

# Relazione progetto

## Table of content

<b>Table of content</b>	<b>1</b>
<b>Introduzione</b>	<b>3</b>
Scopo	3
Definizioni	3
<b>Applicazione Android nativa</b>	<b>3</b>
Requisiti	3
Requisiti funzionali	3
Requisiti versione Android	4
Architettura	5
Architettura singolo fragment	6
Database relazionale	8
data class	8
Il database	9
I type converters	9
I Data access objects (DAO)	9
Il DbRepository	11
I Grafici	12
Il GraphRepository	12
Il GraphBuilder	13
Generazione e renderizzazione dei grafici	14
User interface	15
Dashboard	15
Archivio	16
Pagina aggregato singolo	16
Pagina aggiunta/modifica aggregato	17
Pagina resoconti grafici	17
Sviluppo	18
Struttura della dashboard	18
Recycler View con drag and drop	20
ContentProvider	21
Paging delle immagini dalla galleria	22
Visualizzazione immagini con placeholder	24
Gestione dinamica della richiesta dei permessi	24

Motion Layout	25
Logging	28
Testing	28
Struttura dei test	28
Test del database	29
Test del dbRepository	30
<b>Applicazione cross-platform Flutter</b>	<b>31</b>
Requisiti	31
Requisiti funzionali	31
Requisiti di versione Android	31
Architettura	32
Gestione delle schermate	32
Database relazionale	34
Le data class	34
Il database	35
Il db repository	37
Schermata di aggiunta di un aggregato	38
I grafici	40
Gli oggetti DataSeries	40
Il Charts Builder	40
Il simpleBarChart	41
Il caricamento e la visualizzazione	42
User interface	42
dashboard	42
Pagina aggiunta aggregato	43
Archivio	43
Grafici	44
Sviluppo	44
Problematiche e strategie adottate	44
Future builder	45
Gestione back button	46
Testing	47
Conclusione	47

# Introduzione

## Scopo

Realizzare un'app che permette di tenere traccia di ogni singola spesa che si effettua giornalmente, permettendo all'utente di aggiungere degli scontrini con annesso le sotto voci.

Da tutti questi dati è possibile poi realizzare grafici riassuntivi che evidenziano le abitudini di spesa dell'utente. Per rendere il tutto più utilizzabile la pagina principale dell'app è una dashboard in cui si può personalizzare cosa avere in primo piano.

## Definizioni

Terminologia utilizzata:

- Aggregato: un intero scontrino, una spesa che può essere decomposta in altre spese, ad esempio il latte sarà un elemento dell'aggregato spesa.
- Elemento: il singolo oggetto che è stato acquistato, o la singola spesa.

# Applicazione Android nativa

## Requisiti

### Requisiti funzionali

- Possibilità di aggiungere una spesa al database attraverso un'apposita schermata raggruppando le singole spese in aggregati, i quali devono permettere il salvataggio della data, e l'associazione con un tag che identifica la tipologia di spesa, in oltre tale aggregato dovrà presentare la possibilità di allegare un'immagine dalla galleria, dalla fotocamera, o un pdf.
- Visualizzare tutti gli scontrini inseriti, attraverso una schermata archivio, che ne consenta anche il filtraggio per data e per tag.
- Ottenere un resoconto delle spese fatte, attraverso una pagina che presenti dei grafici che riassumono in maniera semplice e intuitiva la distribuzione delle spese fatte. In tale pagina saranno presenti 8 grafici:
  - Grafico a barre che rappresenti le spese giornaliere del mese corrente.
  - Grafico a barre che rappresenti le spese mensili dell'anno corrente.
  - Grafico a barre che rappresenti le spese divise per tag degli aggregati nel mese corrente.
  - Grafico a barre che rappresenti le spese divise per tag degli aggregati nell'anno corrente.
  - Grafico a barre che rappresenti le spese divise per tag degli elementi nel mese corrente.

- Grafico a barre che rappresenti le spese divise per tag degli elementi nell'anno corrente.
- Grafico a torta che contiene il conteggio del numero di aggregati divisi per tag nel mese corrente.
- Grafico a torta che contiene il conteggio del numero di elementi divisi per tag nel mese corrente.
- Possibilità di personalizzare la dashboard scegliendo quali widget e in quale ordine li si vuole.
- Avere a disposizione le seguenti pagine in cui distribuire le funzionalità descritte:
  - Dashboard: pagina principale con widget personalizzabili
  - Archivio: pagina di visualizzazione di tutti gli aggregati dove è possibile effettuare delle ricerche per tag e per data.
  - AddPage: pagina dove è possibile effettuare l'inserimento dell'aggregato e dei suoi elementi
  - EditPage: pagina di modifica dell'aggregato e dei suoi elementi.
  - GraphPage: pagina dove vengono resi disponibili i riassunti grafici delle spese salvate nel database.

## Requisiti versione Android

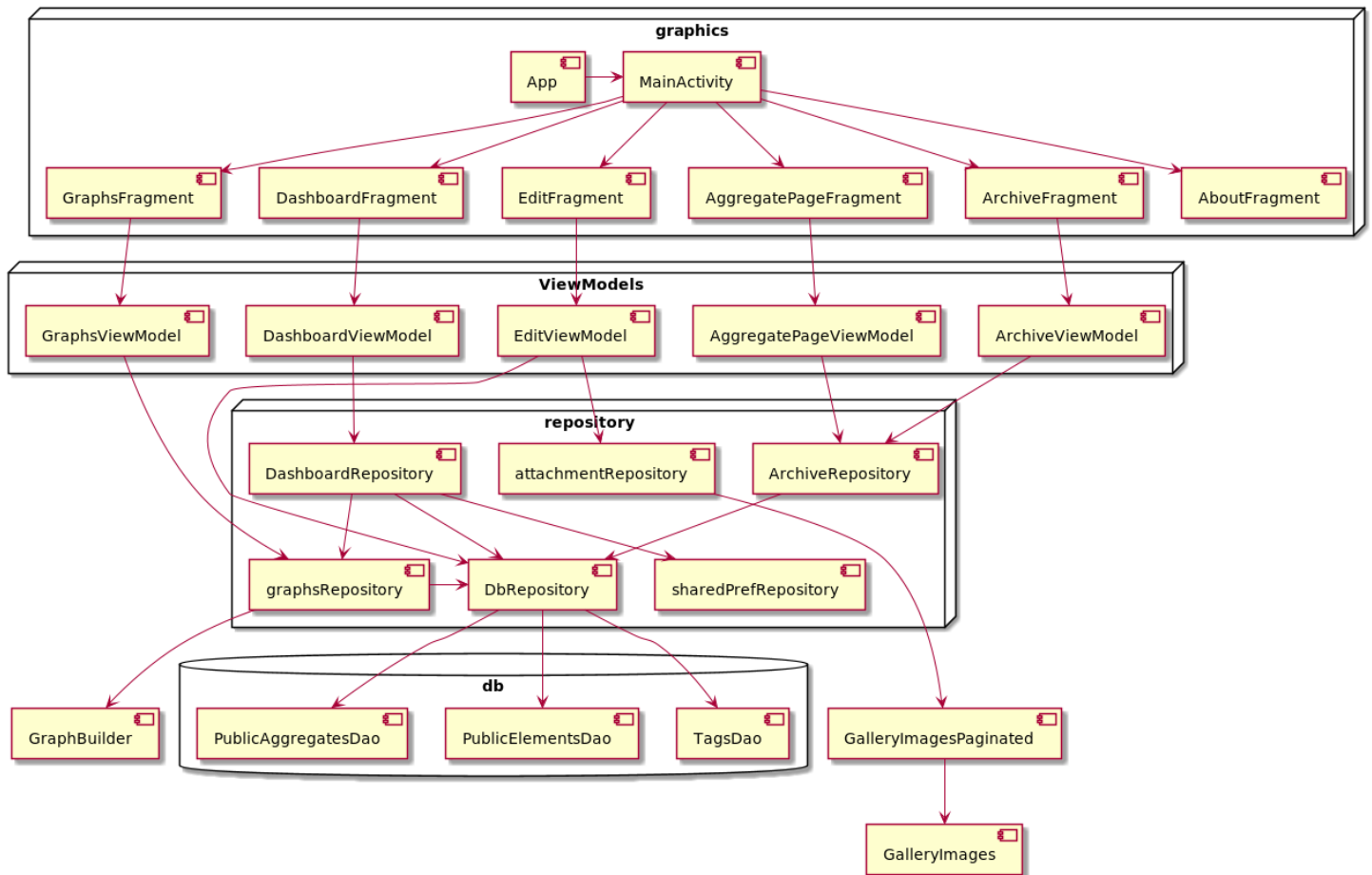
Per lo sviluppo dell'app si è scelto di optare per il supporto dell'api 25 avendo a disposizione una platea di dispositivi di circa il 66.2% del totale. Le funzionalità sono state testate soltanto con Android 10 e 9, dunque sono supportati ma non sono stati testati Android 7.1, 8.0 e 8.1.

## Architettura

Per realizzare l'app abbiamo utilizzato il modello MVVM, separando quindi i layer d'interfaccia grafica con quelli logici. Ogni vista che deve gestire della logica ha il suo view model.

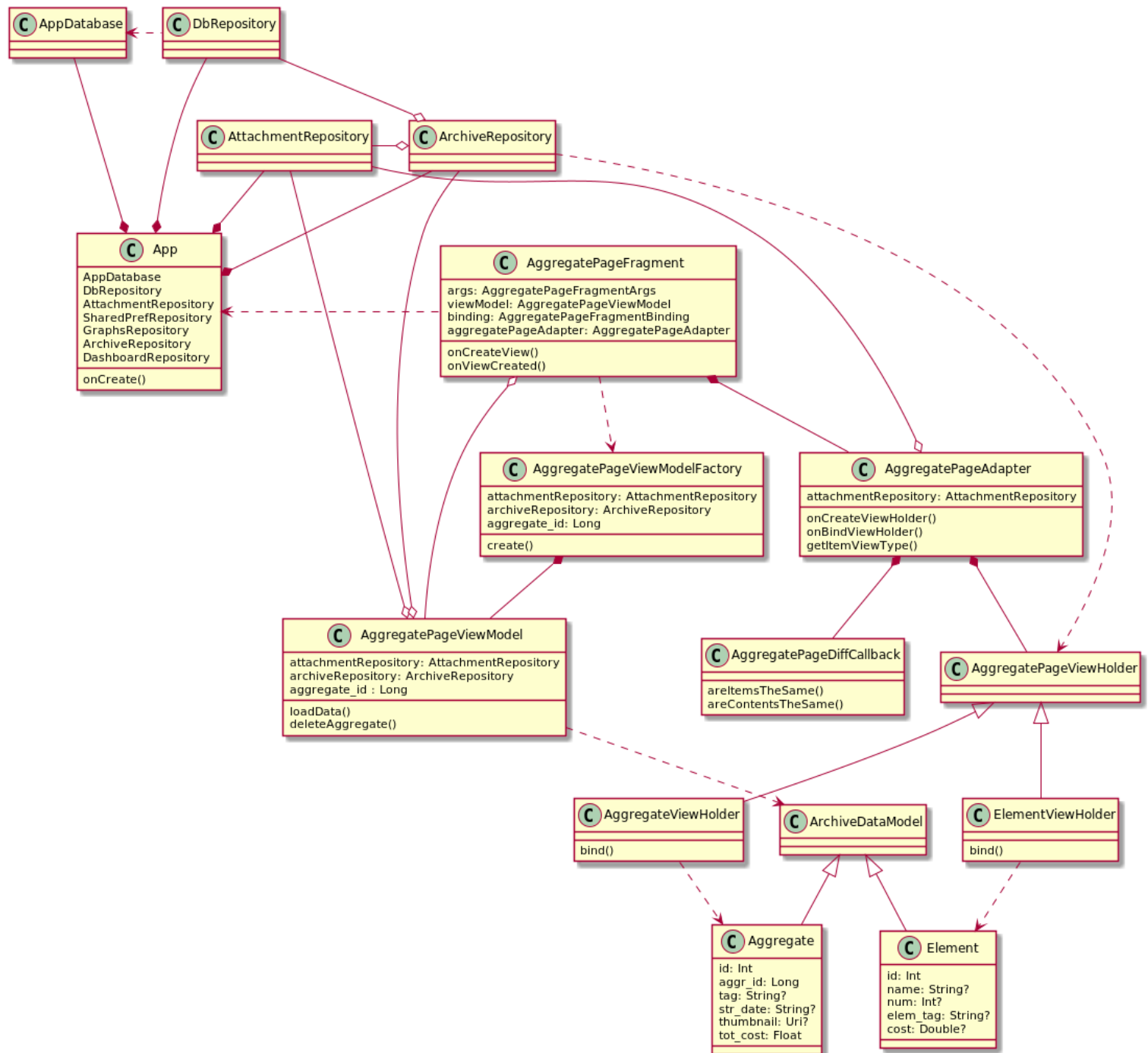
Abbiamo inoltre usato l'approccio di app con singola activity, quindi la MainActivity contiene solo gli elementi comuni a tutte le viste, principalmente la bottom app bar. Mentre tutto il resto dei componenti grafici risiede nei propri fragment.

Di seguito è mostrato lo schema riassuntivo delle componenti principali dell'app, che illustrano l'implementazione del modello MVVM:



## Architettura singolo fragment

Di seguito viene riportato lo schema UML del fragment `AggregatePageFragment`, includendo tutte le connessioni verso le altre parti dell'app.



Come si vede dallo schema,

- Il database e i repository vengono inizializzati all'interno dell'app, in modo lazy.
- Alcuni repository hanno un riferimento ad altri repository, ciò è necessario per specializzare i metodi in base alle necessità dei singoli fragment.
- Il fragment non conserva nessun riferimento ai repository, sono solo passati al viewModel attraverso il viewModelFactory.
- Il view model conserva lo stato, un live data di tipo `List<ArchiveDataModel>`, che viene poi osservato nel fragment.
- Nell'observer si passa la lista all'adapter, composto dal view holder e dalla diffCallback.
- Dato che si devono visualizzare 2 tipi di dati, Aggregate ed Element, è necessario avere un dataModel che abbia 2 classi figlie e un viewholder che abbia una struttura simile. Si hanno infatti 2 ViewHolder, `AggregateViewHolder` ed `ElementViewHolder`, figli di `AggregatePageViewHolder`.

## Database relazionale

Per il database abbiamo scelto di utilizzare la libreria Room, trattata ampiamente nel corso, che ha velocizzato molto lo sviluppo del database grazie al suo paradigma di generazione automatica del codice associato alle operazioni d'input/output con il database e alla possibilità di avere un feedback durante la scrittura delle query in raw.

### data class

Per la costruzione del database sono state utilizzate 3 data class che corrispondono a 3 tabelle contenute nel database, rispettivamente: Aggregate, Element e Tag:

```
@Entity(
    tableName = "aggregate",
    indices = [Index("id")]
)
data class Aggregate @JvmOverloads
constructor(
    @PrimaryKey(autoGenerate = true)
    var id: Long? = null,
    var tag_id: Long? = null,
    var date: Date? = null,
    var location: Location? = null,
    var attachment: Uri? = null,
    var total_cost: Float = 0.0f,
){
    @Ignore
    var tag: String? = null
}

@Entity(
    tableName = "element",
    foreignKeys = [
        ForeignKey(
            entity = Aggregate::class,
            parentColumns = ["id"],
            childColumns = ["aggregate_id"]
        )
    ],
    indices = [Index("elem_id")]
)
data class Element @JvmOverloads
constructor(
    @PrimaryKey(autoGenerate = true)
    var elem_id: Long? = null,
    var aggregate_id: Long? = null,
    var name: String? = null,
    var num: Long = 0L,
    var parent_tag_id: Long? = null,
    var elem_tag_id: Long? = null,
    var cost: Float = 0.0f,
){
    @Ignore
    var parent_tag: String? = null
    @Ignore
    var elem_tag: String? = null
}

@Entity(tableName = "tag")
data class Tag(
    @PrimaryKey(autoGenerate = true)
    var tag_id: Long? = null,
    var tag_name: String? = null,
    var aggregate: Boolean? = null
)
```

Room necessita per la generazione delle tabelle dell'annotazione `@Entity` sulle specifiche data class, le quali dicono alla libreria come generare il codice necessario per la generazione delle tabelle del database. L'annotazione Entity può essere corredata di attributi che indicano quale delle variabili della data class (le quali corrispondono ad ogni colonna della tabella) sono delle chiavi.

Attraverso l'attributo `ForeignKey` possono essere specificate delle relazioni tra le colonne di più tabelle diverse, ad esempio nel database di questa app è stata stabilita una relazione tra la colonna `id` della tabella `Aggregato` e la colonna `aggregate_id` della tabella `Element`, tale relazione previene la creazione di un record `element` con un valore `aggregate_id` che non corrisponde a nessun aggregato esistente nel database.



NOTE: Sono state riscontrate delle incongruenze con la documentazione per quanto riguarda l'utilizzo dell'annotazione `@Ignore`, la quale non può essere utilizzata nel costruttore principale e deve essere associata ad una variabile della data class presente nel corpo in presenza della annotazione `@JvmOverloads constructor`.

## Il database

Per la creazione del database deve essere creata una classe astratta che estende la classe `RoomDatabase`, anche in questo caso sono necessarie delle annotazioni che specificano all'oggetto database a quali Entity deve fare riferimento attraverso la reflection:

```
@Database(entities = [Aggregate::class, Element::class, Tag::class], version = 1, exportSchema = false)
```

L'annotazione `@Database` inoltre consente di specificare quale è la versione del database, infatti in eventuali aggiornamenti dell'app in cui la struttura del database cambia è necessario esplicitare una versione diversa del database o questo darà dei conflitti legati alla struttura del database che è salvato nella memoria del dispositivo.

## I type converters

Nell'oggetto database con una ulteriore annotazione è possibile associare un oggetto `TypeConverters` il quale è estremamente utile nel caso in cui il database deve utilizzare delle tipologie di dato non supportate direttamente da SQLite:

```
@TypeConverters(Converters::class)
```

ad esempio il tipo `Date`, non è supportato direttamente, ma attraverso la definizione di due metodi della classe passata tramite reflection all'annotazione

`@TypeConverters` è possibile effettuare una conversione ad un tipo di dato equivalente, sarà room che automaticamente trovando un tipo di dato non supportato in una data class andrà a cercare nell'oggetto converters un metodo che presenta un argomento del tipo non supportato e ha come tipo di ritorno uno supportato, lo stesso per l'operazione di lettura con il metodo inverso.

In questo caso il tipo `Date` in scrittura viene convertito nel tipo `long`, salvando il suo corrispondente tempo unix, al contrario in lettura il tempo unix nel db verrà usato per la generazione di un oggetto `Date`.

## I Data access objects (DAO)

I Dao sono delle interfacce opportunamente annotate con l'annotazione `@Dao` utilizzate da room come prototipi per l'implementazione dei metodi di scrittura, lettura update ed eliminazione.

I prototipi di metodi con una opportuna annotazione saranno implementati automaticamente da room a tempo di compilazione, le annotazioni che richiamano questo comportamento sono:

`@Insert`, `@Update`, `@Delete`, `@Query`.

possiamo vedere di seguito un esempio:

```
@Insert(onConflict = OnConflictStrategy.REPLACE)
suspend fun _insertElement(element: Element): Long

@Update
suspend fun _updateElement(element: Element): Int

@Delete
suspend fun _deleteElement(element: Element): Int

@Query("SELECT SUM(element.cost * element.num) FROM element WHERE element.elem_tag_id = :elem_tag_id")
suspend fun _countAllExpensesByElementTagId(elem_tag_id: Long): Float
```

Room determinerà come dovrà implementare la funzione in base agli argomenti, al tipo di valore di ritorno e in base all'annotazione. I tipi più semplici sono ovviamente i primi tre mentre l'ultimo, Query, lascia più libertà permettendo di esprimere una query più complessa al quale inoltre è possibile passare degli argomenti da inserire nella query SQL.

Oltre a queste 4 tipologie di annotazione, ne è stata utilizzata un'altra la @Transaction, la quale consente di effettuare una serie di operazioni sul database in modo sicuro, ovvero effettuando il ripristino di tutte le operazioni svolte durante la procedura se si verificano degli errori nelle chiamate o se la procedura viene interrotta.

Nell'implementazione di questa app i Dao sono stati strutturati con un sistema gerarchico, troviamo infatti che le varie interfacce partono da quella più basilare, i Tag, e vengono implementati via via da interfacce di più alto livello che tengono conto della gestione delle interconnessioni tra i vari elementi.

Eccetto che per i tag che contano un solo dao, la Entity Element e Aggregate contano rispettivamente 3 interfacce che si implementano consecutivamente:

- BaseAggregatesDao -> AggregatesDao -> PublicAggregatesDao
- BaseElementsDao -> AggregatesDao -> PublicAggregatesDao

Nell'interfaccia Base\* sono stati inseriti i metodi basilari che sfruttano principalmente le annotazioni di Room senza implementare il corpo delle funzioni, mentre nelle successive sono presenti quasi esclusivamente dei metodi Transaction che si occupano della gestione delle interconnessioni tra i vari elementi, ad esempio della gestione del tag che accompagna un nuovo aggregato, effettuando il controllo della sua esistenza e l'aggiunta nel caso questo non sia presente e il contrario per l'operazione di eliminazione.

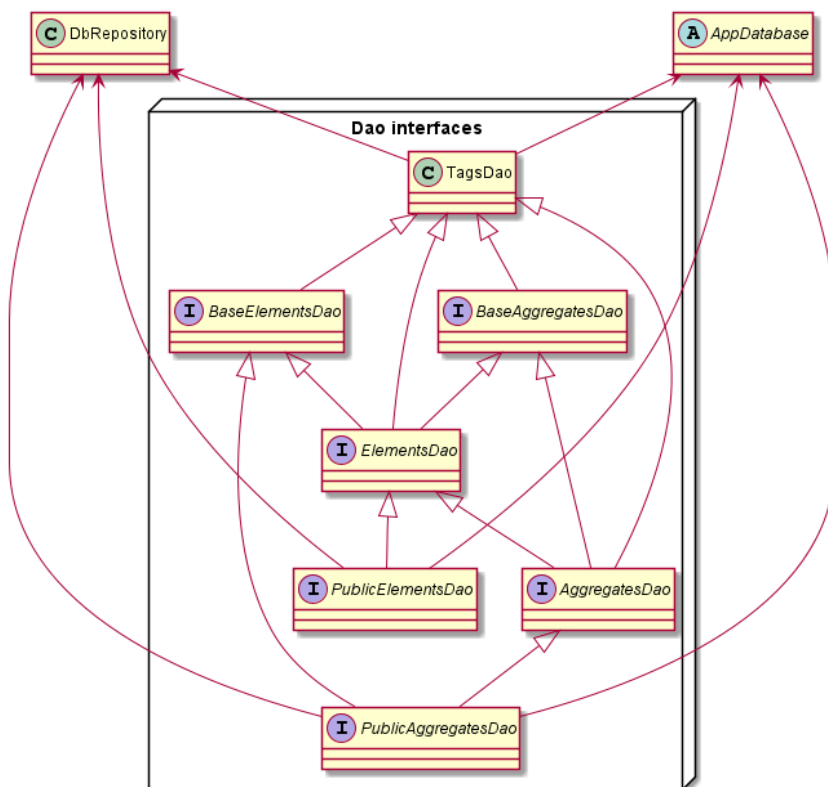
Si noti infine che l'unica differenza tra le interfacce Public e le altre è che nelle prime i metodi sono pensati per poter essere utilizzati all'esterno dei dao, dunque esenti da qualunque tipo di operazione di gestione. I metodi delle interfacce di secondo e primo livello sono pensati per essere utilizzati internamente, con qualche eccezione

per metodi per uso esterno che vengono utilizzati anche internamente da più interfacce.

NOTA: I metodi che non possono essere usati esternamente iniziano con un trattino basso, ad indicare la finalità di uso interno dal momento che nell'interfaccia, Room non accetta la creazione di metodi privati i quali non potranno essere implementati dalla libreria.

## II DbRepository

Il repository del database funge da ulteriore layer di astrazione tra il database e il backend dell'applicazione. Mentre la logica relazionale delle tabelle è gestita all'interno dei dao, nel repository del database vengono resi disponibili i metodi del database wrappati da opportune funzioni che semplificano la traduzione di alcune strutture dati a quelle del database, in oltre nel repository viene effettuata la generazione degli intervalli temporali necessari per la richiesta dei dati a partire da dei valori enum, come month or year.



Rappresentazione delle relazioni tra le varie interfacce che compongono il dao del database.

## I Grafici

### Il GraphRepository

Il graph repository è utilizzato per la generazione degli oggetti della libreria che costruiscono i grafici. Ogni grafico presente nell'app ha un suo metodo per la generazione, prendiamo ad esempio il metodo che genera il grafico delle spese mensili divise per giorno:

```
suspend fun monthExpensesHistogram(): AChartModel{  
  
    val values =  
TypesHelper.float2DoubleArray(dbRepository.getPeriodExpenses(DbRepository.Period.  
.MONTH))  
    values.mapInPlace { n -> n.round(2) }  
  
    return graphBuilder.category_graph(  
        categories = graphBuilder.generate_month_labels(),  
        y_axis_name = getStrings(R.string.expenses),  
        values_name = getStrings(R.string.expense),  
        values = values  
    )  
}
```

i metodi del graph repository richiamano i dati necessari dal database tramite il database repository e li passano al metodo category\_garph() dell'oggetto graphBuilder il quale costruisce l'oggetto effettivo che determina le caratteristiche del grafico.

Il GraphBuilder si occupa della generazione degli oggetti grafici tramite la libreria AChartCore-Kotlin, la quale per la renderizzazione di un grafico necessita della creazione di un oggetto AChartModel e del passaggio di questo ad un oggetto del layout, simile ad un canvas, `AChartView`.

Tramite il metodo aa\_drawChartWithChartModel() è possibile passare il modello del grafico al canvas che ne effettua il rendering.

## Il GraphBuilder

Analizziamo a questo punto la struttura dell'oggetto GraphBuilder, il quale contiene i metodi per la generazione degli istogrammi e per la generazione dei grafici a torta. Vediamo qui in seguito il metodo per la generazione degli istogrammi:

```
fun category_graph(  
    categories: Array<String>,  
    values_name: String = "series 1",  
    y_axis_name: String = "series",  
    values: Array<Double>,  
): AChartModel {  
  
    val stopsArr: Array<Any> = arrayOf(  
        arrayOf(0.00, "#FFFFFF"),  
        arrayOf(0.40, "#6200ee"),  
        arrayOf(1.00, "#6200ee")  
    )  
  
    val linearGradientColor = AAGradientColor.LinearGradient(  
        AALinearGradientDirection.ToTop,  
        stopsArr  
    )  
  
    return AChartModel.Builder(context)  
        .setChartType(AChartType.Column)  
        .setBackgroundColor("#ffffff")  
        .setDataLabelsEnabled(false)  
        .setYAxisGridLineWidth(0f)  
        .setYAxisTitle(y_axis_name)  
        .setYAxisAllowDecimals(false)  
        .setLegendEnabled(false)  
        .setTouchEventEnabled(true)  
        .setCategories(*categories)  
        .setSeries(  
            AASeriesElement()  
                .name(values_name)  
                .data(values as Array<Any>)  
                .color(linearGradientColor)  
        )  
        .build()  
}
```

Il metodo richiama un oggetto della libreria AAGradientColor.linearGradient() il quale serve per definire un gradiente di colore da applicare al grafico, il quale può essere definito tramite una array di array di double e string.

Infine il fulcro del metodo è la generazione effettiva dell'oggetto AChartModel, il quale attraverso il metodo Builder() richiama un oggetto configurabile tramite la chiamata a cascata di una serie di metodi. Per settare ad esempio il tipo di grafico si utilizza il metodo `setChartType(AChartType.Column)`, il quale specifica che il grafico da

generare è un istogramma, `.setBackgroundColor("#ffffff")`, setta il colore dello sfondo, mentre `.setCategories(*categories)`, setta i label delle colonne e in fine `setSeries()` imposta il nome dei valori singoli, il valore di ogni colonna, che è esplicitamente richiesto come Array di Any e il colore che devono avere le colonne, in questo caso un gradiente di colore. Attraverso la chiamata del metodo `build` si genera l'oggetto, che a questo punto non è più modificabile e può essere passato all'oggetto del layout `AAChartView`.

## Generazione e renderizzazione dei grafici

I grafici sono generati all'interno di una coroutine che viene eseguita all'interno del thread di IO tramite il seguente comando all'interno del view model:

```
viewModelScope.launch(Dispatchers.IO) {
    _rvList.postValue(
        listOf(
            GraphsDataModel.Histogram(
                id = 0,
                name =
graphsRepository.getStrings(R.string.histo_month_expenses),
                aaChartModel = graphsRepository.monthExpensesHistogram()
            ),
            ...
        )
    )
}
```

Tutti i grafici vengono generati e caricati all'interno di una lista di `graphsDataModel`, i quali vengono caricati alla fine dell'esecuzione della funzione nel `liveData _rvList` grazie al metodo `postValue()` che permette il caricamento dei dati in un live Data che risiede in un thread diverso.

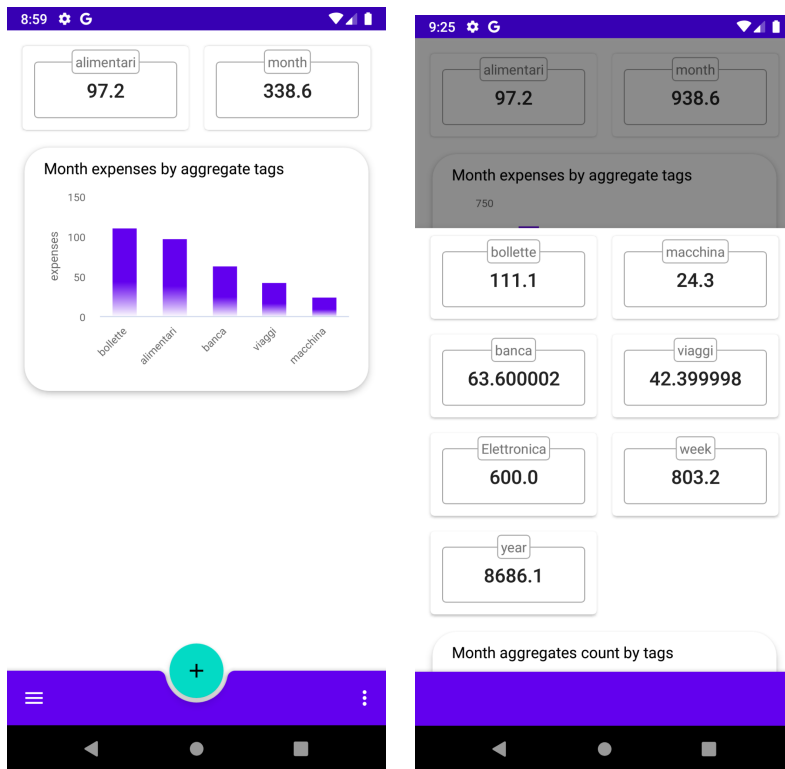
Una volta che la lista di oggetti viene aggiornata sarà opportunamente gestita dalla `RecyclerView` della pagina dei grafici, la quale durante il binding dell'oggetto `GraphsDataModel` effettuerà il caricamento del grafico nel `AAChartView` come mostrato di seguito nella funzione binding dell' `HistogramViewHolder`:

```
fun bind(histogram: GraphsDataModel.Histogram) {
    with(binding)
    {
        textView.text = histogram.name
        aaChartView.aa_drawChartWithChartModel(histogram.aaChartModel)
    }
}
```

## User interface

L'app come mostrato nella sezione dell'architettura è composta da 6 fragment i quali sono illustrati di seguito con una breve descrizione di ciò che è visibile lato utente.

### Dashboard



La dashboard è la prima schermata che l'utente vede quando apre l'app. Per renderla personalizzabile dall'utente abbiamo realizzato 3 modalità di funzionamento:

- normal mode
- edit mode
- store mode

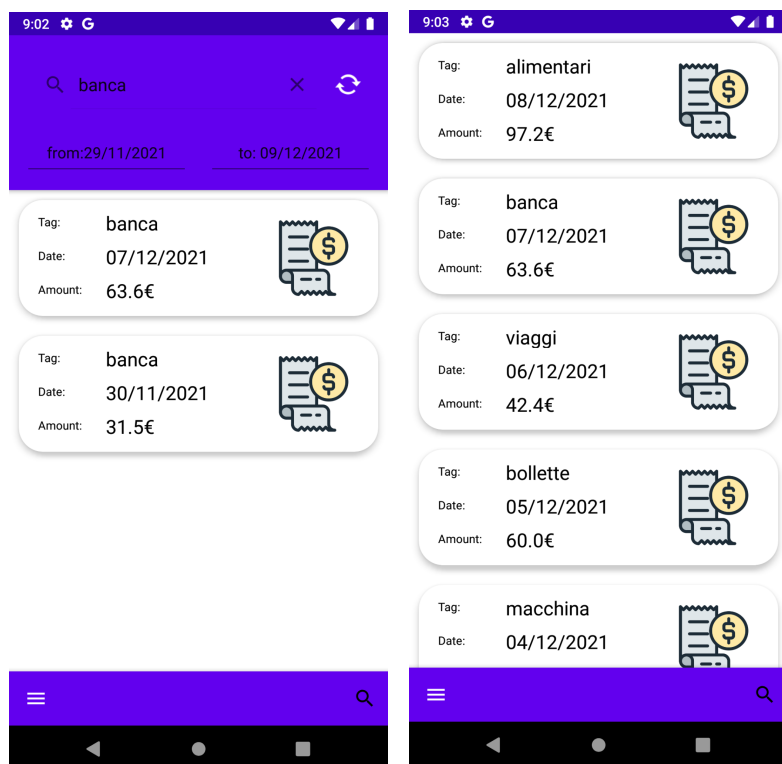
Nella prima immagine si può vedere un esempio di normal mode, si vede infatti una serie di widget scelti e ordinati dall'utente secondo le sue preferenze.

Per consentire la personalizzazione da parte dell'utente si deve passare in edit mode, con un long click

su uno degli widgets. A questo punto si possono riordinare gli widgets attraverso un drag and drop e si può inoltre accedere allo store attraverso un bottone sulla bottom app bar. Inoltre per eliminare un widget che non si vuole più basta uno swipe. Nella seconda immagine si vede la dashboard con lo store aperto, per aggiungere un widget dallo store alla dashboard è sufficiente un click sul widget scelto.

Nella prima immagine è mostrata la dashboard in modalità normale settata con due widget label, i quali illustrano la somma delle spese alimentari di questo mese, mentre l'altro il totale delle spese di questo mese, e di seguito un grafico che illustra le spese divise per aggregato, di questo mese.

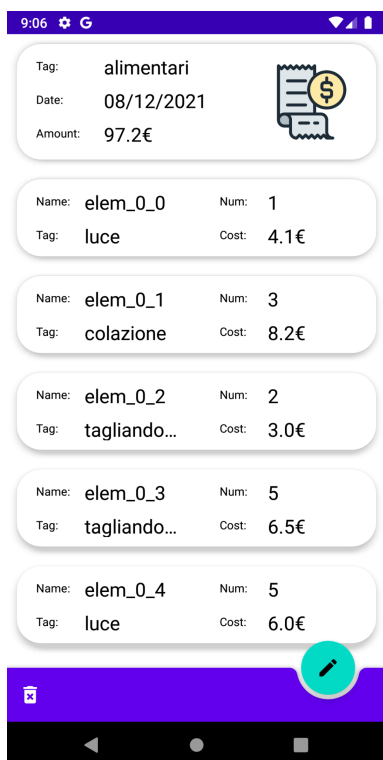
## Archivio



L'archivio permette di visualizzare tutti gli aggregati presenti all'interno del database. Come si vede nella prima immagine premendo il tasto cerca in basso a sinistra si apre una Collapsing Toolbar, la quale permette di filtrare gli aggregati per tag, scrivendo viene illustrata la lista di nomi dei tag compatibili con il testo inserito per facilitare la ricerca. Inoltre grazie agli ulteriori due label è possibile settare la data d'inizio e fine dell'intervallo di interesse in cui effettuare la ricerca. Nella barra è presente inoltre un tasto di reset della ricerca che riporta i valori dei

filtri allo stato di default e con essi anche gli elementi mostrati nella recyclerview. Si noti inoltre che nel caso in cui con l'aggregato sia stata inserita un'immagine questa verrà visualizzata nella card dell'aggregato al posto dell'immagine di default.

## Pagina aggregato singolo



Quando ci si trova nell'archivio è possibile cliccare un qualunque aggregato per aprire una seconda pagina che mostra oltre ai valori dell'aggregato anche i suoi elementi. In tale pagina è possibile richiedere l'eliminazione dell'aggregato e dei suoi elementi o è possibile entrare nella modalità modifica. Cliccando il fab in basso a destra verrà aperta la pagina di edit, che mostrerà le stesse informazioni in una interfaccia che ne permetterà la modifica. In questa schermata oltre a ciò che viene illustrato nell'archivio è possibile vedere ogni elemento associato all'aggregato con il nome associato, il suo tag elemento, il multiplo di quell'elemento e infine il costo unitario di tale elemento.



## Pagina aggiunta/modifica aggregato

9:11 G

← edit\_fragment

tag Elettronica

date 08/12/2021

name nuova tv num 1

tag tv cost 600

name num

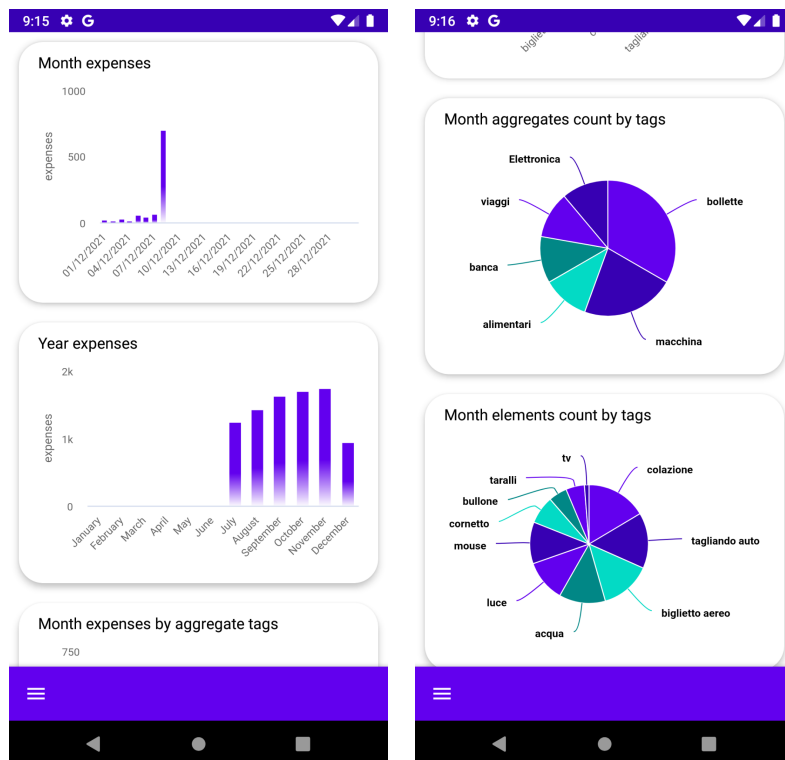
tag cost

✓

Questa schermata viene utilizzata con un layout identico per l'update e per l'add ma ovviamente viene richiamata con due backend diversi, infatti in un caso effettuerà l'aggiunta nel database e ritornerà alla dashboard mentre nel caso in cui venga richiamata per la modifica di un aggregato questa ritornerà nella pagina dell'aggregato appena modificato, attraverso il clic del fab che confermerà la modifica dell'elemento.

Con il bottone sulla bottom app bar si apre il pannello per allegare una foto dalla galleria, dalla fotocamera oppure per allegare un pdf.

## Pagina resoconti grafici



La pagina dei grafici non ha particolari funzioni interattive, gli 8 grafici sono visualizzati su di una recyclerview, con i dati che vengono caricati all'apertura della pagina. Oltre alla visualizzazione è possibile cliccare le colonne per ottenere un label con informazioni più chiare, che descriva il valore della spesa effettuata in tale giorno. Cliccando su uno spicchio del grafico a torta viene illustrato tramite un label il numero degli elementi o degli aggregati associati a quel tag.

## Sviluppo

Per lo sviluppo abbiamo seguito il modello MVVM, includendo i layer repository che in alcuni casi è stato necessario stratificare ulteriormente per riutilizzare dei metodi di accesso a diverse risorse necessarie in più schermate.

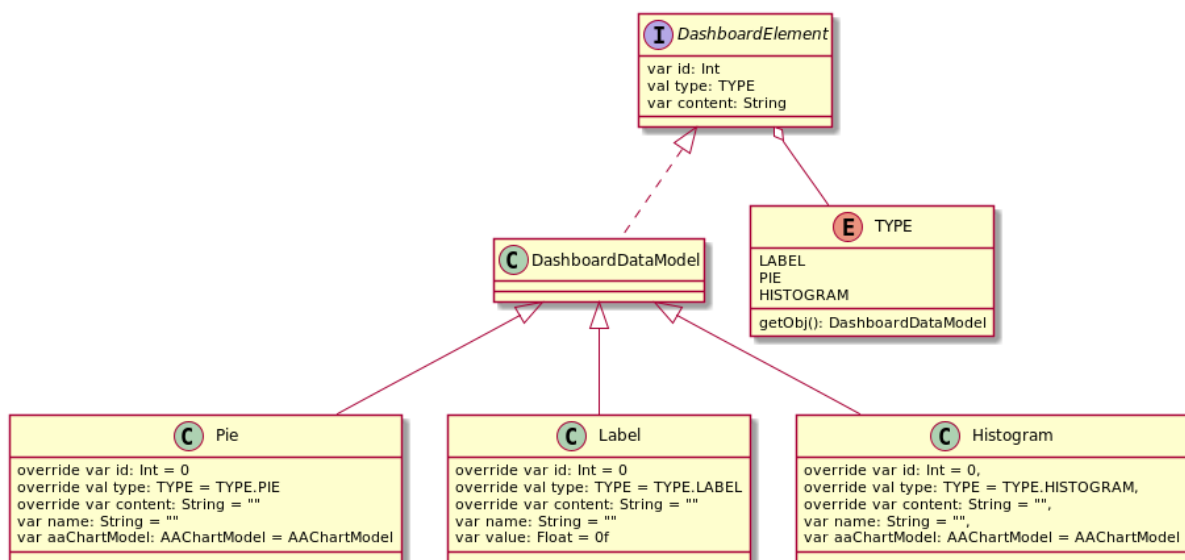
### Struttura della dashboard

La dashboard è composta da un RecyclerView con uno StaggeredGridLayoutManager, ciò permette di avere elementi di dimensione diversa e gap tra elementi.

Il recyclerView della dashboard è in grado di ospitare multipli tipi di viewholder, in questo momento sono implementati:

- LabelViewHolder -> Label
- CakeCardViewHolder -> Pie
- HistogramViewHolder -> Histogram

Ogni viewholder fa riferimento a uno specifico tipo di oggetto. Ciò è realizzato facendo in modo che tutti gli oggetti che si vogliono visualizzare sulla dashboard facciano parte di una sealed class: DashboardDataModel



Per rendere la dashboard permanente è stato necessario rendere i suoi elementi salvabili nel disco, nello specifico nelle shared preferences, ma si nota subito che ogni oggetto ha dei campi specifici e generati a partire dai dati contenuti nel database. È stato quindi necessario creare un'interfaccia comune a tutti gli elementi della dashboard, che contenesse le informazioni minime per ricreare lo stesso oggetto al successivo avvio dell'app: DashboardElement.

In questo modo tutti i tipi di oggetti che sono presenti sulla dashboard possono essere scritti su disco allo stesso modo, indipendentemente da quali campi specifici possiedono.

I campi di DashboardElement sono:

- id: l'id che l'elemento ha nella recyclerview
- type: il tipo di questo widget
- content: una stringa contenente cosa va effettivamente visualizzato in quel widget. Ad esempio se nel db è presente il tag "auto", una stringa content valida potrebbe essere: "sumTag:auto:MONTH", che indica di visualizzare la somma del tag auto nel periodo di 1 mese.

Quando un elemento deve essere salvato nelle sharedPreferences:

```
val base = "el_${index}"

putInt("${base}_id", element.id)
putString("${base}_type", element.type.name)
putString("${base}_content", element.content)
```

il tipo viene salvato come stringa. Inoltre per conservare lo stesso ordine gli elementi vengono salvati usando un prefisso dipendente dalla loro posizione nella recyclerview.

Al ricaricamento della dashboard:

```
var element: DashboardElement
val base = "el_${index}"

// i read the values the same way i wrote them
val id = getInt("${base}_id", -1)
val type = getString("${base}_type", "")
val content = getString("${base}_content", "") ?: return null
// to convert the generic DashboardElement to the real type of this object i use
TYPE.getObj()
element = type?.let {
    // if something changed and the values in the shared preferences are not valid
    anymore i just return
    // null and this particular element will be skipped
    kotlin.runCatching {
        TYPE.valueOf(it).getObj()
    }.getOrElse(null)
} ?: return null

// fill the other data into the newly created object
element.id = id
element.content = content
```

Si parte sempre dalla stessa base, ma per specializzare il DashboardElement e farlo diventare un oggetto del tipo dettato dal campo type usiamo il metodo getObj(). In questo modo l'oggetto è già pronto per essere riempito con i campi specifici del tipo e visualizzato.

La dashboard è strutturata per essere robusta alle variazioni di codice. Quindi se in un futuro aggiornamento dell'app un widget viene rimosso / rinominato, al

caricamento viene ignorato. Inoltre non appena l'utente modifica la dashboard tutte le sharedPreferences relative alla dashboard vengono sovrascritte.

## Recycler View con drag and drop

Per realizzare il drag and drop delle tiles nella dashboard abbiamo usato:

```
ItemTouchHelper(DragManager(viewModel)).attachToRecyclerView(binding.recyclerViewDashboard)
```

dove DragManager e' una classe figlia di ItemTouchHelper.Callback().

In questo modo si hanno a disposizione diverse callback per gestire il drag and drop e lo swipe degli elementi della recyclerView.

Per settare come gli elementi della dashboard possono essere spostati si deve usare il metodo:

```
override fun getMovementFlags(recyclerView: RecyclerView, viewHolder: RecyclerView.ViewHolder): Int
{
    val dragFlags = ItemTouchHelper.UP or ItemTouchHelper.DOWN or ItemTouchHelper.LEFT or ItemTouchHelper.RIGHT
    val swipeFlags = ItemTouchHelper.START or ItemTouchHelper.END

    return makeMovementFlags(dragFlags, swipeFlags)
}
```

nel nostro caso ogni tipo di movimento e' possibile quindi abbiamo abilitato tutte le direzioni per il drag, e tutte le direzioni per lo swipe.

Dopodiche basta implementare i metodi onMove e onSwiped:

```
override fun onMove(
    recyclerView: RecyclerView,
    viewHolder: RecyclerView.ViewHolder,
    target: RecyclerView.ViewHolder
): Boolean
{
    val from = viewHolder.bindingAdapterPosition
    val to = target.bindingAdapterPosition

    viewModel.swapItems(from, to)
    return true
}

override fun onSwiped(viewHolder: RecyclerView.ViewHolder, direction: Int)
{
    // when the user swipe an element, we delete the element
    viewModel.removeItemFromDashboard(viewHolder.absoluteAdapterPosition)
}
```

In questo modo si rende possibile il drag and drop e lo swipe di elementi di una recyclerView.

## ContentProvider

Per caricare le immagini della galleria renderle disponibili all'intento della nostra app abbiamo usato la suite dei content providers.

Un content provider fornisce dai dati alle applicazioni esterne come delle tabelle, la logica è molto simile a quella di un database relazionale. Quindi ogni colonna rappresenta una caratteristica dei dati richiesti e ogni riga rappresenta un elemento. In questo modo ci è stato possibile fare una query stile SQL al sistema, ciò restituisce un cursore, nello stesso stile di un database, da cui si possono poi estrarre tutti i dati richiesti.

Di seguito è riportato un estratto del codice necessario per ottenere il cursore della query:

```
val images = MediaStore.Images.Media.getContentUri(MediaStore.VOLUME_EXTERNAL)

val projection: Array<String> = arrayOf(
    // -> needed to get the thumbnail on older versions of SDK
    MediaStore.Images.Media._ID,
    // -> the name of the file
    MediaStore.Images.Media.DISPLAY_NAME,
    // -> i need only images to i need this column to filter
    MediaStore.Images.Media.MIME_TYPE,
    // -> sort descending
    MediaStore.Images.Media.DATE_MODIFIED
)

val selectionArgs = arrayOf(
    "image/jpeg",
    "image/jpg",
    "image/png"
)

val selection = "${MediaStore.Images.Media.MIME_TYPE} IN (?, ?, ?)"

val bundle = Bundle().apply {
    putString(ContentResolver.QUERY_ARG_SQL_SELECTION, selection)
    putStringArray(ContentResolver.QUERY_ARG_SQL_SELECTION_ARGS, selectionArgs)
    putString(ContentResolver.QUERY_ARG_SORT_COLUMNS,
        MediaStore.Images.Media.DATE_MODIFIED)
    putInt(ContentResolver.QUERY_ARG_SORT_DIRECTION,
        ContentResolver.QUERY_SORT_DIRECTION_DESCENDING)
    putInt(ContentResolver.QUERY_ARG_LIMIT, limit)
    putInt(ContentResolver.QUERY_ARG_OFFSET, offset)
}

// and then use the bundle in the query
curr: Cursor? = contentResolver.query(
    images,
    projection,
    bundle,
    null
)
```

Nello specifico la query che è stata effettuata permette di leggere tutte le immagini ( *images* ), chiedendo le colonne specificate da *projection*, quindi l'ID, il nome, il mime-type e la data di ultima modifica.

Possiamo quindi strutturare la query in modo che usi le colonne richieste per filtrare e riordinare il risultato, il sistema ci restituirà infatti una tabella stile SQL con le colonne richieste.

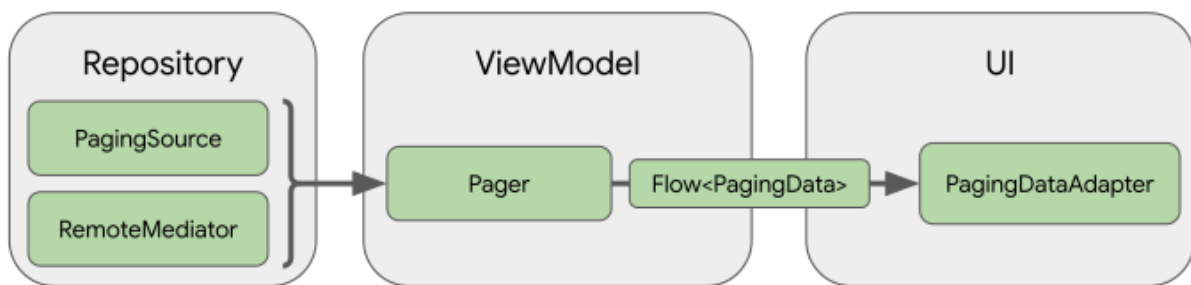
Con *selection* possiamo filtrare solo i mime-type che ci servono: jpeg, jpg e png ( *selectionArgs* ).

Inoltre effettuiamo il sort dei risultati in modo da avere le immagini più recenti in cima ( *QUERY\_ARG\_SORT\_COLUMNS*, *DATE\_MODIFIED* e *QUERY\_ARG\_SORT\_DIRECTION*, *QUERY\_SORT\_DIRECTION\_DESCENDING* ). Abbiamo inoltre aggiunto ulteriori parametri: *offset* e *limit*, che permetteranno poi di effettuare la paginazione dei risultati.

## Paging delle immagini dalla galleria

Dato che un'utente può avere un numero arbitrario d'immagini nella galleria non è possibile pensare di leggerle tutte e fare il display di tutte le immagini in un colpo solo.

Per risolvere questo problema abbiamo fatto ricorso alla “Paging library”:



Nel nostro caso la PagingSource è “GalleryImagesPaginated”, in questa classe si effettua la paginazione di “GalleryImages”, passando *offset* e *limit* come parametri.

Dopodiché la PagingSource viene usata del ViewModel del fragment “Edit” e convertita in un Pager:

```
private val flow = Pager(
    PagingConfig(
        pageSize = 32,
    ),
) { attachmentRepository.galleryImagesPaginated }.flow.cachedIn(viewModelScope)
```

Per evitare che l'operazione di lettura sia effettuata sul thread principale bloccando l'UI, è stato aggiunto un'ulteriore passaggio in cui vengono letti i dati dal pager e trasferiti su di un Flow interno al ViewModel:

```
fun galleryCollect()
{
    _galleryState.value = GalleryDataState.Loading
    viewModelScope.launch {
        withContext(Dispatchers.IO) {
            flow.collectLatest { pagingData ->
                _galleryState.value = GalleryDataState.Data(pagingData)
            }
        }
    }
}
```

In questo modo si cambia contesto della coroutine, passando sul thread di IO. Per visualizzare questi dati si deve implementare “PagingDataAdapter”, sottoclasse di “RecyclerView.Adapter”, che implementa la logica specifica per gestire una sorgente paginata. L'unica differenza sostanziale rispetto a un normale adapter è il modo in cui vengono caricati i dati:

```
viewLifecycleOwner.lifecycleScope.launch {
    viewModel.galleryState.collectLatest { state ->

        when (state)
        {
            is GalleryDataState.Error ->
            {
                Toast.makeText(activity, "there was an error with the
                attachments", Toast.LENGTH_SHORT).show()
                binding.addMotionLayout.transitionToState(R.id.start)
            }
            is GalleryDataState.Data -> galleryAdapter.submitData(state.tasks)
            GalleryDataState.Idle -> { }
            GalleryDataState.Loading -> { }
        }
    }
}
```

Cioè in modo asincrono con una coroutine da lanciare nel fragment che possiede l'adapter.

Questo approccio permette di gestire eventuali errori nel caricamento e nella fase di Loading, ciò è stato in parte implementato ma è comunque riservato a una futura espansione dell'app.

## Visualizzazione immagini con placeholder

La visualizzazione delle immagini della galleria è un'operazione piuttosto onerosa, quindi se l'utente scorre velocemente il PagingDataAdapter potrebbe fare fatica a stare al passo con le richieste delle anteprime. Per evitare che l'UI si blocchi il caricamento delle immagini è effettuato attraverso Glide con l'uso di un placeholder.

```
val size = binding.imageView.width
Glide.with(binding.root.context)
    .load(item.thumbnail)
    .apply(
        RequestOptions
            .centerCropTransform()
            .override(size)
    )
    .apply(
        RequestOptions()
            .placeholder(
                ContextCompat.getDrawable(
                    binding.root.context,
                    R.drawable.ic_baseline_image_24
                )
            )
            .override(size)
            .dontAnimate()
    )
    .into(binding.imageView)
```

Per essere certi che le dimensioni dell'imageView non cambino al caricamento del placeholder è stato fatto l'override delle dimensioni dell'icona alla dimensione dell'imageView.

## Gestione dinamica della richiesta dei permessi

Per gestire più facilmente la richiesta dei permessi è stato creato un oggetto apposito: PermissionsHandling. In questo oggetto è stata wrappata la logica di gestione dei permessi usando il classico registerForActivityResult.

Questo oggetto va inizializzato necessariamente nell'onViewCreated() o comunque in una funzione che sia eseguita all'avvio del fragment.

Dopodiché in qualunque punto del fragment si può eseguire il metodo:

```
setCallbacksAndAsk(
    permissions: Array<String>,
    granted: (() -> Unit)? = null,
    denied: (() -> Unit)? = null
)
```



passando un'array contenente tutti i permessi necessari, es:

```
permissions = arrayOf(  
    Manifest.permission.READ_EXTERNAL_STORAGE,  
    Manifest.permission.ACCESS_MEDIA_LOCATION,  
)
```

una callback da chiamare in caso l'app abbia già i permessi, o l'utente abbia concesso i permessi

```
granted = {  
    ...  
}
```

la seconda callback viene chiamata in caso l'utente non dia tutti i permessi necessari all'app.

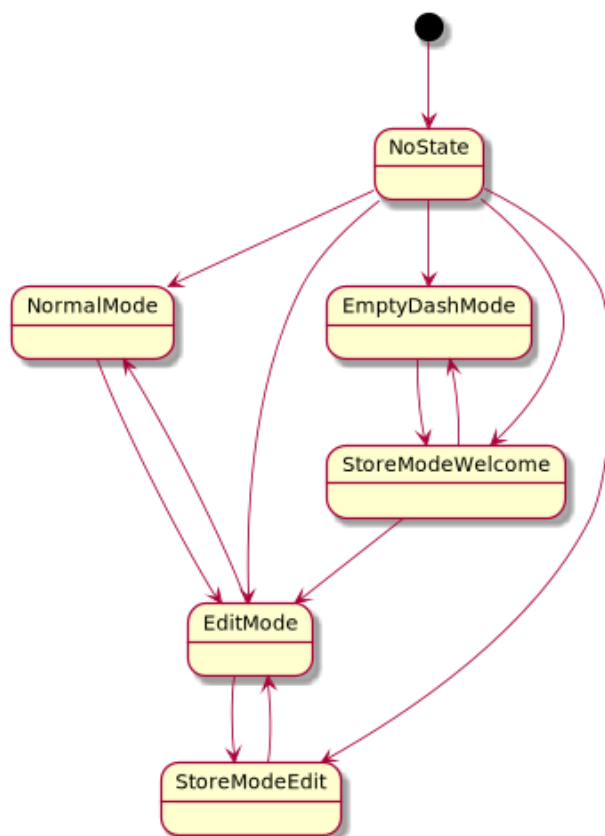
```
denied = {  
    ...  
}
```

## Motion Layout

Per realizzare le animazioni della dashboard e della pagina di Edit abbiamo usato un motion layout. Nella pagina di Edit ci sono solo 2 stati data la semplicità della struttura del fragment, mentre nella Dashboard è stato necessario l'uso di 6 stati. Analizziamo il caso della dashboard, vi sono 6 stati nel MotionLayout e 5 nel codice, 2 stati hanno lo stesso comportamento e cambia solo la grafica da visualizzare. Per quanto sembri semplice da usare è stato comunque necessario considerare tutti i casi possibili e aggiungerli al MotionLayout, lo stato iniziale NoState viene forzato e serve da entry point per gli altri stati:

```
binding.homeMotionLayout.setState(R.id.baseConstraint, -1, -1)
```

Da questo stato si decide poi in quale stato si deve andare, EmptyDashMode o NormalMode. Il collegamento tra NoState e tutti gli altri stati esiste solamente per evitare che quando l'app venga messa in pausa il motion layout perda lo stato corrente e la grafica vada in qualche stato instabile non desiderato.



Dai 2 stati iniziali, NormalState o EmptyDashMode, si procede in base alle interazioni dell'utente con la dashboard. Quindi ad esempio se sono in EmptyDashMode e tocco lo schermo passero' in StoreModeWelcome etc.

Lo stato risiede nel view model e il fragment lo osserva, tutta la logica dei cambiamenti di stato è quindi spostata nel view model. Il fragment è strutturato in modo da non avere nessuno stato, ciò permette che alla distruzione e ricostruzione del fragment l'app dovrebbe riprendere dallo stato in cui era.

Le transizioni sono definite nell'apposito file XML: home\_fragment\_scene.xml

```

<MotionScene xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto">
  <ConstraintSet android:id="@+id/welcomeScreenConstraint" >
    <Constraint
      [...]
    </Constraint>
  </ConstraintSet>
  <Transition
    app:constraintSetStart="@+id/[...]"
    app:constraintSetEnd="@+id/[...]"
    app:duration="@integer/durationJump"/>
</MotionScene>

```

Per modificare la visibilita' degli oggetti senza che venisse animata la transizione [GONE] -> [VISIBLE], che parte dall'angolo in alto a sinistra e viene zoomata fino in

posizione, abbiamo usato una combinazione di trasparenza, cliccabilità e la visibilità stessa.

Quindi una vista che deve essere non visibile e non cliccabile deve avere nel constraint set:

```
<Constraint
    [...]
    android:alpha="0"
    <CustomAttribute
        app:attributeName="clickable"
        app:customBoolean="false" />
    <CustomAttribute
        app:attributeName="visibility"
        app:customIntegerValue="2" />
</Constraint>
```

Da notare come la visibilità deve essere aggiunta come CustomAttribute, altrimenti l'animazione parte dall'angolo in alto a sinistra.

Dopodiché nel momento in cui deve diventare visibile:

```
<Constraint
    [...]
    android:alpha="1">
    <CustomAttribute
        app:attributeName="clickable"
        app:customBoolean="true" />
    <CustomAttribute
        app:attributeName="visibility"
        app:customIntegerValue="0" />
</Constraint>
```

Ciò consente di avere un'animazione stile fade in / out.

La visibility viene trattata come intero da interpolare tra 0 e 2, cioè tra VISIBLE e GONE, passando per INVISIBLE.

## Logging

Per il log in console abbiamo usato la libreria Timber, data la facilità e semplicità di utilizzo. Per far funzionare questa libreria è necessario eseguire il metodo statico “Timber.plant” da qualche parte all’avvio dell’applicazione, abbiamo quindi scelto di eseguirlo all’avvio dell’applicazione stessa. Cioè nell’onCreate della classe App:

```
override fun onCreate()
{
    super.onCreate()

    Timber.plant(object : Timber.DebugTree()
    {
        override fun createStackElementTag(element: StackTraceElement): String
        {
            return (super.createStackElementTag(element)
                + "->" + element.methodName
                + ":" + element.lineNumber)
        }
    })
}
```

Dopodiché per stampare in console una linea è sufficiente usare:

```
Timber.d("NORMAL STATE")
```

la cui linea di log corrispondente è:

```
D/DashboardFragment->onViewCreated$lambda-18:149: NORMAL STATE
```

Come si vede vengono aggiunte informazioni aggiuntive rispetto al normale logging, il nome del metodo e la riga di codice dove questo Log è stato chiamato. Ciò è estremamente utile se la logica è asincrona e complessa.

## Testing

Nello sviluppo dell’applicativo sono stati sviluppati 2 gruppi di test relativi al database, il primo per il debug delle funzioni dei dao, il secondo per il debug delle funzioni del DbRepository.

### Struttura dei test

I test sono stati realizzati richiamando metodi del database i quali sono per necessità asincroni, per tale ragione i test implementati fanno uso del metodo [runBlocking](#) del package [kotlinx.coroutines](#) il quale permette di eseguire una coroutine (in questo caso un metodo di tipo suspend) bloccando l’esecuzione del thread corrente e facendo sì che il test venga eseguito correttamente.

Inoltre, essendo questi dei test che fanno uso del database, è stato necessario creare un database temporaneo in memoria attraverso il metodo `inMemoryDatabaseBuilder()` della libreria Room, il quale permette la creazione di un

database fittizio che viene utilizzato durante i test con la struttura data dalla classe AppDatabase del progetto. Tale procedura di creazione viene eseguita prima di ogni test, creando un database nuovo per l'esecuzione di ogni test, tale comportamento viene definito grazie all'utilizzo dell'annotazione @Before la quale indica a Junit di eseguire questo metodo prima di ogni metodo marcato con l'annotazione @Test, di seguito invece viene lanciato il metodo con l'adorazione @After che in questo caso effettua la chiusura del database.

Per i test inoltre è stata realizzata una classe specifica per il test dei risultati dei test con dei metodi di assert realizzati ad hoc per le strutture dati del database.

## Test del database

Nei test del database sono state verificate le operazioni principali utilizzate nell'applicativo, ovvero: l'inserimento, il conteggio degli elementi, l'eliminazione e l'update, queste operazioni sono distribuite in 4 metodi distinti:

- insertAndGetAggregatesWithElements
  - In questo test vengono generati 10 aggregati con valori casuali, con altrettanti elementi associati a ciascuno e vengono salvati tutti in delle apposite strutture dati che poi vengono salvate nel database. Successivamente viene preso tutto il contenuto del database e viene fatto il confronto tra le strutture dati generate e quelle ottenute dalla query nel database, il controllo viene effettuato attraverso i metodi dell'oggetto CustomAsserts.
- insertCountAndDeleteAggregatesWithElements
  - Anche in questo test viene effettuata la generazione casuale degli aggregati nel database con rispettivi elementi e tag. Successivamente viene effettuato il conteggio di aggregati, elementi e tag nel database, viene poi confrontato con quelli generati. Viene infine effettuata la cancellazione di tutti gli elementi e viene verificato che il conteggio degli aggregati, elementi e tag nel database sia zero.
- insertAndDeleteSingleElements
  - Anche questo test comincia con la generazione casuale di alcuni aggregati ed elementi nel database, per poi proseguire con l'effettuazione e la verifica della procedura di eliminazione di un singolo elemento da un aggregato, seguita dalla verifica che tutti i parametri dell'aggregato vengano modificati e che l'elemento sia stato eliminato correttamente.
  - Di seguito l'elemento eliminato viene aggiunto nuovamente e viene verificato che l'aggregato ha il numero corretto di elementi e ha ripristinato il costo totale dell'aggregato.
- insertAndUpdateAggregatesAndElements
  - In questo ultimo test viene verificata la correttezza della procedura di update sia dell'aggregato che degli elementi.

## Test del dbRepository

In questo test vengono testati alcuni metodi del dbRepository i quali fungono da wrapper a dei metodi dei dao per poter estrarre le informazioni necessarie per la costruzione dei grafici pronte per l'utilizzo.

Questi metodi si differenziano dai metodi dei dao in quanto, in particolare per il filtro temporale, non richiedono una data d'inizio o di fine ma un enum che specifica quale intervallo temporale è d'interesse, ovvero month o year. In base a questa variabile il metodo genera internamente l'intervallo temporale. I test puntavano anche alla verifica del corretto comportamento di queste sotto funzioni.

Per il dbRepository test è stata utilizzata la stessa tipologia di creazione del database in memoria, nella procedura di pre-test. Questa volta dovevano essere verificati dei dati cumulativi, perciò è stato utilizzato un metodo di generazione degli aggregati predicibile, ed è stato incluso nella procedura di pre-test.

I test effettuati sono i seguenti:

- `getPeriodExpensesSumTest`
  - Verificava il corretto comportamento della funzione che restituisce il totale delle spese salvate in tutti i record del database.
- `getPeriodExpensesTest`
  - Verifica il corretto comportamento delle funzioni che restituiscono un array di spese, per ogni giorno dell'ultimo mese e per ogni mese dell'anno corrente.
- `getAggregateTagsAndCountByPeriodTest`
  - Verifica il comportamento delle funzioni che restituiscono il numero di aggregati salvati sotto un dato tag, nel mese corrente o nell'anno corrente.
- `getAggregateTagsAndExpensesByPeriodTest`
  - Verifica il comportamento delle funzioni che restituiscono la somma delle spese associate agli aggregati salvati sotto un dato tag, nel mese corrente o nell'anno corrente.
- `getElementTagsAndCountByPeriodTest`
  - Verifica il comportamento delle funzioni che restituiscono il numero di elementi salvati sotto un dato tag, nel mese corrente o nell'anno corrente.
- `getElementTagsAndExpensesByPeriodTest`
  - Verifica il comportamento delle funzioni che restituiscono la somma delle spese associate agli elementi salvati sotto un dato tag, nel mese corrente o nell'anno corrente.

# Applicazione cross-platform Flutter

## Requisiti

Quasi tutte le funzionalità dell'app sono state ridotte ma è presente una dotazione minima, le funzionalità che mancano per permettere l'utilizzo effettivo dell'app sono la pagina di modifica dell'aggregato e la possibilità di eliminare un aggregato.

### Requisiti funzionali

- Possibilità di aggiungere una spesa al database attraverso un'apposita schermata raggruppando le singole spese in aggregati, i quali devono permettere il salvataggio della data, e l'associazione con un tag che identifica la tipologia di spesa, esclusivamente per gli aggregati e non per gli elementi. Gli aggregati non devono consentire di allegare file o immagini.
- Visualizzare tutti gli scontrini inseriti, attraverso una schermata archivio.
- Ottenere un resoconto delle spese fatte, attraverso una pagina che presenti dei grafici che riassumono in maniera semplice e intuitiva la distribuzione delle spese fatte. In tale pagina saranno presenti 4 grafici:
  - Grafico a barre che rappresenti le spese giornaliere del mese corrente.
  - Grafico a barre che rappresenti le spese mensili dell'anno corrente.
  - Grafico a barre che rappresenti le spese divise per tag degli aggregati nel mese corrente.
  - Grafico a barre che rappresenti le spese divise per tag degli aggregati nell'anno corrente.
- Una dashboard statica con 1 grafico e due label.
- Avere a disposizione le seguenti pagine in cui distribuire le funzionalità descritte:
  - Dashboard: pagina principale con widget fissi.
  - Archivio: pagina di visualizzazione di tutti gli aggregati senza possibilità di filtrarli.
  - AddPage: pagina dove è possibile effettuare l'inserimento dell'aggregato e dei suoi elementi.
  - GraphPage: pagina dove vengono resi disponibili i riassunti grafici delle spese salvate nel database.

### Requisiti di versione Android

Con flutter non si ha il controllo diretto su quale sdk minimo si può supportare, vedendo nel build.gradle della parte specifica per android si vede:

```
minSdkVersion 16
targetSdkVersion 30
```

Ciò offre un'ampissima gamma di dispositivi supportati.

## Architettura

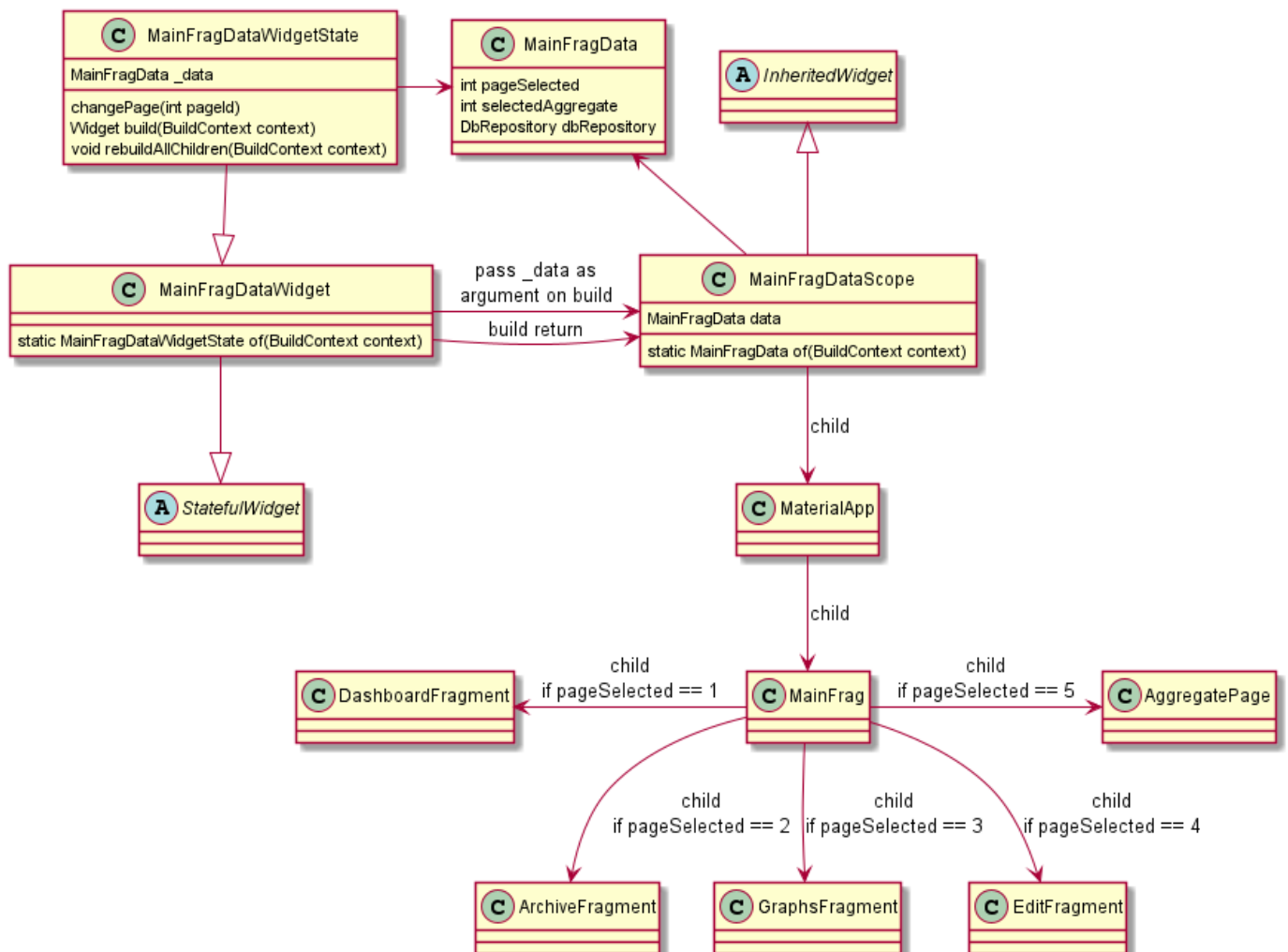
### Gestione delle schermate

La gestione delle schermate avviene attraverso la lettura delle variabili di un inherited widget e l'update delle variabili di uno stateful widget. Questa struttura emula in modo semplificato il comportamento del navigator.

La costruzione dell'interfaccia parte dal richiamo del widget `MainFragDataWidget`, il quale nell'operazione di build restituisce un inherited widget `MainFragDataScope`, passando a quest'ultimo come argomento il suo oggetto interno di tipo `MainFragData` e l'oggetto che gli è stato passato come figlio, in questo caso `MaterialApp`.

Come child `MaterialApp` ha un widget chiamato `MainFrag` il quale funge semplicemente da switch tra le varie schermate dell'app, la quale viene scelta in base al valore della variabile `pageSelected`, la quale è contenuta nell'oggetto `MainFragData` contenuto nel inherited widget.

Questa struttura consente di leggere e modificare i dati contenuti all'interno dell'oggetto `MainFragData` da tutto il sotto albero di widget senza necessità di dover passare come argomenti variabili o funzioni ma utilizzando degli appositi metodi statici.





Vediamo nel dettaglio la struttura della testa dell'albero di widget e il suo funzionamento. Entrambi gli oggetti `MainFragDataScope` e `MainFragDataWidget` contengono una copia dell'oggetto `MainFragData`.

E' possibile accedere da tutto l'albero di widgets all'oggetto `MainFragData` del inherited widget attraverso il suo metodo illustrato in seguito:

```
static MainFragData of(BuildContext context) {  
    return context.dependOnInheritedWidgetOfExactType<MainFragDataScope>()!.data;  
}
```

Tale metodo essendo statico effettuerà l'accesso al primo widget di tipo `MainFragDataScope` che incontrerà lungo l'albero partendo dalla cima e restituirà il suo oggetto di tipo `MainFragData`.

L'oggetto `MainFragData` del inherited widget è una copia di quello posseduto dallo widget `MainFragDataWidget`, il quale gli passa come argomento a ogni rebuild, dunque a ogni aggiornamento dello stato del widget `MainFragDataWidget` l'inherited widget figlio avrà sempre dati aggiornati.

Di seguito un esempio di accesso alle variabili del inherited widget:

```
MainFragDataScope.of(context).pageSelected
```

Per effettuare l'aggiornamento dei dati, si deve richiamare lo stato del widget stateful e non si devono modificare le variabili contenute all'interno dell'inherited widget, almeno se l'intento è quello di innescare una rebuild con il nuovo stato.

Per fare ciò si deve utilizzare il metodo statico del widget `MainFragDataWidget` mostrato in seguito:

```
static MainFragDataWidgetState of(BuildContext context){  
    return context.findAncestorStateOfType<MainFragDataWidgetState>()!;  
}
```

Tale metodo permette di richiamare lo stato del primo widget `MainFragDataWidget` nell'albero, e dunque di accedere ai suoi metodi per effettuare l'aggiornamento dello stato. L'aggiornamento della schermata infatti avviene attraverso la chiamata di tale stato e l'esecuzione della funzione `void changePage(int pageId)` che tramite la funzione `setState()` effettua la rebuild, come mostrato in seguito:

```
MainFragDataWidget.of(context).changePage(PageMap.archiveId)
```

## Database relazionale

Il database è stato sviluppato attraverso la libreria Sqflite, un plugin che permette l'utilizzo di SQLite all'interno dell'ambiente flutter, il quale permette la compilazione dell'applicativo in Android, in IOS e in Mac OS.

Sqflite è una libreria che consente di eseguire la gestione di un database, eseguendo tutte le operazioni basilari di inserimento, lettura update e eliminazione, eseguendo tali operazioni su di un thread in background.

### Le data class

Il mattone base del database è composto dalle classi dei dati che anche in questo caso come nel caso Android hanno la stessa conformazione dei record all'interno della tabella.

Rispetto al caso Android con Room, gli oggetti in questione necessitano di metodi di encoding e decoding embedded, dal tipo map alla data class, e dalla data class al tipo map, in quanto la tipologia di strutture dati che è supportata dal database sono esclusivamente i map, con dei campi chiave che sono semplicemente i nomi delle colonne del database e come valori i contenuti delle celle del record.

Le strutture dati utilizzate all'interno del database sono 3, come nel caso precedente: Aggregate, Element e Tag. Vediamo nel dettaglio la struttura dell'oggetto Tag riportato per semplicità in quanto più breve ma con struttura identica agli altri:

```
class Tag{

  int? tag_id = null;
  final String tag_name;

  Tag({
    required this.tag_name,
    this.tag_id
  });

  Map<String, dynamic> toMap() {
    return {
      'tag_id': tag_id,
      'tag_name': tag_name
    };
  }

  static Tag fromMap(Map<String, dynamic> map){
    return Tag(
      tag_id: map['tag_id'],
      tag_name: map['tag_name']
    );
  }

  ...
}
```

NOTA: in Dart la data class non esiste come in Android e se ne parla solo per similitudine con il caso precedente.

Oltre ai campi dell'oggetto e la struttura del costruttore è importante notare come, la classe presenti un metodo toMap() che serve ad effettuare un dump dell'oggetto nel database, partendo da un oggetto esistente, mentre il metodo fromTag() è statico, e permette di essere chiamato direttamente senza un'istanza esistente della classe Tag per generarne una a partire da un map restituito da una query al database.

## Il database

L'implementazione della classe che gestisce il database che abbiamo scelto, segue un approccio statico, dunque non necessita della creazione del database in un determinato momento del ciclo di vita dell'applicazione, ma semplicemente viene gestito automaticamente ad ogni chiamata, statica, dello stesso.

Per semplicità è stato creato un oggetto di gestione del database separato per ogni tabella, ovvero: Aggregate, Element e tag. Vediamo nel dettaglio la struttura del database manager dei tag:

```
class DbTagMng{

  static final DbTagMng instance = DbTagMng._init();

  static Database? _database;

  DbTagMng._init();

  Future<Database> get database async {
    if (_database != null) return _database!;

    _database = await _initDB('tags.db');
    return _database!;
  }
  ...
}
```

Ogni database manager presenta un attributo instance, del suo stesso tipo statico, in modo che sia accessibile dovunque venga importata la classe. Oltre all'istanza statica dell'oggetto stesso è presente un riferimento statico ad una istanza del database inizialmente nulla, il quale può essere richiamato attraverso un getter, che effettua il controllo della sua inizializzazione, nel caso in cui il database sia già inizializzato restituisce l'istanza statica del database già inizializzata, altrimenti richiama il metodo initDB() che si occupa della creazione dell'istanza del database, creandolo nella memoria del dispositivo se non è già stato creato, oppure aprendo quello esistente.

Ovviamente come ogni altra operazione che legge o scrive dalla memoria non volatile del dispositivo, sqflite effettua tali operazioni su di un thread apposito che

impone di dichiarare il getter async e di restituire un tipo di ritorno `Future<Database>`.

Di seguito analizziamo più nel dettaglio la funzione `_initDB()` e la funzione `_createDB()`:

```
class DbTagMng{
    ...
    Future<Database> _initDB(String filePath) async {
        final dbPath = await getDatabasesPath();
        final path = join(dbPath, filePath);

        return await openDatabase(path, version: 1, onCreate: _createDB);
    }

    Future _createDB(Database db, int version) async {
        final idType = 'INTEGER PRIMARY KEY AUTOINCREMENT';
        final textType = 'TEXT NOT NULL';
        final boolType = 'BOOLEAN NOT NULL';
        final integerType = 'INTEGER NOT NULL';

        await db.execute('''
            CREATE TABLE tag (
                tag_id $idType,
                tag_name $textType
            )''');
    }
    ...
}
```

Analizziamo il metodo `_initDB()`, il quale prende come argomento il nome del database che si vuole leggere/creare e attraverso il metodo `getDatabasePath()` e `join()` ricostruisce il percorso di destinazione del database. Il secondo passaggio effettuato da questo metodo è la chiamata del metodo `openDatabase()` che richiede come argomenti il percorso del database, la versione di tale database e un metodo da chiamare nel caso in cui il database non sia presente nella memoria del dispositivo, nel nostro caso definito con il nome di `_createDB()`.

Il metodo `_createDB()` effettua semplicemente una query in raw per la creazione di una tabella con le caratteristiche desiderate, codificate direttamente nel linguaggio SQL. Nel caso specifico della tabella tag si impongono tutti i campi non nullable e l'id come chiave primaria autoincrementante.

Vediamo di seguito un esempio di semplice inserimento che sfrutta il getter del database sopra illustrato:

```
class DbTagMng{
    ...
    Future<int> insert (Tag tag) async {
        final db = await instance.database;
        final id = await db.insert("tag", tag.toMap());
        return id;
    }
    ...
}
```

Tale metodo effettua l'inserimento di una struttura dati Tag come nuovo record del database. La prima operazione che viene svolta dalla funzione è la chiamata del getter dell'istanza statica `instance.database`, la quale effettua il controllo dell'inizializzazione del database ed ritorna tale istanza, ovviamente essendo un'operazione asincrona è necessario attendere la risposta del metodo con la parola chiave `await`. Una volta ottenuta l'istanza del database, richiama esegue il metodo `insert()` della libreria che richiede il nome della tabella e una struttura dati map che si ottiene dall'oggetto tag attraverso il metodo `toMap()` illustrato in precedenza. Tale funzione restituisce di default l'id del nuovo record inserito, che ovviamente è un future e anche in questo caso deve essere atteso con `await`. Notiamo che anche in questo caso come nella creazione del database la procedura è asincrona e dunque anche il metodo `insert` dell'oggetto richiede che la funzione si `async` e restituisca un Future.

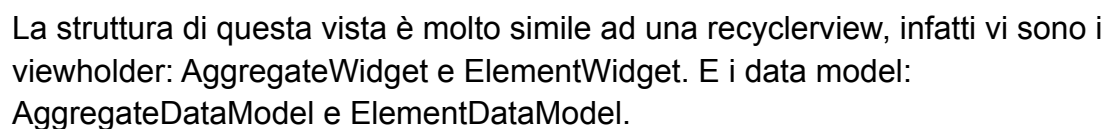
## Il db repository

Il repository del database incorpora la logica relazionale del database, la quale nella versione Android era stata lasciata all'interno dei Dao.

In questa implementazione sono presenti soltanto i metodi principali di lettura, scrittura e eliminazione degli aggregati con i relativi elementi, e annessa gestione dei tag.

In questa versione semplificata non sono state implementate le funzioni per la ricerca degli aggregati in base alla data o al tag, o tutte le altre funzioni per la richiesta del conteggio di elementi o di somma cumulativa dei costi che in questo caso viene effettuata nel ChartsBuilder.

3 stateless widgets, 1 stateful widget e varie dataClass di supporto.



La lista viene inoltre costruita con una funzione simile ad `onBindViewHolder`, solo che in questo caso gli oggetti non sono riciclati ma ricreati ogni volta:

```
Widget buildSingleElement(EditDataModel data) {
    if (data is AggregateDataModel) {
        return AggregateWidget(data: data, update: updateList, selectDate:
            _selectDate);
    }

    if (data is ElementDataModel) {
        return ElementWidget(data: data, update: updateList);
    }
    throw UnsupportedError("");
}
```

Il list builder non fa altro che chiamare il metodo `buildSingleElement` per ogni elemento:

```
ListView.separated(
    padding: const EdgeInsets.all(8),
    itemCount: elements.length,
    itemBuilder: (BuildContext context, int index) {
        return buildSingleElement(elements[index]);
    },
    separatorBuilder: (BuildContext context, int index) => const Divider(),
);
```

Ciò è esattamente quello che fa internamente il `recyclerView`.

Per realizzare l'effetto lista infinita, cioè che quando si modifica l'ultimo elemento ne viene aggiunto uno al di sotto automaticamente, abbiamo usato:

```
void updateList(EditDataModel value) {
    var index = (value as HasIndex).index;
    setState(() {
        elements[index] = value;
        if (index == elements.length - 1) {
            elements.add(ElementDataModel(
                index: elements.length, name: "", cost: 0, num: 0));
        }
    });
}
```

cioè nella callback che viene chiamata quando l'utente modifica il testo vi è un `if` per controllare se l'elemento corrente è l'ultimo e in caso aggiunge un `ElementDataModel` vuoto in fondo.

## I grafici

Nell'applicativo flutter per la gestione dei grafici è stata utilizzata la libreria `charts_flutter`, una libreria distribuita direttamente da google.

La pipeline di creazione dei grafici si basa sull'utilizzo di 3 componenti principali: gli oggetti `DataSet`, la classe `chartsBuilder` e infine gli widget `simpleBarCharts`.

Si noti che per l'applicativo flutter sono stati realizzati esclusivamente istogrammi.

### Gli oggetti `DataSet`

Gli oggetti `DataSet` sono delle classi che contengono 3 tipologie di dati:

```
class DoubleDataSet {
    final String label;
    final double data;
    final charts.Color barColor;

    DoubleDataSet({
        required this.label,
        required this.data,
        required this.barColor
    });
}

class IntDataSet {
    final String label;
    final int data;
    final charts.Color barColor;

    IntDataSet({
        required this.label,
        required this.data,
        required this.barColor
    });
}
```

il label, il quale contiene la stringa associata ad ogni colonna del grafico, il dato che codifica l'altezza della colonna e la grandezza da rappresentare, infine `barColor` codifica il colore della colonna. Per ogni grafico viene costruita una lista di questi oggetti. Sono state realizzate due tipologie di oggetti per la gestione di serie di interi e di double ma nell'applicativo sono state utilizzate esclusivamente serie di double.

### Il Charts Builder

La classe `ChartsBuilder`, è l'oggetto che si occupa della preparazione dei dati e dei label per ogni grafico presente nell'app. Dal momento che la complessità del database in questa app è stata ridotta, gran parte della logica di filtraggio e preparazione dei dati è stata spostata in questa classe per velocizzare lo sviluppo.

Questa classe infatti si occupa di effettuare il caricamento di tutti i dati dal database e di filtrarli e dividerli in delle sotto liste che poi vengono trasformate nelle serie di dati da mostrare in ogni grafico. Tale operazione avviene tramite il metodo `loadData()` il quale è un metodo `async`, in quanto deve gestire l'interazione con il database e restituisce un `Future<bool>`, il quale è necessario per rendere noto all'interfaccia quando i dati dell'oggetto sono pronti per essere visualizzati.



## Il simpleBarChart

Il simpleBarChart è una tipologia di widget stateless sviluppata per prendere come argomento una lista di DataSeries e restituire tramite il metodo build un grafico ben dimensionato all'interno di una cardview. Anche in questo caso sono stati realizzati due widget, uno per la visualizzazione delle serie di `DoubleDataSeries` e uno per gli `IntDataSeries`.

Vediamo nel dettaglio la funzione build dell'oggetto `SimpleDoubleBarChart`:

Il quale costruisce una lista di oggetti charts.Series che prende un oggetto di vario tipo e attraverso gli argomenti domainFn, measureFn, colorFn ne estrae i dati e li utilizza per la costruzione del grafico.

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

  List<charts.Series<DoubleDataSeries, String>> series = [
    charts.Series(
      id: chartId,
      data: data,
      domainFn: (DoubleDataSeries series, _) => series.label,
      measureFn: (DoubleDataSeries series, _) => series.data,
      colorFn: (DoubleDataSeries series, _) => series.barColor
    )
  ];

  return Container(
    height: 350,
    padding: const EdgeInsets.all(5),
    child: Card(
      child: Padding(
        padding: const EdgeInsets.all(9.0),
        child: Column(
          children: <Widget>[
            Text(
              chartName,
              style: Theme.of(context).textTheme.bodyText1,
            ),
            Expanded(
              child: charts.BarChart(
                series,
                animate: true,
                domainAxis: charts.OrdinalAxisSpec(
                  renderSpec: charts.SmallTickRendererSpec(
                    labelRotation: 70,
                    labelStyle: charts.TextStyleSpec(
                      fontSize: (littleLabels) ? 8 : 14,
                      color: charts.MaterialPalette.black
                    )
                  ),
                ),
              ),
            ),
          ],
        ),
      ),
    ),
  );
}
```

In seguito si vede come il widget predispone la `cardView` in un container e setta tutta una serie di widget per la corretta visualizzazione del grafico.

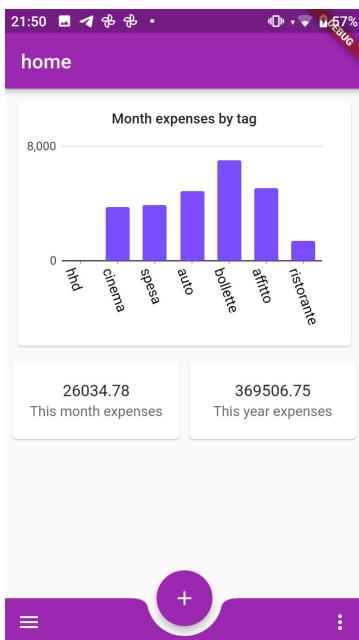
L'ultimo passaggio consiste nell'inserimento del grafico all'interno dell'albero di widget con, il passaggio della serie di dati, e la configurazione di alcune impostazioni per la visualizzazione dei label.

### Il caricamento e la visualizzazione

Un'ultimo passaggio degno di nota nella costruzione dei grafici è rappresentato dal metodo di caricamento e visualizzazione che sfrutta il widget `FutureBuilder`, tale widget prende come argomento un future e una funzione builder. Come future viene passata la funzione `loadData()` della classe `ChartsBuilder`, la quale restituisce un `Future<bool>` nel momento in cui le serie dei grafici sono state costruite. La funzione Builder ha un comportamento particolare, viene chiamata una prima volta all'apertura della pagina, ed effettua il caricamento di un determinato sotto albero di widget in base allo stato del future, se infatti questo non è già stato restituito in questa implementazione carica l'oggetto `CircularProgressIndicator()`, altrimenti effettua la visualizzazione dei grafici, estraendo dall'oggetto `ChartsBuilder` le liste di `DataSet`, e le passa agli widget `SimpleDoubleBarChart`.

## User interface

### dashboard




La dashboard ha lo stesso look della controparte Android, ma in questo caso è fissa e non personalizzabile.

Premendo sul fab si va nella pagina che permette di aggiungere un nuovo scontrino al database.

Premendo sull'hamburger si apre il bottom navigator che permette di spostarsi sull'archivio oppure sulla pagina dei grafici.

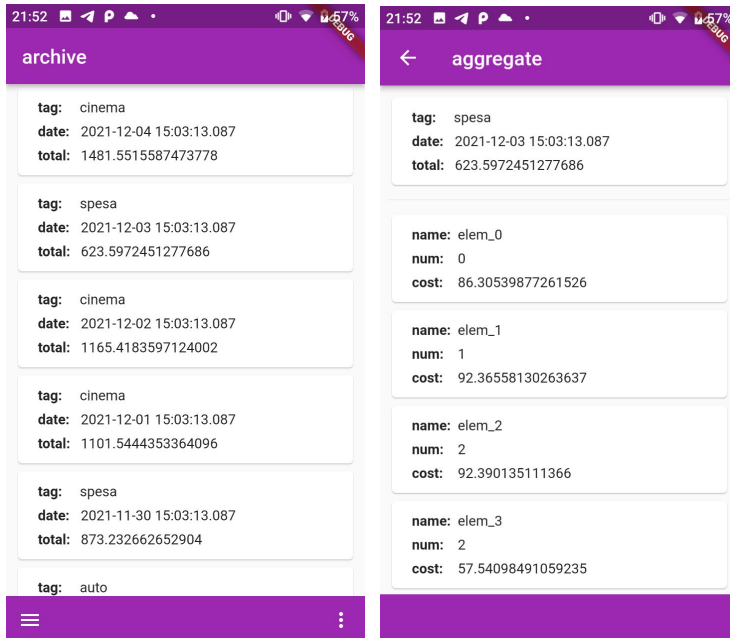
## Pagina aggiunta aggregato



Il look e il funzionamento sono identici a quello della controparte android. Quindi premendo sulla data si aprirà un time picker, e ogni volta che si modifica l'ultimo elemento ne apparirà sempre uno al di sotto.

Non è possibile però aggiungere un allegato. Premendo il fab viene salvato tutto sul db.

## Archivio



tag	date	total
cinema	2021-12-04 15:03:13.087	1481.5515587473778
spesa	2021-12-03 15:03:13.087	623.5972451277686
cinema	2021-12-02 15:03:13.087	1165.4183597124002
cinema	2021-12-01 15:03:13.087	1101.5444353364096
spesa	2021-11-30 15:03:13.087	873.232662652904
auto		

L'archivio permette di visualizzare tutti gli scontrini inseriti nel database. Premendo su uno di essi si viene trasportati su una pagina che permette di visualizzare l'aggregato e i suoi elementi in dettaglio. Nell'app flutter non è possibile la modifica né la cancellazione.

## Grafici



La pagina dei grafici nell'applicativo flutter presenta 4 grafici che possono essere visualizzati su di una listView, con i dati che vengono caricati all'apertura della pagina.

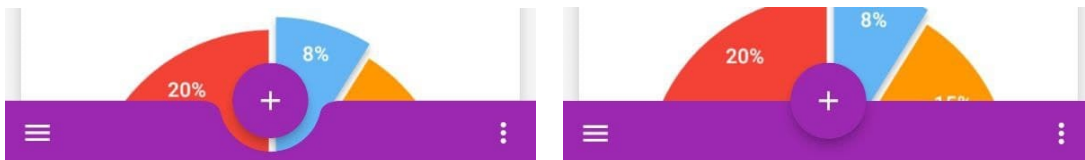
La pagina non è interattiva e permette la visualizzazione dei seguenti grafici come istogrammi:

- Spese dell'ultimo mese divise per giorni.
- Spese dell'ultimo anno divise per mese.
- Spese dell'ultimo mese divise per tag.
- Spese dell'ultimo anno divise per tag.

## Sviluppo

### Problematiche e strategie adottate

Per lo sviluppo dell'applicazione si è adottato un approccio senza l'utilizzo del Navigator, in quanto sono stati riscontrati dei bug legati all'interazione del notch del fab con il cambiamento della route.



Nelle due immagini è illustrato il comportamento della proprietà cradle del fab dopo una rebuild eseguita nella stessa route (a sinistra), e il risultato del rendering ottenuto alla prima rebuild che segue un cambio di ruote (a destra).

Dopo una lunga serie di ricerche, il problema sembrava ancora non avere una soluzione pubblicata, dunque valutando che l'app non aveva bisogno dello stack delle schermate, lo sviluppo si è mosso verso una architettura a singola route, utilizzando un inherited widget per la gestione delle pagine e per il passaggio dei dati tra le schermate.

## Future builder

Per gestire la natura asincrona del database abbiamo usato il Widget FutureBuilder. Questo Widget permette di gestire funzioni che ritornano un Future<> con una callback che in base allo stato della variabile Future deve restituire il widget da visualizzare.

Ad esempio se la funzione “getAggregates” restituisce un “Future<List>”, per poter visualizzare la lista si deve usare “FutureBuilder” passando la funzione come argomento. Il secondo argomento e’ una callback che deve restituire il widget da visualizzare in base allo stato del Future, in questo caso sono stati implementati gli stati: ConnectionState.done e snapshot.hasError, se non e’ nessuno di questi 2 stati suppongo sia in stato di loading.

```
FutureBuilder(  
  future: getAggregates(),  
  builder: (context, snapshot) {  
    if (snapshot.connectionState == ConnectionState.done) {  
      final data = snapshot.data as List?;  
      [...]  
    } else if (snapshot.hasError) {  
      throw snapshot.error ?? Error();  
    } else {  
      return const Center(child: CircularProgressIndicator());  
    }  
  });
```

Nel momento in cui “snapshot.connectionState == ConnectionState.done” sono sicuro che il future e’ sttato esaurito e posso quindi prendere i dati:

```
final data = snapshot.data as List?;  
if (data == null) {  
  return const Center(  
    child: Text("errrrrr"),  
  );  
}
```

Il casting e il test per la nullability sono comunque necessari, si ha infatti che “snapshot.data” e’ “Object?”, devo quindi convertirlo al giusto tipo.

## Gestione back button

Per gestire il comportamento dell'app alla pressione del pulsante fisico indietro abbiamo usato il widget apposito WillPopScope.

Ad esempio se non si deve chiedere conferma all'utente:

```
WillPopScope(  
  onWillPop: () {  
    MainFragDataWidget.of(context).changePage(PageMap.archiveId);  
    return Future.value(false);  
  },  
  child: [...]  
)
```

in questo caso si cambia pagina non appena l'utente preme il pulsante indietro. Il "return Future.value(false);" è necessario altrimenti viene fatto il pop dallo stack e si esce dell'app.

Per altre schermate viene chiesta la conferma con un dialog:

```
Future<bool> sureToExit(BuildContext context, String content,  
  Function positiveButton, Function negativeButton) async {  
  return (await showDialog(  
    context: context,  
    builder: (context) => AlertDialog(  
      title: const Text('Are you sure?'),  
      content: Text(content),  
      actions: <Widget>[  
        TextButton(  
          onPressed: () {  
            print("sureToExit neg");  
            negativeButton();  
          },  
          child: const Text('No'),  
        ),  
        TextButton(  
          onPressed: () {  
            print("sureToExit pos");  
            positiveButton();  
          },  
          child: const Text('Yes'),  
        ),  
      ],  
    ),  
  )) ??  
  false;  
}
```

```
WillPopScope(
  onWillPop: () {
    return sureToExit(
      context,
      'Do you want to exit an App',
      () => Navigator.of(context).pop(true),
      () => Navigator.of(context).pop(false)
    );
  },
  child: [...]
)
```

E in questo caso si esce dall'app in caso l'utente preme su "yes".

## Testing

Per quanto riguarda i test sono state fatte delle prove per effettuare dei test con i metodi del database che non sono stati portati avanti a causa di alcune problematiche riscontrate. Nello specifico durante l'esecuzione dei test la libreria utilizzata Sqlite non trovava l'implementazione del metodo `getDatabasesPath` il quale doveva restituire il path del file del database, dopo diversi tentativi di farlo funzionare è stato deciso di non scrivere dei test per l'applicazione flutter.

## Conclusione

Alla fine del percorso di sviluppo delle due applicazioni, è estremamente evidente il divario tra i due framework.

L'sdk di android nativo presenta grandi potenzialità e possibilità di personalizzazione. Si presta più agevolmente alle interazioni con il sistema, di contro ha una elevata complessità nella gestione delle componenti grafiche, ciò rende complessa l'implementazione di grafiche dinamiche e reattive.

L'sdk Flutter presenta al contrario una grande semplicità di utilizzo per la gestione della grafica, dato che ogni componente viene definita e gestita con una quantità minore di codice rispetto alla controparte nativa. Di contro la giovane età del framework non ci sono librerie mature come in android.