Upravljanje robotskim sustavima pomoću mobilnih uređaja

Umiljanović, Terezija

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Sveučilišni studij

UPRAVLJANJE ROBOTSKIM SUSTAVIMA POMOĆU MOBILNIH UREĐAJA

Diplomski rad

Terezija Umiljanović

Osijek, 2023.

SADRŽAJ

1. UVOD

U današnjem svijetu, robotika je postala značajan i brzorastući sektor tehnologije koji se primjenjuje u različitim područjima kao što su industrija, medicina, istraživanje i mnoga druga. U sklopu robotike, posebnu važnost imaju servisni roboti koji obavljaju razne zadatke poput patroliranja i nadzora. Upravljanje tim robotskim sustavima putem mobilnih aplikacija omogućuje korisnicima jednostavan pristup robotima i njihovim funkcionalnostima, čime se omogućuje daljinsko nadgledanje, upravljanje i optimizacija rada servisnih robota. Ove aplikacije pružaju praktičnost i omogućuju korisnicima da nadziru prostor, upravljaju robotima i unaprijede radne procese, sve to putem njihovog prijenosnog uređaja.

U sklopu diplomskog rada, fokus je stavljen na razvoj programskog rješenja za upravljanje robotskim sustavom putem mobilne aplikacije. Cilj rada je implementirati funkcionalnosti koje će omogućiti korisnicima prijavu u aplikaciju, pregled dostupnih robota, spajanje s odabranim robotom te upravljanje njime. Osim toga, aplikacija će omogućiti prikaz informacija o spojenom robotu, prijenos uživo videa s kamere robota te sučelje za navigaciju i upravljanje patrolnim točkama. Uzimajući u obzir brz rast robotike i potrebu za učinkovitim upravljanjem robotskim sustavima, razvoj mobilne aplikacije za upravljanje robotima ima veliki potencijal i važnost. Kroz ovaj rad, istražit će se tehnologije i alati koji omogućuju razvoj takve aplikacije, s fokusom na Flutter radni okvir, koji je jedan od najpopularnijih okvira za razvoj višeplatformskih mobilnih aplikacija. Osim toga, bit će korištena i *Riverpod* biblioteka za upravljanje stanjem aplikacije te *Dio* biblioteka za izvršavanje mrežnih zahtjeva i komunikaciju sa serverom. Protokoli i alati za prijenos uživo bit će korišteni za prikaz videa s kamere robota u stvarnom vremenu. Kroz implementaciju funkcionalnosti i testiranje, cilj je pružiti pouzdanu i intuitivnu mobilnu aplikaciju koja će omogućiti korisnicima učinkovito i praktično upravljanje robotskim sustavom.

1.1. Zadatak diplomskog rada

U teorijskom dijelu rada potrebno je proučiti i opisati sustave i aplikacije za upravljanje robotskim sustavima i tehnologije za izradu mobilnih aplikacija za Flutter platformu. U praktičnom dijelu rada potrebno je koristeći Flutter radno okruženje izraditi mobilnu aplikaciju koja omogućuje upravljanje s robotskim sustavom.

2. UPRAVLJANJE ROBOTSKIM SUSTAVOM

Kroz povijest, napredak civilizacije uvijek je bio usmjeren prema olakšavanju svakodnevnih poslova i smanjenju rizika za ljude, posebno u opasnim ili izazovnim situacijama. U tom kontekstu, razvoj tehnologije i napredak znanosti donose inovativna rješenja koja značajno oblikuju naš život. Jedno od takvih rješenja, čija se prisutnost već itekako osjeti, a opet se može reći da je tek pred procvatom, je robotski sustav.

Robotski sustav je kompleksan skup međusobno povezanih hardverskih i softverskih komponenti koje omogućuju autonomno ili poluautonomno djelovanje robota u izvršavanju različitih zadataka i funkcija. Hardver najčešće čini robotsko tijelo, aktuatori i senzori, dok softver obuhvaća programske algoritme koji sve više uključuju i umjetnu inteligenciju (engl. *Artificial intelligence, AI*). Robotika, kao grana znanosti koja se bavi projektiranjem, konstruiranjem, upravljanjem i primjenom robota [1], ima sve veću važnost u suvremenom svijetu i postala je neizostavni dio strategije rasta i napretka svake razvijene zemlje. Robotski sustavi su već nekoliko desetljeća prisutni u industrijskoj proizvodnji, a prema dostupnim podacima Statističkog odjela Međunarodne Federacije Robotike (engl. *International Federation of Robotics, IFR*) do 2022. godine, broj industrijskih robota koji su u upotrebi ili aktivno postavljeni u različitim industrijskim sektorima iznosi oko 3 i pol milijuna jedinica [2]. Azija, uključujući Australiju i Novi Zeland, dominira globalnim tržištem industrijskih robota, pri čemu je Kina najveće tržište koje čini 52% ukupnih instalacija u 2021. godini. Japan, Sjedinjene Američke Države, Republika Koreja i Njemačka su također značajni igrači na tržištu. Elektro - elektronička industrija glavni je kupac industrijskih robota zadnje dvije godine, potisnuvši tako automobilsku industriju, koja je prvo mjesto držala od samih početaka [2]. Slika 2.1. prikazuje izgled nekolicine različitih industrijskih robota.

Sl. 2.1. Prikaz različiti industrijski roboti kompanije KUKA [4]

Osim industrijskih robota, unazad nekoliko godina značajan rast na tržištu bilježe i servisni roboti. Razlog tomu je brz napredak tehnologije, sve veća potreba za automatizacijom te povećanje svjesnosti o prednostima koje takvi roboti mogu donijeti, kao što su veća preciznost, smanjenje grešaka i poboljšanje sigurnosti. Za razliku od industrijskih robota koji imaju dobro definirane primjene, servisni roboti se koriste u širokom rasponu područja poput transporta i logistike, sigurnosti, medicine, ugostiteljstva, poljoprivrede, kućanstva, itd. Statistički odjel IFR-a do 2022. godine bilježi 1,010 proizvođača servisnih robota diljem svijeta i to isključujući prototipiranja i integratore sustava [3]. Očekuje se da će industrija servisnih robota, iako mlada i u razvoju, doživjeti sve veći porast u budućnosti. Razlog je tome što mnoge tvrtke još uvijek prolaze kroz rane faze razvoja svojih proizvoda te aktivno traže investicije i sredstva kako bi osigurale potrebno financiranje. U skupinu servisnih robota spadaju i roboti za patroliranje i skeniranje okoline radi nadzora, detekcije i sprječavanja potencijalnih sigurnosnih prijetnji.

2.1. Robotski sustavi za patroliranje i skeniranje okoline

Robotski sustavi za patroliranje i skeniranje okoline su opremljeni kamerama, senzorima za detekciju pokreta, termalnim kamerama i drugim sigurnosnim tehnologijama. Za mapiranje prostora, takvi robotski sustavi se najčešće oslanjaju na LiDAR (engl. *Light detection and Raginig*) tehnologiju jer koristi visoko precizne senzore za registraciju povratka laserskih zraka što rezultira stvaranjem vrlo detaljnih i preciznih mapa okoline. LiDAR tehnologija donosi nekoliko prednosti, uključujući sposobnost izbjegavanja geometrijske distorzije i mogućnost korištenja tijekom dana i noći [4]. Ovo zadnje je iznimno korisno jer omogućuje robotima da nesmetano snimaju i prikupljaju podatke i tijekom noćnih sati, kada najčešće nema ljudi u prostorima koji se snimaju. U svijetu postoje brojne kompanije koja se bavi razvojem robota za pružanje sigurnosti od potencijalnih prijetnji, a neke od njih su:

- *Boston Dynamics* - poznat po svojim inovativnim i visokotehnološkim robotskim rješenjima. Njihov prepoznatljiv robot *Spot*, koji se između ostalog oslanja na LiDAR tehnologiju, može se koristit za patroliranje i nadzor okoline.
- *Knightscope* – specijaliziran za razvoj autonomnih robotskih rješenja za sigurnost. Njihovi roboti, poput modela *Knightscope K5*, opremljeni su raznim senzorima i tehnologijama za nadzor i patroliranje javnih prostora.
- *Cobalt Robotics* – fokus stavlja na stvaranje robota za nadzor sigurnosti u poslovnim okruženjima.

- *SMP Robotics* – njihov autonomni robot *SMP S5* ima mogućnost samostalnog navigiranja u složenim okolinama, prepoznavanja prepreka i interakciju s korisnicima.
- *OTSAW Digital* - bavi se razvojem robotskih rješenja za sigurnost i ostale usluge. Njihov model *O-R3* namijenjen je za patroliranje i nadzor prostora.

Sl. 2.2. Servisni roboti redom: *Spot[6]*, *Knightscope K5[7], SMP S[8], O-R3[9]*

Svi navedeni modeli robota za nadzor i pružanje sigurnosti, osim što koriste najnovija tehnološka rješenja u pogledu hardvera i softvera, imaju integriranu umjetnu inteligenciju. Integracija AI-a omogućuje tim robotima učenje, planiranje, rješavanje problema, donošenje odluka i korištenje računalnog vida [10]. Pomoću AI-a, roboti mogu prepoznati, detektirati i klasificirati objekte, interpretirati senzorske podatke te učiti iz iskustva i prilagođavati se promjenjivim uvjetima. Iako ovi roboti imaju visoku razinu autonomnosti zahvaljujući umjetnoj inteligenciji, često nisu potpuno autonomni i zahtijevaju ulaz od korisnika vezan za zadatke koje trebaju izvršiti. To može uključivati planiranje misije, postavljanje patrolnih točaka ili vremenskog rasporeda patroliranja. Postoje različita rješenja kroz koja je to moguće kao što su nadzorna ploča, web ili desktop aplikacije, zasebni softver ili mobilni uređaj. U ovom radu naglasak je stavljen na upravljanje robotskim sustavima mobilnim uređajima te je u sljedećem odlomku detaljnije obrađena tematika takvog rješenja.

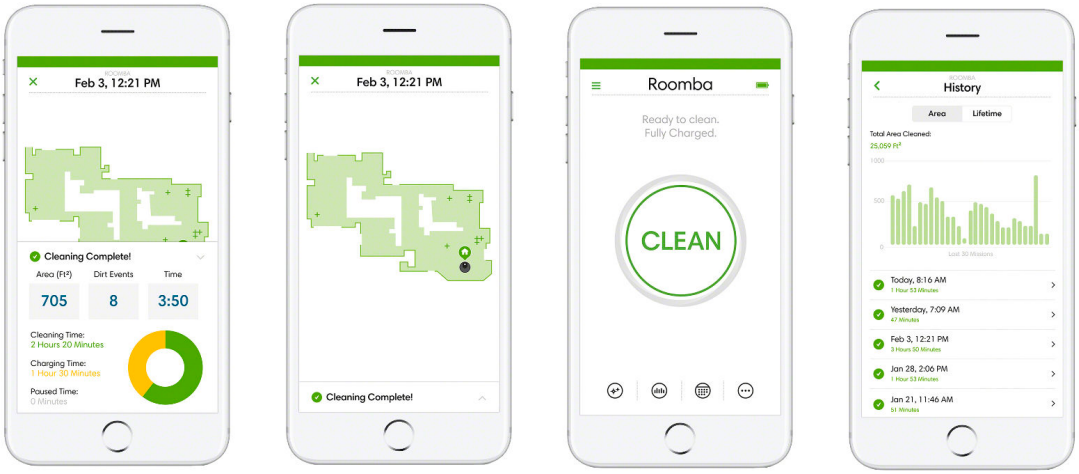
2.2. Upravljanje robotskim sustavom mobilnom aplikacijom

Korištenje mobilnog uređaja kao sučelja za upravljanje robotskim sustavom pruža nekoliko prednosti. Ono omogućuje korisnicima, poput upravitelja objekta ili osoblja za sigurnost, da s lakoćom daljinski nadgledaju i upravljaju robotskim sustavima, pristupaju ključnim podacima i dodjeljuju zadatke. Zahvaljujući praktičnosti mobilnih uređaja, upravljanje robotskim sustavima postaje fleksibilnije i dostupnije, omogućujući besprijekornu integraciju u postojeće sigurnosne okvire. Neke od ključnih točaka koje mobilna aplikacija za upravljanje robotskim sustavom može implementirati su:

- **Nadgledanje i upravljanje** – omogućuje korisnicima da u stvarnom vremenu prate aktivnosti robota, gledaju uživo prijenos s kamere robota te pristupaju senzorskim podacima. Također, korisnici mogu putem digitalnog upravljačkog sučelja upravljati kretnjom robota, simulirajući pokrete upravljačke palice povlačenjem prsta po zaslonu.
- **Kartografiranje i navigacija** – roboti za patroliranje i nadzor često se oslanjaju na sustave za kartografiranje i navigaciju kako bi se mogli autonomno kretati unutar zadane okoline. Mobilna aplikacija treba pružiti korisnicima sučelje s generiranom kartom, obično putem LiDAR tehnologije, na kojoj mogu definirati ili ažurirati rute kretanja robota, označiti ograničena područja, barijere ili postaviti točke na putanji.
- **Planiranje i dodjela zadataka** – korisnici mogu stvarati i upravljati popisom zadataka, postavljati prioritete i dodjeljivati zadatke robotu. Aplikacija može također pružiti obavijesti i upozorenja kako bi informirala korisnike o napretku zadatka, završetku ili bilo kakvim problemima koji se mogu pojaviti.
- **Održavanje i dijagnostika** – kako bi se osigurala optimalna izvedba robota potrebno je redovito održavanje komponenata sustava. Stoga, mobilna aplikacija može obuhvaćati funkcionalnosti za praćenje rasporeda održavanja, generiranje izvješća o održavanju te pružanje priručnika za rješavanje problema. Također, aplikacija može pružati dijagnostičke podatke o komponentama robotskog sustava, omogućujući korisnicima da pravovremeno identificiraju i riješe eventualne probleme.
- **Analitike** – pružanje sučelja s vizualnom prezentacijom metrika performansi i aktivnosti uslužnog robota.

Primjer mobilne aplikacije za upravljanje robotskog sustava za patroliranje i sigurnost je aplikacija za upravljanje već spomenutim robotom *Spot* tvrtke Boston Dynamics. Kroz svoje bogato korisničko sučelje (engl. *user interface, UI*), ova aplikacija nudi sve ranije spomenute značajke za upravljanje robotskim sustavom (Slika 2.3.). Još jedan primjer mobilne aplikacije za upravljanje robotskim sustavom je i aplikacija *iRobot Home.* Ova aplikacija namijenjena je za upravljanje vrlo popularnog Roomba robotskog usisavača te pruža korisničko sučelje za daljinsko upravljanje, planiranje i praćenje usisavanja, sve kako bi unaprijedila cjelokupno iskustvo čišćenja (Slika 2.4.).

Sl. 2.3. Korisničko sučelje aplikacije za upravljanje robotom *Spot* [11]



Sl. 2.4. Korisničko sučelje aplikacije za upravljanje robotskog usisavača Roomba [12]

3. KORIŠTENE TEHNOLOGIJE

Potrebno je izraditi mobilnu aplikaciju koja se može pokretati na mobilnim uređajima koji koriste iOS ili Android operativni sustav. Pored raznih radnih okvira (engl. *framework*) za razvoj višeplatformskih (engl. *multiplatform*) mobilnih aplikacija, za realizaciju ove aplikacije odabran je Flutter radni okvir. Programski jezik koji se koristi za implementaciju Flutter aplikacija je Dart. Aplikacije je napravljena u razvojnom okruženju Android Studio s instaliranim Flutter i Dart priključcima (engl. *plugins*). U ovom poglavlju bit će dan uvid u Flutter radni okvir, njegovu arhitekturu i komponente, programski jezik Dart, razvojno okruženje Android Studio te nekoliko biblioteka bez kojih ne bi bilo moguće implementirati potrebne funkcionalnosti aplikacije.

3.1. Višeplatformski razvoj mobilnih aplikacija

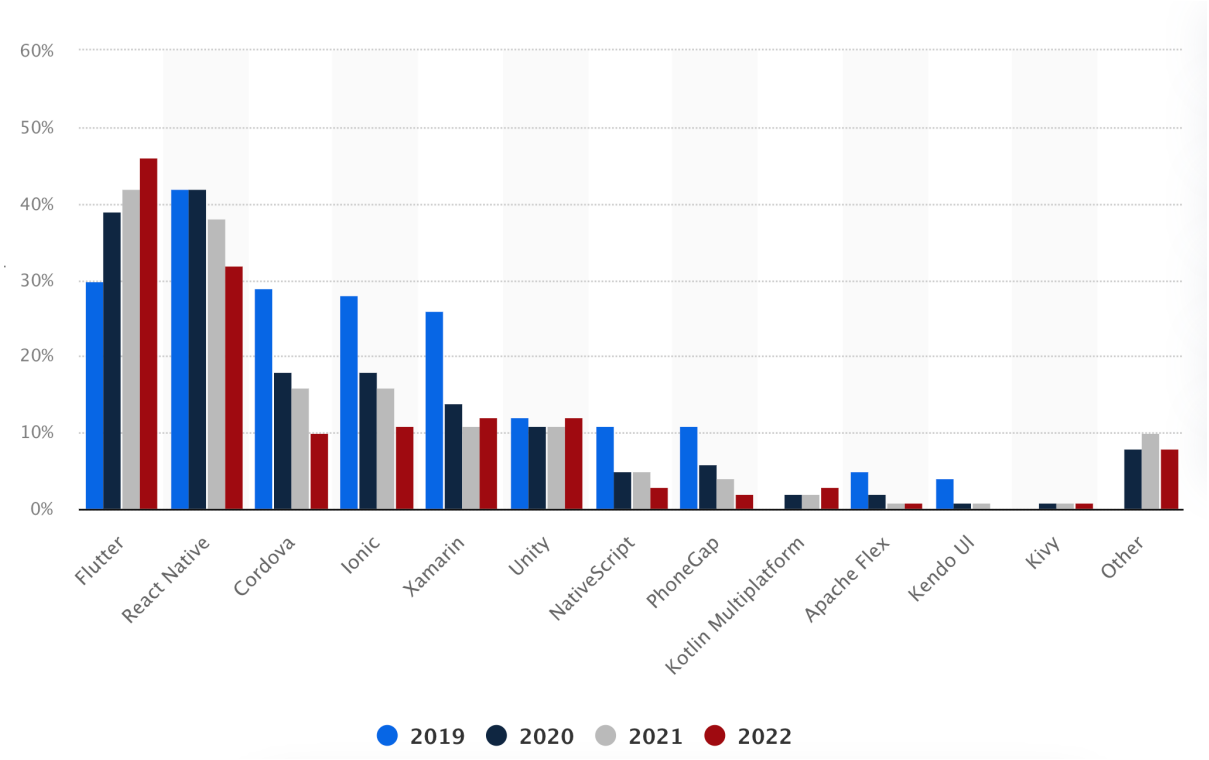
Dijeljenje programskog koda (engl. *codebase*) na više platformi postaje sve popularnije u razvoju programskih rješenja za mobilne aplikacije. Razlog je tome što se takvim pristupom može značajno smanjiti vrijeme i trošak razvoja aplikacije te još više njezinog ažuriranja i održavanja. Prema statistici više od 99% svjetskog tržišta mobilnih operativnih sustava čine iOS i Android operativni sustav (Tab. 3.1.). Stoga se višeplatformski razvoj mobilnih aplikacija većinom bazira na dijeljenju dijela ili cijelog koda između te dvije platforme.

Tab. 3.1. Prikaz stanja svjetskog tržišta mobilnih operativnih sustava u svibnju 2023. [13].

Operacijski sustav	Android	iOS	Samsung	KaiOS	Unknown	Windows
Postotak korisnka u svijetu	67.56	31.6	0.43	0.2	0.15	0.02

Prvi okviri za razvoj višeplatformskih aplikacija temeljili su se na web tehnologijama *HTML*, *JavaScript* i *WebView*, a primjer su *PhoneGap*, *Ionic* i *Apache Cordova*. Druga skupina okvira za razvoj višeplatformskih aplikacija više ne koristi *HTML* za prikaz već se temelji na preslikavanju svojih komponenti u izvorne (engl. *native*) komponente iOS i Android operativnih sustava [14]. U tu skupinu spadaju *Xamarin* i *ReactNative*. Flutter radni okvir spada u treću skupinu koja se temelji na vlastitom iscrtavanju grafičkih elemenata koristeći izvornu komponentu platna za iscrtavanje

(engl. *canvas*). Slika 3.1. jasno prikazuje postepeni porast popularnosti Flutter radnog okvira tijekom godina, te njegovu trenutnu poziciju kao vodećeg višeplatformskog mobilnog okvira u 2022. godini. Dok će mnogi i dalje težiti k izvornom razvoju mobilnih aplikacija smatrajući najbitnijim korisničko iskustvo i snažne performanse koje takve aplikacije pružaju, drugi će zbog već spomenutih razloga reduciranja troška razvoja i održavanja aplikacije na obje platforme ipak odabrati višeplatformski razvoj.



Sl. 3.1. Usporedba trendova korištenja višeplatformskih mobilnih okvira od 2019. do 2022. godine [15]

3.2. Flutter radni okvir

Flutter je radni okvir otvorenog koda (engl. *open source code*) za razvoj višeplatformskih izvorno kompiliranih (engl. *natively compiled*) aplikacija iz jednog dijeljenog programsko koda [16]. Prvenstveno su ga razvili inženjeri kompanije Google, a repozitorij projekta se može pronaći na općepoznatom servisu za verzioniranje koda – GitHub [17], što omogućuje drugim kompanijama i individualcima sudjelovanje u razvoju i poboljšanju okvira. Flutter 1.0, to jest prva stabilna verzija okvira s kojom je podržan razvoj višeplatformskih aplikacija za iOS i Android platforme lansirana je u prosincu 2018. godine. S verzijom Flutter 2.0 koja je objavljena u ožujku 2021. godine predstavljena je podrška za još tri platforme: Windows, macOS i Linux. Ažuriranja

Flutter radnog okvira, koja uključuju poboljšanje stabilnosti i performansi te često i rješenje nekog korisničkog zahtjeva, događaju se u pravilu kvartalno [16]. Flutter radni okvir svakim danom stječe sve veću popularnost, a da je on ozbiljan trend koji raste svjedoče i brojne velike kompanije koje su svoje aplikacije izradile oslanjajući se na njega, to su primjerice *BMW*, *Toyota*, *Alibaba Group* i *eBay*. Također, na službenoj internet stranici Flutter radnog okvira stoji da je do sada više od 400.000 aplikacija isporučeno pomoću njega, što je samo pokazatelj da su mnogi razvojni programeri prepoznali širok spektar prednosti koje im taj radni okvir nudi.

3.2.1. Ključne karakteristike

Osim što omogućava kreiranje višeplatformskih aplikacija dijeljenim programskim kodom, Flutter radni okvir prate još mnoge prednosti, od kojih svakako valja izdvojiti mogućnost brzog ponovnog učitavanja (engl. *hot reload*), lijepo i prilagodljivo grafičko sučelje, te odlične performanse.

Hot reload je izuzetno korisna značajka koju Flutter radni okvir pruža razvojnim programerima. Pomoću nje je moguće vidjeti promjene u aplikaciji u stvarnom vremenu bez potrebe za ponovnim pokretanjem cijelog projekta i gubljenju stanja aplikacije. Nakon što se promjena napravi u kodu i klikne gumb za pokretanje *hot reload* značajke, učitavaju se samo napravljene izmjene i za manje od sekunde ažurira se korisničko sučelje. Ovo omogućava programerima da u vremenu izvršavanja aplikacije eksperimentiraju i poboljšavaju svoj kod bez gubljenja vremena na dugotrajne procese kompilacije i pokretanja aplikacije, kao što je slučaj kod većine drugih radnih okvira. Zahvaljujući ovoj značajki programerima je olakšano otkrivanje i ispravljanje pogreški što uvelike ubrzava razvojni proces aplikacije.

Za lijepo i prilagodljivo grafičko sučelje u Flutter aplikacijama zaslužno je vjerno kopiranje izvornih mobilnih grafičkih elemenata prilikom iscrtavanja korisničkog sučelja. Takve je komponente gotovo nemoguće razlikovati od pravih izvornih mobilnih komponenti. Dodatno tome, Flutter radni okvir nudi širok spektar vlastitih predefiniranih grafičkih komponenti, poznatih kao *widget*, koje pojednostavljuju i ubrzavaju proces slaganja i kreiranja korisničkog sučelja. Više o ovoj osnovnoj gradivnoj jedinici Flutter radnog okvira opisano je u poglavlju 3.2.3.

Flutter radni okvir ističe se po pružanju iznimnih performansi prilikom izvođenja aplikacija. Za to je zaslužna kombinacija visoko optimizirane mašine za iscrtavanje (engl. *rendering engine*), učinkovitost *widget frameworka* i mogućnost korištenja JIT (engl. *just-in-time*) i AOT (engl. *ahead-of-time*) kompilacija. Mašina za iscrtavanje Flutter radnog okvira koristi grafičku biblioteku (engl. *library*) *Skia* koja pruža brzo i učinkovito iscrtavanje 2D grafike na platno zaslona uređaja

eliminirajući nedostatke povezane s tradicionalnim UI radnim okvirima koji često ovise o platformskim komponentama. Učinkovitost *widget frameworka* postignuta je kroz reaktivan pristup koji omogućuje ažuriranje samo onih jedinica na koje se odnosi promjena. Ovaj visoko optimizirani pristup osigurava minimalnu potrošnju resursa i maksimalne performanse. Osim toga, Flutter radni okvir nudi dvije vrste kompilacije, JIT i AOT, koje se mogu odabrati ovisno o potrebama i fazi razvoja aplikacije. JIT kompilacija omogućuje brzo ponovno učitavanje i fleksibilnost tijekom faze razvoja i otklanjanja pogrešaka, dok AOT kompilacija maksimalno optimizira izvršne datoteke, pružajući iznimne performanse u produkcijskim verzijama aplikacije.

Također, bitno je napomenuti da Flutter radni okvir iza sebe ima jednu snažnu i veoma aktivnu zajednicu programera. To znači da postoji obilje izvora resursa, materijala za učenje te velik broj biblioteka koje mogu biti izuzetno korisne u razvoju aplikacija. Ova zajednica neprestano raste iz dana u dan, a jedan od razloga za to je redovito ažuriranje, ispravljanje pogrešaka i poboljšanje Flutter radnog okvira, uz prateću kvalitetnu dokumentaciju.

3.2.2. Arhitektura

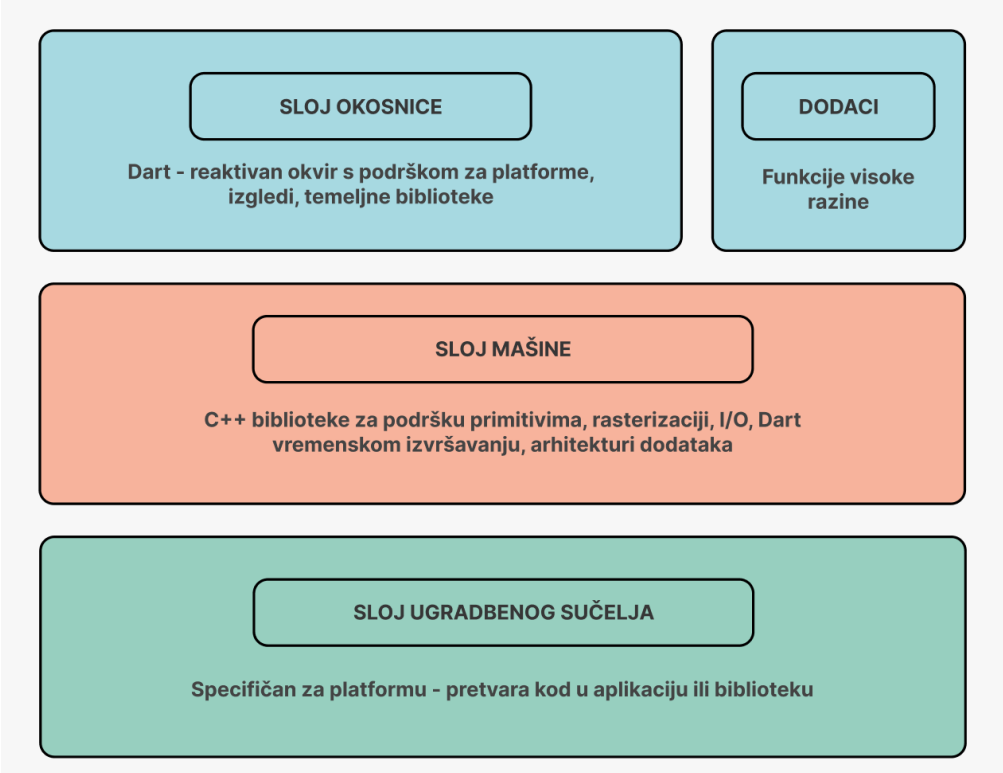
Arhitektura Flutter radnog okvira je modularna i slojevita. Takva arhitektura osigurava dosljedno ponašanje jednom napisane logike aplikacije na svim platformama bez obzira što se kôd na kojem radi mašina (engl. *engine*) razlikuje po platformi [18].

Glavna tri sloja na koja se dijeli arhitektura Flutter radnog okvira su (Slika 3.2.):

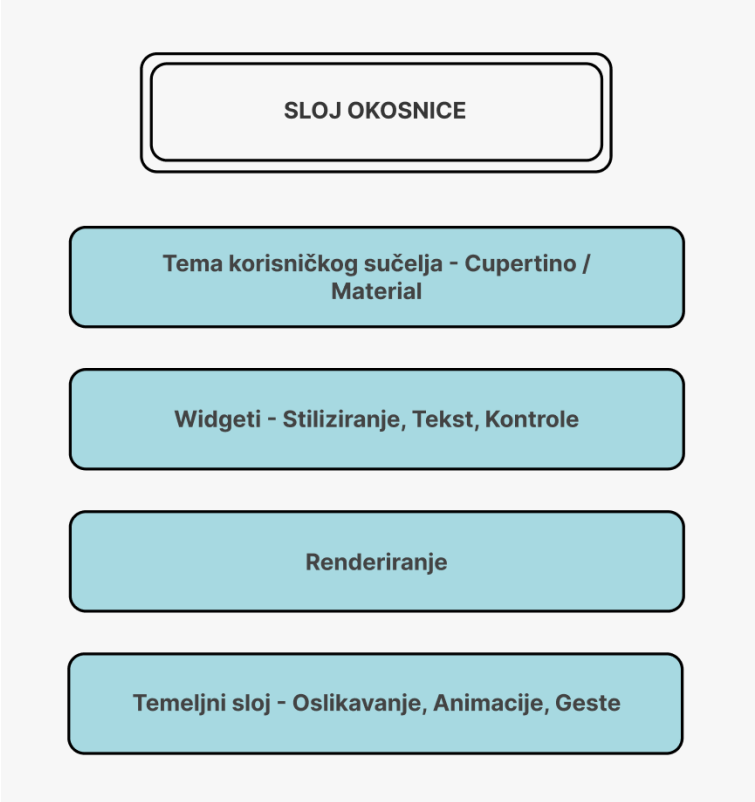
- **Sloj ugradbenog sučelja** (engl. *Embedder layer*) – nalazi se na dnu i služi za integriranje Flutter koda u različite operacijske sustave koje podržava Flutter radni okvir. Pisan je u programskom jeziku specifičnom za platformu (Java i C++ za Android, Objective-C/ Objective-C++ za iOS i macOS i C++ za Windows i Linux). Flutter kod može biti pakiran kao samostalna aplikacija ili kao ugrađeni modul.
- **Sloj Mašine** (engl. *Engine layer*) – napisan je većinom u programskom jeziku C++, a implementira komponente niske razine Flutter aplikacijskog programskog sučelja (engl. *API - Application Programming Interface*) uključujući biblioteke za iscrtavanje 2D grafike, raspored teksta, datotečne i mrežne ulazno/izlazne operacije, podršku za pristupačnost, arhitekturu priključaka i vrijeme izvođenja (engl. *runtime*) Dart programskog jezika.
- **Sloj okosnice** (engl. *Framework layer*) – nalazi se na vrhu i napisan je u programskom jeziku Dart te sadrži biblioteke komponenti visoke razine koje se koriste za izgradnju aplikacija. Na istoj razini nalaze se i priključci za značajke (engl. *feature*) visoke razine

kao što su JSON serijalizacija, geolokacija, pristup kameri i slično. Takav način omogućuje uključivanje samo onih značajki koje su potrebne aplikaciji. Sloj okosnice sastoji se od nekoliko podslojeva (Slika 3.3.):

- Temeljni sloj (engl. *Foundation layer*) – sadrži osnovne građevne elemente kao što su animacije i geste, koje grade više slojeve.
- Sloj renderiranja – apstrahira rad s izgledom (engl. *layout*) tako da omogućava gradnju stabla objekata kojima se može upravljati dinamički tako da stablo uvijek odražava trenutno stanje.
- Sloj *widget* objekata – pruža widget objekte koji predstavljaju objekte renderiranja iz sloja renderiranja preko kojih se sastavlja dizajn i interaktivni elementi aplikacije.
- Teme korisničkog sučelja (engl. *UI theme*) – biblioteke koje se oslanjaju na Material (Android) ili Cupertino (iOS) dizajn smjernice kako bi omogućile kreiranje aplikacija koje izgledaju što sličnije onim izvornima.



Sl. 3.2. Vizualni prikaz arhitekture Flutter radnog okvira

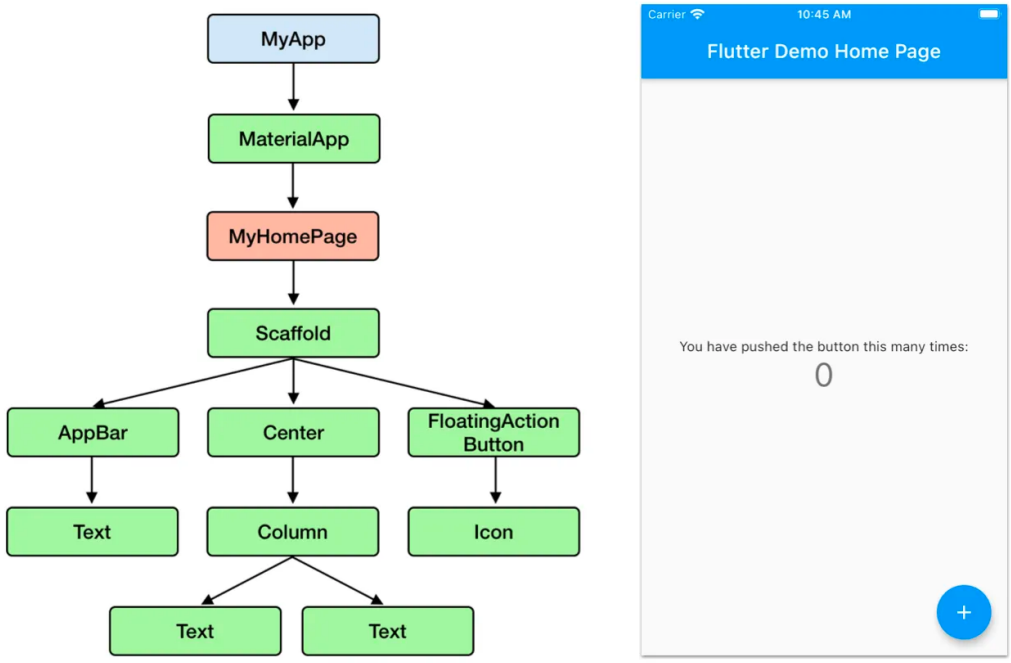


Sl. 3.3. Prikaz slojeva okosnice Flutter radnog okvira

3.2.3. Osnovna gradivna jedinica

Osnovna gradivna jedinica kompozicije u Flutter radnom okviru naziva se *widget* [19]. Na internetu, u člancima, knjigama pa čak i samoj službenoj dokumentaciji uz radni okvir Flutter nerijetko se veže fraza *Sve je widget* (engl. *Everything is a widget*). To nikako ne znači da u Flutter radnom okviru ne postoje drugi objekti osim *widget* objekata, već da je svaki dio korisničkog sučelja aplikacije izgrađen od njih. U svojoj srži *widget* je Dart klasa koja zna kako iscrtati određenu komponentu korisničkog sučelja kao što je gumb, element za unos teksta, slika itd. Hijerarhija *widget* objekata koji čine korisničko sučelje naziva se još i stablo widget objekata. Primjer jednog stabla widget objekata vidljiv je na slici 3.4. Korijenski *widget* obično je *MaterialApp* ili *CupertinoApp*, ovisno o jeziku dizajna aplikacije, i u njemu se najčešće postavljaju teme boje i teksta koje se koriste kroz cijelu aplikaciju. *Widget Scaffold* koristi se kao *widget* najviše razine za izradu zaslona aplikacije te omogućuje fleksibilan sustav izgleda koji olakšava dodavanje i raspoređivanje podređenih *widget* objekata kao što su *AppBar*, *FloatingActionButton*, *Drawer*, *SnackBar*.

Widget objekti se mogu kombinirati kako bi se formirali složeniji elementi te ih je moguće ponovno koristiti unutar iste aplikacije ili čak u drugim aplikacijama. *Widget* objekti u Flutter radnom okviru su deklarativni, što znači da opisuju kako bi korisničko sučelje trebalo izgledati na temelju njihove trenutne konfiguracije i stanja. Kada se stanje *widget* objekta promijeni, okvir automatski ažurira stablo *widget* objekata te dolazi do odražavanja promjena. Widget objekti imaju metodu *build* koja definira strukturu i sadržaj *widget* objekta. Metoda *build* odgovorna je za vraćanje *widget* objekta koji predstavlja element korisničkog sučelja. Najznačajnija podjela *widget* objekata je na one bez stanja (engl. *stateless*) i one sa stanjem (engl. *stateful*). *Stateless widget* objekti su nepromjenjivi s vremenom, dok *stateful widget* objekti mogu promijeniti svoj izgled ili ponašanje kao odgovor na interakciju korisnika, promijene podataka ili druge podražaje. Uz osnovni skup *widget* objekata koje nudi Flutter radni okvir, moguće je također kreirati i vlastite *widget* objekte prilagođene zahtjevima aplikacije. Prilagođeni *widget* objekti (engl. *custom widgets*) mogu biti kreirani od nule ili se se mogu temeljiti na već postojećim *widget* objektima, koji se modificiraju i proširuju kako bi im se dodale neke nove funkcionalnosti. *Widget* objekti u Flutter radnom okviru dizajnirani su da budu brzi, učinkoviti i fleksibilni, što programerima olakšava stvaranje prekrasnih i responzivnih korisničkih sučelja za njihove aplikacije.



Sl. 3.4. Prikaz korisničkog sučelja demo Flutter aplikacije i hijerarhijskog stabla njezinih *widget* elemenata [20]

3.3. Programski jezik Dart

Dart je suvremeni programski jezik otvorenog koda razvijen od Google inženjera s ciljem pružanja efikasne platforme za izgradnju mobilnih, web i desktop aplikacija [21]. Korijeni Dart programskog jezika mogu se pratiti sve do 2011. godine, kada je započeo njegov razvoj. Nakon nekoliko godina rada i iteracija, prva stabilna verzija objavljena je 2013. godine. Od tada, Dart se kontinuirano razvija i unaprjeđuje kako bi pružio programerima moćan alat za razvoj aplikacija. Danas je Dart izuzetno popularan i sveprisutan u razvoju višeplatformskih aplikacija. Trenutno, najnovija verzija Dart programskog jezika je verzija 3.0.4.

Dart se ponosi svojom podrškom za nekoliko ključnih značajki koje ga čine iznimno atraktivnim u programerskoj zajednici. Jedna od tih značajki je da Dart podržava dinamičko tipiziranje (engl. *dynamic typing*). Time omogućuje programerima da deklariraju varijable bez eksplicitnog navođenja tipa ili da mijenjaju tip varijable tijekom izvođenja programa. Ova fleksibilnost omogućuje brzi razvoj i prototipiranje, jer se efikasno mogu dodavati nove funkcionalnosti ili mijenjati postojeće bez potrebe za promjenom tipova ili deklaracija. Primjer dinamičkog tipiziranja vidljiv je u programskom kodu 3.1. Iako Dart podržava dinamičko tipiziranje, statički je tipizirani jezik (engl. *statically typed language*). To znači da se tijekom faze kompilacije provjerava ispravnost tipova varijabli i izraza. Ova statička provjera tipova pruža prednosti kao što su poboljšana pouzdanost koda i pojednostavljeno refaktoriranje. Otkrivanje grešaka tijekom faze kompilacije omogućuje programerima da ih isprave prije nego što se aplikacija uopće pokrene, smanjujući rizik od grešaka tijekom izvođenja programa i poboljšavajući ukupnu kvalitetu koda. Primjer statičkog tipiziranja dan je u programskom kodu 3.2.

```
dynamic broj = 5; // varijabla "broj" može biti bilo kojeg tipa
dynamic ime = "Pero"; // također i varijabla "ime"


dynamic rezultat = broj + ime; // Izvršava se bez greške,
                               // rezultat je "5Pero"
```

Programski kod 3.1. Primjer dinamičkog tipiziranja u Dart programskom jeziku

```
int broj = 5; // varijabla "broj" je tipa int
String ime = "Pero"; // varijabla "ime" je tipa String


int rezultat = broj + ime; // Greška! Ne možemo zbrojiti int i String
```

Programski kod 3.2. Primjer statičkog tipiziranja i prevencije greške prije izvršavanja programa

Osim toga, Dart podržava i sigurnost od nulabilnih vrijednosti (engl. *null safety*). Nulabilne vrijednosti često uzrokuju probleme u programima, kao što su iznimke nulabilne vrijednosti (engl. *Null pointer exceptions*) i rušenje aplikacije. Dart je dizajniran s fokusom na smanjenje takvih pogrešaka. Sigurnost od nulabilnih vrijednosti omogućuje programerima da jasno definiraju koje varijable mogu biti nulabilne i koje ne mogu. To govori kompajleru da provjeri pravilnost koda i upozori na potencijalne probleme prije izvođenja. Programski kod 3.3. prikazuje primjer deklaracije nenulabilne i nulabilne varijable.

```
String ime = "Pero"; // varijabla koja nije nullable
String? nullableIme = null; // nullable varijabla

ime = null; // Greška: Vrijednost tipa 'Null' se ne može
      // dodijeliti varijabli tipa 'String'

nullableIme = "Luka"; // Nema greške
nullableIme = null; // Nema greške
```

Programski kod 3.3. Primjer deklariranja nenulabilne i nulabilne varijable

Također, Dart podržava osnovne principe objektno orijentiranog programiranja, što programerima omogućava korištenje klasa, polimorfizma i nasljeđivanja za izgradnju organiziranog, modularnog i ponovno upotrebljivog koda. Pored toga, Dart sadrži efikasan mehanizam za automatsko upravljanje memorijom. Njegov sakupljač otpada (engl. *garbage collector*) veoma brzo uklanja stare i stvara nove objekte. U Flutter radnom okviru ovo rezultira veoma glatkim animacijama sa stalnom brzinom od 60 okvira po sekundi (engl. *frames per second, fps*) [22]. Sustav automatskog upravljanja memorijom u Dart programskom jeziku oslobađa programere brige o dealokaciji memorije i smanjuje rizik od curenja memorije (engl. *memory leak*). Važno je napomenuti da dvije vrste kompilacije u Flutter radnom okviru, JIT i AOT, ne bi

bile moguće bez podrške Dart programskog jezika u pozadini. Stoga se može reći da Dart igra ključnu ulogu u postizanju visokih performansi koje Flutter radni okvir pruža.

Sintaksa Dart programskog jezika vrlo je slična popularnim programskim jezicima kao što su Java, C, C#, Swift, Kotlin i drugi. Zbog toga je programerima koji su već upoznati s tim jezicima prijelaz na Dart iznimno lagan. Dart podržava i značajke funkcionalnog programiranja kao što su lambda izrazi, funkcije višeg reda, funkcionalni operatori za mapiranje, filtriranje i izvršavanje ostalih operacija nad kolekcijama. Osim toga, Dart ima snažnu podršku za asinkrono programiranje putem *async/await* mehanizma. To omogućuje jednostavno rukovanje blokirajućim operacijama, odnosno operacijama koje zahtijevaju više vremena za izvršavanje, kao što su mrežni pozivi ili čitanje/pisanje datoteke.

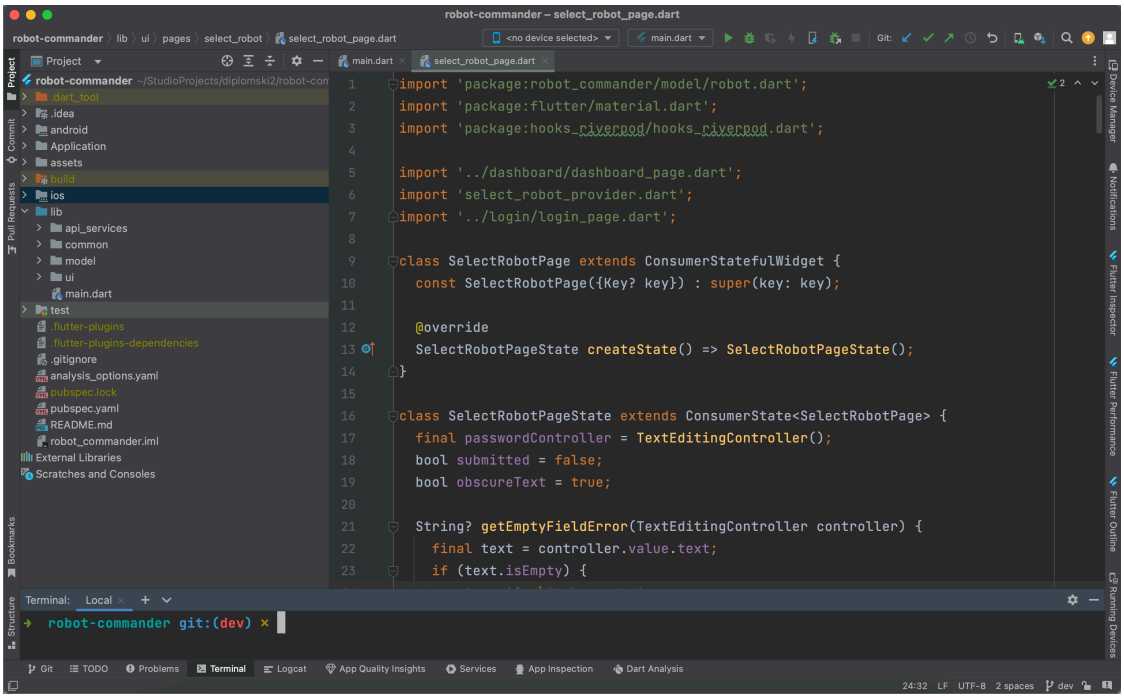
3.4. Razvojno okruženje Android Studio

Izrada Flutter aplikacija moguća je u raznim razvojnim okruženjima, a najčešći izbor razvojnih programera su Android Studio, Visual Studio Code i IntelliJ IDEA. Razlog tomu je njihova popularnost i specifična podrška za Flutter radni okvir. Za izradu ove aplikacije izabran je Android Studio.

Android Studio je integrirano razvojno okruženje (engl. *Integrated Development Environment, IDE*) prvenstveno dizajnirano za razvoj Android aplikacija. Razvila ga je tvrtka Google na temelju IntelliJ IDEA softveru tvrtke JetBrains i pustila u javnu upotrebu 2014. godine. Tijekom godina, Android Studio je kontinuirano ažuriran i unaprjeđivan, integrirajući brojne nove značajke u svoj skup alata [23]. Neke od bitnijih značajki Android Studio razvojnog okruženja su:

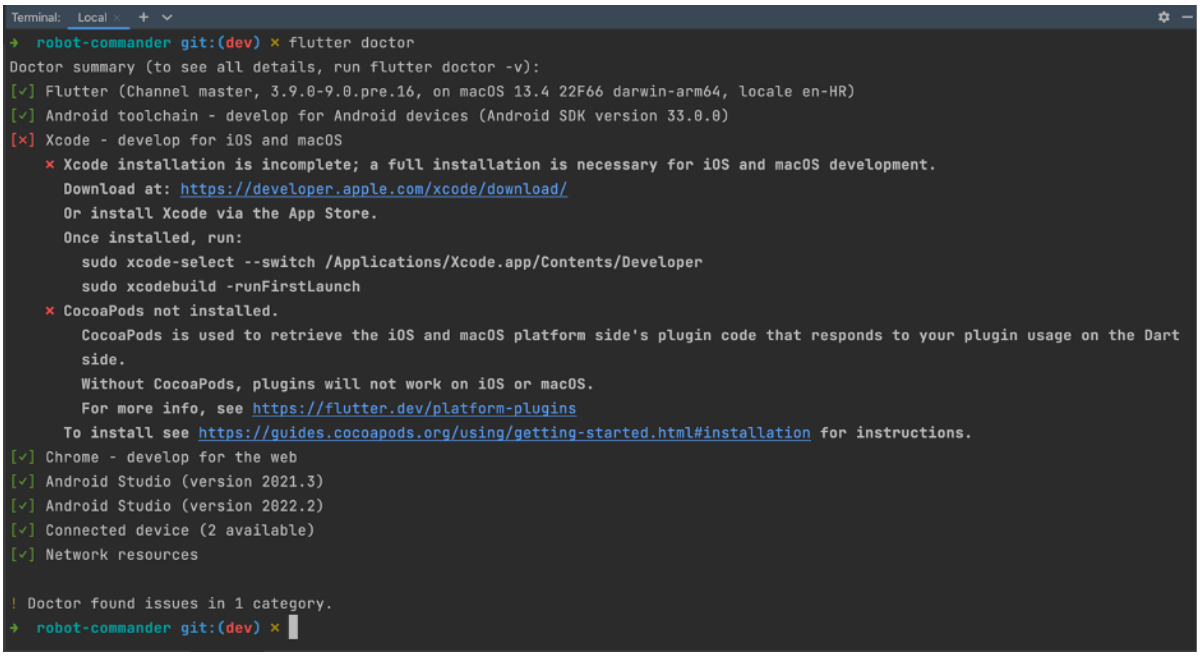
- Uređivač koda – sadrži bogatu podršku za sintaksu, refaktoriranje, automatsko dovršavanje i pretraživanje koda, podržavajući razne programske jezike.
- Način kompiliranja i izgradnje – koristi sustav izgradnje temeljen na Gradle alatu koji omogućava prilagodbu procesa izgradnje i rukovanje ovisnostima projekta.
- Otklanjanje pogrešaka – ugrađeni alat za otklanjanje pogrešaka (engl. *debugger*) omogućava praćenje izvršavanja aplikacije korak-po-korak, te inspekciju varijabli što programerima uvelike olakšava pronalaženje i ispravljanje pogrešaka u kodu.
- Testiranje – Android Studio dolazi s Android Emulatorima koji omogućuju pokretanje i testiranje aplikacija na virtualnom Android uređaju. Također podržava i povezivanje fizičkih uređaja za testiranje.

- Analiza performansi – pruža alate za profiliranje performansi aplikacije kako bi bilo moguće identificirati i ispraviti probleme s performansama, uključujući upotrebu glavne procesorske jedinice sustava, memorije i mreže.



Sl. 3.5. Prikaz korisničkog sučelja Razvojnog okruženja Android Studio

Za razvoj Flutter aplikacije u razvojnom okruženju Android Studio potrebno je prvo instalirati Flutter paket za razvoj softvera (engl. *Software development kit, SDK*) i dodati Flutter priključak u Android Studio. Nakon konfiguracije odmah je moguće početi sa stvaranjem Flutter projekta, pisanjem Dart koda i korištenjem svih gore navedenih značajki. Kako Flutter SDK dolazi s Flutter komandnom linijom, s naredbom *flutter doctor* moguće je provjeriti uspješnost postavljanja i konfiguracije Flutter radnog okvira s odabranim okruženjem. Ova naredba identificira probleme ili nedostajuće komponente, te pruža smjernice o tome kako riješiti te probleme kao što je vidljivo na slici 3.6.



Sl. 3.6. Prikaz generiranog odgovora nakon pokrenute *flutter doctor* naredbe

3.5. Riverpod biblioteka

Upravljanje stanjem je veoma bitan aspekt izrade Flutter aplikacije jer uključuje upravljanje i ažuriranje podataka te njihovog prikazivanje na korisničkom sučelju. Flutter pruža različite opcije za upravljanje stanjem, a jedan popularan izbor je korištenje biblioteke *Riverpod*.

Riverpod je izuzetno korisna biblioteka koja pruža jednostavan i intuitivan načina za upravljanje stanjem Flutter aplikacije. Slijedi principe arhitekture zasnovane na pružateljima (engl. *provider-based architecture*), što olakšava upravljanje ovisnostima i dohvaćanje stanja sa različitih mjesta. Pomoću *Riverpod* biblioteke moguće je definirati pružatelje koji čuvaju stanje aplikacije. Pružatelji djeluju kao izvor istine za podatke i mogu im pristupiti svi *widget* objekti u stablu *widget* objekata. To omogućuje da se bez ponavljajućeg koda (engl. *boilerplate code*) prosljeđuje i ažurira stanje. Pružatelji su potpuna zamjena za obrasce kao što su *Singletone*, *Service Locators*, *Dependency Injection* i slično. *Riverpod* također podržava različite pristupe upravljanju stanjem, uključujući tako pružatelje za nepromjenjivo, promjenjivo i asinkrono stanje. Uz to, *Riverpod* se dobro slaže s drugim Flutter paketima i značajkama, poput već spomenute značajke *hot reload*.

Integriranje *Riverpod* biblioteke u Flutter aplikaciju je iznimno jednostavno. Prvo je potrebno instalirati paket na način da se u *pubspec.yaml* datoteci projekta specificira *flutter_riverpod* biblioteka sa verzijom i zatim pokrene njezino dohvaćanje s *flutter pub get* naredbom. Nakon što

je to izvršeno potrebno je korijenski *widget* aplikacije zamotati u *ProviderScope widget* čime je onda automatski omogućen rad pružateljima. Primjer stvaranja pružatelja i čitanja vrijednosti objekta iz pružatelja dan je programskim kodom 3.7. Varijabla pružatelja trebala bi uvijek biti označena kao konačna (engl. *final*). *Provider* je najosnovniji oblik pružatelja i on izlaže objekt koji nije moguće promijeniti. Osim njega postoje i pružatelji čije je objekte moguće promijeniti kao što su *StreamProvider* ili *StateNotifierProvider.* Kako bi mogli pročitati vrijednost objekta pružatelja, potrebno je prvo imati referencu na *ref* objekt. Ovaj objekt omogućuje interakciju s pružateljima, bilo da je riječ o *widget* objektu ili nekom drugom pružatelju. *Widget* objekti sami po sebi nemaju referencu na *ref* objekt i zato je potrebno, umjesto klasičnih *StatelessWidget* i *StatefullWidget objekata,* koristiti *ConsumerWidget* i *ConsumerStatefullWidget* objekte koji su dio *flutter_riverpod* paketa. *ConsumerWidget* objekt razlikuje se od *StatelessWidget* objekta po jednoj stvari, a to je da sadrži dodatan *WidgetRef* parametra u svojoj metodi izrade (engl. *build*) preko kojeg je onda moguće čitati vrijednost objekta pružatelja. Slično je i s *ConsumerStatefullWidget* objektom. Njegov *ConsumerState* sadrži referencu na *ref* objekt koju je moguće koristiti za čitanje vrijednosti pružatelja u bilo kojoj životnoj metodi (engl. *life-cycle method*) *StatefullWidget* objekta [24].

```dart
import 'package:flutter/cupertino.dart';
import 'package:hooks_riverpod/hooks_riverpod.dart';

final nameProvider = Provider((ref) => 'Tena');

class HomeView extends ConsumerWidget {
  const HomeView({Key? key}): super(key: key);

  @override
  Widget build(BuildContext context, WidgetRef ref) {
    // koristimo ref kako bi čitali vrijednost pružatelja
    final name = ref.watch(nameProvider);
    return Text('Bok! Moje ime je $name!');
  }
}
```

Programski kod 3.4. Primjer instanciranja pružatelja i čitanja njegove vrijednosti unutar *widget* objekta

3.6. Dio biblioteka

Umrežavanje (engl. *networking*) aplikacije s udaljenim serverom radi stvaranja komunikacije neizostavni je dio skoro pa svake aplikacije. Umrežavanje omogućuje aplikaciji da šalje i prima potrebne podatke. Najčešći način umrežavanja je komunikacija putem REST (engl. *Representational State Transfer*) aplikacijskog programskog sučelja (engl. *Application Programming Interface, API*). REST API pruža standardiziran način komunikacije između klijenta i poslužitelja putem protokola za prijenos hiperteksta (engl. *Hypertext Transfer Protocol, HTTP*).

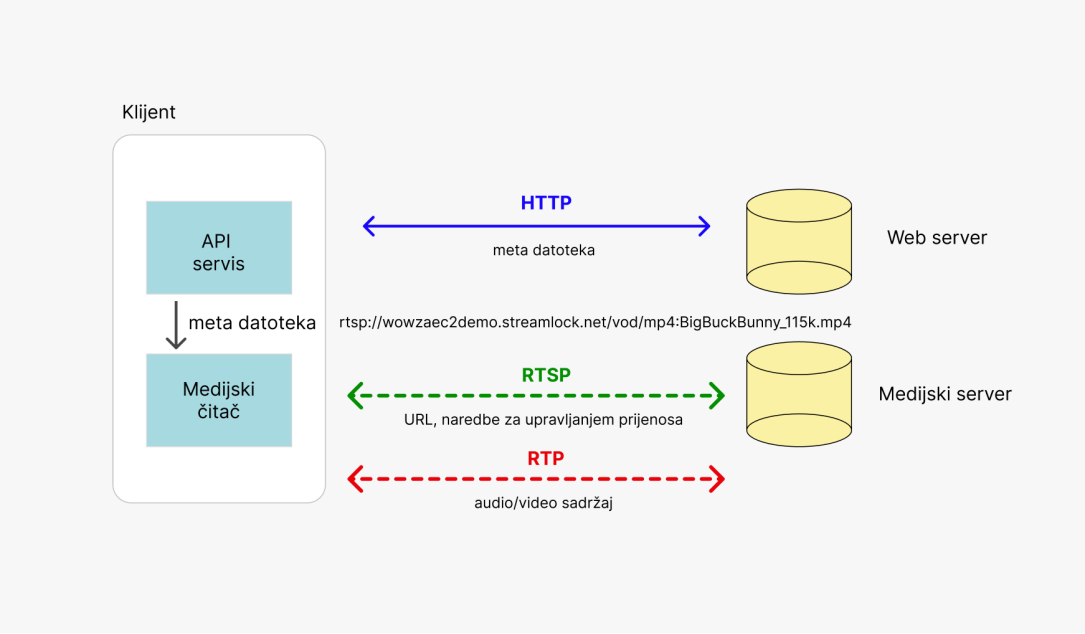
Flutter radni okvir pruža razne biblioteke i alate koji olakšavaju implementaciju umrežavanja. Biblioteka korištena za umrežavanje u razvoju ove aplikacije je *Dio*. *Dio* je popularna biblioteka koja omogućuje izgradnju HTTP klijenta i obavljanje HTTP zahtjeva. Ona pruža jednostavan i intuitivan API za izvršavanje GET, PUT, POST, DELETE i drugih HTTP metoda, kao i za rukovanje odgovorima i pogreškama. Uz to, *Dio* biblioteka podržava različite mogućnosti konfiguracije, poput postavljanja zaglavlja zahtjeva, slanja parametara i tijela zahtjeva, upravljanja kolačićima, rukovanja s HTTPS certifikatima i još mnogo toga. Također podržava asinkrono izvršavanje zahtjeva i omogućuje praćenje napretka zahtjeva. Programski kod 3.5. prikazuje primjer korištenja *Dio* biblioteke za dohvaćanje podataka jednostavnom GET metodom [25].

```
void fetchData() async {
  try {
    // Stvaranje instance "dio" objekta
    Dio dio = Dio();

    // Slanje GET zahtjeva na određeni URL
    Response response = await dio.get('https://api.example.com/data');

   // Dobivanje odgovora
   if (response.statusCode == 200) {
     // Uspješan odgovor
     var data = response.data;
     print('Podaci: $data');
   } else {
     // Neuspješan odgovor
     print('Greška u dohvaćanju podataka.');
   }
 } catch (e) {
   // Greška prilikom slanja zahtjeva
   print('Greška: $e');
 }
}
```

Programski kod 3.5. Primjer korištenja *Dio* biblioteke za dohvaćanje podataka

3.7. Protokoli i alati za prijenos uživo

Jedna od zadaća aplikacije je omogućiti prijenos uživo s kamere robota. Za tu svrhu primijenjen je protokol za prijenos uživo u stvarnom vremenu (engl. *Real-Time Streaming Protocol, RTSP*), koji se koristi za upravljanje prijenosom multimedijskog sadržaja uživo preko IP mreže (engl. *Internet Protocol*) u stvarnom vremenu. RTSP je često korišten protokol u aplikacijama koje zahtijevaju prijenos video sadržaja u stvarnom vremenu, poput video nadzora, televizijskih emitiranja uživo, video konferencija i slično. Protokol definira komunikacijski model između RTSP klijenta i RTSP poslužitelja, omogućujući kontrolu nad reprodukcijom, prikazom i upravljanjem medijskog sadržaja. Klijent koristi URL (engl. *Uniform Resource Locator*) kako bi uspostavio komunikaciju sa serverom i identificirao željeni medijski sadržaj. URL obično sadrži informacije o adresi servera, portu i putanji do medijskog sadržaja, čime se omogućuje precizno lociranje i pristupanje željenom sadržaju. Za prenošenje samih medijskih podataka koristi se protokol za prijenos u stvarnom vremenu (engl. *Real-Time Transport Protocol, RTP*). RTP paketi sadrže važne informacije poput vremenskih oznaka i vrsti podataka, što omogućuje primatelju da rekonstruira kontinuirani tok medijskog sadržaja. Zahvaljujući tome, primatelj može započeti s reprodukcijom prvog RTP paketa dok istovremeno prima i obrađuje sljedeće pakete [26]. Slika 3.7. vizualno prikazuje cijeli proces prijenosa uživo putem RTSP i RTP protokola.



Sl. 3.7. Vizualizacija prijenosa uživo s RTSP i RTP protokolima

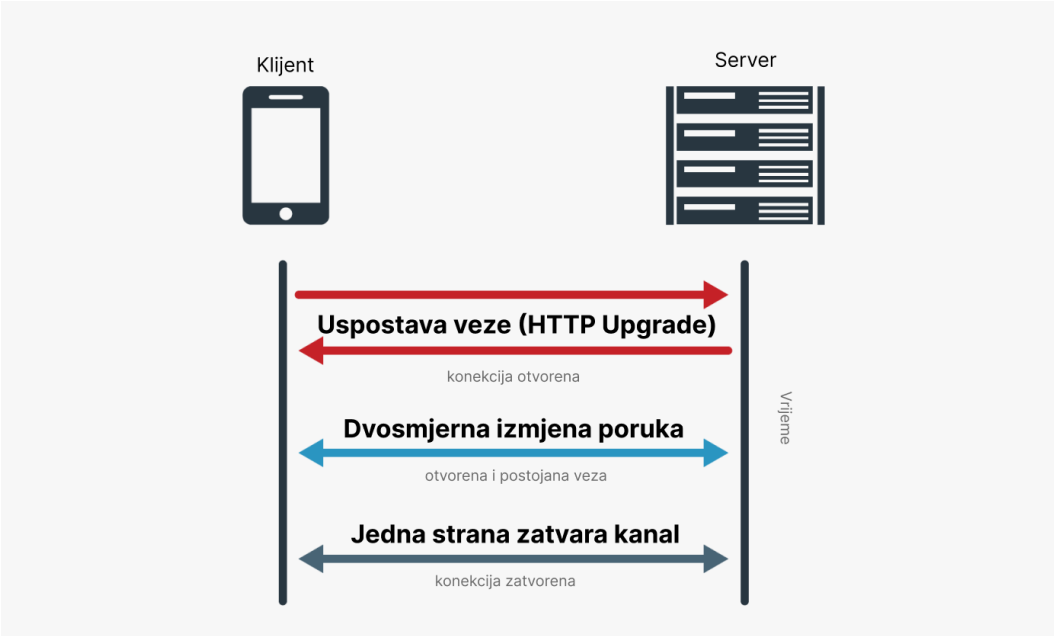
Za integraciju prijenosa uživo putem RTSP i RTP protokola u Flutter aplikaciji korišten je *flutter_vlc_player* priključak. Ovaj priključak pruža podršku za reprodukciju širokog spektra medijskih formata i nudi napredne značajke za bogato iskustvo reprodukcije medija, a kompatibilan je s različitim platformama. Za implementaciju medijskog čitača, za početak potrebno je dodati *flutter_vlc_player* kao zavisnost u *pubspec.yml* datoteku i pokrenuti *flutter pub get* naredbu kako bi paket bio integriran u projekt. Nakon toga moguće je koristiti *VlcPlayer widget* koji je odgovoran za prikaz video sadržaja. *VlcPlayer widget* objektu se prilikom instanciranja prosljeđuje *VlcPlayerController* koji je zadužen za konfiguraciju i reprodukciju videa unutar *VlcPlayer widget* objekta. *VlcPlayerController* omogućuje pozivanje reprodukcijskih metoda nad medijskim sadržajem kao što su pokreni, pauziraj i prekini. Također, putem njega moguće je promijeniti i sam izvor medija [27].

3.8. WebSocket komunikacijski protokol

WebSocket je komunikacijski protokol koji omogućuje dvosmjernu interakciju između klijenta i poslužitelja putem jedne veze protokola za kontrolu prijenosa (engl. *Transmission Control Protocol, TCP*). Umjesto tradicionalnog pristupa koji koristi pojedinačne HTTP zahtjeve i odgovore, *WebSocket* pruža stalnu i neprekidnu vezu između klijenta i poslužitelja, omogućavajući brzi prijenos podataka u stvarnom vremenu [28]. To čini *WebSocket* idealnim za aplikacije koje zahtijevaju ažuriranja u stvarnom vremenu poput chat aplikacija, igrama uživo, nadzoru u stvarnom vremenu, sinkronizaciji podataka i slično. Način na koji *WebSocket* komunikacija funkcionira prikazan je slikom 3.8. Klijent inicijalno šalje HTTP zahtjev prema poslužitelju koji uključuje posebno zaglavlje *Upgrade* s vrijednosti *websocket.* Ako poslužitelj podržava *WebSocket,* on odgovara s HTTP odgovorom koji potvrđuje uspješnu promjenu protokola na *WebSocket.* Ovaj postupak se još naziva uspostava veze ili rukovanje (engl. *handshake*). Nakon uspješnog rukovanja, klijent i poslužitelj imaju otvorenu *WebSocket* vezu. Ona im omogućuje da šalju podatke bez potrebe za ponovnim uspostavljanjem veze za svaku razmjenu. Poruke se šalju u obliku binarnih ili tekstualnih podataka i mogu sadržavati različite informacije ili naredbe. Kada klijent ili poslužitelj žele prekinuti vezu, oni mogu poslati poruku za zatvaranje. Nakon što obje strane razmjene takve poruke, veza se zatvara na pravilan način.

U aplikaciji se koristi *WebSocket* komunikacija kako bi se omogućilo upravljanje kretnjom robota. Odabir *WebSocket* protokola za ovu funkcionalnost ima najviše smisla jer se radi o radnji koja zahtjeva ažuriranje u stvarnom vremenu. Kroz *WebSocket* se šalju paketi koji sadrže naredbe za kretanje, odnosno vrijednosti parametara koji odražavaju položaj upravljačke gljivice. Te

parametre poslužitelj onda koristi za kontroliranje robota. WebSocket je u Flutter aplikaciji vrlo jednostavno implementirati putem *web_socket_channel* paketa. Ovaj paket pruža intuitivan API koji omogućuje jednostavno korištenje svih funkcionalnosti *WebSocket* veze. Kako bi paket bio integriran, potrebno je, kao i kod svih ostalih ovisnosti, prvo specificirati *web_socket_channel* ovisnost u *pubspec.yaml* datoteci. Unutar *web_socket_channel* paketa dostupna je *WebSocketChannel* klasa koja omogućuje kreiranje *WebSocket* konekcije, slanje i primanje podatkovnih paketa, te zatvaranje konekcije [30].



Sl. 3.8. Vizualizacija *WebSocket* komunikacije

4. RAZVOJ PROGRAMSKOG RJEŠENJA

U ovom dijelu diplomskog rada prikazana je primjena navedenih tehnologija u izradi konkretne Flutter aplikacije nazvane *UpraviBot*. Prvo su postavljeni zahtjevi na sustav i definirane funkcionalnosti koje aplikacija treba obavljati. Nakon toga, koristeći *Figma* alat, izrađen je prototip aplikacije koji je služio kao temelj za razvoj prvotne verzije aplikacije. U ovom poglavlju također su obuhvaćeni detalji implementacije specifičnih zahtjeva i funkcionalnosti, prikazujući relevantni programski kod i korisničko sučelje.

4.1. Zahtjevi na sustav

Definiranje zahtjeva na sustav ovisi o mogućnostima koje pruža poslužitelj s kojeg aplikacija dohvaća i šalje podatke. Glavni cilj ove aplikacije je učinkovito upravljanje robotskim sustavom, omogućujući korisnicima kontrolu nad različitim aspektima robotskih operacija i zadaća. Kako bi se postigao cilj, potrebno je prvo raščlaniti zahtjeve na manje cjeline koje precizno definiraju uvjete koji su neophodni za ispravno funkcioniranje sustava. Prema tome su definirani sljedeći zahtjevi:

- **Prijava korisnika** – kroz proces autentifikacije, aplikacija osigurava da samo ovlašteni korisnici imaju pristup informacijama o robotima ili obavljaju određene operacije nad robotima. Da bi se ispunio taj zahtjev, aplikacija treba implementirati ekran za prijavu korisnika. Na tom ekranu korisnik treba unijeti svoje korisničko ime i lozinku kako bi se autentificirao. Nakon uspješne provjere identiteta, korisniku se dodjeljuju odgovarajuće ovlasti i pristup resursima. Također, korisnik mora imati opciju odjave iz sustava.
- **Prikaz liste robota** – nakon uspješne autentifikacije, korisnik treba imati mogućnost pregleda liste svih robota kojima ima pristup. Svaki robot na listi treba biti prikazan svojim imenom i statusom dostupnosti. U slučaju da robot nije dostupan, dodatno treba naznačiti koji korisnik trenutno upravlja tim robotom.
- **Spajanje s robotom** – odabirom dostupnog robota korisnik treba imati mogućnost povezivanja s njime. Proces povezivanja s robotom zahtjeva unos odgovarajuće lozinke definirane za tog robota. Na taj način predstavljen je dodatan sigurnosni sloj u sprječavanju neovlaštenog pristupa. Također, korisnik mora imati opciju prekida veze sa povezanim robotom.

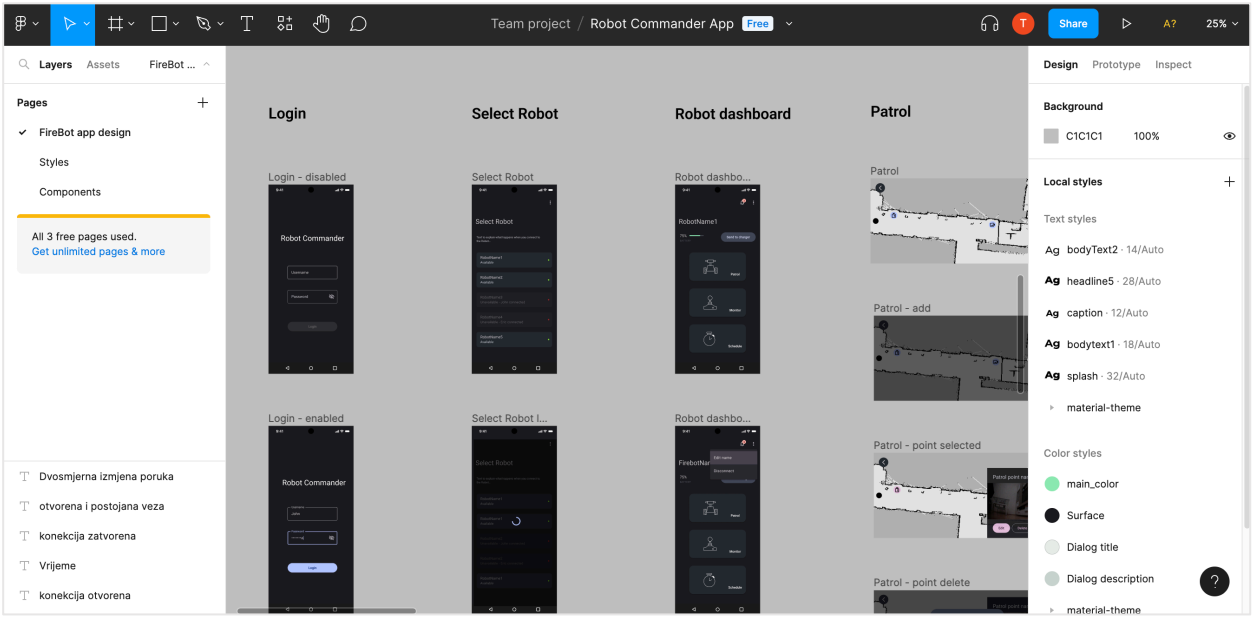
- **Prikaz detalja spojenog robota** – potrebno je na jednom mjestu prikazati podatke robota kao što su njegovo ime, status i stanje baterije, kako bi korisnici imali sveobuhvatni uvid u njegovo stanje i karakteristike.
- **Izmjena imena spojenog robota** – korisnik treba imati mogućnost prilagoditi ime robota prema vlastitim preferencijama ili kontekstu korištenja.
- **Prikaz LiDAR mape i upravljanje patrolnim točkama** – aplikacija treba omogućiti prikaz LiDAR mape okoline u kojoj robot trenutno radi te dodavanje, brisanje i izmjenu patrolnih točaka koje će robot obilaziti tijekom svog vremena rada.
- **Prikaz prijenosa uživo i sučelje za navigaciju** – korisnik treba imati mogućnost u aplikaciji vidjeti prijenos uživo s kamere robota na koji je spojen kako bi mogao pratiti događanja u stvarnom vremenu. Također, korisnik treba moći navigirati robota putem korisničkog sučelja.

Zahtjevi koji su gore navedeni odnose se na minimalno održiv proizvod (engl. *Minimum Viable Product*, MVP), odnosno prvotnu verziju aplikacije. Ti su zahtjevi osnovni i ključni za početno funkcioniranje aplikacije. U prvoj verziji neće biti uključeni zahtjevi poput omogućavanja pravljena rasporeda za patroliranje, primanja i prikaza notifikacija, slanje robota na punjenje, dodavanje barijera na kartu i slično. Iako su neke od tih funkcionalnosti uzete u obzir prilikom kreiranje dizajna aplikacije, njihova potpuna implementacija bit će provedena tek u 2.0 verziji aplikacije.

4.2. Dizajn korisničkog sučelja

Dizajn korisničkog sučelja ključan je za postizanje dobro strukturiranog i intuitivnog korisničkog iskustva (engl. *user experience, UX*). Razmišljanje o korisničkom iskustvu unaprijed omogućuje identifikaciju i rješavanje potencijalnih UX problema i prije same implementacije. Na taj način ubrzava se proces razvoja te osigurava konzistentnost u izgledu i funkcionalnostima aplikacije. Postoje brojni alati i uređivači za kreiranje korisničkog sučelja, a za potrebe ove aplikacije odabran je *Figma* alat jer pruža bogate značajke i mogućnosti za stvaranje kvalitetnog dizajna čak i u svojoj besplatnoj inačici. Svi elementi korisničkog sučelja integrirani su iz *Material Desing* kompleta koje je razvio Google kako bi pomogao dizajnerima u stvaranju dosljednih, atraktivnih i intuitivnih korisničkih sučelja. Nakon uvoza *Material Design* kompleta u *Figma* alat, moguće je kopirati i koristiti već gotove elemente korisničkog sučelja, kao što su gumbovi, kartice, ikone, dijalozi i tako dalje. Također, moguće je vrlo jednostavno promijeniti temu i tipografiju cijele aplikacije i onda ih izvesti iz *Figma* alata i uvesti u Flutter aplikaciju. Slika 4.1. prikazuje

skup ekrana aplikacije koji su kreirani unutar Figma uređivača. Ekrani postavljeni jedan ispod drugoga prikazuju stanja u kojima taj ekran može biti. Na taj način su obuhvaćeni različiti scenariji kao što je učitavanje, prikaz grešaka, unos teksta i slične operacije koje mogu rezultirati izmjenama na ekranu.



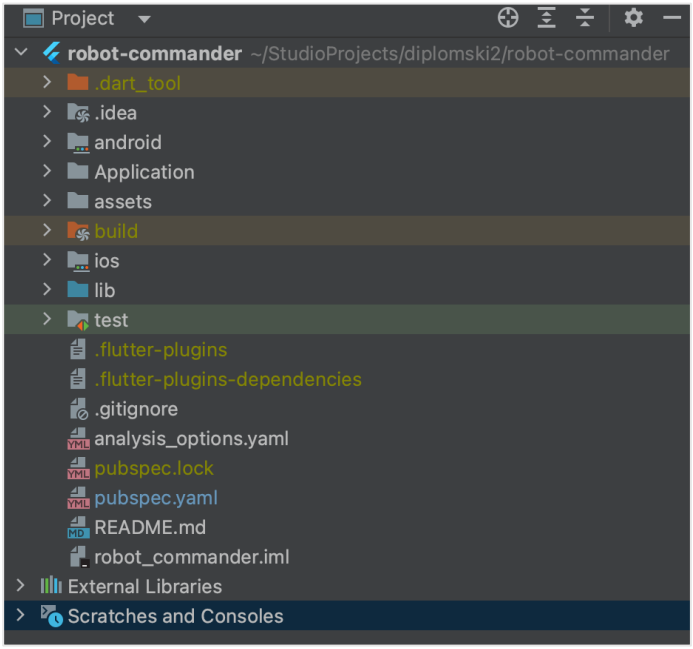
Sl. 4.1. Prikaz sučelja *Figma* alata

4.3. Struktura projekta

Prije nego što započne opisivanje implementacijskih detalja predodređenih zahtjeva potrebno je dati uvid u strukturu Flutter projekta i organizaciju mapa. Kreiranje Flutter projekta u razvojnom okruženju Android Studio je vrlo jednostavno putem čarobnjaka kroz kojeg se postavljaju osnovne informacije o projektu kao što su ime, lokacija, opis, tip i ostalo. Prilikom kreiranja projekta generirani su predefinirane mape i datoteke koji slijede određenu strukturu koja organizira različite komponente aplikacije (Slika 4.2.):

- ***android*** – sadrži Android – specifičan kod i uključuje datoteku *AndroidManifest.xml*, koja definira konfiguraciju i dozvole za Android aplikaciju. Također unutar ove mape nalaze se *.gradle* skripte za izgradnju te resursi specifični za Android platformu.
- ***ios*** – slično kao *android* mapa, sadrži iOS – specifičan kod i uključuje *Xcode* projekt datoteku koja se koristi za izgradnju i pokretanje iOS aplikacije. Unutar ove mape također se nalaze resursne i konfiguracijske datoteke te datoteka *AppDelegate.swift*, koja upravlja životnim ciklusom iOS aplikacije.

- ***lib*** – mjesto gdje se nalazi Dart kod Flutter aplikacije. Obično uključuje datoteku *main.dart*, koja služi kao ulazna točka Flutter aplikacije. Dart kod može biti organiziran u više datoteka i mapa, ovisno o složenosti i strukturi aplikacije.
- ***test*** – mapa u koju se smještaju testovi za Flutter aplikaciju. Prvenstveno je namijenjena za pisanje *widget*, jediničnih (engl. *unit*) i integracijskih testove.
- ***asssets*** – koristi se za pohranu statičkih datoteka, poput slika, fontova ili JSON[1] podataka, koji se ugrađuju u aplikaciju. Ove resurse najbolje je podijeliti u poddirektorije radi bolje organizacije.
- ***pubspec.yaml*** – predstavlja konfiguracijsku datoteku projekta napisanu u YAML[2] formatu. U njoj se navode ovisnosti potrebne za aplikaciju, uključujući Flutter SDK verziju, vanjske pakete i resurse. Također, u njoj je moguće specificirati meta podatke projekta, poput imena aplikacije, verzije i opisa.

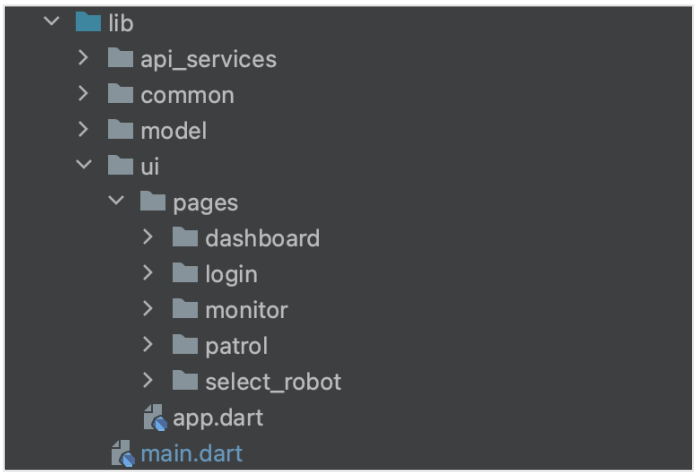


Sl. 4.2. Prikaz strukture paketa Flutter aplikacije unutar razvojnog okruženja Android studio

Osim opisa strukture glavnog projektnog direktorija potrebno je objasniti i strukturu unutar *lib* mape u kojoj se nalazi sav kod aplikacije. Zbog preglednosti programskog rješenja, sve datoteke, osim *main.dart* datoteke, organizirane su u nekoliko mapa. U mapi *api_service* se nalaze Dart klase namijenjen za komunikaciju s poslužiteljem preko HTTP metoda ili *WebSocket* veze. U mapi

[1] Standardni format za razmjenu strukturiranih podataka temeljen na principu *ključ-vrijednost*
[2] Čitljivi format za serijalizaciju podataka često korišten za konfiguracijske datoteke

c*ommon* se nalaze dijeljene komponente i klase koje se koriste u različitim dijelovima aplikacije. U mapi *model* se nalaze klase koje opisuju modele objekata domene, poput robota, LiDAR mape i patrolne točke. U mapi *ui* se nalaze komponente koje su povezane s korisničkim sučeljem, kao što je *app.dart* datoteka koja sadrži korijenski widget, temu i tipografiju aplikacije. Osim *app.dart* datoteke, tu se nalazi i mapa *pages* koja sadrži mape s widget objektima koji predstavljaju specifične ekrane i njihove pružatelje stanja. Na slici 4.3. vidljiva je, iznad opisana, struktura *lib* mape.



Sl. 4.3. Prikaz strukture *lib* mape

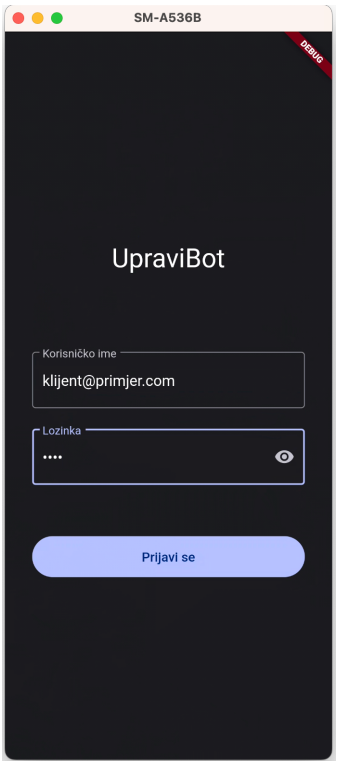
4.4. Implementacija zahtjeva na sustav

U narednih nekoliko poglavlja bit će dan sveobuhvatan pregled implementacijskih detalja zadanih zahtjeva. Prilikom izrade ove aplikacije fokus je bio ne samo na postizanju funkcionalnost, već i na održavanju kvalitete koda i pridržavanju principa čistog koda. Implementacija je vođena načelima čiste arhitekture s naglaskom na čitljivost, održivost i proširivost. Jedan od ključnih koraka u ovom procesu bilo je jasno razdvajanje različitih slojeva arhitekture aplikacije, kao što su komponente korisničkog sučelja od komponenti koje su zadužene za komunikaciju s poslužiteljem.

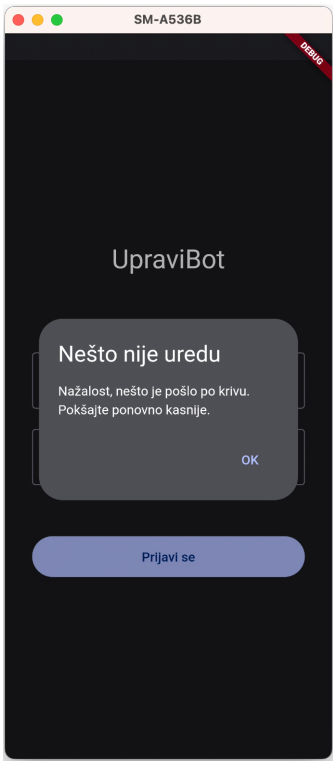
4.4.1. Prijava korisnika

Prilikom pokretanja aplikacije korisniku se pojavljuje zaslon za autentifikaciju (Slika 4.4.). Kako bi se uspješno autentificirao korisnik mora unijeti korisničko ime i lozinku s kojima je registriran u sustavu. U slučaju unosa krivog korisničkog imena ili lozinke na sredini ekrana prikazuje se dijaloški okvir koji obavještava o greški (Slika 4.5.). Također, dijaloški okvir se

prikazuje i ako dođe do mrežne pogreške. Dok se mrežna operacija prijave u sustav izvršava, ekran je blokiran i na njemu je prikazan element učitavanja.



Sl. 4.4. Prikaz ekrana za prijavu korisnika



Sl. 4.5. Prikaz dijaloškog okvira za grešku

Svaki ekran predstavljen je klasom koja nasljeđuje *ConsumerStatefulWidget,* odnosno widget objekt s promjenjivim stanjem. *ConsumerStatefulWidget* sadrži klasu stanja koja proširuje *ConsumerState* i prebrisuje (engl. *override*) *build* metodu koja vraća widget kojim je definiran izgled korisničkog sučelja. Također, svakom ekranu korijenski *widget* je *Scaffold,* kako bi se osigurala konzistencija i olakšalo slaganje ostalih UI komponenti. Komponente za unos korisničkog imena i lozinke implementirane su pomoću *TextFormField widget* objekta. *TextFormField* sprema tekst unosa u kontroler tipa *TextEditingController*, koji se predaje prilikom stvaranja. Nakon klika na gumb za prijavu, provjerava se vrijednost unosa iz kontrolera za oba polja, te se, u slučaju praznog unosa, prikazuje poruka ispod odgovarajućeg polja. Programski kod 4.1. prikazuje implementaciju gumba za prijavu pomoću *FilledButton widget* objekta i logike za provjeru unosa u kontrolerima prilikom poziva *onPressed* metode. Kontrolerima je moguće pristupiti iz bilo kojeg widget objekta u stabla jer su deklarirani kao konačne varijable unutar klase

stanja *_LoginPageState*[3]. Ako su unesene obje vrijednosti, poziva se metoda *login* iz klase *LoginNotifier* pomoću *ref* instance i *Riverpod* pružatelja *loginProvider*.

```
FilledButton(
    onPressed: () {
      _submitted = true;
      FocusScope.of(context).unfocus();
      if (_usernameController.text.isNotEmpty &&
          _passwordController.text.isNotEmpty) {
        ref.watch(loginProvider.notifier).login(
            _usernameController.text,
            _passwordController.text);
      }
    },
    child: Text('Prijavi se'))
```

Programski kod 4.1. Deklariranje gumba u Flutter radnom okviru

Klasa *LoginNotifier* proširuje klasu *StateNotifier* i koristi se kao upravitelj stanja ekrana za prijavu. Sadrži *enum*[4] klasu *LoginState* koja predstavlja različita stanja kao što su *početno*, *učitavanje*, *autentificirano* i *greška*. Također, *LoginNotifier* pruža metodu login koja pokreće proces prijave (programski kod 4.2.). Dok se pokušava izvršiti prijava na server, postavljeno je stanje *učitavanje*. Ako je prijava uspješna, stanje se ažurira na *autentificirano* koje zatim okida promjenu na sljedeći ekran. U slučaju stanja *greška* prikazuje se dijaloški okvir s porukom.

```
void login(String username, String password) async {
  state = LoginState.loading;
  try {
    await userService.login(username, password);
    state = LoginState.authenticated;
  } catch (err) {
    state = LoginState.error;
  }
}
```

Programski kod 4.2. *login* metoda unutar *LoginNotifier* klase

Login metoda poziva se nad *UserService* klasom koja pruža funkcionalnosti za upravljanje korisničkim operacijama kao što su prijava, odjava i dohvaćanje liste robota putem REST API komunikacije, uključujući upravljanje autentifikacijskim tokenima. Klasa *UserService* proširuje

[3] U Dart programskom jeziku donjom podvlakom se označava privatni identifikator
[4] Posebna klasa koja predstavlja skup definiranih konstantnih vrijednosti

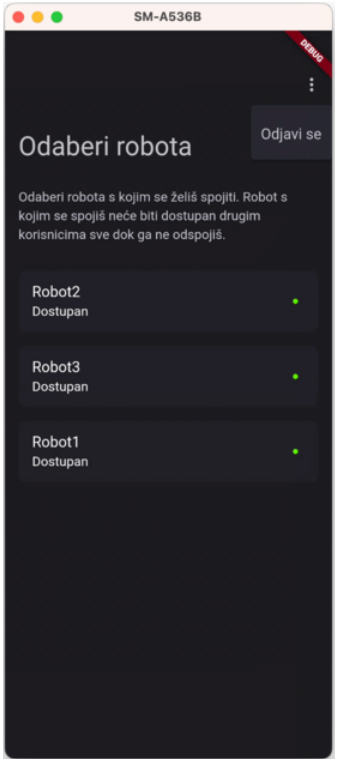
BaseApiService klasu, unutar koje se deklarira i konfigurira *Dio* objekt za slanje HTTP zahtjeva na server. U okviru *BaseApiService* klase je postavljen presretač (engl. *interceptor*) za HTTP odgovore koji provjerava istek autentifikacijskog tokena i automatski ga osvježava ako je istekao, kako bi korisnik ostao prijavljen u aplikaciji bez ručnog ponovnog prijavljivanja. Programski kod 4.3. prikazuje način na koji se osluškuje promjena stanja unutar *LoginNotifier* klase preko *loginProvider* pružatelja. Ako je stanje autentificirano onda se pomoću *Navigator* klase i metode *pushReplacement* mijenja trenutni ekran s ekranom za odabir robota. Ovaj dio programskog koda nalazi se na početku build metode *_LoginPageState*.

```
ref.listen(loginProvider, (previous, next) {
  if (next == LoginState.authenticated) {
    Navigator.pushReplacement(
      context,
      MaterialPageRoute(builder: (context) =>
       const SelectRobotPage()),
    );
  }
}
```

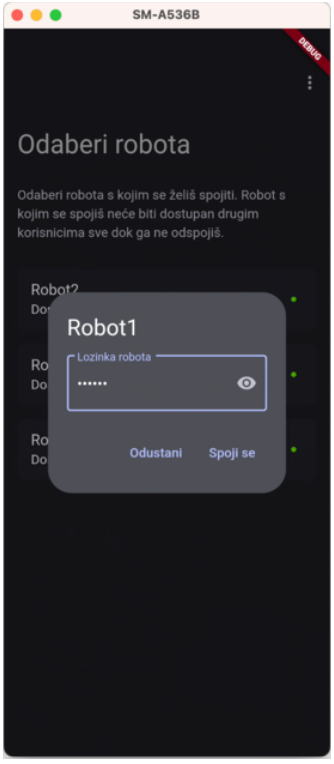
Programski kod 4.3. Osluškivanje izmjene stanja ekrana prijave

4.4.2. Prikaz liste robota

Nakon uspješne prijave korisnika, prikazuje se lista dostupnih robota s kojima je korisnik ovlašten upravljati (Slika 4.6.). Korisnik može odabrati samo jednog robota s kojim će se povezati, pri čemu su nedostupni roboti označeni crvenom točkicom i natpisom *nedostupan.* Robot je nedostupan ako je spojeni s drugim korisnikom. Na istom ekranu, korisniku je omogućena opcija za odjavu iz sustava. Da bi se spojio s robotom korisnik mora prvo unijeti lozinku za tog robota (Slika 4.7.). Klasa koja upravlja stanjima widget stabla ovog ekrana nazvana je *SelectRobotNotifier* i proširuje *StateNotifier* klasu. Ona sadrži instance klasa *UserService* i *RobotService*. Preko metode *getAllRobots* klase *UserService* dohvaćaju se svi roboti s kojima se korisnik može potencijalno spojiti. U programskom kodu 4.4. prikazano je tijelo metode *getAllRobots*, koja je asinkrona i ima povratni tip *Future<List<Robot>>*. Prvo se iz objekta *storage* tipa *FlutterSecureStorage* dohvaća korisnički token za pristup. Taj token je neophodan za uspješno izvršavanje HTTP zahtjeva. Zatim se konfigurira i pokreće HTTP zahtjev tipa GET s odgovarajućom putanjom i autorizacijskim tokenom. Nakon što se dobije odgovor u JSON formatu, podaci se mapiraju u listu objekata domene *Robot*.



Sl. 4.6. Prikaz ekrana s listom robota



Sl. 4.7. Prikaz dijaloškog okvira za upis lozinke robota

```
Future<List<Robot>> getAllRobots() async {
  String accessToken = await storage.read(key: 'ACCESS_TOKEN')??'';

  final Response response = await api.get(
    'client/robots',
    options: Options(headers: {
      'Authorization': "Bearer $accessToken",
    }),
  );

  return (response.data as List)
      .map((item) => RobotListItemResponse.fromJson(item))
      .map((item) => Robot(
          id: item.id ?? 0,
          username: item.username ?? "",
          battery: item.battery ?? 0,
          connectedUserName: item.connectedClient ?? ""))
      .toList();
}
```

Programski kod 4.4. Tijelo funkcije *getAllRobots* unutar *UserService* klase

4.4.3. Spajanje s robotom

Za spajanje s robotom odgovorna je *RobotService* klasa koja proširuje *BaseApiService*. Osim metode za spajanje, klasa također sadrži i druge metode za različite operacije nad robotom poput prekida veze, promjene imena i dohvaćanja podataka o robotu. Metoda *connectToRobot*, prikazana u programskom kodu 4.5, služi za povezivanje s robotom. Tijelo metode vrlo je slično tijelu funkcije *getAllRobots*. Prvo se dohvaća pristupni token, zatim se šalje HTTP POST zahtjev prema poslužitelju s korisničkim imenom robota i lozinkom. U zaglavlju zahtjeva šalje se autorizacijski token. Nakon uspješnog povezivanja, informacije o spojenom robotu se spremaju u pohranu kako bi se prilikom sljedećeg otvaranja aplikacije znalo da je korisnik prijavljen na određenog robota. Unutar BaseApiService klase prilikom konfiguracije *Dio* objekta definiran je osnovni dio URL-a pa je tako u HTTP zahtjevima potrebno samo specificirati dodatni dio putanje kao što je *robot/*.

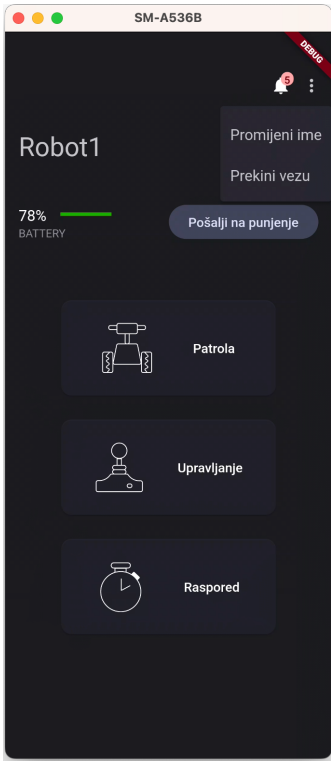
```
Future<void> connectToRobot(Robot robot, String password) async {
  String accessToken = await storage.read(key: "ACCESS_TOKEN")?? "";

  await api.post(
    'robot/',
    data: {
      "username": robot.username,
      "password": password,
    },
    options: Options(headers: {
      'Authorization': "Bearer $accessToken",
    }),
  );

  await storage.write(key: "ROBOT_ID", value: robot.id.toString());
  await storage.write(key: "ROBOT_NAME", value: robot.username);
}
```

Programski kod 4.5. Tijelo funkcije *connectToRobot* unutar *RobotService* klase

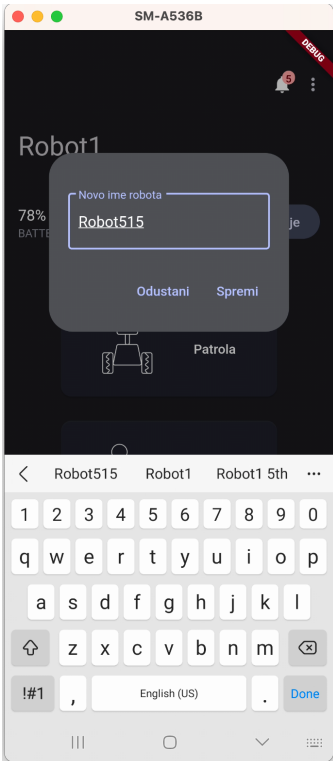
4.4.4. Prikaz podataka spojenog robota

Nakon spajanja s izabranim robotom prikazuje se nadzorna ploča (engl. *dashboard*) s imenom robota i statusom baterije. Također na nadzornoj ploči nalaze se ulazne točke za određene akcije kao što je postavljanje patrolnih točaka ili upravljanje robotom. Osim toga sadrži i izbornik s opcijama za mijenjanje imena robota i prekidanje veze s robotom, tj. od spajanje. Slika 4.8. prikazuje izgled nadzorne ploče, a slika 4.9. dijaloški okvir za promjenu imena robota.

Programski kod 4.6. prikazuje način na koji se rukuje prikazivanjem dijaloškog okvira za promjenu imena robota. Vidljivost *AlertDialog widget* objekta određuje se uvjetom *state.pageState == DashboardState.editName* na način da je je on omotan u *Visibility widget.* Sam *AlertDialog* sadrži naslov, sadržaj i akcije. Sadržajni dio se sastoji od *TextFormField* widgeta koji omogućuje korisniku unos novog imena robota. Akcije sadrže dva gumba, jedan za odustajanje od promijene imena, a drugi za spremanje novog imena. Zanimljivo je primijetiti kako se za naslov unutar *AlertDialog widget* objekta očekuje *widget* koji ne mora nužno biti *Text widget*, na taj način pružena je fleksibilnost u kreiranju widget objekata korisničkog sučelja.



Sl. 4.8. Prikaz nadzorne ploče robota



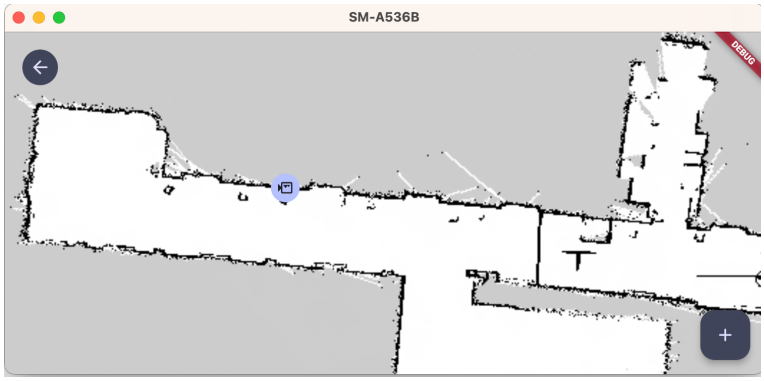
Sl. 4.9. Prikaz dijaloškog okvira za promjenu imena robota

```
Visibility(
    visible: state.pageState == DashboardState.editName,
    child: Stack(children: [
      SizedBox(…),
      AlertDialog(
        title: SizedBox(height: 8.0),
        content: TextFormField(
          controller: robotNameController,
          decoration: InputDecoration(
              label: Text("Novo ime robota"),
              errorText: submitted
                  ? getEmptyFieldError(robotNameController)
                  : null,
              border: OutlineInputBorder()),
          onChanged: (text) {
            submitted = false;
          },
        ),
        actions: […],
      ),
    ]))
```

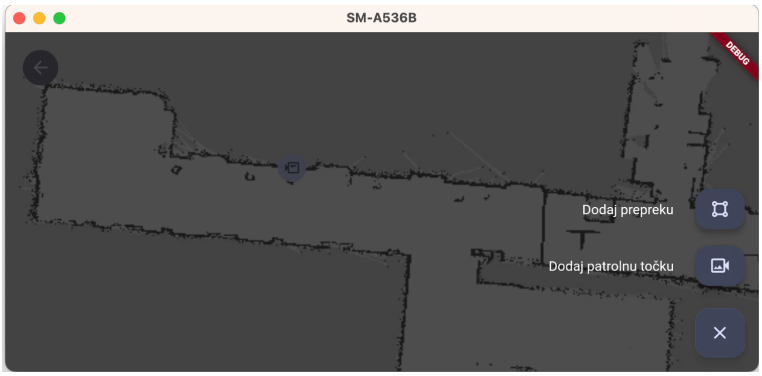
Programski kod 4.6. Prikaz kreiranja dijaloškog okvira za promjenu imena robota

4.4.5. Prikaz LiDAR mape i upravljanje patrolnim točkama

Klikom na *Patrola* karticu na nadzornoj ploči robota, prikazuje se ekran s LiDAR mapom, a njegova orijentacija postavljena je vodoravno radi preglednosti mape (Slika 4.10.). Na mapi su vidljive pozicije postojećih patrolnih točaka. Dužim klikom na patrolnu točku prikazuje se kartica s detaljima o toj točki i akcijama za ažuriranje ili brisanje patrolne točke. U donjem desnom kutu ekrana nalazi se plutajući gumb koji pruža opciju za dodavanje patrolne točke (Slika 4.11.). Pošto je ovaj ekran dodan na stog (engl. *stack*) ekrana, klikom na gumb sa strelicom u gornjem lijevom kutu prikazuje se prethodni ekran, što je nadzorna ploča.



Sl. 4.10. Prikaz LiDAR mape i postojeće patrolne točke



Sl. 4.11. Prikaz opcija plutajućeg gumba

Za dohvaćanje LiDAR mape zadužena je *LidarMapService* klasa koja je uz pomoć riverpod pružatelja injektirana u varijablu unutar *StateNotifier* klase zadužene za upravljanje stanja ovog ekrana. Programski kod 4.7. prikazuje kako se uz pomoć *Provider* funkcije iz *Riverpod* paketa kreira pružatelj instance *LidarMapService* klase te kako se uz pomoć *ref.watch* osluškuje njegova vrijednost i postavlja u varijablu *lidarMapService* unutar *PatrolNotifier* klase.

```
// lidar_map_service.dart
final lidarMapServiceProvider = Provider((ref)=> LidarMapService());


// patrol_provider.dart
class PatrolNotifier extends StateNotifier<PatrolStateWithData> {
  final AutoDisposeStateNotifierProviderRef ref;

  late final LidarMapService lidarMapService;
  late final PatrolPointService patrolPointService;

  PatrolNotifier(this.ref) : super(PatrolStateWithData.initial()) {
    lidarMapService = ref.watch(lidarMapServiceProvider);
    patrolPointService = ref.watch(patrolPointServiceProvider);
    getPatrolPointsState();
   }
...
}
```

Programski kod 4.7. Prikaz kreiranja pružatelja i osluškivanja njegove vrijednosti

Programskim kodom 4.8. izlistana je funkcija *getLidarMap* za dohvaćanje LiDAR mape robota koja se nalazi unutar *LidarMapService* klase. U putanji HTTP GET zahtjeva za dohvaćanje mape potrebno je specifirati ID robota za koji se dohvaća mapa. Nakon spajanja s robotom njegov ID je bio spremljen u storage pa ga je sada moguće dohvatiti od tamo. Dobiveni mrežni odgovor sadrži

informaciju o URL-u mape i njezinoj veličini. Te informacije koriste se za stvaranje objekta domene *LidarMap,* koji se vraća iz funkcije.

```
Future<LidarMap> getLidarMap() async {
  String robotId = await storage.read(key: "ROBOT_ID") ?? "";
  String accessToken = await storage.read(key: "ACCESS_TOKEN")?? "";

  Response response = await api.get(
    'maps/$robotId/',
    options: Options(headers: {
      'Authorization': "Bearer $accessToken",
    }),
  );

  return LidarMap(
      url:response.data["url"],
      size:Size.fromJson(response.data["size"])
      );
}
```

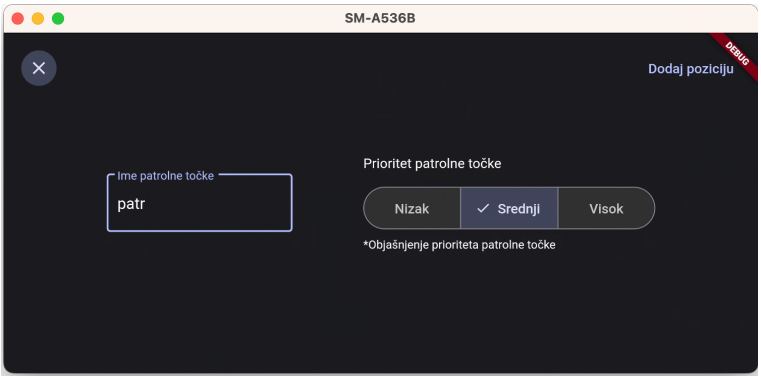
Programski kod 4.8. Tijelo funkcije *getLidarMap*

Za prikaz mape na korisničkom sučelju korišten je *Image.network widget* koji prikazuje sliku dohvaćenu s interneta. Pošto slika mape nije dostupna svima preko URL-a već joj samo imaju pristup ovlašteni korisnici, bilo je potrebno, osim izvora slike, postaviti i korisnikov token u zaglavlje zahtjeva radi pristupa. Kreiranje *Image.network widget* objekta prikazano je programskim kodom 4.9.

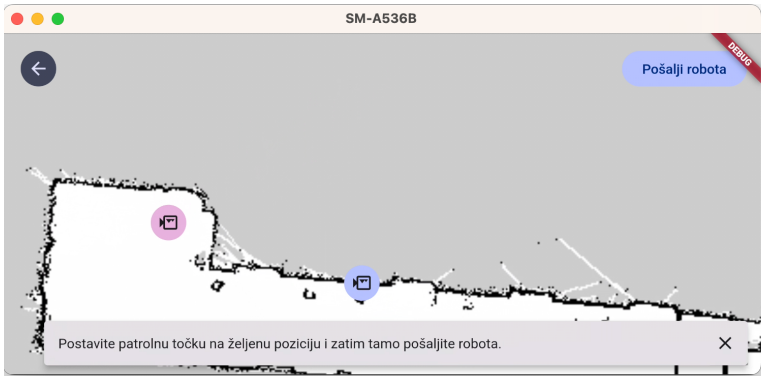
```
Image.network(
  state.lidarMap.url,
  headers: {'Authorization': "Bearer ${state.token}"},
  fit: BoxFit.cover,
  width: double.infinity,
  height: double.infinity,
)
```

Programski kod 4.9. Prikaz kreiranja *Image.netwrok widget* objekta

Dohvaćanje patrolnih točaka vrši se preko *PatrolPointService* klase i metode *getPatrolPoints*. Dobiveni JSON odgovor se mapira u listu objekata domene *PatrolPoint*. PatrolPoint klasa sadrži varijable id, ime, stupanj i poziciju koja sadrži vrijednosti x i y osi. Svaka patrolna točka na mapu je stavljena pomoću *Positioned widget* objekta. On prima x i y koordinate i prema njima pozicionira ikonu patrolne točke. Proces dodavanja nove patrolne točke sastoji se od nekoliko koraka, pri čemu niti jedan korak ne može biti preskočen. Prvi korak uključuje dodavanje imena i označavanje prioriteta patrolne točke (Slika 4.12.), što je važno prilikom kreiranja rasporeda patroliranja. Nakon toga, na mapi se otprilike označava pozicija patrolne točke, a zatim se koristi akcija *Pošalji robota* kako bi se robot uputio do te točke (Slika 4.13.). Nakon što je robot stigao do zadane pozicije, moguće je postavljanje točnih koordinata i prilagođavanje zakrivljenosti pan-tilt sustava robota putem odgovarajućih kontrolnih gljivica na korisničkom sučelju i gledanjem prijenosa uživo s kamere. Kada je sve namješteno, patrolnu točku se može spremiti klikom na gumb *Spremi točku*.



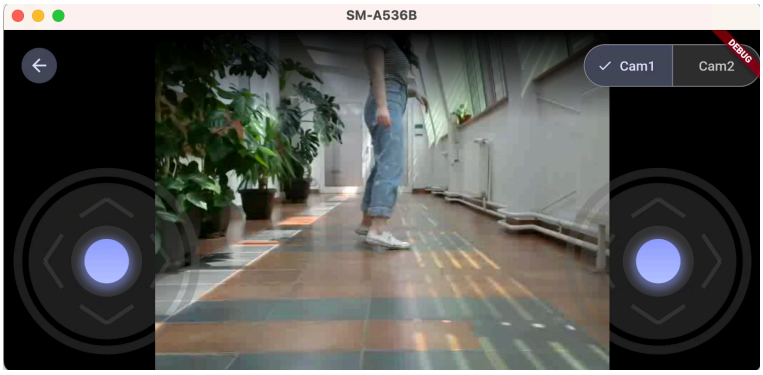
Sl. 4.12. Prikaz ekrana za dodavanje imena i prioriteta patrolne točke



Sl. 4.13. Prikaz ekrana za biranje pozicije patrolne točke

4.4.6. Prikaz prijenosa uživo i sučelje za navigaciju

Klikom na karticu *Upravljanje* otvara se zaslon za navigaciju robota, čiji je prikaz vidljiv na slici 4.14. Na ovom zaslonu moguće je pratiti prijenos uživo s robota, budući da je robot opremljen s dvije kamere, omogućena je i promjena izvora prikaza. Također zaslon sadrži i dvije upravljačke gljivice. Lijeva upravljačka gljivica namijenjena je za promjenu pozicije robota, dok je desna namijenjena za rotaciju te podizanje i spuštanje kamere na pan-tilt sustavu.



Sl. 4.14. Prikaz ekrana za upravljanje robotom

Za prikaz prijenosa uživo putem RTSP protokola, korišten je *VlcPlayer widget* iz *flutter_vlc_player* priključka. Prvo je potrebno stvoriti *VlcPlayerController* putem metode *network*, kojoj se pruža odgovarajući URL prijenosa. Nakon toga, taj se kontroler predaje *VlcPlayer widget* objektu kako bi on znao što treba prikazati na zaslonu. S obzirom da svaka kamera ima svoj URL prijenosa, prilikom klika na gumb za odabir određene kamera, primjenom metode *setMediaFromNetwork*, u kontroler se postavlja URL odabrane kamere. Ova promjena se odmah odražava ažuriranjem prikaza *VlcPlayer widget* objekta. Programski kod 4.10. prikazuje inicijalizaciju *VlcPlayerController* objekta i *VlcPlayer widget* objekta.

```
  _videoPlayerController = VlcPlayerController.network(
    _url,
    autoPlay: true,
  );

 VlcPlayer(
   controller: _videoPlayerController!,
   aspectRatio: screenSize.width / screenSize.height,
   placeholder: Center(child:  CircularProgressIndicator()),
)
```

Programski kod 4.10. Prikaz kreiranja *VlcPlayerController* objekta i *VlcPlayer widget* objekta

Programskim kodom 4.11. prikazan je način kreiranja virtualnih upravljačkih gljivica pomoću *Joystick widget* objekta iz *flutter_joystick* priključka. Ovaj widget prima nekoliko parametara, uključujući *stick, mode* i *listener.* Parametar *stick* postavljen je na instancu *CustomJoystickStick widget* objekta, koji defininira grafičke karakteristike gljivice kao što su radijus i boja gljivice. Parametar *mode* postavljen je na opciju *JoystickMode.all,* što ukazuje da se gljivicu može slobodno kretati u svim smjerovima. Parametar *listener* postavljen je na povratnu funkciju koja se poziva svaki put kad se promijeni položaj gljivice. Unutar povratne funkcije poziva se metoda nad *MonitorNotifier* objetom s parametrom *details*, koji sadrži informacije o trenutnom položaju gljivice.

```
Joystick(
  stick: CustomJoystickStick(),
  mode: JoystickMode.all,
  listener: (details) {
    ref
        .watch(monitorProvider.notifier)
        .updatePosition(details);
  },
)
```

Programski kod 4.11. Prikaz kreiranja gljivice za upravljanje pozicijom robota

MonitorNotifier proslijeđuje detalje o akcijama nad gljivicama klasi *NavigationWebSocket,* koja putem *WebSocket* komunikacije upravlja robotom. *NavigationWebSocket* klasa pruža potrebne metode za uspostavu *WebSocket* veze, slanje naredbi za kretanje robota i upravljanje pan-titl mehanizmom, te zatvaranje veze. Programski kod 4.12. prikazuje tijelo metode za uspostavu *WebSocket* veze koja se poziva prilikom inicijalizacije *MonitorNotifier* objekta. Prvo je potrebno stvoriti kanal veze pozivajući *WebSocketChannel.connect* metodu s URI-jem robota. Zatim je potrebno stvoriti vezu slanjem paketa za povezivanje. U njemu se šalju parametri robota dobiven iz *FlutterSecureStorage* objekta, uključujući ime robota, njegovu ip adresu i port. Programskim kodom 4.13. izlistana je metoda za slanje akcije mijenjanja pozicije robota putem *WebSocket* paketa. U paketu se šalje ime akcije, ime robota, upravljačke x i y komande i trenutno vrijeme kao identifikator zahtjeva. Radi oslobađanja resursa i ispravnog završavanja komunikacije potrebno je zatvoriti *WebSocket* vezu, a to se čini pozivanjem *_channel.sink.close().*

```dart
class NavigationWebSocket {
  final storage = const FlutterSecureStorage();

  late WebSocketChannel _channel;

  Future<void> createWebSocketChannel(String link) async {
    String robotName = await storage.read(key: 'ROBOT_NAME') ?? '';
    String robotIp = await storage.read(key: 'ROBOT_IP') ?? '';
    String robotPort = await storage.read(key: 'ROBOT_IP') ?? '';

    _channel = WebSocketChannel.connect(
      Uri.parse(link),
    );

    final payload = {
      "action": "robot_connect",
      "robot_name": robotName,
      "robot_ip": robotIp,
      "robot_port": robotPort,
      "request_id": DateTime.now().millisecondsSinceEpoch,
    };
    _channel.sink.add(jsonEncode(payload));
  }
…
```

Programski kod 4.12. *NavigationWebSocket* klasa s metodom za kreiranje *WebSocket* veze

```dart
Future<void> moveRobot(double x, double y) async {
  String robotName = await storage.read(key: 'ROBOT_NAME')  ?? '';
  final payload = {
    'action': 'move_robot_action',
    'robot_name': robotName,
    'cmd': {'x': x, 'y': y},
    'request_id': DateTime.now().millisecondsSinceEpoch,
  };
  _channel.sink.add(jsonEncode(payload));
}
```

Programski kod 4.13. Tijelo metode unutar *NavigationWebSocket* klase za mijenjanje pozicije robota

5. TESTIRANJE

U ovom poglavlju bit će prikazano testiranje i evaluacija razvijene aplikacije za upravljanje robotom. Svrha testiranja bila je procijeniti funkcionalnost, performanse i upotrebljivost aplikacije te osigurati da zadovoljava specificirane zahtjeve i pruža pouzdana i učinkovita rješenja za upravljanje robotom.

Glavni ciljevi faze testiranja bili su sljedeći:

- Verifikacija ispravnog funkcioniranja ključnih značajki, uključujući prikaz uživo, upravljanje robota gljivicama i dodavanje patrolnih točaka.
- Procjena kompatibilnosti aplikacije s različitim uređajima i operacijskim sustavima.
- Evaluacija performansi aplikacije u različitim scenarijima.
- Provjera odzivnosti, stabilnosti i pouzdanosti aplikacije.

Metodologije korištene prilikom testiranja su sljedeće:

- Testiranje jedinica: Pojedini dijelovi i funkcionalnosti aplikacije testirani su izolirano kako bi se potvrdila njihova ispravnost.
- Testiranje integracije: Integracija različitih modula aplikacije i vanjskih ovisnosti, poput *WebSocket* veze i *VlcPlayer* biblioteke, temeljito je testirana kako bi se osigurala besprijekorna komunikacija i kompatibilnost.
- Funkcionalno testiranje: Ključne značajke poput prikaza uživo, upravljanja robota gljivicama, promjene kamere i kretanja robota testirane su prema unaprijed definiranim testnim slučajevima kako bi se potvrdilo njihovo ispravno funkcioniranje.
- Testiranje performansi: Performanse aplikacije i njena odzivnost evaluirane su simuliranjem različitih scenarija, uključujući veliku mrežnu prometnost.
- Testiranje kompatibilnosti: Aplikacija je testirana na različitim uređajima, veličinama zaslona i operacijskim sustavima kako bi se osigurala kompatibilnost na više platformi.

Faza testiranja donijela je uspješne rezultate, što dokazuje učinkovitost i pouzdanost aplikacije za upravljanje patrolnim robotom. Aplikacija je uspješno prošla testne scenarije i zadovoljila definirane zahtjeve. Ključni zaključci iz faze testiranja uključuju:

- Sve ključne značajke poput prikaza uživo, upravljanja robota gljivicama i promjene kamere funkcionirale su prema očekivanjima, pružajući besprijekorno korisničko iskustvo.

- Aplikacija je pokazala odlične performanse s brzim vremenom odziva i minimalnim kašnjenjem tijekom reprodukcije prijenosa uživo i upravljanja robotom.
- Testiranje kompatibilnosti otkrilo je da je aplikacija kompatibilna s različitim uređajima i operacijskim sustavima, što osigurava široku bazu korisnika.

Na temelju uspješnog testiranja i pozitivnih rezultata može se zaključiti da je razvijena aplikacija za upravljanje patrolnim robotom ispunila svoje ciljeve i zadovoljila zahtjeve. Aplikacija se pokazala pouzdanom, učinkovitom i jednostavnom za korištenje, pružajući učinkovito rješenje za upravljanje i nadzor patrolnog robota u stvarnom vremenu.

6. ZAKLJUČAK

U ovom diplomskom radu razvijena je mobilna aplikacija za upravljanje robotskim sustavom koja omogućuje korisnicima daljinsko upravljanje i nadzor robota. Korištene tehnologije poput Flutter radnog okvira, *Riverpod* biblioteke i *Dio* biblioteke pružile su osnovu za razvoj aplikacije i omogućile efikasno upravljanje stanjem aplikacije, izvršavanje HTTP zahtjeva i komunikaciju s API-jem. Aplikacija omogućuje korisnicima pristup robotima i njihovim funkcionalnostima putem mobilnih uređaja, pružajući udobnost i fleksibilnost u upravljanju robotima. Integracija RTSP protokola i alata za prijenos uživo omogućuje korisnicima praćenje videa s kamera robota u stvarnom vremenu, a za uspješno navigiranje robota zadužena je *WebSocket* veza.

Aplikacija je postigla svoj osnovni cilj omogućavanjem korisnicima pristup robotima i njihovim funkcionalnostima na jednostavan i fleksibilan način. Međutim, postoji potencijal za daljnje unaprjeđenje aplikacije. Jedan od mogućih unaprjeđenja je implementacija funkcionalnosti za pravljenje rasporeda za patroliranje. Ovo bi omogućilo korisnicima da programiraju rutu i vrijeme patroliranja robota putem mobilne aplikacije, čime bi se olakšalo automatizirano obavljanje zadataka. Također, integracija notifikacija bi korisnicima omogućila primanje obavijesti o bitnim događajima ili statusima robota, što bi unaprijedilo praćenje i nadzor. Ove nadogradnje bi dodale autonomiju i prilagodljivost robotskom sustavu, te bi korisnicima omogućile veću kontrolu i učinkovitost.

Razvijena mobilna aplikacija je podvrgnuta temeljitom testiranju koje je potvrdilo njezinu sposobnost u daljinskom upravljanju i nadzoru robotskih sustava putem mobilnih aplikacija. Testiranje je pokazalo zadovoljavajuće rezultate u pogledu funkcionalnosti i performansi aplikacije. U cilju daljnjeg unaprjeđenja, moguće je dodavanje novih funkcionalnosti koje bi pridonijele praktičnosti i učinkovitosti za korisnike. Također, takve nadogradnje bi omogućile kontinuirani razvoj i istraživanje u području upravljanja robotskim sustavima putem mobilnog uređaja.

LITERATURA

[1] Hrvatska tehnička enciklopedija, mrežno izdanje, Leksikografski zavod Miroslav Krleža, 2021., https://enciklopedija.hr/natuknica.aspx?ID=53102 (pristupljeno 24. svibnja 2023.).

[2] C. Müller, „Executive Summary World Robotics 2022 - Industrial Robots“, Statistički odjel Međunarodne Federacije Robotike, VDMA Services GmbH, Frankfurt am Main, Njemačka 2022.

[3] C. Müller, B. Graf i K. Pfeiffer, „Executive Summary World Robotics 2022 - Service Robots“, Statistički odjel Međunarodne Federacije Robotike, VDMA Services GmbH - Frankfurt am Main, Njemačka, 2022.

[4] kuka, https://www.kuka.com/en-de/products/robot-systems/industrial-robots (pristupljeno 10. lipnja 2023.).

[5] P. Dong, Q. Chen, „LiDAR Remote Sensing and Applications“, CRC PRESS - Taylor & Francis Group, Florida, 2018.

[6] G. Bienasz, „Autonomous Delivery Robots Are Heading For The University of Texas at Austin Campus“, Entrepreneuer, https://www.entrepreneur.com/news-and-trends/dog-like-robots-to-make-deliveries-around-ut-austin-campus/437628 (pristupljeno 26. svibnja 2023.).

[7] M. Oitzman, „Knightscope robots to patrol Fortune 1000 hotel properties“, Mobile Robot Guide, https://mobilerobotguide.com/2022/02/08/knightscope-robots-to-patrol-fortune-1000-hotel-properties/ (pristupljeno 26. svibnja 2023.).

[8] R. Salsman, „SMP Robotics and Team 1st: Utilizing Swarm Intelligence In Security Robots“, LinkedIn, https://www.linkedin.com/pulse/smp-robotics-team-1st-utilizing-swarm-intelligence-security-salsman/ (pristupljeno 26. svibnja 2023.).

[9] T. Livingstone, „Singapore using robots to enforce social distancing“, 9news, https://www.9news.com.au/world/coronavirus-or3-robot-used-in-singapore-to-enforce-social-distancing/21c6ea5f-7f27-4cf9-b358-0d9072e8e8f4 (pristupljeno 26. svibnja 2023.).

[10] R. R. Murphy, „Introduction to AI Robotics“, The MIT Press, Cambridge, 2019.

[11] „Doing more with Spot“, bostondynamics, https://www.bostondynamics.com/resources/blog/doing-more-spot (pristupljeno 10. lipnja 2023.).

[12] „Now Roombas check in with 'Clean map' reports to your phone“, engadget, https://www.engadget.com/2017-03-15-roomba-clean-map-reports-alexa.html (pristupljeno 10. lipnja 2023.)

[13] „Mobile Operating System Market Share Worlwide“, statcounter, https://gs.statcounter.com/os-market-share/mobile/worldwide (pristupljeno 29. svibnja 2023.).

[14] W. Leler, „What's Revolutionary about Flutter“, hackernoon, https://hackernoon.com/whats-revolutionary-about-flutter-946915b09514 (pristupljeno 29. svibnja 2023.).

[15] ANALITIKE

[16] „Flutter FAQ“, Flutter, https://docs.flutter.dev/resources/faq (pristupljeno 29. svibnja 2023.).

[17] GitHub, https://github.com/flutter/flutter (pristupljeno 29. svibnja 2023.).

[18] M. Katz, „Flutter Apprentice - Section I: Build your first Flutter App“ , mrežno izdanje, Kodeco, 2022., https://www.kodeco.com/books/flutter-apprentice/v3.0/chapters/1-getting-started (pristupljeno 29. svibnja 2023.).

[19] E. Windmill, „Flutter in Action“, Manning Publications, Sjedinjene Američke Države, 2020.

[20] N. S. Monnankulama, „Flutter - main.dart & Widgets“, medium, 2021., https://medium.com/tech-learn-share/flutter-main-dart-widgets-e2c0083f3b5e (pristupljeno 29. svibnja 2023.).

[21] „Dart Overview“, dart, https://dart.dev/overview (pristupljeno 11. lipnja 2023.).

[22] Flutter performance profiling, flutter, https://docs.flutter.dev/perf/ui-performance (pristupljeno 11. lipnja 2023.).

[23] „Meet Android Studio“, android developers, https://developer.android.com/studio/intro?gclid=CjwKCAjwkLCkBhA9EiwAka9QRgU7xoMPE-p2ZuRyWLzmmGCk6mcGmYDKFLXtXrB-kR8ciRQie1sHhhoCksAQAvD_BwE&gclsrc=aw.ds (pristupljeno 16. lipnja 2023.).

[24] „Providers“, riverpod, https://riverpod.dev/docs/concepts/providers (pristupljeno 17. lipnja 2023.).

[25] „Dio“, pub dev, https://pub.dev/packages/dio (pristupljeno 17. lipnja 2023.).

[26] H. Schulzrinne, A. Rao, R. Lanphier, „Real Time Streaming Protocol (RTSP)“, The Internet Society, 1998., RFC, https://www.rfc-editor.org/rfc/rfc2326.html (prisupljeno 17. lipnja 2023.).

[27] „VLC Player Plugin“, pub dev, https://pub.dev/packages/flutter_vlc_player (pristupljeno 17. lipnja 2023.).

[28] I. Fette, A. Melnikov, „The WebSocket Protocol“, Internet Engineering Task Force, 2011., RFC, https://www.rfc-editor.org/rfc/rfc6455.html (pristupljeno 17. lipnja 2023.).

[29] „Work with WebSockets“, flutter, https://docs.flutter.dev/cookbook/networking/web-sockets (pristupljeno 17. lipnja 2023.)

SAŽETAK

Mobilne tehnologije su donijele revoluciju u načinu interaktivne komunikacije s različitim sustavima i uređajima, uključujući i upravljanje i kontrolu robotskih sustava putem mobilnih aplikacija. U ovom radu prikazan je razvoj višeplatformske mobilne aplikacije za daljinsko upravljanje i nadzor robotskog sustava, izrađene pomoću Flutter radnog okvira. Cilj aplikacije je pružiti korisnicima jednostavan pristup robotima i njihovim funkcionalnostima putem mobilnih uređaja. Ključne značajke aplikacije uključuju autentifikaciju, prikaz liste robota, povezivanje s robotom, prikaz podataka u stvarnom vremenu, vizualizaciju LiDAR mape, prijenos uživo s kamera robota i navigaciju robota. Ove funkcionalnosti omogućuju korisnicima daljinsko upravljanje i nadzor robota, olakšavajući zadatke poput patroliranja i nadzora. Uzimajući u obzir povratne informacije korisnika i praćenje novih trendova, moguće je proširiti funkcionalnosti aplikacije te pružiti još veću vrijednost i poboljšano korisničko iskustvo. Iskorištavanje Flutter radnog okvira i pripadajućih tehnologija pokazuje potencijal za inovacije u području mobilnih aplikacija za robotiku.

Ključne riječi: Flutter radni okvir, mobilna aplikacija, nadzor u stvarnom vremenu, robotski sustavi

ABSTRACT

Title: Managing robotic systems using mobile devices

Mobile technologies have revolutionized the way we interact with various systems and devices, including the management and control of robotic systems through mobile applications. This paper presents the development of a cross-platform mobile application for remote control and monitoring of a robotic system, built using the Flutter framework. The aim of the application is to provide users with easy access to robots and their functionalities via mobile devices. Key features of the application include authentication, robot list display, robot connection, real-time data visualization, LiDAR map visualization, live streaming from robot cameras, and robot navigation. These functionalities enable users to remotely control and monitor robots, facilitating tasks such as patrolling and surveillance. By considering user feedback and monitoring emerging trends, it is possible to expand the application's functionalities and provide even greater value and enhanced user experience. Leveraging the Flutter framework and its associated technologies demonstrates the potential for innovation in the field of mobile applications for robotics.

Key words: Flutter framework, mobile application, real-time monitoring, robotic systems

ŽIVOTOPIS

Terezija Umiljanović rođena je 26. lipnja 1997. godine u Našicama, gdje je pohađala Osnovnu Školu Dore Pejačević i Srednju Školu Isidora Kršnjavog. Nakon završene opće gimnazije upisuje preddiplomski studij na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek, smjer Računarstvo. Uz drugu godinu pohađa i završava Android Akademiju što joj omogućava prasku i zaposlenje u informatičkoj tvrtci Plava tvornica. Na trećoj godini preddiplomskog studija odlučuje se za odlazak na Erasmus praksu u Valenciju, Španjolska, u trajanju od 5. mjeseci. Tamo radi kao Android developer u jednom startupu. Nakon prakse upisuje diplomski sveučilišni studij na FERIT-u, smjer Programsko inženjerstvo. U međuvremenu se zapošljava u informatičkoj tvrtci Infinum, gdje i danas radi kao Android developer.

_____

Potpis autora

PRILOZI

Programski kod je priložen na CD-u uz ovaj rad.