RAZVOJ IoT MOBILNE APLIKACIJE

Jeličić, Ivo

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE
STRUČNI DIPLOMSKI STUDIJ ELEKTROTEHNIKA

IVO JELIČIĆ

ZAVRŠNI RAD

RAZVOJ IOT MOBILNE APLIKACIJE

Split, ožujak 2024.

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE
STRUČNI DIPLOMSKI STUDIJ ELEKTROTEHNIKA

Predmet: Senzorske mreže

ZAVRŠNI RAD

Kandidat: Ivo Jeličić

Naslov rada: Razvoj IoT mobilne aplikacije

Mentor: Marko Meštrović, mag. ing. el.

Split, ožujak 2024.

SADRŽAJ

SAŽETAK

Razvoj IoT mobilne aplikacije

U današnjem svijetu tehnološki napredak ima sve veći utjecaj na naš način života i poslovanja. S razvojem tehnologija poput Interneta stvari (IoT), vidimo transformaciju načina interakcije s okolinom. IoT se odnosi na mrežu fizičkih uređaja, vozila, kućanskih aparata i drugih objekata koji su opremljeni s tehnologijom koja omogućava prikupljanje i razmjenu podataka.IoT otvara put za razvoj novih poslovnih modela i usluga. Na primjer, u području zdravstva, IoT omogućuje praćenje zdravstvenih podataka pacijenata na daljinu, što može poboljšati dijagnostiku i praćenje bolesti. U prometu, pametni sustavi upravljanja prometom mogu optimizirati protok vozila i smanjiti gužve, u industriji IoT omogućuje praćenje i upravljanje strojevima i procesima na daljinu, što može rezultirati povećanom produktivnošću i smanjenjem troškova održavanja u automatizaciji. U domaćinstvima, pametni uređaji povezani putem IoT-a omogućuju udobnije, sigurnije i energetski učinkovitije okruženje.

Važno je napomenuti da s napretkom dolaze i izazovi, poput pitanja sigurnosti podataka i privatnosti. Potrebno je osigurati da se podaci prikupljeni putem IoT uređaja koriste na odgovoran i siguran način kako bi se zaštitila privatnost korisnika.

Ukratko, tehnologije poput Interneta stvari donose mnoge prednosti u naš svakodnevni život i poslovanje, omogućujući nam da budemo povezaniji, učinkovitiji i pametniji u načinu na koji koristimo resurse i upravljamo okolinom.

Ključne riječi: Internet of Things, tehnologija, napredak, automatizacija, senzori

SUMMARY

IoT mobile application development

In today's world, technological advancement is exerting an increasingly significant impact on our way of life and business operations. With the development of technologies such as the Internet of Things (IoT), we are witnessing a transformation in how we interact with our environment. IoT refers to a network of physical devices, vehicles, household appliances, and other objects equipped with technology that enables data collection and exchange. IoT paves the way for the development of new business models and services. For example, in the healthcare sector, IoT enables remote monitoring of patients' health data, which can enhance diagnosis and disease tracking. In traffic management, smart traffic control systems can optimize vehicle flow and reduce congestion. In industry, IoT allows remote monitoring and management of machinery and processes, resulting in increased productivity and reduced maintenance costs in automation. In households, smart devices connected via IoT enable a more comfortable, secure, and energy-efficient environment.

It is important to note that with progress come challenges, such as data security and privacy concerns. It is necessary to ensure that data collected through IoT devices is used in a responsible and secure manner to protect users' privacy.

In summary, technologies like the Internet of Things bring many advantages to our everyday lives and business operations, enabling us to be more connected, efficient, and intelligent in how we use resources and manage our environment.

Keywords: Internet of Things, technology, progress, automation, sensors

1. UVOD

Konstantna želja za unapređenjem kvalitete života, društva i gospodarstva dovodi do potrebe za kontinuiranim razvojem tehnologije. Ovaj završni rad fokusiran je na razvoj IoT (Internet stvari) android aplikacije. IoT predstavlja ključni korak ka stvaranju pametnih sustava koji mijenjaju način života i rada, te obuhvaća umrežavanje uređaja koji su opremljeni senzorima i mogu komunicirati putem interneta, pružajući nove mogućnosti u područjima poput domaćinstava, industrije, zdravstva i poljoprivrede.

Razvoj IoT aplikacija otvara vrata poboljšanju efikasnosti i sigurnosti, ali također donosi izazove poput upravljanja velikim količinama podataka i osiguranja privatnosti. Važno je pristupiti ovoj tehnologiji holistički, uzimajući u obzir hardverske, softverske i mrežne aspekte.

U nastavku će se istražiti ključni koraci u razvoju IoT aplikacija, kako bi se stvorila pouzdana, skalabilna i učinkovita IoT rješenja. Unapređenje svakodnevnog života, i povezanost svijeta u kojem svaki uređaj doprinosi globalnoj mreži informacija i usluga je upravo cilj razvoja interneta stvari.

1.1. Zadatak diplomskog rada

Osnovni cilj ovog projekta jest implementirati IoT mobilnu aplikaciju pomoću objektno orijentiranog programiranja (OOP) u Android Studiu, pružajući istovremeno priliku za učenje Kotlin programskog jezika, koji predstavlja rastući trend u svijetu programiranja i često se koristi u Android razvoju. Arhitektonska struktura aplikacije bit će temeljena na Model-View-ViewModel (MVVM) obrascu, što omogućava organizaciju koda kroz jasno odvojene poslovne logike (ViewModel), korisničkog sučelja (View) te podataka (Model).

Razvojem ove aplikacije, naglasak će biti stavljen na pružanje pregleda meteoroloških podataka za različite gradove diljem svijeta kao i dohvaćanje podataka s lokalnog senzora u stvarnom vremenu. Kroz projekt će biti obuhvaćeni različiti ključni aspekti pristupa stvaranju IoT i Android aplikacija, postavljajući čvrstu osnovu za daljnji razvoj meteorološke aplikacije. Projekt će također poslužiti kao praktičan primjer kako iskoristiti koncepte OOP-a, MVVM arhitekture te Kotlin programskog jezika u integraciji s Android platformom.

2. INTERNET OF THINGS

Internet stvari (IoT) opisuje dinamičnu mrežu senzora sa procesorskom sposobnošću i softverom koji se povezuju i razmjenjuju podatke putem interneta ili drugih mreža, te imaju mogućnost samokonfiguriranja. Razvoj IoT-a rezultat je spajanja tehnologija poput računalstva, senzorike i ugrađenih (embedded) sustava, uključujući strojno učenje. Svi fizički i virtualni elementi ključno imaju jedinstvene identitete na mreži. Jedan od glavnih razloga raširenosti i razvoja IoT tehnologije je adaptivnost i dinamičnost, odnosno promjena načina rada ovisno o uvjetima Ovo područje obuhvaća elektroniku, komunikaciju i računalstvo. Iako se naziva "Internet stvari" često smatra pogrešnim jer uređaji ne moraju biti povezani s javnim internetom, već samo s mrežom, važno je da budu pojedinačno adresirani. Primjena IoT-a vidljiva je u "pametnim kućama" i zdravstvenim sustavima, logistici, industriji i slično. Unatoč rastućoj popularnosti, postoji zabrinutost zbog rizika u vezi s privatnošću i sigurnošću, što potiče razvoj standarda i regulatornih okvira kako bi se riješili ti izazovi.

2.1. Povijest IoT

Glavna ideja Interneta stvari leži u ugradnji mobilnih prijenosnika kratkog dometa u razne uređaje i svakodnevne potrepštine kako bi se omogućile nove oblike komunikacije između ljudi i stvari, te između samih stvari. Koncept izraza "Internet stvari" prvi su se puta pojavio 1985. godine. IoT predstavlja integraciju ljudi, procesa i tehnologije s povezivim uređajima i senzorima radi udaljenog praćenja, statusa, manipulacije i analize trendova. Sam izraz "Internet stvari" je iskorišten od strane Procter & Gamble, koji su smatrali radiofrekvencijsku identifikaciju (RFID) ključnom za IoT, omogućavajući računalima upravljanje svakom pojedinom stvari. A jedna od prvih pojava povezane mreže pametnih uređaja prvi se put pojavila već 1982., kada je modificirani Coca-Cola aparat na američkom sveučilištu postao prvi uređaj povezan na ARPANET čime je dobio mogućnost izvještavanja o inventaru i temperaturi pića.

2.1 Arhitektura IoT

Internet of Things se po svojoj arhitekturi može podijeliti u tri ključne razine: Uređaji, Internet of Things (IoT) framework te gateway i cloud. Dodatno vrijedi napomenuti da je bitno imati i neko sučelje preko kojeg se podatci daju prikazati (npr. mobilna aplikacija).

Uređaji obuhvaćaju povezane stvari poput senzora i aktuatora prisutnih u IoT opremi, posebno onih koji koriste protokole kao što su Modbus, Bluetooth ili vlastiti protokoli kako bi se povezali s Gateway-om. Ovi senzori obavljaju raznovrsno prikupljanje informacija, kao lokaciju, uvjete vremena i okoline, rad strojeva, podatke o ljudskom tijelu, te podatke o održavanju motora ili temeljnim zdravstvenim aspektima vozila.

Gateway do interneta za sve uređaje (senzore/aktuatore) s kojima vršimo interakciju. Gateway-i služe kao prijenosni medij između mreže odnosno čvorova uređaja i vanjske mreže kao npr World Wide Web.

Informacije iz okoline prikupljene senzorima se šalju putem gatewaya te spremaju i procesuiraju unutar cloud servera (u centrima za baze podataka). Ovako pripremljeni podatci su dalje korišteni da bi izvršavali „pametne" radnje. U cloudu se sve odluke i analitika odvija uzimajući u obzir UX (user experience) odnosno korištenje podatka mora biti jasno i ugodno za krajnjeg korisnika.

2.3. Slojevi IoT

IoT aplikacije koriste četiri sloja:

Fizički sloj koji predstavlja temelj arhitekture Interneta stvari, te je ključno osigurati da su uređaji sposobni za prikupljanje preciznih podataka, učinkovitu komunikaciju te da imaju dovoljno snage i procesnih sposobnosti. Fizički sloj i sloj mrežnog pristupa – PHY i MAC) su definirani protokoli koji određuju na koji način se podaci šalju preko fizičkog sloja. Fizički sloj npr: bakrene parice, koaksijalnim kabel ili radio valovi i sloj mrežnog pristupa (MAC sloj) definiraju protokole koji zadovoljavaju načine kodiranja paketa za slanje i pristup mreži. IoT može koristiti Ethernet, 3G,4G,5G, WiFi i vrlo često LoRaWAN (Long Range Wide Area Network).

Mrežni sloj omogućuje komunikaciju između uređaja i oblaka. Sloj IoT arhitekture koji obuhvaća operatore na strani poslužitelja koji povezuju uređaje s pametnim objektima, poslužiteljima i mrežnim uređajima. Mrežni sloj je zadužen za slanje podataka iz jedne u drugu mrežu pri čemu vrši adresiranje i usmjeravanje paketa. Najčešće se koriste IPv4 odnosno 32-bitne adrese i IPv6 odnosno 128-bitne adrese, a osim toga još postoji i protokol za uređaje male potrošnje i procesorske moći.

Transportni sloj arhitekture Interneta stvari djeluje kao most između mrežnog i aplikacijskog sloja, prevođenjem podataka s različitih uređaja i protokola u zajednički format koji mogu obraditi aplikacije, te omogućava komunikaciju između uređaja i sloja aplikacije. Različite komponente poput posrednika podataka, redova poruka i pretvarača protokola koji upravljaju obradom podataka, pohranom i sigurnošću su sve dio ovog sloja. U transportnom sloju se koristi TCP (Transmission Control Protocol) koji omogućava detekciju pogreške te ponovno slanje ukoliko je došlo do pogreške i tako osigurava pouzdan prijenos paketa podataka te je upravo stoga najšire primijenjen transportni protokol (koristi ga HTTP). UDP (User Datagram Protocol) je transakcijski orijentiran protokol koji ne garantira dostavu ni redoslijed prilikom prijenosa paketa ali izrazito koristan u „RealTime" aplikacijama gdje je prioritet da paketi ne kasne (npr. gaming i streaming).

Aplikacijski sloj arhitekture Interneta stvari obuhvaća aplikacije koje analiziraju podatke prikupljene s uređaja kako bi pružile informacije, automatizirale procese i unaprijedile proces donošenja odluka. Od jednostavnih pametnih telefon aplikacija za upravljanje kućnom automatizacijom do kompleksnih algoritama strojnog učenja optimiziranih za industrijske procese, ovaj sloj predstavlja ključnu vrijednost IoT-a. Važno je osigurati da su aplikacije na ovom sloju pouzdane, skalabilne i sigurne. Najraširenij primjer aplikacijskog sloja je HTTP (Hypertext Transfer Protocol). Ovaj protokol se temelji na principu request – response u kojem klijent šalje zahtjev serveru putem HTTP zahtjeva. Ovaj protokol je temelj World Wide Weba.

2.4. Problemi IoT ekosistema

Osnovni i najkritičniji problem IoT sustava je sigurnost. Mnogi IoT uređaji, zbog svojih računalnih ograničenja, suočavaju se s izazovima u implementaciji osnovnih sigurnosnih mjera, poput vatrozida i snažnog šifriranja, sve je to dodatno otežano rapidnom napredovanju tehnologije. Tehnički sigurnosni problemi, poput slabe autentifikacije, neenkriptiranih poruka i ranjivosti na napade SQL ubacivanja, namjerno injektiranje pogrešaka u IoT uređaje te „man in the middle" napada su vrlo česti. Osim toga, IoT uređaji s pristupom raznolikim podacima i kontrolom nad fizičkim uređajima izazivaju zabrinutost za privatnost jer neki uređaji potencijalno mogu špijunirati pojedince u njihovim domovima kao npr. zvona na ulaznim vratima s ugrađenim videokamerama.

Mogućnost pojave grešaka u aplikacijama, nepredviđenih interakcija ili kvarova uređaja koji mogu stvoriti nesigurna fizička stanja, poput otključavanja vrata ili narušavanje rada grijalica su pojava koju smo mogli uvelike vidjeti kod popularnih uređaja kao npr. Amazonova Alexa koji omogućavaju interakciju potaknutu podatcima senzora, korisničke unose ili vanjske okidače, te zapovijedati aktuatorima za različite oblike automatizacije. Senzori uključuju uređaje poput detektora dima i senzora pokreta, dok aktuatori obuhvaćaju pametne brave i utičnice. IotSan je sustav koji je dizajniran da spriječi takve pojave uvodeći levele interakcije.

Mnogi uređaji zahtijevaju neprekidno napajanje i internet povezanost kako bi pravilno funkcionirali. Kada dođe do prekida u jednom od tih elemenata, isto vrijedi i za uređaje i sve što je s njima povezano. S obzirom na duboku integraciju ovih uređaja u suvremeno poslovanje, nedostupnost može uzrokovati zastoje u radu. Kako bi se tvrtke pripremile za takve situacije, važno je razumjeti kako će prekidi utjecati na uređaje te proaktivno planirati mjere za otklanjanje problema i upravljanje incidentima. Različite strategije i postupci mogu pomoći u ublažavanju posljedica, dok je ključno osigurati da zaposlenici znaju kako djelovati u situacijama kada uređaji nisu dostupni.

Proizvođači IoT aplikacija suočavaju se s izazovom čišćenja, obrade i interpretacije velike količine podataka prikupljenih senzorima. Rješenje bi bilo da bežične senzorske mreže omogućuju dijeljenje podataka između senzorskih čvorova za distribuiranu analizu. Pohrana

velikih količina podataka predstavlja dodatni izazov s visokim zahtjevima koji proizlaze iz visokih potreba za prikupljanjem podataka. Problemi s podacima česti su u implementaciji IoT uređaja, a rješavanje tih izazova zahtijeva usklađivanje s načelima autonomije, transparentnosti i interoperabilnosti.

3. OBJEKTNO ORIJENTIRANO PROGRAMIRANJE

Način programiranja koji nazivamo objektno orijentirano programiranje (OOP) temelji se na konceptu objekata, koji mogu sadržavati podatke i kod. Podaci su obično organizirani u polja (poznati kao svojstva/atributi), dok se kod oblikuje kao skup procedura (metode). U okviru OOP-a, računalni programi su dizajnirani tako da se sastoje od objekata koji međusobno surađuju. Mnogi od najčešće korištenih programskih jezika kao što su C++, Java, Kotlin i Python podržavaju objektno orijentirano programiranje u različitim mjerama, često u kombinaciji s drugim vrstama programiranja.

Tijekom početka i sredine 1990-ih, objektno orijentirano programiranje postalo je dominantan način programiranja s pojavom široko dostupnih programskih jezika koji podržavaju te tehnike. Porastom popularnosti grafičkih korisničkih sučelja, koja se često oslanjaju na objektno orijentirane tehnike programiranja dominacija OOP dodatno je pojačana. OOP alati također su pridonijeli porastu popularnosti programiranja vođenog događajima.

Različiti izazovi u objektno orijentiranom dizajnu rješavaju se kroz nekoliko pristupa. Najčešći od njih je poznat kao obrazac dizajna. Taj izraz može se koristiti za opisivanje bilo kojeg ponovljivog uzorka rješenja problema koji se često pojavljuje u dizajnu softvera. Neki od tih problema često proizlaze s implikacijama i rješenjima specifičnim za objektno orijentirani razvoj.

Ne postoji stroga definicija što je točno to objektno orijentirano programiranje (OOP) te nije svaki jezik koji tvrdi da podržava OOP izravno povezan s tehnikama i strukturama karakterističnim za ovaj paradigmatski pristup. Ono što je jasno je da je to način programiranja koji se oslanja na korištenje objekata.

3.1. Objekti

Objekti u programiranju najčešće odražavaju entitete stvarnog svijeta. Ponekad objekti predstavljaju apstraktne entitete, poput objekta za zatvoren folder ili usluge koja pretvara Fahrenheite u Celzijuse. Pristup objektima sličan je varijablama, sa složenim unutarnjim strukturama; mnogi ih jezici tretiraju kao pokazivače koji služe kao reference na instancu u memorije. Procedure se nazivaju metodama, a varijable su poznate kao polja, članovi, atributi ili svojstva.

Objekti mogu obuhvatiti druge objekte kroz varijable instance, poznate kao objekti kompozicije. Na primjer, objekt Pas može sadržavati objekt Dlaka, koji koristi pokazivače ili izravno ugrađivanje, zajedno s vlastitim varijablama kao što su "dužina" i "boja". Kompozicija objekta simbolizira odnos, gdje svaki objekt Pas ima pristup lokaciji za pohranu objekta Dlaka.

Kritičari tvrde da OOP paradigma pretjerano naglašava dizajn softvera usmjeren na objekte, zanemarujući ključne aspekte kao što su računanje i algoritmi te da OOP jezici često prebacuju fokus sa struktura podataka i algoritama na tipove. Tendencija OOP-a da daje prednost imenicama nad glagolima, što stručnjaci smatraju čudno iskrivljenom perspektivom u usporedbi s funkcionalnim programiranjem. Još jedan fokus kritika za objektne sustave kao pretjerano pojednostavljene modele stvarnog svijeta, ističući problem za točnim modeliranjem vremena, posebno kako softverski sustavi postaju sve konkurentniji.

3.2. Nasljeđivanje

OOP jezici variraju u svojim specifičnostima, ali uobičajeno dopuštaju koncept nasljeđivanja kako bi se olakšala ponovna upotreba koda i proširivost kroz definiciju klasa ili prototipova. Raznoliki oblici nasljeđivanja često se koriste, ali unatoč značajnim razlikama među njima, koristi se analogna terminologija za definiranje ključnih pojmova poput objekta i instanciranja. U programiranju temeljenom na klasama, najčešćem stilu, svaki objekt se definira kao instanca određene klase. Klasa postavlja format podataka ili vrstu, te pruža dostupne postupke (metode klase ili funkcije članice) za određenu vrstu ili klasu objekta. Stvaranje objekata obavlja se

pozivanjem specifične metode u klasi, poznate kao konstruktor. Klase mogu nasljeđivati od drugih klasa, stvarajući hijerarhiju koja odražava odnose "je-a-tip-od". Na primjer, klasa Pas može naslijediti od klase Životinja. Svi podaci i metode dostupni u roditeljskoj klasi također su prisutni u podređenoj klasi s istim imenima. Na primjer, klasa životinja može definirati varijable "ime" i "veličina" te metodu "sortiraj_po_veličini()". Ove će karakteristike također biti dostupne u klasi Pas, koja može dodati varijable "starost" i "vrsta". Time se osigurava se da će sve instance klase Pas dijeliti iste atribute, poput imena, vrste i veličine.

Varijable klase pripadaju cijeloj klasi, dijeleći jednu kopiju među svim instancama klase, dok su varijable ili atributi instance specifični za pojedinačne objekte, svaki posjedujući svoju kopiju tih atributa. Pojam "varijable članice" odnosi se na zajedničke varijable klase i instance, definirane određenom klasom. Metode klase pripadaju cjelokupnoj klasi, pristupajući samo varijablama klase i ulazima procedure, dok metode instance pripadaju pojedinačnim objektima s pristupom varijablama instance, ulazima i varijablama klase.

U kontekstu nasljeđivanja, potklase mogu, ovisno o definiciji jezika, nadjačati metode superklasa, a višestruko nasljeđivanje je dopušteno u nekim jezicima, iako može stvarati složenosti.

U programiranju temeljenom na prototipu, objekti su osnovni entiteti, bez jasnih pojmova "klasa". Objekti se stvaraju na temelju postojećih objekata koji djeluju kao prototipovi, s mogućnošću da objekti imaju više roditelja ili samo jednog, ovisno o jeziku. Različiti objekti mogu dijeliti zajedničke osobine putem zajedničkog prototipa. U suprotnosti s programiranjem temeljenim na klasama, programiranje temeljeno na prototipu omogućava definiranje atributa i metoda koji nisu dijeljeni s drugim objektima. Doktrina sastava nad nasljeđivanjem zagovara implementaciju odnosa korištenjem sastava umjesto nasljeđivanja. Na primjer, umjesto nasljeđivanja od klase Životinja, klasa Pas može svakom objektu Pas dodijeliti interni objekt Životinja, koji zatim ima priliku sakriti od vanjskog koda, čak i ako klasa Životinja ima mnogo javnih atributa ili metoda.

3.3. Enkapsulacija i apstrakcija

Apstrakcija podataka je obrazac dizajna u kojem su podaci vidljivi samo funkcijama koje su semantički povezane, s ciljem sprječavanja pogrešne upotrebe. Uspješna primjena apstrakcije podataka često rezultira primjenom načela skrivanja podataka u objektno orijentiranom i čistom funkcionalnom programiranju. Slično tome, enkapsulacija sprečava vanjski kod da se direktno bavi internim radom objekta, olakšavajući refaktoriranje koda bez utjecaja na vanjski kod, pod uvjetom da "javni" pozivi metoda ostaju nepromijenjeni. Ova praksa također potiče organizaciju koda, potičući programere da grupiraju sve linije koda koji se odnose na određeni skup podataka u istu klasu, što olakšava razumijevanje drugim programerima. Enkapsulacija, kao tehnika, potiče na odvajanje.

U objektno orijentiranom programiranju, objekti djeluju kao sloj koji razdvaja unutarnji od vanjskog koda, implementirajući pritom apstrakciju i enkapsulaciju. Vanjski kod može koristiti objekt samo pozivajući određene metode instance s određenim skupom ulaznih parametara, čitanjem varijabli instance ili pisanjem u varijable instance. Ova metoda omogućuje stvaranje mnogo neovisnih instanci objekata koje mogu raditi neovisno. Iako tvrdnje da OOP paradigma poboljšava ponovnu upotrebu i modularnost koda nisu bez kritika, tvrdi se da ovaj pristup omogućuje intuitivnije modeliranje odnosa u stvarnom svijetu, koristeći objekte iz specifične aplikacijske domene umjesto tradicionalnih tablica baze podataka i potprograma.

Implementacija ograničenja pristupa, poput označavanja privatnih podataka ključnom riječi "private" u jezicima kao što su Java, te označavanje metoda s ključnom riječju "public", pruža dodatnu razinu skrivanja informacija. U nekim jezicima, poput C#, Swift i Kotlin, dodatna kontrola pristupa postiže se korištenjem ključnih riječi poput "internal". U drugima, kao što je Python, ovo se postiže konvencijom imenovanja (npr. metode koje počinju podvlakom smatraju se privatnima).

3.4. Rast popularnosti

Iako su C++, Java i Python među najšire korištenim jezicima s objektno orijentiranim značajkama, mnogi kritiziraju OOP paradigmu, preporučujući funkcionalno programiranje. Kritike sugeriraju da popularnost OOP-a u velikim tvrtkama proizlazi iz potrebe za rizikom ograničenim timovima.

Iako su jezici s OOP značajkama možda poboljšali modularnost, kritičari, tvrde da su postali popularni iz drugih razloga, noseći nepotreban teret složenosti. Studije produktivnosti ne pokazuju značajne razlike između OOP i proceduralnih pristupa. Objektno orijentirano programiranje postaje posebno popularno u dinamičkim jezicima poput Pythona, PowerShella, Rubyja i Groovyja. HTML, XHTML i XML dokumenti koriste objektni model povezan s JavaScriptom/ECMAScriptom, koji se temelji na prototipu umjesto nasljeđivanja klase.

4.ANDROID STUDIO

Android Studio je službeno integrirano razvojno okruženje (IDE) za Googleov operativni sustav Android, temeljeno na softveru IntelliJ IDEA tvrtke JetBrains i posebno dizajnirano za razvoj Android aplikacija. Dostupan je za preuzimanje na operativnim sustavima Windows, macOS i Linux. Android Studio je zamijenio Eclipse Android Development Tools (E-ADT) kao primarni IDE za izvorni razvoj Android aplikacija. Licenciran je pod licencom Apache, ali se isporučuje s nekim ažuriranjima SDK-a koja su pod drugačijom licencom, čime nije otvorenog koda.

Prva stabilna verzija objavljena je u prosincu 2014., počevši od verzije 1.0. Krajem 2015. Google je prestao podržavati Eclipse ADT, čime je Android Studio postao jedini službeno podržani IDE za Android razvoj.

Kotlin je 7. svibnja 2019. zamijenio Javu kao preferirani jezik za razvoj Android aplikacija od strane Googlea. Java i dalje ima podršku, kao i C++.
Značajke U trenutnoj stabilnoj verziji Android Studija dostupne su sljedeće značajke:
Podrška za izgradnju temeljena na Gradleu, Refactoring i brzim popravcima specifičnim za Android, integracija ProGuarda i mogućnosti potpisivanja aplikacija, template wizard temeljeni na predlošcima za stvaranje uobičajenih Android dizajna i komponenti, uređivač izgleda koji omogućuje povlačenje i ispuštanje komponenti korisničkog sučelja, s mogućnošću pregleda izgleda na više konfiguracija zaslona, podrška za izradu Android Wear aplikacija, lint alati za otkrivanje performansi, upotrebljivosti, kompatibilnosti verzija i drugih problema, ugrađena podrška za Google Cloud Platform, omogućujući integraciju s Firebase Cloud Messaging (prethodno poznat kao Google Cloud Messaging) i Google App EngineAndroid virtualni uređaj (emulator) za pokretanje i otklanjanje pogrešaka u aplikacijama u Android Studiju.
Nakon kompajliranja pomoću Android Studija, aplikacija se može objaviti u Trgovini Google Play uz uvjet da je u skladu s pravilima za razvojne programere Trgovine Google Play.

5. KOTLIN PROGRAMSKI JEZIK

Kotlin je statički tipiziran, objektno orijentiran programski jezik koji pokazuje interoperabilnost s Java virtualnim strojem (JVM), Java Class Libraries, te se često koristi u Android razvoju.

Kotlin je općenito koristan za razvoj mobilnih aplikacija za Android, ali također nalazi primjenu u razvoju na strani poslužitelja, kompletnom web razvoju (uključujući JavaScript za front-end), multiplatformskom mobilnom razvoju (uključujući iOS, watchOS i Linux), te zadacima znanosti o podacima poput izgradnje cjevovoda podataka i implementaciji modela strojnog učenja.

Organizacije koje koriste Kotlin za Android aplikacije mogu također iskoristiti te vještine za upravljanje resursima temeljenim na cloudu.

5.1. Prednosti i mane Kotlin programskog jezika

Prednosti Kotlina za razvoj Android aplikacija:

- Interoperabilnost:

Kotlin se može lako integrirati s Javom, koristeći isti bajt kod, te se može kompajlirati u JavaScript ili LLVM koder, omogućavajući pravovremeno kompiliranje i glatko funkcioniranje u drugim programima. Osim toga, dijeli alate s Javom, olakšavajući migraciju Java aplikacija u Kotlin.

- Sigurnost:

Kotlin je osmišljen kako bi izbjegao uobičajene pogreške kodiranja koje mogu uzrokovati probleme u kodu ili stvoriti ranjivosti. Jezik uključuje null sigurnost, eliminirajući greške izuzetaka null pokazivača.

- Jasnoća:

Kotlin smanjuje redundanciju u sintaksi popularnih jezika poput Jave, pružajući programerima sažetiji kod. Ovo štedi vrijeme programerima i povećava njihovu produktivnost.

- Podrška alata:

Kotlin podržava alate iz Android ekosustava, optimizirane za Android razvoj, uključujući Android Studio, Android KTX i Android SDK.

- Podrška zajednice:

Unatoč svojoj relativnoj novosti u usporedbi s Javom, Kotlin ima aktivnu zajednicu programera koja doprinosi poboljšanju jezika i pruža korisnu dokumentaciju.

Nedostaci Kotlina za razvoj Android aplikacija:

- Ograničeni resursi za učenje:

Iako se sve više programera prebacuje na Kotlin, dostupnost resursa za učenje ovog programskog jezika još uvijek nije dovoljna. Nedostaju alati i podrška za pitanja tijekom razvoja softvera.

- Usporena brzina kompilacije:

Iako Kotlin može biti brži od Jave u nekim situacijama, posebno pri inkrementalnim konstrukcijama, važno je napomenuti da se u slučaju potpune izgradnje Java uvijek može ponašati brže.

- Razlika u odnosu na Javu:

Unatoč nekim sličnostima, Kotlin i Java imaju značajne razlike. Nakon što se programeri mobilnih aplikacija potpuno posvete Kotlinu, prelazak na drugi programski jezik može predstavljati izazov.

- Manjak stručnjaka za Kotlin:

Iako je Kotlin iznimno relevantan, broj dostupnih programera u ovom području još uvijek nije dovoljan. Pronalaženje stručnjaka za Kotlin može biti izazovno, što čini zapošljavanje programera za Kotlin teže nego za neke druge jezike.

5.2. Kotlin i Java

Kotlin i Java su statički tipizirani programski jezici opće namjene, s Kotlinom često smatranim alternativom Javi. Iako nisu isti po sintaksi, Kotlin je interoperabilan s Java kodom te ima vlastite biblioteke za Android razvoj. Kotlin se ističe po jednostavnijoj i funkcionalnijoj sintaksi, čime olakšava učenje i smanjuje redundanciju u odnosu na Java-u.

Jedna od ključnih razlika između ova dva jezika je način instanciranja varijabli. Kotlin koristi manje ponavljanja i automatsko zaključivanje tipova, što rezultira kraćim i čišćim kodom. Npr. Kotlin koristi `val` za varijable čija se vrijednost ne mijenja.

Unatoč sličnostima, Kotlin se ističe svojom jednostavnom sintaksom. Na primjer, deklaracija funkcije i dodjela varijabli u Kotlinu zahtijeva manje redaka koda u usporedbi s Javom. Osim toga, Kotlin podržava fleksibilnost u dodjeli varijabli i korištenju operatora `var`. Usporedba sintakse pokazuje kako Kotlin može biti čitljiviji i manje redundantan, nudeći programerima efikasniji način izražavanja ideja. Sintaktičke razlike ogledaju se u primjerima koji koriste Kotlin kod za deklaraciju funkcije, dodjelu varijabli i definiciju klase.

Sve u svemu, Kotlin pruža modernu alternativu Javi s naglaskom na čistoći, funkcionalnosti i smanjenju suvišnog koda. Iz prethodnog se da zaključiti da i Kotlin i Java nude jedinstvene prednosti, s Kotlinom koji se posebno ističe u Android razvoju. Dok Kotlin donosi modernu sintaksu, smanjenje redundantnosti, brzu kompilaciju i podršku za suvremene koncepte programiranja, Java ostaje snažna opcija opće namjene s dugom poviješću, velikim ekosustavom i širom zajednicom programera.

Kotlin je popularan izbor za Android razvoj, a njegova upotreba raste. Ipak, Java i dalje igra ključnu ulogu u programiranju općenito. Konačna odluka o tome koji jezik odabrati ovisi o specifičnostima projekta, preferencijama razvojnog tima te potrebama i ciljevima aplikacije. Važno je uzeti u obzir i činjenicu da oba jezika imaju svoje mjesto u programerskom ekosustavu, a odluka između Kotlina i Jave često će ovisiti o kontekstu i okolnostima projekta.

6. SENZORI UNUTAR METEOROLOŠKIH STANICA

6.1. Meteorološka stanica

Meteorološka postaja predstavlja objekt na kopnu ili na moru, opremljen instrumentima i uređajima za precizna mjerenja atmosferskih uvjeta. Svojim funkcijama pridonosi prikupljanju podataka nužnih za prognozu vremena te istraživanje vremenskih i klimatskih fenomena. Među ključnim parametrima koje mjeri su temperatura, atmosferski tlak, vlažnost zraka, brzina vjetra, smjer vjetra te količina oborina. Mjerenja brzine vjetra provode se uz minimalne prepreke kako bi rezultati bili što precizniji. S druge strane, mjerenja temperature i vlažnosti zahtijevaju očuvanje od izravnog sunčevog zračenja ili insolacije kako bi rezultati bili pouzdani. Ručna promatranja obavljaju se najmanje jednom dnevno, dok su automatska mjerenja učestala, provode se barem jednom na sat. Uz kopnene postaje, i brodovi te plutače na moru sudjeluju u prikupljanju podataka o vremenskim uvjetima. Ova plovila bilježe različite meteorološke veličine, poput temperature površine mora, visine valova te perioda valova. Važno je napomenuti da plutajuće meteorološke plutače znatno nadmašuju svoje usidrene verzije u učinkovitosti pri prikupljanju podataka.

6.1.1. Vrste instrumenata unutar meteoroloških stanica

Termometar: Tradicionalni termometri koriste staklenu cijev sa živinim stupcem. Elektronički termometri koriste senzore osjetljive na promjene u električnom otporu ili naponu. Živin stupac raste ili pada ovisno o temperaturi, dok elektronički senzori pretvaraju promjene u električne signale. Neki moderni termometri omogućuju bežično praćenje podataka.

Barometar: Živin barometar sastoji se od cijevi ispunjene živinom koja reagira na promjene tlaka. Aneroidni barometri koriste metalne kutije koje se deformiraju pod utjecajem tlaka. Živin stupac ili deformacija aneroidne kutije daju očitanja atmosferskog tlaka. Moderni digitalni barometri koriste senzore za precizna mjerenja.

Higrometar: Kapilarni higrometri koriste tanku niti koja se steže ili opušta ovisno o vlažnosti. Elektrostatski i termalni higrometri mjere promjene električne provodljivosti ili toplinske vodljivosti zraka. Promjene u konfiguraciji, uzrokovane vlagom, prate se kako bi se odredila vlažnost zraka.

Anemometar: Tradicionalni anemometri koriste okretni set lopatica pričvršćen na vratilo. Ultrazvučni anemometri koriste ultrazvučne valove za mjerenje brzine vjetra. Okretanjem lopatica ili mjerenjem promjene u frekvenciji ultrazvučnih valova, anemometar daje očitanja brzine vjetra.

Piranometar: Sastoji se od crne površine koja apsorbira sunčevu svjetlost. Može imati termopar za precizna mjerenja. Mjerenje promjene temperature crne površine odražava količinu apsorbirane sunčeve energije.

Kišomjer: Često ima lijevak povezan s mjernom posudom. Elektronički senzori također se koriste za precizna mjerenja. Skupljanjem oborina u lijevku i mjerenjem razine u posudi, kišomjer daje podatke o količini padalina.

Vjetrokaz: Tradicionalni vjetrokazi imaju strelice koje pokazuju smjer vjetra. Elektronički vjetrokazi koriste senzore. Očitanja smjera vjetra bilježe se na temelju strelica ili elektroničkih senzora.

Posuda za isparavanje: Otvorena posuda s vodom postavljena na određenu visinu. Mjerenje promjene razine vode u posudi pruža podatke o isparavanju, što je važan parametar za vodni ciklus.

Na automatiziranim meteorološkim postajama zračnih luka, dodatni instrumenti uključuju senzor trenutnog vremena, disdrometar, transmisometar i ceilometar, koji koriste različite tehnologije za identifikaciju trenutnih vremenskih uvjeta, precizno mjerenje veličine i brzine kapi u padavinama, mjerenje izgubljene svjetlosti u atmosferi te određivanje visine stropa oblaka, čime pružaju sveobuhvatne podatke o atmosferskim uvjetima.

7.ESP 32 SoC, DHT22 I BMP180 SENZORI

7.1. ESP SoC

ESP32 je serija pristupačnih mikrokontrolera s integriranim Wi-Fi i dual-mode Bluetoothom, dizajniranih za male potrošnje energije. Ova serija koristi Tensilica Xtensa LX6 mikroprocesor u varijacijama s jednom ili dvije jezgre, Xtensa LX7 dvojezgreni mikroprocesor ili jednojezgreni RISC-V mikroprocesor. Osim toga, ESP32 uključuje ugrađene antenske prekidače, RF priključak, pojačalo snage, niskošumno prijamno pojačalo, filtre i module za upravljanje napajanjem. Razvila ju je kineska tvrtka Espressif Systems sa sjedištem u Šangaju, a proizvodi je TSMC koristeći njihov 40 nm proces. ESP32 je nasljednik mikrokontrolera ESP8266. Unutarnja memorija ESP32 uključuje 448 KB ROM-a za pokretanje i osnovne funkcije, 520 KB SRAM-a na čipu za podatke i upute, te 8 KB SRAM-a u RTC-u koji se naziva RTC FAST Memory i može se koristiti za pohranu podataka tijekom RTC pokretanja iz načina dubokog mirovanja, dok 8 KB SRAM-a u RTC-u, poznat kao RTC SLOW Memory, pristupa mu se putem ULP koprocesora tijekom načina dubokog sna. Također, ima 1 Kbit eFuse, od kojih se 256 bitova koristi za sustav (MAC adresa i konfiguracija čipa), a preostalih 768 bitova rezervirano je za korisničke aplikacije, uključujući flash enkripciju i ID čipa, te Flash ili PSRAM u pakiranju.

7.1.1. ROM i RAM memorija

ESP32 podržava više vanjskih QSPI flash memorija i vanjskih RAM (SRAM) čipova. Detaljnije informacije mogu se pronaći u Tehničkom priručniku ESP32. Također podržava hardversko šifriranje/dešifriranje temeljeno na AES-u kako bi zaštitio programe i podatke razvijatelja u flash memoriji. ESP32 može pristupiti vanjskoj QSPI flash memoriji i SRAM-u putem visokih brzih predmemorija. Do 16 MB vanjske flash memorije može biti mapirano u prostor instrukcijske memorije CPU-a i prostor memorije samo za čitanje istovremeno. Kada je vanjska flash memorija mapirana u prostor instrukcijske memorije CPU-a, do 11 MB + 248 KB može biti mapirano istovremeno, pri čemu će se smanjiti performanse predmemorije zbog pretpostavljenih čitanja od strane CPU-a ako je mapirano više od 3 MB + 248 KB. Kada je vanjska flash memorija mapirana u prostor memorije samo za čitanje podataka, do 4 MB može biti mapirano

istovremeno, podržavaju se čitanja od 8-bit, 16-bit i 32-bit. Vanjska RAM može biti mapirana u prostor podataka CPU-a. Podržava se SRAM do 8 MB, a istovremeno se može mapirati do 4 MB. Podržana su čitanja i pisanja od 8-bit, 16-bit i 32-bit.

7.1.2. Power Management Unit

Jedinica za upravljanje napajanjem (PMU) ESP32 omogućuje promjenu različitih načina snage uz korištenje naprednih tehnologija. Ovi načini uključuju aktivni način rada, način mirovanja modema, način laganog mirovanja, način dubokog mirovanja i način hibernacije. Na primjer, u načinu dubokog mirovanja, uključeni su samo RTC memorija i RTC periferni uređaji, dok su Wi-Fi i Bluetooth podaci o vezi pohranjeni u RTC memoriju. U načinu hibernacije, interni oscilator od 8 MHz i ULP koprocesor su onemogućeni, dok su samo jedan RTC mjerač vremena na sporom satu i određeni RTC GPIO-ovi aktivni.

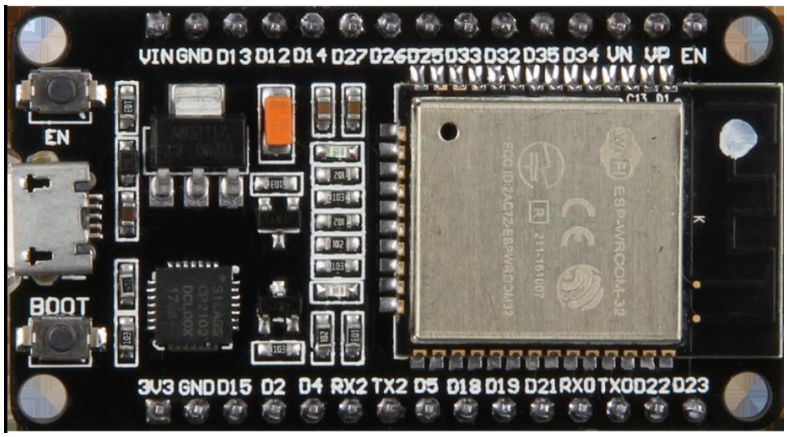
7.1.3. Ultra low power coprocessor

Tijekom načina dubokog mirovanja, ULP koprocesor i RTC memorija ostaju aktivni, omogućujući programeru pohranu programa za ULP koprocesor u RTC sporu memoriju. To omogućuje pristup perifernim uređajima, unutarnjim mjeračima vremena i senzorima tijekom mirovanja, što je korisno u aplikacijama gdje CPU treba biti aktiviran vanjskim događajem ili mjeračem vremena uz minimalnu potrošnju energije.

7.1.4. General purpose input output pinovi

ESP32 raspolaže s 34 GPIO pina koja se programskim putem mogu prilagoditi za različite funkcije putem određenih registara. GPIO pinovi dolaze u više varijanti, uključujući isključivo digitalne, analogne i one osjetljive na dodir, pri čemu se analogni i kapacitivni dodirni GPIO pinovi mogu postaviti da djeluju kao digitalni. Većina digitalnih GPIO pina omogućava konfiguraciju kao unutarnji pull-up ili pull-down otpori, ili se mogu postaviti na visoku impedanciju. Kada su postavljeni kao ulazni, njihove vrijednosti se mogu čitati iz registra, a moguće ih je konfigurirati za generiranje prekida CPU-a na temelju promjene stanja ili razine

signala. Većina digitalnih IO pinova su dvosmjerni, neinvertirajući i s mogućnošću postavljanja u tri stanja, uključujući ulazne i izlazne buffere s kontrolom tri stanja. Ti se pinovi također mogu koristiti zajedno s drugim funkcijama poput SDIO, UART, SPI itd. Za aplikacije s niskom potrošnjom, GPIO pinovi se mogu konfigurirati da zadrže svoje stanje.



***Slika 1**.ESP32 modul*

7.2. DHT22

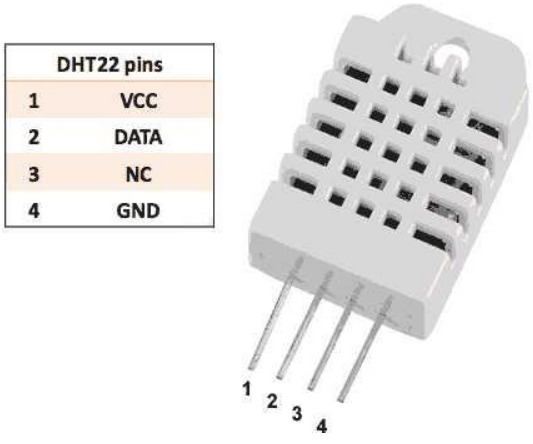
DHT22 je popularan senzor temperature i vlažnosti koji se često koristi u različitim projektima i aplikacijama gdje su potrebni točni podaci o okolišnim uvjetima. Njegove glavne karakteristike uključuju:

1. Digitalni izlazni signal: DHT22 koristi digitalni signal za prijenos podataka, što eliminira potrebu za digitalno-analognom konverzijom i smanjuje mogućnost grešaka u signalu.
2. Ekskluzivna tehnika prikupljanja digitalnog signala i tehnologija senzora vlažnosti: Ove tehnike osiguravaju visoku pouzdanost i stabilnost senzora.
3. Integracija s 8-bitnim mikročipom: Senzorski elementi DHT22 su integrirani s 8-bitnim mikročipom koji obrađuje podatke, omogućujući precizno čitanje temperature i vlažnosti.
4. Temperaturna kompenzacija i kalibracija: Svaki DHT22 senzor je temperaturno kompenziran i kalibriran u kontroliranim uvjetima, što osigurava točnost mjerenja. Kalibracijski koeficijenti su pohranjeni u OTP (One-Time Programmable) memoriji na čipu, što omogućuje senzoru da pristupi ovim podacima tijekom rada.

5. Mala veličina i niska potrošnja energije: Zahvaljujući svojoj kompaktnoj veličini i niskoj potrošnji energije, DHT22 je pogodan za upotrebu u različitim aplikacijama, uključujući one gdje su prostor ili energija ograničeni.
6. Duga udaljenost prijenosa: DHT22 može prenositi podatke na udaljenosti do 20 metara, što ga čini pogodnim za aplikacije gdje senzor i mikrokontroler nisu fizički blizu jedan drugome.
7. Jednostavna veza s četiri pina: DHT22 dolazi u pakiranju s četiri pina, što olakšava povezivanje s mikrokontrolerima ili drugim elektroničkim sklopovima.

   Zbog ovih karakteristika, DHT22 je izuzetno popularan među hobistima, istraživačima i profesionalcima koji razvijaju projekte ili proizvode koji zahtijevaju praćenje okolišnih uvjeta, poput sustava za automatizaciju doma, staklenika, meteoroloških stanica i mnogih drugih aplikacija.

DHT22 je dizajniran za rad s napajanjem od 3,3 do 6V DC, pri čemu je preporučljivo dodati kondenzator od 100nF između VDD i GND za bolju stabilnost. Komunikacija između senzora i mikrokontrolerske jedinice (MCU) odvija se preko jedne sabirnice, zauzimajući 5 mS za prijenos. Format podataka koji DHT22 šalje uključuje integralne i decimalne podatke o relativnoj vlažnosti i temperaturi, te kontrolni zbroj koji potvrđuje ispravnost prijenosa. Senzor šalje podatke počevši s viših bitova, a MCU mora poslati startni signal kako bi inicirao prijenos. Nakon slanja podataka, DHT22 se vraća u način rada s niskom potrošnjom energije, osim ako ne primi novi startni signal.

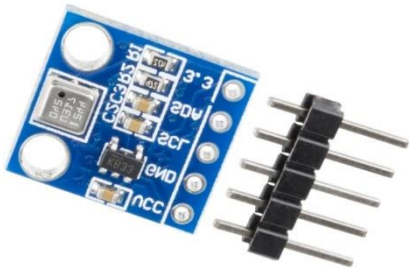


Slika 2. *DHT22*

7.3. BMP180

BMP180 je digitalni senzor tlaka visoke preciznosti, dizajniran kao nasljednik modela BMP185, usmjeren prema potrošačkim aplikacijama. Ovaj senzor karakterizira ultra mala potrošnja energije i niskonaponski rad, što ga čini idealnim za upotrebu u mobilnim uređajima poput telefona, osobnih digitalnih asistenata (PDA), GPS navigacijskih sistema i razne vanjske opreme. Posebno se ističe niskom visinskom bukom od samo 0,25 metara te brzim vremenom pretvorbe, nudeći izuzetne performanse. Zbog svojeg I2C sučelja, BMP180 se lako integrira s mikrokontrolerima, što olakšava korištenje u raznim aplikacijama.

Razvijen na temelju piezorezistivne tehnologije, BMP180 pruža izvrsnu elektromagnetsku kompatibilnost (EMC), visoku preciznost, linearnost te dugoročnu stabilnost. Izgradnja na bogatom iskustvu Roberta Boscha, globalnog lidera u proizvodnji senzora tlaka za automobilsku industriju, koji se temelji na više od 400 milijuna senzora tlaka implementiranih u stvarnim uvjetima, BMP180 predstavlja novu generaciju mikro-strojnih senzora tlaka, nastavljajući tradiciju inovacija i kvalitete.



Slika 3. *BMP180*

BMP180 je sofisticirani digitalni senzor tlaka koji je optimiziran za korištenje u uređajima koji zahtijevaju minimalnu potrošnju energije, poput mobilnih telefona, osobnih digitalnih asistenata (PDA), GPS navigacijskih sistema i slične prijenosne opreme. Ovaj senzor zahtijeva napajanje u rasponu od 1,8V do 3,6V, što ga čini izuzetno prilagodljivim za različite vrste elektroničkih projekata, posebice one na baterije.

Za komunikaciju s mikrokontrolerima, BMP180 koristi I2C sučelje, standard za serijsku komunikaciju koja omogućava povezivanje s dva pina: SDA (Serial Data) i SCL (Serial Clock). Ove karakteristike čine BMP180 vrlo jednostavnim za integraciju u različite elektroničke sustave.

Senzor nudi visoku rezoluciju mjerenja tlaka, koja mu omogućava da pruža precizne informacije o atmosferskom tlaku s niskom visinskom bukom od samo 0,25 metara, čineći ga izuzetno korisnim za aplikacije koje zahtijevaju precizno određivanje nadmorske visine. Brzina uzorkovanja i vrijeme pretvorbe podataka mogu se prilagoditi ovisno o potrebama projekta, dodatno poboljšavajući fleksibilnost senzora.

Korištenje piezorezistivne tehnologije dodatno povećava vrijednost BMP180, osiguravajući visoku razinu elektromagnetske kompatibilnosti (EMC), točnost, linearnost i dugoročnu stabilnost. Ove karakteristike čine BMP180 pouzdanim izborom za širok spektar aplikacija, od meteoroloških stanica do sportske i rekreativne opreme, gdje je točno mjerenje tlaka ključno.

8. MVVM i API POZIV

8.1. MVVM

Model-view-ViewModel je arhitektonski obrazac u računalnom softveru koji olakšava odvajanje razvoja grafičkog korisničkog sučelja od razvoja poslovne logike ili pozadinske logike (odnosno modela).

ViewModel MVVM-a je pretvarač vrijednosti, što znači da je odgovoran za izlaganje (konvertiranje) podatkovnih objekata iz modela na takav način da se njima može lako upravljati i prezentirati. U tom pogledu, model prikaza je više model nego prikaz, i upravlja većinom (ako ne i svom) logikom prikaza pogleda. Model prikaza može implementirati obrazac posrednika, organizirajući pristup pozadinskoj logici oko skupa slučajeva korištenja koje podržava pogled.
MVVM je varijacija obrasca dizajna prezentacijskog modela Martina Fowlera. Izumili su ga Microsoftovi arhitekti Ken Cooper i Ted Peters kako bi pojednostavili programiranje korisničkih sučelja vođenih događajima. Uzorak je ugrađen u Windows Presentation Foundation (WPF) (Microsoftov .NET grafički sustav) i Silverlight, WPF-ov derivat internetske aplikacije.

Model–view–ViewModel također se naziva model–view-binder, posebno u implementacijama koje ne uključuju .NET platformu. ZK, okvir web aplikacije napisan u Javi, i JavaScript biblioteka KnockoutJS koriste model–view–binder.

8.1.1. Model

Model može se odnositi na sloj pristupa podacima, koji predstavlja sadržaj (pristup usmjeren na podatke), ili model domene, koji odražava sadržaj stvarnog stanja (pristup usmjeren na objekte).

8.1.2. View

View je organizacija, dizajn i vizualni prikaz onoga što korisnik vidi na ekranu, slično dizajnu model-pogled-kontroler i model-pogled-prezenter. Prikazuje izgled modela i obrađuje korisnički

unos mišem, tipkovnicom, dodirivanjem i drugim uređajima. Zatim prosljeđuje podatke na obradu.

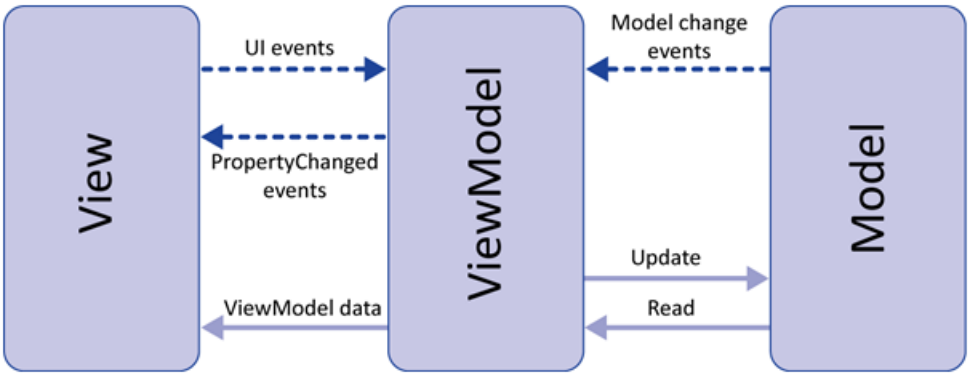
8.1.3. View model

View model predstavlja apstrakciju pogleda koji izlaže javna svojstva i naredbe. Za razliku od kontrolera u MVC obrascu ili prezentera u MVP obrascu, MVVM koristi povezivač koji automatizira komunikaciju između pogleda i pripadajućih svojstava u modelu pogleda. Model prikaza definira stanje podataka u modelu. Ključna razlika između modela pogleda i prezentera u MVP obrascu je što prezenter ima referencu na pogled, dok model pogleda nema. Umjesto toga, pogled se direktno povezuje sa svojstvima modela pogleda za primanje i slanje ažuriranja. Za efikasno funkcioniranje potrebna je tehnologija vezivanja ili generiranje osnovnog koda za vezivanje.

U kontekstu objektno orijentiranog programiranja, model prikaza ponekad se može nazvati objektom za prijenos podataka.

8.1.4. Binder

Deklarativni podaci i vezivanje naredbi su implicitni u MVVM obrascu. Binder oslobađa razvojnog programera od potrebe pisanja standardne logike za sinkronizaciju ViewModela i viewa.



***Slika 4**. Ilustracija rada MVVM arhitekture*

8.2. API poziv

API, skraćenica od aplikacijsko programsko sučelje, predstavlja skup pravila i specifikacija koje programeri prate kako bi koristili usluge ili resurse operacijskog sustava ili drugih složenih programa. To može uključivati standardne biblioteke rutina, struktura podataka, objekata i protokola. Korištenje API poziva omogućava programerima efikasno koristiti rad drugih, štedeći vrijeme i trud potreban za izradu složenih programa. S napretkom operacijskih sustava, posebno u grafičkom korisničkom sučelju, API postaje neizostavan u stvaranju novih aplikacija, omogućujući programerima da nadograđuju na postojeći rad umjesto da pišu nove programe od temelja.

8.2.1. Proces upućivanja API poziva

Proces poziva API-ju odvija se kroz sljedeće korake:

1. Korisnik upućuje poziv API-ju putem njegovog Uniform Resource Identifiera (URI), pridružujući zahtjev, zaglavlja te tijelo zahtjeva.
2. S valjanim zahtjevom, API šalje poziv vanjskom programu za dobivanje podataka.
3. API prima odgovor od vanjskog programa.
4. API predaje dobivene podatke početnom programu koji je poslao zahtjev.

Uniform Resource Identifier (URI) je digitalna adresa aplikacije, slična na primjer fizičkoj adresi. Program koji zahtijeva mora znati odgovarajući URI kako bi pristupio API-ju.

Analogijom adrese, pozivanje API-ja je poput odlaska nekome i traženja usluge. Vrsta zahtjeva određena funkcijama kao što su GET, POST, PUT, PATCH i DELETE, označavajući željenu vrstu akcije.

Zahtjevi imaju određeni format, određen zaglavlja, poput Content-Type i Accept. Content-Type informira API o vrsti podataka u tijelu zahtjeva, dok Accept određuje željeni format odgovora. Pravilno postavljanje tih zaglavlja ključno je za uspješnu komunikaciju s API-jem, s obzirom na važnost preciznosti u formatiranju podataka.

API, uz ispravan zahtjev, preusmjerava poziv prema vanjskom programu za podatke. Važno je napomenuti da većina API-ja ne pohranjuje informacije, već pruža povezivanje s vanjskim programom. Uloga API-ja je osluškivanje, provjera valjanosti zahtjeva te, ako je valjan, pružanje odgovora. Svi podaci u odgovoru potječu iz vanjskog programa; API ih samo obradi i predstavi korisnicima.

U analogiji posjeta nečijoj kući, kucanje na vrata predstavlja poznavanje URI-ja API-ja. Osoba koja otvara vrata ima ulogu API-ja, odgovarajući na zahtjev programa. Kada se traži kuhinjski nož za maslac, vanjska kuhinja postaje ekvivalent vanjskom programu, od kojeg API traži podatke. Kada osoba otvara izvlačenje u potrazi za nožem, to je analogno API-ju koji traži odgovor od vanjskog programa. Međutim, u primjeru dolazi do problema jer je vanjski program već izgubio jedan nož, što ilustrira potencijalni nedostatak efikasnosti u procesu traženja podataka.

API može eventualno vratiti podatke u neobrađenom obliku, ali format može biti ključan. Ako program koji zahtijeva podatke ne podržava određeni format, postaje beskoristan. Zbog toga će API analizirati informacije u zaglavljima zahtjeva, posebno koristeći primjer zaglavlja Accept koji pokazuje željeni format podataka. Ako je potrebna promjena formata, API će izvršiti konverziju prema svojim mogućnostima. Na primjer, ako je program koji zahtijeva želio podatke u JSON formatu, a vanjski program ih isporučio u XML formatu; zadatak API-ja je konvertirati XML u JSON.

Dakle koraci poziva upućenog API-ju su sljedeći: podnesete zahtjev s određenim metapodacima za URI, provjerite valjanost zahtjeva. API prima zahtjev i šalje vlastiti upit vanjskom programu. Program pruža podatke koje je API tražio. Iako pretvorba može ili ne mora biti potrebna, API konačno dostavlja podatke vanjskom programu u obliku odgovora koji je prvotno tražen. To je jednostavno i izuzetno učinkovito.

9. FIREBASE

Firebase, je skup pozadinskih usluga računalstva u oblaku i platformi za razvoj aplikacija koje pruža Google. Uključuje baze podataka, usluge, autentifikaciju i integraciju za razne aplikacije.

Firebase je nastao iz Envolvea, prethodnog startupa osnovanog 2011. godine. Envolve je pružao programerima API za integraciju funkcionalnosti online chata na njihove web stranice. Nakon što su lansirali chat uslugu, osnivači su primijetili da se koristi i za razmjenu podataka aplikacija koji nisu bili chat poruke. Programeri su koristili Envolve za sinkronizaciju podataka aplikacija, kao što je stanje igre u stvarnom vremenu među korisnicima. Zato su osnivači odlučili odvojiti sustav za chat od arhitekture u stvarnom vremenu koja ga je podržavala. Firebase je osnovan kao zasebna tvrtka 2011. godine i lansiran u javnost u travnju 2012. godine.

Prvi proizvod Firebasea bio je Firebase RealTime Database, API koji sinkronizira podatke aplikacija na iOS-u, Androidu i web uređajima te ih pohranjuje u Firebaseov oblak. Ovaj proizvod pomaže programerima u izgradnji suradničkih aplikacija u stvarnom vremenu.

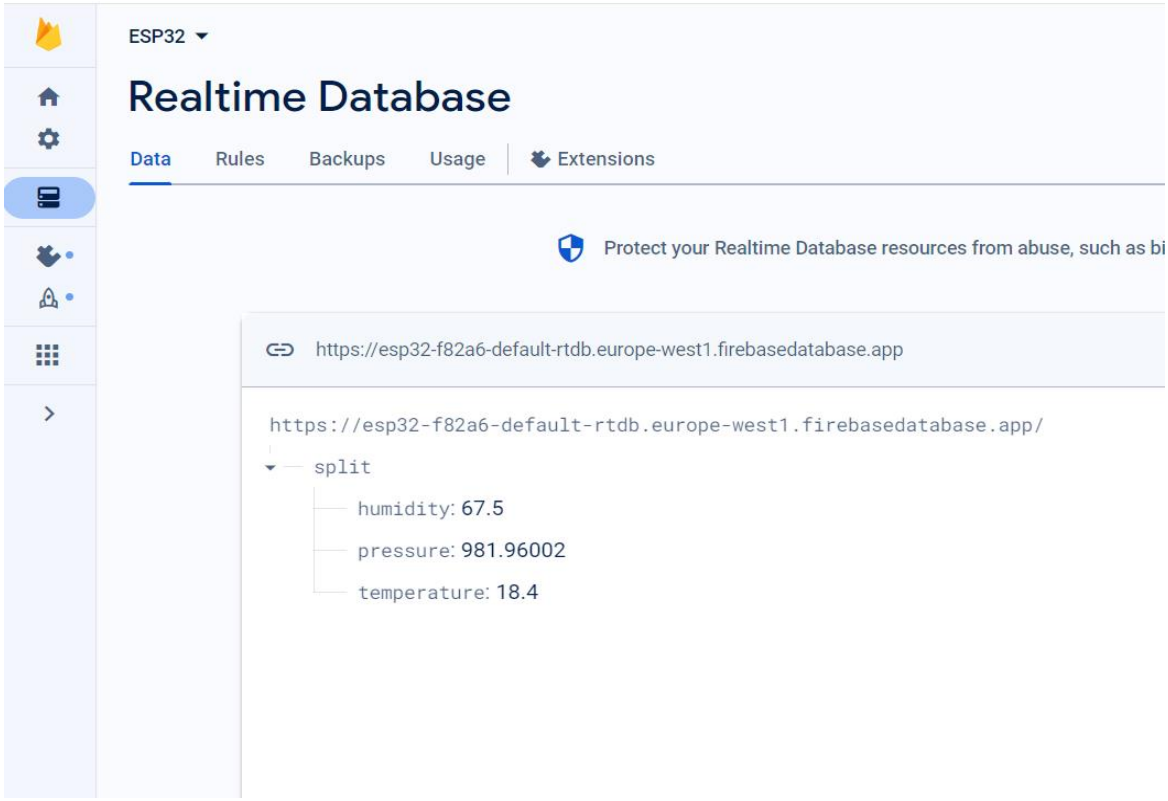
U svibnju 2012., samo mjesec dana nakon lansiranja beta verzije, Firebase je dobio početna sredstva od 1,1 milijun dolara od rizičnih kapitalista. U lipnju 2013., tvrtka je dodatno prikupila 5,6 milijuna dolara .

2014. godine Firebase je predstavio dva nova proizvoda: Firebase Hosting i Firebase Authentication, što je tvrtku pozicioniralo kao vodećeg pružatelja mobilnih pozadinskih usluga. U listopadu iste godine, Firebase je akviziran od strane Googlea.

U svibnju 2016., na Google I/O konferenciji za razvojne programere, Firebase je predstavio Firebase Analytics i najavio širenje svojih usluga kako bi postao sveobuhvatna platforma kao usluga (BaaS) za mobilne programere. Integrirajući se s raznim Googleovim uslugama poput Google Cloud Platform, AdMob i Google Ads, Firebase je ponudio širi spektar proizvoda i povećao dostupnost za programere. Firebase Cloud Messaging je zamijenio Google Cloud Messaging kao usluga za slanje push obavijesti na Android uređajima, dodajući funkcionalnost za isporuku push obavijesti na Android, iOS i web uređajima.

U srpnju 2016., Google je preuzeo LaunchKit, mobilnu razvojnu platformu specijaliziranu za marketing aplikacija razvojnih programera, te ju integrirao u tim Firebase Growth Tools. U siječnju 2017., Google je kupio Fabric i Crashlytics od Twittera kako bi dodao ove usluge u Firebase.

U listopadu 2017., Firebase je lansirao Cloud Firestore, bazu podataka dokumenata u stvarnom vremenu, kao nasljednika izvorne Firebase baze podataka u stvarnom vremenu.



***Slika 5**. Firebase baza podataka*

9.1. Slanje podataka lokalnih senzora u Firebase

Koristeći Arduino IDE programiramo način rada ESP32 SoC na kojeg su spojeni senzori DHT22 za očitanje vlage i temperature, te BMP180 za očitanje tlaka zraka. Kod koristi potrebne biblioteke za ESP32, Wi-Fi, Firebase, DHT22, BMP180 i Wire (za I2C komunikaciju).
Definira konstante i varijable za Wi-Fi podatke, Firebase API ključ, URL Firebase RTDB-a te interval ažuriranja za slanje podataka. Inicijalizira objekte za Firebase autentifikaciju, konfiguraciju i podatke.Postavlja raspored pinova na ESP32, povezuje se na Wi-Fi i inicijalizira Firebase. Inicijalizira DHT22 senzor i provjerava uspješnost inicijalizacije BMP180 senzora. U ponavljajućoj petlji provjerava je li Firebase spreman, je li registracija uspješna i je li vrijeme za slanje podataka. Čita temperaturu i vlagu s DHT22 senzora, te tlak s BMP085 senzora. Upisuje dobivene podatke koristeći definirane putanje u Firebase RTDB. Važno je napomenuti da su varijable YOUR_SSID i YOUR_PASSWORD moraju biti ime i zaporka WiFi mreže na koju se spaja ESP. YOUR_API_KEY i DATABASE_URL moraju odgovarati API keyu i URL-u koji nam generira Firebase.

```
platform: ESP32
  sensors:
    - DHT22
    - BMP180
  used libraries:
    - name=Adafruit BMP180 Unified, version=1.1.3, author=Adafruit
    - name=Adafruit Unified Sensor, version=1.1.14, author=Adafruit+
    - name=DHT sensor library, version=1.4.6, author=Adafruit
    - name=Firebase Arduino Client Library for ESP8266 and ESP32, version=4.4.11,
author=Mobizt
  pinout:
    - DHT22:
      - VCC: +3.3V
      - GND: GND
      - DATA: GPIO 13
    - BMP180:
      - VCC: GPIO 23
      - GND: GND
      - SCL: GPIO 22
      - SDA: GPIO 21
*/
```

```
#include <Arduino.h>
#if defined(ESP32)
  #include <WiFi.h>
#elif defined(ESP8266)
  #include <ESP8266WiFi.h>
#endif
#include <Firebase_ESP_Client.h>

#include "addons/TokenHelper.h"
#include "addons/RTDBHelper.h"

// Insert your network credentials
#define WIFI_SSID "YOUR_SSID"
#define WIFI_PASSWORD "YOUR_PASSWORD"

// Insert Firebase project API Key
#define API_KEY "YOUR_API_KEY"

// Insert RTDB URLefine the RTDB URL */
#define DATABASE_URL "https://YOUR_PROJECT.REGION.firebasedatabase.app/"

//Define Firebase Data object
FirebaseData fbdo;

FirebaseAuth auth;
FirebaseConfig config;

// DHT22
#include "DHT.h"
#define DHTPIN 13       // Digital pin connected to the DHT sensor
#define DHTTYPE DHT22   // DHT 22  (AM2302), AM2321
DHT dht(DHTPIN, DHTTYPE);

float h;  //  variable used for Humidity
float t;  // variable used for Temperature

// BMP180
#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BMP085_U.h>

Adafruit_BMP085_Unified bmp = Adafruit_BMP085_Unified(10085);

int updateInterval = 60000;
unsigned long sendDataPrevMillis = 0;
int count = 0;
```

```
bool signupOK = false;

void setup() {
  // Use PIN 23 as VCC for BMP180
  pinMode(23, OUTPUT);
  digitalWrite(23, HIGH);

  // Connect to Wi-Fi
  Serial.begin(115200);
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
  Serial.print("Connecting to Wi-Fi");
  while (WiFi.status() != WL_CONNECTED){
    Serial.print(".");
    delay(300);
  }
  Serial.println();
  Serial.print("Connected with IP: ");
  Serial.println(WiFi.localIP());
  Serial.println();

  /* Assign the api key (required) */
  config.api_key = API_KEY;

  /* Assign the RTDB URL (required) */
  config.database_url = DATABASE_URL;

  /* Sign up */
  if (Firebase.signUp(&config, &auth, "", "")){
    Serial.println("ok");
    signupOK = true;
  }
  else{
    Serial.printf("%s\n", config.signer.signupError.message.c_str());
  }

  /* Assign the callback function for the long running token generation task */
  config.token_status_callback = tokenStatusCallback; //see addons/TokenHelper.h

  Firebase.begin(&config, &auth);
  Firebase.reconnectWiFi(true);

  // Initialize DHT22
  dht.begin();

  // Initialize BMP180
  if(!bmp.begin())
```

```
  {
    /* There was a problem detecting the BMP180 ... check your connections */
    Serial.print("Ooops, no BMP180 detected ... Check your wiring or I2C ADDR!");
    while(1);
  }
}

void loop() {
  if (Firebase.ready() && signupOK && (millis() - sendDataPrevMillis > updateInterval ||
sendDataPrevMillis == 0)){
    sendDataPrevMillis = millis();

    // read from DHT22
    float h = dht.readHumidity();
    float t = dht.readTemperature();

    if (isnan(h) || isnan(t)) {
      Serial.println(F("Failed to read from DHT sensor!"));
      return;
    }

    // read from BMP180
    sensors_event_t event;
    bmp.getEvent(&event);

    // Write Temperature to the database path split/temperature
    if (Firebase.RTDB.setFloat(&fbdo, "split/temperature", t)){
      Serial.println("PASSED");
      Serial.println("PATH: " + fbdo.dataPath());
      Serial.println("TYPE: " + fbdo.dataType());
    }
    else {
      Serial.println("FAILED");
      Serial.println("REASON: " + fbdo.errorReason());
    }

    // Write Pressure to the database path split/pressure
    if (Firebase.RTDB.setFloat(&fbdo, "split/pressure", event.pressure)){
      Serial.println("PASSED");
      Serial.println("PATH: " + fbdo.dataPath());
      Serial.println("TYPE: " + fbdo.dataType());
    }
    else {
      Serial.println("FAILED");
      Serial.println("REASON: " + fbdo.errorReason());
    }
```

```
    // Write Humidity to the database path split/humidity
    if (Firebase.RTDB.setFloat(&fbdo, "split/humidity", h)){
      Serial.println("PASSED");
      Serial.println("PATH: " + fbdo.dataPath());
      Serial.println("TYPE: " + fbdo.dataType());
    }
    else {
      Serial.println("FAILED");
      Serial.println("REASON: " + fbdo.errorReason());
    }
  }
}
```

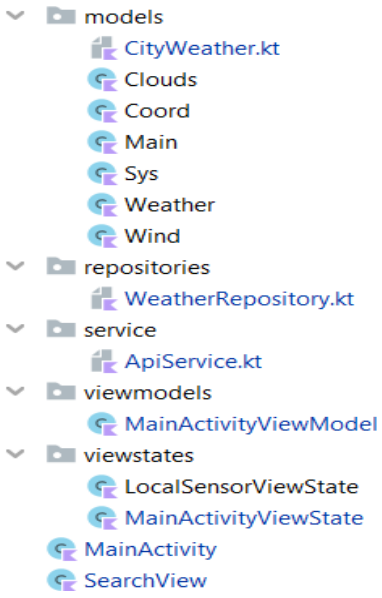
10.ANDROID APLIKACIJA ZA PRAĆENJE LOKALNIH I GLOBALNIH VREMENSKIH UVJETA

U ovom poglavlju će biti opisana android aplikacija koju koristimo da bi pratili lokalne podatke sa senzora kao i vremenske podatke određenih gradova na globalnoj razini. Osvrnut ćemo se na sve što smo prethodno opisivali i kako je implementirano i objedinjeno unutar same aplikacije.

Sama struktura aplikacije temelji se na MVVM arihtekturi koja je prethodno opisana. MVVM koristi različite slojeve kao što su data odnosno podatkovni sloj, domain sloj, UI sloj. U našoj aplikaciji domain sloj se ne koristi. Naša aplikacija je podjeljena na pet razina koje uključuju: modele, repozitorije, servise, viewmodel, viewstates, te activity-je.

10.1. Podatkovni sloj

Modeli, repozitoriji i servisi spadaju u data odnosno podatkovni sloj. Modeli su klase koje opisuju kako trebaju izgledati podatci koje primamo sa, naprimjer, nekakvog servera. Repozitorij je klasa koja obavlja poziv prema serveru i dohvaćanje podataka. API service definira retrofit klijenta. Retrofit je library odnosno biblioteka koju koristimo kada želimo ustupiti komunikaciju s nekim serverom odnosno API-em.



Slika 6. *Struktura aplikacije*

Izgled jednog od modela korištenog u programu je prikazan u sljećim linijama koda. Jasno se vidi kako trebaju biti strukturirani podatci odnosno kojeg su tipa.

```
@Parcelize
data class Main (

  @SerializedName("temp"     ) var temp     : Double? = null,
  @SerializedName("pressure" ) var pressure : Int?    = null,
  @SerializedName("humidity" ) var humidity : Int?    = null,
  @SerializedName("temp_min" ) var tempMin  : Double? = null,
  @SerializedName("temp_max" ) var tempMax  : Double? = null

) : Parcelable
```

Sljedeće linije koda prikazuju izgled API servisa. Koristeći private val client na kojeg postavljamo interceptor koji u ovom slučaju logira api pozive i dodaje api key. U funkciji addApiKeyToRequest dodajemo querry parametre odnosno u nasem slucaju appid api key te konvertiranje u metričke jedinice (jer iz baze podataka dobivamo imperijske). Ovaj api poziv koristi se za dohvaćanje vremenskih podataka u gradovima diljem svijeta. U ApiService interfaceu dodajemo putanju kojom pristupamo našem baseUrl i time mu definiramo s kim točno želimo komunicirat, odnosno želimo od tog URL-a dobiti podatke. @Query “q” je parametar upita kojemu šaljemo cityName odnosno ime grada, jer je prilikom pozivanja (@GET(“2.5/weather”)) moramo poslati cityName da bi zapravo i dobili parametre za trazeni grad.

```
interface ApiService {
   @GET("2.5/weather")
   fun getCityWeather(
      @Query("q") cityName: String,
      ): Call<CityWeather>
}

class RetrofitFactory {
   companion object {
      private val client = OkHttpClient.Builder().addInterceptor {
```

```
            HttpLoggingInterceptor().apply {
                level = HttpLoggingInterceptor.Level.BASIC
            }
            addApiKeyToRequests(it)
        }.build()
        private val retrofit: Retrofit by lazy {
            Retrofit.Builder()
                .baseUrl("https://api.openweathermap.org/data/")
                .client(client)
                .addConverterFactory(GsonConverterFactory.create())
                .build()
        }
        val service: ApiService by lazy {
            retrofit.create(ApiService::class.java)
        }
        private fun addApiKeyToRequests(chain: Interceptor.Chain): Response {
            val request = chain.request().newBuilder()
            val originalHttpUrl = chain.request().url
            val newUrl = originalHttpUrl.newBuilder()
                .addQueryParameter("appid", "1f63accde196ec0e77f1fb6899e38b48")
                .addQueryParameter("units", "metric").build()
            request.url(newUrl)
            return chain.proceed(request.build())
        }
    }
}
```

Sljedeće linije koda prikazuju repozitorij u kojem se definira kako se dohvaćaju podatci I što učiniti ukoliko je neuspješno dohvaćanje. Repozitorij nam služi za definiranje komunikacije sa serverom.

```
class WeatherRepositoryImp(private val apiService: ApiService) : WeatherRepository {


    override suspend fun getCityWeather(cityName: String): CityWeatherResponse {
        return try {
            val data = apiService.getCityWeather(cityName).await()
            CityWeatherResponse(
                isSuccess = true,
                message = "Success",
```

```
                data = data,
                code = 200,
            )
        } catch (e: Exception) {
            CityWeatherResponse(isSuccess = false, message = e.localizedMessage ?: "")
        }
    }
}

interface WeatherRepository {
    suspend fun getCityWeather(cityName:String): CityWeatherResponse
}
```

10.2. Sloj korisničkog sučelja

ViewModel je temeljni dio MVVM arhitekture te upravo njime odvajamo programske i radne logike. Prilikom upita od UI odnosno korisničkog sloja pristupa podatkovnom sloju.ViewModel služi kao posrednik te drži podatke u memoriji koje prilikom upita prosljeđuje drugim elementima korisničkog sučelja. Upravo njime odvajamo programske i radne logike. U našem primjeru ViewModel je također zadužen za promatranje podataka, odnosno reagira na promjenu vrijednosti korištenjem LiveData-e.

```
class MainActivityViewModel(private val repository: WeatherRepository) : ViewModel() {
    private val query = MutableLiveData<String>()
    private var cityWeather: MutableLiveData<CityWeatherResponse> = MutableLiveData()

    fun fetchData() {
        query.value = ""
    }

    fun getCityWeather() = cityWeather

    suspend fun getCityWeatherFromRepo(cityName: String){
        val weather = repository.getCityWeather(cityName)
        cityWeather.postValue(weather)
```

```
        println("MYTAG t ${cityWeather.value}")
    }


    @Suppress("UNCHECKED_CAST")
    class Factory(private val repository: WeatherRepository) : ViewModelProvider.Factory {
        override fun <T : ViewModel> create(modelClass: Class<T>): T {
            return MainActivityViewModel(repository) as T
        }
    }
}
```

ViewStateovi su klase koje definiraju elemente koje ćemo koristiti na user interfaceu. Sljedeći kod prikazuje elemente potrebne za prikaz vrijednosti na lokalnom senzoru. I definiramo ih kao String odnosno tekstualni tip varijable. Ukoliko neko od očitanja ne bude registrirano vraća vrijednost null.

```
class LocalSensorViewState: BaseObservable() {


    @Bindable
    var pressure: String? = null
    @Bindable
    var temperature: String? = null
    @Bindable
    var humidity: String? = null


}
```

MainActivity se kreira tako da u SearchViewu (opisan naknadno) klikom na gumb nakon što smo upisali ime traženog grada. Time se pokreće novi intent koji nam ustvari otvara main activity te se podatci šalju koristeći komandu intent.getStringExtra(„Search Querry”). LifecycleScope.launch pokreće novu korutinu koja pokreće proces dohvaćanja podataka sa servera. Observer promatra podatke, ukoliko je uspješno dohvaćanje podataka prikazuje ih na

ekranu, ukoliko nije uspješno dohvaćanje podataka šalje grešku te komandom finish() zatvara prozor MainActivitya i vraća nas u SearchView.

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding
    private lateinit var viewState: MainActivityViewState

    private val repository: WeatherRepositoryImp by lazy {
        WeatherRepositoryImp(RetrofitFactory.service)
    }

    private val viewModel: MainActivityViewModel by lazy {
        val factory = MainActivityViewModel.Factory(repository = repository)
        ViewModelProviders.of(this, factory)[MainActivityViewModel::class.java]
    }

    @SuppressLint("SetTextI18n")
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        val cityName = intent.getStringExtra("searchQuery")
        binding = DataBindingUtil.setContentView(this, R.layout.activity_main)

        viewState = MainActivityViewState()

        lifecycleScope.launch {
            viewModel.getCityWeatherFromRepo(cityName ?: "")
        }

        binding.viewState = viewState


            viewModel.getCityWeather().observe(this, Observer {
                viewState.apiInProgress = false
                if (it != null) {
                    if (it.isSuccess) {
                        binding.tvCityname.text = it.data?.name
                        binding.tvWeather.text = it.data?.weather?.get(0)?.description
                        binding.tvPressurevalue.text = "${it.data?.main?.pressure.toString()} hPa"
                        binding.tvTemperaturevalue.text = "${it.data?.main?.temp?.toString()}°C"
                        binding.tvHumidityvalue.text = "${it.data?.main?.humidity}%"
                        binding.tvDailyminmaxvalue.text ="${it.data?.main?.tempMin.toString()}°C -
```

```
${it.data?.main?.tempMax.toString()}°C"
                binding.tvWindvalue.text = "${it.data?.wind?.speed?.toString()} m/s"
                binding.tvCoordinatesvalue.text = "${it.data?.coord?.lon.toString()},
${it.data?.coord?.lat.toString()}"

                Glide.with(this)

.load("https://openweathermap.org/img/w/${it.data?.weather?.get(0)?.icon}.png")
                    .apply(RequestOptions())
                    .into(binding.imgWeather)
            } else {
                viewState.setError(it.message)
                finish()
                Toast.makeText(this, "We couldn't find this city. Mind your spelling or try a
different city.", LENGTH_LONG).show()
            }
        }
    })

    binding.btBack.setOnClickListener {
        finish()
    }
  }
}
```

Google Services folder koji koristimo za dohvaćanje podataka sa Firebase-a u kojemu su definirani parametri potrebni za sinkroniziranje podataka na našoj aplikaciji i onima u bazi podataka firebasea koje dobivamo sa našeg lokalnog senzora. Ovako dohvaćene podatke šaljemo na korisničko sučelje koristeći SearchView.


```
{
  "project_info": {
    "project_number": "535962118700",
    "firebase_url": "https://esp-iot-ij-default-rtdb.europe-west1.firebasedatabase.app",
    "project_id": "esp-iot-ij",
    "storage_bucket": "esp-iot-ij.appspot.com"
  },
  "client": [
    {
      "client_info": {
        "mobilesdk_app_id": "1:535962118700:android:d48309b8aac054a62abf89",
        "android_client_info": {
          "package_name": "com.example.aplikacija"
        }
      },
      "oauth_client": [],
      "api_key": [
        {
          "current_key": "AIzaSyC0Brb8qKsN72_iUy1uXMX32tHorJGbBl4"
        }
      ],
      "services": {
        "appinvite_service": {
          "other_platform_oauth_client": []
        }
      }
    }
  ],
  "configuration_version": "1"
}
```


SetOnClickListener koristimo za prelazak u prethodno opisani MainActivity. Koristeći Firebase SDK podatci sa firebasea se interno procesuiraju. Firebase.database.reference nam služi za

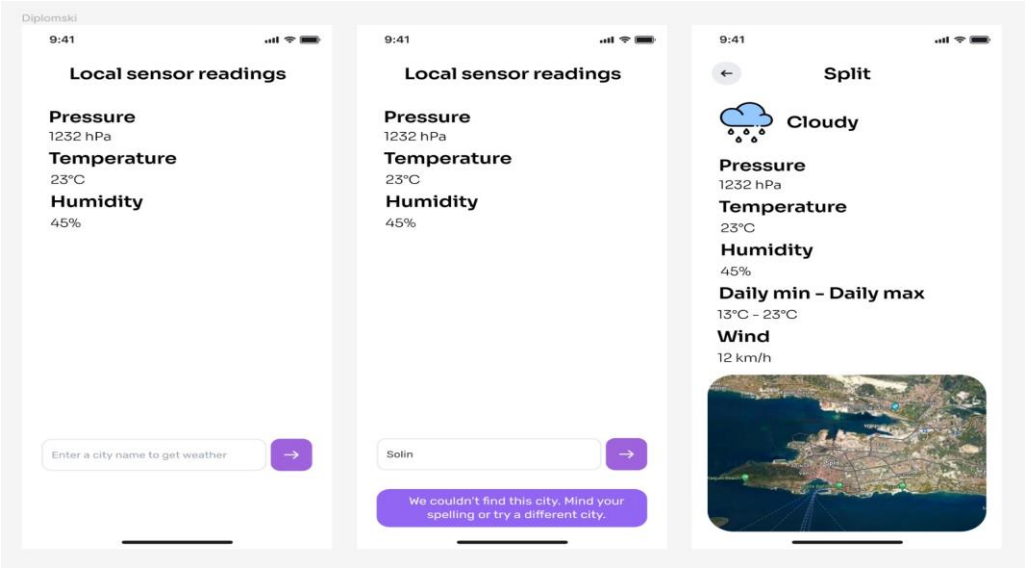
dohvaćanje reference baze koja je postavljena u pozadini. Inicijalizacijom listenera myRef.addValueEventListener koji na svaku promjenu podataka (override fun onDataChange) ažurira podatke.

```
class SearchView: AppCompatActivity() {
    private lateinit var binding: ViewSearchBinding
    private lateinit var viewState: LocalSensorViewState

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ViewSearchBinding.inflate(layoutInflater)
        val view = binding.root
        setContentView(view)

        viewState = LocalSensorViewState()

        viewState.pressure = ""
        viewState.humidity = ""
        viewState.temperature = ""

        binding.btWeather.setOnClickListener {
            if (!binding.etCity.text.isNullOrBlank()) {
                preformSearch(binding.etCity.text.toString())
            }
        }

        val database = Firebase.database
        val myRef = database.reference


        val sensorListener = object : ValueEventListener {
            @SuppressLint("SetTextI18n")
            override fun onDataChange(dataSnapshot: DataSnapshot) {
                val data = dataSnapshot.getValue<Map<String, Double>>()

                binding.tvPressurevalue.text = "${data?.get("pressure").toString()} hPa"
                binding.tvHumidityvalue.text = "${data?.get("humidity").toString()}%"
                binding.tvTemperaturevalue.text = "${data?.get("Temperature").toString()}°C"
            }

            override fun onCancelled(databaseError: DatabaseError) {

                Log.w(TAG, "loadPost:onCancelled", databaseError.toException())
            }
```

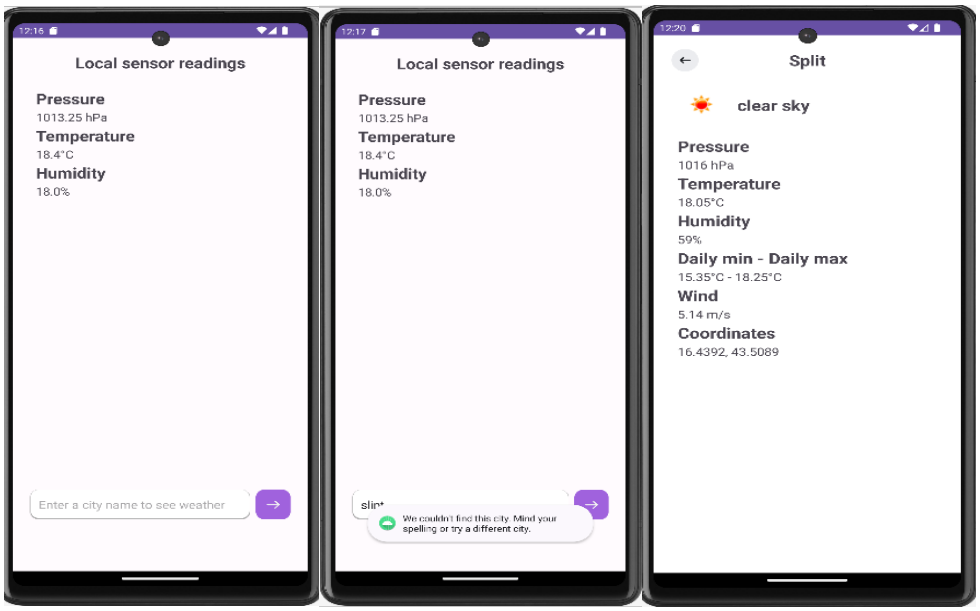
```
        }
        myRef.addValueEventListener(sensorListener)

        binding.viewState = viewState
    }
        private fun preformSearch(query: String){
            val intent = Intent(this, MainActivity::class.java)
            intent.putExtra("searchQuery",query)
            startActivity(intent)
        }
    }
```

10.3. Izgled aplikacije

Koristeći alat Figma za dizajniranje korisničkog iskustva i korisničkog doživljaja (UI i UX) napravljen je prototip dizajna naše aplikacije. Koristeći ideju dizajna na slici 6. formiramo kod XML dokumenata unutar naše aplikacije. Završni izgled aplikacije se blago razlikuje od prototipa i to je vidljivo na slici 7. Prvi ekran prilikom pokretanja aplikacije nam prikazuje pritisak, temperaturu i vlažnost zraka sa lokalnih senzora spojenih na ESP32, drugi ekran nam prikazuje poruku upozorenja ukoliko se upiše nepostojeći grad, a treći ekran nam pokazuje trenutne vremenske uvjete u odabranom gradu kao i njegove koordinate.



Slika 7. *Prototip dizajna napravljen Figma alatom*



Slika 8. *Stvarni izgled završene aplikacije*

11.ZAKLJUČAK

Ovaj rad detaljno istražuje i primjenjuje rastuću IoT (Internet of Things) tehnologiju u kontekstu razvoja Android aplikacije koja omogućuje praćenje različitih podataka o vremenskim uvjetima. Opisani su bitni pojmovi u kontekstu Android programiranja kao: arhitektura Model-View-ViewModel koja omogućuje jasnu separaciju logike aplikacije od korisničkog sučelja, što olakšava održavanje i proširivost aplikacije dugoročno

Koristi se Firebase platforma za pohranu podataka koja omogućuje sinkronizaciju između korisničkih uređaja i servera. To osigurava brzu i pouzdanu komunikaciju između senzora koji prikupljaju podatke i same aplikacije na korisničkom uređaju. Implementacijom API poziva omogućujemo aplikaciji pristup različitim podacima koji nisu direktno dostupni putem senzora, poput vremenskih prognoza za različita geografska područja.

Dodatno, naglašava se skalabilnost, jednostavna proširivost i modularnost aplikacije. Ova karakteristika omogućuje aplikaciji da lako evoluira kako bi zadovoljila promjenjive potrebe korisnika i industrije. Praćenje vremenskih uvjeta može biti od vitalnog značaja za na primjer poljoprivrednike kako bi optimizirali poljoprivredne operacije, planirali sadnju i žetvu te upravljali resursima učinkovitije.

Ova aplikacija predstavlja snažan temelj za daljnje istraživanje i primjenu IoT tehnologije u mobilnom okruženju, otvarajući vrata novim inovacijama i mogućnostima u različitim sektorima.

POPIS SLIKA

LITERATURA

1. Kumar, Ramesh (2020) **Android MVVM Architecture (ViewModel, Repository, RepositoryImp, LiveData, Kotlin) Complete Running code from Scratch**, https://medium.com/@ramesh.approxen/android-mvvm-architecture-viewmodel-repository-repositoryimp-livedata-kotlin-complete-571989cc0618 (16.6.2023.)
2. Allurwar, Naval (2022) **How IoT works – explanation of IoT Architecture & layers**, https://iotdunia.com/iot-architecture/ (15.10.2023.)
3. https://openweathermap.org/api (1.12.2023)
4. Borgini, Julia (2023) **Top advantages and disadvantages of IoT in business** (17.1.2024)
5. Gillis, Alexander (2021), **What is internet of things (IoT)?.IOT Agenda.** (17.1.2024.)
6. https://forgeahead.io/blog/4-stages-of-iot-architecture/ (20.1.2024.)
7. https://en.wikipedia.org/wiki/Object-oriented_programming (20.1.2024)
8. Lutkevich, Ben **Kotlin** (21.1.2024)
9. Arsić, Marko (2017) **Uvod u Kotlin — Koje su prednosti novog zvaničnog jezika za razvoj Android aplikacija?** (21.1.2024)
10. Mariana Berga, Rute Figueiredo (2023)**Kotlin vs Java: the 12 differences you should know** (22.1.2024)
11. Saleh, Hazem (2017) **MVVM architecture, ViewModel and LiveData (Part 1)** (30.1.2024)
12. Ramotion (2023) **Understanding MVVM: Model-View-ViewModel Architecture Explained** (30.1.2024)
13. Ivić, Slaven (2019) **Primjena tehnologije internet stvari (IoT) za udaljeno praćenje mjerenja senzora** (5.2.2024)
14. https://en.wikipedia.org/wiki/Weather_station (5.2.2024)
15. Beckintosh (2019) **How does an API call work?** (6.2.2024)
16. https://firebase.google.com/docs/database (7.2.2024)
17. https://randomnerdtutorials.com/esp32-firebase-realtime-database/ (8.2.2024)
18. https://www.youtube.com/watch?v=LaUzGdtLFiQ&ab_channel=EducationisLife (8.2.2024)

19. https://firebase.google.com/docs/database/android/lists-of-data#child-events (8.2.2024)
20. Aosong Electronics Co.,Ltd **Digital-output relative humidity & temperature sensor/module DHT22 (DHT22 also named as AM2302)** (10.2.2024)
21. Bosch Sensortec (2013) **Data sheet BMP180 Digital pressure sensor BMP180 Data sheet Document revision 2.5** (11.2.2024)
22. https://my.cytron.io/p-dht22-temperature-and-humidity-sensor/ (11.2.2024)