TEMA 3

EXTINDEREA APLICAȚIEI CALCULATOR ASTFEL ÎNCÂT SĂ STOCHEZE DATELE INTERN ȘI SĂ TRIMITĂ/PRIMEASCĂ INFORMAȚII ÎN/DIN INTERNET

STUDENT: Vasile Andrei – Daniel

SPECIALIZAREA: Informatică

ANUL DE STUDII: II

GRUPA: 40322

DISCIPLINA: Dezvoltarea Aplicațiilor Mobile

TITULAR DISCIPLINĂ: conf. dr. ing. Zoran Constantinescu

Cuprins

STRUCTURA APLICAȚIEI **2**

NAVIGAREA ÎNTRE FRAGMENTE **2**

FRAGMENTUL CALCUL **4**

Stocarea datelor în SQLite **5**

Trimiterea și primirea datelor din Internet **8**

FRAGMENTUL ISTORIC **10**

FRAGMENTUL EMAIL **12**

FRAGMENTUL PENTRU LOG-URI **17**

ÎNCĂRCAREA DATELOR DIN BD LA PORNIREA APLICAȚIEI **19**

SALVAREA ȘI RESTAURAREA DATELOR **19**

STRUCTURA APLICAȚIEI

Aplicația are o singură activitate, în care am creat patru fragmente, și anume:

- Calcul
- Istoric
- Email
- Log

Practic, am modificat tema anterioară astfel: am mutat ce aveam în activitatea principală în fragmentul Calcul, respectiv ce aveam în activitatea secundară în fragmentul Istoric. Pe lângă acestea am mai adăugat posibilitatea de a trimite prin email istoricul specificând direct în aplicație adresa destinatarului și subiectul mesajului (în fragmentul Email) și vizualizarea unor log-uri de rețea în fragmentul Log.

NAVIGAREA ÎNTRE FRAGMENTE

Navigarea între fragmente am implementat-o folosind un TabLayout și o componentă numită ViewPager2. TabLayout-ul este pentru tab-urile vizibile (apare ca o bară de meniu în partea de sus a fiecărui fragment), iar ViewPager2 pentru găzduirea fragmentelor, permițând totodată navigarea între ele prin swipe sau prin selectarea acestora din TabLayout-ul de sus. Aceste două componente au fost adăugate în fișierul XML al activității principale:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <com.google.android.material.tabs.TabLayout
        android:id="@+id/tab_layout"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        app:tabMode="fixed"
        app:tabGravity="fill"
        android:contentDescription="descriere" />

        <androidx.viewpager2.widget.ViewPager2
            android:id="@+id/view_pager"
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1" />
</LinearLayout>
```

În **MainActivity.kt** am creat un **ViewPagerAdapter** care furnizează fragmentele și am conectat TabLayout și ViewPager2 printr-un **TabLayoutMediator** (o clasă din biblioteca Material Components) care se ocupă automat de actualizarea sincronizată între swipe-ul din ViewPager2 și selectarea manuală a tab-urilor din bara de navigare.

```
        val tabLayout = findViewById<TabLayout>(R.id.tab_layout)
        val viewPager = findViewById<ViewPager2>(R.id.view_pager)

        //setam adapter-ul care gestioneaza fragmentele
        viewPager.adapter = ViewPagerAdapter( activity: this)

        //asociem tab-urile cu ViewPager-ul
        val tabTitles = listOf("Calcul", "Istoric", "Email", "Log")
        TabLayoutMediator(tabLayout, viewPager) { tab, position ->
            tab.text = tabTitles[position] //seteaza textul fiecarui tab în functie de pozitie
        }.attach() //porneste legatura dintre cele doua
    }
```

Am mai adăugat un fișier Kotlin separat unde am suprascris metodele **getItemCount()** și **createFragment()**. Când utilizatorul schimbă tab-ul (prin swipe sau apăsare), ViewPager2 apelează **createFragment(position)** cu indexul acelui tab iar funcția returnează fragmentul corespunzător. Codul poate fi observat mai jos:

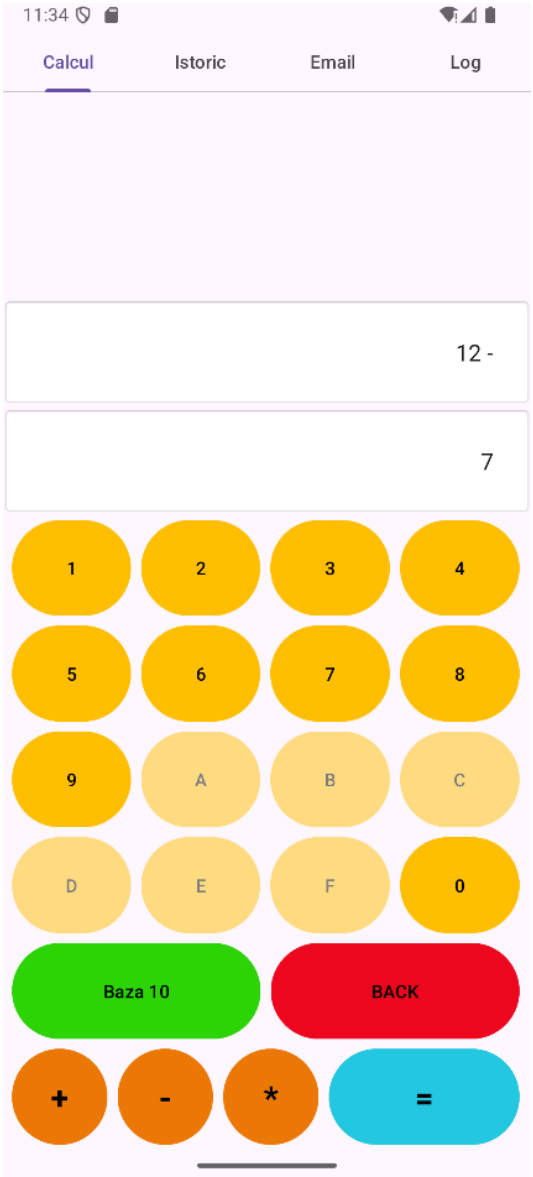
```
class ViewPagerAdapter (activity: FragmentActivity) : FragmentStateAdapter(activity){

    override fun getItemCount(): Int = 4 //sunt 4 tab-uri
    override fun createFragment(position: Int): Fragment {
        return when (position) {
            0 -> Calcul()
            1 -> Istoric()
            2 -> Email()
            3 -> Log()
            else -> throw IllegalArgumentException("Invalid position $position")
        }
    }
}
```

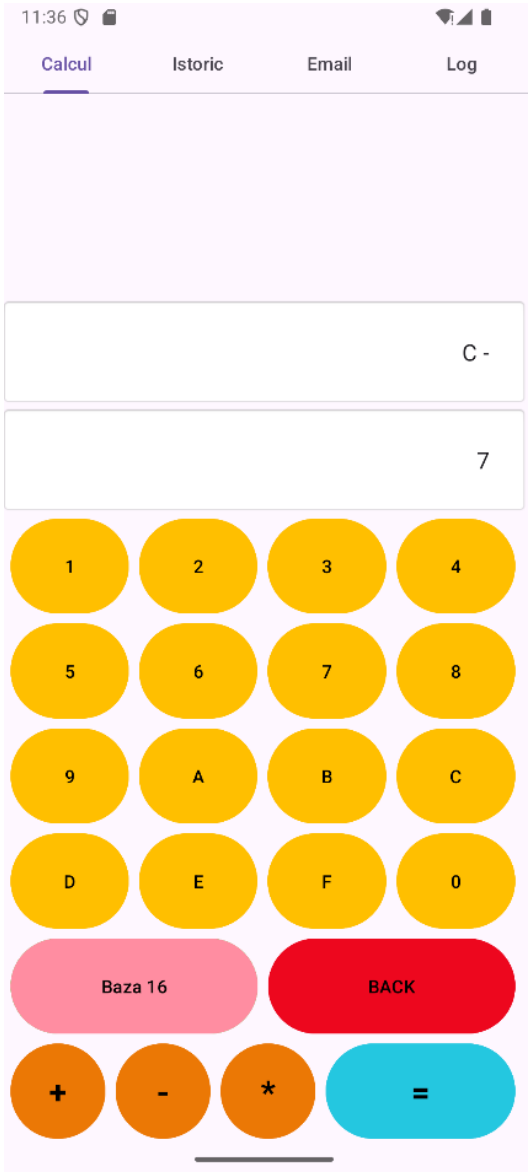
Și pentru ca să pot să trec la alt fragment și din cod (de exemplu, a trebuit să implementez ca atunci când utilizatorul selectează un element din istoric, acesta să fie mutat automat în tab-ul calcul), am mai adăugat în MainActivity.kt o funcție **goToTab()** care permite schimbarea programatica (adică din cod) a fragmentului activ, funcție care poate fi observata mai jos.

```
//declaram o functie pentru a naviga intre taburi din alte parti ale aplicatiei
fun goToTab(index: Int) {
    val viewPager = findViewById<ViewPager2>(R.id.view_pager)
    viewPager.currentItem = index
}
```

FRAGMENTUL CALCUL

În acest fragment este implementată funcționalitatea calculatorului. Acesta are două layout-uri (două fișiere XML), al doilea fiind, bineînțeles, pentru orientarea tip „peisaj” (landscape). Cele doua fișiere XML nu au fost modificate foarte mult față de tema precedentă, doar a fost eliminat din ele componentele **AppBarLayout** și **MaterialToolbar** deoarece acum navigarea se face diferit, așa cum am explicat anterior.





Pe partea de programare voi prezenta în continuare doar ce am adăugat la această temă și nu voi relua ce era deja implementat de la tema trecută.

Stocarea datelor în SQLite

Prin urmare, a trebuit să implementez următoarea funcționalitate: de fiecare dată când adăugam ceva în lista de istoric să se adauge acel element și în baza de date. Pentru aceasta am creat o clasă numită **CalculatorDatabaseHelper**, clasă care este un **helper** pentru baza de date SQLite. Practic ea extinde „**SQLiteOpenHelper**” care ne ajută să creăm și să actualizăm automat baza de date. În această clasă am suprascris metoda „**onCreate()**”, metodă care este apelată automat, o singură dată, atunci când baza de date este creată pentru prima dată. În ea am creat cele 3 tabele: **calcul**, **istoric** și **emailuri**.

```
override fun onCreate(db: SQLiteDatabase) {
    //creem tabela culcul si pun """ pentru ca sa pot scrie un string pe mai multe linii
    db.execSQL ("""
        CREATE TABLE calcul (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        operatie TEXT,
        operand1 TEXT,
        operand2 TEXT,
        bazanumeratie TEXT,
        rezultat TEXT,
        dataora TEXT
        )
        """.trimIndent()) //trimIndent() doar pentru formatarea frumoasa a codului, curata indentarile inutile din string

    //creem tabela pentru istoric
    db.execSQL("""
        CREATE TABLE istoric (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        linie TEXT,
        culoare TEXT
        )
        """.trimIndent())
```

```
    //creem tabela pentru email-uri
    db.execSQL("""
        CREATE TABLE emailuri (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        adresa_email TEXT)
        """.trimIndent())
}
```

De asemenea, în această clasă am adăugat o funcție **get_all_emails()** care deschide baza de date în mod readable, citește toate înregistrările din tabela emailuri și le returnează ca listă de String.

```
fun get_all_emails(): List<String> {
    val db = this.readableDatabase
    val list = mutableListOf<String>()
    val cursor = db.rawQuery( sql: "SELECT adresa_email FROM emailuri",  selectionArgs: null)

    if(cursor.moveToFirst())
    {
        do {
            val email = cursor.getString(cursor.getColumnIndexOrThrow( columnName: "adresa_email"))
            list.add(email)
        } while (cursor.moveToNext())
    }

    cursor.close()
    return list
}
```

Și am suprascris și metoda **onUpgrade()**, metodă care se apelează automat dacă versiunea bazei de date crește. Metoda șterge toate tabelele existente și apelează din nou metoda **onCreate()** pentru a le recrea.

```
override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
    db.execSQL( sql: "DROP TABLE IF EXISTS calcul")
    db.execSQL( sql: "DROP TABLE IF EXISTS istoric")
    db.execSQL( sql: "DROP TABLE IF EXISTS emailuri")
    onCreate(db)
}
```

Revenind la fragmentul calcul, de fiecare dată când adaug ceva în lista operații din obiectul History, acum îl adaug și în tabela istoric. Tabelei i-am mai adăugat eu un câmp în plus, pentru a memora și culoarea cu care trebuie să apară acel element în istoric atunci când aplicația va porni (verde pentru baza 10 si roz pentru baza 16). În imaginea următoare poate fi observat un exemplu:

```
if(History.isBase10)
{
    val sb = SpannableStringBuilder( text: "Select baza 10")
    sb.setSpan(ForegroundColorSpan(Color.parseColor( colorString: "#2cd406")),  start: 0, "Select baza 10".length, Spannable.SPAN

    History.operatii.add(sb)
    //adaugam si in baza de date
    val values = ContentValues().apply {
        put("linie", "Select baza 10")
        put("culoare", "#2cd406")
    }
    db?.insert( table: "istoric",  nullColumnHack: null, values)
}
else
{
    val sb = SpannableStringBuilder( text: "Select baza 16")
    sb.setSpan(ForegroundColorSpan(Color.parseColor( colorString: "#FF8DA1")),  start: 0, "Select baza 16".length, Spannable.SPAN

    History.operatii.add(sb)
    //adaugam si in baza de date
    val values = ContentValues().apply {
        put("linie", "Select baza 16")
        put("culoare", "#FF8DA1")
    }
    db?.insert( table: "istoric",  nullColumnHack: null, values)
}
```

Apărea o mică problema: atunci când introduceam un număr și un operator, elementul respectiv era salvat atât în lista operații cât și în tabela istoric din baza de date. Ulterior, daca eu alegeam alt operator, trebuia să modific ultimul element atât din lista menționată cât și din tabelă. Pentru aceasta am definit o funcție care întoarce ID-ul ultimei înregistrări din tabela istoric:

```
//definim o functie pentru a afla id-ul ultimului element selectat in tabela istoric din bd
//functia primeste ca parametru un obiect de tip SQLiteDatabase
//returneaza un int (reprezentand id-ul) sau null, daca tabela este goala
private fun getLastInsertedId(db: SQLiteDatabase): Int? {
    //cursor este un obiect care contine rezultatul interogarii
    val cursor = db.rawQuery( sql: "SELECT id FROM istoric ORDER BY id DESC LIMIT 1",  selectionArgs: null)
    var lastId: Int? = null
    if(cursor.moveToFirst()) { //verificam daca cursor are cel putin un rand si il mutam pe primul rand
        lastId = cursor.getInt(cursor.getColumnIndexOrThrow( columnName: "id"))//obtine valoarea din coloana id a primului rand gasit
    }
    cursor.close()
    return lastId //returnam ultimul id gasit sau null daca tabela este goala
}
```

Și am folosit această funcție pentru a insera noua valoare la înregistrarea cu ID-ul respectiv:

```
//si inlocuim si in tabela istoric din bd:
val lastId = db?.let { getLastInsertedId(it)  }//apelam functia pe care am definit-o mai jos si care ne introarce id-ul
if(lastId!=null) {
    val newLine = "$operandAnterior $operatieCurenta"
    val culoare = if(History.isBase10) "#2cd406" else "#FF8DA1"

    val updatedValues = ContentValues().apply {
        put("linie", newLine)
        put("culoare", culoare)
    }
    db?.update( table: "istoric", updatedValues,  whereClause: "id = ?", arrayOf(lastId.toString()))
}
```

La apăsarea butonului egal (=), se adaugă în tabela calcul datele: operație, operand1, operand2, baza, rezultatul, ora și data.

```
//-------------------------------------------------------------------------------------
//adaugam calculul si in tabela calcul a bazei de date


val values = ContentValues().apply {
    put("operatie", operatieCurenta)
    put("operand1", numarAnterior)
    put("operand2", operand2)
    put("bazanumeratie", if(History.isBase10) "10" else "16")
    put("rezultat", rezultatText)
    put("dataora", SimpleDateFormat( pattern: "yyyy-MM-dd HH:mm:ss", Locale.getDefault()).format(Date()))
}

//inseram calculul in tabela
db?.insert( table: "calcul",  nullColumnHack: null, values)
//-------------------------------------------------------------------------------------
```

La distrugerea vederii fragmentului (în **onDestroyView**) am închis baza de date pentru a evita scurgerile de memorie și pentru a ne asigura că nu rămâne deschisă inutil.

```
//inchidem baza de date la distrugerea fragmentului
override fun onDestroyView() {
    super.onDestroyView()
    db?.let {
        if(it.isOpen) {
            it.close()
        }
    }
}
```

Trimiterea și primirea datelor din Internet

În listener-ul pus pe butonul egal, am implementat ca să se trimită datele către server printr-o **cerere HTTPS GET** folosind biblioteca **Volley**, bibliotecă pe care am adăugat-o în fișierul ”**build.gradle.kts**” prin linia de cod **implementation ("com.android.volley:volley:1.2.1")**. Se construiește un mesaj care conține utilizatorul, baza de numerație, operandul anterior, operatorul și operandul curent, mesaj ce va fi codificat pentru a putea fi transmis prin URL. Se creează cererea HTTPS cu Volley și s-au definit două posibilități: succes, dacă serverul răspunde corect și error, dacă apare vreo eroare (fără internet, server picat, etc). Dacă serverul răspunde, preluăm răspunsul și, după ce îl verificăm, îl inserăm în câmpul text curent. În ambele situații adăugăm și log-urile corespunzătoare în fișier.

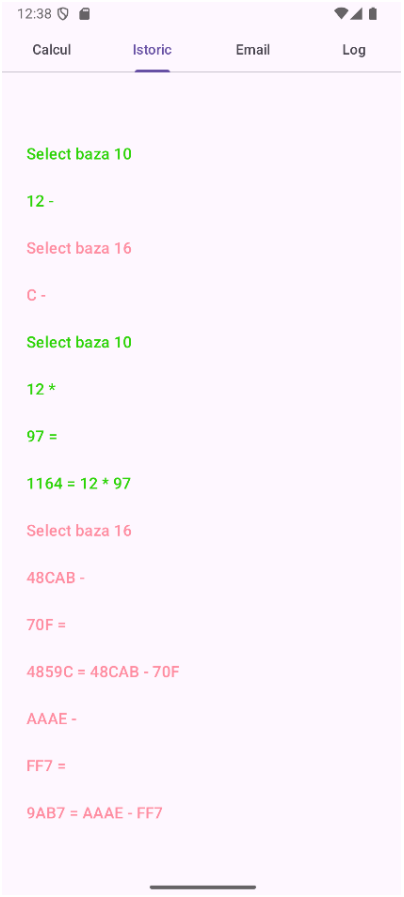
```
//URMEAZA IMPLEMENTAREA INTERACTIUNII CU SERVERUL
//================================================================================================
val user = "user49"
val bazaText = if(History.isBase10) "dec" else "hex"
val mesajTrimis = "$user $bazaText $numarAnterior $copieOperatieCurenta $operand2"
val urlEncoded = Uri.encode(mesajTrimis) //codificam textul intr-un format sigur pentru a fi transmis intr-un URL
val url = "https://timf.upg-ploiesti.ro/idp/z/zdemo9/calc?q=$urlEncoded"

//Adaugam in Log incercarea de conectare
scrieInLog( mesaj: "Connect server: attempting.")

val coadaRaspunsuri = Volley.newRequestQueue(context)
val stringRequest = StringRequest(
    Request.Method.GET, url,
    { response ->

        //Loguri
        scrieInLog( mesaj: "Connect server: succes.")
        scrieInLog( mesaj: "Operation send: $bazaText $numarAnterior $copieOperatieCurenta $operand2.")

        //extragem valoarea rezultatului
        val parti = response.trim().split(Regex( pattern: "\\s+")) //imparte sirul la unul sau mai multe spatii, taburi sau newline-uri
        if (parti.size >= 3 && parti[0] == "result") {
            val rezultatServer = parti[2]
            text_curent.setText("$rezultatServer".uppercase()) //afisam rezultatul primit in campul text_curent
```

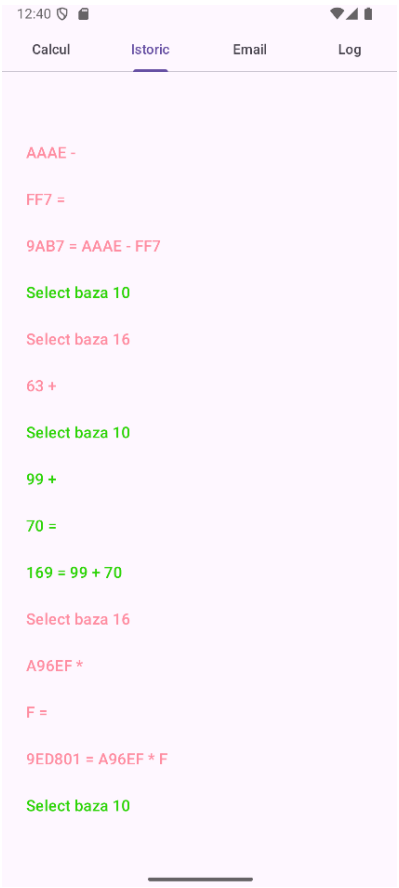
```
            //scriem si logurile
            scrieInLog( mesaj: "Result received: $response")
            val rezultatLocalCuEgal = text_anterior.text.toString()
            val rezultatLocal = rezultatLocalCuEgal.split( ...delimiters: " ")[1]
            if(rezultatServer.trim().uppercase() == rezultatLocal.trim())
                scrieInLog( mesaj: "Result correct.")
            else
                scrieInLog( mesaj: "Result wrong.")
        }
        else
        {
            Toast.makeText(requireContext(), text: "Serverul nu raspunde bine.", Toast.LENGTH_SHORT).show()
        }

        //Log
        scrieInLog( mesaj: "Disconnect.")
    },
    {
        error ->
        scrieInLog( mesaj: "Connect server: failed - ${error.message}.")
        scrieInLog( mesaj: "Disconnect.")
    }
)

coadaRaspunsuri.add(stringRequest) //adaugam cererea HTTPS in coada de executie, solicitand bibliotecii Volley sa o trimita catre server

//================================================================================================
}
```

Pentru a scrie log-urile în fișier s-a definit o funcție numită **scrieInLog()** care primește ca și parametru text-ul log-ului:

```
//DEFINIM O FUNCTIE CARE PRIMESTE CA SI ARGUMENT LOG-UL SI IL SALVEAZA IN FISIER
private fun scrieInLog(mesaj: String) {
    val timestamp = SimpleDateFormat(
        pattern: "yyyyMMdd HH:mm:ss",
        Locale.getDefault()
    ).format(Date()) //preluam data si ora curenta
    val logFinal = "$timestamp $mesaj\n" //formam mesajul final cu newline la sfarsit

    try {
        val file = File(context?.filesDir, child: "log.txt") //se creează (sau se acceseaza daca deja este creat) fisierul „log.txt"
        file.appendText(logFinal) //adaugam mesajul la sfarsitul fisierului
    } catch (e: IOException) {
        //gestionam erorile (spatiu insuficient, fisier blocat, ...)
        Log.e( tag: "LogFragment", msg: "Eroare la scriere log: ${e.message}", e)
    }
}
```

FRAGMENTUL ISTORIC






```
//preluam elementul selectat din istoric alaturi de baza in care este numarul din el
//practic ascultam pentru rezultatul trimis de fragmentul istoric
parentFragmentManager.setFragmentResultListener( requestKey: "requestKey", viewLifecycleOwner) { requestKey, bundle ->
    //preluam si procesam datele
    elementSelectat = bundle.getString( key: "element_selectat")
    bazaElementului = bundle.getInt( key: "baza",  defaultValue: 10)

    elementSelectat?.let {
        val tablouElemente = it.trim().split( ...delimiters: " ")
        if (tablouElemente.size >= 1) {
            val valoare = when {
                History.isBase10 && bazaElementului == 16 -> tablouElemente[0].toIntOrNull( radix: 16)?.toString()
                History.isBase10 && bazaElementului == 10 -> tablouElemente[0] //elementul ramane neschimbat
                !History.isBase10 && bazaElementului == 16 -> tablouElemente[0] //elementul ramane neschimbat
                !History.isBase10 && bazaElementului == 10 -> {
                    val numarInt = tablouElemente[0].toIntOrNull( radix: 10)
                    numarInt?.let { Integer.toString(it,  radix: 16).uppercase() }
                }
                else -> null
            }

            valoare?.let { text_curent.setText(it) } //punem valoarea in campul text curent
        }
    }
}
```

În acest fragment una dintre modificările făcute a fost schimbarea modului în care este trimis elementul selectat alături de baza sa către fragmentul Calcul. Ideea de bază a rămas tot aceeași (folosind un bundle) dar codul s-a schimbat puțin, nemaifiind vorba despre trimiterea datelor între doua activități (acum avem două fragmente). Prin urmare, am definit o funcție în fragmentul Calcul (se poate observa în imaginea de mai sus) care ascultă un rezultat cu cheia „requestKey", rezultat care este trimis din fragmentul Istoric. Funcția preia cele două elemente din bundle (**element_selectat** și **baza**) și, în funcție de baza elementului selectat și de baza curentă a aplicației, va converti (dacă este nevoie) elementul adus din istoric în baza calculatorului și îl va introduce în câmpul text curent al fragmentului Calcul.

Iar ca să trimitem datele către această funcție, le-am pus într-un bundle și le-am asociat aceleași chei. Ulterior trimiterii datelor, am redirecționat utilizatorul către fragmentul Calcul, pentru a putea continua acolo activitatea cu elementul adus din istoric.

```
val bundle = Bundle()
bundle.putString("element_selectat", elementSelectat)
bundle.putInt("baza", baza)

// Trimitem rezultatul la fragmentul Calcul
parentFragmentManager.setFragmentResult( requestKey: "requestKey", bundle)

//navigam la fragmentul calcul
(activity as? MainActivity)?.goToTab( index: 0)
```

Am mai observat ceva. După deschiderea fragmentului istoric (dar si a fragmentelor Email și Log), dacă efectuam calcule și mă întorceam din nou să le văd în istoric, acestea nu apăreau...asta pentru că funcția **onViewCreated()** unde eu afișam conținutul listei operații (lista din memoria RAM unde în timpul rulării aplicației am permanent încărcat istoricul) nu se apelează decât odată, la deschiderea fragmentului. După aceea am înțeles că trebuie să afișez istoricul și în funcția **onResume()**, funcție care se apelează de fiecare dată când revin într-un fragment care, după ce am plecat din el a rămas activ, doar că era ascuns (de exemplu, în ViewPager2 sau când se navighează între tab-uri). Practic, atunci când mă mut într-un alt fragment în ViewPager2, fragmentele nevizibile (inactive) nu sunt distruse, ci doar ascunse, interfața lor rămânând în memorie. Așa că în funcția **onResume()** am creeat din nou adapter-ul, am suprascris **getView()**, am atașat adapter-ul la ListView și am selectat ultimul element din listă, pentru a vedea ultimele operații efectuate. În felul acesta utilizatorul se poate duce de câte ori dorește din fragmentul Istoric în fragmentul Calcul ca să facă alte operații. Când va reveni, istoricul afișat în **ListView** se va actualiza automat, derulându-se mereu la ultimul element introdus și făcând vizibile ultimele operații efectuate. Suprascrierea metodei **onResume()** poate fi observată mai jos.

```kotlin
override fun onResume() {
    super.onResume()

    //facem un adapter personalizat pentru listview
    val adapter = object : ArrayAdapter<CharSequence>(
        requireContext(),
        android.R.layout.simple_list_item_1,
        History.operatii
    ) {
        override fun getView(position: Int, convertView: View?, parent: ViewGroup): View {
            val view = super.getView(position, convertView, parent)
            (view as TextView).text = getItem(position)
            return  view
        }
    }

    listView.adapter = adapter
    //selectam ultimul element din lista pentru a vedea cele mai recente operatii
    listView.setSelection(adapter.count - 1)

}
```

FRAGMENTUL EMAIL

În fișierul XML al acestui fragment am folosit două componente **TextView** pentru a afișa textele „Email”, respectiv „Subiect” înaintea câmpurilor de introducere a adresei de email, respectiv a subiectului. Cu ajutorul unei componente numită **AutoCompleteTextView** am implementat câmpul care trebuie să se autocompleteze atunci când utilizatorul alege o adresă de email din lista adreselor sugerate. Pentru introducerea subiectului am folosit EditText.

```xml
<TextView
    android:text="Email:"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textStyle="bold" />

<AutoCompleteTextView
    android:id="@+id/emailAdress"
    android:layout_width="match_parent"
    android:layout_height="50dp"
    android:hint="Adresă email"
    android:inputType="textEmailAddress"
    android:completionThreshold="1"/>
```

```
<TextView
    android:text="Subiect:"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textStyle="bold" />

<EditText
    android:id="@+id/emailSubject"
    android:layout_width="match_parent"
    android:layout_height="50dp"
    android:layout_marginBottom="10dp"
    android:hint="Subiectul email-ului" />
```

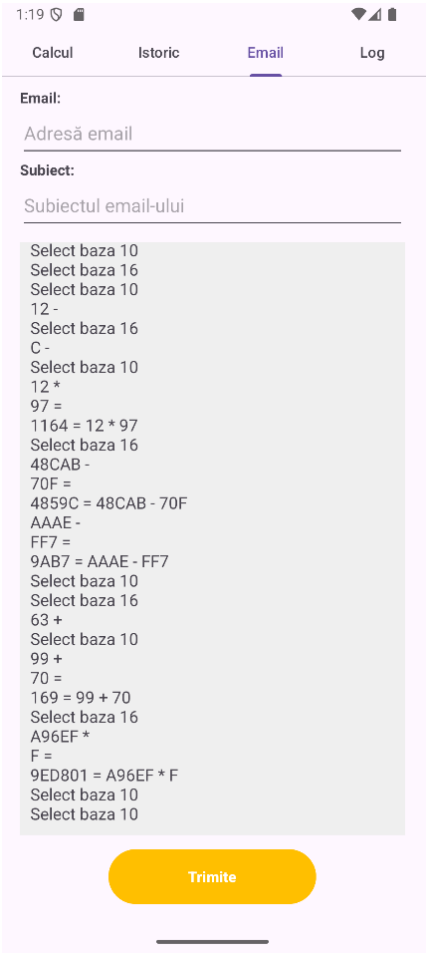
Pentru a afișa mesajul email-ului am folosit un **TextView** plasat într-un ScrollView pentru a putea face scroll mai ușor și mai sigur.

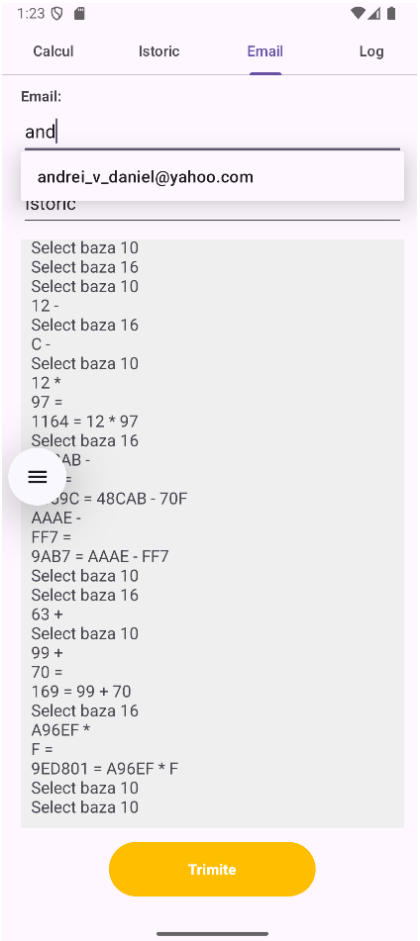
```
<ScrollView
    android:id="@+id/emailScrollView"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    android:layout_weight="1"
    android:background="#EFEFEF"
    android:fillViewport="true">

    <TextView
        android:id="@+id/emailBody"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="10dp"
        android:textSize="16sp" />

</ScrollView>
```

De asemenea, am mai adăugat și o componentă Button centrată orizontal pentru a face butonul de trimitere a email-ului:

```
<Button
    android:id="@+id/sendButton"
    android:layout_width="200dp"
    android:layout_height="60dp"
    android:layout_marginTop="10dp"
    android:text="Trimite"
    android:textStyle="bold"
    android:backgroundTint="#FFBF00"
    android:layout_gravity="center_horizontal"/>
```

Toate componentele de mai sus au fost plasate într-un **LinearLayout** orientat vertical. Rezultatul obținut poate fi observat în imaginea de mai jos.





În fișierul Email.kt, în funcția **onViewCreated()** am instanțiat baza de date și am efectuat un SELECT pe tabela emailuri pentru a pune toate înregistrările de acolo în lista de sugestii. Ulterior am creeat un adapter (pentru a prelua datele și a le transforma în View-uri) pe care l-am conectat la **AutoCompleteTextView** (prin emailField) și am apelat funcția **afiseazaIstoric()**.

```
//instantiem baza de date
val dbHelper = CalculatorDatabaseHelper(requireContext())
val db = dbHelper.writableDatabase //obtinem o referinta la baza de date in modul de scriere

//citim toate email-urile din tabela si le punem in lista de sugestii
val cursor = db.rawQuery( sql: "SELECT adresa_email FROM emailuri",  selectionArgs: null)
cursor.use {
    while (it.moveToNext()) { //cat timp mai avem randuri in tabela
        emailuriList.add(it.getString( columnIndex: 0))
    }
}
```

```
//setam adapterul pentru AutoCompleteTextView
emailAdapter = ArrayAdapter(requireContext(),android.R.layout.simple_list_item_1, emailuriList)
emailField.setAdapter(emailAdapter)

afiseazaIstoric() //afisam istoricul
```

Funcția **afiseazaIstoric()** preia într-o constantă String lista „operatii” a obiectului History, separând fiecare element cu caracterul newline. Ulterior setează textul componentei TextView ce este destinată să conțină mesajul email-ului la valoarea acestei constante String și derulează în jos conținutul din ScrollView pentru a fi vizibilă ultima parte a textului.

```
private fun afiseazaIstoric() {
    val continutIstoric = History.operatii.joinToString(separator = "\n")

    if(bodyTextView != null && scrollView !=null) {
        bodyTextView?.text = continutIstoric

        scrollView?.post {
            scrollView?.fullScroll(View.FOCUS_DOWN)
        }
    }
}
```

Pentru butonul „Trimite” am implementat un listener care ascultă evenimentul click. În această funcție am verificat mai întâi dacă avem unul dintre cele 3 câmpuri goale (adresa de email, subiectul si corpul email-ului). Dacă da, afișăm mesaje corespunzătoare și întrerupem execuția funcției.

```
sendButton.setOnClickListener {
    val email = emailField.text.toString().trim()
    val subject = subjectField.text.toString().trim()
    val mesaj = bodyTextView?.text.toString().trim()

    //verificam daca email-ul este gol sau invalid
    if(email.isEmpty() || !android.util.Patterns.EMAIL_ADDRESS.matcher(email).matches()) {
        Toast.makeText(requireContext(), text: "Te rugăm să introduci o adresa de email validă.", Toast.LENGTH_SHORT).show()
        return@setOnClickListener
    }

    //verificam daca subiectul este gol
    if(subject.isEmpty()) {
        Toast.makeText(requireContext(), text: "Subiectul nu poate fi gol.", Toast.LENGTH_SHORT).show()
        return@setOnClickListener
    }

    //verificam daca istoricul este gol pentru precautie, desi acest lucru nu ar trebui sa se intample niciodata
    if(mesaj.isEmpty()) {
        Toast.makeText(requireContext(), text: "Istoricul este gol, nu aveti ce să trimiteti.", Toast.LENGTH_SHORT).show()
        return@setOnClickListener
    }
```

Ulterior am efectuat o instrucțiune SELECT pe tabela emailuri din baza de date pentru a căuta daca avem deja înregistrată adresa respectivă. Dacă nu a mai fost înregistrată o adăugăm printr-o instrucțiune INSERT, afișăm într-un Toast un mesaj de confirmare și actualizăm lista de sugestii, setând pentru ea un nou adapter care a fost creat pentru noua listă (**updateEmailList**, listă care se obține apelând funcția **get_all_emails()** din clasa CalculatorDatebaseHelper, funcție a cărei implementare poate fi observată într-un capitol anterior). Asta deoarece ne dorim ca să putem vedea în lista de sugestii adresa de email imediat ce ne întoarcem din aplicația prin care am trimis email-ul, nu la următoarea pornire a aplicației noastre.

```
//salvam adresa de email daca nu exista deja
val emailExistCursor = db.rawQuery( sql: "SELECT COUNT(*) FROM emailuri WHERE adresa_email = ?", arrayOf(email))
var exists = false
emailExistCursor.use {
    if(it.moveToFirst()) {
        exists = it.getInt( columnIndex: 0) > 0
    }
}

//daca nu exista o adaugam
if(!exists) {
    db.execSQL( sql: "INSERT INTO emailuri(adresa_email) VALUES(?)", arrayOf(email))
    val toast = Toast.makeText(requireContext(), text: "Email salvat cu succes!", Toast.LENGTH_SHORT)
    toast.setGravity(Gravity.CENTER_VERTICAL, xOffset: 0, yOffset: 0) //setam sa apara la centrul ecranului
    toast.show()

    //actualizam si lista de sugestii
    val updateEmailList = dbHelper.get_all_emails()
    val updatedAdapter = ArrayAdapter(requireContext(), android.R.layout.simple_list_item_1, updateEmailList)
    emailField.setAdapter(updatedAdapter)

}
```

În final vom crea un URL special cu schema „mailto: si adresa de email” și un Intent cu acțiunea ACTION_SENDTO (care este specifică pentru aplicațiile de email) în care trimitem subiectul și corpul mesajului. Dacă nu va exista nici o aplicație de email pe telefon se va afișa un mesaj de eroare.

```
    //trimitem email-ul prin aplicatii de tip email
    val uri = Uri.parse( uriString: "mailto:$email") //facem un URL unde punem adresa de email a destinatarului
    val intent = Intent(Intent.ACTION_SENDTO, uri).apply {
        putExtra(Intent.EXTRA_SUBJECT, subject)
        putExtra(Intent.EXTRA_TEXT, mesaj)
    }

    try {
        startActivity(Intent.createChooser(intent, title: "Trimite email..."))
    } catch (ex: android.content.ActivityNotFoundException) {
        Toast.makeText(requireContext(), text: "Nicio aplicatie de email găsită.", Toast.LENGTH_SHORT).show()
    }
}
```

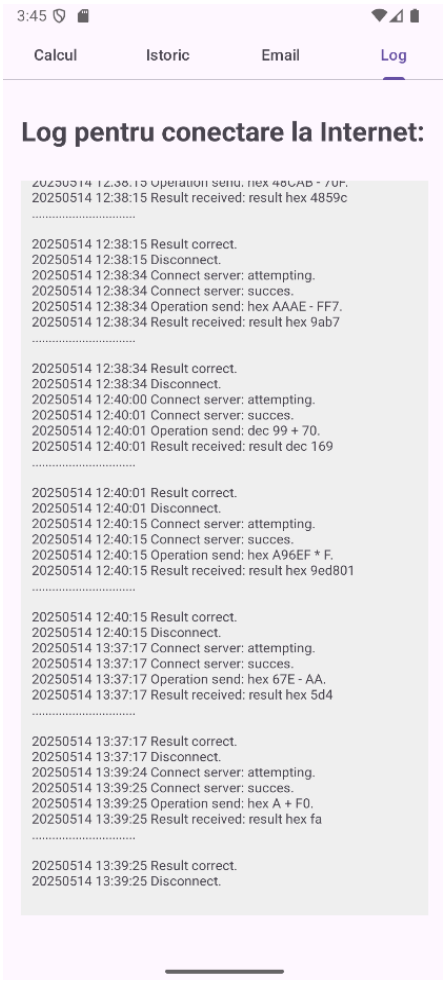
Pentru ca textul mesajului să se actualizeze atunci când se revine în fragmentul Email după ce s-au mai efectuat calcule noi de la deschiderea acestuia, am suprascris funcția **onResume()**, apelând și de aici metoda **afiseazaIstoric()**.

```
override fun onResume() {
    super.onResume()
    afiseazaIstoric()
}
```

FRAGMENTUL PENTRU LOG-URI

Pentru layout-ul acestui fragment am folosit un **LinearLayout** orientat vertical în care am plasat un **TextView** pentru titlul fragmentului (titlul fiind „Log pentru conectare la Internet” și o componentă **ScrollView**, în care din nou am pus un **TextView** pentru afișarea conținutului din fișierul „log.txt”. Componentele precizate pot fi observate mai jos alături de interfața obținută.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/logLayout"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:paddingHorizontal="20dp"
    android:paddingTop="10dp"
    android:paddingBottom="40dp">

    <TextView
        android:text="Log pentru conectare la Internet:"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textStyle="bold"
        android:textSize="25dp"
        android:layout_marginTop="20dp"
        android:layout_marginBottom="30dp" />

    <ScrollView
        android:id="@+id/logScrollView"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:background="#EFEFEF"
        android:fillViewport="true">

        <TextView
            android:id="@+id/logTextView"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:padding="10dp"
            android:textSize="12sp" />

    </ScrollView>
</LinearLayout>
```



În fișierul „**Log.kt**” am definit o funcție care actualizează fișierul de log, fișier numit „log.txt”. Funcția creează o referință către acest fișier aflat în spațiul privat al aplicației (**fileDir**). Dacă fișierul există se citește tot conținutul său cu metoda **readText()** și se afișează în **TextView**. Dacă nu există se afișează un mesaj informativ. După aceste instrucțiuni funcția derulează până la partea de jos a componentei ScrollView (implicit a componentei TextView, care este imbricată în ScrollView), pentru a arăta ultimele înregistrări din log.

```
private fun actualizeazaLog() {
    val file = File(requireContext().filesDir,  child: "log.txt")

    if(file.exists()) {
        val continut = file.readText()
        textView?.text = continut
    }
    else {
        textView?.text = "Încă nu există log-uri."
    }

    //navigam la partea de jos a textului cu log-ul, pentru a vedea ultimele log-uri
    scrollView?.post {
        scrollView?.fullScroll(View.FOCUS_DOWN)
    }
}
```

În funcțiile **onViewCreated()** și **onResume()**, tot ce am făcut a fost să apelez metoda **actualizeazaLog()**, după cum se poate observa și din imaginea următoare.

```
override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    super.onViewCreated(view, savedInstanceState)

    textView = view.findViewById(R.id.logTextView)
    scrollView = view.findViewById(R.id.logScrollView)
    actualizeazaLog()
}

override fun onResume() {
    super.onResume()
    actualizeazaLog()
}
```

ÎNCĂRCAREA DATELOR DIN BD LA PORNIREA APLICAȚIEI

În MainActivity.kt am definit o funcție care citește toate înregistrările din tabela istoric a bazei de date și le încarcă în lista operații declarată în obiectul History. Funcția execută o interogare pe tabela istoric, verifică dacă există cel puțin un rând, dacă da extrage pentru fiecare rând elementul (linie) și culoarea sa și creează un obiect SpannableStringBuilder pe textul elementului. Ulterior adaugă elementul colorat în lista globală **History.operații**, de unde va putea fi folosită de restul aplicației.

```
//declaram o functie care la pornirea aplicatiei citeste inserarile din tabela istoric si le pune in lista mutabila "operatii"
fun incarcaIstoric(db:SQLiteDatabase) {
    val cursor = db.rawQuery( sql: "SELECT linie, culoare FROM istoric", selectionArgs: null)

    if(cursor.moveToFirst()) { //daca cursorul a returnat cel putin un rand
        do {
            val linie = cursor.getString(cursor.getColumnIndexOrThrow( columnName: "linie"))
            val culoare = cursor.getString(cursor.getColumnIndexOrThrow( columnName: "culoare"))

            val sb = SpannableStringBuilder(linie)
            sb.setSpan(ForegroundColorSpan(Color.parseColor(culoare)),
                start: 0,
                linie.length,
                Spannable.SPAN_EXCLUSIVE_EXCLUSIVE
            )

            History.operatii.add(sb)
        } while (cursor.moveToNext()) //trece la urmatorul rand pana cand nu mai sunt
    }
    cursor.close()
}
```

Și am apelat această funcție în metoda onCreate() din activitatea principală astfel:

```
//incarcam istoricul din baza de date
if(History.istoricIncarcat == false)
{
    val dbHelper = CalculatorDatabaseHelper( context: this)
    val db = dbHelper.readableDatabase

    incarcaIstoric(db)
    db.close()
    History.istoricIncarcat = true
}
```

SALVAREA ȘI RESTAURAREA DATELOR

Pentru a nu pierde datele importante atunci când se schimbă orientarea telefonului, am suprascris funcția **onSaveInstanceState()** din fragmentul Calcul, funcție în care am salvat valorile mai multor variabile punând o etichetă pentru fiecare variabilă.

```kotlin
//functie pentru salvarea datelor inainte de inchiderea aplicatiei
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    outState.putString("text_curent", text_curent.text.toString())
    outState.putString("text_anterior", text_anterior.text.toString())
    outState.putBoolean("isBase10", History.isBase10)
    outState.putString("operatieCurenta", operatieCurenta)
    outState.putString("operandAnterior", operandAnterior)
    outState.putBoolean("laDeschidere", History.laDeschidere)
    outState.putBoolean("bazaAdaugata", History.bazaAdaugata)
    outState.putBoolean("istoricIncarcat", History.istoricIncarcat)
}
```

În funcția **onViewCreated()** a aceluiași fragment am verificat dacă există date salvate, iar dacă da, atunci le-am preluat pe fiecare ajutându-mă de etichetele sau cheile cu care au fost asociate și le-am pus din nou în variabilele de unde erau.

```kotlin
//cod pentru restaurarea datelor la repornirea aplicatiei (atunci cand se intoarce ecranul):
if(savedInstanceState != null)
{
    text_curent.setText(savedInstanceState.getString( key: "text_curent",  defaultValue: ""))
    text_anterior.setText(savedInstanceState.getString( key: "text_anterior",  defaultValue: ""))
    History.isBase10 = savedInstanceState.getBoolean( key: "isBase10",  defaultValue: true)
    operatieCurenta = savedInstanceState.getString( key: "operatieCurenta",  defaultValue: null)
    operandAnterior = savedInstanceState.getString( key: "operandAnterior",  defaultValue: "0")
    History.laDeschidere = savedInstanceState.getBoolean( key: "laDeschidere",  defaultValue: true)
    History.bazaAdaugata = savedInstanceState.getBoolean( key: "bazaAdaugata",  defaultValue: true)
    History.istoricIncarcat = savedInstanceState.getBoolean( key: "istoricIncarcat",  defaultValue: true)

    //refacem starea butonului bazei
    button1016.text = if (History.isBase10) "Baza 10" else "Baza 16"
    val culoare = if (History.isBase10) R.color.base10_color else R.color.base16_color
    context?.let {
        button1016.setBackgroundColor(ContextCompat.getColor(it, culoare))
    }

    //refacem si starea butoanelor hexa
    for(buton in butoane_hexa) {
        buton.isEnabled = !History.isBase10
        buton.alpha = if (History.isBase10) 0.5f else 1.0f
    }

}
```