Mobil programozási alapok

Féléves feladat

Költéskövető Android alkalmazás

Készítette: **Palencsár Enikő**

Neptun kód: **YD11NL**

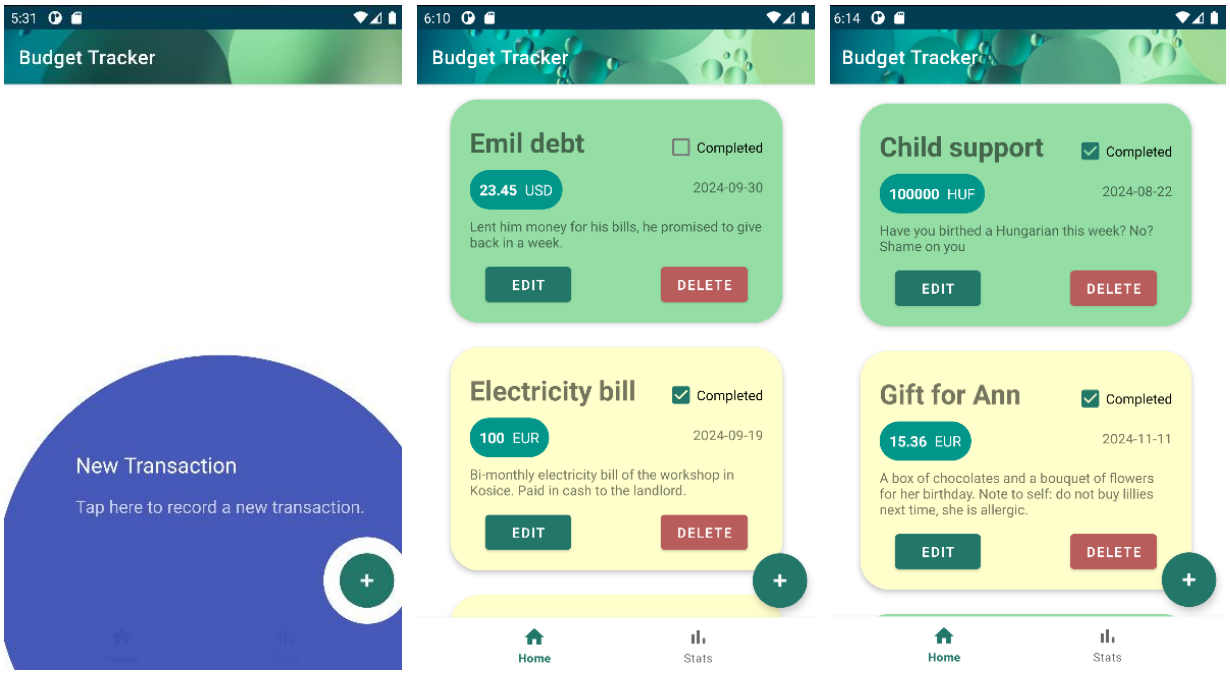
Tartalomjegyzék

1. Felhasználói felület

A *Budget Tracker* egy egyszerű költéskövető mobilalkalmazás, lokális *Room* adatbázissal. Felületén a felhasználók felvehetik a kimenő és beérkező pénzösszegeket tetszőleges pénznemben, megtekinthetik a tranzakciók listáját, illetve szemléletes diagram formájában havi összesítéseket olvashatnak a pénzmozgásokról. Lehetőség nyílik a már felvett tranzakciók módosítására, törlésére is. Az alkalmazáson belüli navigáció a képernyő alján megjelenő menüszalag segítségével valósul meg. A futtatáshoz minimum Android 8.0 szükséges.

1.1. Tranzakciós lista

A tranzakciós lista oldalon a felvett tranzakciók felsorolása tekinthető meg, egymás alatt elhelyezkedő kártyák formájában, részleteikkel együtt. Egy tranzakció főbb attribútumai a rövid, leíró jellegű címke, az érintett összeg és pénznem, a pénzmozgás dátuma, illetve a tranzakcióra vonatkozó hosszabb megjegyzés. A kimenő tranzakciókat sárga, a bejövő tranzakciókat zöld háttérszín jelzi. A tranzakciós kártya jobb felső sarkában egy jelölőnégyzettel megadható, hogy aktív tartozásról vagy már teljesített kifizetésről beszélünk. Az adott tranzakció módosítására és törlésére szolgáló gombok a kártya alján találhatók. Új tranzakciót a jobb alsó sarokban az oldaltartalom felett lebegő gombra kattintva vehetünk fel. A telepítést követően az alkalmazás első megnyitásakor a gomb körül kék kiemelés és magyarázó szöveg jelenik meg.

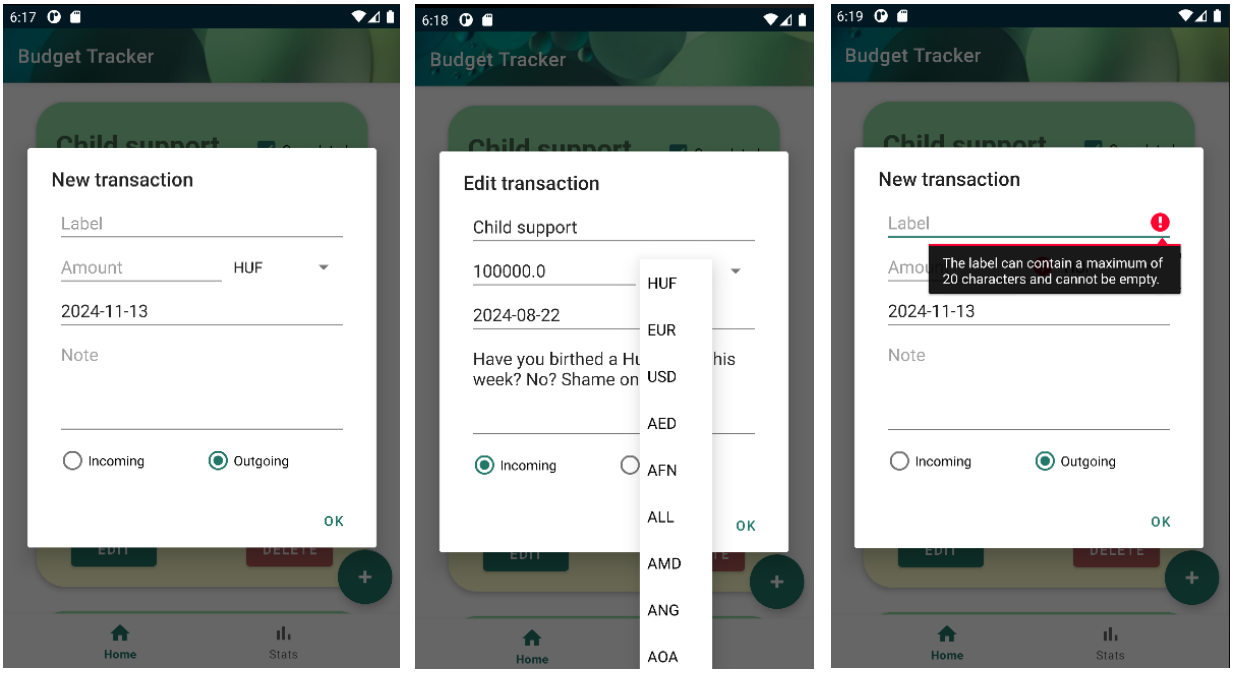


A tranzakciós lista elemeinek rendezése dátum szerint, csökkenő sorrendben történik. Az újonnan felvett tranzakciók első megjelenésükkor dátumtól függetlenül a lista elején kapnak helyet, míg a módosított dátumú tranzakciók módosítás után az eredeti helyükön maradnak. A következő teljes rendezés csak a lista oldal újbóli megnyitásakor zajlik le.

1.2. Felvételi és módosítási űrlap

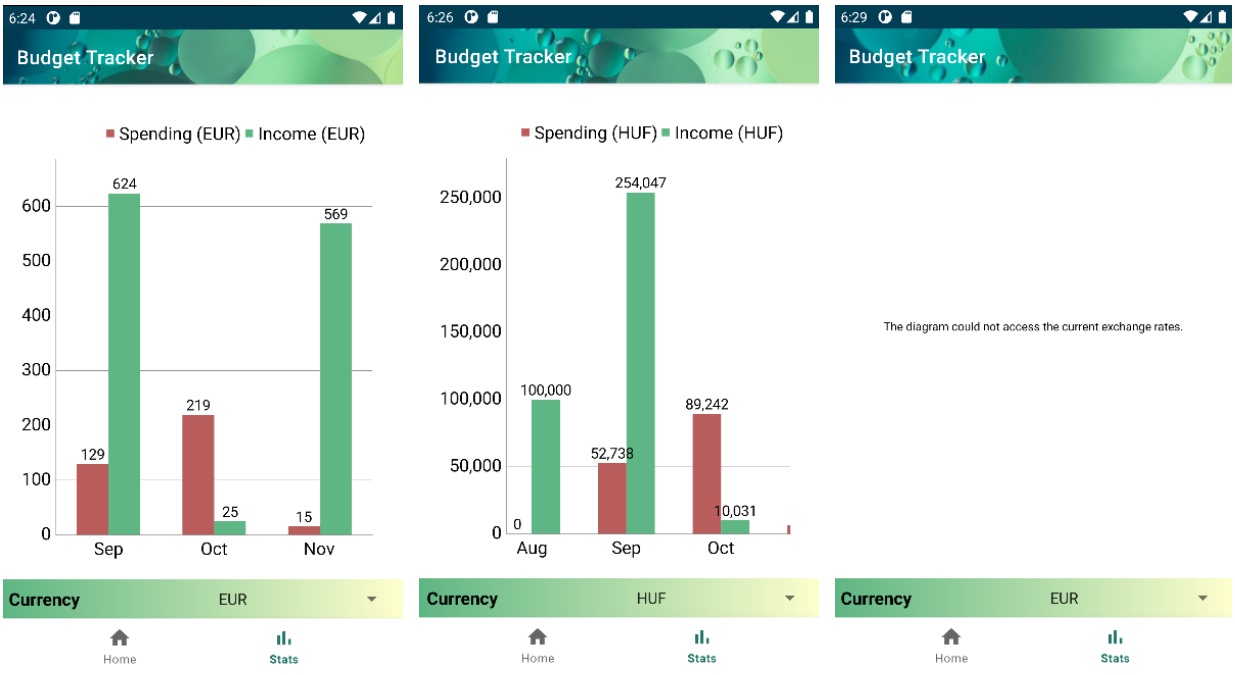
A tranzakciók felvételére és módosítására szolgáló űrlap felugró dialógusablak formájában került megvalósításra. Szöveges beviteli mezők szolgálnak a tranzakció címkéjének és a kapcsolódó megjegyzésnek a megadására, ez utóbbi többsoros, jelezve, hogy szükség esetén hosszabb szöveg

is megadható. A pontos összeg lebegőpontos számmezőbe vihető fel, míg a pénznem legördülő listából választható ki, melynek elején a leggyakoribb pénznemek (HUF, EUR, USD) szerepelnek, a többi valuta pedig abc rendben követi őket. Az alapértelmezett pénznem a forint. Az, hogy a tranzakció bejövő vagy kimenő, rádiógombokkal specifikálható. A tranzakció dátuma felugró naptár segítségével adható meg, alapértelmezett értéke a mai nap. Az új tranzakció a létrejöttekor mindenképpen teljesítetlen, ez a tulajdonság csak a lista oldalon módosítható.



A címke maximális hossza 20 karakter lehet, az összeg pedig mindenképpen pozitív szám. A címke, az összeg és a dátum semmiképpen sem maradhatnak üresen, de a megjegyzés mezőt nem kötelező kitölteni. Sikeres beküldést követően a dialógusablak bezárul, sikertelen próbálkozás esetén a hibás beviteli mezők mellett tájékoztató hibaüzenetek jelennek meg.

1.3. Statisztika oldal



A statisztika menüpont alatt az elmúlt egy év összes beérkező és kimenő tranzakciójának összegzése jelenik meg havi felbontásban, csoportosított oszlopdiagram formájában. A diagramon egyszerre három hónap adatai láthatók, a többi adatsor horizontális görgetéssel tekinthető meg. Az összesítéskor használt pénznemet a felhasználó választhatja ki egy legördülő listából. Az árfolyamok lekérdezéséhez internetkapcsolatra van szükség. Sikertelen generálás esetén a diagram helyén hibaüzenet jelenik meg.

2. Kód

A *Budget Tracker* alkalmazást az Android Studio Bumblebee (2021.1.1) verziójában, Kotlin nyelven valósítottam meg. A tranzakciók mentését Room adatbázis biztosítja. Az aktuális átváltási árfolyamok lekérdezését az alkalmazás a Fixer API segítségével végzi.

2.1. Elérhetőség

Az alkalmazáshoz tartozó forrásfájlokat tartalmazó Github repository linkje: **https://github.com/enikop/MobileProgramming2024**

Futtatás előtt annak érdekében, hogy az oszlopdiagram megjelenítése hibamentes legyen, létre kell hozni az */app/src/main/res/raw/key.txt* állományt, amelyben el kell helyezni a saját, érvényes Fixer API kulcsunkat. Amennyiben ez nem történik meg, az alkalmazásban a diagram helyén hibaüzenet lesz olvasható. A kulcs külön fájlba izolálására azért volt szükség, hogy ne szerepeljen a saját titkos API kulcsom egy nyilvános Github repositoryban.

2.2. Adatbázis

Az adatbáziskapcsolatot az alkalmazáson belül a singleton *AppDatabase* osztály menedzseli, melynek egyetlen példánya a *getInstance()* függvénnyel történő első hivatkozáskor jön létre. Az adatbázis adatai a helyi *budgetTracker.db* állományban kerülnek tárolásra.

```
@Database(entities = [Transaction::class], version = 1)
@TypeConverters(DateConverter::class, CurrencyConverter::class)
abstract class AppDatabase : RoomDatabase() {
    abstract fun transactionDao(): TransactionDAO

    companion object {
        private var INSTANCE: AppDatabase? = null

        fun getInstance(context: Context): AppDatabase {
            if (INSTANCE == null) {
                INSTANCE = Room.databaseBuilder(context.applicationContext,
                    AppDatabase::class.java, "budgetTracker.db")
                    .build()
            }
            return INSTANCE!!
        }

        fun destroyInstance() {
            INSTANCE = null
        }
    }
}
```

Az adatbázisban tárolandó egyedek definíciója a *Transaction* modell osztályban kapott helyet. Egy

tranzakció elsődleges kulcsa generált Long típusú érték, címkéje String típusú, a tranzakcióhoz fűződő megjegyzés szintén szöveges értéket vehet fel, míg az érintett összeg dupla pontosságú lebegőpontos szám. Minden tranzakciónak van továbbá két Boolean adattagja: az *isIncoming* paraméter jelzi, hogy a pénzmozgás beérkező vagy kimenő jellegű-e, míg az *isCompleted* paraméter megmutatja, hogy tartozásról vagy teljesített kifizetésről beszélünk. A tranzakció dátuma LocalDate típusú, alapértelmezése a mai nap, míg a *currency* adattag enum típusú, alapértelmezetten euró. A *Currency* enum osztály minden olyan valutát tartalmaz, amelyre vonatkozóan a Fixer API árfolyam adatokat továbbít.

```
@Entity(tableName = "transactions")
data class Transaction(
    @PrimaryKey(autoGenerate = true) var id: Long?,
    @ColumnInfo(name = "label") var label: String,
    @ColumnInfo(name = "amount") var amount: Double,
    @ColumnInfo(name = "incoming") var isIncoming: Boolean,
    @ColumnInfo(name = "completed") var isCompleted: Boolean,
    @ColumnInfo(name = "note") var note: String,
    @ColumnInfo(name = "date") var date: LocalDate = LocalDate.now(),
    @ColumnInfo(name = "currency") var currency: Currency = Currency.EUR
    ) : Serializable
```

Az adatbázishoz típuskonverter segédosztályokat rendeltem a nem elemi *Transaction* adattagok String értékre történő leképzésére (*DateConverter*, *CurrencyConverter*), ezek metódusai futnak le az adatbázis és az alkalmazás közötti adatcsere során. A tranzakció egyedek a DB-n belül egy *transactions* nevű táblában kapnak helyet.

Az *AppDatabase* osztály rendelkezik egy *TransactionDAO* típusú taggal, ami az adatokon végezhető alapvető adatbázis-műveleteket definiálja. Az általános CRUD műveleteken túl ebben az interfészben kapott helyet a paraméterként megadott hónap összes tranzakciójának lekérdezése is, melyet később a diagram-generálás során használtam fel.

```
@Dao
interface TransactionDAO {

    @Query("SELECT * FROM transactions")
    fun findAllItems(): List<Transaction>

    @Query("SELECT * FROM transactions WHERE date LIKE :yearMonth || '%'")
    fun findTransactionsForMonth(yearMonth: String): List<Transaction>

    @Insert
    fun insertItem(item: Transaction): Long

    @Delete
    fun deleteItem(item: Transaction)

    @Update
    fun updateItem(item: Transaction)
}
```

2.3. Pénzváltó API

A Fixer API-val való kapcsolatot a network csomag osztályai és a Retrofit könyvtár biztosítják. Az *ExhangeAPI* interfész egy GET kérést definiál a legutóbbi árfolyamok lekérdezésére, melynek query paramétere az API használatához szükséges kulcs.

```
interface ExchangeAPI {
    @GET("/api/latest")
    fun getRates(@Query("access_key") key: String) : Call<ExchangeResult>
}
```

Az *ExhangeService* osztály egy Retrofit objektumot definiál a megfelelő elérési útvonallal, amelyhez hozzácsatolja az *ExchangeAPI* interfészt. A *getRates()* metódusban a *MainActivity*-ben rendszerindításkor beolvasott és eltárolt kulcs segítségével megtörténik az API hívás. A metódusnak egy *RatesCallback* típusú paramétere van, ami egy olyan saját interfész típus, mely egy siker esetén hívódó *onRatesReceived()*, illetve egy hiba esetén futtatandó *onError()* metódust definiál.

```
class ExchangeService {
    private var retrofit: Retrofit
    private var api: ExchangeAPI
    constructor() {
        retrofit = Retrofit.Builder()
            .baseUrl("http://data.fixer.io")
            .addConverterFactory(GsonConverterFactory.create())
            .build()
        api = retrofit.create(ExchangeAPI::class.java)
    }

    fun getRates(callback: RatesCallback) {
        val call = api.getRates(MainActivity.API_KEY)
        call.enqueue(object : Callback<ExchangeResult> {
            override fun onResponse(call: Call<ExchangeResult>, response:
              Response<ExchangeResult>) {
                if (response.body() != null) {
                    callback.onRatesReceived(response.body() as ExchangeResult)
                } else {
                    callback.onError(Throwable("No rates available"))
                }
            }

            override fun onFailure(call: Call<ExchangeResult>, t: Throwable) {
                callback.onError(t)
            }
        })
    }
}
```

Az API hívás *ExchangeResult* objektummal tér vissza, mely az API interfészére illeszkedő objektumként került definiálásra a *model* csomagban.

```
class ExchangeResult (val success: Boolean?, val timestamp: Number?,
                      val base: String?, val date: String?, val rates: Rates?)
```

A *Rates* osztály szintén saját típus, adattagjai között az összes Fixer API által támogatott pénznem szerepel. Egyetlen metódust tartalmaz, amely reflection segítségével egy adott *Currency* objektum által kijelölt valuta árfolyamával tér vissza. Ehhez az szükséges, hogy a *Currency* enum

elemek nevei és a *Rates* osztály adattagjainak nevei megegyezzenek egymással.

```
data class Rates(val AED: Number?, val AFN: Number?, […]){
    fun getRate(currency: Currency): Double? {
        val rateProperty =
            Rates::class.memberProperties.find { it.name == currency.name }
        val rate = (rateProperty?.get(this) as? Number)?.toDouble()

        // If the rate is null, attempt to retrieve the EUR rate as a fallback
        return rate ?: (EUR?.toDouble())
    }
}
```

2.4. MainActivity

A *MainActivity* activity osztály az alkalmazás belépési pontja. Fő feladata a menükezelés, azaz a *HomeFragment* és a *StatsFragment* közötti navigáció lebonyolítása, de az API kulcs beolvasását is ez az osztály végzi el. Az *activity_main.xml* nézet tartozik hozzá, amely függőleges LinearLayout segítségével három fő egységre tagolja a képernyőt.

1. **Fejléc:** A fejlécen az alkalmazás neve alatt Ken Burns effektussal animált kép jelenik meg, ezt AppBarLayoutba ágyazott FrameLayout segítségével valósítottam meg.
2. **Töredék tároló:** A töredék tároló egy sima FrameLayout, melyet a *MainActivity* kezelője cserél ki az aktuális töredékre. Ennek parancsa a *HomeFragment* esetén:

   ```
   supportFragmentManager.beginTransaction()
       .replace(R.id.fragment_container, HomeFragment())
       .commit()
   ```

3. **Navigációs sáv:** A menü BottomNavigationView segítségével kerül megjelenítésre a képernyő alján. Az ikonokkal ellátott menüpontok definíciója a különálló *menu/menu.xml* állományban szerepel.

2.5. HomeFragment

A *HomeFragment* töredék a tranzakciós lista megjelenítéséért felelős, a *fragment_home.xml* nézet tartozik hozzá. A nézetben a tranzakciókat részletező RecyclerView mellett az új tranzakció hozzáadását kezdeményező lebegő gomb is szerepel, melynek lenyomására *TransactionDialog* típusú felugró ablak keletkezik. A gombot az alkalmazás első futtatásakor a *HomeFragment* osztály egy MaterialTapTargetPrompt objektummal egészíti ki.

```
if (isFirstRun()) {
    MaterialTapTargetPrompt.Builder(activity)
        .setTarget(view.fab)
        .setPrimaryText("New transaction")
        .setSecondaryText("Tap here to record a new transaction.")
        .show()
}
```

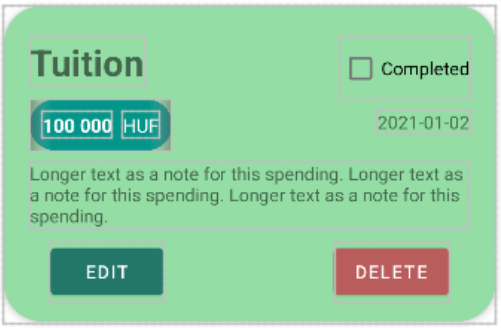
A tranzakciókat tartalmazó RecyclerView inicializálása is a *HomeFragment* feladatai közé tartozik: az osztály a *TransactionDAO findAllItems()* metódusa segítségével egy adatbázis szálon betölti az összes tárolt tranzakciót, majd ezeket felhasználva a UI szálon *BudgetTrackerAdapter* objektumot hoz létre, melyet egy érintési műveleteket definiáló segédobjektummal együtt hozzárendel a RecyclerView-hoz.

```
private fun initRecyclerView() {
    val dbThread = Thread {
        val items = AppDatabase.getInstance(requireContext())
         .transactionDao().findAllItems()
        activity.runOnUiThread {
            adapter = BudgetTrackerAdapter(requireContext(), items, this)
            rvTransaction.adapter = adapter
            val callback = TransactionTouchHelperCallback(adapter)
            val touchHelper = ItemTouchHelper(callback)
            touchHelper.attachToRecyclerView(view?.rvTransaction)
        }
    }
    dbThread.start()
}
```

A *HomeFragment* ezenkívül a *TransactionHandler* interfészt is implementálja, ami tranzakciók dialógusablak segítségével történő létrehozásához és módosításához definiál callback függvényeket. Az interfész *HomeFragment* által megvalósított főbb metódusai a módosítási és létrehozási operációkhoz kapcsolódó adatbázis-műveleteket végzik el külön szálon, illetve a korábban létrehozott adapteren keresztül frissítik a RecyclerView tartalmát. A *showEditItemDialog()* interfész metódus a szerkesztési felugró ablak létrehozásáért, a kiválasztott tranzakció, mint argumentum átadásáért felelős a megfelelő *TransactionDialog* objektumnak.

2.6. BudgetTrackerAdapter

A *BudgetTrackerAdapter* osztály a tranzakciókat listázó RecyclerView tartalmának kezeléséért felelős. Legfontosabb adattagja a *Transaction* objektumok folyamatosan változó listája, melyet az adapter létrehozásakor dátum szerint csökkenő sorrendbe rendez. Az adapter a tranzakciók szerkesztésének kezdeményezéséhez egy *TransactionHandler* taggal is rendelkezik, amely jelen felállásban a szülő *HomeFragment* elemmel egyezik meg.



Az adapter saját belső *ViewHolder* osztály létrehozásával segíti a *row_item.xml* nézethez való kötődést, mely a felhasználói interfész valamennyi kódból módosítandó elemét definiálja. A *row_item.xml* állomány egyetlen tranzakció kártya nézetű ábrázolását írja le, a kártyán belül ConstraintLayout elrendezést alkalmazva. A kártya tetején kiemelt betűmérettel a tranzakció címke szerepel, alatta eltérő háttérszínnel az összeg és a pénznem horizontális LinearLayout elrendezésben. Megjelenítésre kerül a tranzakció dátuma és a többsoros megjegyzés is. A kártya jobb felső sarkában egy Checkbox objektumot, alján egy szerkesztő és egy törlő gombot tartalmaz, melyek kezelőit az adapter osztály definiálja.

Az adapter *onBindViewHolder()* felüldefiniált metódusa átadja a RecyclerView szövegmezőinek az aktuális *Transaction* elem adatait, a tranzakció irányától függően kijelöli a kártya háttérszínét, illetve beállítja a gombok és a jelölőnégyzet kezelőit. A Checkbox kijelölésének módosítása automatikusan adatbázis-műveletet kezdeményez, ahol felülíródik az *incoming* boolean adatmező.

```
holder.cbComplete.setOnClickListener {
    items[holder.adapterPosition].isCompleted = holder.cbComplete.isChecked
    val dbThread = Thread {
```

```
            AppDatabase.getInstance(context).transactionDao()
             .updateItem(items[holder.adapterPosition])
        }
        dbThread.start()
    }
```

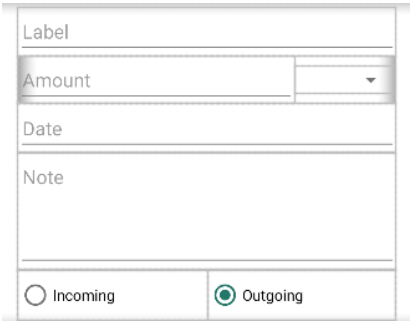
Az adapteren belül 3 alapvető metódus szolgál az adatmanipulációs műveletek kezdeményezésére. Az *addItem()* felveszi a paraméterben megadott új elemet a tranzakciók listájának elejére, az *updateItem()* a paraméterben átadott tranzakcióval felülírja a megfelelő listaelemet, míg a *deleteItem()* metódus maga végzi el az adott pozícióban lévő tranzakció törlését az adatbázisból.

```
fun addItem(item: Transaction) {
    items.add(0, item)
    notifyItemInserted(0)
}

fun deleteItem(position: Int) {
    val dbThread = Thread {
        AppDatabase.getInstance(context).transactionDao().deleteItem(
            items[position])
        (context as MainActivity).runOnUiThread{
            items.removeAt(position)
            notifyItemRemoved(position)
        }
    }
    dbThread.start()
}

fun updateItem(item: Transaction) {
    val idx = items.indexOf(item)
    items[idx] = item
    notifyItemChanged(idx)
}
```

Az érintési műveletek – elemek kipöckölése, áthúzása más pozícióra – megfelelő kezeléséhez az osztály megvalósítja a *TransactionTouchHelperAdapter* interfész *onItemDismissed()* és *onItemMoved()* metódusait, előbbi egyszerűen a *deleteItem()* metódust hívja meg, utóbbi pedig a tranzakciók tömbjében az új pozícióra mozgatja az érintett elemet.

2.7. TransactionDialog

A *TransactionDialog* osztály a tranzakciós űrlap felugró ablak formájában történő megjelenítéséért felelős, kiterjeszti a DialogFragment gyári osztályt. A *dialog_transaction_form.xml* nézet tartozik hozzá, mely a tranzakció adattagoknak megfelelően különböző beviteli mezőket definiál egy vertikális LinearLayout elrendezésen belül. A címke és a dátum elemekhez egysoros, míg a megjegyzés adattaghoz többsoros szöveges beviteli mező tartozik. A dátum mező yyyy-MM-dd formátumú, szabad kézzel nem írható, megérintése egy dátumválasztó dialógusablak megjelenését eredményezi. Az érintett pénzösszeg megadására szolgáló mező decimális formátumú, mellette a valuta specifikálására dropdown típusú Spinner található. A tranzakció irányának kijelölését egy RadioGroup csoporton belüli két rádiógomb teszi lehetővé.



A *TransactionDialog* osztály ezen mezők inicializálásáért, ellenőrzéséért, és a megfelelő

űrlapleadási művelet végrehajtásáért felelős, ez utóbbihoz példányosításakor egy *TransactionHandler* típusú adattagot fogad. Az *onCreateDialog()* örökölt metódusban az űrlaptartalom egy AlertDialogba ágyazva, OK gombbal ellátva jön létre. Itt hívódik meg az *initDialogContent()* függvény, amely a megfelelő nézet csatolását követően – attól függően, hogy a dialógus tranzakció módosítása vagy létrehozása céljából jött létre – beállítja a beviteli mezők kezdeti értékeit, illetve a felugró ablak címét. A célra a dialógusablak argumentumaiból következtethetünk, melyek között módosítás esetén *MainActivity.KEY_ITEM_TO_EDIT* kulccsal szerepel a módosítandó tranzakció. Az *initDialogContent()* függvényben kerül feltöltésre a pénznemet kijelölő Spinner objektum is egy ArrayAdapter segítségével, a *Currency* enumban szereplő lehetséges értékek felhasználásával.

```
val currencies = Currency.values().map { it.name }
val adapter = ArrayAdapter(requireContext(),
  android.R.layout.simple_spinner_item, currencies)
adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
spCurrency.adapter = adapter
```

Az *initDialogContent()* metódusban szerepel a dátum beviteli mező működési logikája is. A dátum alapértelmezett értéke a mai nap, a beviteli mezőre történő kattintás pedig a *showMaterialDatePicker()* metódust hívja meg. Ennek egy olyan callback függvény a paramétere, ami beállítja a dátum mező szövegét a naptár dialógusban kiválasztott dátumra.

```
etDate.setText(dateConverter.dateToString(LocalDate.now()))
etDate.setOnClickListener {
    showMaterialDatePicker { selectedDate ->
        etDate.setText(dateConverter.dateToString(selectedDate))
    }
}
```

A *showMaterialDatePicker()* függvény egy dátumválasztó felugró ablakot generál, melyben alapértelmezetten a dátum EditText mezőben szereplő dátum lesz kijelölve. Az OK gombra kattintva a kiválasztott dátum LocalDate objektummá konvertálódik, majd meghívódik a függvénynek paraméterként átadott callback.

```
private fun showMaterialDatePicker(onDateSelected: (LocalDate) -> Unit) {
    val selectedDate = dateConverter.fromString(etDate.text.toString())?
        .atStartOfDay(ZoneId.systemDefault())?.toInstant()?.toEpochMilli()
        ?: MaterialDatePicker.todayInUtcMilliseconds()
    val datePicker = MaterialDatePicker.Builder.datePicker()
        .setTitleText("Select date")
        .setSelection(selectedDate)
        .build()

    datePicker.addOnPositiveButtonClickListener { selection ->
        // Convert the selected timestamp to LocalDate
        val newSelectedDate = Instant.ofEpochMilli(selection).
            atZone(ZoneId.systemDefault()).toLocalDate()
        onDateSelected(newSelectedDate)
    }

    datePicker.show(requireActivity().supportFragmentManager,
      "MATERIAL_DATE_PICKER")
}
```

A felugró ablak bezárását az *onResume()* örökölt metódus vezényli, mely az OK gomb lenyomásához rendel eseménykezelőt. A kezelő függvény ellenőrzi a beviteli mezőkben szereplő adatokat, és hiba esetén a mezőkhöz rendeli a megfelelő hibaüzeneteket. Helyes adatok megadása esetén a dialógusablak szerepétől függően meghívódik a *handleItemEdit()* vagy a *handleItemCreate()* metódusok valamelyike, majd bezáródik a dialógusablak. A *handleItemEdit()* és a *handleItemCreate()* metódusok kinyerik a beviteli mezőkben szereplő értékeket, ezeket hozzárendelik egy tranzakció objektumhoz, majd meghívják az elvégzendő operációnak megfelelő, *transactionHandler* adattagban definiált callback függvényt.

```
positiveButton.setOnClickListener {
    var isValid = true
    if(etLabel.text.isEmpty() || etLabel.text.length > 20) {
        etLabel.error = "The label can contain a maximum of 20 characters and
          cannot be empty."
        isValid = false
    }
    if(etAmount.text.isEmpty() || etAmount.text.toString().toDouble() <= 0) {
        etAmount.error = "The amount must be a number above 0."
        isValid = false
    }
    if(etDate.text.isEmpty()) {
        etDate.error = "The date cannot be empty."
        isValid = false
    }
    if(isValid) {
        val arguments = this.arguments
        if (arguments != null &&
          arguments.containsKey(MainActivity.KEY_ITEM_TO_EDIT)) {
            handleItemEdit()
        } else {
            handleItemCreate()
        }
        dialog.dismiss()
    }
}
```

2.8. StatsFragment

A *StatsFragment* osztály a havi statisztikát tartalmazó oszlopdiagram megjelenítését végzi el. A hozzá tartozó *fragment_stats.xml* nézet vertikális LinearLayout elrendezésben két fő elemet tartalmaz: magát a BarChart diagramot, melynek magassága a képernyőn fennmaradó területtel egyenlő, illetve a pénznem kiválasztására szolgáló Spinnert. A legördülő menü kiemelésére saját színátmenetes hátteret definiáltam (*drawable/gradient-background.xml*). A diagramokkal kapcsolatos funkcionalitást az alkalmazás com.github.PhilJay:MPAndroidChart:v3.1.0 függősége biztosítja**.**

Az oszlopdiagram felépítése a *MonthlyBarChartBuilder* segédosztály feladata, melynek adattagként átadódik a módosítandó BarChart objektum, az elkészítendő diagram szövegméretének bázisszáma, illetve a két különböző oszloptípus színe. Az osztály két publikus metódusa a *setNoDataText()* és a *build()*. A *setNoDataText()* az üres diagram helyére kerülő placeholder szöveget és annak színét állítja be, míg a *build()* az átadott adatok segítségével felépíti a kért oszlopdiagramot. Ez utóbbi függvény paraméterként a két ábrázolandó adatsort, az x tengelyre helyezendő címkéket, illetve a jelmagyarázatban megjelenítendő pénznemet fogadja.

Az adatsorokat az alábbi logika szerint konvertáltam BarData objektummá, ahol az egyes oszlopok relatív szélességét is kijelöltem:

```
val spendingEntries = spendingData.mapIndexed { index, value ->
  BarEntry(index.toFloat(), value) }
val incomeEntries = incomeData.mapIndexed { index, value ->
  BarEntry(index.toFloat(), value) }

val spendingDataSet = BarDataSet(spendingEntries, "Spending
("+currency.name+")").apply {
    color = spendingColour
    valueTextSize = chartTextSize
}
val incomeDataSet = BarDataSet(incomeEntries, "Income
("+currency.name+")").apply {
    color = incomeColour
    valueTextSize = chartTextSize
}

val data = BarData(spendingDataSet, incomeDataSet)
data.barWidth = 0.3f
barChart.data = data
```

Ezt követően elvégeztem a jelmagyarázat formázását, majd engedélyeztem a diagramon a drag és a scale műveleteket, hogy a régebbi havi adatok vízszintes görgetéssel váljanak elérhetővé. Mindezt a tengelyek személyre szabása követte. Az x tengely értékfeliratait a *labels* tömb elemeire cseréltem úgy, hogy két egymás melletti oszlophoz tartozzon egy címke. Az x tengely teljes tartományát a bejövő adatsorok szélességével állítottam megegyező nagyságúra, a látható tartomány méretét pedig 3 címkében rögzítettem.

```
// X-axis labels and format
val xAxis = barChart.xAxis
xAxis.textSize = chartTextSize * 1.2f
xAxis.valueFormatter = IndexAxisValueFormatter(labels)
xAxis.position = XAxis.XAxisPosition.BOTTOM
xAxis.setDrawGridLines(false)
xAxis.axisMinimum = 0.0f
xAxis.axisMaximum = spendingData.size.toFloat()
xAxis.granularity = 1.0f
xAxis.setCenterAxisLabels(true)
barChart.setVisibleXRangeMinimum(3f)
barChart.setVisibleXRangeMaximum(3f)
```

A diagram létrehozásának záró akkordját az oszlopközök beállítása jelentette. Ezt követően arról is gondoskodtam, hogy a diagram automatikusan az utolsó megjelenített hónaphoz gördüljön.

```
// Spacing between bars
barChart.groupBars(0.0f, 0.4f, 0f)
barChart.invalidate()

//Scroll to this month
val lastIndex = incomeEntries.size
barChart.moveViewToX(lastIndex.toFloat() - 1)
```

A *StatsFragment* tehát egy *MonthlyBarChartBuilder* objektumot definiál a diagram konténer feltöltésére, illetve az *ExhangeService* szolgáltatásait felhasználva előállítja a diagramot generáló adatsorokat. Az osztály örökölt *onViewCreated()* metódusában lekéri a színleíró XML fájlban megadott diagram színeket, a nézetből származó BarChart objektum átadásával felépíti a

későbbiekben használatos *MonthlyBarChartBuilder* példányt, feltölti a Spinnert a választható valutákkal, és eseménykezelőt rendel a kiválasztás megváltozásához. Az eseménykezelő *onItemSelected()* függvénye két esetet különböztet meg: amennyiben az osztály *exchangeResult* adattagja üres, lekéri az átváltási árfolyamokat a hálózaton keresztül, ellenkező esetben megjegyzi a kiválasztott pénznemet, majd megkezdi a diagram felépítéséhez szükséges adatbázis lekérdezéseket.

```
spDiagramCurrency.onItemSelectedListener = object :
  AdapterView.OnItemSelectedListener {
    override fun onItemSelected(parent: AdapterView<*>, view: View?,
      position: Int, id: Long) {
        if (exchangeResult == null)
            exchangeService.getRates(this@StatsFragment)
        else {
            val selectedCurrencyName = parent.selectedItem as String
            selectedCurrency = Currency.valueOf(selectedCurrencyName)
            initTransactions()
        }
    }
    override fun onNothingSelected(parent: AdapterView<*>) {
        // Handle no selection case
    }
}
```

A *StatsFragment* a valutákra vonatkozó hálózati kérés feldolgozásához implementálja a *RatesCallback* interfészt. Amennyiben a kérés során kivétel keletkezik, az *onError()* függvény fut le, amely hibaüzenetre állítja át az oszlopdiagramhoz tartozó placeholder szöveget. Ha a kiszolgáló válasza megérkezik, az *onRatesReceived()* metódus ez alapján ellenőrzi, hogy sikeres volt-e a lekérdezés, és amennyiben igen, beállítja a *StatsFragment exchangeResult* adattagját, majd megkezdi a diagram felépítéséhez szükséges adatbázis lekérdezéseket.

```
override fun onRatesReceived(result: ExchangeResult) {
    if(result.success != null && result.success) {
        this.exchangeResult = result
        initTransactions()
    } else {
        chartBuilder.setNoDataText(
            "The diagram could not access the current exchange rates.",
            Color.BLACK
        )
    }
}
```

A megjelenítéshez az adatok előkészítését az *initTransactions()* metódus végzi, ami egy külön szálon egyesével lekérdezi az előző hónapokra vonatkozó összes költést, az osztály *exchangeResult* tagjának segítségével a megadott pénznemre konvertálja az ezekben szereplő pénzösszegek értékét (*convertAmount()*), majd összegzéssel előállítja az adott havi beérkező és kimenő egyenleget (*calculateMonthlyStat()*). Ezt követően a UI szálon meghívódik a *setupChart()* metódus, ami felszólítja a korábban létrehozott *MonthlyBarChartBuilder* példányt az adott paraméterekkel definiált diagram felépítésére.

```
private fun initTransactions() {
    val dbThread = Thread {
        val today = LocalDate.now()
        val months = (monthsShown - 1 downTo 0).map {
            today.minusMonths(it.toLong()).format(monthFormatter)
        }
```

```
        labels = (monthsShown - 1 downTo 0).map {
            today.month.minus(it.toLong()).getDisplayName(TextStyle.SHORT,
              Locale.getDefault())
        }

        val stats = mutableListOf<Pair<Float, Float>>()
        for (i in 0 until monthsShown) {
            val transactions = AppDatabase
              .getInstance(requireContext()).transactionDao()
              .findTransactionsForMonth(months[i])
            stats.add(calculateMonthlyStat(transactions, selectedCurrency))
        }

        spendingData = stats.map { it.second }
        incomeData = stats.map { it.first }

        activity?.runOnUiThread {
            setupChart()
        }
    }
    dbThread.start()
}

private fun calculateMonthlyStat(
    transactions: List<Transaction>,
    targetCurrency: Currency
): Pair<Float, Float> {

    val incomingSum = transactions
        .filter { it.isIncoming }
        .sumOf {
            convertAmount(it.amount, it.currency, targetCurrency,
              exchangeResult!!.rates!!)
        }

    val outgoingSum = transactions
        .filter { !it.isIncoming }
        .sumOf {
            convertAmount(it.amount, it.currency, targetCurrency,
              exchangeResult!!.rates!!)
        }

    return Pair(incomingSum.toFloat(), outgoingSum.toFloat())
}

private fun convertAmount(amount: Double, fromCurrency: Currency,
  toCurrency: Currency, rates: Rates): Double {
    val rateToBase = rates.getRate(fromCurrency) ?: 1.0
    val rateFromBase = rates.getRate(toCurrency) ?: 1.0
    return amount / rateToBase * rateFromBase
}
```