetima, što znači da su međusobno neovisni i mogu se izvoditi paralelno ili sekvencijalno [33]. Cilj jediničnih testiranja je omogućiti održivi rast i razvoj programskih proizvoda, odnosno služi kao sigurnost da će postojeći kod funkcionirati i prilikom uvođenja novih funkcionalnosti. Bez testiranja i refaktoriranja, programski kod postaje sve složeniji, nepouzdan i naposljetku nečitljiv [33].

Jedna od prednosti jediničnog testiranja programskog proizvoda je da omogućuje refaktoriranje programskog koda uz mogućnost provjere ispravnosti modula. Drugim riječima, greška uzrokovana promjenama u programskog kodu može se brzo uočiti i otkloniti. Budući da se radi o testiranju samo jednog dijela programskog koda, jedinično testiranje je moguće provesti bez čekanja da se razviju drugi dijelovi [35]. Dakle, pisanje jediničnih testova preporučuje se u što ranijoj fazi razvojnog ciklusa. S druge strane, priroda jediničnog testa je takva da je fokus samo na manjem, izoliranom dijelu programskog koda, stoga nije moguće uočiti svaku grešku. Jedinične testove treba kombinirati s drugim vrstama testova, na primjer s integracijskim testiranjem [35].

Jedinični testovi se strukturiraju koristeći uzorak „Organiziraj-djeluj-potvrdi“ *(eng. Arrange-Act-Assert (3A) Pattern)*, odnosno testovi se dijele na tri koraka: organiziraj, djeluj i potvrdi. Khorikov [33] nam daje primjer kako implementirati navedena tri koraka, što je prikazano u primjeru Isječak kôda 20. Primjer je prilagođen, odnosno napisan u programskom jeziku Dart.

Isječak kôda 20: Jedinični test s 3A uzorkom

```
class Calculator {
      double sum(double first, double second) => first + second;
}

void main() {
      test('Provjera zbrajanja dvaju brojeva', () {
            final calculator = Calculator();
            final first = 5.0;
            final second = 7.0;

            expect(calculator.sum(first, second), 12.0);
      });
}
```

U primjeru iznad kreiran je test za metodu `sum` iz klase `Calculator`. Prvi korak „organiziraj“ podrazumijeva instanciranje i inicijalizaciju potrebnih varijabli. Drugi korak „djeluj“ podrazumijeva pozivanje metode koja se želi testirati, a u ovom slučaju radi se o metodi `sum`. U posljednjem koraku „potvrdi“ provjeravamo da li dobiveni rezultat odgovara očekivanoj vrijednosti [33].

5.2. Integracijska testiranja

Integracijski testovi su posljedica nemogućnosti da se jediničnim testovima provjeri rad modula programskog proizvoda s drugim modulima ili vanjskim sustavima [33]. Drugim riječima, integracijski test je metoda testiranja pomoću koje integriramo module i testiramo ih kao cjelinu s ciljem otkrivanja grešaka u vidu njihove međusobne interakcije [36]. U poglavlju 5.1 naučili smo da su jedinični testovi brza metoda provjere manjeg dijela programskog koda neovisno o drugim dijelovima i testovima. Ako određeni test ne zadovoljava jednu od navedenih tri karakteristike, radi se o integracijskom testu [33]. Khorikov [33] i slikovito opisuje koji programski kod se provjerava jediničnim odnosno integracijskim testovima, što je moguće vidjeti u tablici dolje.

Tablica 5: Matrica programskog koda prema složenosti i broju razvojnih programera [33]

Složenost		
	Algoritmi, model domene	Presložen programski kod
	Trivijalan programski kod	Kontroleri
	Broj razvojnih programera	

Trivijalan programski kod se ne testira zbog same prirode istoga, dok presložen programski kod je potrebno refaktorirati u obliku algoritama i kontrolera. Algoritmi i modeli domene provjeravaju se jediničnim testovima, a kontroleri integracijskim testovima. U obzir treba uzeti mogućnost da se ovisnosti odnosno vanjske biblioteke i paketi u kontrolerima mogu oponašati „lažnim“ programskim kodom. Tako se izbjegava međusobna povezanost testova čime se zadovoljavaju spomenute karakteristike jediničnih testova. Drugim riječima, kontrolere je moguće i provjeriti pomoću jediničnih testova ako se stvore odgovarajući uvjeti [33].

5.2.1. Pristupi integracijskog testiranja

Integracijske testove moguće je provesti na dva načina, testiranjem svih modula programskog proizvoda kao cjeline ili testiranjem jednog modula s prethodno testiranim modulima. Ova podjela vodi do dva pristupa integracijskom testiranju, a to su neinkrementalni i inkrementalni pristup [34].

Pod neinkrementalni ili *„big-bang“* pristup razumijevamo integracijske testove koje se provode tako da se testiraju svi moduli programskog proizvoda povezani u cjelinu. Nad svakim modulom prethodno se moraju izvršiti jedinični testovi [34]. Ovaj pristup je prigodan za male projekte jer lokalizacija pogreške je teška u velikim projektima i vrlo lako je moguće propustiti testirati određene dijelove programa. Neinkrementalni pristup, osim toga, ne razlikuje kritične ili prioritetne module od ostalih što utječe na pouzdanost i kvalitetu provedenih testova [36].

Inkrementalni pristup podrazumijeva integraciju dva ili više logički povezana modula i njihovo testiranje s ciljem provjere određene funkcionalnosti programskog proizvoda. Ovaj proces provodi se sve dok se svi logički povezani moduli uspješno ne testiraju [36]. Testovi nad modulima mogu se provoditi strategijama „odozdo prema gore“ *(eng. Bottom Up)* ili „odozgo prema dolje“ *(eng. Top Down)* [34].

Strategijom „odozgo prema dolje“ kreće se s testiranjem prvog modula u hijerarhiji, a u nastavku je moguće odabrati bilo koji drugi modul s uvjetom da je logički povezan s prethodno testiranim modulima [34]. Ovom strategijom se postiže da se prvo testiraju kritični dijelovi programskog proizvoda čime se jednostavnije pronalaze pogreške. Budući da se kreće od samog vrha programa, potrebno je kreirati lažne dijelove programskog koda koji simuliraju komunikaciju logički povezanih modula [34], [36].

S druge strane, strategija „odozdo prema gore“ kreće s testiranjem krajnjih modula koji ne pozivaju druge module programskog proizvoda. U nastavku procesa, kao i u prethodnoj strategiji, ne postoji pravilo odabira sljedećeg modula za testiranje. Jedini uvjet je da su svi podređeni moduli odabranog modula testirani [34]. Problem kod ove strategije je da se cjelokupni sustav ne testira sve dok se ne doda posljednji modul u lanac. No, testove i uvjete, koji se provjeravaju u njima, jednostavnije je kreirati s obzirom na to da nije potrebno razviti lažni programski kod za simulaciju komunikacije modula [34], [36].

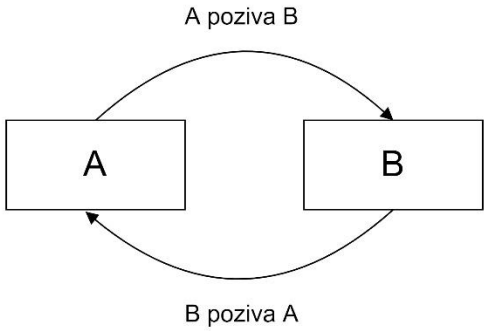
5.2.2. Najbolje prakse integracijskog testiranja

Primjena dobrih praksi u bilo kojoj domeni, a osobito kod testiranja omogućuje razvojnim programerima da pišu čišći kod i redovito ga analiziraju i po potrebi refaktoriraju [33]. Khorikov [33] nudi tri opća principa dobre prakse integracijskog testiranja:

- eksplicitno definiranje granica modela domene,
- uklanjanje kružnih ovisnosti i

- smanjenje broja slojeva u aplikaciji.

Model domene predstavlja cjelokupno znanje o problemu koji se želi riješiti razvojem programskom proizvoda. Definiranje tih granica olakšava vizualizaciju i razvoj testova za pojedine dijelove programskog koda [33]. Pod kružnim ovisnostima razumijevamo odnos dviju ili više klasa u programskog kodu koje eksplicitno ili implicitno ovise jedna o drugoj. Ovakva vrsta ovisnosti vodi do neurednog programskog koda, a u konačnici i do slabog razumijevanja uloge i svrhe klasa [33]. Na slici (Slika 13) vizualno je prikazana kružna ovisnost između klasa A i B koja je ostvarena tako da klasa A poziva članove klase B, a ujedno klasa B poziva članove klase A. Članovi klase mogu biti varijable ili metode.



Slika 13: Kružna ovisnost

Pomoću slojeva u aplikaciji želi se apstrahirati i generalizirati programski kod, no velika, nepotrebna količina slojeva ne olakšava provođenje niti jediničnih ni integracijskih testova. Tako se kreiraju integracijski testovi za svaki sloj zasebno, a njihovi rezultati su niske kvalitete [33].

5.3. Testiranje elemenata korisničkog sučelja

Testiranje elemenata korisničkog sučelja *(eng. widget test)* je proces provjere pojedinačne komponente korisničkog sučelja. Cilj takvih testova je provjeriti ima li element željeni izgled i ponaša li se očekivano [11]. Test elementa korisničkog sučelja sličan je jediničnim testovima, no opsežniji je i zahtjeva nove, dodatne tehnike uključene u ugrađenu biblioteku `flutter_test` [4], [11].

Miola [4] nudi primjer testiranja jednog jednostavnog elementa korisničkog sučelja, a u nastavku je kreiran jedan primjer inspiriran njime. U primjeru Isječak kôda 21 je element korisničkog sučelja koji prikazuje podatke o studentu.

Isječak kôda 21: Primjer elementa za testiranje

```
class StudentScreen extends StatelessWidget {
      final String jmbag;
      final String ime;
      final String prezime;

      const StudentScreen(this.jmbag, this.ime, this.prezime);

      @override
      Widget build(BuildContext context) {
            return Row(
                  children: [
                        Text(jmbag),
                        Text("$ime $prezime"),
                  ]
            );
      }
}
```

Za gornji programski kod potrebno je kreirati novu datoteku koja će sadržavati test. S tim testom, prikazanim u primjeru Isječak kôda 22, provjerit će se imaju li varijable u klasi `StudentScreen` dodijeljene vrijednosti i postoje li tekstualni elementi na korisničkom sučelja s navedenim vrijednostima.

Isječak kôda 22: Test za element korisničkog sučelja

```
void main() {
      testWidgets("Testiranje klase StudentScreen",
      (WidgetTester tester) async {

            await tester.pumpWidget(
StudentScreen("0123456789", "Iva", "Papac"));

            final CommonFinders jmbag = find.text("0123456789");
            final CommonFinders imePrezime = find.text("Iva Papac");

            expect(jmbag, findsOneWidget);
            expect(imePrezime, findsWidgets);
      });
}
```

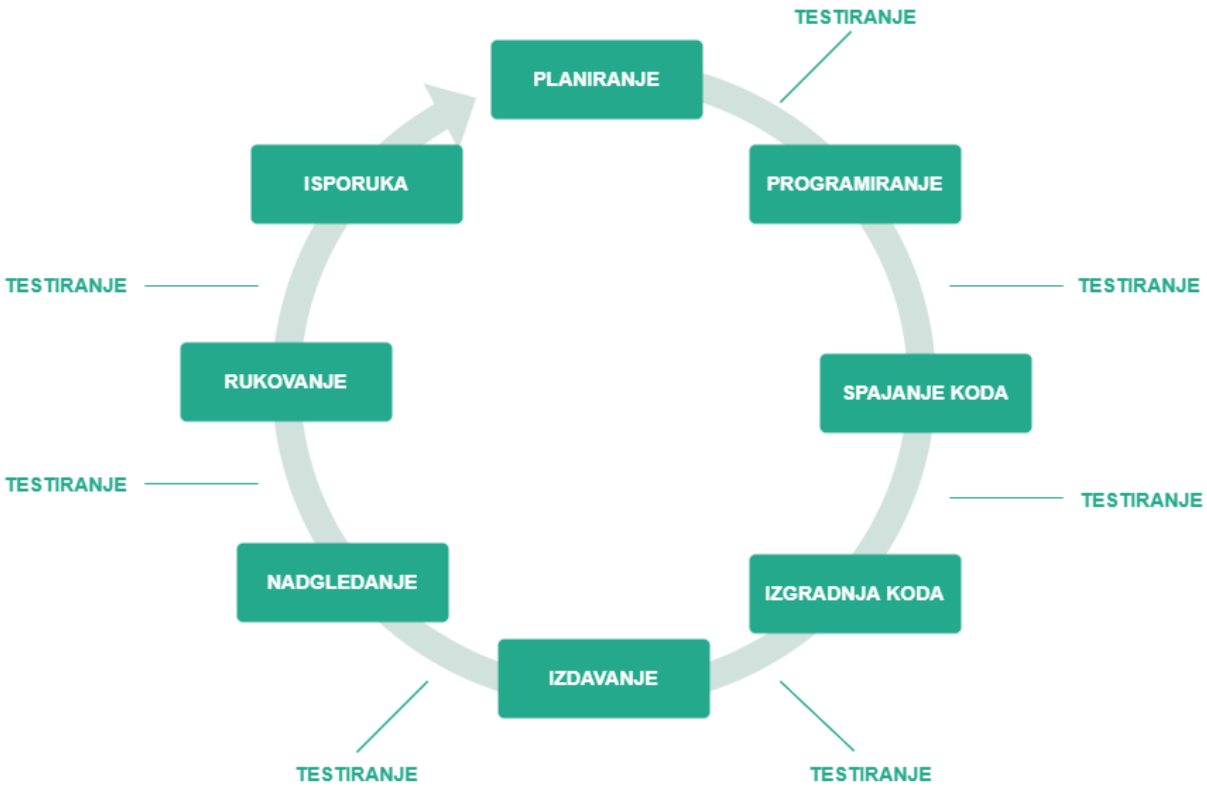
Prva metoda `pumpWidget()` služi za kreiranje i prikazivanje elementa u testnoj okolini. Klasa `CommonFinders` dio je Flutter biblioteke za testiranje, a metoda `find.text()` služi za pronalazak tekstualnog elementa korisničkog sučelja u aplikaciji. Za pronalazak elementa korisničkog sučelja, umjesto spomenute metode, moguće je koristiti metode `find.byWidget()` ili `find.byType()`. Prva navedena metoda traži element prema određenim specifikacijama, dok druga traži element određenog tipa, na primjer `IconButton` [4].

Uz pomoć metode `expect()` provjeravamo koliko tekstualnih elemenata mora biti prikazano na zaslonu prosljeđujući joj jedan od četiri parametra – `findsNothing`, `findsOneWidget`, `findsWidgets` i `findsNWidget`. Parametar `findsNothing` označava da

nije nađen nijedan element, parametar `findsOneWidget` označava da je pronađen točno jedan element, parametar `findsWidgets` označava da je pronađen jedan ili više elemenata i parametar `findsNWidget` označava da je pronađeno točno N elemenata [4].

5.4. Testiranje u kontekstu DevOps-a

Testiranje i automatizacija testiranja jedne su od najvažnijih aktivnosti kontinuirane integracije i kontinuirane isporuke [37]. Ashby [38] smatra da bilo kakva vrsta testiranja se može uklopiti u bilo koji od ciklusa DevOps-a, što je i prikazano na slici (Slika 14).



Slika 14: Testiranje u ciklusima DevOps-a [38]

Testiranje plana podrazumijeva testiranje dizajna programskog proizvoda, a temelji se na postavljanju pitanja, istraživanju i refaktoriranju dizajna prema prikupljenim informacijama. Testiranje programskog koda postižemo redovitim pregledom i analizom napisanog programskog koda, implementacijom automatiziranih jediničnih testova ili programiranjem u paru s testerom. Osim očitih testova, moguće je testirati sustav u ciklusu nadzora provodeći istraživanje o tome prikupljamo li ispravne informacije i nadgledamo li prave dijelove sustava [38]. Cilj je uvijek isti – provjeriti da li aplikacija ispunjava korisničke zahtjeve, provjeriti postoje li greške u programskog kodu i provjeriti da li su nove promjene u kodu utjecale na postojeće funkcionalnosti [37].

Programski proizvodi postali su sve složeniji, a prakse većine metodologija tjeraju izvršavanje programskog koda u različitim okolinama. Pridodaju li se tu i prakse DevOps-a, automatizirano testiranje je neizbježno. Osim toga, principi i prakse kontinuirane integracije također zahtijevaju da se testira svaki put kada je novi programski kod dodan u Git repozitorij. Kontinuirana integracija i provjera novog programskog koda ne bi bila moguća bez automatiziranog testiranja, stoga ga što prije treba uvesti u procese razvoja, integracije i isporuke. Preporuka je da se krene od onih dijelova programskog koda kojeg korisnici najviše koriste [37].

Kontinuiranim testiranjem ostvaruje se jedan od temeljnih principa DevOps-a – griješi brzo i često *(eng. fail fast and often).* Takvom praksom otkriva se više grešaka i izbjegava se pojava incidenata u produkcijskoj okolini. Proces kontinuiranog testiranja počinje u ciklusu razvoja programskog proizvoda, pa zaključujemo da je odgovornost za pisanje testova i na razvojnim programerima. Tako odmah dobivaju povratne informacije o kvalitetu i valjanosti njihovog programskog koda [37].

5.4.1. Principi i prakse

Ovo potpoglavlje orijentirat će se na principe i prakse testiranja u ranoj fazi kontinuirane integracije i kontinuirane isporuke kada razvojni programeri isporuče promjene u zajednički repozitorij. Ovo je važan prvi korak jer smanjuje vrijeme integracije programskog koda. Za razvojne programere ovo je najvažniji dio razvojnog procesa jer je ovo jedini način za brzo dobivanje povratnih informacija o programskom kodu. Isto tako, ova faza je neizbježan početak procesa isporuke programskog proizvoda korisnicima, a ona je jedino moguća ako su svi testovi rezultirali uspjehom [39].

Humble i Farley [39] preporučuju da većina programskog koda mora biti pokrivena jediničnim testovima, iako njihov pozitivan rezultat ne garantira da će aplikacija ispravno raditi. Potrebno je implementirati i manji broj integracijskih testova i testova korisničkog sučelja kako bi se isporučio željeni programski proizvod [39]. Sljedeći principi i prakse preporuka su kako pristupiti testiranju u kontekstu DevOps-a:

- Korištenje injekcije ovisnosti ili inverzije kontrole omogućuje kreiranje fleksibilno i modularnog programskog proizvoda te pojednostavljuje smanjivanje opsega testova. Riječ je o uzorku dizajna kojom se veza između objekata definira izvan njih, a ne unutar [39].
- U testovima treba izbjegavati bazu podataka, a to je jedino moguće ako je sloj poslovne logike odvojen od sloja baze podataka. Komunikacija s bazom podataka otežava i usporava proces pisanja i izvršavanja testova. S druge strane, preporuka

je imati barem jedan integracijski test koji obuhvaća i bazu podataka kako bi bili sigurni da aplikacija zaista ispravno radi [39].

- U jediničnim testovima treba izbjegavati asinkrone pozive metoda. Najbolji pristup tomu je da se u jednom testu testira programski kod do tog poziva, a u drugom testu programski kod nakon tog poziva [39].
- Svaka klasa u programskom kodu komunicira s drugim klasama, što znači da bi se u testovima prvo morale inicijalizirate sve povezane klase i tek onda testirati ponašanje odabrane klase. U testovima svaka komunikacija s drugim klasama treba se lažirati. Taj proces se naziva *Stubbing*, a podrazumijeva zamjenu dijela programskog koda s ručno upisanim podacima [39].
- Testovi bi se trebali fokusirati na ponašanje sustava, no ono što je neizbježno je postavljanje inicijalnih struktura podataka kako bi se ponašanje uopće moglo testirati. Dakle, potrebno je definirati ulazne podatke kako bi se na kraju usporedili očekivani i dobiveni rezultati, no to vrlo lako rezultira kreiranjem testova koji su teški za razumjeti i održavati. Ako testovi izgledaju složeno, velika je vjerojatnost da je i sam programski kod loše strukturiran. Idealni testovi su oni koje jednostavno kreirati i pokrenuti, no to zahtijeva dobro strukturiran kod [39].
- Proces automatiziranog testiranja treba biti kratka jer cilj je da razvojni programeri što brže dobe povratne informacije o kvaliteti i valjanosti programskog koda. Jedan način je da se testovi grupiraju u više paketa koji se onda mogu izvoditi paralelno na više poslužitelja [39].

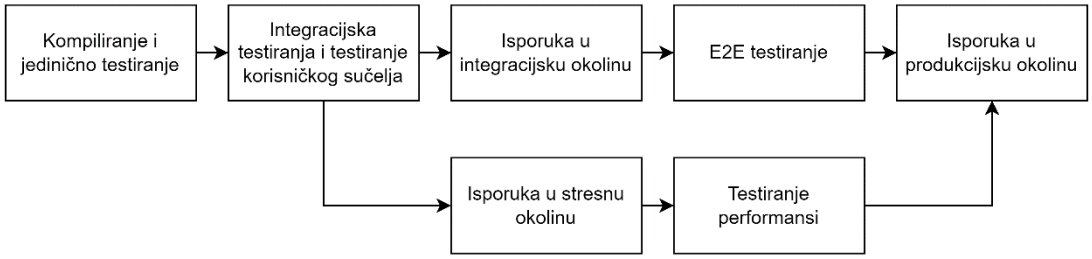
Ovim praksama želi se postići smanjenje opsega bilo kojeg testa kako bi se fokusirali na što manji dio aplikacije. Dakle, treba izbjegavati datotečne sustave, baze podataka, biblioteke ili vanjske sustave u jediničnim testiranjima te treba koristiti testne dvojnike *(eng. mocks and stubs)* [39].

5.4.2. Testiranje u razvoju

Sadržaj ovog potpoglavlja fokusira se na promjene praksi testiranja u procesu razvoja programskih proizvoda uvodeći DevOps prakse. One uključuju cjevovod za isporuku *(eng. deployment pipeline),* uključivanje i isključivanje funkcionalnosti *(eng. feature toggle)*, proces testiranja s korisnicima *(eng. crowd testing)* i proces pronalaska grešaka *(eng. bug bash)* [16].

Clokie [16] navodi da je cjevovod za isporuku jedan način komunikacije razvojnih i operacijskih timova. Jedan tim može provjeriti ispravnost ponašanja programskog proizvoda, dok drugi može provjeriti ispravnost procesa isporuke programskog proizvoda korisnicima. Ranije u radu je spomenuto da proces kontinuirane integracije i kontinuirane isporuke počinje postavljanjem novog programskog koda na zajednički repozitorij. Cjevovod za isporuku

ilustrira taj proces, a sastoji se od izvršavanja jediničnih i integracijskih testova, testova korisničkog sučelja, *end-to-end* testova i isporuke na produkcijsku ili druge okoline. Svaka organizacija ima slobodu samostalno definirati korake kontinuirane integracije i kontinuirane isporuke [16]. Primjer jedne cjevovodi za isporuku prikazan je na slici dolje.



Slika 15: Primjer cjevovodi za isporuku

Uključivanje i isključivanje funkcionalnosti odnosi se na konfiguraciju kojom se određuje treba li određenu funkcionalnost programskog proizvoda izvršiti. Ovom praksom se postiže mogućnost isporuke u produkcijsku okolinu bez da su sve funkcionalnosti odmah dostupne korisnicima. Ovakva praksa je pogodna kada razvojni tim mora implementirati neku funkcionalnost čiji će razvoj trajati danima ili čak tjednima. Ako novi programski kod integriramo na dnevnoj bazi, tada je primjena ove prakse neizbježna. Osim toga, ova praksa smanjuje rizik pri isporuci programskog proizvoda korisnicima i omogućuje razvojnim i operacijskim timovima da lakše otkriju greške, a u slučaju problema funkcionalnost se može isključiti te uključiti naklon otklanjanja grešaka [16].

Proces pronalaska grešaka je vrsta testiranja koja podrazumijeva okupljanje svih članova razvojnih i operacijskih timova kako bi sudjelovali u sesijama testiranja. Obično ovu vrstu testiranja prihvaćaju organizacije koje su fokusirane na isporuku visoko kvalitetnih programskih proizvoda. Preporuka je da se sesije testiranja planiraju unaprijed i da se sve ljude na vrijeme obavijesti kako bi se izolirao programski kod koji se želi testirati i kako bi se napravio predložak za izvještaj o greški. Neke organizacije prakticiraju ovu vrstu testiranja jer omogućuje zajednički rad razvojnih i operacijskih timova ili ako ne postoje timovi specijalizirani za testiranje [16].

Testiranje s pravim korisnicima podrazumijeva da različiti profili ljudi ili potencijalni korisnici testiraju programski proizvod. Ova vrsta testa može se izvršiti procesu razvoja, ali i nakon isporuke programskog proizvoda korisnicima. Prednosti ovakvih testiranja su raznolika testna okruženja temeljena na profilu ljudi, izvještaji o pogreškama, nepristrana perspektiva i raznolika mišljenja o programskom proizvodu, i brzo izvođenje s velikom količinom ljudi. S druge strane, mogu se pojaviti sljedeći nedostaci – povjerljivi dijelovi programskog proizvoda

mogu doći do konkurencije, otežana komunikacija s velikim brojem ispitanika, prijavljivanje velikog broja trivijalnih problema s ciljem veće naplate usluge i problemi povezani s upravljanjem testiranja. Organizacije koje su tek implementirale DevOps kulturu, ovakva vrsta testiranja može biti alternativa *beta* testiranju [16].

5.4.3. Testiranje u produkciji

Testiranje u produkciji predstavlja ravnotežu između brze isporuke i temeljite provjere programskog proizvoda prije isporuke na produkcijsku okolinu. Također, rezultati dobiveni testiranjem u produkciji mogu utjecati na odluke koje su ranije donijeli članovi razvojnog tima s obzirom na to da se povratne informacije temelje na stvarnoj korisničkoj interakciji s programskim proizvodom. U produkciji su najčešće tri prakse – A/B testiranje, beta testiranje i nadzor kao testiranje [16].

A/B testiranje podrazumijeva kreiranje dvije verzije programskog proizvoda koje korisnici testiraju i potom biraju subjektivno bolju verziju. Prije A/B testiranja obje verzije moraju biti verificirane kao ispravne jer cilj nije da korisnici nađu greške, već da li se programski proizvod ispravno razvija. S druge strane, ova praksa ne bi trebala biti jedini izvor povratnih informacija pri donošenju odluka niti bi korisnici trebali biti ti koji donose odluke umjesto razvojnog tima [16].

Beta testiranje je vrsta testiranja koja uključuje isporuku nove verzije programskog proizvoda manjoj grupi korisnika koja taj proizvod i testira. Cilj testiranja je da se pronađu greške prije nego se nova verzija isporuči svim korisnicima. Prednosti ovog testiranja su raznolikost mišljenja o programskog proizvoda, brza isporuka korisnicima, mogućnost testiranja nadgledanja, analitičkih događaja i alarma u produkcijskoj okolini. Iz tih povratnih informacija mogu se izvući greške koje su korisnici i razvojni tim predvidjeli. Uz ovu vrstu testiranja preporučuje se i izvođenje testiranja upotrebljivosti, pristupačnosti, performansi ili penetracijsko testiranje [16]. S druge strane, uz beta testiranje vežemo pet čestih problema:

- Dobrovoljni korisnici u beta testiranju nisu dovoljni jer ne predstavljaju sve profile korisnika. Riječ je o osobama koje lako prihvaćaju nove tehnologije, dok većina korisnika su pragmatici i prihvaćaju nove tehnologije tek kad se one dokažu kao sigurne za korištenje [16].
- Testeri uključeni u beta testiranje mogu samo površno isprobati programski proizvod, što može rezultirati neotkrivanjem grešaka [16].
- Testeri često ne prijavljuju probleme s upotrebljivošću programskog proizvoda jer nisu sigurni uzrokuje li to loš dizajn sustava ili oni sami. Stoga, važno je provoditi i testiranja upotrebljivosti uz beta testiranje [16].

- Testeri neće prijaviti pronađenu grešku ako nisu sigurni što je uzrok, odnosno što su učinili da izazovu pojavu te greške [16].
- Izvještaji o greškama često je teško razumjeti pa je potrebno duže vrijeme da se otkrije uzrok problema [16].

Posljednja praksa testiranja u produkciji je nadzor kao testiranje koja se primjenjuje kada je programski proizvod dostupan korisnicima. Ova praksa omogućuje identifikaciju kada su se problemi javili i kada su oni riješeni. Drugim riječima, nadzor kao testiranje je reaktivna i pasivna praksa jer se problemi otkrivaju tek nakon što se dogode. Pasivna testiranja daju uvid u ponašanje korisnika, dok aktivna testiranja će dati podatke kao rezultat izvršavanja poznatih i ponovljivih scenarija. No, prasku nadzora moguće je učiniti aktivnijom i bržom izvršavanjem automatiziranih testova u produkciji pomoću kojih se mogu otkriti i otkloniti probleme prije nego ih korisnik ima prilike iskusiti. Osim toga, nadgledanjem programskog proizvoda nakon isporuke korisnicima, moguće je pratiti korisničke interakcije i tretirati ih testne scenarija o kojima razvojni tim nije razmišljao [16].

6. Praktični dio

U ovom poglavlju bit će opisani najvažniji dijelovi implementacije i testiranja programskog proizvoda te primjene alata za kontinuiranu integraciju. Taj programski proizvod je Android aplikacija razvijena koristeći programski okvir Flutter i programski jezik Dart. Riječ je o aplikaciji „Cinnamon College“ koja je razvijena u sklopu Flutter tečaja u organizaciji tvrtke Cinnamon. Aplikacija služi da se polaznici tečaja informiraju o vremenu i mjestu održavanja radionica. Polaznik prije svake radionice može rezervirati stol i vidjeti tuđe rezervacije. Nakon radionice, istu je moguće ocijeniti. U sljedećem poglavlju detaljnije su opisane sve funkcionalnosti aplikacije.

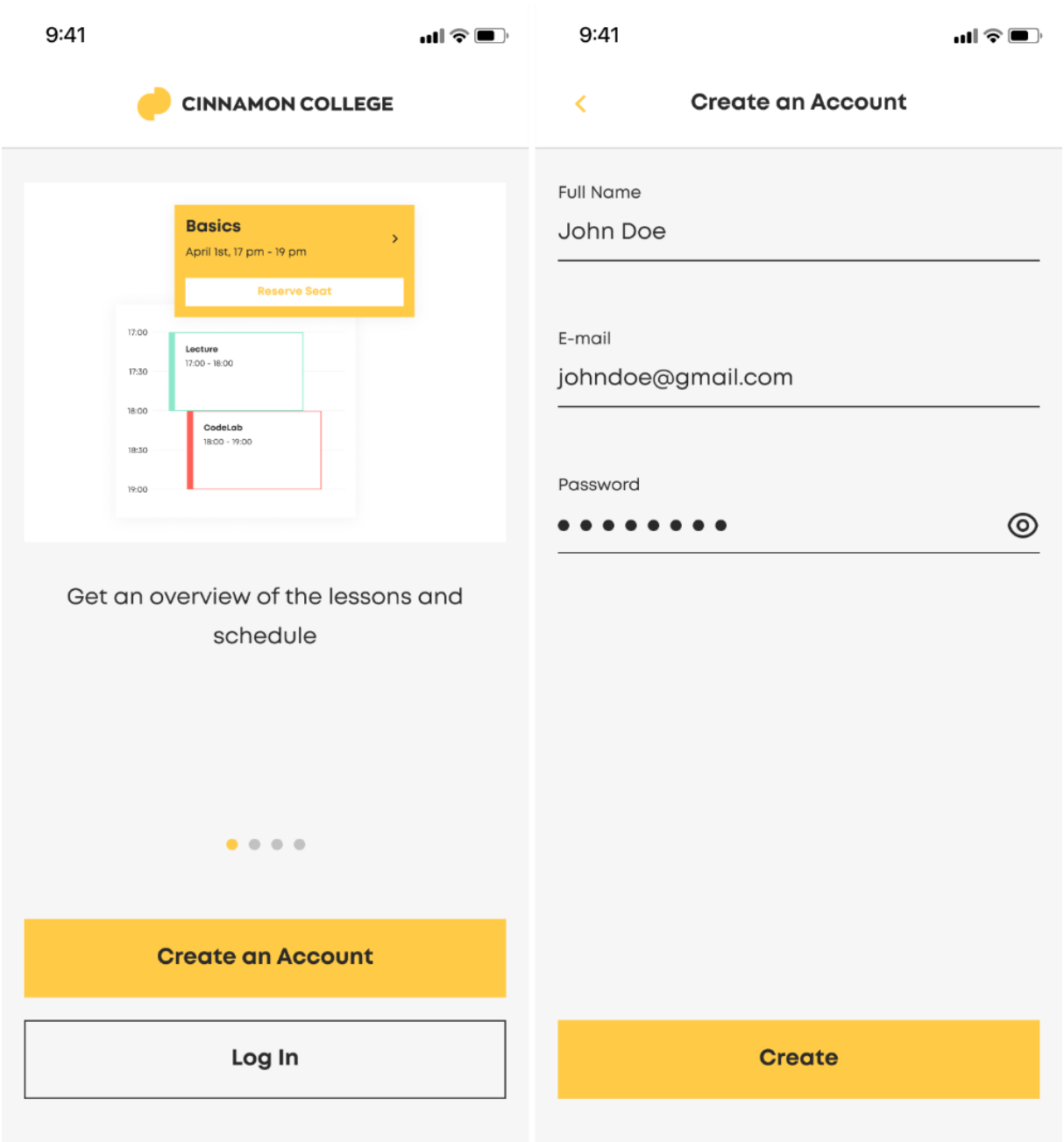
6.1. Funkcionalnosti programskog proizvoda

Aplikacija „Cinnamon College“ namijenjena je polaznicima tečaja u organizaciji tvrtke Cinnamon. U nastavku je priložena Tablica 6 s popisom funkcionalnosti aplikacije.

Tablica 6: Popis funkcionalnosti aplikacije

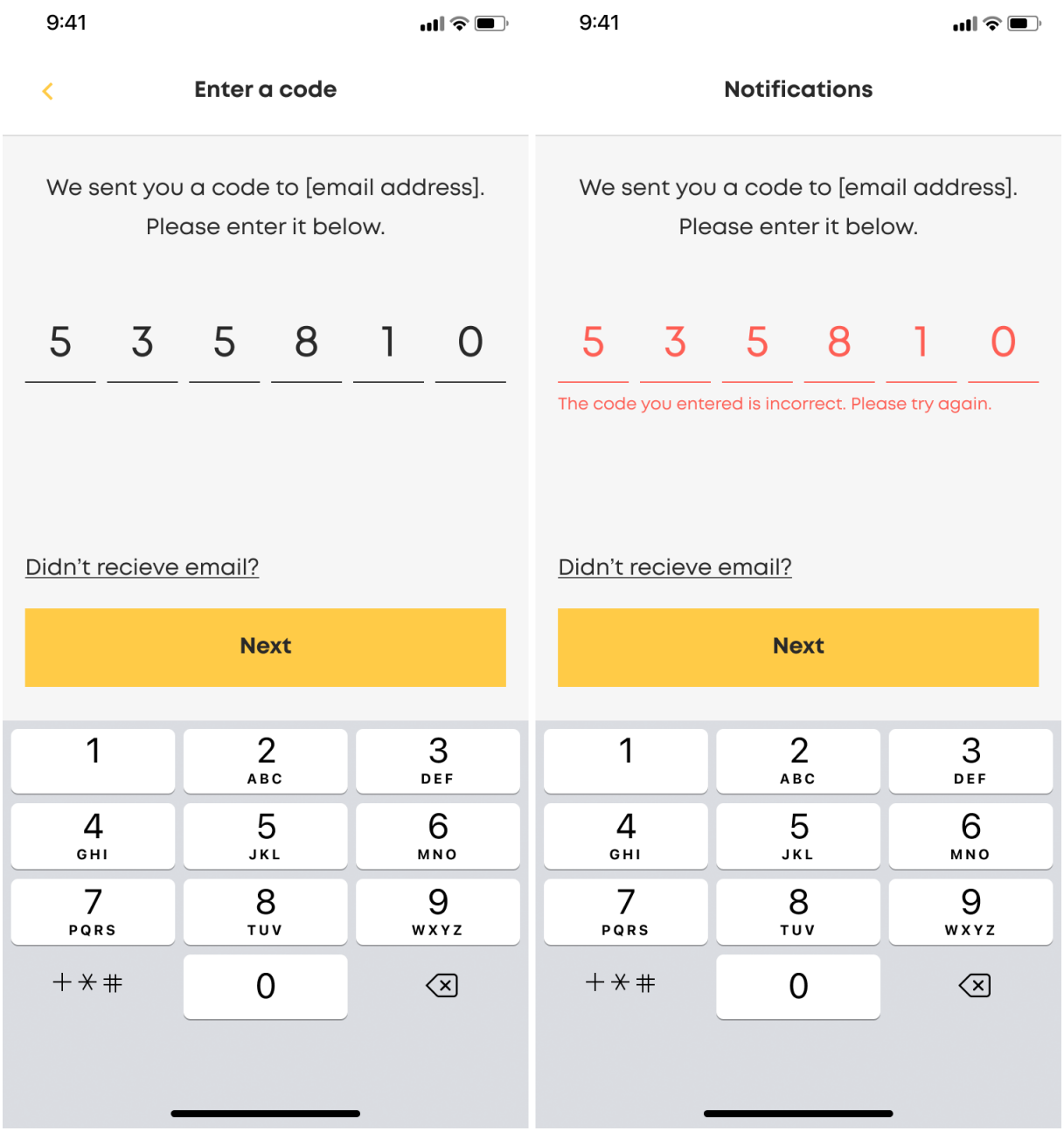
Oznaka	Naziv	Opis
F1	Registracija	Registracija korisnika s imenom, adresom e-pošte i lozinkom.
F2	Prijava	Prijava korisnika s adresom e-pošte i lozinkom.
F3	Pregled nadolazeće radionice	Prikaz osnovnih podataka i rasporeda nadolazeće radionice na početnoj stranici.
F4	Pregled detalja radionice	Prikaz detaljnih podataka i povezanih datoteka.
F5	Rezervacija stola	Odabir i rezervacija stola za nadolazeću radionicu.
F6	Pregled radionica	Popis svih nadolazećih i prošlih radionica na jednom zaslonu sortiranih po vremenu izvođenja.
F7	Ocjenjivanje radionice	Ocjenjivanje prošle radionice ocjenom od jedan do pet.
F8	Pregled radionica na kalendaru	Pregled svake radionice u pripadajućem danu u mjesecu.
F9	Obavijesti za radionice	Primanje obavijesti o nadolazećim radionicama i pregled svih obavijesti unutar aplikacije.
F10	Postavke korisničkog profila	Uređivanje slike profila, imena, adrese e-pošte i ciljeva. Uključivanje i isključivanje obavijesti.

Za početak, korisnici se moraju registrirati kako bi mogli koristiti aplikaciju. Proces registracije sadrži nekoliko koraka, a prvi korak je unos korisničkih podataka (Slika 16). Na zaslon za registraciju dolazi se putem početnog zaslona za neregistrirane korisnike.



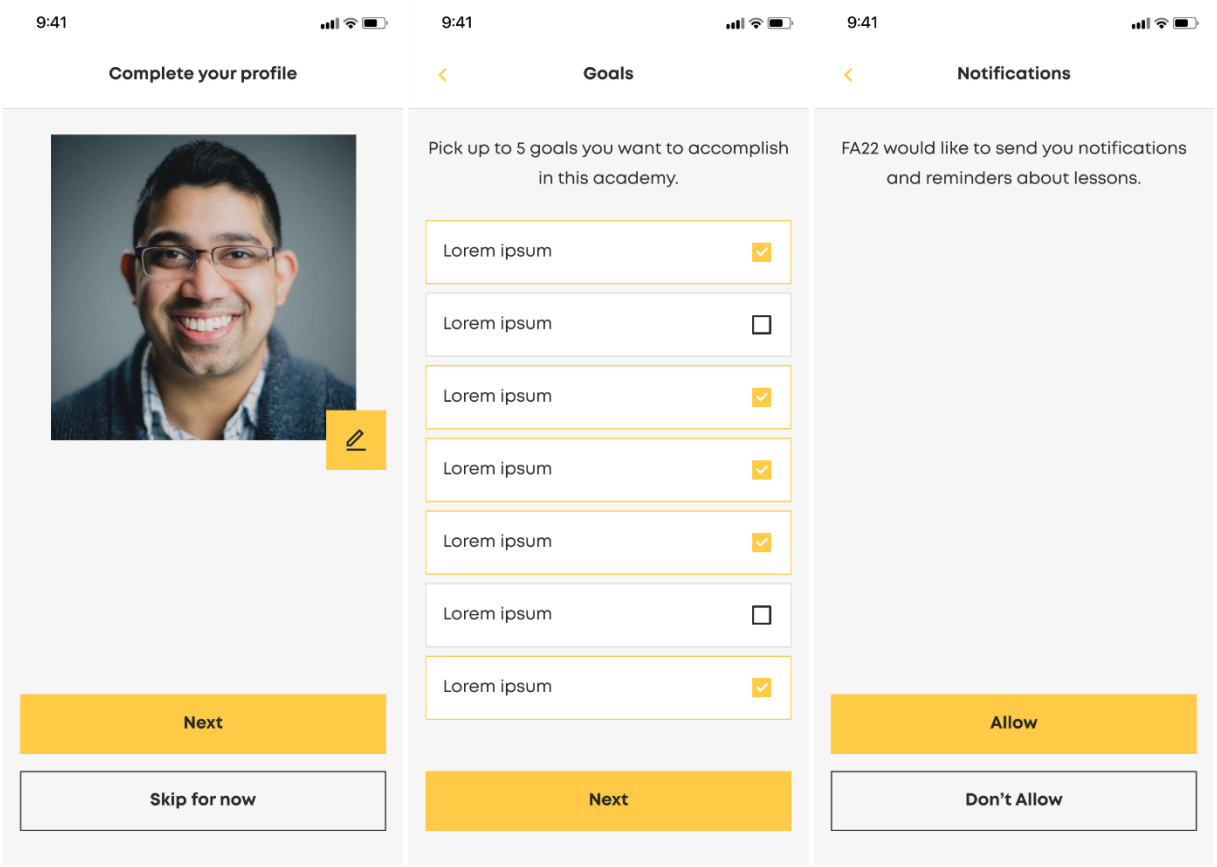
Slika 16: Prvi korak registracije

Drugi korak registracije je unos koda koji se dobije u poruci elektroničke pošte. Taj kod služi za aktivaciju korisničkog računa. Ako je uneseni kod pogrešan, na zaslonu se prikazuje odgovarajuća poruka. Obje situacije su prikazane na slici (Slika 17).



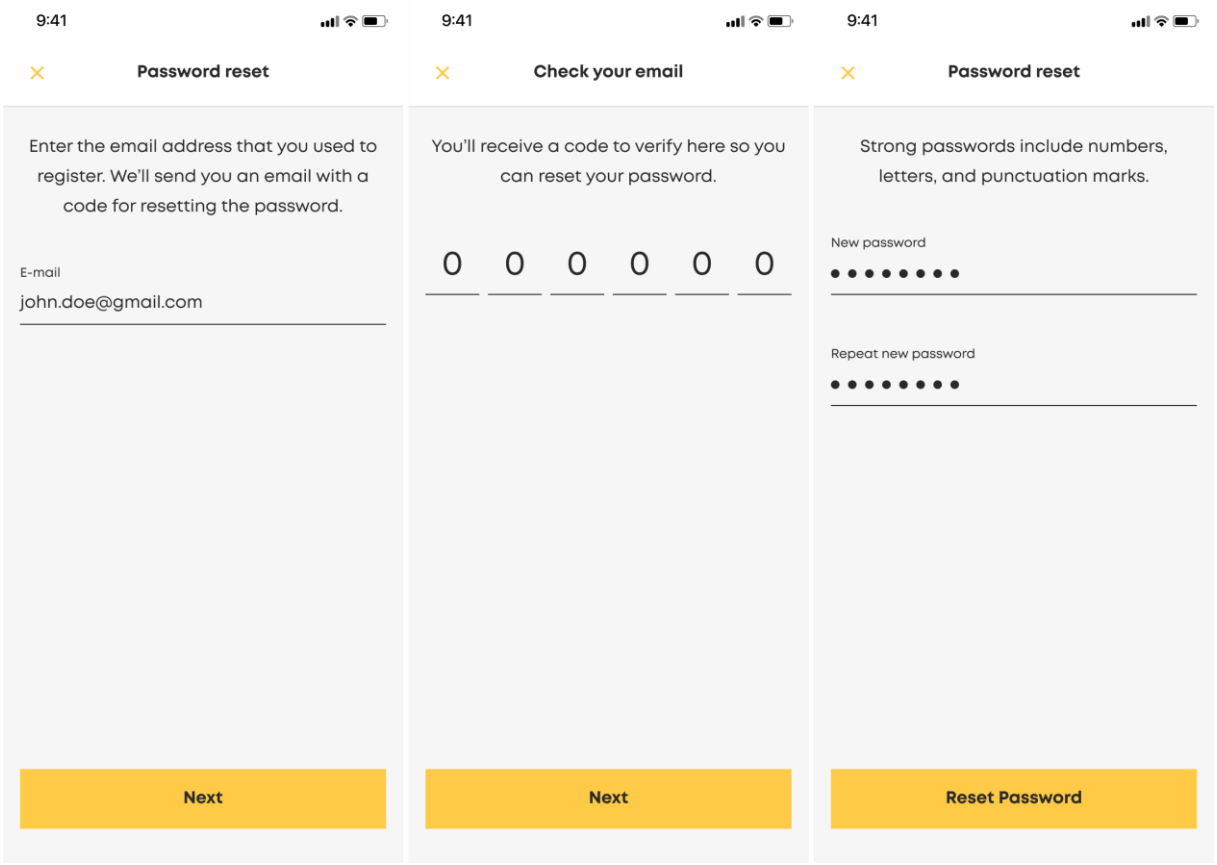
Slika 17: Drugi korak registracije

Treći, posljednji korak registracije je postavljanje korisničkih preferencija koje uključuju postavljanje slike profila, odabir ciljeva i dopuštenje primanja obavijesti. Na slici (Slika 18) su prikazana sva tri potkoraka, a nakon njih slijedi ulazak u aplikaciju.



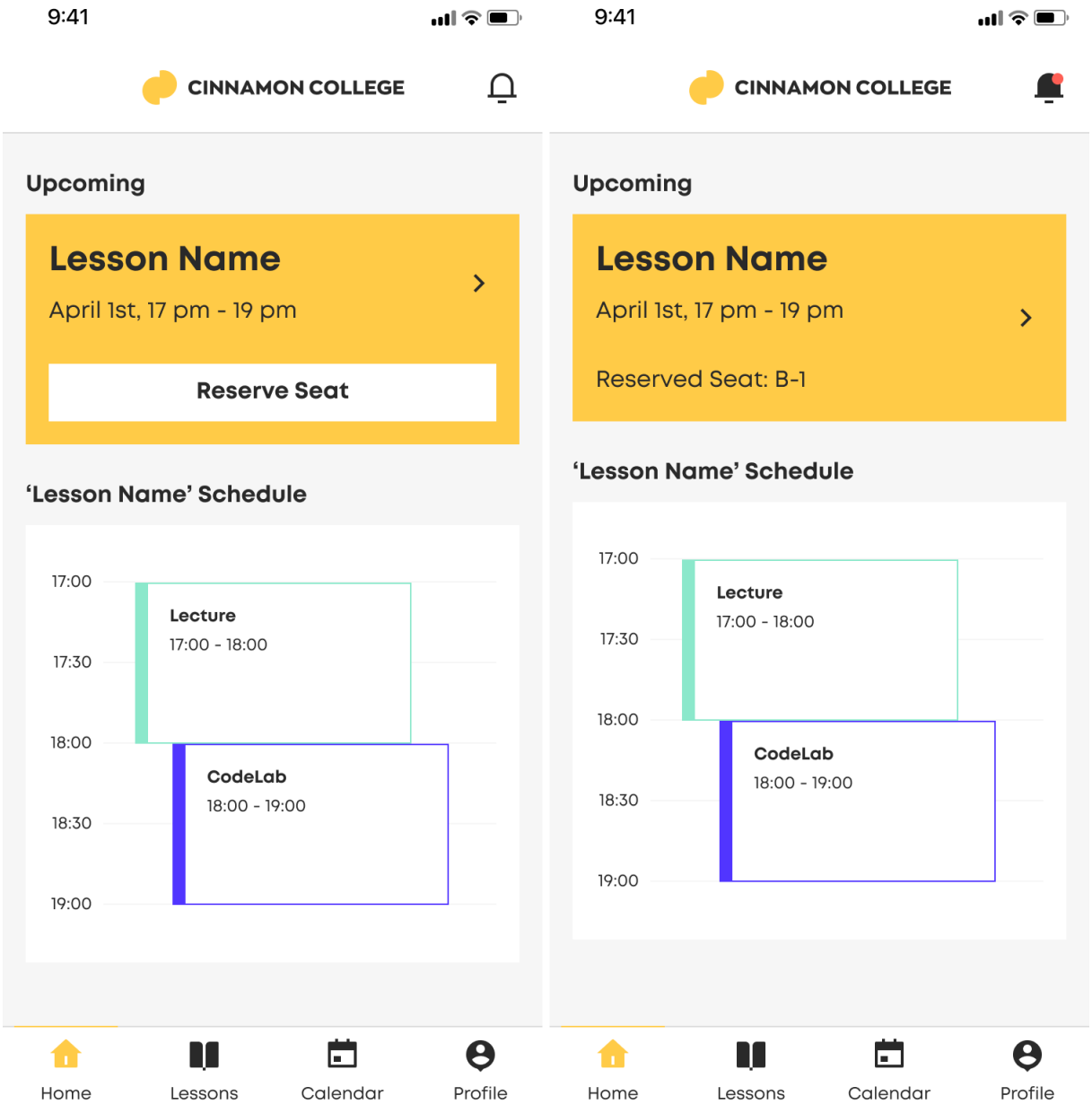
Slika 18: Treći korak registracije

Registrirani korisnik može se prijaviti u aplikaciju i promijeniti lozinku. Zaslon za prijavu istovjetan je zaslonu za registraciju, osim što nije potrebno unositi ime. Za promjenu lozinke potrebno je upisati adresu elektroničke pošte na koji se šalje kod za potvrdu računa. Nakon uspješnog unosa koda, korisniku se otvara zaslon za unos nove lozinke. Navedeni koraci vidljivi su na slici (Slika 19).



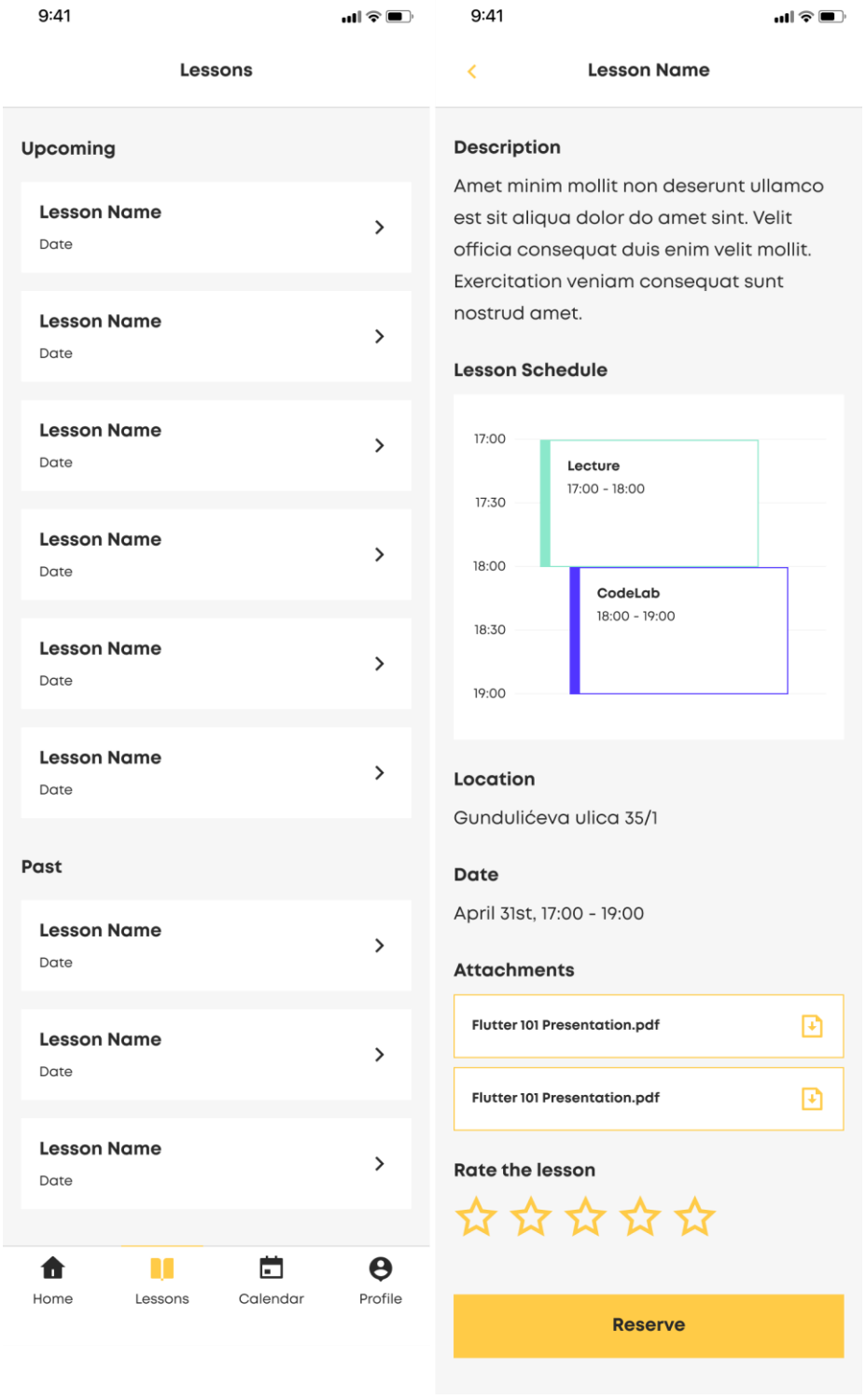
Slika 19: Promjena lozinke

Korisniku se nakon prijave otvara početni zaslon koji prikazuje nadolazeću radionicu i raspored održavanja teorijskog i praktičnog dijela radionice. S početnog zaslona moguće je doći do zaslona za rezervaciju stola, a nakon rezervacije ispisuje se broj rezerviranog stola. Na slici (Slika 20) moguće je vidjeti dvije verzije početnog zaslona.



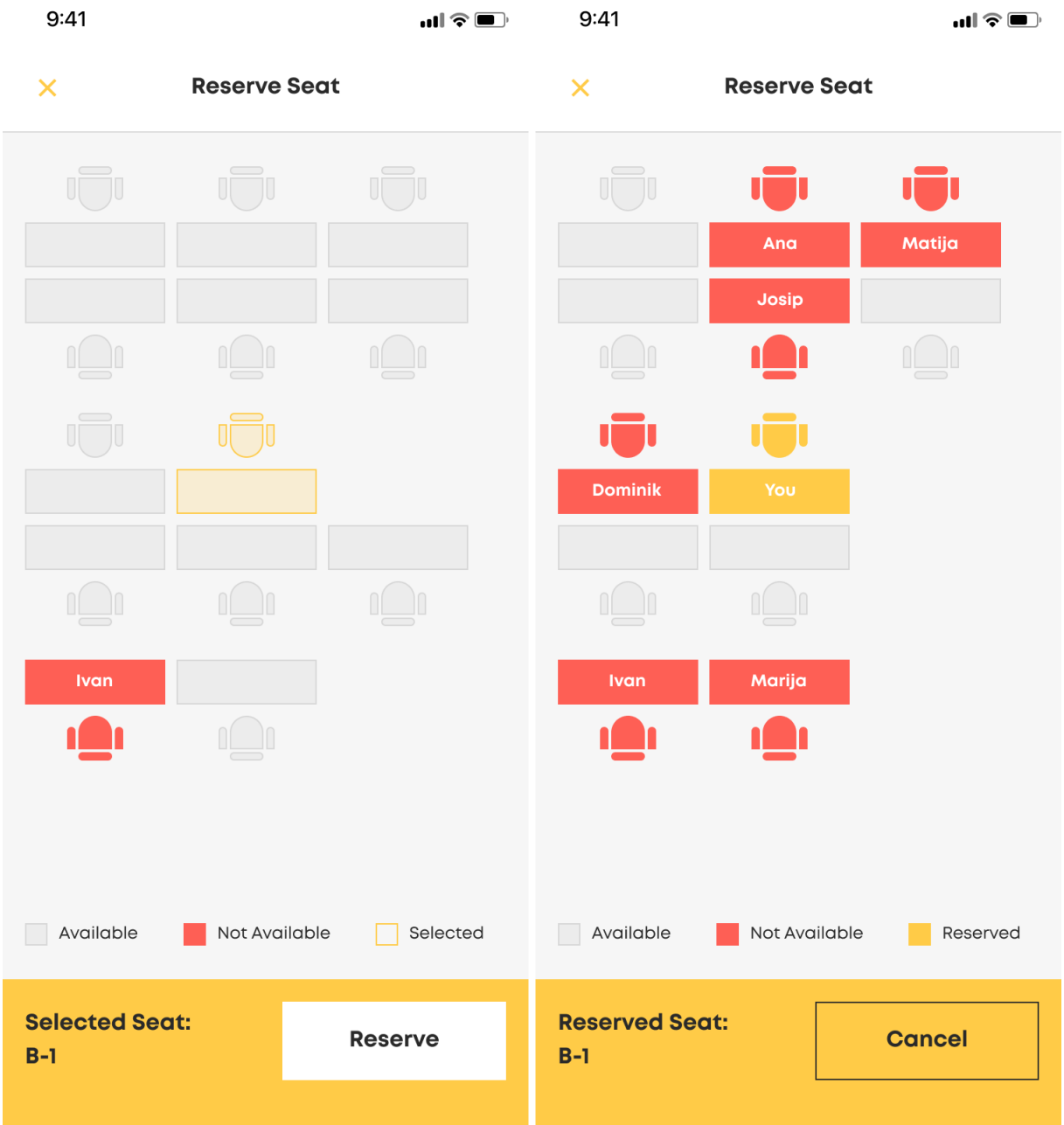
Slika 20: Početni zaslon

Na početnom zaslonu nalazi se navigacijska traka pomoću koje je moguće doći do ostalih zaslona. Prvi takav po redu je zaslon s popisom budućih i prošlih radionica (Slika 21). Odabirom jedne od radionica otvara se zaslon s detaljima radionice poput opisa, vremena i mjesta održavanja te dokumenata za preuzimanje.



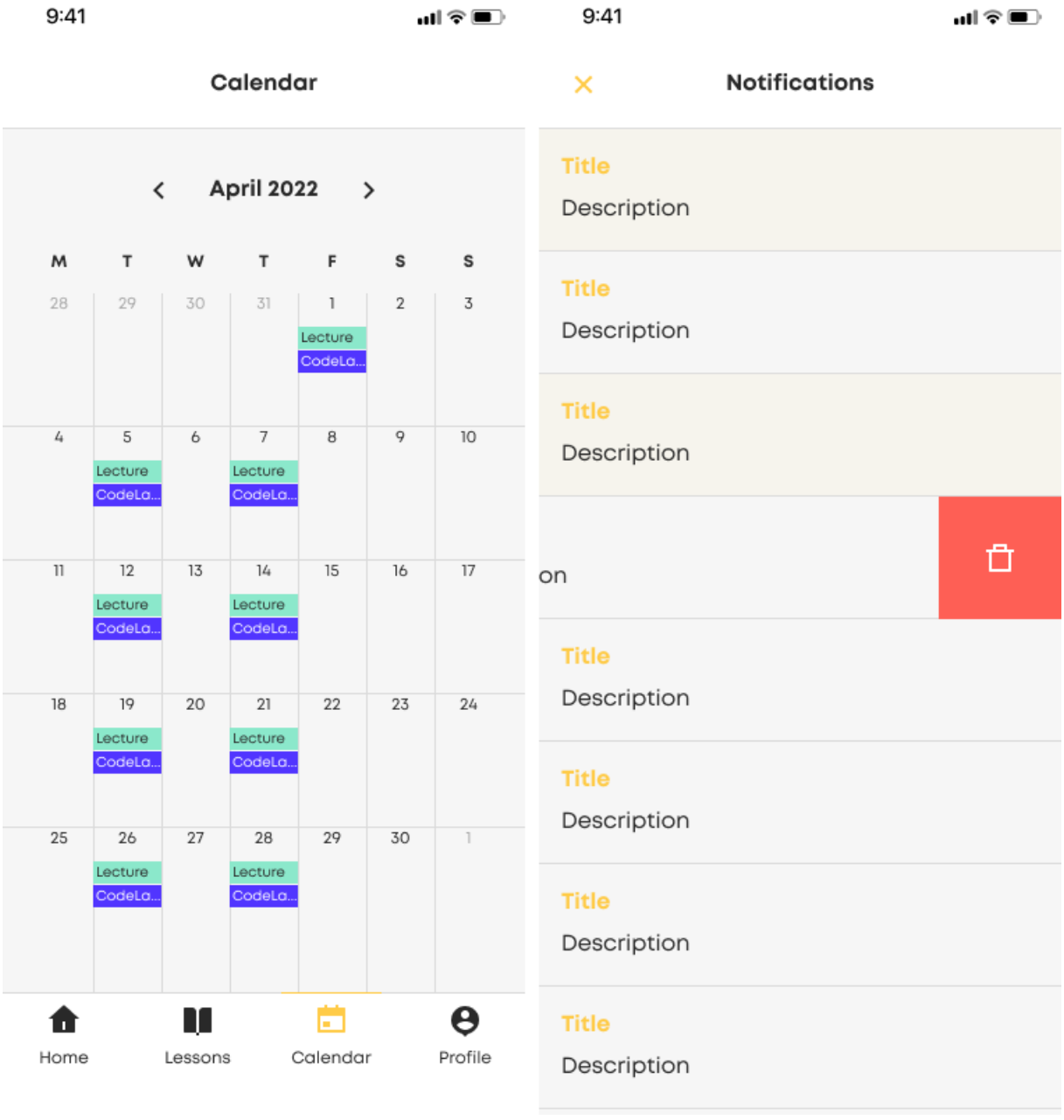
Slika 21: Zasloni s popisom radionica i njihovim detaljima

Na slici (Slika 22) prikazan je zaslon za rezervaciju stola, a do njega se dolazi putem zaslona s detaljima radionice. Crvena boja označava tuđu rezervaciju, dok žuta boja označava rezervaciju prijavljenog korisnika. Ako je stol samo odabran, tada je osjenčan žutom bojom.



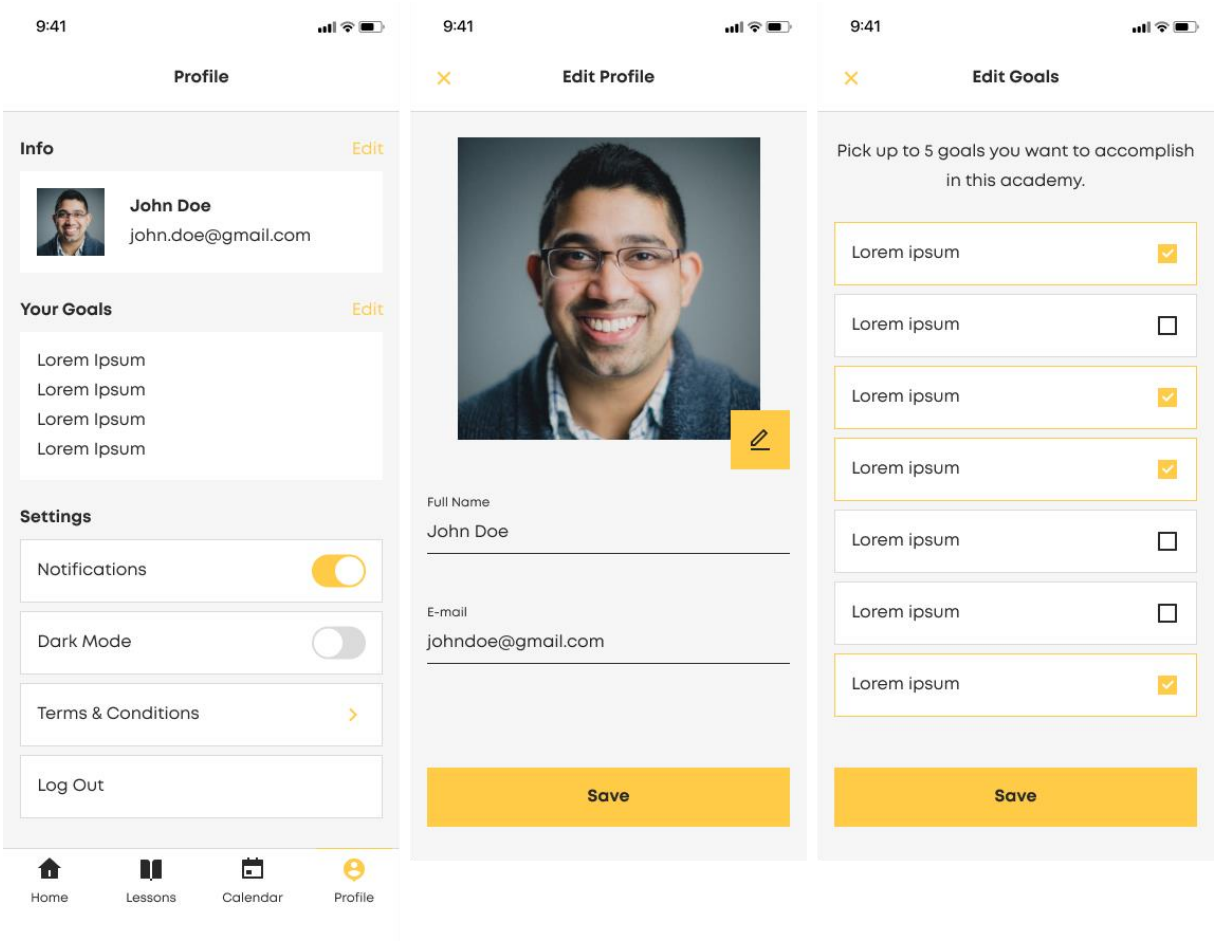
Slika 22: Zaslon za rezervaciju stola

Na slici (Slika 23) prikazane su dvije od tri preostale funkcionalnosti, a to je zaslon s kalendarom (lijevo) i zaslon s obavijestima (desno). Na kalendaru se prikazuje raspored radionica podijeljene na teorijski i praktični dio. Zaslon s obavijestima, kao što i ime govori, prikazuje popis pročitanih i nepročitanih obavijesti.



Slika 23: Zaslon s kalendarom (lijevo) i obavijestima (desno)

Preostala funkcionalnost je uređivanje korisničkog profila. Korisnik može promijeniti sliku profila, ime i adresu elektroničke pošte te odabrati nove ciljeve. Na istom zaslonu može uključiti ili isključiti obavijesti, odjaviti se i uključiti tamni način rada. Sve navedene opcije prikazane su na slici (Slika 24).



Slika 24: Uređivanje korisničkih podataka

U ovom poglavlju vidjeli smo glavne funkcionalnosti aplikacije čija će implementacija biti opisana u nadolazećim poglavljima. Prototip aplikacije odnosno sve korištene slike u ovom potpoglavlju izradio je tim dizajnera tvrtke Cinnamon.

6.2. Implementacija programskog proizvoda

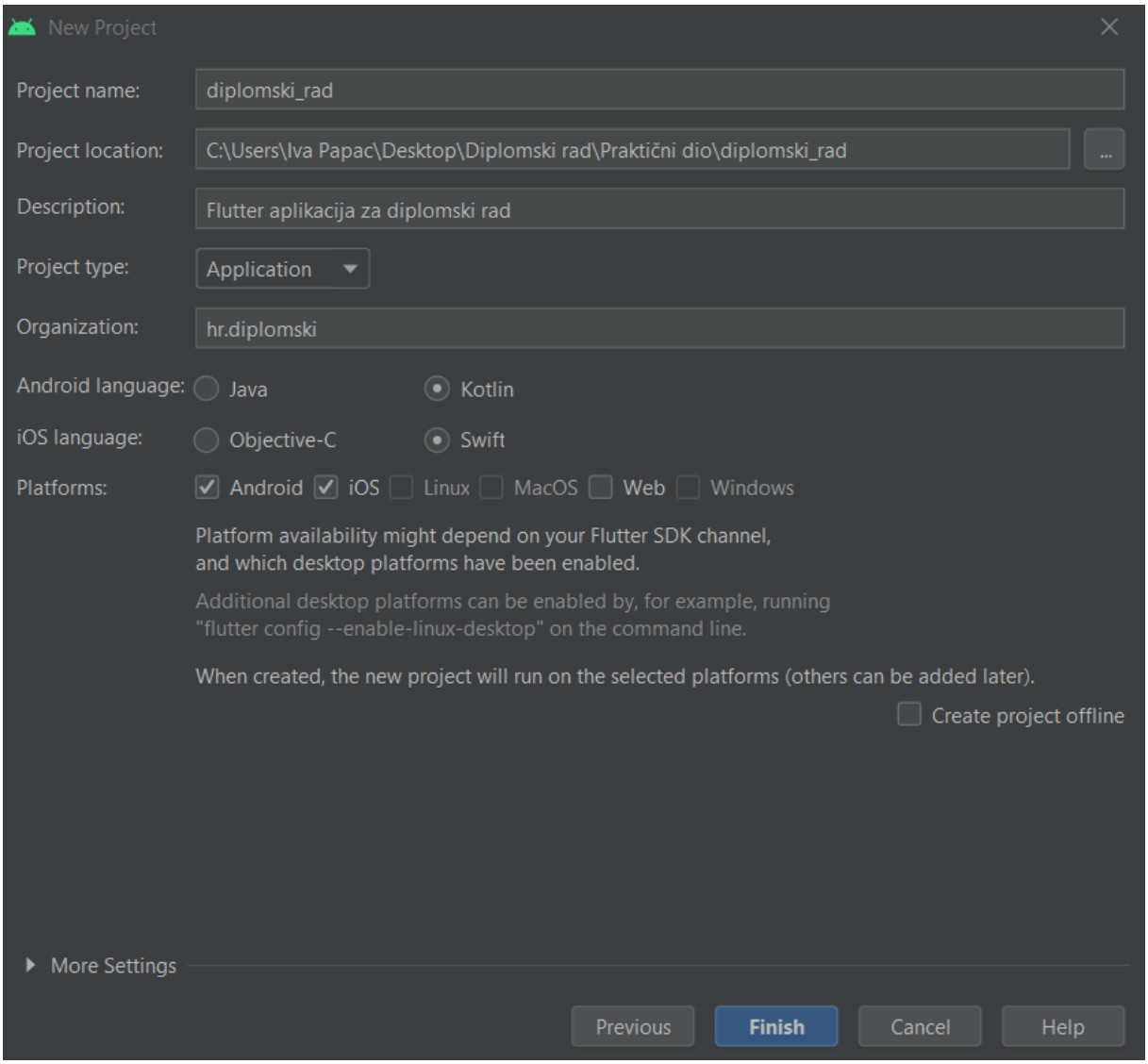
U prethodnom poglavlju opisane su funkcionalnosti programskog proizvoda koje su tijekom procesa razvoja programskog proizvoda implementirane. Proces razvoja započeo je postavljanjem razvojne okoline, odnosno postavljanjem integriranog razvojnog okruženja, sustava za verzioniranje programskog koda i alata za kontinuiranu integraciju. Implementacija programskog proizvoda provodi se u manjim ciklusima koji uključuju pisanje programskog koda i pisanje testova za novi programski kod te verzioniranje i integraciju programskog koda. Pri implementaciji korištena je paradigma MVC arhitekture zbog već spomenutih razloga u teorijskom dijelu rada, na primjer lakša čitljivost i skalabilnost programskog koda. Na poslužiteljskoj strani korištena je platforma Firebase za pohranu podataka i datoteka, registraciju i autentikaciju korisnika te kreiranje JavaScript funkcija za slanje poruka e-pošte i za slanje obavijesti. U nastavku su opisani koraci razvoja programskog proizvoda u kontekstu DevOps-a.

6.2.1. Postavke alata

Za nesmetani rad i implementaciju svih funkcionalnosti potrebno je prvo postaviti sve alate. Za integrirano razvojno okruženje korišten je Android Studio, dok je za verzioniranje programskog koda korišten GitHub. Ono što je novo u procesu razvoja je uvođenje alata za kontinuiranu integraciju, a odabran je alat Codemagic. Detaljni opis i postavke svih alata nalaze se u nastavku poglavlja.

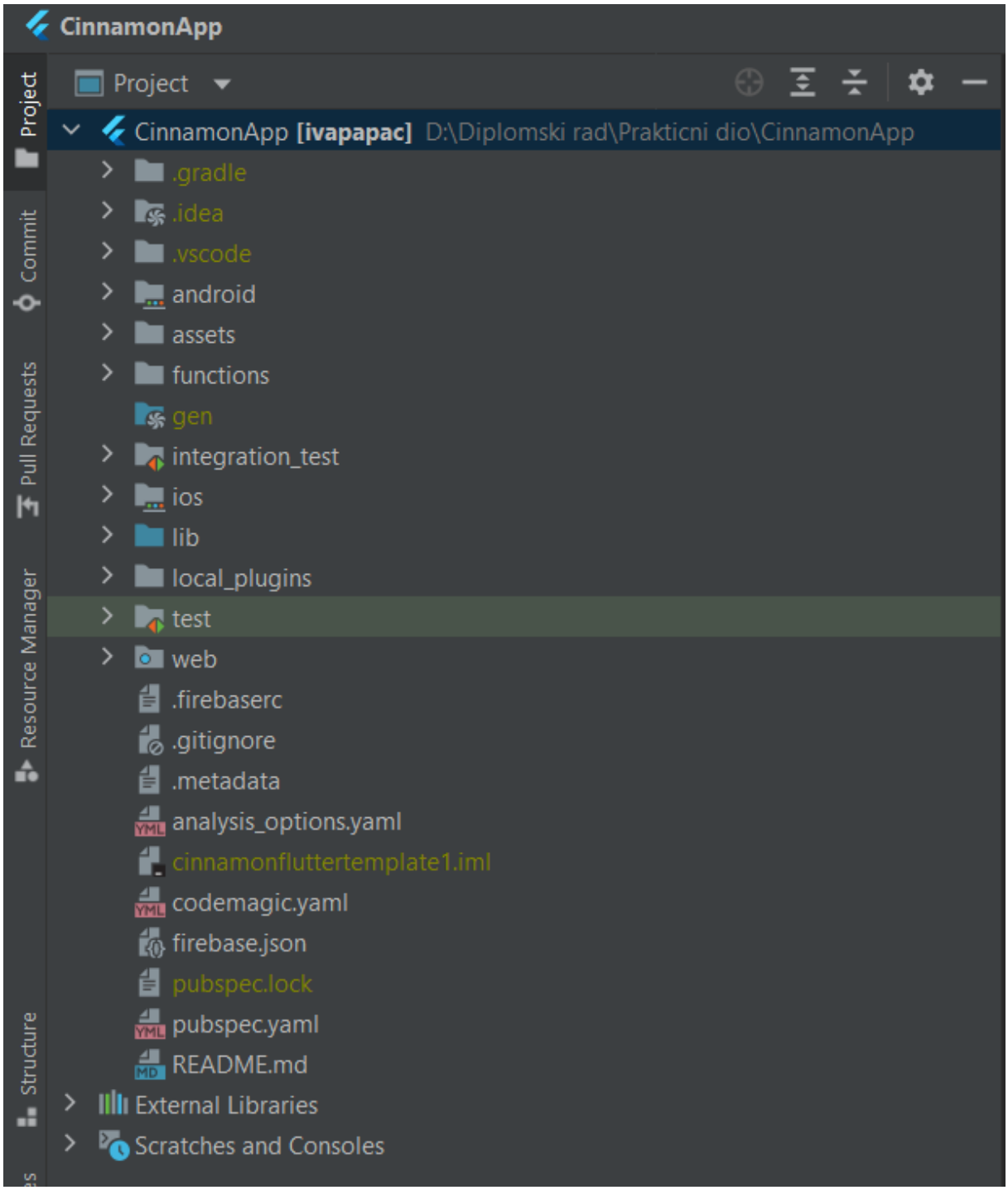
6.2.1.1. Android Studio

Android Studio je integrirano razvojno okruženje za razvoj mobilnih aplikacija, a produkt je suradnje tvrtka Google i JetBrains. Najnovija dostupna verzija zove se Android Studio Chipmunk te je ona korištena u sklopu praktičnog rada [40]. Uz pretpostavku da su instalirane sve tehnologije za razvoj Flutter aplikacije, moguće je započeti s kreiranjem Flutter projekta u alatu Android Studio. Prvo se odabire vrsta projekta i putanja do instalacijske mape Fluttera. Nakon toga slijedi definiranje naziva, putanje i opisa projekta te odabir platforma i programskih jezika za mobilnu aplikaciju (Slika 25).



Slika 25: Kreiranje novog projekta

Android Studio automatski kreira potrebne datoteke i direktorije za razvoj Flutter aplikacije, a među najvažnijima su datoteka `pubspec.yaml` te direktoriji *lib*, *android* i *ios* (Slika 26). U poglavlju 3 detaljno je opisana uloga svake od spomenutih datoteka i direktorija. U procesu razvoja programskog proizvoda najviše se koristi datoteka `pubspec.yaml` za dodavanje biblioteka i paketa trećih strana te direktorij *lib* u kojem se nalazi sav izvorni programski kod. U poglavlju 6.2.2 detaljnije je opisana struktura direktorija *lib*, a struktura i uloga direktorija *integration_test* i *test* opisana je u poglavlju 6.3.



Slika 26: Struktura direktorija Flutter projekta

Android Studio moćan je alat koji nudi brojne mogućnosti, a u ovom praktičnom radu korišteno ih je nekoliko. Korišten je Android Emulator, virtualni Android uređaj, koji služi za pokretanje aplikacije i provođenje integracijskog testiranja. Također je korišten naredbeni redak i Git opcije za postavljanje novog programskog koda na udaljeni GitHub repozitorij.

6.2.1.2. GitHub

Odabrana platforma za verzioniranje programskog koda, GitHub, detaljno je opisana u poglavlju 4.3.2. Kreirani projekt u alatu Android Studio iz prethodnog poglavlja potrebno je povezati s udaljenim repozitorijem. Prvi korak je kreiranje repozitorija putem naredbenog retka pomoću Git naredbi, a u primjeru Isječak kôda 23 moguće je vidjeti naredbe koje je potrebno izvršiti za inicijalizaciju repozitorija i postavljanje prvih promjena.

Isječak kôda 23: Inicijalizacija GitHub repozitorija

```
git init
git add .
git commit -m 'First commit'
```

Drugi korak je povezivanje lokalnog repozitorija s udaljenim GitHub repozitorijem kako bi postavljene promjene bile vidljive i tamo. U primjeru Isječak kôda 24 također je moguće vidjeti Git naredbe koje služe za kreiranje *main* grane, povezivanje lokalne grane s granom na udaljenom repozitoriju te postavljanje novih promjena na udaljeni repozitorij.

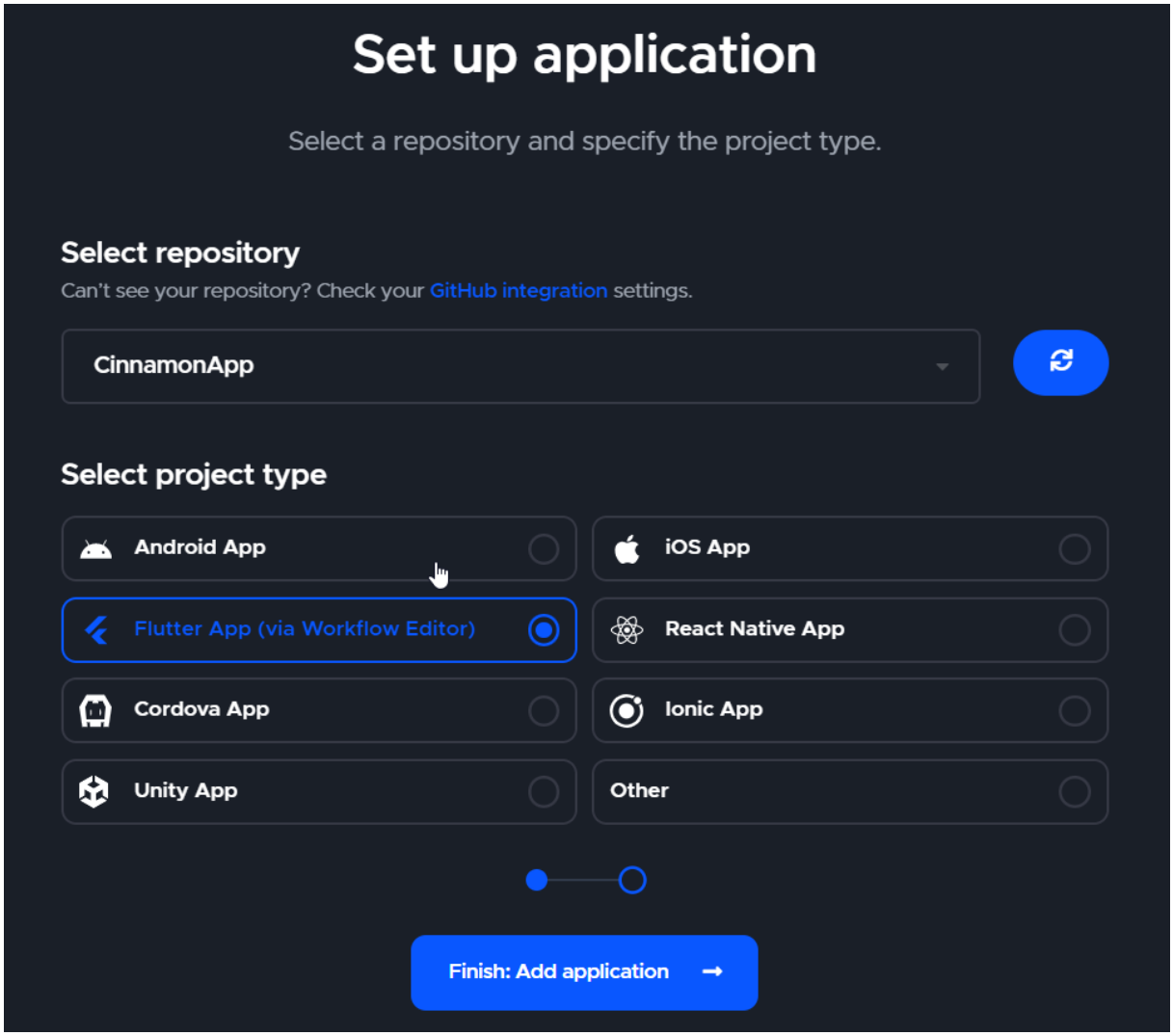
Isječak kôda 24: Povezivanje lokalnog i udaljenog repozitorija

```
git branch -M main
git remote add origin https://github.com/ivapapac/DiplomskiRad.git
git push -u origin main
```

Postavljanjem alata Android Studio i GitHub napravljen je prvi korak u procesu razvoja programskog proizvoda. Ovi alati predstavljaju i temelj za implementaciju praksi kontinuirane integracije i kontinuiranog testiranja, što je već moguće vidjeti u sljedećem poglavlju.

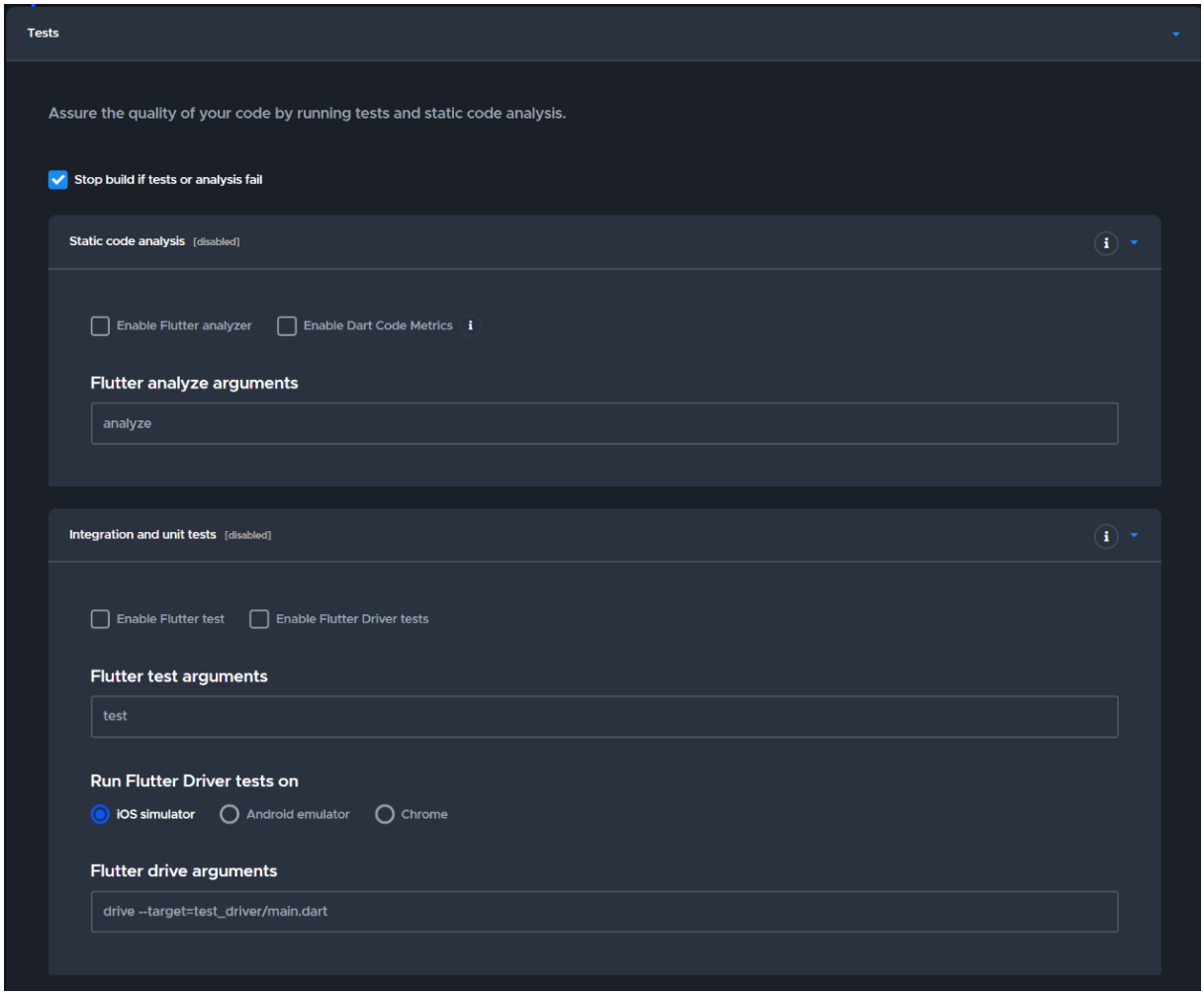
6.2.1.3. Codemagic

Codemagic je alat koji je ekskluzivno kreiran kao rješenje za kontinuiranu integraciju i kontinuiranu isporuku Flutter aplikacija u suradnji Googlea i Nevercodea [41]. Prvi korak postavki je povezati Codemagic račun s aplikacijom koja se nalazi u GitHub repozitoriju. Moguće je odabrati repozitorij i s drugih platforma za verzioniranje programskog koda kao što su Bitbucket i GitLab. Nakon što se odabere platforma, potrebno je odabrati repozitorij i vrstu projekta. U ovom slučaju repozitorij je CinnamonApp i vrsta projekta je Flutter aplikacija (Slika 27).



Slika 27: Dodavanje GitHub repozitorija u Codemagic

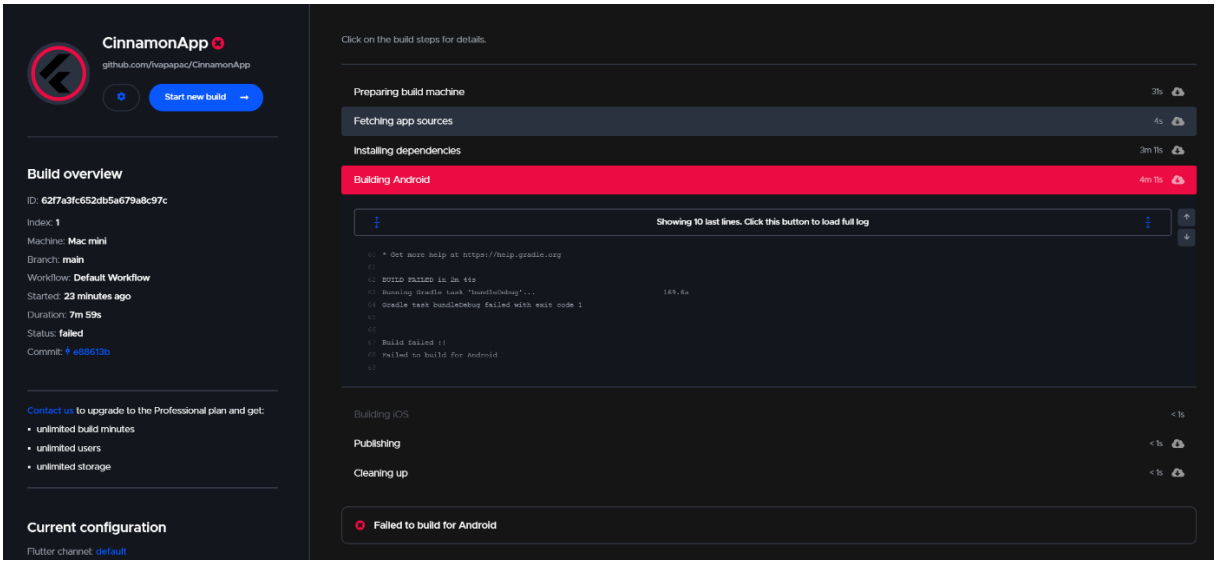
Nakon dodavanja aplikacije, alat vodi do postavka cjevovodi za automatizaciju izgradnje, testiranja i isporuke aplikacije. Postavke se mogu podesiti pomoću grafičkog sučelja ili pisanjem skripte u datoteku codemagic.yaml. Na slici (Slika 28) su prikazane postavke jediničnog i integracijskog testiranja te statičke analize koda putem grafičkog sučelja. Osim toga, moguće je odabrati za koje platforme će se izgraditi aplikacija, definirati okidače za izgradnju, postaviti varijable okoline, urediti postavke izgradnje poput Flutter verzije, putanje projekta i drugih argumenata te podesiti isporuku aplikacije na internetske trgovine mobilnih aplikacija kao što su Google Play i App Store.



Slika 28: Postavke automatiziranog testiranja

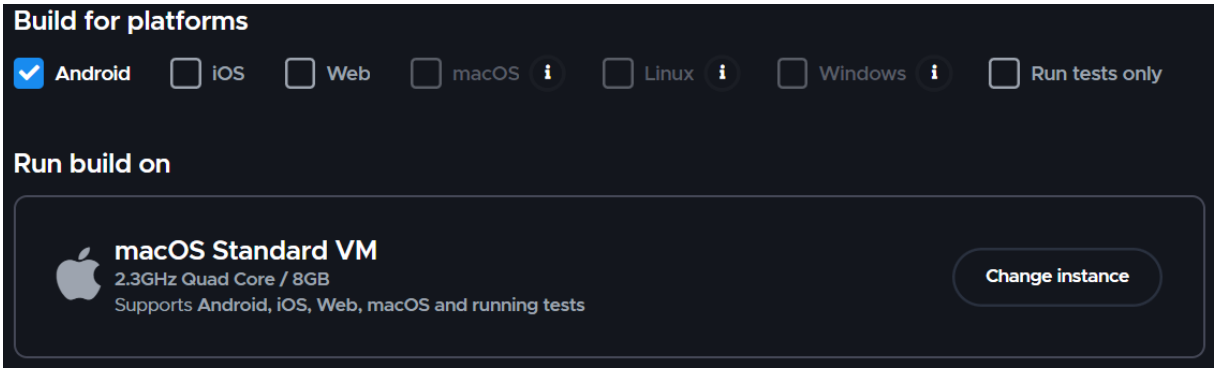
S prikazanim inicijalnim postavkama pokrenut će se prva izgradnja aplikacije. Tako će se vidjeti što sve od postavka treba urediti. Na slici (Slika 29) vidljivo je da izgradnja s inicijalnim postavkama nije bila uspješna, no za svaki od koraka prikazanog na slici moguće je preuzeti datoteku sa zapisima o izvršenim naredbama i eventualnim greškama *(eng. log file)*. U njoj se može pronaći detaljnije obrazloženje zašto izgradnja aplikacije za Android platformu nije bila uspješna. Greška se desila prilikom izvršavanja *Gradle* zadatka, a to dokazuje i sljedeća linija

iz datoteke: `Execution failed for task ':app:compileFlutterBuildDebug'`. Problem je nastao jer nisu postavljene varijable okoline za Firebase bazu podataka.



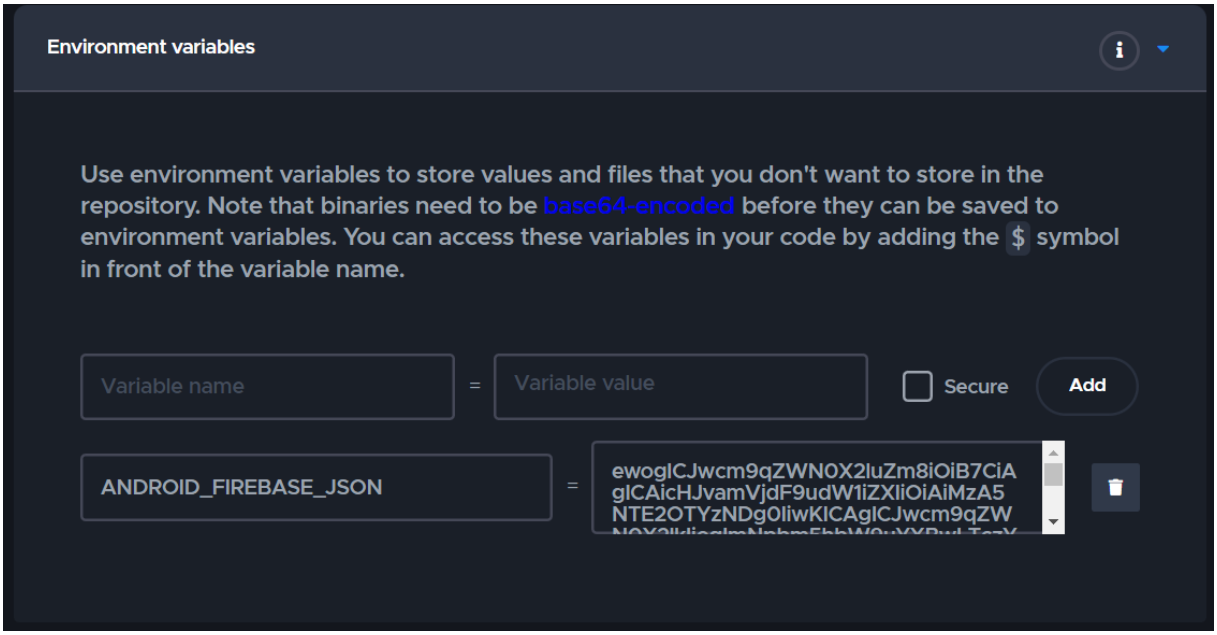
Slika 29: Neuspješna izgradnja aplikacije

Prethodno prikazane inicijalne postavke cjevovodi za automatizaciju izgradnje, testiranja i isporuke promijenjene su uz pomoć grafičkog sučelja. Na slici (Slika 30) je vidljivo da je izgradnja odabrana samo za Android platformu i da će se izvršavati na macOS virtualnoj mašini.



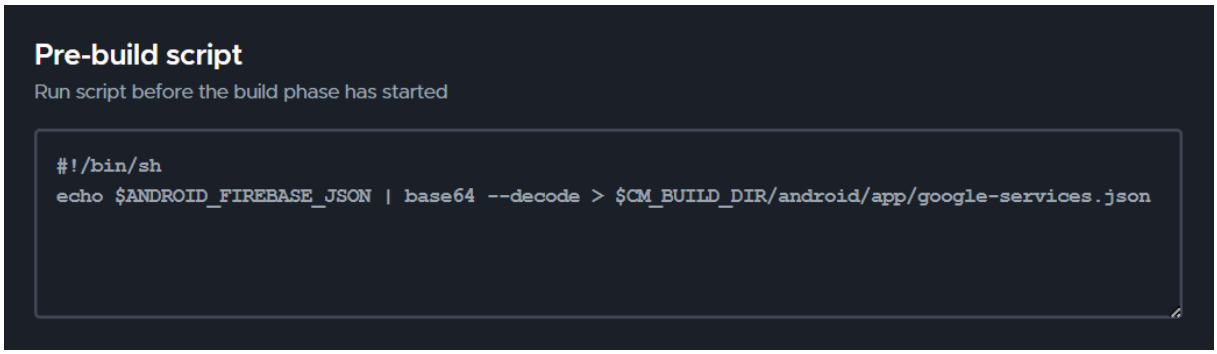
Slika 30: Odabir platforme za izgradnju

Sljedeći korak je podešavanje varijable okoline (Slika 31). Budući da se datoteka `google-services.json` iz sigurnosnih razloga ne nalazi u javnom GitHub repozitoriju, dodana je u obliku šifrirane *(eng. encoded)* vrijednosti varijable okoline `ANDROID_FIREBASE_JSON`.



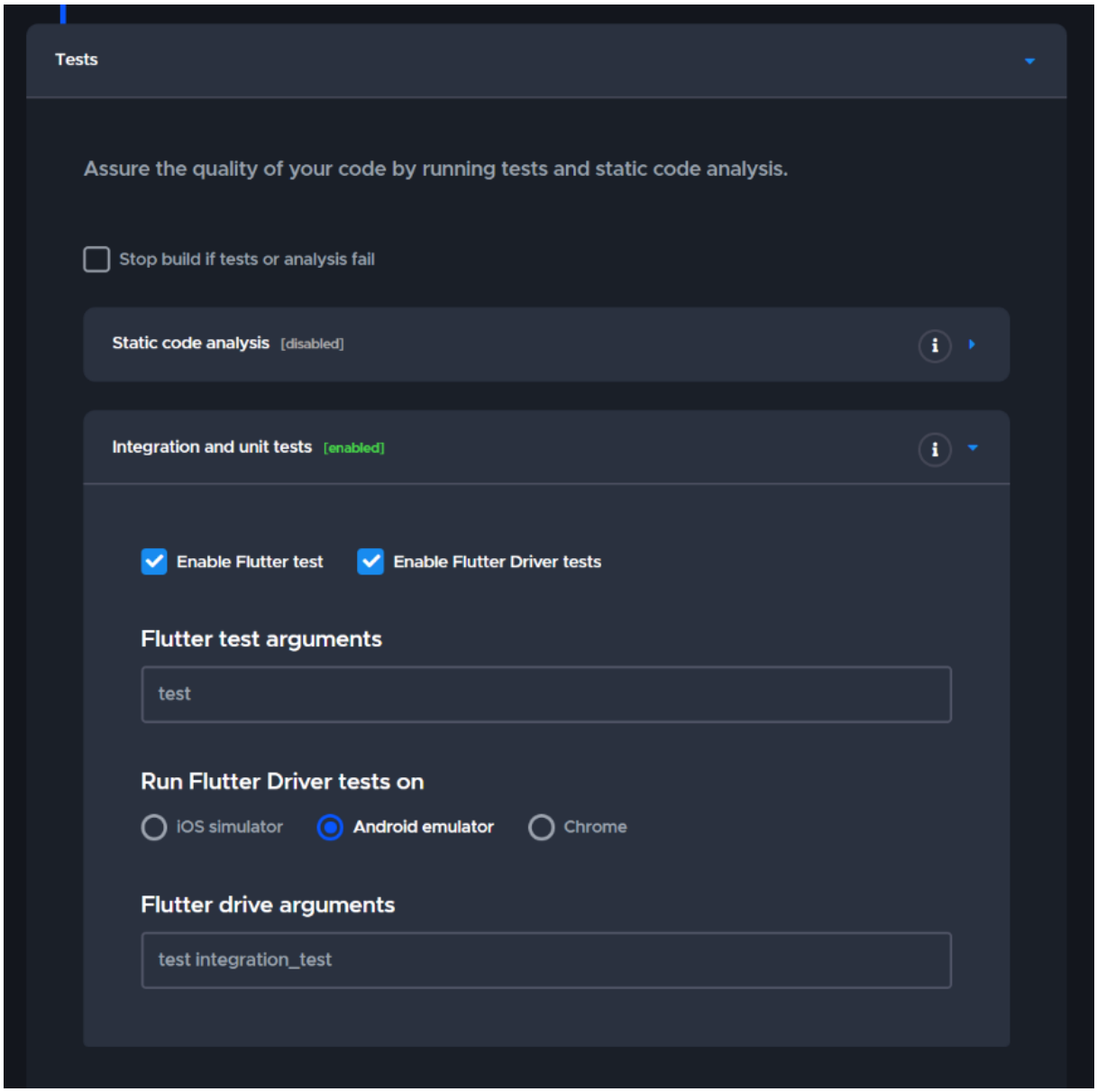
Slika 31: Postavke varijabla okoline

Sada je potrebno šifriranu datoteku dešifrirati pomoću skripte prikazanoj na slici (Slika 32). Vrijednost varijable okoline `ANDROID_FIREBASE_JSON` se dešifrira i sprema u obliku datoteke `google-services.json`. Ova skripta se izvodi prije pokretanja izgradnje aplikacije, ali i prije pokretanja integracijskih testova.



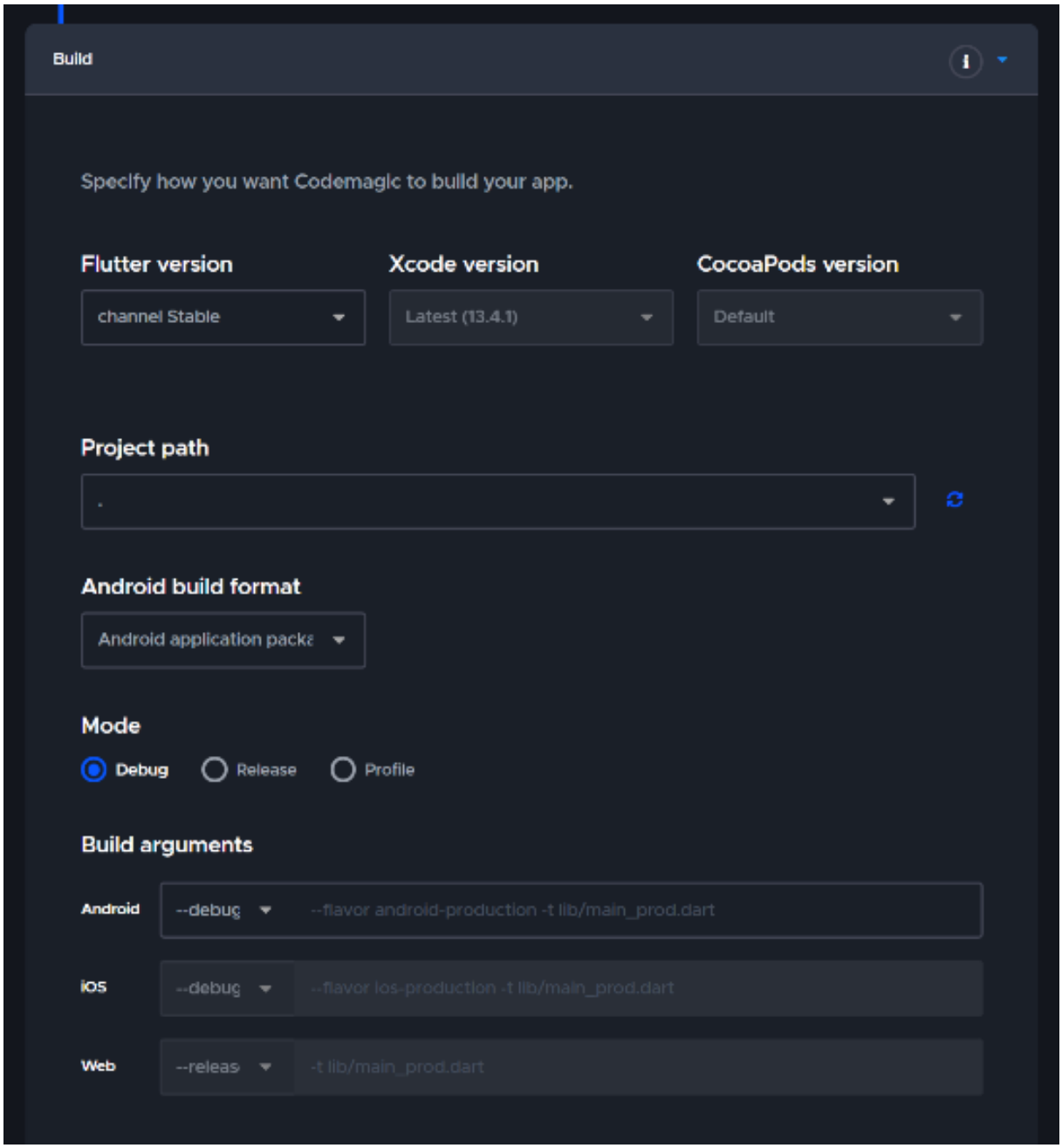
Slika 32: Skripta za dešifriranje datoteke google-services.json

Prije izgradnje programskog koda pokreću se automatizirani jedinični i integracijski testovi te testovi elemenata korisničkog sučelja. Na slici (Slika 33) je vidljivo da su omogućeni integracijski i jedinični testovi koji se pokreću na virtualnom Android uređaju. Opcije statička analiza koda i zaustavljanje izgradnje aplikacije u slučaju neuspjeha analize ili testova su isključene kako bi se uštedjelo na vremenu potrebnom za izvršavanje cijele skripte.



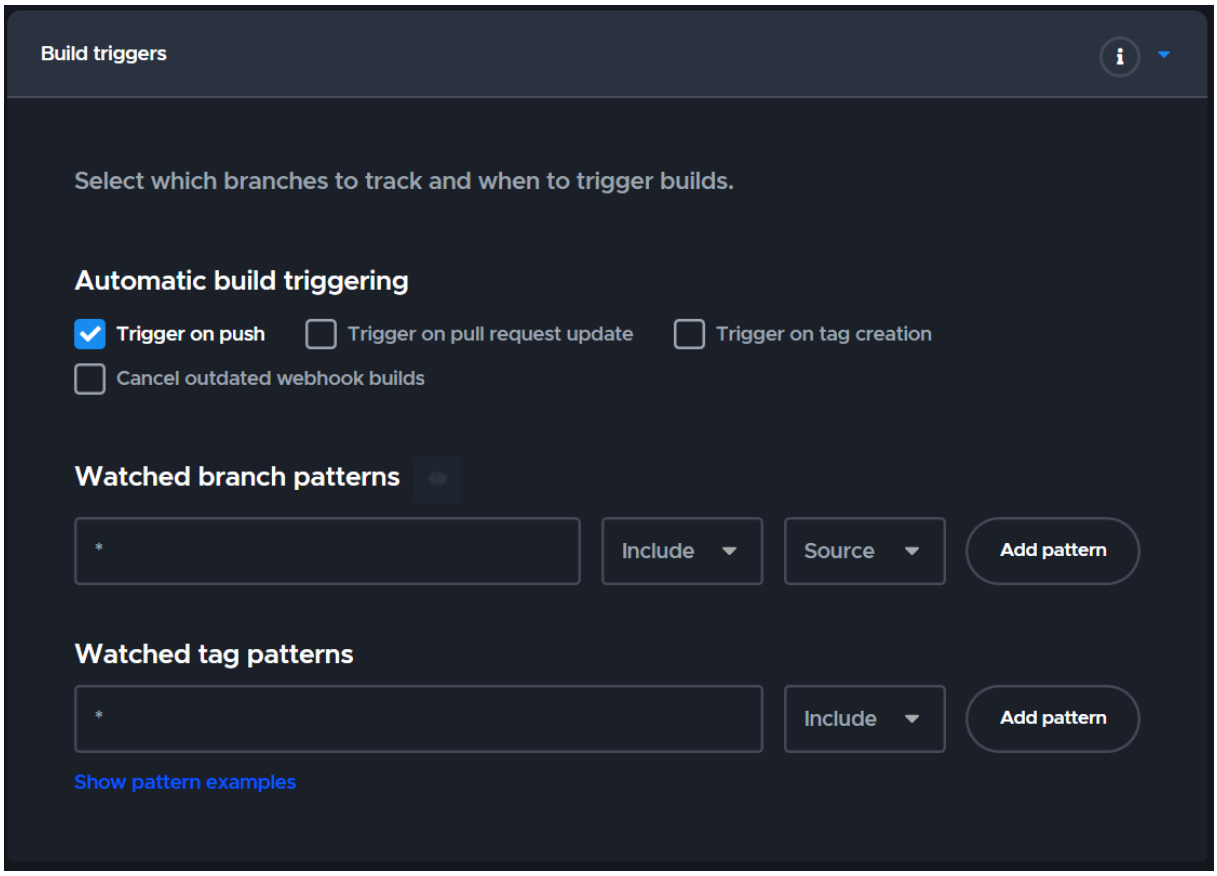
Slika 33: Postavke automatiziranog testiranja

Posljednja opcija koja je uređena je izgradnja aplikacije. Slika 34 prikazuje te postavke. Flutter verzija je postavljena na *channel Stable*, što znači da se radi o stabilnoj verziji. Putanja projekta je ishodišna mapa GitHub repozitorija, dok će se aplikacija izgraditi u APK *(eng. Android application package)* format.



Slika 34: Postavke izgradnje aplikacije

Osim spomenutih postavki, u alatu postoji opcija podešavanja okidača automatske izgradnje i testiranja programskog proizvoda. Riječ je o povezanosti alata s odabranim alatom za verzioniranje programskog koda. U ovom praktičnom radu po potrebi je korištena opcija automatske izgradnje prilikom dodavanja novog programskog koda na povezani GitHub repozitorij. Na slici dolje vidimo da postoje i druge opcije okidanja automatizirane izgradnje aplikacije poput kreiranje zahtjeva za postavljanjem novog programskog koda. Moguće je definirati grane na kojima će se pratiti promjene programskog koda.



Slika 35: Postavke okidača za automatsku izgradnju i testiranje

Sve postavke do sada prikazane u obliku slika, spremljene su u datoteku `codemagic.yaml`. Sadržaj datoteke nalazi se dolje, a dostupna je i u GitHub repozitoriju priloženom uz ovaj rad.

Isječak kôda 25: Cjevovod za automatizaciju testiranja, izgradnje i isporuke

```
workflows:
  default-workflow:
    name: Default Workflow
    max_build_duration: 60
    environment:
      vars:
        ANDROID_FIREBASE_JSON: |-
```

```
          ewogICJwcm9qZWN0X2luZm8iOiB7CiAgICAicHJvamVjdF9udW1iZXIiOiAiMzA5
          NTE2OTYzNDg0IiwKICAgICJwcm9qZWN0X2lkIjogImNpbm5hbW9uYXBwLTczYTY0
          IiwKICAgICJzdG9yYWdlX2J1Y2tldCI6ICJjaW5uYW1vbmFwcC03M2E2NC5hcHBz
          cG90LmNvbSIKICB9LAogICJjbGllbnQiOiBbCiAgICB7CiAgICAgICJjbGllbnRf
          aW5mbyI6IHsKICAgICAgICAibW9iaWxlc2RrX2FwcF9pZCI6ICIxOjMwOTUxNjk2
          MzQ4NDphbmRyb2lkOjAwYjIxYTc1YzNmZDlhNjAzZWE3NTkiLAogICAgICAgICJh
          bmRyb2lkX2NsaWVudF9pbmZvIjogewogICAgICAgICAgInBhY2thZ2VfbmFtZSI6
          ICJhZ2VuY3kuY2lubmFtb24uZmx1dHRlckFjYWRlbXkiCiAgICAgICAgfQogICAg
          ICB9LAogICAgICAib2F1dGhfY2xpZW50IjogWwogICAgICAgIHsKICAgICAgICAg
          ICJjbGllbnRfaWQiOiAiMzA5NTE2OTYzNDg0LXF2dDhva2kyOWxnNDMyNWxncXQ0
          dm9vdHRkOTZ2aXE4LmFwcHMuZ29vZ2xldXNlcmNvbnRlbnQuY29tIiwKICAgICAg
          ICAgICJjbGllbnRfdHlwZSI6IDMKICAgICAgICB9CiAgICAgIF0sCiAgICAgICJh
          cGlfa2V5IjogWwogICAgICAgIHsKICAgICAgICAgICJjdXJyZW50X2tleSI6ICJB
          SXphU3lDYXdFRG9HTFVROFFpVFVLdXBMb0sycXRja1I5c2VhYU0iCiAgICAgICAg
          fQogICAgICBdLAogICAgICAic2VydmljZXMiOiB7CiAgICAgICAgImFwcGludml0
          ZV9zZXJ2aWNlIjogewogICAgICAgICAgIm90aGVyX3BsYXRmb3JtX29hdXRoX2Ns
          aWVudCI6IFsKICAgICAgICAgICAgewogICAgICAgICAgICAgICJjbGllbnRfaWQi
          OiAiMzA5NTE2OTYzNDg0LXF2dDhva2kyOWxnNDMyNWxncXQ0dm9vdHRkOTZ2aXE4
          LmFwcHMuZ29vZ2xldXNlcmNvbnRlbnQuY29tIiwKICAgICAgICAgICAgICAiY2xp
          ZW50X3R5cGUiOiAzCiAgICAgICAgICAgIH0KICAgICAgICAgIF0KICAgICAgICB9
          CiAgICAgIH0KICAgIH0KICBdLAogICJjb25maWd1cmF0aW9uX3ZlcnNpb24iOiAi
          MSIKfQ==
      flutter: stable
      xcode: latest
      cocoapods: default
    triggering:
      events:
        - push
      branch_patterns:
        - pattern: '*'
          include: true
          source: true
      tag_patterns:
        - pattern: '*'
          include: true
    scripts:
      - flutter emulators --launch emulator
      - |
        # set up debug keystore
        rm -f ~/.android/debug.keystore
        keytool -genkeypair \
          -alias androiddebugkey \
          -keypass android \
          -keystore ~/.android/debug.keystore \
          -storepass android \
          -dname 'CN=Android Debug,O=Android,C=US' \
          -keyalg 'RSA' \
          -keysize 2048 \
          -validity 10000
      - |
        # set up local properties
        echo "flutter.sdk=$HOME/programs/flutter" >
"$FCI_BUILD_DIR/android/local.properties"
      - flutter packages pub get
      - |
        #!/bin/sh
        echo $ANDROID_FIREBASE_JSON | base64 --decode >
$CM_BUILD_DIR/android/app/google-services.json
      - flutter test integration_test
      - flutter test
      - |
```

```
        #!/bin/sh
        echo $ANDROID_FIREBASE_JSON | base64 --decode >
$CM_BUILD_DIR/android/app/google-services.json
      - flutter build apk --debug
    artifacts:
      - build/**/outputs/apk/**/*.apk
      - build/**/outputs/bundle/**/*.aab
      - build/**/outputs/**/mapping.txt
      - '*.snap'
      - build/windows/**/*.msix
      - flutter_drive.log
    publishing:
      email:
        recipients:
          - ipapac@foi.hr
```

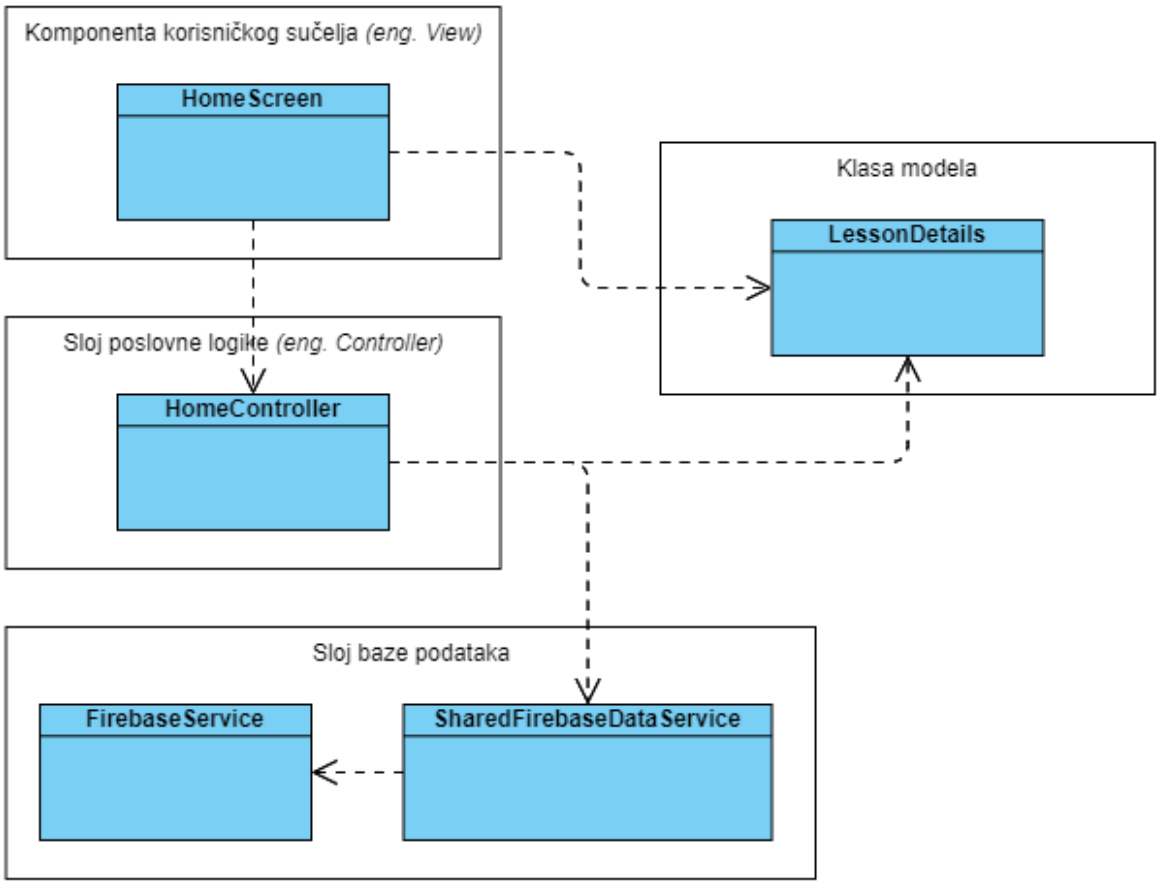
U skripti gore je moguće vidjeti prethodno definirane postavke kroz grafičko sučelje alata. Pod atributom `environment` definirane su varijable okoline i verzije platforma. Atribut `scripts` sadrži sve podešene skripte, a to su u ovom slučaju skripta za pokretanje Android virtualnog uređaja, skripta za pokretanje testova, skripta za dešifriranje varijable okoline `ANDROID_FIREBASE_JSON` i skripta za pokretanje izgradnje aplikacije.

6.2.2. Arhitektura programskog proizvoda

Nakon postavljanja alata za razvoj programskog proizvoda i implementaciju kontinuirane integracije, slijedi programiranje. U procesu razvoja programskog proizvoda potrebno je osigurati jednostavno održavanje i čitljivost programskog koda, a jedan od razloga je pisanje testova. U poglavlju 5 već je objašnjeno da se sav složeni programski kod mora refaktorirati u jednostavnije metode, a to je moguće postići implementacijom arhitektonskog uzorka dizajna. Arhitektonski uzorci dizajna u ovom radu spomenuti su u kontekstu upravljanja stanjem u Flutter aplikacijama u poglavlju 3.3. U praktičnom radu korišten je MVC uzorak dizajna za odvajanje poslovne logike od korisničkog sučelja, dok je za upravljanje stanjem korišten paket treće strane Get. Korišteni paket dodan je u `pubspec.yaml` datoteku kako bi se mogao koristiti u projektu. Pomoću ovog paketa kreirana su sučelja s pripadajućim kontrolerima i klasama modela te je posebno kreirana klasa koja komunicira s bazom podataka. Osim paradigme MVC uzorka dizajna, ovaj paket nudi upravljanje navigacijom i usmjeravanjem te mogućnosti lokalizacije. U nastavku je opisana implementacija spomenutog MVC uzorka dizajna, odnosno upotreba paketa za upravljanje stanjem Get na primjeru početnog zaslona (Slika 20).

Na slici dolje moguće je vidjeti sve komponente MVC uzorka dizajna na primjeru početnog zaslona. Klasa `HomeScreen` predstavlja komponentu korisničkog sučelja, dok klasa `HomeController` predstavlja kontroler u kojem se nalazi sva poslovna logika vezana uz početni zaslon. Kontroler komunicira s klasama `FirebaseService` i

`SharedFirebaseDataService`, koje predstavljaju sloj baze podataka. Primijetimo da je i komunikacija s bazom podataka odvojena od sloja poslovne logike, odnosno kreirane su posebne klase zadužene samo za komunikaciju s bazom podataka i one kontrolerima prosljeđuju dohvaćene podatke. Klasa `LessonDetails` predstavlja komponentu modela u MVC uzorku dizajna.



Slika 36: Dijagram klasa koje sudjeluju u MVC uzorku dizajna

U primjeru Isječak kôda 26 prikazan je dio implementacija početnog zaslona u kojem se koriste podaci o radionici obrađeni u kontroleru. Dakle, komponenta korisničkog sučelja komunicira s pripadajućim kontrolerom s ciljem prikazivanja odgovarajućih podataka na zaslon.

Isječak kôda 26: Implementacija dijela početnog zaslona

```
Column(
  crossAxisAlignment: CrossAxisAlignment.stretch,
  children: [
    /// text Upcoming
    Align(
      alignment: Alignment.topLeft,
```

```
      child: Text(
        FAStrings.homeUpcoming,
        style: FATextStyles.headline,
      ),
    ),

    /// lesson description
    Obx(
      () => UpcomingLesson(
        lessonName: homeController.upcomingLesson.lessonName,
        lessonTime:
dashboardController.writeLessonDate(homeController.upcomingLesson),
      ),
    ),

    /// text lesson name
    Obx(
      () => Text(
        '${homeController.upcomingLesson.lessonName}
${FAStrings.homeLessonNameSchedule}',
        style: FATextStyles.headline,
      ),
    ),

    /// blank space
    SizedBox(
      height: 8.h,
    ),

    /// calendar
    Obx(
      () => Expanded(
        child: Container(
          color: FCColors.white,
          child: CustomPaint(
            painter: SchedulePainter(
              lessonLecture: homeController.upcomingLecture,
              lessonCodeLab: homeController.upcomingCodeLab,
            ),),),),),],)
```

Kontroler navedene podatke o radionici dohvaća tako da poziva metodu `getAllLessons` klase `SharedFirebaseDataService`. Implementacija metode nalazi se u primjeru Isječak kôda 27. Primijetimo da ova metoda komunicira s klasom `FirebaseService`, odnosno poziva generičku metodu za dohvaćanje svih dokumenata iz odabrane kolekcije.

Isječak kôda 27: Metoda za dohvaćanje svih radionica

```
Future<void> getAllLessons() async {
  final snapshot = await firebaseService.getDocuments(
      collectionPath: FCFirestoreCollections.lessonsCollection);

  final lessonsDetails = snapshot?.docs.map((doc) =>
LessonDetails.fromJson(doc.data())).toList();

  if (lessonsDetails != null) {
    lessons.clear();
```

```
    for (final lesson in lessonsDetails) {
      lessons.add(Lesson(
          lessonName: lesson.lessonName,
          lessonStart: lesson.lectureStart,
          lessonEnd: lesson.codeLabEnd,
          lessonDetails: lesson));
    }
  }
}
```

Metoda iz primjera gore također pretvara listu JSON objekata u listu objekata klase `LessonDetails` pomoću metode `LessonDetails.fromJson`. U primjeru Isječak kôda 28 moguće je vidjeti koji podaci o radionici se dohvaćaju iz baze podataka. Primijetimo da je klasa anotirana s anotacijom `@freezed` istoimenog paketa. Taj paket omogućava automatizirano kreiranje metoda za pretvorbu JSON objekata u objekte anotirane klase.

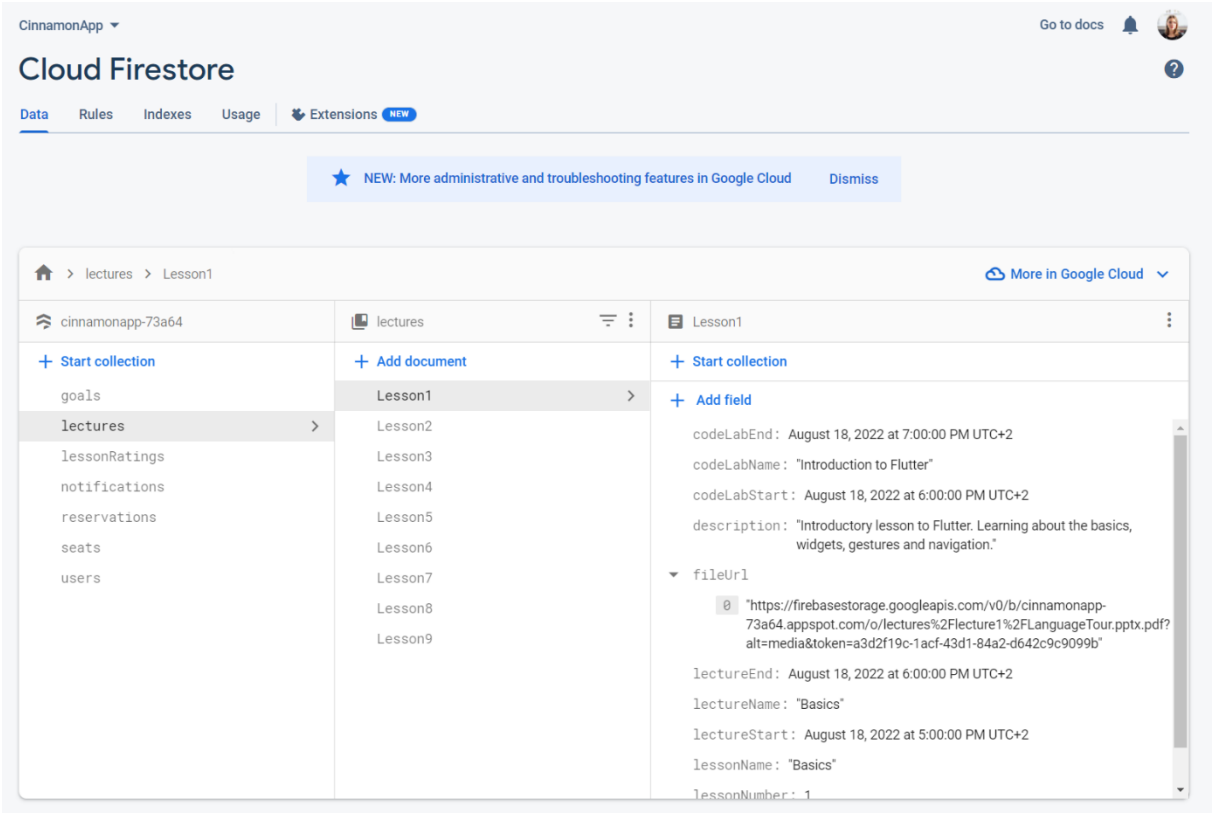
Isječak kôda 28: Implementacija klase modela

```
DateTime timestampToDate(Timestamp timestamp) => timestamp.toDate();

Timestamp dateToTimestamp(DateTime dateTime) =>
Timestamp.fromDate(dateTime);

@freezed
class LessonDetails extends Equatable with _$LessonDetails {
  const LessonDetails._();

  factory LessonDetails({
      required String lessonName,
      required String description,
      required String location,
      required String lectureName,
      @JsonKey(fromJson: timestampToDate, toJson: dateToTimestamp) required
DateTime lectureStart,
      @JsonKey(fromJson: timestampToDate, toJson: dateToTimestamp) required
DateTime lectureEnd,
      required String codeLabName,
      @JsonKey(fromJson: timestampToDate, toJson: dateToTimestamp) required
DateTime codeLabStart,
      @JsonKey(fromJson: timestampToDate, toJson: dateToTimestamp) required
DateTime codeLabEnd,
      required List<String> fileUrl,
      required int lessonNumber}) = _LessonDetails;

  factory LessonDetails.fromJson(Map<String, dynamic> json) =>
_$LessonDetailsFromJson(json);

  factory LessonDetails.blank() => LessonDetails(...);

  @override
  List<Object?> get props => [...];

  @override
  bool get stringify => true;
}
```

Ovim trima primjerima opisana je priča o MVC uzorku dizajna kojim se postiže odvajanje sloja korisničkog sučelja od sloja poslovne logike i baze podataka. Opisane prakse primijenjene su na sve implementirane funkcionalnosti programskog proizvoda. Implementacija arhitektonskog uzorka dizajna i korištenje paketa za upravljane stanjem pokazalo se korisnim tijekom cijelog procesa razvoja, a posebice tijekom testiranja programskog proizvoda.

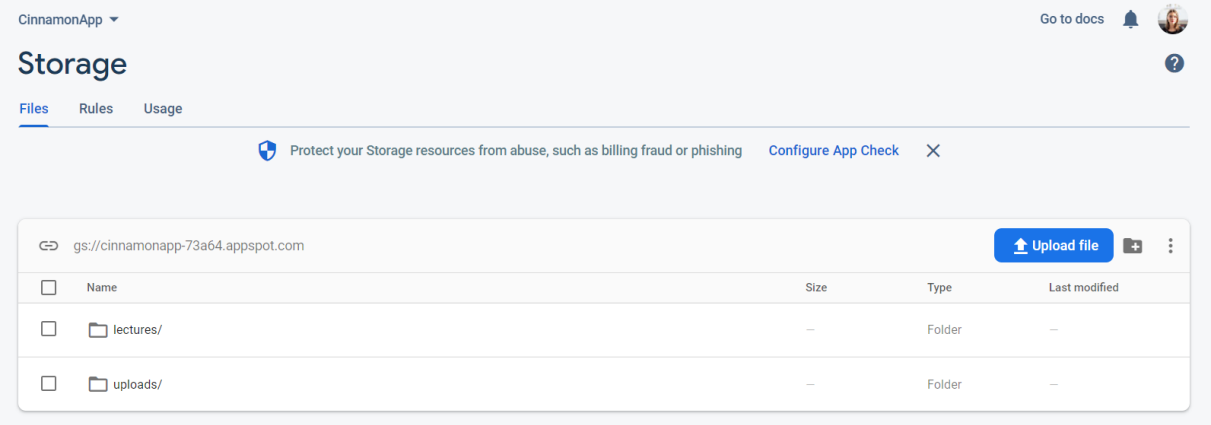
6.2.3. Poslužiteljska strana

U sklopu ovog praktičnog rada kreiran je i projekt na platformi Firebase, a predstavlja poslužiteljsku stranu programskog proizvoda. Firebase je platforma koja nudi korisnicima da izgrade cjelokupni stražnji *(eng. backend)* dio mobilnih aplikacija pomoću mnogobrojnih alata. Prvi takav alat je nerelacijska, tzv. NoSQL baza podataka Cloud Firestore. U takvim bazama podataka podaci se spremaju u obliku kolekcija i dokumenata, što je moguće poistovjetiti s relacijama i slogovima u relacijskim bazama podataka [42]. Na slici dolje moguće je vidjeti bazu podataka za aplikaciju CinnamonApp te popis svih kolekcija unutar nje.



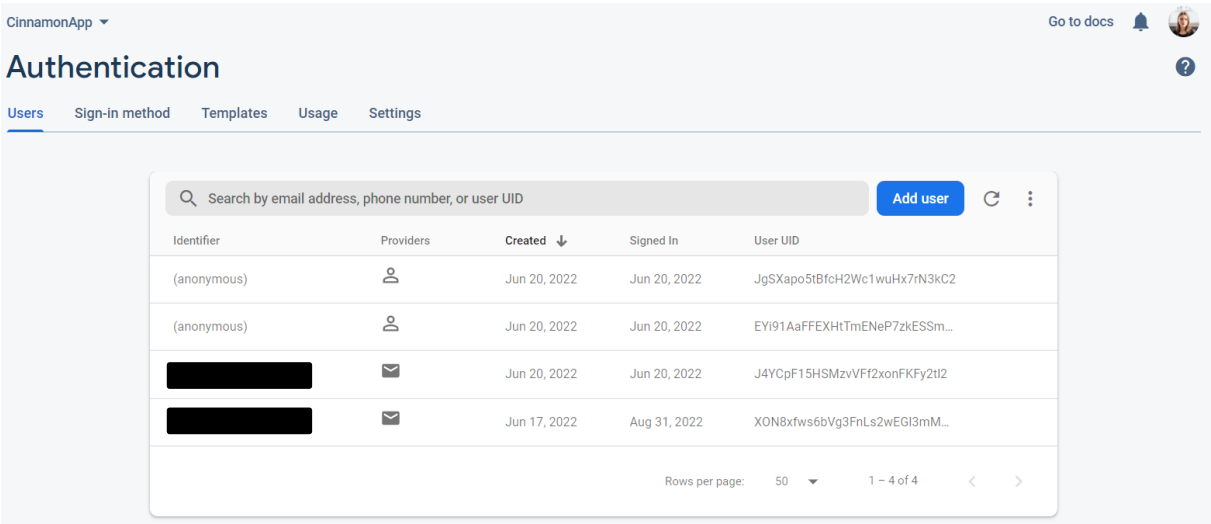
Slika 37: Baza podataka – Cloud Firestore

Za pohranu većih datoteka, poput fotografija ili videozapisa, korišten je alat Cloud Storage [42]. U ovom projektu korišten je za pohranu fotografija korisnika i PDF dokumenata priloženih uz radionice. Na slici dolje moguće je vidjeti dva direktorija, *lectures* i *uploads*. Prvi direktorij sadrži PDF dokumente, dok drugi sadrži fotografije korisnika.



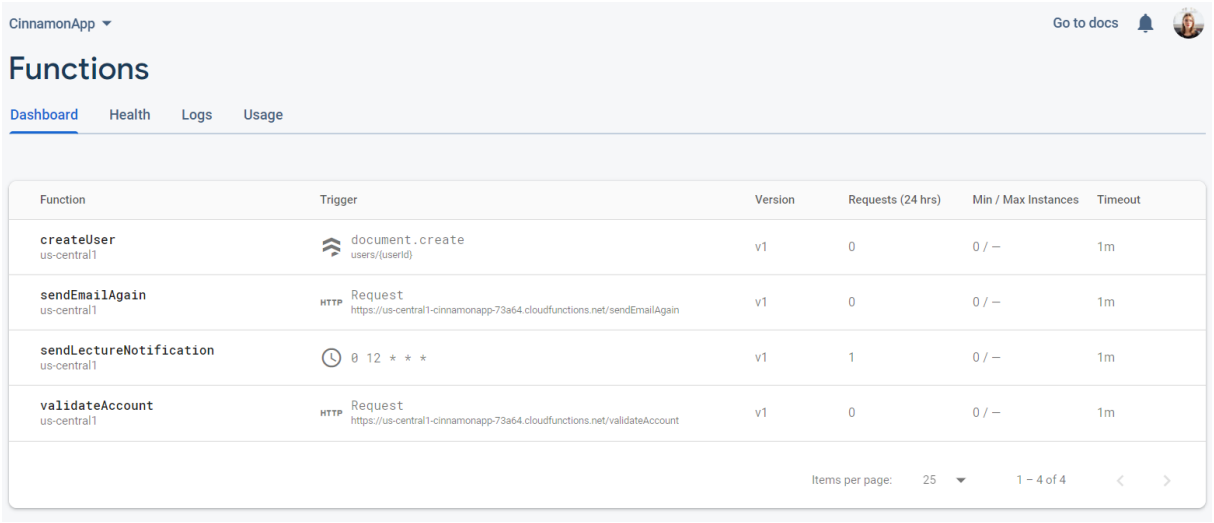
Slika 38: Alat za pohranu datoteka – Cloud Storage

U poglavlju 6.1 opisane su funkcionalnosti registracije i prijave korisnika, a alat Firebase Authentication služi za pohranu i upravljanje registriranim korisnicima. Ovaj alat podržava registraciju i prijavu pomoću adrese e-pošte i lozinke, broja mobitela te Facebook, Google ili Twitter korisničkog računa [42]. Na slici dolje moguće je vidjeti izgled kontrolne ploče alata s popisom registriranih korisnika.



Slika 39: Alat za registraciju korisnika – Firebase Authentication

Posljednji korišteni Firebase alat je Cloud Functions, a radi se o okviru bez poslužitelja. Programski kod sprema se i izvršava na Googleovim serverima. Oni automatski pokreću definirane JavaScript ili Typescript funkcije na jedan od okidača, a obično je riječ o događajima uzrokovanima promjenama u bazi podataka Cloud Firestore. Na slici dolje moguće je vidjeti tri funkcije koje se koriste u procesu registracije korisnika i jedna za slanje obavijesti o radionicama. Druga i četvrta funkcija se izvršavaju kada klijent pošalje HTTP zahtjev, prva se izvršava prilikom kreiranja novih dokumenata u bazi podataka, a treća se izvršava svaki dan u 12 sati. Sve funkcije definirane su u datoteci `index.js` u direktoriju *functions*, a moguće ih je pronaći u GitHub repozitoriju priloženim uz rad.



Slika 40: Alat za izradu funkcija – Cloud Functions

U primjeru Isječak kôda 29 nalazi se JavaScript funkcija za potvrdu korisničkog računa, odnosno ažuriranja vrijednosti atributa `isValid`. Svaka funkcija se definira pomoću ključne riječi `exports`, a u nastavku se piše okidač koji pokreće funkciju. U ovom slučaju radi se o HTTP zahtjevu, dakle funkcija će se izvršiti kada klijent pošalje zahtjev na odgovarajuću poveznicu. Tu poveznicu moguće je pronaći ispod naziva funkcije na slici gore. Funkcija vraća odgovor u obliku JSON objekta.

Isječak kôda 29: Funkcija za potvrdu korisničkog računa

```
exports.validateAccount = functions.https.onRequest(async (req, res) => {
    const userId = req.query.userId || 'Unknown';

    try {
        let result = await
admin.firestore().collection('users').doc(userId).update({
            isValid: true
        });

        res.status(200).json({ message: 'You successfully verified your
```

```
account!' });
    } catch (error) {
        console.log(error);
        res.status(404).json({ error: error });
    }
});
```

Svi opisani alati neophodni su za ispravno funkcioniranje programskog proizvoda, a iz priloženog možemo zaključiti da je cjelokupna pozadinska strana razvijena pomoću jedne platforme, tzv. „poslužitelj kao usluga“ *(eng. Backend as a Service, BaaS)*.

6.3. Testiranje programskog proizvoda

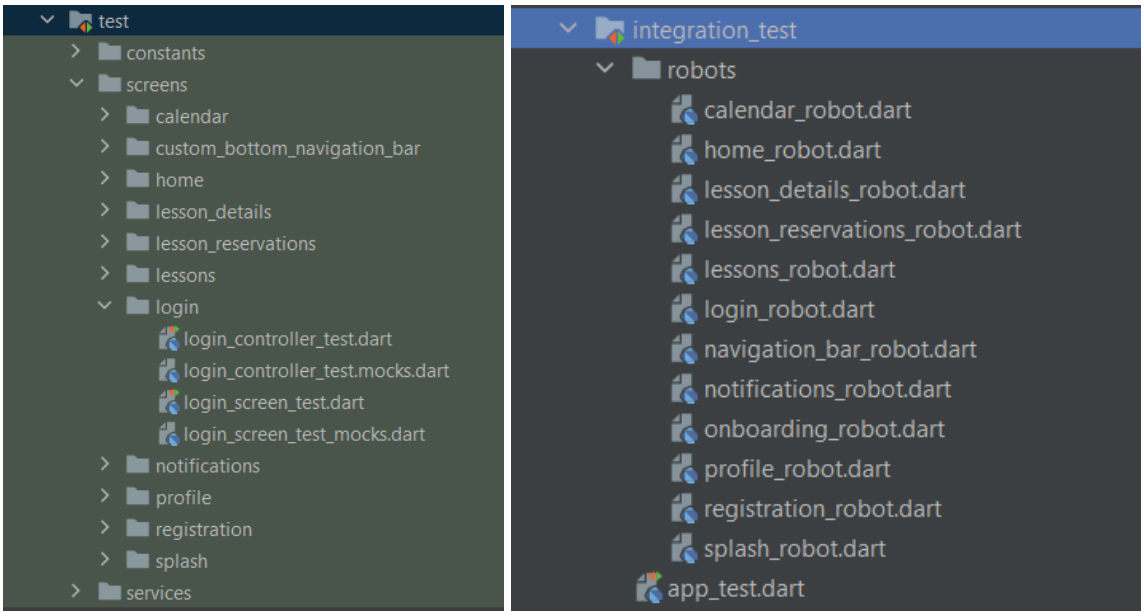
U nastavku poglavlja prikazani su i opisani dijelovi izvornog programskog koda. Uz svaki dio prikazan je i opisan primjer implementacije jedne od metoda testiranja, stoga naslovi potpoglavlja nose naziv prema odabranoj metodi. Prije pisanja i provođenja automatiziranih testova potrebno je postaviti razvojnu okolinu. To podrazumijeva definiranje odgovarajućih paketa i direktorija te postavljanje alata za automatizirano testiranje.

6.3.1. Postavke okoline za testiranje

Postavljanje okoline za testiranje Flutter aplikacije vrlo je jednostavno jer nema potrebe za korištenjem dodatnih alata, već je sve izvedivo uz pomoć Flutter biblioteka. Testove je moguće pokrenuti u odabranom integriranom razvojnom okruženju ili putem naredbenog retka, no u ovom radu testovi se automatski pokreću pri svakom dodavanju promjena na GitHub repozitorij [11]. Dakle, postavljanje okoline za testiranje uključuje postavljanje alata Codemagic i cjevovodi za automatizirano testiranje, izgradnju i isporuku, što je opisano u poglavlju 6.2.1.3.

Preostaje uključiti Flutter biblioteke za testiranje u projekt, kreirati odgovarajuće direktorije i datoteke s testovima. Za jedinične testove potrebno je dodati Flutter biblioteku `test` u datoteku `pubspec.yaml`, a ovaj paket pruža standard u pisanju i izvršavanju testova [11]. Kako bi alat Codemagic prilikom izvršavanja skripte za automatizaciju testova prepoznao jedinične testove, potrebno je sve datoteke s testovima dodati u direktorij *test*. U alatu Codemagic potrebno je omogućiti opciju *Flutter test* i u argumente naredbe dodati *test* (Slika 33) [43]. Postavke za testiranje elemenata korisničkog sučelja iste su kao i za jedinične testove, ali potrebno je još dodati Flutter biblioteku `flutter_test`. Ovaj paket nudi klase i funkcije specijalizirane za kreiranje i testiranje elemenata korisničkog sučelja, interakciju s korisničkim sučeljem te klase za traženje elemenata korisničkog sučelja na zaslonu [11]. Na slici dolje s lijeve strane (Slika 41) moguće je vidjeti strukturu direktorija *test* te direktorij koji sadrži testove vezane uz funkcionalnost prijave u aplikaciju. Svi jedinični testovi navedene funkcionalnosti nalaze se u datoteci `login_controller_test.dart`, dok se svi testovi

elemenata zaslona za prijavu nalaze u datoteci `login_screen_test.dart`. Datoteka `login_controller_test.mocks.dart` sadrži lažirane klase kreirane uz pomoć paketa `mockito`, dok datoteka `login_screen_test_mocks.dart` sadrži prilagođenu klasu povezanog kontrolera bez korištenja paketa treće strane.



Slika 41: Struktura direktorija test i integration_test

Postupak se razlikuje za integracijske testove, a prvi korak postupka je dodati Flutter biblioteku `integration_test` u datoteku `pubspec.yaml`. Integracijski testovi nalaze se u datoteci `app_test.dart` unutar direktorija *integration_test*. Oni se također mogu pokrenuti unutar integriranog razvojnog okruženja ili putem naredbenog retka [11]. Za automatizirano pokretanje u alatu Codemagic, osim što se testovi moraju nalaziti u spomenutoj datoteci i direktoriju, potrebno je omogućiti opciju *Flutter Driver tests* i u argumente naredbe dodati *test integration_test* (Slika 33). Tako alat automatski prepoznaje integracijske testove i izvršava ih u sklopu cjevovodi za automatizirano testiranje, izgradnju i isporuku [43]. Na slici gore s desne strane (Slika 41) moguće je vidjeti strukturu direktorija *integration_test* u kojoj se nalazi već spomenuta datoteka `app_test.dart`. Primijetimo da u direktoriju postoje datoteke sa sufiksom *robot*, a dobile su naziv prema uzorku dizajna za testiranje – Robot. Taj uzorak omogućuje da kreiramo robote po zaslonu ili funkcionalnosti koji sadrže testne metode. Te metode se onda pozivaju u integriranom testu unutar datoteke `app_test.dart` i oponašaju ljudsku interakciju s aplikacijom [44]. Implementacija navedenog uzorka dizajna detaljnije je opisana u poglavlju 6.3.4.

6.3.2. Jedinično testiranje

U ovom poglavlju jedinično testiranje bit će opisano na primjeru funkcije za dohvaćanje rezerviranog stola koja se koristi u sklopu pregleda detalja radionice. Jedinični testovi vezani uz navedenu funkcionalnost nalaze se u datoteci `lesson_details_controller_test.dart`, a u primjeru Isječak kôda 30 moguće je vidjeti njenu strukturu. Svi testovi se pišu unutar metode `main`, a pojedini test unutar metode `test`. Primijetimo da su testovi grupirani prema funkcionalnosti pomoću metode `group`. Osim toga, prije i poslije svakog jediničnog testa izvršavaju se metode `setUp` i `tearDown` koje služe za inicijalizaciju i brisanje povezanih objekata.

Isječak kôda 30: Struktura datoteke s jediničnim testovima

```
void main() {
  setUp(() {...});

  tearDown(() {...});

  group('Lesson ratings', () {

    test('Get rating snapshot by user id and lecture id', () async {...});

    test('Get lesson rating - Lesson rating should be 4', () async {...});

    test('Get lesson rating - Lesson rating should be 0', () async {...});

    test('Rate lesson - Create new rating - Result should be true', ()
async {...});

    test('Rate lesson - Update rating - Result should be true', () async
{...});
  });

  group('Seat reservations for selected lesson', () {

    test("Seat reservation doesn't exist in database", () async {...});

    test('Seat reservation exists in database', () async {...});
  });
}
```

U primjeru Isječak kôda 31 moguće je vidjeti implementaciju metoda `setUp` i `tearDown` za spomenutu funkcionalnost. Primijetimo da se u metodi `setUp` inicijalizira kontroler koji se želi testirati, dok se u metodi `tearDown` briše. Osim toga, inicijaliziraju se i lažirane klase `SharedFirebaseDataService` i `FirebaseService` koje komuniciraju s bazom podataka.

Isječak kôda 31: Primjer metoda `setUp` i `tearDown` u jediničnim testovima

```
setUp(() {
  _lessonDetailsController = LessonDetailsController();
```

```
  sharedFirebaseDataService = _mockSharedFirebaseDataService;
  firebaseService = _mockFirebaseService;

  _lessonDetailsController.lesson =
MockRepository.homeInitialUpcomingLesson.lessonDetails!;
});

tearDown(() => _lessonDetailsController.dispose());
```

Slijedi pisanje testa za odabranu metodu, a implementaciju te metode moguće je vidjeti u primjeru Isječak kôda 32. Ta metoda služi za dohvaćanje stola iz baze podataka koji je rezervirao trenutno prijavljeni korisnik. Prvo se dohvaćaju sve rezervacije, pa se traži rezervacija prijavljenog korisnika. Ako rezervacija postoji, tada se rezervirani stol dohvaća iz baze podataka. Na temelju tih promjena adekvatno se ažurira i korisničko sučelje. Ova metoda dio je klase `LessonDetailsController` te se u njoj nalazi sva poslovna logika vezana uz zaslon s odabranom radionicom (Slika 21).

Isječak kôda 32: Metoda za dohvaćanje rezerviranog stola

```
Future<void> getReservedSeat() async {
  final firebaseReservations = await firebaseService.getDocuments(
      collectionPath: FCFirestoreCollections.reservationsCollection);

  if (firebaseReservations == null) {
    return;
  }

  final reservations =
      firebaseReservations.docs.map((doc) =>
Reservation.fromJson(doc.data())).toList();
  final reservation = reservations
      .where((reservation) =>
          reservation.lectureId ==

'${FCFirestoreCollections.lessonsCollection}/Lesson${lesson.lessonNumber}')
      .single;

  String? seatId;
  for (final student in reservation.students) {
    student.forEach((key, value) {
      if (key == 'userId' &&
          value ==

'${FCFirestoreCollections.usersCollection}/${firebaseService.firebaseUser.v
alue!.uid}') {
        seatId = student['seatId'];
      }
    });
  }

  if (seatId != null) {
    isSeatReserved = true;

    final firebaseSeat = await firebaseService.getDocument(docPath:
seatId!);
    reservedSeat = Seat.fromJson({'id': firebaseSeat?.id,
```

```
...?firebaseSeat?.data()!});
  }
}
```

Isječak kôda 33 prikazuje jedinične testove za prethodno opisanu metodu za dohvaćanje rezerviranog stola. Prvi test provjerava ponašanje metode ako rezervacija ne postoji u bazi podataka, dok drugi test provjerava ponašanje metode ako rezervacija postoji u bazi podataka. Prvi test je uspješan ako je varijabla `isSeatReserved` jednaka `false`, a varijabla `reservedSeat` jednaka praznom objektu klase `Seat`. Drugi test je uspješan ako je varijabla `isSeatReserved` jednaka `true`, a varijabla `reservedSeat` je jednaka objektu klase `Seat` s pozicijom 1. Provjera vrijednosti tih varijabli provodi se uz pomoć metode `expect`.

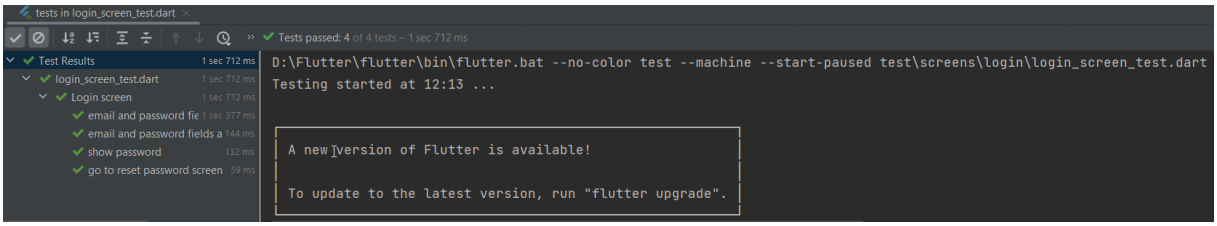
Isječak kôda 33: Jedinični testovi za odabranu metodu

```
group('Seat reservations for selected lesson', () {
  /// Arrange
  /// -- Mock snapshosts
  final QuerySnapshot<Map<String, dynamic>> querySnapshot =
MockQuerySnapshot();
  final QueryDocumentSnapshot<Map<String, dynamic>> queryDocumentSnaphsot =
      MockQueryDocumentSnapshot();

  /// -- Mock document references
  final DocumentReference lecture = MockDocumentReference();
  final DocumentReference seat = MockDocumentReference();
  final DocumentReference user = MockDocumentReference();

  /// -- Mock user
  final User mockUser = MockUser();

  /// -- Stubbing
  when(_mockFirebaseService.getDocuments(collectionPath:
anyNamed('collectionPath')))
      .thenAnswer((_) async => querySnapshot);

  when(querySnapshot.docs).thenReturn([queryDocumentSnaphsot]);

  when(queryDocumentSnaphsot.data()).thenReturn({
    'lectureId': lecture,
    'students': [
      {
        'seatId': seat,
        'userId': user,
      }
    ],
  });

  when(lecture.path).thenReturn('lectures/Lesson1');
  when(seat.path).thenReturn('seats/6hzAdnZPnUplxyvBuXgg');
  when(user.path).thenReturn('users/XON8xfws6bVg3FnLs2wEGl3mMPv1');

  test("Seat reservation doesn't exist in database", () async {
    /// -- Stubbing

when(_mockFirebaseService.firebaseUser).thenReturn(Rx<User?>(mockUser));
```

```
    when(mockUser.uid).thenReturn('1234');

    /// Act
    await _lessonDetailsController.getReservedSeat();

    /// Assert
    expect(_lessonDetailsController.isSeatReserved, false);
    expect(_lessonDetailsController.reservedSeat, Seat.blank());
  });

  test('Seat reservation exists in database', () async {
    /// -- Stubbing

when(_mockFirebaseService.firebaseUser).thenReturn(Rx<User?>(mockUser));
    when(mockUser.uid).thenReturn('XON8xfws6bVg3FnLs2wEGl3mMPv1');

    /// -- Mock seat
    final DocumentSnapshot<Map<String, dynamic>> firebaseSeat =
MockDocumentSnapshot();

    /// -- Stubbing seat
    when(_mockFirebaseService.getDocument(docPath: anyNamed('docPath')))
        .thenAnswer((_) async => firebaseSeat);

    when(firebaseSeat.id).thenReturn('6hzAdnZPnUplxyvBuXgg');
    when(firebaseSeat.data()).thenReturn({
      'name': 'A1',
      'position': 1,
    });

    /// Act
    await _lessonDetailsController.getReservedSeat();

    /// Assert
    expect(_lessonDetailsController.isSeatReserved, true);
    expect(
      _lessonDetailsController.reservedSeat,
      Seat(id: '6hzAdnZPnUplxyvBuXgg', name: 'A1', position: 1),
    );
  });
});
```

Primijetimo da je sva komunikacija sa slojem baze podataka lažirana, odnosno svi odgovori vanjskih metoda su ručno uneseni. Taj proces se još naziva *stubbing,* a podaci se ručno unose uz pomoć metoda `when` i `thenReturn`. Ako je riječ o asinkronoj metodi, tada se koristi metoda `thenAnswer`. Na temelju tih podataka i na temelju očekivanog ponašanja metode, uneseni su očekivani rezultati. Lažne klase kreirane su uz pomoć paketa `mockito` tako da je klasa u kojoj se nalaze jedinični testovi anotirana s anotacijom `@GenerateMocks`. Ova anotacija prima listu klasa koje se žele lažirati.

Ove testove moguće je pokrenuti u integriranom razvojnom okruženju, a u ovom slučaju to je Android Studio. Na slici dolje moguće je vidjeti rezultat izvršavanja jediničnih testova iz primjera te naredbu koja se izvršava prilikom pokretanja testova.



Slika 42: Rezultat izvršavanja jediničnih testova u alatu Android Studio

Nakon ovih koraka, programski kod je spreman za postavljanje na GitHub repozitorij i automatsko izvršavanje testova pomoću alata Codemagic. Testovi se automatski pokreću pri postavljanju promjena na Github repozitorij, a kako izgleda rezultat tih aktivnosti moguće je vidjeti na slici dolje.

Slika 43: Rezultat izvršavanja jediničnih testova u alatu Codemagic

Primijetimo da su svi definirani testovi unutar datoteke `lesson_details_controller_test.dart` bili uspješni. Ako su razvojnom programeru potrebni detalji izvršavanja pojedinih testova, moguće je preuzeti datoteku sa zapisima o izvršenim naredbama, testovima i eventualnim greškama.

6.3.3. Testiranje elemenata korisničkog sučelja

Za primjer testiranja elemenata korisničkog sučelja odabran je zaslon za prijavu u aplikaciju (Slika 16). Struktura testa korisničkog sučelja jednaka je strukturi jediničnih testova, osim što se testovi korisničkog sučelja pišu unutar metode `testWidgets`. Strukturu datoteke s testovima korisničkog sučelja moguće je vidjeti u primjeru Isječak kôda 34.

Isječak kôda 34: Struktura datoteke s testovima korisničkog sučelja

```
void main() {
  setUp(() {...});

  tearDown(() {...});

  group('Login screen', () {

    testWidgets('email and password fields are valid, sign in method is
called', (tester) async {...});

    testWidgets('email and password fields are not valid, sign in method is
not called',
        (tester) async {...});

    testWidgets('show password', (tester) async {...});

    testWidgets('go to reset password screen', (tester) async {...});
  });
}
```

Metode `setUp` i `tearDown` također se razlikuju od prethodnog primjera, a to je moguće vidjeti u primjeru Isječak kôda 35. U ovom slučaju prilagođena je klasa `LoginController` koja sadrži poslovnu logiku zaslona za prijavu te je inicijalizirana u metodi `setUp`. Osim toga, inicijaliziran je element korisničkog sučelja koji se želi testirati, odnosno zaslon za prijavu. Zaslon, odnosno element korisničkog sučelja, potrebno je ugnijezditi u element `MaterialApp` kako bi se testovi mogli izvršiti. Kasnije se taj element izgradi pomoću metode `pumpWidget`.

Isječak kôda 35: Primjer metoda `setUp` i `tearDown` u testovima korisničkog sučelja

```
setUp(() {
  _mockLoginController = MockLoginController();

  _loginScreen = ScreenUtilInit(
    designSize: const Size(375, 812),
    builder: (context, child) => MaterialApp(
      home: LoginScreen(
        controller: _mockLoginController,
      ),
    ),
  );
});

tearDown(() => _mockLoginController.dispose());
```

Isječak kôda 36 prikazuje implementaciju zaslona za prijavu, odnosno elemente korisničkog sučelja koji će se testirati. Primijetimo da su u programskog kodu definirana polja za unos adrese e-pošte i lozinke pomoću komponente `TextFormField`, a gumb za prijavu pomoću personalizirane komponente `YellowButton`.

Isječak kôda 36: Implementacija zaslona za prijavu

```
class LoginScreen extends StatelessWidget {
  static const routeName = '/login_screen';
  late final LoginController loginController;

  LoginScreen({LoginController? controller}) {
    loginController = controller ?? Get.find<LoginController>();
  }

  @override
  Widget build(BuildContext context) => Scaffold(
        appBar: AppBar(...),
        body: SafeArea(
          child: Padding(
            padding: EdgeInsets.symmetric(horizontal: 24.w),
            child: Obx(
              () => Column(
                children: [
                  SizedBox(height: 30.h),

                  /// E-mail TextFormField
                  Expanded(
                    child: TextFormField(
                      key: FAKeys.loginEmail,
                      onChanged: (value) => loginController
                        ..email = value
                        ..validateLoginFields(),
                      decoration: InputDecoration(
                        focusedErrorBorder:
                            const UnderlineInputBorder(borderSide:
                              BorderSide(color: FCColors.red)),
                        labelText: FAStrings.registrationValidationEmail,
                        labelStyle: FATextStyles.description,
                        floatingLabelStyle: TextStyle(height: 0.5.h),
                        errorText: !loginController.errorTextEmail
                            ? null
                            : !loginController.email.isEmail
                                ?
                                 FAStrings.registrationValidationValidEmail
                                : null,
                        errorStyle: FATextStyles.errorDescription))),

                  /// Password TextFormField
                  Expanded(
                    child: TextFormField(
                      key: FAKeys.loginPassword,
                      onChanged: (value) => loginController
                        ..password = value
                        ..validateLoginFields(),
                      obscureText: loginController.obscureText,
```

```
                    obscuringCharacter: '●',
                    decoration: InputDecoration(
                      focusedErrorBorder: const UnderlineInputBorder(
                        borderSide: BorderSide(color: FCColors.red),
                      ),
                      labelText:
                        FAStrings.registrationValidationPassword,
                      labelStyle: FATextStyles.description,
                      errorText: !loginController.errorTextPassword
                          ? null
                          : loginController.password.isEmpty
                              ?
                            FAStrings.registrationValidationRequiredField
                              : loginController.password.length < 6 ||
                      !loginController.password.contains(RegExp('[0-9]'))
                                  ?
                            FAStrings.registrationValidationShortPassword
                                  : null,
                      errorStyle: FATextStyles.errorDescription,
                      errorMaxLines: 2,
                      floatingLabelStyle: TextStyle(height: 0.5.h),
                      suffixIcon: GestureDetector(
                        key: FAKeys.loginShowPassword,
                        onTap: loginController.showPassword,
                        child: Padding(
                          padding: EdgeInsets.all(15.r),
                          child: SvgPicture.asset(FCIcons.eye))))))),

                  /// Forgot password text/link
                  GestureDetector(
                    onTap: loginController.goToPasswordReset,
                    child: Align(
                      alignment: Alignment.topRight,
                      child: Text(
                        FAStrings.loginForgotPassword,
                        key: FAKeys.loginForgotPassword,
                        style: FATextStyles.linkDescription))),

                  /// Login Button
                  Expanded(
                    flex: 3,
                    child: Padding(
                      padding: EdgeInsets.only(bottom: 30.h),
                      child: Align(
                        alignment: Alignment.bottomCenter,
                        child: YellowButton(
                          key: FAKeys.loginButton,
                          text: FAStrings.buttonLogIn,
                          enabled: loginController.validated,
                          onPressed: loginController.signIn)))),
                ],
              ))));
}
```

U primjeru Isječak kôda 37 nalazi se grupa s četiri testa zaslona za prijavu, a predstavljaju moguće scenarije korisničke interakcije na tom zaslonu. S prvim testom provjerava se poziva li se metoda za prijavu nakon što se unesu ispravni podaci za adresu e-pošte i lozinku, dok u drugom testu se unose neispravni podaci i metoda za prijavu se ne smije

pozvati. U trećem testu se provjerava da li je lozinka vidljiva nakon što korisnik pritisne ikonu za prikaz lozinke. U posljednjem, četvrtom testu provjerava se preusmjeravanje na ispravan zaslon nakon što se pritisne poveznica koja vodi do zaslona za promjenu lozinke (Slika 19). Testovi su uspješni ako vrijednosti varijabli odgovaraju očekivanim rezultatima, što se provjerava pomoću metode `expect`.

Isječak kôda 37: Testovi zaslona za prijavu

```
group('Login screen', () {
  /// Arrange
  final _loginEmail = find.byKey(FAKeys.loginEmail);
  final _loginPassword = find.byKey(FAKeys.loginPassword);
  final _loginButton = find.byKey(FAKeys.loginButton);
  final _loginShowPassword = find.byKey(FAKeys.loginShowPassword);
  final _loginForgotPassword = find.byKey(FAKeys.loginForgotPassword);

  testWidgets('email and password fields are valid, sign in method is
called', (tester) async {
    /// Act
    await tester.pumpWidget(_loginScreen); // Build widget

    await tester.tap(_loginEmail);
    await tester.enterText(_loginEmail, 'ipapac@foi.hr');
    await tester.tap(_loginPassword);
    await tester.enterText(_loginPassword, '123456789');
    await tester.pump();

    FocusManager.instance.primaryFocus?.unfocus();

    await tester.tap(_loginButton);
    await tester.pump();

    /// Assert
    expect(_mockLoginController.errorTextEmail, false);
    expect(_mockLoginController.errorTextPassword, false);
    expect(_mockLoginController.validated, true);
    expect(_mockLoginController.isSignInCalled, true);
  });

  testWidgets('email and password fields are not valid, sign in method is
not called',
      (tester) async {
    /// Act
    await tester.pumpWidget(_loginScreen);

    await tester.tap(_loginEmail);
    await tester.enterText(_loginEmail, 'ipapac');
    await tester.tap(_loginPassword);
    await tester.enterText(_loginPassword, '12345');
    await tester.pump();

    FocusManager.instance.primaryFocus?.unfocus();

    await tester.tap(_loginButton);
    await tester.pump();

    /// Assert
    expect(_mockLoginController.errorTextEmail, true);
```

```
    expect(_mockLoginController.errorTextPassword, true);
    expect(_mockLoginController.validated, false);
    expect(_mockLoginController.isSignInCalled, false);
  });

  testWidgets('show password', (tester) async {
    /// Arrange
    final isObscure = _mockLoginController.obscureText;

    /// Act
    await tester.pumpWidget(_loginScreen);
    await tester.tap(_loginPassword);
    await tester.enterText(_loginPassword, '12345');
    await tester.pump();
    await tester.tap(_loginShowPassword);

    /// Assert
    expect(_mockLoginController.obscureText, !isObscure);
  });

  testWidgets('go to reset password screen', (tester) async {
    /// Act
    await tester.pumpWidget(_loginScreen);
    await tester.tap(_loginForgotPassword);

    /// Assert
    expect(_mockLoginController.isGoToPasswordResetCalled, true);
  });
});
```

Kako bi se simulirala korisnička interakcija s elementima korisničkog sučelja, potrebno je dohvatiti te elemente pomoću njihovog ključa koji se prosljeđuje metodi `find.byKey`. Nad dohvaćenim elementima simuliraju su sljedeći pokreti – dodir tekstualnih polja, unos teksta u tekstualna polja i dodir gumba. Pokreti se simuliraju uz pomoć metoda `tap` i `enterText` koje su dio klase `WidgetTester`. Ta klasa dio je `fluttter_test` paketa i omogućuje simulaciju korisničke interakcije s korisničkim sučeljem.

Testove korisničkog sučelja također je moguće pokrenuti u integriranom razvojnom okruženju te njihovo izvršavanje i ispis rezultata jednako je kao i kod jediničnih testova. Programski kod je spreman za postavljanje na GitHub repozitorij čime se automatski pokreću testovi u alatu Codemagic. Postupak i rezultat automatskog testiranja jednak je kao i kod jediničnih testova jer ih alat tretira kao jedinične testove. Jedina razlika je da jedinični testovi provjeravaju ponašanje metoda i klasa, dok testovi korisničkog sučelja provjeravaju ponašanje elemenata korisničkog sučelja.

6.3.4. Integracijsko testiranje

Integracijsko testiranje obuhvaća provjeru ponašanja svih funkcionalnosti aplikacije te njihov međusobni odnos. Tako da je za potrebe ovog rada kreiran scenarij koji obuhvaća veći dio funkcionalnosti. Scenarij, odnosno korake integracijskog testa moguće je vidjeti u tablici dolje. S obzirom na to da scenarij sadrži veliki broj koraka, u primjerima u nastavku bit će prikazani samo dijelovi integracijskog testa. Primjeri su vezani uz funkcionalnost rezervacije stola.

Tablica 7: Scenarij za integracijsko testiranje

ID	Opis	Koraci	Očekivani rezultat
T1	Provjeri učitavanje ulaznog zaslona *(eng. splash screen)*	1. Otvori aplikaciju	Učitane su sve animacije i početni zaslon.
T2	Provjeri povlačenje slajdova na početnom zaslonu	1. Povlači ulijevo slajdove	Učitani su svi slajdovi.
T3	Provjeri unos pogrešne adrese e-pošte i lozinke	1. Unesi pogrešnu adresu e-pošte 2. Unesi pogrešnu lozinku	Prikazane su pogreške s porukama i gumb za prijavu je onemogućen.
T4	Provjeri unos nepostojećeg korisnika	1. Unesi adresu e-pošte i lozinku nepostojećeg korisnika 2. Klikni gumb za prijavu	Prikazana je pogreška s porukom o nepostojećem korisniku.
T5	Provjeri unos postojećeg korisnika s neispravnom lozinkom	1. Unesi adresu e-pošte i neispravnu lozinku 2. Klikni gumb za prijavu	Prikazana je pogreška s porukom o neispravnoj lozinki.
T6	Provjeri prijavu korisnika	1. Unesi adresu e-pošte i ispravnu lozinku 2. Klikni gumb za prijavu	Korisnik je prijavljen u aplikaciju.
T7	Provjeri zaslon s obavijestima	1. Klikni na ikonu za obavijesti 2. Pomiči zaslon gore-dolje 3. Zatvori zaslon s obavijestima	Učitane su obavijesti i označene kao pročitane.
T8	Provjeri zaslon s kalendarom	1. Klikni na ikonu za kalendar u navigacijskoj traci 2. Pomiči zaslon lijevo-desno	Učitan je zaslon s kalendarom, a pri pomicanju lijevo i desno učitavaju se odgovarajući mjeseci.
T9	Provjeri zaslon s radionicama	1. Klikni na ikonu za radionice u navigacijskoj traci 2. Pomiči zaslon gore-dolje	Učitan je zaslon s radionicama i učitane su buduće i prošle radionice.

T10	Provjeri ocjenjivanje prošle radionice	1. Klikni na prošlu radionicu 2. Pomakni zaslon gore 3. Ocijeni radionicu	Učitan je zaslon s detaljima radionice i spremljena je ocjena u bazu podataka.
T11	Provjeri učitavanje PDF datoteke	1. Klikni na karticu s PDF datotekom	Učitan je zaslon s odabranom PDF datotekom.
T12	Provjeri rezervaciju stola na budućoj radionici	1. Klikni na buduću radionicu 2. Klikni na gumb za rezervaciju 3. Odaberi stol 4. Rezerviraj stol 5. Zatvori zaslon	Učitan je zaslon s detaljima radionice i zaslon s rezervacijama; odabran je i rezerviran stol; zaslon s rezervacijama je zatvoren i ponovno je učitan zaslon s detaljima radionice.
T13	Provjeri otkazivanje rezervacije stola	1. Klikni na gumb za promjenu rezervacije 2. Klikni na gumb za otkazivanje rezervacije 3. Zatvori zaslon	Učitan je zaslon s rezervacijama i rezerviranim stolom; rezervacija je uklonjena; zatvoren je zaslon s rezervacijama i ponovno učitan zaslon s detaljima radionice.
T14	Provjeri postavke profila	1. Klikni na ikonu za profil u navigacijskoj traci 2. Uključi obavijesti 3. Uključi tamni način rada	Učitan je zaslon s korisničkim profilom i pripadajućim korisničkim podacima; uključene su obavijesti i tamni način rada.
T15	Provjeri uvjete i načine korištenja aplikacije	1. Klikni na karticu s uvjetima i korištenjima aplikacije 2. Povlači zaslon gore-dolje	Učitan je zaslon s uvjetima i korištenjima aplikacije te je učitan čitav tekst na zaslon.
T16	Provjeri promjenu imena i adrese e-pošte	1. Klikni na gumb za promjenu podataka 2. Unesi novo ime 3. Unesi novu adresu e-pošte 4. Klikni na gumb za spremanje promjena	Podaci su uspješno promijenjeni i promjene su vidljive na zaslonu profila.
T17	Provjeri promjenu ciljeva	1. Klikni na gumb za promjenu ciljeva 2. Odaberi nove ciljeve 3. Klikni na gumb za spremanje promjena	Ciljevi su uspješno promijenjeni i na zaslonu profila vidljivi su novoodabrani ciljevi.

Nakon definiranja scenarija integracijskog testiranja, slijedi njegova implementacija koja se puno ne razlikuje od implementacije jediničnih testova i testova korisničkog sučelja. Integracijski testovi pišu se unutar `main` metode u kojoj je potrebno inicijalizirati servis `IntegrationTestWidgetsFlutterBinding`. Taj servis omogućuje izvršavanje integracijskog testa na fizičkom uređaju. Pojedini testovi pišu se unutar `testWidgets` metode kao i testovi korisničkog sučelja. Strukturu datoteke integracijskog testa moguće je vidjeti u primjeru Isječak kôda 38.

Isječak kôda 38: Struktura datoteke integracijskog testa

```
void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized();

  group('Integration tests', () {
    testWidgets('First scenario - whole app', (tester) async {...});
  });
}
```

U poglavlju 6.3.1 spomenuto je da je za implementaciju integracijskog testa korišten uzorak dizajna Robot pri čemu je za svaku funkcionalnost kreirana klasa s testnim metodama (Slika 41). Te metode nazvane su prema aktivnostima koje se dešavaju prilikom korištenja aplikacije kako bi testovi bili razumljivi svim sudionicima razvojnog procesa. Testne metode vezane uz funkcionalnost rezervacije stola vidljive su u primjeru Isječak kôda 39, a iz njihovih naziva odmah se može zaključiti o čemu se radi. Na primjer, naziv metode `selectSeat` implicira da je riječ o testiranju odabira stola na zaslonu.

Isječak kôda 39: Integracijski test – rezervacija stola

```
testWidgets('First scenario - whole app', (tester) async {
  /*...*/

  /// --- go to lesson (upcoming) details screen, click reserve a seat
  await lessonsRobot.scrollPageVertically(direction: AxisDirection.down);
  await lessonsRobot.clickOnUpcomingLesson();
  await lessonDetailsRobot.findLessonDetails();
  await lessonDetailsRobot.clickReserveButton();

  /// --- find seat & reserve button, make a reservation, close screen
  await lessonReservationsRobot.findSeats();
  await lessonReservationsRobot.findSeatLegend();
  await lessonReservationsRobot.findReserveButton();
  await lessonReservationsRobot.selectSeat();
  await lessonReservationsRobot.reserveSeat();
  await lessonReservationsRobot.closeScreen();

  /// --- find reserved seat, change reservation
  await lessonDetailsRobot.findReservedSeat();
  await lessonDetailsRobot.changeReservation();

  /// --- cancel reservation, close screen
```

```
  await lessonReservationsRobot.cancelReservations();
  await lessonReservationsRobot.closeScreen();

  /// --- find reserve button, go back
  await lessonDetailsRobot.findReserveButton();
  await lessonDetailsRobot.goBack();

  /*...*/
});
```

Testne metode iz prethodnog primjera implementirane su u klasi `LessonReservationsRobot`, a dio implementacije moguće je vidjeti u primjeru Isječak kôda 40. Primijetimo da klasa putem konstruktora prima objekt klase `WidgetTester` koja je ranije spomenuta prilikom testiranja korisničkog sučelja u poglavlju 6.3.3. Pomoću članova i metoda te klase simulira se korisnička interakcija s aplikacijom i traže se odgovarajući elementi na korisničkom sučelju.

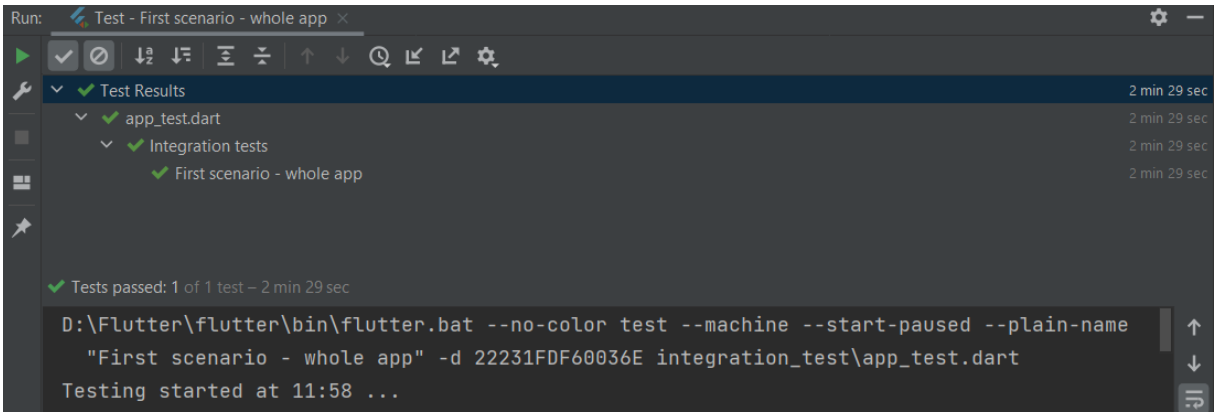
Isječak kôda 40: Klasa `LessonReservationsRobot` s implementiranim testnim metodama

```
class LessonReservationsRobot {
  final WidgetTester _tester;

  LessonReservationsRobot(this._tester);

  Future<void> findSeats() async {...}

  Future<void> findSeatLegend() async {...}

  Future<void> findReserveButton() async {...}

  Future<void> selectSeat() async {
    final seat = find.byType(SeatCard).at(0);

    await _tester.tap(seat);
    await _tester.pump(const Duration(milliseconds: 500));

    final seatWidget = _tester.widget<SeatCard>(seat);
    final seatName = seatWidget.reservation.seat.name;
    final selectedSeat =
find.textContaining('${FAStrings.lessonsSelectedSeat}\n$seatName');

    expect(seatWidget.isSelected, true);
    expect(selectedSeat, findsOneWidget);
  }

  Future<void> reserveSeat() async {
    final seat = find.byType(SeatCard).at(0);
    final reserveButton = find.widgetWithText(WhiteButton,
FAStrings.buttonReserve);
    final cancelButton = find.widgetWithText(OutlinedGrayButton,
FAStrings.buttonCancel);

    await _tester.tap(reserveButton);
    await _tester.pump(const Duration(seconds: 1));

    final seatWidget = _tester.widget<SeatCard>(seat);
```

```
    expect(reserveButton, findsNothing);
    expect(cancelButton, findsOneWidget);
    expect(seatWidget.isReserved, true);
  }

  Future<void> closeScreen() async {...}

  Future<void> cancelReservations() async {...}
}
```

Primijetimo da u integracijskom testu nisu korištene lažirane klase, već testne metode isključivo komuniciraju sa sustavom pomoću simuliranja pokreta korisnika. Nema direktnih poziva klasama sloja poslovne logike, već se oni izvršavaju na odgovarajući događaj, na primjer pomicanje ekrana ili dodir na gumb. Većina metoda tih klasa izvršava se asinkrono, odnosno rezultat tih metoda potrebno je čekati određeno vrijeme. Asinkrono izvođenje predstavlja problem prilikom izvršavanja bilo kojih testova jer se izvode u okolini koja simulira protek vremena i rezultat tih metoda se ne čeka. Takva okolina uzrokuje da testovi padaju jer većina aplikacija dohvaća podatke iz baze podataka. Rješenje je u metodama `pump` i `pumpAndSettle` koje su dio klase `WidgetTester`, a one omogućuju da se odgovarajući elementi korisničkog sučelja ponovno izgrade nakon određenom vremena. Drugim riječima, čeka se određeno vrijeme da se izvrše asinkrone metode i da se rezultati tih metoda učitaju na zaslon [45]. U primjeru gore vidimo da se nakon dodira na gumb za rezervaciju čeka jedna sekunda jer je toliko otprilike potrebno da se podaci u bazi podataka ažuriraju i prikažu na zaslonu.

Integracijski test spreman je za izvršavanje u integriranom razvojnom okruženju. Uvjet je da je povezan barem jedan fizički ili virtualni uređaj nakon čega je moguće pokrenuti integracijski test. Na slici dolje moguće je vidjeti rezultat integracijskog testa te naredbu kojom ga je moguće pokrenuti u naredbenom retku.



Slika 44: Rezultat izvršavanja integracijskog testa u alatu Android Studio

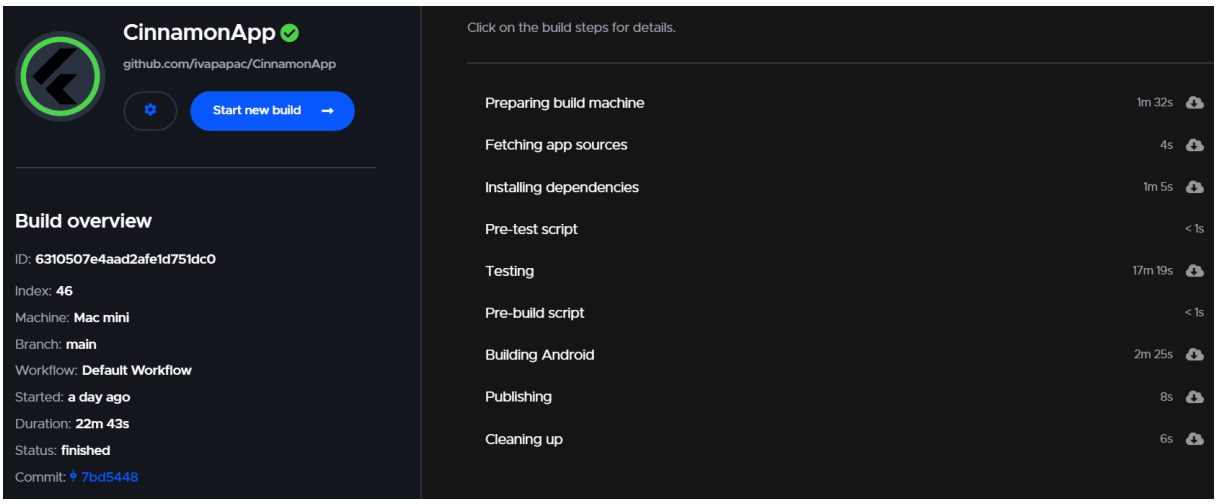
Posljednji korak je postavljanje novog programskog koda na GitHub repozitorij što će aktivirati skriptu za automatizirano testiranje i izgradnju u alatu Codemagic. U poglavlju 6.3.1 navedeno je da alat automatski detektira integracijske testove ako se nalaze u direktoriju *integration_test*. Na slici dolje moguće je vidjeti ispis rezultata integracijskog testa u alatu Codemagic. Primijetimo da se ne razlikuje od ispisa rezultata jediničnih testova, a isto tako je moguće preuzeti datoteku sa zapisima o izvršenim naredbama i eventualnim greškama.

Slika 45: Rezultat izvršavanja integracijskog testa u alatu Codemagic

Sve metode testiranja bile su uspješne prilikom izvršavanja skripte za automatizaciju testiranja i izgradnje, a nakon toga slijedi automatsko pokretanje izgradnje mobilne aplikacije. Cjelokupni rezultat automatiziranog izvršavanja jediničnih testova, testova korisničkog sučelja i integracijskog testa te automatizirane izgradnje analiziran je u sljedećem poglavlju.

6.4. Analiza rezultata

Alat Codemagic u ovom razvojnom procesu omogućio je da se sa svakom promjenom programskog koda pokrene automatizirani proces testiranja, izgradnje i isporuke programskog proizvoda. Najviše je ubrzan proces testiranja s obzirom na to da se u integriranom razvojnom okruženju svaki od testova mora pokrenuti ručno. Na slici dolje moguće je vidjeti koje sve aktivnosti je alat automatizirao i izvršio te koliko je svaka od aktivnosti trajala.



Slika 46: Rezultat izvršavanja skripte za automatizirano testiranje i izgradnju

Prvi korak je pripremanje okoline za cjelokupni proces što uključuje pripremanje alata, dohvaćanje izvornog koda s GitHub repozitorija te instalaciju paketa koji su definirani u datoteci `codemagic.yaml`. Nakon toga kreće testiranje programskog proizvoda koje je prikazano u prethodnim poglavljima. Kako bi se testovi uopće mogli pokrenuti, Codemagic prvo pokreće izvršavanje skripte za dešifriranje datoteke `google-services.json`. Ako su testovi uspješni, automatski se pokreće izgradnja aplikacije i isporuka artefakata. Izgradnja se izvršava pomoću naredbe `flutter build apk -debug`, a nakon uspješne izgradnje moguće je preuzeti artefakte. Popis svih artefakata moguće je pronaći u datoteci `publishing.log`, a sadržaj te datoteke moguće je vidjeti dolje.

Isječak kôda 41: Sadržaj datoteke `publishing.log`

```
== Gathering artifacts ==

== Publishing artifacts ==

Publishing artifact flutter_drive.log
Publishing artifact app-debug.apk
Publishing artifact flutter_plugin_android_lifecycle-debug.aar
Publishing artifact integration_test-debug.aar
Publishing artifact sensors-debug.aar
```

```
Publishing artifact firebase_auth-debug.aar
Publishing artifact wakelock-debug.aar
Publishing artifact package_info_plus-debug.aar
Publishing artifact syncfusion_flutter_pdfviewer-debug.aar
Publishing artifact location-debug.aar
Publishing artifact cloud_firestore-debug.aar
Publishing artifact sensors_plus-debug.aar
Publishing artifact url_launcher_android-debug.aar
Publishing artifact firebase_messaging-debug.aar
Publishing artifact connectivity_plus-debug.aar
Publishing artifact firebase_core-debug.aar
Publishing artifact better_player-debug.aar
Publishing artifact sqflite-debug.aar
Publishing artifact flutter_local_notifications-debug.aar
Publishing artifact permission_handler-debug.aar
Publishing artifact image_picker_android-debug.aar
Publishing artifact firebase_storage-debug.aar
Publishing artifact path_provider_android-debug.aar
Publishing artifact device_info_plus-debug.aar
Publishing artifact open_file-debug.aar
Publishing artifact audioplayers-debug.aar
Publishing artifact share-debug.aar
```

Primijetimo da se radi o dvadesetak datoteka i većina njih sadrži sufiks .aar koji označava Android arhivu, odnosno vrstu datoteke koje pohranjuju resurse za Android aplikaciju. U datoteci `app-debug.apk` nalazi se izgrađena aplikacija spremna za instalaciju na Android uređaje. Naravno, ovdje razvojnom procesu u kontekstu DevOps-a ne dolazi kraj, već svakom novom nadogradnjom programskog koda ciklus razvoja, testiranja, izgradnje i isporuke se ponavlja.

7. Zaključak

Ovaj diplomski rad spoj je relativno novih tehnologija s procesom razvoja programskog proizvoda. Riječ je o programskom okviru Flutter i programskom jeziku Dart, a obje tehnologije su obrađene s teorijskog aspekta nakon čega je izrađena i testirana mobilna aplikacija za Android platformu. Pri razvoju i testiranju spomenute mobilne aplikacije korištene su prakse DevOps-a, a cilj je bio primijeniti kontinuiranu integraciju i isporuku uključujući automatizaciju procesa izgradnje, testiranja i isporuke programskog proizvoda. Principi i prakse DevOps-a bili su najizazovniji dio pri izradi ovog rada s obzirom na to da ne postoji literatura s recepturom primjene DevOps praksi, već način primjene ovisi o pojedinim osobama, timovima ili organizacijama.

Flutter je programski okvir za višeplatformski razvoj pomoću kojeg je moguće vrlo brzo kreirati minimalno vitalne programske proizvode, tzv. MVP aplikacije *(eng. Minimum viable product).* Vrlo je jednostavan za naučiti, posebice razvojnim programerima s iskustvom u bilo kojem drugom programskom jeziku. Ono što se pokazalo najkorisnijim je bogata dokumentacija i velika Flutter zajednica, dok tijekom samog procesa razvoja treba istaknuti novitete poput funkcija koje imaju vlastiti tip i mogu se prosljeđivati kao parametri drugim funkcijama. Osim toga, Googleovi razvojni programeri kreirali su biblioteke za lakše pisanje i izvršavanje jediničnih i integracijskih testova te testova korisničkog sučelja. Nije bilo potrebe koristiti dodatne pakete trećih strana. Iako se danas sve češće testiranje naglašava kao važna aktivnost procesa razvoja programskog proizvoda, što pritom sve više zvuči kao floskula, zaista je vrijedna aktivnost koja štedi puno vremena. Tijekom razvoja programskog proizvoda, pisanje testova primoralo me da razmislim o kvaliteti napisanog programskog koda i naposljetku da redovito provodim refaktoriranje.

Najizazovniji dio rada bila je obrada DevOps-a s teorijskog aspekta s obzirom na to da sam se rijetko susretala s tim pojmom. Izazov je predstavljala sama činjenica je DevOps set principa i praksi koji se implementiraju kao dio procesa razvoja, izgradnje, testiranja i isporuke programskog proizvoda. Dakle, ne postoji set koraka koje je potrebno pratiti za uspješnu implementaciju spomenutih principa i praksi, već se primjenjuje ono što ja ili bilo tko drugi smatra prikladnim za određeni projekt. Bez obzira na loš početak s DevOps-om, primjena kontinuirane integracije prilikom razvoja programskog proizvoda pokazala se učinkovitom. Sve što sam morala je pripremiti skriptu za automatizaciju testiranja, izgradnje i isporuke programskog proizvoda, a odabrani alat je učinio ostalo. U manje od 30 minuta razvijeni programski proizvod je testiran i isporučen s najosnovnijim postavkama okoline.

Na kraju, mogu reći da sam zadovoljna s odabranom temom i primjenom naučenog u praktičnom dijelu rada. S lakoćom sam naučila nove tehnologije i vjerujem da ću naučene DevOps prakse primijeniti u budućim projektima.

Popis literature

[1] T. Uchida Molin, „Why is software testing so important?“, *theICEway*, 2021. [Na internetu]. Dostupno: https://www.theiceway.com/blog/why-is-software-testing-so-important [pristupano 29.03.2022.].

[2] S. Pandey, „Why Flutter is important in 2020“, *Medium*, 2020. [Na internetu]. Dostupno: https://medium.flutterdevs.com/why-flutter-is-important-in-2020-8400394a59f3 [pristupano 29.03.2022.].

[3] „Metode znanstvenih istraživanja“ (bez dat.), *Sveučilište u Zadru*. [Na internetu]. Dostupno: http://www.unizd.hr/portals/4/nastavni_mat/1_godina/metodologija/metode_znanstvenih_istrazivanja.pdf [pristupano 02.09.2022.].

[4] A. Miola, *Flutter Complete Reference: Create Beautiful, Fast and Native Apps for Any Device*. Independently Published, 2020.

[5] M. L. Napoli, *Beginning Flutter: A Hands On Guide To App Development*. Indianapolis, IN, USA: John Wiley & Sons, 2020.

[6] R. Payne, *Beginning App Development with Flutter: Create Cross-Platform Mobile Apps*. Dallas, TX, USA: Apress, 2019.

[7] „Dart službena dokumentacija“ (bez dat.), *Dart*. [Na internetu] Dostupno: https://dart.dev/guides [pristupano 17.03.2022.].

[8] B. Eisenman, *Learning React Native: Building Native Mobile Apps with JavaScript*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2016.

[9] W. Wu, *React Native vs Flutter, cross-platform mobile application frameworks* [Završni rad]. Metropolia University of Applied Sciences, Helsinki, Finska, 2018.

[10] M. Olsson, „A Comparison of Performance and Looks Between Flutter and Native Applications“, Blekinge Institute of Technology, Faculty of Computing, Department of Software Engineering, Švedska, 2020. [Na internetu]. Dostupno: https://www.diva-portal.org/smash/get/diva2:1442804/FULLTEXT01.pdf [pristupano 27.04.2022.].

[11] „Flutter službena dokumentacija“ (bez dat.), *Flutter*. [Na internetu]. Dostupno: https://docs.flutter.dev/ [pristupano 16.03.2022.].

[12] T. Asodariya, „7 things you need to know about Flutter State Management“, *Medium*, 2021. [Na internetu]. Dostupno: https://medium.com/dhiwise/7-things-you-need-to-know-about-flutter-state-management-42f840ef022e [pristupano 14.04.2022.].

[13] J. Mohite, „Flutter: MVVM Architecture“, *Medium*, 2020. [Na internetu]. Dostupno: https://medium.com/flutterworld/flutter-mvvm-architecture-f8bed2521958 [pristupano 20.07.2022.].

[14] T. Gürel, „MVC+S Design Pattern in Flutter“, *Medium*, 2022. [Na internetu]. Dostupno: https://itnext.io/mvc-s-design-pattern-in-flutter-6eba15169413 [pristupano 20.07.2022.].

[15] A. Syal, „BLoC pattern in Flutter“, *Medium*, 2019. [Na internetu]. Dostupno: https://medium.flutterdevs.com/bloc-pattern-in-flutter-part-1-flutterdevs-128f90059f5c (pristupljeno srp. 22, 2022).

[16] K. Clokie, *A Practical Guide to Testing in DevOps*. Victoria, Kanada: Leanpub, 2017.

[17] J. Davis i K. Daniels, *Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling at Scale*, 1. izd., Sebastopol, CA, USA: O'Reilly Media, Inc., 2015.

[18] L. Bass, I. Weber, i L. Zhu, *DevOps: A Software Architect's Perspective*, 1. izd., Westford, MA, USA: Addison-Wesley Professional, 2015.

[19] G. Kim, J. Humble, P. Debois, J. Willis, i N. Forsgren, *The Devops Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*, 1. izd., Portland, OR, USA: IT Revolution, 2016.

[20] P. Swartout, *Continuous Delivery and DevOps – A Quickstart Guide*, 2. izd., Birmingham, UK: Packt Publishing, 2014.

[21] C. Harris, „Microservices vs. monolithic architecture“, *Atlassian*, 2020. [Na internetu]. Dostupno: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith [pristupano 12.08.2022.].

[22] „DevOps Tools for each phase of the DevOps lifecycle“, *Atlassian*, 2022. [Na internetu]. Dostupno: https://www.atlassian.com/devops/devops-tools [pristupano 10.08.2022.].

[23] „What is Jira Software used for?“, *Atlassian*, 2022. [Na internetu]. Dostupno: https://www.atlassian.com/software/jira/guides/use-cases/what-is-jira-used-for#Jira-for-requirements-&-test-case-management [pristupano 10.08.2022.].

[24] „A brief overview of Confluence“, *Atlassian*, 2022. [Na internetu]. Dostupno: https://www.atlassian.com/software/confluence/guides/get-started/confluence-overview#about-confluence [pristupano 10.08.2022.].

[25] „Atlassian Integrations for Slack“, *Atlassian*, 2022. [Na internetu]. Dostupno: https://www.atlassian.com/partnerships/slack/integrations [pristupano 10.08.2022.].

[26] J. Campbell i C. Harris, „Kubernetes vs. Docker“, *Atlassian*, 2020. [Na internetu]. Dostupno: https://www.atlassian.com/microservices/microservices-architecture/kubernetes-vs-docker?utm_campaign=service-desk_devops16-blog [pristupano 10.08.2022.].

[27] M. Courtemanche, „What is GitHub?“, *TechTarget*, 2018. [Na internetu]. Dostupno: https://www.techtarget.com/searchitoperations/definition/GitHub [pristupano 10.08.2022.].

[28] M. Heller, „What is Jenkins? The CI server explained“, *InfoWorld*, 2020. [Na internetu]. Dostupno: https://www.infoworld.com/article/3239666/what-is-jenkins-the-ci-server-explained.html [pristupano 11.08.2022.].

[29] A. Paul, „What is Appium & How it Works? | Beginners Guide To Appium“, *Edureka*, 2020. [Na internetu]. Dostupno: https://www.edureka.co/blog/what-is-appium/ [pristupano 11.08.2022.].

[30] T. Bechtel, „What Is AppDynamics? Cisco’s APM Solution Explained“, *World Wide Technology*, 2021. [Na internetu]. Dostupno: https://www.wwt.com/article/what-is-appdynamics-ciscos-apm-solution-explained [pristupano 11.08.2022.].

[31] S. Newell, „What is Pendo?“, *ProductPlan*, 2021. [Na internetu]. Dostupno: https://www.productplan.com/glossary/pendo/ [pristupano 10.08.2022.].

[32] T. Hamilton, „END-To-END Testing Tutorial: What is E2E Testing with Example“, *Guru99*, 2022. [Na internetu]. Dostupno: https://www.guru99.com/end-to-end-testing.html [pristupano 07.06.2022.].

[33] V. Khorikov, *Unit Testing Principles, Practices, and Patterns*. Shelter Island, NY, USA: Manning Publications Co., 2020.

[34] G. J. Myers, C. Sandler, i T. Badgett, *The Art of Software Testing*. Hoboken, NJ, USA: John Wiley & Sons, 2012.

[35] T. Hamilton, „Unit Testing Tutorial: What is, Types, Tools & Test Example“, *Guru99*, 2022. [Na internetu]. Dostupno: https://www.guru99.com/unit-testing-guide.html [pristupano 02.06.2022.].

[36] T. Hamilton, „Integration Testing: What is, Types, Top Down & Bottom Up Example“, *Guru99*, 2022. [Na internetu]. Dostupno: https://www.guru99.com/integration-testing.html [pristupano 07.06.2022.].

[37] E. Freeman, *DevOps for dummies*. Hoboken, NJ, USA: John Wiley & Sons, 2019.

[38] D. Ashby, „Continuous Testing in DevOps...“, 2016. [Na internetu]. Dostupno: https://danashby.co.uk/2016/10/19/continuous-testing-in-devops/ [pristupano 14.08.2022.].

[39] J. Humble i D. Farley, *Continuous Delivery*. Crawfordsville, IN, USA: Addison-Wesley, 2010.

[40] „Android Developers Blog: Android Studio: An IDE built for Android“ (bez dat.). [Na internetu]. Dostupno: https://android-developers.googleblog.com/2013/05/android-studio-ide-built-for-android.html [pristupano 28.08.2022.].

[41] „Continuous Integration for Flutter with Codemagic“, Codemagic, *Medium*, 2019. [Na internetu]. Dostupno: https://medium.com/@codemagicio/continuous-integration-for-flutter-with-codemagic-239aa206a70 [pristupano 13.08.2022.].

[42] „Firebase službena dokumentacija “, *Google*, 2022. [Na internetu]. Dostupno: https://firebase.google.com/docs [pristupano 31.08.2022.].

[43] „Codemagic službena dokumentacija“, 2022. [Na internetu]. Dostupno: https://docs.codemagic.io/ [pristupano 01.09.2022.].

[44] A. Wangoo, „Integration Testing in Flutter“, *Medium*, 2021. [Na internetu]. Dostupno: https://medium.com/flutter-community/integration-testing-in-flutter-b25c62ec287c [pristupano 01.09.2022.].

[45] A. Bullard, „Understanding Async in Flutter Tests“, *Medium*, 2019. [Na internetu]. Dostupno: https://medium.com/flutter/understanding-async-in-flutter-tests-a304a7604b3c [pristupano 02.09.2022.].

Popis slika

Popis tablica

Popis isječaka kôda

Prilozi

1. Repozitorij aplikacije – https://github.com/ivapapac/CinnamonApp
2. Testiranje i izgradnja aplikacije – https://codemagic.io/app/62f79ab3b44a1ffc4d8afdf7/build/6310507e4aad2afe1d751dc0