MINISTERUL EDUCAȚIEI

UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

DEZVOLTARE DE APLICAȚII MOBILE. COMPARAȚIE REACT NATIVE ȘI FLUTTER

LUCRARE DE LICENŢĂ

Absolvent: **Stefan Vasile MURESAN**

Coordonator ştiinţific: **Eng. Cristian VICAS**

2021

MINISTERUL EDUCAȚIEI
UNIVERSITATEA TEHNICĂ
DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE
DEPARTAMENTUL CALCULATOARE

Cuprins

1. Introducere

Odată cu creșterea performanței dispozitivelor mobile, acestea devin modul principal de comunicare, divertisment și gestionare a activităților de zi cu zi. Acest lucru duce la o necesitate a utilizatorilor de consumare a aplicațiilor calitative, rapide și diverse.

Pentru a putea satisface cererea utilizatorilor de pe platformele majoritare, Android si iOS, dezvoltatorii sunt obligați să dezvolte de multe ori aceeași aplicație în două limbaje native diferite care duce la implementarea funcționalităților similare de doua ori: utilizarea Java sau Kotlin pentru aplicațiile pe Android si Swift sau Objective-C pentru iOS. Deși dezvoltarea nativă aduce cu sine performanțe excepționale și amplifică experiența utilizatorului, aceasta necesită doua seturi de dezvoltatori pentru fiecare platformă, costurile fiind foarte ridicate.

Cu scopul de a reduce timpul de dezvoltare și costurile atașate acestuia, apar opțiuni pentru dezvoltarea multiplatformă. Aceste opțiuni permit utilizarea aceluiași cod sursa pentru lansarea pe mai multe platforme. Procesul de dezvoltare este unul mai scurt și mai puțin costisitor dar produce aplicații cu performanță mai scăzută fiind necesare straturi de traducere a codului multiplatformă în cel nativ. De aceea alegerea unui ecosistem multiplatformă trebuie sa fie una înțeleaptă, luand in considerare scopul final al aplicației și constrângerile.

Se pot menționa următoarele ecosisteme de dezvoltare multiplatformă:

- React Native
- Flutter
- Ionic
- Xamarin

Avand optiuni multiple, apare o decizie în ceea ce privește soluția de dezvoltare potrivită care să scurteze timpul de dezvoltare și costurile.

1.1 Sumarul proiectului

În acest proiect se dorește compararea a doua opțiuni de dezvoltare populare, mentionate mai sus, din punctul de vedere al dezvoltatorului cu scopul final de a contura o imagine de ansamblu a avantajelor și dezavantajelor utilizării uneia în detrimentul celeilalte. Se va dezvolta o aplicație utilizând ecosistemele React Native si Flutter care să implementeze funcționalități elementare precum compoziția unei aplicații și stilul acesteia, dar și funcționalități mai complexe de obicei disponibile în dezvoltarea nativă.

Pentru funcționalitățile mai complexe am decis utilizarea perifericelor dispozitivului mobil pentru executarea unor sarcini simple cu posibilitatea de

îmbunătățire ulterioară. Pe baza acestui proces de dezvoltare în doua ecosisteme diferite se vor trage concluzii legate de performanță, stil și ușurința în dezvoltare.

2. Obiectivele proiectului

2.1 Tema proiectului

Tema acestui proiect este examinarea a doua ecosisteme diferite de dezvoltare a aplicațiilor utilizate pe dispozitivele mobile, cu scopul de a prezenta asemănările și diferențele din perspectiva dezvoltatorului. Cele doua soluții alese sunt React Native, un framework JavaScript, si Flutter, care este un kit de dezvoltare software scris în Dart, un limbaj de programare orientat pe obiect, bazat pe clase, atat cu tipizare dinamica indulgenta(permite modificarea tipului odată setat în timpul rulării) cât și statică(posibilitatea adăugarii unui tip).

Pentru o mai bună exemplificare și posibilitatea de a oferii rezultate experimentale, se vor dezvolta doua aplicații, fiecare implementând una din soluțiile menționate mai sus. Aplicațiile vor avea ca și scop principal, utilizarea perifericelor telefonului mobil:

- NFC – Near-field communication
- BLE – Bluetooth Low Energy
- Camera de fotografie
- Senzori de mișcare – Accelerometru și Giroscop
- GPS – locația curentă

Cele doua soluții de dezvoltare vor fi trecute prin prisma mai multor termeni de comparație pentru o mai bună conturare a deciziei finale în ceea ce privește alegerea uneia dintre acestea în detrimentul celeilalte pentru implementarea cerințelor unui client. Acești termeni de comparație vor scoate în evidență atât argumentele pro cat si cele contra. O soluție aleasă corect, pentru un caz anume, poate oferii un avantaj dezvoltatorului prin scurtarea timpului de îndeplinire a sarcinilor și furnizarea timpului necesar pentru creșterea performanței și eficacității aplicației.

Criteriile de comparatie menționate mai sus sunt:

- Ușurință în dezvoltare
- Popularitate
- Portabilitate
- Suport pentru backend
- Performanță
- Estetică
- Access la functiile native ale platformei (Android/iOS)

2.2 Obiectivele proiectului

Principalul obiectiv al proiectului este dezvoltarea unei aplicații utilizând doua ecosisteme diferite și compararea acestora cu ajutorul criteriilor menționate mai sus. Dezvoltatorii au la dispoziție o sumedenie de soluții de dezvoltare, printre care React Native și Flutter, și de aici necesitatea alegerii potrivite pentru un anume proiect. De asemenea, pentru a ajuta la îndeplinirea obiectivului principal se definesc următoarele obiective secundare prezentate în tabelul de mai jos:

r.	Obiectiv secundar	Descriere
	NFC	Citirea informațiilor de pe un card NFC (Ex: badge access companie)
	BLE	Scanarea și afișarea informațiilor despre dispozitivele bluetooth compatibile din apropiere.
	GPS	Citirea latitudinii și longitudinii locației curente și afișarea țării, orașului și străzii.
	Camera	Implementarea acțiunii de a face o poza, afișarea într-o galerie și ștergerea acestora.
	Senzori	Afișarea valorilor în timp real a accelerometrului și a giroscopului.

Tabelul 1.1: Obiective secundare

2.3 Cerințe funcționale

Aplicațiile vor avea următoarele cerințe funcționale principale:

- **Securitate:** Accesul la aplicație se face după înregistrare și autentificare
- **NFC:** la apasarea unui buton de enable, se citește cardul NFC dacă acesta este disponibil
- **BLE:** la apasarea unui buton de enable, se afișează dispozitivele Bluetooth din apropiere
- **GPS:** la apăsarea unui buton se obține locația curentă sub forma de coordonate
- **GPS:** la apăsarea butonului de salvare, locația curentă se salvează în baza de date
- **Camera:** la apasarea unui buton, se realizează o fotografie care mai apoi se salvează
- **Senzori:** la apăsarea unui buton de enable, se afișează valorile în timp real a accelerometrului și a giroscopului
- **Senzori:** la apăsarea unui buton de disable, valorile senzorilor nu se mai actualizează

2.4 Cerinte non-functionale

Aplicațiile vor avea următoarele cerințe non-functionale principale:

- **Securitate**: dacă datele de autentificare sunt greșite sau nu există, accesul la aplicație nu este permis
- **Performanța**: timpul de răspuns la afișarea datelor și procesarea comenzilor va fii sub 1s. Aceasta este una din cele 3 bariere definite de catre Jakob Nielsen: în acest timp de răspuns, utilizatorul poate observa întârzierea dar nu este afectat de către acesta.
- **Portabilitate**: aplicația poate fi utilizată pe toate dispozitivele care suportă nivelul API 29 sau mai ridicat
- **Limba**: aplicația va fii dezvoltată în limba romană cu posibilitatea de introducere a limbii engleze într-o dezvoltare ulterioară
- **Utilizabilitate**: aplicația trebuie sa fie ușor de folosit, intuitiva și să ofere utilizatorului o experientă plăcută
- **NFC:** dacă nu se găsește nici un NFC card timp de 5 secunde, căutarea este oprită
- **BLE:** căutarea de dispozitive din apropiere se face pentru 5 secunde
- **BLE:** același dispozitiv nu este afișat de mai multe ori
- **GPS:** la navigarea pe ecranul specific GPS-ului, locațiile stocate anterior în baza de date sunt afișate

3. Studiu bibliografic

3.1 Istoria telefoanelor mobile

Istoria dezvoltării aplicațiilor mobile este strâns legată de istoria telefoanelor mobile. Înainte de concretizarea termenului de "smartphone", primul telefon mobil comercial a fost Motorola DynaTAC 8000X, adus în ochii publicului larg printr-o demonstrație făcută chiar de către Martin Cooper, cel care a condus echipa de dezvoltare și producere a telefonului, cum e mentionat in articolul [1]. Deși telefonul acesta avea dimensiuni impresionante, neputand fii comparat cu telefoanele mobile moderne, a fost revoluționar fiind primul telefon care nu necesita sa fie instalat într-o masina sau purtat în serviete masive și nu necesita un operator ca intermediar pentru efectuarea unui apel, utilizatorul putea sa facă asta de unul singur. Acest prim dispozitiv de comunicare portabil avea o baterie care necesita 10 ore de încărcare, pentru 30 de minute de comunicare și avea integrată doar o singura "aplicatie", cea de contacte, unde utilizatorul putea sa stocheze până în 30 de numere diferite. [2]

Primele telefoane mobile își îndeplineau doar rolul de adjuvant al comunicarii, dar urmașii acestora au început sa introduca ceea ce numim noi azi aplicații deși sub un alt nume, cel de caracteristici. Primele astfel de caracteristici au fost ceasul digital, calculatorul, alarma, lista de contacte si chiar e-mail. Acestea au fost introduse cu primul telefon "deștept" care avea sa includa funcțiile unui PDA (asistent personal digital), "Simon Comunicator Personal" promovat in tandem de IBM si BellSouth in anul 1993 [3]. Deși Simon avea primele functionalitati ale unui telefon modern, acesta era în continuare de dimensiuni mari.

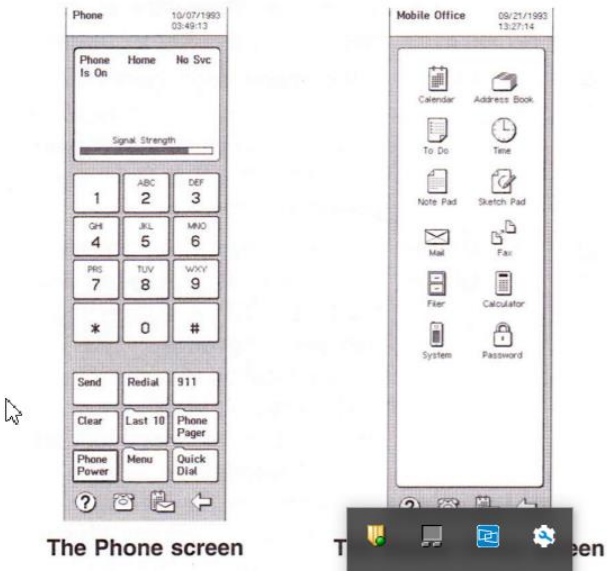

Figura 2.1: Cele doua ecrane principale ale telefonului Simon(add ref. user man.)

Nevoia unor telefoane mult mai portabile, a dus la apariția telefonului considerat de mulți ca fiind primul de dispozitiv cu clapeta. Motorola StarTAC, aparut in anul 1996, a demonstrat importanța stilului în realizarea unui telefon, înainte de acesta accentul cădea pe functionalitate, nu si pe estetica. Telefonul cu clapeta, de dimensiuni reduse de la Motorola, a pavat drumul pentru telefoanele moderne atât ca și "asistent personal" cât și ca accesoriu. [3]

Un alt telefon mobil de cotitură este anunțat în anul următor, Nokia 6110. Acesta este primul telefon cu celebrul joc pe telefon, "Snake" preinstalat asa cum este mentionat in articolul. Deși nu a fost primul joc introdus pe un telefon mobil (Tetris, pe telefonul mobil Hagenuk MT-2000, 1994 [5]) acesta a fost un succes incredibil. Inițial dezvoltat ca și joc video, odată cu introducerea lui pe telefonul produs de cei de la Nokia, acesta și-a recâștigat popularitatea avuta în anii 70'.[4] Snake este considerat de mulți ca fiind jocul care a deschis porțile unei noi industrii, cea a jocurilor pe telefon, o industrie care în anul 2020 a adus venituri de 77.2 miliarde de dolari.

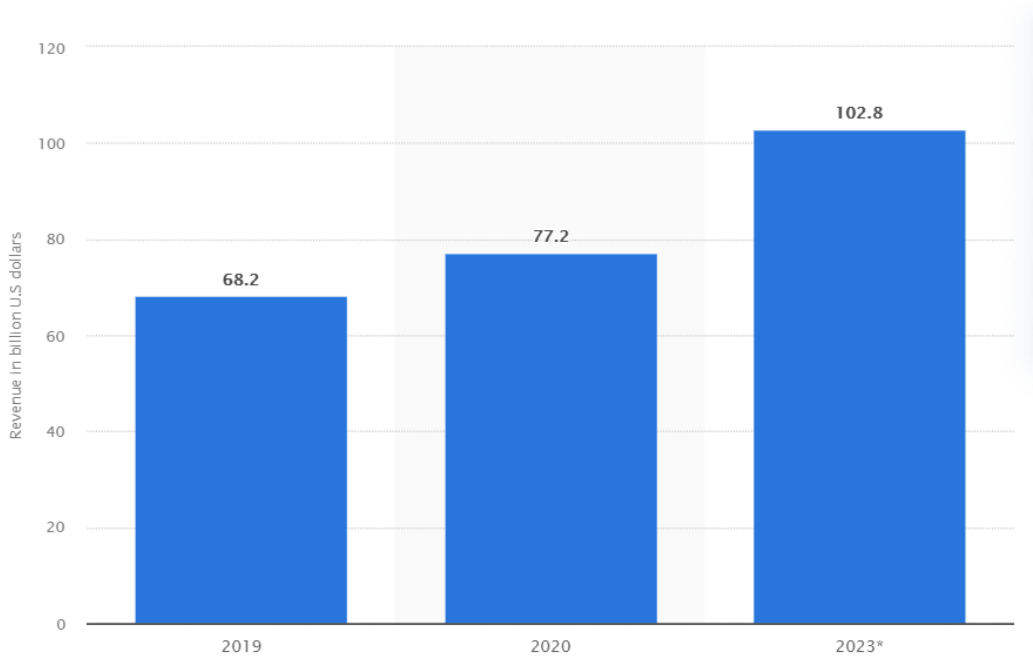

Figura 2.2: Venituri anuale estimative pentru industria jocurilor pe telefon

3.2 Internetul pe mobil

După apariția primelor telefoane cu funcționalități PDA și cu scop final divertismentul utilizatorilor, aceștia din urmă pun o presiune indirectă pe companiile de

dezvoltare pentru a crește numărul caracteristicilor și a jocurilor disponibile. Din lipsa resurselor necesare pentru a satisface necesitățile tuturor utilizatorilor, companiile caută o soluție pentru a putea oferii clienților un portal pentru divertisment și servicii fără ca aceștia să aibă acces direct la telefon.

Se adopta ideea utilizării internetului pentru împărtășirea de conținut pentru divertisment dar accesul direct la acesta era puțin probabil. Telefoanele mobile încă sunt îngrădite de specificațiile joase ale acestora:

- ecrane mici monochrome
- rezoluție scăzută
- memorie limitata (volatilă și nevolatilă)
- putere de procesare redusă
- periferice de intrare diferite
- mai puțină lățime de bandă
- latență ridicată

La sfârșitul anilor ‘90, site-urile web profesionale erau pline de culoare, încărcate cu text, imagini și filmări video. Dispozitivele mobile nu suportau operațiunile de transmitere a datelor venite din rețeaua globală de calculatoare interconectate. Chiar dacă rețelele își imbunatatesc capacitatea de a transmite o lățime de banda mai mare pentru telefoanele mobile, acestea vor fii afectate de consumul mai ridicat de baterie, ceea ce duce la o limitare a vieții bateriei. O soluție era necesară care să țină cont de specificațiile dispozitivelor, limitarea rețelelor și să țină cont și de standardizare, o aplicație să poată fi folosită pe o gamă largă de echipamente.

Pentru a adresa problemele menționate mai sus, marile companii au început să dezvolte standarde independente dar au ajuns să realizeze ca ar fii mult mai productiv și mai puțin costisitor, financiar și din punct de vedere al timpului, să lucreze împreună la un standard comun. Așa a luat naștere forumul WAP, un conglomerat format din Nokia, Ericsson, Unwired Planet, Motorola și nu numai, care au dezvoltat standardul WAP.

Standardul WAP, asa cum mentioneaza articolul [6], este un standard tehnic pentru accesarea informației de pe rețelele fără fir de telefonie mobile. Scopul principal al acestuia era de a aduce tehnologii multiple sub același standard și de a optimiza utilizarea internetului pe telefon, oferind utilizatorilor o experiența mult mai plăcută. WAP era o versiune mai simplistă a HTTP, care este fundația pentru comunicarea de date pe internet. Acest standard tehnic era bazat pe standarde ale Internetului deja existente, precum XML si IP si folosea serverele web HTTP 1.1 pentru a putea pune la dispoziție conținut create cu sintaxa dezvoltata de catre forum. WAP defineste o sintaxa bazata pe XML (eXtensible Markup Language) numita WML (Wireless Markup Language).

Cu ajutorul WML, marii distribuitori de conținut pe Internet au creat site-uri web cu utilizare specifica pe telefoanele mobile. Utilizatorii cu dispozitive capabile WAP, pot accesa doar site-urile web realizate cu aceasta sintaxa. Acestea sunt mult mai simpliste din punct de vedere al design-ului decat paginile WWW(World Wide Web), in

pare parte formate doar din text, imagini de dimensiuni reduse și optimizate în așa fel incat informatia sa fie sintetizata pentru ecranele mult mai mici ale telefoanelor mobile. Pentru a le putea accesa, producătorii au create navigatoarele web WAP, cu un design care permite rularea în limitele de memorie și constrângerile de lățime de banda de pe telefon.

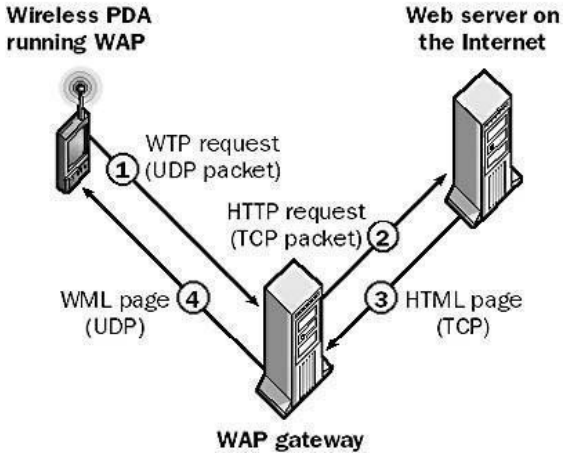

Figura 2.3: Reprezentare a protocolului WAP

WAP parea sa fie solutia perfecta. Producătorii puteau sa creeze navigatoare web preinstalate pe telefoanele mobile și se puteau baza pe programatori sa conceapa conținutul dorit de către utilizatori fără sa aibă acces la telefonul propriu-zis. Operatorii de telefonie mobile ofereau un portal personalizat WAP pentru redirectionarea clienților la conținutul pe care aceștia doreau să-l ofere, aducand profituri uriașe datorate costurilor mari associate cu navigarea pe internet. Dar utilizatorii nu erau multumiti, facand o comparatie direct cu navigarea pe internet. Programatorii și distribuitorii de conținut nu s-au ridicat la inaltimea asteptarilor [7]:

- Comercializarea de aplicatii WAP era dificila. Nu exista un mechanism de plata integrat, plata se face cu ajutorul SMS, EMS, MMS.
- Programatorii nu puteau sa dezvolte o experiența personalizată pentru utilizator. Majoritatea paginilor WAP aveau doar o singura versiune, indiferent de specificatiile telefonului.

Nici utilizatorii nu erau multumiti, isi doreau mai mult. Navigarea pe internet era frustrantă și lentă, introducerea de URL-uri cu tastatura numerica era o corvoada. Producătorii de dispozitive mobile realizeaza necesitatea schimbării politicii de protectie a detaliilor interne despre telefoanele mobile și expunerea acestora la publicul larg într-o oarecare masura.

Avand popularitate ridicată la începutul anilor 2000, în următorii ani WAS a fost inlocuit de standard moderne, navigatoarele web suportand în întregime HTML, CSS si in mare parte JavaScript. Deși a fost înlocuit de soluții mai moderne, WAP a adus oportunitatea de care aveau nevoie distribuitorii de servicii mobile, pentru a oferii suportul necesar pentru aplicatiile web precum: posta electronica, noutati din sport, noutati din lume, prețurile acțiunilor etc.

3.3 Sisteme de operare pentru dispozitivele mobile

Odată cu expunerea detaliilor tehnice ale dispozitivelor mobile, au început să apară sistemele de operare private, aflate în proprietatea unor companii care nu produc telefoanele propriu-zise. Această apariție a dus la o dezvoltare rapidă a numeroase sisteme de operare pentru telefoanele "inteligente", PDA-uri și sisteme de gaming de dimensiuni reduse.

Industria sistemelor de operare pentru telefoane, a devenit fragmentată. Fiecare companie avand propria bucată din piața. Majoritatea acestor platforme se foloseau de diferite programe de dezvoltare, diferite limbaje de programare, pe platforme diferite pentru crearea de aplicații. Comunitatea de programatori era obligată sa achiziționeze aceste unelte adjuvante dezvoltării de la companii. Aceasta diviziune de piață, a îngreunat munca producătorilor. Nu exista o platformă care să acopere nevoile întregii piețe, toate aveau atat beneficii cât și limitări. De aceea, liniile de productie au devenit complexe, într-o perioada scurtă de timp. Pentru a putea sa reziste competiției, aceștia au fost forțați sa vândă telefoane cu platforme diverse.

3.3.1 PalmOS

Asa cum este mentionat in articolul [8], compania Palm Computing a fost infiintata în 1992 și inițial se ocupă cu dezvoltarea de aplicații software pentru PDA-urile nou apărute. Dupa o fuziune cu U.S Robotics, compania a fost achizitionata de catre 3Com și a devenit o filială a celei din urma mentionate.

In anul 2002, odată cu trecerea de la PDA-uri la telefoane "inteligente", Palm creeaza o filială care sa se ocupe exclusiv de dezvoltarea unui sistem de operare, numit PalmSource sau PalmOS. Aceasta mai apoi devine o entitate independenta, in anul 2003. În următorii 3 ani compania trece prin diferite schimbări, in anul 2005 fiind achizitionata de ACCESS dar reușește sa obtina drepturi depline asupra mărcii "PalmOS", în așa fel incat doar ei puteau sa lanseze noi actualizari a sistemului de operare sub numele de PalmOS. Ca si consecinta, ACCESS sunt nevoiți sa schimbe numele sistemului de operare în Garnet OS.

PalmOS a fost inițial utilizat pentru PDA-uri produse de catre Handspring [9], companie fondata de catre fondatorii Palm Computing, datorită unor diferențe de viziune după achiziționarea Palm Computing de către 3Com. Incepand cu seria Treo, Handspring începe sa produca telefoane inteligente. Primele doua telefoane lansate de către aceștia sunt Treo 180 si Treo 180g utilizand PalmOS 3.5.2H. Treo 180 utiliza o tastatura completa iar modelul 180g se folosea de tehnologia de recunoaștere a scrisului de mana, Graffiti. Ambele telefoane fiind telefoane cu clapetă.

Toate versiunile predecesoare PalmOS 5.0, rulează pe procesoare Motorola/Freescale DragonBall iar toate versiunile ulterioare rulează pe procesoare bazate pe arhitectura ARM. PalmOS a fost implementat pe o multitudine de dispozitive precum: telefoane inteligente, ceasuri inteligente, console de jocuri de dimensiuni mici, GPS etc.

Cateva din functionalitatile cheie [9] ale ultimei versiuni de PalmOS Garnet, lansata in 2007, sunt:

- Interfață simplă, cu un singur task care permitea lansarea în modul fullscreen a aplicațiilor
- Ecrane monocrom sau color cu rezoluții de până la 480x320
- Sistem de recunoaștere a scrisului de mana (Graffiti 2)
- HotSync – tehnologie utilizată pentru sincronizarea datelor cu calculatoarele personale
- Capacitate de redare și înregistrare sunet
- Securitate simplistă: dispozitivul poate fi blocat cu parolă
- Port USB, infraroșu, Bluetooth și conexiune Wi-Fi
- Suport pentru card de memorie
- Set de aplicații standard: Contacte, Calculator, Calendar, Urmarire cheltuieli, Lista de sarcini
- Blazer – navigator web

Dezvoltare aplicații: Cum este mentionat in articolul [10], aplicațiile PalmOS Garnet sunt dezvoltate în mare parte in C/C++ dar exista instrumente care nu necesita limbaje de programare de nivel scazut precum PocketC, AppForge Crossfire(care utilizeaza Visual Basic sau C#). De asemenea o masina virtuala Java a fost disponibilă pentru platforma PalmOS până în 2008. Pentru dezvoltarea cu C/C++, exista doua compilatoare oficiale, CodeWarrior și un lanț de instrumente cu sursa deschisa “prc-tools”, bazat pe o versiune mai veche a gcc. Pentru testarea și depanarea aplicațiilor, PalmOS pune la dispoziție emulatorul POSE.

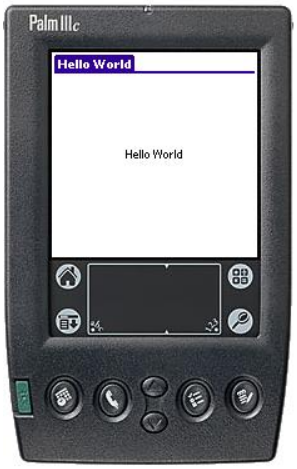

Figura 2.4: Program "Hello,World!" in PalmOS

3.3.2 Symbian OS

Considerat de mulți ca fiind primul sistem de operare modern pentru telefoane mobile, SymbianOS a fost cea mai folosită platformă pentru telefoanele inteligente din lume până în anul 2010. Începând ca și EPOC, un sistem de operare dezvoltat de către Psion, devine SymbianOS în anul 1998 fiind susținut de marii producători de telefoane mobile precum Nokia, Ericsson și Motorola. A deschis porțile cu adevărat spre telefonul inteligent modern, conceptul de ecran de dimensiuni ridicate care rulează aplicații multiple. In anul 2006, Nokia împreuna cu Symbian conduceau piața de telefonie mobilă. Fara sa aiba o competitie majora(PalmOS și Windows Mobile erau considerați companii mici în comparație cu acest gigant), Symbian ajunge să dețină 67% din piața. [11]

In 2008, Nokia achizitioneaza Symbian și într-o încercare de a atrage programatorii, pentru a dezvolta aplicații terțe, eliberează codul sursa sub o licență sursa deschisa. Deși avea o bucata uriașă din piața de telefonie mobile, aplicațiile erau greu de dezvoltat pentru Symbian, în primul rand din cauza complexității singurelor limbaje de programare native OPL și Symbian C++ și în al doilea rand din cauza prețurilor ridicate a mediilor de dezvoltare și a pachetelor de dezvoltare software. Toate acestea au descurajat dezvoltatorii de aplicații terțe și au stopat evolutia naturala a sistemului de operare spre un spatiu de desfacere a aplicațiilor pentru utilizator, un magazin digital de aplicatii.

Deși Symbian a cochetat cu deschiderea unui magazin de aplicații și cu extinderea sistemului de operare dincolo de telefonie mobilă, în direcția consolelor de jocuri, nici una dintre aceste idei nu s-a concretizat. Pe langa cele mentionate mai sus, Symbian este

tot prima companie care realizeaza importantă ecranelor de dimensiuni ridicate si cu suport la atingere. Symbian nu reușește sa fructifice avantajul de a fii prima companie capabilă de a implementa aceste caracteristici într-un sistem de operare.

Symbian avea un ecosystem care îmbrățișa dezvoltatorii de aplicații terțe, chiar avand o inițiativă de aprobare a aplicațiilor mult mai rapid, Symbian Signed, eliminand nevoia de a fii testate de către ei înainte. In anul 2008, erau disponibile 10000 de aplicatii Symbian. Dar neavand un magazin digital să țină locul unei piețe de desfacere, un singur loc de unde utilizatorii pot să achiziționeze aplicații dezvoltarea era una lentă. Ca si comparatie, Apple a ajuns la incredibilul număr de 100000 de aplicații în puțin peste un an de la lansarea primului kit de dezvoltare software.

Dezvoltare aplicații: Cum este mentionat in articolul [12] SymbianOS fiind scris in C++, folosind propriile standard de dezvoltare, dezvoltarea aplicațiilor este posibila folosind Symbian C++. Din pacate utilizarea acestui limbaj este una anevoioasa, curba de învățare fiind una abruptă, Symbian C++ necesitand tehnici speciale precup descriptori, obiecte active și stiva de curatare. O alta functionalitate lipsa este aceea de Excepții din standard C++, utilizandu-se mecanismul de “Trap and leave” ca și substitute. Ca și mediu de dezvoltare, primele versiuni folosesc IDE-ul commercial CodeWarrior care mai apoi este inlocui de Carbide.c++ sau Microsoft Visual Studio 2003/2005 prin intermediul conectorului Carbide.vs. Problemele de dezvoltare introduse odată cu utilizarea Symbian C++ dispar in 2010, cand pentru dezvoltare se poate utiliza C++ standard cu ajutorul Qt, un kit pentru crearea de interfețe grafice sau aplicații cu Qt Creator sau Carbide ca si in cazul Symbian C++.

Intr-un final, Symbian ajunge să fie depășit de către giganții Android și iOS, dezvoltarea de aplicații pentru Symbian fiind oprită în anul 2014.

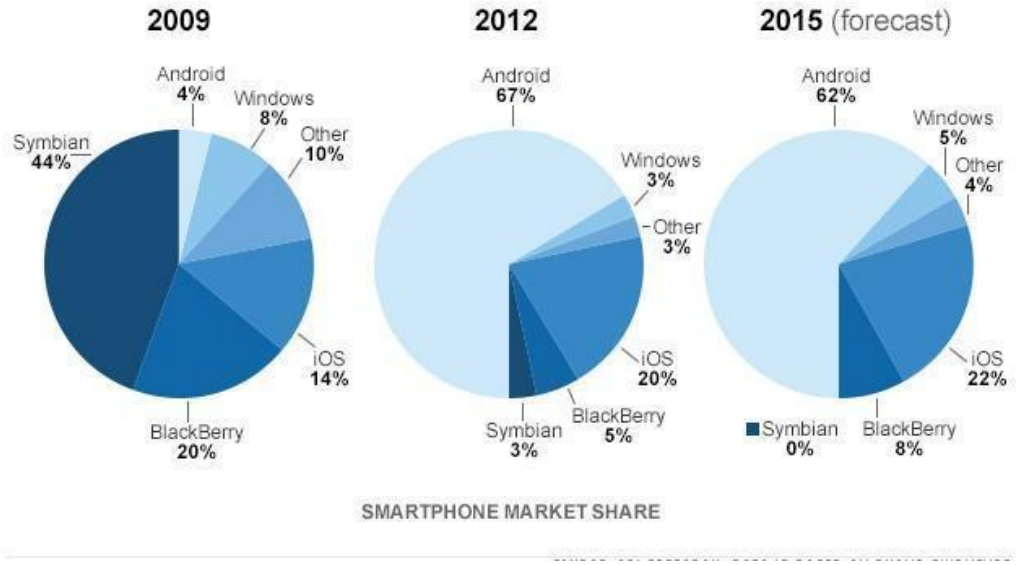

Figura 2.5: Cota de piata la nivel global între anii 2009-2015. A se urmarii Symbian [13]

3.3.3 iPhoneOS (iOS)

Sistemul de operare pentru dispozitive mobile, dezvoltat de cei de la Apple Computer, a fost anunțat în 2007 de către Steve Jobs co-fondatorul companiei împreuna cu Steve Wozniak. In același timp, compania este redenumită doar Apple, marcând solidificarea mărcii.

În timpul anunțului oficial, se subliniază faptul ca OS-ul pentru telefon împărtășește multe dintre aceleași caracteristici cu Mac OS X(calculatorul personal dezvoltat de Apple) și rulează pe același nucleu Unix dar că era destul de puternic să primească propria marcă. Inițial sistemul de operare a fost numit iPhone OS, păstrând numele acesta pentru 4 ani pana la lansarea iOS 4 în anul 2010.

Deși la data apariției, iOS era cu mult în urma competitorilor deja bine stabiliți pe piață precum Symbian, BlackBerry, Windows Mobile, PalmOS în materie de caracteristici, iOS reușește să ofere o experiența inedită, concentrându-se pe experiența centrală de utilizare a unui telefon: viteză, consistență și optimizarea celor cateva caracteristici prezente într-un fel în care le făcea radical mai bune decât orice altceva prezent pe piață. Noul telefon mobil cu noul sistem de operare avea să paveze drumul spre viitorul telefoanelor inteligente, subțiri și cu metoda principala de interacțiune, ecranul tactil.

Câteva dintre cele mai importante și revoluționare caracteristici ale iOS 1.0 sunt dupa cum mentioneaza articolul [14]:

- Interfața cu utilizatorul - Apple introduce atingerea ca mijloc principal de interacțiune a utilizatorului cu sistemul de operare, elimină majoritatea butoanelor și adaugă diferite functionalitati la atingere
- Navigatorul web Safari – funcționalitățile la atingere prin diferite gesturi menționate mai sus, sunt folosite în Safari, acesta fiind primul navigator web pe telefon care se simțea la fel de capabil și puternic ca unul de pe calculatoarele personale. În timp ce competiția adapta paginile web la dimensiunile telefonului, Safari prezenta întreaga pagină și oferind funcționalități de zoom și derulare simpliste cu ajutorul degetelor.
- Funcționalitatea de iPod – Apple introduce funcția de iPod pe un telefon inteligent, posibilitatea de a asculta muzica cu dispozitivul cândva utilizat doar pentru comunicare.
- Tastatura software – prima tastatură software care este disponibilă pe ecran doar cand e necesar, neocupând spațiu important care poate fi utilizat pentru alte funcționalități. Deși sisteme ca PalmOS aveau funcționalitatea de Graffiti deja introdusă, această tastatură putea fi folosită cu ajutorul degetelor.
- Introducerea ecranului principal trambulină – indiferent în ce aplicație te aflii, prin apăsarea butonului de “acasă”, navighezi la ecranul principal

Sistemul de operare are câteva actualizări intermediare după versiunea 1.0 dar una din cele mai importante lansări a fost cea a versiunii 2.0. Spre deosebire de Symbian, Apple nu pierde timpul și lansează “App Store”, un magazin digital cu 500 de aplicații inițial, deschis și pentru aplicațiile terțe. Nota inovativă este dată de faptul că magazinul de aplicații exista atât pe telefon, cât și în iTunes, platforma de muzică deja bine înrădăcinată. Utilizatorii deja înregistrați pe iTunes, puteau să înceapă să caute și să instaleze aplicații instant fără sa fie nevoiți să-și introducă datele bancare, lucru care a dus la o usurința în cumpararea aplicațiilor nemaivăzută până în acel moment. Un an mai târziu se adaugă și functionalitatea de cumpărături în interiorul aplicațiilor, devenind rapid principala metodă de monetizare a aplicațiilor/jocurilor.

O alta caracteristică majoră introdusă odată cu versiunea 2.0 este kit-ul de dezvoltare software iOS cu o suită de instrumente pentru dezvoltatorii de aplicații, pentru a putea crea aplicații mai funcționale, mai estetice si mai avansate decat a altor competitori. Cele doua mari functionalitati introduse merg mana-n mana.

Introducerea magazinului digital și a kitului de dezvoltare, vine și cu anumite reglementări atat pentru utilizatori cat si pentru dezvoltatorii de aplicații. Utilizatorii nu au permisiunea de a instala aplicațiile prin alta metoda decat prin cea a magazinului digital. Dezvoltatorii de aplicații trebuie sa respecte o suita de reguli, pentru a-si putea vedea creațiile încărcate în magazine. Unele reguli fiind înțelese de utilizatori dar unele au starnit controverse, precum respingerea continuă a aplicațiilor care ar permite conectarea calculatorului personal la iPhone pentru acces la internet.

Ca și un prim pas în distribuirea de aplicații, dezvoltatorii trebuie sa se înscrie în așa numitul program dezvoltator Apple, printr-un abonament anual de 99$. Odată intrat în acest program, dezvoltatorul are acces atât la magazinul digital pentru distribuire cât și la caracteristica "zbor de test" (TestFlight) utilizată pentru testarea aplicației. Este posibilă invitarea printr-un link sau e-mail a până la 10.000 de testeri.

Odată intrat în programul de dezvoltare Apple, dezvoltatorii trebuie să se asigure că aplicațiile lor respectă suita de reglementări, atât de stil cât și de funcționalitate. Dacă aplicația respectă aceste reguli, ea trebuie să fie semnată, lucru necesar pentru a asigura utilizatorii că aplicația vine dintr-o sursă sigură și nu a fost modificată de la ultima "semnare". Semnarea se poate face atât automat cât și manual cu Xcode(mediul de dezvoltare integrat Mac).

Pentru a putea face încărcarea propriu-zisă și a gestiona aplicația după încărcare, se folosește suita de instrumente "App Store Connect" unde se creează o înregistrare a aplicației. Aici dezvoltatorii au posibilitatea de a integra plata și de a accesa date analitice după ce aplicația este deja disponibilă în magazinul digital.

Ultimul pas este de a incarca aplicatia pe "App Store Connect", unde sunt necesare date relevante despre aplicație, printre care și documentele pentru termeni si conditii. Odata completate aceste date, aplicația poate fi trimis pentru verificare și aprobare. Acest proces durează între una și trei zile în majoritatea cazurilor și dezvoltatorii pot primii doua răspunsuri:

- Aplicatie aprobata: în maxim 24 ore aplicația o sa fie vizibilă în magazinul digital
- Aplicatie respinsa: dezvoltatorul poate sa facă reparațiile necesare și să o trimită din nou spre verificare sau poate sa escaleze și să facă apel la decizie.

De-a lungul anilor, Apple a continuat sa imbunatateasca, sa adauge si sa revolutioneze caracteristicile telefoanelor mobile. Deși interfața utilizatorului s-a îmbunătățit, se pastreaza aceeasi interacțiune fără asperități. La momentul actual, ultima versiune de iOS este 14.6.

Dezvoltarea aplicațiilor [15]**:** Inițial dezvoltarea aplicațiilor pe iOS, s-a realizat utilizând Objective-C, un limbaj de programare care combină avantajele a doua limbaje mai vechi, C si Smalltalk. Objective-C a fost adoptat de către Apple odată cu achiziționarea companiei NeXT, companie fondata chiar de catre Steve Jobs dupa acesta a fost obligat sa renunte la rolul său în compania Apple Computer. Instrumentele de dezvoltare de la NeXT utilizau Objective-C, instrumente care mai apoi au fost integrate in Xcode, mediul de dezvoltare integrat de către Apple în macOS și utilizat mai apoi în dezvoltarea de aplicații și pentru iOS.

În combinație cu Xcode, se utilizează kitul de dezvoltare iOS. Acest kit se poate descărca gratuit pentru utilizatorii de Mac dar nu e disponibil pentru utilizatorii

Windows. Kitul conține seturi care oferă dezvoltatorilor de aplicații access la diferite funcții și servicii, pentru a putea folosii capacitățile software și hardware ale telefonului mobil. Pe deasupra acest pachet mai conține și un simulator pentru testare și depanare a aplicațiilor pe calculatorul personal Mac.

Datorită faptului ca Objective-C a rămas neschimbat din anii ’80 și ducea lipsa unor caracteristici moderne, Apple decide sa dezvolte un limbaj de programare pentru a-l înlocui. Asa ca in 2014 este anuntat Swift, care împreuna cu Cocoa si Cocoa Touch cele două schelete de dezvoltare pentru Mac, vor fii folosite de către dezvoltatori pentru realizarea noilor aplicații. Deși Swift a fost conceput ca înlocuitor pentru Objective-C, cel din urma încă este folosit pentru dezvoltarea de aplicații. În functie de necesitatile clientilor, Objective-C poate fi o optiune viabila datorită faptului ca este un limbaj matur, cu documentatie precisa. O functionalitate cheie a limbajului Swift e ca are access și se poate folosii de către codul deja scris în Objective-C.

Atat Swift cat si Objective-C sunt native pentru iOS și se folosesc în continuare ambele deși din lipsa de suport pentru dezvoltarea limbajului Objective-C, Swift o sa continue sa crească în popularitate.

Pe langa limbajele de programare native, dezvoltatorii de aplicații au la dispoziție și schelete de dezvoltare sau întregi kituri de dezvoltare software. Utilitatea lor este dată de faptul ca majoritatea se pot folosii pentru dezvoltarea aplicațiilor pe cele doua mari platforme Android și iOS, iar pana in 70% din cod poate fi refolosit, ceea ce poate fi un plus pentru partea financiara a dezvoltarii aplicatiilor. Ca si exemple avem, mentionate in articolul [16]:

- Ionic – schelet de dezvoltare bazat pe HTML, CSS si JavaScript, construit peste Cordova, care înglobează aplicația realizata cu HTML/JavaScript într-un container nativ pentru a putea utiliza functiile native precum: geolocatie, notificari, camera etc. Doua caracteristici demne de menționat ale Ionic sunt: Ionic Creator, un editor care permite crearea rapidă a unui prototip cu drag and drop și Ionic View, o aplicație gratuită pe Android și iOS care permite dezvoltatorilor sa impartasească aplicațiile cu utilizatori, testeri sau alți dezvoltatori fara a fii necesara incarcarea aplicației în magazinul digital.
- React Native – schelet de dezvoltare creat de către cei de la Facebook pe baza bibliotecii JavaScript, React. React Native permite utilizarea de bibliotecilor native și scrierea de cod în Objective-C, Swift sau Java pentru Android pentru optimizarea performanței.
- Flutter – kit de dezvoltare software creat de către cei de la Google și este utilizat pentru crearea de aplicații Android și iOS din același cod sursa. Flutter utilizează limbajul de programare Dart, are o documentatie bine pusa la punct si are acces la toate funcțiile native

3.3.4 Android

Compania Android Inc. a fost fondată in 2003 de 4 fondatori. Inițial dorindu-se utilizarea sistemului de operare pentru îmbunătățirea camerelor digitale, compania decide să schimbe obiectivul spre telefoane mobile. Odată cu aceasta decizie, în 2005 compania este achiziționată de către Google. Cei 4 fondatori rămân în continuare în companie să continue dezvoltarea noului produs. Google și echipa de dezvoltare Android decid sa utilizeze Linux ca și bază pentru acesta. Decizie care a fost luată pe baza dorinței de a oferi sistemul de operare gratuit producătorilor de telefoane mobile, având încredere în capacitatea financiară a altor servicii și aplicații. Pe lângă faptul că sistemul de operare avea să fie cu sursă deschisă, creatorii decid să lase cu sursă deschisă însă și logo-ul proiectului. [17]

La câteva luni de la lansarea sistemului de operare dezvoltat de către Apple, Android 1.0 este lansat pentru dezvoltatori și nici un an mai târziu este lansat telefonul care avea să fie primul dintre multe altele, cu sistemul de operare Android: HTC Dream. Deși dispozitivul mobil, nu se apropia de cel dezvoltat de catre Apple în materie de design și într-o oarecare măsură funcționalitate, Google și-a lăsat amprenta pe sistemul acesta de operare integrând produse și servicii dezvoltate de către companie precum: Google Maps, Youtube și un navigator web HTML, precursor Chrome, care folosea serviciile de căutare Google.

Pe lângă caracteristicile menționate mai sus, primul telefon Android introducea și prima versiune a pieței Android, un magazin digital de aplicatii, de unde utilizatorii puteau sa descarce aplicații și jocuri. Magazinul a adăugat suport pentru aplicațiile cu plată abia un an mai târziu. Spre deosebire de competitorul direct, iOS, dezvoltatorii puteau sa incarce aplicațiile pe magazinul digital mult mai ușor, Android avand mai putine reglementari si un proces mai putin complex si dificil.

In anii urmatori, două noi magazine au fost lansate de către cei de la Google: Google eBookstore si Google Music. Datorită faptului ca existenta magazinelor multiple putea sa cauzeze confuzie, în 2012 Google integrează cele 3 magazine într-unul singur, sub denumirea de Google Play Store. La lansarea acestei noi versiuni, magazinul avea peste 450000 de aplicații și jocuri Android, număr care avea sa creasca în perioada următoare.

La fel ca și competitorul său, Android avea sa aiba o imbunatatire continua, atat în materie de design al interfeței cât și a functionalitatilor și experienței utilizatorului. Ultima versiune disponibila de Android este Android 11 iar în decursul anului actual, Android 12 va ajunge pe piata. Datorită faptului ca sistemul de operare a fost oferit gratuit producătorilor de telefonie mobilă, Android deține procentul majoritar din piața de sisteme de operare pentru telefoane mobile, la momentul actual singurul sau competitor fiind iOS-ul. [17]

Deși una dintre caracteristicile sistemului de operare Android, care a dus la dominarea pieței (71% in ianuarie, 2021), este faptul că Google a lansat produsul cu sursă deschisă, totodată aceasta este una din problemele majore ale platformei, fragmentarea. Fragmentarea sistemului de operare Android, este dată de către multitudinea de dispozitive și versiuni ale acestuia. Fiind cu sursă deschisă, producătorii pot să modifice sistemul de operare după placul lor și acest lucru afectează adoptarea noilor versiuni de către utilizatori, acestea nefiind tot timpul disponibile pentru unele dispozitive mobile deoarece producătorii trebuiau să-și introducă mai intai propriile lor modificări. Google este afectat de acest lucru deoarece neavand o rata de adoptare a noilor actualizari ridicată, nu pot sa facă o analiza realista a felului în care nouă versiune afectează utilizatorii, ce probleme apar și ce caracteristici ar trebui dezvoltate în viitor. Aceasta fragmentare a ecosistemului afecteaza si dezvoltatorii care au doua opțiuni: dezvoltarea de aplicații cu suport pentru versiuni mai vechi, creșterea costurilor de dezvoltare, sau utilizarea ultimelor versiuni, pierzand clienti.

Ca si solutie la problema fragmentarii, Google propune proiectul Treble în 2017 care presupune o modificare a arhitecturii în așa fel incat actualizările sa ajunga mult mai ușor la utilizări, indiferent de producator. Soluție care după ultimele date pare sa functioneze, Android 8 avand o rata de adoptare la apariția versiunii 9 de 8.9% iar Android 9 avand o rata de adoptare de 22.6% la apariția Android 10. Eforturile proiectului Treble sunt continuate de către proiectul Mainline care are ca scop trimiterea de actualizari ale securității prin intermediul magazinului digital, eliminand astfel producatorii din ecuatie.[18] [19]

Dezvoltarea aplicațiilor: Dupa cum mentioneaza articolul [20], Pentru dezvoltarea de aplicații software pentru telefoanele mobile care rulează Android, Google afirmă că aplicațiile pot fi scrise utilizând Kotlin, Java sau C++ utilizând kitul de dezvoltare software de la Android. Toate limbajele care nu rulează codul pe JVM, precum Go, JavaScript, C, C++, au nevoie de cod scris în limbaj JVM care poate fi obținut cu ajutorul unor anumite instrumente deși Google pune la dispoziție și kitul de dezvoltare nativ(NDK), care permite scrierea părților unde performanța este critică, în cod nativ C/C++ fără a avea nevoie de mecanisme intermediare precum JVM, codul fiind compilat direct. Această optiune nu o sa fie benefică tuturor aplicațiilor, crescând complexitatea codului.

Kitul de dezvoltare de la Android conține un set de instrumente de dezvoltare: un depanator, biblioteci, un emulator de telefon mobil bazat pe QEMU, documentație, monstre de cod si tutoriale. Spre deosebire de competitorul său, Android permite dezvoltarea și pe platforma Mac OS X.

Ca și mediu de dezvoltare oficial suportat pentru Android a fost Eclipse până în 2014, utilizând un conector de instrumente de dezvoltare Android. Incepând cu 2015, mediul de dezvoltare integrat este Android Studio, suportul pentru Eclipse încheindu-se. Ca si alternativă, dezvoltatorii pot sa folosească orice editor de text pentru modificarea fișierelor Java sau XML, după care să utilizeze instrumentele din linia de comanda pentru

a crea, construi și depana aplicațiile Android dar și de a controla dispozitivele Android conectate la calculatorul personal. Pentru distribuire, aplicațiile Android sunt împachetate în format .apk și stocate în directorul app al Android OS, director accesibil doar utilizatorului root din motive de securitate.

La fel ca si in cazul iOS, dacă nu se dorește utilizarea limbajelor de programare native pentru dezvoltarea de aplicații pentru telefon, dezvoltatorii au la dispoziție dezvoltarea multiplatforma. Utilizarea unui schelet sau a unui kit de dezvoltare pentru scrierea de cod refolosibil pe ambele platforme, reducerea costurilor și a timpului de dezvoltare. Pe lângă opțiunile menționate în subcapitolul iOS, mai avem:

- PhoneGap/Cordova – similar cu Ionic deoarece permite construirea de aplicații mobile cu tehnologii web și e construit cu Cordova la baza dar spre deosebire de Ionic, care este legat de Angular, PhoneGap nu este legat de un schelet JavaScript anume, oferind dezvoltatorilor mai multă versatilitate
- Xamarin – aplicațiile dezvoltate cu Xamarin sunt scrise în totalitate în C#, acesta compilează codul în distribuiri native pentru Android și iOS. Xamarin are la bază pe scheletul Mono, o implementare cu sursă deschisă a .NET. Aplicațiile Xamarin au acces la toate capabilitațile native ale dispozitivului, după compilare comportându-se ca o aplicație nativă. Deși există un grad ridicat de împărtășire a codului, uneori o sa fie necesara scrierea de cod specific Android/iOS.
- Progressive Web Apps – o alternativă adusă de către Google și se dorește utilizarea aplicațiilor web ca aplicații native. Nu este necesară instalarea lor, ci odată ce utilizatorul viziteaza această aplicație web, va avea posibilitatea de a o adăuga pe ecran.

Pentru a încarca aplicația în magazinul digital Android, dezvoltatorii trebuie sa creeze un cont de dezvoltator. Acest lucru este posibil cu contul Google deja existent sau prin crearea unui cont nou. Spre deosebire de iOS, in cazul Android se plateste o suma unica de 25$. Dacă aplicația se dorește a fi una cu plata, un cont de comerciant trebuie creat de asemenea.

Dupa crearea acestui cont, este necesară pregătirea documentelor de termeni și condiții și politică de confidențialitate. De asemenea, dezvoltatorul trebuie să treacă prin toate politicile de dezvoltare Google pentru a se asigura ca aplicația e în conformitate cu standardele Android si a nu fi respinsă la verificare. Spre deosebire de iOS, aceste politici sunt mai permisive.

Următorul pas din proces este de a verifica necesitățile tehnice, printre care semnarea aplicației cu un certificat care este folosit pentru identificarea autorului și nu poate fi generat din nou, dimensiunile aplicației și formatul de lansare (.apk sau .aab).

Ca și ultim pas, dezvoltatorul trebuie sa creeze aplicația cu consola Google în magazin, sa adauge informațiile despre aceasta precum: descriere, capturi de ecran, icoana, pret etc. și după sa incarce aplicația propriu-zisă ci să o trimită spre verificare. Există trei opțiuni: productie, alpha si beta.

Este recomandată încărcarea aplicației inițial sub forma alpha sau beta, deoarece odată ce verificarea este finalizată, aceasta o sa fie disponibilă doar pentru anumite tipuri de utilizator. Pentru alpha, aplicația o să fie disponibilă persoanelor pe care dezvoltatorul le invita la testare iar beta, pentru orice utilizator care dorește să intre în procesul de testare. Procesul de verificare durează aproximativ două zile, dar în ultima perioadă pentru o mai buna protectie a utilizatorilor, Google a anunțat că perioada poate fi extinsă până la o saptamană pentru a putea verifica în detaliu aplicațiile.

4. Analiză și fundamentare teoretică

4.1 React Native

4.1.1 Introducere

React Native este un schelet JavaScript utilizat pentru dezvoltarea de aplicații cu performanțe native, bazat pe biblioteca JavaScript, React. Unicitatea acestei soluții de dezvoltare este dată de faptul ca întreg codul sursă este utilizat de către cele doua mari platforme, Android si iOS, lucru care duce la o reducere a costului și a timpului de dezvoltare, simultan dezvoltând ambele aplicații.

Similar cu React, aplicațiile scrise în React Native folosesc un amestec de JavaScript și JSX, o extensie de sintaxă asemănătoare XML sau HTML dezvoltată pentru JavaScript cu scopul de a permite existența HTML-ului în cod, care mai apoi la rulare sunt convertite in elemente react. Partea nativă utilizând un limbaj diferit față de partea JavaScript, Objective C/Swift pentru iOS și Java pentru Android, React Native se folosește de așa numitul pod, care face legatura dintre cele doua părti ale aplicației. [21]

4.1.2 Arhitectura React Native

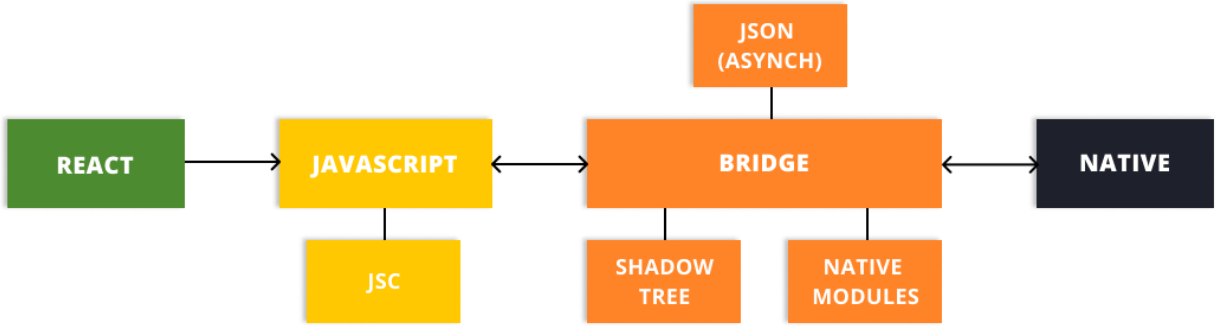

Figura 3.1: Reprezentare grafica a arhitecturii React Native

Asa cum mentioneaza articolul [22], aplicație React Native este compusă din două părți, componenta JavaScript și partea nativă. Pentru a se putea realiza comunicarea

dintre cele doua avem nevoie de biblioteca pod, fără aceasta nu se pot trimite informații între cele două.

Podul RN este scris în C++/Java și rolul său este de a transmite informația procesată de la JavaScript la partea nativă. Odată cu deschiderea unei aplicații React Native, un fir principal de execuție(UI thread) este creat care mai apoi generează alte doua fire: firul de execuție JavaScript, unde întreg codul RN scris este executat, și firul de executie umbra "shadow thread" sau "shadow tree" pentru a afișa compoziția aplicației. În firul JavaScript definim compozitiile, care sunt compilate și preluate de către firul umbra, unde se calculează, și mai apoi sunt trimise la firul de execuție principal(partea nativa). Pentru a trimite actualizări la partea nativă, se utilizează obiecte JSON care sunt înglobate și trimise la fiecare sfârșit de iterație a buclei de evenimente.

Există și posibilitatea codului JavaScript de a reacționa la un eveniment apărut pe partea nativă, utilizând conceptul de funcție declanșată de eveniment("callback"). O functie este scrisa in JavaScript, atașată unui obiect și este serializată și trimisă peste pod. Odată aparut un eveniment, acesta este trimis cu ajutorul funcției peste podul React înapoi la JavaScript pentru execuția codului specific evenimentului.

Pentru rularea codului JavaScript, React Native utilizează motorul JavaScriptCore(JSC), același motor care stă la baza navigatorului web de pe iOS, Safari. În cazul aplicațiilor iOS, RN utilizează JSC-ul integrat în platforma iar în cazul Android acesta vine odată cu aplicația, motiv pentru care aplicația simpla "Hello, World!" are dimensiune mai ridicată.

Pentru a putea păstra performanța, viteza și sentimentul de aplicație nativă, uneori este necesar să se acceseze o interfață de programare a aplicației (API) care nu este disponibilă în JavaScript sau chiar să fie creată de către dezvoltator pentru a crește performanța aplicației. Sistemul de module native dezvăluie instanțe ale claselor native Java/Objective-C/C++ sub forma unor obiecte JavaScript pentru a putea fi folosite în codul JS. Pentru scrierea unui modul nativ în sistem există două opțiuni:

- Direct in aplicația React Native, în proiectele Android/iOS
- Ca și packet NPM(node package manager) și instalat ca și dependință la aplicațiile RN

Ca și în cazul unui pod real, pot să apară blocaje. Un exemplu este afișarea animațiilor, din cauza podului RN și "distanței" pe care trebuie sa o parcurga evenimentele apar probleme care afectează performanța aplicației. In majoritatea cazurilor acest lucru poate fii optimizat cu ajutorul NativeDriver, care trimite animatiile pentru a fi procesate în partea nativă. Din cauza acestor probleme mentionate mai sus, echipa de la Facebook lucrează la o actualizare a arhitecturii lucrand la fiecare secțiune a ei în parte pentru a îmbunătăți performanța.

4.1.3 Modalităţi de implementare

Pentru dezvoltarea de aplicații, React Native vine cu două interfețe linie de comanda, ambele având atat beneficii cât și limitări și decizia utilizării uneia în detrimentul celeilalte, trebuie luată ținând cont de obiectivul final al aplicației și cerințele sau caracteristicile pe care dezvoltatorul trebuie sa le implementeze. Soluția aleasă este utilizarea interfeței linie de comanda Expo CLI, alternativa fiind React Native CLI. [23]

Expo CLI – Flux de lucru gestionat

Interfața linie de comanda oferită de RN, oferă o dezvoltare a aplicației rapidă, ocupându-se de schimbările de configurație, conflicte între versiuni și certificate. Pe langa acestea, Expo oferă posibilitatea de a împărtăși proiectul încă în faza de dezvoltare fără a genera fișierul .apk sau .ipa, prin trimiterea unui cod QR sau adresa. Aceeași funcționalitate poate fii folosită și pentru testarea aplicației pe telefonul personal, fără a fi necesară o conexiune directă la mediul de dezvoltare sau utilizarea unui emulator, lucru dificil mai ales în cazul dezvoltării aplicațiilor pentru iOS dacă nu se dispune de un calculator Mac. Acest lucru este posibil descărcând aplicația gratuită pentru Android și iOS de pe magazinul digital și doar scanand acel cod. De asemenea, Expo integrează și unele biblioteci elementare în proiectul standard precum notificari “push”, manager de resurse etc.

Pe lângă cele menționate mai sus, Expo CLI dispune de o modalitate ușoară de construire a fișierelor .apk/.ipa, lansare și depanare a aplicațiilor, aplicațiile Expo automatizand procesul de semnare a aplicațiilor atat pe iOS cat si pe Android, lucru care poate fii suprascris la nevoie.

Testarea aplicației cu ajutorul aplicației gratuite ExpoGo se face cu ajutorul interfeței linie de comanda Expo CLI. Aceasta înglobează rularea serverului de dezvoltare Expo și a serviciului de împachetare Metro, care combina tot codul JavaScript într-un singur fișier și resursele(imagini) in obiecte care pot fii afisate. Pentru ca acest lucru sa functioneze este necesar ca atat mediul de dezvoltare cât și dispozitivul mobil de testarea, sa fie in aceeasi retea.

Serverul de dezvoltare Expo, oferă o interfață între aplicația aflată pe dispozitivul mobil/simulator și ExpoCLI și de a trimite fișierul de manifest Expo. Acest server este punctul în care se ajunge prima data după scanarea codului QR sau introducerea adresei. Manifestul Expo contine toate configuratiile necesare pentru ca ExpoGo sa știe în ce fel sa ruleze aplicatia. În același fișier se afla și calea spre serverul de dezvoltare local care comunica direct cu serviciul de împachetare Metro.

La lansarea aplicației, aceasta preia mai intai fișierul manifest și după care pe baza caii spre serverul local, preia codul JavaScript împachetat într-un singur fișier de catre Metro.

Metro este folosit atât pentru aplicațiile realizate cu ExpoCLI cât și pentru cele cu React Native CLI și are doua roluri importante:

- combină tot codul JavaScript într-un singur fișier și translatează orice cod JavaScript pe care dispozitivul nu-l poate interpreta (precum codul JSX).
- resursele(ex: imagini) sunt transformate în obiecte care pot fi utilizate

Figura 3.2: Reprezentarea comunicarii dintre aplicația ExpoGo și mediul de dezvoltare

Dacă se dorește trimiterea spre testare la alți utilizatori, Expo oferă și funcționalitatea de publicare direct din aplicația lor. Codul JavaScript împachet de către Metro, resursele necesare și manifestul sunt salvate de către Expo pe CloudFront, serviciu de stocare date oferit de Amazon. Odata publicata aplicatia, dezvoltatorul primește o adresa pe care oricine cu aplicația ExpoGo instalată poate sa o folosească pentru accesarea aplicației și să o testeze. Din pacate exista si limitari, în cazul Android după publicarea aplicației, oricine cu adresa respectivă poate sa acceseze aplicația prin intermediul ExpoGo dar pe iOS din cauza restricțiilor impuse de acestea, acest lucru nu este posibil, cea mai buna opțiune ramane împachetarea aplicației într-un .ipa si lansarea din magazinul digital, eventual folosirea functionalitatii "zbor de test" pentru testare.

Pe aceeași notă a confidențialității, în fișierul de configurare al proiectului se poate seta starea aplicației: nelistată, doar cei cu adresa primită de la dezvoltator pot să o testeze, sau publică, unde exista posibilitatea ca aplicația să fie dezvaluita și altor utilizatori. In mod implicit, aplicația nu e listata dar exista și posibilitatea de a o șterge de pe serverele Expo.

Există posibilitatea de a oferi și actualizări prin intermediul Expo, acestea fiind gestionate diferit pe cele doua platforme: pe android, actualizările se fac în timpul rulării luând efect la a doua rulare iar pe iOS actualizarea se face sincron fiind disponibilă la prima lansare.

Accesul la aplicațiile publicate Expo se poate face și offline, aplicația o sa aducă ultima versiune salvată pe dispozitiv dacă încercarea de a o aduce de pe server eșuează.

Există și posibilitatea de impachetare a aplicației într-un fișier .apk/.ipa pentru a fi lansate în magazinele digitale. În spatele "cortinei" o sa fie o versiune modificata a aplicației ExpoGo cu o singură adresă spre aplicația dezvoltatorului și fără să afișeze ecranele de încărcare sau sigla Expo.

Ca și dezavantaje se pot menționa următoarele:

- Adaugarea de module native este imposibilă – una din cele mai importante dezavantaje, care poate sa descurajeze dezvoltatorii
- Nu se pot folosii bibliotecile care utilizează cod nativ Object-C/Java/C++
- Aplicațiile au dimensiuni ridicate din cauza bibliotecilor integrate
- Unele functionalitati la integrarea serviciului pentru partea din spate a aplicației, Firebase, sunt indisponibile și trebuie folosite alternative sau alte soluții

Din fericire, React Native Expo oferă posibilitatea de a elimina aceste dezavantaje dacă o limitare neprevazuta oprește dezvoltarea aplicației prin introducerea fluxului de lucru liber (bare workflow).

Expo CLI – Flux de lucru liber

Dacă dezvoltatorul de aplicație React Native Expo, se lovește de o limitare, are la dispoziție funcționalitatea de evacuare din fluxul de lucru gestionat în cel liber, unde are control complet al dezvoltării aplicației împreuna cu complexitatea acesteia. Se pot folosi majoritatea interfețelor de programare din Expo gestionat, dar configurația ușoară și serviciul de construire al aplicației nu este disponibil deocamdată.

O funcționalitate care se păstreaza dar nu în totalitate este cea de testare și depanare a aplicației pe telefonul personal cu ajutorul aplicației gratuite ExpoGo. Aplicația se poate folosii în continuare pentru testarea aplicației dar nu și pentru acele bucăți care conțin cod nativ, a se evita orice apel spre o zona nativă în timpul testarii. Aceasta functionalitate parțială permite testarea codului JavaScript și pentru iOS utilizând chiar și platforma Windows, dacă avem acces la un telefon mobil cu sistemul de operare iOS.

Deși se pierd anumite avantaje specifice fluxului de lucru gestionat, acum dezvoltatorul are acces la codul nativ și la bibliotecile care îl implementează, lucru care vine cu noi posibilități în materie de functionalitati ale aplicației, mai ales în cazul utilizarii periferice lor.

4.2 Flutter

4.2.1 Introducere

Asa cum este mentionat in articolul [24], Flutter este un kit de dezvoltare software a interfețelor cu utilizatorul cu sursa deschisa, prima versiune stabilă fiind lansată de Google în 2018 deși a fost anunțată în 2015 sub numele de cod “Sky”. Acest kit este utilizat pentru dezvoltarea de aplicații pe multiple platforme, printre care Android si iOS, din aceeasi sursa.

Soluția de dezvoltare de la Google se folosește de limbajul de programare orientat pe obiect creat de aceeași companie, Dart. Acest limbaj de programare are o sintaxa similară cu cea a JavaScript, oferind dezvoltatorilor o curba de învățare lină, dar cu o viteza de rulare mai mare decat cea a programelor scrise in JavaScript, create inițial cu scopul de a randa consistent la 120 de cadre pe secunda.

Un mare avantaj al kitului Flutter este performanța lui nativă. Spre deosebire de React Native, aplicațiile Flutter sunt native în adevăratul sens al cuvântului și nu necesită un pod JavaScript, utilizându-se de conceptul AOT (Ahead-of-time), codul fiind compilat în cod nativ înainte de execuția programului.

Pe partea de interfață cu utilizatorul, kitul de la Google utilizează elemente de interfață grafică(widgets), care sunt componente structurale, de compoziție și stil, utilizate pentru crearea unui tot unitar în ceea ce privește aplicația precum butoane, fonturi text, culori, meniuri etc. Aceste componente sunt oferite de către Flutter in doua stiluri stilizate în așa fel incat sa se conformeze cu regulile de stil impuse de Android si iOS.

4.2.2 Arhitectura Flutter

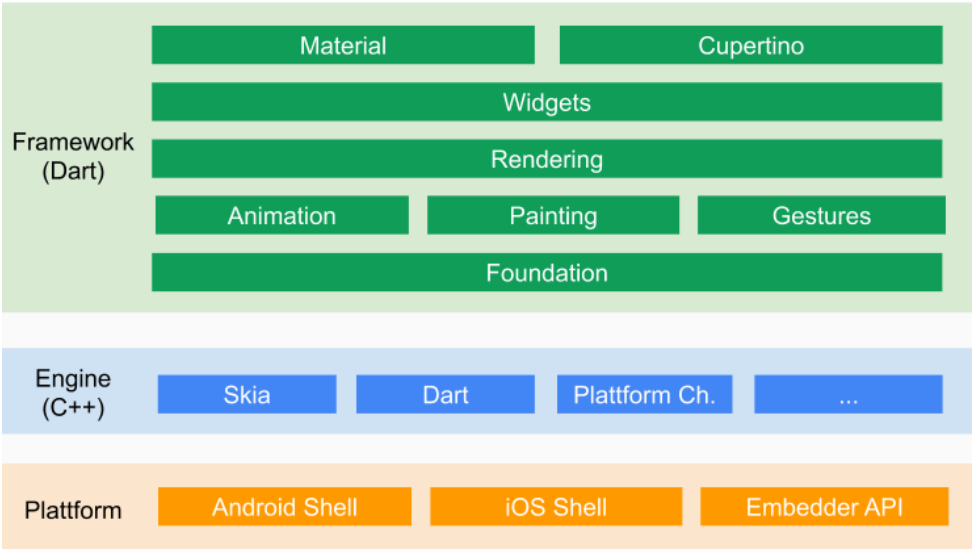

Figura 3.6: Arhitectura generala Flutter

Aplicațiile Flutter sunt văzute de sistemul de operare pe care rulează ca fiind aplicații native. Un sistem de incorporare specific platformei pe care se dorește rularea aplicației oferă un punct de access la API-urile native si la servicii precum randarea de suprafete, accesibilitate, input etc. Acest access se face prin așa numitul canal de platformă, care este un mecanism simplu de comunicare între codul Dart și codul specific platformei gazda. Pe lângă cele menționate, acest sistem gestionează și bucla de evenimente. [25]

Fiind scris într-un limbaj specific platformei, Java si C++ pentru Android și Objective-C pentru iOS, poate fi folosit în așa fel incat codul Flutter sa fie integrat într-o aplicație existența ca și un modul sau ca și o aplicație de sine stătătoare. Kitul bazat pe Dart are inclus un număr de sisteme de incorporare pentru platformele comune dar exista și sisteme dezvoltate de comunitate precum go-flutter.

La nucleul aplicațiilor Flutter se afla motorul Flutter, scris in C++. Acesta este un mediu de rulare portabil care implementează biblioteci esentiale Flutter printre care animatii si grafica, intrari si iesiri, accesibilitate și un mediu de rulare Dart pentru dezvoltare, compilare si rulare a aplicațiilor.

Pentru randarea compozițiilor și graficelor low-level, motorul se folosește de Skia, care este o biblioteca grafica cu sursa deschisa dezvoltată tot de Google care abstractizeaza partea de grafica specifica platformei.

Ca scheletul Flutter propriu-zis să aibă acces la motor, se foloseste dart:ui, care încastrează codul C++ al motorului în clase Dart. Dart:ui este o bibliotecă care dezvăluie serviciile de nivel scăzut folosite de către aplicații.

Ultima parte a arhitecturii este scheletul Flutter, care este interfața cu dezvoltatorul și utilizatorul. Aceasta este compusă de diferite straturi cu diferite scopuri:

- Fundația (foundation) – este o bibliotecă care conține toate pachetele necesare pentru a scrie blocuri de construire esentiale pentru servicii precum animație, afișare și gesturi. Aceste blocuri sunt abstractizări ale motorului scrise în Dart
- Stratul de randare – oferă o abstractizare care se ocupă cu compozitiile și este utilizat pentru crearea schemei de obiecte randabile.
- Stratul de elemente de interfață grafică sau componente – este o abstractizare a compozițiilor din stratul menționat anterior. Fiecare obiect randabil din stratul de randare are o clasa corespondenta aici. De asemenea, stratul acesta permite combinarea diferitelor clase pentru a putea fi refolosite în diferite feluri. Tot aici este introdus modelul de programare reactiv, care foloseste curenti asincroni de date (un curent este un obiect care emite date într-o perioada de timp)
- Stratul de elemente de interfață grafică specifici platformei – bibliotecile Material și Cupertino oferă elemente/componente realizate în așa fel incat sa se conformeze cu regulile de design specifice Android/iOS.

Scheletul Flutter este relativ mic deoarece o mare parte din caracteristicile de nivel mai înalt sunt implementate ca și pachete pentru utilizarea lor opțională, precum camera foto sau webview(care permite încorporarea paginilor web în aplicație). [26]

4.2.3 Procesul de dezvoltare

Începerea dezvoltării unei aplicații Flutter este una simpla datorita sintaxei asemănătoare cu cea JavaScript, odată instalat kitul de dezvoltare se poate ajunge rapid la o aplicație “Hello, World!” gata de testat.

Un avantaj major pe care îl are Flutter este documentația foarte bine pusă la punct care ghidează dezvoltatorul de la instalare, la alegerea mediului de dezvoltare, configurare editor și scrierea unei aplicații care implementează unele din cele mai esentiale caracteristici ale unei aplicații.

Un alt factor adjuvant al dezvoltării este existența extensiilor în cele mai utilizate editoare. In cazul Visual Studio Code, odată instalată extensia aveam acces la toate instrumentele necesare dezvoltării direct din editor, construire, testare și depănare, execuția emulatoarelor etc.

O caracteristică existentă și în React Native, este posibilitatea de a face reîncarcare la “cald” (hot reload). Odată facute modificari in cod nu este necesara reconstruirea aplicatiei, la salvarea codului aplicația este reincarcata, lucru care scurtează timpul de dezvoltare.

Flutter este un mediu care utilizează mai multe paradigme de dezvoltare, tehnici de programare dezvoltate si testatea de-a lungul anilor sunt folosite în kitul de la Google. Acestea sunt utilizate unde pot fi cat mai avantajoase. Putem menționa următoarele paradigme:

- Programare funcțională – aplicația este compusă din funcții pure, evitand modificarile de stare, efectele la anumite acțiuni și modificările datelor. Aplicațiile Flutter pot fii compuse în totalitate din componente fără stare bazandu-se pe primitive pentru computare și afișare a graficii (aceste aplicații sunt de obicei non-interactive)
- Programare asincronă – Flutter utilizează conceptul de “Future”, un obiect care este folosit pentru a reprezenta existența unei valori sau a unei erori care o sa fie valabilă cândva în viitor, o computare întârziată, și alte API-uri asincrone. Spre exemplu, un ecran de încărcare este finalizat prin terminarea computatiei unui “viitor”.
- Programare declarativă – Accentul cade pe ce vrem sa obtinem nu și cum vrem sa obtinem. Flutter permite dezvoltatorilor să descrie starea următoare fără a se ocupă manual de tranziție din starea curentă în starea următoare, aceasta este realizata de catre schelet. Spre deosebire de alte schelete care utilizează programarea imperativă, în Flutter este acceptata reconstruirea completa a unei întregi părți din interfata de la zero in loc sa o modificam. Flutter este destul de rapid pentru a permite acest lucru.
- Programarea imperativă – Folosita acolo unde este necesara, deobicei utilizata în colaborare cu o starea incapsulata într-un obiect. Spre exemplu, testele sunt scrise într-un mod imperativ.
- Programare orientată pe obiect bazată pe clase – Majoritatea API-urilor sunt construite cu ajutorul claselor și cu principiul OOP, mostenire: se definesc clase de nivel înalt după care se specializeaza în funcție de necesități.
- Programare compozițională – principala paradigma în Flutter, unde se utilizează obiecte mărunte cu comportament restrâns care compuse impreuna realizeaza efecte complicate

Alte avantaje ale procesului de dezvoltare în Flutter:

- Scriere rapidă a codului – datorită funcției de hot reload și a extensiilor care oferă bucati de cod configurabile pentru elementelele de interfata grafica.
- Același cod pentru ambele platforme Android și iOS – este posibilă și diferențiere în ceea ce privește stilizarea
- Performanța ridicată datorită motorului – documentația Flutter oferă o intreaga pagina pentru ajutor în ceea ce privește performanța aplicației
- Aceeași interfață și pe modele mai vechi de telefoane – Flutter suportă începând cu Android Jelly Bean iar iOS incepand cu iOS 8.
- O opțiune avantajoasă de a crea un produs minim viabil – produsele minim viabile au doar cateva caracteristici, de obicei utilizate pentru a putea primi

feedback în ceea ce privește partea de interfata pentru a putea continua mai departe cu dezvoltarea

Ca și dezavantaje putem menționa următoarele:

- Dificil de realizat un stil specific al platformei. O aplicație dezvoltată cu elemente grafice din Material, o sa ruleze pe iOS dar nu o sa aiba exact aceeași experiență pentru utilizator ca la o aplicație nativă.
- Actualizările iOS și Android vin mai târziu pe Flutter datorită faptului că dezvoltatorii ecosistemului trebuie sa le adauge manual
- Nu suportă atingerea 3D – spre exemplu pe iOS atingerea 3D este utilizata pentru afișarea meniurilor contextuale ascunse.

4.3 Șabloane de proiectare

Șabloanele de proiectare sunt soluții pentru problemele de design des intalnite in dezvoltarea software. Datorită apariției repetate a acestor probleme în contexte specifice, anumite metode de rezolvarea a acestora au apărut, pentru a putea fi refolosite de către dezvoltatori. Folosirea de șabloane de proiectare duce la o înțelegere ridicată a codului, încurajează scalabilitatea aplicației și definește o conexiune între dezvoltatori.

Una din cele mai bune cărți care oferă o prezentare amănunțită și o categorizare a sabloanelor de proiectare este cea scrisă de Erich Gamma, Richard Help, Ralph Johnson si John Vlissides, aceștia având să fie cunoscuți ca și “gașca celor patru” (Gang of Four) de unde și denumirea șabloane de proiectare GoF [27].

Șabloanele sunt împărțite în trei categorii:

- Șabloane de creație – pentru crearea de obiecte
- Șabloane structurale – legăturile dintre obiecte
- Șabloane de comportament – cum interacționează obiectele între ele

Utilizarea șabloanelor de proiectare nu este una obligatorie, ci opțională în cazul în care problema pe care acestea o rezolvă apare în aplicația dezvoltatorului. Trebuie folosite în cazul în care utilizarea lor simplifică codul, nu îi ridică complexitatea.

În subcapitolul curent vor fi prezentate doar șabloanele de proiectare specifice ecosistemelor utilizate sau limbajelor pe care acestea sunt bazate și care sunt utilizate în implementarea aplicației curente.

4.3.1 Șabloanele de proiectare în Flutter/Dart

Șablonul fațadă – șablon structural GoF

În cazul în care avem mai multe componente sau funcții cu implementări complexe, putem sa abstractizam implementarea pentru o mai bună lizibilitate a codului și depanare mai rapidă.

Pentru a putea implementa acest șablon, putem folosii unul din cele doua tipuri de componente puse la dispoziție de către Flutter, componente fără stare și componente cu stare.

Componentele fără stare sunt utilizate în cazul în care conținutul acestuia nu se schimba, folosit adesea ca și container pentru componente cu stare precum ecranele diferite dintr-un meniu al aplicației. Ca și exemple avem următoarele componente esentiale: icoana, buton icoana, text. Toate acestea sunt componente fără stare și mostenesc clasa “Stateless Widget”.

```
4   class Exemplu extends StatelessWidget {
5
6     @override
7     Widget build(BuildContext context) {
8
9       return Scaffold();
10    }
11  }
```

Figura 3.6: Exemplificare componenta fără stare

Componentele cu stare sunt dinamice, dacă utilizatorul interacționează cu acestea sau primesc date noi, ele își pot schimba aspectul. Ca si exemple, avem următoarele componente esentiale: butoane de bifat, butoane radio, campuri text etc. Toate acestea sunt componente cu stare și mostenesc clasa “StatefulWidget”.

Starea unei componente cu stare este stocată într-un obiect de tipul stare, separand starea componentei de aparența ei. Pentru a se realiza schimbarea de stare, odată cu apariția unui eveniment care ar declanșa o schimbare sau a unei actualizari de date, metoda de setare stare este apelata, lucru care declanșează rerandarea componentei. [28]

```dart
class _ExempluStare extends State<ExempluStare> {
  var culoareExemplu = Colors.black;

  void apasareButon() {
    setState(() {          // cu setState modificam starea, in cazul de fata culoarea textului
      culoareExemplu = Colors.red;
    })
  }

  @override
  Widget build(BuildContext context){
    return Row(
      children: [
        Text('Doar un exemplu', style: TextStyle(color: culoareExemplu)),    // textul
        ElevatedButton(child: Text('Schimba culoarea textului', onPressed: apasareButon()))
        // la apasarea butonului schimbam culoarea
      ]
    );
  }
}

```

Figura 3.7: Exemplificare componenta cu stare

Șablonul furnizor

Există cazuri în care este necesar ca mai multe clase din structura aplicației să aibă acces la starea celorlalte clase. Spre exemplu, vrem să aflăm dacă un anumit serviciu este activ în altă parte a aplicației. Pentru a putea face acest lucru avem nevoie de acces la starea curentă.

În cazul Flutter, care se folosește de o paradigmă declarativă, componentele sunt reconstruite nu modificate, această stare curentă, ca și localizare în sistem, trebuie să fie deasupra componentelor, care au nevoie de anumite date din aceasta.

O metodă simplă de a face acest lucru este prin utilizarea conceptului de callback, atașând o funcție ca și parametru care să fie executată cu datele de care avem nevoie la apariția unui eveniment. Dar odată cu creșterea în complexitate a aplicației acest lucru devine greu de urmărit, codul nefiind la fel de lizibil.

Pentru a gestiona starea curentă, dezvoltatorul dispune de mai multe metode printre care Redux, o bibliotecă utilizată pentru gestionarea stării aplicațiilor complexe, sau prin utilizarea pachetului furnizor, o opțiune mult mai simplă și recomandată de către Flutter în cazul aplicațiilor cu grad de complexitate scăzut. [28]

Utilizând pachetul furnizor, este necesară utilizarea a trei concepte:

Notificatorul de schimbări – este o clasă inclusă în fundația ecosistemului Flutter ce poate fi utilizata și fără pachetul provider. Are un rol similar cu cel de observat din tiparul observatorului, acest rol este de a trimite notificări ascultatorilor sai. Clasa care are informațiile de care avem nevoie la alt nivel trebuie extinsă cu aceasta clasa iar

de fiecare data cand se modifica o informație utilizată și în alta parte a aplicației, trebuie apelată funcția specifică notifyListeners().

```
42
43  class A extends ChangeNotifier {
44
45  final List<String> lista = [];
46
47  void adauga(String cuvant){
48    lista.add(cuvant);  //adaugam in lista
49    notifyListeners();  //trimitem notificare ca lista s-a actualizat
50  }
51
52  }
```

Figura 3.8: Exemplificare notificator de schimbări

Furnizorul de notificator – este o componentă din pachetul furnizor, folosită pentru a oferi descendenților săi o instanță a notificatorului. Așa cum am menționat mai sus, acesta trebuie poziționat în sistem deasupra componentelor care trebuie să se aboneze la notificator, poate fi deasupra întregii aplicații ca și în exemplul prezentat mai jos sau doar deasupra unei alte componente.

```
54  void main(){
55    runApp(
56      ChangeNotifierProvider(
57        create: (context) => A(),
58        child: AplicatiaMea(), //AplicatiaMea este componenta
59      )                        // de deasupra componentelor care au nevoie de datele respective.
60    );
61  }
```

Figura 3.9: Exemplificare furnizor de notificator

Consumatorul – pentru a putea avea acces la notificările venite de la furnizor, consumatorul trebuie să se aboneze la acesta. Acest lucru se realizează folosind componenta consumator (Consumer) care are un singur argument obligatoriu, constructor(builder), o funcție apelată de fiecare dată când metoda specifică a notificatorului “notifyListeners” este apelată.

Metoda constructor are trei parametrii, primul reprezintă contextul în care componenta a fost construită, al doilea argument este instanța notificatorului, pe care o putem folosii pentru a obține datele iar al treilea argument este utilizat pentru optimizare, în cazul în care consumatorul are un subsistem mare sub el, si nu se doreste a fii reconstruit de asemenea.

```dart
@override
 Widget build(BuildContext context) {

   return Scaffold(
     child: Consumer<A>(    //este obligatoriu de specificat generalitatea <A>
       builder: (context, lista, child) { //prin intermediul argumentului al doilea se acceseaza datele
         return Text("Ultimul element din lista este ${lista.last}");
       }
     )
   );
}
```

Figura 3.10: Exemplificare consumator

Șablonul prototipului – șablon de creație GoF

Acest șablon este utilizat pentru crearea de obiecte. În unele cazuri este necesară clonarea unui obiect pentru a nu utiliza originalul pentru acțiuni care ar putea să dăuneze obiectul. O opțiune ar fii să creăm o instanța nouă și să copiem proprietățile una câte una. Această opțiune este viabilă dar nu în toate cazurile.

Cea mai bună opțiune pentru a realiza aceasta acțiune de clonare, este prin implementarea unei metode de clonare direct în obiect. Dacă un obiect are o metoda de clonare, doar obiectul respectiv o sa aibă cunoștințe de structura lui internă și acțiunea de copiere se face într-o linie de cod indiferent de numărul de parametrii.

Șablonul strategie – șablon de comportament GoF

În cadrul sablonului de proiectare strategie, comportamentul unei componente este schimbat în timpul rulării, obiectul da senzația unei schimbări totale de componenta.

Pentru a realiza acest lucru în Flutter, se utilizează o componentă cu stare iar cu ajutorul metodei specifice setState(). La setarea stării, interfața este reconstruită și comportamentul modificat. Ca și exemplu, se prezintă utilizarea șablonului în cazul modificării comportamentului unui buton.

```dart
class _ExempluStare extends State<ExempluStare> {
  var culoareExemplu = Colors.verde;
  var text = 'Schimba culoare rosu';
  var activ = 'SCHIMBA_ROSU';

  void apasareButon() {
    setState(() {          // cu setState modificam starea, in cazul de fata culoarea textului
    switch (activ) {       // modificam si comportamentul butonului
      case 'SCHIMBA_ROSU':
        culoareExemplu = Colors.red;
        text = 'Schimba culoare verde';
        break;
      case 'SCHIMBA_VERDE':
        culoareExemplu = Colors.green;
        text = 'Schimba culoare rosu';
        break;
      case default:
        break;
    }

    activ = !activ;
    });
    }

```

Figura 3.12: Exemplificare a sablonului strategie

4.3.2 Şabloane de proiectare în React Native/JavaScript

Datorită utilizarii JavaScript ca limbaj de programare pentru scheletul React Native, unele șabloane de proiectare sunt "împrumutate". [29]

Șablonul componentelor de prezentare

Componentele de prezentare au rolul doar de a prezenta informația primită ca și parametru și a o stiliza în modul cerut, acestea nu conțin logica au rolul doar de prezentare după cum le spune și numele. De asemenea ele nu au stare.

Utilizarea acestei componente permite reutilizarea codului și separarea codului în entități cu rol bine stabilit, deoarece nu au dependinte de restul aplicației primind date doar prin intermediul parametrilor.

Șablonul componentelor container

Componentele de tipul container sunt acele componente care se ocupă de logica aplicației și de procesare. Acestea analizează datele proprii sau primite din restul aplicației ca mai apoi sa le trimita componentelor de prezentare pentru afișarea lor sub forma finală sau alte componente container.

Aceste componente au următoarele caracteristici:

- Se ocupă de felul în care aplicația funcționează
- Nu au niciodată stil definit în interiorul lor
- Sunt de obicei componente cu stare
- Se ocupă cu preluarea datelor din baza de date sau de la alte componente
- Este utilizat de obicei cu biblioteca Redux, dezvoltata initial pentru React dar utilizata acum si-n alte ecosisteme și folosită pentru păstrarea persistentă a stării aplicației într-un magazin, fiecare componenta avand acces la aceasta.

Șablonul utilizării cârligelor

În React Native există două moduri de a scrie o componentă, utilizând o funcție sau o clasă. Componentele funcționale, care folosesc o funcție, sunt doar funcții JavaScript care returnează alte componente fundamentale sau create de către dezvoltator și pot fi definite cu ajutorul cuvantului cheie "function" sau folosindu-se de funcțiile lambda, care sunt functii anonime cu o sintaxa mai scurta. Acestea nu au stare deci sunt utilizate de cele mai multe ori ca și componente de prezentare.

Componentele clasa sunt clase care mostenesc clasa de baza Component din React, acest lucru dandu-le atat access la metode precum randare constructor etc. cât și functionalitatea de stare și parametrii. Acestea sunt folosite ca și componente container datorită faptului ca au access la gestionarea stărilor. La scrierea unei astfel de componente, singura metoda obligatory este cea de randare, restul fiind optionale.

Desi componentele clasa și cele funcționale aveau rolurile bine stabilite în ceea ce privește starea, odată cu apariția cârligelor (hooks) în versiunea 0.59, componentele funcționale pot să folosească funcționalitatea de stare și alte caracteristici ale ciclului de viață a aplicației, fără a modifica componentele deja existente.

Cârligele sunt funcții utilizate pentru a "agața" starea React și caracteristicile ciclului de viata din componentele funcționale. Câteva din cârligele cele mai folosite sunt următoarele:

- Cârligul stare – utilizat pentru modificarea stării aplicației în interiorul unei componente funcționale. Spre exemplu, pentru a putea modifica starea unui buton.
- Cârligul efect – adaugă abilitatea de a folosi efecte secundare în interiorul unei componente funcționale. Spre exemplu, la randarea unui ecran al aplicației, o anumită bucata de cod sa fie executată.
- Cârligul context – mai puțin utilizat dar permite abonarea la contextul React.
- Cârligul reducer – utilizat pentru gestionarea stării locale a componentelor complexe
- De asemenea, React introduce și posibilitatea de a construire a propriilor cârlige, cu ajutorul celor menționate mai sus.

Apariția cârligelor și folosirea programării funcționale, face ca împărțirea aplicației în componente container și componente de prezentare sa nu mai fie necesara. Deși componentele funcționale pot fii folosite în locul claselor, React Native nu planuieste sa le scoata din folosinta. Acestea pot fii utilizate in paralel cu carligele.

Ca și reguli de utilizarea a cârligelor avem următoarele:

- Carligele trebuie apelate doar la cel mai înalt nivel, nu în interiorul buclelor sau a funcțiilor imbricate
- Carligele trebuie apelate doar în componentele funcționale React nu și în funcțiile JavaScript
- Carligele pot fi apelate și din interiorul carligelor personalizate de către dezvoltatori

Șablonul de furnizor

Există cazuri în care o anumită informație trebuie utilizată in mai multe locuri din structura de componente, în cazul acesta ar trebui sa trimitem informația ca și un parametru de la părinte până la copilul/copiii care au nevoie de acea informație. Aceasta pasare de valori se face din nivel in nivel care nu aduce neapărat probleme ca soluție în sine dar odată cu creșterea în complexitate a aplicației, dezvoltatorul ajunge sa trimita parametrii printr-un număr ridicat de nivele, acest lucru duce la pasarea valorilor prin componente care nu au nevoie de acestea doar trebuie sa le primească și să le distribuie

mai departe. Pentru aceasta problema, dezvoltatorii de aplicații React Native au la dispoziție mai multe opțiuni.

Opțiunea implementată de către mine este interfața de programare a aplicației "Context". Aceasta solutie este folosită pentru a impartasii informația, care este necesara unui numar mare de componente de la diferite nivele, intr-un mod automat.

Pentru implementarea acestei soluții, se creeaza un obiect de tipul Context care are ca și parametrii valoarea sau setul de valori pe care dorim să-l folosim intr-o alta componentă. Componenta o să înglobeze componenta strămoș a componentei unde urmează să folosim setul de valori, în acest fel toți descendenții acesteia sunt consumatori ai valorilor furnizate iar de fiecare dată când setul de valori se modifică, o randare a acestora este inițiata.

Pentru a folosii valoarea sau valorile din setul trimis mai departe se poate utiliza carligul mai sus menționat "useContext" care primește ca parametru obiectul Context și returnează setul de date.

Aceasta soluție este ideală a fi implementată în proiecte relative mici, unde valorile pasate sunt utilizate de un număr mai mare de componente deoarece toate se vor randa la schimbarea valorii, în aplicații mari putând să apară întarzieri. Printre avantajele acestei soluții se numără și simplitatea implementării odata cu apariția carligelor.

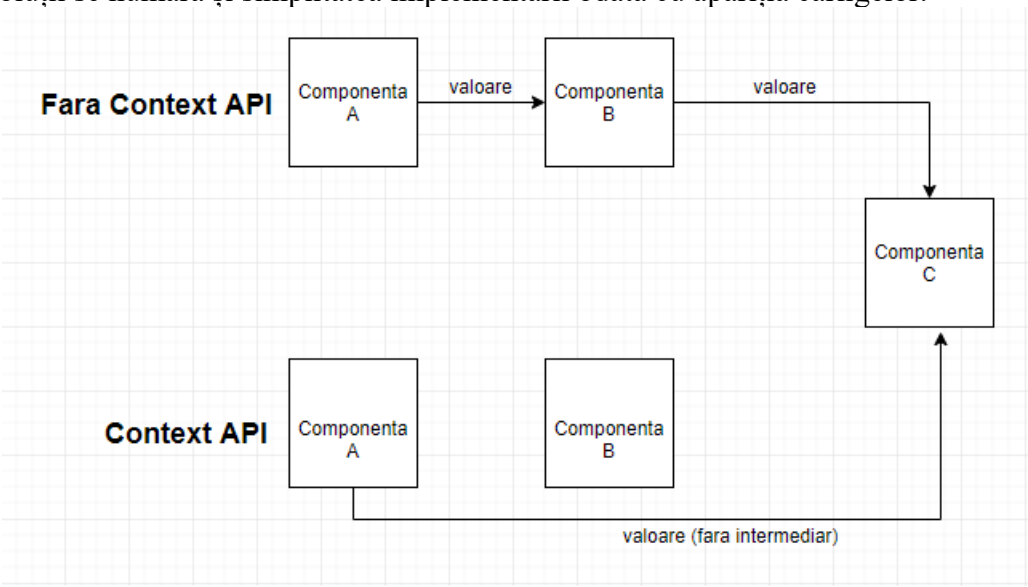

Figura 3.2: Diagrama reprezentare utilizarea a Context API

Alternative la utilizarea API-ului Context:

- Redux – bibliotecă de gestionare a stării, original creata pentru React, dar utilizata acum in mai multe schelete JavaScript. Implementare mai complexa decat Context API si ideala pentru proiecte mai mari.

- Variabile globale – utilizarea variabilelor(constante) globale soluție acceptată în cazul în care acestea nu se schimba, dacă acestea se schimba componentele nu se rerandeaza (lucru care se poate realiza cu carligul stare)

Șablonul modul

Limbajul de programare JavaScript nu suportă modificatori de access pentru variabile. Pentru a imita acest comportament, dezvoltatorii folosesc la atribuirea unei variabile o funcție care returnează valoarea dorită. In acest fel, avem acces la valoarea returnată a funcției, partea publica, dar nu și la variabilele declarate în funcția respectivă, partea privata. Aceste funcții poarta numele de module.

```
4   var incrementare = () => {
5       var contor = 0 //membru privat
6
7       return () => {++contor}; // membru public
8   }
9
10  console.log(incrementare()); // se printeaza 1
11  console.log(incrementare()); // se printeaza 2
12  console.log(incrementare()); // se printeaza 3
13
```

Figura 3.3: Exemplificare sablon modul

Șablonul observator – șablon de comportament GoF

Șablonul observator este folosit pentru a îmbunatatii comunicarea dintre componente într-un mod optimizat, promovand cuplajul slab. Un sistem avand cuplajul slab este un sistem în care componentele au cât mai puține date sau depind cat mai puțin de celelalte componente.

Acest șablon este împărțit în doua componente, subiectul și observatorii. Subiectul realizeaza anumite operații, computari iar observatorii au posibilitatea de a se abona la actualizari venite din partea acestuia.

In React Native, implementarea acestui șablon de proiectare se poate face cu ajutorul API-ului context, Redux, MobX (o alta biblioteca utilizata in react native pentru gestionarea stării) sau prin utilizarea carligului effect, care este de obicei folosit în colaborare cu metodele de gestionare a stării menționate mai devreme.

```
// Se presupune existenta unui context "Context"
var variabilaExemplu = useContext(Context);

useEffect(() => {
    //cod de executat la modificarea contextului
},
[variabilaExemplu]); //parametrul al doilea al carligului efect specifica dependintele
```

Figura 3.4: Exemplificare utilizarea sablonului observator cu ajutorul API-ului context si carligului effect

Șablonul fatada – sablon structural GoF

Sablonul fațada este utilizat pentru a ascunde complexitatea unui sistem și a prezenta doar o interfata pentru o mai bună lizibilitate a codului, care duce la o depanare mai lină și rapidă.

În React Native acest lucru este implementat cu ajutorul componentelor funcționale, cu ajutorul cărora putem sa ascundem atat complexitate computationala cât și de design, acestea putand sa returneze si componente esentiale puse la dispoziție de echipa React precum “View”, “Text”, “Button” etc.

Utilizat de obicei în cazul proiectelor medii și mari, acest șablon este esențial pentru a ascunde implementările unde este cazul, un echilibru fiind necesar.

```
const App = () => {

const AfiseazaText = () => {
    return <View>
        <Text>Text 1</Text>
        <Text>Text 2</Text>
        <Text>Text 3</Text>
    </View>
}

return <View>
        <AfiseazaText />
    </View>

}
```

Figura 3.5: Exemplificare sablonul fațada

Șablonul strategie – șablon comportamental GoF

Șablonul strategie are aceleași caracteristici în React Native ca si in Flutter, diferenta fiind doar de limbaj și metoda de implementare. In React Native, pentru a obține acest comportament se utilizează carligul stare. Ca si exemplu, îl folosim același ca și la Flutter cu schimbarea de comportament a butonului.

```
24  const App = () => {
25  const [text, setareText] = useState('Schimba culoare rosu');
26  const [culoareText, setareCuloareText] = useState('green');
27  const optiune = 'SCHIMBA_ROSU';
28
29  const schimbaButonul = () => {
30  switch (optiune){
31      case 'SCHIMBA_ROSU':
32          setareText('Schimba culoare negru');
33          setareCuloareText('red');
34          optiune = 'SCHIMBA_VERDE'
35          break;
36      case 'SCHIMBA_VERDE':
37          setareText('Schimba culoare rosu');
38          setareCuloareText('green');
39          optiune = 'SCHIMBA_ROSU';
40          break;
41      default:
42          break;
43  }
44  }
45
```

Figura 3.13: Exemplificare sablon strategie React Native

5. Proiectare de detaliu și implementare

În acest capitol se va prezenta procesul de dezvoltare a aplicației pe sistemul de operare Android, cu cele doua ecosisteme alese: React Native si Flutter. Scopul este de a prezenta implementarea functionalitatilor, tehnicile de programare importante folosite și argumentarea deciziilor luate.

5.1 Implementare în React Native

5.1.1 Instalare React Native

Inițial am ales Expo cu flux de lucru gestionat pentru dezvoltare. După dezvoltarea interfeței grafice, din necesitatea de a folosii biblioteci care utilizau cod nativ, am evacuat aplicația din fluxul de lucru gestionat în cel liber pierzand astfel accesul la testarea directă pe dispozitivul personal, cu ajutorul aplicației ExpoGo.

Pentru instalare interfeței linie de comanda ExpoCLI, am utilizat managerul de pachete “npm” (alternativa yarn).

Odata instalat CLI, am putut crea proiectul folosit un sablon gol, lucru care genereaza doar un ecran cu titlul “Hello, World!”. Comanda utilizata este aceasta: *expo init rn_licenta*. (rn_licenta fiind numele proiectului). După inițializarea proiectului, acesta se poate testa folosind comanda: *expo start* care lansează atat aplicația cât și serviciul de împachetare Metro.

Ca si testare, am utilizat un emulator de Android integrat in Android Studio și dispozitivul mobil personal(in fluxul gestionat). Datorită funcționalității de reincarcare la cald (hot reload) modificările făcute în aplicație se pot vedea instant in emulator/dispozitiv personal.

5.1.2 Editor de cod

Pentru scrierea codului, am utilizat editorul de cod de la Microsoft: Visual Studio Code configurat cu extensii specific React Native. Printre extensiile folosite se numără:

- Instrumente React Native (React Native Tools) – care transforma editorul general într-unul specific React Native cu caracteristici precum: rulare comenzi React Native direct din paleta de comenzi, oferă modalități de configurare a mediului de depanare, completare cod și posibilitatea de personalizare
- ESLint – utilizat pentru gasirea erorilor de sintaxa în momentul scrierii codului
- Auto închiderea etichetelor(Auto close tags) – utilizat pentru sintaxa JSX, închiderea automată a etichetelor după scriere

- Auto redenumire a etichetelor – utilizat tot pentru sintaxa JSX, în cazul componentelor complexe elimină necesitatea modificării manuale a doua linii de cod.
- Auto formatare (Prettier) – indenteaza codul pentru o mai bună lizibilitate

5.1.3 Navigare în aplicatie

Datorită faptului ca aplicația este compusă din ecrane multiple și cu scopul de a oferi interactivitate în cadrul acesteia, am decis sa folosesc conceptul de navigatie. Aceasta functionalitate poate fi implementată cu ajutorul bibliotecii React Native Navigare (Navigation). React Native pune la dispoziție aceasta solutie care are abilitatea de a prezenta tranziții între ecrane specific platformei pe care se rulează.

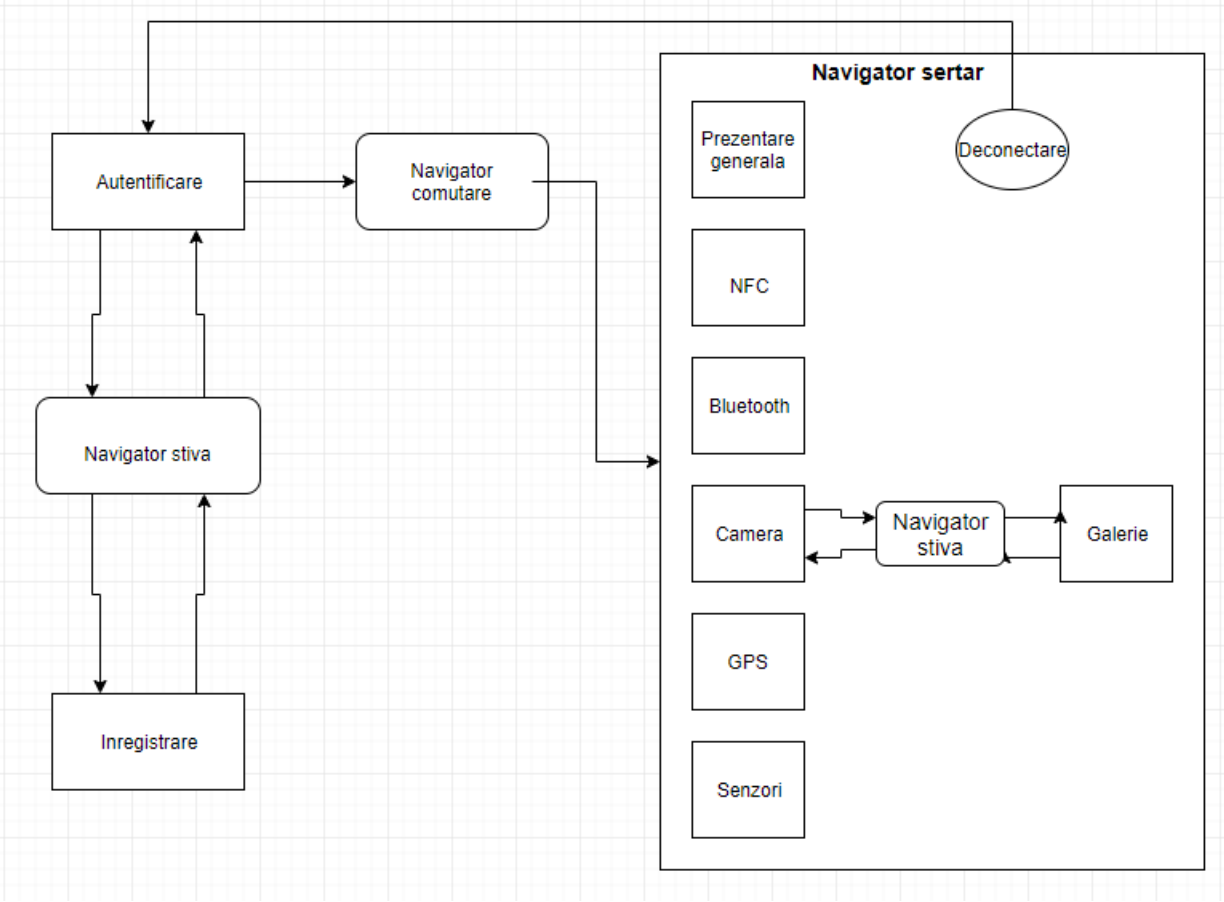

Figura 4.1: Diagrama a ecranelor și navigatoarele folosite

Pe lângă biblioteca “react-native-navigation”, este necesara instalarea unor dependinte:

- React-native-reanimated – o biblioteca utilizata pentru crearea animatiilor și executarea lor pe firul principal de executie(UI thread) pentru a ușura munca firului de execuție JavaScript și a nu provoca intarzieri. Utilizata la navigatie pentru tranzițiile dintre ecrane.
- React-native-gesture-handler – o biblioteca care dispune de o interfața de programare a aplicației pentru gestionarea gesturilor cu capabilitati native. Ca si exemplu, gestul de rotire a dispozitivului mobil sau gestul de apasare.
- React-native-screens – o biblioteca care oferă primitive native pentru reprezentarea ecranelor. Folosită de obicei de dezvoltatorii de biblioteci, prea putin de utilizatori.

Există diferite tipuri de navigatoare, dintre care pentru interactivitate în aplicație am utilizat trei: navigatorul de comutare(Switch), navigatorul stiva(Stack) si navigatorul sertar(Drawer). Acestea se pot utiliza și impreuna: un navigator sertar poate sa conțină un navigator stiva ca și exemplu.

Navigatorul de comutare afișează un singur ecran la un moment dat și nu are suport în mod implicit pentru acțiunea de întoarcere la ecranul anterior.

Navigatorul stiva este utilizat în cazul în care avem ecrane multiple cu o legatura de functionalitate între ele. Acesta permite tranziția între doua ecrane prin plasarea ecranului care urmează a fi afișat, deasupra in stiva. Oferă posibilitatea acțiunii de întoarcere la ecranul anterior, prin scoaterea din stiva a ultimului element adaugat.

Navigatorul sertar este utilizat de cele mai multe ori ca meniu al aplicației și este reprezentat grafic ca un “sertar” derulat din partea stanga sau dreapta a ecranului unde dispune de o lista de elemente cu legatura la un ecran diferit.

5.1.4 Autentificare și înregistrare

Pentru realizarea autentificării și a înregistrării, am utilizat un serviciu care oferă logica din spate(backend), Firebase. Aceste tipuri de servicii sunt utilizate pentru a simplifica ce se intampla în “spatele cortinei”, dezvoltatorul concentrandu-se pe interfața grafica.

Firebase este o platformă oferita de Google, iar printre functionalitatile pe care le oferă se afla și autentificarea și înregistrarea. Ca sa implementez aceste caracteristici, am utilizat React Native Firebase, o colectie de module, cu acces la kiturile Firebase native.

Înainte de utilizarea ei propriu-zisa, crearea unui proiect este necesara din consola Firebase, după autentificarea/înregistrarea cu un cont google. Crearea unui proiect este realizata prin completarea unei serii de date, precum numele proiectului și identificatorul aplicației. Odata finalizati pasii, firebase oferă un fișier de configurare a serviciului în interiorul aplicației.

După crearea proiectului, am instalat biblioteca React Native Firebase cu ajutorul expo: *expo install firebase.* Pentru instalare se poate folosi și managerul de pachete, npm, dar "expo install" se asigura ca versiunea instalată a bibliotecii este una compatibila.

Pentru configurarea Firebase, nu am folosit fișierul descărcat ci doar datele de configurare disponibile și în consola serviciului pentru a-mi crea propriul fișier de configurare unde am și actualizat serviciul și funcționalitatea oferită de Firebase, Firestore, care este o baza de date bazata pe documente. Aceasta functionalitate am utilizat-o pentru stocarea de date.

```
1  import React from 'react';
2  import * as firebase from 'firebase';
3  import '@firebase/auth';
4  import '@firebase/firestore';
5
6  const firebaseConfig = {
7      apiKey: "AIzaSyBlMBiFV9cdzG5Mg7FfFZ0NO-s-xSj-kY4",
8      authDomain: "rn-licenta-2021.firebaseapp.com",
9      projectId: "rn-licenta-2021",
10     storageBucket: "rn-licenta-2021.appspot.com",
11     messagingSenderId: "832647924617",
12     appId: "1:832647924617:web:6d4eba62960574155bef58",
13     measurementId: "G-T9D4GB90Z8"
14   };
15
16   if (!firebase.apps.length) {
17     firebase.initializeApp(firebaseConfig);
18     }
19
20    export const database = firebase.firestore();
```

Figura 4.1: Configurare Firebase/Firestore in React Native

Odată inițializat serviciul și definită configurarea, Firebase se poate utiliza. Pentru implementarea functionalitatii avem nevoie de doua componente: componenta pentru ecranul de autentificare și componenta pentru ecranul de înregistrare.

Componenta de autentificare este formată din doua componente esentiale de introducere text, pentru campurile e-mail și parola, cea pentru introducerea parolei avand parametrul de securizare a introducerii setat pe adevărat, o componenta esențială de tipul buton și un text pentru navigarea la componenta de înregistrare.

Pentru componentele de introducere text avem nevoie de utilizarea carligul state, nefolosind componentele de tipul clasa care au stare integrată. Deoarece la introducerea textului, ecranul nu se randeaza singur, avem nevoie de doua stări care sa fie actualizate de fiecare data cand se introduce/șterge o litera și să declanșeze schimbarea de stare.

În cazul componentei de tip buton, avem nevoie de o funcție care sa apeleze metoda specifica autentificării din modulele puse la dispoziție de către biblioteca Firebase. Aceasta funcție este legată la evenimentul de apasare a butonului.

Prin utilizarea metodei "signInWithEmailAndPassword" din Firebase, adresa de e-mail și parola sunt verificate dacă exista și dacă sunt corect introduse. Dacă utilizatorul este înregistrat, navigatorul o sa schimbe ecranul afișat în cel al aplicației și o să afișeze un mesaj de success de tipul "toast", o bara specifica Android cu un mesaj personalizat care apare din partea de sus a ecranului cu durata definită de variabile *"ToastAndroid.LONG"*. Dacă utilizatorul este inexistent sau parola greșită, un mesaj de același tip este afișat cu un text care descrie problema pentru ca aceasta sa fie remediata de catre utilizator.

În cazul în care utilizatorul dorește sa se inregistreze, are la dispoziție o componenta esentiala de tipul text, situata sub butonul de autentificare, care la apasare declanșează rutarea la componenta responsabilă de înregistrare.

Componenta de înregistrare este similară cu cea de autentificare dar cu componenta de tipul buton schimbată, atât ca și aspect cat si ca functionalitate. Aceasta componenta are legată la evenimentul de apasare a butonului, o funcție în care metoda Firebase pentru înregistrarea utilizatorului pe baza datelor introduse este apelată.

```
const onLoginPress = () => {
    firebase.auth()
    .signInWithEmailAndPassword(email, password)
    .then(() => {props.navigation.navigate({routeName: 'App'}),
        ToastAndroid.showWithGravityAndOffset(
        "Welcome!",
        ToastAndroid.LONG,
        ToastAndroid.TOP,
        25,
        50
      )})
    .catch(error => ToastAndroid.showWithGravityAndOffset(
        error,
        ToastAndroid.LONG,
        ToastAndroid.TOP,
        25,
        50
      ))

};
```

Figura 4.2: Metoda folosită pentru autentificare atașată butonului de logare

Metoda utilizata pentru înregistrarea utilizatorului este "createUserWithEmailAndPassword", metoda care verifica corectitudinea datelor: adresa de e-mail să fie de forma " XX@XX.XX" și parola sa indeplineasca o anumită lungime ca număr de caractere. Verificari adiționale se pot face înainte de rularea metodei, spre exemplu verificarea existenței caracterelor speciale în interiorul parolei.

Dacă datele sunt corecte, utilizatorul este creat și datele stocate în firebase unde avem acces la informații precum data creării, un număr de identificare unic auto-generat și diferite comenzi de resetare parola, stergere cont sau suspendare cont.

Pe langa cele de mai sus, firebase oferă posibilitatea configurarii unor sabloane pentru diferite acțiuni specific utilizatorului că și verificarea adresei de e-mail după înregistrare, resetare parola, adresa de e-mail schimbată sau verificare prin mesaj text.

Trecerea între cele doua componente autentificare și înregistrare este realizata cu ajutorul navigatorului de tip stiva. Acesta permite tranziția dintre ecranul de autentificare și cel de înregistrare prin plasarea ecranului de înregistrare deasupra în stiva. Acțiunea de întoarcere din ecranul de autentificare se realizeaza prin scoaterea elementului reprezentat de ecranul înregistrare din varful stivei. Navigatorul tip stiva adaugă în mod automat un buton sub forma unei săgeți în bara de titlu pentru aceasta actiune.

```
const onRegisterPress = () => {
    firebase.auth()
    .createUserWithEmailAndPassword(email, password)
    .then(() => {
        props.navigation.navigate('Login'),
        ToastAndroid.showWithGravityAndOffset(
            "Account created succesfully!",
            ToastAndroid.LONG,
            ToastAndroid.TOP,
            25,
            50
      )})
    .catch(error => ToastAndroid.showWithGravityAndOffset(
        "Account created succesfully!",
        ToastAndroid.LONG,
        ToastAndroid.TOP,
        25,
        50
      ))
};
```

Figura 4.3: Metoda de înregistrare a utilizatorului cu ajutorul Firebase

5.1.5 Meniul principal

Meniul principal al aplicației este realizat cu ajutorul unui navigator de tipul sertar. Deschiderea acestuia se face prin apăsarea butonului din bara de titlu specific

platformei. La apasare o componenta care acoperă jumătate din suprafața ecranului, este afisata si prezinta o lista de elemente legate direct la ecranele aplicației.

Ecranele principale ale aplicației, fiecare implementand functionalitati diferite, sunt următoarele:

- Prezentare generala – afișează toate functionalitatile aplicației și ultima data cand au fost active
- NFC – implementeaza functionalitatea de citire NFC
- Bluetooth – implementează functionalitatea Bluetooth
- Camera – implementează functionalitatea de camera foto și un navigator de tipul stiva, pentru a se putea face tranziția în ecranul Galerie
- GPS – implementează functionalitatea de locație curenta
- Senzori – implementează functionalitatea de giroscop si accelerometru

Realizarea navigatorului de tip sertar se face cu ajutorul funcției "*createDrawerNavigator*" din biblioteca de navigatie. Metoda primeste doi parametrii. În primul parametru o sa se introducă ecranele care fac parte din "sertar", acestea pot fii atat componente React Native cât și alte navigatoare, în cazul meu navigatorul sertar conține câte un navigator stivă pentru fiecare componenta. Al doilea parametru este utilizat pentru personalizarea navigatorului precum stilul elementelor din sertar sau poziționarea acestora. Pentru a putea utiliza navigatorul sertar ca și rădăcina a aplicației, utilizăm metoda "*createAppContainer"* care transforma obiectul returnat de către creatorul navigatorului sertar intr-o componenta React pentru a putea fi exportată și utilizata ca și container pentru restul aplicației.

```
47  const AppNavigator = createDrawerNavigator({
48      Overview: OverviewNavigator,
49      NFC: NFCNavigator,
50      GPS: GPSNavigator,
51      Camera: CameraNavigator,
52      Gyroscope: GyroscopeNavigator,
53      Bluetooth: BluetoothNavigator
54  },
55      {
56          contentOptions: {
57              activeTintColor: '#3F855B',
58              itemStyle: {
59                  marginVertical: 5
60                  //marginTop: 40
61              },
62              itemsContainerStyle: {
63                  marginTop: 50
64              }
65          }
66
67      });
68
69  export default createAppContainer(AppNavigator);
```

Figura 4.4: Crearea navigatorului sertar

5.1.6 Ecranul GPS

Ecranul GPS implementează funcționalitatea de locație curentă. Utilizatorul este întrebat dacă oferă permisiunea de accesare a locației în momentul navigarii la acest ecran. Acesta are posibilitatea de accesare a coordonatelor geografice, latitudine si longitudine, și salvarea locației aflate la acele coordinate, in baza de date oferita de Firebase, Firestore. Locatia va fi salvata sub forma unui document, cu un identificator auto-generat, cu trei campuri: tara, oras si strada.

Componentele interfeței grafice:

- O componentă esențială de tip buton
- O componentă creată de către mine pentru afișarea latitudinii și longitudinii, doua componente de tipul buton și locațiile salvate anterior în baza de date

Prima componentă, de tipul buton, a interfeței este utilizata pentru preluarea coordonatelor geografice la apasare. Pentru a avea access la aceste date se utilizează o biblioteca specifica Expo, numita expo-location și este instalată cu ajutorul comenzii "*expo install expo-location*". Înainte de utilizarea funcțiilor din aceasta biblioteca,

permisiunile pentru acces la locație trebuie introduse în manifestul Android, pentru aplicatiile cu destinatie platforma Android:

```
<manifest ... >
  <!-- To request foreground location access, declare one of these permissions. -->
  <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
</manifest>
```

Figura 4.5: Permisiunile în manifestul Android (figura preluata din documentatia Android)

Pentru ca utilizatorul să primească cererea de acces la locație utilizăm carligul efect, cu al doilea parametru o lista goala deoarece dorim să se execute o singură dată la navigarea pe ecranul GPS. Cererea pentru permisiuni se face asincron, asteptându-se un răspuns din partea utilizatorului.

```
useEffect(() => {
    (async () => {
        let {status} = await Location.requestForegroundPermissionsAsync();
        if(status !== 'granted'){
            alert('Permissions not granted');
            return;
        }
    })().catch(err => console.log(alert(err.toString())));
}, []);
```

Figura 4.6: Cerere pentru permisiuni locație cu ajutorul carligului effect

Odata permisiunile acceptate, componenta buton pentru access la coordinate poate fi utilizata. Componentei i se atașează o funcție lambda care se rulează la un eveniment de tipul apasare, unde cu ajutorul unei metode din biblioteca “locație” se preia locatia curenta asincron, se modifica starea componentelor de afișare a latitudinii și longitudinii cu noua stare care afiseaza coordonatele. Cu ajutorul bibliotecii se poate implementa și modul de abonat, primind actualizari ale locației constant, acest lucru nu este implementat în aceasta aplicatie nefiind necesar pentru scopul ei.

Metoda utilizată pentru obținerea poziției are un singur argument unde se poate introduce acuratețea cu care se dorește obținerea alocației. Avem la dispoziție mai multe opțiuni printre care: acuratete scazuta, acuratețe echilibrata și acuratețe pentru navigare.

Testarea funcționalității se poate face și cu ajutorul emulatorului Android dar necesită activarea serviciului de locație și configurarea unei adrese care în momentul citirii, o sa fie returnată sub forma de coordonate.

```
const [coords, setCoords] = useState(0);

const onPressTurnOn = () => {
    (async()=>{
        //Obtinere a locatiei curente si setarea starii coordonatelor
        let location = await Location.getCurrentPositionAsync({});
        setCoords(location.coords);
    })();
};
```

Figura 4.7: Funcția lambda utilizata pentru obținerea locației curente

A doua componentă a ecranului GPS, este cea creata de catre mine pentru afisarea coordonatelor, prezentarea locatiilor din baza de date și a acțiunilor posibile pe acestea. Componenta este impacheta intr-o componenta esentiala de derulare (ScrollView), în cazul în care numărul locatiilor din baza de date depășesc dimensiunile ecranului.

Pentru a avea acces la datele obținute în componenta principala utilizam conceptul de recuzita a componentelor (props) care se comporta ca un parametru sau o lista de parametrii. Coordonatele obținute sunt trimise la aceasta componenta din componenta principala prin intermediul recuzitei. Aici aceste date sunt afișate sub forma de text.

Aceasta componentă implementează și două alte componente esentiale de tipul buton cu functionalitati diferite: un buton pentru opțiunea de a salva locația în baza de date și unul pentru a goli baza de date de locatii salvate.

La butonul de salvare îi este atașată o funcție lambda, utilizata la apăsarea butonului, unde o alta metoda din biblioteca “locație” care are ca si intrare coordonatele primite ca recuzita, in format latitudine, longitudine, și returnează un obiect, in continutul careia, printre multe altele, se afla tara, orașul și strada locației curente. Aceste informații se preiau și cu ajutorul metodelor bazei de date, Firestore, sunt utilizate pentru a salva locatia. La baza de date initializata la instalarea bibliotecii firebase îi este specificata colecția de documente în care sunt stocate locațiile și cu ajutorul metodei de adăugare (add) sunt introduse informațiile în campurile potrivite: tara, oras si strada. Acest lucru se face asincron iar la finalizarea adaugarii un mesaj de success este afișat.

```
const saveButtonHandler = () => {
    (async () => {
        let geo = await Location.reverseGeocodeAsync(props.coords);
        await database.collection('location').add({
            country: "" + geo[0].country,
            city: "" + geo[0].city,
            street: "" + geo[0].street
        }).then(() => {
            alert('Location added succesfully!');
        })

        let location = await database.collection('location').get().then((val) => {
            setLocations(val.docs);
        })
    })();
}
```

Figura 4.7: Funcția lambda utilizata pentru salvarea locației în baza de date

Butonul de golire a bazei date are o funcție lambda asemănătoare atașată acestuia și are ca acțiune ștergerea tuturor documentelor din colecția de locații și randarea listei afișate cu locatiile.

Pentru ștergerea documentelor din colecție, avem nevoie de o alta metoda a bazei de date, metoda preluare(get). Din nou este necesara specificarea colectiei inainte de apelarea metodei de preluare. Această metodă returnează un instantaneu al interogarii care contine o lista de documente. Baza de date dispune de o metoda pentru ștergerea întregii colecții(acțiune recomandată a se face de pe dispozitive mobile) dar nu și de ștergere a tuturor documentelor dintr-o colecție. Asa ca vom itera lista de documente din instantaneul de interogare, si vom sterge fiecare document in parte.

La sfarșitul ștergerii documentelor, în aceeași funcție folosim și carligul stare utilizat pentru lista de locații pentru a goli și lista prezenta în aplicație și a randa componenta.

```
const clearAllHandler = () => {
    try{
            database.collection('location'). get().then(snapshot => {
        snapshot.docs.forEach(element => {
            console.log(element.id);
            database.collection('location').doc("" + element.id).delete();
        });
    });
    setLocations([]);
}catch(e){
    alert(e.toString());
}
}
```

Figura 4.7: Funcția lambda utilizata pentru ștergerea tuturor locatiilor salvate in baza de date

În aceeași componentă creată de către mine, am utilizat etichetele “vedere” și “text” pentru a afișa fiecare intrare în baza de date sub forma unui tabel cu trei coloane reprezentand datele stocate: tara, oras si strada. Am afisat inițial numele coloanelor și

după care utilizând metoda de mapare pe lista de locații, am afisat fiecare locație pe un rand nou, ca un element al tabelului.

Pentru a nu se randa componenta de un numar prea mare de ori, lista de locații este actualizata doar in doua locuri: actualizată mai intai la prima randare a componentei pe ecran utilizând carligul efect și la fiecare salvare în baza de date. La stergere nu este necesara, deoarece ștergem toate locațiile nu doar una pentru a necesita actualizare, putem doar sa golim lista.

```
const GPSDataEntry = props => {
    const [locations, setLocations] = useState([]);
    useEffect(() => {
        (async () => {
            let location = await database.collection('location').get().then((val) => {
                setLocations(val.docs);
                console.log(locations);

            })
            //console.log(location);
        })();
    }, []);
```

Figura 4.8: Initializarea listei la prima randare a componentei

5.1.7 Ecranul Senzori

Ecranul Senzori implementează funcționalitatea de citire a senzorilor accelerometru și giroscop în timp real la momentul mișcării telefonului.

Accelerometrul măsoară accelerația pe axele x, y si z in metrii pe secunda si este utilizat pentru functionalitati precum afișarea unui meniu ascuns prin scuturarea dispozitivului mobil iar giroscopul măsoară rata de rotație în jurul axelor x,y si z in radiani pe secunda și este utilizat pentru functionalitati precum rotirea automată a unei poze în galeria unui dispozitiv mobil.

Pentru implementarea acestor doua funcționalități am utilizat biblioteca Expo pentru senzori “expo-sensors” care oferă diferite metode cu scopul de a accesa senzorii dispozitivului mobil printre care și accelerometrul și giroscopul.

Interfața grafică a ecranului Senzori este compusă din:

- O componenta de tipul buton
- Doua compozitii de afișare a valorilor pentru axele x, y si z, una pentru fiecare senzor în parte

Componenta de tipul buton este implementată în așa fel încat la apasarea ei aceasta să-și schimbe atat aspectul cat și comportamentul pentru a obține un efect de pornire/oprire a functionalitatii.

Pentru obținerea acestui comportament, utilizăm o funcție lambda atașată la evenimentul de apasare a butonului. Singurul lucru pe care-l facem în aceasta funcție este

de a seta intervalul de actualizare a valorilor date de senzori și modificarea unei variabile create cu ajutorul carligului stare, care reprezinta starea butonului, buton de pornire sau buton de oprire.

La apăsarea butonului facem comutarea dintr-o stare în alta, utilizând funcția de setare a carligului stare care genereaza randarea butonului cu noua valoare. În funcție de aceasta valoare afisam atat un text diferit pe buton cât și o alta culoare.

Odată schimbat aspectul, trebuie sa schimbam și comportamentul. Acest lucru poate fi făcut cu ajutorul unui alt carlig de tipul efect. De data aceasta ca si dependinta la carligul efect adaugam variabila, ceea ce rezulta intr-o rulare a codului din carligul efect de fiecare data cand starea butonului se schimba. În interiorul carligului efect, facem verificarea pentru a vedea în ce stare se afla butonul în acest moment și a rula comportamentul potrivit.

Dacă butonul este în starea de pornire, acesta rulează o alta functie lambda care se ocupă cu legarea variabilelor utilizate pentru afișare, la datele venite de la senzori prin conceptul de abonare. Procedeul de abonare se face cu ajutorul metodei clasei Giroscop respectiv Accelerometru, adauga ascultator (addListener).

Datele primite sunt legate de niște variabile de tipul stare, create cu ajutorul carligului stare, prin funcția de setare stare pentru randarea componentei de fiecare data cand avem valori actualizate. De asemenea, un lucru important este salvarea obiectelor returnate de aceasta metoda, atat pentru accelerometru cat și pentru giroscop deoarece vom avea nevoie de aceste obiecte pentru procedeul de dezabonare.

Dacă butonul este în starea de oprire, acesta dezaboneaza variabilele de la senzori cu ajutorul obiectelor salvate mai sus la abonare, actualizarea acestora pe ecran fiind oprită.

```
useEffect(()=> {
    if(isEnabled){

        _subscribe();
    }
    else{
        console.log('else ' + isEnabled);
        _unsubscribe();
        Gyroscope.removeAllListeners();
        Accelerometer.removeAllListeners();
    }
},[isEnabled]);
const onPressTurnOn = () => {
    console.log("Merge turn on!");
    _fast;
    setIsEnabled(previousState => !previousState);
};
```

Figura 4.8: Functia apelata la apasarea butonului de pornire a citirii de senzori

5.1.8 Ecranul NFC

În ecranul NFC, am implementat funcționalitatea de citire a unui card capabil de NFC cu ajutorul dispozitivului mobil. NFC este o tehnologie de comunicare între doua dispozitive, capabile de NFC, pe distanțe scurte prin intermediul undelor radio. Producătorii de dispozitive mobile utilizează aceasta tehnologie ca instrument de impartasire a datelor sau metoda de plata fără fir.

La momentul actual Expo nu are nici o biblioteca care să suporte functionalitatea NFC, așa că este necesară trecerea din fluxul de lucru gestionat în cel liber în cadrul Expo. Asta înseamnă pierderea unor beneficii, printre care și testarea codului nativ pe un dispozitiv mobil care nu este conectat direct la mediul de dezvoltare.

Procedeul de trecere din fluxul de lucru gestionat în cel liber se numește evacuare (eject) și se poate realiza cu ajutorul comenzii "*expo eject*". După executarea comenzii, dezvoltatorului i se cere un nume pentru pachetul android și un identificator pentru iOS, aici alegand sa pastrez valorile implicite. Odata finalizat procesul, folderul android este adaugat în contextul proiectului (iOS este adaugat doar în cazul dezvoltării pe dispozitive cu sistem de operare MacOS) și fluxul de lucru este acum unul liber.

În fluxul de lucru liber, avem posibilitatea de a folosii biblioteci care lucrează cu codul nativ ca și biblioteca de care avem nevoie în acest caz: react-native-nfc-manager, folosită pentru a avea access la functionalitatea de NFC.

Ecranul NFC, este compus din:

- O componenta de tipul buton
- O componenta de tipul modal – care apare "deasupra" ecranului pentru afișarea unui anumit mesaj
- O componenta de prezentare a listei
- O componenta de derulare

Componentă de tipul buton o folosim pentru activarea sau dezactivarea serviciului de citire NFC prin intermediul evenimentului de apasare, o functie lambda este apelata.

Funcția lambda are rolul important de a modifica aspectul și comportamentul butonului. Utilizând culoarea butonului ca și indicator al stării, vom modifica atat culoarea cat și textul butonului și în funcție de culoarea curentă(dacă este activ sau nu) modificam si comportamentul.

Dacă butonul este activ, trebuie sa setam vizibilitatea componentei de tipul modal pe adevărat, acest lucru fiind posibil prin adaugarea unei stări create cu carligul stare ca și valoare la parametrul de vizibilitate a modalului. Odată setata vizibilitatea cu ajutorul funcției de setare a stării, se instantiaza o clasa creata de catre mine pentru o utilizare mai usoara a metodelor din biblioteca NFC.

Odată clasa instanțiata, se pot utiliza metodele din aceasta. Mai întâi utilizăm metoda de inițializare pentru pornirea managerului NFC, după care încercăm să citim un dispozitiv NFC din apropiere.

Metoda de citire fiind una asincronă care returnează o promisiune, avem nevoie de metoda “then” pentru a gestiona valoarea returnată odată ce aceasta este disponibilă. In metoda “then”, adaugam informațiile despre dispozitivul gasit intr-o lista ca si text. În cazul în care avem și alte dispozitive deja scanate in lista, funcția de setare oferită de carligul stare permite obținerea valorii curente a listei ca mai apoi noul element pe care dorim sa il adaugam, sa fie introdus la finalul acesteia. Pentru a putea adăuga un element la finalul listei, avem nevoie de operatorul introdus în React “extindere” (spread). Acest operator permite extinderea unui obiect peste care se poate itera, în cazul meu l-am folosit pentru a extinde lista de dispozitive NFC deja existente și crearea unei noi liste cu elementul meu la final.

Componenta de tipul modal este utilizata pentru a anunța utilizatorul ca o scanare de dispozitive NFC este in progres si este necesara apropierea dispozitivului NFC de dispozitivul mobil. Aceasta este pusă la dispoziție de către biblioteca “react-native-modal”.

Componenta de prezentare este doar un text stilizat, cu un buton pentru golirea listei și are rol de prezentarea a listei dispozitivelor NFC scanate. Butonul are atasat o funcție lambda în care cu ajutorul funcției de setare stare, setam starea listei ca fiind goala.

Componenta de derulare este utilizată ca un container pentru lista de dispozitive NFC scanate, in cazul in care avem un numar ridicat de intrari in lista si acestea ies din dimensiunile ecranului. Utilizăm un carlig stare pentru lista, deoarece de fiecare data cand găsim un dispozitiv NFC nou, aceasta lista trebuie randata, pentru a fi afișată pe ecran în timp real

```
const addNFCReading = (val) => {
    setNFCReading(currentReadings => [...currentReadings, val + '    ' + dateTime.toString()]);
}

const onPressTurnOn = () => {
    let items = { ...state };
    console.log("Merge turn on!");
    if (color === color1) {
        //setare aspect buton
        setColor(color2);
        setButtonText("Disable NFC");
        //setare vizibilitate modal
        setVisible(true);

        //initializare NFCManager
        nfc.initNfc().catch(err => {
            //setVisible(false);
            alert('Init ' + err.toString());
        })

        //citire dispozitiv NFC
        nfc.readNdef().then(tag => {
            addNFCReading(tag.toString());
            setVisible(false);
        }).catch(err => {
           alert(err.toString());
        })
    }
    else {
        setColor(color1);
        setButtonText("Enable NFC");
    }
};
```

Figura 4.10: funcția lambda care gestionează comportamentul butonului, din ecranul NFC, la apasare

Clasa NFC creată are urmatoarele metode:

- initNFC – o metoda asincronă, utilizată pentru pornirea managerului NFC cu metoda managerului NFC “start”. Aceasta metodă este înconjurată cu un mecanism de tratarea a exceptiilor incearca-prinde (try-catch)
- checkEnabled – o metoda asincronă, utilizata pentru verificarea disponibilității funcției NFC, dacă funcția NFC este dezactivata pe dispozitivul mobil aceasta o sa returneze fals.
- readNdef – utilizata pentru citirea propriu-zisă a cardului NFC

Metoda de citire a cardului NFC returnează o promisiune. Promisiunea este ca și un container pentru un cod asincron, valoarea returnată de către metoda de citire NFC e

posibil sa nu fie cunoscuta la momentul apelarii. Acest concept returnează un obiect cu o metodă care permite gestionarea valorii odată ce aceasta este disponibila si poate fii prelucrata, numita "*then*". De asemenea, acest obiect returnat de promisiune oferă o metoda pentru gestionare în cazul în care valoarea ajunge sa nu fie disponibilă din cauza unei erori, numita "*catch*".

În interiorul containerului promisiune, se seteaza un ascultător cu ajutorul metodei din cadrul managerului NFC, setareAscultatorEveniment(setEventListener) în care se încearcă detectarea unui card/dispozitiv NFC din apropiere. Dacă acest dispozitiv NFC este găsit, promisiunea este rezolvata si valoarea returnată. După încercarea de descoperire, se seteaza încă un ascultător de eveniment dar de data aceasta pe evenimentul de sesiune închisă unde se face curatarea de ascultători(se seteaza un ascultător cu null, pe evenimentele de descoperire și sesiune închisă) iar dacă un dispozitiv nu a fost găsit, promisiunea este rezolvata cu valoare nula.

```
class Nfc {
    async initNfc() {
        try {
            await NfcManager.start();
        } catch (e) {
            throw e;
        }
    }

    async checkEnabled() {
        await NfcManager.isEnabled();
    }

    readNdef() {
        const cleanUp = () => {
            NfcManager.setEventListener(NfcEvents.DiscoverTag, null);
            NfcManager.setEventListener(NfcEvents.SessionClosed, null);
        }
        return new Promise((resolve) => {
            let tagFound = null;
            NfcManager.setEventListener(NfcEvents.DiscoverTag, (tag) => {
                tagFound = tag;
                resolve(tagFound);
                NfcManager.setAlertMessage('Ndef tag found ' + tagFound.name);
                NfcManager.unregisterTagEvent().catch(() => 0);
            });
            NfcManager.setEventListener(NfcEvents.SessionClosed, () => {
                cleanUp();
                if (!tagFound) {
                    resolve();
                } else {
                    console.log(tagFound.toString());
                }
            });
            NfcManager.registerTagEvent();
        });
    }
}
```

Figura 4.9: Clasa NFC creata pentru utilizarea bibliotecii NFC

5.1.9 *Ecranul Cameră*

În ecranul Cameră am implementat funcționalitatea de cameră foto, cu posibilitatea de a realiza o fotografie și afișarea acesteia într-un ecran separat numit Galerie.

Funcționalitatea de cameră este oferită atat de Expo cat si de React Native. Am decis sa aleg biblioteca oferită de React Native datorită faptului ca aceasta permite testarea functionalitatii pe emulatoare și aplicația a fost deja evacuata pentru implementarea functionalitatii de NFC.

Instalarea bibliotecii "react-native-camera" se face cu managerul de pachete npm și prin adaugarea permisiunilor necesare în manifestul Android:

```
<uses-permission android:name="android.permission.CAMERA" />
```

Figura 4.10: Permisiunea CAMERA în manifestul Android

Interfața grafică a ecranului Cameră este compusă din următoarele:

- O componentă de tipul buton pentru accesarea camerei
- O componentă de tipul buton pentru accesarea galeriei
- O componentă de tipul modal pentru cameră – inițial cu vizibilitatea setată pe fals
- O componentă de tipul modal pentru galerie – inițial cu vizibilitatea setată pe fals

Cele doua componente de tipul buton utilizează pentru evenimentul apasare o funcție lambda care au rolul doar de a seta vizibilitatea componentei de tipul modal pe adevărat, butonul pentru camera seteaza vizibilitatea pentru modalul camera iar cel pentru galerie seteaza vizibilitatea pentru modalul galerie.

Modalul cameră are parametrul de vizibilitate setat cu o valoare de tipul stare, pentru a putea fi modificat în timpul rulării. Aceasta componenta, odată setata vizibilitatea pe adevărat, acoperă întreg ecranul cu o componenta de tip RNCamera pusa la dispozitie de catre biblioteca instalata mai sus, react-native-camera. Pe langa componenta RNCamera, avem și două componente de tipul buton pentru capturarea fotografiei respectiv părăsirea modului camera.

Componenta RNCamera are doar un singur parametru care reprezinta referinta la camera. Valoarea parametrului este o funcție cu declansare la eveniment. La afisarea acestei componente, în funcție primim o valoare referinta pe care o salvăm pentru a o putea utiliza mai târziu pentru captura de imagine.

Butonul de captură de imagine are atașat o funcție lambda la evenimentul de apasare. Funcția lambda este una asincrona și în cazul în care referinta la camera nu este nulă, folosește o metoda din biblioteca pentru a captura fotografia.

Metoda folosită este “takePictureAsync” și returnează un obiect de tipul promisiune dacă o utilizăm în mod sincron iar dacă o utilizăm într-o funcție asincrona, returnează un obiect care are mai multe proprietăți printre care:

- Latime si inaltime
- Calea la imaginea salvata in memoria temporara cache

Obiectul obținut în urma capturii îl dăm ca și parametru unei alte funcții, care preia doar calea la imaginea salvată în memorie și îl adaugă într-o listă cu stare, creata cu ajutorul carligului stare. M-am folosit de același operator special de extindere pentru adaugarea caii în lista cu căile adaugate la fotografiile capturate anterior.

Butonul de părăsire a modului cameră utilizează doar o funcție lambda pentru a seta starea vizibilității pe fals pentru modalul camera.

```
const addPhoto = (data) => {
    setImageArray(currentImages => [...currentImages, { key: uuidv4(), value: data.uri }])
}

let camera;

const takePicture = async () => {
    if (camera) {
        const data = await camera.takePictureAsync();
        alert('Success!');
        addPhoto(data);
        console.log(imageArray);
    }
};
```

Figura 4.11: Funcțiile de capturare a fotografiilor și de a adăugare a imaginii în galerie

La fel ca și în cazul modalului camera, modalul galerie are ca valoare pentru parametrul de vizibilitate o variabila de tip stare pentru declanșarea randarii la schimbarea acesteia. Odată setata vizibilitatea pe adevărat, un nou ecran este afisat, proces asemanator cu afișarea unui ecran cu ajutorul navigatorului de comutare.

Noul ecran are în componență un buton de inchidere a modalului galerie și o componenta de derulare. Componenta de derulare este utilizata ca și container pentru prelucrarea listei de fotografii capturate. Fiecare imagine din lista este afișată cu ajutorul a doua componente, o componenta galerie creata de catre mine si un buton cu functionalitate de ștergere.

Componenta galerie creată de către mine primește ca și recuzită (props), calea la imagine și identificatorul unic și are rol doar de stilizare a felului în care imaginea este afișată pe ecran. Butonul cu functionalitatea de ștergerea imaginii, șterge mai intai imaginea din lista cu ajutorul filtrarii după identificatorul unic și după randeaza componenta pentru afișarea listei fără imaginea respective.

Ștergerea imaginii din memoria temporara se face cu ajutorul unei alte biblioteci React Native, react-native-fetch-blob, pentru accesarea fișierelor locale. Din biblioteca aceasta utilizam metoda de deconectare(unlink) pentru stergerea fotografiei din memorie, metoda se poate aplica fără a lua în considerare cazul în care imaginea nu exista deoarece nu arunca nici o eroare.

Pentru a putea face operațiunea de ștergere este necesar sa adaugam și permisiunile specifice in manifestul Android:

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
```

Figura 4.12: Permisiunile pentru acces la mediul de stocare extern in manifestul Android

```
<Button raised={true} title={"Delete"} color={COLORS.disableButton} onPress={() => {
    const newArray = imageArray.filter((item) => item.key !== image.key);
    setImageArray(newArray);
    var path = image.value.split("///").pop();
    RNFetchBlob.fs
        .unlink(path)
        .then(() => { alert("Image deleted!")})
        .catch(err => {
            alert(err);
        });
}} />
```

Figura 4.13: Functia de stergere a imaginii din memoria temporara

5.1.10Ecranul Bluetooth

Ecranul Bluetooth implementează funcționalitatea de descoperire a dispozitivelor Bluetooth cu consum de energie scăzut (Bluetooth Low Energy).

Expo nu oferă suport pentru Bluetooth deocamdată, așa ca pentru implementare am utilizat biblioteca dezvoltată pentru React Native “react-native-ble-plx”. Aceasta biblioteca impacheteaza o alta biblioteca care lucrează direct cu codul nativ, Java pentru Android și Swift pentru iOS.

Biblioteca oferă funcționalități precum:

- Scanarea de dispozitive
- Descoperirea de servicii disponibile la un anumit dispozitiv
- Posibilitatea de citire/scriere

De menționat că această opțiune nu este ideală în cazul în care se dorește comunicarea între doua dispozitive mobile BLE.

Interfața grafică a ecranului Bluetooth este compusă din:

- O componentă de tip buton

- O componentă esențială care cuprinde o componenta de tip text si una de tip buton
- O componentă de derulare

Componenta de tipul buton este utilizata pentru pornirea/oprirea scanarii de dispozitive. Butonul este unul cu stare și are atașat o funcție lambda care gestionează schimbarea de aspect și de comportament.

Funcția lambda atașată utilizează culoarea butonului pentru a decide comportamentul curent și aspectul stării următoare a butonului. Pentru a putea modifica aspectul butonului în timpul rulării, pentru titlu și stilizare se utilizează o variabila de tipul stare. În funcție de culoarea butonului, functionalitatea de scanare este oprita sau pornita.

Când se dorește pornirea functionalitatii, pe langa schimbarea stilizarii se apelează și o funcție separată pentru gestionarea scanării.

În interiorul funcției de gestionare a scanarii, se utilizează o clasă manager BLE din biblioteca instalata mai sus. Aceasta clasa este instantiata în afara componentei. Din aceasta clasa utilizam metoda startDeviceScan la care atasam o funcție la declansare eveniment(callback). Acest callback proceseaza doua variabile, o variabilă contine eroarea(dacă este prezentă) iar cealalta reprezinta obiectul dispozitiv scanat. Fiecare variabila este verificata, dacă variabila eroare exista, afisam un mesaj de eroare, dacă variabila dispozitiv exista o adaugat in lista cu ajutorul unei funcții de expediere.

```
const [scannedDevices, dispatch] = useReducer(reducer, []);

const reducer = ( state, action ) => {
    switch(action.type){
        case 'add':
            const device = action.device;
            if(device && !state.find((dev) => dev.id === device.id)) {
                return [...state, device];
            }
            return state;
        case 'clearAll':
            return [];
        default:
            return state;
    }
};

const scanDevices = () => {
    manager.startDeviceScan(null, null, (error, scannedDevice) => {
        if(error){
            alert(error.reason.toString());
        }

        if(scannedDevice){
            dispatch({type: 'add', device: scannedDevice});
        }

    });
    setTimeout(() => {
        manager.stopDeviceScan();
    }, 5000);
}
```

Figura 4.12: Funcția de scanare dispozitive și reductorul

Deoarece e posibil ca scanarea să aducă rezultate duplicate, avem nevoie de o metoda pentru a introduce in lista doar acele dispozitive care nu sunt deja acolo. Pentru acest lucru am decis sa utilizez carligul reductor (useReducer), o alternativa a carligului stare în cazul în care avem o logica mai complexă și trebuie sa gestionam mai multe stări.

Cârligul reductor primește ca și parametrii doua valori, o funcție de declansare eveniment numita reductor(reducer) și o stare inițială a obiectului, în cazul nostru al listei de dispozitive scanate. Acesta returnează doua valori, o lista care o sa o utilizam pentru stocarea dispozitivelor și o funcție de expediere (dispatch) pe care o utilizăm pentru trimiterea unor parametrii la reductor.

Reductorul procesează două variabile, variabila stare care reprezinta lista curenta si variabila acțiune unde primim parametrii trimiși de funcția expediere, parametrul tip și

parametrul dispozitiv care o sa contina dispozitivul scanat. În interiorul acestuia, tratam două cazuri bazate pe parametrul tip:

- Acțiunea de tip adaugare – o utilizăm pentru verificarea existenței dispozitivului în lista. Dacă dispozitivul exista deja în lista, returnam starea curentă iar dacă acesta nu exista, folosindu-ne de același parametru special de extindere, il adaugi in lista.
- Acțiunea de curățare – o utilizăm pentru a goli lista de dispozitive

Funcționalitatea de scanare, odata pornita, se opreste dupa 5 secunde sau la apăsarea butonului de către utilizator. Aceasta se oprește cu ajutorul unei alte metode a managerului BLE, metoda opresteScanareDispozitiv (stopDeviceScan).

Următoarea componentă utilizata este una de prezentare cu rolul de a introduce lista afișată de dispozitive. Aceasta cuprinde o componenta esentiala de tip text folosită pentru titlul listei și o componenta de tip buton care are functionalitatea de a goli lista. Pentru a goli lista ne folosim de cealalta actiune a reductorului, cea de curatare.

```
const clearBlueDevices = () => {
    dispatch({type: 'clearAll'});
};
```

Figura 4.14: Funcția de curatare a listei

5.2 Implementare în Flutter

5.2.1 Instalare Flutter

Flutter are o documentație vasta, bine pusă la punct care ghidează dezvoltatorul de la instalare la testarea aplicației.

Pentru utilizarea ecosistemului Flutter, este necesară descărcarea kitului de dezvoltare Flutter. Dezvoltatorul are posibilitatea de a alege între descarcarea unei arhive a ultimei versiuni stabile sau accesarea codului sursa utilizand platforma git. Am ales sa descarc arhiva pe care am dezarhivat-o intr-o locatie pe calculatorul utilizat pentru dezvoltare.

Odată dezarhivat, Flutter pune la dispoziție o comanda “flutter doctor” care verifica mediul de dezvoltare și oferă un raport cu ce este în neregula și pași de rezolvare. Dupa rularea comenzii, se rezolva problemele apărute și se mai rulează odată pentru a te asigura ca totul e în regula.

Pentru testarea aplicației, ca și instalare aditionala, am avea nevoie de programul Android Studio care pune la dispoziție functionalitatea de emulator Android. Odata instalat programul, se creeaza un dispozitiv virtual.

5.2.2 *Editor de cod*

Deși codul Dart se poate scrie în orice editor de cod/text, Flutter oferă suport pentru patru editoare de cod, fiecare avand propria secțiune de configurare în documentația ecosistemului: Android Studio, IntelliJ, Visual Studio Code si Emacs. Am decis utilizarea aceluiași editor de cod ca și în cazul React Native, pentru o mai buna comparatie între procesele de dezvoltare.

Pentru configurarea editorului de cod cu scopul dezvoltării de aplicații Flutter, documentația sugerează instalarea extensiei VS Code "Flutter". Aceasta extensie pune la dispoziție în paleta de comenzi, comenzi preconfigurate utile in ecosistemul Flutter precum "flutter doctor". Pe langa acestea, extensia configurează un depanator și o secțiune de alegere a platformei pe care se dorește testarea. În aceasta secțiune o sa apara toate emulatoarele deschise, dispozitivele mobile conectate la mediul de dezvoltare si platforma web, care permite testarea aplicației în navigatorul web.

Pe lângă această extensie principală, am utilizat și altele pentru o dezvoltare mai lină:

- Pubspec Assist – utilizat pentru instalarea pachetelor Flutter și Dart, nefiind necesara adaugarea manuala a lor în fișierul de configurare pubspec.yaml. Permite instalarea pachetelor multiple
- Bracket Pair Colorizer – extensie utilizata pentru a prezenta parantezele din cod cu diferite culori. Foarte utilă la scrierea de cod care implica elemente grafice (widgets)
- Colors – utilizata pentru afisarea culorii reprezentate de codul HEX.

5.2.3 *Navigare în aplicație*

Aplicațiile Flutter pot sa utilizeze atat elemente de interfata de tipul Material(specifice Android) cat si de tipul Cupertino(specific iOS). Aceste doua tipuri de elemente au fost introduse pentru a oferii un sentiment nativ aplicației, și o consistentă cu restul aplicațiilor instalate pe dispozitivul mobil. Flutter ofera doua clase, MaterialApp is CupertinoApp, care au rol de container pentru grupuri de elemente de interfață des utilizate în dezvoltarea aplicațiilor. În cazul de fata am utilizat clasa MaterialApp, aplicația fiind dezvoltată pentru Android deși atat MaterialApp cat si CupertinoApp pot fi utilizate pe ambele platforme ba chiar utilizate împreuna în aceeași aplicație.

Navigarea în Flutter se poate face cu ajutorul Navigatorului. Navigatorul este un element de interfata(widget) utilizat pentru gestionarea unei stive de ecrane. Aceste ecrane poarta numele de rute (routes) si sunt adaugate sau eliminate din stiva, elementul din varf reprezentand ecranul curent. Clasa MaterialApp are un navigator integrat si nu este necesara instalarea unui pachet separat pentru aceasta functionalitate.

Putem utiliza doua tipuri de rute, rute cu nume sau rute anonime. Pentru ecranele principale am decis sa utilizez rute cu nume. Fiecare element de interfata grafica care reprezinta un ecran defineste o constanta. Aceasta constanta este o valoarea de tipul string de forma "/nume_ecran" si reprezinta ruta pe care trebuie sa o urmeze navigatorul ca sa ajungă la acest ecran.

După definirea fiecărei rute în interiorul ecranelor principale, le adun pe toate într-un obiect de tipul clasa pentru includerea mult mai ușoară a rutelor unde am nevoie.

Pentru înlocuirea unui ecran, Navigatorul oferă mai multe metode printre care și:

- pushReplacementNamed – utilizat pentru adăugarea în stiva a unei rute cu nume; starea rutei curente nu se mai pastreaza
- pushNamed – utilizat pentru adăugarea în stiva a unei rute cu nume, pastrand posibilitatea de a te întoarce la ecranul curent ; starea rutei curente se pastreaza
- pop – elimină ecranul curent, afisandu-l pe cel de dinaintea lui în stiva;

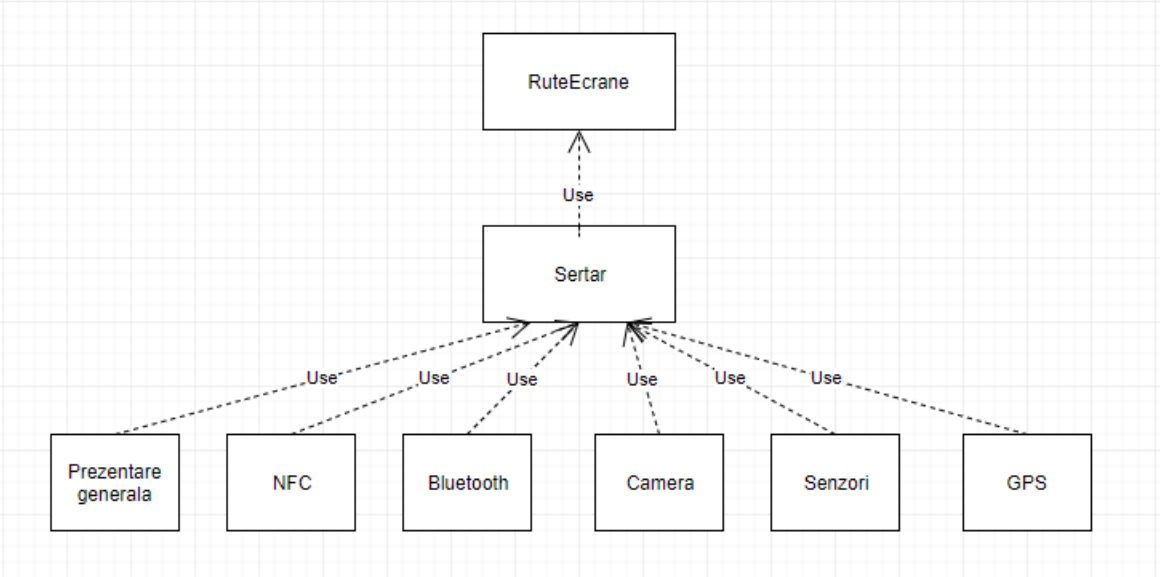

Figura 4.14: Reprezentare a navigarii in aplicație

5.2.4 *Autentificare și înregistrare*

Pentru ecranele de autentificare și înregistrare am folosit doua elemente de interfață grafică cu stare. Navigarea între acestea am realizat-o cu ajutorul unei rute anonime și păstrarea stării ecranului de autentificare în momentul tranziției la cel de înregistrare.

La fel ca și la implementarea aplicației React Native, am folosit serviciul Firebase. Instalarea pachetelor necesare am facut-o direct din editorul de cod cu ajutorul extensiei Pubspec Assist.

Atât ecranul de autentificare cât și cel de înregistrare, conțin un formular pentru introducerea datelor necesare, email si parola, si un buton de functionalitate.

În cazul autentificarii, butonul de functionalitate utilizează o funcție lambda asincrona pentru validarea formularului cu ajutorul metodei validare a acestuia și verificarea accesului la aplicație a utilizatorului. Verificarea accesului la aplicație se face cu ajutorul metodei Firebase signInWithEmailAndPassword. Dacă metoda aceasta returnează o excepție de tipul FirebaseAuth aceasta este "prinsa" cu ajutorul conceptului "încearcă-prinde" și o eroare specifica este afișată. Dacă datele sunt corecte, aplicatia tranzitioneaza la ecranul principal de prezentare generala.

În cazul în care utilizatorul nu este înregistrat, accesul la elementul de interfața înregistrare se face cu ajutorul unui alt buton de pe ecranul autentificare. Butonul de functionalitate din ecranul înregistrare apelează o funcție lambda asincrona care preia datele din campurile formularului și cu ajutorul unei alte metode din Firebase createUserWithEmailAndPassword se creeaza noul utilizator și ecranul este scos din stiva, ramanand afisat cel de autentificare. Dacă apare o excepție de tipul FirebaseAuth din nou aceasta este capturata și eroarea specifica afișată.

Pentru a se recunoaște câmpurile potrivite pentru preluarea de informații, fiecare formular are generata o cheie globală unică, pe post de identificator.

```
ElevatedButton(
    child: Text('Login'),
    onPressed: () async {
      final form = _formKey.currentState;
      form!.save();

      if (form.validate()) {
        try {
          await FirebaseAuth.instance
              .signInWithEmailAndPassword(
            email: _email,
            password: _password,
          );
          ScaffoldMessenger.of(context).showSnackBar(snackBar);
          Navigator.pushReplacementNamed(
              context, pageRoutes.overview);
        } on FirebaseAuthException catch (e) {
          showAlert(context, e.toString());
        }
      }
    },
    style: ElevatedButton.styleFrom(primary: enableButton)),
```

Figura 4.15: Elementul de interfața grafica de tipul buton cu functionalitatea de autentificare

```
ElevatedButton(
    child: Text('Register'),
    onPressed: () async {
      try {
        var credential = await FirebaseAuth.instance
            .createUserWithEmailAndPassword(
                email: email, password: password);
        Navigator.pop(context);
      } on FirebaseAuthException catch (e) {
        showAlert(context, e.toString());
      }
    },
    style: ElevatedButton.styleFrom(primary: enableButton))
],
)));
```

Figura 4.16: Elementul de interfața grafica de tipul buton cu functionalitatea de înregistrare

5.2.5 Meniul principal

La fel ca și în cazul React Native, pentru meniul principal am utilizat conceptul de "sertar" (Drawer). În cazul Flutter, clasa MaterialApp are un element de interfata grafica cu aceasta functionalitate. Pentru implementarea lui am creat o clasa care extinde clasa de element de interfața fără stare care returnează acest sertar. Sertarul returnat are o proprietate numita "copil" (child) unde putem specifica lista de elemente care reprezinta ecranele principale.

Lista de elemente este una personalizată fiecare avand un titlu, numele ecranului, și o funcție atașată la declanșarea evenimentului de apasare. In aceste functii, sertarul este închis și se face rutarea la ecranul specificat cu ajutorul rutelor cu nume.

```
ListTile(
    title: Text('NFC'),
    onTap: () {
      Navigator.pop(context);
      Navigator.pushReplacementNamed(context, pageRoutes.nfc);
    }),
```

Figura 4.17: Elementul sertarului pentru ecranul principal NFC

La crearea unui element de interfață grafică care să reprezinte un ecran, este necesara utilizarea unui container pentru restul elementelor. Acest container se numeste schela (Scaffold) si reprezinta structura compozitionala a ecranului.

Pentru a utiliza sertarul ca si meniu principal, schela menționată mai sus vine cu proprietatea speciala "schela" care primește ca și parametru clasa creata de catre mine.

Schela se asigura ca sertarul este prezent sub forma unui buton în bara de titlu care la apasare afișează clasa personalizată sertar. Clasa sertar creata este adaugata în fiecare schela a ecranelor principale pentru acces la meniul principal de oriunde din aplicație.

5.2.6 Ecranul GPS

Pagina GPS implementează funcționalitatea de locație curentă și la fel ca oricare element grafic de interfata care reprezinta un ecran principal, are la baza o schela. Pentru schela utilizăm trei proprietăți:

- Bara de titlu(appBar)
- Corpul schelei (body)
- Sertar (Drawer) – unde utilizam clasa creata special pentru asta

În bara de titlu, adaugam un element esențial cu același nume unde avem posibilitatea setarii anumitor acțiuni. O actiune necesara este cea de delogare a utilizatorului. Acțiunea este reprezentată de o icoana cu functionalitate de buton în care cu ajutorul navigatorului și a metodei Firebase delogam utilizatorul.

Corpul schelei este format din elementele de interfață grafică care aparțin acestui ecran. Pentru o mai buna parcurgere a codului, am extras toata functionalitatea necesara in ecranul GPS, într-un element cu stare cu numele ActivareGPS(EnableGPS).

Elementul ActivareGPS este compus din următoarele:

- Buton de functionalitate pentru activare serviciu
- Elemente esentiale de tipul text
- Butoane pentru salvarea in baza de date respectiv golirea ei
- Un alt element cu stare creat de către mine pentru prelucrarea locatiilor din baza de date

Butonul de funcționalitate are rolul de a prelua locația curentă și de a afisa coordonatele, latitudine si longitudine in elementele esentiale de tipul text. Pentru a putea avea access la acest serviciu nativ vom utiliza metoda, din pachetul Dart “geolocator”, de preluare a locației curente getCurrentPosition.

La butonul de funcționalitate atașăm o funcție lambda asincronă care să gestioneze preluarea de date și afișarea lor. Odată preluate datele cu metoda preiaPozitieCurenta, acestea sunt salvate în variabile care mai apoi sunt preluate de către elementele esentiale de tip text pentru afișare. Pentru afișarea lor la fiecare schimbare a valorilor se utilizează funcția disponibilă implicit a elementelor de interfață grafică cu stare, setareStare(setState) pentru reconstruirea elementelor cu noile valori.

Odată afișate datele, avem nevoie de coordonate pentru aflarea informațiilor necesare pentru salvarea in baza de date: tara, oras si strada. Acest lucru este posibil cu ajutorul pachetului Dart “geocoding” care oferă funcția “locatieDinCoordonate”

(placemarkFromCoordinates). Aceasta functie ia ca și parametrii latitudinea și longitudinea aflata anterior.

Având toate informațiile necesare, trebuie sa le salvăm într-un obiect pentru procesare ulterioară. Am decis crearea unei clase Locație cu un constructor care creeaza obiecte de tipul Locație.

Declarăm un obiect de tipul Locație cu modificatorul "tarziu" (late) în afara funcției lambda utilizate pentru butonul de functionalitate si îl definim în interiorul funcției după obținerea informațiilor necesare. Modificatorul "târziu" permite definirea valorii obiectului în viitor.

```
Future<void> _getLocation() async {
  var currentLocation;

  try {
    currentLocation = await Geolocator.getCurrentPosition(
        desiredAccuracy: LocationAccuracy.best);
    List<Placemark> placemarks = await placemarkFromCoordinates(
        currentLocation.latitude, currentLocation.longitude);
    print(placemarks.toString());
    latitude = currentLocation.latitude.toString();
    longitude = currentLocation.longitude.toString();
    loc = new Location(
        country: placemarks.first.country.toString(),
        city: placemarks.first.locality.toString(),
        street: placemarks.first.street.toString(),
        latitude: latitude,
        longitude: longitude);
    setState(() {
      items.add(loc);
    });
  } catch (e) {
    showAlert(context, e.toString());
  }
}
```

Figura 4.18: Preluarea informațiilor necesare legate de locația curentă

Odată afișate coordonatele, utilizatorul are opțiunea de a le salva în baza de date prin apăsarea butonului "salvare" (save). Pentru stocarea datelor am utilizat serviciul oferit de Firebase, Firestore la fel ca si in cazul React Native. Firestore oferă o metoda de adaugare a datelor sub forma de documente.

Butonul de salvare are atașat o funcție care la declanșarea evenimentului de apasare, utilizează metoda Firestore de adăugare (add) pentru a salva informațiile din obiectul definit anterior in baza de date sub forma de document.

O alta opțiune pe care o are utilizatorul este cea de ștergere a tuturor locatiilor salvate anterior prin apăsarea butonului stergeTot (clearAll). Acesta are atasat o funcție lambda în care se preia colecția "locații" din Firestore, se itereaza peste aceasta și pentru fiecare document din colecția se apelează metoda ștergere(delete), ștergerea întregii colecții nefiind recomandata de pe dispozitivele mobile sau din navigatorul web, doar din consola pusa la dispozitie de catre Firebase.

```
void saveLocation() {
  CollectionReference locations =
      FirebaseFirestore.instance.collection('locations');
  try {
    locations.add({
      'country': loc.country,
      'city': loc.city,
      'street': loc.street
    }).catchError((error) => print('Erroareee'));
  } catch (e) {
    showAlert(context, 'Eroare');
  }
}

void deleteAll() {
  CollectionReference locations =
      FirebaseFirestore.instance.collection('locations');

  locations.get().then((value) => {
        for (DocumentSnapshot ds in value.docs) {ds.reference.delete()}
      });
}
```

Figura 4.19: Funcțiile lambda pentru functionalitatile de salvare și ștergere

Pentru afișarea listei de locații din baza de date, am creat un nou element grafic de interfata cu stare pe care l-am utilizat in elementul ActivareGPS, LocationsData în interiorul căruia am utilizat conceptul de curent (Stream) utilizat pentru ascultarea schimbărilor din baza de date fara a fi necesara apelarea functiei de setare a stării la o anumită perioada.

Acest concept este implementat cu ajutorul elementului de interfața StreamBuilder. Acest element are doua proprietăți obligatorii:

- Stream – care primește un obiect curent de tipul instantaneu al interogării. Obiectul este definit la construirea elementului de interfata cu ajutorul metodei instantanee(snapshots) al bazei de date Firestore. Aceasta metoda returnează un instantaneu al interogării în care regasim toate documentele din colecția "locații".

- Builder – in aceasta proprietate procesam instantaneul, afisand sub forma de tabel toate locațiile din baza de date. Pentru afișarea datelor utilizăm elementul de interfața grafica predefinit DataTable care are de asemenea două proprietăți: coloane(columns) unde setam numărul și numele coloanelor, si randuri(rows) unde cu metoda de mapare a listelor afisam fiecare locație pe un rand nou.

Deoarece există posibilitatea ca în interiorul bazei de date sa avem un număr mare de locații și prin afișarea lor într-un tabel acesta sa depaseasca dimensiunile ecranului, tot ecranul este adaugat într-un element de interfata grafica cu functionalitate de derulare, care are rolul de container.

```
class _LocationsDataState extends State<LocationsData> {
  final Stream<QuerySnapshot> _locationStream =
      FirebaseFirestore.instance.collection('locations').snapshots();

  @override
  Widget build(BuildContext context) {
    return StreamBuilder(
        stream: _locationStream,
        builder: (BuildContext context, AsyncSnapshot<QuerySnapshot> snapshot) {
          if (snapshot.hasError) {
            return Text('Error');
          }

          if (snapshot.connectionState == ConnectionState.waiting) {
            return CircularProgressIndicator();
          }

          return new DataTable(
              columns: [
                DataColumn(label: Text('Country')),
                DataColumn(label: Text('City')),
                DataColumn(label: Text('Street'))
              ],
              rows: snapshot.data!.docs.map((DocumentSnapshot doc) {
                Map<String, dynamic> data = doc.data() as Map<String, dynamic>;
                return new DataRow(cells: [
                  new DataCell(Text(data['country'])),
                  new DataCell(Text(data['city'])),
                  new DataCell(Text(data['street']))
                ]);
              }).toList());
        });
  }
}
```

Figura 4.20: Afișarea listei de locații din baza de date

5.2.7 Ecranul Senzori

Ecranul Senzori implementează funcționalitățile de accelerometru și giroscop, afișarea valorilor în timp real.

La fel ca si ecranul GPS, ecranul Senzori are la bază un schelet cu aceleași proprietăți. Singurul lucru diferit este corpul scheletului. Pentru corpul scheletului am create un nou element de interfața grafica ActivareSenzori(EnableSensors in care implementam intreaga functionalitate.

Elementul ActivareSenzori este format din următoarele subelemente:

- Element de tipul buton
- Element prezentare de tipul card pentru afișarea datelor accelerometrului
- Element prezentare de tipul card pentru afișarea datelor giroscopului

Butonul de funcționalitate se ocupă de activarea citirii valorilor în timp real de la senzori. Acesta are doua stări cu aspect și comportament diferite care sunt gestionate cu funcția lambda atașată acestuia.

Funcția lambda verifică dacă butonul este activ sau nu, schimband comportamentul și aspectul ca atare. În cazul în care butonul este activ, folosind funcția de setare a stării pentru reconstruirea elementului, se schimba titlul, starea butonului si variabilele care reprezinta valorile senzorilor pe axele x,y și z.

```
void buttonHandler() {
  if (isEnabled == false) {
    setState(() {
      title = 'Disable';
      isEnabled = true;
    });
    subscription.add(accelerometerEvents.listen((AccelerometerEvent event) {
      setState(() {
        accX = double.parse(event.x.toStringAsFixed(3));
        accY = double.parse(event.y.toStringAsFixed(3));
        accZ = double.parse(event.z.toStringAsFixed(3));
      });
    }));

    subscription.add(gyroscopeEvents.listen((GyroscopeEvent event) {
      setState(() {
        gyroX = double.parse(event.x.toStringAsFixed(3));
        gyroY = double.parse(event.y.toStringAsFixed(3));
        gyroZ = double.parse(event.z.toStringAsFixed(3));
      });
    }));
  } else {
    setState(() {
      title = 'Enable';
      isEnabled = false;
    });
    subscription.forEach((element) {
      element.cancel();
    });
  }
}
```

Figura 4.21: Funcția care gestionează activarea și dezactivarea citirii senzorilor

Pentru obținerea valorilor de la senzori se utilizează un pachet Dart numit “sensors_plus” care permite adaugarea unui observator pe cei doi senzori. Din nou se utilizează conceptul de curent.

Avand nevoie de doi curenți pentru ascultarea evenimentelor venite de la ambii senzori, o sa utilizăm un curent de abonamente. Aplicam metoda statica de ascultare(listen) a claselor evenimenteGiroscop(gyroscopeEvents) si evenimenteAccelerometru(accelerometerEvents) și după care adaugam curenții returnati de aceștia în curentul de abonamente. Utilizăm aceasta metoda pentru a putea opri ascultarea la nevoie.

În interiorul metodei de ascultare, utilizăm o funcție cu declansare la eveniment, unde cu ajutorul funcției disponibilă implicit în elementele grafice cu stare, setareStare(setState) ecranul se reconstruieste afisand de fiecare data valorile in timp real pe axele x,y si z a celor doi senzori.

Dacă butonul este în starea de dezactivare, la apasare acesta își revine la starea inițială și itereaza prin curentul de abonamente apeland pentru fiecare metoda de oprire a ascultării (cancel). Odata oprita ascultarea, valorile senzorilor nu se mai actualizează.

Pentru ca observatorii să nu rămână activi după schimbarea ecranului, se suprascrie funcția mostenita de la elementul de interfata grafica cu stare, “distrugere”(dispose) care se rulează la distrugerea elementului. În aceasta functie mostenita adăugăm și iterarea pe curentul de abonamente, pentru a opri ascultarea.

5.2.8 Ecranul NFC

Ecranul NFC implementează funcționalitatea de citire a unui card NFC. Pentru utilizarea serviciului nativ NFC, am decis sa utilizez pachetul Dart “nfc_manager” .

Pentru corpul scheletului, creăm un nou element de interfață grafică cu stare numit Lista(List) unde o sa procesam intreaga functionalitate. Elementul are o interfata grafica simpla compusă din următoarele:

- Un element de tipul buton pentru activare/dezactivare
- Un element de tipul lista pentru afișarea informațiilor despre cardurile NFC scanate

Butonul are rolul de a gestiona citirea cardului NFC și de a schimba comportamentul și aspectul butonului. La declanșarea evenimentului de apasare se apelează o funcție asincrona în care este implementată logica de citire.

Pentru a putea urmării starea butonului, utilizam o variabila de activare (isEnabled) pe care o verificam la intrarea in functia asincrona. Dacă nu este activat încă butonul, schimbăm starea variabilei în adevărat și afisam un element de încărcare afișat din zona inferioară a ecranului (BottomLoader). Acest element îl utilizăm din pachetul Dart “bottom_loader” și oferă o reprezentare grafica a scanarii cardului NFC.

În timp ce elementul de încărcare este activ, se verifica disponibilitatea functionalitatii NFC. Dacă aceasta este disponibilă se începe sesiunea de căutare a unui dispozitiv NFC care poate fi scanat. Sesiunea se porneste cu ajutorul metodei pornireSesiune(startSession) a instanței managerului NFC din pachetul “nfc_manager”. Dacă un card NFC este descoperit acesta este adaugat unei liste cu ajutorul funcției de setare a stării pentru a reconstrui elementul cu un nou membru al listei.Indiferent dacă un dispozitiv NFC a fost găsit sau nu, elementul de încărcare este închis și sesiunea de căutare oprită.

În cazul în care funcționalitatea NFC nu este disponibilă, elementul de încărcare se închide direct și se afișează o alertă cu un mesaj specific.

Cealalta componenta a ecranului este un constructor de lista cu două proprietăți importante: itemCount care primește lungimea listei și itemBuilder care returnează fiecare element al listei cu ajutorul unui element grafic.

```
if (isEnabled) {
  BottomLoader bl = new BottomLoader(context, isDismissible: true);
  bl.style(message: 'Scanning NFC...');
  await bl.display();

  bool isAvailable = await NfcManager.instance.isAvailable();

  print(isAvailable);
  if (isAvailable) {
    NfcManager.instance.startSession(onDiscovered: (NfcTag tag) async {
      setState(() {
        items.add(tag.toString());
      });

      NfcManager.instance.stopSession();
      bl.close();
      return;
    });
  } else {
    Future.delayed(Duration(seconds: 5)).whenComplete(() => {
          bl.close(),
          showAlert(context, 'NFC not available. Check if it is enabled')
        });
  }
```

Figura 4.22: Functionalitatea de citire NFC

5.2.9 *Ecranul Cameră*

Ecranul Cameră implementează funcționalitatea de captură foto, salvare imagini în memoria dispozitivului mobil și afișarea acestora într-o galerie de imagini. Pentru implementarea funcționalității utilizăm pachetul Dart “camera”.

Scheletul corpului acestui ecran este compus dintr-un element de interfață grafică cu stare care afișează doua elemente esentiale de tipul buton. Fiecare buton integrează functionalitatea de navigare spre ecrane diferite, utilizând conceptul de rutare anonimă.

Butonul de funcționalitate Cameră tranzitează din ecranul Camera în cel de captura foto. Navigarea se face cu ajutorul metodei de adaugare in stiva de navigare(push) care pastreaza starea ecranului anterior. Navigatorul permite trimiterea de informații prin parametrii.

În acest caz avem nevoie sa trimitem din ecranul principal de Camera la cel de captura foto a informațiile despre camerele disponibile în dispozitivul mobil. Aceste informații sunt preluate asincron la inițializarea elementului de interfata grafica cu funcția care aparține pachetului “camera”, availableCameras. Functie returneaza o lista de obiecte care reprezinta camerele de fotografiat disponibile si tipul acestora, fata sau spate.

Odată preluate camerele le pasam navigatorului la adaugarea ecranului de captura foto în varful stivei pentru afișare.

La construirea elementului de interfață grafică pentru acest ecran, se apelează funcția mostenita de inițializare, pe care o suprascriem și adaugam inițializarea camerei de pe poziția 0 din lista, care în majoritatea cazurilor este cea din spate. Pentru a utiliza camera avem nevoie de un obiect de gestionare a camerei (CameraController) pe care îl putem obține prin instanțierea clasei din pachetul "camera", ControlCamera(CameraController). Constructorul clasei ia ca și parametrii, obiectul de tip camera pe care dorim sa-l gestionam si rezoluția dorită.

Inițializarea camerei se face cu metoda obiectului de control camera, initializare(initialize) care returnează un viitor (conceptul de Future). Acest viitor îl utilizăm într-un element de interfața grafica numit ConstructorViitor(FutureBuilder) care utilizează un instantaneu (snapshot) al unui viitor pentru a construi alte elemente de interfata grafica. Elementul ConstructorViitor are doua proprietăți importante:

- Viitor (future) – care primește obiectul de tipul viitor
- Constructor (builder) – care primește o funcție cu declansare la eveniment

În funcția cu declansare la eveniment, se prelucrează starea conexiunii instantaneului. Dacă conexiunea este realizată și avem acces la valoarea variabilei sau erorii returnate de viitor, returnam un container cu proprietatea copil avand valoarea unei alte clase utilizate din pachetul "camera", clasa PrevizualizareCamera (CameraPreview).

Aceasta clasă primește ca și parametru obiectul de control al camerei obtinut la initializare și afișează camera în timp real. Pentru afișarea camerei pe ecranul complet, utilizăm alte două proprietăți a containerului, cea de înălțime și cea de lățime, care iau ambele aceeași valoarea de infinit (double.infinity).

```
if (snapshot.connectionState == ConnectionState.done) {
  return Container(
      child: CameraPreview(ctrlCurrentCamera),
      height: double.infinity,
      width: double.infinity);
} else {
  return const Center(child: CircularProgressIndicator());
}
```

Figura 4.23: Verificare starea conexiunii viitorului și afișarea elementului potrivit

Dacă conexiunea nu este încă realizată, afisam un element grafic de interfata din clasa MaterialApp, numit indicator de progres circular (CircularProgressIndicator).

Odată afișată previzualizarea camerei, avem nevoie de un element de tipul buton pentru capturarea fotografiei. Am decis utilizarea unui element de interfata grafica din clasa MaterialApp, numit ButonActiunePlutitor(FloatingActionButton). Acesta permite afișarea unui buton cu fundal transparent și are atasat o functie asincrona pentru functionalitatea de capturare care este realizata cu ajutorul metodei obiectului de gestionare a camerei capturaFoto(takePicture).

Metoda de captură foto returnează un obiect în care se afla informațiile legate de stocarea imaginii. Aceasta este stocata initial in memoria temporara. Dupa captura foto, utilizatorul îi este prezentat, cu ajutorul unei ferestre de tipul dialog, o copie a imaginii de dimensiuni reduse si doua optiuni:

- Salvare (save) – care salveaza imaginea capturata în memoria internă a dispozitivului mobil, în directorul aplicației într-un folder nou creat cu numele "galerie". Pentru acțiunile de tipul accesare fisiere și directoare am utilizat doua pachete Dart "path" și "path_provider".
- Anulare (cancel) – care elimina fereastra de tipul dialog și nu salvează fotografia

```
Future<void> takePhoto() async {
  try {
    final image = await ctrlCurrentCamera.takePicture();
    currentPhoto = image;
    showPhotoPreview(image.path);
  } catch (e) {
    print(e);
  }
}

Future<void> savePhoto(BuildContext context) async {

  final appDir = await getApplicationDocumentsDirectory();
  final filename = Path.basename(currentPhoto.path);
  final path = appDir.path;
  final savedImage =
      await currentPhoto.saveTo('${appDir.path}/gallery/$filename');

  Navigator.of(context).pop();
}
```

Figura 4.24: Funcțiile utilizate pentru captura foto și salvare fotografie

Pentru reîntoarcerea în ecranul principal Camera, avem nevoie de un buton cu funcționalitatea de eliminare din stiva a ecranului curent. Acest lucru l-am realizat prin adaugarea unei bare de titlu cu opacitate 0 și un element de tipul icoana, o sageata la stanga. Icoana are atașată funcționalitatea oferită de navigator de eliminare a ecranului din varful stivei.

Pe lângă butonul de captura foto și cel de eliminare a ecranului, am implementat, in plus fata de aplicația React Native, și functionalitatea de schimbare cameră. La apăsare, utilizatorul are acces la camera de pe poziția întâi din lista (camera frontala) daca aceasta exista. Functionalitatea este prezentata cu ajutorul unui element de tipul icoana și adaugata ca și acțiune în bara de titlu.

În cazul în care se dorește vizualizarea fotografiilor capturate și salvate, am implementat ecranul de galerie. Accesul la ecranul galerie se face prin apăsarea butonului din ecranul principal dedicat galeriei, care are atasat o funcție care realizeaza două acțiuni. Mai intai, preia calea fiecărei imagini din directorul creat special pentru salvarea pozelor și le salvează într-o lista iar apoi, utilizând navigatorul face tranziția de la ecranul principal la cel al galeriei. Utilizandu-se funcția simpla de adăugare, navigatorul implementează functionalitatea de reintoarcere la ecranul principal automat in bara de titlu. Ca și parametru pentru galerie, se trimite lista cu calea fiecărei fotografii salvate.

Pentru interfața grafică a elementului galerie am utilizat o compoziție de tipul coloana. Utilizand un constructor de lista separata (adaugă posibilitatea de a adăugare a unor componente goale intre elementele listei) am afisat fiecare imagine determinată de către calea din lista cu ajutorul elementelor de tipul card. Fiecare element de tipul card continue următoarele componente:

- Imagine(Image) – componenta utilizata pentru afișarea unei imagini pe baza caii.
- Buton (ElevatedButton) – pentru implementarea functionalitatii de ștergere

Funcționalitatea de ștergere am implementat-o prin obținerea unui obiect de tipul Fișier(File) pe baza caii din lista și utilizarea metodei “ștergere” (delete) specifice obiectului fișier. Odata imaginea ștearsă se utilizează funcția de setare stare pentru reconstruirea elementului de interfața grafica.

```
void deleteFile(path) async {
  final appDir = await getApplicationDocumentsDirectory();
  final file = File(path);
  try {
    await file.delete();
    setState(() {
      list.removeWhere((item) => item.path == path);
    });
  } catch (e) {
    print(e);
  }
}
```

Figura 4.24: Funcția de ștergere a imaginilor

5.2.10Ecranul Bluetooth

Ecranul Bluetooth implementează funcționalitatea de scanare a dispozitivelor Bluetooth cu utilizare de energie scăzută(BLE) din apropiere. Pentru implementare se utilizează pachetul Dart "flutter_blue" care oferă funcționalități precum: verificarea disponibilității serviciului pe dispozitivul mobil, scanarea dispozitivelor, conectarea la un dispozitiv și descoperirea de servicii și caracteristici a dispozitivelor. Scheletul este unul asemanator cu cel al celorlalte ecrane principale, singura trasatura care diferă este conținutul corpului acestuia.

Corpul scheletului este reprezentat de un element de interfata grafica cu stare numit CitireBluetooth(BluetoothRead). Interfata grafica a acestui element contine doua elemente, un element buton de functionalitate si un element de tipul constructor de lista.

Butonul de funcționalitate este utilizat pentru pornirea functionalitatii de căutare a dispozitivelor și are atasat o functie asincrona care procesează comportamentul și aspectul butonului în funcție de starea acestuia.

La apelarea funcției asincrone, se obține o instanță a clasei FlutterBlue din pachetul Dart Bluetooth și se obține un obiect utilizat pentru afișarea elementului de interfața grafica de tipul incarcare din partea de jos a ecranului(BottomLoader). După instantierea claselor necesare, se verifica starea butonului.

Dacă butonul este în starea de pornire, aspectul acestuia se schimbă sa reflecte faptul ca scanarea este în desfășurare și se verifica disponibilitatea functionalitatii Bluetooth utilizând metoda obiectului FlutterBlue, esteDisponibil(isAvailable). În cazul in care functionalitatea este disponibilă se continua procesarea, dacă nu, se afișează o alertă cu un mesaj personalizat. Odată verificata disponibilitatea, se afișează elementul de încărcare, pentru a oferi utilizatorului o imagine a ce se intampla in acel moment, si se porneste scanarea intr-un mod asincron cu ajutorul metodei obiectului FlutterBlue pornesteScanarea (startScan) cu o perioada limitata de timp de 5 secunde.

După finalizarea scanării, pentru a avea access la dispozitivele scanate utilizăm obiectul clasei instantiate FlutterBlue, rezultateScanare(scanResults) care returnează un curent de rezultate la care ne putem abona utilizând metoda ascultare(listen). Metoda ascultare primește ca parametru o funcție cu declansare la eveniment unde întreaga lista de rezultate este iterata. Pentru fiecare dispozitiv din lista de rezultate, se preia numele și se adaugă într-o lista pe care o sa o utilizam la afisarea dispozitivelor.

Ca și pas final, se oprește abonarea la rezultatele scanării, se opreste scanarea și se închide elementul de încărcare din partea de jos a ecranului. De asemenea, butonul își schimba starea și aspectul în cel inițial, pentru a putea fi pregatit pentru o nouă scanare.

Dacă butonul este în starea de scanare, scanarea este în curs de desfășurare și nu exista un comportament atașat acestei stări.

Elementul de interfață constructor de lista utilizează lista populata la scanare pentru afișarea dispozitivelor după scanare. Lista este actualizată constant datorită

utilizării funcției de setare stare în interiorul scanării care reactualizeaza elementul de interfața grafica.

```
if (isEnabled) {
bool isAvailable = await fb.isAvailable;
if (isAvailable) {
  await bl.display();
  try {
    await fb.startScan(timeout: Duration(seconds: 5));
  } catch (e) {
    showAlert(context, 'e');
  }

  var subscription = fb.scanResults.listen((results) {
    for (ScanResult r in results) {
      items.add(r.device.name.toString());
      print('${r.device.name} found! rssi: ${r.rssi}');
    }
  });
  setState(() {
  });

  subscription.cancel();
  fb.stopScan();
  bl.close();
} else {
  showAlert(context, 'Not available! Check if it is enabled.');
}
}
```

Figura 4.25: Functionalitatea de scanare a dispozitivelor BLE

6. Concluzii

6.1 Introducere

In acest capitol, se vor prezenta concluziile comparației dintre cele doua ecosisteme utilizate pentru dezvoltarea aplicației prezentate în capitolele anterioare: React Native, bazat pe JavaScript, și Flutter, bazat pe Dart. De asemenea tot în acest capitol vor fi prezentate posibile dezvoltări și îmbunătățiri ulterioare.

Concluziile se vor trage prin prisma criteriilor de comparație menționate mai jos:

- Ușurință în dezvoltare
- Portabilitate
- Suport pentru backend
- Performanță
- Estetică
- Popularitate
- Acces la funcțiile native ale platformei (Android/iOS)

6.2 Criterii de comparație

6.2.1 Ușurință în dezvoltare

În ceea ce privește ușurința în dezvoltare, ambele ecosisteme oferă soluții mult mai ușor de implementat decât utilizarea limbajelor native. Deși Flutter utilizează limbajul de programare Dart, care este unul cu o popularitate mai scăzută, acesta are o sintaxa familiara cu care dezvoltatorul se obișnuiește rapid.

Un element cheie în ușurința în dezvoltare, o face documentatia pusa la dispozitie de cele doua solutii. Ambele soluții au o documentație vasta cu exemple interactive integrate în navigatorul web care duc dezvoltatorul de la stadiul de instalare la testarea conceptelor mai dificile.

Deși documentația React Native dispune de exemple la explicarea fiecărui concept, Flutter oferă posibilitatea de a crea o aplicație ghidat, în care majoritatea conceptelor fundamentale lucrează împreună pentru implementarea unei întregi aplicații.

Un alt avantaj Flutter peste React Native este existența unei galerii de aplicații dezvoltate cu Flutter. Aceasta prezintă elementele de interfață grafică cu cod demonstrativ și întregi aplicații pentru a arăta capabilitățile acestui ecosistem. Deși Flutter vine cu o documentație mult mai elegant împachetată, React Native oferă mult

mai multe detalii tehnice în documentație, care amplifica viziunea de ansamblu a dezvoltatorului.

Din punctul de vedere al opțiunilor de dezvoltare, React Native este cel avantajat punând la dispoziție două interfețe linie de comanda, pentru dezvoltatorul incepator și cel avansat. ExpoCLI poate fi utilizat atât de către dezvoltatorii începători cât și de cei avansați, datorită abstractizării în ceea ce privește configurația proiectului și aducand functionalitati utile în procesul de dezvoltare precum aplicația de testare ExpoGo.

Ca și configurare inițială a proiectului, Flutter este mult mai bine pregatit avand un instrument de verificare a mediului de dezvoltare, identificare a problemelor și oferirea de sfaturi în rezolvarea lor. Acest instrument se numeste “flutter doctor”.

React Native pune dificultăți uneori în dezvoltarea unor anumite componente, bazându-se prea mult pe bibliotecile terțe. Spre exemplu, în cadrul componentelor esențiale nu exista o componentă care să implementeze un tabel, este necesara instalarea unei biblioteci sau crearea unei componente de către utilizator.

6.2.2 Portabilitate

Ambele ecosisteme sunt utilizate ca și alternative pentru dezvoltarea nativă, datorită faptului ca se poate utiliza același cod sursa pentru dezvoltarea aplicațiilor pe cele doua platforme majoritare: Android si iOS.

Pe langa cele doua platforme ale dispozitivelor mobile, Flutter si React Native oferă suport și pe alte sisteme de operare. React Native ofera suport, prin intermediul librăriilor, pentru iOS, Android, Web si Windows 10. Flutter oferă o mai buna portabilitate deoarece același cod sursa poate oferi aplicații native pe cinci sisteme de operare: iOS, Android, macOS, Windows si Linux. Pe langa acestea, Flutter dispune de suport și pentru navigatoarele web, codul putand fi rulat și testat pe web. Expansiunea Flutter in diferite domenii nu se oprește aici, producătorul de automobile Toyota anuntand posibilitatea introducerii Flutter in sistemele de informare din acestea.

5.2.3 Suport pentru backend

Atât în cazul Flutter cat și în cazul React Native, pentru partea de backend am utilizat serviciul de backend Firebase datorită usurintei de integrare în aplicații, ambele ecosisteme oferind suport prin intermediul bibliotecilor/pachetelor, utilizarea gratuită cu spațiu de stocare limitat, opțiuni în ceea ce privește bazele de date și suport pentru autentificare sau înregistrare. Toate avantajele de mai sus pot scurta timpul de dezvoltare considerabil.

În cazul aplicațiilor mici, Firebase este o opțiune viabilă, dar în cazul aplicațiilor mari, din cauza capabilităților de interogare limitate și a tipului de stocare, JSON, poate fi un inconvenient.

În ceea ce privește suportul pentru backend, ambele ecosisteme pot suporta atat servicii de backend cât și dezvoltarea acestuia într-un limbaj la alegere iar mai apoi integrarea cu React Native și Flutter.

5.2.4 Performanța

Ambele soluții oferă performanță ridicată care ajunge aproape de nivelul dezvoltării în limbaj nativ dar în unele cazuri Flutter pare să aibă o performanță mai ridicată decât React Native deoarece codul este transformat în cod nativ înainte de rularea aplicației (ahead of time) si de asemenea, nu se utilizează podul JS pentru schimbul de informație. Podul fiind un interpretor al codul JavaScript este necesar dezvoltării aplicațiilor dar acest lucru rezultă în durată mai mare de așteptare pentru procesare. Flutter neavand nevoie de acest intermediar, rulează codul mult mai rapid se ofera tranziții mai line în cazul animațiilor. Acest lucru poate fi realizat și în React Native dar cu configurări adiționale.

Problemele de performanță în cazul React Native ar putea să afecteze aplicația și felul în care este percepută dacă se realizează computații intense pe fundalul aplicației.

5.2.5 Acces la funcțiile native

Dezvoltând o aplicație care necesită acces la perifericele dispozitivului mobil, accesul la funcțiile native este foarte important.

Deși atât React Native cât și Flutter, au acces la biblioteci/pachete care utilizează interfețe de programare a aplicației pentru implementarea functionalitatilor de nfc, Bluetooth, camera etc, în cazul React Native unele functionalitati nu sunt prezente. Spre exemplu, in cazul Bluetooth, comunicarea intre dispozitive conectate Bluetooth este dificila de implementat.

Un dezavantaj al ecosistemului React Native, este faptul ca acesta utilizează biblioteci terte pentru accesul la funcțiile native, spre deosebire de Flutter care are integrat în kitul de dezvoltare componentele necesare pentru a avea acces la majoritatea funcțiilor native.

5.2.6 Estetică

Ambele soluții de dezvoltare, dau impresia unei aplicații dezvoltate nativ. React Native face acest lucru prin utilizarea de componente esențiale care la rularea aplicației afișează compoziția potrivită pentru fiecare platforma, ceea ce rezultă în actualizari a interfeței grafice odată cu actualizarea sistemelor de operare.

Flutter utilizează propriile elemente de interfață grafică, împartită în doua seturi pentru fiecare platforma: setul Material pentru Android și setul Cupertino care imită stilul iOS. Acest lucru rezulta într-un aspect nativ al aplicației atat pe dispozitivele care rulează versiuni actualizate cât și pe cele care rulează versiuni mai vechi. Acest lucru poate fi obtinut și în React Native prin utilizarea unor librării terțe precum: React Native Paper sau React Native Elements.

Deși Flutter oferă un aspect mai similar cu cel nativ, stilul aplicației este de cele mai multe ori specific platformei ceea ce duce la necesitatea de modificare a interfețelor grafice pentru a utiliza Cupertino pe iOS în cazul dezvoltării inițiale cu Material.

Există de asemenea și posibilitatea de utilizare a celor două seturi in aceeași aplicație prin utilizarea anumitor setari. Acest aspect poate deveni o problema în cazul în care se dorește ca aplicația să aibă un aspect standard iOS, deoarece Material a fost dezvoltat pentru orice platforma nu doar Android.

Ca și concluzie, ambele ecosisteme necesită efort în direcția stilului aplicației, pentru a putea obține stilul nativ pe fiecare platforma.

5.2.7 Popularitate

Deși existând o diferenta de 3 ani între lansarea React Native(2015) si lansarea Flutter(2018), ca și popularitate pe git Flutter conduce cu o diferenta de aproximativ 20.000 de “stele” ceea ce semnifica o creștere a comunității Flutter.

React Native deține avantajul stabilității pe piață, multe din marile companii precum Airbnb sau Facebook, utilizat ecosistemul pentru dezvoltarea aplicațiilor lor. Comunitatea este una mai vastă și mai matura, care dezvolta biblioteci terte simplificand procesul de dezvoltare.

Atât vechimea pe piață, cât și adoptarea React Native de către marile companii face ca cererea de dezvoltatori React Native să fie mult mai mare ca cea de dezvoltatori Flutter, acest lucru fiind datorat în mare parte și faptului ca React Native este bazat pe JavaScript, dezvoltatorii fiind mult mai familiarizați cu acest limbaj de programare decât cu Dart. Dart castiga de asemenea popularitate, nefiind utilizat prea des fara Flutter dar are o sintaxa familiara care aminteste de Java.

La momentul actual, React Native conduce piața avand mult mai multe aplicații și descarcari în magazinele digitale și per total o mai mare bucata din piața. Dintre aplicațiile dezvoltate cu React Native putem menționa: Instagram, Netflix, Amazon. Dintre aplicațiile dezvoltate cu Flutter putem menționa: Google Ads, Google Pay, Ebay.

5.3 Dezvoltări ulterioare și îmbunătățiri

Aplicația dezvoltată fiind una utilizată ca și exemplu pentru dezvoltarea a unor funcționalități în doua ecosisteme diferite, se pot aduce îmbunătățiri considerabile. Se pot menționa următoarele îmbunătățiri și dezvoltări ulterioare:

- Securitate îmbunătățită la autentificarea utilizatorilor și păstrarea sesiunii curente după autentificare.
- Implementarea autentificarii cu amprentă pe dispozitivele care suportă acest lucru
- Configurarea stilului în așa fel încât să prezinte componente/elemente de interfață cât mai apropiate de cele native
- Implementarea funcției de scriere pe carduri NFC
- Implementarea suportului pentru diferite tehnologii NFC
- Comunicarea între doua dispozitive mobile prin Bluetooth(dificil de implementat pe React Native)
- Utilizarea serviciilor și caracteristicilor unui dispozitiv Bluetooth
- Adaugarea funcționalității de schimbare cameră în React Native și salvarea în memoria internă
- Adăugarea diferitelor moduri pentru cameră în ambele ecosisteme: bliț, focus etc
- Utilizarea senzorilor de mișcare pentru implementarea unei funcționalități precum raportarea de erori.
- Integrarea unei hărți cu ajutorul funcționalității GPS
- Eliminarea reconstruirii inutile a elementelor/componentelor de interfață
- Implementarea unei mai bune separari intre interfața grafică și logica din spate
- Implementarea suportului pentru limba engleză

Bibliografie

[1] Bates, S. (2014, January 14). *Manifesto*. Retrieved from https://manifesto.co.uk/history-mobile-application-development/

[2] Best, J. (2013, April 4). *ZDNet*. Retrieved from https://www.zdnet.com/article/android-before-android-the-long-strange-history-of-symbian-and-why-it-matters-for-nokias-future/

[3] Callaham, J. (2021, May 1). *AndroidAuthority*. Retrieved from androidauthority.com: https://www.androidauthority.com/history-android-os-name-789433/

[4] Cassavoy, L. (2007, May 7). *PCWorld*. Retrieved from pcworld.com: https://www.pcworld.com/article/131450/in_pictures_a_history_of_cell_phones.html#slide9

[5] Clark, P. J. (n.d.). *edu*. Retrieved from uky.edu: https://www.uky.edu/~jclark/mas490apps/History%20of%20Mobile%20Apps.pdf

[7] Cowley, S. (2013, January 29). *CNN*. Retrieved from money.cnn.com: https://money.cnn.com/gallery/technology/mobile/2013/01/29/smartphone-market-share/index.html

[8] Design Patterns. (n.d.). In R. H. Erich Gamma.

[9] Expo. (n.d.). Retrieved from docs.expo.io: https://docs.expo.io/introduction/managed-vs-bare/

[10] Flutter. (n.d.). Retrieved from flutter.dev: https://flutter.dev/docs/resources/architectural-overview

[11] Flutter. (n.d.). Retrieved from https://flutter.dev/docs

[12] Kaczworoski, M. (2021, April 29). *IdeaMotive*. Retrieved from ideamotive.co: https://www.ideamotive.co/blog/picking-the-best-language-for-ios-app-development

[13] Karczewski, D. (2020, August 19). *IdeaMotive*. Retrieved from ideamotive.co: https://www.ideamotive.co/blog/swift-vs-objective-c-which-should-you-pick-for-your-next-ios-mobile-app

[14] Karczewski, D. (2020, September 14). *IdeaMotive*. Retrieved from ideamotive.co: https://www.ideamotive.co/blog/flutter-app-development-everything-you-need-to-know

[15] Khorososhulia, S. (n.d.). *MobiDev*. Retrieved from mobidev.biz: https://mobidev.biz/blog/how-react-native-app-development-works

[16] Kodytechnolab. (2020, December 23). Retrieved from kodytechnolab.com: https://kodytechnolab.com/how-flutter-works

[17] MobileInfo. (n.d.). *MobileInfo*. Retrieved from mobileinfo.com: http://www.mobileinfo.com/wap/what_is.htm

[18] O'Reilly. (n.d.). Retrieved from oreilly.com: https://www.oreilly.com/library/view/learning-react-native/9781491929049/ch01.html

[19] Poulson, S. (2005, December 30). *CodeProject*. Retrieved from https://www.codeproject.com/Articles/12561/Palm-OS-3-5-Development-Part-1-Introduction-to-Pal

[20] PubNative. (2020, April 20). *PubNative*. Retrieved from https://pubnative.net/blog/from-tetris-to-candy-crush-the-history-of-mobile-gaming/

[21] Raphael, J. (2020, april 1). *ComputerWorld*. Retrieved from computerworld.com: https://www.computerworld.com/article/3306443/what-is-project-treble-android-upgrade-fix-explained.html

[22] ReactNative. (n.d.). Retrieved from https://reactnative.dev/docs/getting-started

[23] Siddiqui, A. (2020, October 10). *XDA*. Retrieved from xda-developers.com: https://www.xda-developers.com/android-project-mainline-modules-explanation/

[24] Staff, V. (2013, september 16). *TheVerge*. Retrieved from theverge.com: https://www.theverge.com/2011/12/13/2612736/ios-history-iphone-ipad

[25] The editors of Encyclopaedia. (2018, December 19). *Britannica*. Retrieved from britannica.com: https://www.britannica.com/technology/Palm-OS

[26] Wikipedia. (n.d.). Retrieved from en.wikipedia.org: https://en.wikipedia.org/wiki/Motorola_DynaTAC#cite_ref-7

[27] Wikipedia. (n.d.). Retrieved from en.wikipedia.org: https://en.wikipedia.org/wiki/Snake_(video_game_genre)

[28] WIkipedia. (n.d.). Retrieved from en.wikipedia.org: https://en.wikipedia.org/wiki/Handspring_(company)

[29] Wikipedia. (n.d.). *Wikipedia*. Retrieved from en.wikipedia.org: https://en.wikipedia.org/wiki/Palm_OS

[30]Wikipedia. (n.d.). *Wikipedia*. Retrieved from en.wikipedia.org: https://en.wikipedia.org/wiki/Symbian

[31] Wikipedia. (n.d.). *Wikipedia*. Retrieved from en.wikipedia.org: https://en.wikipedia.org/wiki/Android_software_development